Simpira Gets Simpler: Optimized Simpira on Microcontrollers

Minjoo Sim[1], Siwoo Eum[1], Hyeokdong Kwon[1], Kyungbae Jang[1], Hyunjun Kim[1], Hyunji Kim[1], Gyeongju Song[1], Wai-Kong Lee[2], and Hwajeong Seo[1][0000−0003−0069−9061]

[1]IT Department, Hansung University, Seoul (02876), South Korea,
`{ minjoos9797, shuraatum, korlethean, starj1023, khj930704, khj1594012, thdrudwn98, hwajeong84}@gmail.com`
[2]Department of Computer Engineering,
Gachon University, Seongnam, Incheon (13120), Korea,
`waikonglee@gachon.ac.kr`

Abstract. Simpira Permutation is a Permutation design using the AES algorithm. The AES algorithm is the most widely used in the world, and Intel has developed a hardware accelerated AES instruction set (AES-NI) to improve the performance of encryption. By using AES-NI, Simpira can be improved further. However, low-end processors that do not support AES-NI require efficient implementation of Simpira optimization. In this paper, we introduce a optimized implementation of a Simpira Permutation in 8-bit AVR microcontrollers and 32-bit RISC-V processors, that do not support the AES instruction set. We firstly pre-computed round keys and omitted the Addroundkey. Afterward, the MixColumn and InvMixColumn of the final round (i.e. 12-th), which were added unnecessarily due to characteristics of Simpira using AES-NI, were omitted. In the AVR microcontroller, the Addroundkey consists of 16 operations, but it has been optimized by eliminating operations where the value of roundkeys is `0x00`, omitting Addroundkey to 4 operations. In the RISC-V processor, it is implemented using a same optimization technique of AVR implementation. We have carried out experiments 8-bit ATmega128 microcontroller and 32-bit RISC-V processor, which shows up-to `5.76`× `and 37.01`× better performance enhancement than reference codes for the Simpira Permutation, respectively.

Keywords: AES · Software Implementation · Simpira Permutation · 8-bit AVR Microcontroller · 32-bit RISC-V Processor

1 Introduction

AES (Advanced Encryption Standard) is an encryption algorithm that adopted by the National Institute of Standards and Technology (NIST) in 2001 [1]. Since then, AES block cipher has become the most used encryption algorithm in the world. In 2008, Intel developed an instruction set (AES-NI [2]) 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 [3]. However, the proposed Simpira permutation cannot be used generally, because many kinds of processors do not support the AES instruction set. In this paper, we propose an optimal implementation of Simpira Permutation on an 8-bit AVR microcontroller and a 32-bit RISC-V processor that does not provide the AES instruction set. Main contributions of this work are as follow:

1.1 Contributions

- **Optimized Simpira on the 8-bit AVR architecture.** An 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 Permutation uses AES algorithm, but the target processor does not support AES-NI instruction set. Since AES-NI is not available, we have used existing AES to enable Simpira to behave the same as AES-NI instruction set to operate on the target processor.
  To implement the Simpira, some offset functions are optimized away. For instance, the last round of Mixcolumns and InvMixcolums can be omitted because it can be operated in the opposite way. Some of AddRoundKey functions uses `0x00` roundkeys. Since it has no effect on the result value, it is operated by 16 times. With optimization techniques, the proposed implementation requires only 4 times of computations. We have carried out experiments shows up-to `5.76`× better performance enhancements than reference code for the Simpira Permutation.
- **Optimized Simpira on the 32-bit RISC-V architecture.** A RISC-V is an open source computer CPU architecture. We presents the optimal implementation of Simpira, whose permutation is implemented with the AES algorithm. However, on a 32-bit RISC-V processor does not support AES-NI instruction sets. In particular, we optimized by omitting the operation using the optimized AES algorithm. We have carried out experiments shows up-to `37.01`× better performance enhancement than reference code for the Simpira Permutation.
- **First optimized-implementation for Simpira on 8-bit AVR mirco-controller and 32-bit RISC-V processor.** The implementation on the low-end processor for Simpira, an algorithm used inside SPHINCS+ [4] and an algorithm that advanced to the NIST PQC Round3, has not yet been explored before except for the implementation on the ARM processor.

