



kpqC 구현기술 워크숍

Session1: 격자기반암호 최적화 구현

2022.09.30

국민대학교 금융정보보안학과
서 석 충



- ❖ NIST PQC 격자기반암호 개요
- ❖ 격자기반암호에서의 다항식 곱셈 알고리즘 소개
 - Karatsuba, Toom-Cook 기반 다항식 곱셈
 - NTT 기반 다항식 곱셈
- ❖ 격자기반암호에 적용된 정수 연산 감산 방법
- ❖ 레퍼런스 코드에 적용된 운영환경 별 최적화 사항 분석 (Cortex-M4/AVX2)
- ❖ 레퍼런스 코드에 적용된 샘플링 방법 소개

❖ NIST PQC 격자기반암호 개요

※ 격자기반암호 상수시간 구현 및 상수시간 구현 체크 방안

❖ 격자기반암호에서의 다항식 곱셈 알고리즘 소개

○ Karatsuba, Toom-Cook 기반 다항식 곱셈

○ NTT 기반 다항식 곱셈

※ Toom-Cook k-way, Karatsuba의 성능 비교 ($|q|=23\text{-bit}$)

※ 연산 구현 시, 메모리 점유율과 연산속도

❖ 격자기반암호에 적용된 정수 연산 감산 방법

※ 격자기반암호 상수시간 구현

❖ 레퍼런스 코드에 적용된 운영환경 별 최적화 사항 분석 (Cortex-M4/AVX2)

※ AVX 및 ARM CPU 구조 상 레티스 기반 PQC 알고리즘 효율적인 구현 기법

❖ 레퍼런스 코드에 적용된 샘플링 방법 소개

NIST PQC 격자기반암호 개요



❖ NIST PQC 공모전 표준화 대상 알고리즘

○ KEM: Crystals-Kyber

○ DS: Crystals-Dilithium, Falcon, SPHINCS+

분류	이름	기반문제
KEM	Classic McEliece	Code (Goppa)
	Crystals-Kyber	Lattice (Module-LWE)
	NTRU	Lattice (NTRU)
	SABER	Lattice (Module-LWR)
DS	Crystals-Dilithium	Lattice (Module-LWE)
	Falcon	Lattice (NTRU)
	Rainbow	MQ (UOV)
	SPHINCS+	Hash (2nd Preimage resistance)

NIST PQC 격자기반암호 개요



❖ NIST PQC 격자기반암호 주요 특징 (데이터 크기: 바이트단위)

알고리즘	기반문제	다항식 링	Degree n	Modulus q
Crystals-Kyber	Module LWE	$\mathbb{Z}_q[X]/\langle X^n + 1 \rangle$	256	3329
Crystals-Dilithium	Module LWR	$\mathbb{Z}_q[X]/\langle X^n + 1 \rangle$	256	8380417
Falcon	NTRU Lattice	$\mathbb{Z}_q[X]/\langle X^n + 1 \rangle$	512, 1024	12289

특징	Kyber			Dilithium			Falcon		
	SL	SK	PK	SL	SK	PK	SL	SK	PK
키 사이즈	1	1632	800	2	2528	1312	1	1281	897
	3	2400	1184	3	4000	1952	-	-	-
	5	3168	1568	5	4864	2592	5	2305	1793
암호문 or 서명 크기	1	768		2	2420		1	690	
	3	1088		3	3293		-	-	
	5	1568		5	4595		5	1330	

NIST PQC 격자기반암호 개요



❖Crystals-Kyber

	n	k	q	η_1	η_2	(d_u, d_v)	δ
KYBER512	256	2	3329	3	2	(10, 4)	2^{-139}
KYBER768	256	3	3329	2	2	(10, 4)	2^{-164}
KYBER1024	256	4	3329	2	2	(11, 5)	2^{-174}

- ❖ $R = \mathbb{Z}_q[X]/\langle X^{256} + 1 \rangle$, $q = 3329$
- ❖ dimension of A (k, k): Lv1(2), Lv3(3), Lv5(4)
- ❖ Public Key : $(A, t = A \cdot s + e)$, Secret Key : (s)
- ❖ Enc : $(u, v) = (A^T r + e_1, t^T r + e_2 + Decompress_q(m, 1))$
- ❖ Dec : $Compress_q(v - s^T u, 1)$

특징

- Module RLWE 구조
- NTT 기반 다항식 곱셈 적용 가능 (Incomplete NTT)

```

1: Gen
2:  $A \leftarrow R_q^{k \times k}$ 
3:  $s, e \leftarrow R_q^k$ 
4:  $t \leftarrow As + e$ 
5: Return  $(pk = (A, t, sk = s))$ 
6:
7: Enc
8:  $r \leftarrow R_q^k$ 
9:  $e_1 \leftarrow R_q^k, e_2 \leftarrow R_q$ 
10:  $u \leftarrow A^T r + e_1$ 
11:  $v \leftarrow t^T r + e_2 + Decompress_q(m, 1)$ 
12: Return  $C = (u, v)$ 
13:
14: Dec
15:  $m \leftarrow Compress_q(v - s^T u, 1)$ 
16: Return  $m$ 
    
```

NIST PQC 격자기반암호 개요



❖ Crystals-Dilithium

파라미터/보안레벨	dilithium2	dilithium3	dilithium5
q [modulus]	$8380417 = 2^{23} - 2^{13} + 1$		
d [dropped bits from t]	13		
τ [# of ± 1 's in c]	39	49	60
challenge entropy $\log \binom{256}{\tau} + \tau$	192	225	257
γ_1 [y coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, l) [dimensions of A]	(4, 4)	(6, 5)	(8, 7)
η [secret key range]	2	4	2
$\beta[\tau \cdot \eta]$	78	196	120
w [max. # 's in the hint h]	80	55	75
Repetition	4.25	5.1	3.85

- ❖ $R = \mathbb{Z}_q[X]/\langle X^{256} + 1 \rangle$, $q = 8380417$
- ❖ dimension of A (k, l): Lv2(4, 4), Lv3(6, 5), Lv5(8, 7)
- ❖ Public Key : $(A, t = A \cdot s_1 + s_2)$, Secret Key : (s_1, s_2)
- ❖ 서명 : $(z, c) = (y + c \cdot s_1, c = H(M || [A \cdot y]_d))$
- ❖ 검증 : 1. Compute: $A \cdot z - c \cdot t = A \cdot y - c \cdot s_2 \rightarrow [A \cdot y - c \cdot s_2]_d = [A \cdot y]_d$
2. Check if $c \neq \hat{c} (= H([A \cdot y]_d || M))$

Gen

```

01  $A \leftarrow R_q^{k \times \ell}$ 
02  $(s_1, s_2) \leftarrow S_n^\ell \times S_\eta^k$ 
03  $t := As_1 + s_2$ 
04 return  $(pk = (A, t), sk = (s_1, s_2))$ 

```

Sign(sk, M)

```

05  $z := \perp$ 
06 while  $z = \perp$  do
07    $y \leftarrow S_{\gamma_1-1}^\ell$ 
08    $w_1 := \text{HighBits}(Ay, 2\gamma_2)$ 
09    $c \in B_\tau := H(M || w_1)$ 
10    $z := y + cs_1$ 
11   if  $\|z\|_\infty \geq \gamma_1 - \beta$  or  $\|\text{LowBits}(Ay - cs_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ , then  $z := \perp$ 
12 return  $\sigma = (z, c)$ 

```

Verify($pk, M, \sigma = (z, c)$)

```

13  $w'_1 := \text{HighBits}(Az - ct, 2\gamma_2)$ 
14 if return  $\|z\|_\infty < \gamma_1 - \beta$  and  $c = H(M || w'_1)$ 

```

NIST PQC 격자기반암호 개요



❖ Falcon

❖ $R = Z_q[X]/\langle X^n + 1 \rangle$, $n = 512, 1024$, $q = 12289$

❖ f, g 에 대해 다음을 만족하는 F, G 를 찾음

- $[fG - gF = q \bmod \phi]$
- $h = gf^{-1} \bmod (q, \phi)$ 을 계산
- $B = \begin{bmatrix} f & g \\ F & G \end{bmatrix}$, $P = \begin{bmatrix} 1 & h \\ 0 & q \end{bmatrix}$ 생성
(B : 비밀 키, P : 공개 키)

Algorithm 4 Keygen(ϕ, q)

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus q

Ensure: A secret key sk , a public key pk

- 1: $f, g, F, G \leftarrow \text{NTRUGen}(\phi, q)$ ▷ Solving the NTRU equation
- 2: $B \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$
- 3: $\hat{B} \leftarrow \text{FFT}(B)$ ▷ Compute the FFT for each of the 4 components $\{g, -f, G, -F\}$
- 4: $\hat{G} \leftarrow \hat{B} \times \hat{B}^*$
- 5: $T \leftarrow \text{fFLDL}^*(\hat{G})$ ▷ Computing the LDL* tree
- 6: for each leaf $leaf$ of T do ▷ Normalization step
- 7: $leaf.value \leftarrow \sigma / \sqrt{leaf.value}$
- 8: $sk \leftarrow (\hat{B}, T)$
- 9: $h \leftarrow gf^{-1} \bmod q$
- 10: $pk \leftarrow h$
- 11: return sk, pk

Algorithm 10 Sign($m, sk, \lfloor \beta^2 \rfloor$)

Require: A message m , a secret key sk , a bound $\lfloor \beta^2 \rfloor$

Ensure: A signature sig of m

- 1: $r \leftarrow \{0, 1\}^{320}$ uniformly
- 2: $c \leftarrow \text{HashToPoint}(r \| m, q, n)$
- 3: $t \leftarrow \left(-\frac{1}{q} \text{FFT}(c) \odot \text{FFT}(F), \frac{1}{q} \text{FFT}(c) \odot \text{FFT}(f) \right)$ ▷ $t = (\text{FFT}(c), \text{FFT}(0)) \cdot \hat{B}^{-1}$
- 4: do
- 5: do
- 6: $z \leftarrow \text{ffSampling}_n(t, T)$
- 7: $s = (t - z)B$ ▷ At this point, s follows a Gaussian distribution: $s \sim D_{(c,0)+\Lambda(B), \sigma, 0}$
- 8: while $\|s\|^2 > \lfloor \beta^2 \rfloor$ ▷ Since s is in FFT representation, one may use (3.8) to compute $\|s\|^2$
- 9: $(s_1, s_2) \leftarrow \text{invFFT}(s)$ ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
- 10: $s \leftarrow \text{Compress}(s_2, 8 \cdot \text{sbytelen} - 328)$ ▷ Remove 1 byte for the header, and 40 bytes for r
- 11: while $(s = \perp)$
- 12: return $\text{sig} = (r, s)$

Algorithm 16 Verify($m, \text{sig}, pk, \lfloor \beta^2 \rfloor$)

Require: A message m , a signature $\text{sig} = (r, s)$, a public key $pk = h \in \mathbb{Z}_q[x]/(\phi)$, a bound $\lfloor \beta^2 \rfloor$

Ensure: Accept or reject

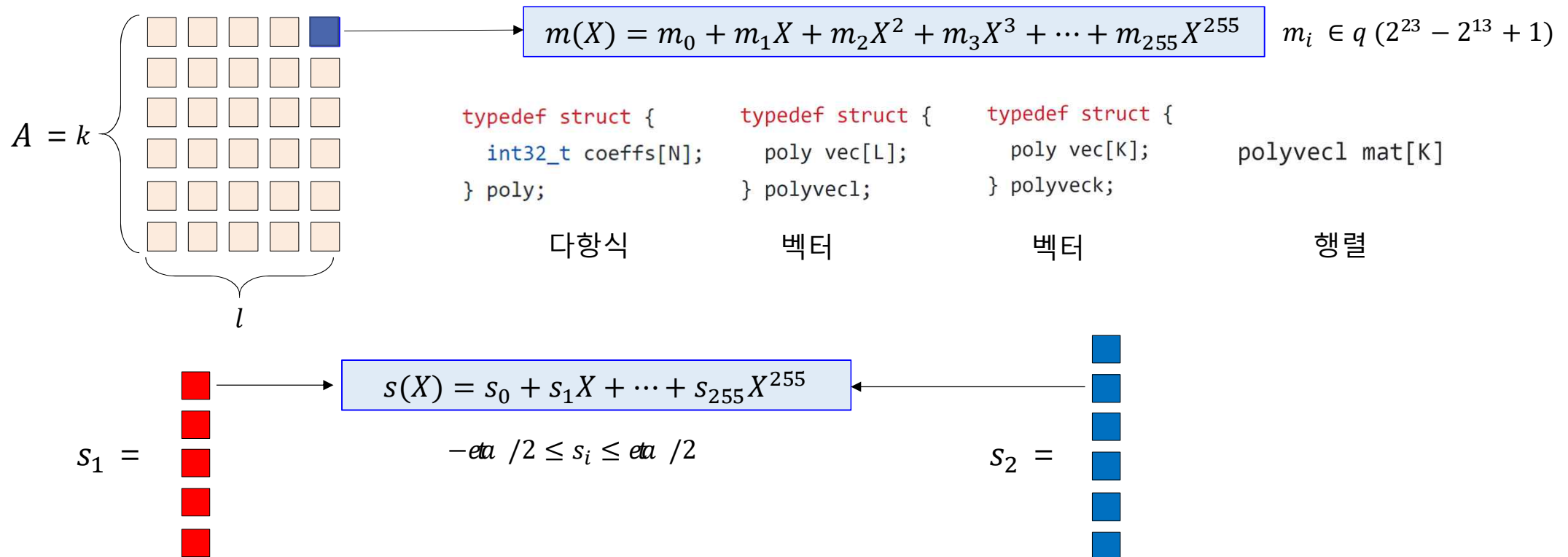
- 1: $c \leftarrow \text{HashToPoint}(r \| m, q, n)$
- 2: $s_2 \leftarrow \text{Decompress}(s, 8 \cdot \text{sbytelen} - 328)$
- 3: if $(s_2 = \perp)$ then ▷ Reject invalid encodings
- 4: reject ▷ s_1 should be normalized between $\left[-\frac{q}{2}\right]$ and $\left[\frac{q}{2}\right]$
- 5: $s_1 \leftarrow c - s_2 h \bmod q$
- 6: if $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ then
- 7: accept
- 8: else
- 9: reject ▷ Reject signatures that are too long

