SNOVA Multiplication on ARMv8

유튜브 주소: https://youtu.be/orR_OoyAJfw

SNOVA

- SNOVA: Simple Noncommutative unbalanced Oil and Vinegar scheme with randomness Alignment
- NIST PQC Additional Signature Round 1 제출된 알고리즘
  - 다변수 기반 전자서명(1라운드 10개 제출)
  - UOV 스킴 기반(UOV 포함 6개 제출)
    - UOV: Unbalanced Oil and Vinegar: 1999년 공개된 다변수 기반 알고리즘
- 공개키가 큰 UOV 알고리즘의 단점을 상쇄하여 설계
  - UOV 기반 알고리즘 중 가장 작은 공개키 크기를 지님
- 다양한 보안 수준에 대해 서로 다른 파라미터 옵션을 제공
  - esk 9개, ssk 9개로 총 18개의 파라미터 옵션 제공
- 장점
  - 구현이 간결하며 서명 생성 및 검증 속도가 상대적으로 빠름
- 단점
  - 상대적으로 서명의 크기가 크며 공개키의 크기가 매우 큼

SNOVA 곱셈기

- GF16 상에서의 a * b 연산 수행
  - 룩업 테이블을 활용하여 연산 구현
    - 16 * 16의 모든 경우의 수를 미리 계산한 테이블
  - 인덱스를 계산해 값을 조회하는 방식의 구현

```
#define mt(p, q) mt4b[((p) << 4) ^ (q)]
#define inv(gf16) inv4b[(gf16)]
#define gf16_get_add(a, b) ((a) ^ (b))
#define gf16_get_mul(a, b) (mt((a), (b)))

static uint8_t mt4b[256] = {0};
static uint8_t inv4b[16] = {0};

typedef uint8_t gf16_t;
```

```
/**
 * Multiplying GF16 Matrices. c = a * b
 */
static inline void gf16m_mul(gf16m_t a, gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) {
        for (int j = 0; j < rank; ++j) {
            set_gf16m(c, i, j,
                      gf16_get_mul(get_gf16m(a, i, 0), get_gf16m(b, 0, j)));
            for (int k = 1; k < rank; ++k) {
                set_gf16m(c, i, j,
                          gf16_get_add(get_gf16m(c, i, j),
                                       gf16_get_mul(get_gf16m(a, i, k),
                                                    get_gf16m(b, k, j))));
            }
        }
    }
}
```

```
void init_gf16_tables() {
    uint8_t F_star[15] = {1, 2,  4, 8,  3,  6,  12, 11,
                          5, 10, 7, 14, 15, 13, 9};  // Z2[x]/(x^4+x+1)
    for (int i = 0; i < 16; ++i) {
        mt(0, i) = mt(i, 0) = 0;
    }

    for (int i = 0; i < 15; ++i)
        for (int j = 0; j < 15; ++j)
            mt(F_star[i], F_star[j]) = F_star[(i + j) % 15];
    {
        int g = F_star[1], g_inv = F_star[14], gn = 1, gn_inv = 1;
        inv4b[0] = 0;
        inv4b[1] = 1;
        for (int index = 0; index < 14; index++)
            inv4b[(gn = mt(gn, g))] = (gn_inv = mt(gn_inv, g_inv));
    }
}
```

→ 기약다항식(x^4+x+1)에 의해 생성된 GF16 원소

→ 룩업 테이블 초기화 과정

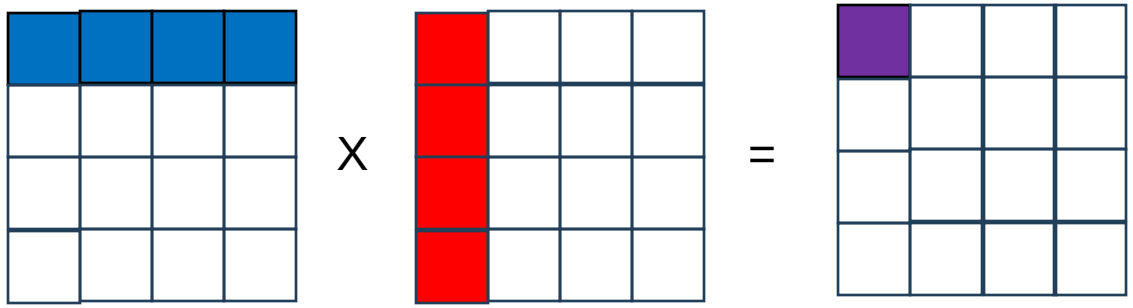
→ 곱셈 테이블 생성 과정
각 원소의 곱셈 결과는 F_star 배열에서 두 인덱스의 합을 15로 모듈러 연산하여 얻은 값

