Ncc Sign 코드 분석

유튜브 주소 : https://youtu.be/dyoGEg7057I

NCC Sign

- KpqC공모전에 제출된 전자서명 알고리즘
  - Round1을 통과하여 Round 2에 진출
- 격자 기반으로 개발
  - RLWE (Ring Learning With Errors)
    - 고차원의 격자 문제(LWE)를 다항식 구조로 확장한 형태
    - 비순환 다항식(Non-Cyclotomic Polynomial)을 사용하여 RLWE 문제를 설정
    - 기존 순환 다항식 구조의 대수적 약점을 제거
  - RSIS (Ring Short Integer Solution)
    - 기존의 Short Integer Solution (SIS) 문제를 다항식 링 구조로 확장한 형태
    - 서명 생성 시 RSIS 난제 활용
  - SelfTargetRSIS
    - RLWE와 RSIS를 조합한 형태로, 서명 검증 시 활용
- 다항식 및 NTT(수 이론 변환)를 활용하여 설계

NCC Sign 주요 연산

- 다항식 연산(NTT 포함)
  - 다항식 곱셈(pointwise_mul, base_mul, poly_mul_schoolbook 등)
  - 다항식 덧셈, 뺄셈(poly_add, poly_sub, poly_modadd, poly_modsub, poly_shiftl 등)
- NTT 연산
  - 전방 변환(Forward Transform), 역변환(Inverse Transform)
  - 위 변환 과정에서 Radix-2, Radix-3 변환 및 몽고메리 폼 최적화가 활용
- 모듈러 연산
  - montgomery_reduce, mod_add, mod_sub 함수 등으로 구현
- 샘플링
  - NCC는 난수와 시드(Seed)를 활용하여 다항식 계수를 샘플링
  - poly_uniform, poly_uniform_eta, poly_uniform_gamma1 함수 등으로 구현
- 해시 연산
  - SHAKE-256 사용
- 연산 비중 및 중요도: NTT > 모듈러 > 샘플링 >= 해시

다항식 연산(덧셈,뺄셈)

- poly_add
  - 두 다항식 a와 b의 계수를 더하여 다항식 c에 결과 저장
  - 반복문은 다항식의 길이 N 만큼 반복 수행
  - 모듈러 연산은 포함하지 않음(poly_modadd에서 수행)
  - poly는 구조체로, int32_t 배열로 구성되어있음
  - neon 인트린직 함수 중 poly_add를 사용하여 최적화?

```c
void poly_add(poly *c, const poly *a, const poly *b) {
    unsigned int i;
    DBENCH_START();

    for (i = 0; i < N; ++i)
        c->coeffs[i] = a->coeffs[i] + b->coeffs[i];

    DBENCH_STOP(*tadd);
}
```

- poly_sub
  - 두 다항식 a와 b의 계수를 뺄셈하여 다항식 c에 결과 저장
  - 모듈러 연산 포함 X(poly_modsub에서 수행)
  - neon 인트린직 함수 중 poly_sub를 사용하여 최적화?

```c
void poly_sub(poly *c, poly *a, poly *b) {
    unsigned int i;
    DBENCH_START();

    for (i = 0; i < N; ++i)
        c->coeffs[i] = a->coeffs[i] - b->coeffs[i];

    DBENCH_STOP(*tadd);
}
```

- poly_shiftl
  - 다항식의 각 계수를 D 비트만큼 ShiftLeft 연산 수행
  - 즉, 다항식의 각 계수를 2의 D제곱으로 곱하는 연산

```c
void poly_shiftl(poly *a) {
    unsigned int i;
    DBENCH_START();

    for (i = 0; i < N; ++i)
        a->coeffs[i] <<= D;

    DBENCH_STOP(*tmul);
}
```

