symmetry

MDPI

Article

Optimized Implementation of Simpira on Microcontrollers for Secure Massive Learning

Minjoo Sim [1], Siwoo Eum [1], Hyeokdong Kwon [1], Kyungbae Jang [1], Hyunjun Kim [1], Hyunji Kim [1], Gyeongju Song [1], Waikong Lee [2] and Hwajeong Seo [1,*]

[1] Division of IT Convergence Engineering, Hansung University, Seoul 02876, Korea
[2] Department of Computer Engineering, Gachon University, Seongnam 13306, Korea
* Correspondence: hwajeong@hansung.ac.kr; Tel.: +82-760-8033

Abstract: Internet of Things (IoT) technology, in which numerous devices cooperate, has a significant impact on existing industries, such as smart factories, smart cars, and smart cities. Massive learning and computing using data collected through the IoT are also being actively performed in these industries. Therefore, the security of low-end microcontrollers used in the Internet of Things should be highly considered due to their importance. Simpira Permutation is a Permutation design using the AES algorithm designed to run efficiently on 64-bit high-end processors. With the efficient implementation of Simpira algorithm, we can ensure secure massive learning in IoT devices without performance bottleneck. In nature, Simpira exploited the part of AES algorithm. The AES algorithm is the most widely used in the world, and Intel has developed hardware accelerated AES instruction set (AES-NI) to improve the performance of encryption. By using AES-NI modules, Simpira can be improved further on high-end devices. On the other hand, low-end processors do not support AES-NI modules. For this reason, an optimized implementation of efficient Simpira should be considered. In this paper, we present an optimized implementation of Simpira on 8-bit AVR microcontrollers and 32-bit RISC-V processors, which are low-end processors that do not support AES-NI features. There are three new techniques applied. First, Addroundkey is computed efficiently through pre-computation. Second, it takes advantage of the characteristics of round keys to omit some of the operations. Third, we omit unnecessary operations added to use AES-NI features. We have carried out performance evaluations on 8-bit ATmega128 microcontrollers and 32-bit RISC-V processors, which show up-to 5.76× and 37.01× better performance enhancements than the-state-of-art reference C codes for the Simpira, respectively.

Keywords: AES; software implementation; simpira permutation; 8-bit AVR microcontroller; 32-bit RISC-V processor

Citation: Sim, M.; Eum, S.; Kwon, H.; Jang, K.; Kim, H.; Kim, H.; Song, G.; Lee, W.; Seo, H. Optimized Implementation of Simpira on Microcontrollers for Secure Massive Learning. *Symmetry* **2022**, *14*, 2377. https://doi.org/10.3390/sym14112377

Academic Editors: Milan Milosavljević, Takeshi Koshiba, Yuan Ping and Yuri Borissov

Received: 27 September 2022
Accepted: 8 November 2022
Published: 10 November 2022

1. Introduction

With the development of Internet of Things (IoT) technology, low-end processors used in the IoT communicate with each other to provide useful services [1]. For example, autonomous vehicles collect driving state data through various sensors in the vehicle. After analyzing the collected data, real-time decisions can be made [2]. In this way, vast amounts of important data are collected in various industries, such as drones and transportation to perform massive learning and computing [3,4]. The importance of security for low-end processors used in IoT increases day by day. Therefore, research on encryption for efficient operation in low-end processors is being actively conducted.

AES (Advanced Encryption Standard) is an encryption algorithm that was adopted by the National Institute of Standards and Technology (NIST) in 2001 [5]. Since then, AES block cipher has become the most used encryption algorithm in the world. In 2008, Intel developed an instruction set (AES-NI [6]) to improve the performance of AES encryption/decryption as an extension of the x86 instruction set. In addition, the AES instruction set was developed to improve AES performance in ARM processors.

Simpira Permutation proposed an efficient permutation with AES Round function using the AES instruction set [7]. AES algorithm is a symmetric-key algorithm. Therefore, Simpira, which uses the round function of the AES encryption algorithm, is also a symmetric encryption algorithm. Most microcontrollers use symmetric encryption algorithms to provide security for transmitted information [8–10]. Therefore, Simpira block cipher is suitable for use as a cryptographic algorithm (i.e., symmetric encryption) to provide security on microcontrollers. However, the proposed Simpira cannot be used generally, because many kinds of processors do not support the AES instruction set. In this paper, we propose optimized implementations of Simpira on 8-bit AVR microcontrollers and 32-bit RISC-V processors that do not provide the AES instruction set.

The remainder of this paper is structured as follows. Section 2 describes the AES Block Cipher, the Simpira Permutation, the target 8-bit AVR microcontrollers, the target 32-bit RISC-V processors, and related works. Section 3 describes the proposed implementation method. Section 4 shows a performance comparison with the-state-of-art works. Finally, Section 5 describes the conclusion of this work and future plan.

1.1. Contribution

1.1.1. Optimized Simpira on the 8-Bit AVR Architecture

ATmega128 processor is one of Atmel AVR family, which is the most commonly used in practice. We propose an optimized implementation of Simpira on the ATmega128 processor. The Simpira uses AES algorithm, but the target processor does not support the AES-NI instruction set. Since AES-NI is not available, we have used existing AES to enable Simpira to have the same AES-NI instruction set to operate on the target processor. To implement the Simpira, some offset functions are optimized away. We optimized by omitting the final rounds of Mixcolumns and InvMixcolums as they could work oppositely. Furthermore, some Addroundkey functions use the 0x00 round key. Using this, we optimized the Addroundkey operation by omitting it to 4 operations. We have carried out experiments that show up-to 5.76× better performance enhancements than reference C code for the Simpira.

