

NTRU+

SMAUG-T

KEM 구현 목차 상세

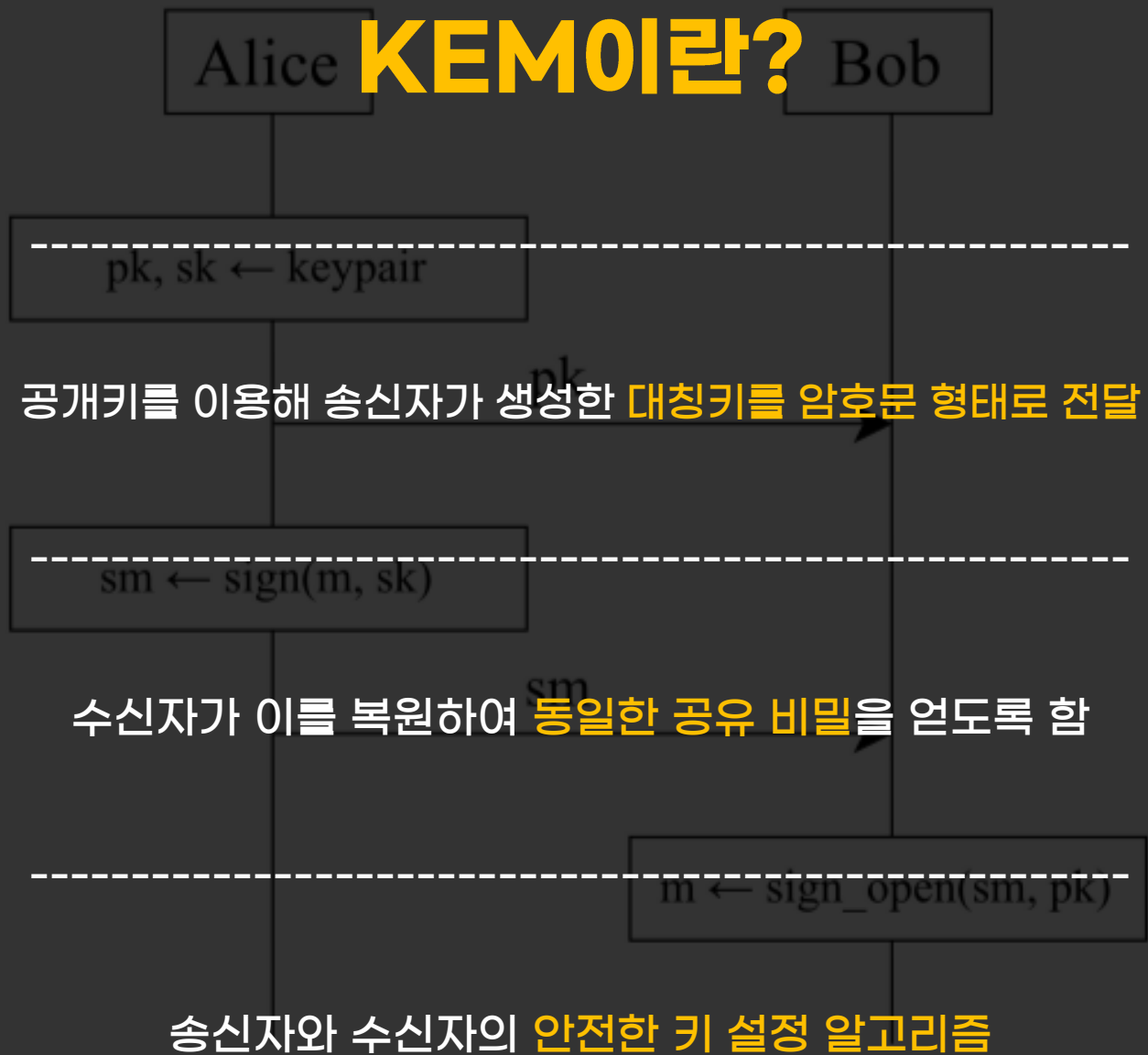
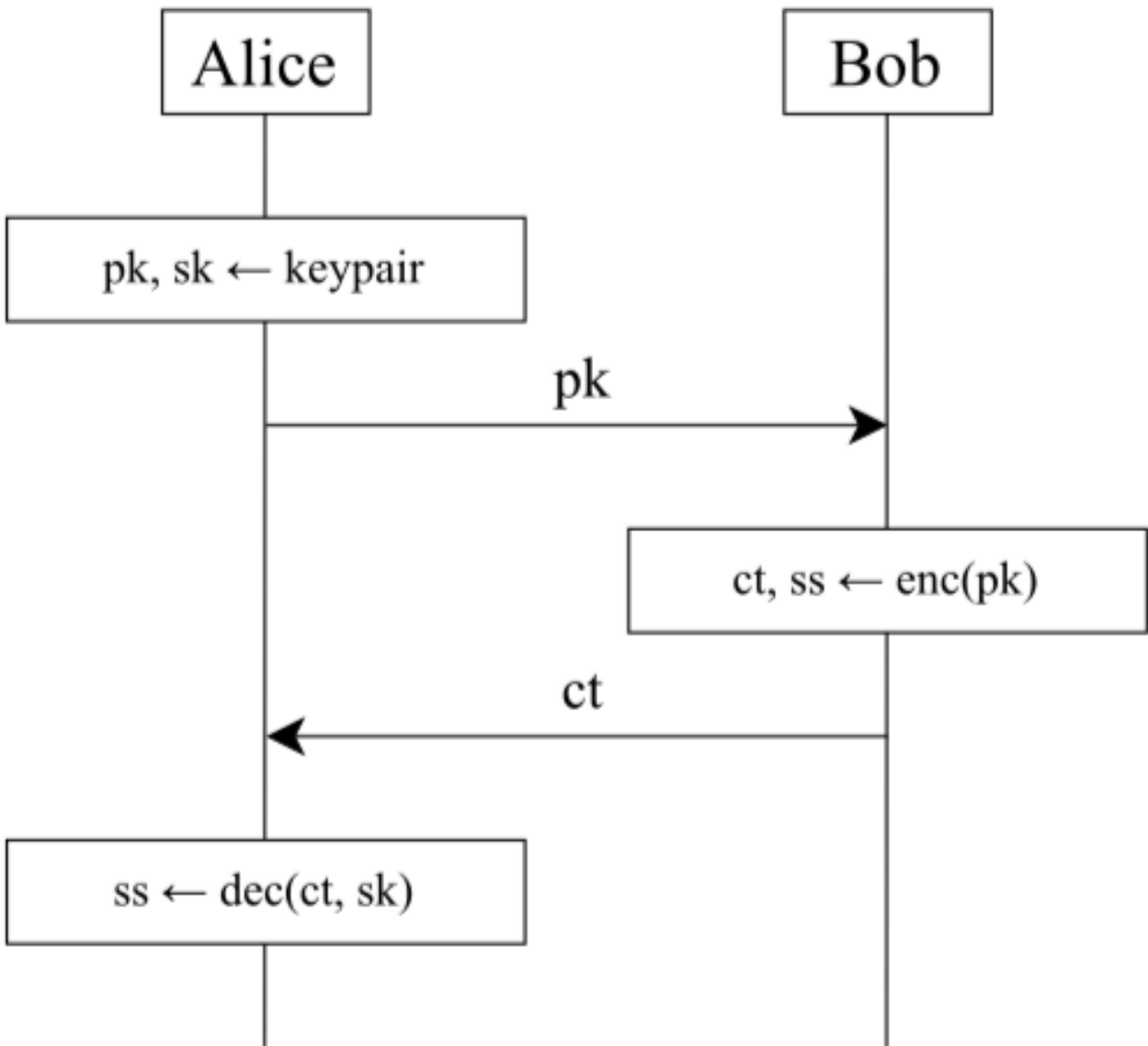
KEM 개요 및 동작 원리

NTRU+ 기반 PQC KEM 구조

NTRU+ NTT 구현 최적화

SMAUG-T 기반 MLWE/MLWR 구조

SMAUG-T 다항식 연산 최적화



Alice

Bob

$pk, sk \leftarrow \text{keypair}$

pk

$ct, ss \leftarrow \text{enc}(pk)$

ct

$ss \leftarrow \text{dec}(ct, sk)$

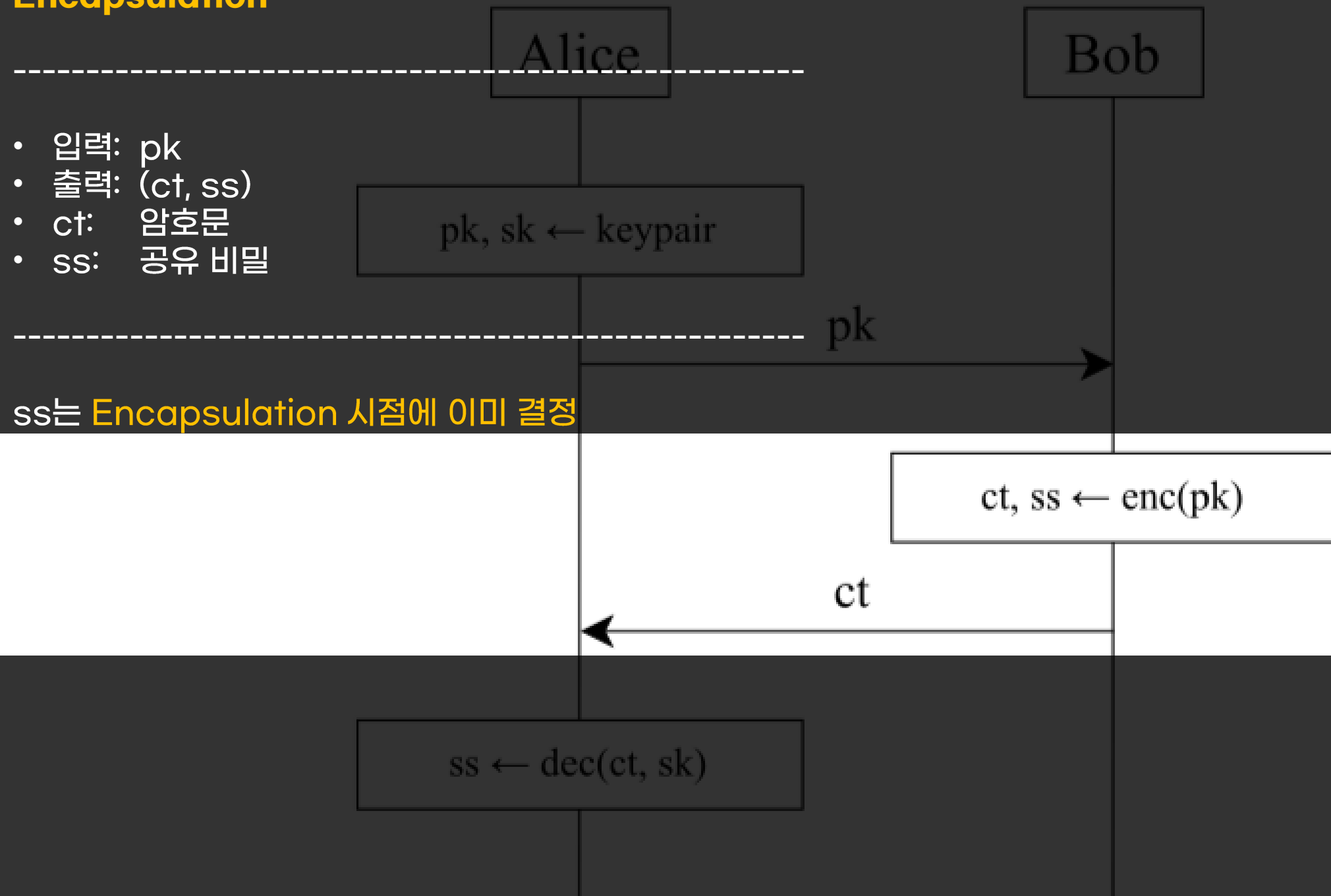
Key Generation

- 출력: (pk, sk)
- pk: 공개키
- sk: 개인키

PQC KEM에서는 공개키 크기가 수 KB 수준인 경우가 많음

Encapsulation

- 입력: pk
- 출력: (ct, ss)
- ct : 암호문
- ss : 공유 비밀



Alice

Bob

Decapsulation

$pk, sk \leftarrow \text{keypair}$

- 입력: 자신의 sk , 수신된 ct
- 출력: ss'

pk

$ct, ss \leftarrow \text{enc}(pk)$

ct

$ss \leftarrow \text{dec}(ct, sk)$

정상 구현 시 $ss' = ss$

NTRU+ 기반 문제 (두 가지 격자 문제에 동시에 기반)

