Optimized Implementation of Encapsulation and Decapsulation of Classic McEliece on ARMv8

No Author Given

No Institute Given

Abstract. We propose an efficient software implementation of Classic McEliece, a code-based cipher, on ARMv8. Classic McEliece can be divided into Key Generation, Encapsulation, and Decapsulation. Among them, we propose an optimal implementation for Encapsulation and Decapsulation. Optimized Encapsulation implementation utilizes vector registers to perform 16-byte parallel operations, and optimize using the specificity of the identity matrix. Decapsulation implemented efficient Multiplication and Inversion on $\mathbb{F}_{2^m}$. Compared with the previous results, Encapsulation showed a performance improvement of $7.64\times \sim 15.33\times$, and Decapsulation showed a performance improvement of $1.92\times \sim 2.24\times$.

Keywords: ARMv8 · Code based Cipher · Classic McEliece · NIST PQC · Parallel implementation · KEM

1 Introduction

Classic McEliece is the only code-based cipher in the NIST PQC finalists. The basic structure is based on the McEliece [1] 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 [2].

Recently, NIST PQC final algorithms was decided. `CRYSTALS-KYBER`[3] was selected as public key encryption and key establishment algorithms. And as digital signature algorithms, `CRYSTALS_DILITHIUM[4], FALCON[5], and SPHINCS+` [6] were selected. Although Classic McEliece is secure, it was not selected as a finalist due to the large size of the public key. However, NIST PQC is conducting the 4th candidate KEM, and it is judged that there is sufficient possibility because it is the only code-based cipher among candidate algorithms [7].

In [8], Classic McEliece optimization was implemented on ARM Cortex-M4. Due to the small RAM size of 192KB, the public key was stored in the ROM and implemented. In addition, performance improvement was shown by using Quick sort when generating errors in the encapsulation process and applying the bitslicing technique for matrix multiplication optimization. Decapsulation was also optimized for bitslicing and Radix-16 implementation. As a result, compared to `FrodoKEM`, which has similar security strength, performance improvement was 79 times faster in Encapsulation and 17 times faster in Decapsulation.

In this paper, we implement optimized Classic McEliece on ARMv8 processor. Our contributions are as follows:

1.1 Contribution

First Implementation of Classic McEliece on ARMv8 As far as we know, there is no Classic McEliece optimization implementation on ARMv8 yet. We present the first Classic McEliece optimization implementation of ARMv8.

Optimized Implementation of Encapsulation on ARMv8 We present optimized implementation of Encapsulation on ARMv8. Optimized the Encapsulation process by optimizing the computation when generating the syndrome. Most of the identity matrices in the syndrome generation process are zero, so we use the omitting possibility for optimization.

Optimized Implementation of Decapsulation on ARMv8 We present optimized implementation of Decapsulation on ARMv8. During decapsulation, Multiplication operations and Inversion operations are performed on extended binary finite-field $\mathbb{F}_{2^m}$, where m is 12 or 13. Multiplication operations and Inversion operations take a lot of time. Therefore, in this paper, Multiplication and Inversion operations on $\mathbb{F}_{2^m}$ operating in decapsulation are efficiently implemented using ARM instructions.

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 Tabel **??**.

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.

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	86	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

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 $w30$. Vector registers can be operated in parallel. 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). 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 [9].

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

asm	Operands	Description	Operation
ADD	Xd, Xn, Xm	Add	Xd ← Xn + Xm
AND	Xd, Xn, Xm(,shift #amount)	Bitwise AND(shifted register)	Xd ← Xn & (Xm <<#amount or Xm >>#amount)
SUB	Xd, Xn, imm(,shift)	Substact (immediate)	Xd ← Xn - Xm
EOR	Xd, Xn, Xm	Bitwise Exclusive OR	Xd ← Xn ⊕ Xm
EOR	Xd, Xn, Xm(,shifted #amount)	Bitwise Exclusive OR (shift register)	Xd ← Xn ⊕ (Xm <<#amount or Xm >>#amount)
ORR	Xd, Xn, Xm(,shift #amout)	Bitwise OR(shifted register)	Xd ← Xn | (Xm <<#amount or Xm >>#amount)
LD1	Vt.T, [Xn]	Load multiple single-element structures to one, two, three, or four registers	Vt ← [Xn]
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
MOV	Xd, Xn	Move(register)	Xd ← Xn
MOV	Xd, imm	Move(immediate)	Xd ← imm
RET	{Xn}	Return from subroutine	Return
LSL	Xd, Xn, #shift	Logical Shift Left(immediate)	Xd ← Xn <<#shift
LSR	Xd, Xn, #shift	Logical Shift Right(immediate)	Xd ← Xn >>#shift
MUL	Xd, Xn, Xm	Multiply	Xd ← Xn × Xm
BIC	Vd.T, Vn.T, Vm.T	Bitwise bit Clear(vector, register)	Clear
LDRB	Wt, [Xn/SP, (Wm$\|Xm), extendamount$]	Load Register Byte	Wt ← [Xn]
STRB	Wt, [Xn/SP, (Wm$\|Xm), extendamount$]	Store Register Byte	Wt ← [Xn]
CBNZ	Wt, Label	Compare and Branch on Nonzero	Go to Label