NIST PQC 격자기반암호 개요



❖ 격자기반암호 핵심 연산

○ (행렬 × 벡터) 연산 또는 (벡터 × 벡터) 연산

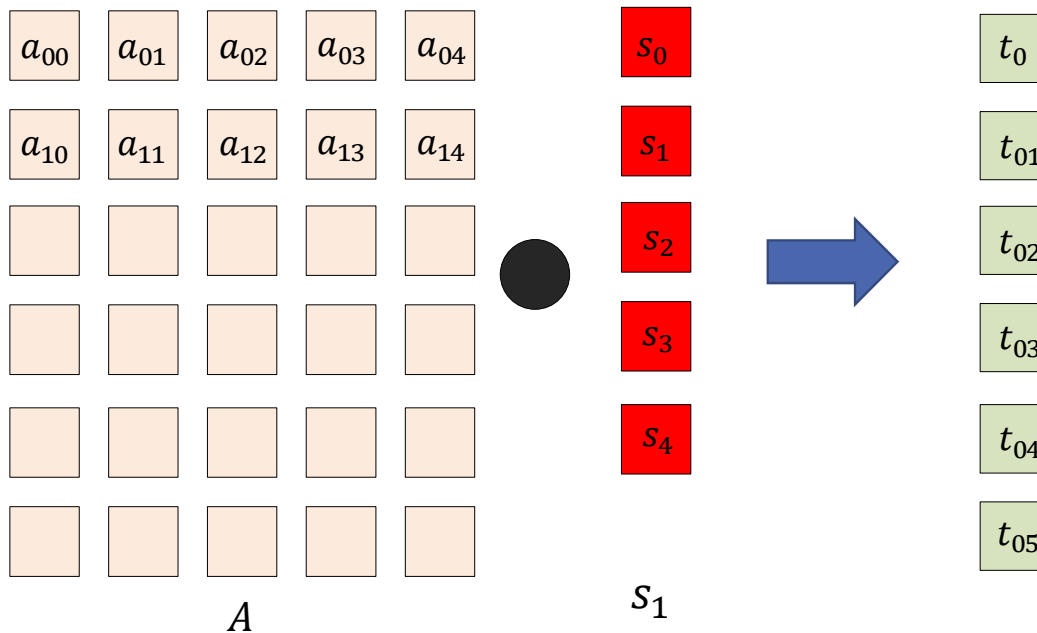


NIST PQC 격자기반암호 개요



❖ 격자기반암호 핵심 연산

○ (행렬 × 벡터) 연산 또는 (벡터 × 벡터) 연산



$$t_i = a_{i0} \times s_0 + a_{i1} \times s_1 + a_{i2} \times s_2 + a_{i3} \times s_3 + a_{i4} \times s_4$$

256차 다항식 × 256차 다항식

NIST PQC 격자기반암호 개요



❖ 격자기반암호 핵심 연산

○ (행렬 × 벡터) 연산 또는 (벡터 × 벡터) 연산

▪ n 차로 구성된 다항식 A 와 B 의 곱셈 연산 ($n = 256, 512, 1024$)

▪ 연산량: $O(n^2)$ (n 은 차수)

School-book 기반 다항식 곱셈

• $C(X) = \sum_{i=0}^{255} c_i X^i \bmod \langle X^{256} + 1 \rangle, A(X) = \sum_{i=0}^{255} a_i X^i, B(X) = \sum_{i=0}^{255} b_i X^i, a_i, b_i \in \mathbb{Z}_q$

$$(a_{255} X^{255} + \dots + a_3 X^3 + a_2 X^2 + a_1 X + a_0) \cdot b_0$$

$$(a_{255} X^{255} + \dots + a_3 X^3 + a_2 X^2 + a_1 X + a_0) \cdot b_1 X$$

$$(a_{255} X^{255} + \dots + a_3 X^3 + a_2 X^2 + a_1 X + a_0) \cdot b_2 X^2$$

...

$$+ (a_{255} X^{255} + \dots + a_3 X^3 + a_2 X^2 + a_1 X + a_0) \cdot b_{255} X^{255}$$

$\mathbb{Z}_q[X] / \langle x^{256} + 1 \rangle$
상에서의 곱셈 연산
(감산필요)

$$\boxed{a_{255} \cdot b_{255} X^{255+255} + \dots + a_i \cdot b_j X^{i+j} + \dots + a_0 \cdot b_0}$$

\mathbb{Z}_q 상에서 감산 필요

\mathbb{Z}_q 상에서의 곱셈 연산(감산 필요)

NIST PQC 격자기반암호 개요



❖ R_q 상에서의 감산

○ n 차 이상의 항들에 대해 $f(X) = X^n + 1$ 의 성질을 이용하여 감산

$$T(X) = \sum_{i=0}^{510} (t_i X^i), \quad T(X) = \sum_{i=0}^{255} (t_i - t_{i+256} X^i) \text{ where } f(X) = X^{256} + 1$$

❖ Z_q 상에서의 감산

○ q 범위를 벗어난 계수들에 대해 $\text{mod } q$ 연산 수행

▪ Kyber case ($q = 3329$)

- 12-bit \times 12-bit = 24-bit
→ 16-bit \times 16-bit = 32-bit → 32-bit 자료형에 대한 '%' 연산 수행

▪ Dilithium case ($q = 8380417$)

- 23-bit \times 23-bit = 46-bit
→ 32-bit \times 32-bit = 64-bit → 32-bit 자료형에 대한 '%' 연산 수행

▪ SABER case ($q = 2^{13}$)

- 13-bit \times 13-bit = 26-bit
→ 16-bit \times 16-bit = 32-bit → 32-bit 자료형에 대한 '&' 연산 수행

NIST PQC 격자기반암호 개요



❖% 연산자

○아키텍처마다 연산 지연 시간이 가변적이거나 느림

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Divide	SDIV, UDIV	4 - 20	1/20 - 1/4	M	1

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized pipelines	Notes
Divide, W-form	SDIV, UDIV	4-20	1/20 – 1/4	M	1
Divide, X-form	SDIV, UDIV	4-36	1/36 - 1/4	M	1

- 참고문헌 : <https://developer.arm.com/documentation/uan0015/b/>
https://www.agner.org/optimize/#manual_instr_tab

NIST PQC 격자기반암호 개요



❖ NIST 격자기반 암호에서는 상수시간 구현을 기본 조건으로 함

○ 상수시간 구현 조건

- variable-time 명령어 사용 금지
 - C언어의 % 연산자 (udiv in Cortex-M4 명령어), UMULL 곱셈 명령어 (Cortex-M3 명령어)
- 비밀 정보에 의존적인 분기문 사용 금지
 - Balanced 분기문이라도 해도, BPU (Branch Prediction Unit)에 의해 실행시간 차이 존재
- 비밀 정보에 의존적인 테이블 접근 금지
 - Microarchitectural Attacks

○ 상수시간 구현 포인트

- 다항식 곱셈 연산에서 정수 상에서의 감산 연산 → 상수시간 감산 방법 적용
- KEM Decap 과정에서의 조건문 → **CMOV** 방법 적용

NIST PQC 격자기반암호 개요



❖상수시간 구현 체크 방법 (Valgrind 이용: <https://valgrind.org/>)

○비밀 정보를 담는 변수를 초기화하지 않고 취약점 탐지 시, 분기문/LUT 연산에 대한 탐지 수행

```
// An example function for timing leak (2)
unsigned char leaky_branch1(unsigned char v)
{
    // branching depending on v.
    // if v is secret, this leaks.
    if(v > 42)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

unsigned char LUT[] = {3, 1, 4, 1};

// An example function for timing leak (3)
unsigned char leaky_lookup(unsigned char idx)
{
    // memory access at an address depending on idx.
    // if idx is secret, this leaks.
    return LUT[idx & 3];
}
```

```
gegehe@ubuntu:~/Programming/test_valgrind$ valgrind ./a.out
==3096== Memcheck, a memory error detector
==3096== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3096== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3096== Command: ./a.out
==3096==
==3096== Conditional jump or move depends on uninitialised value(s)
==3096==    at 0x4005A3: leaky_branch1 (test2.c:11)
==3096==    by 0x40063B: main (test2.c:66)
==3096==
0
==3096== Use of uninitialised value of size 8
==3096==    at 0x4005C5: leaky_lookup (test2.c:28)
==3096==    by 0x40065A: main (test2.c:67)
==3096==
3
```

NIST PQC 격자기반암호 개요



❖상수시간 구현 체크 방법(Timecop: <https://www.post-apocalyptic-crypto.org/timecop/>)

○Vagrind를 이용하여 Supercop에 제출된 암호구현물에 대해 상수시간 구현여부 체크

Results

TIMECOP applied the described analysis to over 2,700 implementations of cryptographic algorithms. The results are sorted hierarchically, following the structure of the SUPERCOP benchmarking suite.

Authenticated ciphers ([crypto_aead](#)) >

[crypto_auth](#) >

[crypto_box](#)

[crypto_core](#)

Diffie-Hellman ([crypto_dh](#))

Public-key encryption ([crypto_encrypt](#))

Hash functions ([crypto_hash](#))

[crypto_hashblocks](#)

Key-encapsulation mechanisms ([crypto_kem](#))

[kindi512321](#)

Implementations:

ref

0 / 4 / 5



[kyber1024](#)

Implementations:

ref

0 / 8 / 1



[kyber512](#)

Implementations:

ref

0 / 8 / 1



[kyber768](#)

Implementations:

ref

0 / 8 / 1



■ 한계점

- 가변시간에 동작하는 명령어로 인한 취약점 탐지 어려움
- C/C++ 코드에 대해서만 테스트
- 공개키/비밀키가 혼용되어 사용될 때, 공개키로 인해 시간차가 발생하는 경우도 탐지 (오탐)

NIST PQC 격자기반암호 개요



❖상수시간 구현 체크 방법(Timecop: <https://www.post-apocalyptic-crypto.org/timecop/>)

○Vagrind를 이용하여 Supercop에 제출된 암호구현물에 대해 상수시간 구현여부 체크

Output

[Download](#)

Operation `crypto_kem`
Primitive `kyber768`
Implementation `ref`
Compiler options `clang -march=native -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments`
Host `venus`

```
1 error 0
2 b35411b279c770f10f332fc1d41543e1b65ee2fe0ad868e1fce8d0962f669b3a 46802420 414739953 4300000000 crypto_kem/kyber768/ref
3 ==15269== Memcheck, a memory error detector
4 ==15269== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
5 ==15269== Using Valgrind-3.15.0.GIT and LibVEX; rerun with -h for copyright info
6 ==15269== Command: ./try
7 ==15269==
8 ==15269== Conditional jump or move depends on uninitialised value(s)
9 ==15269== at 0x403E39: gen_matrix (in /home/moritz/Uni/2015-2016/research-internship/letstrythisagain/supercop-20181113/bench/venus/work/compile/try)
10 ==15269== by 0x40453B: indcpa_enc (in /home/moritz/Uni/2015-2016/research-internship/letstrythisagain/supercop-20181113/bench/venus/work/compile/try)
11 ==15269== by 0x402750: crypto_kem_kyber768_ref_dec (in /home/moritz/Uni/2015-2016/research-internship/letstrythisagain/supercop-20181113/bench/venus/work/compile/try)
12 ==15269== by 0x400E7E: timecop_doit (in /home/moritz/Uni/2015-2016/research-internship/letstrythisagain/supercop-20181113/bench/venus/work/compile/try)
13 ==15269== by 0x401F44: main (in /home/moritz/Uni/2015-2016/research-internship/letstrythisagain/supercop-20181113/bench/venus/work/compile/try)
14 ==15269== Uninitialised value was created by a client request
15 ==15269== at 0x400DE4: timecop_doit (in /home/moritz/Uni/2015-2016/research-internship/letstrythisagain/supercop-20181113/bench/venus/work/compile/try)
16 ==15269== by 0x401F44: main (in /home/moritz/Uni/2015-2016/research-internship/letstrythisagain/supercop-20181113/bench/venus/work/compile/try)
```

❖상수시간 구현 체크 방법(ct-verif: <https://github.com/imdea-software/verifying-constant-time>)

