Efficient Implementation of the Classic McEliece on ARMv8 processors

Minjoo Sim[1][0000−0001−5242−214X],
Hyeokdong Kwon[1][0000−0002−9173−512X],
Siwoo Eum[1][0000−0002−9583−5427],
Gyeongju Song[1][0000−0002−4337−1843],
Minwoo Lee[2][0000−0002−2356−3055], and
Hwajeong Seo[2][0000−0003−0069−9061]

[1]Department of Information Computer Engineering,
Hansung University, Seoul (02876), South Korea,
[2]Department of Convergence Security,
Hansung University, Seoul (02876), South Korea,
{minjoos9797, korlethean, shuraatum, thdrudwn98, minunejip,
hwajeong84}@gmail.com

Abstract. Classic McEliece is a Code-based Key Encapsulation Mechanisms(KEM) and one of the candidate algorithms in the NIST PQC competition. Based on the McEliece cryptosystem developed in 1978, this system relies on the Niederreiter variant of McEliece. It consists of three phases: Key Generation, Encapsulation, and Decapsulation. In this paper, we propose an optimized implementation of the internal multiplication operations of Classic McEliece on the ARMv8 processor. We utilize parallel computing techniques using vector registers and vector instructions of the ARMv8 processor. We specifically focus on optimizing the multiplication operation, which is a major contributor to the overall execution time of the Classic McEliece algorithm, by leveraging the commutative property and implementing an parallelization technique. As a result, our approach achieves a maximum performance improvement of 2.82× compared to the reference implementation in the multiplication operation.

Keywords: 64-bit ARMv8 Processors, Classic McEliece, Post-quantum Cryptography, Parallel implementation, KEM, Software implementation

1 Introduction

Due to the rapid advancement of quantum computers, conventional modern Cryptography algorithms are facing threats. In preparation for this, efforts are underway to transition existing modern Cryptography algorithms into quantum-resistant Cryptography before the practical realization of quantum computers. Recognizing the need for robust Cryptography solutions, the National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography(PQC) Competition in 2017. The objective of this competition is to identify

and standardize quantum-resistant Cryptography algorithms [1]. As part of the NIST PQC standards, CRYSTALS-KYBER [2], CRYSTALS-DILITHIUM [3], SPHINCS$^+$ [4], and FALCON [5] have been selected. Optimal implementation research for the algorithm selected as a NIST PQC standard is being actively conducted on ARMv8 [6–11]. Classic McEliece was selected as one of the Round 4 candidate algorithms [12]. Classic McEliece is a code-based key encapsulation mechanisms. The basic structure is based on the McEliece [13] cryptosystem in 1978, and its stability has been verified through long-term research. In addition, the German Federal Office for Information Security recommends Classic McEliece as long-term security along with FrodoKEM. In this paper, we propose an efficient implementation of Classic McEliece on a 64-bit ARMv8 processor.

The remainder of this paper is structured as follows. The Section 2 describes the Classic McEliece, the target 64-bit ARMv8 processor, and related works. The Section 3 describes the proposed method. The Section 4 shows a performance comparison. Finally, the Section 5 describes the conclusion of this document and future work.

1.1 Contribution

-Multiplication with Parallel Operations of the Classic McEliece on ARMv8 Processors We propose an efficient parallel multiplication implementation using Classic McEliece's parallel operation on ARMv8 processors. The implementation utilizes 128-bit vector registers of the ARMv8 architecture and NEON vector instructions. Our proposed parallel multiplication leverages the commutative property of XOR operation to rearrange internal operations, enabling simultaneous computation of four values within the internal structure. As a result, we observed a maximum performance improvement of approximately 2.82× compared to the reference code (C code) provided by PQ-Clean project [14].

-First Implementation of the Classic McEliece Multiplier on 64-bit ARMv8 Processors Using Vector Registers. To the best of our knowledge, this paper presents the first implementation of the Classic McEliece multiplier utilizing vector registers supported by 64-bit ARM processors. We believe that our work can serve as a valuable resource for researchers to assess the performance of the Classic McEliece algorithm.

