Look-up the Rainbow: Efficient Table-based Parallel Implementation of Rainbow Signature on 64-bit ARMv8 Processors

No Author Given

No Institute Given

Abstract. Rainbow signature is one of the finalist in National Institute of Standards and Technology (NIST) standardization. It is also the only signature candidate that is designed based on multivariate quadratic hard problem. Rainbow signature is known to have very small signature size compared to other post-quantum candidates. In this paper, we propose an efficient implementation technique to improve performance of Rainbow signature schemes. A parallel polynomial-multiplication on a 64-bit ARMv8 processor was proposed, wherein a look-up table was created by pre-calculating the 4×4 multiplication results. This technique was developed based on the observation that the existing implementation of Rainbow's polynomial-multiplication relies on the Karatsuba algorithm. It is not optimal due to the divide and conquer steps involved, whereby operations on $\mathbb{F}_{16}$ are divided into many small sub-fields of $\mathbb{F}_4$ and $\mathbb{F}_2$. Further investigations reveal that when the polynomial-multiplication in Rainbow signature is operated on $\mathbb{F}_{16}$, its operand is in 4-bit. Since the maximum combinations of a 4×4 multiplication is only 256, we constructed a 256-byte look-up table. According to the 4-bit constant, only 16-byte is loaded from the table at one time. The time-consuming multiplication is replaced by performing the table look-up. In addition, it calculates up-to 16 result values per register using characteristics of vector registers available on 64-bit ARMv8 processor. With the proposed fast polynomial-multiplication technique, we implemented the optimized Rainbow III and V. These two parameter sets are performed on $\mathbb{F}_{256}$, but they use sub-field $\mathbb{F}_{16}$ in the multiplication process. Therefore, the sub-field multiplication can be replaced with the proposed table look-up technique, which in turn omitted a significant number of operations. We have carried out the experiments on the Apple M1 processor, which shows up to 167.2× and 51.6× better performance enhancement at multiplier, and Rainbow signatures, respectively, compared to the previous implementation.

Keywords: Post-quantum Cryptography · Rainbow Signature · 64-bit ARMv8 Processors · Software Implementation.

1 Introduction

Due to the advancement of quantum computers and its related computing algorithms, existing cryptographic schemes are seriously in threats. Some widely

used public key cryptographic schemes (e.g. RSA and ECC) can be easily compromised by quantum computers. To prepare for these threats, the National Institute of Standards and Technology (NIST) is holding a post-quantum cryptography standardization competition to select cryptographic schemes that can be safely used even in the quantum computer era. Rainbow signature is the only multivariate-based public key signature in the finalists (Round 3) of this standardization competition. Rainbow signature has a disadvantage that its implementation performance is heavily influenced by the polynomial-multiplication, which is very slow execution timing.

In this paper, we propose a technique using a look-up table, in which the result of 4×4 multiplication is pre-computed. This look-up table is used to speed up the polynomial-multiplication of Rainbow schemes. In addition, we also propose to use vector registers to perform parallel table look-up operations to speed-up the memory load and store operations. With the proposed fast polynomial-multiplication technique, optimized Rainbow I, Rainbow III and Rainbow V implementations are presented in this paper. Its performance is compared with the previous reference implementation.

The rest part of paper can be written as follows; In Section 2, it shows related works of Rainbow post-quantum cryptography, target 64-bit ARMv8 processor, and previous optimal implementation of Post-Quantum Cryptography on target processor. In Section 3, the proposed method will be described. In Section 4, the performance comparison is carried out. In Section 5, it draws the conclusion of this paper and presents future works.

1.1 Contributions

- **Efficient implementation of polynomial-multiplication operations for Rainbow signature schemes.**
  The proposed method calculates the polynomial-multiplication result for 4×4 cases, and stores it as a look-up table. There are only 256 combinations in this situation. The look-up table only consumes 256-byte. This technique is applicable to all parameter sets in Rainbow schemes, but Rainbow III and Rainbow V requires an additional of 16 bytes to store the look-up table in total 272 bytes. This technique greatly reduces the computational time in polynomial-multiplication compared to the previous implementation using the Karatsuba algorithm. Evaluation results show better performance by up-to 167.2× and 51.6× for multiplier and Rainbow signatures, respectively, compared to the previous implementation.
- **Optimal parallel-implementation on the latest 64-bit ARMv8 processors.** The Apple M1 is one of the state-of-art latest ARM processor. In this paper, we exploited vector registers and vector instructions in Apple M1 processor for the parallel-implementation. The vector register can store up-to 16-byte, and multiple results can be computed simultaneously with vector instructions and combinations. The proposed parallel technique can compute results faster than the conventional multiplication operations, because multiple values can be loaded from the look-up table at once.

