IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 16 August 2022, accepted 7 September 2022, date of publication 21 September 2022,
date of current version 27 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3208247

RESEARCH ARTICLE

Parallel Implementation of PIPO and Its Application for Format Preserving Encryption

HYUNJI KIM, HYUNJUN KIM, SIWOO EUM, HYEOKDONG KWON, YUJIN YANG, AND HWAJEONG SEO
IT Convergence Division, Hansung University, Seoul 02876, South Korea

Corresponding author: Hwajeong Seo (hwajeong84@gmail.com)

This work was supported in part by the Institute for Information and Communications Technology Promotion (IITP) Grant through the Korea Government (MSIT), Research on Blockchain Security Technology for IoT Services (50%), under Grant 2018-0-00264; in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant through the Korea Government (MSIT), Development of Fast Design and Implementation of Cryptographic Algorithms based on GPU/ASIC (25%), under Grant 2021-0-00540; and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant through the Korea Government (MSIT), Development of Lightweight BIoT technology for Highly Constrained Devices (25%), under Grant 2022-0-00627.

ABSTRACT The PIPO block cipher, a domestic lightweight block cipher, was announced at ICISC'20. In particular, the bitslicing technique is implemented in the S-Layer for the PIPO block cipher. Because this is a part that can be operated in parallel, we implemented the PIPO block cipher efficiently in a parallel approach through AVX2 instructions, and provide implementations for ECB and CTR modes. Compared to the existing PIPO implementation, we achieved a performance improvement by 7.345×. In addition, we applied the AVX2-PIPO implementation to the round function of format-preserving encryption. When repeatedly encrypting 128-byte plaintext, we achieved performance similar to that of the existing FF1-AES implementation. The FF1-AVX2-PIPO implementation successfully encrypted the database and enabled efficient database management in terms of memory space and speed factor. Finally, AVX2-PIPO-CTR and FF1-AVX2-PIPO were applied to image processing. In the case of CTR mode, the encryption performance was better than that of ECB mode. Partial encryption with object detection and FF1-AVX2-PIPO was successfully performed, and it is expected that privacy protection in CCTV or image processing can be improved.

INDEX TERMS PIPO block cipher, parallel implementation, format preserving encryption.

I. INTRODUCTION

Recently, with the development of big data and deep learning technologies, the need for large-scale databases encryption and management has increased. Format preserving encryption can enable memory-efficient database encryption through constant data input/output length and format preservation, which are advantages of format-preserving encryption. Parallel operation on large-scale data is also needed to improve the performance [1].

PIPO (Plug-In Plug-Out), a lightweight block cipher presented at ICISC'20 [2], is applied with the bit-slice technique, which is an efficient implementation method for performing parallel operations. Therefore, the PIPO block cipher is suitable for the parallel implementation using AVX2 instruction sets. In addition, it seems that parallel encryption and format-preserving encryption algorithms can be applied to image processing to protect privacy.

The associate editor coordinating the review of this manuscript and approving it for publication was Jiafeng Xie.

A. CONTRIBUTION

1) PARALLEL IMPLEMENTATION OF PIPO BLOCK CIPHER USING AVX2 INSTRUCTIONS

We implemented the PIPO block cipher in a parallel way using AVX2 instructions, and it achieved 7.345× improved performance compared to the reference implementation by applying an efficient rotation method. In addition, it supports

ECB and CTR modes to provide scalability. This can be applied to various applications.

2) FORMAT PRESERVING ENCRYPTION WITH THE PARALLEL PIPO BLOCK CIPHER USING AVX2 INSTRUCTIONS

We successfully applied the parallel implementation of the PIPO block cipher to the format-preserving cipher. This implementation is called FF1-AVX-PIPO. Among pseudo random functions and CIPH functions (encryption function) inside the round function, the CIPH function was changed to the parallel PIPO block cipher using AVX2, and it was confirmed that encryption and decryption were successfully conducted.

3) CASE STUDIES ON VARIOUS APPLICATIONS

We applied the proposed implementation to database encryption and image processing. Database encryption through FF1-AVX2-PIPO showed excellent performance in terms of memory usage. In the case of short plaintext, it achieved faster speed than the method provided by MySQL in the process of encrypting and storing it in the database. In addition, FF1-AVX2-PIPO was successfully applied to image processing such as partial encryption after object detection. This approach also solves problems involved in image encryption, such as the waste of storage space associated with data padding and, not being able to identify an image without decoding.

II. RELATED WORKS

A. LIGHTWEIGHT BLOCK CIPHER PIPO

PIPO is a lightweight block cipher that outperforms other 64-bit lightweight block ciphers in an 8-bit AVR environment [2]. It has 64-bit input/output and 128-bit (64/128) and 256-bit (64/256) key size, and is designed with the SPN (Substitution Permutation Network) structure. Depending on the length of the key, 13 rounds and 17 rounds are performed respectively. The round consisting of the S-layer performing S-box operation, which is a non-linear operation, the R-layer performing rotation, and key addition is repeated. For the S-layer, a bit-slicing implementation using 11 nonlinear operations and 23 linear bit operations and TLU (Table Look-up) implementation using a lookup table are provided.

B. ADVANCED VECTOR EXTENSIONS (AVX)

AVX2 is supported by 64-bit Intel processors. AVX2 is an extension of AVX that includes 256-bit integer arithmetic [3]. We can use the instructions supported by AVX in C/C++ environment through the AVX2 intrinsic function. A 256-bit register is used (epi8 and epi16 represent a vector containing an unsigned integer). As shown in Table 1, AVX2 also provides instructions, such as logic, combination, and permutation in units of each vector size [4].

TABLE 1. AVX2 instruction set for parallel PIPO implementation.

