AIMer 최적 구현 코드 분석(ARMv8)

유튜브 주소 : https://youtu.be/cz3hUPiwYgY

AIMer & AIM

최적 구현 코드 분석

성능 측정 결과

AIMer

- **AIMer**
  - **대칭키 기반**의 전자 서명 알고리즘
  - NIST PQC 전자 서명 공모전 Round 1 후보 알고리즘
    & KPQC 공모전 Round 1 선정 알고리즘들 중 하나

Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process

Updated October 2022 to reflect that IP statements can be accepted digitally.

Symmetric-based Signatures

Algorithm	Algorithm Information
AIMer	Specification IP Statements Zip file (34 MB) Website

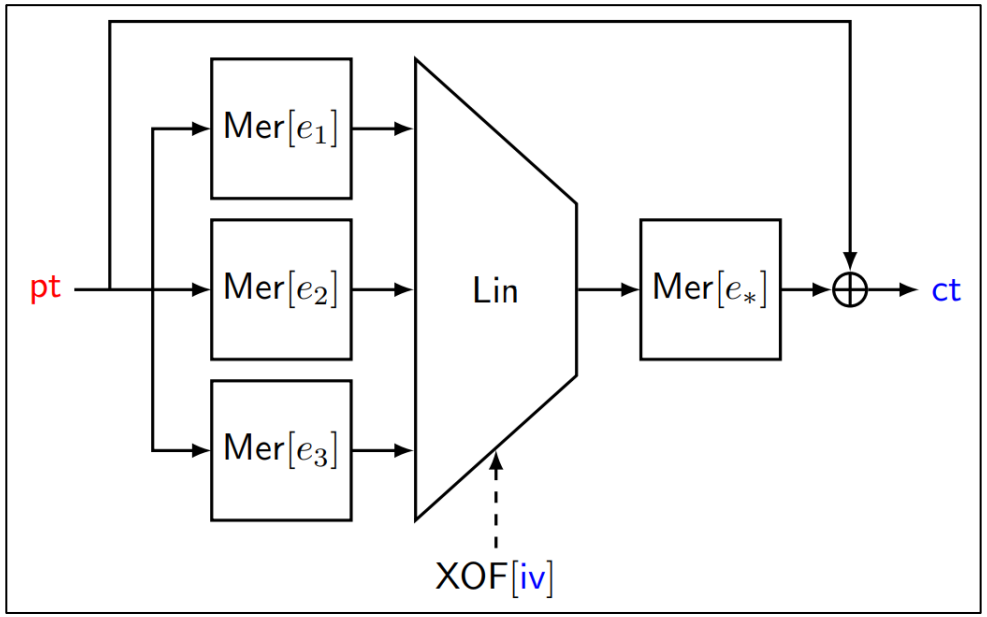
「양자내성암호 국가공모전」라운드 결과로 선정된 알고리즘 8종을 아래와 같이 발표합니다.

A. 전자서명
- AIMer
- HAETAE
- MQ-Sign
- NCC-Sign

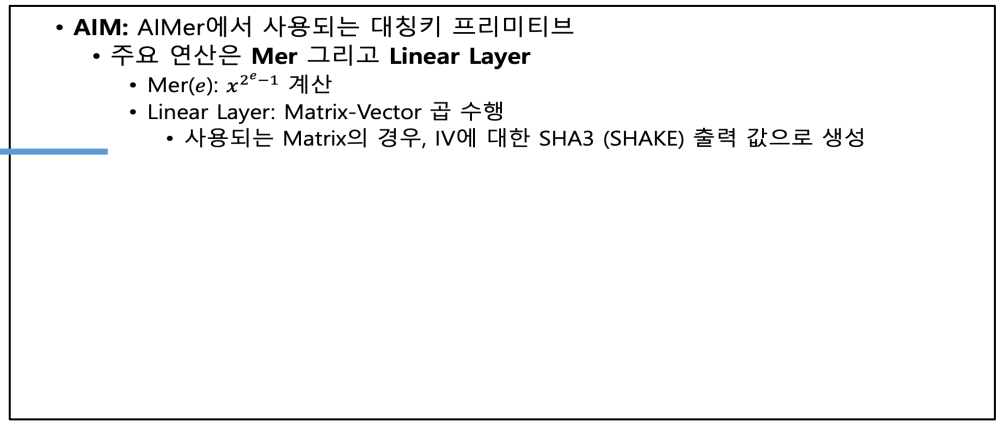
B. 공개키암호/키설정
- NTRU+
- PALOMA
- REDOG
- SMAUG + TiGER (merged)

