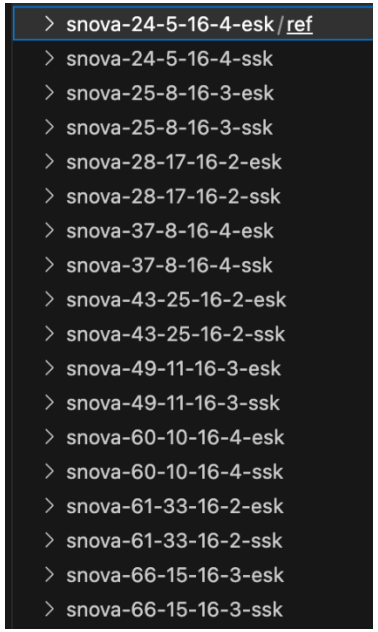
ARMv8 상에서의 SNOVA 전자서명 최적 구현

SNOVA

- SNOVA: Simple Noncommutative unbalanced Oil and Vinegar scheme with randomness Alignment
- NIST PQC Additional Signature 공모전에 제출된 알고리즘
  - 다변수 기반, UOV 스킴 기반
  - 현재 공모전 2라운드 진출
- 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
  - 행렬 크기(rank = 2, 3, 4)에 따라 구현이 달라짐
- 주요 연산: 가산기, 곱셈기



SNOVA 가산기

- 128bit matrix a XOR b -> c
  - 8bit 4*4 행렬 간의 덧셈
    - 유한체 상에서의 덧셈은 비트간 XOR 연산으로 구현 가능
    - 캐리연산 등 불필요
  - 각 위치(i, j)에 해당하는 행렬 a, b의 요소를 가져와 덧셈
    - 반복문의 i, j는 행렬의 인덱스를 설정
      - 인덱스는 get_gf16m 함수 내에서의 shift, xor 연산으로 설정
    - 덧셈은 gf16_get_add 함수를 통해 실행

```c
/**
 * Adding GF16 Matrices. c = a + b
 */
static inline void gf16m_add(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_add(get_gf16m(a, i, j), get_gf16m(b, i, j)));
        }
    }
}
```

```c
#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;
```

$$A + B = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ a_{m1} & a_{m2} & \cdots & a_{mf} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1j} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2j} & \cdots & b_{2n} \\ b_{i1} & b_{i2} & \cdots & b_{ij} & \cdots & b_{in} \\ b_{m1} & b_{m2} & \cdots & b_{mf} & \cdots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{1j}+b_{1j} & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{2j}+b_{2j} & a_{2n}+b_{2n} \\ a_{i1}+b_{i1} & a_{i2}+b_{i2} & a_{ij}+b_{ij} & a_{in}+b_{in} \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & a_{mj}+b_{mj} & a_{mn}+b_{mn} \end{bmatrix} = \begin{bmatrix} a_{ij} + b_{ij} \end{bmatrix}$$

```c
#elif rank == 4
#define get_gf16m(gf16m, x, y) (gf16m[(((x) << 2) ^ (y))])
#define set_gf16m(gf16m, x, y, value) (gf16m[(((x) << 2) ^ (y))] = value)
#endif

typedef gf16_t gf16m_t[sq_rank];
```

sq_rank: rank*rank(16)

SNOVA 가산기 binary field

- a[0~15] XOR b[0~15] => c에 Set
  - 즉, 단순한 a XOR b의 구조

```
#elif rank == 4
#define get_gf16m(gf16m, x, y) (gf16m[(((x) << 2) ^ (y))])
#define set_gf16m(gf16m, x, y, value) (gf16m[(((x) << 2) ^ (y))] = value)
#endif

typedef gf16_t gf16m_t[sq_rank];
```