Instruction	Description
_mm256_loadu_si256	Load 256-bit of integer data from memory
_mm256_set1_epi16	Broadcast 16-bit integer to all elements of dst
_mm256_xor_si256	Bitwise XOR of 256 bits
_mm256_and_si256	Bitwise AND of 256 bits
_mm256_or_si256	Bitwise OR of 256 bits
_mm256_andnot_si256(A,b)	Bitwise NOT of 256 bits in a and then AND with b
_mm256_slli_epi16 / _mm256_srli_epi16	Shift packed 16-bit integers in a left/right

C. FORMAT PRESERVING ENCRYPTION (FPE)

FF1 [5] is a standardized name for FFX[Radix]. It was proposed by Mihir Bellare, Phillp Rogaway, and Terence Spies in 2010 [6]. It is a Feistel structure. Plaintext, key, radix, and tweak are used as input values, and ciphertext is generated through 10-rounds. In the round function, AES-128 [7] is used, and this part can be changed to another block cipher [8]. The algorithm was described together with the proposed method in Section III-B. In [9], a method was proposed to improve the speed of FF1 and FF3-1 [5] by implementing the algorithm by changing to the lightweight encryption algorithm (LEA) and SPECK, which are lightweight block ciphers. They experimented with high performance computers and low-power devices, and FF1 with LEA and SPECK applied in high-performance computer environments shortened the encryption time. However, it was found that the efficiency of using lightweight block ciphers decreases after 10,000 iterations. In the IoT (Internet of Things) environment, except FF1-LEA, it shows faster encryption speed in the IoT environment. In addition, among their proposed methods, implementations of FF1 greatly improved the encryption speed when short data was encrypted, but the speed improvement was low when the length of the plain text was long or was repeated several times. In [10], a domestic format-preserving encryption algorithm (FEA) was implemented in a parallel way through ARM-NEON (ARM's SIMD (Single Instruction Multiple Data) instruction for parallel processing), SSE (SIMD instruction set using 128-bit registers), and AVX2. A parallel processing method for the lookup table was implemented, and an optimal implementation for the low-power IoT environment was also proposed. In [11], they proposed an efficient implementation of Format-preserving Encryption Algorithm (FEA), which is the Korean standard of FPE, and the first-order masked implementation of FEA on both low-end (i.e., AVR microcontroller) and high-end (i.e., ARM processor) IoT devices.

D. PARALLEL IMPLEMENTATION OF PIPO BLOCK CIPHER

There is no parallel implementation of PIPO in the same environment yet, there are implementation on RISC-V [12] and 64-bit ARM processor. In [13], they implemented parallel processing for the PIPO block cipher on RISC-V. An efficient 8-bit unit R-layer function is implemented on a 32-bit register, and parallel implementation is presented in terms of memory optimization and speed optimization. In addition, they proposed optimal implementations for ECB (Electronic Code Block), CBC (Cipher Block Chaining) and CTR (Counter Mode) modes. The performance measurement

of parallel implementation of the ECB operation mode show 84.85cpb (memory optimization) and 59.93cpb (speed optimization). This shows performance improvements of 1.79× (memory optimization) and 2.53× (speed optimization) compared to a single implementation. In addition, performance improvements of 1.34× (memory optimization) and 1.89× (speed optimization) were confirmed compared to the existing implementation [14] that includes key schedules. Similar to the ECB mode, it shows similar performance improvement in the CTR operation mode, and the similar performance is also measured by applying the parallel implementation technique to the decoding process of the CBC operation mode. In [15], they proposed a parallel implementation for block cipher PIPO on the A10× fusion processor. In particular, the rotation operation of the R-layer was implemented using only two instructions, and the operation speed was greatly improved. The original PIPO 64/128 shows 34.6 cpb, and the 64/256 shows 44.7 cpb. On the other hand, the parallel implementation on eight plaintexts of the proposed technique has performance of 12.0 cpb and 15.6 cpb in 64/128 and 64/256 standards, respectively. In addition, parallel implementation on 16 plaintexts has performance of 6.3 cpb and 8.1 cpb in 64/128 and 64/256 standards, respectively. As a result, compared to the original PIPO, the parallel implementations of 8 plaintexts for each 64/128 and 64/256 standard showed higher performance by 65.3%, 66.4%, respectively. And compared to the original PIPO, the parallel implementations of 16 plaintexts are improved 81.8% and 82.1%, respectively.

III. PROPOSED METHOD

A. PARALLEL IMPLEMENTATION OF PIPO BLOCK CIPHER USING AVX2

We propose an AVX2-based parallel PIPO algorithm (AVX2-PIPO) utilizing the bit-slicing implementation of the PIPO block cipher suitable for parallel operation. The original PIPO block cipher has 64-bit plaintext, and in the encryption process (S-Layer, R-layer), it is calculated in units of bytes (8-bit). Therefore, we implemented parallel operation for 8-bit unit operation, and for this, we used AVX2's 256-bit register. That is, the parallel implementation does not perform 8-bit but 256-bit operations. Finally, the existing PIPO implementation encrypts one 64-bit plaintext, but with our parallel implementation, 16 64-bit plaintext can be encrypted at a time.

Figure 1 shows the architecture of AVX2-PIPO. In this work, the input and output of PIPO are 128-byte, and the process of arrangement and packing plaintext is added before entering the r-round. A process of rearranging the ciphertext after encryption is completed is also required. This requires pre-processing and post-processing, which ensure 16 64-bit plaintext in a single instruction. The whole process performs key addition in round 0, and then repeats the S-Layer, R-Layer, and key addition process in every round. Therefore, we implemented plaintext arrangement and ciphertext

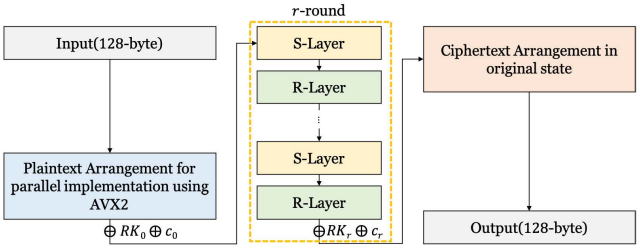


FIGURE 1. Encryption structure of parallel implementation of PIPO block cipher.

rearrangement processes as well as, parallel operations for encryption processes. For efficient parallel implementation, optimized instruction usage, arrangement, and rotation are applied.

1) ARRANGEMENT OF PLAINTEXT FOR PARALLEL IMPLEMENTATION

In this approach, 16 64-bit plaintexts are input by utilizing 8 256-bit registers to implement PIPO's byte unit operation in a parallel way. As shown in Figure 2, each plaintext consists of 8 bytes, which are divided into byte units and input into the temp array. That is, the $n-1^{th}$ bytes of each plaintext are collected and stored in temp[$n-1$] (The length is 16). In other words, the first byte ($n=1$) of the 16^{th} plaintext is input to temp[0][15]. Each row of the temp array (i.e. temp[0], temp[1]) is loaded into each element of the 256-bit register array using the *_mm*256*_loadu_si*256 instruction. As shown in Figure 3, 128-bit data (16 8-bit data) is stored in one 256-bit register. Each 8-bit data is packed into 16-bit. Finally, these 256-bit register arrays ($T256$) are used as input to key addition process.