AIM

- **AIM:** AIMer에서 사용되는 대칭키 프리미티브
  - 주요 연산은 **Mer** 그리고 **Linear Layer**
    - Mer(e): x^{2^e-1} 계산
    - Linear Layer: Matrix-Vector 곱 수행
      - 사용되는 Matrix의 경우, IV에 대한 SHA3 (SHAKE) 출력 값으로 생성

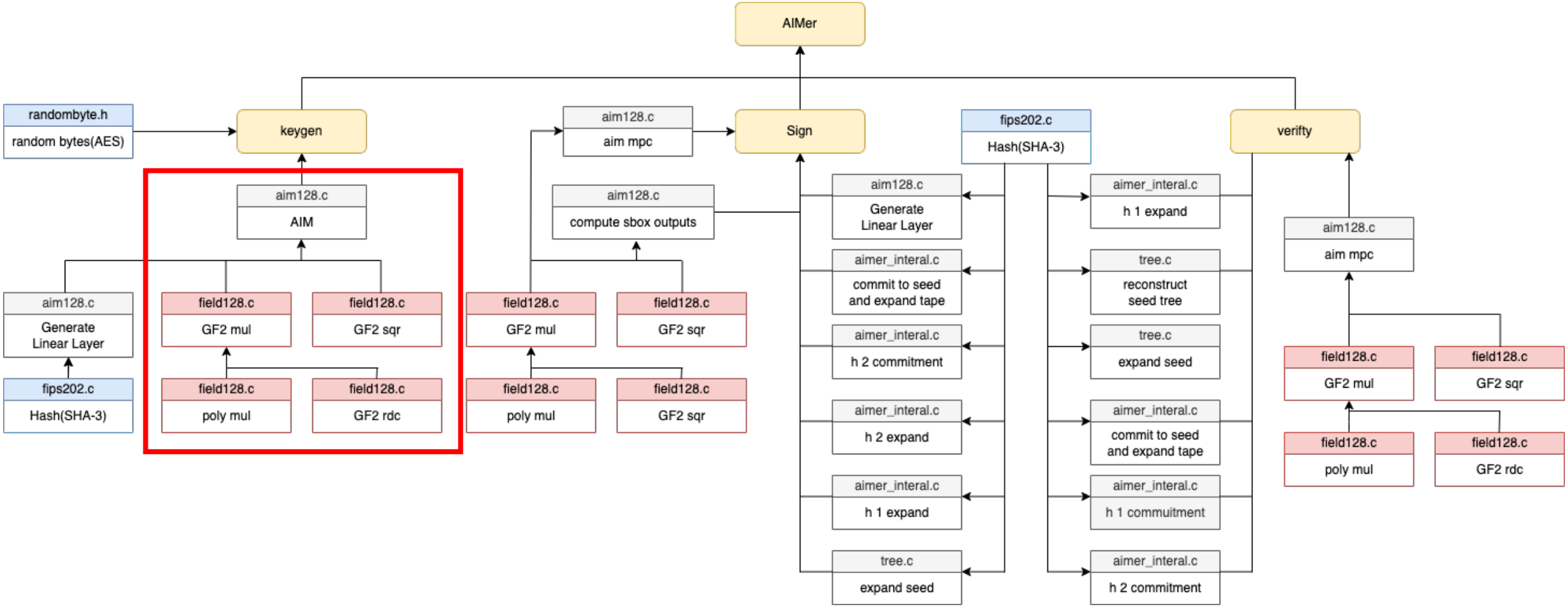


< AIM-V 암호화 >



< AIM 파라미터>

AIMer 함수 관계도


AIMer
randombyte.h
random bytes(AES)
keygen
aim128.c
AIM
aim128.c
Generate Linear Layer
fips202.c
Hash(SHA-3)
field128.c
GF2 mul
field128.c
GF2 sqr
field128.c
poly mul
field128.c
GF2 rdc
aim128.c
aim mpc
aim128.c
compute sbox outputs
field128.c
GF2 mul
field128.c
GF2 sqr
field128.c
poly mul
field128.c
GF2 sqr
Sign
aim128.c
Generate Linear Layer
aimer_interal.c
commit to seed and expand tape
aimer_interal.c
h 2 commitment
aimer_interal.c
h 2 expand
aimer_interal.c
h 1 expand
tree.c
expand seed
fips202.c
Hash(SHA-3)
aimer_interal.c
h 1 expand
tree.c
reconstruct seed tree
tree.c
expand seed
aimer_interal.c
commit to seed and expand tape
aimer_interal.c
h 1 commuitment
aimer_interal.c
h 2 commitment
verifty
aim128.c
aim mpc
field128.c
GF2 mul
field128.c
GF2 sqr
field128.c
poly mul
field128.c
GF2 rdc

