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

HYEOKDONG KWON, Hansung University, Republic of Korea
HYUNJUN KIM, Hansung University, Republic of Korea
MINJOO SIM, Hansung University, Republic of Korea
WAI-KONG LEE, Gachon University, Republic of Korea
HWAJEONG SEO*, Hansung University, Republic of Korea

Rainbow Signature Scheme is one of the finalists in the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography (PQC) standardization competition, but failed to win because it has lack of stability in the parameter selection. It is the only signature candidate based on a multivariate quadratic equation. Rainbow signatures have smaller signature sizes compared to other post-quantum cryptography candidates. However, it requires expensive tower-field based polynomial multiplications. In this paper, we propose an efficient implementation of Rainbow signature using a look-up table-based multiplication method. The polynomial multiplications in Rainbow signatures are performed on the $\mathbb{F}_{16}$ field, which is divided into sub-fields $\mathbb{F}_4$ and $\mathbb{F}_2$ under the tower-field method. To accelerate the multiplication process on target processors, we propose a look-up table-based tower-field multiplication technique. In $\mathbb{F}_{16}$, all values are expressed in 4-bit data format and can be implemented using a 256-byte look-up table access. The implementation uses the TBL and TBX instructions of the 64-bit ARMv8 target processor. For Rainbow III and Rainbow V, they are computed on the $\mathbb{F}_{256}$ field using a additional 16-byte table instead of creating a new look-up table. The proposed technique uses the vector registers of 64-bit ARMv8 processors and can calculate 16 result values with a single instruction. We also proposed implementations that are resistant to timing attacks. There are two types of implementations. The first one is the cache side-attack resistant implementation, which utilizes the 128-byte cache lines of the M1 processor. In this implementation, cache misses do not occur, and cache hits always occur. The second type is the constant-time implementation. This method takes a step-by-step approach to finding the required look-up table value and ensures that the same number of accesses is made regardless of which look-up table value is called. This implementation is designed to be constant-time, meaning it does not leak timing information. Our experiments on modern Apple M1 processors showed up to 428.73× and 114.16× better performance for finite field multiplications and Rainbow signatures schemes, respectively, compared to previous reference implementations. To the best of our knowledge, this proposed Rainbow implementation is the first optimized Rainbow implementation for 64-bit ARMv8 processors.

CCS Concepts: • **Security and privacy → Digital signatures**.

Additional Key Words and Phrases: Post-quantum Cryptography, Rainbow Signature, Software Implementations, 64-bit ARMv8 Processors

Authors' addresses: HyeokDong Kwon, Hansung University, 116, Samseongyo-ro 16-gil, Seoul, Seongbuk-gu, Republic of Korea, 02876, korlethean@gmail.com; HyunJun Kim, Hansung University, 116, Samseongyo-ro 16-gil, Seoul, Seongbuk-gu, Republic of Korea, 02876, khj930704@gmail.com; MinJoo Sim, Hansung University, 116, Samseongyo-ro 16-gil, Seoul, Seongbuk-gu, Republic of Korea, 02876, minjoos9797@gmail.com; Wai-Kong Lee, Gachon University, 1332, Seongnam-daero, Seongnam-si, Sujeong-gu, Republic of Korea, 13306, waikong.lee@gmail.com; Hwajeong Seo, Hansung University, 116, Samseongyo-ro 16-gil, Seoul, Seongbuk-gu, Republic of Korea, 02876, hwajeong84@gmail.com.

1539-9087/2023/7-ART $15.00
https://doi.org/10.1145/3607140

1 INTRODUCTION

With the recent development of quantum computers, modern public key cryptographic algorithms, such as RSA and ECC, are under threat. Quantum computer algorithms like Grover search and Shor algorithm can be implemented on quantum computers and are effective for fast key search or prime factorization. A research presented computational cost of attack for specific block cipher on the simulator [3]. This means that quantum computers can cause the modern cryptography system to collapse. To prevent this threat, the National Institute of Standards and Technology (NIST) is holding a post-quantum cryptography standardization competition to establish a standard post-quantum cryptography that will be safe even in the era of quantum computers. The Rainbow signature is the only multivariate-based public key signature among the Round 3 finalists of the NIST competition. It has a reasonable signature size but has the disadvantage of having a long computation time for signature generation.

In this paper, we propose a look-up table-based polynomial multiplication technique to speed up the polynomial multiplication in the Rainbow signature. The look-up table contains 4×4 multiplication results and is used to accelerate the process. We also utilize vector registers and vector instructions, such as NEON and ASIMD, to perform parallel table look-up operations. The proposed fast polynomial multiplication technique is used to present optimized implementations of Rainbow I, Rainbow III, and Rainbow V. These implementations show significantly better performance compared to the previous reference implementations.

In addition, we have introduced timing attack-resistant implementations. These implementations fall into two categories. The first type is the cache side-attack resistant implementation, which leverages the 128-byte cache lines of the M1 processor. It guarantees the absence of cache misses and ensures that cache hits always occur. The second type is the constant-time implementation. This approach adopts a systematic procedure to locate the necessary look-up table value and guarantees a consistent number of accesses, regardless of the specific value being accessed. By design, this implementation operates in constant time, safeguarding against the leakage of timing information.

The structure of this paper is organized as follows: Section 2 covers related works on Rainbow post-quantum cryptography, the target 64-bit ARMv8 processors, and the optimal implementation of post-quantum cryptography on the target processor. Section 3 discusses the proposed method. In Section 4, a performance comparison is conducted. The conclusion and future work of this paper are presented in Section 5.

2 BACKGROUNDS

2.1 Post-Quantum Cryptography: Rainbow Signature

The Rainbow signature is rooted in the multivariate quadratic equations introduced by Jintai Ding and Dieter Schmidt in 2004 [7]. Table 1 lists the parameters of the Rainbow signature [21].

Table 1. Size of key and signature for Rainbow. 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

Security analysis of Rainbow signature is an ongoing process, and recent research has shown that the secret key can be recovered using a standard laptop within a week when targeting Security level I of the Second submission Rainbow [2]. As a result, the parameters of Rainbow signature have been revised in the Round 3 submission [6], but further security analysis is still necessary.

Utilizing the Unbalanced Oil and Vinegar (UOV) structure, the Rainbow signature offers fast computation for public key algorithms while requiring a small amount of memory [9]. A multivariate quadratic equation ($\mathbb{MQ}$) is a mathematical problem of finding the answer X of P, when there are m quadratic equations with n variables, as shown in Equation 1.

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

The security level of the multivariate cryptography system, which operates in a finite field K, is dependent on $\mathbb{MQ}$. The Rainbow signature, a type of multivariate cryptography, relies on $\mathbb{MQ}$ and comprises key scheduling, signature generation, and signature verification steps. Figure 1 provides an overview of the overall structure of the Rainbow signature.

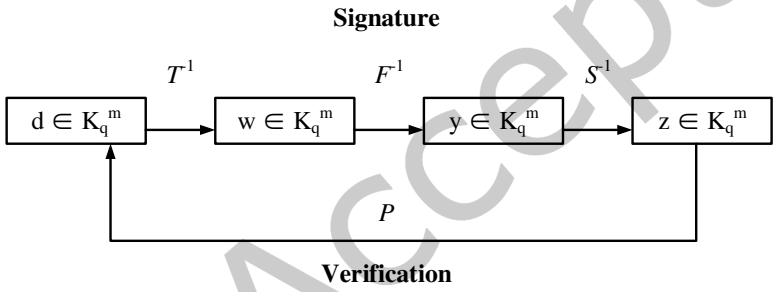

Fig. 1. Overview of the overall structure of Rainbow signature.