2 Preliminaries

2.1 Classic McEliece

Classic McEliece is designed to combine the advantages of McEliece and Niederreiter. The existing McEliece uses a Generator Matrix(G) for the public key, whereas Classic McEliece uses the Parity Check Matrix(H) used as the public

key in Niederreiter. Classic McEliece is designed with a simple matrix multiplication process for Encapsulation and Decapsulation, allowing for fast computation. It also has the advantage of having a shorter Ciphertext compared to the existing Ciphertext. On the other hand, the length of the public key is very long and the key generation process takes a long time. The length of the public key is 256KB to 1.3MB, using it difficult to use on low-end devices with small memory space. Classic McEliece parameters are shown in Table 1.

Table 1. Parameters of Classic McEliece; **m** is $log_2 q$ (q is the size of the field used); **n** is length of code, and **t** is the sizes of guaranteed error-correction capability;

Algorithm	m	n	t	security level	Public key	Secret key
Mceliece 348864	12	3,488	64	1	261,120	6,492
Mceliece 460896	13	4,608	96	3	524,160	13,608
Mceliece 6688128	13	6,688	128	5	1,044,992	13,932
Mceliece 6960119	13	6,960	119	5	1,047,319	13,948
Mceliece 8192128	13	8,192	128	5	1,357,824	14,120

Classic McEliece algorithm can be divided into three processes: a key generation process, an encryption process(Encapsulation), and a decryption process(Decapsulation).

- **Key Generation** In the Key Generation process, first, g(x) of degree t required for Goppa code generation and L called a support set are generated. Generate H(parity check matrix) using g(x) and L. The generated H is converted to binary form and converted to systematic form by performing Gaussian elimination. That is, it is converted to the form H = (I_{n-k}|T), and after removing I_{n-k}(Identity Matrix), the remaining **T** matrix is used as a public key. The private key consists of **g(x)** and **L**, which are used to generate the Goppa code, and a randomly generated **s**. In conclusion, the public key is **T** and the private keys are **g(x)**, **L**, and **s**.
- **Encapsulation** In the Encapsulation process, a random vector(e) with weight t is first generated. A syndrome(C_0) is generated using the generated e and the public key(T). It uses the value of e and the number 2 to generate a hash value(C_1 = Hash(2, e)) and combines the two values(C = C_0|C_1) to finally produce the ciphertext(C). Finally, for the session key, the hash value of the number 1, e, C will be the session key(K = Hash(1, e, C)).
- **Decapsulation** In the Decapsulation process, Decapsulation is performed using the delivered value of C and the owned private key. The value of e(error matrix) can be obtained by performing syndrome decoding with the syndrome(C_0) included in C(ciphertext) and the private key. It is determined whether there is an error by comparing the hash value with the number 2 in front of the e value obtained through syndrome decoding and the C_1 value included in the transmitted C(ciphertext). If the two values are the same, the hash value of the numbers 1, e, and C is computed to obtain the session key.

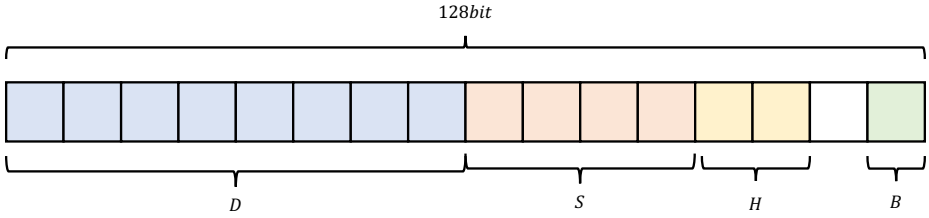

Fig. 1. Register packing of vector registers.

2.2 ARMv8 Processor

ARM is an ISA(Instruction Set Architecture) high-performance embedded processor. ARMv8-A supports both 32-bit AArch32 and 64-bit AArch64 architectures for backward compatibility. ARMv8-A provides 31 64-bit general-purpose registers from $x0$ to $x30$ and 32 128-bit vector registers from $v0$ to $v31$. In this case, the general purpose registers can also be used as 32-bit registers from $w0$ to $w30$. The vector registers can be processed by dividing stored values into specific units. There are four types of units supported: byte (8-bit), half word (16-bit), single word (32-bit), and double word (64-bit). Figure 1 shows register packing of vector registers. A vector instructions (called ASIMD or NEON) is used for the vector register to perform parallel operation. Table 2 shows that instruction lists for proposed implementations [15].