AIM 핵심 연산자

- **Mersenne 연산** 내부 연산
  - **MUL 연산(곱셈)**, **SQR 연산(제곱)** 사용
  - **MUL 연산** 내부에서 **RDC 연산** 사용

```c
void GF2_128_mul(const GF2_128 a, const GF2_128 b, GF2_128 c)
{
  uint64_t temp[4] = {0,};

  poly128_mul(a, b, temp);
  GF2_128_rdc(temp, c);
}
```

```c
void mersenne_exp_3(const GF in, GF out)
{
  GF t1 = {0,};

  // t1 = a ^ (2^2 - 1)
  GF_sqr(in, t1);
  GF_mul(t1, in, t1);

  // out = a ^ (2^3 - 1)
  GF_sqr(t1, t1);
  GF_mul(t1, in, out);
}
```

```c
void poly128_sqr(const GF2_128 a, uint64_t *c)
{
  int i;
  for (i = 0; i < NUMWORDS_FIELD; i++)
  {
    c[2 * i    ] = sqr_table[(a[i] >> 28) & 0xf] << 56 |
                   sqr_table[(a[i] >> 24) & 0xf] << 48 |
                   sqr_table[(a[i] >> 20) & 0xf] << 40 |
                   sqr_table[(a[i] >> 16) & 0xf] << 32 |
                   sqr_table[(a[i] >> 12) & 0xf] << 24 |
                   sqr_table[(a[i] >>  8) & 0xf] << 16 |
                   sqr_table[(a[i] >>  4) & 0xf] <<  8 |
                   sqr_table[(a[i]      ) & 0xf];

    c[2 * i + 1] = sqr_table[(a[i] >> 60)      ] << 56 |
                   sqr_table[(a[i] >> 56) & 0xf] << 48 |
                   sqr_table[(a[i] >> 52) & 0xf] << 40 |
                   sqr_table[(a[i] >> 48) & 0xf] << 32 |
                   sqr_table[(a[i] >> 44) & 0xf] << 24 |
                   sqr_table[(a[i] >> 40) & 0xf] << 16 |
                   sqr_table[(a[i] >> 36) & 0xf] <<  8 |
                   sqr_table[(a[i] >> 32) & 0xf];
  }
}
```

```c
void poly64_mul(const uint64_t a, const uint64_t b, uint64_t *c1, uint64_t *c0
{
  uint64_t table[16];
  uint64_t temp, mask, high, low;
  uint64_t top3 = a >> 61;

  table[0] = 0;
  table[1] = a & 0x1fffffffffffffffULL;
  table[2] = table[1] << 1;
  table[4] = table[2] << 1;
  table[8] = table[4] << 1;

  table[3] = table[1] ^ table[2];

  table[5] = table[1] ^ table[4];
  table[6] = table[2] ^ table[4];
  table[7] = table[1] ^ table[6];

  table[9] = table[1] ^ table[8];
  table[10] = table[2] ^ table[8];
  table[11] = table[3] ^ table[8];
  table[12] = table[4] ^ table[8];
  table[13] = table[5] ^ table[8];
  table[14] = table[6] ^ table[8];
  table[15] = table[7] ^ table[8];

  low = table[b & 0xf];