- **Key Scheduling.** The public key is expressed as $P = T \circ F \circ S : K^n \rightarrow K^m$, and the private key comprises T, F, and S. For multivariate signature schemes, it is necessary that $n \geq m$ to guarantee that every message has a signature.
- **Signature Generation.** To generate a signature for a message (or its hash value) $d \in K^m$, Equation 2 must be calculated recursively.

$$w = T^{-1}(d) \in K^m, y = F^{-1}(w) \in K^n, z = S^{-1}(y) \tag{2}$$

  $z \in K^n$ represents the signature of the message d, and finding one of the pre-images of w under the central map F is indicated by $F^{-1}(w)$.
- **Signature verification.** To validate the authenticity of the signature $z \in K^n$, the verifier simply calculates $d' = P(z)$. If the result of the signature verification matches the message d, the signature is accepted; otherwise it is rejected.

Rainbow signature has an alternative algorithm called CZ-Rainbow (Circumzenithal Rainbow), which was inspired by Petzoldt et al's work [14]. Unlike the Classic Rainbow, the key generation process for CZ-Rainbow is reversed and doesn't use cyclic matrices. It leads to a reduction in public key size by about 70%. However, this

reduction comes with longer computation times during key scheduling and verification due to the inefficiency of its key generation process.

Another alternative is Compressed Rainbow, which has a similar structure to CZ-Rainbow but calculates the central map without storing it. This results in a smaller key size compared to Classic Rainbow and CZ-Rainbow, but longer computation times during signature generation.

2.2 Target Processor: 64-bit ARMv8 Architecture

ARM processors are provide high performance in IoT devices, sensor nodes, smartphones, and smart watches. We targeted the latest ARMv8-A (ARMv8) architecture, which has 64-bit general registers and 128-bit vector registers. Among the registers, vector registers provide parallel operations, and the arrangement specifier determines the packing unit of data. For example, a 16b arrangement specifier indicates that the internal data of the specified register is treated as 16 values of 1-byte [17].

We used three kinds of ARMv8 processors. The first one is the Apple M1 processor (i.e., Apple Silicon), which is intended for use in Apple products such as Macs and iPads. It is the latest ARM processor and is based on 64-bit ARMv8 architecture. The M1 processor is a System-on-Chip (SoC) that integrates a multi-core CPU (Central Processing Unit), GPU (Graphics Processing Unit), DSP (Digital Signature Processor), and Neural engine into a single chip. It was produced using a 5-nanometer process and consists of approximately 16 billion transistors [11].

The second processor is the Apple A13 bionic processor, which is part of the ARMv8 family and was announced in 2019. The A13 was primarily used in Apple smartphones and has a 6-core CPU, with two high-performance cores called Lightning and the remaining ones are energy-efficient cores named Thunder. It has 8.5 billion transistors and was considered one of the best processors at the time [20]. However, it is now seen an older processor compared to the M1 chip and has a significantly lower performance compared to the M1 currently.

The third processor is the Broadcom BCM2711, which is a Cortex-A72 processor and has a big.LITTLE design, similar to the A13 and M1 processors. The big.LITTLE design is a power-optimization technology that combines high-performance and energy-efficient CPUs. It has an Energy Aware Scheduler (EAS) that manages tasks by allocating them to the least energy-consuming CPU. The Cortex-A72 processor is widely used for IoT devices and educational purposes, such as the Raspberry Pi 4 [12].

2.3 Previous Implementations of Post Quantum Cryptography on ARM Processors

Chou et al. proposed an optimized implementation of the Rainbow signature on a Cortex-M4 microcontroller. The Cortex-M4 is a family of 32-bit ARMv7 [5]. To implement their proposal, Chou et al. utilized fast constant-time bit-slice $\mathbb{F}16$ multiplication. It achieved a multiplication of 32 field elements in 32 clock cycles. As two $\mathbb{F}16$ elements fit into one byte, and eight $\mathbb{F}16$ elements can be stored in one 32-bit register, they proposed a significantly faster $\mathbb{F}16$ multiplication routine. The routine runs in constant time and each field element is implemented in bit-slice form on four separate registers, holding a total of 32 elements.

Kim et al. also targeted Cortex-M4 environment and proposed a revised polynomial multiplication technique [8]. This approach reduced the number of XOR operations by 13.7% compared to a previous work by Chou et al. Additionally, while Chou et al. used table look-up for inversion, Kim et al. used a 4×4 matrix inverse method, resulting in constant-time computation.

Sanal et al. presented an optimized implementation of the Kyber encryption schemes for two specific processors: the 64-bit ARM Cortex-A and the Apple A12 processors [15]. In their study, the authors aimed to improve the performance of three components of the Kyber algorithm, namely the Number Theoretic Transform (NTT), noise sampling, and symmetric function implementations, by leveraging the capabilities of an AES accelerator. The results of their implementation demonstrated a significant improvement over previous work, with the performance gains of 1.72×, 1.88×, and 2.29×, respectively, in key generation, encapsulation, and decapsulation.

Nguyen et al. presented an optimized implementation of three lattice-based NIST post-quantum cryptography key encapsulation mechanism finalists, CRYSTALS-Kyber, NTRU, and Saber, in an ARMv8 environment [13]. This implementation utilized the NEON instruction (a vector instruction), resulting in significant speed-up compared to a reference implementation written solely in the C language. Nguyen et al. also noted that NTT and Toom-Cook are suitable algorithms for implementing polynomial multiplication, and their optimization using the ARMv8 NEON instruction was confirmed through experimental results.

In their study, Streit et al. demonstrated an optimized implementation of the New Hope post-quantum key exchange algorithm on the ARMv8-A platform [19]. The authors implemented full vectorization for all ring operations. To improve performance, three alternative modular reductions were employed in the implementation, enabling parallel execution of the Number Theoretic Transform (NTT). The results showed that the vectorized NTT required 18,909 clock cycles on an ARM Cortex-A53 processor when using a 16-bit unsigned integer.

In terms of efficient implementation on other processors, Chen et al. proposed effective $\mathbb{MQ}$ computation on the Intel CPU with SIMD instructions [4]. They implemented $\mathbb{MQ}$ using `VPSHUFB` instruction for parallel implementation. In this paper, we show table based implementation on ARMv8 processors. Unlike previous work, we re-design the method with new ARMv8 instructions and fully utilize registers of ARMv8 processors. Furthermore, we introduce novel side-channel attack countermeasures.

3 PROPOSED TECHNIQUE

In this section, we present a look-up table-based parallel method for polynomial multiplication, which enhances the computational performance of the Rainbow signature at all security levels, including its alternate versions CZ-Rainbow and Compressed Rainbow. This technique is optimized for 64-bit ARMv8 processors, utilizing the vector registers and vector instructions of the ARMv8 architecture to enable parallel computation structures.

3.1 Instruction Set Summary and Register Scheduling Plan

The target processors, including Apple M1, Apple A13 Bionic, and Cortex-A72, are part of the 64-bit ARMv8 processor family and offer many powerful instructions. These instructions can be classified into two categories: general instructions or vector instructions (e.g., NEON or ASIMD). Vector instructions have the ability to perform operations in a parallel manner. Table 2 lists instructions used in the proposed implementations.

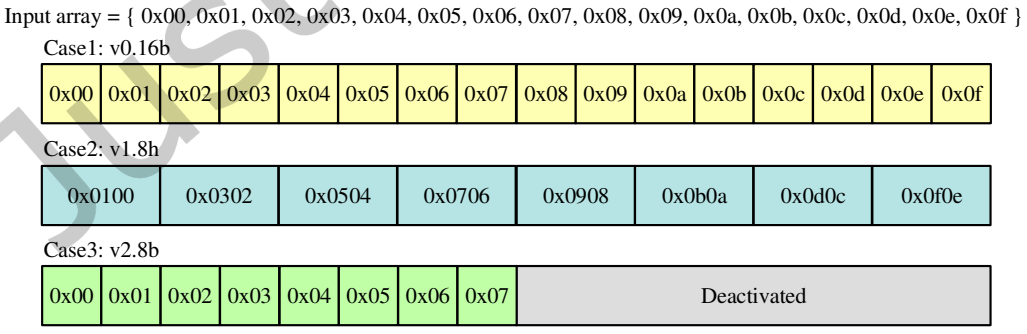

Fig. 2. Example of using arrangement specifiers. Case 1: 16-byte specifier; each value is treated as 8-bit. Case 2: 8-half-word specifier; each value is treated as 16-bit. Two 8-bit input values are treated as one 16-bit value. Case 3: 8-byte specifier; each value is treated as 8-bit. Last 8-byte value is not loaded, and half of vector register will be deactivated, storing 0 values.

