ARMing-sword: Scabbard on ARM*

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

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

Abstract. Scabbard, one of the Post-quantum Key Encapsulation Mechanisms (KEM), is a improved version of Saber that Lattice-based Key Encapsulation Mechanism. Scabbard has three schemes, called Florete, Espada, and Sable. Florete is a Ring-LWR-based KEM that effectively reuses the hardware architecture module used in Saber. Espada is a Module-LWR-based KEM that can be parallelized, requires very little memory, and is advantageous for operating in a resource-constrained environment. Finally, Sable adjusted the parameters to reduce the standard deviation of errors occurring in the Saber. In this paper, we propose ARMing-sword that optimized implementation of Scabbard on ARM processor. For the efficient implementation, a parallel operation technique using vector registers and vector instructions of the ARM processor is used. We focused on optimizing the multiplier, which takes majority execution time for Scabbard computation, and propose a Direct Mapping and Sliding Window methods for accumulating computation results. ARMing-sword has a performance difference of up-to 6.34× in the multiplier and a performance difference of up-to 2.17× in the encryption algorithm to which the optimization technique is applied.

Keywords: Post-quantum Cryptography · Saber · Scabbard · Software implementation · 64-bit ARMv8 processors.

1 Introduction

Quantum computers can compute huge data at once through qubit that based on the quantum mechanics principles [1]. Therefore, the computation power of

* 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 the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2020R1F1A1048478, 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, 25%).

quantum computers is stronger than classical computers. With quantum computers, Shor and Grover algorithms utilized this characteristic [2]. So the advent of quantum computers, public key cryptographic systems and symmetric key cryptographic algorithms are threatened [3].

Therefore, the National Institute of Standards and Technology (NIST) held a post-quantum cryptography standardization competition to create a safety online environment under the quantum computer era [4]. This competition, which started in 2016, reached Round 3 in 2020, and in the Public-Key Encryption (PKE) category, Classic McEliece [5], CRYSTALS-Kyber [6], NTRU [7], and Saber [8] were selected as final algorithms [9].

Among them, Saber is a lattice-based cipher based on the Module ring learning with rounding (Module-LWR) problem. It has advantages that short key length and fast computation. Due to these advantages, Saber guarantees even under a resource-constrained environment. Afterwards, the Scabbard, which improved the internal structure of Saber, was proposed. Scabbard utilizes parallel operators provided on a specific platform and is designed to operate appropriately in a resource-constrained environment like Saber. In this paper, we propose ARMing-sword that optimized implementation of Scabbard on ARMv8 processor. The main contributions of this paper is as follows.

1.1 Contributions

- **Step-by-step optimization of multiplier.** In this paper, we focused on high-speed implementation of multiplier, which is heavily used in Scabbard operations. Multiplier of the Scabbard consists of three steps (i.e. Evaluation, Multiplication, and Interpolation). In the proposed method, evaluation and multiplication are implemented efficiently by using parallel operators. In the implementation, we reduce the number of iterative operations by utilizing vector registers and vector instructions of ARM processor. Finally we reduce the computational load. The target processor is Apple M1 processor (@3.2GHz). As a unit, we use the average of the clock cycles where we run each algorithm a million (for multiplier) or ten thousand (for cipher) times. In the case of Evaluation it took up to 137.6 (Single), 329.6 (3-way), and 92.8 (4-way) clock cycles, respectively. In total, the multiplier takes 8,736 (Espada) and 425.6 (Florete, Sable) clock cycles, respectively. Performance improvements of multiplier are 3.35× and 1.17× then Scabbard multiplier. When we applied proposed multiplier to ARMing-sword algorithm, the result of measuring the operation time of Keygen+Encap+Decap, ARMing-sword Florete takes 203.8 clock cycles, Espada shows 720.4 clock cycles, and Sable needs 247.5 clock cycles. The case with the best performance improvement is Espada Decapsulation, which shows a performance improvement by 2.17×.
- **Optimized implementation for each scheme.** Scabbard has three schemes, including Florete, Espada, and Sable. Each has a different internal structure. Florete and Sable have similar structures, but Espada has a completely different form. Therefore, different approaches are applied to each algorithm.

