

Parallel Implementation of PIPO Block Cipher on 32-bit RISC-V Processor*

YuJin Kwak¹, YoungBeom Kim², and Seog Chung Seo^{2*}

¹Department of Information Security, Cryptology, and Mathematics,
Kookmin University

²Department of Financial Information Security, Kookmin University
{youjk, darania, scseo*}@kookmin.ac.kr

Abstract. In ICISC 2020, 64-bit PIPO lightweight block cipher was presented. The main design goals of PIPO are the implementation-friendly and the ease of designing side-channel protection techniques. Until now, the performance of PIPO has been investigated only on 8-bit AVR environment. Thus, optimization strategies on various embedded devices need to be investigated for PIPO's wide usage in various IoT applications. For filling this gap, in this paper, we present an optimized software implementation of PIPO cipher on 32-bit RISC-V processor being widely considered as an alternative embedded processor for ARM processors. For optimal performance, we propose several novel techniques: optimal register scheduling for minimizing the number of memory accesses, data parallel processing approach by using 32-bit register set, and combined internal process. In result, our software shows 128.5% improved performance on PIPO-64/128 basis than the simple ported version of RISC-V. In addition, our implementation showed 393.52% improvement over the encryption performance of the PIPO reference code, despite including the key scheduling process. As far as we know, this is the first optimal implementation of PIPO block cipher on RISC-V environment.

Keywords: PIPO · Optimization · 32-bit RISC-V · Embedded · Efficient Implementation · Parallel Implementation

1 Introduction

In ICISC 2020, a 64-bit lightweight PIPO block cipher was proposed for secure communication in IoT environment [1]. PIPO block cipher consists of an S-Layer designed with bit slicing and an R-Layer which is a permutation process. It has the advantage of providing security with fewer rounds compared to other

* This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.A2021-0270, 6G autonomous security internalization-based technology research to ensure security quality at all times, 50%) and the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. 2019R1F1A1058494, 50%)

lightweight ciphers [2–6]. In addition, PIPO block cipher can be implemented for countermeasures against side-channel attacks with relatively low computational cost. Low-end devices are particularly vulnerable to side-channel implementation [7–10]. Therefore, PIPO block cipher has great advantages from an optimization point of view. In particular, PIPO block cipher is designed specifically for the 8-bit AVR environment so that it achieves a small code size and high performance compared to the existing lightweight ciphers [1, 2].

RISC-V is a open source architecture developed at UC Berkeley. RISC-V have advantages that users can freely design and apply it through the open modular instruction set and hardware structure. Due to these characteristics, many companies are currently developing RISC-V architecture. In addition, as interest in RISC-V architecture as a sensor node increases, research on the application of various cryptographic systems and protocol demonstration in RISC-V architecture is being actively conducted [3, 11–14].

In this paper, we present the first optimized implementation of PIPO block cipher in RISC-V architecture. The various optimization methods are introduced by analyzing RISC-V architecture and overall structure of PIPO block cipher.

Organization Section 2 describes PIPO block cipher and RISC-V Architecture; also, we describe some related block cipher implementations on RISC-V environment. In Section 3, we present a new optimization methods for PIPO block cipher on RISC-V. Section 4 evaluates our software with respect to speed and memory consumption. Finally, Section 5 concludes this paper with future works.

2 Preliminaries

2.1 Overview of 64-bit PIPO Block cipher

PIPO (Plug-In and Plug-Out) is Korea block cipher presented at ICISC 2020, an international major security conference [1]. PIPO provides reference codes implemented as a Bit-slicing method and Table Look Up method to be applicable to various platforms or programming languages. In particular, it boasts excellent performance in the implementation of 8-bit AVR platform. In addition, it is expected to be highly portable to multiple platforms such as big data, cloud, database, and blockchain by proving the security of side-channel analysis.

PIPO performs encryption and decryption in units of 64-bit blocks, and the number of rounds varies according to the size of the key. One round of PIPO's encryption process consists of S-Layer, R-Layer, and AddRoundKey. Figure 1 schematically illustrates the overall structure of PIPO encryption algorithm. S-Layer is a non-linear layer that transforms using an 8-bit S-box. PIPO's Sbox is designed using an unbalanced bridge structure and provides an efficient bit-slicing implementation method in 8-bit Sbox. R-Layer is a linear layer process that rotates left by n -bit for each byte sequentially from the least significant

byte. Rotate left by 0, 7, 4, 3, 6, 5, 1, 2-bit in order from the least significant byte.