Table 2. Instructions for 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	Bit-wise 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	Bit-wise 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

The main difference between vector instructions and general instructions is that vector instructions have an arrangement specifier. The reason for this is due to the characteristic of vector registers, which can hold up to 128-bit, with the unit of operation being determined by arrangement specifier. This makes it easy to change the handling unit within the register by changing arrangement specifier. Although vector registers can store up-to 128-bit, it is possible to use only half of vector registers (64-bit) if it is desired. In that case, half of the vector register is deactivated and contains 0 values. Figure 2 illustrates an example of handling units with arrangement specifier, showing the difference in handling units as arrangement specifier changes.

The 64-bit ARMv8 processors have 31 general-purpose registers and 32 vector registers. To increase computational efficiency, we first make a register scheduling plan, as the number of registers is limited. If no plan is in place, the the store and load operations for intermediate values can decrease computational efficiency. Figure 3 shows the register scheduling plan for the proposed implementation. In the proposed Rainbow I implementation, 16 vector registers are used for operands and one vector register is used to hold look-up table values. In the case of Rainbow III and Rainbow V, these security levels require additional registers, so 24 vector registers are used for operands, 3 vector registers are used for look-up table values, and one vector register is used for a constant. The proposed Rainbow I implementation also uses 6 general-purpose registers for housekeeping, managing the address pointer and temporary variables. Rainbow III and Rainbow V require one more general-purpose register.

3.2 Look-up Table based Polynomial Multiplication

Rainbow signature needs polynomial multiplication and its process is based on tower-field method. Rainbow I is operated on $\mathbb{F}_{16}$ field. A distinctive feature of tower field calculations is that it move to sub-field in the course of the calculation. $\mathbb{F}_{16}$ field operations are performed on the sub-field $\mathbb{F}_4$ field. Also, $\mathbb{F}_4$ moves to the sub-field $\mathbb{F}_2$. These process can be expressed by the following formula 3.

$$\mathbb{F}_{16} := \mathbb{F}_4[y]/(y^2 + y + x), \mathbb{F}_4 := \mathbb{F}_2[y]/(x^2 + x + 1) \tag{3}$$

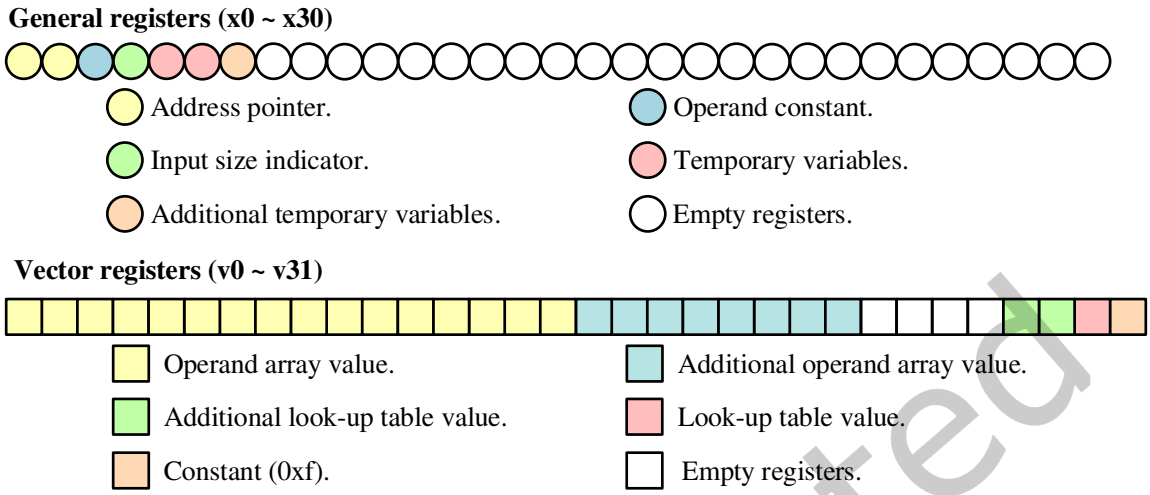

Fig. 3. Register scheduling plan. Additional registers are only used for Rainbow III or Rainbow V.

The implementation submitted by the Rainbow team adopted the Karatsuba algorithm for implementing tower-field operations. This implementation is described in 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 multiplication and addition to compute intermediate values for Karatsuba algorithm. Finally, it collects all intermediate results of each operation and accumulates them into a final result, *C*. During computation on $\mathbb{F}_4$, the modular reduction is applies to the carry, because Rainbow signatures are based on tower-field operations. Consequently, polynomial multiplication can be executed efficiently with the Karatsuba algorithm based on the tower-field method [1]. Detailed descriptions of tower-field multiplication are shown in Figure 4.