• NTRU Problem

- 고전적인 NTRU 격자 문제
- 모듈러 q 에서 정의된 다항식 환 $R_q = \mathbb{Z}_q[x] / \langle f(x) \rangle$ 상에서 어떤 공개된 다항식 h 가 주어졌을 때

$$h = g \cdot f^{-1} \pmod{q}$$

로부터 계수가 짧은 두 다항식 f, g 를 찾는 문제

- 여기서 f^{-1} 은 환 R_q 에서의 곱셈 역원 (즉 $h \cdot f \equiv g \pmod{q}$)

NTRU+ 기반 문제 (두 가지 격자 문제에 동시에 기반)

짧은 다항식이란?

- f, g 의 계수들은 $\{-1, 0, 1\}$ 또는 centered binomial distribution
→ 절댓값이 매우 작은 정수들 문제
- 모듈러 q 에서 정의된 다항식 환 $R_q = \mathbb{Z}_q[x] / \langle f(x) \rangle$ 상에서
- 단순히 해가 존재하는 것이 아니라 작은 해를 찾아야 한다는 점이 문제를 어렵게 함

$$h = g \cdot f^{-1} \pmod{q}$$

로부터 계수가 짧은 두 다항식 f, g 를 찾는 문제

단순 대수 문제라면 쉬움 → 임의의 f 를 고르고 $g = hf \pmod{q}$ 로 계산하면 끝
(하지만 이렇게 얻은 g 는 전혀 짧지 않음)

실제 문제의 본질 → 선형방정식 $hf - g = 0 \pmod{q}$ 를 만족하면서
 f, g 가 동시에 매우 작은 벡터가 되도록 해야함 → 전형적인 격자 문제

NTRU+ 기반 문제 (두 가지 격자 문제에 동시에 기반)

• Ring-LWE Problem

- NTRU+의 보안 분석은 RLWE 관점에서도 동시 평가
→ NTRU+는 구조적으로는 NTRU이지만
공개키와 암호문이 Ring-LWE 샘플과 거의 동일한 분포를 가지도록
설계되었기 때문에 보안 분석을 Ring-LWE 난이도로 환원

정보 유형	RLWE	NTRU+
식	$a, b = a \cdot s + e \pmod{q}$	$g = hf \pmod{q}$
주어진 값	a	h
숨겨진 값	s	f
	e	g

수학적 구조 (Ring & Polynomial)

- **Polynomial Ring**

- $R_q = \mathbb{Z}_q[x] / \langle f(x) \rangle$
- $f(x) = x^n - x^{\frac{n}{2}} + 1$: cyclotomic trinomial
- $q = 3457$
- 기존 NTRU의 $x^n - 1$, Kyber의 $x^n + 1$ 과 다름

NTRU+768 ref를 중심으로 (상위 API 관점 @kem.c)

```
int crypto_kem_keypair(uint8_t *pk, uint8_t *sk)
{
    uint8_t coins[NTRUPLUS_SYMBYTES];

    poly f, finv;
    poly g, ginv;

    do {
        randombytes(out: coins, outlen: sizeof coins);
    } while (genf_derand(f: &f, finv: &finv, coins));

    do {
        randombytes(out: coins, outlen: sizeof coins);
    } while (geng_derand(g: &g, ginv: &ginv, coins));

    crypto_kem_keypair_derand(pk, sk, f: &f, finv: &finv, g: &g, ginv: &ginv);
    return 0;
}
```

```
int crypto_kem_enc(uint8_t *ct, uint8_t *ss, const uint8_t *pk)
{
    uint8_t coins[NTRUPLUS_N / 8];

    randombytes(out: coins, outlen: sizeof coins);
    return crypto_kem_enc_derand(ct, ss, pk, coins);
}
```

```
int crypto_kem_dec(uint8_t *ss, const uint8_t *ct, const uint8_t *sk)
{
    uint8_t msg[NTRUPLUS_N / 8 + NTRUPLUS_SYMBYTES];
    uint8_t buf1[NTRUPLUS_POLYBYTES];
    uint8_t buf2[NTRUPLUS_POLYBYTES];
    uint8_t buf3[NTRUPLUS_POLYBYTES + NTRUPLUS_SYMBYTES];

    int8_t fail;

    poly c, f, hinv;
    poly r1, r2;
    poly m1, m2;

    poly_frombytes(r: &c, a: ct);
    poly_frombytes(r: &f, a: sk);
    poly_frombytes(r: &hinv, a: sk + NTRUPLUS_POLYBYTES);

    poly_basemul(r: &m1, a: &c, b: &f);
    poly_invntt(r: &m1, a: &m1);
    poly_crepmod3(r: &m1, a: &m1);

    poly_ntt(r: &m2, a: &m1);
    poly_sub(r: &c, a: &c, b: &m2);
    poly_basemul(r: &r2, a: &c, b: &hinv);
```

NTRU+/
├──

api.h	# KEM API 인터페이스
params.h	# 파라미터 정의
kem.c	# 키 생성, 암호화, 복호화 로직
poly.h, poly.c	# 다항식 연산 (샘플링 포함)
ntt.h, ntt.c	# Number Theoretic Transform 구현
symmetric.h, symmetric.c	# 대칭키 연산 (해시 함수)
fips202/	# SHAKE256 해시 함수 구현
├── fips202.h	
└── fips202.c	
randombytes.h, randombytes.c	# 랜덤 바이트 생성
rng.h, rng.c	# 난수 생성기 (테스트용)
test.c	# 테스트 프로그램
PQCgenKAT_kem.c	# NIST KAT 생성 프로그램
cpucycles.h	# CPU 사이클 측정
Makefile	# 빌드 설정

NTRU KEM

```
static inline int crypto_kem_enc_derand(uint8_t *ct, uint8_t *ss,  
                                         const uint8_t *pk,  
                                         const uint8_t *coins)
```

- NTRU+의 기반 문제: Module-NTRU

- 환구조: $R_q = Z_q[x] / (x^n + 1)$
- 공개키: $h = g \cdot f^{-1} \pmod{q}$
- 암호문: $c = r \cdot h + m \pmod{q}$
- r : ephemeral secret (랜덤 다항식)
- m : 세션용 랜덤 메시지
- ss : 최종 공유 비밀

NTRU KEM1 (메시지 *msg* 생성)

```
static inline int crypto_kem_enc_derand(uint8_t *ct, uint8_t *ss,  
                                         const uint8_t *pk,  
                                         const uint8_t *coins)
```

```
    for (size_t i = 0; i < NTRUPLUS_N / 8; i++)  
        msg[i] = coins[i];  
  
    hash_f(buf: msg + NTRUPLUS_N / 8, msg: pk);  
    hash_h(buf: buf1, msg);
```

- *coins*를 해시하여 내부 메시지 *msg* 생성
- *msg*은 보통 소수 q 보다 작은 계수 다항식

NTRU KEM2 (ephemeral secret 다항식 r 생성)

```
static inline int crypto_kem_enc_derand(uint8_t *ct, uint8_t *ss,  
                                         const uint8_t *pk,  
                                         const uint8_t *coins)
```

```
poly_cbd1(r: &r, buf: buf1 + NTRUPLUS_SYMBYTES);  
poly_ntt(r: &r, a: &r);
```

- $buf1$ 에서 비트들을 읽어 중심 이항 분포로 r 의 계수들을 샘플링
- r 을 NTT 도메인으로 올림

NTRU KEM3 (r을 바이트로 내린 뒤 hash해서 인코딩)

```
static inline int crypto_kem_enc_derand(uint8_t *ct, uint8_t *ss,  
                                         const uint8_t *pk,  
                                         const uint8_t *coins)
```

```
    poly_tobytes(r: buf2, a: &r);  
    hash_g(buf: buf2, msg: buf2);  
    poly_sotp_encode(r: &m, msg, buf: buf2);  
    poly_ntt(r: &m, a: &m);
```

- $msg (= coins || H(pk))$ 와 $buf2 (= H(r))$ 를 이용해 m 다항식 인코딩
- $m \leftarrow Encode(msg, H(r))$

NTRU KEM4 (암호문 c 계산)

```
static inline int crypto_kem_enc_derand(uint8_t *ct, uint8_t *ss,  
                                         const uint8_t *pk,  
                                         const uint8_t *coins)  
  
    poly_frombytes(r: &h, a: pk);  
    poly_basemul_add(r: &c, a: &h, b: &r, c: &m);  
    poly_tobytes(r: ct, a: &c);
```