- **Efficient instructions usage and implementation.** The proposed implementation uses only 18 instructions to increase the readability of the code and facilitate maintenance. Each of the instructions perform 4-bit masking, table address allocation, table look-up, and accumulation of result values.
- **First optimal-implementation for Rainbow III and Rainbow V on 64-bit ARMv8 processors.** Most of the Rainbow implementations targets only Rainbow I. Rainbow III and Rainbow V are usually omitted. This paper implemented not only Rainbow I, but also Rainbow III and V. Since the implementation technique of Rainbow I cannot be equally applied to Rainbow III and Rainbow V, some modifications are required and it can be applied to all parameter sets. However, the look-up table used in Rainbow I can also be used in Rainbow III and Rainbow V for the multiplication on $\mathbb{F}_{16}$. Rainbow III and Rainbow V require an additional 16-byte table for high 4-bit squaring operations. Therefore, the total table size of Rainbow III and Rainbow V is 272-byte.

2 Related Works

2.1 Post Quantum Cryptography: Rainbow Signatures

Rainbow is a polynomial signature scheme proposed by Jintai Ding and Dieter Schmidt in 2004, which is a based on multivariate quadratic problem [1]. Parameters of Rainbow algorithm are shown in Table 1. Rainbow signature adopted the Unbalanced Oil and Vinegar (UOV) structure that requires small size of memory but this provides fast operation speed for public key algorithms [2].

The multivariate quadratic problem is a mathematical hard problem to find the answer X of P when there are m quadratic equations with n variables as shown in the following equation.

$$P^{(m)}(x_1, ..., x_n) = \sum_{i,j=1}^{n} p_{i,j}^{(m)} x_i x_j + \sum_{i=1}^{n} p_i^{(m)} x_i + p_0^{(m)}$$

Multivariate cryptography is developed based on a system of multivariate quadratic polynomials over a finite field K. The security of multivariate systems relies on the multivariate quadratic (MQ) problem, which is to find a solution of multivariate system for field K. Rainbow signature is based on MQ-problem, and consist of Key Scheduling, Signature, and Verification. Overall scheme of Rainbow signature is shown in Figure 1.

- **Key Scheduling.** The public key is given as $P = T \circ F \circ S : K^n \to K^m$, and the private key consists of T, F and S. For multivariate signature schemes, we require $n \geq m$, which ensures that every message has a signature.
- **Signature Generation.** To generate a signature for a message (or its hash value) $d \in K^m$, we need to recursively calculate the following expression.

$$w = T^{-1}(d) \in K^m, y = F^{-1}(w) \in K^n, z = S^{-1}(y)$$

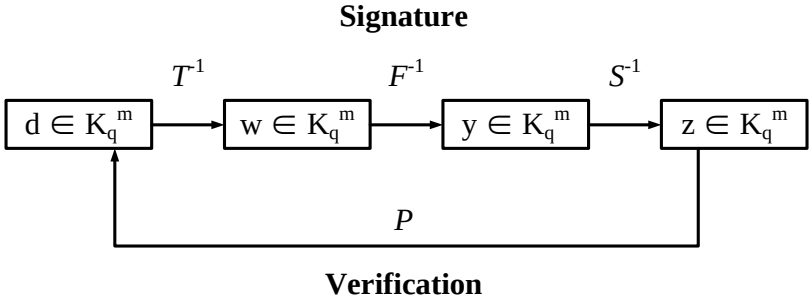

Fig. 1. Overall scheme of Rainbow signature.

Table 1. Length of key and signature for Rainbow schemes. Internal brackets indicate the private key size when linear maps S and T are generated through 256-bit seed.

Type	Security	Parameters	Public key size (KB)	Private key size (KB)	Signature size (bit)
Standard	I	$(GF(16), 36, 32, 32)$	157.8	101.2	528
	III	$(GF(256), 68, 32, 48)$	861.4	611.3	1,312
	V	$(GF(256), 96, 36, 64)$	1,885.4	1,375.7	1,632
CZ	I	$(GF(16), 36, 32, 32)$	58.8	101.2 (99.0)	528
	III	$(GF(256), 68, 32, 48)$	258.4	611.3 (603.0)	1,312
	V	$(GF(256), 96, 36, 64)$	523.5	1,375.7 (1,361.8)	1,696

$z \in K^n$ means the signature of message d, and $F^{-1}(w)$ means finding one (of possibly many) pre-images of w under the central map F.

– **Signature verification.** To check the authenticity of the signature $z \in K^n$, the verifier simply computes $d' = P(z)$. If the signature verification result is equal to the message d, the signature is accepted, otherwise it is rejected.

Rainbow signature has alternative version that called CZ-Rainbow (i.e. Circumzenithal Rainbow or Cyclic Rainbow), it has reversed key generation process then Classic Rainbow signature. CZ-Rainbow is inspired by Petzoldt et al. that proposed cyclic structure inside the Rainbow public key [3]. CZ-Rainbow does not use cyclic matrices at all, and it reduces public key size about 70% then Classic Rainbow. However, it takes more time spend because inefficient computation during Key Scheduling and Verification.

Another alternative version is Compressed Rainbow. Compressed Rainbow has the same internal structure as CZ-Rainbow, but it don't store the computed central map. Therefore, Compressed Rainbow needs a lot of time for Signature, but it drastically reduced size of public key.