Algorithm 1 Pseudo-code for previous 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
```

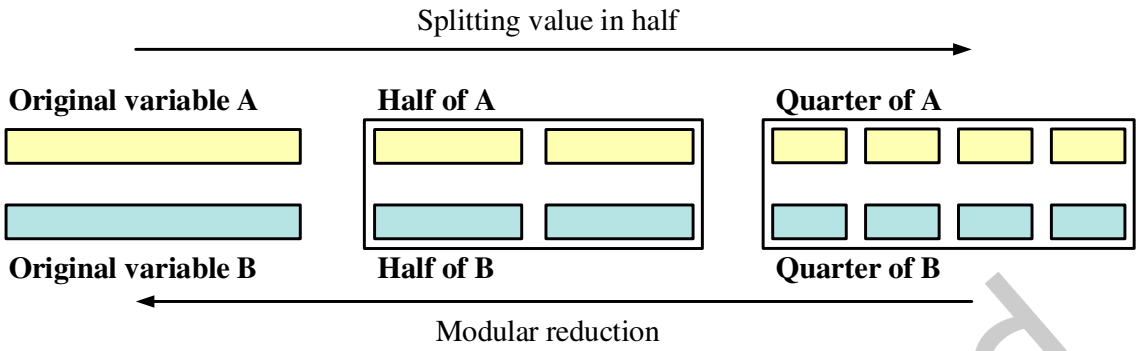

Fig. 4. The description of tower-field based multiplication process. The width of square indicates the size of variable. White squares indicate multiplication on each value from within the squares. When the variable is 8-bit, splitting is performed twice. And when the variable is 4-bit, splitting is performed once.

The ARMv8 processors have the `PMUL` and `PMULL` instructions for performing polynomial multiplications in a parallel. However, Rainbow signature is based on tower-field method and requires the modular reduction process if a carry is generated in the sub-field. Although `PMUL` and `PMULL` instructions can perform modular reduction, they have the assumption that operands must be at least 8-bit. But these instructions cannot reduced the carry that occurs in the sub-field $\mathbb{F}_2$, which is represented by 2-bit. An implementation of tower-field with `PMUL` and `PMULL` can be described in pseudo-code form in Algorithm 2. Therefore, we did not use `PMUL` and `PMULL` and they are not our comparison target.

Instead of using the `PMUL` and `PMULL` instructions, we implemented a polynomial multiplication based on a look-up table. The proposed look-up table pre-calculates all multiplications of variables that can be expressed as 4-bit, which can represent 16 cases with an output size of 4-bit. This results in 256 different multiplications, requiring 256-byte of storage space (since the minimum storage unit is 8-bit). Rainbow I is calculated in the $\mathbb{F}_{16}$, where all values are expressed in 4-bit, so the look-up table can utilized. During look-up table access, one of the multiplication operands is a constant, so instead of loading the entire table, only 16 values are loaded according to the constant operand. For example, if the constant operand is `0x3`, the fourth 16 values of the table would be loaded.

The look-up table values are loaded according to the pointer address stored in a general purpose register. This can be easily implemented using a branch statement. However, this approach is simple but not efficient and vulnerable to side channel attacks (e.g. timing attacks). For example, if it is implemented as a branch statement, the table loading speed depends on the constant operand, creating a vulnerability for timing attacks. To mitigate this, the address pointer is directly changed. When declaring and saving the look-up table, the lower address value can be specified as `0x00`. If the lower address is then multiplied by 16, the pointer can be moved to the desired look-up table position. This structure is expressed in Algorithm 3 in pseudo-code form. The algorithm 3 starts by initializing the address pointer to the start address of the look-up table. The results of the 0 constant multiplication are stored at this address. When the address pointer increased by 16, it moves to the next constant (e.g. $0 \rightarrow 1$) multiplication result. This is achieved by multiplying the operand constant by 16 and adding it to the address pointer. If the address pointer is completed, the look-up table values can be loaded by referencing the address pointer. The overall process is illustrated in Figure 5. The look-up table table values are loaded into a vector register can store up to 128-bit (i.e. 16-byte), which is the size of one table.

The 4×4 polynomial multiplication is replaced by a table look-up. The entire process is described in Algorithm 4. In line 1, the lower 8-bit value of the address pointer for the look-up table is initialized to `0x00`. This is necessary because the table values are loaded in 16-byte units and the address pointer must be moved in of multiples of 16. The stable movement is only possible when the lower address value is set to `0x00`. In line 2, the operand constant is multiplied by 16. In line 3, the address pointer and the constant operand are added together. In line 4, 16-byte of table values are loaded from the address pointer for look-up table. These steps (lines 2-4) are explained in Algorithm 3 and Figure 5. In line 5, the loop condition is established. In general, table look-up iteratively for the length of input array. However, the proposed implementation performs parallel operations in 16-byte units. The number of iterations is (length of the input array / 16). In line 7, a variable is set to adjust operand array index. In lines 8-9, 8-bit are divided into two 4-bit variables. Rainbow I operates in the $\mathbb{F}_{16}$, where values are expressed as 4-bit, but the minimum unit of data storage is 8-bit. Two 4-bit values are stored in one byte. In lines 10-11, table look-up is performed, replacing the polynomial multiplication process. In line 12, the divided values are concatenated and restored. In line 14, the results are returned.

Algorithm 5 is the source code that implements Algorithm 4. In lines 1-5, it loads a 16 bytes table according to the operand constant. In lines 6-8, it loads operand array values and separates them into high/low 4 bits. In lines 9-10, it performs the table look-up for polynomial multiplication on $\mathbb{F}_{16}$. In line 11-15, it combines the two 4 bits results into 8 bits, and accumulates results to the output array. The entire process can be simply represented in Figure 6.

3.3 Optimized Implementations of Rainbow III and Rainbow V

Algorithm 2 Process of `PMUL` instruction based polynomial-multiplication on $\mathbb{F}_{16}$.

Input: Operand array A(4-bit||4-bit), accumulated array B.
Output: Accumulated array B.

1: Loop ← |A| / 16
2: **for** i from 0 to Loop **do**
3: j ← i * 16
4: $A1$[j~j+15] ← A[j~j+15] & 0x0f
5: $A2$[j~j+15] ← A[j~j+15] » 4
6: $A1_{lower}$[j~j+15] ← $A1$[j~j+15] & 0x03
7: $A1_{higher}$[j~j+15] ← $A1$[j~j+15] » 2
8: $A1_{lower}$[j~j+15] ← `PMUL`($A1_{lower}$[j~j+15], C & 0x03)
9: $A1_{lower}$[j~j+15] ← `Modular reduction`($A1_{lower}$[j~j+15])
10: $A1_{lower}$[j~j+15] ← $A1_{lower}$[j~j+15] & 0x03
11: $A1_{higher}$[j~j+15] ← `PMUL`($A1_{higher}$[j~j+15], C » 2)
12: $A1_{higher}$[j~j+15] ← `Modular reduction`($A1_{higher}$[j~j+15])
13: $A1_{higher}$[j~j+15] ← $A1_{higher}$[j~j+15] &0x03
14: $A1$[j~j+15] ← `PMUL`($A1_{lower}$[j~j+15], $A1_{higher}$[j~j+15])
15: $A1$[j~j+15] ← `Modular reduction`($A1$[j~j+15])
16: $A1$[j~j+15] ← $A1$[j~j+15] & 0x0f
17: $A2$ `proceeds in the same way from line 6 to 16`
18: B[j~j+15] ← B[j~j+15] ⊕ (A_{lower} & (A_{higher} « 4))
19: **end for**
20: **return** B

Algorithm 3 Pseudo-code of table address setting step.

Input: 4 bits constant C, address pointer P which aligned lower address to 0x00, 256 bytes look-up table (LUT).
Output: address pointer P.

1: $P \leftarrow$ `first address of LUT`
2: $C \leftarrow C \times 16$
3: $P \leftarrow P + C$
4: **return** P

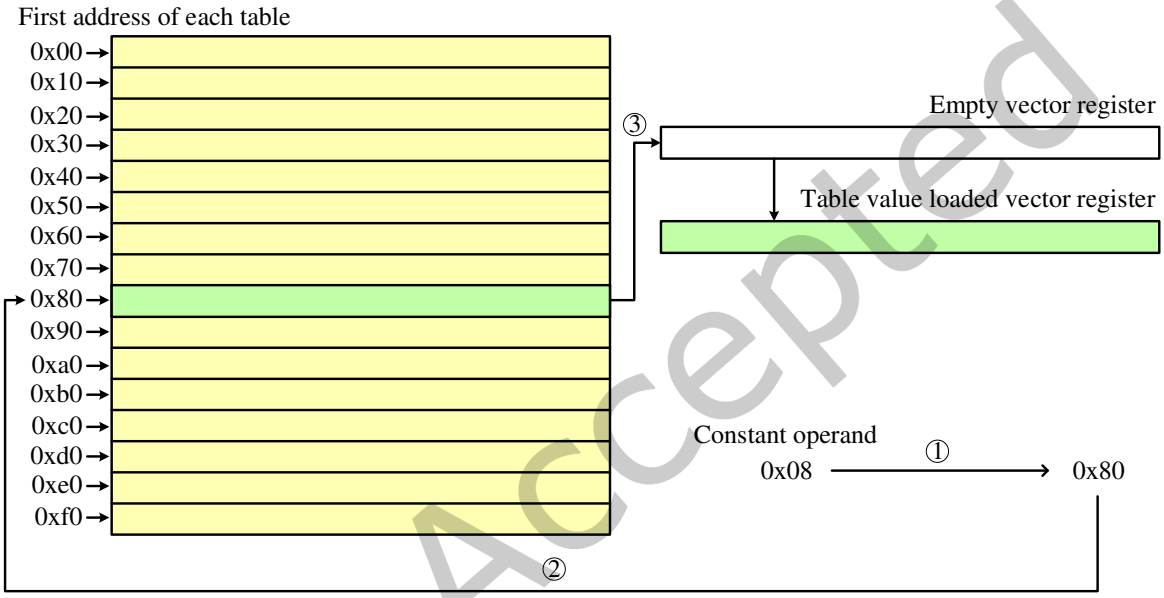

Fig. 5. The look-up table load/replacement technique. Step 1: Multiplying constant by 16. Step 2: checking the first address of the look-up table. Step 3: loading a 16 bytes table by referring to the address value.

Rainbow III and Rainbow V have 8-bit operands because they operate in the $\mathbb{F}_{256}$. 8-bit wise multiplications are performed, resulting in 65,536 calculation results and a table size of $65,536 \times 8$. However, this table is not used because it is too large size. Instead, Rainbow III and Rainbow V use Rainbow I look-up table and an additional 16-byte table. To perform the multiplication, each 8-bit value is decomposed into two 4-bit values for sub-field $\mathbb{F}_{16}$ operation. At this point, polynomial multiplication in $\mathbb{F}_{16}$ can be performed by table look-up. However, Rainbow III and Rainbow V also include 4-bit squaring, which requires an additional 16-byte look-up table for the squaring operation (shown in the last row of appendix Table 6). This makes the total size of Rainbow III and Rainbow V look-up table becomes 272-byte. After the $\mathbb{F}_{16}$ operations are completed, the remaining operations are performed.

The overall process is similar to the Rainbow I implementation. However, there are some differences which can be seen in Algorithm 6. In lines 1-11, table values are loaded. Rainbow III and Rainbow V use 8-bit operands. The constant is divided into 4-bit values, and then two table values are loaded, one lower and one higher. Finally, an additional 16-byte table is loaded for squaring. In line 14, a variable is set to adjust the index of the operand array. In lines 15-19, table look-up is performed in the same manner as in Rainbow I, but additional values are

Algorithm 4 Process of look-up table based polynomial multiplication on $\mathbb{F}_{16}$.

Input: Operand array A(4-bit||4-bit), accumulated array B, constant C, table address pointer P.
Output: Accumulated array B.

Initialize lower 8-bit of P to 0x00
$C \leftarrow C$ * 16
$P \leftarrow P + C$
$T[16] \leftarrow$ Load from P
Loop $\leftarrow |A|$ / 16
for i from 0 to Loop **do**
  j $\leftarrow$ i * 16
  A_{lower}[j~j+15] $\leftarrow A$[j~j+15] & 0x0f
  A_{higher}[j~j+15] $\leftarrow A$[j~j+15] » 4
  A_{lower}[j~j+15] $\leftarrow T[A_{lower}]$
  A_{higher}[j~j+15] $\leftarrow T[A_{higher}]$
  B[j~j+15] $\leftarrow B$[j~j+15] $\oplus$ (A_{lower} & (A_{higher} « 4))
end for
return B

Algorithm 5 Implementation code of polynomial multiplication on $\mathbb{F}_{16}$.

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