→ 역원 생성 과정

SNOVA 곱셈기

```
#define mt(p, q) mt4b[((p) << 4) ^ (q)]
#define inv(gf16) inv4b[(gf16)]
#define gf16_get_add(a, b) ((a) ^ (b))
#define gf16_get_mul(a, b) (mt((a), (b)))

static uint8_t mt4b[256] = {0};
static uint8_t inv4b[16] = {0};

typedef uint8_t gf16_t;
```

- i 루프: a행렬의 요소(행)를 반복
- j 루프: b행렬의 요소(열)를 반복



행렬 곱셈

```
/**
 * Multiplying GF16 Matrices. c = a * b
 */
static inline void gf16m_mul(gf16m_t a, gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) {
        for (int j = 0; j < rank; ++j) {
            set_gf16m(c, i, j,
                      gf16_get_mul(get_gf16m(a, i, 0), get_gf16m(b, 0, j)));
            for (int k = 1; k < rank; ++k) {
                set_gf16m(c, i, j,
                          gf16_get_add(get_gf16m(c, i, j),
                                       gf16_get_mul(get_gf16m(a, i, k),
                                                    get_gf16m(b, k, j))));
            }
        }
    }
}
```

각 행렬의 첫 번째 요소간의 곱셈
e.g. a[0][0] * b[0][0]

각 행렬의 2,3,4 번째 요소간의 곱셈
각 곱셈 결과를 누산하여 결과 도출

SNOVA 곱셈기

- GF16 상에서의 곱셈 진행
- 기약다항식 사용
  - $x^4 + x + 1 = 0$

```c
/**
 * Multiplying GF16 Matrices. c = a * b
 */
static inline void gf16m_mul(gf16m_t a, gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) {
        for (int j = 0; j < rank; ++j) {
            set_gf16m(c, i, j,
                      gf16_get_mul(get_gf16m(a, i, 0), get_gf16m(b, 0, j)));
            for (int k = 1; k < rank; ++k) {
                set_gf16m(c, i, j,
                          gf16_get_add(get_gf16m(c, i, j),
                                       gf16_get_mul(get_gf16m(a, i, k),
                                                    get_gf16m(b, k, j))));
            }
        }
    }
}
```