1.1.2. Optimized Simpira on the 32-Bit RISC-V Architecture

RISC-V is open-source computer architecture. We present the optimal implementation of Simpira, whose permutation is implemented with the AES algorithm. However, on the 32-bit RISC-V processor it does not support AES-NI instruction sets. Since we are not using AES-NI, we can omit the Mixcolumns operation in the last round. As a result, Mixcolumns and InvMixcolumns operations are omitted. Furthermore, certain rounds in Simpira use a round key of 0x00. The Addroundkey operation is omitted for those rounds. As a result, we showed that the experiment is up to 37.01× better than the reference C code for Simpira.

1.1.3. First Optimized Implementation for Simpira on 8-Bit AVR Microcontrollers and 32-Bit RISC-V Processors

The implementation on the low-end processor for Simpira, an algorithm used inside SPHINCS+ [11] and an algorithm that advanced to the NIST PQC Round3 [12], has not yet been explored before except for the implementation on the ARM processors [7]. To the best of our knowledge, there are no studies regarding Simpira on the optimized 8-bit AVR microcontrollers and 32-bit RISC-V processors. In this paper, we have optimized Simpira on 8-bit AVR microcontrollers and 32-bit RISC-V processors, for the first time.

2. Related Works

2.1. AES Block Cipher

The AES (Advanced Encryption Standard) is a symmetric block cipher that uses an identical key for encryption and decryption. It is composed of 128-bit blocks, and the number of rounds is 10, 12, and 14 according to the key length of 128-bit, 192-bit, and 256-bit, respectively. In the encryption process, the MixColumns step is performed in all

rounds except the last round, and every round goes through the SubBytes, ShiftRows, and AddRoundKey steps.

Each encryption step proceeds as follows. SubBytes apply the same 8-bit S-Box to each byte of the internal state. ShiftRows shifts the k-th row to the left by k-bytes. MixColumns multiplies each column by a diffusion matrix through $GF(2^8)$. AddRoundKey adds the round key, which is derived from key extension using secret key [5]. The overall operation codes are detailed in Algorithm 1.

Algorithm 1: AES Algorithm.

procedure AESroundkey($state, rk$)

1: $state \leftarrow Addroundkey(state, rk[0])$
2: $R \leftarrow Round - 1$
3: **for** $i = 1$ **to** R **do**
4: $state \leftarrow SubBytes(state)$
5: $state \leftarrow ShiftRows(state)$
6: $state \leftarrow MixColumns(state)$
7: $state \leftarrow Addroundkey(state, rk[R])$
8: **end for**
9: $state \leftarrow SubBytes(state)$
10: $state \leftarrow ShiftRows(state)$
11: $state \leftarrow Addroundkey(state, rk[Round])$
12: **return** $state$

end procedure

2.2. Simpira Permutation

Simpira Permutation uses the AES round functions. If the roundkey used in AddRoundKey in the AES block cipher is set to a publicly known fixed value, it can be used as an encryption permutation with the same output value when the input value is the same. AES encryption spreads all bits to other bytes during 2 rounds. For this reason, one round of Simpira consists of 1 and 2 rounds of AES. To use it as the permutation, a fixed value is used for the roundkey used in AddRoundKey of AES. Therefore, the output value is fixed, because the roundkey is fixed [7].

At this time, it is not safe to set the fixed roundkey value to 0x00. A fixed roundkey is used by utilizing the round constant. The overall algorithm is the same as Algorithm 2. The roundkey Z used in line 5 of the Algorithm 3 represents a roundkey in which all roundkey values are 0x00. That is, a fixed roundkey using a round constant and a round key with 0x00 are used alternately. Simpira block size increases in 128-bit units because it uses the round function of AES. In $b \times 128$-bit, there is a difference in the algorithm depending on the parameter b. In this paper, b is set to 1 where b is a number of blocks, it is used as a standard. Figure 1 shows the structure of Simpira for $b = 1$.

Algorithm 2: Simpira Algorithm.

procedure Simpira($state, rk$)

1: $R \leftarrow 6$

2: **for** $c = 1$ **to** R **do**

3: $\quad state \leftarrow \mathrm{F}_{c,b}(state)$

4: **end for**

5: $state \leftarrow InvMixColumns(state)$

6: **return** $state$

end procedure

Algorithm 3: $\mathrm{F}_{c,b}$ Algorithm ($b = 1$).

procedure $\mathrm{F}_{c,b}(state)$

1: $RK[0] = 0x00 \oplus c \oplus b$

2: $RK[4] = 0x10 \oplus c \oplus b$

3: $RK[8] = 0x20 \oplus c \oplus b$

4: $RK[12] = 0x30 \oplus c \oplus b$

5: **return** $AES(AES(state, RK), Z)$

end procedure

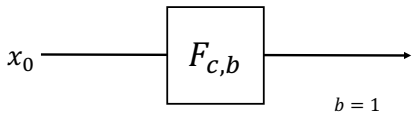

Figure 1. Structure of Simpira about $b = 1$; c is a counter that is initialized by one, and incremented after every use of $F_{c,b}$. Every $F_{c,b}$ consists of two AES round, where the round constants that are determined from (c, b) where b is number of blocks.

2.3. 8-Bit AVR Microcontroller

A low-end 8-bit AVR microcontroller is an 8-bit RISC single chip based on Harvard architecture. Mainly used in low-power environments, there are currently several types of AVR microcontrollers, with various peripherals and memory sizes. In this paper, ATmega128, which is the most widely used in the ATmega class, is used. The ATmega128 can use 133 RISC instructions and has 32 8-bit general-purpose registers. It has 128 KB of flash memory, 4 KB of EEPROM, and 4 KB of SRAM [13]. Instructions used to implement the optimized Simpira are summarized in Table 1.