2.2 Target Processor: 64-bit ARMv8 Architecture

The ARM processor is one of lightweight processors that provides high performance in resource-constrained Internet of Things (IoT) environment, such as

sensor nodes. In this paper, we targeted the latest ARMv8-A (ARMv8) architecture, which can be separated into two versions; 32-bit AArch32 (A32) and 64-bit AArch64 (A64). Among them, A64 has 64-bit general registers and 128-bit vector registers. Vector registers provide parallel operation with arrangement specifier, which determines the packing unit of data. For example, `16b` arrangement means that the internal data of the specified register is treated as 16-bytes [4]. Apple M1 processor is one of the latest of ARM processors. The M1 is a processor designed by Apple, intended for use in devices such as Macs and iPads. M1 processor is a kind of System on Chip (SoC) that has multi-core CPU, GPU, DSP and Neural engine on a single chip. It is produced on a 5nm process and consists of about 16 billion transistors [5].

2.3 Previous Implementations of Post Quantum Cryptography on ARM Processors

Chou et al. implemented the Rainbow post-quantum cryptography on Cortex-M4, which is a family of 32-bit ARMv7 [6]. Chou et al. proposed fast constant-time bit-slice $\mathbb{F}_{16}$ multiplication allowing multiplication of 32 field elements in 32 clock cycles. As two $\mathbb{F}_{16}$ elements fit into one byte, eight $\mathbb{F}_{16}$ elements can be mounted in one 32-bit register. They proposed a significantly faster $\mathbb{F}_{16}$ multiplication routines that run in constant time, in which each field element is implemented in bit-slice form on four separate registers, holding a total of 32 elements.

Kim et al. proposed a renewed polynomial-multiplication technique that can reduce number of XOR operations by more 13.7% than previous work Chou et at. on Cortex-M4 environments [7]. In addition, Chou et al. used table look-up for inverse operation, but Kim et al. achieved this by using a 4×4 matrix inverse method.

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

Nguyen et al. implemented an optimized implementation of three lattice-based NIST post-quantum cryptography Key Encapsulation Mechanisms Finalists (CRYSTALS-Kyber, NTRU, Saber) on ARMv8 environment [9]. This optimized implementation involves an explicit call to the NEON instruction (vector instruction), and the results obtained show a significant speedup compared to the implementation written only in C language. Nguyen et al. stated that NTT and NTRU for CRYSTALS-Kyber and Toom-Cook for Saber are the optimal algorithms for implementing polynomial multiplication in ARMv8 using the NEON instruction through the experimental results.

Streit et al. implemented a New Hope post-quantum key exchange on ARMv8-A [10]. New Hope is based on Ring-LWE (Ring-Learning With Errors) problem,

so researchers fully vectorized all ring operations. Proposed method has three alternative modular reduction, to makes the Number Theoretic Transform (NTT) in parallel-way. The results shows vectorized NTT takes 18,909 clock cycles on ARM Cortex-A53 processor when using a 16-bit unsigned integer.

3 Proposed Method

In this section, we describe the parallel polynomial-multiplication with look-up table technique. The implementation is targeting the 64-bit ARMv8 processor.

3.1 Instruction Set and Register Allocation Plan

The target processor Apple M1 is one of the 64-bit ARMv8 processors that provides many powerful instructions. These instructions can be classified into two types: general instructions and vector instructions (i.e. NEON). Among them, vector instructions can be provide the operation in parallel-way. In Table 2, instructions used for the proposed implementation are summarized. An easy distinction between vector instructions and general instructions is the presence of an arrangement specifier. Since the vector instruction treats the value inside the register by dividing it into an arrangement unit, the arrangement is expressed after the instruction or after the operand register.

64-bit ARMv8 architecture has 31 general registers and 32 vector registers. Since the number of register is limited, an efficient allocation plan for registers should be required. Figure 2 shows register scheduling plan for proposed implementation. Our proposed implementation utilized 24 vector registers for operands, three for holding look-up table values and one for constant value, for Rainbow III, and Rainbow V. For Rainbow I, 16 vector registers for operands and only single vector register needed to store look-up table values. Only six general registers are utilized in our implementation, which are mainly for housekeeping purposes like address pointer, temporary variables and etc. Rainbow III and Rainbow V require one more general register for temporary value.

3.2 Look-up Table based Polynomial Multiplication

Rainbow I is operated on the $\mathbb{F}_{16}$, and the computation is based on tower-field. That is, $\mathbb{F}_{16}$ operates on the sub-field $\mathbb{F}_4$, while the sub-field $\mathbb{F}_4$ operates on the sub-field $\mathbb{F}_2$. This can be expressed by the following formula.

$$\mathbb{F}_{16} := \mathbb{F}_4[y]/(y^2 + y + x),\ \mathbb{F}_4 := \mathbb{F}_2[y]/(x^2 + x + 1)$$

The reference implementation provided by Rainbow in NIST submission adopted the Karatsuba algorithm for implementing the tower-field operation. The implementation can be described with pseudo code form as shown in Algorithm 1. It first divides two 4-bit values (A and B) into two sets of 2-bit values (*a0*, *a1*, *b0* and *b1*). Then, it performs a series of multiplication/addition to

