Look-up the Rainbow

Efficient Table-based Parallel Implementation of Rainbow Signature on 64-bit ARMv8 Processors

2021 국가암호공모전

Opening

- Target cipher
- Target environment

MiddleGame

- Core point of optimization
- Table based multiplication
- Parallel implementation
- Evaluation

EndGame

- Conclusion
- Future works

2021 국가암호공모전

Opening

Look-up the Rainbow

Efficient Table-based Parallel Implementation of Rainbow Signature on 64-bit ARMv8 Processors

2021 국가암호공모전

Target cipher

- 양자 내성 암호의 **원활한 사용·보급**을 위해 양자 내성 암호 최적 구현을 주제로 선택
- Round 3 **Finalist** 중 하나인 **Rainbow**를 선정
  - Signature 분야에는 Rainbow 포함 3개의 알고리즘이 선정
  - Rainbow의 매개변수는 [표 1]을 따름

Type	Security Level	Parameters	Public key size	Private key size	Signature size
STD	I	(GF(16), 36, 32, 32)	157.8 KB	101.2 KB	528 bit
	III	(GF(256), 68, 32, 48)	861.4 KB	611.3 KB	1,312 bit
	V	(GF(256), 96, 36, 64)	1,885.4 KB	1,357.7 KB	1,632 bit
CZ	I	(GF(16), 36, 32, 32)	58.8 KB	101.2 (99.0) KB	528 bit
	III	(GF(256), 68, 32, 48)	258.4 KB	611.3 (603.0) KB	1,312 bit
	V	(GF(256), 96, 36, 64)	523.5 KB	1,357.7 (1,361.8) KB	1,696 bit

Table 1. Parameters of Rainbow signature

Opening:

Target environment

- **64-bit ARMv8**을 대상 프로세서로 선정
  - 광범위하게 사용되는 고성능 프로세서
  - Vector instruction을 통한 병렬 구현
  - ARMv8 아키텍처가 적용된 최신 Apple M1 chip을 사용
  - 기본적인 설정은 [그림 1]을 따름

- Rainbow **III, V**는 ARMv8 상에서 **최초로 최적 구현 시도**
  - 기존 ARMv8 상에서의 Rainbow 최적 구현은 I에 대해서만 시도 되었음
  - 본 제안에서는 **III, V 구현** 뿐만 아니라, **Cyclic, Compressed**에 대해서 모두 구현

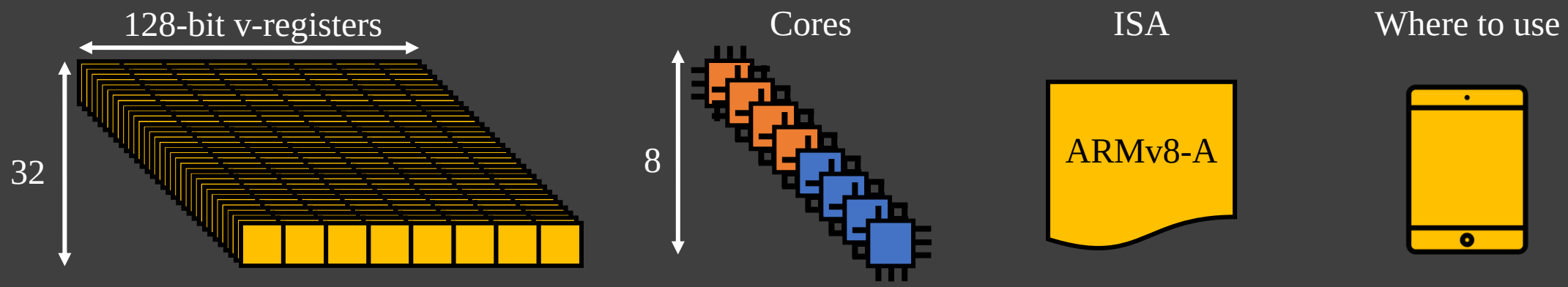


Figure 1. Specification of Apple M1

MiddleGame

Look-up the Rainbow

Efficient Table-based Parallel Implementation of Rainbow Signature on 64-bit ARMv8 Processors