○Verifying Constant-Time Implementations, Usenix 2016

- <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>

○상수 구현에 대한 정형검증

- Crypto primitives: TEA, NaCl
- TLS (record protocol)
- Fixed-point arithmetic: libfixedtimefixedpoint
- ECC: curve25519-donna, FourQLib

NIST PQC 격자기반암호 개요



❖ NIST PQC에 적용된 상수시간 구현

○ Lazy reduction, CMOV, Control-bit 기반 연산

이름	유한체 연산	추가 적인 대응 부분
Kyber	- Signed Montgomery/Barrett 감산에서 Lazy reduction 적용	KEM.decap 연산에서 CMOV 적용
SABER	-	KEM.decap 연산에서 CMOV 적용
Dilithium	Signed Montgomery/Barrett 감산에서 Lazy reduction 적용	-
Falcon	- unsigned Montgomery 감산에서 control-bit을 이용한 reduction 적용 - 덧셈/뺄셈 연산에서 control-bit을 이용한 reduction 적용	- $h \leftarrow gf^{-1} \bmod q$ 연산에서 지수승 기반 역원계산 ($y^{-1} = y^{q-2} \bmod q$) - NTRU equation solve($fG - gF = q \bmod \phi$)에서 constant-time binary GCD 이용

NIST PQC 격자기반암호 개요



❖NIST PQC에 적용된 상수시간 구현 (KEM.decap cmov 함수)

```
int crypto_kem_dec(unsigned char *ss,
                  const unsigned char *ct,
                  const unsigned char *sk)
{
    size_t i;
    int fail;
    uint8_t buf[2*KYBER_SYMBYTES];
    /* Will contain key, coins */
    uint8_t kr[2*KYBER_SYMBYTES];
    uint8_t cmp[KYBER_CIPHTEXTBYTES];
    const uint8_t *pk = sk+KYBER_INDCPA_SECRETKEYBYTES;

    indcpa_dec(buf, ct, sk);

    /* Multitarget countermeasure for coins + contributory KEM */
    for(i=0;i<KYBER_SYMBYTES;i++)
        buf[KYBER_SYMBYTES+i] = sk[KYBER_SECRETKEYBYTES-2*KYBER_SYMBYTES+i];
    hash_g(kr, buf, 2*KYBER_SYMBYTES);

    /* coins are in kr+KYBER_SYMBYTES */
    indcpa_enc(cmp, buf, pk, kr+KYBER_SYMBYTES);

    fail = verify(ct, cmp, KYBER_CIPHTEXTBYTES);

    /* overwrite coins in kr with H(c) */
    hash_h(kr+KYBER_SYMBYTES, ct, KYBER_CIPHTEXTBYTES);

    /* Overwrite pre-k with z on re-encryption failure */
    cmov(kr, sk+KYBER_SECRETKEYBYTES-KYBER_SYMBYTES, KYBER_SYMBYTES, fail);

    /* hash concatenation of pre-k and H(c) to k */
    kdf(ss, kr, 2*KYBER_SYMBYTES);
    return 0;
}
```

```
/* returns 0 for equal strings, 1 for non-equal strings */
int verify(const unsigned char *a, const unsigned char *b, size_t len)
{
    uint64_t r;
    size_t i;
    r = 0;

    for (i = 0; i < len; i++)
        r |= a[i] ^ b[i];

    r = (-r) >> 63;
    return r;
}

/*****
* Name:          cmov
*
* Description:   Copy len bytes from x to r if b is 1;
*               don't modify x if b is 0. Requires b to be in {0,1};
*               assumes two's complement representation of negative integers.
*               Runs in constant time.
*
* Arguments:    uint8_t *r:      pointer to output byte array
*               const uint8_t *x: pointer to input byte array
*               size_t len:      Amount of bytes to be copied
*               uint8_t b:       Condition bit; has to be in {0,1}
*****/
void cmov(uint8_t *r, const uint8_t *x, size_t len, uint8_t b)
{
    size_t i;

    b = -b;
    for(i=0;i<len;i++)
        r[i] ^= b & (r[i] ^ x[i]);
}
```

NIST PQC 격자기반암호 개요



❖ NIST PQC에 적용된 상수시간 구현 (역원 연산)

○ 지수승 기반의 역원 연산 ($q = 12289$ 이기 때문에, $q - 2 = 12287 = e$ 에 대한 addition chain 구성)

- Montgomery 도메인 상에서 곱셈과 제곱 연산을 반복 사용

```
* e0 = 1
* e1 = 2 * e0 = 2
* e2 = e1 + e0 = 3
* e3 = e2 + e1 = 5
* e4 = 2 * e3 = 10
* e5 = 2 * e4 = 20
* e6 = 2 * e5 = 40
* e7 = 2 * e6 = 80
* e8 = 2 * e7 = 160
* e9 = e8 + e2 = 163
* e10 = e9 + e8 = 323
* e11 = 2 * e10 = 646
* e12 = 2 * e11 = 1292
* e13 = e12 + e9 = 1455
* e14 = 2 * e13 = 2910
* e15 = 2 * e14 = 5820
* e16 = e15 + e10 = 6143
* e17 = 2 * e16 = 12286
* e18 = e17 + e0 = 12287

y0 = mq_montymul(y, R2);
y1 = mq_montysqr(y0);
y2 = mq_montymul(y1, y0);
y3 = mq_montymul(y2, y1);
y4 = mq_montysqr(y3);
y5 = mq_montysqr(y4);
y6 = mq_montysqr(y5);
y7 = mq_montysqr(y6);
y8 = mq_montysqr(y7);
y9 = mq_montymul(y8, y2);
y10 = mq_montymul(y9, y8);
y11 = mq_montysqr(y10);
y12 = mq_montysqr(y11);
y13 = mq_montymul(y12, y9);
y14 = mq_montysqr(y13);
y15 = mq_montysqr(y14);
y16 = mq_montymul(y15, y10);
y17 = mq_montysqr(y16);
y18 = mq_montymul(y17, y0);
return mq_montymul(y18, x);
```

- 참고문헌: mq_div_12289 함수 in keygen.c in Falcon-512 project

NIST PQC 격자기반암호 개요



❖ NIST PQC에 적용된 상수시간 구현 (constant-time binary GCD)

○ $fG - gF = q \bmod \phi$ 연산 중, binary GCD 이용

```
/*
 * rt = 1 if a_hi > b_hi, 0 otherwise.
 */
rz = b_hi - a_hi;
rt = (uint32_t)((rz ^ ((a_hi ^ b_hi)
                      & (a_hi ^ rz))) >> 63);

/*
 * cAB = 1 if b must be subtracted from a
 * cBA = 1 if a must be subtracted from b
 * cA = 1 if a must be divided by 2
 *
 * Rules:
 *
 *   cAB and cBA cannot both be 1.
 *   If a is not divided by 2, b is.
 */
oa = (a_lo >> i) & 1;
ob = (b_lo >> i) & 1;
cAB = oa & ob & rt;
cBA = oa & ob & ~rt;
cA = cAB | (oa ^ 1);
```

```
/*
 * Conditional subtractions.
 */
a_lo -= b_lo & -cAB;
a_hi -= b_hi & -(uint64_t)cAB;
pa -= qa & -(int64_t)cAB;
pb -= qb & -(int64_t)cAB;
b_lo -= a_lo & -cBA;
b_hi -= a_hi & -(uint64_t)cBA;
qa -= pa & -(int64_t)cBA;
qb -= pb & -(int64_t)cBA;

/*
 * Shifting.
 */
a_lo += a_lo & (cA - 1);
pa += pa & ((int64_t)cA - 1);
pb += pb & ((int64_t)cA - 1);
a_hi ^= (a_hi ^ (a_hi >> 1)) & -(uint64_t)cA;
b_lo += b_lo & -cA;
qa += qa & -(int64_t)cA;
qb += qb & -(int64_t)cA;
b_hi ^= (b_hi ^ (b_hi >> 1)) & ((uint64_t)cA - 1);
```

```
* - If a is even, then:
*   a <- a/2
*   u0 <- u0/2 mod y
*   v0 <- v0/2 mod x
*
* - Otherwise, if b is even, then:
*   b <- b/2
*   u1 <- u1/2 mod y
*   v1 <- v1/2 mod x
*
* - Otherwise, if a > b, then:
*   a <- (a-b)/2
*   u0 <- (u0-u1)/2 mod y
*   v0 <- (v0-v1)/2 mod x
*
* - Otherwise:
*   b <- (b-a)/2
*   u1 <- (u1-u0)/2 mod y
*   v1 <- (v1-v0)/2 mod y
```

• 참고문헌: zint_bezout 함수 in keygen.c in Falcon-512 project

❖ NIST 격자기반 암호에 적용된 연산 알고리즘

이름	파라미터	다항식 곱셈 연산	유한체 감산
Kyber	$n = 256, q = 3329 \text{ such that } (n q - 1)$	(incomplete) NTT 기반	(signed) Montgomery, (signed) Barret
Dilithium	$n = 256, q = 8380417 \text{ such that } (2n q - 1)$	(complete) NTT 기반	(signed) Montgomery, (signed) Barret
Falcon	$q = 12289, \phi = X^n + 1, n = 512, 1024$ $\text{such that } (2n q - 1)$	(complete) NTT 기반, FFT 기반	(unsigned) Montgomery
SABER	$q = 2^{13}, p = 2^{10}, n = 256$	Toom-Cook 기반 또는 NTT 기반 ($q' (> q)$ 이용)	Masking 또는 (signed) Montgomery

❖다항식 곱셈 방법 종류와 복잡도

○Schoolbook: $O(n^2)$

○Karatsuba: $O(n^{\lg(3)/\lg(2)}) \approx O(n^{1.58})$

○Toom-Cook

- Toom-3: $O(n^{\lg(5)/\lg(3)}) \approx O(n^{1.46})$

- Toom-4: $O(n^{\lg(7)/\lg(4)}) \approx O(n^{1.40})$

○NTT: $O(n \lg n)$

❖ Karatsuba 곱셈

- 하나의 다항식을 균등하게 2개의 $(n/2)$ 차 다항식으로 분할
- 3번의 $(n/2)$ 차 다항식 곱셈과 추가적인 덧셈/뺄셈을 이용하여 n 차 다항식 곱셈을 수행
- 재귀적으로도 구성 가능

- 즉, $(n/2)$ 차 다항식 곱셈에서 다시 Karatsuba 곱셈 적용 (2-Way Karatsuba)

- $a = a^{(1)}X^{n/2} + a^{(0)}, b = b^{(1)}X^{n/2} + b^{(0)}$

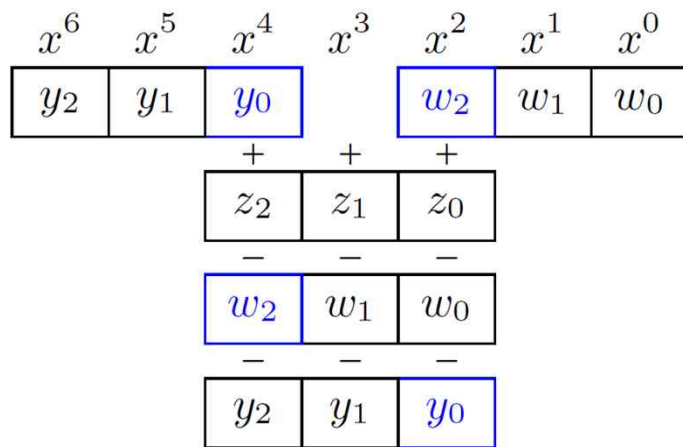
- $c = a \cdot b = \left(a^{(1)}X^{\frac{n}{2}} + a^{(0)}\right) \left(b^{(1)}X^{\frac{n}{2}} + b^{(0)}\right)$

$$= a^{(1)}b^{(1)}X^n + (a^{(1)}b^{(0)} + a^{(0)}b^{(1)})X^{n/2} + a^{(0)}b^{(0)}$$

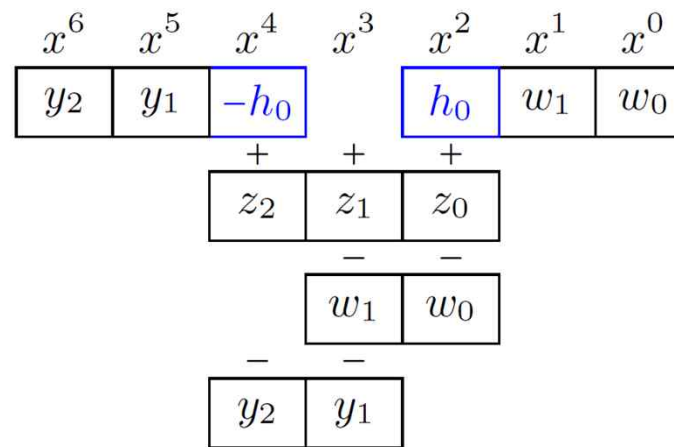
$$= a^{(1)}b^{(1)}X^n + \left((a^{(0)} + a^{(1)})(b^{(0)} + b^{(1)}) - a^{(1)}b^{(1)} - a^{(0)}b^{(0)}\right)X^{n/2} + a^{(0)}b^{(0)}$$

❖ Refined Karatsuba 곱셈

○ 뺄셈의 횟수를 줄일 수 있음 (공통되는 연산을 치환하여 계산)