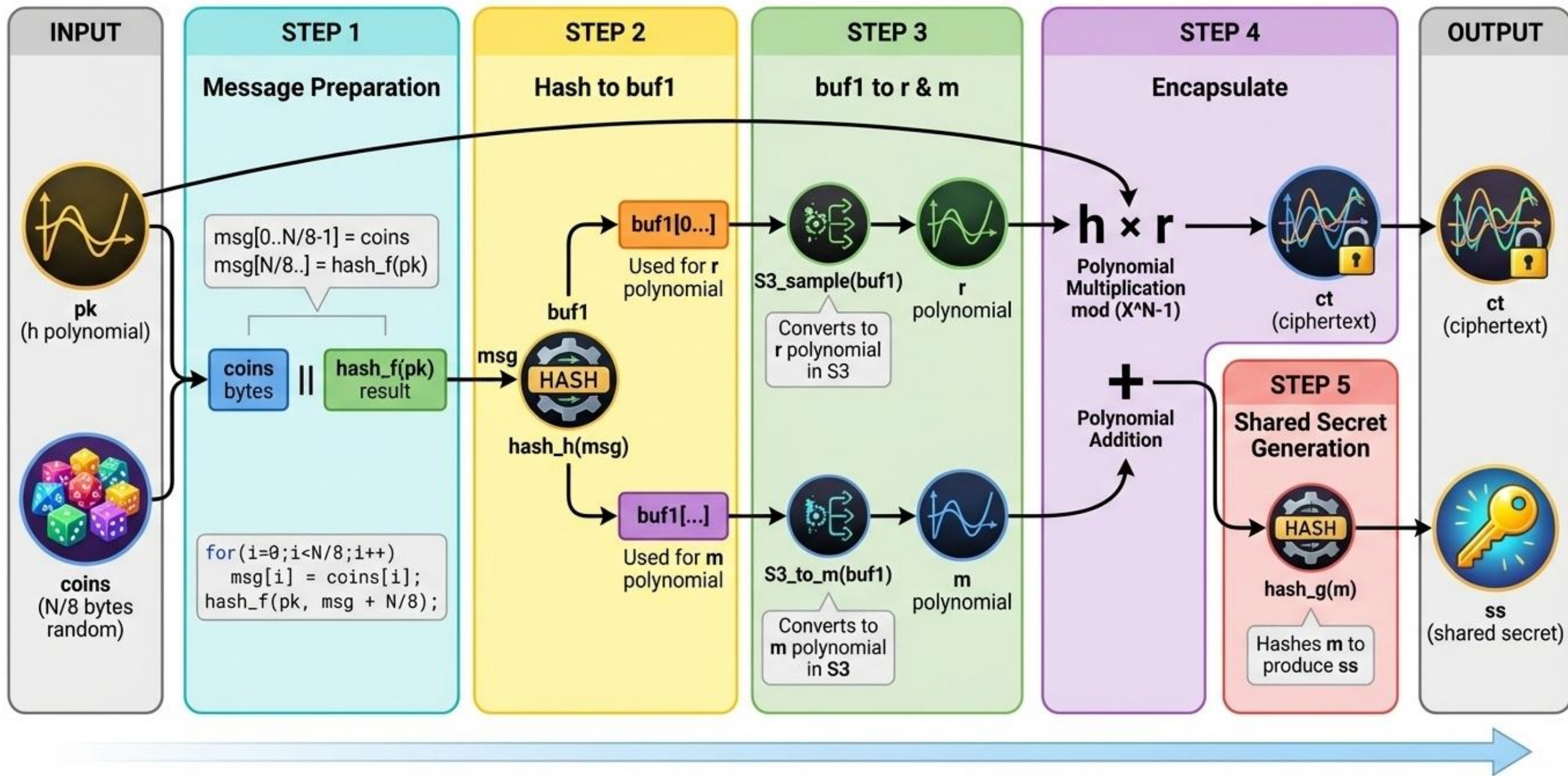
- $c = h \cdot r + m$ 를 NTT 도메인에서 수행하는 함수

NTRU KEM5 (shared secret 추출)

```
static inline int crypto_kem_enc_derand(uint8_t *ct, uint8_t *ss,  
                                         const uint8_t *pk,  
                                         const uint8_t *coins)  
  
    for (size_t i = 0; i < NTRUPLUS_SSBYTES; i++)  
        ss[i] = buf1[i];
```

- *buf1* 앞부분을 그대로 shared secret *ss*로 사용

NTRU+ Encapsulation - crypto_kem_enc_derand() Detailed Flow



NTRU+ NTT

- **NTRU+에서 다루는 환**

- $R_q = \mathbb{Z}_q[X] / (X^{768} - X^{384} + 1)$

- **NTT의 목적**

- 다항식 $a(X) \in R_q$ 를 작은 차수의 다항식 환들의 곱표현으로 바꾸는 것

- 전체 환 분해의 큰 그림

- $\mathbb{Z}_q[X] / (X^{768} - X^{384} + 1) \rightarrow \prod_{i=1}^{192} \mathbb{Z}_q[X] / (X^4 - \xi_i)$

- 차수 768짜리 하나의 큰 환 → 차수 4짜리 작은 환 192개의 곱

- **쪼개어지는 구조** ($f(X) = X^{768} - X^{384} + 1$) $\rightarrow 768 = 2 \times 3 \times 2^7$

- $768 = 2 \times 384$

- $384 = 3 \times 128$

- $128 = 2^7$

NTRU+ NTT 수학적 해석

NTRU+에서 사용하는 환

$$R_q = Z_q[X] / \langle f(X) \rangle \text{ where } f(X) = X^{768} - X^{384} + 1$$

• **1단계 분해:** $768 = 2 \times 384$

• 다항식을 두 부분으로 나눔

$$a(X) = a_0(X) + X^{384}a_1(X), \quad \deg a_0, \deg a_1 < 384$$

• 핵심 관계식

$$X^{768} = X^{384} - 1 \rightarrow (X^{384})^2 - X^{384} + 1 = 0$$

• 즉

$$Y^2 - Y + 1 = 0 \text{ where } Y = X^{384}$$

NTRU+

NTRU+에서 사용하는 환

NTT 수학적 해석

$$R_q = \mathbb{Z}_q[X] / \langle f(X) \rangle \text{ where } f(X) = X^{768} - X^{384} + 1$$

• 1단계 분해: $768 = 2 \times 384$

• 해당 2차 다항식의 근을 α_1, α_2 라 두면

$$Y^2 - Y + 1 = (Y - \alpha_1)(Y - \alpha_2)$$

• 다항식

• 따라서 환은 다음처럼 분해

• 핵심

$$\mathbb{Z}_q[X] / (X^{768} - X^{384} + 1) \cong \prod_{j=1}^2 \mathbb{Z}_q[X] / (X^{384} - \alpha_j)$$

• 1단계 NTT의 최종 연산

• 즉

$$a_0(X) + X^{384} a_1(X) \rightarrow a_0(X) + \alpha_j a_1(X)$$

• 즉 $X^{384} = \alpha_j$ 평가를 수행

NTRU+ NTT 수학적 해석

NTRU+에서 사용하는 환

$$R_q = Z_q[X] / \langle f(X) \rangle \text{ where } f(X) = X^{768} - X^{384} + 1$$

• 2단계 분해: $384 = 3 \times 128$

- 현재 환의 구조: $Z_q[X] / (X^{384} - \alpha)$

- 다항식을 다음과 같이 정리

- $b(X) = b_0(X) + X^{128}b_1(X) + X^{256}b_2(X)$

- 여기서 $Z = X^{128} \rightarrow Z^3 = \alpha$

- 다음 3차 다항식 분해 필요

- $Z^3 - \alpha = (Z - \beta_1)(Z - \beta_2)(Z - \beta_3)$

- 따라서 환은 다음과 같이 분해

- $Z_q[X] / (X^{384} - \alpha) \cong \prod_{k=1}^3 Z_q[X] / (X^{128} - \beta_k)$

- 2단계 NTT (radix-3)는 다음을 계산하는 선형 변환

- $b_0 + b_1Z + b_2Z^2 \rightarrow b_0 + \beta_k b_1 + \beta_k^2 b_2$

NTRU+ NTT 수학적 해석

NTRU+에서 사용하는 환

$$R_q = \mathbb{Z}_q[X] / \langle f(X) \rangle \text{ where } f(X) = X^{768} - X^{384} + 1$$

• 3단계 분해: $128 = 2^7$

- 현재 환의 구조: $\mathbb{Z}_q[X] / (X^{128} - \beta)$
- 이를 반복적으로 분해
 - $X^{2m} - \gamma = (X^m - \delta)(X^m + \delta), \delta^2 = \gamma$
 - $\mathbb{Z}_q[X] / (X^{2m} - \gamma) \cong \mathbb{Z}_q[X] / (X^m - \delta) \times \mathbb{Z}_q[X] / (X^m + \delta)$
 - 이를 $128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4$ 까지 반복

NTRU+ NTT 수학적 해석

NTRU+에서 사용하는 환

$$R_q = Z_q[X] / \langle f(X) \rangle \text{ where } f(X) = X^{768} - X^{384} + 1$$

• 최종 분해 결과

$$\bullet Z_q[X] / (X^{768} - X^{384} + 1) \xrightarrow{\times 2} \prod_{j=1}^2 Z_q[X] / (X^{384} - \alpha_j) \xrightarrow{\times 3} \prod Z_q[X] / (X^{128} - \beta) \xrightarrow{\times 2^5} \prod_{i=1}^{192} Z_q[X] / (X^4 - \xi_i)$$

• NTT가 실제로 계산하는 것

- $\bullet a(X) \rightarrow \left(a(\xi_i, 0), a(\xi_i, 1), a(\xi_i, 2), a(\xi_i, 3) \right)_{i=1}^{192}$
- \bullet 여기서 각 4개 묶음은 $X^4 = \xi_i$ 라는 관계를 갖는 작은 환의 좌표 표현

• NTRU+ NTT 요약

- $\bullet X^{768} - X^{384} + 1$ 을 아래와 같이 인수분해하여 $\prod Z_q[X] / (X^4 - \xi_i)$ 로 바꾸는 단계적 환 동형 계산
 - $\bullet (X^{384} - \alpha)$
 - $\bullet (X^{128} - \beta)$
 - $\bullet (X^4 - \xi)$

NTRU+ NTT 코드적 해석 (1단계 분해)

- NTRU+에서 사용하는 환

$$R_q = \mathbb{Z}_q[X] / \langle f(X) \rangle \text{ where } f(X) = X^{768} - X^{384} + 1$$

- 다항식을 다음과 같이 뒀 $a(X) = a_0(X) + X^{384}a_1(X), \deg < 384$

- 환 관계식에서 $Y = X^{384}$ 는 $Y^2 - Y + 1 = 0$ 을 만족

- 해당 2차식의 근을 α_1, α_2 라 하면

$$R_q \cong \prod_{j=1}^2 \mathbb{Z}_q[X] / (X^{384} - \alpha_j)$$

- 동형의 좌표는 $X^{384} = \alpha_j$ 를 대입하는 형태

$$a_0(X) + X^{384}a_1(X) \rightarrow a_0(X) + \alpha_j a_1(X)$$

NTRU+

- NTRU+에
- 다항식을 D
- 환 관계식에

```
int k = 1;
```

```
zeta1 = zetas[k++];
```

```
for (int i = 0; i < NTRUPLUS_N / 2; i++)
```

```
{
```

```
    t1 = fqmul(a: zeta1, b: a[i + NTRUPLUS_N / 2]);
```

```
    r[i + NTRUPLUS_N / 2] = a[i] + a[i + NTRUPLUS_N / 2] - t1;
```

```
    r[i] = a[i] + t1;
```

```
}
```

< 384

zeta는 다항식의 근 (root)

여기에 수식을 입력하십시오.

1단계의 출발 다항식: $f(X) = X^{768} - X^{384} + 1$

• 동형의 좌표는 $X^{384} = \alpha_j$ 를 대입하는 형태

여기서 $Y = X^{384}$ 로 치환 $f(X) = Y^2 - Y + 1$

따라서 zeta는 다음을 만족하는 값:

$$\alpha^2 - \alpha + 1 \equiv 0 \pmod{q}$$

zeta는 체 Z_q 에서 $Y^2 - Y + 1$ 의 해 (root)

NTRU+

- NTRU+에
- 다항식을 D
- 환 관계식

```
int k = 1;
```

```
zeta1 = zetas[k++];
```

```
for (int i = 0; i < NTRUPLUS_N / 2; i++)
```

```
{
```

```
    t1 = fqmul(a: zeta1, b: a[i + NTRUPLUS_N / 2]);
```

```
    r[i + NTRUPLUS_N / 2] = a[i] + a[i + NTRUPLUS_N / 2] - t1;
```

```
    r[i] = a[i] + t1;
```

```
}
```

< 384

이차방정식 $Y^2 - Y + 1$ 의 판별식은 $\Delta = (-1)^2 - 4 \cdot 1 \cdot 1 = -3$

zeta가 존재하려면 -3 이 Z_q 에서 **제곱 잉여**여야 함

- $\exists x \in Z_q$ such that $x^2 \equiv -3 \pmod{q}$

zeta를 이차방정식에서 구하는 방식

$$\alpha = \frac{1 \pm \sqrt{-3}}{2} \pmod{q}$$

제곱잉여

어떤 수가 제곱해서 나올 수 있는 값인가?