다항식 연산(곱셈)

```c
void pointwise_mul(int32_t* C, int32_t* A, int32_t* B){
    for(int i=0;i<N;i++){
        C[i] = montgomery_reduce((int64_t)A[i] * B[i]);
    }
}
```

```c
int32_t montgomery_reduce(int64_t a) {
  int32_t t;

  t = (int64_t)(int32_t)a*QINV;
  t = (a - (int64_t)t*Q) >> 32;
  return t;
}
```

- pointwise_mul
  - 다항식의 계수를 순서대로 서로 곱하는 연산 수행(점별 곱셈)
  - NTT 변환 이후 계수를 곱할 때 사용
  - 몽고메리 reduction – 곱셈 결과에 모듈러 연산 적용

```c
void base_mul(int32_t* C, int32_t* A, int32_t* B, int32_t zeta){ //2차식 곱셈
    C[0]=montgomery_reduce((int64_t)A[2]*B[1]);
    C[0]+=montgomery_reduce((int64_t)A[1]*B[2]);
    C[0]=montgomery_reduce((int64_t)C[0]*zeta);
    C[0]+=montgomery_reduce((int64_t)A[0]*B[0]);

    C[1]  = montgomery_reduce((int64_t)A[2]* B[2]);
    C[1]  = montgomery_reduce((int64_t)C[1]* zeta);
    C[1] += montgomery_reduce((int64_t)A[0]* B[1]);
    C[1] += montgomery_reduce((int64_t)A[1]* B[0]);

    C[2]  = montgomery_reduce((int64_t)A[2]* B[0]);
    C[2] += montgomery_reduce((int64_t)A[1]* B[1]);
    C[2] += montgomery_reduce((int64_t)A[0]* B[2]);
}
```

- base_mul
  - 두 다항식의 곱셈을 수행
  - NTT 변환 과정에서 사용
  - Neon의 vmulq_s32 인트린직 함수 사용하여 최적화?
    - Vmulq_32: 4개의 32bit 정수를 한 번에 곱셈

```c
void poly_mul_schoolbook(poly* res, poly* a, poly* b)
{
    // Polynomial multiplication using the schoolbook method, c[x] = a[x]*b[x]
    // SECURITY NOTE: TO BE USED FOR TESTING ONLY.
    uint32_t i, j;

    int32_t c[N << 1];
    int32_t t0;
    for (i = 0; i < (N << 1); i++) c[i] = 0;

    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            t0 = ((int64_t)(Q+a->coeffs[i]) * (Q+b->coeffs[j])) % Q;
            c[i + j] = (c[i + j] + t0) % Q;
        }
    }

    for (i = N + (N >> 1) - 1; i < 2 * N - 1; i++) {
        c[i - (N>>1)] = (c[i - (N>>1)] + c[i]) % Q;
        c[i - N] = (Q + c[i - N] - c[i]) % Q;
    }


    for (i = N; i < N + (N >> 1) - 1; i++)
    {
        c[i - (N>>1)] = (c[i - (N>>1)] + c[i]) % Q;
        c[i - N] = (Q + c[i - N] - c[i]) % Q;
    }

    for (i = 0; i < N; i++)
        res->coeffs[i] = c[i];
}
```

- poly_mul_schoolbook
  - 스쿨북 알고리즘 구현 함수
  - 크기가 큰 다항식에 대해서는 비효율적
    - 계산 복잡도가 O(N^2)임
  - 테스트를 위해 구현되어있는 함수
  - 최적화 할 필요는 X