2 Related Works

2.1 AES

Algorithm 1 AES Algorithm

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

The AES(Advanced Encryption Standard) is a symmetric block cipher that uses the 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 applies 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 [1]. The overall operation codes are detailed in Algorithm 1.

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. Also, 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 a 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 [3].

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 the 5 line of the Algorithm 3 means 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

Algorithm 2 Simpira Algorithm

procedure Simpira($state, rk$)
1: $R \leftarrow 6$
2: **for** $c = 1$ **to** R **do**
3: $state \leftarrow \mathrm{F}_{c,b}(state)$
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: $RK[0] = \texttt{0x00} \oplus c \oplus b$
2: $RK[4] = \texttt{0x10} \oplus c \oplus b$
3: $RK[8] = \texttt{0x20} \oplus c \oplus b$
4: $RK[12] = \texttt{0x30} \oplus c \oplus b$
5: **return** $AES(AES(state, RK), Z)$
end procedure

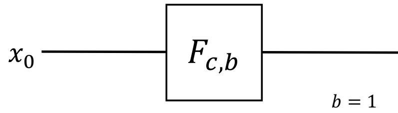

Fig. 1: Structure of Simpira about $b = 1$; c is a counter that is initialized by one, and incremented after every use of $F_{c,b}$, Every $F_{c,b}$ consists of two AES round, where the round constants that are determined from (c, b) where b is number of blocks

(b). In this paper, b is set to 1 where b is number of blocks, it is used as a standard.

2.3 8-bit AVR Microcontroller

The 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 [5]. Instructions used to implement the optimized Simpira cipher 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

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

2.4 32-bit RISC-V Processor

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 as shown in Table 2. Among them, there are sp registers and $s0 \sim s11$ registers as callee-saved registers that preserve the value before using the register and return the value after use.

3 Proposed Method

3.1 Optimized Implementation of Simpira on 8-bit AVR microcontroller

Constant Roundkey Pre-compute. Since the AES algorithm used in Simpira uses a round constant unlike the original AES extended roundkey, it is possible

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. [6]

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

to calculate the value used as the roundkey in advance. Before entering the AES round function in Algorithm 3, the roundkey is calculated in advance and the AES round function operation is performed. In this paper, parameter (b) is set to 1 because it is implemented for the case where the value of b (the number of blocks) is 1. Since the roundkey always uses a fixed value due to the fixed value of b, it is possible to calculate the roundkey in advance without having to recalculate the roundkey every round during the operation of the $F_{c,b}$ function. For this reason, it is possible to pre-compute the roundkey. The operation (operation of the round key) performed in lines 1 to 4 of Algorithm 3 can be omitted.

Omit 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 round keys are used per round and a total of 12 round keys. Since one round key per round is `0x00`, there are 6 round keys 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 the existing Simpira study was implemented using AES-NI. When using AES-NI instructions, the Addroundkey function cannot be omitted. If the value of roundkey is `0x00`, the Addroundkey function is executed. Since proposed implementation does not use AES-NI and implements each AES function, individually. In proposed implementation, it is possible to omit the Addroundkey operation in which Z, where all roundkey values are `0x00`, is used among the Addroundkey functions. For this reason, we omitted a total of 12 Addroundkeys as 6 Addroundkeys.