– **First implementation of Scabbard on ARMv8 processors.** Saber, the previous implementation of Scabbard, has an optimized implementation on ARMv8, and the implementation is optimized by implementing the Toom-cook multiplication algorithm used inside in parallel. Scabbard is an improved algorithm of Saber. However, the same implementation techniques cannot be applied due to the different internal structure. In this paper, we propose ARMing-sword, which is the optimized implementation of Scabbard, and focused on the optimized implementation of all schemes of Scabbard.

1.2 Organization of the paper

The organization of the paper is as follows. Section 2 shows the backgrounds of Scabbard algorithm and target processor ARMv8. In Section 3, we introduce ARMing-sword that optimized implementation of Scabbard on ARMv8. Detailed optimization techniques and implementation methods are described. In Section 4, we present performance evaluation of the previous Scabbard and the proposed ARMing-sword. Finally, Section 5 concludes this paper.

2 Backgrounds

2.1 Scabbard: a suite of post-quantum key-encapsulation mechanisms

Scabbard was proposed in 2021 as an improved key-encapsulation mechanism (KEM) of Saber. Scabbard is a lattice-based cryptography that uses ring learning with rounding the same as the original cipher, Saber. Scabbard has three schemes, and they are called Florete, Espada, and Sable, respectively [10].

Florete. Scabbard Florete focused on reusing the hardware architecture and software modules developed for Saber to ensure efficient KEM. In lattice-based cryptography, such as Scabbard, the part that generates the largest computational load is polynomial multiplication, which is necessary because of the binomial distribution of lattice-based cryptography. In Florete, polynomial multiplication based on an asymptotically faster Number Theoretic Transform (NTT) cannot be used because the moduli have already been selected. In order to apply NTT technique to implementation, it should use a larger NTT friendly prime. Instead, Florete used a generic Toom-Cook polynomial multiplication [11].

To efficiently implement Scabbard Florete, quotient ring $\mathbb{R}_q^n$ was changed to $\mathbb{Z}_q[x]/(x^{768} - x^{384} + 1)$. Before multiplying the two polynomials a and $b \in \mathbb{R}_q^n$, Toom-Cook 3-way evaluation and b is applied. Through this, the polynomials of length 256 are divided into 5 polynomials. After that, polynomial multiplication of 256×256 is performed, and then $c = a \times b \in R_q^n$ is calculated through Toom-Cook 3-way interpolation. Since the computational load of Toom-Cook 3-way evaluation and interpolation is small, the time to perform multiplication of five 256×256 polynomials is similar to the time to calculate one 768×768 polynomial

multiplication. Table 1 shows that Florete has fewer multiplications than Saber and works effectively in pseudo-random number generation.

Table 1: Size of pseudo random number and number of 256×256 polynomial multiplications in Saber and Scabbard-Florete.

Name	Pseudo-random number (Bytes)	256×256 multiplications		
		KeyGen	Encap	Decap
Florete	1152	5	10	15
Saber	4512	9	12	15

However, polynomial coefficients need to be within 16-bit for efficient multiplication operation on FPGA or IoT devices. Therefore, while applying Toom-Cook interpolation, it is sometimes necessary to divide field elements by $r = 2d \times m$ with $gcd(m, 2) = 1$. Last, Florete targets NIST security level 3 and does not support level 5, but can be provided as 7 polynomial partitions of length 256 using Toom-Cook 4-way.

Espada. Scabbard Espada aimed to have a system that has a small memory footprint and can be parallelized in a resource-constraint environment. In this way, Espada maintains the performance of other platforms within a lightweight environment and aims to achieve 128-bit or higher post-quantum security.

Espada further reduced the length of the polynomial (n). Parallel operations can be performed, effectively. In particular, as n decreases, the faster operation is possible and requires a smaller area. In this way, multiple instances of $n \times n$ polynomial multipliers can be executed in batches. Therefore, using this method is similar to applying Toom-Cook evaluation to decompose larger multiplications and then using small polynomial multipliers in parallel. Considering this point, Espada uses n as an optimal value of 64.

Sable. Scabbard Sable is a lattice-based KEM manufactured in another form of Saber. Previous Saber used rounding error with continuous uniform distribution rather than discrete uniform distribution. At this time, the parameters are adjusted in order to prevent a decrease in security due to different standard deviations of the error. First, Sable sample the secret value from the Centered Binomial Distribution (CBD) where $\eta = 1$. In this case, the secret coefficient is -1, 0, or 1. When $\eta = 1$, we can store the secret value using only 2 bits per coefficient, reducing the memory requirements for the Sable.