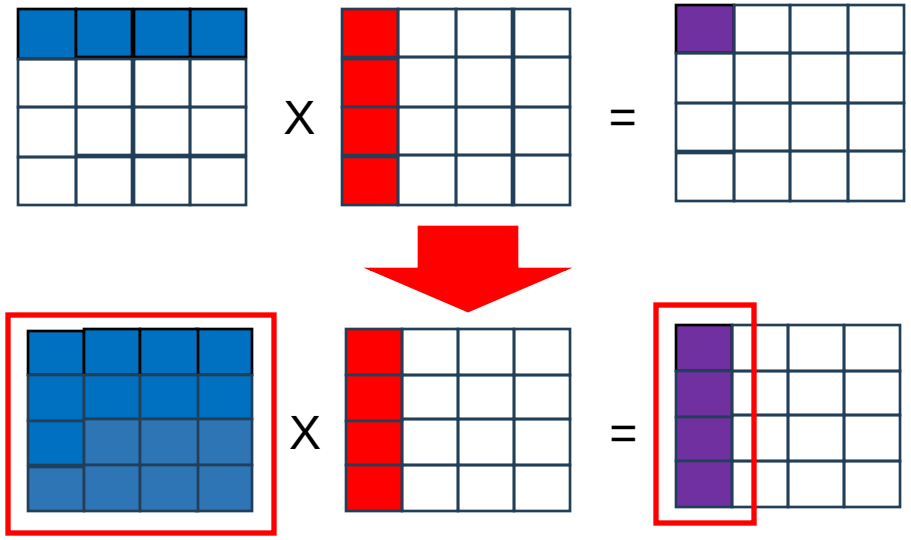
```
1st mul (0,0): 8 * 8 = C
2 mul (0,0): C + (7 * 7 = 6) = A
3 mul (0,0): A + (6 * 6 = 7) = D
4 mul (0,0): D + (5 * 5 = 2) = F
1st mul (0,1): 8 * 7 = D
2 mul (0,1): D + (7 * 6 = 1) = C
3 mul (0,1): C + (6 * 5 = D) = 1
4 mul (0,1): 1 + (5 * 4 = 7) = 6
1st mul (0,2): 8 * 6 = 5
2 mul (0,2): 5 + (7 * 5 = 8) = D
3 mul (0,2): D + (6 * 4 = B) = 6
4 mul (0,2): 6 + (5 * 3 = F) = 9
1st mul (0,3): 8 * 5 = E
2 mul (0,3): E + (7 * 4 = F) = 1
3 mul (0,3): 1 + (6 * 3 = A) = B
4 mul (0,3): B + (5 * 2 = A) = 1
1st mul (1,0): 7 * 8 = D
2 mul (1,0): D + (6 * 7 = 1) = C
3 mul (1,0): C + (5 * 6 = D) = 1
4 mul (1,0): 1 + (4 * 5 = 7) = 6
1st mul (1,1): 7 * 7 = 6
2 mul (1,1): 6 + (6 * 6 = 7) = 1
3 mul (1,1): 1 + (5 * 5 = 2) = 3
4 mul (1,1): 3 + (4 * 4 = 3) = 0
1st mul (1,2): 7 * 6 = 1
2 mul (1,2): 1 + (6 * 5 = D) = C
3 mul (1,2): C + (5 * 4 = 7) = B
4 mul (1,2): B + (4 * 3 = C) = 7
1st mul (1,3): 7 * 5 = 8
2 mul (1,3): 8 + (6 * 4 = B) = 3
3 mul (1,3): 3 + (5 * 3 = F) = C
4 mul (1,3): C + (4 * 2 = 8) = 4
1st mul (2,0): 6 * 8 = 5
2 mul (2,0): 5 + (5 * 7 = 8) = D
3 mul (2,0): D + (4 * 6 = B) = 6
4 mul (2,0): 6 + (3 * 5 = F) = 9
1st mul (2,1): 6 * 7 = 1
2 mul (2,1): 1 + (5 * 6 = D) = C
3 mul (2,1): C + (4 * 5 = 7) = B
```

- 연산 원리 예시
  - 8 * 8 = C
    - 8 -> 1000 -> x^3
    - 8 * 8 -> $x^3 * x^3 = x^6$
    - x^6 을 기약다항식으로 나눈 나머지 계산
      - $x^4 + x + 1 = 0$ -> $x^4 = -x - 1$ -> $x^4 = x + 1$으로 치환 가능
    - $x^6 = (x^2 * x^4) = (x^2 * (x + 1)) = x^3 + x^2$ -> 1100 -> C
  - 7 * 6 = 1
    - 7 -> 0111 -> $x^2 + x + 1$, 6 -> 0110 -> $x^2 + x$
    - 7 * 6 -> $(x^2 + x + 1) * (x^2 + x) = x^4 + 2x^3 + 2x^2 + x = x^4 + x$ (2진수 연산이므로 $2x^3$, $2x^2$은 0으로 취급 가능)
    - $x^4 + x$를 기약다항식으로 나눈 나머지 계산
    - $x^4 + x = (x + 1) + x = 1$

SNOVA 곱셈기 최적 구현 코드

- NEON 인트린직 함수를 사용한 병렬 구현

- 1개의 행에 대해 4개의 열을 병렬 연산
  - 128bit 벡터 레지스터에 32bit 열 4개를 넣어 병렬화
  - 곱셈 연산은 인덱스만 계산 후 look up table 이용
    - gf16_get_mul 함수 사용하여 미리 계산되어있는 결과를 가져옴



