ARX-based Korean Block Ciphers with CTR Mode and CTR_DRBG on Embedded Processors and GPU

No Author Given

No Institute Given

Abstract. As the development of Internet of Things (IoT), the volume of transmitted data has been significantly increased. To protect sensitive data including users' personal information, it is necessary to encrypt the data in secure and efficient way. Since IoT services make use of a number of heterogeneous devices, such as wireless sensors, smart phones, and application servers, dealing with a number of IoT clients. Under this condition, it is required to optimize the performance of cryptographic algorithms which are computation-intensive tasks for providing communication security. In this paper, we present efficient implementations of ARX-based Korean Block Ciphers (HIGHT, LEA, and revised CHAM) with CounTeR (CTR) mode of operation and CTR_DRBG, one of the most widely used DRBGs (Deterministic Random Bit Generators), on IoT devices. As target devices, we select 8-bit AVR microcontrollers (MCUs) and CUDA-enabled GPU as a representative resource-constrained client device and a high-end application server in IoT services. We optimize the performance of target block ciphers by utilizing the property of CTR mode that the nonce value is fixed and the only counter value changes during the encryption. By using this property, the computation of the initial few rounds can be omitted, which results in performance improvements. With respect to CTR_DRBG, we identify several parts that do not need to be computed. Thus, pre-computing those parts in offline and using them in online can result in performance improvements of CTR_DRBG. Furthermore, we apply several optimization techniques by making the full use of target devices' characteristics with AVR assembly codes on 8-bit AVR MCUs and PTX assembly codes on CUDA-enabled GPUs. Our optimized revised CHAM(except in case of 16-bit counter), LEA, and HIGHT with CTR mode on 8-bit AVR provides 12.8%, 5.9%, and 3.8% of improved performance in the best case, compared with the most efficiency previous results, respectively. The proposed CTR_DRBG implementation on AVR provides better performance by 37.2%, 21.1%, and 8.7% when the underlying block cipher is LEA, revised CHAM, and HIGHT, respectively. On GPU environments, our block cipher implementations on CTR mode with revised CHAM, LEA, and HIGHT provide 2,219, 664, and 477 times of improved performance compared with CPU implementations. Our GPU-based CTR_DRBG with CHAM, LEA, and HIGHT is capable of generating secure random bits at least 34.2, 24.8, and 52.2 times enhanced speed compared over CPU implementation, respectively.

Keywords: Counter Mode of Operation · Block Cipher · ARX · Efficient Implementation · CTR_DRBG · AVR · GPU · CUDA

1 Introduction

As the advent of Internet of Things (IoT), new types of services have been available. In IoT services, a number of small and wireless devices around users collect user data or surrounding information and transmit the information to the connected application server. Since the collected data is transmitted in wireless communication, the data can be easily revealed to attackers. Thus, the data needs to be properly encrypted by secure cryptographic algorithm before transmission. In other words, since it is required to transmit data in encrypted form rather than original form, applying cryptographic algorithm is fundamental for providing robust and secure communication in IoT services. However, implementing cryptograhic algorithm on IoT devices is not easy task because typical client devices in IoT equip only limited CPU, RAM, and ROM. For example, a 8-bit AVR-based sensor node used in wireless sensor networks (WSNs) has 4KB of RAM, 128KB of ROM, and 7.3728 MHz of computing capability. Furthermore, since application servers need to securely provide services with a number of connected clients, they have to execute a number of cryptographic operations concurrently. GPUs (Graphic Processing Units) have been considered as proper candidate for crypto accelerator in situation that needs to handle a number of concurrent cypto operations.

Recently, several lightweight block ciphers have been developed for efficient performance on IoT devices. The lightweight block ciphers ensures low-cost and simple computations and their computational structure is based on ARX (Addition, Rotation, and XOR) architecture. Actually, in South Korea, three ARX-based lightweight block ciphers have been developed such as HIGHT [10], LEA [9], and CHAM [15, 23]. When data is larger than basic processing block (64-bit in HIGHT, 128-bit in LEA and CHAM), mode of operation needs to be applied. There are several mode of operations and CTR mode is the most popular among them. Until now, these ARX-based lightweight block ciphers have been optimized on various platforms including embedded MCUs, GPU-enabled servers, etc.

In addition to applying block ciphers to data to be transmitted, it is also important to securely generate secret keys used in block ciphers. If weak key is used in a block cipher, the ciphertext can be revealed to attackers even if the underlying block cipher is secure. DRBGs (Deterministic Random Bit Generators) are widely used to generate secret information including keys. Among standardized DRBGs, CTR_DRBG provides strong resistance against backward security and forward security. Since CTR_DRBG makes use of CBC-MAC and CTR mode of operation in its derivation function and output generation function, it takes execution time larger than executing block cipher algorithms. Thus, optimizing the performance of CTR_DRBG on IoT clients and servers is important for constructing secure and robust communicationse between clients and servers in IoT services.

In this paper, we present optimized implementation of ARX-based lightweight block ciphers (HIGHT, LEA, and CHAM) on CTR mode and CTR_DRBG using them on 8-bit AVR Microcontrollers and GPU-enabled PCs as target devices of IoT client and server, respectively. With respect to efficient implementation of CTR mode with the target block ciphers, we present optimization methodology using CTR mode's property that in Initial Vector (IV) the only counter part except for the nonce part changes. By pre-computing some values by using this property, the computation of the initial few rounds can be omitted, which result in performance improvement. Regarding CTR_DRBG optimization, we identify several parts that can be pre-computed. In other words, pre-computing those parts and using them when executing CTR_DRBG gives performance improvement. In addition, we apply several optimization techniques by using both block ciphers' algorithmic characteristics and target devices' properties.

The following subsection describes our contribution points and presents improved performance with respect to CTR mode of operation and CTR_DRBG on 8-bit AVR MCUs and GPUs.

1.1 Contribution

The contributions of this paper are given as follows.

Fast counter mode of operation for Korean block ciphers on 8-bit AVR MCUs In this paper, counter mode of operation for Korean block ciphers such as CHAM, LEA, and HIGHT are optimized to improve performance by taking advantages of static data (nonce) in input data. For optimal implementation, we made of use characteristics of the CTR mode input values. CTR mode of operation use nonce and counter value as input. As this moment, nonce is fixed value and counter is refers to block number. For this reason, result always has identical when nonce part encrypted. From on optimal implementation view, if the calculation result is identical, encryption performance can be improved by skipping calculation procedure, and loading only the calculation result from cache table. Also, led to better performance through optimization of rotation operation and efficiently memory access. To conclude, this paper achieved CHAM, LEA, and HIGHT the Korean block ciphers about 8.9∼5.8%, 6.1∼5.0%, and 3.9% higher performance respectively than the previous works.

Fast counter mode of operation for Korean block ciphers on CUDA-enabled GPU Significant performance improvement could be achieved by computing multiple plaintext blocks in parallel in the GPU environment for targeted Korean block ciphers. In addition to simple parallel encryption, the counter value is replaced by plaintext using the thread ID, a unique number inside the GPU, to significantly reduce the memory that needs to be copied from the CPU to the GPU. Also, by using the CUDA stream command to operate the kernel through asynchronous execution, the CPU idle time during GPU operation was reduced to further improve performance. Inside the GPU, memory access time is optimized by utilizing various memories such as shared memory. In addition, by utilizing the features of the CTR mode, it is possible to reduce encryption rounds by 2, 3, and 2 rounds, respectively, inside the GPU while eliminating unnecessary calculations for blocks that do not change during a certain round. This led to about 2, 6 and 5% performance improvements in internal encryption time, respectively. By applying various GPU optimization methods, up to 95, 59, and 67% performance improvement was achieved for each of CHAM, LEA, and HIGHT compared to a simple parallel implementation when memory copy time was included. When only the encryption performance was excluded except the memory copy time, the performance was up to 2219, 664, and 477 times faster than the existing CPU implementation. Since CHAM and HIGHT have few existing studies optimized through GPU, the results of this optimization study can be used as indicators for other studies.

Optimized CTR_DRBG Implementation for fast random bit generation on 8-bit AVR MCUs For the constant data generated in the Derivation Function and Update Function, the performance improvement of CTR_DRBG was achieved using the Look-Up table. In CBC-MAC of Derivation Function, a Look-Up table is created for the encryption results of data as much as the initial Block Bit, and the optimization of Update Function was achieved using Look-Up table using the initial Operational Status is zero. In addition, the Look-Up table does not require an update, but also requires a low cost of 96-byte, making it effectively applicable to 8-bit AVR-Microcontroller environments. Moreover, we present methods to optimize Korean block cipher in Extract Function for 8-bit AVR Microcontroller. Extract Function is optimized by utilizing table in CTR mode that uses fixed keys to effectively reduce block cycle. Optimized Derivation Function shows up to 13.3% performance improvement, and optimized Update Function shows up to 72.4% performance improvement. By applying CTR optimization methods, up to 13.6, 36.4, and 3.5% performance evaluation was applied for approach of CHAM, LEA, and HIGHT compared to standard implementation of Extract Function. The optimization method of Derivation Function and Update Function has the advantage of being applicable in various platforms as well as 8-bit AVR Microcontroller. In addition, the optimization method applied to Extraction Function has the advantage of creating Look-Up tables during a series of encryption processes in a limited environment, 8-bit AVR Microcontroller.

Optimized CTR_DRBG Implementation providing high throughput of random bit generation on CUDA-enabled GPUs CTR_DRBG is optimized with the GPU by utilizing the optimization technique of the CTR operation mode, and significant performance improvement is confirmed. Partial constants in the derivative and update functions are applied in the same way, and the plaintext input in the extract function was performed by using the counter value so that random numbers could be calculated and output in parallel. In the random number output process, a single CTR_DRBG function is called, and the long random number output part is executed by the GPU. As a result of the optimization implementation using GPU for CTR_DRBG, it is confirmed that the performance was up to 34.2, 24.8, and 52.2 times faster than the existing CPU implementation based on CHAM-128/128, LEA-128/128, and HIGHT-64/128, respectively. Also, when generating random numbers from 1 MB to 128 MB, it is shown that the performance increases up to 5 times as the size of random numbers output from one CTR_DRBG increases.

The remainder of this paper is organized as follows. In Section 2, target platform, target Korean block ciphers, target mode of operation, CTR_DRBG, and previous implementations are given. In Section 3, optimized implementation of counter mode of operation is presented. In Section 4, optimized implementation of CTR_DRBG is presented. In Section 5, the performance is evaluated. Finally, Section 6 concludes the paper.

2 Related Works

2.1 Target Platforms

Low-end Platform: 8-bit AVR Microcontroller AVR is a modified Havard architecture 8-bit RISC single-chip microcontroller [17]. An AVR microcontrollers find many applications as embedded systems, such as IoT devices, sensor nodes, and Arduino development boards. The ATmega128 microcontroller is one of AVR family that supports an 8-bit instruction set, 128 KB `FLASH` memory, 8 MHz working frequency, two-stage pipeline design, and 4 KB RAM. The number of available registers is 32. Among them, 6 registers (i.e `R26` $\sim$ `R31`) are reserved for address pointers, and the remaining registers are used for general purpose registers. Moreover the 133 kinds of instructions are served, they are most takes one clock cycle, while the memory access takes two clock cycles per byte.

The most important function of AVR CPU core is to secure precise program execution. Thus CPU can perform the following functions; Memory accessing, calculations performing, peripheral controlling, and interrupts handling. AVR applied Harvard architecture that is memories and buses of program and data are separated. Consequently, AVR has powerful performance and parallelism possible. Instructions are executed with a single level pipe-lining. As a detail, if one instruction is being performed, then the follow instruction is pre-fetched from the program memory. This structure ensures that instructions always can be executed in single cycle.

There are 32 general purpose registers with a single clock cycle access time. Of the 32 registers, 6 registers used for 16-bit indirect address register pointer for addressing, which they are called X-register, Y-register, and Z-register, respectively. Also these address pointers can be used as a pointer for Flash Program memory.

Table 1: Instruction set summary for efficient CHAM/LEA/HIGHT implementations on 8-bit AVR microcontrollers.

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
LSR	Rd	Logical Shift Right	Rd|C ← 1>>Rd	1
ROL	Rd	Rotate Left Through Carry	C|Rd ← Rd<<1||C	1
ROR	Rd	Rotate Right Through Carry	Rd|C ← C||1>>Rd	1
BST	Rd, b	Bit store from Bit in Reg to T Flag	T ← Rd(b)	1
BLD	Rd, b	Bit load from T Flag to a Bit in Reg	Rd(b) ← T	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

High-end Platform: GPGPU Even though Graphics Processing Units (GPU) was originally developed for graphic and image processing, nowadays they are widely used for general purpose applications including acceleration of crypto operations, machine learning, and so on. The NVIDIA is a representative GPU manufacturer, and GPUs produced by NVIDIA are classified according to their architectures. NVIDIA TITAN RTX, the flagship GPU of Turing architecture released in 2018, among many architectures, has 4,608 CUDA cores with 1,770 MHz boost clock. It also has 24 GB of GDDR6 graphics memory and has a memory clock of 1,750 MHz.

The CPU uses most of the chip area for cache, while the GPU uses most of the area for arithmetic logic units (ALU). GPUs use hardware threads that run the same instruction stream on different data sets. There are multiple streaming multiprocessors (SM) within the GPU, and a collection of threads running on one multiprocessor is called a block. Therefore, the number of all threads running inside the GPU is the product of the number of blocks and the number of threads per block. GPUs utilize these numerous threads to perform high-level parallelism in their applications.

GPU memory is made up of many different types. Figure 1 shows the memory structure of GPU devices. Basically, GPU has a memory area shared by all threads, global memory, constant memory, and texture memory. Since these memory types are the first memory areas to be accessed in data copy with the CPU using PCIe, so the memory size is very large. Global memory is enormous because it uses the DRAM area of the GPU. However, it has the disadvantage

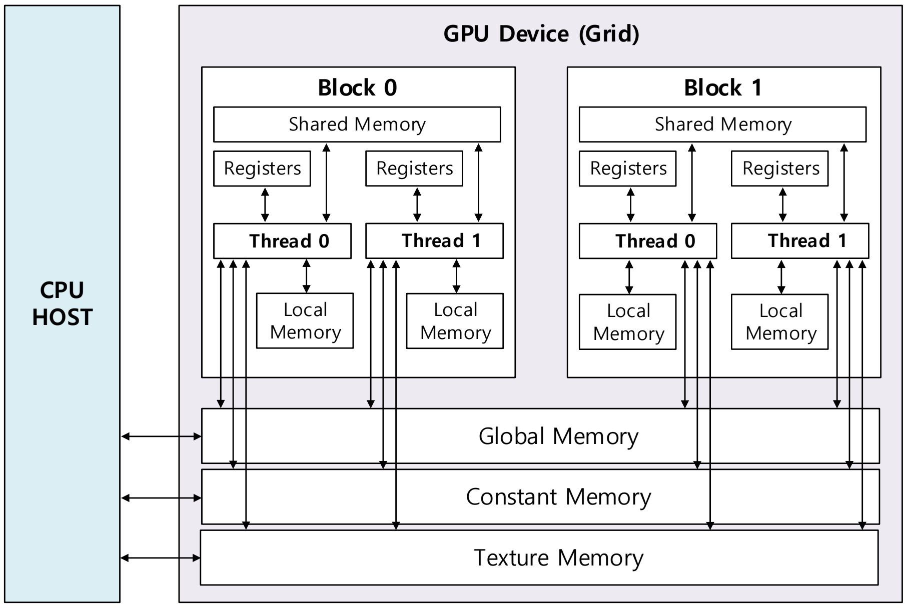


Fig. 1: Memory structure of GPU devices.

that the memory access speed is very slow compared to other memory areas. To solve this shortcoming, from the Fermi architecture GPU, it is possible to caching and use global memory by adding a cache to the SM. However, because the cache size is very small, there are limitations to actively use it. A register is a memory area used by threads responsible for parallel operations in a block. Although small in size, it is very fast. Because there is an upper limit to the size of registers per block, if threads use a lot of registers, some of the memories in the registers are stored as local memory. Since local memory is a memory existing in the DRAM area, access speed is slower than register. Shared memory is a memory shared by threads within a block, and has the advantage of fast access. Since the threads in a block are shared memory, the values in the memory can be affected by other threads.

In 2006, NVIDIA announced Compute Unified Device Architecture (CUDA). CUDA is a parallel computing platform and API model that enables the use of the general-purpose computing on graphics processing units (GPGPU) technique, which is used for general-purpose computation of applications that process GPUs used only for traditional graphics tasks. CUDA can be used in various language such as C and C++, and new versions are updated whenever a new GPU or architecture is released. CUDA compute capability supported version varies depending on the GPU used.

CUDA programming uses a language that adds CUDA-specific syntax as a library. CUDA code consists of calling a function on the host CPU called a kernel that only works inside the GPU. When processing data in the GPU kernel, the memory on the host CPU is not immediately available, requiring an additional process of copying the required memory area from the host to the device in advance. it may also involve copying data back from the device to the host after completing the operation. Figure 2 shows the command execution process between the CPU and GPU kernel.