In Saber, polynomial multiplication is performed in the form of $a \times s$, where a is randomly determined from R_q^n or R_p^n, and s is sampled according to the distribution β_η. Saber uses η as 3, 4, and 5 according to each scheme. Sable keeps η because it is advantageous to keep η and change p and q. It is easy to use this distribution for efficient implementation because secret values have a specific distribution.

2.2 Target processor: Apple M1 processor

The Apple M1 processor is a processor belonging to the ARMv8 family and was first shown in 2020. Same as ARMv8 processor, it has 64-bit general purpose register and 128-bit vector register capable of parallel operation. One vector register can store up to 128-bit. However, according to the arrangement specifier, the data inside the register is treated as the size specified by the specifier. If $16B$ specifier is used, the vector register treats the internal value as 16 bytes. The overall arrangement specifier can be found in Table 2 [12]. However, it is specified in the instruction, not in the arrangement specifier register. This allows the value in the register to be treated as a different unit each time an instruction is executed.

Table 2: List of arrangement specifiers specifying register packing of vector registers.

Data type	Unit	Specifier	Number of data
Byte	8-bit	8B	8
		16B	16
Half-word	16-bit	4H	4
		8H	8
Single-word	32-bit	2S	2
		4S	4
Double-word	64-bit	1D	1
		2D	2

2.3 Optimized implementations of Post-Quantum Cryptography on ARM processors

Kwon et al. [12] implemented FrodoKEM on Apple A10X Fusion processor, one of ARMv8 processors. The proposed method parallelizes matrix multiplication. The multiplication is performed at a high speed, and the AES module used for random number generation is operated at a high speed using an AES accelerator. As a result of performance comparison, it was shown that it operates up to 10.22× faster than the reference FrodoKEM-640.

Song et al. [13] implemented Saber, the predecessor of Scabbard, on ARM Cortex-A72, an ARMv8 processor. The proposed method operates the multiplication algorithm Toom-Cook at high speed using vector instructions and registers. The proposed method improves the operation speed of the multiplier up-to 5× faster than the existing multiplier of Saber.

Kim et al. [14] implemented Falcon, an NTRU based cryptography. The implementation targets the Jetson Xavier CPU, which is widely used in autonomous driving environments. Jetson Xavier CPU is ARMv8.2 and uses the same instruction set as ARMv8, but it has added functions such as half-precision floating-point processing. The proposed method implements FFT-based multiplication

and NTT-based multiplication, separately, and utilizes the NEON engine. As a result of performance comparison, the proposed method showed a performance improvement of up-to 65.4% in the verification process compared to the existing Falcon-512.

Kwon et al. [15] optimally implemented the rainbow signature algorithm, a multivariate based cryptography, on Apple M1 processor. The main optimization technique is to change polynomial multiplication to multiplication based on look-up table. In addition, by using vector instructions and registers, the table look-up is executed in parallel. The multiplication is completed quickly. The multiplication performance of the proposed method was improved by a maximum of 167.2× compared to the existing method. When the multiplier was applied to the Rainbow signature, there was a performance improvement of up-to 51.6×.

3 Proposed techniques

The multiplier used in Scabbard consists of three stages (Evaluation, Multiplication, and Interpolation). The proposed method is to improve the performance by applying the optimal implementation to each step.

3.1 Instruction set

ARMv8 processor provides various instructions, and these instructions divided into general instructions and vector instructions. In this paper, vector instructions are mainly used together with some general instructions. Table 3 lists instructions used to implement the ARMing-sword, and it can easily check the usage and effect of each instruction.

Table 3: List of ARMv8 instructions used for proposed implementation in alphabetical order [16]; `Xd, Vd`: destination register (general, vector), `Xn, Xm, Vn, Vm`: source register (general, general, vector, vector), `Xt, Vt`: transferred register (general, vector), `T`: Arrangement specifier, `i`: Index.

