electronics

MDPI

Article

# Designing a CHAM Block Cipher on Low-End Microcontrollers for Internet of Things

**Hyeokdong Kwon [1], SangWoo An [2], YoungBeom Kim [2], Hyunji Kim [1], Seung Ju Choi [1], Kyoungbae Jang [1], Jaehoon Park [1], Hyunjun Kim [1], 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 (H.K.); 1594012@hansung.ac.kr (H.K.); bookingstore3@hansung.ac.kr (S.J.C.); starj1234@hansung.ac.kr (K.J.); 20213201@hansung.ac.kr (J.P.); amdjd0704@hansung.ac.kr (H.K.)
2 Department of Financial Information Security, Kookmin University, Seoul 02707, Korea; pinksnail06@kookmin.ac.kr (S.A.); darania@kookmin.ac.kr (Y.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

Received: 9 August 2020; Accepted: 19 September 2020; Published: 22 September 2020

check for updates

**Abstract:** As the technology of Internet of Things (IoT) evolves, abundant data is generated from sensor nodes and exchanged between them. For this reason, efficient encryption is required to keep data in secret. Since low-end IoT devices have limited computation power, it is difficult to operate expensive ciphers on them. Lightweight block ciphers reduce computation overheads, which are suitable for low-end IoT platforms. In this paper, we implemented the optimized CHAM block cipher in the counter mode of operation, on 8-bit AVR microcontrollers (i.e., representative sensor nodes). There are four new techniques applied. First, the execution time is drastically reduced, by skipping eight rounds through pre-calculation and look-up table access. Second, the encryption with a variable-key scenario is optimized with the on-the-fly table calculation. Third, the encryption in a parallel way makes multiple blocks computed in online for CHAM-64/128 case. Fourth, the state-of-art engineering technique is fully utilized in terms of the instruction level and register level. With these optimization methods, proposed optimized CHAM implementations for counter mode of operation outperformed the state-of-art implementations by 12.8%, 8.9%, and 9.6% for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively.

**Keywords:** cham block cipher; microcontroller; counter mode of operation; parallel computation; round based encryption

## 1. Introduction

As the Internet of Things (IoT) environment evolves, the embedded processor has rapidly developed. The information transmitted throughout IoT devices is encrypted before transmission for safe services. The encryption requires an expensive computational overhead on low-end IoT devices with limited computing power, low energy capacity, and small storage. For this reason, lightweight block ciphers have been actively studied, which can be implemented in an efficient manner even under certain limitations.

In this paper, we proposed the optimized CHAM block cipher on 8-bit Alf and Vegard's RISC (AVR) microcontrollers that applied suitable optimization techniques for Addition, Rotation, and eXclusive (ARX)-based block ciphers. First, the counter mode of CHAM block cipher was accelerated with pre-calculation. Secondly, the encryption with a variable key was optimized with on-the-fly table calculation. Thirdly, parallel computations for encrypting multi-plaintext blocks were implemented. To achieve the compact implementation, we used 8-bit AVR microcontrollers-specific assembly-level optimizations for CHAM block ciphers (e.g., word size, number of registers, and instruction set).

The remainder of this paper is organized as follows. In Section 2, the contribution is summarized. In Section 3, the background is described. In Section 4, specifications of CHAM block cipher and target AVR microcontrollers are described. In Section 5, the compact implementation of CHAM block cipher on AVR microcontrollers are described. In Section 6, the performance of proposed implementations in terms of execution timing is evaluated. Finally, Section 7 concludes the paper.

## 2. Contribution

Proposed implementation techniques for CHAM block cipher can be used for other ARX-based block ciphers, such as SPECK and SIMON without difficulty. Detailed contributions are as follows:

- Optimized counter mode of operation for CHAM block cipher: Due to the characteristic of counter mode of operation, the certain amount of input value is constant (i.e., nonce). Through pre-calculation, 5 left rotation by 1-bit, 10 XOR, 3 ADD, and 3 left rotation by word-wise operations are replaced to 5 word-wise look-up table accesses. These techniques optimize whole parameters of CHAM block cipher;
- Variable-key based CHAM-CTR (Counter) implementations: When the key is fixed, there is no update of look-up table and the pre-calculation is performed offline. For practical implementation, variable-key scenarios should be considered. Therefore, we proposed a pre-computation for look-up table in on-the-fly way;
- Parallel implementation of CHAM-64/128: When implementing CHAM-64/128, there are many registers that are not used in an optimal way. By utilizing all of these registers, the parallel encryption is available. It allows multiple plaintexts to be encrypted, simultaneously. We proposed an optimized register allocation. This ensures efficient performance for parallel computation;
- Highly optimized source code and 8-round based implementation: The state-of-art engineering technique is fully utilized with available instruction sets and register files to achieve high performance. The partially unrolled 8-round based implementation is adopted, which optimizes the left rotation by 8-bit wise operations.

### *Extended Version of WISA'20*

In this paper, we extended our previous work published in WISA'20 [1]. In WISA'20, only fixed-key based CHAM-CTR implementations for 128-bit, 192-bit, and 256-bit security levels on low-end microcontrollers were investigated (i.e., FACE-LIGHT). This work further extended CHAM-CTR implementations to variable-key scenario. In particular, the variable-key based implementation is optimized with on-the-fly computations for look-up tables. The proposed implementation is implemented in the constant timing, which is secure against Timing Attack (TA).

## 3. Backgrounds

### *3.1. Block Cipher*

The block cipher can be categorized into two structures, including Feistel and Substitution Permutation Network.

- Feistel: Feistel structure divides plaintext. One block is used as the round function input. Feistel structure is simple to implement encryption because the structure of decryption and encryption is identical. However it requires a number of move operations;
- Substitution Permutation Network: Substitution Permutation Network (SPN) performs encryption by repeating the process, which substitutes the plaintext through S-BOX and permutates its result. SPN requires the reversed function for decryption. This is practical for hardware implementation.

### *3.2. Mode of Operation*

The block cipher should perform several encryption operations for blocks if the plaintext exceeds the block size. To overcome this limitation, the mode of operation was proposed. In addition, some mode of operations provide authentication and integrity provision. Types of block encryption operation modes include Electronic codebook (ECB), Cipher-block chaining (CBC), Cipher feedback (CFB), and Counter (CTR). Among them, the CTR uses nonce and number of block (counter), instead of plaintext. In the counter mode, nonce and counter are used as input values of the encryption algorithm. After that, the XOR operation is performed with encryption result and plaintext to get ciphertext.

## **4. Related Works**

### *4.1. CHAM Block Cipher*

For low-end Internet of Things (IoT) devices, the lightweight ciphers were proposed. Lightweight block ciphers are a primitive technology to optimize the hardware chip size and reduce execution timing. According to the current trend, a number of block cipher algorithms have been designed for being in a lightweight environment.

In ICISC'17, the Korean block cipher CHAM, one of lightweight block cipher, was presented by the Attached Institute of ETRI [2]. The CHAM family consists of three cipher standards, including CHAM-64/128, CHAM-128/128, and CHAM-128/256. Each CHAM block ciphers operate 80, 80, and 96 rounds, respectively. The CHAM block ciphers are of the generalized 4-branch Feistel structure based on ARX operations.