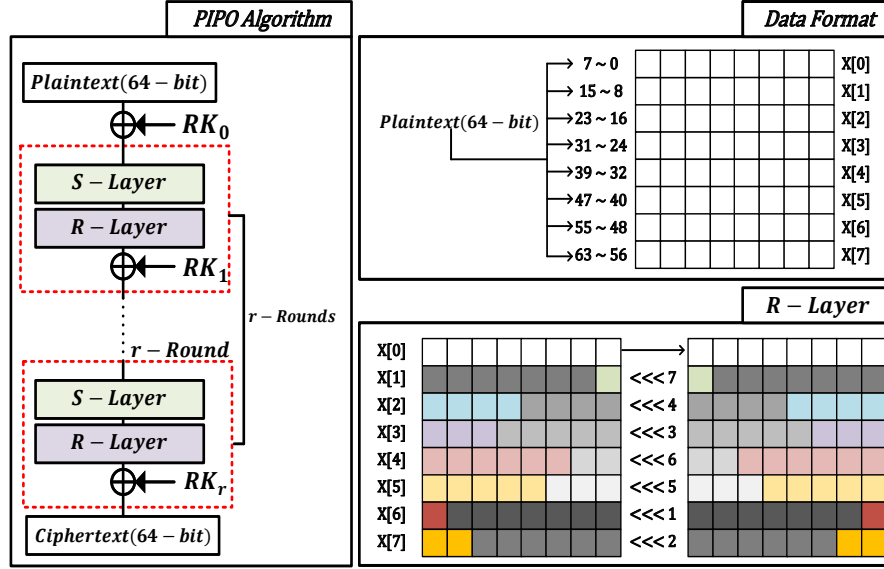


Fig. 1: PIPO Algorithm [1]

$$K^{128} = K_1 || K_0, RK_i = K_{i \bmod 2} (K_i = 64 \text{ bits}, i = 0 \sim 13) \quad (1)$$

$$K^{256} = K_3 || K_2 || K_1 || K_0, RK_i = K_{i \bmod 4} (K_i = 64 \text{ bits}, i = 0 \sim 15) \quad (2)$$

Key schedule process performed before encryption process uses 2 or 4 master keys, respectively, depending on the key size according to the even and odd rounds. To generate a round key, XOR(eXclusive OR) master key and Rcon. Equation 1 is PIPO-64/128 key schedule process, and Equation 2 is PIPO-64/256 key schedule process.

2.2 RISC-V Architecture

RISC-V is a new computer architecture based on RISC(Reduced Instruction Set Computer) that has been under development at UC Berkeley since 2010 [15]. RISC-V supports a free, open instruction set architecture based on open standards collaboration. In addition, since the hardware structure and ISA(Instruction Set Architecture) of RISC-V are freely available, it is possible to extend without restriction through the ISA configured in a modular way, and the freedom of hardware and software development is guaranteed [16]. Because of these advantages, many companies are actively investing in RISC-V development. Interest in RISC-V in the IoT environment has also been increasing by these aspects.

Table 1: 32-bit RISC-V Assembly Instructions (*cc* : clock cycle)

Asm	Operands	Description	Operation	cc
LB	Rd, Ra(+Imm12)	Load bytes	$Rd \leftarrow Ra(+Imm12)$	2
SB	Rd, Ra(+Imm12)	Store bytes	$Ra(+Imm12) \leftarrow Rd$	2
ADD	Rd, Rs1, Rs2	Add	$Rd \leftarrow Rs1 + Rs2$	1
AND	Rd, Rs1, Rs2	And	$Rd \leftarrow Rs1 \& Rs2$	1
XOR	Rd, Rs1, Rs2	eXclusive OR	$Rd \leftarrow Rs1 \wedge Rs2$	1
OR	Rd, Rs1, Rs2	OR	$Rd \leftarrow Rs1 \mid Rs2$	1
SLL	Rd, Rs1, Rs2	Shift left logical	$Rd \leftarrow Rs1 \ll Rs2$	1
SRL	Rd, Rs1, Rs2	Shift right logical	$Rd \leftarrow Rs1 \gg Rs2$	1
BNE	Rs1, Rs2, Ra(+Imm12)	Branch not equal	if($Rs1 \neq Rs2$) $PC \leftarrow PC + Ra(+Imm12 \ll 1)$	1/2

RISC-V consists of 32 registers. Register size is supported in 32-bit and 64-bit with the same instruction set according to the RV32I and RV64I models. In this paper, the RV32I model that supports 32-bit registers is used. A characteristic of RISC-V is that the integer ISA is like other RISC processors, but there is no delay slot for branches, and it supports instructions of variable length. Also, unlike other embedded devices, RISC-V does not have a status code for the operation result [17]. Therefore, there is a characteristic that a conditional jump instruction exists for this purpose. Table 1 specifies the clock cycle and execution method for RISC-V instructions.