3 Proposed Method

3.1 Optimized Implementation of Encapsulation

Excluding the hash process from encapsulation, it can be divided into two processes: random vector generation and syndrome generation. In this paper, we optimize the syndrome generation process. This process is called the ENCODE process. The encoding process adds an identity matrix to the public key T to

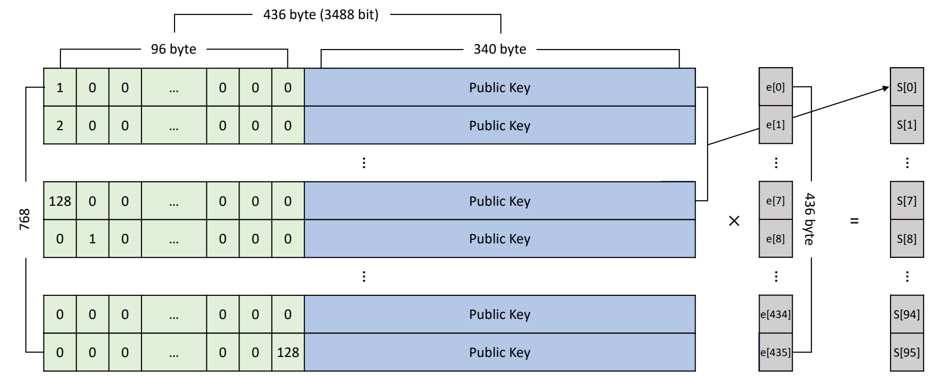

Fig. 1: `ENCODE` process of Encapsulation(In Classic McEliece-348864)

Algorithm 1 Assembly code implementing a macro that calculates only 1-byte of the identity matrix part(x3:error matrix address, x5:non-zero index in identity matrix, n:(1,2,4,8,16,32,64,128))

```
.macro row_front n
1: movi.16b v0, #0
2: add x3, x2, x5
3: ldrb w6, [x3]
4: and w6, w6, #\n
5: mov.b v0[0], w6
6: add x3, x2, #96
.endm
```

create a parity check matrix. Then the parity check matrix is multiplied by a randomly generated e matrix. `ENCODE` is defined as follow:

$$\text{Define } H = (I_{n-k}|T).$$
$$\text{Compute and return } C_0 = He \in \mathbb{F}_2^{n-k}$$

Figure 1 shows the `ENCODE` process. 8 Rows are each matrix multiplied by the error and then combined into 1 S. At this time, it can be seen that 96-byte corresponding to the identity matrix are 0 except for 1-byte. Of course, most of the values of the error matrix are also 0, but it is impossible to know which index has a 0 value. That is, the operation of the identity matrix part except for the public key may be omitted except for 1-byte.

Algorithm 1 shows the implementation of the identity matrix part. For non-zero values, the eight values (1,2,4,8,16,32,64,128) are used repeatedly. This repeated value is **n** used in line 4. Since the error only needs to be computed for non-zero values, the operation is performed by calling only the error values that have an index equal to the non-zero matrix index. The index of the non-zero matrix is stored in the **x5** register used in line 3. This index is incremented by 1 after 8 iterations (1,2,4,8,16,32,64,128). **x3** is the address of the error matrix. It

Algorithm 2 Syndrome 1-bit value operation macro(n:(1,2,4,8,16,32,64,128), i:Bit index when storing as bytes in S, v0: (public key × error))

```
.macro calculate_s_1bit n, i
 1: row_front \n
 2: row_process_21
 3: row_last

 4: mov.d v3[0], v0[1]
 5: eor.16b v0, v0, v3
 6: mov.s v3[0], v0[1]
 7: eor.16b v0, v0, v3
 8: mov.h v3[0], v0[1]
 9: eor.16b v0, v0, v3
10: mov.b v3[0], v0[1]
11: eor.16b v0, v0, v3
12: mov.s w9, v0[0]
13: and x9, x9, #0xff
14: lsr x10, x9, #4
15: eor x9, x9, x10
16: lsr x10, x9, #2
17: eor x9, x9, x10
18: lsr x10, x9, #1
19: eor x9, x9, x10
20: and x9, x9, #1
21: lsl x9, x9, #\i
22: orr x8, x8, x9
.endm
```

is computed by incrementing this value by **x5**(index) and then calling the value. Finally, correct the address of the error so that it is the same as the index where the public key value is stored.