(i, j)	0 0	0 1	0 2	0 3	1 0	1 1	1 2	1 3	2 0	2 1	2 2	2 3	3 0	3 1	3 2	3 3
binary	0000 0000	0000 0001	0000 0010	0000 0011	0001 0000	0001 0001	0001 0010	0001 0011	0010 0000	0010 0001	0010 0010	0010 0011	0011 0000	0011 0001	0011 0010	0011 0011
i left shift 2	0000 0000	0000 0001	0000 0010	0000 0011	0100 0000	0100 0001	0100 0010	0100 0011	1000 0000	1000 0001	1000 0010	1000 0011	1100 0000	1100 0001	1100 0010	1100 0011
I XOR j	0b0000	0b0001	0b0010	0b0011	0b0100	0b0101	0b0110	0b0111	0b1000	0b1001	0b1010	0b1011	0b1100	0b1101	0b1110	0b1111
Result	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

```
int add_count = 0;
static inline void gf16m_add(gf16m_t a, gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) {
        for (int j = 0; j < rank; ++j) {
            if (add_count < 16) {
                uint8_t value_a = get_gf16m(a, i, j);
                uint8_t value_b = get_gf16m(b, i, j);
                uint8_t result = gf16_get_add(value_a, value_b);
                printf("(%d,%d) A:%X B:%X C:%X\n",
                        i,j,value_a, value_b, result);
                set_gf16m(c, i, j, result);
                add_count++;
            } else {
                set_gf16m(c, i, j, gf16_get_add(get_gf16m(a, i, j), get_gf16m(b, i, j)));
```

```
(0,0) A:F B:A C:5
(0,1) A:0 B:2 C:2
(0,2) A:0 B:E C:E
(0,3) A:0 B:9 C:9
(1,0) A:0 B:2 C:2
(1,1) A:F B:E C:1
(1,2) A:0 B:9 C:9
(1,3) A:0 B:5 C:5
(2,0) A:0 B:E C:E
(2,1) A:0 B:9 C:9
(2,2) A:F B:5 C:A
(2,3) A:0 B:7 C:7
(3,0) A:0 B:9 C:9
(3,1) A:0 B:5 C:5
(3,2) A:0 B:7 C:7
(3,3) A:F B:B C:4
```

```
LD1.16b {v0}, [x0]
LD1.16b {v1}, [x1]

EOR.16b v1, v0, v1

ST1.16b {v1}, [x2]

RET
```

rank=4
ARMv8 최적 구현

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

```c
#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;
```

```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
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

→ 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

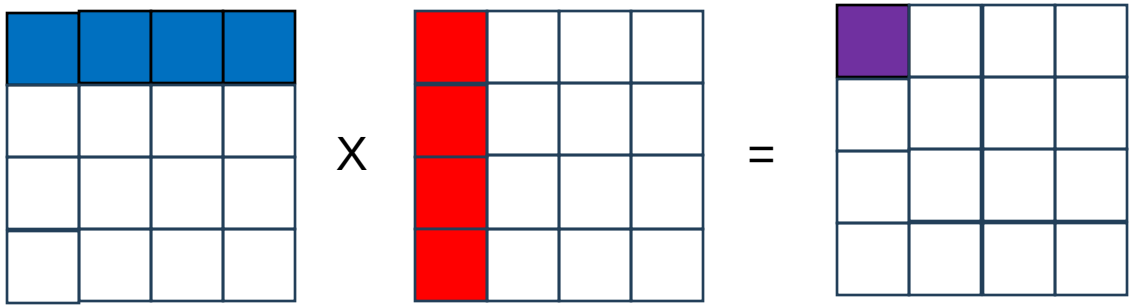
SNOVA 곱셈기

