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]

[1]Hansung University, [2]Gachon University

22. 08. 24.

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

Introduction

Backgrounds

ARMing-sword

Evaluation

Conclusion

Introduction

- With the development of quantum computers,
  classic cryptography algorithms are threatened. (e.g. RSA, ECC)
- In preparation for the quantum computer era,
  the Post-quantum cryptography (PQC) became necessary.
- The **Scabbard** is one of the PQC.
  - Originated from Saber.
  - Round 3 candidate of Public-key Encryption.
- In this paper, we proposed **ARMing-sword.**
  - Optimized implementation of Scabbard.

Introduction

- Our main contributions

1. Step-by-step optimization of multiplier.
   - Scabbard multiplier consists of three steps: Evaluation, Multiplication, Interpolation.
   - We focused on **Evaluation** and **Multiplication** steps.
   - Implemented with **revised internal structure** and **using parallel operations.**
2. Customized optimized implementation for each scheme.
   - Scabbard has three schemes: Florete, Espada, Sable.
   - Each scheme has different structure, so **our approaches also applied in various way s.**
3. The first implementation of Scabbard on the ARMv8 processors.
   - It might be **helpful to following researchers.**

Background: Saber and Scabbard

- Lattice-based cryptography.
- Round 3 candidate of NIST PQC standardization for KEM.
  - Other candidates: Classic McEliece, **CRYSTALS-Kyber**, NTRU.
  - Failed to advance to Round 4.

Common parameters: $q = 2^{13}, \ p = 2^{10}, \ f(x) = x^{256} + 1$

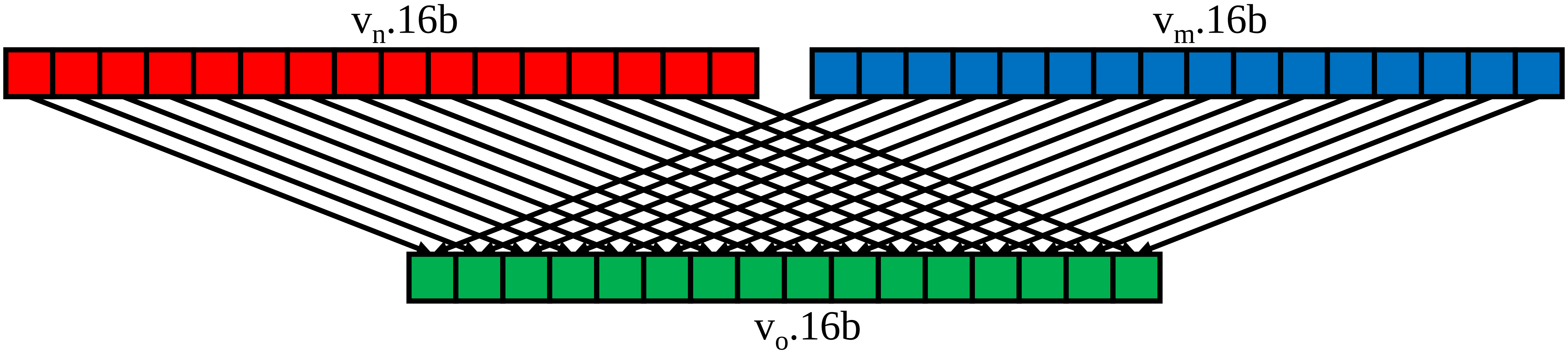
Category	Failure Probability	Classical Core-SVP	Quantum Core-SVP	Pk(Byte)	Sk(Byte)	Ct(Byte)
LightSaber-KEM: $l = 2, \ T = 2^3, \ \mu = 5$						
1	2^{-120}	2^{118}	2^{107}	672	1568 (992)	736
Saber-KEM: $l = 3, \ T = 2^4, \ \mu = 4$						
2	2^{-136}	2^{189}	2^{172}	992	2304 (1440)	1088
FireSaber-KEM: $l = 4, \ T = 2^6, \ \mu = 3$						
3	2^{-165}	2^{260}	2^{236}	1312	3040 (1760)	1472

Background: Saber and Scabbard

- Scabbard is originated from Saber and has three schemes.
- Florete
  - **Reusing the hardware architecture and software modules** developed for Saber.
  - Quotient ring $\mathbb{R}_q^n$ changed to $\mathbb{Z}_q[x]/(x^{768} - x^{384} + 1)$.
  - 768×768 polynomial multiplication → five of 256×256 polynomial multiplicatio ns.
- Espada
  - **Small memory** footprint.
  - Can be **parallelized in a resource-constraint environment**.
- Sable
  - Sample the secret value from the **Centered Binomial Distribution**.