2) KEY ADDITION

In the key addition process, as shown in Figure 4, the 256-bit register array ($T256$) received as an input and the round key performs XOR operation. Because the round key is the same for all plaintext, it is not implemented in a parallel way. The round key performs bitwise XOR with $T256$. The n-th bytes of the round key are set in the 256-bit register array for mapping to the n-th byte of the plaintext. For this, the n-th byte of the round key is packed by 16-bit and set using the _mm256_set1_epi16 instruction. It is possible to add round keys for a total of 16 plaintexts by eight operations.

3) S-LAYER

In the S-Layer, the substitution operation can be performed through bit-slice or lookup table. In the implementation, we use the bit-slice method, which is suitable for parallel implementation. Algorithm 1 shows the bit-slice method for s-layer using AVX2. The output of the key addition process is used as the input of the S-Layer. Because the same bytes were gathered, the same array index as that in the existing implementation is used for access. The operation between the bytes of 1 plaintext is also the same for 16 plaintexts. However,

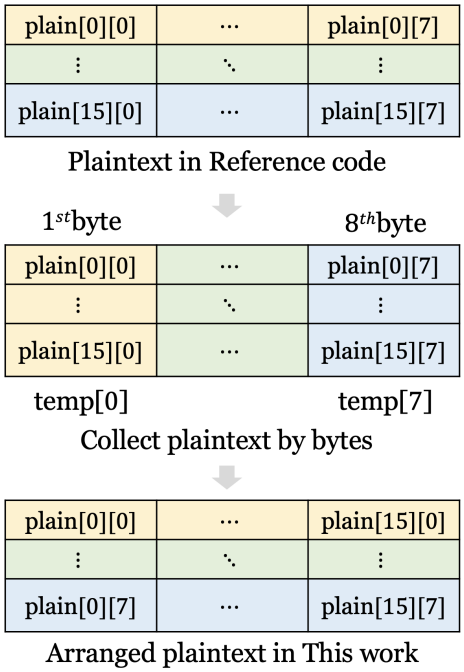


FIGURE 2. Plaintext arrangement by bytes for using AVX2.

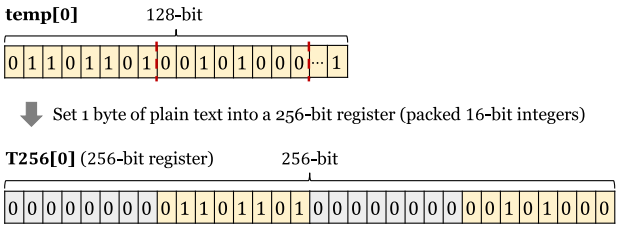


FIGURE 3. Allocation of 1-byte data to 256-bit registers packed in 16-bit units.

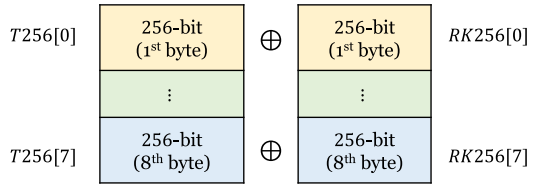


FIGURE 4. Key addition operation of proposed implementation.

the AVX2 instruction does not support the NOT operation used in the S-Layer. Therefore, the _mm_andnot_si256(A,b) instruction in line 17, is used to perform a NOT operation for the value stored in the A register, and then an AND operation is performed with b. An *ANDmask* (b) is needed to preserve the value of A after the NOT operation. Therefore, the value of *ANDmask* is set for all bits to 1.

4) R-LAYER

The PIPO block cipher performs byte unit operations. Because the same operation is performed on the same byte, 32 plaintexts must be processed concurrently to maximize the utilization of the 256-bit registers. However, the AVX2 instruction does not support 8-bit shift operation for the 8-bit rotation operation. Therefore, it is necessary to change the 256-bit register packed into 8-bit units before and after the operation of the R-layer in 16-bit units. For this reason, additional registers are needed. Registers must be re-packed and unnecessary parts are added to the operation for the actual 8-bit rotation. To prevent this, we propose parallel processing for 16 plaintext blocks. As mentioned above, the re-packing process to 16-bit is not necessary because plaintext blocks are

Algorithm 1 Bitslice for S-Layer Using AVX2

Input: 256-bit register array of size 8 packed in 16-bit units ($T256$), 256-bit temp register array (T), and 256-bit register for masking (*ANDmask*).

Output: 256-bit register array of size 8 packed in 16-bit units ($T256$).

1: Let *XOR* be the notation for *_mm*256*_xor_si*256, which is an *xor* operation between 256-bit registers.
2: Let *AND* be the notation for *_mm*256*_and_si*256, which is an *and* operation between 256-bit registers.
3: Let *OR* be the notation for *_mm*256*_or_si*256, which is an *or* operation between 256-bit registers.
4: $T[3] = \_mm256\_set1\_epi16(0 \times 00)$;
5: *__m256iANDmask*;
6: $Notmask = \_mm256\_set1\_epi16(0xFF)$;
7: $T256[5] = XOR(T256[5], AND(T256[7], T256[6]))$;
8: $T256[4] = XOR(T256[4], AND(T256[3], T256[5]))$;
9: $T256[7] = XOR(T256[7], T256[4])$;
10: $T256[6] = XOR(T256[6], T256[3])$;
11: $T256[3] = XOR(T256[3], OR(T256[4], T256[5]))$;
12: $T256[5] = XOR(T256[5], T256[7])$;
13: $T256[4] = XOR(T256[4], AND(T256[5], T256[6]))$;
14: $T256[2] = XOR(T256[2], AND(T256[1], T256[0]))$;
15: $T256[0] = XOR(T256[0], OR(T256[2], T256[1]))$;
16: $T256[1] = XOR(T256[1], OR(T256[2], T256[0]))$;
17: $T256[2] = \_mm256\_andnot\_si256(T256[2], ANDmask)$;
18: $T256[7] = XOR(T256[7], T256[1])$;
19: $T256[3] = XOR(T256[3], T256[2])$;
20: $T256[4] = XOR(T256[4], T256[0])$;
21: $T[0] = T256[7]$; $T[1] = T256[3]$; $T[2] = T256[4]$;
22: $T256[6] = XOR(T256[6], AND(T[0], T256[5]))$;
23: $T[0] = XOR(T[0], T256[6])$;
24: $T256[6] = XOR(T256[6], OR(T[2], T[1]))$;
25: $T[1] = XOR(T[1], T256[5])$;
26: $T256[5] = XOR(T256[5], OR(T256[6], T[2]))$;
27: $T[2] = XOR(T[2], AND(T[1], T[0]))$;
28: $T256[2] = XOR(T256[2], T[0])$;
29: $T[0] = XOR(T256[1], T[2])$;
30: $T256[1] = XOR(T256[0], T[1])$;
31: $T256[0] = T256[7]$; $T256[7] = T[0]$;
32: $T[1] = T256[3]$; $T256[3] = T256[6]$; $T256[6] = T[1]$;
33: $T[2] = T256[4]$; $T256[4] = T256[5]$; $T256[5] = T[2]$;
34: **return** $T256$