2021 국가암호공모전

MiddleGame:

Core point of optimization

- Rainbow signature는 **Tower-field 기반의 Karatsuba** 알고리즘을 사용
  - [그림 2]를 기반으로 [그림 3] 코드를 따라서 구현
- GF(16) 연산 → GF(4) 연산 → GF(2) 연산 반복
- 여러 단계를 반복하여 비효율적인 연산
  - Matrix 크기가 크기 때문에 효율이 더욱 감소
- 입력 값은 matrix와 constant로 구분
  - Matrix는 다수의 값 존재
  - Constant는 **1개**의 값

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

Figure 2. Tower-field based multiplication

```
static inline uint8_t gf16_mul(uint8_t a, uint8_t b) {
    uint8_t a0 = a & 3;
    uint8_t a1 = (a >> 2);
    uint8_t b0 = b & 3;
    uint8_t b1 = (b >> 2);
    uint8_t a0b0 = gf4_mul(a0, b0);
    uint8_t a1b1 = gf4_mul(a1, b1);
    uint8_t a0b1_a1b0 = gf4_mul(a0 ^ a1, b0 ^ b1) ^ a0b0 ^ a1b1;
    uint8_t a1b1_x2 = gf4_mul_2(a1b1);
    return ((a0b1_a1b0 ^ a1b1) << 2) ^ a0b0 ^ a1b1_x2;
}
static inline uint8_t gf4_mul_2(uint8_t a) {
    uint8_t r = a << 1;
    r ^= (a >> 1) * 7;
    return r;
}

static inline uint8_t gf4_mul(uint8_t a, uint8_t b) {
    uint8_t r = a * (b & 1);
    return r ^ (gf4_mul_2(a) * (b >> 1));
}
```

Figure 3. Multiplication algorithm of Rainbow I signature

MiddleGame:
Core point of optimization

- ARMv8 프로세서에는 **polynomial multiplication**을 제공하는 명령어가 존재
  - Assembly 명령어 중, PMUL 또는 PMULL
- Rainbow signature는 **Tower-field** 기반 연산이므로 사용 불가
  - 조건부 사용 가능 → GF(4) 상에서 carry가 존재하지 않을 때
  - Rainbow III, V의 경우 → GF(16), GF(4) 상에서 carry가 없어야 사용 가능
  - 결과적으로 PMUL(PMULL)로 구현은 할 수 없음
- 입력 값을 **2-bit 단위로 분리**하여 PMUL(PMULL) 사용시 구현 가능
  - Rainbow III, V의 경우 → 4-bit로 분리 → 2-bit로 분리
  - 기존 구현물과 같은 형태 → 비효율적
- 새로운 유형의 곱셈기를 제안
  - 테이블 기반의 곱셈

MiddleGame:
Table based multiplication

- GF(16)의 값은 4-bit로 저장되며, 16가지의 값을 표현 가능
- GF(16)상에서 4-bit간의 곱셈의 가짓수는 **총 256가지**
- 256가지 유형의 곱셈을 연산하여 Look-up table(LUT) 생성
  - 생성한 LUT는 [표 2]와 같음
- 결과 값은 4-bit이지만, 데이터를 저장하는 최소 크기는 8-bit(1-byte)
  - 전체 가짓수가 256이므로, **LUT의 전체 크기는 256-byte**
- 곱셈 연산을 진행하는 대신, **Table look-up을 통해 결과 값을 획득**
  - Sub-field로 이동할 필요가 없으므로, 연산 시간 대폭 감소
  - 기기의 메모리 중 256-byte만 사용하기 때문에 메모리 소모 측면에서 경제적
- Rainbow III, V의 경우, 16-byte의 추가 LUT가 요구됨
  - 총 272-byte가 필요

MULTABLE

*	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

ADDI TABLE

Table 2. Look-up table for proposed technique, ‘addi’ is additional table values for Rainbow III, V

MiddleGame:

Table based multiplication

- 구현을 위해 [그림 4]와 같이 레지스터 확보
  - Additional 레지스터는 Rainbow III, V 구현시에만 사용

- 입력 값이 ‘Matrix * Constant’ 형태이므로, 결과 값 중 **16개만 필요**
  - Table 호출 시, **256-byte를 모두 호출하지 않고, 16-byte만 호출**
  - Table 호출 방법은 [그림 5]의 알고리즘을 따름

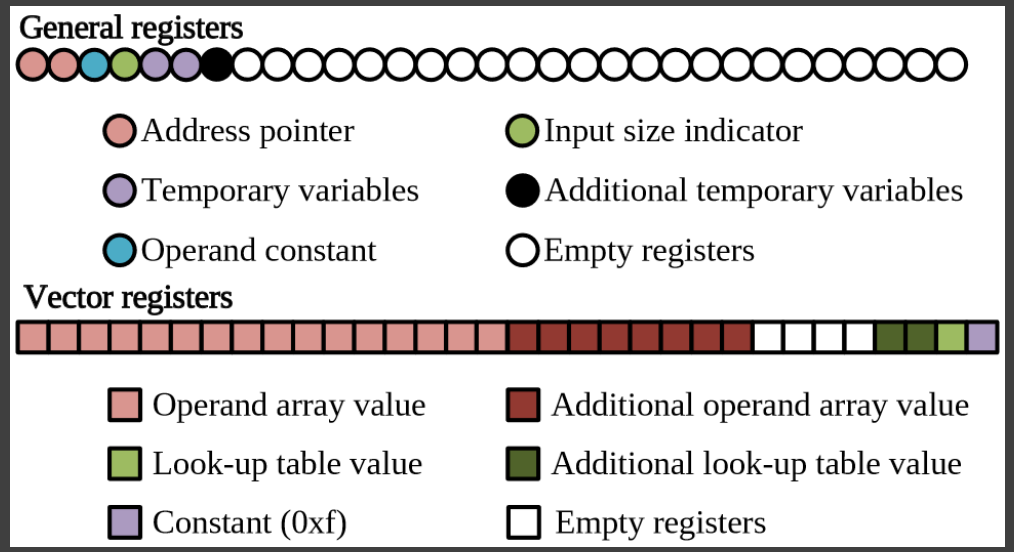


Figure 4. Register scheduling plan

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

```
1: P ← first address of LUT
2: C ← C × 16
3: P ← P + C
4: return P
```

Figure 5. Pseudocode of table address setting method

MiddleGame:

Table based multiplication

- Table 호출이 완료 되었으면, 각 **Matrix가 가진 값을 확인 후 Look-up 진행**
  - [그림 6]과 같은 형태로 도식화, [그림 7]의 코드 사용

- ARMv8에서 제공하는 **Assembly 명령어**를 사용
  - C 코드에 비해 **매우 빠른 속도**로 동작
  - 원하는 레지스터를 사용하는 것으로 효율적인 레지스터 관리

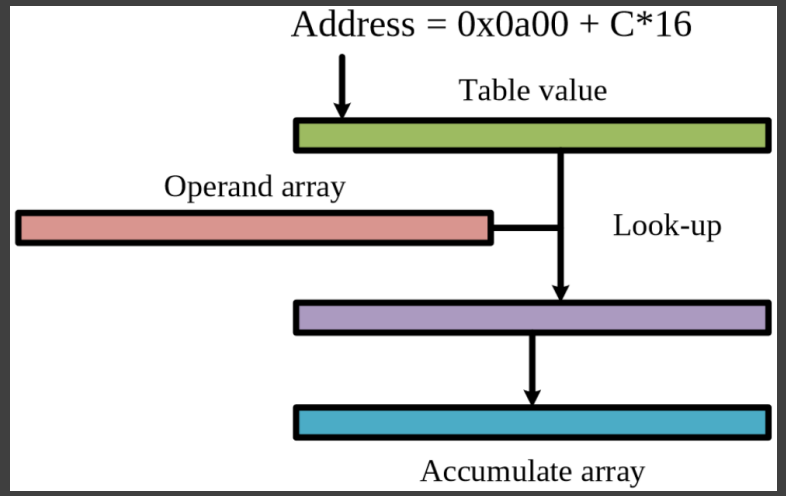


Figure 6. Table-based polynomial multiplier operation process

```
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]
```