```
#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 binary field

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

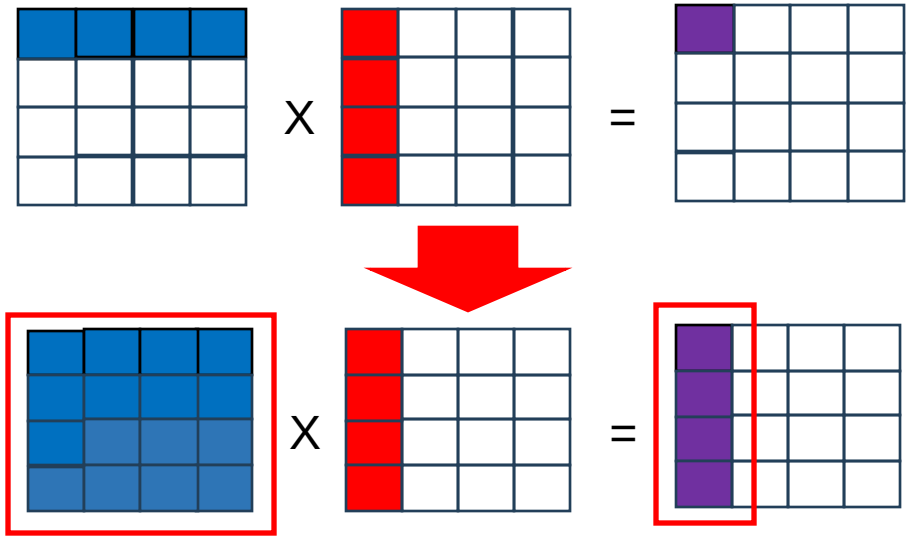
```c
/**
 * Multiplying GF16 Matrices. c = a * b
 */
static inline void gf16m_mul(gf16m_t a, gf16m_t b, gf16m_t c) {
    for (int i = 0; i < rank; ++i) {
        for (int j = 0; j < rank; ++j) {
            set_gf16m(c, i, j,
                      gf16_get_mul(get_gf16m(a, i, 0), get_gf16m(b, 0, j)));
            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
  - **vdupq_n_u8():** 8비트 값을 16개의 슬롯에 복사(128bit)
  - **veorq_u8():** 두 벡터의 각 원소를 XOR 연산
  - **vgetq_lane_u8():** 벡터에서 스칼라 값을 추출
- 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



```
// 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))));
            }
        }
    }
}
```

```
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
    }
}
```

AES 가속기

- ARMv8 어셈블리를 활용한 aes128-ctr 모드 가속기 적용
- Armv8에서 지원하는 aese, aesmc 명령어 사용
- 한 번에 4개의 블록 병렬 연산
- aese: AES 각 라운드 암호화 수행
  - SubBytes 수행
  - ShiftRows 수행
  - AddRoundKey 수행
- aesmc: MixColums 연산 수행

```
aes128_ctr_multi_asm:
_aes128_ctr_multi_asm:

    stp x29, x30, [sp, #-16]!
    mov x29, sp

    // 매개변수
    // x0: output 포인터
    // x1: length (바이트 수)
    // x2: nonce 포인터 (16 바이트)
    // x3: roundkey 포인터 (176 바이트)

    // 넌스 및 초기 카운터 로드
    ldr q16, [x2]                   // nonce + counter 초기값

    // 카운터 증가 초기화 (v17 = {0, 0, 0, 0})
    movi v17.4s, #0

    // 카운터 증가 설정 (첫 번째 요소에 1 삽입)
    mov x6, #1
    ins v17.d[0], x6                // v17.s[0] = 1

    // AES 라운드 키 로드
    ldp q6, q7, [x3]                // 라운드 키 0, 1
    ldp q8, q9, [x3, #32]           // 라운드 키 2, 3
    ldp q10, q11, [x3, #64]         // 라운드 키 4, 5
    ldp q12, q13, [x3, #96]         // 라운드 키 6, 7
    ldp q14, q15, [x3, #128]        // 라운드 키 8, 9
    ldr q18, [x3, #160]             // 라운드 키 10

    // 블록 수 계산 (16 바이트 블록 기준)
    mov x4, #0                      // 블록 인덱스 초기화
    lsr x5, x1, #4                  // 바이트 수 -> 블록 수 계산
```

```
encrypt_loop:
    cmp x4, x5                      // 현재 블록 수 확인
    bge encrypt_done                // 완료 시 종료

    // 병렬 카운터 설정
    mov v0.16b, v16.16b             // v0 = 카운터 + 넌스
    mov v1.16b, v16.16b
    mov v2.16b, v16.16b
    mov v3.16b, v16.16b

    // 카운터 증가
    add v1.4s, v1.4s, v17.4s
    add v2.4s, v1.4s, v17.4s
    add v3.4s, v2.4s, v17.4s

    // 병렬 AES 암호화
    aese v0.16b, v6.16b
    aesmc v0.16b, v0.16b
    aese v1.16b, v6.16b
    aesmc v1.16b, v1.16b
    aese v2.16b, v6.16b
    aesmc v2.16b, v2.16b
    aese v3.16b, v6.16b
    aesmc v3.16b, v3.16b