$\sqrt{-3} \pmod{3457}$ 을 구해야 함

$$-3 \equiv 3454 \pmod{3457}$$

$$1445^2 \equiv -3 \pmod{3457}$$

$$\sqrt{3} \equiv \pm 1445 \pmod{3457}$$

$$2^{-1} \equiv 1729 \pmod{3457}$$

$$\text{왜냐하면 } (2 \cdot 1729 = 3458 \equiv 1)$$

따라서 $\alpha_{\pm} = \frac{1 \pm 1445}{2} \pmod{3457}$

$$\alpha_1 = \frac{1 + 1445}{2} \equiv 723 \pmod{3457}, \alpha_2 = \frac{1 - 1445}{2} \equiv 2735 \pmod{3457}$$

$\alpha_1 + \alpha_2 \equiv 1 \pmod{3457}, \alpha_2 = 1 - \alpha_1$ 이며 두 값 모두 $\alpha^2 - \alpha + 1 \equiv 0 \pmod{3457}$ 만족

$$Y^2 - Y + 1 = 0 \pmod{3457}$$

$$522,729 - 723 + 1 = 0 \pmod{3457}, 7,480,225 - 2,735 + 1 = 0 \pmod{3457}$$

```
// zetas: Montgomery-form twiddle factors
```

```
const int16_t zetas[192] = {
```

```
[0]=-147, [1]=-1033, [2]=-682, [3]=-248, [4]=-708, [5]=682,
```

```
#define NTRUPLUS_R -147 // R = 2^16 mod q
```

```
#define NTRUPLUS_RINV -682 // (R)^(-1) mod q
```

```
#define NTRUPLUS_RSQ 567 // (R^2) mod q
```

```
#define NTRUPLUS_QINV 12929 // (q)^(-1) mod (2^16)
```

```
[16]=1611, [17]=222, [18]=1164, [19]=-1346, [20]=1716, [21]=
```

```
#define NTRUPLUS_OMEGA -886 // (omega * R) mod q
```

```
[24]=-455, [25]=639, [26]=502, [27]=-655, [28]=-603, [29]=
```

```
#define NTRUPLUS_ZMINUSZ5INV -1665 // (z - z^5)^(-1) * R mod q
```

```
[32]=-1241, [33]=550, [34]=-44, [35]=39, [36]=-820, [37]=
```

```
// where z = zeta^((n/d)/6)
```

```
[40]=-348, [41]=937, [42]=895, [43]=387, [44]=-603, [45]=
```

```
#define NTRUPLUS_NINV -811 // (n/d)^(-1) * R mod q
```

```
[48]=1449, [49]=837, [50]=901, [51]=1637, [52]=569, [53]=
```

```
#define NTRUPLUS_MINV -1622 // 2 * (n/d)^(-1) * R mod q
```

```
[56]=50, [57]=-830, [58]=-625, [59]=4, [60]=176, [61]=-156, [62]=1257, [63]=-1507,
```

```
[64]=-380, [65]=-606, [66]=1293, [67]=661, [68]=1428, [69]=-1580, [70]=-565, [71]=-992,
```

```
[72]=548, [73]=-800, [74]=64, [75]=-371, [76]=961, [77]=641, [78]=87, [79]=630,
```

```
[80]=675, [81]=-834, [82]=205, [83]=54, [84]=-1081, [85]=1351, [86]=1413, [87]=-1331,
```

```
[88]=-1673, [89]=-1267, [90]=-1558, [91]=281, [92]=-1464, [93]=-588, [94]=1015, [95]=436,
```

```
[96]=223, [97]=1138, [98]=-1059, [99]=-194, [100]=-83, [101]=1655, [102]=559, [103]=-1674,
```

```
[104]=277, [105]=933, [106]=-1723, [107]=-457, [108]=-145, [109]=-1242, [110]=1640, [111]=-432,
```

```
[112]=-1583, [113]=696, [114]=774, [115]=1671, [116]=927, [117]=514, [118]=512, [119]=-489,
```

```
[120]=297, [121]=601, [122]=1473, [123]=1130, [124]=1322, [125]=871, [126]=760, [127]=-1212,
```

```
[128]=-312, [129]=-352, [130]=443, [131]=943, [132]=8, [133]=1250, [134]=-100, [135]=-1660,
```

```
[136]=-31, [137]=1206, [138]=-1341, [139]=-1247, [140]=444, [141]=235, [142]=1364, [143]=-1209,
```

```
[144]=361, [145]=230, [146]=673, [147]=582, [148]=1409, [149]=1501, [150]=1401, [151]=-251,
```

```
[152]=1022, [153]=-1063, [154]=1053, [155]=1188, [156]=417, [157]=-1391, [158]=-27, [159]=-1626,
```

```
[160]=1685, [161]=-315, [162]=1408, [163]=-1248, [164]=400, [165]=274, [166]=-1543, [167]=-32,
```

```
[168]=-1550, [169]=1531, [170]=-1367, [171]=-124, [172]=1458, [173]=1379, [174]=-940, [175]=-1681,
```

```
[176]=22, [177]=1709, [178]=-275, [179]=1108, [180]=354, [181]=-1728, [182]=-968, [183]=-858,
```

```
[184]=1221, [185]=-218, [186]=294, [187]=-732, [188]=-1095, [189]=892, [190]=1588, [191]=-779,
```

```
[192]=1221, [193]=-218, [194]=294, [195]=-732, [196]=-1095, [197]=892, [198]=1588, [199]=-779,
```

```
[200]=1221, [201]=-218, [202]=294, [203]=-732, [204]=-1095, [205]=892, [206]=1588, [207]=-779,
```

```
[208]=1221, [209]=-218, [210]=294, [211]=-732, [212]=-1095, [213]=892, [214]=1588, [215]=-779,
```

```
[216]=1221, [217]=-218, [218]=294, [219]=-732, [220]=-1095, [221]=892, [222]=1588, [223]=-779,
```

```
#define NTRUPLUS_OMEGA -886 // (omega * R) mod q
```

```
#define NTRUPLUS_ZMINUSZ5INV -1665 // (z - z^5)^(-1) * R mod q
```

```
// where z = zeta^((n/d)/6)
```

```
#define NTRUPLUS_NINV -811 // (n/d)^(-1) * R mod q
```

```
#define NTRUPLUS_MINV -1622 // 2 * (n/d)^(-1) * R mod q
```

$$\alpha = \text{zetas}[1] \cdot R^{-1} \bmod q$$

$$\alpha \equiv (-1033) \cdot (-682) \equiv 723 \pmod{3457}$$

$$\alpha_1 = \frac{1 + 1445}{2} \equiv 723 \pmod{3457}$$

NTRU+ M

- NTRU+에

$$R_q = Z_q[X]$$

- 다항식을 다

- 환 관계식에

- 해당 2차식의 근을 α_1, α_2 라 하면

$$a(X) = a_0(X) + X^{384} a_1(X), Y := X^{384}$$

$$R_q \cong \prod_{j=1}^2 Z_q[X] / (X^{384} - \alpha_j)$$

환의 관계식: $Y^2 - Y + 1 = 0$

- 동형의 좌표는 $X^{384} = \alpha_j$ 를 대입하는 형태

해당 2차식의 근을 α_1, α_2 라 두면 $a_1(X) \rightarrow a_0(X) + \alpha_j a_1(X)$

$$R_q \cong Z_q[X] / (X^{384} - \alpha_1) \times Z_q[X] / (X^{384} - \alpha_2)$$

따라서 일반적인 좌표 선택은 $\begin{cases} u_1 = a_0 + \alpha_1 a_1 \\ u_2 = a_0 + \alpha_2 a_1 \end{cases}$

```
int k = 1;
```

```
zeta1 = zetas[k++];
```

```
for (int i = 0; i < NTRUPLUS_N / 2; i++)
```

```
{
```

```
t1 = fqmul(a: zeta1, b: a[i + NTRUPLUS_N / 2]);
```

```
r[i + NTRUPLUS_N / 2] = a[i] + a[i + NTRUPLUS_N / 2] - t1;
```

```
r[i] = a[i] + t1;
```

```
}
```

< 384

NTRU+ N

• NTRU+에

$$R_q = Z_q[X]$$

```
t1 = α * a1;
```

```
r[i] = a0 + α a1; // 첫 branch
```

```
r[i+384] = a0 + a1 - α a1; // a0 + (1-α) a1
```

```
t1 = tqmul(a: zeta1, b: a[i + NTRUPLUS_N / 2]);
```

```
r[i + NTRUPLUS_N / 2] = a[i] + a[i + NTRUPLUS_N / 2] - t1;
```

```
r[i] = a[i] + t1;
```

하지만 코드에서는 한번의 곱셈만을 취함

즉 두 번째 좌표가 $a_0 + \alpha_2 a_1$ 가 아니라 $a_0 + (1 - \alpha) a_1$ 임

다항식의 근들의 관계: $Y^2 - Y + 1 = 0$ 의 두 근 $\alpha_1 + \alpha_2 = 1$, $(-(\alpha_1 + \alpha_2) = -1)$ 을 만족

따라서 $\alpha_2 = 1 - \alpha_1$ 이며 이를 정리하면 $a_0 + \alpha_2 a_1 = a_0 + (1 - \alpha_1) a_1$

해당 2차식의 근을 α_1, α_2 라 두면 $1(X) \rightarrow a_0(X) + \alpha_j a_1(X)$

$$R_q \cong Z_q[X]/(X^{384} - \alpha_1) \times Z_q[X]/(X^{384} - \alpha_2)$$

따라서 일반적인 좌표 선택은 $\begin{cases} u_1 = a_0 + \alpha_1 a_1 \\ u_2 = a_0 + \alpha_2 a_1 \end{cases}$

NTRU+ NTT 코드적 해석 (2단계 분해)

```
for (int start = 0; start < NTRUPLUS_N; start += 384)
{
    zeta1 = zetas[k++];
    zeta2 = zetas[k++];

    for (int i = start; i < start + 128; i++)
    {
        t1 = fqmul(a: zeta1, b: r[i + 128]);
        t2 = fqmul(a: zeta2, b: r[i + 256]);
        t3 = fqmul(a: NTRUPLUS_OMEGA, b: t1 - t2);

        r[i + 256] = r[i] - t1 - t3;
        r[i + 128] = r[i] - t2 + t3;
        r[i] = r[i] + t1 + t2;
    }
}
```

$t1 = \text{fqmul}(\text{zeta}1, r[i+128]);$ // $B' = \xi_1 \cdot B$
 $t2 = \text{fqmul}(\text{zeta}2, r[i+256]);$ // $C' = \xi_2 \cdot C$
 $t3 = \text{fqmul}(\text{OMEGA}, t1 - t2);$ // $t3 = \Omega \cdot (B' - C')$