In ICISC'19, the revised version of CHAM block cipher was announced [3]. Revised CHAM using a SAT solver to prevent new related-key differential characteristics and differentials attack. In addition, numbers of rounds of CHAM-64/128, CHAM-128/128, and CHAM-128/256 are increased to 88, 112, and 120 than original CHAM block ciphers, respectively. In this paper, we target to the revised CHAM. For convenience, we will use CHAM instead.

### *4.2. Previous Block Cipher Implementations on 8-bit AVR Microcontrollers*

The low-end 8-bit AVR platform working at 8 MHz supports 8-bit instruction set, 128 KB FLASH memory, and 4 KB RAM. The number of available registers is 32. Among them, six registers (i.e., R26∼R31) are reserved for address pointers and the other registers are used for a general purpose. The basic arithmetic instruction takes one clock cycle, while the memory access takes two clock cycles per byte. Detailed instruction set summary for efficient CHAM-CTR implementation is given in Table 1.

**Table 1.** Summary of instruction set for optimized CHAM implementations on 8-bit Addition, Rotation, and eXclusive (ARX) microcontrollers [4,5].

| asm | Operands | Description | Operation | #Clock |
|---|---|---|---|---|
| ADD | Rd, Rr | Add without Carry | Rd ← Rd+Rr | 1 |
| ADC | Rd, Rr | Add with Carry | Rd ← Rd+Rr+C | 1 |
| EOR | Rd, Rr | Exclusive OR | Rd ← Rd⊕Rr | 1 |
| LSL | Rd | Logical Shift Left | C\|Rd ← Rd«1 | 1 |
| ROL | Rd | Rotate Left Through Carry | C\|Rd ← Rd«1\|\|C | 1 |
| MOV | Rd, Rr | Copy Register | Rd ← Rr | 1 |
| MOVW | Rd, Rr | Copy Register Word | Rd+1:Rd ← Rr+1:Rr | 1 |
| LDI | Rd, K | Load Immediate | Rd ← K | 1 |
| LD | Rd, X | Load Indirect | Rd ← (X) | 2 |
| LPM | Rd, Z | Load Program Memory | Rd ← (Z) | 3 |
| ST | Z, Rr | Store Indirect | (Z) ← Rr | 2 |
| PUSH | Rr | Push Register on Stack | STACK ← Rr | 2 |
| POP | Rd | Pop Register from Stack | Rd ← STACK | 2 |

A number of works have devoted to improve the performance of lightweight cryptography implementations on low-end microcontrollers (e.g., 8-bit AVR). The structure of block cipher is largely divided into two categories. First, Addition, Rotation, and eXclusive-or (ARX) based block ciphers were efficiently implemented on low-end microcontrollers.

In WISA'13, LEA block cipher was by the attached institute of ETRI [6]. The first implementation of LEA-128 on the 8-bit AVR microcontroller achieved 190 clock cycles per byte for encryption [6]. In WISA'15, speed-optimized and memory-efficient LEA implementations were presented [4]. In [7], the number of general purpose registers and the instruction set of the AVR microcontroller were fully utilized to optimize LEA block cipher implementation. In WISA'18, general purpose registers are efficiently utilized to cache intermediate results of delta variables during key scheduling of LEA [5].

In CHES'06, HIGHT block cipher was introduced [8]. The basic implementation of HIGHT was firstly introduced in [9]. The execution timing for encryption and decryption is 2438 and 2520 clock cycles per byte, respectively. In [7], efficient rotation operations were suggested and achieved high performance. In [10], speed-optimized and memory-efficient HIGHT implementations were presented.

In ICISC'17, the original CHAM on 8-bit AVR microcontrollers achieved 172, 148, and 177 clock cycles per byte for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively [2]. In [11], 2-round based memory-efficient implementation was suggested. The work achieved 211, 187, and 223 clock cycles per bytes for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively, with a reasonably small memory footprint. In ICISC'19, revised CHAM was presented by modifying the round of CHAM [3]. AVR implementations achieved 188, 203, and 219 clock cycles per byte for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively.

Second, Substitution Permutation Network (SPN)-based block ciphers were also actively investigated. Among them, AES implementations received high attention since the block cipher is the international standard.

In [12], S-box pointer was maintained in Z address pointer for fast memory access. The mix-column computation was efficiently handled with the conditional branch skip. In ICISC'19, the compact implementation of AES-CTR on microcontrollers (i.e., FACE-LIGHT) was presented [13]. With the newly designed cache table for low-end microcontrollers, implementations of AES-CTR achieved 138, 168, and 199 clock cycles per byte for 128-bit, 192-bit, and 256-bit security levels, respectively. In [14], implementations of AES-GCM achieved 415, 466, and 477 clock cycles per byte for 128-bit, 192-bit, and 256-bit security levels, respectively. In [15], implementations of ARIA-CTR achieved 187.1, 216.8, and 246.6 clock cycles per byte for 128-bit, 192-bit, and 256-bit security levels, respectively.

## 5. Proposed Method

### *5.1. Optimized CHAM-CTR Mode Encryption*

In this section, we present the optimized CHAM encryption with counter mode of operation (CTR mode) on 8-bit AVR microcontrollers. In general, block ciphers use plaintext as an input value. While performing the CTR mode of operation, both nonce and counter are used instead of plaintext. At this point, the nonce is a fixed value and the counter indicates order of blocks. Since CHAM divides the input into quarters, the length of block is assigned 16-bit, 32-bit, and 32-bit for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively. Thus, the length of counter is also the same as block length and the remainder is set to nonce (i.e., 48-bit, 96-bit, and 96-bit for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively. A 32-bit counter can also be used for CHAM-64/128. In this case, the pre-calculated part was slightly reduced to half. However the performance is better than the basic implementation). Among the input values, nonce has the characteristic of being fixed during computations. Utilizing this characteristic, some operations for several rounds can be simplified to a look-up table access. On the other hand, the counter value increases in each time move to the next block. This incurs the update of an intermediate result. In other words, blocks affected by a counter cannot be cached. Figure 1 shows that the counter value flow. Optimal implementation techniques for each round are as follows.