Background: Apple M1 processor

- One of the ARMv8 processor.
- 64-bit general purpose registers, **128-bit vector registers.**
- Vector registers can be calculated values in parallel.
  - Internal values treated in different units depending on the **arrangement specifier.**
  - 8-bit(8b, 16b), 16-bit(4h, 8h), 32-bit(2s, 4s), 64-bit(1d, 2d).
  - Arrangement specifiers belong to instructions.
  - **Operational units can be changed by specifiers of instructions.**



ARMing-sword

- We propose ARMing-sword.
  - Optimized implementation of **Scabbard on ARM** processors.
- Two kinds of optimization techniques.
  - Evaluation step: **Direct Mapping.**
  - Multiplication step: **Sliding Window**.

Saber

Scabbard

ARMing-sword

ARMing-sword

- In the Evaluation stage
  - All of input arrays are traversed and stored in different variables.
  - N length of 16-bit array → **N/4 length of 16-bit arrays of 9.**
  - It takes very long time.
- **Is there any additional method besides parallelization?**

Algorithm 1 Pseudo-code of Scabbard Espada Evaluation step.

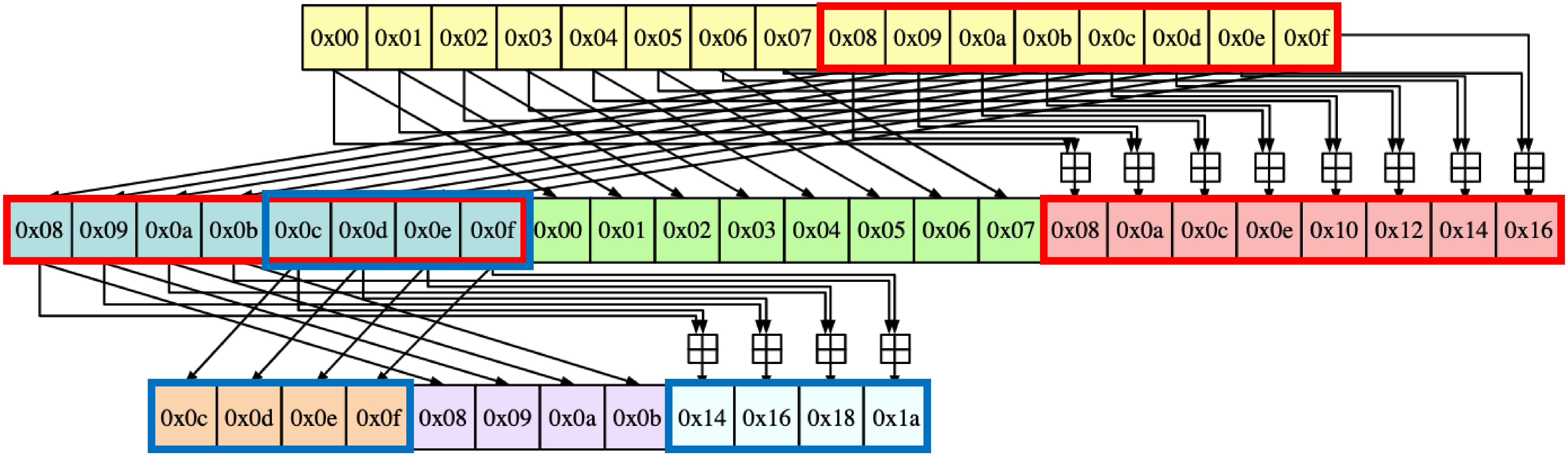
Input: N length of 16-bit array $A1$, middle length $M = N/2$, output length $L = N/4$.

Output: L length of 16-bit array $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$.

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

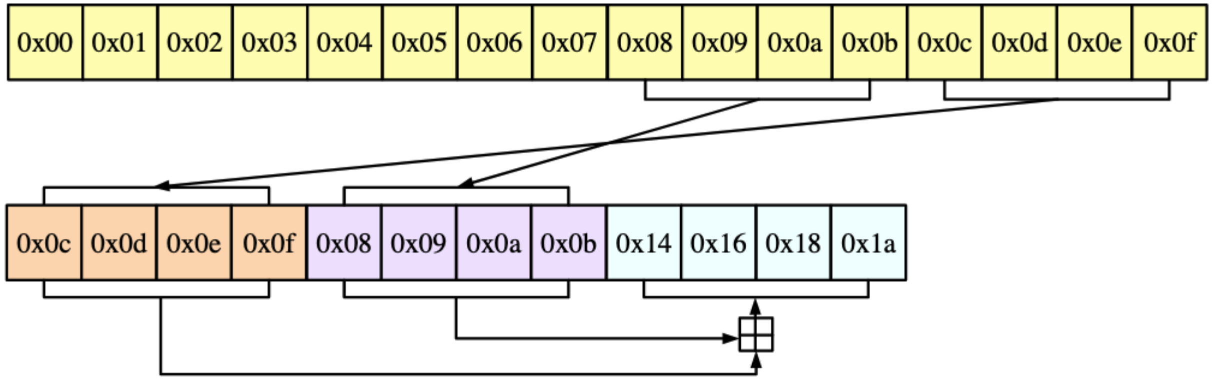
ARMing-sword

- Evaluation consists of three stage.
  - N length of 1 → N/2 length of 3 → N/4 length of 9
  - At each step, **half of the array is moved to the same way.**
  - Each variable **moves in two places**.