```
MOVI v31.16b, #15
ADR x4, MUL_TABLE
LSL w2, w2, #4
ADD x4, x4, x2
LD1.16b {v30}, [x4]
LD1.16b {v1}, [x1]
AND.16b v0, v1, v31
USHR.16b v1, v1, #4
TBL.16b v0, {v30}, v0
TBL.16b v1, {v30}, v1
SHL.16b v1, v1, #4
EOR.16b v0, v0, v1
LD1.16b {v1}, [x0]
EOR.16b v1, v1, v0
ST1.16b {v1}, [x0]
```

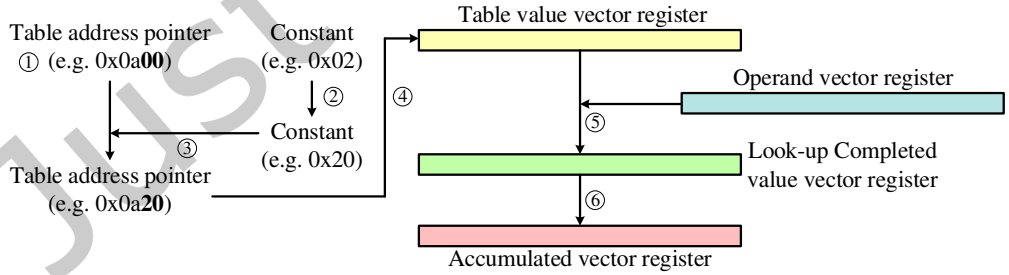

Fig. 6. Table-based polynomial multiplier operation process. Step 1: lower address of look-up table pointer is initialized to 0x00. Step 2: operand constant is multiplied by 16. Step 3: a table pointer is set by adding constant to table pointer. Step 4: 16 bytes of table value is loaded from address pointer. Step 5: table look-up is performed for the polynomial multiplication. Step 6: the result is accumulated.

Algorithm 6 Process of look-up table based polynomial multiplication on $\mathbb{F}_{256}$.

Input: Operand array A, accumulated array B, constant C, table address pointer P, additional table address pointer P_A.
Output: Accumulated array A.

1: $C_{lower} \leftarrow C$ & 0x0f
2: $C_{higher} \leftarrow C$ » 4
3: $C_{lower} \leftarrow C_{lower}$ * 16
4: $C_{higher} \leftarrow C_{higher}$ * 16
5: Initialize lower 8-bit of P to 0x00.
6: $P \leftarrow P + C_{lower}$
7: $T_{lower}[16] \leftarrow$ Load from P
8: Initialize lower 8-bit of P to 0x00.
9: $P \leftarrow P + C_{higher}$
10: $T_{higher}[16] \leftarrow$ Load from P
11: $T_{addi}[16] \leftarrow$ Load from P_A
12: Loop $\leftarrow |A|$ / 16
13: **for** i from 0 to Loop **do**
14: j $\leftarrow$ i * 16
15: A_{lower}[j~j+15] $\leftarrow A$[j~j+15] & 0x0f
16: A_{higher}[j~j+15] $\leftarrow A$[j~j+15] » 4
17: A_{square}[j~j+15] $\leftarrow A_{lower}$[j~j+15] $\oplus$ A_{higher}[j~j+15]
18: A_{lower}[j~j+15] $\leftarrow T_{lower}[A_{lower}]$
19: A_{higher}[j~j+15] $\leftarrow T_{higher}[A_{higher}]$
20: $C_{lower} \leftarrow C$ & 0x0f
21: $C_{higher} \leftarrow C$ » 4
22: $C_{square} \leftarrow C_{lower} \oplus C_{higher}$
23: Initialize lower 8-bit of P to 0x00.
24: $P \leftarrow P + C_{square}$
25: $T_{square}[16] \leftarrow$ Load from P_A
26: A_{square}[j~j+15] $\leftarrow T_{square}[A_{square}]$
27: A_{lower}[j~j+15] $\leftarrow A_{lower}$[j~j+15] $\oplus$ A_{square}[j~j+15]
28: A_{higher}[j~j+15] $\leftarrow T_{addi}[A_{higher}]$
29: B[j~j+15] $\leftarrow$ B[j~j+15] $\oplus$ (A_{lower} & (A_{higher} « 4))
30: **end for**
31: **return** B

also calculated in preparation for squaring. In lines 20-25, additional table values are loaded for squaring. In lines 26-27, squaring is performed using the look-up table and XORed with the lower 4-bit. In line 28, table look-up is performed for the higher 4-bit using the additional table. In line 29, the divided values are combined and the results are accumulated. In line 31, the results are returned.

Algorithm 7 is an implementation code for polynomial multiplication of Rainbow III and Rainbow V. Comparing lines 6 of Algorithm 4 with lines 14-15 of Algorithm 7, Rainbow III and Rainbow V load operands twice, as the minimum multiplication unit for Rainbow III and Rainbow V is 32-byte. In lines 1-11, the algorithm separates the operand into high/low 4-bit values and loads two 16-byte tables according to the separate operand. In lines 12-13, it loads an additional table for Rainbow III and Rainbow V. Then, the instructions load the operand array values

Algorithm 7 Implementation code of polynomial multiplication on $\mathbb{F}_{256}$.

Input: x0 = output array address A, x1 = operand array address 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, v31
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
```

and separate them into high/low 4-bit values (lines 14-19). In lines 20-23, the algorithm performs table look-up with the previous table, calculates intermediate values, and switches table according to intermediate result values in lines 24-32. In lines 33-36, it performs table look-up through the switched table. In lines 37-38, it performs table look-up through the additional table. In lines 39-44, it combines result values into 8-bit values. Finally, the results are accumulated into the output array in lines 45-51. Rainbow III and Rainbow V require intermediate value multiplication on $\mathbb{F}_{16}$, so table replacement occurs once, which is different from multiplication in Rainbow I.