$r[i] = A + B' + C';$ // $y_0 = A + B' + C'$
 $r[i+128] = A - C' + t3;$ // $y_1 = A - C' + \Omega \cdot (B' - C')$
 $r[i+256] = A - B' - t3;$ // $y_2 = A - B' - \Omega \cdot (B' - C')$

ξ 는 global root

Ω 는 local root (3차 원시 단위근)

$$\Omega^3 \equiv 1 \pmod{3457}, \Omega \neq 1, \Omega \equiv 2734 \equiv -723 \pmod{3457}$$
$$\Omega = \xi^{N/3}$$

$$\Omega^3 = \xi^N = 1$$

$$\Omega^3 = 1, \Omega \neq 1$$

$$\Omega^3 - 1 = (\Omega - 1)(\Omega^2 + \Omega + 1) = 0$$

여기서 $\Omega \neq 1$ 이므로 $\Omega^2 + \Omega + 1 = 0$

$$\Omega^2 + \Omega + 1 = 0 \rightarrow \Omega^2 = -(1 + \Omega)$$

```
// zetas: Montgomery-form twiddle factors
```

```
const int16_t zetas[192] = {
```

```
    [0]=-147, [1]=-1033, [2]=-682, [3]=-248, [4]=-708, [5]=682, [6]=1, [7]=-722,  
    [8]=-723, [9]=-257, [10]=-1124, [11]=-867, [12]=-256, [13]=1484, [14]=1262, [15]=-1590,
```

```
    t2 = fqmul(zeta2, r[i+256]); // C' =  $\xi_2 \cdot C$ 
```

```
    t3 = fqmul(OMEGA, t1 - t2); // t3 =  $\Omega \cdot (B' - C')$ 
```

```
for (int start = 0; start < NTRUPLUS_N; start += 384)
```

zetas[2]: $\alpha = 2735$ 의 3제곱근 β

zetas[2] = -682

$\beta = (-682) \cdot 2775 \pmod{3457} = 1886$

$1886^3 \pmod{3457} = 2735 = \alpha$

```
    r[i + 256] = r[i] - t1 - t3;
```

```
    r[i + 128] = r[i] - t2 - t3;  $\beta^3 = \alpha$ 
```

```
    r[i] = r[i] + t1 + t2;
```

```
}
```

```
}
```

```
    r[i] = A + B' + C'; //  $y_0 = A + B' + C'$ 
```

```
    r[i+128] = A - C' + t3; //  $y_1 = A - C' + \Omega \cdot (B' - C')$ 
```

```
    r[i+256] = A - B' - t3; //  $y_2 = A - B' - \Omega \cdot (B' - C')$ 
```

zetas[3] = -248

$\xi \triangleq$ global root

$\Omega \triangleq$ local root (3차 원시 단위근)
 $\beta^2 = (-248) \cdot 2775 \pmod{3457} = 3200$

$$1886^2 \pmod{3457} \stackrel{\Omega = \xi^{N/3}}{=} 3200 = \beta^2$$

$$\Omega^3 = \xi^N = 1$$

$$\Omega^3 = 1, \Omega \neq 1$$

$$\Omega^3 - 1 = (\Omega - 1)(\Omega^2 + \Omega + 1) = 0$$

여기서 $\Omega \neq 1$ 이므로 $\Omega^2 + \Omega + 1 = 0$

$$\Omega^2 + \Omega + 1 = 0 \rightarrow \Omega^2 = -(1 + \Omega)$$

NTRU+ NTT 코드적 해석 (2단계 분해)

$$y_0 = A + B' + C'$$

$$\begin{aligned} y_1 &= A - C' + \Omega \cdot (B' - C') \\ y_1 &= A + \Omega \cdot B' + (-(1 + \Omega)) \cdot C' \end{aligned}$$

$$\begin{aligned} y_2 &= A - B' - \Omega \cdot (B' - C') \\ y_2 &= A + (-(1 + \Omega)) \cdot B' + \Omega \cdot C' \end{aligned}$$

$$\Omega^3 = 1, \Omega \neq 1$$

따라서

$$\Omega^0 = 1, \Omega^1 = \Omega, \Omega^2 = \Omega^2, \Omega^3 = 1, \Omega^4 = \Omega$$

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & \Omega & -(1 + \Omega) \\ 1 & -(1 + \Omega) & \Omega \end{pmatrix} \begin{pmatrix} A \\ B' \\ C' \end{pmatrix}$$

$$\Omega^2 + \Omega + 1 = 0 \rightarrow \Omega^2 = -(1 + \Omega)$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & \Omega & \Omega^2 \\ 1 & \Omega^2 & \Omega \end{pmatrix}$$

$$f(x) = A + B'x + C'x^2$$

해당 다항식을 $x = 1, \Omega, \Omega^2$ 에서 평가하면

1. $x = 1$ $f(1) + A + B' + C' = y_0$
2. $x = \Omega$ $f(\Omega) + A + B'\Omega + C'\Omega^2 = y_1$
3. $x = \Omega^2$ $f(\Omega^2) + A + B'\Omega^2 + C'(\Omega^2)^2 = y_2$

NTRU+ NTT 코드적 해석 (3단계 분해)

- 남은 각 factor는 $Z_q[X]/(X^{128} - \beta)$ 형식을 가짐
- $128 = 2^7$ 이므로 표준적인 2-way 분해를 반복
- 일반적으로 $X^{2m} - \gamma = (X^m - \delta)(X^m + \delta)$
 - $\delta^2 = \gamma$ 에 대응하는 동형이 있고 NTT의 butterfly는 그 좌표 변환을 계산

```
for (int step = 64; step >= 4; step >>= 1)
{
    for (int start = 0; start < NTRUPLUS_N; start += (step << 1))
    {
        zeta1 = zetas[k++];

        for (int i = start; i < start + step; i++)
        {
            t1 = fqmul(a: zeta1, b: r[i + step]);

            r[i + step] = barrett_reduce(a: r[i] - t1);
            r[i] = barrett_reduce(a: r[i] + t1);
        }
    }
}
```

zeta 값 결정

$$(X^{128})^3 = \alpha \rightarrow X^{128} = \rho$$

$$X^{64} = \sqrt{\rho}$$

$$X^{32} = \sqrt[4]{\rho}$$

$$X^{16} = \sqrt[8]{\rho}$$

$$X^8 = \sqrt[16]{\rho}$$

$$X^4 = \sqrt[32]{\rho}$$

$$X^{384} = \alpha$$

↓ (radix-3)

$$X^{128} \in \{ \beta, \beta\Omega, \beta\Omega^2 \}$$

↓ (radix-2)

$$X^{64} \in \{ \sqrt{\beta}, \sqrt{(\beta\Omega)}, \sqrt{(\beta\Omega^2)} \}$$

↓

$$X^{32} \in \{ \dots \}$$

↓

$$X^4$$

SMAUG-T

- 격자기반 KEM

- 기반 문제

- Module-LWE (MLWE): 공개키 보안
- Module-LWR (MLWR): 암호문 보안

- 특징

- Kyber 대비 더 작은 암호문 / 공개키
→ 암호문 크기 최소화 (통신 비용 절감), IoT 임베디드 환경 적합
- 낮은 Decryption Failure Probability (DFP)와 고속 Encapsulation

Set	Sec	(n, k)	q	p	p'	Ciphertext	PK
TiMER	1	(256,2)	1024	256	8	608 B	672 B
T128	1	(256,2)	1024	256	32	672 B	672 B
T192	3	(256,3)	2048	512	16	992 B	1088 B
T256	5	(256,4)	2048	512	128	1376 B	1440 B

SMAUG-T: 수학적 구조

- **다항식 환**

- $R = \mathbb{Z}[x] / (x^n + 1), n = 256$

- **모듈 격자 구조**

- MLWE: $A \in R_q^{k \times k}, b = -A^T s + e$

- MLWR: $[\frac{p}{q} Ar]$

- **Module 구조 채택 이유**

- RLWE 대비 보안-성능 파라미터 조절 자유도 증가
 - Kyber/Saber와 유사한 분석 프레임워크

공개키 (MLWE) & 암호문 (MLWR) 구조

• 공개키

- $A \leftarrow R_q^{k \times k}$ (XOF 기반 생성)
- s : 고정 Hamming weight ternary secret
- e : 근사 이산 가우시안
- $b = -A^T s + e$

• 공개키 저장

- b 와 A 의 seed

• 암호문:

- $c = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \lfloor \frac{p}{q} \begin{bmatrix} A \\ b^T \end{bmatrix} r \rfloor + \frac{p}{t} \begin{bmatrix} 0 \\ \mu \end{bmatrix}$
- r : 비고정 Hamming weight sparse ternary
- MLWR 기반 \rightarrow ciphertext 압축 가능
- Public key는 압축하지 않음

NTT 사용도 가능하지만 특정한 변환없이
SMAUG-T는 기본적으로 toomcook 적용이 용이

```
reference_implementation/
├── src/                                # 소스 코드 파일
│   ├── toomcook.c                    # Toom-Cook 4-way 및 Karatsuba 곱셈 알고리즘
│   ├── poly.c                        # 다항식 연산 (벡터/행렬 곱셈)
│   ├── hwt.c                         # Hamming Weight Sampling
│   ├── dg.c                         # Discrete Gaussian Sampling
│   ├── cbd.c                        # Centered Binomial Distribution Sampling
│   ├── key.c                        # 키 생성 (공개키/비밀키)
│   ├── indcpa.c                    # IND-CPA 보안 공개키 암호화
│   ├── kem.c                       # Key Encapsulation Mechanism (KEM)
│   ├── ciphertext.c                # 암호문 계산 (c1, c2)
│   ├── pack.c                      # 키/암호문 패킹
│   ├── packring.c                  # 링 원소 패킹
│   ├── hash.c                      # 해시 함수
│   ├── fips202.c                   # SHA-3 (SHAKE256, SHAKE128)
│   ├── verify.c                    # 검증 함수
│   └── randombytes.c
├── include/                          # 헤더 파일
│   ├── params.h                    # 파라미터 정의 (모드별)
│   ├── poly.h                      # 다항식 타입 및 함수 선언
│   ├── toomcook.h                  # 다항식 곱셈 함수 선언
│   ├── hwt.h                      # HWT 샘플링 함수 선언
│   ├── dg.h                       # Discrete Gaussian 샘플링 함수 선언
│   ├── cbd.h                      # CBD 샘플링 함수 선언
│   ├── key.h                      # 키 생성 함수 선언
│   ├── indcpa.h                   # IND-CPA 암호화 함수 선언
│   ├── kem.h                      # KEM 함수 선언
│   ├── ciphertext.h               # 암호문 구조체 및 함수 선언
│   ├── pack.h                     # 패킹 함수 선언
│   ├── packring.h                 # 링 원소 패킹 함수 선언
│   ├── hash.h                     # 해시 함수 선언
│   ├── fips202.h                  # SHA-3 함수 선언
│   ├── verify.h                   # 검증 함수 선언
│   ├── common.h                   # 공통 매크로 정의
│   └── config.h                   # 빌드 설정
├── CMakeLists.txt                  # CMake 빌드 설정
├── benchmark/                      # 벤치마크 코드
└── test/                          # Known Answer Test, 테스트 코드

poly_mul_acc(a[256], b[256]) // 최상위 API
├─ toom_cook_4way(a, b) // 256을 4개로 쪼개서 평가점 7개
├─ karatsuba_simple(64,64) x 7회 // 각 평가점 곱셈은 64차
└─ 내부는 16차 단위 직접곱 (schoolbook) 다수
```