Figure 7. Implementation code of table look-up based polynomial multiplication on GF(16)

MiddleGame:

Table based multiplication

- Rainbow III, V의 경우 **추가적인 단계**가 존재
  1. 4-bit 단위로 분할 → Table이 4-bit 계산에 맞춰져 있기 때문
  2. Additional table 사용

- [그림 8]과 같이 추가 사항이 있는 코드 사용
  - 원본 값이 8-bit이므로 추가 연산이 늘어남

- **GF(256)용 Table을 생성하지 않음**
  - Operand 값 종류 256가지

  → 곱셈 결과 값 종류 65,536가지
  - **Table의 크기가 64MB**로 과도하게 큼
  - Vector register의 사이즈 제약 16-byte
  - Table 저장에 16개의 vector register 요구

  → Table 로드에 과도한 시간 소요

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
```

Figure 8. Implementation code of GF(256) multiplication

MiddleGame:

Parallel implementation

- Vector register / instruction은 **병렬 연산을 지원**
- 하나의 register에 대해 하나의 instruction으로 다량의 데이터 생성이 가능
  - Vector register = 128-bit, Variable = 8-bit → **16개 데이터 저장 가능**
  - **1개의 명령어로 16개의 output 생성**
- Vector register / instruction을 사용한 동작은 [그림 9]로 도식화
- C언어로 구현된 기존 구현물에 비해 **더 적은 loop로 연산 종료**

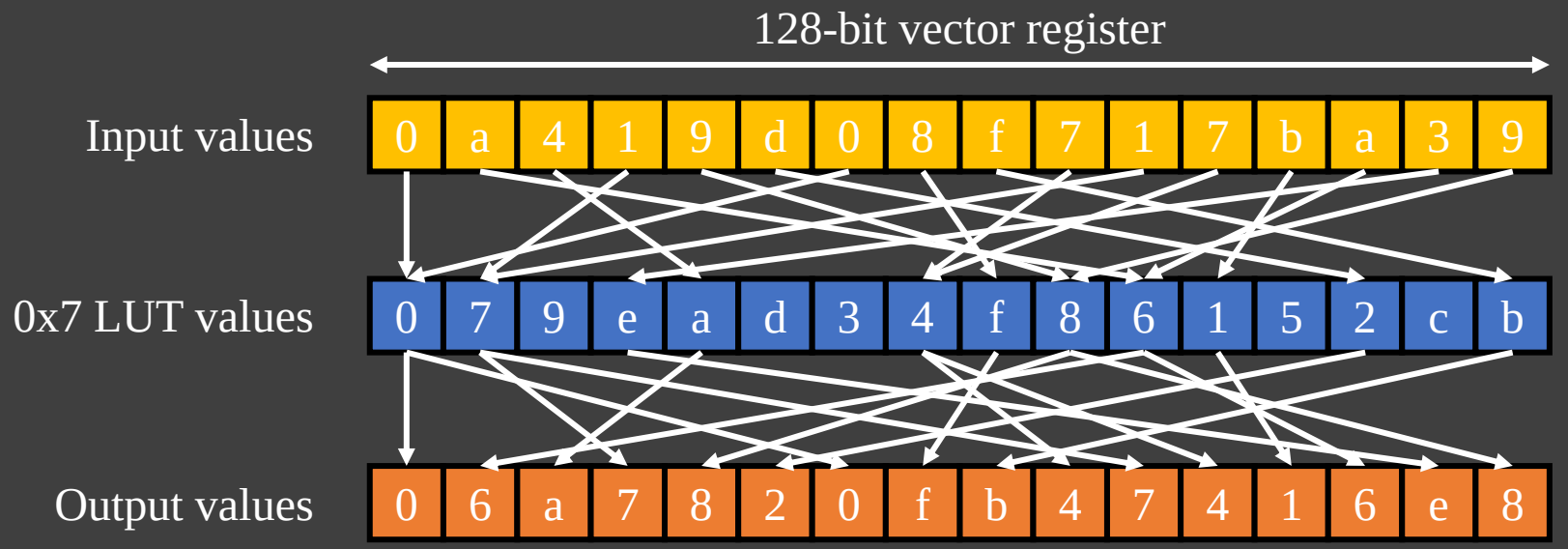


Figure 9. Schematic of parallel table look-up

Parallel implementation

- Rainbow signature의 matrix 크기는 일정하게 고정
- 하지만 내부 연산 과정에서 일부 변수는 일정하지 않은 matrix 크기 보유
- 모든 경우를 고려하기 위해 **arrangement specifier** 활용을 고려
- Arrangement specifier에 따라서 vector register의 packing 단위가 변화
  - 16b: 16-bytes / 8b: 8-bytes
  - 8h: 8-half word / 4h: 4-half word
  - [그림 10]과 같은 형태로 확인 가능
- Rainbow 이외에 다른 암호 알고리즘 구현 시에 코드 활용이 용이함
  - 코드의 재사용, 유지보수성 상승

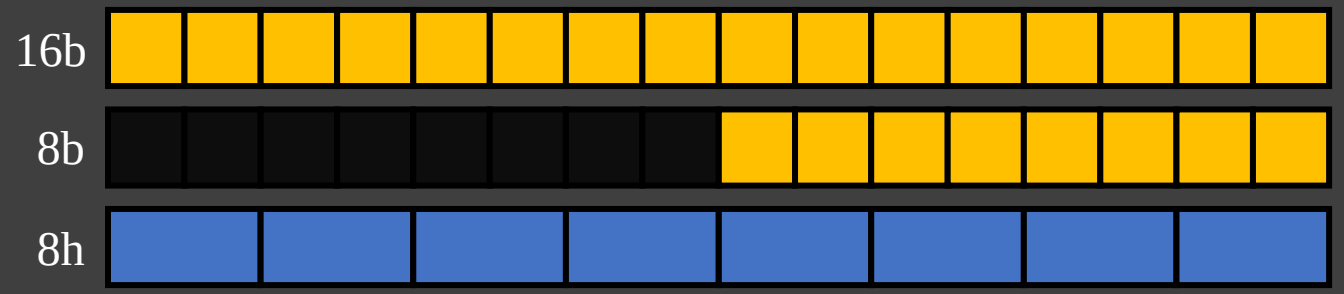


Figure 10. State of vector registers according to arrangement specifier

MiddleGame: Evaluation

- 구현 환경
  - Machine: Apple iPad Pro Gen. 5 with Apple M1 chip (3.2GHz)
  - Framework: Xcode Integrated Development Environment