Table 2. Instructions for to implement parallel polynomial-multiplication based look-up table; `Xd, Vd`: destination register (general, vector), `Xn, Vn, Vm`: source register (general, vector, vector), `Vt`: transferred vector register, `T`: arrangement specifier.

asm	Operands	Description	Operation
ADD	Xd, Xn, #imm	Add registers immediate	Xd ← Xn + #imm
ADR	Xd, (Label)	Form PC-relative address	Xd ← address
AND	Vd.T, Vn.T, Vm.T	Bitwise AND	Vd ← Vn & Vm
B	(Label)	Branch	Go to Label
BEQ	(Label)	Branch if it is equal	Go to Label
CBNZ	Xt, (Label)	Compare and Branch on Nonzero	Go to Label
CMP	Xd, #imm	Compare	Flags ← result
EOR	Vd.T, Vn.T, Vm.T	Bitwise Exclusive OR	Vd ← Vn ⊕ Vm
LD1	Vt.T, [Xn]	Load multiple single-element structures	Vt ← [Xn]
LSL	Xd, Xn, #shift	Logical Shift Left immediate (general)	Xd ← Xn << #shift
MOV	Xd, #imm	Move immediate (general)	Xd ← #imm
MOVI	Vt.T, #imm	Move immediate (vector)	Vt ← #imm
RET	{Xn}	Return from subroutine	Return
SHL	Vd.T, Vn.T, #shift	Shift Left immediate (vector)	Vd ← Vn << #shift
ST1	Vt.T, [Xn]	Store multiple single-element structures	[Xn] ← Vt
SUB	Xd, Xn, #imm	Subtract immediate	Xd ← Xn - #imm
TBL	Vd.T, {Vn.16B}, Vm.T	Table vector Lookup	Vd ← Vn[Vm]
USHR	Vd.T, Vn.T, #shift	Unsigned Shift Right immediate	Vd ← Vn >> #shift

compute the intermediate values for Karatsuba algorithm. Lastly, it collects the intermediate results of each operation and accumulates it into a final result C. Since Rainbow signature uses the tower-field operation, the modular reduction is applied to the carry generated during the computation on $\mathbb{F}_4$. Consequently, polynomial-multiplication can be executed efficiently with the Karatsuba algorithms based on this tower-field arrangement [11].

`PMUL` (or `PMULL`) instruction performs polynomial-multiplication in a parallel-way, and store the result into the vector register according to arrangement specifier. Since Rainbow I multiplication variable are 4-bit, `8b` or `16b` arrangement specifier is needed. Since the polynomial-multiplication of Rainbow I is based on tower-field, modular reduction is performed if carry occurs in the sub-field $\mathbb{F}_4$. On the other hand, the minimum computation unit of `PMUL` instruction is byte-wise. Since this is not a tower-field operation, the carry occurring in the sub-field cannot be reflected correctly. Thus, it is difficult to implement Rainbow I multiplication efficiently by using the `PMUL` instruction. Considering that the input variable is divided into 2-bit units, one can still use `PMUL` instruction with modular reduction manually, but it is inefficient. Therefore, a new efficient technique for 4-bit unit polynomial-multiplication is proposed.

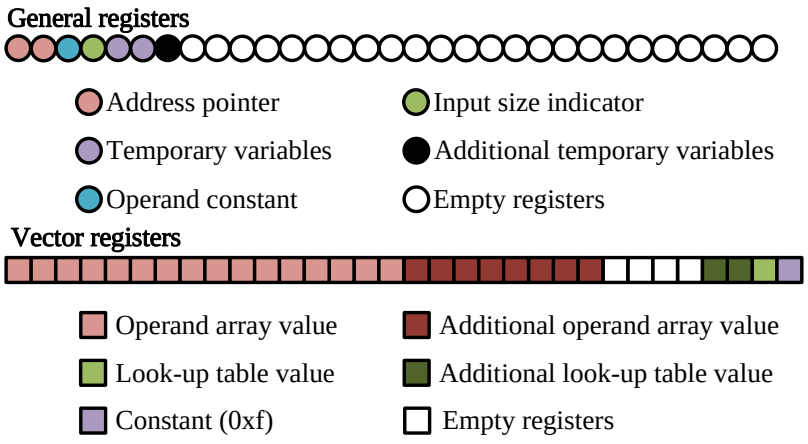

Fig. 2. Register scheduling plan.

To resolve this issue, we propose a technique based on look-up table. The table can be calculated by multiplying the cases of all variables. Since the polynomial-multiplication of Rainbow I operates on $\mathbb{F}_{16}$, each variable can express only 4-bit, and the output value is also 4-bit. Since each 4-bit value can represent 16 distinct numbers, so there is only 256 multiplication results. The proposed technique creates a table by pre-calculating the multiplication results of all possible combinations (256). During the look-up access, one of the multiplication operand is a constant. Instead of loading the whole table, only 16 values are loaded from the table according to the input constant operand. For example, operand is `0x3`, the fourth 16 table values will be loaded. Listing 1.1 represents the multiplication table used for implementation. Each value of the table is 4-bit, but the minimum data storage unit is 8-bit. So the total size of the table is 256-byte.