packed in 16-bit units in the plaintext arrangement process. In this condition, an efficient 8-bit rotation operation can be performed. Algorithm 2 shows details of the proposed 16-bit rotation. Here, *num* is an array that stores rotation constants for each byte to rotate. Because the first byte does not perform the rotation operation, only operations on the other 7 registers need to be performed. For the rotation operation, after left and right shift operations are performed on the original register, OR operation is performed. For this, $MASK_R$ is needed. The 0's are allocated to the upper 8-bits of $MASK_R$. The 1's are allocated to $num[i]$-bits starting with the most significant bit of the lower 8-bit of $MASK_R$. The value of the AND operation of the original register ($T256[i+1]$) and $MASK_R$ is shifted right by $8 - num[i]$, and $T256[i+1]$ is shifted left by the rotation constant. The OR operation is performed for these two results, and then the final result is stored in the original register ($T256[i+1]$). This approach enables efficient 8-bit rotation. In addition, after all rounds are finished, only the lower 8-bits of each 16-bit are used as output.

Algorithm 2 16-Bit Rotation Mechanism for R-Layer Using AVX2

Input: 256-bit register array of size 8 packed in 16-bit units ($T256$), 256-bit register packed in 16-bit units for masking ($MASK_R$), and array of rotation constants (num).

Output: 256-bit register array of size 8 packed in 16-bit units ($T256$).

1: $num[7] = \{7, 4, 3, 6, 5, 1, 2\}$
2: **for** $i = 0$, **to** 6 **do**
3: Allocate 0's to upper 8-bits of $MASK_R$
Allocate 1's to $num[i]$-bits starting with the most significant bit of lower 8-bit of $MASK_R$
4: $RES_0 \leftarrow T256[i+1] \ll num[i]$
$RES_1 \leftarrow (T256[i+1]\ \&\ MASK_R) \gg 8 - num[i]$
5: $T256[i+1] \leftarrow RES_0\ or\ RES_1$
6: **end for**
7: **return** $T256$

5) PARALLEL PIPO WITH CTR MODE USING AVX2

We implemented a counter mode (CTR mode) of the PIPO algorithm in a parallel way by using AVX2 instructions, and we call this implementation AVX2-PIPO-CTR mode. Figure 5 shows the system configuration of AVX2-PIPO-CTR mode. For the AVX2-PIPO implementation, 16 concatenated values of 32-bit nonce and 32-bit counter are required. In CTR mode, the nonce and counter are used as input to this parallel implementation, not the plaintext to be encrypted. Then, 16 encrypted 64-bit ciphertexts are XORed with 16 plaintexts to be encrypted. That is, 64-bit values combined with 32-bit nonce and 32-bit counter become input to AVX2-PIPO implementation. So we can encrypt 16 concatenated values (32-bit nonce and 32-bit counter) at once. The nonce is the same, and the counter is incremented by 1. The round key and encryption process is the same as mentioned above, and the output is used as a key to encrypt

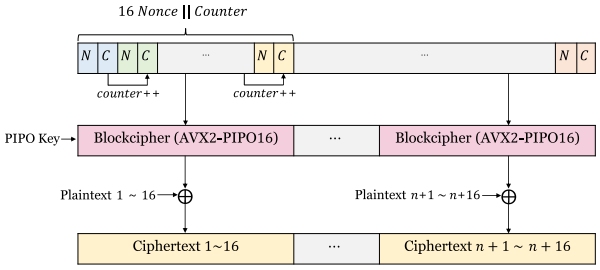


FIGURE 5. CTR mode of AVX2-PIPO.

plaintext. Therefore, the output of AVX2-PIPO and the 128-byte plaintext perform XOR operation to generate a 128-byte ciphertext. Because AVX2-PIPO also provides CTR mode to have scalability, it can be utilized in various applications.

B. FORMAT PRESERVING ENCRYPTION WITH PARALLEL PIPO

The existing FF1 [5] performs a round function every round with a feistel structure. In the round function, pseudo random function (PRF) and encryption function ($CIPH\_k(X)$) are performed, and AES-128-CBC mode and AES-128 are used respectively. In this paper, we proposed a method using AVX2-PIPO implementations for $CIPH\_k(X)$, which is used in the round function of FF1, one of the format preserving encryption methods.

Algorithm 3 shows the process of the proposed FF1 with AVX2-PIPO implementations. Inputs are tweak, tweak length, radix, key, and plaintext. The input plaintext X is split into A and B. Then, b (length of B), d, and the initial block P are set. These values are used for the encryption. Q is also generated using T, round (i), and B. Here, R is generated with initial blocks P and Q through PRF ($AES\_Encrypt$ in line 8). AVX2-PIPO is not applied to the PRF part, because PRF receives the input of a fixed length of 16 bytes. Up to line 8 in Algorithm 3, it is the same as the existing FF1 algorithm [8]. Afterward, R is encrypted through $CIPH\_k(X)$ to generate S. We change this $CIPH\_k(X)$ to the implementation of AVX2-PIPO algorithm. In the existing FF1, the values obtained by performing XOR operation with R and Q by 16 bytes were encrypted. In this work, 128-bytes are encrypted at once in a parallel way. The AVX2-PIPO encryption process consists of arranging plaintext, repeating rounds, and rearranging ciphertext, as described above. Then, S and A are added on the mod $radix^m$, and then converted to the original radix frequency of length m. Finally, the concatenated value of the last A and B becomes the ciphertext, and this process is repeated for 10 rounds.