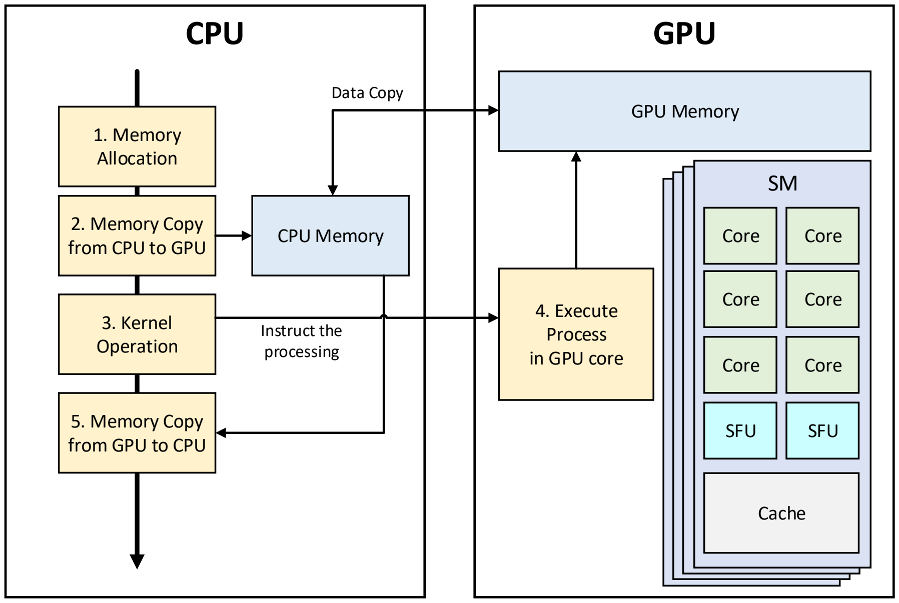


Fig. 2: Command execution process between CPU and GPU.

Table 2: Specifications of CUDA-enabled NVIDIA GPU architectures.

Architecture	Maxwell	Pascal	Turing
GPU Chips example	GTX 980 Ti	GTX 1080 Ti	RTX 2080 Ti
SM count	24	28	68
Core count	2,816	3,584	4,352
Memory Size	6 GB	11 GB	11 GB
Base clock	1,000 MHz	1,480 MHz	1,350 MHz
CUDA compute capability	5.2	6.1	7.5

Currently, various of latest NVIDIA GPUs that can process GPGPU using CUDA library have been released, and CUDA also provides various functions and functions as the version is continuously updated. NVIDIA GPUs have different features for different architecture generations, and different CUDA versions are available. Table 2 describes the examples and features from the latest Turing architecture to the Maxwell architecture. The CUDA compute capability means the CUDA version that the GPU card must be satisfied to the minimum to perform GPGPU.

2.2 ARX-based Korean Block Ciphers

Lightweight cryptography is a fundamental technology to optimize the hardware chip area and reduce the execution timing for low-end Internet of Things (IoT) devices. Recently, a number of block cipher algorithms have been designed for satisfying lightweight features. These are mainly divided into two architectures, Substitute Permutation Network (SPN) and Addition-Rotation-XOR (ARX). For the SPN architecture, PRESENT and GIFT block ciphers have received the attention. The SPN block ciphers utilize the simple 4-bit S-BOX and permutation to achieve the high security as well [3, 7]. For ARX architecture, HIGHT, LEA, SPECK, SIMON, and CHAM block ciphers have been proposed. The ARX block cipher exploit simple ARX operations [11, 6, 9, 15]. In particular, HIGHT, LEA, and CHAM block ciphers are Korean block ciphers. In this paper, we focused on the optimized implementation for these block ciphers.

HIGHT Block Cipher In CHES'06, ultra lightweight block cipher HIGHT was presented [10]. HIGHT is a 64-bit block cipher with 128-bit keys. HIGHT block cipher performs 8-bit wise ARX operations and the encryption/decryption operation consists of 32 rounds. During every single round, 4 blocks are encrypted, and the other blocks are used as input. At this point, some blocks are through by the F_0 or F_1 function. F_0 and F_1 functions performs left shift operation, that expressed in the following equations. Round function structure of HIGHT block cipher is shown in Figure 3.

$$F_0(X) = X \lll 1 \oplus X \lll 2 \oplus X \lll 7$$

$$F_1(X) = X \lll 3 \oplus X \lll 4 \oplus X \lll 6$$

In each round, a 64-bit round key is required, and in total, 2,048-bit of round keys are needed to process a 64-bit block. Parameters for HIGHT are represented at Table 3.

LEA Block Cipher In WISA'13, lightweight block cipher LEA was presented [9]. LEA is a 128-bit block cipher supporting three key lengths, i.e., 128-bit, 192-bit, and 256-bit. LEA-128/128, LEA-128/192, and LEA-128/256 require 24, 28, and 32 rounds, respectively. In each round, 192-bit round keys are required. For more information on parameters are shown in Table 4. The word size is 32-bit and primitive operations consist of 32-bit wise addition, rotation, and exclusive-or.

LEA block cipher performs encryption for 3 blocks in one round, that repre-

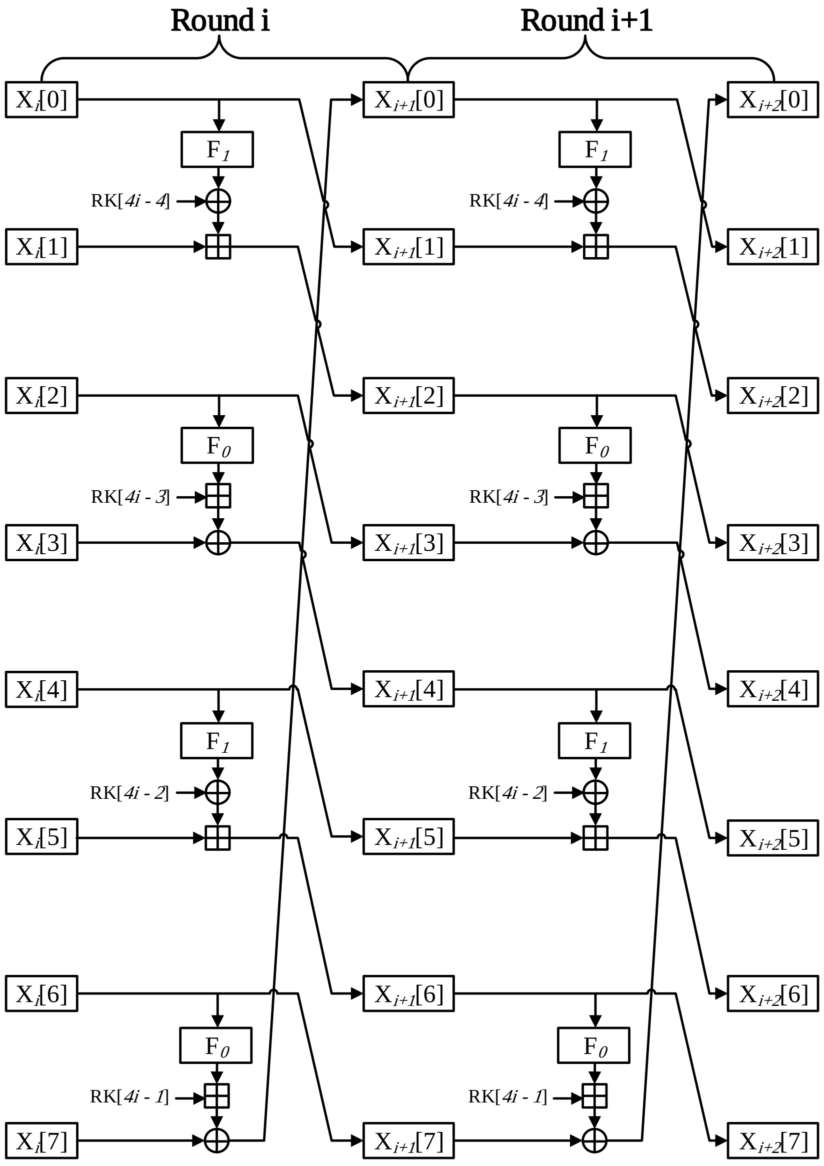


Fig. 3: Round function scheme for HIGHT block cipher.

Table 3: Parameters of HIGHT block cipher, where n, k, rk, and r represent block size (bit), key size (bit), round key size (bit), and number of rounds, respectively.

Cipher	n	k	rk	r
HIGHT-64/128	64	128	64	32

sented at Figure 4. The block[0] does not encrypt, but used for input value to block[1]. All blocks are moves to the left every end of rounds.

Table 4: Parameters of LEA block cipher, where n, k, rk, and r represent block size (bit), key size (bit), round key size (bit), and number of rounds, respectively.

Cipher	n	k	rk	r
LEA-128/128	128	128	192	24
LEA-128/192	128	192	192	28
LEA-128/256	128	256	192	32

CHAM Block Cipher In ICISC'17, a family of lightweight block ciphers CHAM was announced by the Attached Institute of ETRI [15]. The family consists of three ciphers, including CHAM-64/128, CHAM-128/128, and CHAM-128/256. 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 presented [23]. In order to prevent new related-key differential characteristics and differentials of CHAM using a SAT solver, the numbers of rounds of CHAM-64/128, CHAM-128/128, and CHAM-128/256 are increased from 80 to 88, 80 to 112, and 96 to 120, respectively. Whole parameters are given in Table 5. CHAM block cipher has a slightly different structure for odd and even rounds, as shown in Figure 5. In one round of CHAM, block[0] and block[1] participate in the encryption. At the end of round, each block moves to the left. The difference of odd and even rounds is number of left rotation operation bits.

In this paper, we target to 'revised CHAM', but for convenience, write it as 'CHAM'.

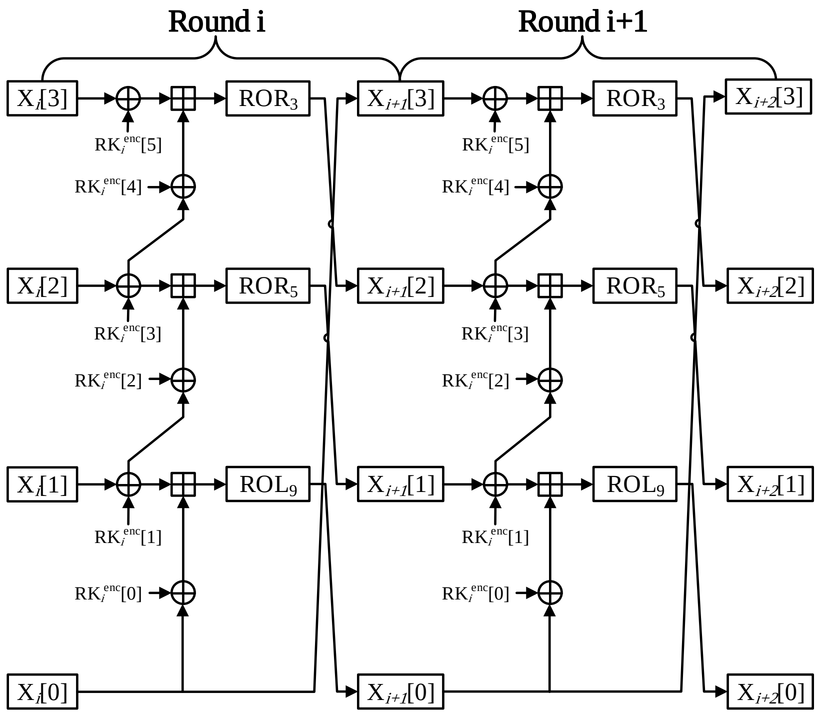


Fig. 4: Encryption process of LEA block cipher.

Table 5: Parameters of CHAM block cipher, where n, k, rk, and r represent block size (bit), key size (bit), round key size (bit), and number of rounds, respectively.

Cipher	n	k	rk	r
CHAM-64/128	64	128	16	88
CHAM-128/128	128	128	32	112
CHAM-128/256	128	256	32	120

2.3 Target Block Cipher Mode of Operation

In order to securely encrypt data larger than basic block size, a mode of operation needs to be applied. There have been several mode of operations. Among them, the simplest one is electronic codebook (ECB) mode. In ECB mode, the long

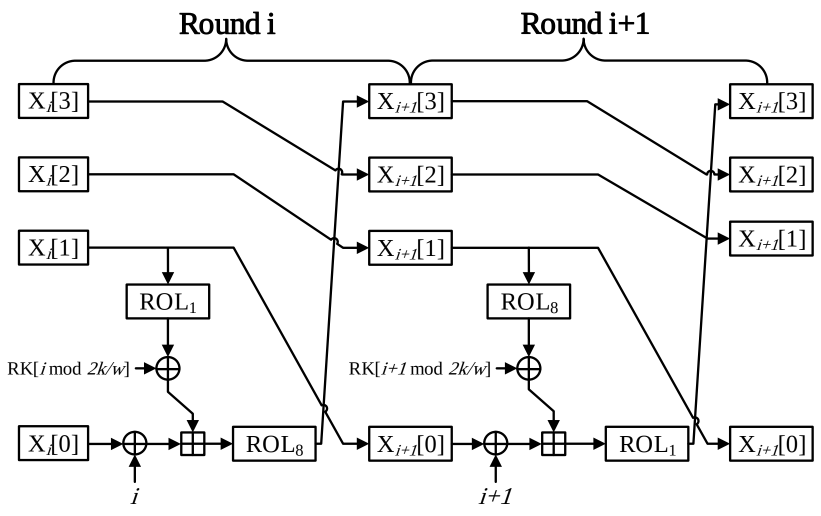


Fig. 5: Encryption round function structure of CHAM block cipher.

message is divided by basic block size into multiple blocks and the divided blocks are encrypted separately. It is well known that ECB mode is not secure because encrypting the same plaintext with the same key generates the same ciphertext.

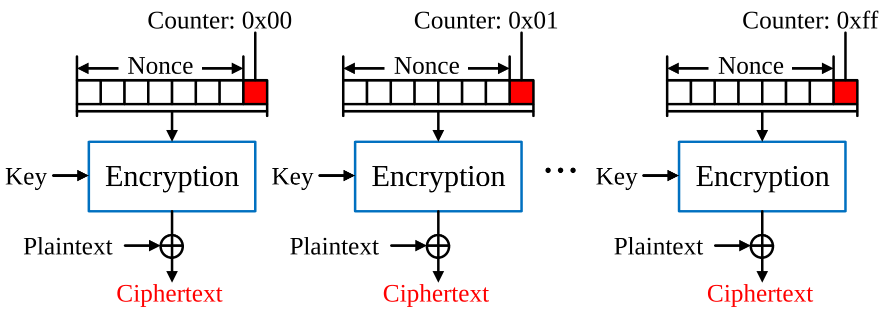


Fig. 6: CTR mode of operation structure.

Therefore, the counter (CTR) mode or the cipher block chaining (CBC) mode is typically used for practical applications rather than ECB. CTR mode turns a block cipher into a stream cipher and it generates the next keystream block by encrypting successive values of a counter value. In other words, In CTR mode, the counter value for each block is combined with the fixed nonce part and used as the input of the block cipher. The encrypted input is XORed with plaintext block. The operation of CTR mode is shown in Figure 6. Lastly, CTR mode has advantages that encryption and decryption processes are identical and they are executed in parallel. Thus, in this paper, we makes use of CTR mode of operation by considering its advantages.

2.4 CTR_DRBG

Table 6: Notations for CTR_DRBG.

Notation	Descriptions
Personalization String	Information for differentiating the instances being created, non-confidential input (optional).
Nonce	Input information used to generate a seed during instance Function.
Internal State	Information used during CTR_DRBG. It consists of Operational Status and Control Information.
Operational Status	Information directly used for random number output. Consisting of C and V, C is the key used for block cipher, and V is the plain text used for block cipher.
Control Information	Information consists of security strength, Prediction Resistance flag and Derivation Function flag.
Prediction Resistance	Characteristics of the exposure of internal status information of the CTR_DRBG without affecting future output.
Instantiate Function	Function to create and initialize CTR_DRBG instances as needed.
Derivation Function	Function called from an Instantiate Function to generate a seed using entropy input, Nonce and Personalization String.
Update Function	Function to update the internal state.
Reseed Function	Function to update Internal State using entropy and additional input. This function is affected by the Reseed Counter.
Generate Function	Function to generate an output(random number) using the Internal State and update the Internal State.
Extract Function	Function to generate random number sequence, using CTR mode.
Uninstantiate Function	Function to make Internal State zero and delete instances after CTR_DRBG use.

The secure key is fundamental prerequisite for operating secure block cipher algorithms to protect secret information. In order to securely generate secret key,

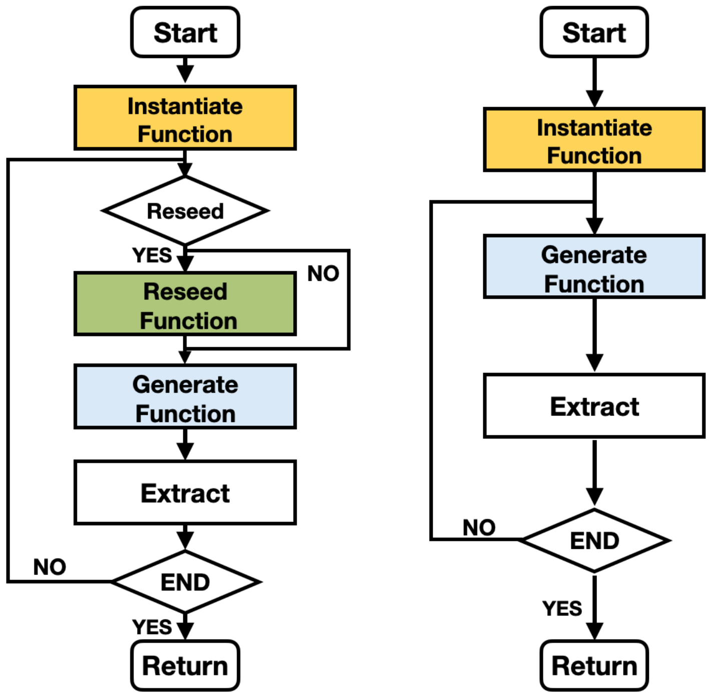


Fig. 7: Operational structure of CTR_DRBG.

random bit generator needs to be utilized. The ideal random bit generator must be satisfied following three things; unpredictability, unbiasedness, and inter-bit independence. The random bit that satisfies these three conditions is called the `true random bit`. Theoretically, however, the creation of a true random bit is almost impossible. Therefore, it creates and uses pseudo-random bit that are difficult to distinguish from true random numbers. As pseudo-random bit generators, DRBGs (Deterministic Random Bit Generators) are used to securely generate random bit information including secret keys, initial vectors, nonces, and so on. There are CTR_DRBG using block cipher algorithm, HASH_DRBG using hash function, and HMAC_DRBG using HMAC among other types of pseudo-random number generators. We note the cryptographic algorithm used in CTR_DRBG and the CTR mode used in Update Function and Extract Function. Various symmetric key algorithms used in CTR_DRBG, unlike the HASH functions used in HASH_DRBG and HMAC_DRBG, can store the internal state in the general purpose register and reduce memory access efficiently in 8-bit AVR Microcontroller. From on the perspective of GPU, CTR mode can be applied in parallel during the process of extracting random numbers.