(a) Karatsuba



(b) Refined Karatsuba

- $a = a_3X^3 + a_2X^2 + a_1X^1 + a_0, b = b_3X^3 + b_2X^2 + b_1X^1 + b_0$
- $w = (a_1X + a_0)(b_1X + b_0), y = (a_3X + a_2)(b_3X + b_2), z = ((a_2 + a_0)(a_3 + a_1)X)((b_2 + b_0)(b_3 + b_1)X)$
- $h = w_2 - y_0$

❖ Toom-Cook 곱셈

- 다항식을 k 개로 분할하여 곱셈 수행 (k -way Toom-Cook)
- NTT를 적용할 수 없는 파라미터에서 사용 가능 (예: SABER)
- 보간법을 이용하여 곱셈 결과에 대한 계수 복원
 - $n - 1$ 차 다항식은, 서로 다른 n 개의 점으로 표현 가능
- 곱셈 과정
 - 다항식 분할
 - 분할된 다항식에 대해 (임의의 값에 대한) 평가
 - Point value 표현
 - 점별 곱셈 (Point-wise 곱셈)
 - 보간, 재구성
 - $a(X) \cdot b(X)$ 의 결과값으로부터 $c(X)$ 의 계수 복원
- 감산 다항식을 이용한 감산 수행

❖Toom-Cook 곱셈

○Toom-Cook 3-way

- $a = a^{(2)}Y^2 + a^{(1)}Y + a^{(0)}, b = b^{(2)}Y^2 + b^{(1)}Y + b^{(0)}$ where $Y = X^{n/3}$
- $Y = \{0, 1, -1, -2, \infty\}$ 점 상에서 다항식 평가 (곱셈 결과는 항이 5개)

$$\begin{bmatrix} a(0) \\ a(1) \\ a(-1) \\ a(-2) \\ a(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a^{(0)} \\ a^{(1)} \\ a^{(2)} \end{bmatrix}$$

$$a(0) = a^{(0)}$$

$$a(1) = a^{(0)} + a^{(1)} + a^{(2)}$$

$$a(-1) = a^{(0)} - a^{(1)} + a^{(2)}$$

$$a(-2) = a^{(0)} - 2a^{(1)} + 4a^{(2)}$$

$$a(\infty) = a^{(2)}$$

$$\begin{bmatrix} b(0) \\ b(1) \\ b(-1) \\ b(-2) \\ b(\infty) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b^{(0)} \\ b^{(1)} \\ b^{(2)} \end{bmatrix}$$

$$b(0) = b^{(0)}$$

$$b(1) = b^{(0)} + b^{(1)} + b^{(2)}$$

$$b(-1) = b^{(0)} - b^{(1)} + b^{(2)}$$

$$b(-2) = b^{(0)} - 2b^{(1)} + 4b^{(2)}$$

$$b(\infty) = b^{(2)}$$

❖Toom-Cook 곱셈

○Toom-Cook 3-way

- 점별 곱셈 수행: $r(0), r(1), r(-1), r(-2), r(\infty)$ 를 계산
 $(r(X) = a(X) \cdot b(X) = r_4X^4 + r_3X^3 + r_2X^2 + r_1X + r_0$ 로 가정)
- 보간: $r(0), r(1), r(-1), r(-2), r(\infty)$ 로 부터 r_i 복원

$$\begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix} = \begin{pmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ (-1)^0 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ (-2)^0 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix}.$$

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix}$$

unknown

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & -1 & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & \frac{1}{6} & \frac{1}{2} & -\frac{1}{6} & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r(0) \\ r(1) \\ r(-1) \\ r(-2) \\ r(\infty) \end{pmatrix}$$

known

❖Toom-Cook 곱셈 (SABER Case)

○Toom-Cook 4-way (256차 다항식을 4개의 64차 다항식으로 분할)

- $\{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$ 의 점에 대해 평가 및 곱셈 → 7개의 64차 다항식 곱셈 수행
- 64차 다항식에 대한 곱셈 수행 시, Karatsuba 방법 이용 (2Level: 16차 곱셈 9번 수행)

Input: Two polynomials $A(x)$ and $B(x)$ of degree $n = 256$

Output: $C(x) = A(x) * B(x)$

// Splitting $A(x)$ into four polynomials of size 64

1 $A(y) = A_3 \cdot y^3 + A_2 \cdot y^2 + A_1 \cdot y + A_0$ where $y = x^{64}$

// Splitting $B(x)$ into four polynomials of size 64

2 $B(y) = B_3 \cdot y^3 + B_2 \cdot y^2 + B_1 \cdot y + B_0$

// Evaluation of the polynomials at $y = \{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$.

3 $w_1 = A(\infty) * B(\infty) = A_3 * B_3$

4 $w_2 = A(2) * B(2) = (A_0 + 2 \cdot A_1 + 4 \cdot A_2 + 8 \cdot A_3) * (B_0 + 2 \cdot B_1 + 4 \cdot B_2 + 8 \cdot B_3)$

5 $w_3 = A(1) * B(1) = (A_0 + A_1 + A_2 + A_3) * (B_0 + B_1 + B_2 + B_3)$

6 $w_4 = A(-1) * B(-1) = (A_0 - A_1 + A_2 - A_3) * (B_0 - B_1 + B_2 - B_3)$

7 $w_5 = A(\frac{1}{2}) * B(\frac{1}{2}) = (8 \cdot A_0 + 4 \cdot A_1 + 2 \cdot A_2 + A_3) * (8 \cdot B_0 + 4 \cdot B_1 + 2 \cdot B_2 + B_3)$

8 $w_6 = A(\frac{-1}{2}) * B(\frac{-1}{2}) = (8 \cdot A_0 - 4 \cdot A_1 + 2 \cdot A_2 - A_3) * (8 \cdot B_0 - 4 \cdot B_1 + 2 \cdot B_2 - B_3)$

9 $w_7 = A(0) * B(0) = A_0 * B_0$

// Interpolation

10 $w_2 = w_2 + w_5$

11 $w_6 = w_6 - w_5$

12 $w_4 = (w_4 - w_3)/2$

13 $w_5 = w_5 - w_1 - 64 \cdot w_7$

14 $w_3 = w_3 + w_4$

15 $w_5 = 2 \cdot w_5 + w_6$

16 $w_2 = w_2 - 65 \cdot w_3$

17 $w_3 = w_3 - w_7 - w_1$

18 $w_2 = w_2 + 45 \cdot w_3$

19 $w_5 = (w_5 - 8 \cdot w_3)/24$

20 $w_6 = w_6 + w_2$

21 $w_2 = (w_2 + 16 \cdot w_4)/18$

22 $w_3 = w_3 - w_5$

23 $w_4 = -(w_4 + w_2)$

24 $w_6 = (30 \cdot w_2 - w_6)/60$

25 $w_2 = w_2 - w_6$

26 **return** $C(y) = w_1 \cdot y^6 + w_2 \cdot y^5 + w_3 \cdot y^4 + w_4 \cdot y^3 + w_5 \cdot y^2 + w_6 \cdot y + w_7$

- 참고문헌 : SABER on ARM, TCHES 2018

❖Toom-Cook 곱셈 (SABER Case)

○Toom-Cook 4-way (256차 다항식을 4개의 64차 다항식으로 분할)

Input: Two polynomials $A(x)$ and $B(x)$ of degree $n = 256$

Output: $C(x) = A(x) * b(x)$

// Splitting $A(x)$ into four polynomials of size 64

1 $A(y) = A_3 \cdot y^3 + A_2 \cdot y^2 + A_1 \cdot y + A_0$ where $y = x^{64}$

// Splitting $B(x)$ into four polynomials of size 64

2 $B(y) = B_3 \cdot y^3 + B_2 \cdot y^2 + B_1 \cdot y + B_0$

// Evaluation of the polynomials at $y = \{0, \pm 1, \pm \frac{1}{2}, 2, \infty\}$.

3 $w_1 = A(\infty) * B(\infty) = A_3 * B_3$

4 $w_2 = A(2) * B(2) = (A_0 + 2 \cdot A_1 + 4 \cdot A_2 + 8 \cdot A_3) * (B_0 + 2 \cdot B_1 + 4 \cdot B_2 + 8 \cdot B_3)$

5 $w_3 = A(1) * B(1) = (A_0 + A_1 + A_2 + A_3) * (B_0 + B_1 + B_2 + B_3)$

6 $w_4 = A(-1) * B(-1) = (A_0 - A_1 + A_2 - A_3) * (B_0 - B_1 + B_2 - B_3)$

7 $w_5 = A(\frac{1}{2}) * B(\frac{1}{2}) = (8 \cdot A_0 + 4 \cdot A_1 + 2 \cdot A_2 + A_3) * (8 \cdot B_0 + 4 \cdot B_1 + 2 \cdot B_2 + B_3)$

8 $w_6 = A(\frac{-1}{2}) * B(\frac{-1}{2}) = (8 \cdot A_0 - 4 \cdot A_1 + 2 \cdot A_2 - A_3) * (8 \cdot B_0 - 4 \cdot B_1 + 2 \cdot B_2 - B_3)$

9 $w_7 = A(0) * B(0) = A_0 * B_0$

```
karatsuba_simple(aw1, bw1, w1);
karatsuba_simple(aw2, bw2, w2);
karatsuba_simple(aw3, bw3, w3);
karatsuba_simple(aw4, bw4, w4);
karatsuba_simple(aw5, bw5, w5);
karatsuba_simple(aw6, bw6, w6);
karatsuba_simple(aw7, bw7, w7);
```

// 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;
}
```

```
for (j = 0; j < N_SB; ++j) {
    r0 = B0[j];
    r1 = B1[j];
    r2 = B2[j];
    r3 = B3[j];
    r4 = r0 + r2;
    r5 = r1 + r3;
    r6 = r4 + r5;
    r7 = r4 - r5;
    bw3[j] = r6;
    bw4[j] = r7;
    r4 = ((r0 << 2) + r2) << 1;
    r5 = (r1 << 2) + r3;
    r6 = r4 + r5;
    r7 = r4 - r5;
    bw5[j] = r6;
    bw6[j] = r7;
    r4 = (r3 << 3) + (r2 << 2) + (r1 << 1) + r0;
    bw2[j] = r4;
    bw7[j] = r0;
    bw1[j] = r3;
}
```


❖Toom-Cook 곱셈 (SABER Case)

○Toom-Cook 4-way (256차 다항식을 4개의 64차 다항식으로 분할)

```
// Interpolation
10 w2 = w2 + w5
11 w6 = w6 - w5
12 w4 = (w4 - w3)/2
13 w5 = w5 - w1 - 64 · w7
14 w3 = w3 + w4
15 w5 = 2 · w5 + w6
16 w2 = w2 - 65 · w3
17 w3 = w3 - w7 - w1
18 w2 = w2 + 45 · w3
19 w5 = (w5 - 8 · w3)/24
20 w6 = w6 + w2
21 w2 = (w2 + 16 · w4)/18
22 w3 = w3 - w5
23 w4 = -(w4 + w2)
24 w6 = (30 · w2 - w6)/60
25 w2 = w2 - w6
26 return C(y) = w1 · y6 + w2 · y5 + w3 · y4 + w4 · y3 + w5 · y2 + w6 · y + w7;
```

```
// 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 = r1 + r4;
    r5 = r5 - r4;
    r3 = ((r3 - r2) >> 1);
    r4 = r4 - r0;
    r4 = r4 - (r6 << 6);
    r4 = (r4 << 1) + r5;
    r2 = r2 + r3;
    r1 = r1 - (r2 << 6) - r2;
    r2 = r2 - r6;
    r2 = r2 - r0;
    r1 = r1 + 45 * r2;
    r4 = (uint16_t)((r4 - (r2 << 3)) * (uint32_t)inv3) >> 3;
    r5 = r5 + r1;
    r1 = (uint16_t)((r1 + (r3 << 4)) * (uint32_t)inv9) >> 1;
    r3 = -(r3 + r1);
    r5 = (uint16_t)((30 * r1 - r5) * (uint32_t)inv15) >> 2;
    r2 = r2 - r4;
    r1 = r1 - r5;
```