C. APPLICATION

1) IMAGE ENCRYPTION WITH AVX2-PIPO-CTR MODE

We propose image encryption using AVX2-PIPO-CTR mode. Image processing is usually time consuming and involves doing the same and repetitive work on multiple pixels. Multiple pixels are simultaneously worked through AVX2-PIPO

Algorithm 3 FF1 Encryption Process Using AVX2-PIPO

Input: Tweak (T), Length of tweak (t), Radix (r), Key (k), and Plaintext (X)
Output: Encryption value ($A||B$)

1: Let $u = \lfloor n/2 \rfloor$; $v = n - u$
2: Let $A = X[1 \ldots u]$; $B = X[u+1 \ldots n]$
3: Let $b = \lceil \lceil v \cdot LOG(r) \rceil / 8 \rceil$
4: Let $d = 4\lceil b/4 \rceil + 4$
5: Let $P = [1]^1||[2]^1||[1]^1||[r]^3||[10]^1||[u \bmod 256]^1||[n]^4||[t]^4$
6: **for** $i = 0$, **to** 9 **do**
7: Let $Q = T||[0]^{(-t-b-1) \bmod 16}||[i]^1||[NUM_{radix}(B)]^b$
8: Let $R = AES\_Encrypt(P||Q)$
9: Let $pad = ((-t-b-1)\%16 + 16)\%16$
Let $Qlen = t + pad + 1 + b$
Let $count = Qlen$ / 16
unsigned char $R_i[16]$, unsigned char $*Q_i = Q$, unsigned char $R_i256[256]$
10: **for** $block = 0$, **to** $count - 1$ **do**
11: **for** $j = 0$, **to** 15 **do**
12: $R_i[j] = Q_i[j] \oplus R[j]$
13: $R_i256[16 * block + j] \leftarrow R_i[j]$
14: **end for**
15: $Q_i += 16$
16: **end for**
17: AVX2-PIPO_k(R_i256, R, *roundkey*); R is for ciphertext
18: *memcpy* (S, R,16)
19: Let $y = NUM(S)$
20: **if** i is even **then**
21: $m = u$
22: **else**
23: $m = v$
24: **end if**
25: $c = (NUM_r(A) + y) \bmod r^m$
26: $C = \mathrm{STR}_r^m(c)$
27: $A = B$, $B = C$
28: **end for**
29: **return** $A||B$

implemented in a parallel way. We implemented the CTR mode with high parallelism. First, the nonce, counter, and key of PIPO are inserted into AVX2-PIPO implementation to perform the encryption. The result value becomes the key used to encrypt input pixels. Then, image pixels are loaded into a 128-byte buffer. They are divided into 16 64-bit plaintexts, and the XOR operation is performed in a parallel way. Encrypted values are entered in the original place of the corresponding plain pixels. Because the 128-byte pixel to be input is independent, the speed can be further improved through the parallel implementation of the counter mode.

2) DATABASE ENCRYPTION USING FF1-AVX2-PIPO

In this paper, we propose a database encryption method using FF1-AVX2-PIPO. Because format-preserving encryption is used, the size of the data to be encrypted is not padded to match the block size of the encryption algorithm. Therefore, the input and output are maintained in the same length, preventing waste of memory storage space at the database. Because it has the same form as the original data, there is no need to change the schema of the database. When trying to store the password 'A1234' in the database table that stores the user ID and password, if the domain of the password column is a character string that combines letters and numerals, it is encrypted within that range (letters and numeral), to preserve the length and format. No additional database schema changes are required. Thus, database administration costs are not increased and system modifications are not required.

MySQL (one of the relational database management systems) provides encryption algorithms, such as AES-128, SHA2, and DES. However, as mentioned above, these cryptographic schemes waste database memory because the input and output sizes are fixed. Furthermore, these cryptographic algorithms are not suitable for maintaining database schema. To solve these problems, we use the FF1 implementation which has advantages in terms of database memory management and schema maintenance.

Algorithm 4 shows the database encryption mechanism. MySQL can be implemented in various languages. In particular, C language is used for the implementation of this work. First, the data to be encrypted through the FF1 algorithm in plain text is set. Before encryption, the MySQL server is connected, and the database to be used is selected through a query in line 2. The ciphertext generated after encryption is inserted into a table chosen by the user in the database using the query in line 4.

Algorithm 4 Database Encryption Mechanism Using FF1-AVX2-PIPO

Input: Tweak (T), Length of tweak (t), Radix (r), Key (k), Plaindata (X), and Length of plaindata ($Xlen$)
Output: Database with encrypted data added (*new_DB*)

1: $MySQL * conn = mysql\_init(NULL)$
Set_Connection(*conn*); Connect to a MySQL server
2: MySQL_query(USE `DATABASE NAME`)
3: $Y \leftarrow$ FF1-AVX2-PIPO(T, t, r, k, X, $Xlen$)
4: MySQL_query(INSERT INTO %s VALUES(%s), `DB_TABLE`, Y)
5: **return** *new_DB*

3) PARTIAL IMAGE ENCRYPTION USING FF1-AVX2-PIPO

We propose a method to encrypt the detected object in the entire image after object detection through deep learning. Figure 6 shows the system configuration. We use Yolov3 (an object detection model) to detect a region where a specific object is located in the entire image. The detected area is represented by the (x, y) coordinates of the top-left and bottom-right corners. We use these coordinates to get pixels inside a particular area ($(left\_x < row)$ && $(row < right\_x)$ && $(left\_y < col)$ && $(col < right\_y)$). Because this area has a different size depending on the target image or model used, only pixels within the area are encrypted using

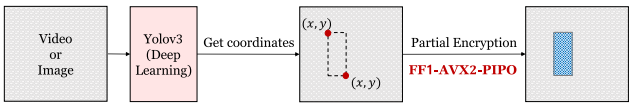


FIGURE 6. Partial encryption using FF1-AVX2-PIPO and object detection.

the FF1-AVX2-PIPO implementation. Therefore, a padding process is not required, and only a specific area can be encrypted. In addition, it is possible to efficiently preserve the privacy of image data due to the use of the PIPO algorithm, which is robust against side-channel attacks, and the fast encryption speed achieved through parallel implementation.

IV. EVALUATION

We experimented with parallel Implementation of PIPO using AVX2, format preserving encryption with parallel PIPO implementation and several applications. Because each experimental environment is different, we refer to the experimental environment in each subsection. Also, the source code is open to our github https://github.com/khj1594012/AVX2-PIPO.

A. PARALLEL IMPLEMENTATION OF PIPO USING AVX2

For the performance evaluation, we measured the performance of the parallel processing of 32 plaintext blocks (32 PT blocks), 16 plaintext blocks (16 PT blocks), and the reference code. This experiment was performed on a MacBook Pro @2.6GHz 6 core Intel Core i7 (16GB RAM), and the optimization options -O1 and -mavx2 were applied. As a performance comparison method, the execution time and cycles to process 1 byte (Cycle Per Byte, cpb) were used.