Vector instructions			
asm	Operands	Description	Operation
ADD	Vd.T, Vn.T, Vm.T	Add vector	Vd ← Vn + Vm
LD1	Vt.T, [Xn]	Load multiple single-element structures	Vt ← [Xn]
LD1R	Vt.T, [Xn]	Load one single-element structures and Replicate	Vt ← [Xn][i]
SHL	Vd.T, Vn.T #shift	Shift Left	Vd ← Vn ¡¡ #shift
ST1	Vt.T, [Xn]	Store multiple single-element structures	[Xn] ← Vt
SUB	Vd.T, Vn.T, Vm.T	Subtract vector	Vd ← Vn - Vm
General instructions			
asm	Operands	Description	Operation
ADD	Xd, Xn, Xm	Add a register values	Xd ← Xn + Xm
CBNZ	Xt, (Label)	Compare and Branch on Nonzero	Go to Label
MOV	Xd, #imm	Move immediate maximum 64-bit	Xd ← #imm

3.2 Evaluation: Direct Mapping

In the Evaluation stage, the variable is set to a 256-bit length and a set number of values is secured. At this time, the composition of evaluation is different for each scheme. Espada divides the initial input value for each section and creates it, and Florete and Sable create a value by performing a specific operation on the initial input value.

In the Evaluation process, all the values of each array are traversed and stored in different variables. It takes a lot of time. In the case of Espada, Evaluation is performed twice in succession, and this requires more time than Florete and Sable. Vector registers and instructions are used to optimally implement Evaluation step. Since Scabbard operates in 16-bit units, a maximum of 8 variables are stored using $8h$ arrangement specifier in single vector register. This can reduce the number of loops by 8 times.

For the efficient operation, the structure of Evaluation is changed and it performed once instead of twice. Evaluation stage of the Scabbard Espada can be confirmed in the form of pseudo code of Algorithm 1. It shows the for loop is run twice. As such, Evaluation of Espada has a larger computational load compared to Florete and Sable. Visualizing this one shown in Figure 1(a). Figure 1(a) shows only an example for $AW0$, but $AW1$ and $AW2$ go through the same process.

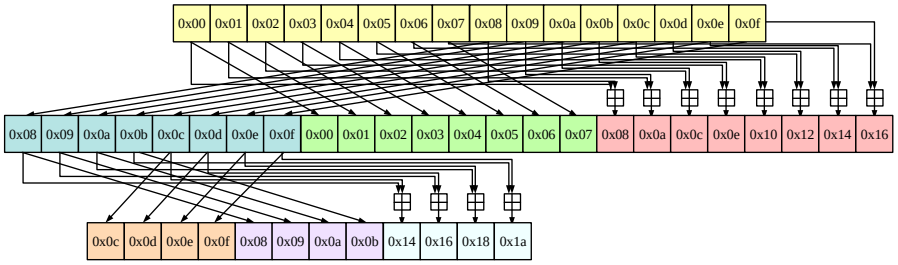
Algorithm 1 Pseudo-code of Scabbard Espada Evaluation step.

Input: N length of 16-bit array $A1$, middle length $M = N/2$, output length $L = N/4$.
Output: L length of 16-bit array $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$.

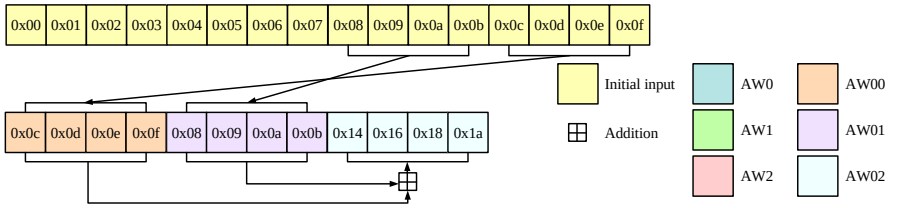
$i \leftarrow 0$
for i to M **do**
  $AW0[i] \leftarrow A1[i + L]$
  $AW2[i] \leftarrow A1[i]$
  $AW1[i] \leftarrow A1[i] + A1[i + L]$
  $i \leftarrow i + 1$
end for
$j \leftarrow 0$
for j to L **do**
  $AW00[j] \leftarrow AW0[j + M]$
  $AW01[j] \leftarrow AW0[j] + AW0[j+M]$
  $AW02[j] \leftarrow AW0[j]$
  $AW10[j] \leftarrow AW1[j + M]$
  $AW11[j] \leftarrow AW1[j] + AW1[j+M]$
  $AW12[j] \leftarrow AW1[j]$
  $AW20[j] \leftarrow AW2[j + M]$
  $AW21[j] \leftarrow AW2[j] + AW2[j+M]$
  $AW22[j] \leftarrow AW2[j]$
  $j \leftarrow j + 1$
end for
return $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$

The proposed method is the **Direct Mapping** that directly stores the initial input value in the output value. To implement Direct Mapping, we first figure out saving rules of each array value. Values stored in $AW00$ and $AW02$ in Algorithm 1 are as follows.

$$AW00[0] \leftarrow AW0[0+M] = A1[0+L+M],\ AW02[0] \leftarrow AW0[0] = A1[0+L]$$



(a) $AW0$ is passed to generate $AW00 \sim 02$. All values are moved one at a time and iterates through the array until it is traversed.