2.3 Related Works

Becker et al. implemented an optimization for Barrett multiplication using the 64-bit ARM Cortex-A NEON vector instruction [7]. They are the combination of Montgomery multiplication and Barrett reduction resulting in Barrett multiplication which allows particularly efficient modular one-known-factor multiplication using the NEON vector instructions. And proposed novel techniques combined with fast two-unknown-factor Montgomery multiplication, Barrett reduction sequences, and interleaved multi-stage butterflies result in significantly faster code. As a result, in the Saber, NTTs are far superior to Toom-Cook multiplication on the ARMv8-A architecture, outrunning the matrix-to-vector polynomial multiplication by 2.0×. On the Apple M1, the matrix-vector products run 2.1× and 1.9× faster for Kyber and Saber respectively.

Sanal et al. implemented CRYSTAL-Kyber encryption on 64-bit ARM Cortex-A and Apple A12 processors [8]. They improved the performance of noise sampling, Number Theoretic Transform (NTT), and symmetric function implementations based on an AES accelerator. As the result, the proposed Kyber512 implementation on ARM64 improved the previous work by 1.72×, 1.88×, and 2.29× for key generation, encapsulation, and decapsulation, respectively. And, the proposed Kyber512-90s implementation(using AES accelerator) is improved by 8.57×, 6.94×, and 8.26× for key generation, encapsulation, and decapsulation, respectively.

Table 2. Summarized instruction set of ARMv8 for Classic McEliece multiplier implementation; `Xd, Vd`: destination register (general, vector), `Xn, Vn, Vm`: source register (general, vector, vector), `Vt`: transferred vector register.

asm	Operands	Description	Operation
ADD	Vd.T, Vn.T, Vm.T	Add	Vd ← Vn + Vm
AND	Vd.T, Vn.T, Vm.T	Bitwise AND	Vd ← Vn & Vm
EOR	Vd.T, Vn.T, Vm.T	Bitwise Exclusive OR	Vd ← Vn ⊕ Vm
LD1	Vt.T, [Xn]	Load multiple single-element structures	Vt ← [Xn]
LD1R	Vt.T, [Xn]	Load single 1-element structure and replicate to all lanes (of one register).	Vt.T ← [Xn]
MOV	Xd, #imm	Move(immediate)	Xd ← imm
MOV	Vd.T, Vn.T	Move(vector)	Vd ← Vn
MOV	Vd.Ts[index1], Vn.Ts[index2]	Move vector element to another vector element	Vd ← Vn
MOVI	Vt.T, #imm	Move immediate (vector)	Vt ← #imm
MUL	Xd, Xn, Xm	Multiply	Xd ← Xn × Xm
RET	{Xn}	Return from subroutine	Return
SHL	Vd.T, Vn.T, #shift	Shift Left immediate (vector)	Vd ← Vn <<#shift
SRI	Vd.T, Vn.T, #shift	Shift Right and immediate (vector)	Vd ← Vn >>#shift
ST1	Vt.T, [Xn]	Store multiple single-element structures from one, two, three, or four registers	[Xn] ← Vt
SUB	Xd, Xn, #imm	Subtract immediate	Xd ← Xn - #imm
REV32	Vd.T, Vn.T	Reverse elements in 32-bit words	Vd ← Vn of Reverse
CBNZ	Wt, Label	Compare and Branch on Nonzero	Go to Label
CBZ	Wt, Label	Compare and Branch on Zero	Go to Label
ZIP1	Vd.T, Vn.T, Vm.T	Zip vectors primary	Vd ← Vn[even], Vm[even] Vd ← Vn[odd], Vm[odd]
UZP1	Vd.T, Vn.T, Vm.T	Unzip vectors primary	Vd ← Vn[even], Vm[even] Vd ← Vn[odd], Vm[odd]

Kwon et al. implemented the Rainbow signature schemes on 64-bit ARM Cortex-A processor [16]. Rainbow signature is based on the multivariate-based public key signature. They proposed a technique using a look-up table, in which the result of 4×4 multiplication is pre-computed. The techniques used parallel operation of vector registers and vector instructions. As a result, the proposed multiplier by using look-up table performances improvement was improved previous work by maximum of 167.2×.