  temp = table[(b >> 4) & 0xf];
  low ^= temp << 4;
  high = temp >> 60;
  temp = table[(b >> 8) & 0xf];
  low ^= temp << 8;
  high ^= temp >> 56;
  temp = table[(b >> 12) & 0xf];
  low ^= temp << 12;
  high ^= temp >> 52;
  temp = table[(b >> 16) & 0xf];
  low ^= temp << 16;
  high ^= temp >> 48;
  temp = table[(b >> 20) & 0xf];
  low ^= temp << 20;
  high ^= temp >> 44;
  temp = table[(b >> 24) & 0xf];
  low ^= temp << 24;
  high ^= temp >> 40;
  temp = table[(b >> 28) & 0xf];
  low ^= temp << 28;
  high ^= temp >> 36;
  temp = table[(b >> 32) & 0xf];
  low ^= temp << 32;
  high ^= temp >> 32;
  temp = table[(b >> 36) & 0xf];
  low ^= temp << 36;
  high ^= temp >> 28;
  temp = table[(b >> 40) & 0xf];
  low ^= temp << 40;
  high ^= temp >> 24;
  temp = table[(b >> 44) & 0xf];
  low ^= temp << 44;
  high ^= temp >> 20;
  temp = table[(b >> 48) & 0xf];
  low ^= temp << 48;
  high ^= temp >> 16;
  temp = table[(b >> 52) & 0xf];
  low ^= temp << 52;
  high ^= temp >> 12;
  temp = table[(b >> 56) & 0xf];
  low ^= temp << 56;
  high ^= temp >> 8;
  temp = table[b >> 60];
  low ^= temp << 60;
  high ^= temp >> 4;

  mask = -(int64_t)(top3 & 0x1);
  low ^= mask & (b << 61);
  high ^= mask & (b >> 3);
  mask = -(int64_t)((top3 >> 1) & 0x1);
  low ^= mask & (b << 62);
  high ^= mask & (b >> 2);
  mask = -(int64_t)((top3 >> 2) & 0x1);
  low ^= mask & (b << 63);
  high ^= mask & (b >> 1);

  *c0 = low;
  *c1 = high;
}
```

```c
void poly128_mul(const GF2_128 a, const GF2_128 b, uint64_t *c)
{
  uint64_t temp0 = 0;
  uint64_t temp1 = 0;

  poly64_mul(a[1], b[1], &c[3], &c[2]);
  poly64_mul(a[0], b[0], &c[1], &c[0]);

  poly64_mul((a[0] ^ a[1]), (b[0] ^ b[1]), &temp1, &temp0);
  c[1] ^= temp0 ^ c[0] ^ c[2];
  c[2] = temp0 ^ temp1 ^ c[0] ^ c[1] ^ c[3];
}
```

```c
void GF2_128_rdc(const uint64_t *a, GF2_128 c)
{
  uint64_t temp;

  // irreducible polynomial f(x) = x^128 + x^7 + x^2 + x + 1
  temp = a[2] ^ ((a[3] >> 57) ^ (a[3] >> 62) ^ (a[3] >> 63));

  c[1] = a[1] ^ a[3];
  c[1] ^= (a[3] << 7) | (temp >> 57);
  c[1] ^= (a[3] << 2) | (temp >> 62);
  c[1] ^= (a[3] << 1) | (temp >> 63);

  c[0] = a[0] ^ temp;
  c[0] ^= (temp << 7);
  c[0] ^= (temp << 2);
  c[0] ^= (temp << 1);
}
```