(b) Direct Mapping technique of ARMing-sword. $AW00 \sim 02$ are created directly without intermediate values. $AW01$ is created by the sum of $AW00$ and $AW02$. The number of iterations is reduced by shifting multiple values at once.

Fig. 1: Evaluation step of Scabbard.

$$AW00[1] \leftarrow AW0[1+M] = A1[1+L+M],\ AW02[1] \leftarrow AW0[1] = A1[1+L]...$$
$$AW00[j] \leftarrow AW0[j+M] = A1[j+L+M],\ AW02[j] \leftarrow AW0[j] = A1[j+L]$$

It can be seen that $AW02$ gets the value from the middle value of $A1$, which is the initial input value, and $AW00$ gets the value from the 3/4 point of $A1$. In the case of $AW01$, it can be calculated by adding the values of $AW00$ and $AW02$. Similarly, in the case of $AW20$ and $AW22$, it is similar to $AW00 \sim 02$, except that the starting value of $A1$ is imported. Therefore, the overall form is the same as $AW00 \sim 02$. Finally, the following rules can be found for the values of $AW10$ and $AW12$.

$$AW10[0] \leftarrow AW1[0+M] = A1[0+M] + A1[0+M+L],\ AW12[0] \leftarrow AW1[0] = A1[0] + A1[0+L]$$
$$AW10[1] \leftarrow AW1[1+M] = A1[1+M] + A1[1+M+L],\ AW12[1] \leftarrow AW1[1] = A1[1] + A1[1+L]...$$
$$AW10[j] \leftarrow AW1[j+M] = A1[j+M] + A1[j+M+L],\ AW12[j] \leftarrow AW1[j] = A1[j] + A1[j+L]$$

$AW10$ is derived from the sum of the intermediate values of the initial input value $A1$, and $AW12$ consists of the sum of the starting values of $A1$. Therefore, it can be seen that all values from $AW00$ to $AW22$ can be calculated directly from $A1$ without going through $AW0$, $AW1$, and $AW2$. However, the values of $A1$ requested by $AW10$, $AW11$, and $AW12$ have already been stored by calculating $AW00 \sim 02$, $AW20 \sim 22$. Therefore, $AW01 \sim 12$ can be calculated by adding $AW00 \sim 02$ and $AW20 \sim 22$. The Direct Mapping technique calculates the

final value directly without going through an intermediate step like this. The Evaluation step of ARMing-sword to which Direct Mapping is applied can be expressed as Algorithm 2. If it is expressed visually, it can be expressed as Figure 1(b), and it shows how the values of $A1$ correspond from $AW00$ to $AW02$ at once. Figure 1(b) In Figure 1(b), only $AW00 \sim 02$ case is expressed, but the rest of the values are calculated in the same way.

Algorithm 2 Pseudo-code of Direct Mapping technique for ARMing-sword.

Input: N length of 16-bit array $A1$, middle length $M = N/2$, output length $L = N/4$.
Output: L length of 16-bit array $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$.

1: $i \leftarrow 0$
2: **for** i to L **do**
3: $AW00[i] \leftarrow A1[i + L + M]$
4: $AW02[i] \leftarrow A1[i + L]$
5: $AW01[i] \leftarrow AW00[i] + AW02[i]$
6: $AW20[i] \leftarrow A1[i + M]$
7: $AW22[i] \leftarrow A1[i]$
8: $AW21[i] \leftarrow AW20[i] + AW21[i]$
9: $AW10[i] \leftarrow AW00[i] + AW20[i]$
10: $AW12[i] \leftarrow AW01[i] + AW21[i]$
11: $AW11[i] \leftarrow AW02[i] + AW22[i]$
12: $i \leftarrow i + 1$
13: **end for**
14: **return** $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$