The look-up table can be loaded by adjusting pointer address value, this can be easily implemented with branch statements. However, this approach is time consuming and vulnerable to side channel attack. Since it moves to the table call statement by forming a branch according to the operand constant, additional time is required to adjust the address value. This additional time increases further as the operand constant value is later in the branch condition. Therefore, we proposed to directly changing the address pointer. The implementation method is as follow. Firstly, pointer to the first address of look-up table is initialized, which points to the results of operand constant 0. There are 16 possible values in the look-up table that one operand constant may load. If the address pointer is increased by 16, it becomes the table of the next constant value. These steps are implemented simply by multiplying the operand constant by 16 and adding

Algorithm 1 Pseudo-code for reference polynomial multiplication.

Input: 4-bit array A, 4-bit constant B.
Output: 4-bit accumulated output C.

```
a0 ← low 2-bit of A
a1 ← high 2-bit of A
b0 ← low 2-bit of B
b1 ← high 2-bit of B
a0b0 ← a0 × b0
a1b1 ← a1 × b1
middle ← a0 ⊕ a1 × b0 ⊕ b1
square ← a1b1 × a1b1
C ← ((middle ⊕ a1b1) ≪ 2) ⊕ a0b0 ⊕ square
return C
```

it to the address pointer. Table 2 shows the address pointer setting in pseudo-code form. If the address pointer has been adjusted, it loads table value into the vector register using the `LD1` instruction. The vector register can store up to 128-bit (16-byte), which can be completely stored in one table.

```
.balign 256
MUL_TABLE:
.byte 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, \
      0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, \
      0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, \
      0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf, \
      0x0, 0x2, 0x3, 0x1, 0x8, 0xa, 0xb, 0x9, \
      0xc, 0xe, 0xf, 0xd, 0x4, 0x6, 0x7, 0x5, \
      0x0, 0x3, 0x1, 0x2, 0xc, 0xf, 0xd, 0xe, \
      0x4, 0x7, 0x5, 0x6, 0x8, 0xb, 0x9, 0xa, \
      0x0, 0x4, 0x8, 0xc, 0x6, 0x2, 0xe, 0xa, \
      0xb, 0xf, 0x3, 0x7, 0xd, 0x9, 0x5, 0x1, \
      0x0, 0x5, 0xa, 0xf, 0x2, 0x7, 0x8, 0xd, \
      0x3, 0x6, 0x9, 0xc, 0x1, 0x4, 0xb, 0xe, \
      0x0, 0x6, 0xb, 0xd, 0xe, 0x8, 0x5, 0x3, \
      0x7, 0x1, 0xc, 0xa, 0x9, 0xf, 0x2, 0x4, \
      0x0, 0x7, 0x9, 0xe, 0xa, 0xd, 0x3, 0x4, \
      0xf, 0x8, 0x6, 0x1, 0x5, 0x2, 0xc, 0xb, \
      0x0, 0x8, 0xc, 0x4, 0xb, 0x3, 0x7, 0xf, \
      0xd, 0x5, 0x1, 0x9, 0x6, 0xe, 0xa, 0x2, \
      0x0, 0x9, 0xe, 0x7, 0xf, 0x6, 0x1, 0x8, \
      0x5, 0xc, 0xb, 0x2, 0xa, 0x3, 0x4, 0xd, \
      0x0, 0xa, 0xf, 0x5, 0x3, 0x9, 0xc, 0x6, \
      0x1, 0xb, 0xe, 0x4, 0x2, 0x8, 0xd, 0x7, \
      0x0, 0xb, 0xd, 0x6, 0x7, 0xc, 0xa, 0x1, \
      0x9, 0x2, 0x4, 0xf, 0xe, 0x5, 0x3, 0x8, \
      0x0, 0xc, 0x4, 0x8, 0xd, 0x1, 0x9, 0x5, \
      0x6, 0xa, 0x2, 0xe, 0xb, 0x7, 0xf, 0x3, \
      0x0, 0xd, 0x6, 0xb, 0x9, 0x4, 0xf, 0x2, \
```

Algorithm 2 Pseudocode of table address setting method.

Input: 4-bit constant C, address pointer P, 256-byte look-up table(LUT).
Output: address pointer P.

```
P ← first address of LUT
C ← C × 16
P ← P + C
return P
```

Algorithm 3 Look-up table based polynomial-multiplication on $\mathbb{F}_{16}$.