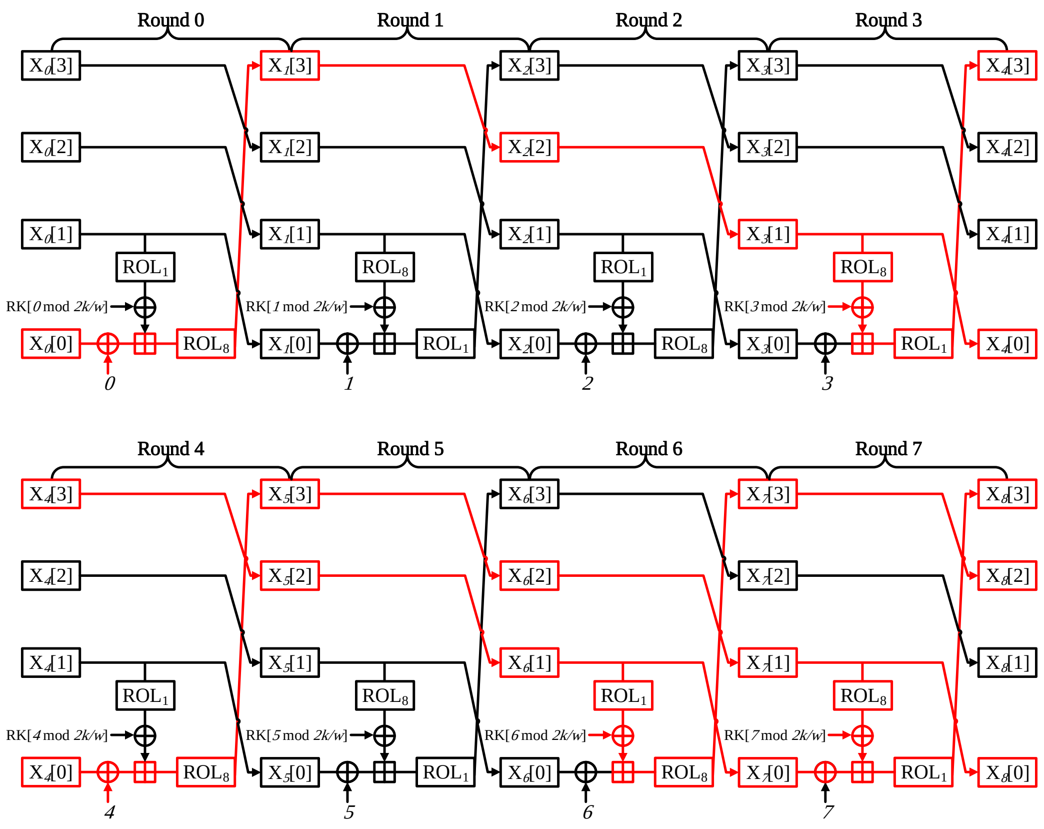


**Figure 1.** Path tracking for counter value.

- Round 0: Since $X_0[0]$ block is counter block, the pre-computation is not feasible as described above. In contrast, $X_0[1]$ which has the nonce value block, 1-bit left rotation, and XOR with round key operations are performed. At this time, both the nonce and round key are fixed, the result of the operation is always identical. The pre-computation is available at this part. This leads to a simplified round. LDI instruction is used for implementation; LDI instruction can directly assign the 8-bit value in a single clock cycle. This approach reduces 2 clock cycles then memory access in each 8-bit assignment;
- Round 1: Since all of Round 1 inputs are of a fixed value (i.e., nonce, round key, and round counter), whole operations can be skipped. The result of Round 1 is obtained by accessing the look-up table at $X_4[1]$ block during Round 4.
- Round 2: Similar to Round 1, Round 2 is also skipped. The result value is obtained from the look-up table to $X_5[1]$ of Round 5.
- Round 3:

$$((X_3[0] \oplus i) \boxplus ((X_3[1] \lll 8) \oplus \text{RoundKey}[3]) \lll 1)$$

  In Round 3, the $(X_3[0] \oplus i)$ part can be pre-computed. In more detail, $i$ stands for the round counter which is a fixed value. In Round 0, $X_0[3]$ is a fixed nonce value and it is moved to $X_3[0]$. Therefore, it can be seen that all values in the $(X_3[0] \oplus \text{i})$ part are a fixed value, which can be pre-calculated. It only requires a word assignment. $X_3[1]$ originated from $X_0[0]$ (i.e., counter). Thus $X_3[1]$ cannot be pre-computed;
- Round 4: Each block is word-wise rotate to the left at every end of a round. After Round 3, all blocks return to their own original place. In other words, Round 4 is structurally identical to

Round 0. Thus, computations with $X_4[1]$ (i.e., rotation left by 1-bit and add-round key) can be optimized;

- Round 5: It is able to skip all operations at Round 5, since it has the same structure of Round 1. The $X_5[0]$ result is brought to the look-up table access after Round 7. This takes 2 clock cycles.
- Round 6:

$$((X_6[0] \oplus i) \boxplus ((X_6[1] \lll 1) \oplus \text{RoundKey}[6]) \lll 8)$$

  The difference between Round 3 and Round 6 is only the number of rotation. In other words, it has almost the same structure of Round 3. Thus, the optimization is possible at the $X_6[0]$ operation part like Round 3;
- Round 7: In Figure 1, all values for the operation are affected by the counter value at Round 7. All operations must be implemented;
- Round 8: After Round 7, there is still a remaining part for pre-computation. In Round 8, $X_8[1]$ has no relation to the counter value. For this reason it can be pre-calculation. However, the performance improvement is 0.6, 0.7, and 0.7 clock cycles per byte for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively. This is a very small performance improvement. If the pre-calculation is applied after Round 8, the 8-way based structure will be broken. It makes the code size larger than the proposed implementation. Although there is an optimization room after the Round 7, it is not considered in this paper.

The optimized CHAM-CTR is shown in Figure 2. The blue line means the pre-calculation part and the red line is the counter value flow. The optimized design needs only 5 memory accesses for look-up table during Round 0, 3, 4, 6, and 7. Since the CHAM block cipher uses 8-bit word size, the optimized CHAM-CTR requires a 5-byte memory size for the look-up table.