RISC-V uses FreedomStudio, an optimized development environment [18]. FreedomStudio is an integrated development environment that targets SiFive-based processors and can be used to write and debug software. FreedomStudio is based on Eclipse and is bundled with a pre-built RISC-V GCC Toolchain, OpenOCD, and freedom e-sdk.

2.3 Block Cipher Implementations on RISC-V

There are several implementations of crypto systems on RISC-V environment. [3] proposed an optimization method on RISC-V using AES-based Table and Bit-slicing. By using T-table, AES was efficiently implemented. However, since T-table-based AES implementation is not secure against side-channel attacks such as timing attack, an optimization method using the Bit-slicing technique was also proposed. in [3] AES-128 based on T-table achieves 57 CPB.

In addition, [3] presented some optimal techniques of ChaCha20 and Keccak-f[1600] on RISC-V. Every round in ChaCha20 includes 4 quarter rounds and every quarter round consists of 4 extras, 4 XORs, and 4 turns. Since RISC-V's basic ISA does not have a rotation instruction, every rotation instruction must be replaced with two shift instructions and an OR instruction; therefore ChaCha uses 20 single cycle commands to comprise the quarter round. By the optimal method, the Implementation of ChaCha20 achieves 27.9 CPB on RISC-V platforms. Keccak-f[1600] is implemented using efficient register scheduling

and bit interleaving. [3] also uses Lane Complementing's method. Through this technique, Keccak-f achieves 68.9 CPB.

[14] presented the benchmark results of the Korea block cipher algorithm on RISC-V. [14] provided the performance of ARIA, LEA, and PIPO encryption results for one block. As a benchmarking technique of RISC-V, [14] are benchmarking using efficient register scheduling and minimize memory access to plain text. By using this, in RISC-V, ARIA achieved a performance of 295.0 CPB, LEA achieved a performance of 48.4 CPB, and PIPO achieved a performance of 259.8 CPB.

3 Proposed PIPO Implementation on RISC-V Platform

In this section, we present a parallel optimal implementation of the first PIPO in a RISC-V processor. Section 3.1 overall introduces our parallel strategy, and the remaining subsections introduce detailed implementation details.

3.1 Overview of Optimization Strategy

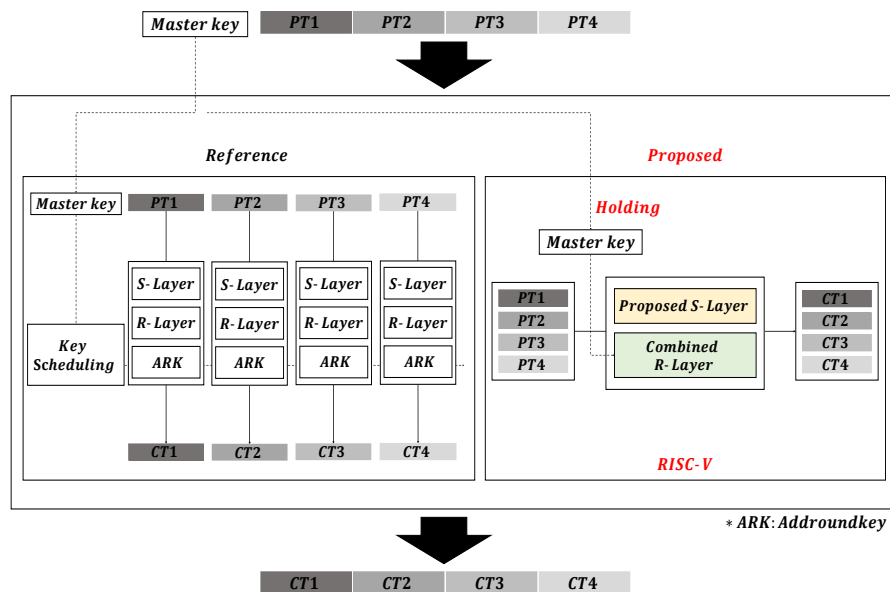


Fig. 2: Overview of Optimization Strategy

Figure 2 shows the overview of our optimization strategy. We propose three optimization strategies for efficient PIPO execution on 32-bit RISC-V processor. Followings are our strategies for optimizing the performance of PIPO.