Input: x0 = address of A, x1 = address of B, x2(w2) = constant C.
Output: 4-bit accumulated output to A.

```
1: MOVI v31.16b, #15
2: ADR, x4, MUL_TABLE
3: LSL, w2, w2, #4
4: ADD, x4, x4, x2
5: LD1.16b {v30}, [x4]
6: LD1.16b {v30}, [x1]
7: AND.16b v0, v1, v31
8: USHR.16b v1, v1, #4
9: TBL.16b v0, {v30}, v0
10: TBL.16b v1, {v30}, v1
11: SHL.16b v1, v1, #4
12: EOR.16b v0, v0, v1
13: LD1.16b {v1}, [x0]
14: EOR.16b v1, v1, v0
15: ST1.16b {v1}, [x0]
```

```
30      0xe, 0x3, 0x8, 0x5, 0x7, 0xa, 0x1, 0xc, \
31      0x0, 0xe, 0x7, 0x9, 0x5, 0xb, 0x2, 0xc, \
32      0xa, 0x4, 0xd, 0x3, 0xf, 0x1, 0x8, 0x6, \
33      0x0, 0xf, 0x5, 0xa, 0x1, 0xe, 0x4, 0xb, \
34      0x2, 0xd, 0x7, 0x8, 0x3, 0xc, 0x6, 0x9
```

Listing 1.1. Pre-calculation result table of 4-bit tower-field polynomial multiplication results on $\mathbb{F}_{16}$ with hexadecimal notation.

With this precomputed table, the 4×4 multiplication in polynomial-multiplication is replaced by a table look-up. In this case, the implementation uses `TBL` instruction, which replaces the value of a vector register with the look-up table value. For example, if the value stored in the register is `0x3`, it is replaced with the fourth value of the table. Finally, the operation is completed by accumulating result of multiplication. Each vector register stores 16 values, so a single instruction can generate 16 results in parallel-way. The overall operation codes is detailed in Algorithm 3. When we need to parallelize more data, it is completed by using more registers or writing loop statement. In line 1-5, it loads 16-byte table according to operand constant. In line 6-8, it loads operand array values and separates into high/low 4-bit. In line 9-10, it performs the table look-up for polynomial-multiplication on $\mathbb{F}_{16}$. In line 11-15, it combines the two 4-bit results into 8-bit, and accumulates the results to output array. The entire process can be simply represented in Figure 3

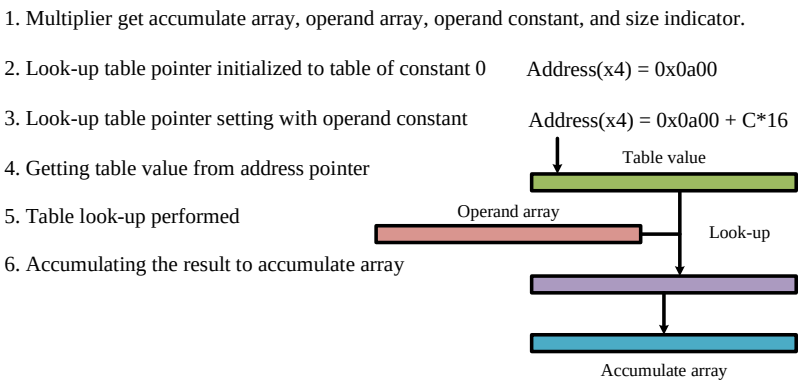

Fig. 3. Table-based polynomial-multiplier operation process.

3.3 Optimized Implementations of Rainbow III and Rainbow V

Rainbow III and Rainbow V is operated on $\mathbb{F}_{256}$, where each operand has 8-bit. Therefore, 8-bit multiplication is performed, and the number of calculation results is 65,536. In this case, the size of the additional table becomes too large that takes 65,536-byte (64KB), and it takes a lot of time accessing this look-up table. However, since Rainbow III and Rainbow V are also based on tower-field calculations, the look-up table used in Rainbow I can be used as is. The proposed technique is simple. First, each 8-bit value is decomposed into two 4-bit for sub-field $\mathbb{F}_{16}$ operation. At this moment, polynomial-multiplication on $\mathbb{F}_{16}$ can be performed by table look-up. However, since Rainbow III and Rainbow V include 4-bit squaring operation, it required an additional 16 bytes in the look-up table, which is shown in Listing 1.2. Therefore, the size of the entire table becomes 272-byte for Rainbow III and Rainbow V implementation. After the operations on $\mathbb{F}_{16}$ are completed, the rest of the operations take place. The entire process is shown in Algorithm 4, which computes 32-byte results in a parallel-way. Referring to Algorithm 4, lines 1-11 it separates the operand constant into high/low 4-bit and loads two 16-byte tables according to separate constant. In lines 12-13, it calls additional table for Rainbow III and Rainbow V, followed by steps to load the operand array values and separate them into high/low 4-bit (lines 14-19). In lines 20-23, the algorithm performs the table look-up with the previous table, and then calculates the intermediate values and switching table according to this one in lines 24-32. In lines 33-36, it performs the table look-up through switched table. In line 37-38, it operates the table look-up through the additional table. In lines 39-44, it combines result values into 8-bit. Finally, the results are accumulated into the output array in lines 45-51. Since Rainbow III and Rainbow V require the intermediate value multiplication on $\mathbb{F}_{16}$, table replacement occurs once, unlike Rainbow I. To compute with a parallel-way larger than 32-bytes, one can use more registers or loop statement.

```
.balign 16
ADDI_TABLE:
.byte 0x0, 0x8, 0xc, 0x4, 0xb, 0x3, 0x7, 0xf, \
      0xd, 0x5, 0x1, 0x9, 0x6, 0xe, 0xa, 0x2
```

Listing 1.2. Additional table of Rainbow III and Rainbow V with hexadecimal notation.