In this paper, 16-byte parallel operation was performed using ARMv8 vector registers. Algorithm 2 is the assembly code to calculate the 1-bit syndrome. In lines 1-3, the public key and the error are computed in parallel and stored in the v0 register divided into 16 bytes. lines 4-11 collect the divided values into 1-byte. Finally, lines12-22 perform an operation to obtain a 1-bit value from the calculated value. In line 21, i means the bit position in the byte. If this is repeated 8 times, one syndrome byte is calculated.

3.2 Optimized Implementation of Decapsulation

Decapsulation uses a decoder of the constant-time Berlekamp-Massey (BM) algorithm. Inside the BM algorithm, Multiplication and Inversion are 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 and inversion on $\mathbb{F}_{2^m}$ used in decapsulation 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 $\mathbb{F}_2[x]/(x^{13}+x^4+x^3+x+1)$ [8].

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

Algorithm 3 is an optimization implementation code for multiplication on $\mathbb{F}_{2^{13}}$. As shown in Table 2, ARMv8 general-purpose registers can implement

Algorithm 3 Multiplication on $\mathbb{F}_{2^{13}}$($x0$, $x1$ is input register; $x13$, $x14$ is temporary registers).

Input: a($\mathbb{F}_{2^{13}}$), b($\mathbb{F}_{2^{13}}$)
Output: a*b($\mathbb{F}_{2^{13}}$)

```
 1: mov x10, x0
 2: mov x20, x1
 3: mov x3, #1

 4: and x14, x20, x3, lsl #1
 5: mul x13, x10, x14

 6: and x14, x20, x3, lsl #2
 7: mul x14, x10, x14
 8: eor x13, x13, x14
    ⋮
 9: and x14, x20, x3, lsl #12
10: mul x14, x10, x14
11: eor x13, x13, x14

12: and x14, x13, #0x1FF0000
13: eor x13, x13, x14, lsr #9
14: eor x13, x13, x14, lsr #10
15: eor x13, x13, x14, lsr #12
16: eor x13, x13, x14, lsr #13

17: and x14, x13, #0x000E000
18: eor x13, x13, x14, lsr #9
19: eor x13, x13, x14, lsr #10
20: eor x13, x13, x14, lsr #12
21: eor x13, x13, x14, lsr #13

22: lsl x14, x3, #13
23: sub x14, x14, #1
24: and x0, x13, x14
```

logical operations and shift operations using one instruction. The part corresponding to lines 4-11 of Algorithm 3 implements the Multiplication part. The omitted part proceeds as follows. If you look at line 6, you can see that the SHIFT operation is performed by 1 to the left before the and operation is performed. This SHIFT operation is a process of increasing the value by 1 up to $m-1$ and performing the operations from lines 6 to 8 in the same way. Finally, it can be seen that the $m-1$ value of 12 is applied in line 9. And if multiplication is finished through this process, there may be a result of multiplication out of $\mathbb{F}_{2^{13}}$. Therefore, after performing modular reduction, the operation corresponding to lines 12 to 24, on the result of multiplication, the result of the operation is returned. As such, it was possible to efficiently implement the corresponding part according to the characteristics of the ARM instructions.

Inversion on $\mathbb{F}_{2^{13}}$. Inversion operation on $\mathbb{F}_{2^{13}}$ can obtain by dividing $1(\mathbb{F}_{2^{13}})$ input value. Algorithm 4 represents the $(\mathbf{S}^2)^2$ operation on the input value as part of the inversion operation(in this case, the input value is referred to as $\mathbf{S}(\mathbb{F}_{2^{13}})$. Operation on the square of the input value $\mathbf{S}$ can be calculated by changing the OR operation, SHIFT operation, and AND operation. This can be implemented as lines 3-10 of Algorithm 4. The OR operation, which is one of the logical operations, was also efficiently implemented so that the OR operation proceeds after the SHIFT operation with one ORR instruction in the same way as the AND instruction. And, as in multiplication on $\mathbb{F}_{2^m}$, the result obtained

Algorithm 4 Partial operation process of inversion operation on $\mathbb{F}_{2^{13}}$($x0$ is input register; $x11$, $x13$, $x14$ is temporary registers).