MUL 연산 최적화

a와 b를 곱셈하는 과정
a,b: 64bit 정수 2개로 이루어진 128bit 배열

```c
void poly128_mul(const GF2_128 a, const GF2_128 b, uint64_t *c)
{
  uint64_t temp0 = 0;
  uint64_t temp1 = 0;

  poly64_mul(a[1], b[1], &c[3], &c[2]);
  poly64_mul(a[0], b[0], &c[1], &c[0]);

  poly64_mul((a[0] ^ a[1]), (b[0] ^ b[1]), &temp1, &temp0);
  c[1] ^= temp0 ^ c[0] ^ c[2];
  c[2] = temp0 ^ temp1 ^ c[0] ^ c[1] ^ c[3];
}
```

```c
void poly64_mul(const uint64_t a, const uint64_t b, uint64_t *c1, uint64_t *c0)
{
  uint64_t table[16];
  uint64_t temp, mask, high, low;
  uint64_t top3 = a >> 61;

  table[0] = 0;
  table[1] = a & 0x1fffffffffffffffULL;
  table[2] = table[1] << 1;
  table[4] = table[2] << 1;
  table[8] = table[4] << 1;

  table[3] = table[1] ^ table[2];

  table[5] = table[1] ^ table[4];
  table[6] = table[2] ^ table[4];
  table[7] = table[1] ^ table[6];

  table[9] = table[1] ^ table[8];
  table[10] = table[2] ^ table[8];
  table[11] = table[3] ^ table[8];
  table[12] = table[4] ^ table[8];
  table[13] = table[5] ^ table[8];
  table[14] = table[6] ^ table[8];
  table[15] = table[7] ^ table[8];

  low = table[b & 0xf];
  temp = table[(b >> 4) & 0xf];
  low ^= temp << 4;
  high = temp >> 60;
  temp = table[(b >> 8) & 0xf];
  low ^= temp << 8;
  high ^= temp >> 56;
  temp = table[(b >> 12) & 0xf];
  low ^= temp << 12;
  high ^= temp >> 52;
  temp = table[(b >> 16) & 0xf];
  low ^= temp << 16;
  high ^= temp >> 48;
  temp = table[(b >> 20) & 0xf];
  low ^= temp << 20;
  high ^= temp >> 44;
  temp = table[(b >> 24) & 0xf];
  low ^= temp << 24;
  high ^= temp >> 40;
  temp = table[(b >> 28) & 0xf];
  low ^= temp << 28;
  high ^= temp >> 36;
  temp = table[(b >> 32) & 0xf];
  low ^= temp << 32;
  high ^= temp >> 32;
  temp = table[(b >> 36) & 0xf];
  low ^= temp << 36;
  high ^= temp >> 28;
  temp = table[(b >> 40) & 0xf];
  low ^= temp << 40;
  high ^= temp >> 24;
  temp = table[(b >> 44) & 0xf];
  low ^= temp << 44;
  high ^= temp >> 20;
  temp = table[(b >> 48) & 0xf];
  low ^= temp << 48;
  high ^= temp >> 16;
  temp = table[(b >> 52) & 0xf];
  low ^= temp << 52;
  high ^= temp >> 12;
  temp = table[(b >> 56) & 0xf];
  low ^= temp << 56;
  high ^= temp >> 8;
  temp = table[b >> 60];
  low ^= temp << 60;
  high ^= temp >> 4;

  mask = -(int64_t)(top3 & 0x1);
  low ^= mask & (b << 61);
  high ^= mask & (b >> 3);
  mask = -(int64_t)((top3 >> 1) & 0x1);
  low ^= mask & (b << 62);
  high ^= mask & (b >> 2);
  mask = -(int64_t)((top3 >> 2) & 0x1);
  low ^= mask & (b << 63);
  high ^= mask & (b >> 1);

  *c0 = low;
  *c1 = high;
}
```

기존 Reference

```c
.macro mul128_p64 r0, r1, a, b, t0, t1, z
    //r0 = a0 * b0
    pmull \r0\().1q, \a\().1d, \b\().1d
    //r1 = a1 * b1
    pmull2 \r1\().1q, \a\().2d, \b\().2d
    //Reverse low and high parts
    ext.16b \t0, \b, \b, #8
    //t1 = a0 * b1
    pmull \t1\().1q, \a\().1d, \t0\().1d
    //t0 = a1 * b0
    pmull2 \t0\().1q, \a\().2d, \t0\().2d
    //t0 (a0 * b1) + (a1 * b0)
    eor.16b \t0, \t0, \t1
    //xor into place
    ext.16b \t1, \z, \t0, #8
    eor.16b \r0, \r0, \t1
    ext.16b \t1, \t0, \z, #8
    eor.16b \r1, \r1, \t1
.endm
```

ARMv8 최적화

MUL 연산 최적화

기존 Reference

```
void poly64_mul(const uint64_t a, const uint64_t b, uint64_t *c1, uint64_t *c0)
{
  uint64_t table[16];
  uint64_t temp, mask, high, low;
  uint64_t top3 = a >> 61;

  table[0] = 0;
  table[1] = a & 0x1fffffffffffffffULL;
  table[2] = table[1] << 1;
  table[4] = table[2] << 1;
  table[8] = table[4] << 1;

  table[3] = table[1] ^ table[2];

  table[5] = table[1] ^ table[4];
  table[6] = table[2] ^ table[4];
  table[7] = table[1] ^ table[6];

  table[9] = table[1] ^ table[8];
  table[10] = table[2] ^ table[8];
  table[11] = table[3] ^ table[8];
  table[12] = table[4] ^ table[8];
  table[13] = table[5] ^ table[8];
  table[14] = table[6] ^ table[8];
  table[15] = table[7] ^ table[8];
```