Table 1. Summarized instructions set of efficient Simpira implementations on 8-bit AVR microcontrollers; Rd: destination register, Rr: source register, X, Y, Z: indirect address register ($X\{R27 : R26\}$, $Y\{R29 : R28\}$ and $Z\{R31 : R30\}$), PC: loaded with the contents of the Z-register, C: carry flag, K: constant data, k: constant address.

Instruction	Operands	Description	Operation	#Clock
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	1
EOR	Rd, Rr	Exclusive OR	Rd ← Rd ⊕ Rr	1
MOV	Rd, Rr	Copy Register	Rd ← Rr	1
MOVW	Rd, Rr, Rr	Copy Register Pair	Rd+1:Rd ← Rr+1:Rr	1
LPM	Rd, Z	Load Program Memory	Rd ← Z	3
BRCC	k	Branch if Carry Cleared	if(C = 0) then PC ← PC + k + 1	1/2
LD	Rd, X(or Y, Z)	Load Indirect	Rd ← X(or Y, Z)	2
LDI	Rd, K	Load Immediate	Rd ← K	1
ST	X(or Y, Z), Rr	Store Indirect	X(or Y, Z) ← Rr	2
PUSH	Rr	Push Register on Stack	STACK ← Rr	2
POP	Rd	Pop Register from Stack	Rd ← STACK	2

2.4. 32-Bit RISC-V Processor

A RISC-V is an open-source developed at UC Berkeley since 2010. Unlike ARM processors, which have the greatest influence, this is a computer CPU structure that can be used for free without paying a license. RISC-V has developed 32-bit, 64-bit, and 128-bit devices. The RISC-V instruction set architecture (ISA) is divided into RV32I, RV64I, and RV128I according to the supported bit size. In this paper, the 32-bit RV32I instruction set is used. A 32-bit RISC-V processor has 32 32-bit registers. The purpose of each register is shown in Table 2. Among them, there are *sp* registers and s0∼s11 registers as callee-saved registers that preserve the value before using the register and return the value after use [14]. The intruction set for RISC-V is given in Table 3.

Table 2. Purpose of registers in 32-bit RISC-V processor.

Register	Description	Saver
zero(x0)	zero register	caller
ra(x1)	return address register	caller
sp(x2)	stack pointer register	callee
gp(x3)	global pointer register	caller
tp(x4)	thread pointer register	caller
a0∼a7	function arguments and return value registers	caller
s0∼s11	saved registers	callee
t0∼t6	temporal registers	caller

Table 3. Summarized instructions set of efficient Simpira implementations on 32-bit RISC-V processors; Rd: destination register, Rs: source register, K: constant data, J: constant address [15].

Instruction	Operands	Description	Operation
ADD	Rd, Rs1, Rs2	Add	Rd ← Rs1 + Rs2
XOR	Rd, Rs1, Rs2	Exclusive OR	Rd ← Rs1 ⊕ Rs2
MV	Rd, Rs1	Copy Register	Rd ← Rs1
SLLI	Rd, Rs1, K	Shift left logical immediate	Rd ← Rs1 « K
SRLI	Rd, Rs1, K	Shift right logical immediate	Rd ← Rs1 » K
BNE	Rs1, Rs2, J	Branch not equal	if(Rs1!=Rs2) Jump to J
JAL	J	Jump and link	Jump to J
LW	Rd, K(J)	Load word	Rd ← J + K
SW	Rs1, K(J)	Store word	Rs1 → J + K
ANDI	Rd, Rs1, K	AND immediate	Rd ← Rs1 & K

2.5. Related Works

There are many kinds of optimization implementations on AVR and RISC-V processors. In the case of Simpira, since it has never been optimally implemented on AVR and RISC-V processors, we introduce an optimized implementation of other block cipher algorithms instead.

Kim et al. [16] suggested FACE-LIGHT that revised version of Fast AES-CTR mode Encryption (FACE [17]). The FACE-LIGHT targeted AES with the 32-bit counter mode of operation in constant timing. It used pre-computation techniques that some variables pre-calculated. Some of the operation are omitted. The FACE-LIGHT targeted AVR microcontrollers, and it showed 22% faster than the previous FACE.

Kwon et al. [18] showed an optimized implementation of the Korean block cipher CHAM [19]. It proposed a pre-computation technique that skipped the first eight rounds of CHAM block cipher. Furthermore, the proposed method was implemented considering the key update and the fixed key situation. Results showed that the proposed implementation had 12.8%, 8.9%, and 9.6% higher performance than previous CHAM implementation for CHAM-64/128, CHAM-128/128, and CHAM-128/256, respectively.

Kim et al. [20] was implemented the Korean block cipher PIPO that announced in 2020 [21]. The proposed implementation targeted AVR microcontrollers, and it had Side-Channel Attack(SCA) countermeasure by 2-byte random masking. In addition, they presented a new kind of masking technique that using OR operation, so to implement the S-layer the presented implementation used 23 AND operations, 5 OR operations, and 47 XOR operations. Evaluation results showed 1.5× faster than reference PIPO implementation without SCA countermeasure. In the case of SCA countermeasure implementation, the proposed implementation had 2.2× faster performance than previous implementations.

Eum et al. [22] optimally implemented the Korean block cipher LEA [23] on RISC-V processors. The proposed implementation applied Counter mode of operation and Galois counter mode of operation. It mainly used pre-computation method and state fixing that makes omitted state move operations. Moreover, it showed applicability to GCM. As a result, the suggested implementation showed 2% performance improvement over previous LEA implementations with counter mode of operation. In the case of GCM, it becomes an indicator for the following researchers.