Kwon et al. implemented the FrodoKEM on 64-bit ARM Cortex-A processor [17]. FrodoKEM is Public-key Encryption and Key-establishment Algorithms, which is selected NIST PQC Round 3 alternate candidates. They proposed the parallel matrix-multiplication and built-in AES accelerator for AES encryption. They applied these techniques to the FrodoKEM640 scheme, utilizing vector registers and vector instructions. And the implementation with all of proposed techniques was improved previous C implementation by maximum of 10.22×.

3 Proposed Method

3.1 Multiplication on $\mathbb{F}_{2^m}$.

In Classic McEliece, Multiplication is performed on the extended binary finite-filed $\mathbb{F}_{2^m}$. The expensive operations on public keys are multiplication and inversion on finite-field. Therefore, in this paper, optimization of multiplication on $\mathbb{F}_{2^m}$ is performed (m is 12 or 13). In the specification, $\mathbb{F}_{2^{12}}$ consists of $\mathbb{F}_2[x]/(x^{12}+x^3+1)$ and $\mathbb{F}_{2^{13}}$ consists of $\mathbb{F}_2[x]/(x^{13}+x^4+x^3+x+1)$ [18].

Multiplication on $\mathbb{F}_{2^m}$ proceeds as follows. Multiplication is performed on two m-bit values. At this time, since the multiplication result may be out of the range of $\mathbb{F}_{2^m}$, the multiplication is completed on $\mathbb{F}_{2^m}$ by performing modular reduction on the multiplication result value.

To ensure accurate multiplication of two 16-bit values, it is necessary to apply a masking operation using 32-bit variables. By masking the 16-bit variables, multiplication can be performed accurately. Subsequently, the resulting 16-bit values are obtained by unmasking the computed values. Figure 2 is the masking and unmasking operations performed for the multiplication operation in Classic McEliece. After masking four 16-bit with four 32-bit, multiplication is performed. After the multiplication operation is completed, the multiplication result is unmasked again and 4 values of 16 bits are returned.

Algorithm 1 is an optimization implementation code for multiplication on $\mathbb{F}_{2^m}$. Load four 16 bits into one vector register. At this time, four 16-bit values are loaded into each of two vector registers. The having the same value four 16-bit value is loaded into one vector register(`v0`) and four different 16-bit values are loaded into another vector register(`v1`)in lines 5-6. Lines 7-8 are performed a masking operation is applied to the four 16-bit values received as input. This masking process ensures that the variables are appropriately extended to 32 bits. Lines 9-26 perform multiplication operations. Lines 27-37 perform the multiplication operations, reduction operations are performed to obtain the desired

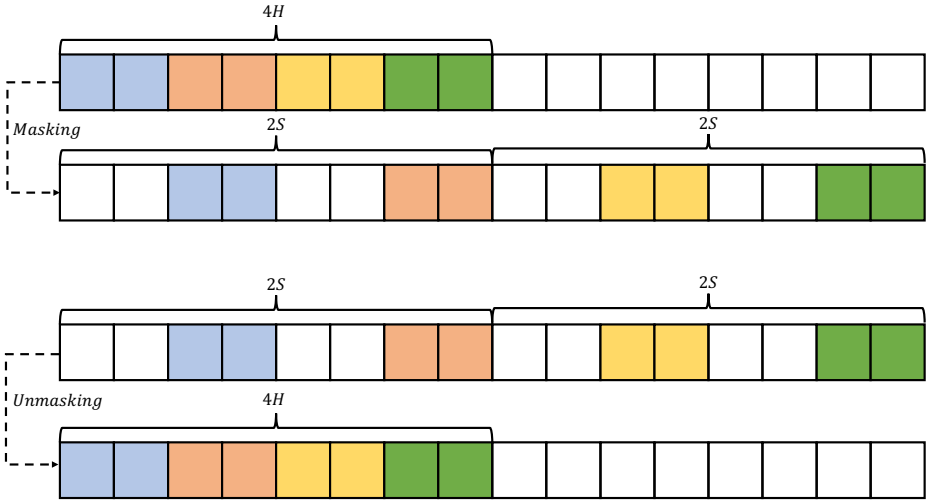

Fig. 2. Masking and Unmasking (In Classic McEliece)

results. Line 38 performs unmasking on the multiplication result. Lines 39-41 call the result of the previous multiplication operation, perform the multiplication operation and xor operation as above, and store the result of the multiplication operation in `x0`. Lines 42-43 return to Line 4 to Label if the value of `x8` is not 0, and repeat the operation until the value of `x8` becomes 0. Lines 44-48 When the value of `x8` becomes 0, the value of x9 is subtracted by one. The address value of `x2`, which is loaded into `v1`, and the address value of `x0`, where the operation result is stored, are moved appropriately. Move the address value of `x1` loaded into `v0` by 16 bits. The above operations are repeated until the value of `x9` becomes 0.