Table 6 defines notations used in this paper and Figure 7 shows the overview of CTR_DRBG operation process. The left side of Figure 7 is the operating process of CTR_DRBG, which supports Prediction Resistance, and the right

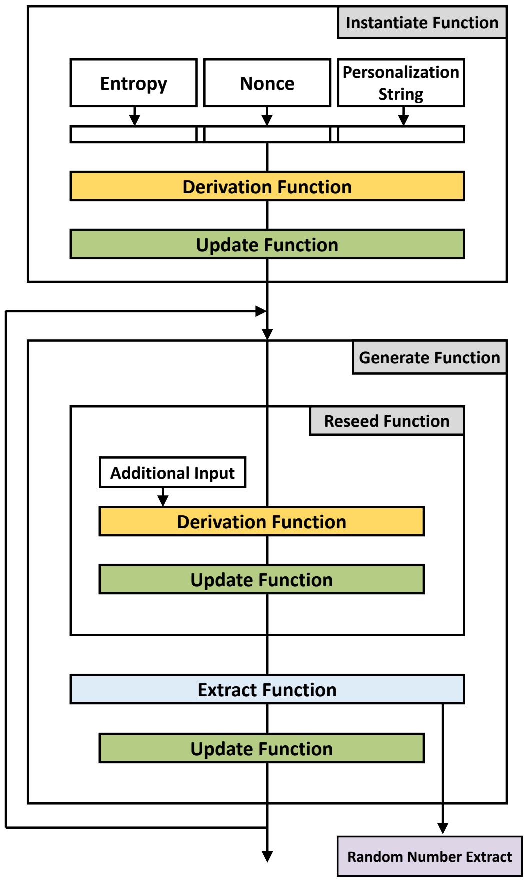


Fig. 8: Detailed procedures of CTR_DRBG.

side is the operating process of CTR_DRBG, which does not support Prediction Resistance. To Extract random number, First, initialize the Internal State by executing the Instantiate Function. Afterwards, if the Reseed Counter is greater than the Reseed interval or supports Prediction Resistance, perform a Reseed Function. If without support for Prediction Resistance, Reseed Function can be omitted. And, it uses the Generate Function to extract random numbers. If there is no additional random number generation, then the use of CTR_DRBG is terminated, otherwise the process for additional random number generation is returned.

Table 7 shows parameters based the used block ciphers in CTR_DRBG. Seed Bit is the addition of Key bit and Block Bit, and N is the representation of Seed Bit in bytes. Len_seed is the value of Seed Bit divided by Block Bit. Figure 8 show the detailed operational process of CTR_DRBG. The Instantiate Function

Table 7: Constant parameters of CTR_DRBG depending on block cipher.

Parameters	**CHAM-64/128**	**CHAM-64/128(32)**	**CHAM-128/128**	**CHAM-128/256**
Key Bit	128	128	128	256
Block Bit	64	64	128	128
Seed Bit	192	192	256	384
N	0x18	0x18	0x20	0x40
Len_seed	3	3	2	3
Parameters	**HIGHT-64/128**	**LEA-128/128**	**LEA-128/192**	**LEA-128/256**
Key Bit	128	128	192	256
Block Bit	64	128	128	128
Seed Bit	192	256	320	384
N	0x18	0x20	0x30	0x40
Len_seed	3	2	3	3

generates seed with the Derivation Function and updates the Internal State using the seed and Update Function. The Reseed Function consists of the Derivation Function and Update Function,and the Generate Function consists of the Reseed Function, Extract Function and Update Function. Note that Update Function is called from Instantiate Function and Generate Function and Reseed Function.

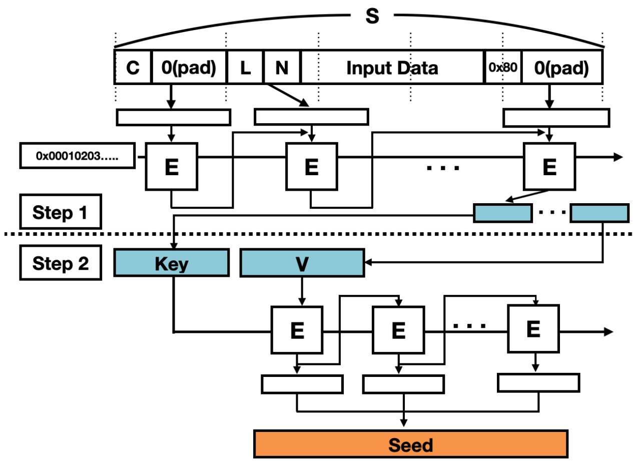


Fig. 9: Derivation function of CTR_DRBG.

Figure 9 shows the structure of the Derivation Function of CTR_DRBG. Derivation Function makes use of CBC-MAC in order to produce an output of seed length by inputting variable length S. The Derivation Function is called in the Instantiate Function and the Reseed Function. Step 1 is a CBC-MAC encryption process by using counter value C. Step 2 is the process of CBC mode which encrypts V with Key generated from Step 1. First, it generates input S for CBC-MAC using Input Data consisting of Entropy, Nonce, and Personalization

String. C, L and N are 32-bit data. The initial C is zero, and padded to zero after C by the length of Block Bit minus 32-bit. L and N are byte lengths of Input Data and seed. The Derivation Function make S using C, L, N and Input Data. At this sequence of generating S, the length of S is padded to 0 so that it is a multiple of Block Bit. Then, the Derivation Function increase C of S by 1 and repeat CBC-MAC as many times as Len_seed. Using the results of CBC-MAC as key and V, the Derivation Function perform CBC mode encryption as many times as Len_seed to generate seed.

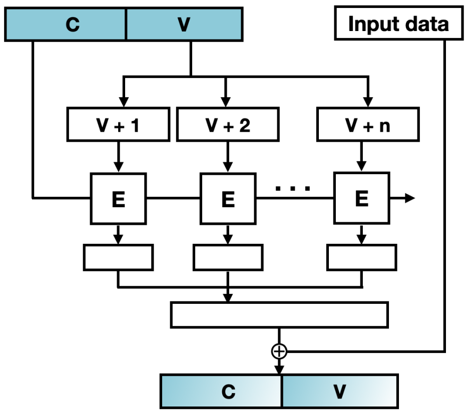


Fig. 10: Update function of CTR_DRBG.

Figure 10 shows the structure of Update Function in CTR_DRBG. Update Function updates C and V of Operational Status in Internal State through CTR mode using input data. Update Function is called by some functions, such as Instantiate Function, Generate Function, and Reseed Function. Using C as the Key and V as the counter, Update Function performs with CTR mode encryption as many times as Len_seed. And, the Update Function performs XOR operation on the generated result value and Input data. If there is no Input data, the XOR operation can be omitted. C and V are zero on the first Update Function call in Instantiate Function.

The Generate Function consists of Reseed Function, Extract Function, and Update Function. Note that Figure 8 shows details of General Function in CTR_DRBG. The Generate Function generates random number using Operational Status C and V. And, updates the Operational Status using Update Function. When the Generate Function is executed, if it supports Prediction Resistance or if the Reseed Counter is greater than the Reseed Interval, the Generate Function calls the Reseed Function to update the Operational Status. In the opposite case, the Reseed Function can be omitted. Then, the Generate Function calls Extract Function for generate random number, and then calls the Update Function to finally update the Operational Status. Figure 11 shows the

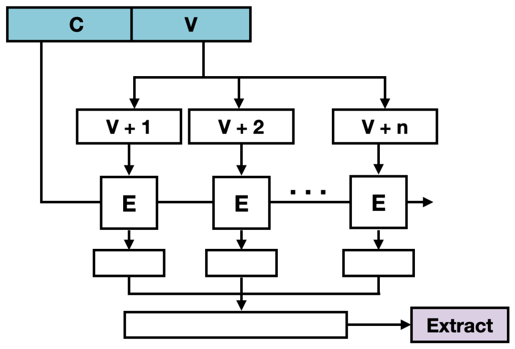


Fig. 11: Extract function of CTR_DRBG.

structure of Extract Function called in the Generate Function in CTR_DRBG. Extract Function is a function that uses the same CTR mode as Update Function. Therefore, using Operational Status C as the key and V as the counter, Extract Function generate random number by using CTR mode by the length of random numbers set by the user.

2.5 Previous Implementations

Block Cipher Implementations on AVR A number of implementation studies have been conducted to improve the performance of block ciphers on 8-bit AVR microcontrollers. Block cipher structures are largely divided into two categories. First, Addition, Rotation, and eXclusive-or (ARX) based block ciphers have been efficiently implemented on low-end microcontrollers [9, 26, 25, 24, 10, 8, 12, 4, 6].

In CHES'06, the HIGHT block cipher was introduced [10]. 64-bit plaintext and 128-bit key are supported, and ARX operations are performed in 8-bit wise. The basic implementation of HIGHT was first introduced in [8]. The execution timing for encryption and decryption is 2,438 and 2,520 clock cycles per byte, respectively. In [25], efficient rotation operations were introduced, and they achieved high performance. The result won the second round of FELICS. In [12], speed-optimized and memory-efficient HIGHT implementations were presented. For the speed-optimized implementation, the delta update, F0 function, and F1 function were replaced by an 8-bit aligned LUT. For the memory-efficient implementation, the delta update, F0 function, and F1 function were written in bit-wise operations.

In WISA'13, the LEA block cipher was introduced by the attached institute of Electronics and Telecommunications Research Institute (ETRI) [9]. The word size and plaintext size are 32-bit and 128-bit, respectively. Three security levels (128-bit, 192-bit, and 256-bit) are supported. The first implementation of LEA-128/128 on an 8-bit AVR microcontroller achieved 190 clock cycles per byte for encryption [9]. In WISA'15, speed-optimized and memory-efficient LEA implementations were presented [26]. The speed-optimized implementation utilizes a byte-wise rotation operation. For the memory-efficient implementation, a partially unrolled approach is used for small code size and reasonable execution timing. In [25], the number of general purpose registers and the instruction set of the AVR microcontroller was fully utilized to optimize the LEA block cipher implementation. The implementation was evaluated on the Fair Evaluation of Lightweight Cryptographic Systems (FELICS) framework. It achieved the best implementation in the first round of the competition. In WISA'18, general purpose registers were efficiently utilized to cache the intermediate results of delta variables during the key scheduling of LEA [24].

The US National Security Agency (NSA) presented two lightweight block ciphers, namely SIMON and SPECK [6]. The SIMON and SPECK block ciphers are intended for efficient hardware and software implementations, respectively. They support various block sizes (32-bit, 48-bit, 64-bit, 96-bit, and 128-bit) and various key sizes (64-bit, 72-bit, 96-bit, 128-bit, 144-bit, 192-bit, and 256-bit). RAM-minimizing, high-throughput/low-energy, and flash-minimizing implementations for 8-bit AVR microcontrollers were presented in [4].

Second, Substitution Permutation Network (SPN) based block ciphers have also been actively investigated. Among them, AES implementations have received considerable attention because the block cipher is an international standard. In [19], the S-box pointer was maintained in the `Z` address pointer for fast memory access. The mix-column computation was efficiently handled with the conditional branch skip. However, previous implementations have mainly focused on ECB mode of operation. However, the CTR mode of operation is most widely used in practice (e.g. TLS/SSL) [18]. In CHES'18, the compact implementation of AES-CTR (i.e. FACE) was presented [20]. The FACE method takes advantage of repeated data in IV by caching a certain amount of the pre-computed result. However, the implementation method is intended for high-end processors and table updating is frequent during computations. For a resource constrained environment, a lightweight variant of FACE (i.e. FACE-LIGHT) implementation was suggested by [14]. With a 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.

Block Cipher Implementations on GPU In the case of Korean block cipher, only a few studies have been optimized on GPUs. As we have seen, few studies have optimized HIGHT and CHAM on GPUs. In the case of LEA, in ICISC'2013, parallel implementation of LEA by using CUDA GPU has performed[27]. The LEA was optimized using the various optimization methods, such as coalesced memory access and inline PTX code. In JKIISC'2015, A LEA optimization study using GPU shared memory was conducted[21]. In [16], excellent performance improvement was achieved through coarse-grain optimization using thread warp. By utilizing the characteristics of warp actively, terabit throughput was proposed for various block ciphers. In [1], It presented optimization performance for CHAM and LEA on GPU environment. Terabit throughput was achieved by integrating and resolving various memory problems that could occur in the GPU environment.

DRBG Implementations on AVR To the best of our knowledge, the implementation of DRBG is not presented in academic papers. The commercial product provides the AES-DRBG but the performance is much slower than our work[1]. Furthermore, the detailed information is not available. For this reason, our work is the-state-of-art work.

DRBG Implementations on GPU To the best of out knowledge, this is the first paper to optimize CTR_DRBG in GPU environment. Also, since only CTR_DRBG among DRBGs generates random numbers using block ciphers, we present the first GPU CTR_DRBG implementation using Korean block ciphers. This can be an indicator for future studies.

3 Optimized Implementation of Counter Mode of Operation

3.1 Main Idea

The optimized implementation of counter mode of operation for block cipher utilizes unique features of fixed nonce and variable counter values instead of the plaintext. At this point, the counter value indicates block number, but the nonce is fixed random value. Thus, every input block has same nonce value which means that the calculation result of nonce block is always identical. Following this idea, we can be pre-calculate the processed nonce block, until before the round that affected by the counter block.

Based on the above idea, the implementation is categorized into two different scenarios that fixed-key and variable-key. In this paper, we optimize both implementations for various applications.

Optimization for Fixed-key Implementation In the fixed-key scenario, it is assumed that the key value does not change. For this reason, the pre-calculation result is always the same, so table update unnecessary.

In the implementation, `LDI` instruction was used instead of `LD` instruction, when loading the pre-calculation value from table. `LDI` instruction operates at one cycle faster than the `LD` instruction. Through this, it is possible to implement a better performance under the fixed-key scenario.

[1] http://cryptovia.com/cryptographic-libraries-for-avr-cpu/

Optimization for Variable-key Implementation On the contrary previous one, it is impossible to use `LDI` instruction when applying variable-key. For the reason that `LDI` instruction can load only the fixed value, but the table values change if the key is updated. Thus `LDI` instruction cannot be used because it cannot load the updated value.

And next, we also implement the pre-calculation part. It is for the purposes of to create pre-calculation table, when the key is updated.

As a final point, it has lower performances than fixed-key implementation. However variable-key implementation is able to process cryptographic operations, even if the key is updated. In other words, variable-key implementation is more suitable for practical use than fixed-key implementation.

3.2 Optimized CTR Mode Implementation based on ARX-based Block Ciphers on AVR

3.2.1 CHAM-CTR on AVR The CHAM divides the plaintext into quarters. Accordingly CHAM-64/128, CHAM-128/128, and CHAM-128/256 use 16-bit, 32-bit, and 32-bit block size respectively. Then the counter block uses 16-bit, 32-bit, and 32-bit block each types, and the nonce has 48-bit, 96-bit, and 96-bit respectively. Figure 12 is represent that the part affected by counter block.

- Round 0, 4 The pre-calculation is not applicable to $X_0[0]$ block, because it has the counter value. However, the input value from $X_0[1]$ has nonce value. In this part, 1-bit left rotation and XOR operations are performed. Due to nonce value is fixed, the result of this operations are always same. That is to say we can be applied pre-calulation. Under the fixed-key scenario, it will be need 2 clock cycles to loads cache table value, but in variable-key scenario, it takes 4 clock cycles. Because, the fixed-key scenario uses `LDI` instruction which needs 1 clock cycle to operate. On the other hand, the variable-key scenario implement with `LD` instruction. It takes 2 clock cycles to work. Round 4 structure is identical with Round 0, eventually it also implemented same way.

- Round 1, 2, 5 In Round 1, the whole operations can be skipped. Because, this round uses only fixed values which they are nonce($X_1[1]$, and $X_1[2]$), round key($RK[1]$), and round counter(1). Finally, we skip this round but result will be loaded $X_4[1]$ at Round 4. Round 2 and Round 5 are also skipped for the same reason as Round 1.

- Round 3, 6 Round 3 performs with $X_3[0]$, and $X_3[1]$ data. At this point, $X_3[0]$ has fixed nonce value, and it is operated XOR with round counter value. The fact that, XOR instruction can be skipped, because both $X_3[0]$ and round counter value are constant. By way of contrast $X_3[1]$ is affected by counter value, so the computation result of $X_3[1]$ not able to implement pre-calculate.

- Round 7 Since all of input values are influenced by counter value, the whole part must be implemented.