```
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;
}
```

✓ 1/2, 1/24, 1/18, 1/60에 대해서는 사전 계산된 값 이용
 → 1/24 = 1/3 * 1/8, 1/18 = 1/9*1/2, 1/60 = 1/15*1/4

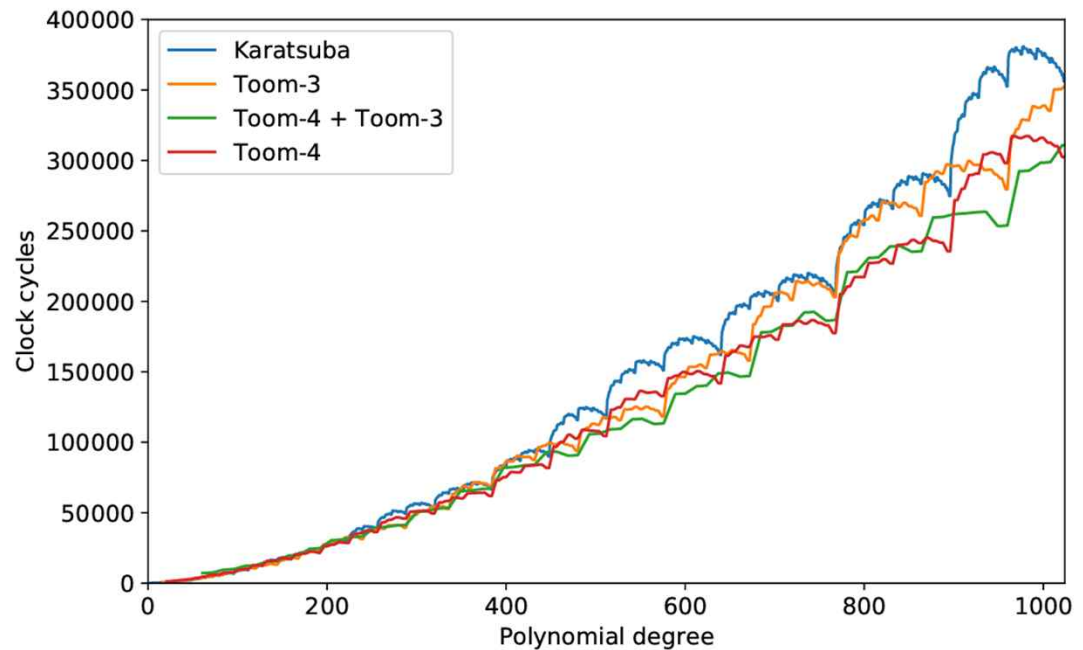
LBC 다항식 곱셈 방법



❖다항식 곱셈 연산 속도 비교 ($Rq = Z_q[X]/(X^n + 1), q = Z_{2^m}[X]$)

○Schoolbook, Karatsuba, Toom-Cook의 조합에 대한 연산 성능 Benchmarking (다항식 감산 제외)

	approach	schoolbook	clock cycles	stack usage [bytes]
Saber ($n = 256, q = 2^{13}$)	Karatsuba only	16	38 000	2 020
	Toom-3	11	39 043	3 480
	Toom-4	16	36 274	3 800
	Toom-4 + Toom-3	-	-	-
Kindi-256-3-4-2 ($n = 256, q = 2^{14}$)	Karatsuba only	16	38 000	2 020
	Toom-3	11	39 043	3 480
	Toom-4	-	-	-
	Toom-4 + Toom-3	-	-	-
NTRU-HRSS ($n = 701, q = 2^{13}$)	Karatsuba only	11	202 889	5 676
	Toom-3	15	205 947	9 384
	Toom-4	11	172 882	10 596
	Toom-4 + Toom-3	-	-	-
NTRU-KEM-743 ($n = 743, q = 2^{11}$)	Karatsuba only	12	217 130	6 012
	Toom-3	16	211 588	9 920
	Toom-4	12	186 639	11 208
	Toom-4 + Toom-3	16	192 503	12 152
RLizard-1024 ($n = 1024, q = 2^{11}$)	Karatsuba only	16	356 046	8 188
	Toom-3	11	352 770	13 756
	Toom-4	16	302 504	15 344
	Toom-4 + Toom-3	11	310 712	16 816



참고문헌: Faster multiplication in $Z_{2^m}[X]$ on Cortex-M4 to speed up NIST PQC candidate, eprint2018

❖ Number Theoretic Transform (NTT)

○ 정수 상에서의 FFT 변환

○ 조건

- 적절한 modulus q 를 선택해 n -th root of unity 가 \mathbb{Z}_q 에 존재하도록
- $q \equiv 1 \pmod{2n}$
 - $x^{2n} \equiv 1 \pmod{q}$ 를 만족하는 모든 정수가 \mathbb{Z}_q 안에 있음 (primitive $2n$ -th root of unity)

○ 연산량: $O(n \log_2 n)$

○ 절차: NTT 변환 ($O(n \log_2 n)$) \rightarrow pointwise 곱셈 ($O(n)$) \rightarrow INVNTT 변환 ($O(n \log_2 n)$)

- NTT 변환은 다항식을 차수가 1 또는 2인 subring들의 나머지로 표현 (CRT 개념 적용)
$$R[x]/(x^n - 1) \rightarrow (R[x]/(x - 1)) \times (R[x]/(x - \zeta)) \times \cdots \times (R[x]/(x - \zeta^{n-1}))$$
- Pointwise 곱셈은 동일한 subring의 나머지 값끼리 곱하는 연산
- INVNTT 변환 subring의 나머지 값들로부터 원본 ring의 원소로 복원

❖ Complete NTT vs Incomplete NTT

○Dilithium Case

- $q = 8380417$, primitive $2n$ -th root of unity 존재 → **Complete** NTT (8-Layer)

○Kyber Case

- $q = 3329$, primitive n -th root of unity 존재 → **Incomplete** NTT (7-Layer)

다항식 차수		다항식 개수
1	256	1
2	128	2
3	64	4
4	32	8
5	16	16
6	8	32
7	4	64
8	2	128
9	1	256

Kyber

Dilithium

[Ring 분할 \rightarrow Twiddle factors 사전 계산]

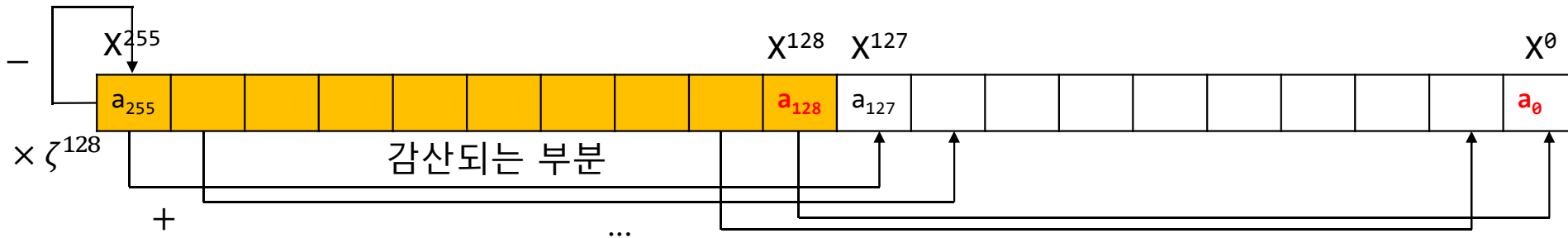
$$(X^{256} + 1) = (X - \zeta)(X + \zeta)(X - \zeta^{129}) \cdots (X - \zeta^{127})(X + \zeta^{127})(X - \zeta^{255})(X + \zeta^{255})$$

❖ Dilithium Case ($q = 8380417, \zeta = 1753$)

[NTT 변환: $O(n \log_2 n)$]

$$\mathbf{a} \in \mathbb{Z}_q[X]/(X^{256} + 1)$$

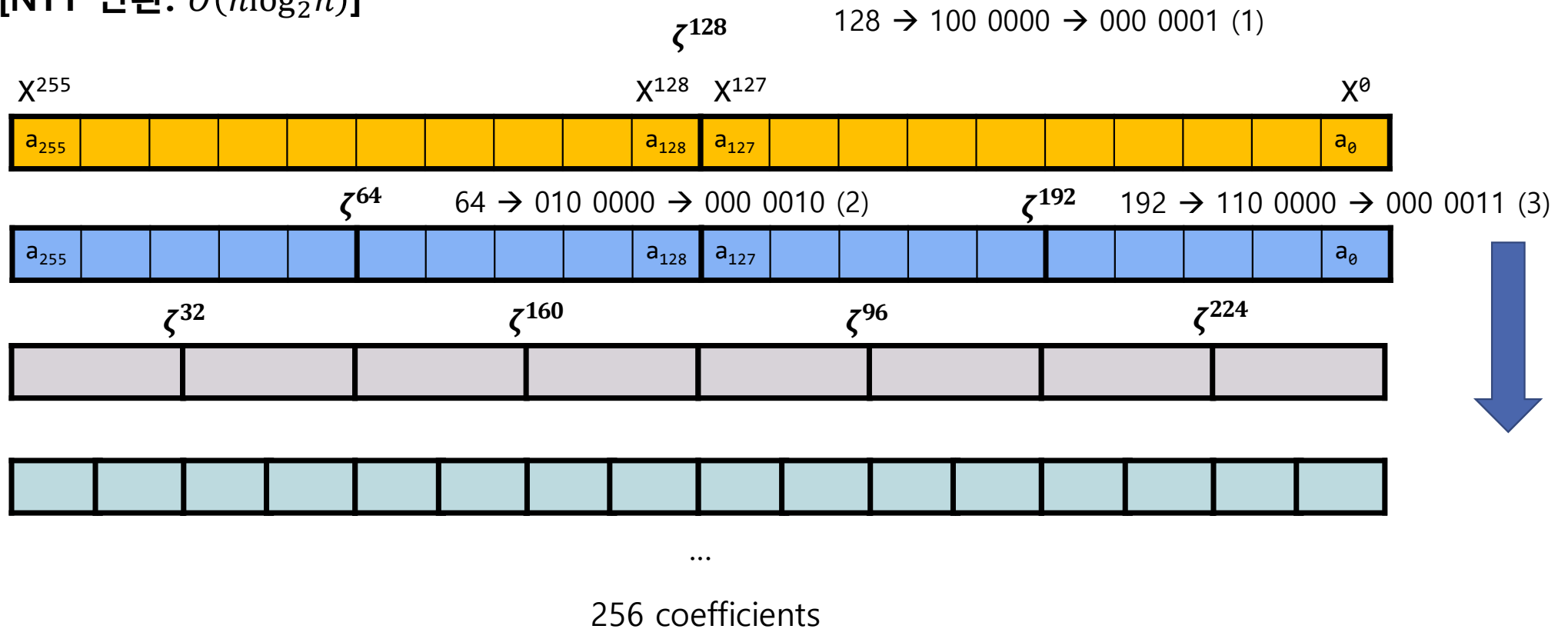
$$\rightarrow \mathbf{a}_L^{(1)} \in \mathbb{Z}_q[X]/(X^{128} - \zeta^{128}), \mathbf{a}_R^{(1)} \in \mathbb{Z}_q[X]/(X^{128} + \zeta^{128})$$



$$\mathbf{a}_L^{(1)} = (a_0 + \zeta^{128} a_{128}) + (a_1 + \zeta^{128} a_{129})X + \cdots + (a_{127} + \zeta^{128} a_{255})X^{127},$$

$$\mathbf{a}_R^{(1)} = (a_0 - \zeta^{128} a_{128}) + (a_1 - \zeta^{128} a_{129})X + \cdots + (a_{127} - \zeta^{128} a_{255})X^{127}$$

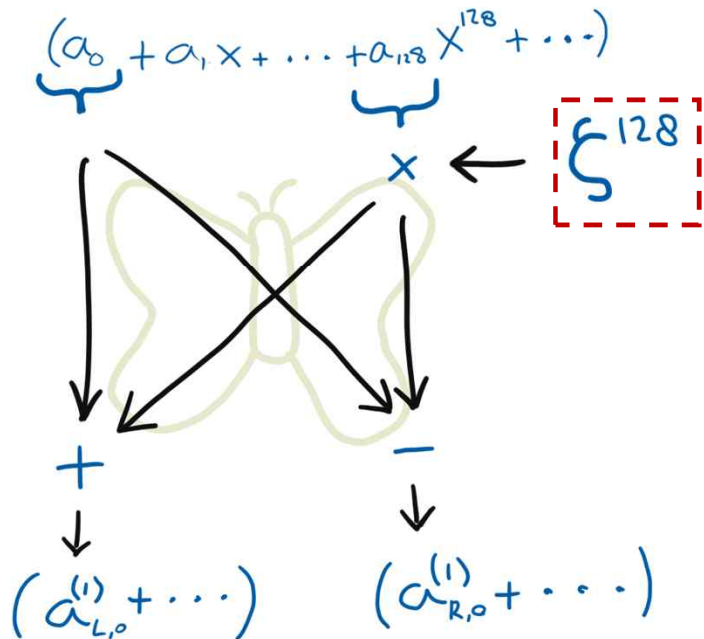
[NTT 변환: $O(n \log_2 n)$]



❖ Dilithium Case ($q = 8380417, \zeta = 1753$)

[NTT 변환: $O(n \log_2 n)$]

$$\begin{aligned} \mathbf{a}_L^{(1)} &= (a_0 + \zeta^{128} a_{128}) + (a_1 + \zeta^{128} a_{129})X + \cdots + (a_{127} + \zeta^{128} a_{255})X^{127}, \\ \mathbf{a}_R^{(1)} &= (a_0 - \zeta^{128} a_{128}) + (a_1 - \zeta^{128} a_{129})X + \cdots + (a_{127} - \zeta^{128} a_{255})X^{127} \end{aligned}$$



✓ ζ 는 사전에 계산되어 저장된 값 (Twiddle factor)

✓ ζ 와 곱하는 부분은 \mathbb{Z}_q 상에서의 곱셈 (곱셈 결과 감산 필요)

- Montgomery 기반 곱셈 적용: $\text{Mont}(a_i, \zeta) = a_i \cdot \zeta \cdot R^{-1} \bmod q$

- ζ 는 원본값에 R 을 곱하고 $\bmod q$ 를 한 값을 저장함 ($\zeta' = \zeta R \bmod q$)

LBC 다항식 곱셈 방법



❖ Dilithium Case ($q = 8380417$, $\zeta = 1753$)

[점별 곱셈: $O(n)$]

$$\overline{A(x)} \left\{ \begin{array}{l} \widetilde{a_0} \in Z_q[x]/(x - \xi), \\ \widetilde{a_1} \in Z_q[x]/(x + \xi), \\ \widetilde{a_2} \in Z_q[x]/(x - \xi^{129}), \\ \widetilde{a_3} \in Z_q[x]/(x + \xi^{129}), \\ \dots \\ \widetilde{a_{254}} \in Z_q[x]/(x - \xi^{255}) \\ \widetilde{a_{255}} \in Z_q[x]/(x + \xi^{255}) \end{array} \right. \quad \overline{B(x)} \left\{ \begin{array}{l} \widetilde{b_0} \in Z_q[x]/(x - \xi), \\ \widetilde{b_1} \in Z_q[x]/(x + \xi), \\ \widetilde{b_2} \in Z_q[x]/(x - \xi^{129}), \\ \widetilde{b_3} \in Z_q[x]/(x + \xi^{129}), \\ \dots \\ \widetilde{b_{254}} \in Z_q[x]/(x - \xi^{255}) \\ \widetilde{b_{255}} \in Z_q[x]/(x + \xi^{255}) \end{array} \right.$$

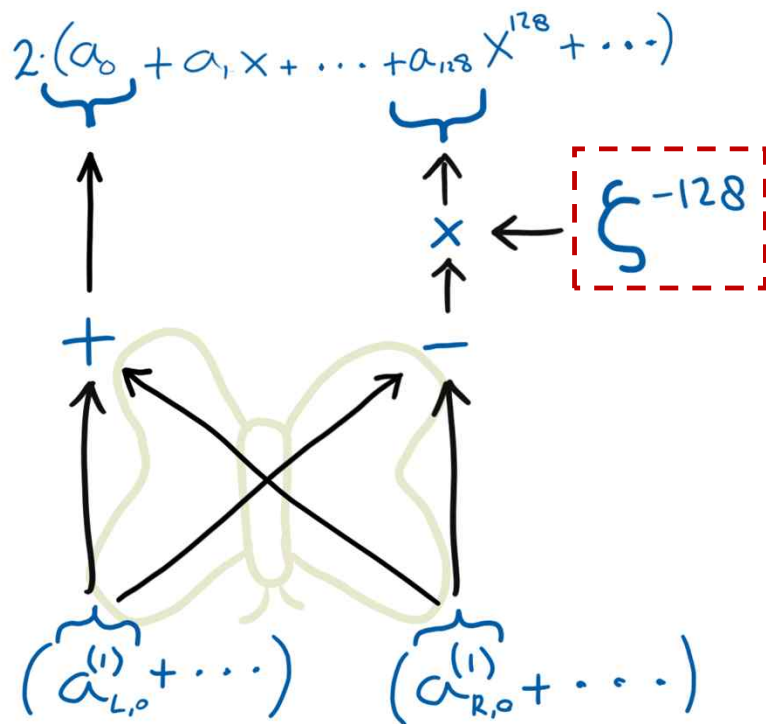
$$\overline{A(x)} \times \overline{B(x)} = \overline{C(x)} \left\{ \begin{array}{l} \widetilde{a_0} \times \widetilde{b_0} \in Z_q[x]/(x - \xi), \\ \widetilde{a_1} \times \widetilde{b_1} \in Z_q[x]/(x + \xi), \\ \widetilde{a_2} \times \widetilde{b_2} \in Z_q[x]/(x - \xi^{129}), \\ \widetilde{a_3} \times \widetilde{b_3} \in Z_q[x]/(x + \xi^{129}), \\ \dots \\ \widetilde{a_{254}} \times \widetilde{b_{254}} \in Z_q[x]/(x - \xi^{255}) \\ \widetilde{a_{255}} \times \widetilde{b_{255}} \in Z_q[x]/(x + \xi^{255}) \end{array} \right. \quad \longrightarrow \quad \widetilde{a_i} \times \widetilde{b_i} \text{ 각 곱셈 결과는 } q \text{ 로 감산}$$

LBC 다항식 곱셈 방법



❖ Dilithium Case ($q = 8380417$, $\zeta = 1753$)

[INVNTT: $O(n \log_2 n)$]



$$a_{L,0}^{(1)} = a_0 + \zeta^{128} a_{128}$$

$$a_{R,0}^{(1)} = a_0 - \zeta^{128} a_{128}$$

$$a_{L,0}^{(1)} + a_{R,0}^{(1)} = a_0 + \cancel{\zeta^{128} a_{128}} + a_0 - \cancel{\zeta^{128} a_{128}}$$

$$2a_0 = a_{L,0}^{(1)} + a_{R,0}^{(1)}$$

$$a_0 = 2^{-1} (a_{L,0}^{(1)} + a_{R,0}^{(1)})$$

$$a_{L,0}^{(1)} - a_{R,0}^{(1)} = \cancel{a_0} + \zeta^{128} a_{128} - \cancel{a_0} + \zeta^{128} a_{128}$$

$$2\zeta^{128} a_{128} = a_{L,0}^{(1)} - a_{R,0}^{(1)}$$

$$a_{128} = \boxed{2^{-1} \zeta^{-128}} (a_{L,0}^{(1)} - a_{R,0}^{(1)})$$

LBC 다항식 곱셈 방법



❖Dilithium Case ($q = 8380417$, $\zeta = 1753$)