3.2 Multiplication on $\mathbb{F}_{2^{13t}}$.

The commutative property, also known as the exchange law, states that the order of operands in a mathematical operation can be interchanged without affecting the result(e.g. a * b = b * a). In other words, for any given operation, the outcome remains the same regardless of the order in which the operands are arranged. Based on the commutative property, which holds for the `XOR` operation as well, we can utilize it to rearrange the order of existing operations. This allows us to interchange the operands involved in `XOR` operations without affecting the final result. Therefore, we performed operations by modifying the order of multiplication operations by utilizing these properties.

Figure 3 represents the original order of multiplication operations performed. The existing multiplication operations follow a sequential process for performing 16-bit multiplication, as illustrated in Figure 3 (a). Once the operations depicted

Algorithm 1 In Classic McEliece-348864, 16-bit value multiplication operation;(x0 : Result of Multiplication Operation, x1, x2 : Input of Multiplication Operation)

```
 1: mov x9, #64
 2: loop_i:
 3: mov x8, #64

 4: loop_j:
 5: ld1R {v0.4h}, [x1]
 6: ld1 {v1.4h}, [x2], #8

    //masking
 7: zip1.8h v0, v0, v3
 8: zip1.8h v1, v1, v3

    //Multiplication
 9: and.16b v8, v1, v4
10: mul.4s v6, v8, v0

11: shl.4s v5, v4, #1
12: and.16b v8, v1, v5
13: mul.4s v8, v0, v8
14: eor.16b v6, v6, v8

15: shl.4s v5, v4, #2
16: and.16b v8, v1, v5
17: mul.4s v8, v0, v8
18: eor.16b v6, v6, v8

    ⋮

19: shl.4s v5, v4, #10
20: and.16b v8, v1, v5
21: mul.4s v8, v0, v8
22: eor.16b v6, v6, v8

23: shl.4s v5, v4, 11
24: and.16b v8, v1, v5
25: mul.4s v8, v0, v8
26: eor.16b v6, v6, v8

    //Reduction
27: and.16b v9, v6, v14
28: sri.4s v10, v9, #9
29: eor.16b v6, v6, v10
30: sri.4s v10, v9, #12
31: eor.16b v6, v6, v10

32: and.16b v9, v6, v15
33: sri.4s v10, v9, #9
34: eor.16b v6, v6, v10
35: sri.4s v10, v9, #12
36: eor.16b v6, v6, v10

37: and.16b v6, v6, v11

    //unmasking
38: uzp1.8h v6, v6, v7

39: ld1.4h {v2}, [x0]
40: eor.16b v2, v2, v6

41: st1.4h {v2}, [x0], #8

42: add x8, x8, #-4
43: cbnz x8, loop_j

44: add x0, x0, #-126
45: add x2, x2, #-128
46: add x1, x1, #2

47: add x9, x9, #-1
48: cbnz x9, loop_i
```

in Figure 3 (a) are completed, the multiplication computations proceed in the order presented in Figure 3 (b).

Figure 4 shows the sequence of the proposed multiplication operation using four 16-bit loaded vector registers. To leverage the commutative property of the XOR operation, the paper adopts a modified approach as illustrated in Figure 4. The results of the multiplication operations, denoted as (a), (b), (c),