SMAUG-T KEM

```
int crypto_kem_enc_internal(uint8_t *ctxt, uint8_t *ss, const uint8_t *pk,
                           const uint8_t *mu) {
    int ret = 0;
    uint8_t seed_r[SMAUGT_DELTA_BYTES + SMAUGT_CRYPTO_BYTES] = {0};
    hash_h(h: seed_r, in: pk, inlen: SMAUGT_PUBLICKEY_BYTES);
    hash_g(out: seed_r, outlen: SMAUGT_DELTA_BYTES + SMAUGT_CRYPTO_BYTES, in1: mu,
          inlen1: SMAUGT_MSG_BYTES, in2: seed_r, inlen2: SHA3_256_HashSize);

    memset(s: ss, c: 0, n: SMAUGT_CRYPTO_BYTES);
    indcpa_enc(ctxt, pk, mu, seed: seed_r);
    cmov(r: ss, x: seed_r + SMAUGT_DELTA_BYTES, len: SMAUGT_CRYPTO_BYTES, b: 1);
    return ret;
}
```

- hash_h: $H(pk)$ 를 seed_r 앞부분에 저장
- hash_g: 랜덤 메시지 (mu)와 $H(pk)$ 값을 해시하여 저장

SMAUG-T KEM

```
int crypto_kem_enc_internal(uint8_t *ctxt, uint8_t *ss, const uint8_t *pk,
                           const uint8_t *mu) {
    int ret = 0;
    uint8_t seed_r[SMAUGT_DELTA_BYTES + SMAUGT_CRYPT0_BYTES] = {0};
    hash_h(h: seed_r, in: pk, inlen: SMAUGT_PUBLICKEY_BYTES);
    hash_g(out: seed_r, outlen: SMAUGT_DELTA_BYTES + SMAUGT_CRYPT0_BYTES, in1: mu,
          inlen1: SMAUGT_MSG_BYTES, in2: seed_r, inlen2: SHA3_256_HashSize);

    memset(s: ss, c: 0, n: SMAUGT_CRYPT0_BYTES);
    indcpa_enc(ctxt, pk, mu, seed: seed_r);
    cmov(r: ss, x: seed_r + SMAUGT_DELTA_BYTES, len: SMAUGT_CRYPT0_BYTES, b: 1);
    return ret;
}
```

- CPA-PKE 암호화로 ciphertext (c1, c2) 생성

SMAUG-T KEM (공개키 pk 언팩, ephemeral 비밀 벡터 r 생성)

```
void indcpa_enc(uint8_t ctxt[SMAUGT_CIPHERTEXT_BYTES],
               const uint8_t pk[SMAUGT_PUBLICKEY_BYTES],
               const uint8_t mu[SMAUGT_MSG_BYTES],
               const uint8_t seed[SMAUGT_DELTA_BYTES]) {
```

```
    uint8_t seed_r[SMAUGT_DELTA_BYTES] = {0};
    public_key pk_tmp;
    unpack_enck(pk: &pk_tmp, input: pk);

    // Compute a vector r = hwt(delta, H'(pk))
    polyvec r;
    memset(s: &r, c: 0, n: sizeof(polyvec));

    if (seed == NULL)
        randombytes(x: seed_r, xlen: SMAUGT_DELTA_BYTES);
    else
        cmov(r: seed_r, x: seed, len: SMAUGT_DELTA_BYTES, b: 1);
    expand_r(r: &r, seed: seed_r);
```

```
    // Compute c1(x), c2(x)
    ciphertext ctxt_tmp;
    memset(s: &ctxt_tmp, c: 0, n: sizeof(ciphertext));
    computeC1(c1: &(ctxt_tmp.c1), A: pk_tmp.A, r: &r);
    computeC2(c2: &(ctxt_tmp.c2), mu, b: &pk_tmp.b, r: &r);

    pack_ct(output: ctxt, ctxt: &ctxt_tmp);
}
```

- pk_tmp: 행렬/벡터 형태의 PKE 공개키 구성요소가 들어감

- seed_r 을 입력으로 shake를 생성 후 sp_cbd()로 희소 분포의 다항식 벡터 생성

```
void expand_r(polyvec *r, const uint8_t *seed) {
    unsigned int i;
    uint8_t buf[SMAUGT_CBDSEED_BYTES] = {0};

    for (i = 0; i < SMAUGT_K; ++i) {
        uint8_t extseed[SMAUGT_DELTA_BYTES + 1];
        memcpy(dest: extseed, src: seed, n: SMAUGT_DELTA_BYTES);
        extseed[SMAUGT_DELTA_BYTES] = i;

        shake256(out: buf, outlen: SMAUGT_CBDSEED_BYTES, in: extseed, inlen: SMAUGT_DELTA_BYTES + 1);
        sp_cbd(r: &r->vec[i], buf);
    }
}
```

SMAUG-T KEM (ciphertext의 첫 성분 c1 계산)

```
void indcpa_enc(uint8_t ctxt[SMAUGT_CIPHERTEXT_BYTES],
               const uint8_t pk[SMAUGT_PUBLICKEY_BYTES],
               const uint8_t mu[SMAUGT_MSG_BYTES],
               const uint8_t seed[SMAUGT_DELTA_BYTES]) {

    uint8_t seed_r[SMAUGT_DELTA_BYTES] = {0};
    public_key pk_tmp;
    unpack_enck(pk: &pk_tmp, input: pk);

    // Compute a vector r = hwt(delta, H'(pk))
    polyvec r;
    memset(s: &r, c: 0, n: sizeof(polyvec));

    if (seed == NULL)
        randombytes(x: seed_r, xlen: SMAUGT_DELTA_BYTES);
    else
        cmov(r: seed_r, x: seed, len: SMAUGT_DELTA_BYTES, b: 1);
    expand_r(r: &r, seed: seed_r);

    // Compute c1(x), c2(x)
    ciphertext ctxt_tmp;
    memset(s: &ctxt_tmp, c: 0, n: sizeof(ciphertext));
    computeC1(c1: &(ctxt_tmp.c1), A: pk_tmp.A, r: &r);
    computeC2(c2: &(ctxt_tmp.c2), mu, b: &pk_tmp.b, r: &r);

    pack_ct(output: ctxt, ctxt: &ctxt_tmp);
}
```

- $A \cdot r$ 을 수행하고 round1 함수를 통해 계수들을 q 에서 p 로 라운딩

```
void computeC1(polyvec *c1, const polyvec A[SMAUGT_K], const polyvec *r) {
    // c1 = A * r
    matrix_vec_mult_add(r: c1, a: A, b: r);

    // Rounding q to p
    round1(a: c1);
}
```

SMAUG-T KEM (ciphertext의 첫 성분 c1 계산)

```
void indcpa_enc(uint8_t ctxt[SMAUGT_CIPHERTEXT_BYTES],
               const uint8_t pk[SMAUGT_PUBLICKEY_BYTES],
               const uint8_t mu[SMAUGT_MSG_BYTES],
               const uint8_t seed[SMAUGT_DELTA_BYTES]) {

    uint8_t seed_r[SMAUGT_DELTA_BYTES] = {0};
    public_key pk_tmp;
    unpack_enck(pk: &pk_tmp, input: pk);

    // Compute a vector r = hwt(delta, H'(pk))
    polyvec r;
    memset(s: &r, c: 0, n: sizeof(polyvec));

    if (seed == NULL)
        randombytes(x: seed_r, xlen: SMAUGT_DELTA_BYTES);
    else
        cmov(r: seed_r, x: seed, len: SMAUGT_DELTA_BYTES, b: 1);
    expand_r(r: &r, seed: seed_r);

    // Compute c1(x), c2(x)
    ciphertext ctxt_tmp;
    memset(s: &ctxt_tmp, c: 0, n: sizeof(ciphertext));
    computeC1(c1: &(ctxt_tmp.c1), A: pk_tmp.A, r: &r);
    computeC2(c2: &(ctxt_tmp.c2), mu, b: &pk_tmp.b, r: &r);

    pack_ct(output: ctxt, ctxt: &ctxt_tmp);
}
```

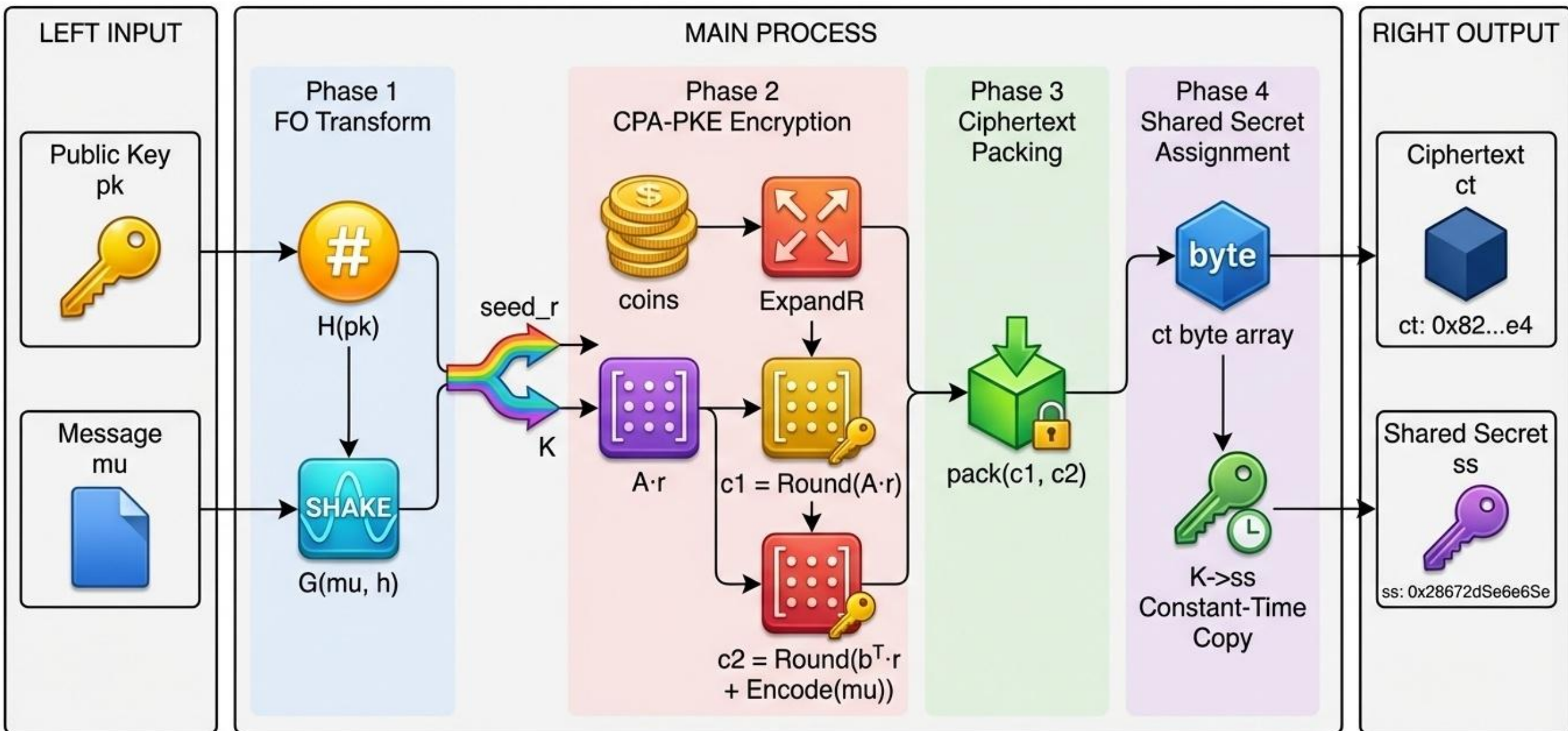
- $q/2 \cdot \text{delta}$ 형식으로 인코딩
- $c2 = \frac{q}{2} \cdot \text{delta} + (b \cdot r)$ 수행
- 최종적으로 q 에서 p' 으로 라운딩

```
void computeC2(poly *c2, const uint8_t delta[SMAUGT_MSG_BYTES],
               const polyvec *b, const polyvec *r) {
    #if SMAUGT_MODE == SMAUGT_MODET
        // c2 = q/2 * delta
        d2_ecd(c2, delta); // EDIT TIMER
    #else
        unsigned int i, j;
        // c2 = q/2 * delta
        for (i = 0; i < SMAUGT_MSG_BYTES; ++i) {
            for (j = 0; j < sizeof(uint8_t) * 8; ++j) {
                c2->coeffs[8 * i + j] =
                    (uint16_t)((delta[i] >> j) << SMAUGT_MODULUS_16_LOG_T);
            }
        }
    #endif

    // c2 = q/2 * delta + (b * r)
    vec_vec_mult_add(r: c2, a: b, b: r, mod: SMAUGT_MODULUS_16_LOG_Q);

    // Rounding q to p'
    round2(a: c2);
}
```