테이블 초기화 과정
table 배열: a의 4비트 단위로 분할된 부분들에 대한 곱셈 결과를 저장

```
low = table[b & 0xf];
temp = table[(b >> 4) & 0xf];
low ^= temp << 4;
high = temp >> 60;
temp = table[(b >> 8) & 0xf];
low ^= temp << 8;
high ^= temp >> 56;
temp = table[(b >> 12) & 0xf];
low ^= temp << 12;
high ^= temp >> 52;
temp = table[(b >> 16) & 0xf];
low ^= temp << 16;
high ^= temp >> 48;
temp = table[(b >> 20) & 0xf];
low ^= temp << 20;
high ^= temp >> 44;
temp = table[(b >> 24) & 0xf];
low ^= temp << 24;
high ^= temp >> 40;
temp = table[(b >> 28) & 0xf];
low ^= temp << 28;
high ^= temp >> 36;
temp = table[(b >> 32) & 0xf];
low ^= temp << 32;
high ^= temp >> 32;
temp = table[(b >> 36) & 0xf];
low ^= temp << 36;
high ^= temp >> 28;
temp = table[(b >> 40) & 0xf];
low ^= temp << 40;
high ^= temp >> 24;
temp = table[(b >> 44) & 0xf];
low ^= temp << 44;
high ^= temp >> 20;
temp = table[(b >> 48) & 0xf];
low ^= temp << 48;
high ^= temp >> 16;
temp = table[(b >> 52) & 0xf];
low ^= temp << 52;
high ^= temp >> 12;
temp = table[(b >> 56) & 0xf];
low ^= temp << 56;
high ^= temp >> 8;
temp = table[b >> 60];
low ^= temp << 60;
high ^= temp >> 4;
```

하위 64bit XOR 상위 64bit 과정
b(입력 데이터)를 4비트 단위로 분할
각 4비트는 table 배열에서 인덱스에 해당하는 결과를 조회
조회 결과는 low와 high에 누적되어 저장(64bit 단위)
low는 결과의 하위 64bit, high는 상위 64bit를 저장
low와 high에 결과 누산 시 shift 연산 수행
이후 XOR 연산을 통해 현재까지의 결과와 XOR
이 과정은 b의 각 4비트 단위마다 반복

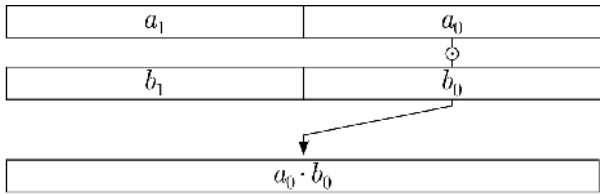
```
  mask = -(int64_t)(top3 & 0x1);
  low ^= mask & (b << 61);
  high ^= mask & (b >> 3);
  mask = -(int64_t)((top3 >> 1) & 0x1);
  low ^= mask & (b << 62);
  high ^= mask & (b >> 2);
  mask = -(int64_t)((top3 >> 2) & 0x1);
  low ^= mask & (b << 63);
  high ^= mask & (b >> 1);

  *c0 = low;
  *c1 = high;
}
```

마스킹 과정(오버플로우 방지?)
top3(a의 상위 3bit)의 상태에 따라 마스킹 실행
mask값은 0또는 -1
각 mask는 b를 shift한 값과 AND 연산 수행
이후 low와 high에 XOR

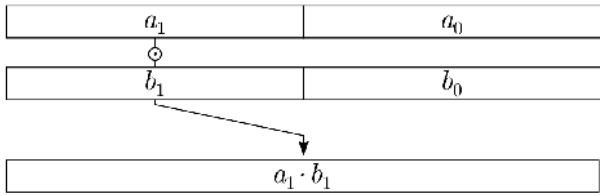
MUL 연산 최적화

ARMv8 최적화

```
.macro mul128_p64 r0, r1, a, b, t0, t1, z
    //r0 = a0 * b0
    pmull \r0\().1q, \a\().1d, \b\().1d
    //r1 = a1 * b1
    pmull2 \r1\().1q, \a\().2d, \b\().2d
    //Reverse low and high parts
    ext.16b \t0, \b, \b, #8
    //t1 = a0 * b1
    pmull \t1\().1q, \a\().1d, \t0\().1d
    //t0 = a1 * b0
    pmull2 \t0\().1q, \a\().2d, \t0\().2d
    //t0 (a0 * b1) + (a1 * b0)
    eor.16b \t0, \t0, \t1
    //xor into place
    ext.16b \t1, \z, \t0, #8
    eor.16b \r0, \r0, \t1
    ext.16b \t1, \t0, \z, #8
    eor.16b \r1, \r1, \t1
.endm
```