```
void ntt(int32_t a[N]) { 경과 시간 1ms 이하
    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 = montgomery_reduce((int64_t)zeta * a[j + len]);
                a[j + len] = a[j] - t;
                a[j] = a[j] + t;
            }
        }
    }
}
```

```
void invntt_tomont(int32_t a[N]) { 경과 시간 1ms 이하
    unsigned int start, len, j, k;
    int32_t t, zeta;
    const int32_t f = 41978; // mont^2/256
    -----

    k = 256;
    for(len = 1; len < N; len <= 1) {
        for(start = 0; start < N; start = j + len) {
            zeta = -zetas[--k];
            for(j = start; j < start + len; ++j) {
                t = a[j];
                a[j] = t + a[j + len];
                a[j + len] = t - a[j + len];
                a[j + len] = montgomery_reduce((int64_t)zeta * a[j + len]);
            }
        }
    }

    for(j = 0; j < N; ++j) {
        a[j] = montgomery_reduce((int64_t)f * a[j]);
    }
}
```

❖ Kyber Case ($q = 3329, \zeta = 17$)

○ Layer 7까지만 NTT 변환 적용 가능

→ 최종 변환 결과는 128개의 1차 다항식

○ 점별 곱셈은 1차 다항식 곱셈을 128번 수행

$$- NTT(A(x)) = \widetilde{A}(x) = (\widetilde{a}_0 + \widetilde{a}_1 X, \widetilde{a}_2 + \widetilde{a}_3 X, \dots, \widetilde{a}_{254} + \widetilde{a}_{255} X)$$

$$- NTT(B(x)) = \widetilde{B}(x) = (\widetilde{b}_0 + \widetilde{b}_1 X, \widetilde{b}_2 + \widetilde{b}_3 X, \dots, \widetilde{b}_{254} + \widetilde{b}_{255} X)$$

$$\widetilde{h}_{2i} + \widetilde{h}_{2i+1} X = (\widetilde{a}_{2i} + \widetilde{a}_{2i+1} X)(\widetilde{b}_{2i} + \widetilde{b}_{2i+1} X) \bmod X^2 - \zeta^{2br_{7(i)}+1}$$

$$\rightarrow \widetilde{h}_{2i} \leftarrow \widetilde{a}_{2i} \cdot \widetilde{b}_{2i} + \widetilde{a}_{2i+1} \cdot \widetilde{b}_{2i+1} \cdot \zeta^{2br_{7(i)}+1}$$

$$\rightarrow \widetilde{h}_{2i+1} \leftarrow \widetilde{a}_{2i+1} \cdot \widetilde{b}_{2i} + \widetilde{a}_{2i} \cdot \widetilde{b}_{2i+1}$$

❖(행렬 \times 벡터)에서의 NTT 변환 횟수

○Dilithium Case

- Ay where $A \in R^{k \times l}$ and $y \in R^l$
- A 는 NTT domain으로 sampling했다고 가정
- Ay 는 l NTT + k INVNTT 요구

○Kyber Case

- As where $A \in R^{k \times k}$ and $s \in R^k$
- A 는 NTT domain으로 sampling했다고 가정
- As 는 k NTT + k INVNTT 요구

❖ SABER에서 NTT 적용

○ NTT 사용 조건을 만족하는 $q' (> q = 2^{13})$ 를 이용하여 다항식 곱셈 연산 수행 $\rightarrow \text{mod } q$ 연산 수행

- 동형암호에서 사용되는 modulus switching 테크닉의 응용
- q' 의 범위를 효율적으로 잡는 것이 중요함
 $\rightarrow q$ 상에서의 곱셈+누적 연산을 분석하여 발생할 수 있는 최대값 분석을 통해 q' 설정 필요
- Incomplete NTT 적용 ($n \mid q' - 1$)

▪ We compute $A^T \cdot s$ as if \mathbb{Z} is the coefficient ring

▪ Bounding the maximum value of the result:

- Max: $2 \cdot 256 \cdot \frac{8192}{2} \cdot \frac{\mu}{2} \cdot l = 2^{20} \cdot \mu \cdot l$
- Cortex-M4: choose a 32-bit prime $q' > 2^{20} \cdot \mu \cdot l$
- AVX2: choose two 16-bit primes p_0, p_1 with $p_0 p_1 > 2^{20} \cdot \mu \cdot l$

[Cortex-M4]

- Saber, Firesaber: $q' = 25166081 = 196610 \cdot 128 + 1$
- Lightsaber: $20972417 = 163841 \cdot 128 + 1$

[AVX2]

- 16-bit 단위의 Montgomery 감산이 더욱 효율적임
- 7681, 10753 for p_0 and p_1

참고문헌: NTT multiplication for NTT unfriendly rings, TCHES 2021

Time-memory Trade-offs for SABER+ on Memory-constrained RISC-V platform, ToC

LBC 다항식 곱셈 방법



❖ SABER에서 NTT 적용

○ SABER 성능향상

- Cortex-M4 상에서 (행렬 X 벡터) 연산의 성능이 TC 대비 최대 61% 향상
- AVX2 상에서 최대 39% 향상

○ SABER에서의 NTT 연산 횟수

- $(k^2 + k)$ NTT + k INVNTT 요구
(A 를 NTT domain으로 샘플링하지 못하기 때문)

Table 1: Cycles for MatrixVectorMul and InnerProd in Saber.

MatrixVectorMul					
	Cortex-M4		Skylake (AVX2)		
	[BMKV20]	Our Work	[BMKV20]	Our Work	
$l = 2$	159k	66k (− 58%)	7 002	5 215 (−25%)	
$l = 3$	317k	125k (− 61%)	14 145	9 579 (−32%)	
$l = 4$	528k	205k (− 61%)	24 342	14 959 (−39%)	
InnerProduct					
	Cortex-M4		Skylake (AVX2)		
	[BMKV20]	Our Work	[BMKV20]	Our Work	
$l = 2$	73k	41k (− 44%)	4 016	2 125 (−47%)	
$l = 3$	99k	57k (− 42%)	5 977	2 706 (−55%)	
$l = 4$	126k	73k (− 42%)	8 040	3 278 (−60%)	

Table 2: Clock cycles for Lightsaber, Saber, and Firesaber.

		Cortex-M4		Skylake (AVX2)	
		[BMKV20]	Our Work	[BMKV20]	Our Work
Lightsaber	K	466k	360k (−23%)	61 325	59 831 (−2%)
	E	653k	513k (−21%)	75 876	72 473 (−4%)
	D	678k	498k (−27%)	70 228	64 859 (−8%)
Saber	K	853k	658 (−23%)	104 832	99 71 (−5%)
	E	1 103k	864 (−22%)	125 835	118 44 (−6%)
	D	1 127k	835 (−26%)	118 553	107 26 (−10%)
Firesaber	K	1 340k	1 008k (−25%)	157 915	148 729 (−6%)
	E	1 642k	1 255k (−24%)	184 322	171 993 (−7%)
	D	1 679k	1 227k (−27%)	177 864	159 950 (−10%)

참고문헌: NTT multiplication for NTT unfriendly rings, TCHES 2021

Time-memory Trade-offs for SABER+ on Memory-constrained RISC-V platform, ToC

LBC 다항식 곱셈 방법



❖구현 방법에 따른 메모리 사용량 측정 방법

○정적 메모리 체크

- 컴파일 타임에 크기가 결정되는 경우
- 범용환경: gcc 컴파일 옵션에 -fstack-usage 옵션 추가
 - 소스코드 파일별로 확장자가 su인 파일 생성 (함수별 정적 스택 사용량 표시)

```
CC=/usr/bin/gcc
CFLAGS += -O0 -g -march=native -fomit-frame-pointer -fstack-usage
LDFLAGS=-lcrypto

SOURCES= cbd.c fips202.c indcpa.c kem.c ntt.c poly.c polyvec.c PQCGenKAT_kem.c reduce.c rng.c verify.c symmetric-shake.c
HEADERS= api.h cbd.h fips202.h indcpa.h ntt.h params.h poly.h polyvec.h reduce.h rng.h verify.h symmetric.h

PQCGenKAT_kem: $(HEADERS) $(SOURCES)
$(CC) $(CFLAGS) -o $@ $(SOURCES) $(LDFLAGS)
```

Kyber768의 Makefile