Optimizing InvMixColumn. In line 6 of Algorithm 2, InvMixColumns operation is performed. The round function of Simpira consists of the AES round

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
```

function. 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. In other words, when the AES round function is used 12 times in Simpira, the Mixcolumn operation is omitted in the final 12-th AES round function, giving the effect of calculating the same InvMixcolumn. Therefore, it is more efficient to omit the Mixcolumn operation once than implementing the InvMixcolumn, separately. As a result, it is possible to optimize away Mixcolumn once and InvMixcolumn.

Three optimization techniques listed above are equally applicable to 32-bit RISC-V processors. The following technique cannot be applied on 32-bit RISC-V processors, where it is applicable only for 8-bit AVR microcontrollers.

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. Addroundkey executes one column at a time, and serves to strengthen security by mixing the bits (current block) that have gone through three stages: SubBytes, ShiftRows, and MixColumns. However, Addroundkey of AES used in Simpira has a characteristic of using a fixed roundkey value. Using a fixed roundkey value is vulnerable to security. 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 the Constant Roundkey Pre-compute section, 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 operation with the round constant. Using these characteristics, $RK[0]$, $RK[4]$, $RK[8]$, and $RK[12]$ can result in different roundkey values for each round by performing XOR operations with the bit values corresponding to the current block. However, other roundkey values are fixed at `0x00`. When we perform XOR operations on bits corresponding to this roundkey and the current block, values do not change when comparing with pre-operation values.

Therefore, using the feature that there is no change in value when XOR operation is performed with `0x00`, except for operations on RK[0], RK[4], RK[8],

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

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

Table 4: Evaluation result of Addroundkey on 8-bit AVR microcontrollers (in terms of speed; clock cycles).

No optimization	Optimization
48	**12**

and RK[12]. Other roundkeys it is possible to omit the XOR operation because the result is the same as when it is not performed. The algorithm applying the optimized Addroundkey can be found at 4.

In this paper, 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 by `4.0`×.

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.

Using an optimized AES implementation. For the optimal implementation of Simpira on the AVR microcontroller, it is necessary to firstly implement the optimization of the AES algorithm. Aoki et al. came up with an approach that greatly reduces the amount of XOR operations required in the Mixcolumns step that works in Grøstl [7]. Through the approach of Aoki et al, Feichtner has been shown that multiplication operations are possible without additional overhead [8]. The hash function Grøstl, an AES-based algorithm, uses the same Sbox as AES, and Grøstl's Mixcolumns and AES's Mixcolumns do similar things. Therefore, in this paper, MixColumns is implemented similarly to Feichtner's approach.

3.2 Optimized Implementation of Simpira on 32-bit RISC-V

Simpira Optimized Implementation. 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. The second is the omission of the Addroundkey function. Since 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.

Using an optimized AES implementation. For the optimal implementation of Simipra on 32-bit RISC-V processors, it is necessary to firstly implement the optimization of the AES algorithm. It is implemented by referring to Ko Stoffelen's [9] implementation of AES optimization on the RISC-V processor. In [9], 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 bytes on the stack to store callee-save registers and $4KiB$ lookup table.

As a result, the optimization implementation is shown in the Figure 3. The Figure 3 shows the basic structure of Simpira and the structure after optimization.

4 Evaluation

This section introduces the evaluation of the proposed implementation. There are no comparative groups because we firstly 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. The performance evaluation is measured in terms of execution timing (i.e. *clock cycle*).

4.1 8-bit AVR Microcontroller

The proposed implementation measures the ATmega128 microcontroller in the AVR microcontroller. The source code was implemented through the Microchip Studio Framework, and compiled with compile option`-O3`. Since Simpira has never been implemented on an AVR microcontroller, the reference code is ported to the AVR microcontroller and results and performance are compared. Comparison results are shown in Table 5. A Reference C code takes 14,334 cycle. And optimized assembly implementation takes 2,862 cycle, while the proposed optimization implementation in assembly language achieved 1,052 cycle. As a result, it confirmed that there is a 5.76× performance improvement over C implementation.

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, $LUT0 \sim 3$: look up table address, C: constant value (`0xff0`) register, $T0 \sim 4$: 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
```