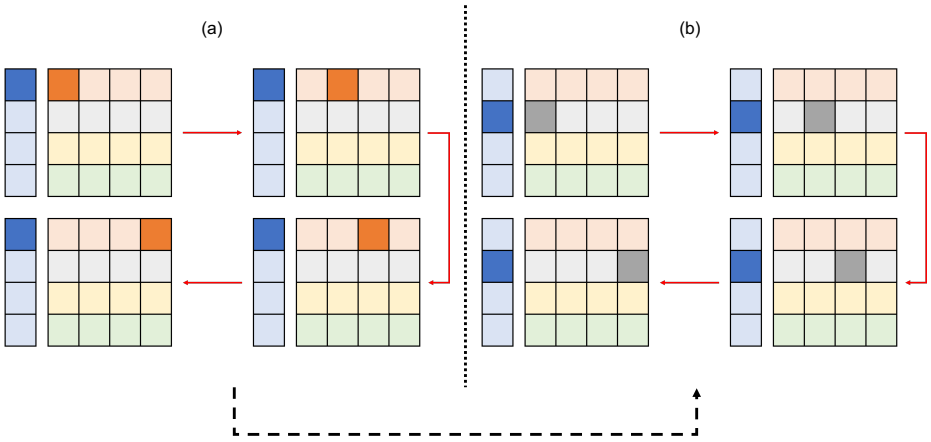

Fig. 3. Order of existing multiplication operations.

and (d) in Figure 4, are stored in each registers designated as temp. Since the `XOR` operation is commutative, the order of the values does not impact the final result, allowing for efficient parallel loading and processing. By adopting this strategy, the paper maximizes the utilization of parallel operations and takes advantage of the commutative property of `XOR` operation, resulting in improved efficiency in the multiplication process.

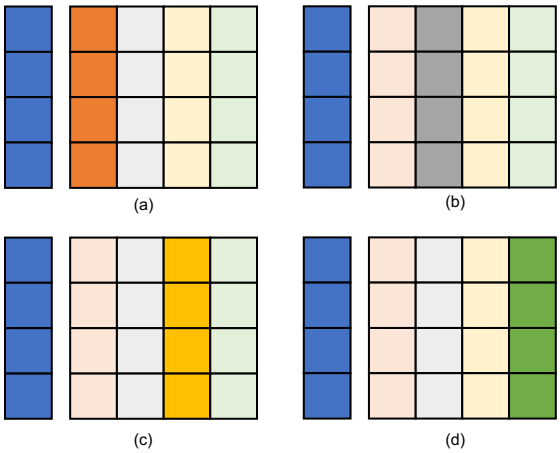

Fig. 4. Order of new multiplication operations.

Algorithm 2 is part of the optimization implementation code for multiplication on $\mathbb{F}_{2^{13^t}}$. Lines 4-7 perform operations corresponding to (a), (b), (c), and (d) in Figure 4 respectively. Lines 8-23 are operations that store the values calcu-

lated through Lines 4-7. Address values are directly calculated for each of them, and the address values are moved and stored. Lines 24-26 perform the operation by adding the direct address value appropriately, and the operation of Lines 4-25 is performed until the value of `x9` becomes 0. Lines 35-59 are stored using the `st1 v2.h[n]` because they need to store values for three 16-bit values(n: 0 to 2).

Algorithm 3 is one of the macros used by Algorithm 2. So, Algorithm 1 Therefore, multiplication operation is possible in the same way as Algorithm 1. However, Algorithm 3 performs a load for four 16 bits and performs one masking process over one whole. Because, the other multiplication is a 32-bit constant value, enter the value directly through Line 2 and use it for operation.

4 Evaluation

In this section, we evaluate the proposed implementation and reference C implementation (PQ-Clean project reference code in C language) [14]. Since the previous work did not conduct a separate performance evaluation of the multiplier, so it was not included in the performance evaluation [19].

The implementation were developed through the Xcode 14.3 framwork and carried out through the Xcode IDE. The implementation were evaluated on a MacBook Pro 13 with the Apple M1 chip that can be clocked up to 3.2 GHz.

Table 3. Evaluation results of multiplier on ARMv8 processors (Apple M1 chip); (cc : Clock Cycle).