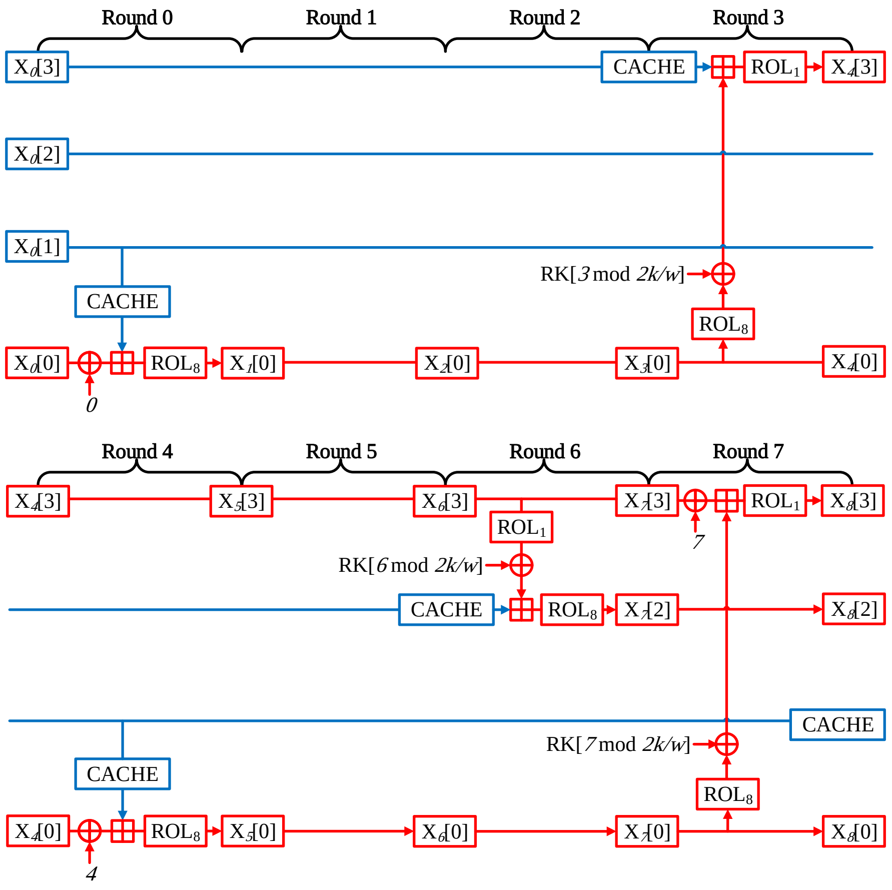


Fig. 12: Optimized eight rounds of CHAM block ciphers.

- Round 8+ After Round 7, pre-calculation scheme is still available for some part of operations. $X_8[1]$ is still not affected by counter value, thus XOR operation part can be pre-computed. However, regardless pre-calculation optimization, the performance improvement does not outstanding. In this time we achieved only 0.6, 0.7, and 0.7 clock cycles per byte enhancement for each CHAM-64/128, CHAM-128/128, and CHAM-128/256. As a result, we decided not to consider only this case.

In general, the CTR mode of operation taking 32-bit counter value, but out implementation using 16-bit counter in case of CHAM-64/128, because the CHAM-64/128 has 16-bit block size specification. Thus we consider the compatibility, it also need to implement 32-bit counter version of CHAM-64/128. For this implementation, the counter value storing into two blocks, then it can be

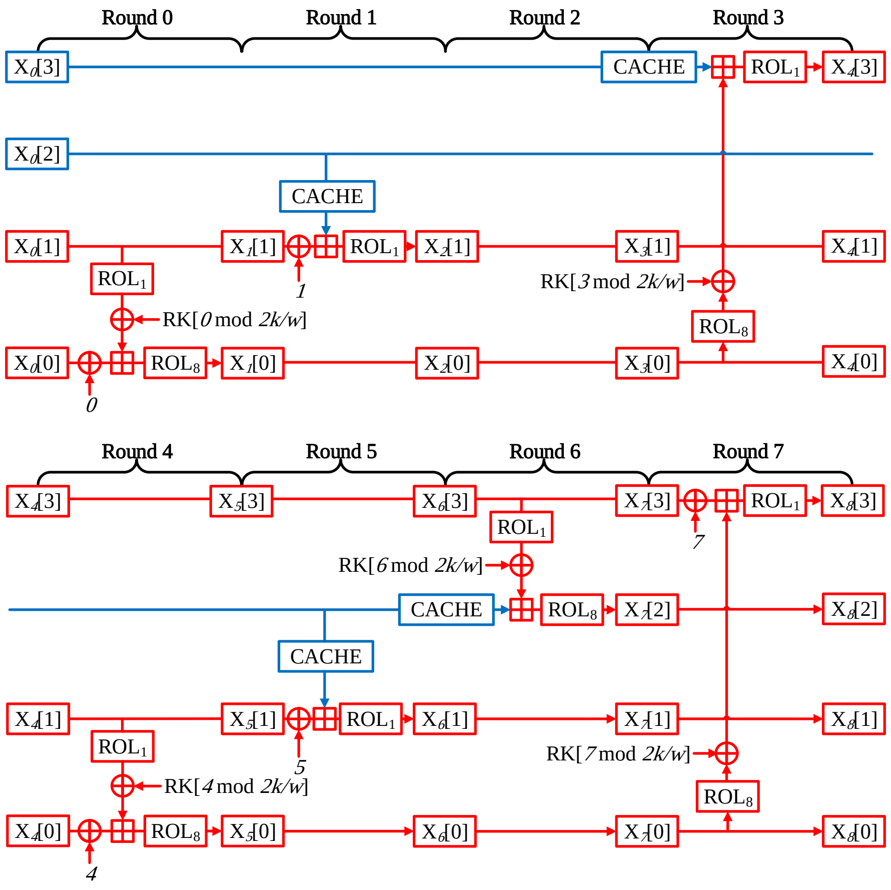


Fig. 13: Optimized eight rounds of CHAM-64/128 block cipher for 32-bit counter.

simply implemented. Figure 13 shows that, the difference from the 16-bit counter implementation.

Parallel Implementations of CHAM block cipher Parallel implementations generate multiple ciphertexts in one implementation. Two types of parallel implementations have been investigated (i.e. 2-parallel and 3-parallel). Since the implementation requires intermediate result caching, the register utilization should be optimized.

Target 8-bit AVR microcontrollers has 32 8-bit registers. For the CHAM-64/128 block cipher, 8 registers are needed. For the case of 2-parallel implementation, it requires 16 registers to save two plaintexts. In addition, the operation requires control variables, including round counter, round key, and address pointer for loading round keys and plaintexts storage. In Figure 14 (a), the register utilization for 2-parallel implementation is given. In this case, all values can be maintained in registers.

On the other hand, 3-parallel implementation requires more registers than 2-parallel implementation. Twenty four registers are used for plaintexts. Since there are not enough registers, `STACK` memory is utilized and some registers are used for multiple purposes. Detailed optimization techniques are as follows:

– **Total round variable**: By using `CPI` instruction, total round value is not maintained in the register.
– **Plaintext block** Only part of plaintext (i.e. $X_i[0]$, and $X_i[1]$) is required to perform the round. Other plaintext values are temporarily stored in `STACK` memory.
– **Address pointer**: Round key and plaintext address pointers are required in computations. However, the address pointer for plaintext is not used throughout computations. The address pointer is stored in `STACK`.
– **Round Key**: Each round key access requires word-wise memory access. By accessing byte by byte, only one register is utilized to access round key.
– **Round counter** Round counter is XORed with data in each round. After the XOR operation, the round counter is stored in `STACK`.
– **ZERO**: `R1` register is `ZERO` register. For 3-parallel implementation, `R1` register is assigned for plaintext. Some registers are temporarily initialized to act as `ZERO` register.

In Figure 14 (b), the register utilization for 3-parallel implementation is given.

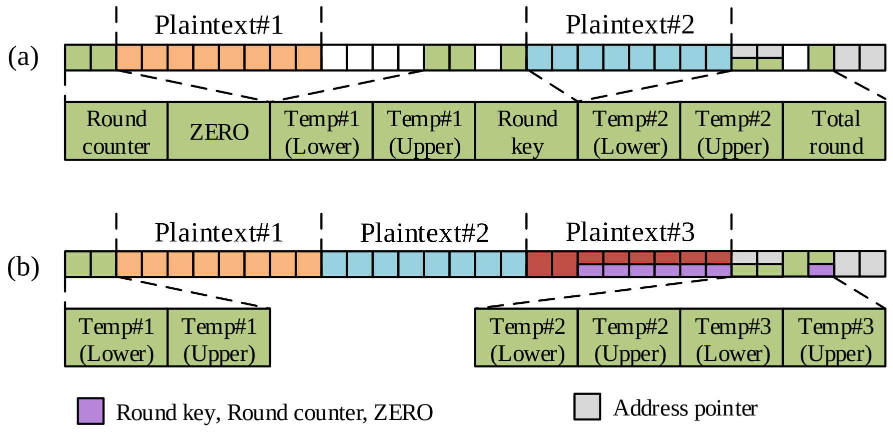


Fig. 14: Register alignment for (a) 2-parallel and (b) 3-parallel of CHAM-64/128 implementations. Each block represents one register. Two color in one register is used for various purposes.

The parallel computation of CHAM-64/128 is given in Algorithm 1. From Step 3 to 7, rotation operations are performed depending on the counter. In Step 8, round key addition is performed. With these round keys, multiple blocks are computed in parallel way. From Step 11 to 15, rotation operations are performed depending on the counter. From Step 16 to 19, the intermediate result is relocated by word-wise.

Algorithm 1 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:         $tmp[j][3] \leftarrow tmp[j][1] tmp[j][2]$

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:         $X[j][0] \leftarrow X[j][1]$
17:         $X[j][1] \leftarrow X[j][2]$
18:         $X[j][2] \leftarrow X[j][3]$
19:         $X[j][3] \leftarrow tmp[j][4]$
20:     end for
21: end for
```

There are limitations to implement the parallel version of CHAM-128/128 and CHAM-128/256. CHAM-128/128 and CHAM-128/256 have twice much longer plaintext then CHAM-64/128. For this reason, parallel implementations for these algorithms are not considered.

Adaptive Encryption of CHAM block cipher The parallel computation is effective for huge data handling. The parallel computation can be applied to adaptive encryption [22]. The adaptive encryption performs parallel encryption operations when the length of data is long enough. When only one block is

Algorithm 2 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 $C$
```

remained, single block encryption is performed. Detailed descriptions for 2-way adaptive encryption are given in Algorithm 2. The method performs 2-parallel computations. From Step 3 to 5, parallel computation is performed. Afterward, from Step 6 to 8, sequential computation is performed for remaining data.

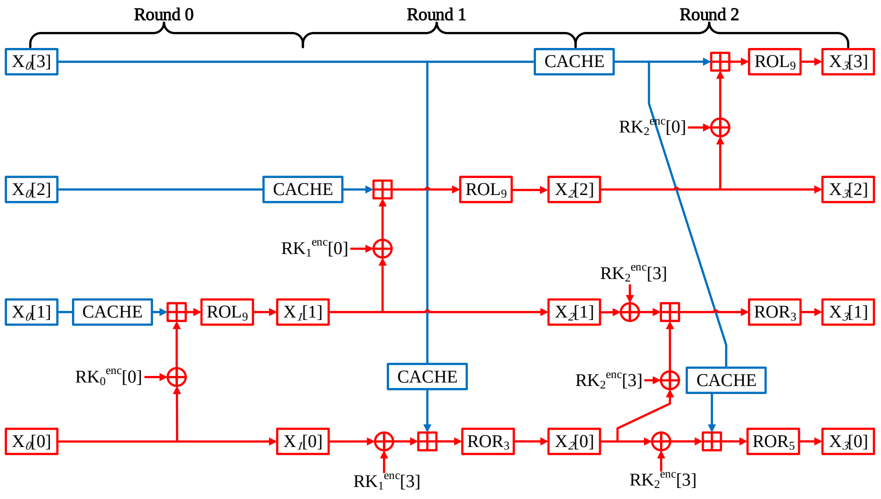


Fig. 15: Optimized three rounds of LEA-128/128 block ciphers.

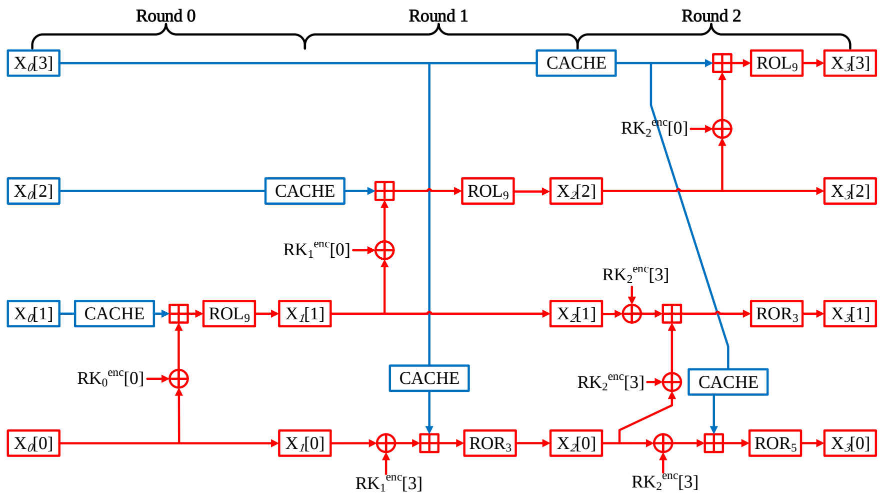


Fig. 16: Optimized three rounds of LEA-128/192 and LEA-128/256 block ciphers.

3.2.2 LEA-CTR The LEA algorithm was suggested [9]. But we use [25] implementation. [25] has optimized key scheduling, especially the LEA-128/128 can be reuse some round keys. Like CHAM algorithm, LEA also divides input data into quarters. The CHAM-64/128 implements a 32-bit counter version apart. But in case of LEA, since only need 128-bit input size, there is no necessary to implement 32-bit counter version separately.

- Round 0 In one round of LEA, the operations is performed in three parts. In Round 0, there only $X_0[0]$ block has counter block. Consequently, two parts can be implemented through the pre-calculation method.

- Round 1 However, due to the Round 0, the $X_1[1]$ block is also beginning to be affected by the counter value. For this reason, it might be thought that the pre-calculation part is only available at $X_1[2]$ part. The part where $X_1[3]$ is used as the input value of $X_1[0]$ in Round 1 can be expressed by the following equation.

$$\mathrm{ROL}_3((X_1[3] \oplus \mathrm{RK}_1^{enc}[2]) \boxplus (X_1[0] \oplus \mathrm{RK}_1^{enc}[3]))$$

At this equation, it can be seen that the blue parts $X_1[3]$ and round key are fixed value. Consequently, XOR instruction between $X_1[3]$ and round key part can be skipped.

- Round 2 In Round 2, Only the $X_2[3]$ block is not affected by counter value. Therefore, pre-calculation is not applicable to as a whole. But like previous round, in order to use $X_2[3]$ as an input value for $X_3[0]$ block, the operation part that performs XOR instruction with round key can be pre-calculation implement. The optimized LEA-128/128 CTR mode of operation is described at Figure 15.

Optimization for LEA-128/192 and LEA-128/256 The LEA-128 from [25] implementation reuses some of round keys, but the LEA-128/192 and LEA-128/256 cannot done this. However, regardless of this, as shown in Figure 16, the overall structure is the same as LEA-128/128. To conclude, the LEA-125/192, and LEA-128/256 implementation can be applied equally like the LEA-128/128 implementation.

3.2.3 HIGHT-CTR The HIGHT-64/128 split input value into eight 8-bit blocks. The CTR mode of operation using 32-bit counter, so four blocks are affected by counter block when initial round. In Figure 17 shows this.

- Round 0 The HIGHT performs four operations in a single round. In Round 0, the operation is performed using the following block pairs; $X_0[0]$ with $X_0[1]$, $X_0[2]$ with $X_0[3]$, $X_0[4]$ with $X_0[5]$, and $X_0[6]$ with $X_0[7]$. First of all, $X_0[0]$, $X_0[1]$, $X_0[2]$, $X_0[3]$ are has counter value, which is variable. So, two of the four operations must be implemented. However, the other operations part use only fixed values, which are nonce, and round keys, so pre-calculation is available for this parts.

- Round 1 Unlike the previous round, the pair of blocks participating in the operation is slightly different. In this time, $X_1[4]$ block is affected by the counter value, then pre-calculation is not possible. $X_1[5]$ and $X_1[6]$ blocks are still have nonce value, so this part is pre-calculation implementation available. And last, $X_1[0]$ block is operate with $X_1[7]$ block that has nonce value. The whole operations cannot skipped, but the result of $X_1[7]$ operation through the F1 function is can be omitted, because $X_1[7]$ has nonce value, and F1 function only does left shift operation.

- Round 2 Round 2 has similar structure with Round 0. But in this time, $X_2[4]$ block is affected by counter value, so the pre-calculation part is reduced by one place then the Round 0.

- Round 3 Likewise this time, Round 3 scheme is alike Round 1. The difference is that $X_4[6]$ block is affected by the counter value. For this reason, pre-calculation implementation is possible in only one part.