- *Processing multiple data blocks in parallel*
Since the size of each register in our target RISC-V processor is 32-bit, it is efficient to encrypt-decrypt 4 plaintexts, simultaneously. By using this, we design register scheduling and parallel logic.
- *Minimizing the number of memory access*
By holding the master key on a register, memory access to the round key is minimized; also we efficiently load 4 plaintexts with minimal instructions. It requires only two memory accesses to load one plaintext and loads 32-bit data at a time. Therefore our implementation can load all 4 plaintexts in just 8 instructions.
- *Combining internal computation processes*
we integrate the AddRoundKey process with the existing R-Layer, using a property that the simplicity of PIPO's AddRoundKey operation and the master key held in the register. therefore, only very simple XOR operations are added to the R-Layer.

3.2 Proposed Parallel Logic

Our main idea is the minimization of memory access in encryption process. Compared to the register-only operation on the ALU, the memory access requires more clock cycles. Thus, encryption should be implemented only with ALU operation through register holding of plaintexts used for encryption to reduce memory access.

PIPO is designed for implementation in 8-bit AVRs. So it would be inefficient to only encrypt one block of PIPO in RISC-V, which provides 32-bit registers. Therefore, we will ultimately implement PIPO to match the register features provided by RISC-V. Because 32-bit registers are used, it is more efficient in terms of memory access if multiple blocks are encrypted in parallel than encrypting only one block. Therefore, the register scheduling method proposed in this paper is shown in the following Figure 3.

After the transpose process, 4 plaintexts can have registers scheduled in parallel logic as shown in Figure 3. In this way, using the transpose process, 4 plaintexts are scheduled and encrypted using a total of 8 registers in parallel. It rearranges the ciphertext scheduled in parallel logic so that a normal ciphertext can come out after all operations are completed. If the plaintext is scheduled using the proposed method, it will provide efficient memory access and operation.

3.3 Optimization for S-Layer

In Reference, the implementation of S-Layer is a single byte as a starting point. However, our implementation follows parallel logic, so there are considerations. The first is an independent computation of 4 plaintexts in one register. Since the S-Layer process is based on bit operation, each plaintext is independently

	<i>Plaintext1</i>			<i>Plaintext2</i>			<i>Plaintext3</i>			<i>Plaintext4</i>		
R_0	pt_1^0	...	pt_1^7	pt_2^0	...	pt_2^7	pt_3^0	...	pt_3^7	pt_4^0	...	pt_4^7
R_1	pt_1^8	...	pt_1^{15}	pt_2^8	...	pt_2^{15}	pt_3^8	...	pt_3^{15}	pt_4^8	...	pt_4^{15}
R_2	pt_1^{16}	...	pt_1^{23}	pt_2^{16}	...	pt_2^{23}	pt_3^{16}	...	pt_3^{23}	pt_4^{16}	...	pt_4^{23}
R_3	pt_1^{24}	...	pt_1^{31}	pt_2^{24}	...	pt_2^{31}	pt_3^{24}	...	pt_3^{31}	pt_4^{24}	...	pt_4^{31}
R_4	pt_1^{32}	...	pt_1^{39}	pt_2^{32}	...	pt_2^{39}	pt_3^{32}	...	pt_3^{39}	pt_4^{32}	...	pt_4^{39}
R_5	pt_1^{40}	...	pt_1^{47}	pt_2^{40}	...	pt_2^{47}	pt_3^{40}	...	pt_3^{47}	pt_4^{40}	...	pt_4^{47}
R_6	pt_1^{48}	...	pt_1^{55}	pt_2^{48}	...	pt_2^{55}	pt_3^{48}	...	pt_3^{55}	pt_4^{48}	...	pt_4^{55}
R_7	pt_1^{56}	...	pt_1^{63}	pt_2^{56}	...	pt_2^{63}	pt_3^{56}	...	pt_3^{63}	pt_4^{56}	...	pt_4^{63}

Fig. 3: Four plaintext scheduling using parallel logic in RISC-V

calculated in our proposed parallel logic. That is, each plaintext has no effect. The second is the use of temporary registers considering operands. In our implementation, the 3 temporary variables in the reference code are replaced with 4 registers. Temporarily 1 register each in S_5^1 , S_5^2 , S_3 constituting S-Layer, and one additional register is used to store the intermediate value. Through this, the S-Layer for 4 plaintexts is calculated at the same time.