SMAUG-T KEM Encapsulation Complete Process Overview



H: Hash Function, G: Key Derivation Function, Round: Rounding Function, Encode: Message Encoding, K: Intermediate Key, $c1$, $c2$: Ciphertext Components, pk : Public Key, mu : Message, r : Random Coins, ct : Final Ciphertext, ss : Shared Secret

NTT friendly Ring

- 대부분 PQC에서 사용하는 환은 다음과 같음
 - $R_q = \mathbb{Z}_q[x] / (x^n \pm 1)$
 - 해당 환이 NTT-friendly 하려면 핵심 조건이 필요
 - $\omega^n \equiv 1 \pmod{q}$ 이고 $\omega^k \not\equiv 1 \pmod{q}$ ($0 < k < n$) 인 primitive n-root of unity가 존재
 - 즉 $n | (q - 1)$ 을 만족해야 함
 - $q \equiv 1 \pmod{n}$

NTT friendly Ring

파라미터	ML-KEM	HAETAE	NTRU+	SMAUG-T
n	256	256	768	256
q	3329	64513	3457	1024
$q \equiv 1 \pmod{n}$	$3329 \bmod 256 = 1$	$64513 \bmod 256 = 1$	3-radix여서 다름	$1024 \bmod 256 = 0$
NTT-friendly ring 여부	○	○	○	X

- SMAUG-T의 경우 NTT friendly-ring 조건을 만족하지 못함
- 따라서 본 설명에서는 Toom-cook과 같은 차선택을 기준으로 설명
- 하지만 SMAUG-T와 같은 NTT unfriendly-ring에서도 NTT 구현 가능
 - NTT 조건을 만족하는 현재 q 보다 큰 값으로 확장 후 NTT 적용하고 다시 원래 q 로 전환
→ 마치 물건을 그대로 택배를 보낼수 없을때 택배상자에 완충제 넣어서 보내기

택배 unfriendly 두유팜키지



택배 friendly 두유팜키지



SMAUG-T ToomCook 4-way 수학적 해석

- **SMAUG-T의 256차 다항식 곱을 64차 블록 다항식의 4-way Toom-Cook으로 바꿔서 평가 7번의 64차 곱만으로 복원하는 구조**
- 1) 4-way 분할: “블록 변수”로 재정의
 - 원래 다항식 (길이 256 계수) $a(x), b(x)$ 를 64개씩 끊어 4개의 블록 다항식으로 분할
 - $n = 256, m = \frac{n}{4} = 64$
 - 블록 다항식 (각각 차수 < 64):
 - $a(x) = a_0(x) + a_1(x)x^m + a_2(x)x^{2m} + a_3(x)x^{3m}$
 - $b(x) = b_0(x) + b_1(x)x^m + b_2(x)x^{2m} + b_3(x)x^{3m}$
 - 여기서 Toom-Cook 관점에서는 $t = x^m$ 를 새로운 변수처럼 보고
 - $\tilde{a}(t) = a_0 + a_1t + a_2t^2 + a_3t^3, \tilde{b}(t) = b_0 + b_1t + b_2t^2 + b_3t^3$ (여기서 a_i, b_i 는 64차 미만의 다항식)
 - 결과는 다음을 계산
 - $\tilde{c}(t) = \tilde{a}(t)\tilde{b}(t) = c_0 + c_1t + c_2t^2 + c_3t^3 + c_4t^4 + c_5t^5 + c_6t^6$
 - 최종 결과는 $c(x) = c_0(x) + c_1(x)x^m + \dots + c_6(x)x^{6m}$

SMAUG-T ToomCook 4-way 수학적 해석

• 2) Evaluation

- 차수 6 다항식을 결정하려면 서로 다른 7개 점에서의 값이 필요

- $\tilde{c}(t) = \tilde{a}(t)\tilde{b}(t) = c_0 + c_1t + c_2t^2 + c_3t^3 + c_4t^4 + c_5t^5 + c_6t^6$

- c_0, \dots, c_6 는 미지의 블록 다항식 (각각 차수 $< m$)

- $\tilde{a}(t) = a_0 + a_1t + a_2t^2 + a_3t^3, \tilde{b}(t) = b_0 + b_1t + b_2t^2 + b_3t^3$

- 서로 다른 스칼라 $\alpha_1, \alpha_2, \dots, \alpha_7$ 를 선택 (e.g., 0, 1, -1, 2, -2, ∞ , β)

- 각 점에서 평가

- $A_i = \tilde{a}(\alpha_i)$

- $B_i = \tilde{b}(\alpha_i)$

- $(i = 1, \dots, 7)$

- $W_i = A_i \cdot B_i = \tilde{c}(\alpha_i)$

- W_i 는 차수 $< 2m$ 인 다항식

SMAUG-T ToomCook 4-way 수학적 해석

- 3) 평가식의 선형 구조

- 각 평가값은 다음 선형 결합: $W_i = C_0 + C_1\alpha_i + C_2\alpha_i^2 + \dots + C_6\alpha_i^6$

- $W_i = A_i \cdot B_i = \tilde{c}(\alpha_i)$

- $\tilde{c}(t) = \tilde{a}(t)\tilde{b}(t) = c_0 + c_1t + c_2t^2 + c_3t^3 + c_4t^4 + c_5t^5 + c_6t^6$

- $\tilde{a}(t) = a_0 + a_1t + a_2t^2 + a_3t^3, \tilde{b}(t) = b_0 + b_1t + b_2t^2 + b_3t^3$

- 이를 모든 i 에 대해 쓰면

$$\begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^6 \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^6 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \alpha_7 & \alpha_7^2 & \dots & \alpha_7^6 \end{bmatrix} \begin{bmatrix} C_0 \\ C_1 \\ \vdots \\ C_6 \end{bmatrix} = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_7 \end{bmatrix}$$

- 미지수: C_0, \dots, C_6

- 이미 계산된 값: W_1, \dots, W_7

SMAUG-T ToomCook 4-way 수학적 해석

- 4) Interpolation (보간)

- $$\begin{bmatrix} C_0 \\ \vdots \\ C_6 \end{bmatrix} = V^{-1} \begin{bmatrix} W_1 \\ \vdots \\ W_7 \end{bmatrix}$$

- 즉 $C_k = \sum_{i=1}^7 \lambda_{k,i} W_i$

- 어떤 상수 $\lambda_{k,i}$ 들의 선형 결합으로 각 계수가 복원

- 실제 구현에서는 이 선형 결합을 정수 연산으로 미리 전개

예시 Toom-3

SMUG-T ToomCook 4-way 수학적 해석

$$A(x) = a_0 + a_1x + a_1x^2 \quad B(x) = b_0 + b_1x + b_1x^2$$

$$C(x) = A(x)B(x)$$

$$C(x) = \begin{bmatrix} c_0 \\ \vdots \\ c_6 \end{bmatrix} = V^{-1} \begin{bmatrix} W_1 \\ \vdots \\ W_7 \end{bmatrix}$$

- 즉 $C_k = \sum_{i=1}^7 \lambda_{k,i} W_i$

- 어떤 상수 $\lambda_{k,i}$ 들의 선형 결합으로 각 계수가 복원

Evaluation 단계

- 실제 구현에서는 이 선형 결합을 정수 연산으로 미리 전개

- Toom-3에서는 보통 다음 점들을 사용 $x = 0, 1, -1, 2, \infty$
- 각 점에서 $W_i = C(x_i)$ 를 계산

Evaluation 단계

- Toom-3에서는 보통 다음 점들을 사용 $x = 0, 1, -1, 2, \infty$
- 각 점에서 $W_i = C(x_i)$ 를 계산

예시) Interpolation (보간)

- $W_0 = C(0) = c_0$
- $W_1 = C(1) = c_0 + c_1 + c_2 + c_3 + c_4$
- $W_{-1} = C(-1) = c_0 - c_1 + c_2 - c_3 + c_4$
- $W_2 = C(2) = c_0 + 2c_1 + 4c_2 + 8c_3 + 16c_4$
- $W_\infty = c_4$

- 즉 $C_k = \sum_{i=1}^7 \lambda_{k,i} W_i$

- 어떤 상수 $\lambda_{k,i}$ 들의 선형 결합으로 각 계수가 복원

Interpolation 단계

• Step1

- $c_0 = W_0$
- $c_4 = W_\infty$

Interpolation 단계 ToomCook 4-way 수학적 해석

• Step1

- $c_0 = W_0$
- $c_4 = W_\infty$

• Step2

- $W_1 + W_{-1} = 2(c_0 + c_2 + c_4)$
- $c_2 = \frac{W_1 + W_{-1}}{2} - c_0 - c_4$

- 즉 $C_k = \sum_{i=1}^7 \lambda_{k,i} W_i$

- 어떤 상수 $\lambda_{k,i}$ 들의 선형 결합으로 각 계수가 복원

• Step3 실제 구현에서는 이 선형 결합을 정수 연산으로 미리 전개

- $W_1 - W_{-1} = 2(c_1 + c_3)$
- $c_1 + c_3 = \frac{W_1 - W_{-1}}{2}$

• Step3

$$W_1 - W_{-1} = 2(c_1 + c_3)$$

$$c_1 + c_3 = \frac{W_1 - W_{-1}}{2} \quad (\text{보간})$$

• Step4

$$W_2 = c_0 + 2c_1 + 4c_2 + 8c_3 + 16c_4$$

• 이미 c_0, c_2, c_4 를 알고 있으므로

$$W_2 - c_0 - 4c_2 - 16c_4 = 2c_1 + 8c_3$$

$$\begin{cases} c_1 + c_3 = A \\ 2c_1 + 8c_3 = B \end{cases}$$

$$c_3 = \frac{B - 2A}{6}$$

$$c_1 = A - c_3$$

$$\text{• 즉 } C_k = \sum_{i=1}^7 \lambda_{k,i} W_i$$

• 어떤 상수 $\lambda_{k,i}$ 들의 선형 결합으로 각 계수가 복원

• 실제 구현에서는 이 선형 결합을 정수 연산으로 미리 전개

SMAUG-T ToomCook 4-way 수학적 해석

- 5) Recomposition (원래 변수로 복원)