Input: a($\mathbb{F}_{2^{13}}$)
Output: $(\mathrm{a}^2)^2(\mathrm{F}_{2^{13}})$

```
mov x10, x0
mov x12, #1

orr x11, x10, x10, lsl #24
and x10, x11, #0x000000FF000000FF
orr x11, x10, x10, lsl #12
and x10, x11, #0x000F000F000F000F
orr x11, x10, x10, lsl #6
and x10, x11, #0x0303030303030303
orr x11, x10, x10, lsl #3
and x10, x11, #0x1111111111111111

and x13, x10, #0x0001FF0000000000
eor x10, x10, x13, lsr #9
eor x10, x10, x13, lsr #10
eor x10, x10, x13, lsr #12
eor x10, x10, x13, lsr #13

and x13, x10, #0x000000FF80000000
eor x10, x10, x13, lsr #9
eor x10, x10, x13, lsr #10
eor x10, x10, x13, lsr #12
eor x10, x10, x13, lsr #13

and x13, x10, #0x000000007FC00000
eor x10, x10, x13, lsr #9
eor x10, x10, x13, lsr #10
eor x10, x10, x13, lsr #12
eor x10, x10, x13, lsr #13

and x13, x10, #0x00000000003FE000
eor x10, x10, x13, lsr #9
eor x10, x10, x13, lsr #10
eor x10, x10, x13, lsr #12
eor x10, x10, x13, lsr #13

lsl x14, x3, #13
sub x14, x14, #1
and x0, x13, x14
```

after the operation may be out of the range of $\mathbb{F}_{2^{13}}$, so the operation is completed by performing modular reduction on the result value.

4 Evaluation

Implementations were evaluated on a MacBook Pro 13 with the Apple M1 chip that can be clocked up to 3.2 GHz. Since ARMv8 does not have a Classic McEliece implementation, the performance is compared with the existing PQ-Clean project reference code [10]. Encapsulation(up to `ENCODE`) and Decapsulation(up to `DECODE`) was repeated by 1,000 times to measure the operation time("up to" means until the hashing process is performed). Performance evaluation is given in Table 3.

Encapsulation and Decapsulation of our implementation Classic Mceliece 348864 are 14.11× and 2.24× higher than [10], respectively. Encapsulation and Decapsulation of our implementation Classic Mceliece 460896 are 15.33× and 1.93× higher than [10], respectively. Encapsulation and Decapsulation of our implementation Classic Mceliece 6688128 are 14.75× and 1.92× higher than [10], respectively. Encapsulation and Decapsulation of our implementation Classic Mceliece 6960119 are 7.64× and 2.04× higher than [10], respectively. Finally, En-

Table 3: Evaluation result on ARMv8 processors(Apple M1) in terms of execution timing (i.e. clock cycles)

Algorithm	Encapsulation(up to `ENCODE`)		Decapsulation(up to `DECODE`)	
	[10]	our work	[10]	our work
mceliece 348864	555,789.625	39,512.469	16,148,425.313	7,204,017.813
mceliece 460896	1,236,170.406	80,643.812	32,543,014.063	16,820,844.063
mceliece 6688128	2,299,348.187	155,855.156	62,416,464.375	32,428,698.438
mceliece 6960119	2,636,024.437	345,173.812	63,814,351.563	31,352,953.750
mceliece 8192128	2,808,967.125	195,299.187	77,021,110.625	39,569,996.875

capsulation and Decapsulation of our implementation Classic Mceliece 8192128 are 14.38× and 1.95× higher than [10], respectively.

5 Conclusion

In this paper, we implemented the optimization of Classic McEliece Encapsulation and Decapsulation on ARMv8. In Encapsulation, sufficient performance improvement was achieved only through optimization of syndrome generation. In addition, in Decapsulation, a sufficient performance improvement result was obtained through optimization of multiplication and inversion operation. As a result, Encapsulation showed a performance improvement of $7.64\times \sim 15.33\times$, and Decapsulation showed a performance improvement of $1.92\times \sim 2.24\times$. we hope that this work will be helpful to implement Classic McEliece.

References

1. R. J. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv*, vol. 4244, pp. 114–116, 1978.
2. 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.
3. 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.
4. 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.
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. 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.

7. "Nist pqc project." https://csrc.nist.gov/Projects/post-quantum-cryptography. Accessed : 2022-07-29.
8. 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.
9. "Armv8-a instruction set architecture." https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets. Accessed: 2022-07-29.
10. "PQClean project." Available online: https://github.com/PQClean/PQClean. Accessed: 2022-07-29.