Direct Mapping changes the order of operations to minimize memory access. In particular, in the case of $AW10 \sim 12$, it can be calculated by adding $AW00 \sim 02$ and $AW20 \sim 22$, respectively. This can shorten the time to access the memory to get a value from $A1$, and it is an effective operation because the number of addition instructions can be reduced only for $AW11$. Algorithm 3 describe the Direct Mapping in source codes. In line 1-4, $A1$ values are loaded from address of $A1$ to vector registers. At that time, the value is loaded into the vector registers corresponding to the output address. In line 5-6, $AW01$ values are calculated and line 7-8, $AW21$ values are calculated. In line 9-14, $AW10$, $AW12$, and $AW11$ values are calculated. In line 15-19, each value is stored to output address. From Algorithm 3, $AW00$, $AW02$, $AW20$, and $AW22$ don't seem to have any operations. Since these values only need to get the specific value of $A1$, the calculation is actually completed as soon as values from lines 1-4 are loaded.

In the case of Florete and Sable, it is implemented according to Toom-Cook 3-way or 4-way. Unlike Evaluation of Espada, Evaluation of Florete and Sable includes a number of processing of input values. Therefore, we focus on minimizing the number of iterations using vector instructions and registers. In some parts, Direct Mapping can be applied. Algorithm 4 is a part of the source code implementing Toom-Cook 3-way Evaluation. Toom-Cook 4-way can be implemented by increasing the scale of 3-way version. In line 1-3, initial input values are loaded from address of $A[0]$, $A[256]$, and $A[512]$. In line 4-5, $p0$ and $p4$ are

Algorithm 3 Source codes for Direct Mapping technique.

Input: $A1$ address = x0, AW address = x1
Output: $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$ values
1: LD1.8h {v16, v17}, [x0], #32
2: LD1.8h {v12, v13}, [x0], #32
3: LD1.8h {v4, v5}, [x0], #32
4: LD1.8h {v0, v1}, [x0], #32
5: ADD.8h v2, v0, v4
6: ADD.8h v3, v1, v5
7: ADD.8h v14, v12, v16
8: ADD.8h v15, v13, v17
9: ADD.8h v6, v0, v12
10: ADD.8h v7, v1, v13
11: ADD.8h v10, v4, v16
12: ADD.8h v11, v5, v17
13: ADD.8h v8, v2, v14
14: ADD.8h v9, v3, v15
15: ST1.8h {v0, v1, v2, v3}, [x1], #64
16: ST1.8h {v4, v5, v6, v7}, [x1], #64
17: ST1.8h {v8, v9, v10, v11}, [x1], #64
18: ST1.8h {v12, v13, v14, v15}, [x1], #64
19: ST1.8h {v16, v17}, [x1], #32

calculated using the Direct Mapping technique. In line 6-14, 15-19, and 20-32, $p1$, $p2$, and $p3$ values are calculated, respectively. Since Algorithm 4 is part of the whole operation process, it completes all values through iteration.

Algorithm 4 Source codes for Toom-Cook 3-way Evaluation.

Input: $A[0]$ address = x0, $p0-4$ address = x1-5, $A[256]$ address = x6, $A[512]$ address = x7.
Output: $p0$, $p1$, $p2$, $p3$, $p4$ values.
1: LD1.8h {v0, v1, v2, v3}, [x0], #64
2: LD1.8h {v4, v5, v6, v7}, [x6], #64
3: LD1.8h {v8, v9, v10, v11}, [x7], #64
4: **ST1.8h {v0, v1, v2, v3}, [x1], #64**
5: **ST1.8h {v8, v9, v10, v11}, [x5], #64**
6: ADD.8h v12, v0, v8
7: ADD.8h v13, v1, v9
8: ADD.8h v14, v2, v10
9: ADD.8h v15, v3, v11
10: ADD.8h v16, v12, v4
11: ADD.8h v17, v13, v5
12: ADD.8h v18, v14, v6
13: ADD.8h v19, v15, v7
14: ST1.8h {v16, v17, v18, v19}, [x2], #64
15: SUB.8h v16, v12, v4
16: SUB.8h v17, v13, v5
17: SUB.8h v18, v14, v6
18: SUB.8h v19, v15, v7
19: ST1.8h {v16, v17, v18, v19}, [x3], #64
20: ADD.8h v16, v16, v8
21: ADD.8h v17, v17, v9
22: ADD.8h v18, v18, v10
23: ADD.8h v19, v19, v11
24: SHL.8h v16, v16, #1
25: SHL.8h v17, v17, #1
26: SHL.8h v18, v18, #1
27: SHL.8h v19, v19, #1
28: SUB.8h v16, v16, v0
29: SUB.8h v17, v17, v1
30: SUB.8h v18, v18, v2
31: SUB.8h v19, v19, v3
32: ST1.8h {v16, v17, v18, v19}, [x4], #64

3.3 Multiplication: Sliding Window

After Evaluation, the multiplication is carried out. The multiplication of Espada is implemented as a nested loop like Algorithm 5 and has a structure in which the calculation result value is accumulated.