PMULL(A, B)



PMULL2(A, B)

다항식의 곱셈(a0 + a1) * (b0 + b1) 과정을 구현
분배 법칙 => (a0*b0) + (a0*b1) + (a1*b0) + (a1*b1)

pmull, pmull2: 벡터 데이터(128bit) 곱셈에 사용하는 명령어
pmull: 하위 64bit 끼리 곱셈 -> r0에 저장
pmull2: 상위 64bit 끼리 곱셈 -> r1에 저장

ext 명령어 사용 -> 하위 64bit와 상위 64bit 반전(b) -> t1에 저장
#8: 8byte(64bit)만큼 이동

pmull, pmull2 -> a와 b(반전된)와 곱셈 -> t0, t1에 저장

t0 eor t1 -> (a0 * b1) + (a1 * b0) -> t0에 저장
다항식에서의 XOR -> 덧셈과 같은 연산

ext -> t0, t1의 값 재배열(z->0으로 초기화된 벡터)
이후 eor 연산 통해 r0, r1에 최종 결과값 저장

최종 결과
r0: a0*b0 + (a0*b1 + a1*b0)
r1: a1*b1 + (a0*b1 + a1*b0)

SQR 연산 최적화

```c
void poly128_sqr(const GF2_128 a, uint64_t *c)
{
  int i;
  for (i = 0; i < NUMWORDS_FIELD; i++)
  {
    c[2 * i    ] = sqr_table[(a[i] >> 28) & 0xf] << 56 |
                   sqr_table[(a[i] >> 24) & 0xf] << 48 |
                   sqr_table[(a[i] >> 20) & 0xf] << 40 |
                   sqr_table[(a[i] >> 16) & 0xf] << 32 |
                   sqr_table[(a[i] >> 12) & 0xf] << 24 |
                   sqr_table[(a[i] >>  8) & 0xf] << 16 |
                   sqr_table[(a[i] >>  4) & 0xf] <<  8 |
                   sqr_table[(a[i]      ) & 0xf];

    c[2 * i + 1] = sqr_table[(a[i] >> 60)      ] << 56 |
                   sqr_table[(a[i] >> 56) & 0xf] << 48 |
                   sqr_table[(a[i] >> 52) & 0xf] << 40 |
                   sqr_table[(a[i] >> 48) & 0xf] << 32 |
                   sqr_table[(a[i] >> 44) & 0xf] << 24 |
                   sqr_table[(a[i] >> 40) & 0xf] << 16 |
                   sqr_table[(a[i] >> 36) & 0xf] <<  8 |
                   sqr_table[(a[i] >> 32) & 0xf];
  }
}
```

기존 Reference

```c
const uint64_t sqr_table[16] = {0x00, 0x01, 0x04, 0x05, 0x10, 0x11, 0x14, 0x15,
                                0x40, 0x41, 0x44, 0x45, 0x50, 0x51, 0x54, 0x55};
```

SQR: 제곱 연산 수행: a에 대한 제곱 연산 수행
a의 4비트 블록을 추출하여 블록의 제곱값을 테이블에서 조회
조회한 제곱값(8비트)을 c 배열에 저장
위 과정을 통해 a배열이 128비트 결과로 확장 -> c 배열에 저장

```
.macro sqr128_p64 r0, r1, a
    //r0 = a0 * a0
    pmull \r0\().1q, \a\().1d, \a\().1d
    //r1 = a1 * a1
    pmull2 \r1\().1q, \a\().2d, \a\().2d
.endm
```

ARMv8 최적화

Pmull을 사용하여 간단하게 제곱 구현
pmull: a0 * a0 -> r0
pmull2: a1 * a1 -> r1

RDC 연산 최적화

- 축약 연산(256bit -> 128bit)

```c
void GF2_128_rdc(const uint64_t *a, GF2_128 c)
{
  uint64_t temp;

  // irreducible polynomial f(x) = x^128 + x^7 + x^2 + x + 1
  temp = a[2] ^ ((a[3] >> 57) ^ (a[3] >> 62) ^ (a[3] >> 63));

  c[1] = a[1] ^ a[3];
  c[1] ^= (a[3] << 7) | (temp >> 57);
  c[1] ^= (a[3] << 2) | (temp >> 62);
  c[1] ^= (a[3] << 1) | (temp >> 63);

  c[0] = a[0] ^ temp;
  c[0] ^= (temp << 7);
  c[0] ^= (temp << 2);
  c[0] ^= (temp << 1);
}
```