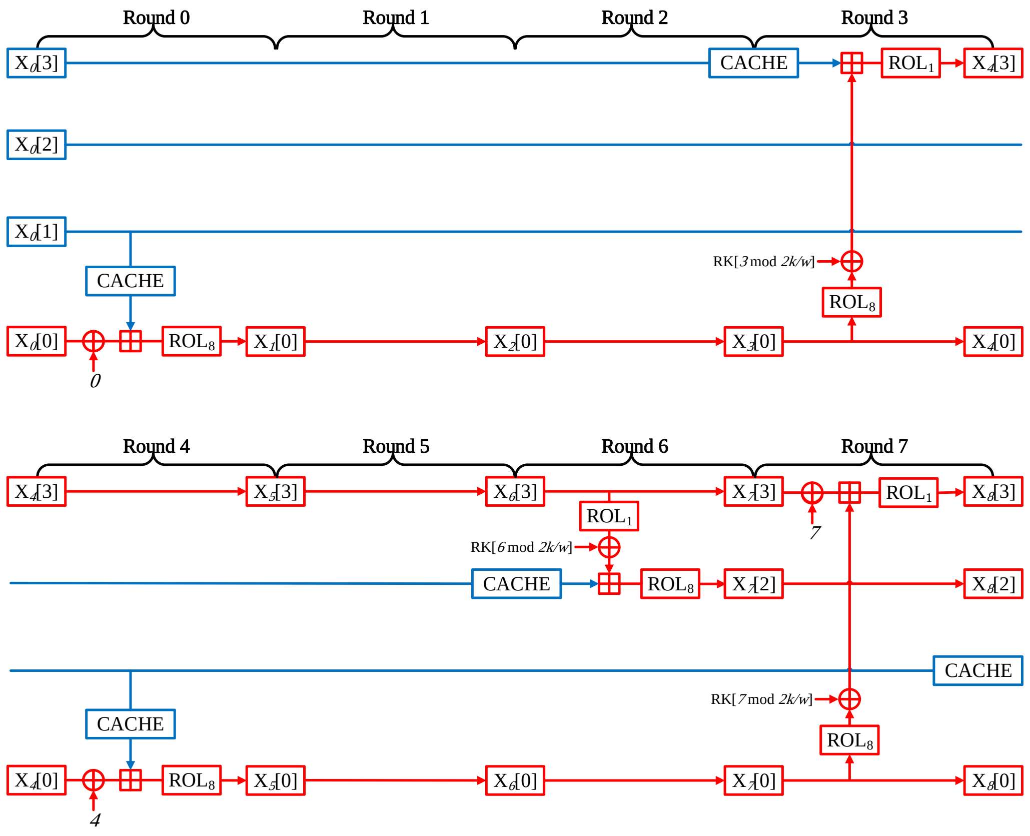


**Figure 2.** Optimized CHAM-Counter (CTR) mode encryption for round function.

### 5.1.1. 32-Bit Counter for CHAM-64/128

The formal counter mode of operation uses the 32-bit counter value. However, the word size of CHAM-64/128 is 16-bit wise. The 16-bit counter value is used for CHAM-64/128. The 32-bit counter for CHAM-64/128 is also required to support the high-security services. It is simply implemented by storing the counter value into two blocks as described in Figure 3. Since there are two counter blocks, the part for pre-computation is reduced. This shows a lower performance than CHAM-64/128 using a 16-bit counter. However, the 32-bit counter version needs a 4-byte memory size for the look-up table because its pre-computation part is lesser than the other implementations by one.

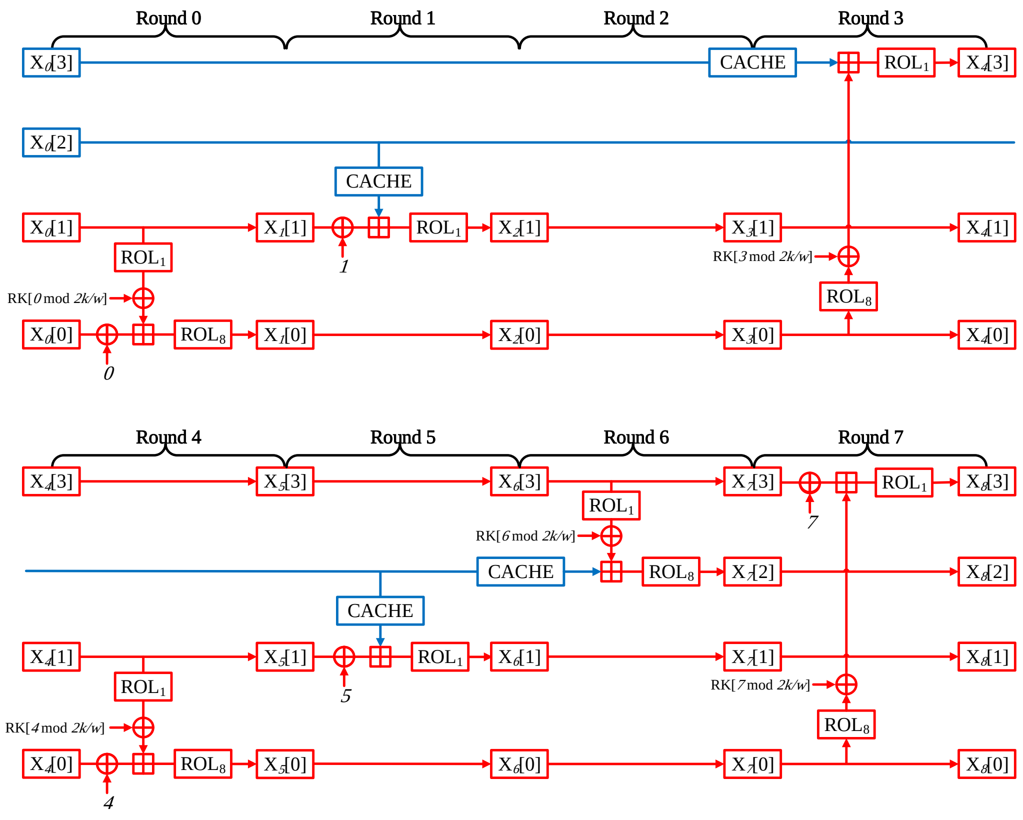


**Figure 3.** The counter value flow of 32-bit counter CHAM-64/128.

### 5.1.2. Countermeasure against Timing Attack

The optimized CHAM-CTR implementation has a resistance against timing attack. The timing attack is one of side-channel attack, utilizing the execution time of the algorithm. The proposed optimized CHAM-CTR implementation ensures that all round functions have constant execution time. The implementation has no conditional branch-statement. The proposed implementation has memory access, but there is no cache on the target AVR microcontrollers. For this reason, a cache timing attack is not available. For this reason, the implementation is resistant against timing attack.

### 5.1.3. Optimized Memory Access

Whole round keys are saved in SRAM. When the key is needed, it can be accessed through the LD instruction. This takes 2 clock cycles per byte for each operate. Aligning the round key in 8-bit wise, the offset of 16-bit address is only modulated with lower byte. To access the pre-calculated value, the LDI instruction is used. The LDI instruction requires only 1 clock cycle and does not need memory pointer control.

#### 5.1.4. Round Counter and Pointer Address Optimization

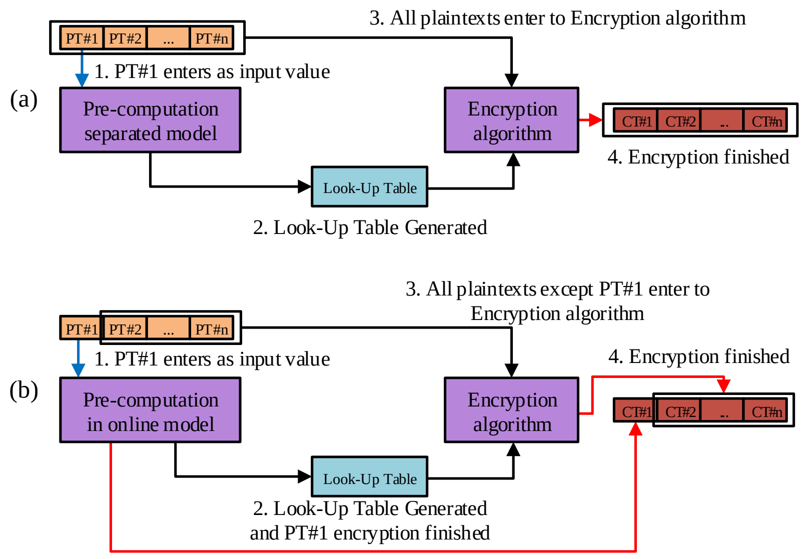
In every round, one of block performs the XOR operation with the round counter or round key. However, the optimized CHAM-CTR has only three times (i.e., Round 0, 4, and 7) of XOR operations with the round counter. The round key is used three times (i.e., Round 3, 6, and 7). Instead of adjusting the counter each time, the value is assigned directly. It benefits greatly in terms of computation time. With this approach, the result is as follows. First, 8 INC instructions for the round counter are replaced to the combination of 1 INC and 2 LDI instructions. Second, ADIW instructions for pointer address setting are reduced from 5 to 2. Third, the ADIW instruction adds for pointer address in multiples of 2 and 4 for CHAM-64/128 and CHAM-128/128 or CHAM-128/256, respectively.