1) IMPLEMENTATION

We implemented the PIPO algorithm with AVX2 instructions in a parallel way, and we used test vectors from the reference code to ensure that this design was implemented, properly. When the same test vector was applied to 16 plaintexts, the output of the last round of the existing PIPO and the output of the implementation were generated identically.

2) EVALUATION

Table 2 shows a comparison of the execution time results according to the number of plaintext blocks. In the case of the reference code, it started to slow down compared to the AVX2 implementation from more than 64 blocks of plaintext. When fewer than 64 plaintext blocks were processed, it was confirmed that the parallel implementation was rather slow due to pre-processing for the parallel operation. As seen in III-A1, the pre-processing is a process of arranging several 64-bit plaintexts for parallel processing. The *n*th bytes (e.g. 1-th byte, 2-th byte) of each plaintext are collected. Next, they are arranged in a form for input to 256-bit AVX2 register. This pre-processing is an additional process that is not necessary if parallel processing is not performed.

Because this pre-process is required, the performance of the AVX2 parallel implementation improved as the number of plaintexts increased. In the case of 32 PT blocks that process more plaintext but require unnecessary pre-processing in the R-Layer, compared to the reference code, performance improvement of 1.003× for 64 plaintexts and 1.801× for processing 32 million plaintexts was achieved. Therefore, 16 PT block implementation showed the best performance. Compared to the reference code, 16 PT block implementation achieved 7.345× the speed for 32 million plaintext blocks. In other words, when processing 32 PT blocks at the same time, pre-processing is required to use the AVX2 instruction in the R-layer (Because there is no instruction 8-bit shift for 8-bit rotation). However, if 16 PT blocks are processed in parallel, rotation using AVX instruction is possible without pre-processing, so speed optimization is possible. In addition, for parallel processing, plaintext arrangement process is required before encryption. In addition, as the number of plaintexts to be processed increases, the time required for the plaintext arrangement process before encryption is reduced, and thus the speed can be improved.

Table 3 compares the CPBs of our implementation (32 PT blocks and 16 PT blocks) and the reference implementation. In the case of 32 PT blocks, there was a performance improvement of 1.8× compared to the reference code. The performance of 16 PT blocks was also the best, a performance improvement of about 7.34× compared to the reference implementation was achieved. Comparing our two implementations, the cpb of the 16 PT block implementation was 4.07× higher than that of the 32 PT block. And, as a result of comparing CPBs for several plaintext numbers (in Table 2), it was found that all of them were almost same.

TABLE 2. Comparison result in terms of speed (unit: ms) between reference code and AVX2-PIPO (`32 PT blocks`/`16 PT blocks`) where the optimization option is–O1.

The number of PT blocks	Ref. [2]	32 PT blocks	16 PT blocks
32	0.501	0.505	0.502
64	0.511	0.509	0.503
320,000	54.647	32.876	9.003
3,200,000	451.136	260.624	73.538
32,000,000	4390.659	2437.528	597.735

TABLE 3. Comparison result in terms of speed (unit: cpb) between related works and AVX2-PIPO (32 PT/16 PT).

Platform	Implementation	cpb
Intel Core i7	Single PT (Ref. code [2])	44.592
AVX2	32 PT blocks (This work)	24.75
AVX2	16 PT blocks (This work)	6.07
RISC-V	Speed opt. (Single) [13]	1918
RISC-V	Speed opt. (Parallel) [13]	59.93
RISC-V	Memory opt. (Single) [13]	2715
RISC-V	Memory opt. (Parallel) [13]	84.85
ARM	Single PT (Ref. code [2])	34.6
ARM	8 PT [15]	12.0
ARM	16 PT [15]	6.3

Also, Table 3 shows a comparison with PIPO implementations implemented in parallel on different platforms. In [13], RV321 model supporting 32-bit was used, and in [15], 64-bit ARM was used. As a result of benchmarking the existing single plaintext encryption reference [2] on each platform, it was measured to be about 44 cpb on our platform. And, it was measured to 1918 for speed optimization on RISC-V, 2715 for memory optimization on RISC-V, and 34.6 on 64-bit ARM. An exact comparison is difficult because of the different platforms and different implementations. Nevertheless, comparing the performance improvement, when using AVX2, 32PT and 16PT were faster than encryption for single plaintext by 1.8× and 7.34× respectively. Next, in RISC-V, speed optimization became 32× faster than single encryption, and in the case of memory optimization, it was 31.9× faster. Finally, in 64-bit ARM, 8PT and 16PT achieved speed improvements of 2.9× and 5.5×, respectively, compared to single plaintext encryption. Comparing only the performance improvement rate, RISC-V improved the most and ARM showed the least performance improvement.

B. FORMAT PRESERVING ENCRYPTION WITH PARALLEL PIPO IMPLEMENTATION

1) IMPLEMENTATION

In this implementation, key, tweak, radix, and plaintext were used as input, and we used the test vector of FF1. A short plaintext, a long plaintext, a plaintext composed of numbers, and a plaintext that is a mixture characters and numbers can be encrypted while maintaining length and domain.

2) EVALUATION

This experiment was performed on an Ubuntu 20.04.2 LTS virtual machine with Intel i5-8250U CPU @2.50GHz and 2GB RAM. Table 4 shows a comparison of the speed result obtained for the existing FF1 and FF1-AVX2-PIPO. When a 16-byte plaintext was encrypted without repetition, the two algorithms performed similarly, and when the encryption was repeated 100,000 times, FF1-AES shows faster performance than the FF1-AVX2-PIPO. AVX2 implementation is inefficient because it encrypts in units of 128 bytes. When a 128-byte plaintext was encrypted 100,000 times, it took almost the same amounts of time for the existing method and proposed method. As seen in Table 2, the performance of AVX2-PIPO is better than that of the reference code when more plaintext is encrypted. In this experiment, the performance of FF1-AVX2-PIPO showed improvement when 128-byte plaintext was performed, repeatedly. In addition, it was confirmed that it is rather inefficient to repeatedly encrypt 16-byte plaintext using FF1-AVX2-PIPO. It showed similar performance to that of FF1-AES when 128-byte plaintext was input.

C. APPLICATION

1) IMAGE ENCRYPTION USING AVX2-PIPO-CTR

We performed image encryption using AVX2-PIPO-CTR mode. As described in Section III-C, multiple pixels can be encrypted at the same time. Figure 7 shows the result of image encryption with AVX2-PIPO CTR and ECB modes. In the ECB mode, the upper pixels of the image have the same ciphertext value for several rows. Because the upper part of the image has the same pixel value, the ECB mode outputs the same ciphertext. However, in the CTR mode, the same input was encrypted with a completely different value due to the characteristics of the CTR operation mode. Because the CTR mode does not have ciphertext feedback, parallel implementation is easy and high-speed encryption is possible. It has a structure that converts a block cipher into a stream cipher, and this can be utilized for the encryption of videos which are composed of a continuous series of frames.