기존 Reference

```
.macro rdc_p64 r, a0, a1, t0, t1, p, z
    // Reduce higher part
    // t0 = a1h * 0x87
    pmull2 \t0\().1q, \a1\().2d, \p\().2d
    // xor into place
    ext \t1\().16b, \t0\().16b, \z\().16b, #8
    eor.16b \a1, \a1, \t1
    ext \t1\().16b, \z\().16b, \t0\().16b, #8
    eor.16b \a0, \a0, \t1
    // Reduce lower part
    // t0 = a1l * 0x87
    pmull \t0\().1q, \a1\().1d, \p\().1d
    // xor into place
    eor.16b \r, \a0, \t0
.endm
```

ARMv8 최적화

RDC 연산 최적화

```
void GF2_128_rdc(const uint64_t *a, GF2_128 c)
{
  uint64_t temp;

  // irreducible polynomial f(x) = x^128 + x^7 + x^2 + x + 1
  temp = a[2] ^ ((a[3] >> 57) ^ (a[3] >> 62) ^ (a[3] >> 63));

  c[1] = a[1] ^ a[3];
  c[1] ^= (a[3] << 7) | (temp >> 57);
  c[1] ^= (a[3] << 2) | (temp >> 62);
  c[1] ^= (a[3] << 1) | (temp >> 63);

  c[0] = a[0] ^ temp;
  c[0] ^= (temp << 7);
  c[0] ^= (temp << 2);
  c[0] ^= (temp << 1);
}
```

기존 Reference

축약 연산(256bit -> 128bit) 수행

a(256bit)를 사용하여 c(128bit) 생성
기약다항식 기반($f(x)=x128+x7+x2+x+1$) -> GCM 모드에서 사용

a[0], a[1]: 하위 128bit, a[2], a[3]: 상위 128bit

temp: 기약 다항식에 해당되는 만큼 shift 연산 수행

c[1]: a[1] XOR a[3]의 값에 OR 연산한 값들 xor
c[0]: a[0]과 temp와 XOR 수행 후 shift된 temp들과 xor

RDC 연산 최적화

```
.macro rdc_p64 r, a0, a1, t0, t1, p, z
    // Reduce higher part
    // t0 = a1h * 0x87
    pmull2 \t0\().1q, \a1\().2d, \p\().2d
    // xor into place
    ext \t1\().16b, \t0\().16b, \z\().16b, #8
    eor.16b \a1, \a1, \t1
    ext \t1\().16b, \z\().16b, \t0\().16b, #8
    eor.16b \a0, \a0, \t1
    // Reduce lower part
    // t0 = a1l * 0x87
    pmull \t0\().1q, \a1\().1d, \p\().1d
    // xor into place
    eor.16b \r, \a0, \t0
.endm
```

ARMv8 최적화

축약 연산(256bit -> 128bit) 수행

상위 64비트 축약 과정
pmull2: a1(상위) * 0x87 -> t0
0x87은 기약다항식에 해당하는 값

ext 연산을 통해 t0의 상위, 하위 64비트간 반전 수행
t1에 저장된 ext 연산 결과를 a1과 XOR
두번째 ext 연산을 통해 t0의 반전 한 번 더 수행
t1에 저장된 ext 연산 결과를 a0과 XOR
위 연산들을 통해 a1h와 a0l에 나머지 연산을 적용

pmull: a1l * 0x87 -> t0
eor: t0 XOR a0 -> r1(최종 결과)에 저장

성능 측정 결과

- 성능 측정 환경
  - CPU: Apple M2(4Core 3.49GHz + 4Core 2.42GHz)
  - IDE: Visual Studio Code
  - 최적화 레벨: O3

AIMer	Optimization	KeyGen	Sign	Verify
128	Optimization	81,856	13,381,888	13,325,632
	Reference	115,200	19,937,792	17,729,280
	Diff.	**28.94%**	**32.88%**	**24.84%**
192	Optimization	174,080	18,502,976	16,925,248
	Reference	200,768	33,674,496	31,699,968
	Diff.	**13.29%**	**45.05%**	**46.61%**
256	Optimization	375,680	34,388,544	32,914,752
	Reference	415,872	69,730,752	67,622,272
	Diff	**9.66%**	**50.68%**	**51.33%**

Q & A