### *5.2. Variable-Key Based CHAM-CTR Implementation*

In Section 5.1, the fixed-key based implementation was covered. Under the fixed-key scenario, the LDI instruction is efficient for high performance. Since the LDI instruction can only assign the fixed value to the target register, the LDI instruction cannot be used in variable-key based implementation where the look-up table should be updated, frequently. In the IoT environment, the fixed-key scenario is not practical. After a number of encryption, the key and nonce should be updated. If not, the information will be repeated and become vulnerable toward potential attacks. For this reason, the variable-key based implementation is required for the practical usage.

For variable-key scenario implementation, the following should be considered. The look-up table is updated frequently. When the key is changed, the look-up table is updated again. The variable-key based implementation has lower performance than fixed-key based implementation. The variable-key based implementation needs to update the pre-calculation table. When the key is updated, the encryption is still available. The variable-key implementation is more suitable than the fixed-key implementation in terms of practical use.

The implementation can be done in two different types. The first one is the separated version for the pre-calculation. The second one is pre-calculation in online model. In Figure 4a shows that the separated version calculates only the look-up table result. Furthermore, the register alignment is given in Figure 5a. The separated version computes the look-up table with minimal instructions. To be more specific, the separated version does not run all rounds of CHAM, only up to 7 rounds. Besides, during 7 rounds, instructions irrelevant to look-up table calculation are removed. When the pre-computation is finished, the look-up table value is used to start encryption. In Algorithm 1, the separated version is given. In Step 1∼7, the look-up table is generated from the first plaintext. In Step 8∼10, the encryption proceeds from the first plaintext to the last plaintext. In Step 11, the result is returned. On the other hand, in Figure 4b, the pre-calculation in the online version encrypts while calculating the look-up table. In addition, the register alignment is represented in Figure 5b. In this time, the look-up table calculation is performed simultaneously with encryption process. The encryption is operated with the first plaintext block, while calculating the look-up table. Consequently, unlike the separated version, the look-up table is calculated through all encryption processes. The creation time of the look-up table is slightly slower than the separated version. However, the encryption of the first plaintext block is performed while creating the look-up table. Thus, the performance of encryption is better than the separated version from the second plaintext block. In Algorithm 2, the integrated version is presented. In Step 1∼7, the look-up table is generated from first plaintext. In Step 8, first plaintext encrypted after look-up table calculation. In Step 9∼11, the encryption proceeds from second plaintext to last plaintext. In Step 12, the result is returned.



(a)
PT#1 PT#2 ... PT#n
3. All plaintexts enter to Encryption algorithm
1. PT#1 enters as input value
Pre-computation separated model
Encryption algorithm
CT#1 CT#2 .. CT#n
4. Encryption finished
Look-Up Table
2. Look-Up Table Generated
(b)
PT#1 PT#2 ... PT#n
3. All plaintexts except PT#1 enter to Encryption algorithm
1. PT#1 enters as input value
Pre-computation in online model
Encryption algorithm
4. Encryption finished
CT#1 CT#2 .. CT#n
Look-Up Table
2. Look-Up Table Generated and PT#1 encryption finished

**Figure 4.** Work flow for (**a**) separated model and (**b**) pre-computation in online implementations.

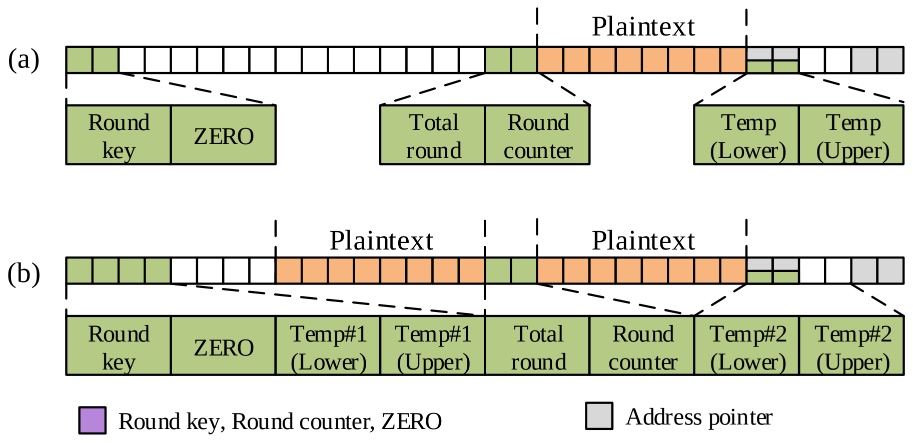


**Figure 5.** Register alignment for (**a**) separated model and (**b**) pre-computation in online implementations. Each block represents one register. Two colors in one register is used for multiple purposes.

**Algorithm 1:** Separated version for variable-key implementation.

```
Input: Number of blocks $n$, Plaintext blocks $P \in \{P_1, P_2, ..., P_n\}$.
Output: Ciphertext blocks $C \in \{C_1, C_2, ..., C_n\}$.
1: $LUT[0] \leftarrow Pre\_Computation\_Round0(P_1[0])$        //pre-computation start
2: $Pre\_Computation\_Round1(P_1[1])$
3: $Pre\_Computation\_Round2(P_1[2])$
4: $LUT[1] \leftarrow Pre\_Computation\_Round3(P_1[3])$
5: $LUT[2] \leftarrow Pre\_Computation\_Round4(P_1[4])$
6: $LUT[3] \leftarrow Pre\_Computation\_Round5(P_1[5])$
7: $LUT[4] \leftarrow Pre\_Computation\_Round6(P_1[6])$   //pre-computation end at CHAM Round 6
8: for $i = 1$ to $n$ do
9:     $C_i \leftarrow ENC(P_i, LUT)$        //encryption from $P_1$ to $P_n$
10: end for
11: return $C$
```

**Algorithm 2:** Integrated version for variable-key implementation.

```
Input: Number of blocks $n$, Plaintext blocks $P \in \{P_1, P_2, ..., P_n\}$.
Output: Ciphertext blocks $C \in \{C_1, C_2, ..., C_n\}$.
1: $LUT[0] \leftarrow Pre\_Computation\_Round0(P_1[0])$          //pre-computation start
2: $Pre\_Computation\_Round1(P_1[1])$
3: $Pre\_Computation\_Round2(P_1[2])$
4: $LUT[1] \leftarrow Pre\_Computation\_Round3(P_1[3])$
5: $LUT[2] \leftarrow Pre\_Computation\_Round4(P_1[4])$
6: $LUT[3] \leftarrow Pre\_Computation\_Round5(P_1[5])$
7: $LUT[4] \leftarrow Pre\_Computation\_Round6(P_1[6])$
8: $C_1 \leftarrow Pre\_Computation(P_1)$
   // After Round 7, remaining rounds are performed with $P_1$ for encryption
9: for $i = 2$ to $n$ do
10:    $C_i \leftarrow ENC(P_i, LUT)$          //encryption start from $P_2$ to $P_n$
11: end for
12: return $C$
```

## 5.3. Parallel Implementations of CHAM Block Cipher

Parallel implementation is helpful in improving performance because it can generate multiple ciphertexts at once. In this paper, two kinds of parallel implementations have been researched, which are 2-parallel and 3-parallel. The implementation requires caching of intermediate results, thus a register utilization should be optimized.