3.4 Timing-attack resistance implementation

The straight-forward operand loading for table based approach may incur cache timing information. To remove this information, there would be two options. First option is disabling cache storage. This is secure but it leads to significant performance reduction. Second option is blinding the information of cache access. In this paper, we propose a technique to access cache every time to blind the cache access information. We access whole table values with minimum memory access. This loads whole values into cache efficiently and eliminates cache timing information efficiently. In order to achieve this goal, we utilized specific cache features of target processor. One of our target processor, the M1 processor, has a 128-byte cache line [22]. When one memory access to the aligned memory, it loads data into cache by 128-byte-wise (i.e. size of cache line). The total size of the table is 256-byte. We aligned memory address and only two times of memory accesses load full table values to cache. The table access is performed by 16-byte-wise data loading (vector register), which is stored in cache along with adjacent 128-byte values. In order to load remaining 128-byte (out of 256-byte) into the cache, we performed another memory access. This technique does not cause a time difference due to cache access because there are no cache misses. However, since it depends on the size of the cache line, it may be difficult to apply to caches other than 128-bytes. Algorithm 8 shows the code used for proposed implementation. In lines 1-4, it gets the 16-byte table

to use the same as algorithm 5. In lines 5-6, it restores the changed address value to its initial state. In line 7, it proceeds XOR `0x8` to the offset value. In lines 8-10, it loads the dummy table according to the modified offset value. The reason for XOR `0x8` is that the offset values range from 0 to 15. If the offset value is from 0 to 7, the value of the front part 128-byte is loaded into the cache when loading the 16-byte table to vector register. The value of the back part 128-byte is loaded into the cache when it is 8 to 15. Therefore, applying XOR `0x8` to a offset value enables caching of all 256-byte tables, regardless of the selected offset value.

Algorithm 8 Implementation code of cache side-attack resistance implementation.

Input: x2(w2) = constant *C*.

```
1:  ADR x4, MUL_TABLE
2:  LSL w2, w2, #4
3:  ADD x4, x4, x2
4:  LD1.16b {v30}, [x4]
5:  SUB x4, x4, x2
6:  ROR w2, w2, #4
7:  XOR w2, w2, #8
8:  LSL w2, w2, #4
9:  ADD x4, x4, x2
10: LD1.16b {v27}, [x4]
```

We also proposed constant-time implementation, which named atomic approach. In generally, the table values are stored in registers, and an `if-else` branch statement is used to determine which table value to use based on the operand constant. However, the execution time can vary when regular branch statements are used, depending on the order in which the conditions are checked. As a result, performance can vary based on the operand value. To address this issue, we designed an atomic approach that ensures consistent execution time across different operands.

The atomic approach works as follows: the input constant ranges from 0 to 15. The first condition checks if it is greater or less than half of the initial constant range of 8. If it is less than 8, values 0 to 7 move to the second condition, and so on, until it is possible to determine which table to use based on the current constant value. The implementation code is shown in Algorithm 9.

To use this approach, we load all table values of 256 bytes into the registers at the beginning of the code. Then, we check if the input constant value fits the first condition and execute the branch syntax according to the condition. At this point, two branch syntaxes may be executed, but the second one is ignored when the first branch syntax is executed. This can result in a difference in execution time. For this reason, we added a dummy branch syntax to match the execution time when the first branch syntax was executed [18]. This process is repeated to determine which table value to use based on the input constant. This process executes only a total of 14 instructions, including 4 `CMP`, 9 `B`, and 1 `MOV`, regardless of the input value. Figure 3 showed the registers used in the proposed implementation. The constant time implementation uses registers `v0-v15` in the same way as the optimized implementation, but registers `v16-v31` are used for table storage. In the optimized implementation, register `v30` was used to load look-up table values, and in the constant time implementation, it is used for the same purpose. Register `v31` is used to store table values and the `AND` instruction used to extract 4-bit values in the optimized implementation are replaced with `SHL` and `USHR` instructions in the constant time implementation.

4 EVALUATION

The implementations were evaluated on three target processors: Apple M1 (iPad Pro 12.9 5th gen.), Apple A13 bionic (iPhone SE 2nd gen.), and Broadcom BCM2711 (Raspberry Pi 4 Model B). These processors operate at a maximum frequencies of 3.2GHz, 2.65GHz, and 1.5GHz, respectively.

The evaluation was done using the Xcode and Visual Studio Code framework, and the code was compiled with the `-O3` optimization option for maximum speed. The comparison was made with the implementation code of

Algorithm 9 Implementation code of constant time (atomic) implementation.

Input: x2(w2) = constant C, x4 = look-up table address.
Output: 4-bit accumulated output to A.

```
ADR x4, MUL_TABLE
LD1.16b {v16}, [x4], #16
Code omitted.
LD1.16b {v31}, [x4], #16
CMP w2, #8
BLT MUL_07
B MUL_815
MUL_BACK:
Code omitted.
RET
MUL_07:
B // Dummy
CMP w2, #4
BLT MUL_03
B MUL_47
MUL_03:
B // Dummy
CMP w2, #2
BLT MUL_01
B MUL_23
MUL_01:
B // Dummy
CMP w2, #1
BEQ MUL_1
B MUL_0
MUL_23:
CMP w2, #3
BEQ MUL_3
B MUL_2
MUL_47:
CMP w2, #6
BLT MUL_45
B MUL_67
MUL_45:
B // Dummy
CMP w2, #5
BEQ MUL_5
B MUL_4
MUL_67:
CMP w0, #7
BLT MUL_811
B MUL_1215
MUL_815:
BLT MUL_811
B MUL_1215
MUL_811:
B // Dummy
CMP w2, #10
BLT MUL_89
B MUL_1011
MUL_89:
B // Dummy
CMP w2, #9
BEQ MUL_9
B MUL_8
MUL_1011:
CMP w2, #11
BEQ MUL_11
B MUL_10
MUL_1215:
CMP w2, #14
BLT MUL_1213
B MUL_1415
MUL_1213:
B // Dummy
CMP w2, #13
BEQ MUL_13
B MUL_12
MUL_1415:
CMP w2, #15
BEQ MUL_15
B MUL_14
MUL_0:
MOV v30.16b, v16.16b
B MUL_BACK
MUL_1:
B // Dummy
MOV v30.16b, v17.16b
B MUL_BACK
MUL_2:
MOV v30.16b, v18.16b
B MUL_BACK
MUL_3:
B // Dummy
MOV v30.16b, v19.16b
B MUL_BACK
MUL_4:
MOV v30.16b, v20.16b
B MUL_BACK
MUL_5:
B // Dummy
MOV v30.16b, v21.16b
B MUL_BACK
MUL_6:
MOV v30.16b, v22.16b
B MUL_BACK
MUL_7:
B // Dummy
MOV v30.16b, v23.16b
B MUL_BACK
MUL_8:
MOV v30.16b, v24.16b
B MUL_BACK
MUL_9:
B // Dummy
MOV v30.16b, v25.16b
B MUL_BACK
MUL_10:
MOV v30.16b, v26.16b
B MUL_BACK
MUL_11:
B // Dummy
MOV v30.16b, v27.16b
B MUL_BACK
MUL_12:
MOV v30.16b, v28.16b
B MUL_BACK
MUL_13:
B // Dummy
MOV v30.16b, v29.16b
B MUL_BACK
MUL_14:
MOV v30.16b, v30.16b
B MUL_BACK
MUL_15:
B // Dummy
MOV v30.16b, v31.16b
B MUL_BACK
```

Rainbow from PQClean library[1]. Our proposed Rainbow was also based on this reference code. The performance evaluation was conducted in two stages. The first stage compared the performance of the multiplication algorithm and proposed table-based parallel multiplication algorithm. The second stage compared the performance of the Rainbow and the proposed optimized Rainbow. Finally, we proceed with the performance evaluation of the Timing-attack resistance implementation.