NTT 연산

- 입력 배열 A에 대해 NTT 변환 수행
  - 결과는 Out 배열에 저장
- 버터플라이 연산 준비 단계
  - 결과는 상,하위 절반으로 나뉨
  - Zeta1은 변환 상수(복소수 루트)
    - NTT 테이블(zetas)에서 선택됨
- Radix-2 버터플라이 연산
  - 데이터 크기를 절반씩 줄여가며 연산
  - Zetas 배열에서 적합한 루트를 선택
- Radix-3 버터플라이 연산
  - Radix-2와 유사
  - 그러나 데이터 집합이 3개
    - Zeta1, zeta2, Wmont의 3부분으로 나뉨
- 딜리시움을 참고하여 최적화?
  - 구조는 유사하나, N값, Radix 등이 다름

```c
void ntt(int32_t * Out, int32_t * A){
    int32_t zeta1;
    int32_t t1;
    int len, start, j, k=0;

    if(Out!=A){
        memcpy(Out,A,sizeof(int32_t)*N);
    }

    zeta1 = zetas[k++];
    for(j = 0; j < N/2; j++)
    {
        t1 = montgomery_reduce((int64_t)zeta1 * Out[j + N/2]);

        Out[j + N/2] = Out[j] + Out[j + N/2] - t1;
        Out[j       ] = Out[j] + t1;
    }

    for (len = N>>2; len > radix2_redlen_ntt; len >>= 1)
    { // radix-2
        for (start = 0; start < N; start += (len << 1))
        {
            zeta1 = zetas[k++];

            for (j = start; j < start + len; j++)
            {
                t1 = montgomery_reduce((int64_t)zeta1 * Out[j + len]);

                Out[j + len] = Out[j] - t1;
                Out[j] = Out[j] + t1;
            }
        }
    }

    for(j = 0; j < N; ++j)
        Out[j] = reduce32(Out[j]);

    for (len = radix2_redlen_ntt; len >=  3 * radix3_len; len >>= 1)
    { // radix-2
        for (start = 0; start < N; start += (len << 1))
        {
            zeta1 = zetas[k++];

            for (j = start; j < start + len; j++)
            {
                t1 = montgomery_reduce((int64_t)zeta1 * Out[j + len]);

                Out[j + len] = Out[j] - t1;
                Out[j] = Out[j] + t1;
            }
        }
    }
```

```c
int32_t zetas[2303] = {
    6691422, 3745396, 1005959, 5560684, 557530, 1359546, 5166674, 5884035, 4422884, 4240425,
5387571, 1046739, 2694230, 3252563, 6563702, 6365869, 2241533, 6130832, 3005748, 2818295,
1059740, 1575156, 6639601, 1563110, 7206912, 1200488, 285221, 8040721, 2282474, 6468071,
4739225, 2021449, 3517630, 7346728, 4905548, 5238671, 3440514, 4703105, 940539, 2892723,
6587485, 4244468, 3541424, 3284715, 964913, 106719, 2439163, 2652420, 2954142, 8379323,
7950516, 8059860, 1193423, 7129183, 7731829, 264713, 8060094, 445982, 8011185, 1369059,
7955478, 5679661, 6136964, 5113226, 3683432, 7938303, 6949595, 5152479, 1033092, 1728878,
8278066, 3908887, 4478964, 1661123, 751008, 3037667, 1719154, 4320384, 7339862, 4426392,
7097598, 7116055, 3331941, 1686410, 6321919, 1086929, 6201985, 1269346, 4028114, 1093944,
3364626, 7532885, 7083882, 6208581, 6588778, 2989190, 4018245, 905487, 7903618, 6818581,
2510402, 3495458, 8269219, 5100927, 4821549, 7197271, 7365006, 7566618, 5426137, 1390757,
4282185, 3438679, 4615431, 4943292, 483603, 8193371, 1190228, 2472636, 7326792, 4641031,
6575361, 6820008, 586507, 8339833, 1302054, 7839232, 2038881, 7951916, 2149249, 4851479,
6738222, 7925831, 1335443, 8252838, 2533493, 351317, 69907, 2530659, 67928, 4628830,
6097080, 6520199, 7059579, 3741732, 5872664, 1844725, 3046182, 6710463, 1533306, 777419,
8184245, 87056, 7368142, 4291236, 613216, 4955376, 1690952, 4474239, 6784644, 2909821,
7943851, 948115, 3618608, 3420707, 4466850, 8249857, 4873863, 2031465, 2812877, 7672067,
1196093, 1679678, 6892943, 280851, 4838437, 286108, 2180160, 5946783, 3280475, 2953005,
5294676, 1176, 5818203, 7589923, 6465761, 5542137, 6290777, 4603252, 7727308, 625905,
6148742, 4934225, 8349238, 6933212, 7729342, 3845758, 1099113, 6380048, 1313510, 1844170,
4995993, 8090989, 460717, 2321474, 6689731, 6053013, 577288, 3209389, 3044372, 2313948,
1145465, 7998760, 2770361, 5631248, 2577595, 7779883, 3552194, 226028, 4944063, 4270717,
4051664, 4953682, 4416567, 1351657, 806825, 574573, 304984, 6607630, 5841098, 6844990,
5184478, 5421164, 4815963, 7879235, 7430274, 6207605, 3475370, 459765, 7611163, 7678294,
```

```c
#if NIMS_TRI_NTT_MODE != 3
    int32_t zeta2;
    int32_t t2,t3,t4;

    for (len = radix3_len; len >= 1; len = len / 3)
    { // radix-3
        for (start = 0; start < N; start += 3 * len)
        {
            zeta1 = zetas[k++];
            zeta2 = zetas[k++];

            for(j = start; j < start + len; j++)
            {
                t1 = montgomery_reduce((int64_t)zeta1 * Out[j + len]);
                t2 = montgomery_reduce((int64_t)zeta2 * Out[j + 2*len]);
                t3 = montgomery_reduce((int64_t)Wmont * t1); //w
                t4 = montgomery_reduce((int64_t)W2mont * t2); //w^2

                t1 = t1 + t2;
                t3 = t3 + t4;

                Out[j + 2*len] = Out[j] - (t1 + t3);
                Out[j + len] = Out[j] + t3;
                Out[j     ] = Out[j] + t1;
            }
        }
    }
#endif
```