However, vector instructions cannot be implemented because the address pointer moves in 32-byte or 64-byte units. For this, the **Sliding Window** technique is utilized. The Sliding Window technique is a technique that processes operations based on the current pointer position. This requires manual adjustment of the pointer. In ARM assembly, address pointers are stored in general registers. They can be adjusted through general instructions. Algorithm 6 is the

Algorithm 5 Pseudo-code of Scabbard Espada Multiplication stage.

Input: Evaluation result array A, input array B.
Output: output result C.
1: $i \leftarrow 0$
2: $j \leftarrow 0$
3: **for** i to 32 **do**
4: **for** j to 63 **do**
5: $C[0][0][i] \leftarrow A[i]$ * $B[0][0][j]$
6: $C[0][1][i] \leftarrow A[i]$ * $B[0][1][j]$
7: $C[0][2][i] \leftarrow A[i]$ * $B[0][2][j]$
8: $C[1][0][i] \leftarrow A[i]$ * $B[1][0][j]$
9: $C[1][1][i] \leftarrow A[i]$ * $B[1][1][j]$
10: $C[1][2][i] \leftarrow A[i]$ * $B[1][2][j]$
11: $C[2][0][i] \leftarrow A[i]$ * $B[2][0][j]$
12: $C[2][1][i] \leftarrow A[i]$ * $B[2][1][j]$
13: $C[2][2][i] \leftarrow A[i]$ * $B[2][2][j]$
14: $j \leftarrow j + 1$
15: **end for**
16: $i \leftarrow i + 1$
17: **end for**C

source code used for Sliding Window implementation. In line 1-2, B and A values are loaded. In line 3-4, multiplication between A and B is performed. In line 5-7, the multiplication result is accumulated and stored to C. There is no register to store the address of C in the source code. This is because it shares an address with A. In line 8, the address pointer is moved to next value. The operation can be completed by repeating this until the last value of A and B. Florete and Sable exist in five different forms of multiplication. Therefore, 5 different types of multipliers to which Sliding Window is applied are implemented.

Algorithm 6 Source codes of Sliding Window technique for ARMing-sword.

Input: B address = x1, A address = x2.
Output: multiplication result C.
1: LD1.8h {v18, v19}, [x1], #32
2: LD1.8h {v21, v22}, [x2]
3: MUL v23.8h, v18.8h, v0.h[0]
4: MUL v24.8h, v19.8h, v0.h[0]
5: ADD.8h v21, v21, v23
6: ADD.8h v22, v22, v24
7: ST1.8h {v21, v22}, [x2]
8: ADD x2, x2, #2

4 Evaluation

This Section shows evaluation result of proposed ARMing-sword and previous Scabbard implementation[1]. The target processor is Apple M1 processor (@3.2GHz) which used for Apple product line (e.g. MacBook and iPad). The implementation is carried out through the Xcode IDE, and compiled with compile option `-O3` (i.e. fastest). The performance is measured through the average time of 1,000,000 iterations (i.e. 10,000 times for multiplier and ARMing-sword algorithm in average timing). The unit is clock cycles.

[1] https://github.com/josebmera/scabbard

Table 4: Performance comparison results of multiplier (Unit: clock cycles)

Algorithm	[10]	**This work**
Evaluation single	272	**137.6**
Evaluation 3-way	1,740.8	**329.6**
Evaluation 4-way	588.8	**92.8**
Multiplier Espada	29,286.4	**8736**
Multiplier Florete/Sable	496	**425.6**

First, performance of the multiplier is compared. Table 4 shows performance comparison results. Evaluation single is only for Espada, and Evaluation single of Scabbard takes 272 clock cycles. Our proposed ARMing-sword Evaluation single needs 137.6 clock cycles. It has 1.97× better performance than Scabbard. Evaluation 3-way and 4-way is used for Florete and Sable, Scabbard show 1,740.8 and 588.8 clock cycles, respectively. On the other hand, ARMing-sword Evaluation 3-way and 4-way take 329.6 and 92.8 clock cycles respectively. It is 5.28× and 6.34 × better performance than Scabbard. Finally, as a result of operating multiplier, Multiplier of Scabbard-Espada takes 29,286.4 clock cycles. However, ARMing-sword-Espada multipliers only takes 8,736 clock cycles, and it 3.35 × performance improvement. However, multipliers of Florete and Sable have almost the same performance. This is because optimized implementation of Evaluation is well done, but the structure of Multiplication Florete/Sable is not friendly to ARM processor compared to Espada.