Algorithm 4 Look-up table based polynomial-multiplication on $\mathbb{F}_{256}$.

Input: x0 = address of array A, x1 = address of array B, x2(w2) = constant C.
Output: 4-bit accumulated output to A.

```
 1: MOVI v31.16b, #15
 2: AND w4, w2, #15
 3: LSR w5, w2, #4
 4: ADR x6, MUL_TABLE
 5: LSL w4, w4, #4
 6: ADD x6, x6, x4
 7: ADR x7, MUL_TABLE
 8: LSL w5, w5, #4
 9: ADD x7, x7, x5
10: LD1.16b {v30}, [x6]
11: LD1.16b {v29}, [x7]
12: ADR x6, ADDI_TABLE
13: LD1.16b {v27}, [x6]
14: LD1.16b {v1}, [x1], #16
15: LD1.16b {v5}, [x1], #16
16: AND.16b v0, v1, v31
17: USHR.16b v1, v1, #4
18: AND.16b v4, v5, v3
19: USHR.16b v5, v5, #4
20: TBL.16b v2, {v30}, v0
21: TBL.16b v3, {v29}, v1
22: TBL.16b v6, {v30}, v4
23: TBL.16b v7, {v29}, v5
24: EOR.16b v0, v0, v1
25: EOR.16b v4, v4, v5
26: AND w4, w2, #15
27: LSR w5, w2, #4
28: EOR w4, w4, w5
29: ADR x6, MUL_TABLE
30: LSL w4, w4, #4
31: ADD x6, x6, x4
32: LD1.16b {v28}, [x6]
33: TBL.16b v0, {v28}, v0
34: EOR.16b v0, v0, v2
35: TBL.16b v4, {v28}, v4
36: EOR.16b v4, v4, v6
37: TBL.16b v3, {v27}, v3
38: TBL.16b v7, {v27}, v7
39: SHL.16b v0, v0, #4
40: EOR.16b v0, v0, v2
41: EOR.16b v0, v0, v3
42: SHL.16b v4, v4, #4
43: EOR.16b v4, v4, v6
44: EOR.16b v4, v4, v7
45: LD1.16b {v1}, [x0], #16
46: LD1.16b {v5}, [x0], #16
47: SUB x0, x0, #32
48: EOR.16b v1, v1, v0
49: EOR.16b v5, v5, v4
50: ST1.16b {v1}, [x0], #16
51: ST1.16b {v5}, [x0], #16
```

4 Evaluation

The implementation was evaluated on a Apple M1 chip, which can be clocked up to 3.2 GHz. Implementation is carried out on the Xcode framework, and compile using the compile option `-O3` (i.e. fastest). The performance evaluation is carried out in two ways. First, the performance of the previous multiplication algorithm

Table 3. Evaluation results of multiplier (unit: clock cycles).

Algorithm	$\mathbb{F}_{16}$ multiplier	$\mathbb{F}_{256}$ multiplier
Previous work [1]	355	16,557
This work	**58**	**99**

Table 4. Comparison of execution timing (unit: $\times 10^6$ clock cycles).

Algorithm	Previous work [1]			**This work**		
	Key Scheduling	Signature	Verification	**Key Scheduling**	**Signature**	**Verification**
Rainbow I Classic	2.53	3.17	3.30	**1.59**	**0.32**	**0.064**
Rainbow I CZ	281.76	3.20	12.42	**16.90**	**0.48**	**9.18**
Rainbow I Compressed	281.79	127.71	12.42	**16.90**	**12.99**	**9.18**
Rainbow III Classic	3,141	27.65	28.67	**88.13**	**1.98**	**5.86**
Rainbow III CZ	3,570	27.65	83.74	**93.73**	**1.98**	**60.96**
Rainbow III Compressed	3,570	1,690	83.71	**93.70**	**75.36**	**60.96**
Rainbow V Classic	8,830	61.12	62.4	**530.14**	**2.46**	**2.69**
Rainbow V CZ	10,140	61.12	186.66	**561.18**	**2.50**	**127.17**
Rainbow V Compressed	10,140	4,850	186.88	**561.06**	**279.94**	**127.14**

and the proposed table-based parallel multiplication algorithm is compared. The second is to compare the performance of previous Rainbow signature and the Rainbow signature adopted the proposed multiplier.

For measuring multiplier performance, 512-byte input was used, and each algorithm was repeated 1,000,000 times to measure operation time. The performance evaluation of the multiplier is shown in Table 3. Previous $\mathbb{F}_{16}$ multiplier takes about 355 clock cycle and $\mathbb{F}_{256}$ multiplier takes about 16,557 clock cycle. Proposed table-based parallel multiplier takes only 58 and 99 clock cycles for $\mathbb{F}_{16}$ multiplication and $\mathbb{F}_{256}$ multiplication, respectively. Therefore, the proposed technique to speed up the polynomial multiplication is 6.12× and 167.2× faster than the reference implementation for $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ respectively.