  - Compiler: LLVM compiler with **–O3(fastest)** option

- 평가 방법
  - 곱셈기를 분리한 다음 기존 곱셈기와 제안하는 곱셈기의 성능 평가
  - GF(16), GF(256) 곱셈기를 별개로 비교 → 총 2개의 비교 결과
  - 곱셈기를 적용한 Rainbow와 기존 **Rainbow**의 성능 평가
  - Rainbow는 I, III, V 레벨 별로 비교
  - Classic, Cyclic(CZ), Compressed 버전 별로 비교
  - Key scheduling, Signature, Verification 단계 별로 비교
  - 레벨 별, 버전 별, 상황 별로 비교 진행 → 총 27개의 비교 결과

MiddleGame: Evaluation

- 곱셈기를 1,000,000회 가동한 값의 평균 값을 사용, 결과는 [표 3]과 동일
- 기존 곱셈기는 355 / 16,557 clock cycles 소요
- 제안하는 곱셈기는 **58 / 99 clock cycles** 소요
- 제안하는 기법은 기존에 비해 **6.12 / 167.2배의 성능**을 지님
- 성능 격차의 요인
  1. 변수를 분리하는데 들어가는 시간 단축
  2. 곱셈을 진행하는데 요구하는 단계 축소
  3. 병렬 연산을 사용하여 다수 데이터 처리

Algorithm	GF(16) multiplier	GF(256) multiplier
Previous work	355	16,557
This work	**58**	**99**

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

Evaluation

- Rainbow signature를 비교, 결과는 [표 4]와 동일
  - 기존 Rainbow는 300회 가동한 값의 평균을 사용
  - 제안하는 기법의 Rainbow는 10,000회 가동한 값의 평균을 사용

Level	Algorithm	Previous work			This work		
		Key scheduling	Signature	Verification	**Key scheduling**	**Signature**	**Verification**
I	Classic	2.53	3.17	3.3	**1.59**	**0.32**	**0.064**
	Cyclic	281.76	3.2	12.42	**16.9**	**0.48**	**9.18**
	Compressed	281.79	127.71	12.42	**16.9**	**12.99**	**9.18**
III	Classic	3,141	27.65	28.67	**88.13**	**1.98**	**5.86**
	Cyclic	3,570	27.65	83.74	**93.73**	**1.98**	**60.96**
	Compressed	3,570	1,690	83.71	**93.7**	**75.36**	**60.96**
V	Classic	8,830	61.12	62.4	**530.14**	**2.46**	**2.69**
	Cyclic	10,140	61.12	186.66	**561.18**	**2.5**	**127.17**
	Compressed	10,140	4,850	186.88	**561.06**	**279.94**	**127.14**

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

- 기존 Rainbow에 비해 다양한 성능 차이를 보여줌