Kwon et al. [24] targeted the Chinese domestic block cipher SM4 [25], and it was implemented on AVR, and RISC-V processors. It presented an efficient register scheduling plan and optimized implementation for 32-bit wise rotation. Furthermore, different implementations were provided depending on whether to optimize the memory or the operating speed in the case of AVR implementations. The AVR implementations showed evaluation

results that 205.2 cycles per byte, 213.3 cycles per byte, and 207.4 cycles per byte for speed optimization, memory optimization, and code size optimization, respectively. All kinds of implementations showed faster performance than reference implementation which had 1670.7 cycles per byte. The RISC-V implementation had 128.8 cycles per byte, it had better performance than 345.7 cycles per byte of the reference implementation.

3. Proposed Method

3.1. Optimized Implementation of Simpira on 8-Bit AVR Microcontroller

3.1.1. Constant Roundkey Pre-Computation

Since the AES algorithm used in Simpira uses a round constant unlike the original AES extended roundkey, it is possible to calculate the value used as the roundkey in advance. Before entering the AES round function in Algorithm 3, the roundkey is pre-computed and the AES round function operation is performed. Therefore, we fixed the b parameter to 1 because we optimized the implementation of Simpira Permutation where the value of b (the number of blocks) is 1. Roundkeys always use the fixed value b. Therefore, during the operation of the $F_{c,b}$ function, the roundkey can be calculated in advance without having to recalculate the roundkey every round. In other words, the operations performed in lines 1 to 4 of Algorithm 3 (operation of the roundkey) can be omitted.

3.1.2. Omitting AddRoundkey Function

Simpira runs 6 rounds. In this case, two AES round functions are performed in one Simpira round. Among the round functions of AES, the roundkey used in the Addroundkey function uses a constant roundkey once and uses Z (all values of roundkey are 0x00) once. In other words, two roundkeys are used per round and a total of 12 roundkeys. Since one roundkey per round is 0x00, 6 roundkeys are using 0x00 in a total of 6 rounds. The operation of the Addroundkey function consists of the XOR operation of State and roundkey. When XOR operation is performed with roundkey of 0x00 and State, the State value does not change.

The implementation of existing Simpira study was implemented using AES-NI. When using AES-NI instructions, the Addroundkey function cannot be omitted. However, we do not use AES-NI and implement each AES function individually. So, We can omit the Addroundkey operation that uses Z where all roundkey values are 0x00 among Addroundkey functions. For this reason, we omit a total of 12 Addroundkeys to 6 Addroundkeys.

3.1.3. Optimizing InvMixColumn

In line 6 of Algorithm 2, InvMixColumns operation is performed. Mixcolumn is not performed in the last round of AES. However, since Simpira's round function is implemented using AES-NI, Mixcolumn is included in the last round. So, after the round function ends, it is implemented by additionally using InvMixcolumn, which is used when decrypting Mixcolumn. In this process, performing InvMixcolumn operations at the end of the round is the same result of omitting the Mixcolumn operations once in the round function of AES. Therefore, it is more efficient to omit the Mixcolumn operation once than implement the InvMixcolumn, separately. As the result, since we directly implement the AES round function, we omit the operation of Mixcolumn and InvMixcolumn once each.

3.1.4. Optimized Addroundkey Function

The Addroundkey step in the existing AES performs an XOR operation on the extended roundkey and the current block bit by bit. However, the Addroundkey of AES used in Simpira has the characteristic of using a fixed roundkey value. The result of XOR operation, the round constant, roundkey, and the number of blocks b (i.e., 1) are used as the roundkey. As mentioned in Section 3.1.1, it is possible to pre-compute the roundkey using the round constant, roundkey, and number of block b (i.e., 1) with this characteristic.

Figure 2 summarizes the values for each roundkey. Among $RK[0] \sim RK[15]$, it can be seen that only the values corresponding to $RK[0]$, $RK[4]$, $RK[8]$, and $RK[12]$ are XOR

operations with the round constant. Using these properties, $RK[0]$, $RK[4]$, $RK[8]$, and $RK[12]$ perform the XOR operation with the bit value corresponding to the current block. Through this process, different roundkey values can be generated for each round. However, other roundkey(except $RK[0]$, $RK[4]$, $RK[8]$, and $RK[12]$) values are fixed at 0x00.

RK[0]	0x00 ⊕ c ⊕ b	RK[4]	0x10 ⊕ c ⊕ b	RK[8]	0x20 ⊕ c ⊕ b	RK[12]	0x30 ⊕ c ⊕ b
RK[1]	0x00	RK[5]	0x00	RK[9]	0x00	RK[13]	0x00
RK[2]	0x00	RK[6]	0x00	RK[10]	0x00	RK[14]	0x00
RK[3]	0x00	RK[7]	0x00	RK[11]	0x00	RK[15]	0x00

Figure 2. Values of each roundkey; RK = Roundkey, c is a counter that is initialized by one, and incremented after every use of $F_{c,b}$, Every $F_{c,b}$ consists of two AES round, where the round constants that are determined from (c, b), b is number of blocks.

Therefore, except for operations for $RK[0]$, $RK[4]$, $RK[8]$, and $RK[12]$, the results of the remaining operations are the same as those when the operation is not performed. The algorithm applying the optimized Addroundkey can be found at Algorithm 4.

Algorithm 4: Optimized Addroundkey in AVR microcontrollers (.macro round); $R0$, $R4$, $R8$, $R12$: input register, $R18$: temporary register, Y: indirect address register.