TABLE 4. Comparison result in terms of speed (unit: ms) between existing FF1-AES and FF1-AVX2-PIPO (`16 PT blocks`) where the optimization option is–O2.

Length of Plaintext	Existing FF1-AES [1].	FF1-AVX2-PIPO (16 PT blocks)
16-byte (1 time)	0.715	0.549
128-byte (1 time)	0.790	0.687
16-byte (100,000 times)	2600.053	4147.026
128-byte (100,000 times)	13236.574	14071.321

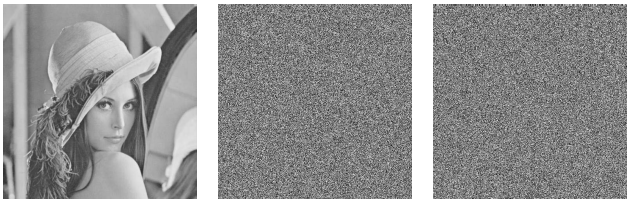


FIGURE 7. Result of image encryption with AVX2-PIPO CTR mode (center) and ECB mode (right).

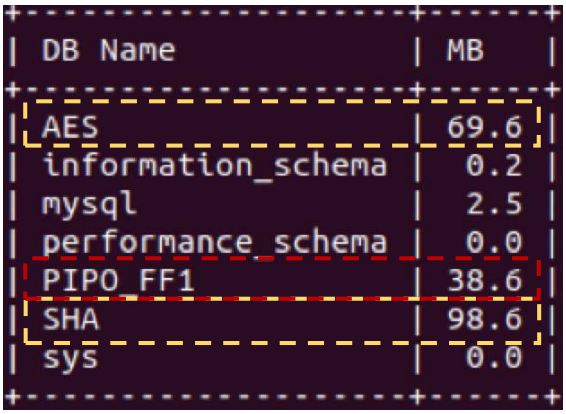
TABLE 5. Comparison result table in terms of speed (unit: ms) between AES, SHA2 of MySQL and FF1-AVX2-PIPO.

Method	$A1234$	$0 \sim 9$ $0 \sim 9$ $0 \sim 9$ $A \sim H$
AES of MySQL	0.755	0.838
SHA2 of MySQL	0.818	0.874
FF1-AVX2-PIPO	0.592	0.852

2) DATABASE ENCRYPTION USING FF1-AVX2-PIPO

This section compares and analyzes database encryption using MySQL's built-in algorithm and database encryption using the FF1-AVX2-PIPO implementation. The experiment was performed on an Ubuntu 16.04 LTS virtual machine with Intel i5-8250U CPU @1.60GHz and 2GB RAM due to the MySQL environment issues. In addition, we selected data storage speed and memory usage after encryption as performance comparison factors.

MySQL provides encryption algorithms, such as AES, SHA2, and DES. We encrypted data stored in our database using FF1-AVX2-PIPO. Table 5 shows the result of



DB Name	MB
AES	69.6
information_schema	0.2
mysql	2.5
performance_schema	0.0
PIPO_FF1	38.6
SHA	98.6
sys	0.0

FIGURE 8. Memory usage for each database (AES, SHA2, and FF1-AVX2-PIPO).

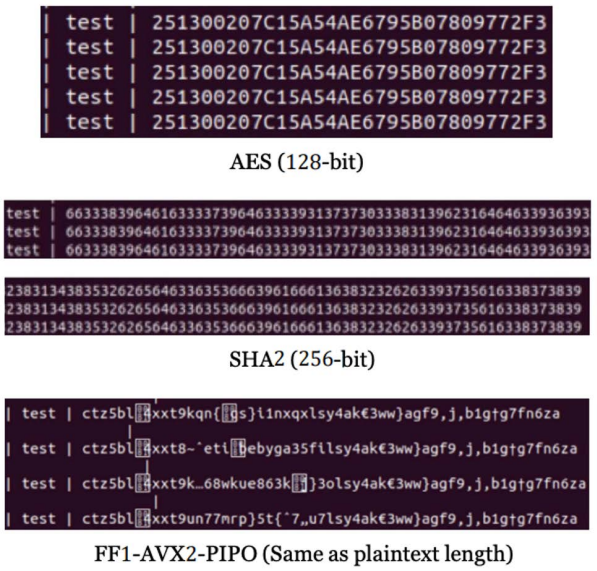


FIGURE 9. Data stored in each database (AES, SHA2, and FF1-AVX2-PIPO).

FIGURE 10. Result of partial encryption (right) with FF1-AVX2-PIPO and object detection (left).

measuring the time it takes to save data to the database after encrypting it using MySQL's AES and SHA2 as well as the proposed implementation of FF1-AVX2-PIPO. It is the time taken to perform 100,000 encryption divided by 100,000 (i.e. average timing). That is the time taken to store single data. For the experiment, we set the data to be encrypted as short plain text and long plain text. As seen in Table 5, in the case of AES and SHA, the block length is adjusted with the padding. This takes a similar amount of time for short plaintext and long plaintext. Because the length of input and output is preserved in the case of FF1-AVX2-PIPO, for the case of short plaintext, the process of encryption and storage in the database was performed in a shorter time than usual. When the long plaintext was used, AES based encryption showed the fastest speed, followed by FF1-AVX2-PIPO, and SHA2 was the slowest. Because plaintext is 256-bits for SHA2, it seems to take the longest. In summary, for short plaintext, FF1-AVX2-PIPO was 0.163ms and 0.226ms faster than AES and SHA2, respectively. For long plaintext, AES was measured to be 0.014ms and 0.036ms faster, respectively, compared to FF1-AVX2-PIPO and SHA2. Combining the results for two lengths of plaintext, AES took 0.7965ms, SHA took 0.846 ms, and FF1-AVX2-PIPO took 0.722ms.

In addition, the memory space occupied by each database is also an important factor. Figure 8 shows the memory usage after three iterations of storing each data in Table 5 with encryption 100,000 times. That is, a total of 600,000 results of encrypting the data are stored in each database. The stored data in the table of a database is shown in Figure 9. It can be seen that when block ciphers are used, the data is converted to a fixed-length (block size) and then stored. SHA, AES, and FF1-AVX2-PIPO took up more memory capacity in that order. For SHA in MySQL, the block length is 256-bit. That of AES is 128-bit. If the length of the plaintext is smaller than the block size, padding is performed to fit the block size. Thus, memory space is wasted. However, in the case of FF1-AVX2-PIPO, because the input and output length are same, the wasted part generated by padding to fit the block size can be eliminated. The proposed method achieves 1.8× less memory usage compared to AES and 2.55× less than SHA2.