3.4 Optimization for R-Layer

R-Layer uses the rotation operator to shuffle the bits of each byte. We have to think about two parts of the R-Layer process. First, there is no rotation operator that handles carry in RISC-V. Therefore, we have to think about the handling of carry when performing the rotation operation. Second, we use parallel logic. Since we are encrypting 4 plaintexts using parallel logic, we need to perform rotation operation in one register in units of bytes. Therefore, in our implementation, we propose a method of constructing the R-Layer by processing the rotation operation using 4 plaintexts scheduled in parallel logic in RISC-V.

When performing the rotation operation, the carry part should be considered. We use a method of masking the data loaded onto each register to handle carry. This method can be useful in register with parallel logic from the 32-bit. Figure 4 are illustrates how to configure a rotation using a bit-masking in RISC-V that we offer.

The main idea is that when right rotation 3-bits in one byte, the actual process that takes place within a byte is that the left 3-bit and right 5-bit are swapped into one chunk. Figure 4 is an example using one register among 8 registers. It has one register which first bytes of the plaintext output after performing the S-Layer process are collected. We first proceed with the masking process, which divides chunks by considering carry. The example shows the process of rotating 3-bits to the right of each of the 4-byte; therefore, we will perform an operation that swaps the right 3-bit chunk and the left 5-bit chunk for each

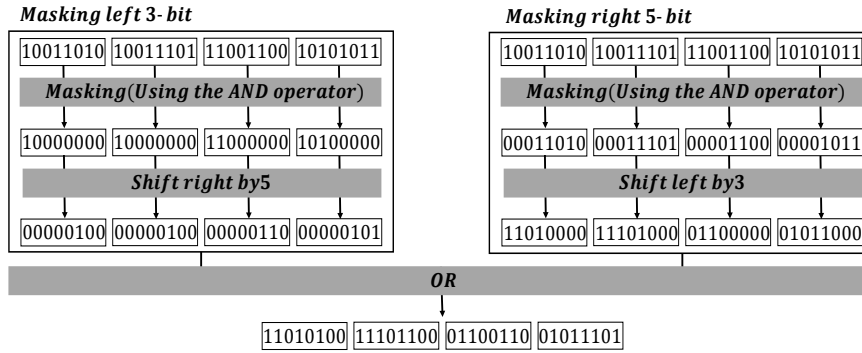


Fig. 4: Rotation operation using masking in RISC-V

byte. First, in order to divide chunks while considering carry, in the masking process, 0xE0 and 0x1F are operated by AND operator for each byte. Then, for 4 bytes, a 3-bit chunk on the left and 5-bit chunk on the right are generated respectively. A shift operation is performed to exchange the chunks of the left 3-bit and the right 5-bit. If the OR operation is performed on the two chunks that have performed the shift operation, the register that has performed the rotation operation for each byte is calculated. In this way, our proposed method has the advantage that it can be extended and applied to 16-bit and 64-bit processors, including 32-bit processors.

3.5 Combined AddRoundKey

To proceed with the encryption process in PIPO, a round key is required for every round. A round key is generated from one master key through a key schedule process in advance. Since this function generates additional computational load independent of the encryption process, we propose a method to optimize the computational amount of key schedule process.

PIPO's key schedule process generates a round key through XOR operation of the master key and Rcon. In RISC-V, PIPO's key schedule function consumes 1146 clock cycles. To reduce the additional load caused by key schedule, the AddRoundKey function and the key schedule process are combined within the encryption process. We have implemented AddRoundKey to proceed with a simple XOR operation by storing the round key in the register. In this case, there is no process of loading the round key, which reduces the number of memory accesses. This operation is integrated into R-Layer. At this time, it should be noted that the master key used for even rounds and odd rounds is different.

4 Performance Analysis

Table 2 compares the performance of the proposed PIPO implementation with the reference PIPO implementation and other block cipher implementations on

32-bit RISC-V processor. To measure the correct performance, we used SiFive’s HiFiveRevB board equipped with 32-bit RISC-V processor. HiFiveRevB board has a 320MHz frequency, 16KB RAM, and 32MB flash memory. As a development tool, we used SiFive’s FreedomStudio (ver. 2020) using GCC Toolchain: gcc version 10.1.0 and gcc version 8.3.0. For fair comparison, we build our software and the reference PIPO implementation with -O3 compile option.