Input: $R0$, $R4$, $R8$, $R12$

Output: $R0$, $R4$, $R8$, $R12$

```
1: ld R18, Y+
2: eor R0, R18
3: ld R18, Y+
4: eor R4, R18
5: ld R18, Y+
6: eor R8, R18
7: ld R18, Y+
8: eor R12, R18
```

As the result, we omit the rest of the operations except $RK[0]$, $RK[4]$, $RK[8]$, and $RK[12]$ whose values change, reducing the operations of Addroundkey of the existing from 16 operations to 4 operations using Simpira's characteristics. Comparison results are shown in Table 4. For the Addroundkey operation, 48 cycles were obtained when the same operation was performed as before, whereas 12 cycles were obtained for this work. As a result, it reflects a performance improvement of 4.0×.

Table 4. Evaluation result of Addroundkey on 8-bit AVR microcontrollers in terms of execution timing (i.e., clock cycles); Notation (*) indicates implementation with optimization techniques.

This Work	This Work *
48	12

We implemented each module for Subbytes, Shiftrows, MixColumns, and Addroundkey of Simpira to call the module as needed. By implementing it as a Modularization, it is possible to efficiently manage the code.

Three optimization techniques of Sections 3.1.1–3.1.3 are equally applicable to 32-bit RISC-V processors. However, the technique in Section 3.1.4 does not apply to the 32-bit

RISC-V processor. Because it takes AVR's structural advantage of the 8-bit register size of the AVR microcontroller.

3.1.5. Using Optimized AES Implementation of AVR

For the optimal implementation of Simpira on the AVR microcontroller, it is necessary to first implement the optimization of the AES algorithm. We implemented Simpira by modifying Johannes Feichtner's [26] optimized code. Feichtner's is implemented by integrating the key extension step and AddRoundKey into one step. In this case, 4 LDD operations, 1 LDI operation, 4 ADD operations, and 16 EOR operations, a total of 25 operations are required.

As mentioned in Section 3.1.1, Simpira uses a round constant instead of an extended round key, unlike AES, so the value used as the round key can be calculated in advance. So, we omit it because we do not use the key expansion step. As mentioned in Section 3.1.2, if we perform *XOR* operation with roundkey and State of 0x00, the State value does not change, so we omit the total 12 Addroundkeys. Therefore, like Algorithm 4, 4 LD operations and 4 EOR operations are required for a total of 8 operations.

MixColumns are the most computationally expensive in AES. To implement MixColumns efficiently, Feichtner reduced the multiplication and XOR operations required for MixColumns to a minimum. Multiplication of 2 is implemented so that ADD operation, BRCC operation, and EOR operation are performed in order. As a result, MixColumns was efficiently implemented through 16 ADD operations, 16 BRCC operations, 64 EOR operations, and 36 MOV operations.

As a result, our Addroundkey implementation omitted 17 operations over Feichtner's, and we implemented Simpira using these optimized MixColumns. The code size to which our optimization technique is applied is as shown in our work * in Table 5.

Our AVR Implementation is available in the public domain at https://github.com/minjoo97/Implementation_AVR (accessed on 7 November 2022).

Table 5. Evaluation result on AVR microcontrollers and RISC-V processors about code size; Notation (*) indicates with optimization techniques.

Implementation	Processor	Code Size
Our work	low-end AVR	2122
	low-end RISC-V	1806
Our work *	low-end AVR	1978
	low-end RISC-V	1781

3.2. Optimized Implementation of Simpira on 32-Bit RISC-V

3.2.1. Simpira Optimized Implementation of RISC-V

The optimization technique in 32-bit RISC-V processors uses the same technique used in 8-bit AVR microcontrollers. The first is the pre-computation of the roundkey. Similar to the AVR optimization implementation, before starting the operation of the $F_{c,b}$ function using the feature of using a fixed b value, we performed optimization through roundkeys pre-computation. The second is the omission of the Addroundkey function. Since the RISC-V processor does not support AES-NI instruction sets, it is possible to omit Addroundkey where Z is used. The implementation when roundkey is Z is the same as Algorithm 5. Algorithm 5 omits four commands to load the roundkey and four commands to XOR operations, rather than when the Constant roundkey is used, resulting in a total of eight commands being optimized. Third, it is omitting InvMixcolumn. As above, since AES-NI is not supported, InvMixcolumn can be omitted by not performing Mixcolumns operation in the last round.

3.2.2. Using Optimized AES Implementation

For the optimal implementation of Simpira on 32-bit RISC-V processors, it is necessary to first implement the optimization of the AES algorithm. It is implemented by referring to Ko Stoffelen's [27] implementation of AES optimization on the RISC-V processor. In [27], the fastest implementation of encryption for a single block utilizes large lookup tables called T-tables, which combine the various steps of a round function. Encryption of a single 16-byte block is performed in 912 clock cycles. This uses 24-byte on the stack to store callee-save registers and a $4KiB$ lookup table.

We implemented Simpira by modifying the optimized code. Algorithm 5 is the code that performs one round of AES except for AddRoundkey. A brief description of the algorithm is as follows. First, calculate the value to add to the table address(LUT) in the state. Corresponds to lines 1–8, 17–24, 37–44, and 57–64 of Algorithm 5. By changing the table address using this value, the value is substituted, and the AES round operation is calculated through the XOR operation of the substituted value. Corresponds to lines 9–16, 25–36, 45–56, 65–76 of Algorithm 5. Many round functions are pre-computed in the T-table. That is, SubBytes, ShiftRows, and Mixcolumns are calculated only by table substitution and XOR operation. This code cannot be used in the last round, as we need to omit the Mixcolumns in the last round. So, in the final round, slightly modified code from that code is used. This does not require adding InvMixcolumns to that implementation.