NTT 연산: dilithium과 비교

- NCC Sign은 Radix2,3를 구현하나, Dilithium은 Radix2만 구현
- Dilithium의 구현이 보다 더 간결
  - 별도의 최종 정규화 과정 X(모든 값을 몽고메리 Reduction으로 처리)
- NCC는 보다 복잡한 연산을 처리할 수 있게 구현

```c
void PQCLEAN_MLDSA44_CLEAN_ntt(int32_t a[N]) {
    unsigned int len, start, j, k;
    int32_t zeta, t;

    k = 0;
    for (len = 128; len > 0; len >>= 1) {
        for (start = 0; start < N; start = j + len) {
            zeta = zetas[++k];
            for (j = start; j < start + len; ++j) {
                t = PQCLEAN_MLDSA44_CLEAN_montgomery_reduce((int64_t)zeta * a[j + len]);
                a[j + len] = a[j] - t;
                a[j] = a[j] + t;
            }
        }
    }
}
```

```c
int32_t PQCLEAN_MLDSA44_CLEAN_montgomery_reduce(int64_t a) {
    int32_t t;

    t = (int32_t)((uint64_t)a * (uint64_t)QINV);
    t = (a - (int64_t)t * Q) >> 32;
    return t;
}
```

```c
void ntt(int32_t * Out, int32_t * A){
    int32_t zeta1;
    int32_t t1;
    int len, start, j, k=0;

    if(Out!=A){
        memcpy(Out,A,sizeof(int32_t)*N);
    }

    zeta1 = zetas[k++];
    for(j = 0; j < N/2; j++)
    {
        t1 = montgomery_reduce((int64_t)zeta1 * Out[j + N/2]);

        Out[j + N/2] = Out[j] + Out[j + N/2] - t1;
        Out[j      ] = Out[j] + t1;
    }

    for (len = N>>2; len > radix2_redlen_ntt; len >>= 1)
    { // radix-2
        for (start = 0; start < N; start += (len << 1))
        {
            zeta1 = zetas[k++];

            for (j = start; j < start + len; j++)
            {
                t1 = montgomery_reduce((int64_t)zeta1 * Out[j + len]);

                Out[j + len] = Out[j] - t1;
                Out[j] = Out[j] + t1;
            }
        }
    }

    for(j = 0; j < N; ++j)
        Out[j] = reduce32(Out[j]);

    for (len = radix2_redlen_ntt; len >=  3 * radix3_len; len >>= 1)
    { // radix-2
        for (start = 0; start < N; start += (len << 1))
        {
            zeta1 = zetas[k++];

            for (j = start; j < start + len; j++)
            {
                t1 = montgomery_reduce((int64_t)zeta1 * Out[j + len]);

                Out[j + len] = Out[j] - t1;
                Out[j] = Out[j] + t1;
            }
        }
    }
```

```c
#if NIMS_TRI_NTT_MODE != 3
    int32_t zeta2;
    int32_t t2,t3,t4;

    for (len = radix3_len; len >= 1; len = len / 3)
    { // radix-3
        for (start = 0; start < N; start += 3 * len)
        {
            zeta1 = zetas[k++];
            zeta2 = zetas[k++];

            for(j = start; j < start + len; j++)
            {
                t1 = montgomery_reduce((int64_t)zeta1 * Out[j + len]);
                t2 = montgomery_reduce((int64_t)zeta2 * Out[j + 2*len]);
                t3 = montgomery_reduce((int64_t)Wmont * t1); //w
                t4 = montgomery_reduce((int64_t)W2mont * t2); //w^2

                t1 = t1 + t2;
                t3 = t3 + t4;

                Out[j + 2*len] = Out[j] - (t1 + t3);
                Out[j + len] = Out[j] + t3;
                Out[j    ] = Out[j] + t1;
            }
        }
    }
#endif
```