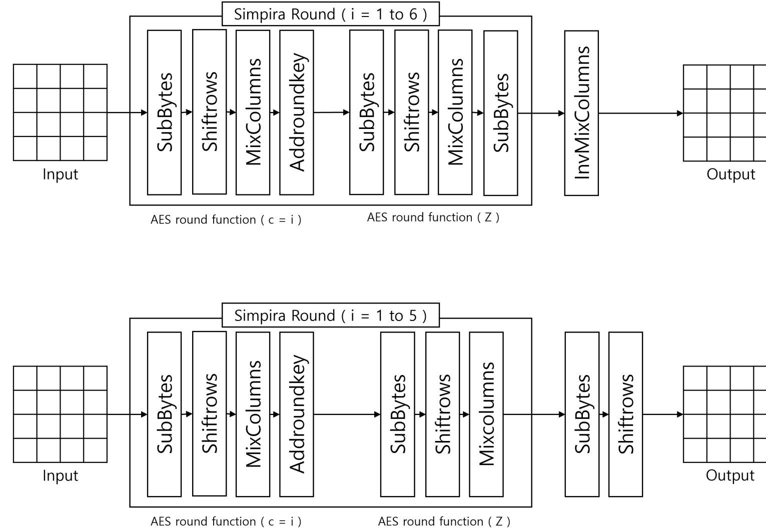

Fig. 3: (Top) original Simpira structure / (Bottom) optimized Simpira structure.

Table 5: Evaluation result on AVR microcontrollers with the optimization level `-O3` in terms of execution timing (i.e. clock cycles); Notation (∗) indicates with optimization techniques.

Reference C code	This work	**This work∗**
14,334	2,862	**2,485**

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 39,842 cycle. And the assembly implementation takes 1,106 cycle, while the optimized implementation in assembly language achieved 1,052 cycle. As a result, it confirmed that there is a 37.01× performance improvement over C implementation.

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

Reference C code	This work	**This work∗**
38942	1106	**1052**

5 Conclusion

In this paper, we propose an optimized implementation of Simpira Permutation on both an 8-bit AVR microcontroller and a 32-bit RISC-V processor. The proposed techniques include the constant roundkey pre-computation and AddRoundKey, InvMixColumns operation omission. In AVR microcontrollers, the operation of the `0x00` part of the Constant roundkey value is omitted. The proposed technique confirmed the performance improvement of 5.7× in AVR microcontrollers and 37.01× in RISC-V processors compared to the 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. As a future research project, we propose the optimal implementation of various block sizes of Simpira.

References

1. J. Daemen and V. Rijmen, “Reijndael: The advanced encryption standard.,” *Dr. Dobb’s Journal: Software Tools for the Professional Programmer*, vol. 26, no. 3, pp. 137–139, 2001.
2. K. Akdemir, M. G. Dixon, W. Feghali, P. G. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, “Breakthrough aes performance with intel ® aes new instructions,” 2010.
3. S. Gueron and N. Mouha, “Simpira v2: A family of efficient permutations using the aes round function.,” in *International Conference on Cryptology and Information Security in Latin America*, pp. 95–125, Springer, 2016.
4. D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, “The sphincs+ signature framework,” in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 2129–2146, 2019.
5. A. Corporation, “Atmega128(l) datasheet,” 2021.
6. A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, “The risc-v instruction set manual, volume i: User-level isa, version 2.1,” 2016.
7. K. Aoki, G. Roland, Y. Sasaki, and M. Schläffer, “Byte slicing grøstl optimized intel aes-ni and 8-bit implementations of the sha-3 finalist grøstl,” in *Proceedings of the International Conference on Security and Cryptography*, pp. 124–133, IEEE, 2011.
8. J. Feichtner, “Efficient grøstl-256 implementations for the avr 8-bit microcontroller architecture,” 2012.
9. K. Stoffelen, “Efficient cryptography on the risc-v architecture,” in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 323–340, Springer, 2019.