ARMing-sword

- Revised structure of Evaluation step. → **Direct Mapping.**
  - **To generate result values directly.**
  - **Move multiple variables at once** using vector instructions.
- It can be reduced number of variables move and calculation.



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 $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$.

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

ARMing-sword

- Implementation codes for Direct Mapping technique.
  - Written by ARM assembly.

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
```

ARMing-sword

- Toom-Cook 3-way with Direct Mapping.
  - Multiplication source codes are based on Scabbard.
  - Toom-Cook 4-way also can be implemented same method.

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.

```
LD1.8h {v0, v1, v2, v3}, [x0], #64
LD1.8h {v4, v5, v6, v7}, [x6], #64
LD1.8h {v8, v9, v10, v11}, [x7], #64
ST1.8h {v0, v1, v2, v3}, [x1], #64
ST1.8h {v8, v9, v10, v11}, [x5], #64
ADD.8h v12, v0, v8
ADD.8h v13, v1, v9
ADD.8h v14, v2, v10
ADD.8h v15, v3, v11
ADD.8h v16, v12, v4
ADD.8h v17, v13, v5
ADD.8h v18, v14, v6
ADD.8h v19, v15, v7
ST1.8h {v16, v17, v18, v19}, [x2], #64
SUB.8h v16, v12, v4
SUB.8h v17, v13, v5
SUB.8h v18, v14, v6
SUB.8h v19, v15, v7
ST1.8h {v16, v17, v18, v19}, [x3], #64
ADD.8h v16, v16, v8
ADD.8h v17, v17, v9
ADD.8h v18, v18, v10
ADD.8h v19, v19, v11
SHL.8h v16, v16, #1
SHL.8h v17, v17, #1
SHL.8h v18, v18, #1
SHL.8h v19, v19, #1
SUB.8h v16, v16, v0
SUB.8h v17, v17, v1
SUB.8h v18, v18, v2
SUB.8h v19, v19, v3
ST1.8h {v16, v17, v18, v19}, [x4], #64
```

ARMing-sword

- In the multiplication stage.
  - It implemented as a **nested loop.**
  - The indexes are moved to 16-bit units.
  - **However vector instructions only can be moved pointer to 32-byte or 64-byte uni ts.**

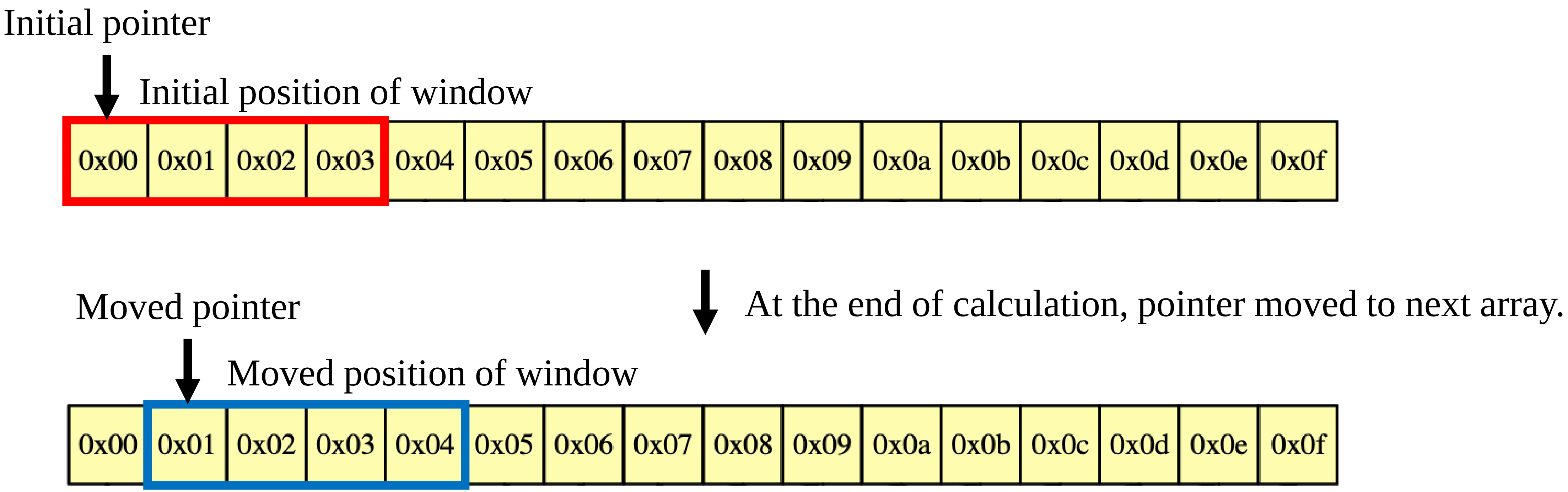
Algorithm 5 Pseudo-code of Scabbard Espada Multiplication stage.

Input: Evaluation result array A, input array B.
Output: output result C.