    // 계속 AES 라운드 수행
    aese v0.16b, v7.16b
    aesmc v0.16b, v0.16b
    aese v1.16b, v7.16b
    aesmc v1.16b, v1.16b
    aese v2.16b, v7.16b
    aesmc v2.16b, v2.16b
    aese v3.16b, v7.16b
    aesmc v3.16b, v3.16b
```

성능 측정

- **키 생성 성능 향상 (crypto_sign_keypair)**
  - Rank 4: 51.7% ~ 68.7%
  - Rank 3: 25% ~ 35%
  - Rank 2: 20% ~ 30%
- **서명 생성 성능 향상 (crypto_sign)**
  - Rank 4: 29.5% ~ 36.7%
  - Rank 3: 10% ~ 25%
  - Rank 2: 5% 미만
- **서명 검증 성능 향상 (crypto_sign_verify)**
  - Rank 4: 42.8% ~ 50.0%
  - Rank 3: 15% ~ 25%
  - Rank 2: 10% 미만

파라미터	Work	레퍼런스	최적 구현
snova-24-5-16-4-esk	키 생성	3,475,085	1,398,483
	서명 생성	15,753,137	10,707,076
	서명 검증	10,629,480	5,898,839
snova-25-8-16-3-esk	키 생성	3,463,081	2,050,620
	서명 생성	6,484,644	6,007,548
	서명 검증	3,158,913	2,843,025
snova-28-17-16-2-esk	키 생성	5,557,774	4,370,283
	서명 생성	2,144,271	2,127,047
	서명 검증	2,282,003	1,330,599
snova-37-8-16-4-esk	키 생성	18,112,642	8,641,329
	서명 생성	59,263,911	41,794,416
	서명 검증	40,590,714	22,410,705
snova-43-25-16-2-esk	키 생성	24,795,480	21,714,931
	서명 생성	7,117,074	7,062,414
	서명 검증	7,609,277	4,451,052
snova-49-11-16-3-esk	키 생성	17,887,340	13,837,133
	서명 생성	23,711,616	23,494,859
	서명 검증	14,493,366	11,888,327
snova-60-10-16-4-esk	키 생성	69,097,561	33,384,333
	서명 생성	168,794,828	115,369,891
	서명 검증	122,557,890	67,624,592
snova-61-33-16-2-esk	키 생성	83,359,185	73,577,806
	서명 생성	17,722,885	17,651,531
	서명 검증	19,220,693	11,221,976
snova-66-15-16-3-esk	키 생성	57,863,043	46,400,037
	서명 생성	60,121,205	60,103,122
	서명 검증	35,164,753	31,463,332

파라미터	Work	레퍼런스	최적 구현
snova-24-5-16-4-ssk	키 생성	3,162,043	1,398,307
	서명 생성	15,532,906	10,813,358
	서명 검증	10,665,857	5,911,721
snova-25-8-16-3-ssk	키 생성	2,964,322	2,047,839
	서명 생성	5,934,865	5,904,548
	서명 검증	3,777,688	2,895,751
snova-28-17-16-2-ssk	키 생성	5,839,063	4,363,678
	서명 생성	2,134,444	2,126,962
	서명 검증	2,270,131	1,335,159
snova-37-8-16-4-ssk	키 생성	20,797,176	8,641,214
	서명 생성	59,903,432	41,767,647
	서명 검증	40,680,194	22,403,252
snova-43-25-16-2-ssk	키 생성	26,863,060	21,603,320
	서명 생성	7,011,836	7,001,149
	서명 검증	7,617,343	4,451,764
snova-49-11-16-3-ssk	키 생성	19,166,980	13,781,456
	서명 생성	23,715,980	23,580,833
	서명 검증	14,569,760	12,809,535
snova-60-10-16-4-ssk	키 생성	69,064,963	33,406,701
	서명 생성	168,849,382	115,361,855
	서명 검증	122,586,802	67,590,588
snova-61-33-16-2-ssk	키 생성	82,685,800	73,589,368
	서명 생성	17,421,722	17,250,398
	서명 검증	19,222,942	11,213,860
snova-66-15-16-3-ssk	키 생성	57,911,107	46,343,915
	서명 생성	60,674,453	59,117,498
	서명 검증	35,254,524	31,426,092

Q & A