As a result, the optimization implementation is shown in Figure 3. Figure 3 shows the basic structure of Simpira and the structure after optimization, and the code size to which our optimization technique is applied is as shown in our work * in Table 5. Our RISC-V Implementation is available in the public domain at https://github.com/ShuRaAtum/Implementation_RICS-V (accessed on 7 November 2022).

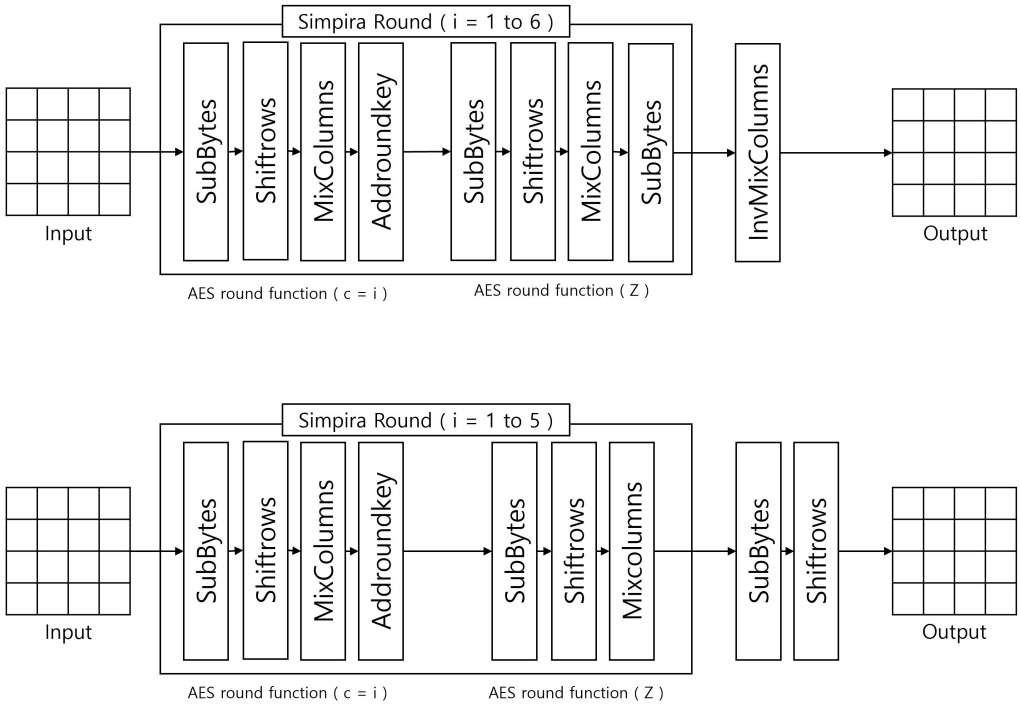

Figure 3. (**Top**) original Simpira structure/(**Bottom**) optimized Simpira structure.

Algorithm 5: Implementation of AES round function when the roundkey is Z in RISC-V processors (.macro zround); $X0 \sim X3$: input state register, $Y0 \sim Y3$: output state register, $LUT0 \sim LUT3$: look up table address, C: constant value (0xff0) register, $T0 \sim T4$: temp registers.

Input: $X0, X1, X2, X3$

Output: $Y0, Y1, Y2, Y3$

```
 1: andi T0, X0, 0xff
 2: andi T1, X1, 0xff
 3: andi T2, X2, 0xff
 4: andi T3, X3, 0xff
 5: slli T0, T0, 4
 6: slli T1, T1, 4
 7: slli T2, T2, 4
 8: slli T3, T3, 4
 9: add T4, T0, LUT1
10: lw Y0, (T4)
11: add T4, T1, LUT1
12: lw Y1, (T4)
13: add T4, T2, LUT1
14: lw Y2, (T4)
15: add T4, T3, LUT1
16: lw Y3, (T4)

17: srli X0, X0, 4
18: srli X1, X1, 4
19: srli X2, X2, 4
20: srli X3, X3, 4
21: and T0, X1, C
22: and T1, X2, C
23: and T2, X3, C
24: and T3, X0, C
25: add T4, T0, LUT3
26: lw T0, (T4)
27: add T4, T1, LUT3
28: lw T1, (T4)
29: add T4, T2, LUT3
30: lw T2, (T4)
31: add T4, T3, LUT3
32: lw T3, (T4)
33: xor Y0, Y0, T0
34: xor Y1, Y1, T1
35: xor Y2, Y2, T2
36: xor Y3, Y3, T3

37: srli X0, X0, 8
38: srli X1, X1, 8
39: srli X2, X2, 8
40: srli X3, X3, 8
41: and T0, X2, C
42: and T1, X3, C
43: and T2, X0, C
44: and T3, X1, C
45: add T4, T0, LUT0
46: lw T0, (T4)
47: add T4, T1, LUT0
48: lw T1, (T4)
49: add T4, T2, LUT0
50: lw T2, (T4)
51: add T4, T3, LUT0
52: lw T3, (T4)
53: xor Y0, Y0, T0
54: xor Y1, Y1, T1
55: xor Y2, Y2, T2
56: xor Y3, Y3, T3

57: srli X0, X0, 8
58: srli X1, X1, 8
59: srli X2, X2, 8
60: srli X3, X3, 8
61: and T0, X3, C
62: and T1, X0, C
63: and T2, X1, C
64: and T3, X2, C
65: add T4, T0, LUT2
66: lw T0, (T4)
67: add T4, T1, LUT2
68: lw T1, (T4)
69: add T4, T2, LUT2
70: lw T2, (T4)
71: add T4, T3, LUT2
72: lw T3, (T4)
73: xor Y0, Y0, T0
74: xor Y1, Y1, T1
75: xor Y2, Y2, T2
76: xor Y3, Y3, T3
```

4. Evaluation

This section introduces the evaluation of the proposed implementation. There are no comparative groups because we first implemented this on target processors. This compares the performance of each platform's Simpira C implementation and Assembly

implementation by setting the optimized level to -O3, and we compare the reference code [7] using AES-NI with the optimized implementation performance for each platform. The performance evaluation is measured in terms of execution timing (i.e., clock cycle).