Besides, reducing the need to modify the database system is important in database management. In the case of a database table, it consists of rows and columns, and the domains of data stored in the same column are the same. For example, a column that stores names stores strings, and a column that stores ages stores numeric data. Thus, it is possible to preserve the entire schema of the database by storing data that maintains its format even after being encrypted.

In conclusion, the results obtained by applying FF1-AVX2-PIPO can be summarized as follows. It can reduce memory usage and process multiple data in a parallel way. In addition, the encryption speed was improved for the short plaintext, and the encryption speed achieved for the long plaintext was similar to that achieved by MySQL's method. Considering that short data such as passwords are mainly contained in actual databases, the necessity of encryption for large databases, and the need to perform encryption on various types of data stored in databases, using FF1-AVX2-PIPO is considered to be efficient.

3) PARTIAL IMAGE ENCRYPTION USING FF1-AVX2-PIPO

We used a deep learning-based object detection algorithm that detects objects, such as people and cars. The area is displayed as a bounding box as shown on the left of Figure 10, and the coordinates of the upper left and the lower right can be obtained. After the area to be encrypted was determined through these coordinates, only that part was successfully encrypted, as shown in the right part of Figure 10. Privacy can be protected in the process of video analysis based on deep learning, and it can prevent waste of storage space as well as inability to recognize images without decoding.

V. CONCLUSION

In this paper, we propose parallel implementation of PIPO, format preserving encryption with PIPO, and its applications. To achieve high-performance, we utilized an AVX2 based parallel implementation of the PIPO block cipher. Afterward, the implementation was applied to the FF1 algorithm. Finally, we presented several case studies based on PIPO and FF1 implementations. We will further explore the optimized implementation on other platforms and services in future works.

REFERENCES

[1] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, Jul. 2015.
[2] H. Kim, "PIPO: A lightweight block cipher with efficient higher-order masking software implementations," in *Proc. Int. Conf. Inf. Secur. Cryptol.* Cham, Switzerland: Springer, 2020, pp. 99–122.
[3] W. Muła, N. Kurz, and D. Lemire, "Faster population counts using AVX2 instructions," *Comput. J.*, vol. 61, no. 1, pp. 111–120, Jan. 2018.
[4] A. Faz-Hernández and J. López, "Fast implementation of Curve25519 using AVX2," in *Proc. Int. Conf. Cryptol. Inf. Secur. Latin Amer.* Cham, Switzerland: Springer, 2015, pp. 329–345.
[5] M. Dworkin, "Recommendation for block cipher modes of operation: Methods for format-preserving encryption," *NIST Special Publication*, vol. 800, p. 38G, Mar. 2016.
[6] M. Bellare, P. Rogaway, and T. Spies, "The FFX mode of operation for format-preserving encryption," *NIST Submission*, vol. 20, no. 19, p. 24, 2010.
[7] J. Daemen and V. Rijmen, "AES proposal: Rijndael," Tech. Rep., 1999.
[8] W. Stallings, "Format-preserving encryption: Overview and NIST specification," *Cryptologia*, vol. 41, no. 2, pp. 137–152, Mar. 2017.
[9] W. Jang and S.-Y. Lee, "A format-preserving encryption FF1, FF3–1 using lightweight block ciphers LEA and, SPECK," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, Mar. 2020, pp. 369–375.
[10] C. Park, S. Jeong, D. Hong, and C. Seo, "Optimal implementation of format preserving encryption algorithm FEA in various environments," *J. Korea Inst. Inf. Secur. Cryptol.*, vol. 28, no. 1, pp. 41–51, 2018.
[11] H. Kim, M. Sim, K. Jang, H. Kwon, S. Uhm, and H. Seo, "Masked implementation of format preserving encryption on low-end AVR microcontrollers and high-end ARM processors," *Mathematics*, vol. 9, no. 11, p. 1294, Jun. 2021.
[12] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-V instruction set manual: User-level ISA, version 2.0," California Univ. Berkeley Dept. Elect. Eng. Comput. Sci., Long Beach, CA, USA, Tech. Rep., 2014, vol. 1.
[13] S.-W. Eum, K.-B. Jang, G.-J. Song, M.-W. Lee, and H.-J. Seo, "Optimized parallel implementation of lightweight blockcipher PIPO on 32-bit RISC-V," in *Proc. Korea Inf. Process. Soc. Conf.*, 2021, pp. 201–204.
[14] Y. Kwak, Y. Kim, and S. C. Seo, "Parallel implementation of PIPO block cipher on 32-bit RISC-V processor," in *Proc. Int. Conf. Inf. Secur. Appl.* Cham, Switzerland: Springer, 2021, pp. 183–193.
[15] S. Eum, H. Kwon, H. Kim, K. Jang, H. Kim, J. Park, G. Song, M. Sim, and H. Seo, "Optimized implementation of block cipher PIPO in parallel-way on 64-bit ARM processors," *KIPS Trans. Comput. Commun. Syst.*, vol. 10, no. 8, pp. 223–230, 2021.

HYUNJI KIM received the B.S. and M.S. degrees in IT convergence engineering from Hansung University, where she is currently pursuing the Ph.D. dgree. Her research interests include artificial intelligence, machine learning, and information security.

HYUNJUN KIM received the B.S. and M.S. degrees in IT convergence engineering from Hansung University, where he is currently pursuing the Ph.D. degree. His research interests include side-channel analysis and cryptography implementation.

SIWOO EUM received the B.S. degree in IT convergence engineering from Hansung University, where he is currently pursuing the M.S. degree. His research interests include cryptography implementation and information security.

HYEOKDONG KWON received the B.S. and M.S. degrees in IT convergence engineering from Hansung University, where he is currently pursuing the Ph.D. degree. His research interests include cryptography implementation, information security, and machine learning.

YUJIN YANG received the B.S. degree in IT convergence engineering from Hansung University, where, she is currently pursuing the M.S. degree. Her research interests include cryptography implementation and information security.

HWAJEONG SEO received the B.S.E.E., M.S., and Ph.D. degrees in computer engineering from Pusan National University. He is currently an Assistant Professor with Hansung University. His research interests include the Internet of Things and information security.