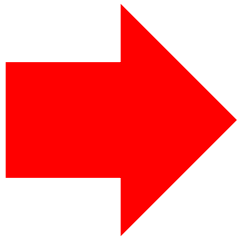
```c
// reference
static inline void gf16m_mul(gf16m_t a, gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) {
        for (int j = 0; j < rank; ++j) {
            set_gf16m(c, i, j,
                      gf16_get_mul(get_gf16m(a, i, 0), get_gf16m(b, 0, j)));
            for (int k = 1; k < rank; ++k) {
                set_gf16m(c, i, j,
                          gf16_get_add(get_gf16m(c, i, j),
                                       gf16_get_mul(get_gf16m(a, i, k),
                                                    get_gf16m(b, k, j))));
            }
        }
    }
}
```

```c
static inline void gf16m_neon_mul(const gf16m_t a, const gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) { // 행
        // 초기화: 각 열에 대한 곱셈 결과를 저장할 벡터 초기화
        uint8x16_t sum0 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[0]));
        uint8x16_t sum1 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[1]));
        uint8x16_t sum2 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[2]));
        uint8x16_t sum3 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[3]));

        for (int k = 1; k < 4; ++k) {
            // 각 요소에 대해 8비트 값을 128비트로 확장 및 병렬 연산
            uint8x16_t prod0 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4]));
            uint8x16_t prod1 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 1]));
            uint8x16_t prod2 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 2]));
            uint8x16_t prod3 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 3]));

            // 덧셈 (XOR 연산) 수행
            sum0 = veorq_u8(sum0, prod0); // veorq_u8는 128비트 벡터 XOR
            sum1 = veorq_u8(sum1, prod1);
            sum2 = veorq_u8(sum2, prod2);
            sum3 = veorq_u8(sum3, prod3);
        }

        // 결과를 스칼라 값으로 추출하여 저장
        c[i * 4 + 0] = vgetq_lane_u8(sum0, 0); // 첫 번째 열에 대한 결과
        c[i * 4 + 1] = vgetq_lane_u8(sum1, 0); // 두 번째 열에 대한 결과
        c[i * 4 + 2] = vgetq_lane_u8(sum2, 0); // 세 번째 열에 대한 결과
        c[i * 4 + 3] = vgetq_lane_u8(sum3, 0); // 네 번째 열에 대한 결과
    }
}
```

SNOVA 곱셈기 최적 구현

- ARMv8 NEON 인트린직 함수 활용
  - **vdupq_n_u8():** 스칼라 값을 벡터화 하여 128bit 레지스터에 복사
    - 8비트 값을 16개의 슬롯에 복사(128bit)
  - **veorq_u8():** 128bit 벡터에 대해 XOR 연산 수행
    - 128bit 벡터 레지스터 간 XOR 연산
  - **vgetq_lane_u8():** 벡터에서 스칼라 값을 추출
    - 결과를 스칼라 값으로 추출하여 저장

```c
/**
 * Multiplying GF16 Matrices. c = a * b
 */
static inline void gf16m_mul(gf16m_t a, gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) {
        for (int j = 0; j < rank; ++j) {
            set_gf16m(c, i, j,
                      gf16_get_mul(get_gf16m(a, i, 0), get_gf16m(b, 0, j)));
            for (int k = 1; k < rank; ++k) {
                set_gf16m(c, i, j,
                          gf16_get_add(get_gf16m(c, i, j),
                                       gf16_get_mul(get_gf16m(a, i, k),
                                                    get_gf16m(b, k, j))));
            }
        }
    }
}
```



```c
static inline void gf16m_neon_mul(const gf16m_t a, const gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) { // 행
        // 초기화: 각 열에 대한 곱셈 결과를 저장할 벡터 초기화
        uint8x16_t sum0 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[0]));
        uint8x16_t sum1 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[1]));
        uint8x16_t sum2 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[2]));
        uint8x16_t sum3 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[3]));

        for (int k = 1; k < 4; ++k) {
            // 각 요소에 대해 8비트 값을 128비트로 확장 및 병렬 연산
            uint8x16_t prod0 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4]));
            uint8x16_t prod1 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 1]));
            uint8x16_t prod2 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 2]));
            uint8x16_t prod3 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 3]));

            // 덧셈 (XOR 연산) 수행
            sum0 = veorq_u8(sum0, prod0); // veorq_u8는 128비트 벡터 XOR
            sum1 = veorq_u8(sum1, prod1);
            sum2 = veorq_u8(sum2, prod2);
            sum3 = veorq_u8(sum3, prod3);
        }

        // 결과를 스칼라 값으로 추출하여 저장
        c[i * 4 + 0] = vgetq_lane_u8(sum0, 0); // 첫 번째 열에 대한 결과
        c[i * 4 + 1] = vgetq_lane_u8(sum1, 0); // 두 번째 열에 대한 결과
        c[i * 4 + 2] = vgetq_lane_u8(sum2, 0); // 세 번째 열에 대한 결과
        c[i * 4 + 3] = vgetq_lane_u8(sum3, 0); // 네 번째 열에 대한 결과
    }
}
```