Second, we compare the performance of the previous work and the proposed method. The overall performance comparison results are shown in Table 5. The proposed method has better performance than the previous work. In particular, the performance improvement in Espada is outstanding. Among them, decapsulation shows the best performance improvement with 2.17×.

Table 5: Performance comparison results of previous implementation and proposed implementation (Unit: $\times 10^4$clock cycles).

Scheme	Algorithm	[10]	**This work**
Sable	KeyGen	80.8	**75.7**
	Encapsulation	89.8	**84.1**
	Decapsulation	93.4	**88.4**
	All	263.2	**247.5**
Espada	KeyGen	475.6	**230.7**
	Encapsulation	505.4	**239.7**
	Decapsulation	521.2	**241.7**
	All	11497.4	**720.4**
Florete	KeyGen	53.2	**50.8**
	Encapsulation	72.5	**72.6**
	Decapsulation	86.0	**78.9**
	All	222.1	**203.8**

5 Conclusion

In this paper, we propose ARMing-sword that optimized version of Scabbard algorithm. ARMing-sword uses Direct Mapping and Sliding Window techniques for multiplication algorithms. As a result of the performance comparison, multiplier shows performance improvement by 6.34× in Evaluation 4-way. In the case of ARMing-sword implementation by applying a multiplier to Scabbard, decapsulation shows most performance improvement that 2.17× than previous works.

References

1. D. Deutsch, "Quantum theory, the Church–Turing principle and the universal quantum computer," *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 400, no. 1818, pp. 97–117, 1985.
2. M. N. Leuenberger and D. Loss, "Quantum computing in molecular magnets," *Nature*, vol. 410, no. 6830, pp. 789–793, 2001.
3. Z. Kirsch and M. Chow, "Quantum computing: The risk to existing encryption methods," *Retrieved from URL: http://www. cs. tufts. edu/comp/116/archive/fall2015/zkir sch. pdf*, 2015.
4. D. Moody, G. Alagic, D. C. Apon, D. A. Cooper, Q. H. Dang, J. M. Kelsey, Y.-K. Liu, C. A. Miller, R. C. Peralta, R. A. Perlner, *et al.*, "Status report on the second round of the nist post-quantum cryptography standardization process," 2020.
5. 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," 2017.
6. 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.
7. C. Chen, O. Danba, J. Hoffstein, A. Hülsing, J. Rijneveld, J. M. Schanck, P. Schwabe, W. Whyte, and Z. Zhang, "NTRU algorithm specifications and supporting documentation," in *Second PQC Standardization Conference*, 2019.
8. J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren, "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," in *International Conference on Cryptology in Africa*, pp. 282–305, Springer, 2018.
9. D. Auten and T. Gamage, "Impact of resource-constrained networks on the performance of NIST round-3 PQC candidates," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 768–773, IEEE, 2021.
10. J. M. B. Mera, A. Karmakar, S. Kundu, and I. Verbauwhede, "Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 474–509, 2021.
11. D. E. Knuth, *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.
12. H. Kwon, K. Jang, H. Kim, H. Kim, M. Sim, S. Eum, W.-K. Lee, and H. Seo, "ARMed Frodo," in *International Conference on Information Security Applications*, pp. 206–217, Springer, 2021.
13. J. Song, Y. Kim, and S. Seo, "Optimization study of Toom-Cook algorithm in NIST PQC SABER utilizing ARM/NEON processor," *Journal of The Korea Institute of Information Security & Cryptology*, vol. 31, no. 3, 2021.

14. Y. Kim, J. Song, and S. C. Seo, "Accelerating Falcon on ARMv8," *IEEE Access*, 2022.
15. 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.
16. "ARMv8-A instruction set architecture." https://documentation-service.arm.com/static/613a2c38674a052ae36ca307. Accessed: 2019-06-26.