CHAM-64/128 using 64-bit plaintext and target 8-bit AVR microcontrollers has 32 8-bit registers. That is to say, CHAM-64/128 requires 8 registers for plaintext blocks. Simply put, 2-parallel implementation needs 16 registers since there are 2 plaintexts. Furthermore, control variables (i.e., round counter, round key, and address pointer) require their own registers. Figure 6a represents the register utilization of 2-parallel implementation. According to Figure 6a, whole values can be assigned in registers.

By way of contrast, since 3-parallel implementation uses 3 plaintexts, it requires 24 registers. At this point, we face the problem of a lack of registers. To overcome the limitations of 8-bit AVR microcontrollers, STACK memory is utilized and some registers are redesigned for multiple purposes. Detailed optimization techniques are as follows:

- Total round indicator: Instead of storing the total round indicator in a register, use CPI instruction;
- 3-Plaintext blocks: Every round, the only two blocks (i.e., X$_i$[0], and X$_i$[1]) are participating encryption. Thus, other plaintext blocks are stored in STACK memory for a short time;
- Address pointer X: Round keys and plaintext can be obtained by accessing them through an address pointer. However, there are only two plaintext blocks participating in the operation in every round. Thus, some plaintext address pointers are stored on STACK;
- Round key: Originally, round key access require word-wise memory access. If accessing byte by byte, instead of word-wise access, only one register requires to load round key;
- Round counter: Round counter is used only when performing XOR operation. After XOR operation, it is stored in STACK;
- ZERO: For 3-parallel implementation, R1 register is used for plaintext, which originally assigned ZERO register. Some registers are used as ZERO register for a while.

The register utilization scheme for 3-parallel implementation is given in Figure 6b.

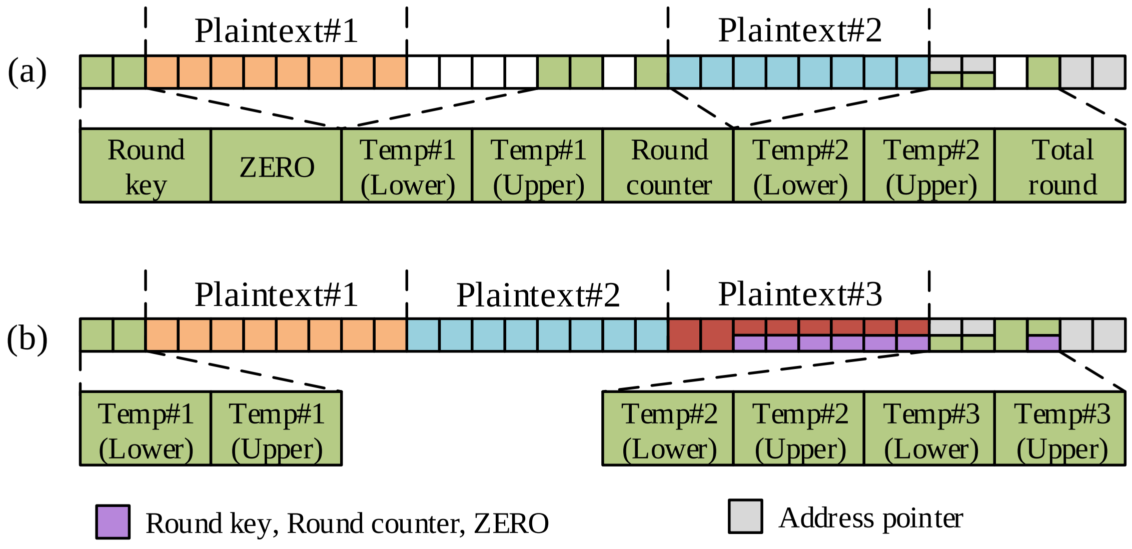


**Figure 6.** Register alignment for (**a**) 2-parallel and (**b**) 3-parallel of CHAM-64/128 implementations. Each block represents one register. Two color in one register is used for various purposes.

In Algorithm 3, the parallel computation for CHAM-64/128 is given. Step 3∼7, left rotation operations are performed depending on if the counter is even or odd. In Step 8, addition with round key is operated. At this point, using these round keys, multiple blocks are calculated in parallel routine. Step 11∼15, left rotation operations are performed depending on if the counter is even or odd.

**Algorithm 3:** Parallel implementation of CHAM-64/128.

```
Input: Plaintext blocks ($X[0][0 \sim 3], ..., X[\#parallel-1][0 \sim 3]$).
Output: Ciphertext blocks ($X[0][0 \sim 3], ..., X[\#parallel-1][0 \sim 3]$).
1:  for $i = 0$ to $\#round$ do
2:    for $j = 0$ to $\#parallel$ do
3:      if $i$ mod $2 == 0$ then
4:        $tmp[j][0] \leftarrow ROL_1(X[j][1])$
5:      else
6:        $tmp[j][0] \leftarrow ROL_8(X[j][1])$
7:      end if
8:      $tmp[j][1] \leftarrow tmp[j][0] \oplus RK[i \bmod 16]$          //round key access optimization
9:      $tmp[j][2] \leftarrow X[j][0] \oplus i$
10:     $tmp[j][3] \leftarrow tmp[j][1] \boxplus tmp[j][2]$
11:     if $i$ mod $2 == 0$ then
12:       $tmp[j][4] \leftarrow ROL_8(tmp[j][3])$
13:     else
14:       $tmp[j][4] \leftarrow ROL_1(tmp[j][3])$
15:     end if
16:   end for
17: end for
```

In case of CHAM-128/128 and CHAM-128/256, it is hard to implement the parallel version, since there are limitations on the number of registers. CHAM-128/128 and CHAM-128/256 have plaintext, which is twice as lengthy than CHAM-64/128. To conclude, parallel implementation for CHAM-128/128 and CHAM-128/256 are not considered.

### *5.4. Adaptive Encryption of CHAM Block Cipher*

The parallel operation is practical for massive data controlling. It can be involved to adaptive encryption [16]. The adaptive encryption operates parallel encryption when data is long enough. If only single block remained, single block encryption is performed. Algorithm 4 is given that circumstantial descriptions for 2-way adapted encryption. Step 3∼5, parallel calculation is operated. Subsequently, Step 6∼8, sequential computation is performed for leftover data.