Table 2: Comparison of other block cipher implementations on RISC-V (*CPB* : Cycles Per Byte, code size measured by byte).

Reference	Algorithm	Plaintext	Security	<i>CPB</i>	Code size
This work	PIPO	64-bit	128-bit	113.7*	10816
			256-bit	136.7*	12352
Kim et al.(Ref) [1]			128-bit	561.13	896
			256-bit	600.38	896
Ko Stoffelen [3]	Table-AES	128-bit	128-bit	57.0	4352
	ChaCha20 enc	512-bit		27.9	12960
	Keccak-f[1600]	-		68.9	7936
Kwak et al. [14]	PIPO	64-bit	128-bit	249.1	6124
	ARIA	128-bit		291.0	8192
	LEA			46.4	2432

*Includes key schedule process

For the comparison to the reference PIPO implementation [1] which was originally developed on 8-bit AVR MCU, we build it to a 32-bit RISC-V processor. Regarding the reference code, when performance was measured only for the encryption process, 561.13 CPB for PIPO-64/128 and 600.38 CPB for PIPO-64/256 were measured. Our work proposes a merged key scheduling process. Despite the inclusion of the key scheduling process, our implementation has a performance improvement of 393.52% and 339.19% for PIPO-64/128 and PIPO-64/256, respectively, compared to the implementation built with the reference code.

In Table 2, we also compare our implementation with other ciphers implementations on the same target devices. In case of Korean block ciphers PIPO, ARIA, and LEA, Kwak et al. implemented them on 32-bit RISC-V with some optimization techniques and benchmarked their performance (The core parts were written in assembly codes) [14]. The PIPO Reference implementation is assembled and ported to RISC-V. The implementation proposed in [14] had a performance improvement over the default implementation, the reference implementation built simply on RISC-V, as the ported implementation. The proposed PIPO implementation outperforms Kwak et al.’s ARIA and PIPO implementations by 159.45% and 128.5%, respectively.

However, it still shows lower performance than LEA, which is an ARX structure composed of only simple bit operations. 32-bit addition operation of LEA shows high efficiency in embedded devices with 32-bit registers, including the RISC-V environment. Implementation of the partitioning of LEA in 32-bit units removes the load on carry operations that have to be considered in other devices; therefore, the actual process is only simple XOR and rotate operation. We further investigated the implementation of LEA in the AVR environment. The encryption performance of LEA in the AVR platforms achieved 190 CPB in the initially proposed code [4], and 167 CPB in the additionally optimized paper [19]. PIPO's performance in PIPO's proposal paper is 197 CPB, which is basically lower than LEA [1]. On the other hand, in the AVR processor, which is one of the most constrained environments in terms of code size, PIPO shows the best performance. Algorithmic performance differences are maintained even in the RISC-V environment. However, our implementation achieves lower performance than benchmarked LEA in a RISC-V environment, but is competitive compared to ARIA and benchmarked PIPO. The code size for parallelization uses only 0.03% in the RISC-V environment with the most commonly used 32MB of flash memory. Therefore, from a practical applicability standpoint, the memory cost is negligible. For a higher performance improvement over the proposed implementation, ISA of RISC-V needs to be further designed. Currently, parallel instruction development for RISC-V is in progress. If development is completed in the future and commercialization takes place in earnest, there is a possibility that the proposed PIPO software will be further optimized.

5 Conclusion

With the development of 5G technology, the importance of high-speed of data communication and security in embedded devices is being stressed. However, the embedded environment has limited resources unlike the high-end computer, so it is challenging to apply the reference implementation of the computation-intensive cryptography. Thus, it is necessary to optimize the crypto implementation considering the characteristics of each embedded environment. In this paper, we propose an optimization method for the PIPO block cipher algorithm in the 32-bit processor RISC-V architecture. Our optimization methods include parallel logic implementation, S-Layer optimization method, rotation instruction implementation method considering the characteristics of RISC-V architecture, and R-Layer configuration that merges the key schedule process and AddRoundKey process. Efficient PIPO in RISC-V architecture suggest implementation plan. Our PIPO-64/128 and PIPO-64/256 optimization implementations provide performance improvements of 393.52% and 339.19% over the existing reference code, respectively. In the future, we plan to conduct the optimized implementation of lightweight block ciphers such as SPECK, SIMON, and SKINNY for secure communication in an IoT environment.