3.2.4 Additional Optimization Techniques of Counter Mode on AVR

Optimized memory access The round key is stored in `SRAM` and accessed through `LD` instruction. This requires 2 clock cycles per byte. By aligning the

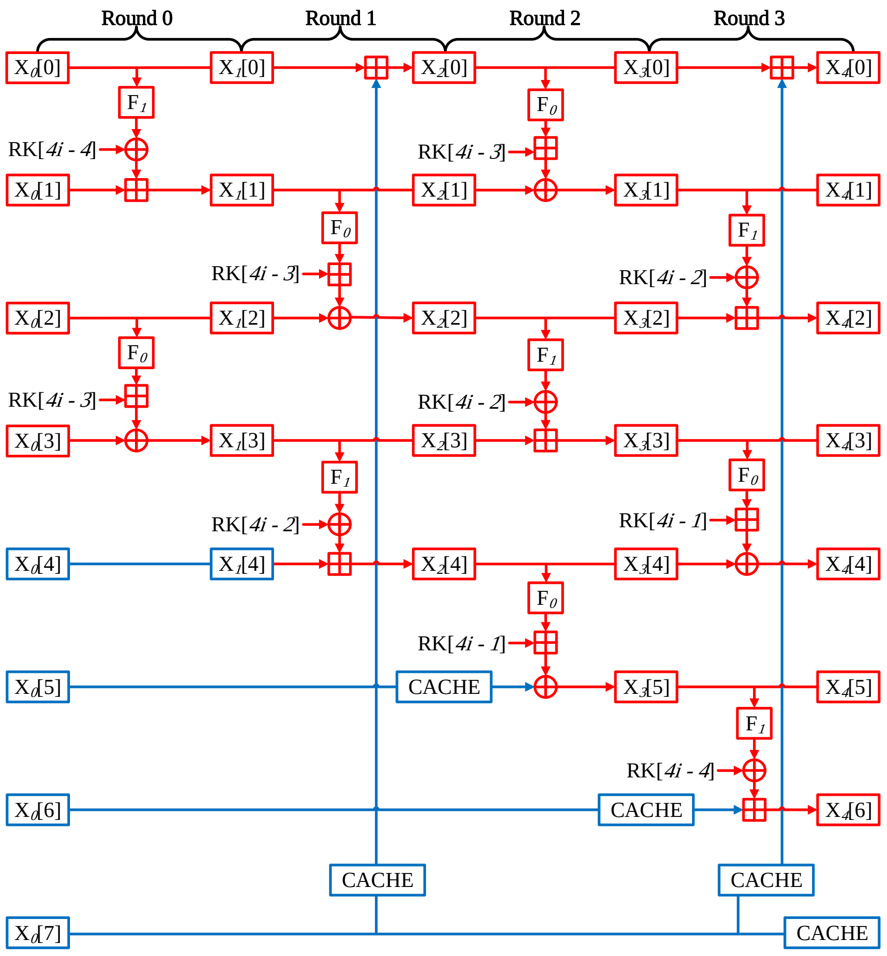


Fig. 17: Optimized four rounds of HIGHT-64/128 block cipher.

round key in 8-bit wise, the offset is only controlled with lower byte. In order to access the pre-computed value, `LDI` instruction is utilized. This requires 1 clock cycle per byte and does not require memory pointer setting.

Optimized rotation operation The other optimized 16/32-bit word rotation operations are given in Table 8.

Table 8: Optimized 16/32-bit word rotation operations on 8-bit AVR microcontroller.

16-bit ROL_1	16-bit ROL_8	32-bit ROL_1	32-bit ROL_8

`LSL LOW`
`ROL HIGH`
`ADC LOW, ZERO`
	`MOV TEMP, LOW`
	`MOV LOW, HIGH`
	`MOV HIGH, TEMP`
		`LSL R0`
		`ROL R1`
		`ROL R2`
		`ROL R3`
		`ADC R0, ZERO`
			`MOV TEMP, R3`
			`MOV R3, R2`
			`MOV R2, R1`
			`MOV R1, R0`
			`MOV R0, TEMP`

3 cycles	3 cycles	5 cycles	5 cycles

3.3 Optimization of Counter Mode of Operation on GPU

In the general block cipher, the plaintext and key are entered as input values of the encryption algorithm, and the ciphertext is output. In the ECB operation mode, the data to be encrypted must always be plaintext. However, in order to handle data in the GPU, data stored in the memory of the CPU must be copied to the GPU in advance. To perform encryption on the GPU through the ECB operation mode, an additional process of copying plaintext values stored in the CPU in advance to the GPU is required.

The CPU and GPU has ultimate fast computation speed. However, PCIe, the transmission path between the CPU and GPU, is relatively slow. Therefore, copying data between the CPU and GPU is time-consuming. Reducing this data copy time can make a significant optimization contribution in GPU implementations. This heavy memory copy time can be reduced through counter mode of operation. A characteristic of the CTR operation mode is that it encrypts the counter value instead of plaintext. There is no need to copy the plaintext from the CPU to the GPU because the plaintext is not used when encrypting on the GPU. In the CTR operation mode, the counter value used for encryption is generated on the GPU and encrypted, so there is no need to separately copy the CPU to the GPU.

Each thread that is in charge of computation inside the GPU has a unique number, the thread ID. The unique number of each thread can be used as a counter value whose value increases by one for each encryption block. As a result, the result of encrypting each thread ID as a counter value is the same as the result of encrypting each block while increasing the counter value by one. The CPU performs encryption by one block while increasing the counter value by one, but the GPU can encrypt the counter values by the number of threads even if each thread encrypts only once. Thus, CTR operation mode encryption using the advantage of these GPUs can show very fast encryption speed.

The round key used for encryption, is generated by the CPU through a key schedule. The generated round key is copied and used by the GPU, and the round key is simply stored in the global memory of the GPU. Global memory is a memory space that all threads can refer to in common, but has a disadvantage that it is very slow compared to other memory spaces. To use the round key efficiently, the round key must be stored in another memory space. Shared memory is a memory shared in block units composed of multiple threads, which is faster than global memory. Shared memory cannot be initialized, and after it is declared in the GPU kernel function, the data in global memory is copied to shared memory. Figure 18 shows an efficient encryption method using shared memory and registers.

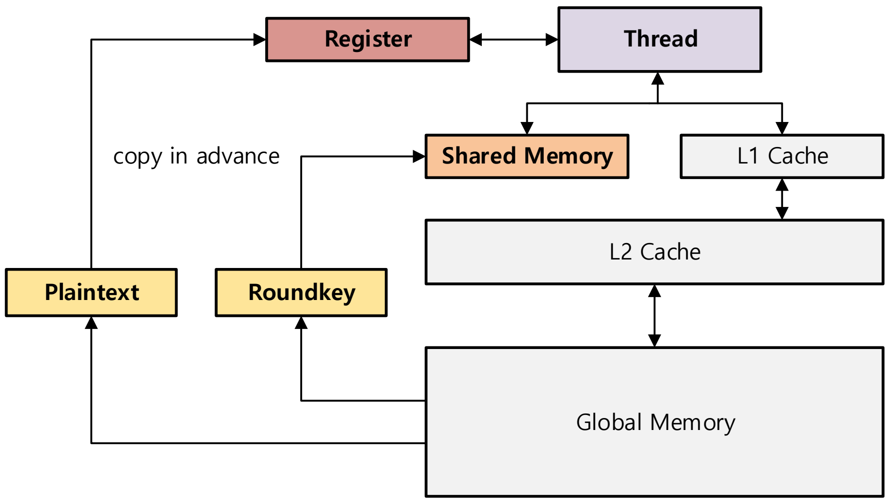


Fig. 18: Efficient encryption method using shared memory and registers.

In each thread block, encryption is performed using the round key copied to shared memory, and shared memory is shared and used for each thread in block units. By the way, when threads refer to the same shared memory, a bank that is a partition of the shared memory area, is accessed simultaneously. For parallel operation, shared memory is divided into multiple memory banks by warp, and parallel access is available in each bank unit. When multiple banks access the same bank, a serialization occurs, which sequentially processes memory access while waiting in sequence. This situation is called a bank conflict. Figure 19 shows the state of the bank conflict problem. In order not to cause the bank conflict problem, the shared memory bank size was adjusted to use a unique bank for each thread, and the round key was stored in each bank location. Therefore, when encryption is performed, each thread can use the round key stored in shared memory within its own bank without duplicate access.

In addition to the optimization method through the CTR mode, an additional optimization method was applied to reduce the memory copy time between the CPU and GPU. As shown in Figure 20, after all the data has been copied from the

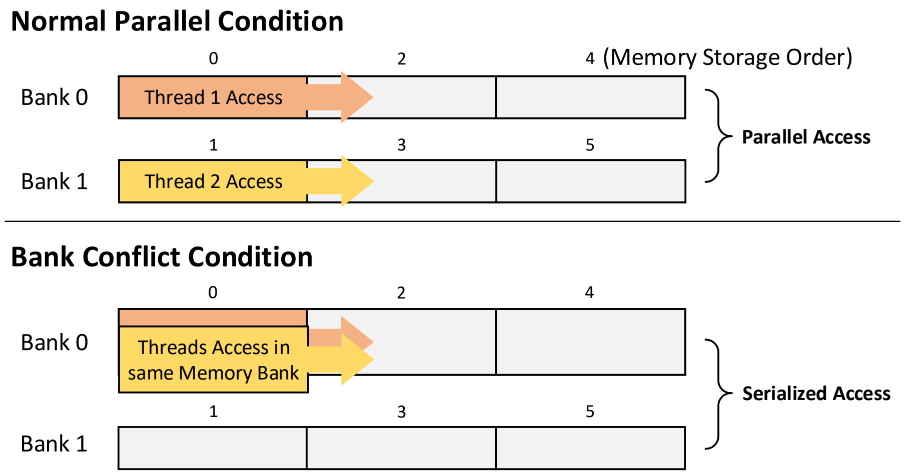


Fig. 19: Bank conflict condition.

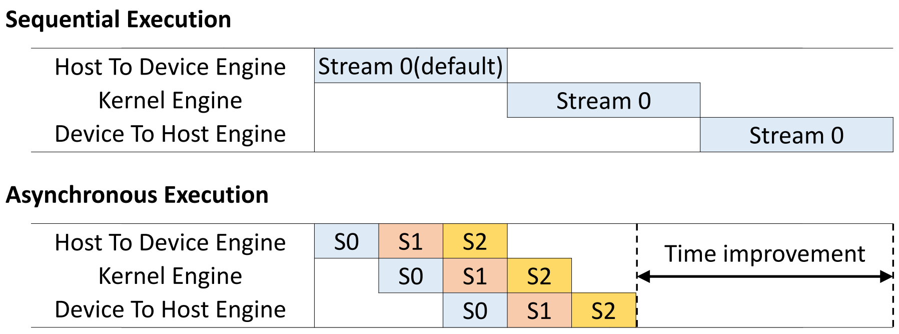
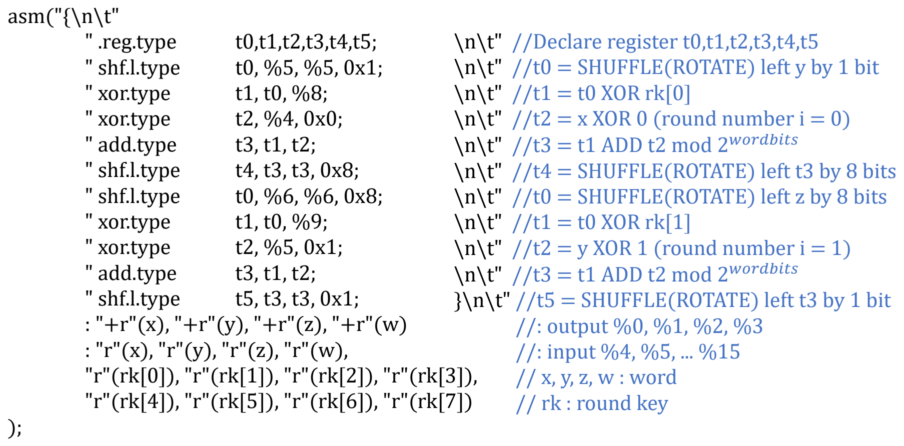


Fig. 20: Asynchronous execution by using CUDA stream.

CPU to the GPU, the CPU enters the idle state and waits until the operation of the GPU is finished while the data is processed on the GPU. When the GPU operation is finished, the GPU data is copied to the CPU. In order to reduce the idle state of the CPU as much as possible, asynchronous instructions can be executed using CUDA streams. Basically, the stream is composed of a single default stream, but CUDA instructions can be used to create multiple streams and divide data to perform operations. Since each stream is managed asynchronously, when the first stream finishes copying data from the CPU to the GPU and enters the kernel engine, the data copy process of the next stream proceeds immediately. In this case, the CPU can continuously process the CPU task while reducing the latency caused by the GPU operation. This optimization technique helps reduce memory copy time between the CPU and GPU.

Additionally, by optimizing and implementing the existing cryptographic algorithm operation with the assembly language of the GPU, PTX assembly code, it was possible to utilize registers to the maximum and reduce unnecessary operations. In particular, by optimizing the rotation operation of the ARX-based block cipher algorithm through PTX assembly code, the optimization method to reduce the number of unnecessary rounds utilizing the characteristics of the CTR mode suggested by microcontrollers was applied, and at the same time, the operation itself was optimized to see an excellent optimization effect.

GPU optimization on CHAM The CHAM algorithm repeatedly uses a short round key several times compared to other algorithms. Therefore, it is possible to have much faster memory access speed by storing all the CHAM plaintext and round keys in a register. However, the total register size is determined on the GPU, and the size is divided as many threads are allowed. At this time, if the number of threads is too large, the maximum register size that can be used by one thread is limited. Therefore, when applying the optimization technique using registers, the number of threads was appropriately selected based on the number of registers to be used for encryption.



```
asm("{\n\t"
        " .reg.type     t0,t1,t2,t3,t4,t5;        \n\t" //Declare register t0,t1,t2,t3,t4,t5
        " shf.l.type    t0, %5, %5, 0x1;          \n\t" //t0 = SHUFFLE(ROTATE) left y by 1 bit
        " xor.type      t1, t0, %8;               \n\t" //t1 = t0 XOR rk[0]
        " xor.type      t2, %4, 0x0;              \n\t" //t2 = x XOR 0 (round number i = 0)
        " add.type      t3, t1, t2;               \n\t" //t3 = t1 ADD t2 mod 2^wordbits
        " shf.l.type    t4, t3, t3, 0x8;          \n\t" //t4 = SHUFFLE(ROTATE) left t3 by 8 bits
        " shf.l.type    t0, %6, %6, 0x8;          \n\t" //t0 = SHUFFLE(ROTATE) left z by 8 bits
        " xor.type      t1, t0, %9;               \n\t" //t1 = t0 XOR rk[1]
        " xor.type      t2, %5, 0x1;              \n\t" //t2 = y XOR 1 (round number i = 1)
        " add.type      t3, t1, t2;               \n\t" //t3 = t1 ADD t2 mod 2^wordbits
        " shf.l.type    t5, t3, t3, 0x1;          }\n\t" //t5 = SHUFFLE(ROTATE) left t3 by 1 bit
        : "+r"(x), "+r"(y), "+r"(z), "+r"(w)                //: output %0, %1, %2, %3
        : "r"(x), "r"(y), "r"(z), "r"(w),                   //: input %4, %5, ... %15
        "r"(rk[0]), "r"(rk[1]), "r"(rk[2]), "r"(rk[3]),     // x, y, z, w : word
        "r"(rk[4]), "r"(rk[5]), "r"(rk[6]), "r"(rk[7])      // rk : round key
);
```

Fig. 21: PTX assembly code example of CHAM

Figure 21 shows the PTX assembly code for a round of CHAM on a CUDA GPU. "r" means register, and all inputs and outputs are entered into the PTX code through registers. In the case of CHAM, since all round keys can be stored in a register, all round keys can be used as an input of a PTX code. On the GPU, the rotate operation on the existing CPU can be replaced with the warp shuffle operation. $Type$ is a variable data type, which is $u16$ for CHAM-64/128 and $u32$ otherwise. In the CHAM encryption process, the process of rotating the word is necessary at the end of each round, but the word rotation process is unnecessary because the PTX code calls the corresponding register instead.

GPU optimization on LEA Since LEA uses many round keys, the number of memory accesses is high. Therefore, if the round key is directly referenced and used in global memory where the memory access speed is slow, the total kernel speed will be slow. To speed up memory access time, shared memory or constant memory can be used. Using shared memory can take advantage of very fast memory access speeds, but be cautious of bank conflicts in this case. Constant memory shows a very fast memory access speed when copying previously cached data, but when accessing data in uncached constant memory, it has similar access speed to global memory.

```
asm("{\n\t"                                                  //i = round
        " xor.type      %2, %6, %12;           \n\t"  //Y = Y XOR rk[i + 4]
        " xor.type      %3, %7, %13;           \n\t"  //Z = Z XOR rk[i + 5]
        " add.type      %3, %6, %7;            \n\t"  //Z = Y ADD Z mod 2^32
        " shf.r.type    %3, %7, %7, 0x3        \n\t"  //Z = SHUFFLE(ROTATE) right Z by 3 bits
        " xor.type      %1, %5, %10;           \n\t"  //X = X XOR rk[i + 2]
        " xor.type      %2, %6, %11;           \n\t"  //Y = Y XOR rk[i + 3]
        " add.type      %2, %5, %6;            \n\t"  //Y = X ADD Y mod 2^32
        " shf.r.type    %2, %6, %6, 0x5;       \n\t"  //Y = SHUFFLE(ROTATE) right Y by 5 bits
        " xor.type      %0, %4, %8;            \n\t"  //W = W XOR rk[i]
        " xor.type      %1, %5, %9;            \n\t"  //X = X XOR rk[i + 1]
        " add.type      %1, %4, %5;            \n\t"  //X = W ADD X mod 2^32
        " shf.l.type    %1, %5, %5, 0x9;       \n\t"  //X = SHUFFLE(ROTATE) left X by 9 bits
        " xor.type      %3, %7, %14;           \n\t"  //Z = Z XOR rk[i + 10]
        " xor.type      %0, %4, %15;           \n\t"  //W = W XOR rk[i + 11]
        " add.type      %0, %7, %4;            \n\t"  //W = Z ADD W mod 2^32
        " shf.r.type    %0, %4, %4, 0x3;       }\n\t"//W = SHUFFLE(ROTATE) right W by 3 bits
     : "+r"(W), "+r"(X), "+r"(Y), "+r"(Z)                              //: output %0, %1, %2, %3
     : "r"(W), "r"(X), "r"(Y), "r"(Z),                                 //: input %4, %5, ... %15
     "r"(rk[i]), "r"(rk[i + 1]), "r"(rk[i + 2]), "r"(rk[i + 3]),       // W, X, Y, Z : word
     "r"(rk[i + 4]),"r"(rk[i + 5]),"r"(rk[i + 10]), "r"(rk[i + 11])    // rk : round key
);
```

Fig. 22: PTX assembly code example of LEA

Figure 22 shows the PTX assembly code for a round of LEA on a CUDA GPU. In the case of LEA, the round key size is very large, so the whole round key cannot be stored in the register. Therefore, part of the round key is stored in a register for use. Since LEA performs right rotate in addition to left rotate, the left rotate and right rotate are performed using $shf.l$ and $shf.r$ instruction in the PTX code.