**Algorithm 4:** 2-way adaptive encryption for CHAM64/128.

```
Input: Number of blocks $N$, Plaintext blocks $P \in \{P_1, P_2, ..., P_N\}$.
Output: Ciphertext blocks $C \in \{C_1, C_2, ..., C_N\}$.
1: $n \leftarrow \frac{N}{2}$
2: $m \leftarrow N \bmod 2$
3: for $i = 0$ to $n$ do
4:     $\{C_{2 \cdot i+1}, C_{2 \cdot i}\} \leftarrow ENC(\{P_{2 \cdot i+1}, P_{2 \cdot i}\})$        //parallel computation
5: end for
6: if $m$ then
7:     $C_{2 \cdot n+2} \leftarrow ENC(P_{2 \cdot i+2})$        //sequential computation
8: end if
9: return $C$
```

### *5.5. Optimized Implementations of Primitive Operations*

Proposed optimized CHAM-CTR is implemented on 8-round based implementation. In each 8 round, the 8-bit rotation operation on data for CHAM-64/128 is the optimized approach. On the other hand, 16-bit wise rotation operation is still needed for CHAM-128/128 and CHAM-128/256. It is performed in 16-bit wise move operation MOVW.

## 6. Evaluation

Proposed implementations of CHAM block cipher were tested on the target device (i.e., low-end 8-bit AVR microcontrollers). The performance was evaluated in execution time (clock cycles per byte). The software was implemented over Atmel Studio 7 and the code was complied in -O2 option.

In Figure 7, the comparison result of optimized CHAM-CTR block cipher is given. First of all, the performance was compared with previous works by [3], proposed implementations of CHAM-64/128, CHAM-128/128, and CHAM-128/256 have better performance by 11.8%, 6.6%, and 7.4%, respectively. The performance improvement comes from the 8-round based implementation and fast memory access for round keys. This is implemented in two scenarios, including fixed-key scenario and variable-key scenario. Results of fixed-key scenario are compared. The implementation improved the performance further by 15.5%, 13.4%, 9.3%, and 10.0% for CHAM-64/128 (16-bit counter), CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively. The implementation utilized the repeated or fixed values (i.e., nonce, round counter, and round key) to improve the performance. Second, variable-key scenario implementations. There are two types under variable-key scenario: Pre-computation in offline (i.e., separated version) and online. The separated version shows almost the same performance as [3]. Since this approach calculates a table through pre-calculation and then proceeds with the encryption with the table. This approach has more operations than the counter mode of operation-based implementation. If the encryption is more than two blocks, it outperforms the original implementation. For the same reason, the pre-computation in the online version has a similar performance as [3]. The look-up table based implementation improved the performance about 14.4%, 12.8%, 8.9%, and 9.6% for each CHAM-64/128 (16-bit counter), CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively. This result is similar to fixed-key scenario because `LD` instruction is used instead of `LDI` instruction. Since the variable-key scenario can be used when the key is changed. This is a much more practical form of implementation than fixed-key scenario.

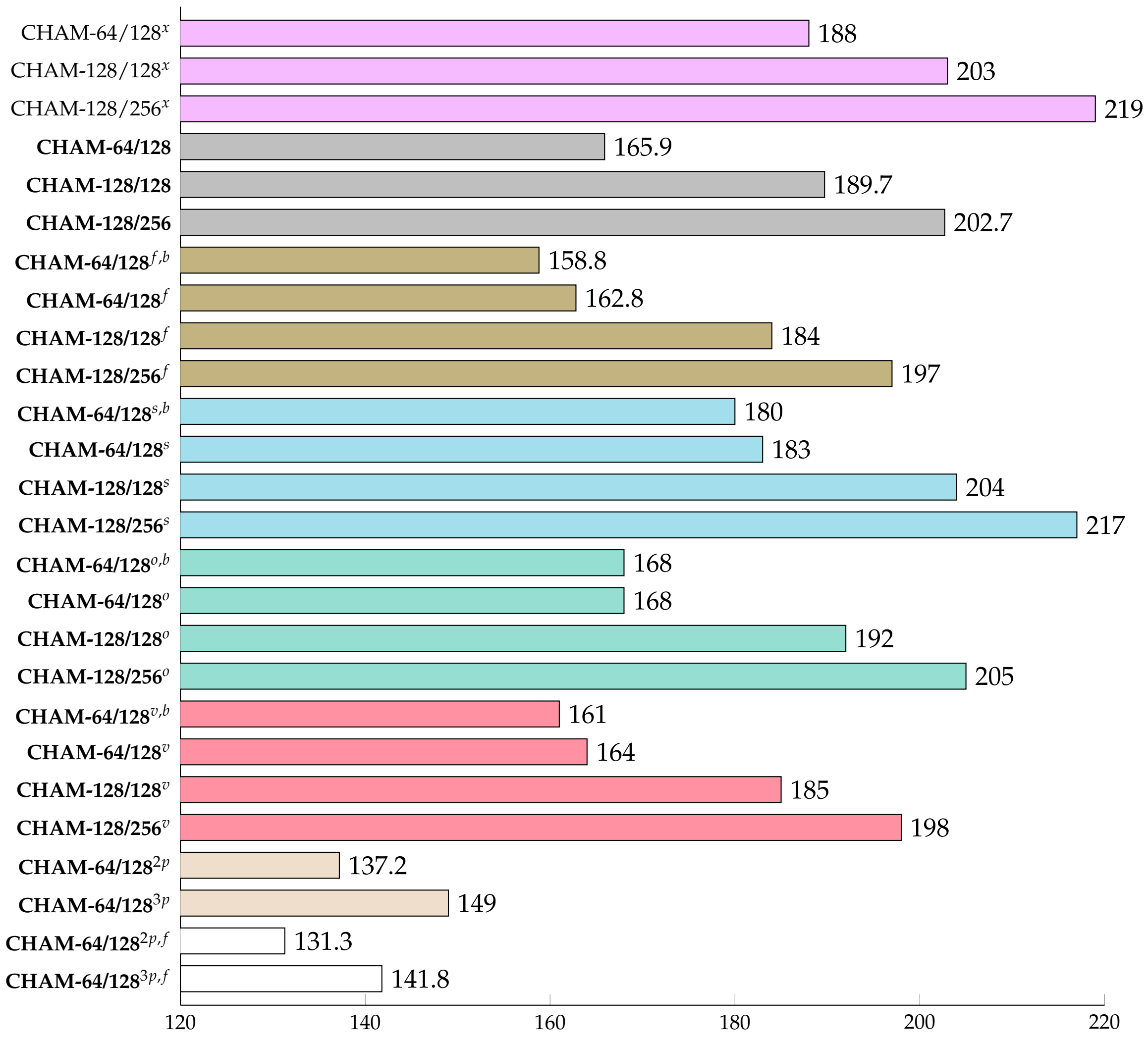


**Figure 7.** Comparison of execution time for CHAM implementations on 8-Bit AVR microcontrollers under the fixed-key scenario in terms of clock cycles per byte, *b*: 16-bit counter for CHAM-64/128, *f*: Fixed-key scenario, *s*: Pre-computation in separated way, *o*: Pre-computation in online, *v*: Variable-key scenario (using look-up table which is generated by *s* or *o*), 2*p*: 2-parallel, 3*p*: 3-parallel, *x*: [3].