4.1 Evaluation of multiplier

The first comparison measured the performance of the multipliers. The algorithms were tested by inputting a length of 512-byte and repeating the process 1 million times to calculate the operation time. The results are shown in Table 3. The previous $\mathbb{F}_{16}$ multiplier took 355 clock cycles to run on the M1 processor, while the

[1]https://github.com/PQClean/PQClean

proposed table-based parallel multiplier only took 58 clock cycles, providing a 6.12× better performance for $\mathbb{F}_{16}$ multiplication compared to the reference implementation. On the A13 processor, the proposed multiplier improved $\mathbb{F}_{16}$ multiplication by 5.78×. The largest improvement was seen on the BCM2711 with a performance improvement of 428.2× for $\mathbb{F}_{16}$ multiplication. The $\mathbb{F}_{256}$ multiplier showed similar results with the proposed multiplier performing 167.2× better on the M1 processor, 135.85× better on the A13 processor, and 428.73× better on the BCM2711 compared to the previous implementation.

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

Processor	Algorithm	$\mathbb{F}_{16}$ multiplier	$\mathbb{F}_{256}$ multiplier
M1	Previous impl. [7]	355	16,557
	This work	**58**	**99**
A13	Previous impl. [7]	445.2	17,254
	This work	**77**	**127**
BCM2711	Previous impl. [7]	132324	221,225
	This work	**309**	**516**

4.2 Evaluation of Rainbow signature

The performance of the Rainbow signature was also evaluated. The previous implementation was repeated 300 times while the proposed method was repeated 10,000 times. This difference in iterations was due to the previous implementation too long to run with a large number of iterations and the proposed implementation ending too quickly with a small number of iterations. To ensure maximum CPU performance, the number of iterations for the proposed technique increased. Note that. The CPU was not overclocked in the experiment. In other words, the maximum performance of the CPU follows the part specified in each CPU official specification. The results are shown in Table 4.

When comparing the implementations on the M1 processor, the proposed technique showed better performance in key scheduling, signature generation, and verification for Rainbow I Classic by 15.9×, 10.19×, and 47.14×, respectively. For Rainbow I Circumzenithal, the proposed implementation had 16.69×, 6.79×, and 1.35× better performance than the reference implementation. In Rainbow I Compressed, the proposed method had 16.67×, 9.83×, and 1.35× better performance. The largest improvement was seen in the verification process of the Rainbow I Classic version, which showed 47.14× better performance than the reference implementation.

The performance evaluation of Rainbow III is similar. The proposed implementation showed higher performance than the reference for key scheduling, signature creation, and verification. In the Classic version, the proposed implementation performed 35.64×, 13.96×, and 4.89× better, respectively, compared to the reference. In the Circumzenithal version, the proposed method was 38.13×, 13.89×, and 13.74× faster. And in the Compressed version, the performance differences were 38.13×, 22.48×, and 1.37×.

For Rainbow V, the proposed technique performed better for key scheduling, signature creation, and verification compared to the reference implementation. In the Classic version, the proposed method was 16.66×, 24.85×, and 23.20× faster. In the Circumzenithal version, the performance improvement was 18.07×, 24.45×, and 1.47×. And in the Compressed version, the proposed technique performed 18.07×, 17.33×, and 1.47× better.

In conclusion, the verification process of Rainbow I Classic showed the highest performance improvement of 47.14×. The results on the A13 processor show a similar trend as those on the M1 processor. However, on the BCM2711, the performance gap is much larger with the best case being 114.16× faster in the Rainbow I

Table 4. Evaluation results of Rainbow signature (unit: $\times 10^6$ clock cycles).

Level	Processor	Algorithm	Previous implementation [7]			**This work**		
			Key Scheduling	Signature	Verification	**Key Scheduling**	**Signature**	**Verification**
I	M1	Classic	253.08	3.16	3.3	**15.91**	**0.31**	**0.07**
		CZ	281.77	3.19	12.41	**16.88**	**0.47**	**9.18**
		Compressed	281.79	127.71	12.42	**16.9**	**13.0**	**9.18**
	A13	Classic	264.93	3.31	3.42	**16.35**	**0.31**	**0.07**
		CZ	295.22	3.29	12.5	**18.1**	**0.35**	**9.23**
		Compressed	314.51	145.49	13.56	**20.9**	**13.63**	**10.68**
	BCM2711	Classic	3,285.6	42.12	42.24	**126.97**	**3.54**	**0.37**
		CZ	3,633.3	42.6	127.35	**135.36**	**3.6**	**82.56**
		Compressed	3,676.95	1,679.7	125.1	**132.9**	**111.95**	**87.29**
III	M1	Classic	3,141.06	27.65	28.67	**88.13**	**1.98**	**5.86**
		CZ	3,574.48	27.65	837.55	**93.73**	**1.99**	**60.97**
		Compressed	3,573.15	1,693.31	83.71	**93.7**	**75.34**	**60.97**
	A13	Classic	3,798.01	28.89	299.55	**120.87**	**2.1**	**6.13**
		CZ	4,382.05	32.18	93.02	**120.05**	**2.39**	**75.92**
		Compressed	4,240.08	1,925.94	84.89	**129.83**	**98.71**	**76.73**
	BCM2711	Classic	4,2393	432.0	381.0	**722.6**	**27.02**	**79.79**
		CZ	4,8162	381.0	892.5	**738.32**	**26.91**	**600.17**
		Compressed	48,588	23,155.5	892.5	**742.37**	**635.93**	**577.89**
V	M1	Classic	8,830	61.12	62.4	**530.14**	**2.46**	**2.69**
		CZ	10,140	61.12	186.66	**561.18**	**2.5**	**127.17**
		Compressed	10,140	4,852.0	186.88	**561.06**	**279.94**	**127.14**
	A13	Classic	9,670.56	63.87	65.46	**659.85**	**3.2**	**3.45**
		CZ	12,732	75.17	207.23	**738.53**	**3.17**	**137.32**
		Compressed	12,050	6,204.6	220.66	**721.79**	**322.7**	**134.48**
	BCM2711	Classic	118,996	843.0	850.5	**6,199.65**	**34.65**	**37.35**
		CZ	136,492	843.0	1,971	**6,596.7**	**34.5**	**1,212.3**
		Compressed	137,713	66,393	1,980	**6,499.2**	**2,965.2**	**1,233.75**

Compressed verification. The proposed method effectively reduces computation time, making it applicable to low-end processors as well as high-end processors.

4.3 Evaluation of timing-attack resistance variant implementation

Two types of variant implementations (i.e. cache side attack resistant and constant time) are evaluated. Table 5 shows the experimental results, which are obtained on an Apple M1 processor under the same conditions as the optimized Rainbow signature. Overall, the cache side-attack resistance implementation shows similar performance to the proposed optimal implementation.

However, the constant-time implementation exhibits about 11%~13% lower performance compared to the optimized implementation. However, due to its fast basic operation speed, it still achieves a significant performance improvement compared to the reference implementation.

5 CONCLUSION

In this paper, we proposed a table-based polynomial multiplication technique to improve the performance of Rainbow signatures. The previous implementations use efficient Karatsuba algorithm for polynomial multiplication,

Table 5. Evaluation results of timing-attack resistance implementations (unit: $\times 10^6$ clock cycles).

Level	Processor	Algorithm	Cache side attack resistance			Constant time implementation		
			Key Scheduling	Signature	Verification	Key Scheduling	Signature	Verification
I	M1	Classic	16.47	0.40	0.09	27.13	0.37	0.12
		CZ	17.20	0.51	9.95	29.70	0.37	8.82
		Compressed	17.07	13.07	9.78	29.53	18.76	8.83
III		Classic	89.04	2.01	5.93	-	-	-
		CZ	94.51	2.01	62.62	-	-	-
		Compressed	94.40	76.20	62.64	-	-	-
V		Classic	531.39	2.46	2.72	-	-	-
		CZ	577.15	2.53	129.63	-	-	-
		Compressed	575.97	282.05	129.70	-	-	-

but this takes a long time to execute due to the large size of Rainbow parameters. Our proposed table look-up based multiplication technique reduced the computational load, with a look-up table size of 256 or 272-byte.