GPU optimization on HIGHT HIGHT, unlike LEA and CHAM, has a Look-Up table. Therefore, the table to be used for encryption as well as the round key can be copied and used in advance in shared memory. In this case, by utilizing the feature that shared memory is shared among the threads in a block, each thread can copy one data without having to copy all of each table. For example, HIGHT's table consists of two arrays of 256 data. If there are 256 threads per block, It is only needed to copy one data for each array, and if there are 512 threads per block, each thread can only copies one data.

```
asm("{\n\t"
        " .reg.type         t                       \n\t"  //Declare register temp
        " mov.type          t, %9;                  \n\t"  //t = XX[1]
        " ld.shared.type    t, array0[t];           \n\t"  //t = HIGHT_F0[t]
        " add.type          t, t, %19;              \n\t"  //t = t + rk[i*4 + 3]
        " xor.type          %0, t, %8;              \n\t"  //XX[0] = t XOR XX[0]
        " mov.type          t, %11;                 \n\t"  //t = XX[3]
        " ld.shared.type    t, array1[t];           \n\t"  //t = HIGHT_F1[t]
        " add.type          t, t, %18;              \n\t"  //t = t + rk[i*4 + 2]
        " xor.type          %2, t, %10;             \n\t"  //XX[2] = t XOR XX[2]
        " mov.type          t, %13;                 \n\t"  //t = XX[5]
        " ld.shared.type    t, array0[t];           \n\t"  //t = HIGHT_F0[t]
        " add.type          t, t, %17;              \n\t"  //t = t + rk[i*4 + 1]
        " xor.type          %4, t, %12;             \n\t"  //XX[4] = t XOR XX[4]
        " mov.type          t, %15;                 \n\t"  //t = XX[7]
        " ld.shared.type    t, array1[t];           \n\t"  //t = HIGHT_F1[t]
        " add.type          t, t, %16;              \n\t"  //t = t + rk[i*4]
        " xor.type          %6, t, %14;             }\n\t" //XX[6] = t XOR XX[6]
        " : "+r"(XX[0]), "+r"(XX[1]), ... , "+r"(XX[7])                 //: output %0, %1, ... , %7
      : "r"(XX[0]), "r"(XX[1]), ... , "r"(XX[7]),                       //: input %8, %9, ... , %19
      "r"(rk[i*4]), "r"(rk[i*4 + 1]), "r"(rk[i*4 + 2]), "r"(rk[i*4 + 3]) // rk : round key
);                                                                      // i = round
```

Fig. 23: PTX assembly code example of HIGHT

Figure 23 shows the PTX assembly code for a round of HIGHT on a CUDA GPU. Two Look-Up tables are referenced by the *ld.shared* instruction after being copied to shared memory. XX is a temporary 64-bit data type array used by HIGHT. It can be stored in a register to perform encryption to achieve fast encryption performance. HIGHT, like LEA, has a large round key size, so it is difficult to store the entire round key in shared memory, and only the round key used for one round can be imported and used.

4 Optimized Implementation of CTR_DRBG

4.1 Optimization of CTR_DRBG on 8-bit AVR Microcontroller

In this section, we present an optimization methods for the Instantiate Function of CTR_DRBG. Our main idea is to optimize the constants data that exist in the Derivation Function and Update Functions that are called from the Instantiate Function. Also, the advantage of our idea is that it can be applied regardless of

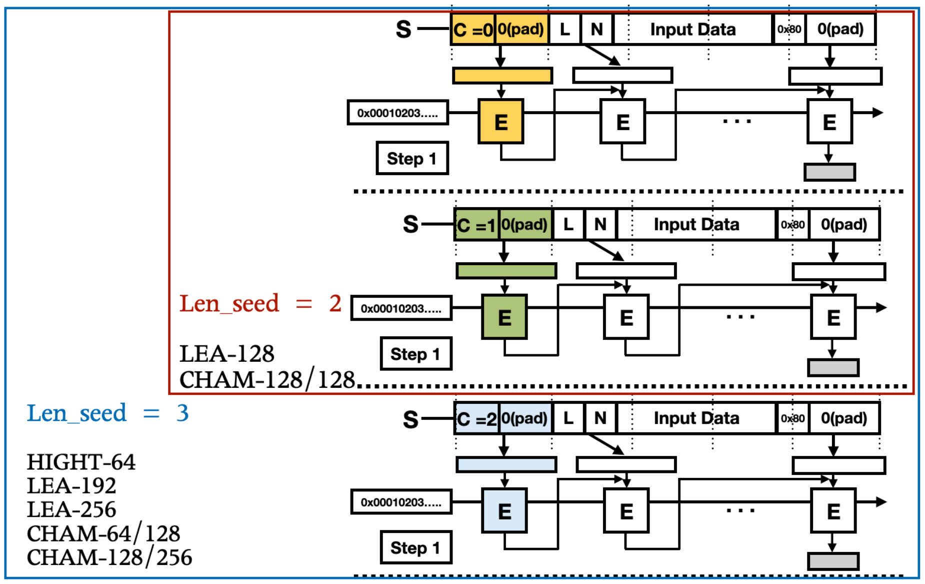


Fig. 24: Optimized implementations for derivation function of instantiate function.

the type of cipher used in CTR_DRBG. Our decision to develop optimizations using constants data for the Derivation Function and Update Function, which are used in the Instantiate Function, is to generate Look-Up table for the determined constant data.

In short, our strategy to speed up the Instantiate Function is to generate a Look-Up table for the result of encryption for the Block Bit size from msb of S, which is used in CBC-MAC of the Derivation Function. In addition, we optimize the Instantiate Function by generating a Look-Up table for the result of the Update Function called from the Instantiate Function by using the fact that the C and V values of the initial Operational Status are zero.

Figure 24 shows optimization method we propose for Derivation Function of Instantiate Function. As mentioned in Section 4, Step 1 of the Derivation Function is a kind of hash function using CBC-MAC. When CBC-MAC is called in the Derivation Function, the data from msb to Block Bit of S is zero. The CBC-MAC execute with encryption, increasing C by the number of times Len_seed. Since, At this time, the key used in the CBC-MAC is fixed(0x00010203..), result for encryption of Block Bit including C can be stored in a Look-Up table. Note that table 7 shows Len_seed of Korean Block Cipher used in this paper. Except for LEA-128/128 and CHAM-128/128, the Block Cipher has a Len_seed value of 3. That is, for LEA-128/128 and CHAM-128/128, we can reduce two encryption process of CBC-MAC. And, for other cases (HIGHT-64/128, LEA-128/192, LEA-128/256, CAHM-64/128, and CHAM-128/256), we can reduce three encryption processes of CBC-MAC. The optimized method of Derivation Function we propose is applicable regardless of the length of Input Data entered in S. In addition, for the CTR_DRBG that supports Prediction Resistance, the Reseed Function is called from the Generate Function, even at this time our main idea for Derivation Function are applicable. The constants data on the Look-Up table is fixed data and requires the cost of Block Byte $*$ Len_seed. Based on the Korean Block Cipher used in this paper, Look-Up table requires up to 48-byte. Since the Look-Up table is used semi-permanent when it is created (Look-Up table is constant data), the generation time of Look-Up table is not considered. Therefore, the Look-Up table can be used in environments where CTR_DRBG is repeatedly called.

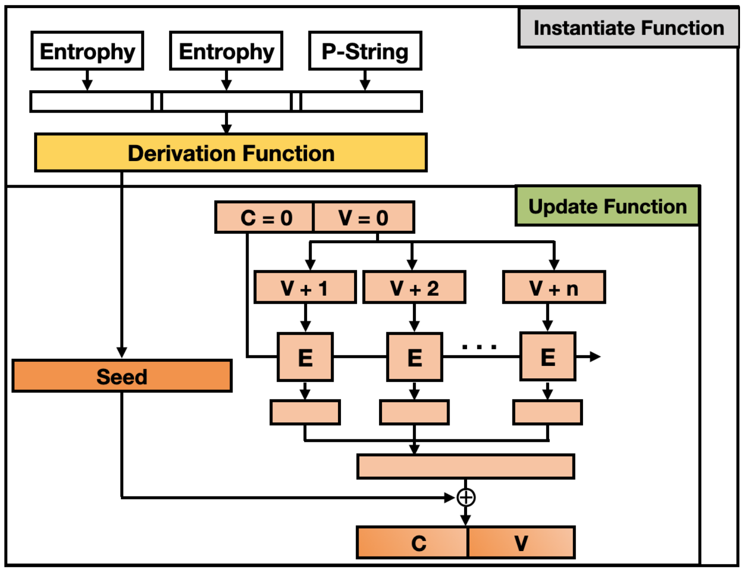


Fig. 25: Optimized implementations for update function of instantiate function.

Figure 25 shows optimization method we propose for Update Function of Instantiate Function. When Instantiate Function is called, Derivation Function generates seed. After generating the seed, Instantiate Function calls Update Function to execute an initial update for the Operational Status. We note that the initial Operational Status state is zero (C = 0, V = 0). Since the Operational Status is zero when the Instantiate Function generates a seed and updates the Internal State, we can store the results for Update Function in the Look-Up table. Optimization method that can be applied regardless of the Block Cipher algorithm, such as the optimization method applied to the Derivation Function we propose. In addition, the Look-Up table for Update Function is a constant data table that can be used regardless of the number of calls made to the CTR_DRBG, just like the Look-Up table generated by the Derivation Function. In other words, our method can be replaced by using the Look-Up table for calling Update Function that must be called in the Instantiate Function. The Look-Up table for Update Function requires up to 48-byte, the same as the Look-Up table generated for the Derivation Function.

Optimization methods we propose for Instantiate Function is to omit the encryption process as much as possible by using Look-Up table. The Look-Up table we propose requires up to 96-byte costs. That is, we can replace up to six encryption process using just 96-byte. In the case of ATmega128, the most popular use in 8-bit AVR Microcontroller environment, as mentioned it section 2.1, It has 4KB of RAM. Therefore, the maximum 96-byte required for the Look-Up table for Instantiate Function can be stored in RAM of ATmega128 sufficiently. In addition, the optimization methods we propose for Instantiate Function has the advantage of being applicable to various platforms such that Low-end-Platforms and High-end-Platforms without relying on specific platforms.

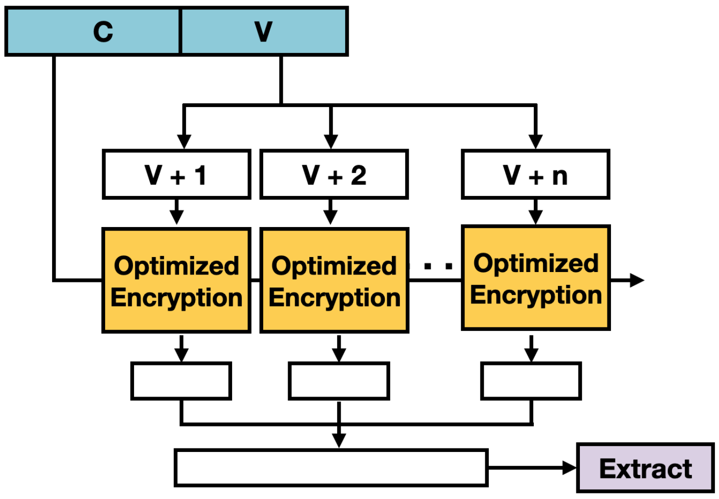


Fig. 26: Optimized implementations for extract function of generate function.

The Extract Function called in the Generate Function is a function of extracting random numbers using CTR mode. Figure 26 shows optimization method we propose for Extract Function of Generate Function. Extract Function uses C and V in Operational Status as a key and a counter to perform CTR mode to extract random numbers. Using optimized CTR mode proposed in section 3 , we apply optimized CTR mode of Extract Function. In order to apply optimized CTR mode using Look-Up table proposed in section 3 to Extract Function, Look-Up table must be generated first. When Counter is V+1, we apply optimized CTR mode with pre-computation, through this process we generate Look-Up table. Note that optimized CTR mode with pre-computation only applies when the counter is V+1. If the counter is V+2 or higher, the optimized CTR mode using Look-Up table is applied for the encryption of all CTR modes during Extraction Function.

4.2 Optimization of CTR_DRBG on GPU

The main operation of CTR_DRBG is CTR mode encryption process, so the method of optimizing block encryption in CTR operation mode can be applied to CTR_DRBG. In CTR_DRBG, CTR mode of operation's encryption process is mainly used in update function and extract function. Figure 27 shows the use of GPU in the update and extract functions. Significant time improvement can be obtained by processing the CTR operation mode encryption process performed in parallel using GPU threads.

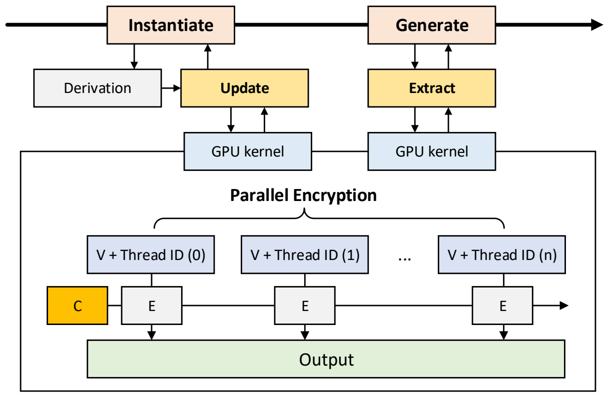


Fig. 27: CTR_DRBG optimization on GPU.

When generating a random number sequence while calling multiple CTR_DRBGs, since the update function performs CTR mode encryption of 2 to 3 blocks depending on the type of the selected block cipher, 2 to 3 threads can implement one CTR_DRBG. In this paper, the optimization strategy for CTR_DRBG on GPU are two categories. First thing that threads output a random number sequence by calling one CTR_DRBG function. In this case, since a different CTR_DRBG internal environment can be built by using a thread ID, which is a unique number for each thread, a more secure random number sequence can be output. However, memory and performance are limited because each thread has to bear the instantiate function, internal derivation function, and update function, which are processes until the random number sequence is output from CTR_DRBG. When each thread calls a number of CTR_DRBGs, the parallel CTR mode optimization method can be applied by encrypting one or two CTR_DRBGs in 2-3 threads according to the CTR block lenseed value of the update function.

Another optimization direction is to call one CTR_DRBG, but set the length of the random number sequence very large inside to process many CTR mode blocks in parallel. That is, the CTR mode encryption process performed by the extract function is optimized and used as efficiently as possible. In this case, the same process is performed in the CPU until inside the CTR_DRBG just before the extract function, but is implemented to process only the extract function with the GPU. This method applies only the CTR mode optimization method in one CTR_DRBG. The CPU only needs to manage one internal state, so it has the advantage of less memory load than the optimization method that calls multiple CTR_DRBGs. Outputting a large number of random numbers through one internal state can cause safety issues. However if the number of internal seed reuses exceeds a certain threshold, the maximum value of the seed counter that calls the seed function is 2^48. This value is extremly big number that is hard to reach. Therefore, even in a method of outputting a large number of random numbers through the GPU, the corresponding seed counter value is not transmitted.

In the extract function, the number of CTR mode blocks is determined according to the output length of the random number sequence. In the case of the optimization method that calls multiple CTR_DRBGs, since 2 3 threads are in charge of one CTR_DRBG in the update function, the random number sequence output length is also implemented to output 2-3 blocks according to the number of threads. In the case of the optimization method of outputting a large number of random number sequences in one CTR_DRBG, the number of threads can be set to the number of CTR mode blocks according to the random number size to be output, so that each thread can encrypt each block in parallel. For example, when outputting a random number sequence having a block size of 16 bytes, in order to output a random number sequence of 1 MB, 65,536 threads can encrypt each CTR mode block once.

Since CTR_DRBG is a random number generator using CTR mode, techniques optimized in the CTR mode environment can be reflected in the CTR mode calculation process inside CTR_DRBG. Therefore, by applying the optimization methods of each block ciphers presented in Section 3 and the constant optimization methods presented in Section 4 to the CTR mode calculation process inside CTR_DRBG. It is possible to obtain an improved performance in addition to the existing GPU optimization method.

5 Evaluation

5.1 CTR on AVR Microcontrollers

Proposed low-end implementations were evaluated on low-end 8-bit AVR microcontollers. The performance was measured in execution time (clock cycles per byte). The software was implemented over `Atmel Studio 7` and the code was complied in `-O2` option.

The comparison criteria are as follows. The first is previous works, the second is a model that separates pre-computation, the third is a model that performs pre-calculation at online, and the last is an optimization implementation.

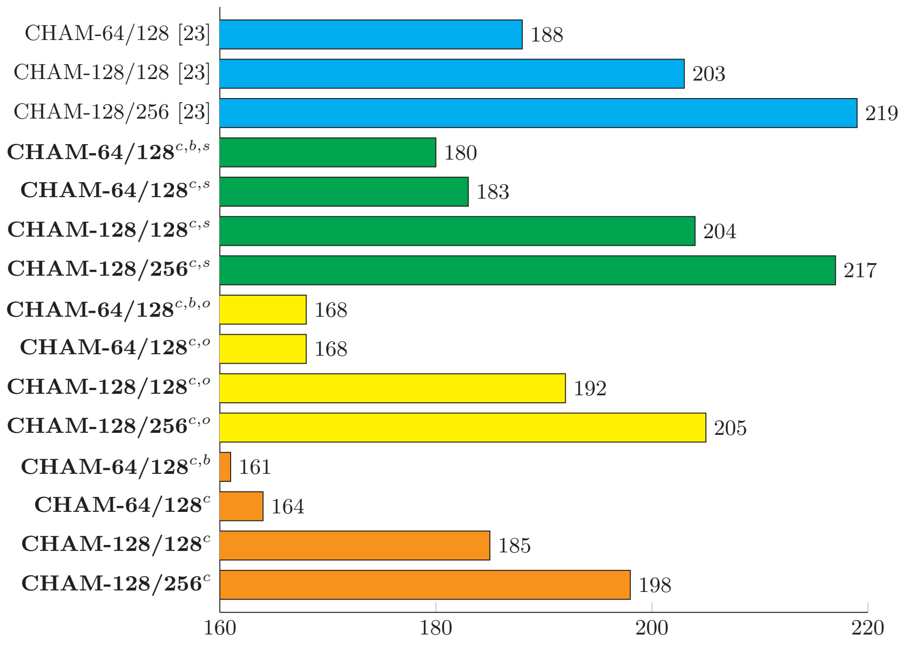


Fig. 28: Comparison of execution time for CHAM implementations on 8-Bit AVR microcontrollers under the variable-key scenario in terms of clock cycles per byte, c: counter mode of operation, b: 16-bit counter for CHAM-64/128, s: pre-computation in separated way, o: pre-computation in online.