The parallel implementation of CHAM-64/128 shows 137.2 and 149 clock cycles per byte for 2-parallel and 3-parallel implementation, respectively. Due to the limited number of registers, 2-parallel based implementation shows better performance than that of 3-parallel. In addition, the performance is slightly improved when the pre-computation is applied to the parallel implementation.

We compared the performance result with other block ciphers in Figure 8. AES implementation achieved the best performance among implementations because AES is designed for 8-bit word architecture. Among ARX based implementations, SPECK shows the fastest performance. Proposed CHAM-CTR block cipher implementation is ranked in second place.

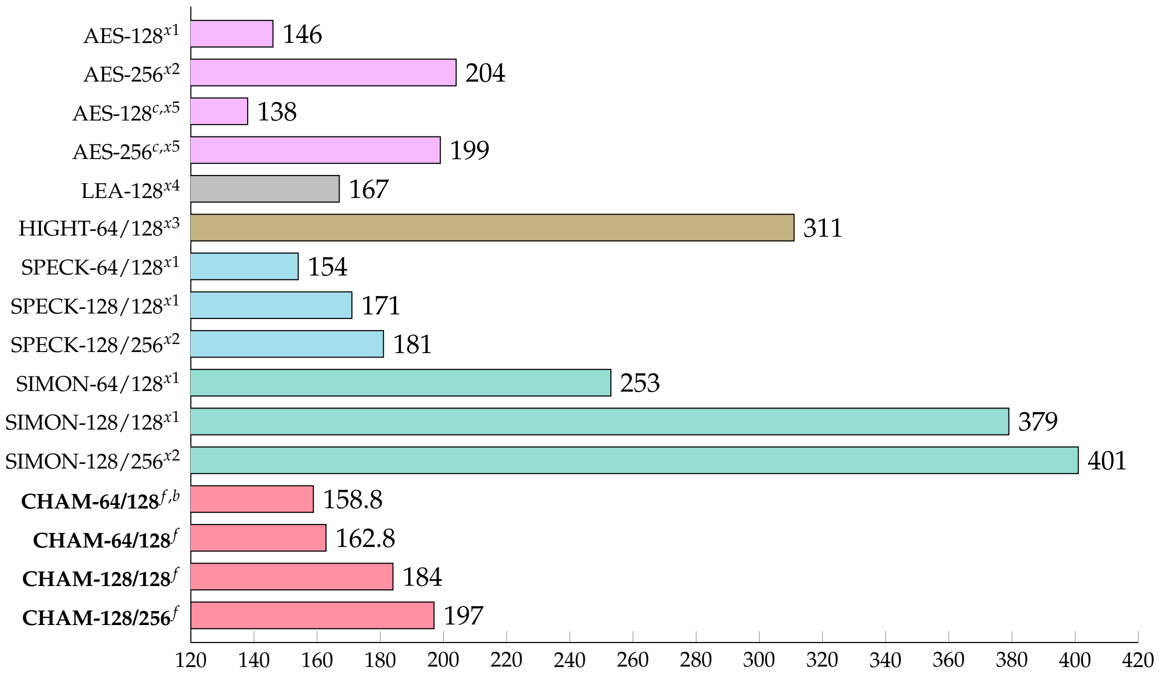


**Figure 8.** Comparison of execution time for CHAM with other block ciphers on 8-Bit AVR microcontrollers in terms of clock cycles per byte, *c*: Counter mode of operation, *b*: 16-bit counter for CHAM-64/128, *f*: Fixed-key scenario, *x*1: [17], *x*2: [18], *x*3: [8], *x*4: [5], *x*5: [13,14].

## 7. Conclusions

In this paper, we concentrated on the high-speed implementation of CHAM-CTR. We presented optimization implementations of lightweight CHAM block cipher on low-end 8-bit AVR microcontrollers. Proposed techniques consisted of pre-calculated counter mode of operation, parallel implementation, on-the-fly table construction, and optimized primitive operations. Unlike previous WISA'20 implementation, the proposed implementation can be operated under the variable-key environment. The presented work showed slightly lower performance than the WISA'20 implementation. However, the previous work has a limitation that it can only be operated on a fixed-key. On the other hand, the proposed implementation supported a variable-key environment, which is useful for practical use. The proposed implementation depends on CHAM block cipher and CTR mode of operation, but optimization techniques for this implementation can be applied to other ARX-based block ciphers. In addition, it can be utilized in the Galois/Counter Mode (GCM) protocol that is most commonly used for Transport Layer Security (TLS). We evaluated proposed implementations in terms of execution time. The result shows that proposed implementations achieved a high performance and fast execution timing for practical IoT applications.

Future work is applying the presented method to other ARX-based block ciphers (i.e., SIMON and SPECK). Moreover, the optimized implementation on other microcontrollers such as 16-bit MSP and 32-bit ARM will be considered.

**Author Contributions:** Investigation, H.K. (Hyeokdong Kwon), H.K. (Hyunji Kim), S.A., Y.K. and J.P.; Software, H.K. (Hyeokdong Kwon), K.J., S.A., Y.K., H.K. (Hyunjun Kim) and H.S.; Supervision, H.S.; Writing—original draft, H.K. (Hyeokdong Kwon), H.S., K.J.; Writing—review and editing, H.K. (Hyeokdong Kwon), S.J.C., S.C.S. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was partly supported as part of Military Crypto Research Center(UD170109ED) funded by Defense Acquisition Program Administration(DAPA) and Agency for Defense Development(ADD) and this work was partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2020R1F1A1048478) and this work 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 work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1F1A1058494) for SangWoo An, YoungBeom Kim, and SeogChung Seo.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Kwon, H.; Kim, H.; Choi, S.; Jang, K.; Park, J.; Kim, H.; Seo, H. Compact Implementation of CHAM Block Cipher on Low-End Microcontrollers. In *International Workshop on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2020.
2. 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 (ICISC'17), Seoul, Korea, 29 November–1 December 2017.
3. 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, Nanjing, China, 6–8 December 2019; pp. 1–19.
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; pp. 28–40.
5. 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; pp. 253–265.
6. 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; pp. 3–27.
7. 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]
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; 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, Ifrane, Morocco, 10–12 July 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. 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.
12. Osvik, D.A.; Bos, J.W.; Stefan, D.; Canright, D. Fast software AES encryption. In Proceedings of the International Workshop on Fast Software Encryption, Graz, Austria, 15–17 March 2010; pp. 75–93.
13. 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; pp. 102–114.
14. 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]
15. Seo, H.; Kwon, H.; Kim, H.; Park, J. ACE: ARIA-CTR Encryption for Low-End Embedded Processors. *Sensors* **2020**, *20*, 3788. [CrossRef] [PubMed]
16. Park, T.; Seo, H.; Lee, S.; Kim, H. Secure data encryption for cloud-based human care services. *J. Sens.* **2018**, *2018*, 6492592. [CrossRef]

17. Beaulieu, R.; Shors, D.; Smith, J.; Treatman-Clark, S.; Weeks, B.; Wingers, L. SIMON and SPECK: Block Ciphers for the Internet of Things. *IACR Cryptol. Eprint Arch.* **2015**, *2015*, 585.
18. 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; pp. 3–20.