SNOVA 곱셈기 최적 구현 코드

```
static inline void gf16m_neon_mul(const gf16m_t a, const gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) { // 행
        // 초기화: 각 열에 대한 곱셈 결과를 저장할 벡터 초기화
        uint8x16_t sum0 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[0]));
        uint8x16_t sum1 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[1]));
        uint8x16_t sum2 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[2]));
        uint8x16_t sum3 = vdupq_n_u8(gf16_get_mul(a[i * 4], b[3]));

        for (int k = 1; k < 4; ++k) {
            // 각 요소에 대해 8비트 값을 128비트로 확장 및 병렬 연산
            uint8x16_t prod0 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4]));
            uint8x16_t prod1 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 1]));
            uint8x16_t prod2 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 2]));
            uint8x16_t prod3 = vdupq_n_u8(gf16_get_mul(a[i * 4 + k], b[k * 4 + 3]));

            // 덧셈 (XOR 연산) 수행
            sum0 = veorq_u8(sum0, prod0); // veorq_u8는 128비트 벡터 XOR
            sum1 = veorq_u8(sum1, prod1);
            sum2 = veorq_u8(sum2, prod2);
            sum3 = veorq_u8(sum3, prod3);
        }

        // 결과를 스칼라 값으로 추출하여 저장
        c[i * 4 + 0] = vgetq_lane_u8(sum0, 0); // 첫 번째 열에 대한 결과
        c[i * 4 + 1] = vgetq_lane_u8(sum1, 0); // 두 번째 열에 대한 결과
        c[i * 4 + 2] = vgetq_lane_u8(sum2, 0); // 세 번째 열에 대한 결과
        c[i * 4 + 3] = vgetq_lane_u8(sum3, 0); // 네 번째 열에 대한 결과
    }
}
```

16bit 4*4 행렬인 a, b를 곱해 c에 저장

rank는 행렬 크기(4), i는 행렬의 각 행을 의미

각 열(0~3)에 대한 곱셈 결과 저장할 벡터 초기화
이 구현은 a의 행과 b의 열에 대한 곱셈 수행
sum0,1,2,3은 b의 1~4 열에 대한 각 곱셈 결과 저장

각 행, 열에 대한 곱셈을 수행 후 누산하는 루프

8비트 값을 128비트 레지스터에 복사(16번)
확장된 레지스터를 이용해 8비트 단위 병렬 연산 가능

곱셈 결과를 누산하여 최종 결과를 계산
덧셈은 XOR로 수행

결과를 스칼라 값으로 추출하여 저장하는 단계

성능 측정

- 성능 측정 환경
  - H/W: Apple M2(8GB RAM)
  - IDE: Visual Studio Code 1.90.1
  - GCC: 14.0.3
  - 최적화 레벨: O3

함수	레퍼런스	덧셈 최적화	곱셈 최적화	덧셈 + 곱셈 최적화
키 생성	**4,049,105**	3,294,436	3,392,110	**2,077,540**
서명 생성	**16,379,266**	15,476,262	11,802,266	**10,780,592**
서명 검증	**12,148,060**	10,700,197	7,646,975	**6,324,007**

- 키 생성: 레퍼런스 대비 **48.7%** 성능 향상
- 서명 생성: 레퍼런스 대비 **34.2%** 성능 향상
- 서명 검증: 레퍼런스 대비 **47.9%**의 성능 향상

Q & A