  - Rainbow I: ×1.35 (Cyclic verification, Compressed verification)
    ~ **×51.6 (Classic verification)**
  - Rainbow III: ×1.37 (Cyclic verification, Compressed verification)
    ~ **×38.10 (Compressed key scheduling)**
  - Rainbow V: ×1.47 (Cyclic verification, Compressed verification)
    ~ **×28.85 (Classic signature)**
- **최소 1.35배에서 최대 51.6배의 성능 향상 제공**

Level	Algorithm	Previous work			This work		
		Key scheduling	Signature	Verification	**Key scheduling**	**Signature**	**Verification**
I	Classic	2.53	3.17	3.3	**1.59**	**0.32**	**0.064**
	Cyclic	281.76	3.2	12.42	**16.9**	**0.48**	**9.18**
	Compressed	281.79	127.71	12.42	**16.9**	**12.99**	**9.18**
III	Classic	3,141	27.65	28.67	**88.13**	**1.98**	**5.86**
	Cyclic	3,570	27.65	83.74	**93.73**	**1.98**	**60.96**
	Compressed	3,570	1,690	83.71	**93.7**	**75.36**	**60.96**
V	Classic	8,830	61.12	62.4	**530.14**	**2.46**	**2.69**
	Cyclic	10,140	61.12	186.66	**561.18**	**2.5**	**127.17**
	Compressed	10,140	4,850	186.88	**561.06**	**279.94**	**127.14**

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

EndGame

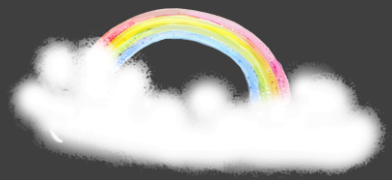
Look-up the Rainbow

Efficient Table-based Parallel Implementation of Rainbow Signature on 64-bit ARMv8 Processors

2021 국가암호공모전

EndGame:
Conclusion & Future works

- 64-bit **ARMv8** 프로세서 상에서 양자 내성 암호 **Rainbow** 최적 구현
- Tower-field 연산에 소요되는 시간을 LUT를 사용하여 단축
  - **256-byte, 272-byte**의 작은 크기의 LUT
  - 1개의 명령어로 **16-byte의 병렬 곱셈** 진행
- 기존 곱셈기보다 빠른 연산 속도, 곱셈기 적용시 효과적인 Rainbow 연산
  - 곱셈기: 기존 대비 **최대 167.2배**의 연산 속도
  - Rainbow: 기존 대비 **최대 51.6배**의 연산 속도
- **ARMv8** 프로세서 상에서의 Rainbow **III, V의 최초 구현**
  - 후속 연구자들을 위한 **발판 마련**
- 후속 과제로 Rainbow 이외의 Round 3 Finalist 알고리즘을 ARMv8 상 구현



Thank you

2021 국가암호공모전