모듈러 연산

- Mod_add
  - 두 값을 더한 결과를 모듈러 특정 모듈러(Q)에 대해 모듈러 연산 수행
  - 두 값을 더한 후 결과가 Q를 초과하면 Q를 빼주는 방식으로 구현

```c
int32_t mod_add(int32_t a, int32_t b)
{
    int32_t t;
    t=(a+b);
    t=t-Q;
    t += (t >> 31) & Q;
    t += (t >> 31) & Q;
    t += (t >> 31) & Q;

    return (uint32_t)t;
}
```

- Mod_sub
  - 두 값을 뺀 결과를 Q에 대해 모듈러 연산 수행
  - add, sub 모두 연산은 단순하지만 조건문 처리로 인해 최적화가 어려울 것으로 예상

```c
int32_t mod_sub(int32_t a, int32_t b)
{
    int32_t t;
    t=a-b;
    t=t-Q;
    t += (t >> 31) & Q;
    t += (t >> 31) & Q;
    t += (t >> 31) & Q;

    return (uint32_t)t;
}
```

- Montgomery_reduce
  - 주어진 곱셈 결과를 Q에 대해 모듈러 연산 수행
  - 대부분의 다항식 곱셈 및 변환에서 사용되는 핵심 연산
  - 어셈블리 명령어 SMLAL 및 쉬프트 연산을 활용하여 최적화?

```c
int32_t montgomery_reduce(int64_t a) {
  int32_t t;

  t = (int64_t)(int32_t)a*QINV;
  t = (a - (int64_t)t*Q) >> 32;
  return t;
}
```

- Poly_reduce
  - 주어진 다항식의 모든 계수를 Q에 대해 모듈러 연산 수행
- Reduce32
  - 특정 값을 32bit 내에서 Q에 대한 모듈러 연산 수행
  - 기본적으로 빠르게 동작하기 때문에 추가 최적화가 가능할지?

```c
void poly_reduce(poly *a) {
  unsigned int i;
  DBENCH_START();

  for(i = 0; i < N; ++i)
    a->coeffs[i] = reduce32(a->coeffs[i]);

  DBENCH_STOP(*tred);
}
```

```c
int32_t reduce32(int32_t a) {
  int32_t t;

  t = (a + (1 << 22)) >> 23;
  t = a - t*Q;
  //t=a%Q;
  return t;
}
```

Q & A