CHAM-CTR on Microcontrollers Comparison result of CHAM block cipher is given in Figure 28. Compared with previous works by [23], First of all, the separation model does not appear to have improved performance compared to [23]. However, since this model calculates a table through pre-calculation, and then proceeds with encryption. So this model has more operation codes than the [23]. If the encryption is more than two blocks, it will has outperform the previous work.

The second is pre-computation in online model. This model has about 10.6%, 10.6%, 5.4%, and 6.4% better performance than [23] at each CHAM-64/128, CHAM-64/128(32-bit counter), CHAM-128/128, and CHAM-128/256 respectively. This model creates a cache table in the same way as the separation model, but it works much more effectively than the separation model because it encrypts one plaintext block together when the table is created.

The last, optimized CTR mode of operation model is using cache table value which calculated from separation model, or pre-computation in online model. This model has the best performance than other models. Numerically, performance improvement achieved 16.8%, 12.8%, 8.9%, and 9.6% than [3], CHAM-64/128, CHAM-64/128(32-bit counter), CHAM-128/128, and CHAM-128/256, respectively. Because this model does not include the calculating for cache table but only use the pre-calculation value. In other words, this model can be skipped many instructions for encryption.

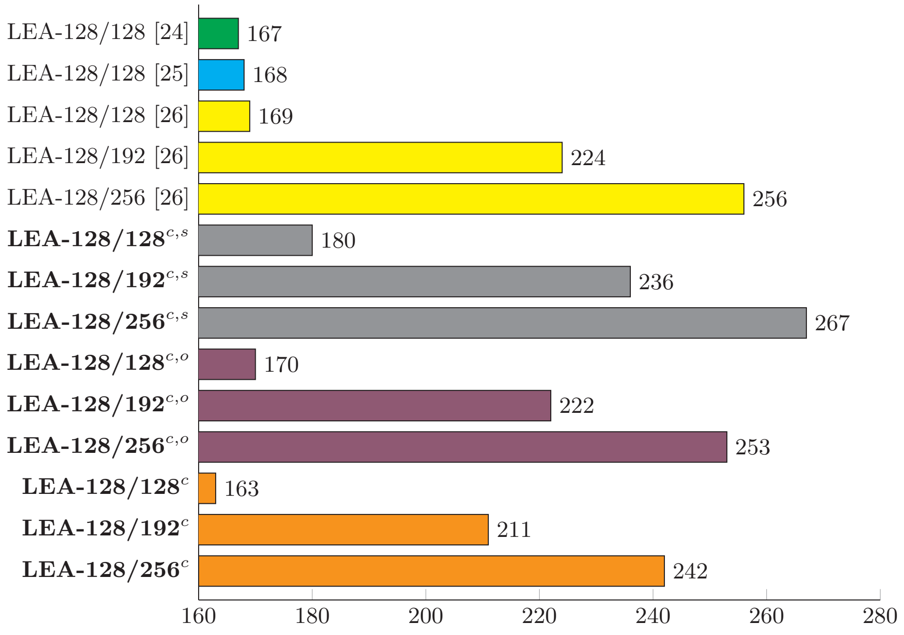


Fig. 29: Comparison of execution time for LEA implementations on 8-Bit AVR microcontrollers under the variable-key scenario in terms of clock cycles per byte, c: counter mode of operation (32-bit counter), s: pre-computation in separated way, o: pre-computation in online.

LEA-CTR on Microcontrollers The LEA implementation result is shown in Figure 29. LEA has three of previous works, that [26], [25], and [24]. The latest research is [24], but it only has LEA-128. Therefore, in case of LEA-128/192 and LEA-128/256, it is compared with [26].

The separation model of LEA-128 has worse performance about 7.3% than the [24]. Also LEA-128/192, and LEA-128/256 has slightly lower speed than the [26].

However, the pre-computation in online model of LEA-128 shows almost similar performance to [24]. The reason for the performance difference is that there is a part that calculates a cache table. That is, without this part, it can be evaluated as having higher performance.

The optimized CTR mode of operation model clearly outperforms [24]. It showed a 2.4% performance improvement. In case of LEA-128/192, and LEA-128/256, it has 5.9% and 5.5% improved performance compared to [26].

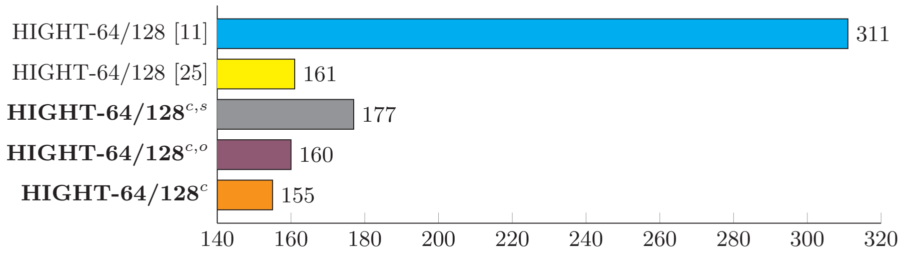


Fig. 30: Comparison of execution time for HIGHT implementations on 8-Bit AVR microcontrollers under the variable-key scenario in terms of clock cycles per byte, c: counter mode of operation (32-bit counter), s: pre-computation in separated way, o: pre-computation in online.

HIGHT-CTR on Microcontrollers The HIGHT only has a 64-bit plaintext type, and there are one previous study for optimization. So we compared to previous work [25], and the result is represented to Figure 30.

Separation model has lower performance about 9.9% than [25]. But pre-computation in online model seem slightly better performance. The reason for this result is that the sepration model performs encryption after calculating the cache table. On the other hand, the pre-computation in online model simultaneously computes the table and encrypt plaintext. Given these points, the pre-computation in online model has better performance.

Last, optimized CTR mode of operation model gets 3.8% better performance than [25]. This is because some of the calculation intervals are skipped through the uses pre-calculated values.

Compared with other block ciphers And also, we compared performance with the other block ciphers, such as AES, SPECK, and SIMON. Its result given in Figure 31. First, AES that based on SPN architecture, has the best performance than the other implementations, owing to AES well designed to 8-bit word processors. In terms of ARX based block ciphers, SPECK block cipher got first rank, and next HIGHT block cipher achieved excellent performance. And followed by CHAM, and LEA block ciphers in order.

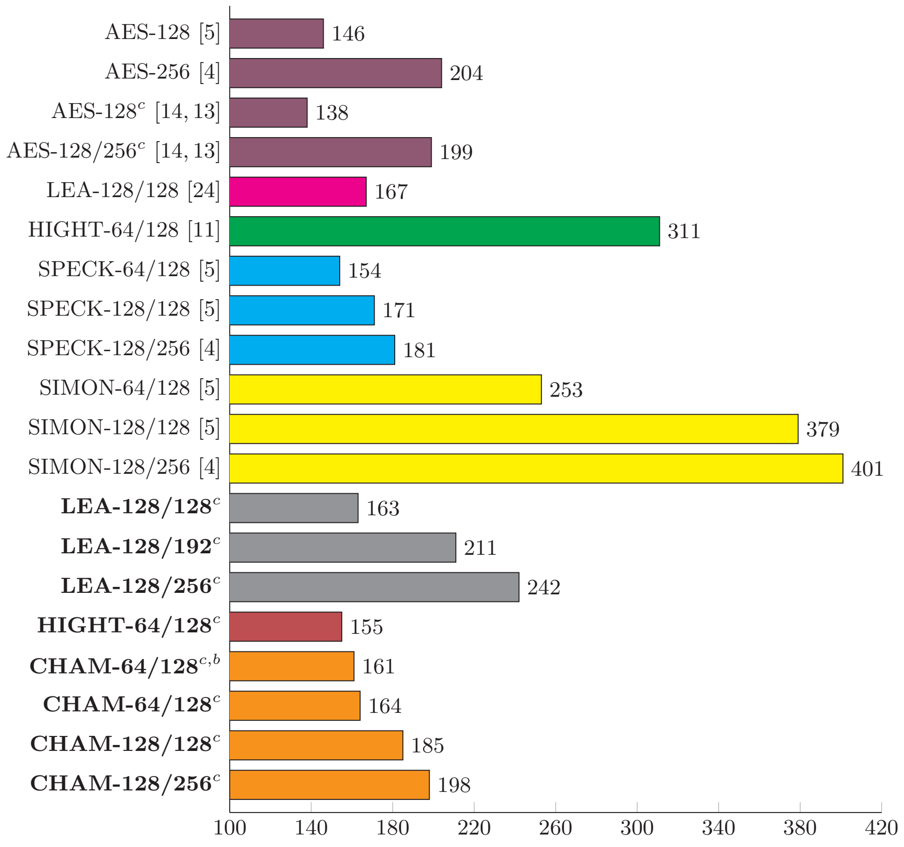


Fig. 31: Comparison of execution time for other block ciphers on 8-Bit AVR microcontrollers under the variable-key scenario in terms of clock cycles per byte, c: counter mode of operation, b: b: 16-bit counter for CHAM-64/128.

5.2 CTR on GPU

The results implemented according to the proposed GPU optimization method were measured in the environment presented in Table 9. All performance measurements on the GPU were measured through throughput, and the unit of measurement is Mbps. The size of data to be encrypted in parallel was measured from a minimum of 128 MB to a maximum of 1 GB, and the number of threads per block was measured by increasing from a minimum of 256 to a maximum of 1024. The graphs presented below were fixed based on the data size and the number of threads per block that showed the best optimization result. The performance was averaged after measuring the repeated time for a total of 100 iterations.

Figure 32, 33, 34 shows the performance improvement according to each optimization method for CHAM, LEA, and HIGHT. Each figure compares the performance of each algorithm, and within each figure, compares each optimization performance according to the key size of the algorithm.

Table 9: GPU optimization implementation test environment.

CPU	AMD RYZEN 5 3600
GPU	RTX 2070
GPU Architecture	Turing
GPU Core count	2,340
GPU Memory Size	8 GB
GPU Base clock	1,410 MHz
OS	Windows 10
CUDA Version	10.2

Table 10: Comparison of encryption kernel on GPU in terms of throughput (Gbps).

Method	Platform	CHAM	LEA	HIGHT
Lee et al. [16]	GTX 980	-	678	-
	GTX 1080		1,478	
Seo et al. [27]	GTX 680	-	139	-
Beak et al. [2]	GTX 470	-	-	46
This work	RTX 2070	3,063	2,593	468
	Ryzen 5 3600(CPU)	1.38	3.9	0.98

Table 10 shows the kernel performance for the target algorithm implemented in the GPU. Kernel operating performance excluding memory copy time was measured by throughput (Gbps). Performance was measured in CHAM-128/128, LEA-128/128, and HIGHT-64/128, and the number of threads was fixed to 256 to properly utilize the GPU memory space. Compared to the existing CPU implementation, it was possible to obtain 2219, 664, and 477 times improved performance for each of CHAM, LEA, and HIGHT when encrypted with GPU. This result showed improved performance for other studies conducted in the past, and in the case of HIGHT and CHAM, few optimization studies were conducted in the existing GPU environment.

CHAM-CTR on GPU In the case of CHAM shown in Figure 32, the performance of applying only the simple ECB mode parallel operation implementation was measured in 2133, 2171, and 2184 MB/s for CHAM-64/128, CHAM-128/128, and CHAM-128/256 respectively. In addition, when the optimization was implemented in the CTR mode that does not copy plaintext from the existing ECB operation mode, it can be seen that the performance of each increased to 2976,

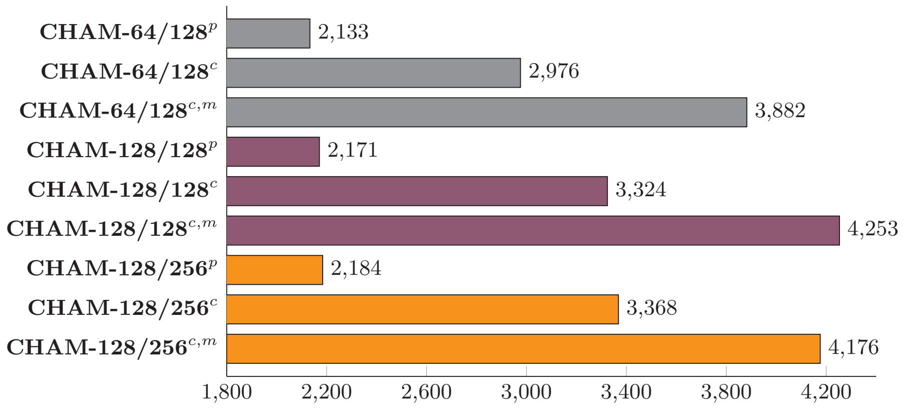


Fig. 32: Comparison of throughput (MB/s) for CHAM implementations on GPU platform including memory copy time, where p, c, and m represent parallel ECB mode of operation, parallel CTR mode of operation, and GPU memory optimization, respectively.

3324, and 3368 MB/s. Finally, when the encryption was split asynchronously using CUDA stream, the performance for each was 3882, 4253, and 4176 MB/s, up to 81, 95, and 91% performance improvement over the simple parallel implementation without optimization.

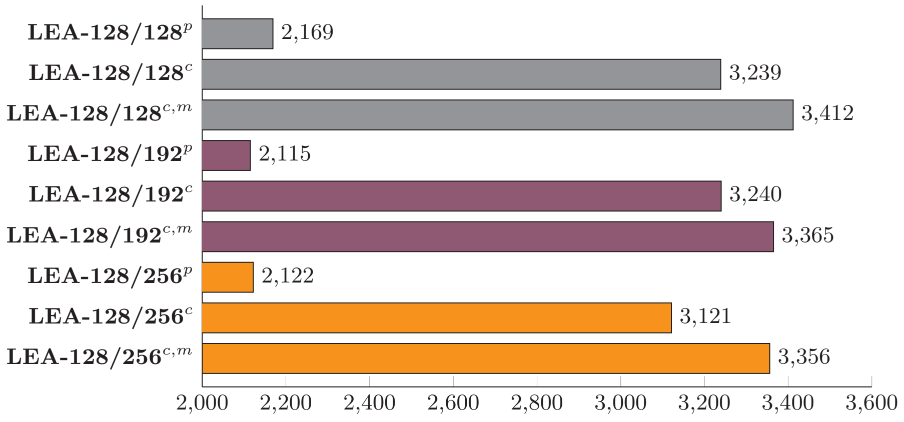


Fig. 33: Comparison of throughput (MB/s) for LEA implementations on GPU platform including memory copy time, where p, c, and m represent parallel ECB mode of operation, parallel CTR mode of operation, and GPU memory optimization, respectively.

LEA-CTR on GPU In the case of LEA shown in Figure 33, the performance of applying only the simple ECB mode parallel operation implementation was measured in 2169, 2115, and 2122 MB/s for LEA-128/128, LEA-128/192, and LEA-128/256 respectively. In addition, when the optimization was implemented in the CTR mode that does not copy plaintext from the existing ECB operation mode, it can be seen that the performance of each increased to 3239, 3240, and 3121 MB/s. Finally, when the encryption was split asynchronously using CUDA stream, the performance for each was 3412, 3365, and 3356 MB/s, up to 57, 59, and 58% performance improvement over the simple parallel implementation without optimization.

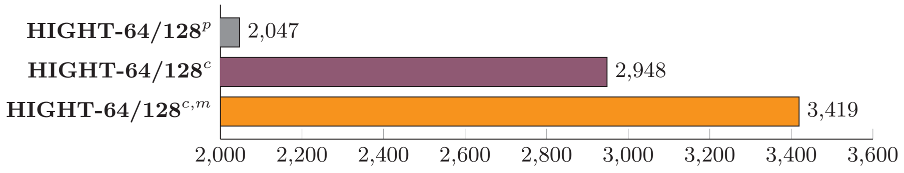


Fig. 34: Comparison of throughput (MB/s) for HIGHT implementations on GPU platform including memory copy time, where p, c, and m represent parallel ECB mode of operation, parallel CTR mode of operation, and GPU memory optimization, respectively.

HIGHT-CTR on GPU In the case of HIGHT shown in Figure 34, the performance of applying only the simple ECB mode parallel operation implementation was measured in 2047 MB/s for HIGHT-64/128. In addition, when the optimization was implemented in the CTR mode that does not copy plaintext from the existing ECB operation mode, it can be seen that the performance of increased to 2948 MB/s. Finally, when the encryption was split asynchronously using CUDA stream, the performance was 3419 MB/s, up to 67% performance improvement over the simple parallel implementation without optimization.

5.3 CTR_DRBG on AVR Microcontrollers

The proposed CTR_DRBG was implemented in `Atmel Studio 7`, the same implementation environment as section 5.1. Also, the code was complied in `-O2` option. We implemented CTR_DRBG using the optimization method proposed in section 3 and 4. Therefore, we measure the ratio of performance improvement by comparing our proposed implementation of CTR_DRBG (using Optimized CHAM-CTR, Optimized LEA-CTR, and Optimized HIGHT-CTR) and CTR_DRBG with CHAM [23], LEA [24][26], and HIGHT[25].