4.1. 8-Bit AVR Microcontroller

The proposed implementation is evaluated on the ATmega128 microcontroller. The source code was implemented through the Microchip Studio Framework and compiled with compile option -O3. Since Simpira has never been implemented on the AVR microcontrollers, the reference code is ported to the AVR microcontroller and results and performance are compared. Comparison results are shown in Table 6. A reference C code takes 14,334 cycles. An optimized assembly implementation takes 2862 cycles, while the optimized implementation in assembly language achieved 2485 cycles. As a result, it confirmed that there is a 5.76× performance improvement over the-state-of-art reference implementation. We compared our implementation with reference code [7] using AES-NI.The comparison results are shown in the Table 6. Most platforms are much better than AVR, so our performance comparison shows that our reference implementation outperforms our implementation. However, we have optimally implemented Simpira on a low-end processor so that it can be used efficiently.

Table 6. Evaluation result on AVR microcontrollers, RISC-V processors with the optimization level -O3 in terms of execution timing (i.e., clock cycles); Notation (*) indicates with optimization techniques.

Implementation	Processor	Clock Cycles
Reference-C	low-end AVR	14,334
	low-end RISC-V	38,942
Our work	low-end AVR	2862
	low-end RISC-V	1106
Our work *	low-end AVR	2485
	low-end RISC-V	1052
[7]	Skylake (high-end Intel Processor)	50

4.2. 32-Bit RISC-V Processor

The proposed implementation is evaluated over the 32-bit RISC-V processor using a RV32I. The source code was implemented through the Freedom Studio Framework provided by SiFive. Similar to the results of AVR microcontrollers, Simpira has no implementation results on 32-bit RISC-V processors. The reference C code is transplanted to RISC-V and the results are compared. Comparison results are shown in Table 6. A Reference C code takes 38,942 cycles. Moreover, the assembly implementation takes 1106 cycles, while the optimized implementation in assembly language achieved 1052 cycles. As the result, it confirmed that there is a 37.01× performance improvement over the-state-of-art reference implementation. We compared our implementation with reference code [7] using AES-NI.The comparison results are shown in the Table 6 Most platforms are much better than RISC-V, so our performance comparison shows that our reference implementation outperforms our implementation. However, we have optimally implemented Simpira on a low-end processor so that it can be used efficiently.

5. Conclusions

In this paper, we propose an optimized implementation of Simpira Permutation on both 8-bit AVR microcontrollers and 32-bit RISC-V processors. The proposed techniques include the constant roundkey pre-computation and AddRoundKey, InvMixColumns operation omission. Taking advantage of AVR's structural features, the optimized implementation of the AVR microcontroller omits the operation of the 0x00 part of the constant

roundkey value. The proposed technique confirmed the performance improvement of 5.7× in AVR microcontrollers and 37.01× in RISC-V processors compared to the-state-of-art reference C implementation, respectively. This paper is the first Simpira Permutaion optimization study on 8-bit AVR and 32-bit RISC-V that does not support AES-NI features. Lastly, we also provide our implementation result in open-source. Any research followers can easily access and utilize it their own purposes.

As a future research project, we will propose an optimal implementation for Simpira Permutation on other IoT platforms.

Author Contributions: Formal analysis, M.S., S.E. and H.K. (Hyeokdong Kwon); Software M.S., S.E. and H.K. (Hyeokdong Kwon); Writing—original draft, M.S.; Writing—review and editing, K.J., H.K. (Hyunjun Kim), H.K. (Hyunji Kim), G.S. and W.L.; Supervision, H.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00264, Research on Blockchain Security Technology for IoT Services, 50%) and this work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00627, Development of Lightweight BIoT technology for Highly Constrained Devices, 50%).