```
fips202.su x indcpa.su x kem.su x ntt.su x poly.su x polyvec.su x PQCGenKAT_kem.su x reduce.su x
1 fips202.c:24:17:load64 8 static
2 fips202.c:42:13:store64 8 static
3 fips202.c:84:13:KeccakF1600_StatePermute 392 static
4 fips202.c:362:13:keccak_absorb 272 static
5 fips202.c:404:13:keccak_squeezeblocks 56 static
6 fips202.c:430:6:pqcrystals_fips202_ref_shake128_absorb 48 static
7 fips202.c:447:6:pqcrystals_fips202_ref_shake128_squeezeblocks 32 static
8 fips202.c:462:6:pqcrystals_fips202_ref_shake256_absorb 48 static
9 fips202.c:479:6:pqcrystals_fips202_ref_shake256_squeezeblocks 32 static
10 fips202.c:494:6:pqcrystals_fips202_ref_shake128 448 static
11 fips202.c:524:6:pqcrystals_fips202_ref_shake256 416 static
12 fips202.c:553:6:pqcrystals_fips202_ref_sha3 256 416 static
13 fips202.c:575:6:pqcrystals_fips202_ref_sha3 512 352 static
```

```
indcpa.su x kem.su x ntt.su x poly.su x polyvec.su x PQCGenKAT_kem.su x reduce.su x rng.su
1 indcpa.c:22:13:pack_pk 64 static
2 indcpa.c:44:13:unpack_pk 64 static
3 indcpa.c:62:13:pack_sk 32 static
4 indcpa.c:77:13:unpack_sk 32 static
5 indcpa.c:94:13:pack_ciphertext 48 static
6 indcpa.c:112:13:unpack_ciphertext 48 static
7 indcpa.c:135:21:rej_uniform 8 static
8 indcpa.c:177:6:pqcrystals_kyber768_ref_gen_matrix 816 static
9 indcpa.c:218:6:pqcrystals_kyber768_ref_indcpa_keypair 9360 static
10 indcpa.c:271:6:pqcrystals_kyber768_ref_indcpa_enc 12400 static
11 indcpa.c:325:6:pqcrystals_kyber768_ref_indcpa_dec 4160 static
12
```

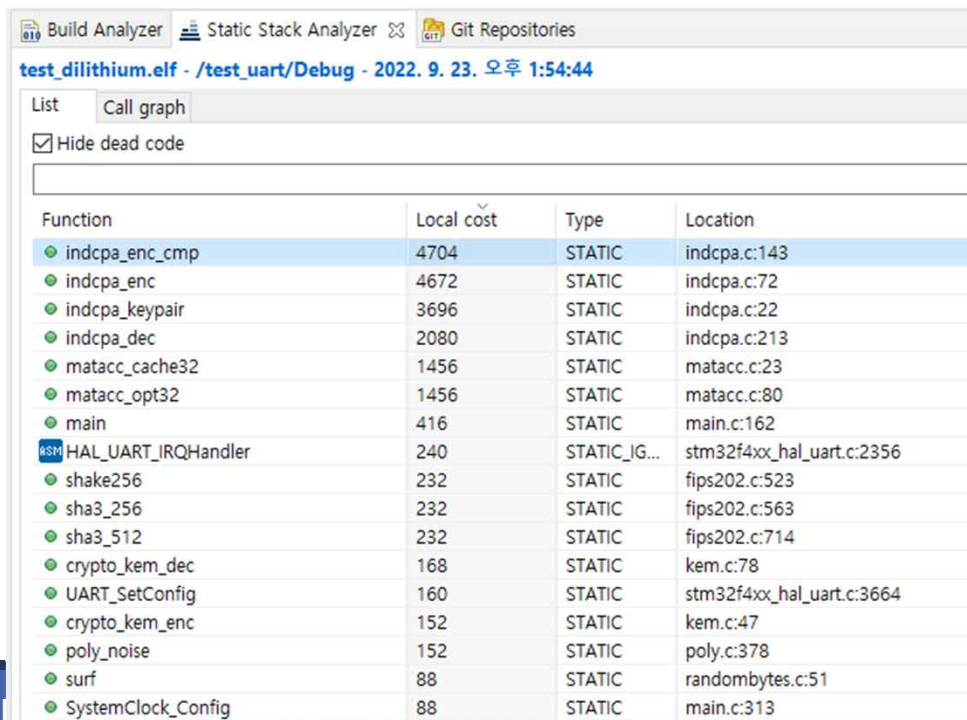
LBC 다항식 곱셈 방법



❖구현 방법에 따른 메모리 사용량 측정 방법

○정적 메모리 체크

- 컴파일 타임에 크기가 결정되는 경우
- 32-bit Cortex-M4 환경: STM32CubeIDE 상에서 빌드 후 확인 가능



Function	Local cost	Type	Location
indcpa_enc_cmp	4704	STATIC	indcpa.c:143
indcpa_enc	4672	STATIC	indcpa.c:72
indcpa_keypair	3696	STATIC	indcpa.c:22
indcpa_dec	2080	STATIC	indcpa.c:213
matacc_cache32	1456	STATIC	matacc.c:23
matacc_opt32	1456	STATIC	matacc.c:80
main	416	STATIC	main.c:162
HAL_UART_IRQHandler	240	STATIC_IG...	stm32f4xx_hal_uart.c:2356
shake256	232	STATIC	fips202.c:523
sha3_256	232	STATIC	fips202.c:563
sha3_512	232	STATIC	fips202.c:714
crypto_kem_dec	168	STATIC	kem.c:78
UART_SetConfig	160	STATIC	stm32f4xx_hal_uart.c:3664
crypto_kem_enc	152	STATIC	kem.c:47
poly_noise	152	STATIC	poly.c:378
surf	88	STATIC	randombytes.c:51
SystemClock_Config	88	STATIC	main.c:313

LBC 다항식 곱셈 방법



❖구현 방법에 따른 메모리 사용량 측정 방법

○동적 메모리 체크

- 프로그램이 실행되면서 사용되는 메모리 측정 (힙, 스택 메모리)
- 운영환경 마다 차이 존재
 - 범용환경: Valgrind의 massif 프로파일러 사용 가능
 - Visual Studio Enterprise: CPU, 메모리에 대한 프로파일링 기능 제공

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
73	792,744,596	16,224	9,296	48	6,880
74	801,461,992	16,232	9,296	48	6,888
75	810,180,722	15,272	9,296	48	5,928
76	818,898,138	24,584	9,296	48	15,240
77	827,615,569	24,088	9,296	48	14,744
78	836,335,156	24,752	9,296	48	15,408
79	845,053,474	14,600	9,296	48	5,256
80	853,770,874	28,416	9,296	48	19,072
81	862,489,360	28,832	9,296	48	19,488
82	871,206,777	28,392	9,296	48	19,048

```
32.74% (9,296B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->28.85% (8,192B) 0x53E11E3: _IO_file_doallocate (filedoalloc.c:127)
->28.85% (8,192B) 0x53EF5A2: _IO_doallocbuf (genops.c:398)
->14.43% (4,096B) 0x53EE906: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:820)
->14.43% (4,096B) 0x53ED298: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1331)
->14.43% (4,096B) 0x53C124F: vfprintf (vfprintf.c:1320)
->14.43% (4,096B) 0x53C9805: fprintf (fprintf.c:32)
->14.43% (4,096B) 0x404CA0: main (PQCgenKAT_kem.c:71)
```

- valgrind -tool=massif --stacks=yes ./PQCgenKAT_kem
- ms_print massif.out.4487
- 성능에 핵심인 함수 만을 프로파일링 수행
(예: Karatsuba recursive VS iterative)

LBC 다항식 곱셈 방법



❖ 구현 방법에 따른 메모리 사용량 측정 방법

○ 동적 메모리 체크

▪ SABER의 TC 기반 다항식 곱셈 (TC-4 + Karatsuba)

```
gegehe@ubuntu:~/Programming/test_mem/SABER$ gcc -o saber -g -fstack-usage main.c poly_mul.c -I./
gegehe@ubuntu:~/Programming/test_mem/SABER$ ls
main.c main.su massif.out.8776 poly_mul.c poly_mul.h poly_mul.su saber SABER_params.h
gegehe@ubuntu:~/Programming/test_mem/SABER$ valgrind --tool=massif --stacks=yes ./saber
==8849== Massif, a heap profiler
==8849== Copyright (C) 2003-2015, and GNU GPL'd, by Nicholas Nethercote
==8849== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==8849== Command: ./saber
==8849==
gegehe@ubuntu:~/Programming/test_mem/SABER$ ls
main.c main.su massif.out.8776 massif.out.8849 poly_mul.c poly_mul.h poly_mul.su saber SABER_params.h
gegehe@ubuntu:~/Programming/test_mem/SABER$ ms_print massif.out.8849
```

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
73	222,621	6,688	0	0	6,688
74	284,757	6,688	0	0	6,688
75	346,893	6,688	0	0	6,688
76	409,029	6,688	0	0	6,688
77	471,165	6,688	0	0	6,688
78	533,301	6,688	0	0	6,688
79	595,437	6,688	0	0	6,688
80	624,022	2,928	0	0	2,928
81	630,944	1,824	0	0	1,824
82	632,468	1,848	0	0	1,848

00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#include "SABER_params.h"
#include "poly_mul.h"

void test1()
{
    int cnt_i;
    uint16_t poly1[SABER_N] = {0x00, };
    uint16_t poly2[SABER_N] = {0x00, };
    uint16_t ret[SABER_N] = {0x00, };

    srand(time(NULL));
    for(cnt_i = 0; cnt_i < SABER_N; cnt_i++)
    {
        poly1[cnt_i] = rand() & 0x1fff;
        poly2[cnt_i] = rand() & 0x1fff;
    }

    poly_mul_acc(poly1, poly2, ret);
}
```

```
main.su ✕ poly_mul.su ✕
1 main.c:8:6:test1 1584 static
2 main.c:25:5:main 16 static
```

```
main.su ✕ poly_mul.su ✕
1 poly_mul.c:14:13:karatsuba_simple 416 static
2 poly_mul.c:113:13:toom_cook_4way 3760 static
3 poly_mul.c:232:6:poly_mul_acc 1104 static
```

LBC 다항식 곱셈 방법



❖ 구현 방법에 따른 메모리 사용량 측정 방법

○ 동적 메모리 체크

▪ Kyber의 NTT 기반 다항식 곱셈

```
gegehe@ubuntu:~/Programming/test_mem/KYBER$ gcc -o kyber -g -fstack-usage main.c poly.c ntt.c
reduce.c
gegehe@ubuntu:~/Programming/test_mem/KYBER$ valgrind --tool=massif --stacks=yes ./kyber
==9141== Massif, a heap profiler
==9141== Copyright (C) 2003-2015, and GNU GPL'd, by Nicholas Nethercote
==9141== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==9141== Command: ./kyber
==9141==
gegehe@ubuntu:~/Programming/test_mem/KYBER$ ls
kyber  main.su      ntt.c  ntt.su  poly.c  poly.su  reduce.h
main.c  massif.out.9141  ntt.h  params.h  poly.h  reduce.c  reduce.su
gegehe@ubuntu:~/Programming/test_mem/KYBER$ ms_print massif.out.9141
```

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
74	412,537	1,936	0	0	1,936
75	416,698	1,928	0	0	1,928
76	420,858	1,968	0	0	1,968
77	425,038	1,936	0	0	1,936
78	429,203	1,936	0	0	1,936
79	433,388	1,936	0	0	1,936
80	437,553	1,936	0	0	1,936
81	441,745	1,936	0	0	1,936
82	445,910	1,936	0	0	1,936
83	450,075	1,936	0	0	1,936

00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>
#include "params.h"
#include "poly.h"

void test1()
{
    int cnt_i;
    poly poly1 = {0x00, };
    poly poly2 = {0x00, };
    poly ret = {0x00, };

    srand(time(NULL));
    for(cnt_i = 0; cnt_i < KYBER_N; cnt_i++)
    {
        poly1.coeffs[cnt_i] = rand() & 0xffff;
        poly2.coeffs[cnt_i] = rand() & 0xffff;
    }

    poly_ntt(&poly1);
    poly_ntt(&poly2);
    poly_basemul_montgomery(&ret, &poly1, &poly2);
    poly_invntt_tomont(&ret);
}
```

LBC 다항식 곱셈 방법



❖구현 방법에 따른 메모리 사용량 측정 방법

○동적 메모리 체크

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
35	78,831	1,352	0	0	1,352
36	80,867	1,352	0	0	1,352
37	82,203	1,352	0	0	1,352
38	84,737	1,448	0	0	1,448
39	86,133	1,352	0	0	1,352
40	88,107	1,096	0	0	1,096
41	90,711	1,152	0	0	1,152
42	92,544	1,192	0	0	1,192
43	95,146	1,192	0	0	1,192
44	97,034	824	0	0	824
45	98,224	1,440	0	0	1,440
46	100,675	272	0	0	272
47	102,526	1,528	0	0	1,528

00.00% (0B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

main.c:5:5:fibonacci	64	statics
main.c:19:5:main	48	static