Scheme		PQ-Clean	This Work
Classic McEliece 348864	ms	4,294	5,429
	cc	13,740	17,372
Classic McEliece 460896	ms	40,579	14,563
	cc	129,852	46,601
Classic McEliece 6699128	ms	71,185	25,343
	cc	227,792	81,097
Classic McEliece 6960119	ms	61,690	21,972
	cc	197,408	70,310
Classic McEliece 819212	ms	71,193	25,222
	cc	227,817	80,710

Therefore, we measured the operation time by repeating the 1,000,000 times and compiled using the compile option `-O2` (i.e.faster). Performance evaluation is given in Table 3. The performance of the implementation of Multiplier optimization using vector registers performed in Classic Mceliece 348864 is 0.79× times lower than [14]. The performance of the implementation of Multiplier optimization using vector registers performed in Classic Mceliece 46089 is 2.79× times higher than [14]. The performance of the implementation of Multiplier optimization using vector registers performed in Classic Mceliece 6699128 is 2.80×

Algorithm 2 In Classic McEliece-348864, 16-bit value multiplication on $\mathbb{F}_{2^{13}t}$ operation.

```
 1: mov x9, #15
 2: add x0, x0, #246

 3: loop:
 4: 4_byte_gf_mul_877
 5: 4_byte_gf_mul_2888
 6: 4_byte_gf_mul_1781
 7: 4_byte_gf_mul_373

 8: add x0, x0, #-110
 9: ld1.4h {v2}, [x0]
10: eor.16b v2, v2, v20
11: st1.4h {v2}, [x0]

12: add x0, x0, #-4
13: ld1.4h {v2}, [x0]
14: eor.16b v2, v2, v21
15: st1.4h {v2}, [x0]

16: add x0, x0, #-4
17: ld1.4h {v2}, [x0]
18: eor.16b v2, v2, v22
19: st1.4h {v2}, [x0]

20: add x0, x0, #-10
21: ld1.4h {v2}, [x0]
22: eor.16b v2, v2, v23
23: st1.4h v2, [x0]

24: add x0, x0, #120

25: add x9, x9, #-1
26: cbnz x9, loop


27: add x0, x0, #2

28: 4_byte_gf_mul_877
29: 4_byte_gf_mul_2888
30: 4_byte_gf_mul_1781
31: 4_byte_gf_mul_373

32: add x0, x0, #-110
33: ld1.4h {v2}, [x0]
34: eor.16b v2, v2, v20

35: st1 v2.h[0], [x0], #2
36: st1 v2.h[1], [x0], #2
37: st1 v2.h[2], [x0]
38: add x0, x0, #-4

39: add x0, x0, #-4
40: ld1.4h {v2}, [x0]
41: eor.16b v2, v2, v21

42: st1 v2.h[0], [x0], #2
43: st1 v2.h[1], [x0], #2
44: st1 v2.h[2], [x0]
45: add x0, x0, #-4

46: add x0, x0, #-4
47: ld1.4h {v2}, [x0]
48: eor.16b v2, v2, v22

49: st1 v2.h[0], [x0], #2
50: st1 v2.h[1], [x0], #2
51: st1 v2.h[2], [x0]
52: add x0, x0, #-4

53: add x0, x0, #-10
54: ld1.4h {v2}, [x0]
55: eor.16b v2, v2, v23

56: st1 v2.h[0], [x0], #2
57: st1 v2.h[1], [x0], #2
58: st1 v2.h[2], [x0]
59: add x0, x0, #-4
```

times higher than [14]. The performance of the implementation of Multiplier optimization using vector registers performed in Classic Mceliece 6960119 is 2.81× times higher than [14]. The performance of the implementation of Multiplier op-

Algorithm 3 In Classic McEliece-348864, multiplication $\mathbb{F}_{2^{13}}$ macro for 16-bit value multiplication for $\mathbb{F}_{2^{13^t}}$.

```
.macro 4_byte_gf_mul_877
 1: ld1 {v0.4h}, [x0]
 2: mov.8h v1, v12

    //masking
 3: zip1.8h v0, v0, v3

    //Multiplication
 4: and.16b v8, v1, v4
 5: mul.4s v6, v8, v0

 6: shl.4s v5, v4, #1
 7: and.16b v8, v1, v5
 8: mul.4s v8, v0, v8
 9: eor.16b v6, v6, v8

10: shl.4s v5, v4, #2
11: and.16b v8, v1, v5
12: mul.4s v8, v0, v8
13: eor.16b v6, v6, v8

    ⋮

14: shl.4s v5, v4, #10
15: and.16b v8, v1, v5
16: mul.4s v8, v0, v8
17: eor.16b v6, v6, v8

18: shl.4s v5, v4, 11
19: and.16b v8, v1, v5
20: mul.4s v8, v0, v8
21: eor.16b v6, v6, v8

22: and.16b v9, v6, v14
23: sri.4s v10, v9, #9
24: eor.16b v6, v6, v10
25: sri.4s v10, v9, #12
26: eor.16b v6, v6, v10

27: and.16b v9, v6, v15
28: sri.4s v10, v9, #9
29: eor.16b v6, v6, v10
30: sri.4s v10, v9, #12
31: eor.16b v6, v6, v10

32: and.16b v6, v6, v11

    //unmasking
33: uzp1.8h v20, v6, v7
.endm
```