References

1. Hangi Kim, Yongjin Jeon, Giyoon Kim, Jongsung Kim, Bo-Yeon Sim, Dong-Guk Han, Hwajeong Seo, Seonggyeom Kim, Seokhie Hong, Jaechul Sung, and Deukjo Hong. Pipos: A lightweight block cipher with efficient higher-order masking software implementations. In Deukjo Hong, editor, *Information Security and Cryptology – ICISC 2020*, pages 99–122, Cham, 2021. Springer International Publishing.
2. Hangi Kim, Yongjin Jeon, Giyoon Kim, Jongsung Kim, Bo-Yeon Sim, Dong-Guk Han, Hwajeong Seo, Seonggyeom Kim, Seokhie Hong, Jaechul Sung, and Deukjo Hong. A new method for designing lightweight s-boxes with high differential and linear branch numbers, and its application. Cryptology ePrint Archive, Report 2020/1582, 2020. <https://eprint.iacr.org/2020/1582>.
3. Ko Stoffelen. Efficient cryptography on the RISC-V architecture. *IACR Cryptol. ePrint Arch.*, 2019:794, 2019.
4. Deukjo Hong, Jung-Keun Lee, Dong-Chan Kim, Daesung Kwon, Kwon Ho Ryu, and Dong-Geon Lee. Lea: A 128-bit block cipher for fast encryption on common processors. In Yongdae Kim, Heejo Lee, and Adrian Perrig, editors, *Information Security Applications*, pages 3–27, Cham, 2014. Springer International Publishing.
5. Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. HIGHT: A new block cipher suitable for low-resource device. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2006.
6. Dongyoung Roh, Bonwook Koo, Younghoon Jung, Il Woong Jeong, Dong-Geon Lee, Daesung Kwon, and Woo-Hwan Kim. Revised version of block cipher cham. In Jae Hong Seo, editor, *Information Security and Cryptology – ICISC 2019*, Cham, 2020. Springer International Publishing.
7. Ilya Kizhvatov. Side channel analysis of AVR XMEGA crypto engine. In Dimitrios N. Serpanos and Wayne H. Wolf, editors, *Proceedings of the 4th Workshop on Embedded Systems Security, WESS 2009, Grenoble, France, October 15, 2009*. ACM, 2009.
8. Anupam Golder, Debayan Das, Josef Danial, Santosh Ghosh, Shreyas Sen, and Arjit Raychowdhury. Practical approaches toward deep-learning-based cross-device power side-channel attack. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2720–2733, 2019.
9. Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. Side channel attack on noc-based mpsoes are practical: Noc prime+probe attack. In *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6, 2016.
10. Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. Keydown: Eliminating keystroke timing side-channel attacks. *CoRR*, abs/1706.06381, 2017.
11. Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoît Viguier. Assembly or optimized C for lightweight cryptography on risc-v? In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings*, volume 12579 of *Lecture Notes in Computer Science*, pages 526–545. Springer, 2020.

12. Ben Marshall, G. Richard Newell, Dan Page, Markku-Juhani O. Saarinen, and Claire Wolf. The design of scalar AES instruction set extensions for RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):109–136, 2021.
13. Etienne Tehrani, Tarik Graba, Abdelmalek Si-Merabet, and Jean-Luc Danger. RISC-V extension for lightweight cryptography. In *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*, pages 222–228. IEEE, 2020.
14. YuJin Kwak, YoungBeom Kim, and Seog Chung Seo. Benchmarking korean block ciphers on 32-bit risc-v processor. *journal of korea institute of information security and cryptology (JKIISC)*, 31(3):331–340, 2021.
15. Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011.
16. Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. PhD thesis, UC Berkeley, 2016.
17. Jongbok Lee. Simulation and synthesis of risc-v processor. *The Journal of The Institute of Internet, Broadcasting and Communication*, 19(1):239–245, 2019.
18. Chat Room. Risc-v architecture. *image*, 2017.
19. Hwajeong Seo, Kyuhwang An, and Hyeokdong Kwon. Compact lea and hight implementations on 8-bit avr and 16-bit msp processors. In Brent ByungHoon Kang and JinSoo Jang, editors, *Information Security Applications*, pages 253–265, Cham, 2019. Springer International Publishing.