The Rainbow signature was implemented using the proposed polynomial multiplication technique based on look-up table. The performance comparison is conducted on Classic, Circumzenithal (CZ), and Compressed versions for three security levels (Rainbow I, III and V). For the performance measurement, the previous work uses the average value of 300 times iteration hours, and the proposed method uses the average value of 10,000 times repetition hours. The difference in number of iterations of the two algorithms is that the proposed method ended before the CPU usage reaches the maximum because the operation speed is too fast. Considering this, the proposed technique was tested by increasing number of iterations count. The implementation result is shown in Table 4.

In Rainbow I classic version, the proposed technique has better performance than previous work about 1.59×, 9.91×, and 51.6× in key scheduling, signature, and verification, respectively. Subsequently, in Rainbow I Circumzenithal version, the proposed implementation has 16.67×, 6.67×, and 1.35× better performance than the previous work, and in Rainbow I Compressed, the proposed method has 16.67×, 9.83×, and 1.35× better performance. Therefore, the biggest

performance difference in Rainbow I is verification process of Classic version, which shows a 51.6× better performance than previous implementations.

Evaluating the Rainbow III in the same way. Classic version of proposed implementation shows higher performance than previous implementations about 35.64×, 13.96×, 4.89× for key scheduling, signature, and verification, respectively. In case of Circumzenithal version of Rainbow III, the proposed Rainbow implementation achieved 38.09×, 13.96× 1.37× higher calculation speed than previous works. Similarly, for Compressed version, there are a performance difference of 38.10×, 22.42×, and 1.37×.

Finally, the performance evaluation for Rainbow V are as follows. First, in the case of the Classic version, the proposed technique has 16.66×, 28.85×, and 23.05× better performance for key scheduling, signature, and verification, respectively, compared to previous implementation. In Circumzenithal version, the proposed method shows better performance than the previous work about 18.07×, 24.45×, 1.47×. In the case of Compressed version, the proposed technique has 18.07× 17.33× 1.47× better performance than the previous one.

Overall, the verification of the Rainbow I Classic version has the highest performance improvement of 51.6×.

5 Conclusion

In this paper, we proposed the table based polynomial-multiplication technique. Previous implementation applies the efficient Karatsuba algorithm to the polynomial multiplication. However the multiplication takes a long time because the size of the parameter is too large. The proposed method reduced computation load from multiplication by using look-up table which takes 256-byte, or 272-byte, and performance of multiplier has a difference of up to 167.2×. As a result of transplanting the proposed multiplier to Rainbow signatures, computational performance could be improved 51.6× in best case. Another reason for performance improvement is parallel operation of vector registers and vector instructions. Therefore, the proposed technique can be improved performance of Rainbow signatures on target processor (i.e. Apple M1). As a future study, we present the implementation of another post-quantum cryptography on the target processor, the latest M1 processor, or using a different processor (i.e. RISC-V) as well [12, 13].

References

1. J. Ding and D. Schmidt, "Rainbow, a new multivariable polynomial signature scheme," in *International conference on applied cryptography and network security*, pp. 164–175, Springer, 2005.
2. A. Kipnis, J. Patarin, and L. Goubin, "Unbalanced Oil and Vinegar signature schemes," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 206–222, Springer, 1999.

3. A. Petzoldt, S. Bulygin, and J. Buchmann, "Cyclicrainbow–a multivariate signature scheme with a partially cyclic public key," in *International Conference on Cryptology in India*, pp. 33–48, Springer, 2010.
4. H. Seo, Z. Liu, P. Longa, and Z. Hu, "SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 1–20, 2018.
5. J. Lee, "VHDL design for out-of-order superscalar processor of a fully pipelined scheme," *The Journal of the Institute of Internet, Broadcasting and Communication*, vol. 21, no. 1, pp. 99–105, 2021.
6. T. Chou, M. J. Kannwischer, and B.-Y. Yang, "Rainbow on Cortex-M4.," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 532, 2021.
7. G.-S. Kim and Y.-S. Kim, "Efficient implementation of finite field operations in NIST PQC Rainbow," *Journal of the Korea Institute of Information Security & Cryptology*, vol. 31, no. 3, pp. 527–532, 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,"
9. D. T. Nguyen and K. Gaj, "Optimized software implementations of CRYSTALS-Kyber, NTRU, and Saber using NEON-based special instructions of ARMv8,"
10. S. Streit and F. De Santis, "Post-quantum key exchange on ARMv8-A: A new hope for NEON made simple," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1651–1662, 2017.
11. D. J. Bernstein and T. Chou, "Faster binary-field multiplication and faster binary-field MACS," in *International Conference on Selected Areas in Cryptography*, pp. 92–111, Springer, 2014.
12. H. Kwon, H. Kim, E. S. Woo, M. Shim, W.-K. Lee, Z. Hu, and H. Seo, "Optimized implementation of SM4 on AVR microcontrollers, RISC-V processors, and ARM processors,"
13. H. Seo, H. Kwon, K. Jang, and H. Kim, "Optimized implementation of scalable multi-precision multiplication method on RISC-V processor for high-speed computation of post-quantum cryptography," *Journal of the Korea Institute of Information Security & Cryptology*, vol. 31, no. 3, pp. 473–480, 2021.