timization using vector registers performed in Classic Mceliece 819212 is 2.82× times higher than [14].

According to the performance evaluation, the experimental results verified that the utilization of the proposed method led to a significant performance enhancement of up to 2.82× compared to the reference implementation

5 Conclusion

In this paper, a parallel multiplication implementation technique for Classic McEliece 348864 was introduced. The proposed method utilizes vector registers and vector instructions of the ARMv8 processor. The proposed parallel multiplication operation efficiently computes multiplication in a parallel manner by exploiting the commutative law and rearranging the order of internal operations. As a result, the proposed method significantly improves the efficiency of the multiplication operation in terms of speed and performance. The performance evalu-

ation results achieve that the proposed multiplication technique achieves a speed improvement of 2.82× compared to the PQ-Clean reference implementation.

6 Acknowledgements

This work was 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 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%)

References

1. "NIST PQC project." https://csrc.nist.gov/Projects/post-quantum-cryptography. Accessed : 2022-07-29.
2. R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Kyber algorithm specifications and supporting documentation," *NIST PQC Round*, vol. 2, no. 4, 2019.
3. L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-Dilithium: A lattice-based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–268, 2018.
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. P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over NTRU," *Submission to the NIST's post-quantum cryptography standardization process*, vol. 36, no. 5, 2018.
6. Y. Kim, J. Song, and S. C. Seo, "Accelerating falcon on ARMv8," *IEEE Access*, vol. 10, pp. 44446–44460, 2022.
7. H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1," *Cryptology ePrint Archive*, 2021.
8. P. Sanal, E. Karagoz, H. Seo, R. Azarderakhsh, and M. Mozaffari-Kermani, "Kyber on ARM64: Compact implementations of Kyber on 64-bit ARM Cortex-A processors," in *International Conference on Security and Privacy in Communication Systems*, pp. 424–440, Springer, 2021.
9. Y. Kim, J. Song, T.-Y. Youn, and S. C. Seo, "Crystals-Dilithium on ARMv8," *Security and Communication Networks*, vol. 2022, 2022.
10. S. Kölbl, "Putting wings on SPHINCS," in *International Conference on Post-Quantum Cryptography*, pp. 205–226, Springer, 2018.
11. H. Becker and M. J. Kannwischer, "Hybrid scalar/vector implementations of Keccak and SPHINCS+ on AArch64," *Cryptology ePrint Archive*, 2022.

12. D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, *et al.*, "Classic McEliece: conservative code-based cryptography," *NIST submissions*, 2017.
13. R. J. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv*, vol. 4244, pp. 114–116, 1978.
14. "PQClean project." Available online: https://github.com/PQClean/PQClean. Accessed: 2022-07-29.
15. "Armv8-A instruction set architecture." https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets. Accessed: 2023-06-07.
16. H. Kwon, H. Kim, M. Sim, W.-K. Lee, and H. Seo, "Look-up the rainbow: efficient table-based parallel implementation of rainbow signature on 64-bit ARMv8 processors," *Cryptology ePrint Archive*, 2021.
17. H. Kwon, H. Kim, M. Sim, S. Eum, M. Lee, W.-K. Lee, and H. Seo, "ARMingSword: Scabbard on ARM," in *International Conference on Information Security Applications*, pp. 237–250, Springer, 2022.
18. M.-S. Chen and T. Chou, "Classic McEliece on the ARM Cortex-M4," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 125–148, 2021.
19. M. Sim, S. Eum, H. Kwon, H. Kim, and H. Seo, "Optimized implementation of encapsulation and decapsulation of Classic McEliece on ARMv8," *Cryptology ePrint Archive*, 2022.