Table 11 shows the ratio of performance improvement to Derivation Function and Update Function, and shows the ratio of performance improvement in

Table 11: Performance Comparison of Derivation function and Update function to the previous best results on AVR [23, 25, 24, 26]. The result is based on the number of extracted random numbers, where B, *D.Fnc*, *U.Fnc*, and *E.Fnc* represent byte, Derivation Function, Update Function, Extract Function, respectively.

Block Cipher		**CHAM-64/128**	**CHAM-64/128(32)**	**CHAM-128/128**	**CHAM-128/256**
D.Fnc		5.9%	5.9%	11.0%	13.3%
U.Fnc		45.6%	45.6%	56.4%	68.6%
32B	*E.Fnc*	6.9%	5.2%	4.3%	5.4%
64B	*E.Fnc*	8.7%	7.4%	5.3%	6.3%
128B	*E.Fnc*	10.3%	9.2%	6.4%	7.2%
256B	*E.Fnc*	11.9%	10.3%	7.3%	8.1%
512B	*E.Fnc*	13.0%	11.8%	8.0%	8.7%
1024B	*E.Fnc*	13.6%	12.0%	8.4%	9.1%
Block Cipher		**LEA-128/128**	**LEA-128/192**	**LEA-128/256**	**HIGHT-64/128**
D.Fnc		10.1%	13.4%	14.1%	5.6%
U.Fnc		51.1%	69.4%	72.4%	40.6%
32B	*E.Fnc*	13.6%	22.0%	23.5%	1.4%
64B	*E.Fnc*	16.7%	25.1%	26.5%	1.9%
128B	*E.Fnc*	20.2%	28.7%	29.9%	2.4%
256B	*E.Fnc*	23.4%	31.9%	32.8%	3.0%
512B	*E.Fnc*	25.8%	34.3%	35.0%	3.3%
1024B	*E.Fnc*	27.3%	35.8%	36.4%	3.5%

Extract Function depending on the length of the extract random number. Actually, since there are no CTR_DRBG implementations on AVR platforms, for comparison we estimate the performance of CTR_DRBG based on the result of block cipher implementations from [23, 25, 24, 26]. When measuring the ratio of performance improvement, Input Data of the Derivation Function was fixed at 64-byte (Which is reasonable because on AVR platforms, the noise data is typically collected from hardware noise sources, which contain large entropy than software noise sources). The encryption process as much as Len_seed is omitted in the Derivation Function proposed in section 4. The block ciphers except CHAM-64/128 and HIGHT-64/128 show a performance improvement of more than 10% as the Block size is 128-bit. The actual computation of the method, proposed in section 4, in Update Function is only the XOR operation of seed length. Since, in Update Function, as much encryption process as Len_seed has been omitted, the ratio of performance improvement is much larger than that of Derivation Function. We measure performance according to the length of extracted random number in Extract Function. Also, we implemented Extract Function using proposed methods by sections 3 and 4. In other words, our implementation generates a Look-Up table when the Counter is V+1, and use the Look-Up table when the Counter is V+2. Therefore, it can be seen that the longer the extract random number length, the greater the ratio of the performance improvement of the Extract Function for each algorithm.

Table 12: Performance Comparison of CTR_DRBG to the previous best results on AVR [23, 25, 24, 26]. The result is based on the number of extracted random numbers, Byte represent number of byte with best performance of CTR_DRBG

Block Cipher	**CHAM-64/128**	**CHAM-64/128(32)**	**CHAM-128/128**	**CHAM-128/256**
Byte	32	32	32	32
CTR_DRBG	**15.3%**	**14.3%**	**16.6%**	**21.1%**
Block Cipher	**LEA-128/128**	**LEA-128/192**	**LEA-128/256**	**HIGHT-64/128**
Byte	1024	1024	1024	32
CTR_DRBG	**26.7%**	**36.2%**	**37.2%**	**8.7%**

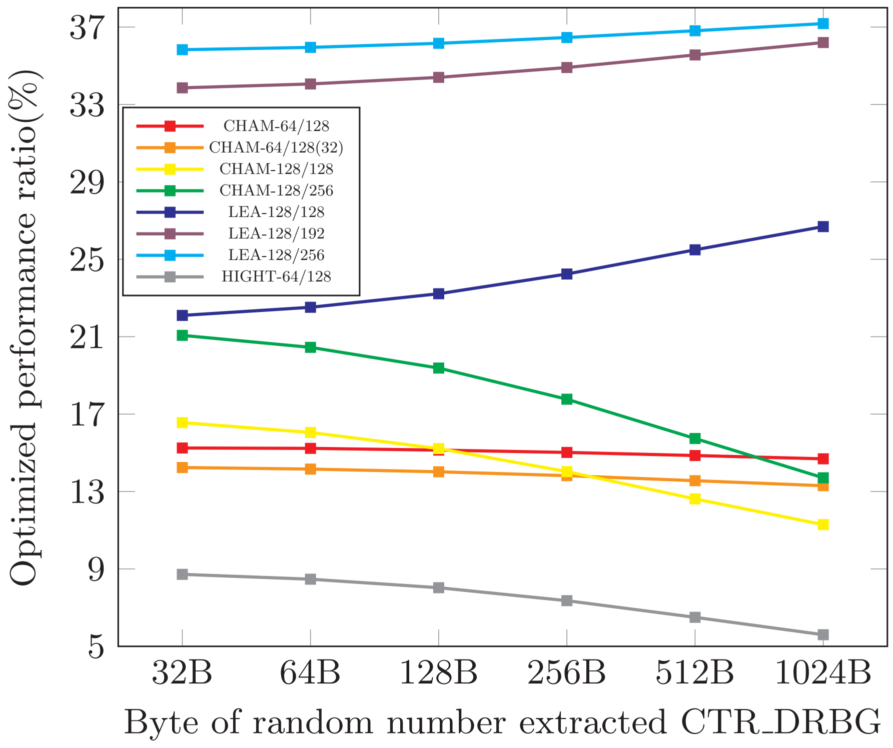


Fig. 35: Comparison of optimized performance ratio (%) for CHAM, LEA, and HIGHT implementations on 8-bit AVR microcontroller [25, 23, 24, 26]. The result is based on the number of extracted random numbers. B represents byte.

Figure 35 and table 12 show the ratio of performance improvement for CTR_DRBG with the target block ciphers according to the length of the extracted random number. And, table 12 shows the ratio of best performance improvement for CTR_ DRBG with the length of the extracted random number which results best performance. The optimized implementation of CTR_DRBG used LEA-128/128, LEA-128/192, and LEA-128/256 increases the ratio of performance improvement as the length of the extracted random number increases. In table 11, the ratio of performance improvement to Extract Function for LEA-128/128, LEA-128/192, and LEA-128/256 increases by a greater width than other Cipher algorithms(CHAM, HIGHT) as the length of the extracted random number increases. Therefore, the ratio of performance improvement to Extract Function for LEA-128/128, LEA-128/192, and LEA-128/256 affects the performance improvement ratio of CTR_DRBG over the ratio of performance improvement for Derivation Function and Update Function. CHAM-64/128 has a difference of approximately 7% in ratio of performance improvement between 32-byte and 1024-byte extracted random number, and the overall ratio of performance improvement of CTR_DRBG does not increase. In other words, the ratio of performance improvement in Extract Function is less affected by CTR_DRBG ratio of performance improvement than ratio of performance improvement for Derivation Function and Update Function.

The ratio of performance improvement of CTR_DRBG in CHAM-128/128, CHAM-128/256, and HIGHT-64/128 drops as the length of the extracted random number increases. Table 11 shows that the performance increase in the Extract Function of the three functions results in a performance improvement of less than 4% as the length of the extracted random number increases. According to our observation, In the ratio of performance improvement to extracted 32-byte random numbers from Extract Function and 1024-byte, If the difference between the ratio of performance improvement when extracting random numbers of is less than 8.3%, the ratio of performance improvement of CTR_DRBG does not increase depending on the length of the extracted random number. The longer the extracted random number is, the more encryption process is added. The longer the extracted random number, the more encryption process is added. Therefore, if a ratio of performance improvement of Extract Function is significantly less than the performance improvement ratio for Derivation Function and Update Function, the performance improvement ratio of CTR_DRBG is lower. The CTR_DRBG optimization method proposed in this paper shows up to 37.2% performance improvement when using LEA, up to 22.1% performance improvement when using CHAM. In addition, the HIGHT-64/128 improves performance by up to 8.7%.

5.4 CTR_DRBG on GPU

CTR_DRBG was also performed in the same environment in which the CTR mode optimization implementation was tested. The performance of the GPU CTR_DRBG optimization implementation was measured based on when the CTR_DRBG was called, and ended when the random number sequence was output. The performance measurements were measured in accordance with various experimental environment variables, and the set experimental environment variables were divided into four types: block cipher type, output random number sequence size, prediction resistance, and additional input.

When the optimization is implemented, the progressing function varies depending on the predict resistance and whether additional input is performed, but the extract function, which is the main element of CTR_DRBG operating as the GPU, is not affected. Therefore, the results provided in Figure 36 measured the performance while changing the size of the random number output while the environment was fixed as one. In the performance measurement, the result of repeating the entire process 1000 times is presented as an average.

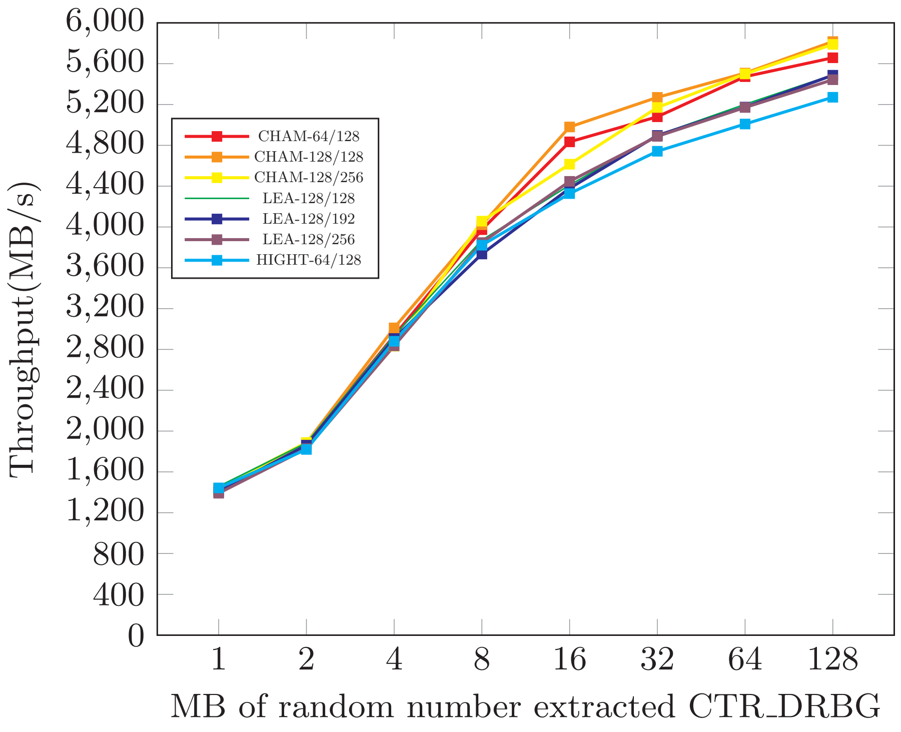


Fig. 36: Comparison of throughput (MB/s) for CHAM, LEA, and HIGHT implementations on GPU platform [26, 25, 23, 24].

Figure 36 shows the result of throughput(MB/s) for the CTR_DRBG optimization implementation on the GPU according to the random number output size. It can be seen that performance of all the cryptographic algorithms implemented increases as the output random number increases in common. When outputting a 128 MB random number compared to the random number output size of 1 MB, it can be seen that the performance increases up to 5 times.

Based on the 128 MB random number output, the throughput of the CPU is 169.6, 235.8, and 100.9 MB/s respectively for CHAM-128/128, LEA-128/128, and HIGHT-64/128, and when the random number is output through the GPU, up to 34.2, 24.8, and 52.2 for each CPU The throughput was doubled.

6 Conclusion

In this paper, we have presented optimized implementations of ARX-based Korean block ciphers such as CHAM/LEA/HIGHT with CTR mode of operation and CTR_DRBG using them on low-end 8-bit AVR microcontrollers and CUDA-enbabled GPU. With respect to CTR mode optimization, our implementations are accelerated with the re-designed look-up table and parallel computations. The look-up table approach skips computations and parallel computation improved the throughput. On AVR platforms, our optimized CTR implementations based on revised CHAM, LEA, and HIGHT provide 8.9∼5.8%, 6.1∼5.0%, and 3.9% improvements compared with the previous best results, respectively. On the target GPU, our implementations with revised CHAM, LEA, and HIGHT provide 2219, 664, and 477 times of improved throughput compared with corresponding CPU implementations. Regarding to CTR_DRBG optimization, we propose to precompute several parts of CTR_DRBG, which results in performance improvement. Through the proposed techniques, our implementations of CTR_DRBG using LEA, revised CHAM, and HIGHT on AVR platforms provide 37.2%, 21.1%, and 8.7% of performance improvement compared with the previous works. On the target GPU platform, our implementations using revised CHAM, LEA, and HIGHT provide 34.2, 24.8, and 52.2 times enhanced throughput over CPU implementations, respectively. We believe that our work can be widely used for building various types of secure IoT services. Furthermore, the optimization techniques from this work can be applied to the other platforms without difficulties.

References

1. S. An and S. Seo. Highly efficient implementation of block ciphers on graphic processing units for massively large data. *Applied Sciences*, 10, 2020.
2. E.-T. Baek and M.-K. Lee. Speed-optimized implementation of hight block cipher algorithm. *Journal of the Korea Institute of Information Security Cryptology*, 22:495–504, 2012.
3. S. Banik, S. K. Pandey, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo. GIFT: a small PRESENT. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.
4. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK block ciphers on AVR 8-bit microcontrollers. In *International Workshop on Lightweight Cryptography for Security and Privacy*, pages 3–20. Springer, 2014.
5. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. Simon and speck: Block ciphers for the internet of things. *IACR Cryptology ePrint Archive*, 2015:585, 2015.
6. R. Beaulieu, S. Treatman-Clark, D. Shors, B. Weeks, J. Smith, and L. Wingers. The SIMON and SPECK lightweight block ciphers. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.
7. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In

International Workshop on Cryptographic Hardware and Embedded Systems, pages 450–466. Springer, 2007.

8. T. Eisenbarth, Z. Gong, T. Güneysu, S. Heyse, S. Indesteege, S. Kerckhof, F. Koeune, T. Nad, T. Plos, F. Regazzoni, et al. Compact implementation and performance evaluation of block ciphers in ATtiny devices. In *International Conference on Cryptology in Africa*, pages 172–187. Springer, 2012.
9. 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*, pages 3–27. Springer, 2013.
10. 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*, pages 46–59. Springer, 2006.
11. 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*, pages 46–59. Springer, 2006.
12. B. Kim, J. Cho, B. Choi, J. Park, and H. Seo. Compact implementations of HIGHT block cipher on IoT platforms. *Security and Communication Networks*, 2019, 2019.
13. K. Kim, S. Choi, H. Kwon, H. Kim, Z. Liu, and H. Seo. PAGE—practical AES-GCM encryption for low-end microcontrollers. *Applied Sciences*, 10(9):3131, 2020.
14. K. Kim, S. Choi, H. Kwon, Z. Liu, and H. Seo. FACE–LIGHT: Fast AES–CTR mode encryption for low-end microcontrollers. In *International Conference on Information Security and Cryptology*, pages 102–114. Springer, 2019.
15. 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 (ICISC'17)*, 2017.
16. W. K. Lee, B.-M. Goi, and R. C.-W. Phan. Terabit encryption in a second: Performance evaluation of block cipher in gpu with kepler, maxwell, and pascal architectures. *Concurrency and Computation Practice and Experience*, 31, 2018.
17. M. A. Mazidi, S. Naimi, and S. Naimi. *AVR Microcontroller and Embedded Systems*. Pearson India, 2010.
18. D. McGrew and J. Viega. The Galois/counter mode of operation (GCM). *submission to NIST Modes of Operation Process*, 20, 2004.
19. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *International Workshop on Fast Software Encryption*, pages 75–93. Springer, 2010.
20. 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*, pages 469–499, 2018.
21. M. Park and J. Yoon. Optimization of lightweight encryption algorithm (lea) using threads and shared memory of gpu. *Journal of the Korea Institute of Information Security Cryptology*, pages 719–726, 2015.
22. T. Park, H. Seo, S. Lee, and H. Kim. Secure data encryption for cloud-based human care services. *Journal of Sensors*, 2018, 2018.
23. D. Roh, B. Koo, Y. Jung, I. W. Jeong, D.-G. Lee, D. Kwon, and W.-H. Kim. Revised version of block cipher CHAM. In *International Conference on Information Security and Cryptology*, pages 1–19. Springer, 2019.
24. H. Seo, K. An, and H. Kwon. Compact LEA and HIGHT implementations on 8-bit AVR and 16-bit MSP processors. In *International Workshop on Information Security Applications*, pages 253–265. Springer, 2018.

25. H. Seo, I. Jeong, J. Lee, and W.-H. Kim. Compact implementations of ARX-based block ciphers on IoT processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(3):1–16, 2018.
26. H. Seo, Z. Liu, J. Choi, T. Park, and H. Kim. Compact implementations of LEA block cipher for low-end microprocessors. In *International Workshop on Information Security Applications*, pages 28–40. Springer, 2015.
27. H. Seo, Z. Liu, T. Park, H. Kim, Y. Lee, J. Choi, and H. Kim. Parallel implementations of lea. *International Conference on Information Security and Cryptology*, pages 256–274, 2013.