On the BCM2711 processor, the performance of the multiplier improved by up to 428.73×. This improvement was reflected in the Rainbow signatures, with a best-case improvement of 114.16×. our proposed technique utilized vector registers and vector instructions, which allow for parallel operations, leading to improved performance on the target processors of Apple M1, Apple A13 bionic, and Cortex-A72. We also presented two variant implementations that resist timing-attack. Both implementations are resistant to timing-attack, but the cache side-attack resistant implementation has a dependency on cache-line and the constant time implementation is only applicable to Rainbow I. As future work, we suggest the implementation of other post-quantum cryptography on the latest M1 processor and other ARMv8 processors, or using a different processor such as RISC-V [10, 16].

6 ACKNOWLEDGEMENT

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, 50%) and 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%).

REFERENCES

[1] Daniel J Bernstein and Tung Chou. 2014. Faster binary-field multiplication and faster binary-field MACS. In *International Conference on Selected Areas in Cryptography*. Springer, 92–111.
[2] Ward Beullens. 2022. Breaking rainbow takes a weekend on a laptop. *Cryptology ePrint Archive* (2022).
[3] Amit Kumar Chauhan and Somitra Kumar Sanadhya. 2020. Quantum resource estimates of grover's key search on aria. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 238–258.
[4] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang, and Chen-Mou Cheng. 2018. Implementing 128-bit secure MPKC signatures. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 101, 3 (2018), 553–569.
[5] Tung Chou, Matthias J Kannwischer, and Bo-Yin Yang. 2021. Rainbow on Cortex-M4. *IACR Cryptol. ePrint Arch.* 2021 (2021), 532.
[6] Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. 2020. Rainbow - Algorithm Specification and Documentation The 3rd Round Proposal. https://troll.iis.sinica.edu.tw/by-publ/recent/Rainbow3round.pdf.
[7] Jintai Ding and Dieter Schmidt. 2005. Rainbow, a new multivariable polynomial signature scheme. In *International conference on applied cryptography and network security*. Springer, 164–175.
[8] Gwang-Sik Kim and Young-Sik Kim. 2021. Efficient Implementation of Finite Field Operations in NIST PQC Rainbow. *Journal of the Korea Institute of Information Security & Cryptology* 31, 3 (2021), 527–532.
[9] Aviad Kipnis, Jacques Patarin, and Louis Goubin. 1999. Unbalanced Oil and Vinegar signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 206–222.

[10] Hyeokdong Kwon, Hyunjun Kim, Siwoo Eum, Minjoo Sim, Hyunji Kim, Wai-Kong Lee, Zhi Hu, and Hwajeong Seo. 2021. Optimized Implementation of SM4 on AVR Microcontrollers, RISC-V Processors, and ARM Processors. *Cryptology ePrint Archive* (2021).
[11] Jongbok Lee. 2021. VHDL Design for Out-of-Order Superscalar Processor of A Fully Pipelined Scheme. *The Journal of the Institute of Internet, Broadcasting and Communication* 21, 1 (2021), 99–105.
[12] Ivan Lin, Brian Jeff, and Ian Rickard. 2016. ARM platform for performance and power efficiency—Hardware and software perspectives. In *2016 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, 1–5.
[13] Duc Tri Nguyen and Kris Gaj. 2021. Optimized Software Implementations of CRYSTALS-Kyber, NTRU, and Saber Using NEON-Based Special Instructions of ARMv8. (2021).
[14] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. 2010. CyclicRainbow–a multivariate signature scheme with a partially cyclic public key. In *International Conference on Cryptology in India*. Springer, 33–48.
[15] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. 2021. Kyber on ARM64: compact implementations of Kyber on 64-bit ARM Cortex-A processors. In *International Conference on Security and Privacy in Communication Systems*. Springer, 424–440.
[16] Hwajeong Seo, Hyeokdong Kwon, Kyoungbae Jang, and Hyunjun Kim. 2021. 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* 31, 3 (2021), 473–480.
[17] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. 2018. SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 1–20.
[18] Seog Chung Seo and HeeSeok Kim. 2019. SCA-resistant GCM implementation on 8-Bit AVR microcontrollers. *IEEE Access* 7 (2019), 103961–103978.
[19] Silvan Streit and Fabrizio De Santis. 2017. Post-quantum key exchange on ARMv8-A: A new hope for NEON made simple. *IEEE Trans. Comput.* 67, 11 (2017), 1651–1662.
[20] Purab Surana, Nishil Madhani, and T Gopalakrishnan. 2020. A Comparative Study on the Recent Smart Mobile Phone Processors. In *2020 7th International Conference on Smart Structures and Systems (ICSSS)*. IEEE, 1–3.
[21] Rainbow Team. 2020. Modified parameters of Rainbow in response to a refined analysis of the Rainbow band separation attack by the NIST team and the recent new MinRank attacks.
[22] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W Fletcher, and David Kohlbrenner. 2022. Augury: Using data memory-dependent prefetchers to leak data at rest. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1491–1505.

A APPENDIX

A.1 Look-up Table Information

Table 6 represents multiplication table utilized in the proposed implementation. While each value of the table is 4-bit, the minimum data storage unit is 8-bit. Therefore, the total size of the table is 256-byte. The `addi` row is an additional table and is employed in Rainbow III and Rainbow V operation. For a detailed description of the table, recommend referring Section 3.

Table 6. Pre-calculation result table of 4 bits tower-field polynomial multiplication results on $\mathbb{F}_{16}$ with hexadecimal notation. 'Addi' row means additional table for Rainbow III and Rainbow V.

*	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0
0x1	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x2	0x0	0x2	0x3	0x1	0x8	0xa	0xb	0x9	0xc	0xe	0xf	0xd	0x4	0x6	0x7	0x5
0x3	0x0	0x3	0x1	0x2	0xc	0xf	0xd	0xe	0x4	0x7	0x5	0x6	0x8	0xb	0x9	0xa
0x4	0x0	0x4	0x8	0xc	0x6	0x2	0xe	0xa	0xb	0xf	0x3	0x7	0xd	0x9	0x5	0x1
0x5	0x0	0x5	0xa	0xf	0x2	0x7	0x8	0xd	0x3	0x6	0x9	0xc	0x1	0x4	0xb	0xe
0x6	0x0	0x6	0xb	0xd	0xe	0x8	0x5	0x3	0x7	0x1	0xc	0xa	0x9	0xf	0x2	0x4
0x7	0x0	0x7	0x9	0xe	0xa	0xd	0x3	0x4	0xf	0x8	0x6	0x1	0x5	0x2	0xc	0xb
0x8	0x0	0x8	0xc	0x4	0xb	0x3	0x7	0xf	0xd	0x5	0x1	0x9	0x6	0xe	0xa	0x2
0x9	0x0	0x9	0xe	0x7	0xf	0x6	0x1	0x8	0x5	0xc	0xb	0x2	0xa	0x3	0x4	0xd
0xa	0x0	0xa	0xf	0x5	0x3	0x9	0xc	0x6	0x1	0xb	0xe	0x4	0x2	0x8	0xd	0x7
0xb	0x0	0xb	0xd	0x6	0x7	0xc	0xa	0x1	0x9	0x2	0x4	0xf	0xe	0x5	0x3	0x8
0xc	0x0	0xc	0x4	0x8	0xd	0x1	0x9	0x5	0x6	0xa	0x2	0xe	0xb	0x7	0xf	0x3
0xd	0x0	0xd	0x6	0xb	0x9	0x4	0xf	0x2	0xe	0x3	0x8	0x5	0x7	0xa	0x1	0xc
0xe	0x0	0xe	0x7	0x9	0x5	0xb	0x2	0xc	0xa	0x4	0xd	0x3	0xf	0x1	0x8	0x6
0xf	0x0	0xf	0x5	0xa	0x1	0xe	0x4	0xb	0x2	0xd	0x7	0x8	0x3	0xc	0x6	0x9
addi	0x0	0x8	0xc	0x4	0xb	0x3	0x7	0xf	0xd	0x5	0x1	0x9	0x6	0xe	0xa	0x2