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

References

1. Xie, C.; Yu, B.; Zeng, Z.; Yang, Y.; Liu, Q. Multilayer internet-of-things middleware based on knowledge graph. *IEEE Internet Things J.* **2020**, *8*, 2635–2648. [CrossRef]
2. Lu, J.; Chen, L.; Xia, J.; Zhu, F.; Tang, M.; Fan, C.; Ou, J. Analytical offloading design for mobile edge computing-based smart internet of vehicle. *EURASIP J. Adv. Signal Process.* **2022**, *2022*, 44. [CrossRef]
3. Alsamhi, S.H.; Shvetsov, A.V.; Kumar, S.; Hassan, J.; Alhartomi, M.A.; Shvetsova, S.V.; Sahal, R.; Hawbani, A. Computing in the Sky: A Survey on Intelligent Ubiquitous Computing for UAV-Assisted 6G Networks and Industry 4.0/5.0. *Drones* **2022**, *6*, 177. [CrossRef]
4. Zhao, M.; Li, J.; Tang, F.; Asif, S.; Zhu, Y. Learning based massive data offloading in the iov: Routing based on pre-rlga. *IEEE Trans. Netw. Sci. Eng.* **2022**, *9*, 2330–2340. [CrossRef]
5. Daemen, J.; Rijmen, V. Reijndael: The Advanced Encryption Standard. *Dr. Dobb's J. Softw. Tools Prof. Program.* **2001**, *26*, 137–139.
6. Akdemir, K.; Dixon, M.G.; Feghali, W.; Fay, P.G.; Gopal, V.; Guilford, J.; Ozturk, E.; Wolrich, G.; Zohar, R. Breakthrough AES Performance with Intel® AES New Instructions. 2010. Available online: https://www.semanticscholar.org/paper/Breakthrough-AES-Performance-with-Intel-%C2%AE-AES-New-Akdemir-Dixon/62116fe84e7360202d4e1cff859c8fc014ef4614 (accessed on 26 September 2022).
7. Gueron, S.; Mouha, N. Simpira v2: A family of efficient permutations using the AES round function. In *International Conference on Cryptology and Information Security in Latin America*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 95–125.
8. Ahmad, S.; Alam, K.M.R.; Rahman, H.; Tamura, S. A comparison between symmetric and asymmetric key encryption algorithm based decryption mixnets. In Proceedings of the 2015 International Conference on Networking Systems and Security (NSysS), Dhaka, Bangladesh, 5–7 January 2015; pp. 1–5.
9. Yassein, M.B.; Aljawarneh, S.; Qawasmeh, E.; Mardini, W.; Khamayseh, Y. Comprehensive study of symmetric key and asymmetric key encryption algorithms. In Proceedings of the 2017 International Conference on Engineering and Technology (ICET), Antalya, Turkey, 21–23 August 2017; pp. 1–7.
10. Rajesh, S.; Paul, V.; Menon, V.G.; Khosravi, M.R. A secure and efficient lightweight symmetric encryption scheme for transfer of text files between embedded IoT devices. *Symmetry* **2019**, *11*, 293. [CrossRef]
11. Bernstein, D.J.; Hülsing, A.; Kölbl, S.; Niederhagen, R.; Rijneveld, J.; Schwabe, P. The SPHINCS+ signature framework. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 2129–2146.
12. NIST PQC Project. Available online: https://csrc.nist.gov/Projects/post-quantum-cryptography (accessed on 29 July 2022).
13. ATmega128 Datasheet. Available online: www.microchip.com/wwwproducts/en/ATmega128 (accessed on 16 August 2022).
14. The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2. Available online: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (accessed on 16 August 2022).
15. Waterman, A.; Lee, Y.; Patterson, D.A.; Asanović, K. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*; Version 2.1. 2016. Available online: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf (accessed on 26 September 2022).
16. Kim, K.; Choi, S.; Kwon, H.; Liu, Z.; Seo, H. FACE–LIGHT: Fast AES–CTR mode encryption for Low-End microcontrollers. In *International Conference on Information Security and Cryptology*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 102–114.

17. Park, J.H.; Lee, D.H. FACE: Fast AES CTR mode encryption techniques based on the reuse of repetitive data. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, *2018*, 469–499. [CrossRef]
18. Kwon, H.; An, S.; Kim, Y.; Kim, H.; Choi, S.J.; Jang, K.; Park, J.; Kim, H.; Seo, S.C.; Seo, H. Designing a CHAM block cipher on low-end microcontrollers for internet of things. *Electronics* **2020**, *9*, 1548. [CrossRef]
19. Roh, D.; Koo, B.; Jung, Y.; Jeong, I.W.; Lee, D.G.; Kwon, D.; Kim, W.H. Revised version of block cipher CHAM. In *International Conference on Information Security and Cryptology*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 1–19.
20. Kim, H.; Sim, M.; Eum, S.; Jang, K.; Song, G.; Kim, H.; Kwon, H.; Lee, W.K.; Seo, H. Masked Implementation of PIPO Block Cipher on 8-bit AVR Microcontrollers. In *International Conference on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 171–182.
21. Kim, H.; Jeon, Y.; Kim, G.; Kim, J.; Sim, B.Y.; Han, D.G.; Seo, H.; Kim, S.; Hong, S.; Sung, J.; et al. PIPO: A lightweight block cipher with efficient higher-order masking software implementations. In *International Conference on Information Security and Cryptology*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 99–122.
22. Eum, S.W.; Kwon, H.D.; Kim, H.J.; Yang, Y.J.; Seo, H.J. Implementation of LEA Lightwegiht Block Cipher GCM Operation Mode on 32-Bit RISC-V. *J. Korea Inst. Inf. Secur. Cryptol.* **2022**, *32*, 163–170.
23. Hong, D.; Lee, J.K.; Kim, D.C.; Kwon, D.; Ryu, K.H.; Lee, D.G. LEA: A 128-bit block cipher for fast encryption on common processors. In *International Workshop on Information Security Applications*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 3–27.
24. Kwon, H.; Kim, H.; Eum, S.; Sim, M.; Kim, H.; Lee, W.K.; Hu, Z.; Seo, H. Optimized Implementation of SM4 on AVR Microcontrollers, RISC-V Processors, and ARM Processors. *IEEE Access* **2022**, *10*, 80225–80233. [CrossRef]
25. Cheng, H.; Ding, Q. Overview of the block cipher. In Proceedings of the 2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control, Washington, DC, USA, 8–10 December 2012; pp. 1628–1631.
26. Efficient Implementations of AES-128 and Grøstl-256 for the AVR 8-Bit Microcontroller Architecture. Available online: https://github.com/Churro/avr-aes128-groestl256/blob/master/Paper.pdf (accessed on 16 August 2022).
27. Stoffelen, K. Efficient Cryptography on the RISC-V Architecture. In *International Conference on the Theory and Application of Cryptology and Information Security*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 323–340.