```
i ← 0
j ← 0
for i to 32 do
    for j to 63 do
        C[0][0][i] ← A[i] * B[0][0][j]
        C[0][1][i] ← A[i] * B[0][1][j]
        C[0][2][i] ← A[i] * B[0][2][j]
        C[1][0][i] ← A[i] * B[1][0][j]
        C[1][1][i] ← A[i] * B[1][1][j]
        C[1][2][i] ← A[i] * B[1][2][j]
        C[2][0][i] ← A[i] * B[2][0][j]
        C[2][1][i] ← A[i] * B[2][1][j]
        C[2][2][i] ← A[i] * B[2][2][j]
        j ← j + 1
    end for
    i ← i + 1
end forC
```

ARMing-sword

- We using general register to pointer. → **Sliding Window.**
  - Calculate by calling the values at the point currently pointed to by the pointer.
  - When the calculation is finished, the value is stored in the memory of that pointer.
  - Move a pointer to calculate the next value.
  - Values of multiplication step can be accumulated in this way.



ARMing-sword

- Implementation codes for Sliding Window technique.
  - Written by ARM assembly.

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

Input: B address = x1, A address = x2.
Output: multiplication result C.

```
LD1.8h {v18, v19}, [x1], #32
LD1.8h {v21, v22}, [x2]
MUL v23.8h, v18.8h, v0.h[0]
MUL v24.8h, v19.8h, v0.h[0]
ADD.8h v21, v21, v23
ADD.8h v22, v22, v24
ST1.8h {v21, v22}, [x2]
ADD x2, x2, #2
```

Evaluation

- Target processor: **Apple M1** processor (@3.2GHz).
- Framework: Xcode IDE.
- Language: Objective-C, C, **ARM assembly**.
- Compare with previous reference C implementation.
  - Scabbard open source C code used.
  - Compiled with –O3 option (fastest).
- Two kinds of result.
  - **Multiplier: average time of 1,000,000 iterations.**
  - **Scabbard and ARMing-sword: average time of 10,000 iterations.**

Evaluation

- Multiplier results (Unit: clock cycles).
  - Performance gap 6.34 when best case.
  - **Multiplier of ARMing-sword Espada shows effective then Scabbard.**
  - Multiplier of ARMing-sword Florete and Sable has similar performances with Scabbard.

Algorithms	Scabbard	ARMing-sword	Improvement
Evaluation single	272	**137.6**	1.97×
Evaluation 3-way	1740.8	**329.6**	5.28×
Evaluation 4-way	588.8	**92.8**	**6.34×**
Multiplier for Espada	29,286.4	**8736**	3.35×
Multiplier for Florete/Sable	496	**425.6**	1.16×

Evaluation

- Result of cryptography algorithm (Unit: $\times 10^4$clock cycles).
  - In almost all cases, ARMing-sword has better performance than Scabbard.
  - **In case of Espada shows best improvement.**

Scheme	Algorithm	Scabbard	ARMing-sword
Sable	KeyGeneration	80.8	**75.7**
	Encapsulation	89.8	**84.1**
	Decapsulation	93.4	**88.4**
	All	263.2	**247.5**
Espada	KeyGeneration	475.6	**230.7**
	Encapsulation	505.4	**239.7**
	Decapsulation	521.2	**241.7**
	All	1497.4	**720.4**
Florete	KeyGeneration	53.2	**50.8**
	Encapsulation	72.5	**72.6**
	Decapsulation	86.0	**78.9**
	All	222.1	**203.8**

Conclusion

- We proposed **ARMing-sword** cryptography algorithms.
  - Optimized implementation version of Scabbard on ARM processors.
- ARMing-sword using two kinds of **optimization techniques for multiplier.**
  - Direct Mapping
  - Sliding Window
- **The evaluation results show better than Scabbard performances.**
  - In case of multiplier, 6.34× improvement is best case.
  - ARMing-sword Espada decapsulation shows most performance improvement.
  - Presented implementation is effective to ARM processors.

Q & A