- 마지막으로 $t = x^m$ 을 되돌리면

- $C(x) = C_0(x) + C_1(x)x^m + C_2(x)x^{2m} + C_3(x)x^{3m} + C_4(x)x^{4m} + C_5(x)x^{5m} + C_6(x)x^{6m}$

SMAUG-T ToomCook 4-way 코드 해석

- 4way로
입력값을 분해

```
static void toom_cook_4way(const uint16_t *a1, const uint16_t *b1,
                           uint16_t *result) {
    uint16_t inv3 = 43691, inv9 = 36409, inv15 = 61167;

    uint16_t aw1[N_SB], aw2[N_SB], aw3[N_SB], aw4[N_SB], aw5[N_SB], aw6[N_SB],
              aw7[N_SB];
    uint16_t bw1[N_SB], bw2[N_SB], bw3[N_SB], bw4[N_SB], bw5[N_SB], bw6[N_SB],
              bw7[N_SB];
    uint16_t w1[N_SB_RES] = {0}, w2[N_SB_RES] = {0}, w3[N_SB_RES] = {0},
              w4[N_SB_RES] = {0}, w5[N_SB_RES] = {0}, w6[N_SB_RES] = {0},
              w7[N_SB_RES] = {0};
    uint16_t r0, r1, r2, r3, r4, r5, r6, r7;
    uint16_t *A0, *A1, *A2, *A3, *B0, *B1, *B2, *B3;

    A0 = (uint16_t *)a1;
    A1 = (uint16_t *)&a1[N_SB];
    A2 = (uint16_t *)&a1[2 * N_SB];
    A3 = (uint16_t *)&a1[3 * N_SB];
    B0 = (uint16_t *)b1;
    B1 = (uint16_t *)&b1[N_SB];
    B2 = (uint16_t *)&b1[2 * N_SB];
    B3 = (uint16_t *)&b1[3 * N_SB];
```

SMAUG-T ToomCook 4-way 코드 해석

// EVALUATION

```
for (j = 0; j < N_SB; ++j) {
```

```
    r0 = A0[j];
```

```
    r1 = A1[j];
```

```
    r2 = A2[j];
```

```
    r3 = A3[j];
```

```
    r4 = r0 + r2;
```

```
    r5 = r1 + r3;
```

```
    r6 = r4 + r5;
```

```
    r7 = r4 - r5;
```

```
    aw3[j] = r6;
```

```
    aw4[j] = r7;
```

```
    r4 = ((r0 << 2) + r2) << 1;
```

```
    r5 = (r1 << 2) + r3;
```

```
    r6 = r4 + r5;
```

```
    r7 = r4 - r5;
```

```
    aw5[j] = r6;
```

```
    aw6[j] = r7;
```

```
    r4 = (r3 << 3) + (r2 << 2) + (r1 << 1) + r0;
```

```
    aw2[j] = r4;
```

```
    aw7[j] = r0;
```

```
    aw1[j] = r3;
```

```
}
```

$$A(1) = r_0 + r_1 + r_2 + r_3$$

$$A(-1) = r_0 - r_1 + r_2 - r_3$$

$$8A\left(\frac{1}{2}\right) = 8r_0 + 4r_1 + 2r_2 + r_3$$

$$8A\left(-\frac{1}{2}\right) = 8r_0 - 4r_1 + 2r_2 - r_3$$

$$A(2) = r_0 + 2r_1 + 4r_2 + 8r_3$$

$$A(0) = r_0 \quad A(\infty) = r_3$$

// MULTIPLICATION

```
karatsuba_simple(a_1: aw1, b_1: bw1, result_final: w1);
```

```
karatsuba_simple(a_1: aw2, b_1: bw2, result_final: w2);
```

```
karatsuba_simple(a_1: aw3, b_1: bw3, result_final: w3);
```

```
karatsuba_simple(a_1: aw4, b_1: bw4, result_final: w4);
```

```
karatsuba_simple(a_1: aw5, b_1: bw5, result_final: w5);
```

```
karatsuba_simple(a_1: aw6, b_1: bw6, result_final: w6);
```

```
karatsuba_simple(a_1: aw7, b_1: bw7, result_final: w7);
```

SMAUG-T ToomCook 4-way 코드 해석

$r0 = W(\infty) = c6$

$r6 = W(0) = c0$

$r2 = W(1), r3 = W(-1)$

$r4 = 64W(1/2), r5 = 64W(-1/2)$

$r1 = W(2)$

$r5 = 64(W(-1/2) - W(1/2)) = -64c1 - 16c3 - 4c5 \rightarrow$ 홀수항 조합

$r4 = 32c2 + 8c4 = 8(4c2 + c4)$

```
// INTERPOLATION
for (i = 0; i < N_SB_RES; ++i) {
    r0 = w1[i];
    r1 = w2[i];
    r2 = w3[i];
    r3 = w4[i];
    r4 = w5[i];
    r5 = w6[i];
    r6 = w7[i];
```

$r1 = W(2) + 64W(1/2)$

$r1 = r1 + r4;$

$r5 = r5 - r4;$

$r3 = ((r3 - r2) >> 1); r3 = (W(-1) - W(1))/2 = -(c1 + c3 + c5) \rightarrow$ 홀수항 조합(부호 포함)

$r4 = r4 - r0;$

$r4 = r4 - (r6 << 6);$

$r4 = (r4 << 1) + r5;$

$r2 = r2 + r3; r2 = W(1) + r3 = (W(1) + W(-1))/2 = c0 + c2 + c4 + c6 \rightarrow$ 짝수항 합

$r1 = r1 - (r2 << 6) - r2;$

$r2 = r2 - r6;$

$r2 = r2 - r0; r2 = c2 + c4$

$r1 = r1 + 45 * r2;$

$r4 = c2$ $r4 = (uint16_t)((r4 - (r2 << 3)) * (uint32_t)inv3) >> 3);$

$r5 = r5 + r1; r5 = c1$

$r1 = (uint16_t)((r1 + (r3 << 4)) * (uint32_t)inv9) >> 1);$

$r3 = -(r3 + r1); r3 = c3$

$r5 = (uint16_t)((30 * r1 - r5) * (uint32_t)inv15) >> 2);$

$r2 = r2 - r4; r2 = c4$

$r1 = r1 - r5; r1 = c5$

$C[i] += r6;$

$C[i + 64] += r5;$

$C[i + 128] += r4;$

$C[i + 192] += r3;$

$C[i + 256] += r2;$

$C[i + 320] += r1;$

$C[i + 384] += r0;$

}

SMAUG-T Karatsuba 수학적 해석

- 두 다항식 $a(x), b(x)$ (차수 $< n$)를 반으로 나누면
 - $a(x) = a_0(x) + a_1(x)x^{n/2}$
 - $b(x) = b_0(x) + b_1(x)x^{n/2}$
- 곱은
 - $a(x)b(x) = a_0b_0 + (a_0b_1 + a_1b_0)x^{n/2} + a_1b_1x^n$
- 여기서 핵심 트릭
 - $(a_0 + a_1)(b_0 + b_1) = a_0b_0 + a_1b_1 + (a_0b_1 + a_1b_0)$
- 따라서 교차항은 다음과 같이 계산
 - $a_0b_1 + a_1b_0 = (a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1$
- 곱셈수가 최종적으로 4번에서 3번으로 감소

SMAUG-T Karatsuba 실제 구현 관점

- Karatsuba_simple()은 64차 다항식을 입력으로 받음
 - 이를 16차 블록 4개로 나눔
 - $a(x) = a_0 + a_1x^{16} + a_2x^{32} + a_3x^{48}$
 - $b(x) = b_0 + b_1x^{16} + b_2x^{32} + b_3x^{48}$
- 즉 블록 변수 $t = x^{16}$ 를 쓰면
 - $\tilde{a}(t) = a_0 + a_1t + a_2t^2 + a_3t^3$
- 여기서 목표는
 - $\tilde{c}(t) = \tilde{a}(t)\tilde{b}(t) = \sum_{k=0}^6 c_k t^k$

SMAUG-T Karatsuba 실제 구현 관점

- 계산 상세: $\tilde{c}(t) = \tilde{a}(t)\tilde{b}(t) = \sum_{k=0}^6 c_k t^k$
 - 직접 계산되는 항
 - $c_0 = a_0 b_0$
 - $c_2 = a_1 b_1$
 - $c_4 = a_2 b_2$
 - $c_6 = a_3 b_3$
 - 교차항
 - $d_{01} = (a_0 + a_1)(b_0 + b_1)$
 - $c_1 = d_{01} - a_0 b_0 - a_1 b_1$
 - $d_{23} = (a_2 + a_3)(b_2 + b_3)$
 - $c_5 = d_{23} - a_2 b_2 - a_3 b_3$
 - $c_3 = a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0$
 - $d_{0123} = (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3)$
 - $d_{0123} = (a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3) + (a_0 b_1 + a_1 b_0) + (a_2 b_3 + a_3 b_2) + (a_0 b_2 + a_2 b_0 + a_1 b_3 + a_3 b_1)$
 - $c_3 = d_{0123} - (a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3) - c_1 - c_5$

SMAUG-T Karatsuba 실제 구현 관점

- 결과 조립

- $\tilde{c}(t) = c_0 + c_1t + c_2t^2 + c_3t^3 + c_4t^4 + c_5t^5 + c_6t^6$

- 이를 $t = x^{16}$ 에 대입해 배치

- $c(x) = c_0 + c_1x^{16} + c_2x^{32} + c_3x^{48} + c_4x^{64} + c_5x^{80} + c_6x^{96}$

SMAUG-T Reduction 코드 해석

- SMAUG-T의 modulus: $1024(2^{10})$ 혹은 $2048(2^{11})$

```
#define OVERFLOWING_MUL(X, Y) ((uint16_t)((uint32_t)(X) * (uint32_t)(Y)))
```

- 내부 연산은 uint16_t
- 자연 $\text{mod } 2^{16}$
- 최종적으로 $\text{mod } 2^{10}$
- $2^{16} \equiv 0 \pmod{2^{10}}$
 - $\text{mod } 2^{16}$ 후 $\text{mod } 2^{10} = \text{mod } 2^{10}$ 과 동일

결론

- NTRU+ 및 SMAUG-T 코드를 중심으로 KEM 핵심 구현 구조를 분석
- NTT, Toom-Cook, Karatsuba 등 다항식 연산 최적화 기법의 수학적, 코드적 연결 확인
- 상위 API 이해를 넘어 내부 연산 구조 파악이 실제 성능 최적화의 핵심
- 제시된 구현 기법만으로도 실무 및 임베디드 환경에서 충분한 적용 가능성을 가짐