```
#include <stdio.h>
#include <stdlib.h>

int fibonacci(int num)
{
    int temp;
    if(num == 1){
        return 0;
    }else if(num == 2){
        return 1;
    }

    temp = fibonacci(num - 1) + fibonacci(num - 2);
    return temp;
}

int main(int argc, char* argv[])
{
    int num = atoi(argv[1]);
    printf("total: %d\n", fibonacci(num));

    return 0;
}
```


❖ 곱셈/제곱 연산에 대한 감산 방법

○ Montgomery 감산

- RSA, ECC 구현에서 많이 사용됨 (Standard 표현 이용 $[0, q - 1]$)
- 나눗셈 연산을 시프트 연산으로 대체
- Montgomery 도메인으로의 변환이 필요함
 - $a \cdot b \bmod q \rightarrow a \cdot b \cdot R^{-1} \bmod q$ where $R > q$ and $R = 2^n$
 - To Mont: $Mont(T, R^2) = (TR^2)R^{-1} \bmod q = TR \bmod q$
 - From Mont: $Mont(TR, 1) = (TR)R^{-1} \bmod q = T \bmod q$
- Final reduction 과정이 존재
 - 감산된 중간값 T의 범위가 $0 < T < 2q$
- NIST 격자기반암호에서는 Centered modulo 표현을 이용함 $([-\frac{q-1}{2}, \frac{q-1}{2}])$

❖ 곱셈/제곱 연산에 대한 감산 방법 (Montgomery 감산)

Mont(\tilde{A}, \tilde{B})

INPUT: integer $q = (q_{n-1}, \dots, q_1, q_0), T < qR$, with $\gcd(q, R) = 1, R = b^n, q' = -q^{-1} \bmod R$

OUTPUT: $TR^{-1} \bmod q$

1. $U = Tq' \bmod R$

2. Uq

3. $T + Uq$ // $(T + Uq) = (T + (Tq' + kR)q)$

4. $(T + Uq)/R$ // $\frac{(T + (Tq' + kR)q)}{R} = \frac{TR + kRq}{R} = (T + kq)$ where $q' \times q = (-1 + R)$

5. *if* $T \geq q$ *then* $T \leftarrow T - q$ // $0 \leq \frac{T + Uq}{R} < \frac{qR + qR}{R} < 2q$

6. *return* (T)

❖ 곱셈/제곱 연산에 대한 감산 방법 (Short Montgomery 감산)

○ Standard 표현 사용으로 인해 비효율 발생

Mont(\tilde{A}, \tilde{B})

INPUT : $q = (q_{n-1}, \dots, q_1, q_0), T = t_1R + t_0 < qR$, with $\gcd(q, R) = 1, R = 2^l (l = 16, 32), q' = -q^{-1} \bmod R$

OUTPUT : $TR^{-1} \bmod q$

1. $U = -U' \bmod R$ // $U' = t_0(q^{-1}) \bmod R, U = R - U'$ (2-words)
2. Uq // (1-word) * (1-word)
3. $T + Uq$ // double-word addition
4. $(T + Uq)/R$ // right shift
5. **if** $T \geq q$ **then** $T \leftarrow T - q$ // $0 \leq \frac{T+Uq}{R} < \frac{qR+qR}{R} < 2q$
6. **return** (T)

LBC 정수 연산 감산 방법



❖ 곱셈/제곱 연산에 대한 감산 방법 (Signed Short Montgomery 감산)

○ $q' = q^{-1} \bmod \pm R$ 을 이용하여 덧셈이 아닌 뺄셈 수행 → signed 사용

○ Signed 자료형 사용으로 인해 Lazy reduction 쉽게 적용 가능 + 연산 이득 존재

Mont(\tilde{A}, \tilde{B})

INPUT: $0 < q < \frac{R}{2}$ odd, $-\frac{R}{2}q \leq T = t_1R + t_0 < \frac{R}{2}q$, where $0 \leq t_0 < R$, $q' = q^{-1} \bmod \pm R$

OUTPUT: $r' \equiv TR^{-1} \bmod q, -q < r' < q$

1. $U = t_0q' \bmod \pm R$

2. $v_1 \leftarrow \left\lfloor \frac{Uq}{R} \right\rfloor$ // (1-word) * (1-word)

3. $r' \leftarrow t_1 - v_1$ // 1-word subtraction, $-q < \frac{T-Uq}{R} = \frac{(t_1-v_1)R+(t_0-v_0)}{R} = (t_1 - v_1) < q$

4. ***return*** (r') • 참고문헌: Faster AVX2 Optimized NTT Multiplication for Ring-LWE Lattice Cryptography, eprint2018

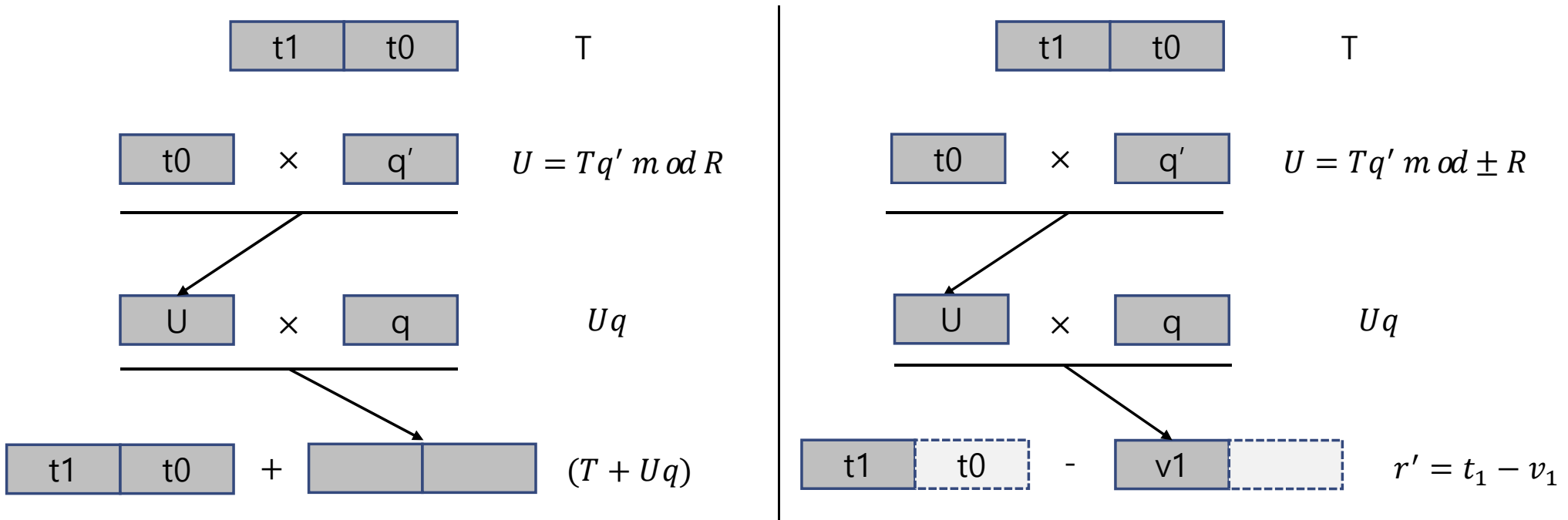
LBC 정수 연산 감산 방법



❖ 곱셈/제곱 연산에 대한 감산 방법 (Signed Short Montgomery 감산)

○ $q' = q^{-1} \bmod R$ 을 이용하여 덧셈이 아닌 뺄셈 수행 → signed 사용

○ Signed 자료형 사용으로 인해 Lazy reduction 쉽게 적용 가능 + 연산 이득 존재



❖참고내용

Algorithm 6 Original unsigned Montgomery reduction [5]; using Montgomery factor $\beta = 2^{18}$.

Input: a (32 bit)

Output: reduced a (16 bit)

```
1: mul t, a, q-1
2: and t, #0x3fff
3: mla a, t, q, a      ▷ a ← a + t · q
4: lsr a, #18
```

Algorithm 7 Signed Montgomery reduction (this work, adapted from [30]); using Montgomery factor $\beta = 2^{16}$.

Input: a (32 bit)

Output: reduced a (16 bit)

```
1: smulbb t, a, q-1 ▷ t ← (a mod β) · q-1
2: smulbb t, t, q      ▷ t ← (t mod β) · q
3: usub16 a, a, t      ▷ atop ← ⌊ $\frac{a}{2^{16}}$ ⌋ - ⌊ $\frac{t}{2^{16}}$ ⌋
```

- 참고문헌: Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4

LBC 정수 연산 감산 방법



❖ 곱셈/제곱 연산에 대한 감산 방법 (Signed Short Montgomery 감산)

○ Constant-time으로 동작

✓ 내부 연산 과정에서는 signed 표현 사용

✓ 저장 시에는 Standard 표현으로 변환

→ 음수의 경우 Q를 더함

```
int16_t montgomery_reduce(int32_t a)
{
    int16_t t;

    t = (int16_t)a*QINV;
    t = (a - (int32_t)t*KYBER_Q) >> 16;
    return t;
}
```

Kyber

```
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;
}
```

Dilithium

```
void poly_tobytes(uint8_t r[KYBER_POLYBYTES], const poly *a)
{
    unsigned int i;
    uint16_t t0, t1;

    for(i=0; i<KYBER_N/2; i++) {
        // map to positive standard representatives
        t0 = a->coeffs[2*i];
        t0 += ((int16_t)t0 >> 15) & KYBER_Q;
        t1 = a->coeffs[2*i+1];
        t1 += ((int16_t)t1 >> 15) & KYBER_Q;
        r[3*i+0] = (t0 >> 0);
        r[3*i+1] = (t0 >> 8) | (t1 << 4);
        r[3*i+2] = (t1 >> 4);
    }
}
```

❖ 곱셈/제곱 연산에 대한 감산 방법 (Barrett 감산)

○ $T = AB \bmod q$ for $A, B \in q$ and $0 \leq T < q$

○ $T = AB - Qq$ where $Q = \left\lfloor \frac{AB}{q} \right\rfloor$

○ Q 를 나눗셈 연산보다 부하가 적은 방법을 통해 추정 (floor 연산은 비트 시프트로 계산)

○ 과정

▪ 1) $Q = \left\lfloor \frac{AB}{q} \right\rfloor = \left\lfloor \frac{AB}{b^{k-1}} \cdot \frac{b^{2k}}{q} \cdot \frac{1}{b^{k+1}} \right\rfloor \rightarrow \hat{Q} = \left\lfloor \frac{\left\lfloor \frac{AB}{b^{k-1}} \right\rfloor \cdot \mu}{b^{k+1}} \right\rfloor$ 를 계산 ($Q - 2 \leq \hat{Q} \leq Q$)

• $\mu = \left\lfloor \frac{b^{2k}}{q} \right\rfloor$: q 에 대해 사전 계산

▪ 2) $r = \{(AB \bmod b^{k+1}) - (\hat{Q} \cdot q \bmod b^{k+1})\} < 3q$

• 최대 2번의 뺄셈 필요

❖ 곱셈/제곱 연산에 대한 감산 방법 (Signed Short Barrett 감산)

○ 덧셈 연산에 대한 감산은 signed Barret 감산 이용 (INVNTT 변환, 다항식 덧셈)

○ Montgomery 감산에서는 감산 후의 범위가 넓기 때문에, Barrett에서는 $r \in \left[-\frac{q-1}{2}, \frac{q-1}{2}\right]$

Algorithm 5 General reduction for signed one word integers

Require: $0 \leq q < \frac{\beta}{2}, -\frac{\beta}{2} \leq a < \frac{\beta}{2}$

Ensure: $r \equiv a \pmod{q}$ with $0 \leq r \leq q$

1: $v \leftarrow \left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1} \beta}{q} \right\rfloor$	▷ precomputed
2: $t \leftarrow \left\lfloor \frac{av}{2^{\lfloor \log(q) \rfloor - 1} \beta} \right\rfloor$	▷ signed high product and arithmetic right shift
3: $t \leftarrow tq \bmod \beta$	▷ signed low product
4: $r \leftarrow a - t$	

$$\hat{Q} = \left\lfloor \frac{\left\lfloor \frac{AB}{b^{k-1}} \right\rfloor \cdot \mu}{b^{k+1}} \right\rfloor, \mu = \left\lfloor \frac{b^{2k}}{q} \right\rfloor$$

$$r = \{(AB \bmod b^{k+1}) - (\hat{Q} \cdot q \bmod b^{k+1})\}$$

$$\left\lfloor \frac{a}{q} \right\rfloor = \left\lfloor \left(\frac{a}{2^{\lfloor \log(q) \rfloor - 1} \beta} \right) \left(\frac{2^{\lfloor \log(q) \rfloor - 1} \beta}{q} \right) \right\rfloor = \hat{Q} = t$$

// 여기서 a 는 one-word임. $v = \left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1} \beta}{q} \right\rfloor$

LBC 정수 연산 감산 방법



❖ 곱셈/제곱 연산에 대한 감산 방법 (Signed Short Barrett 감산)

- constant-time으로 동작
- Rounding 연산으로 인해 $q/2$ 값을 더함

```
int16_t barrett_reduce(int16_t a) {  
    int16_t t;  
    const int16_t v = ((1<<26) + KYBER_Q/2)/KYBER_Q;  
  
    t = ((int32_t)v*a + (1<<25)) >> 26;  
    t *= KYBER_Q;  
    return a - t;  
}
```

Kyber

```
int32_t reduce32(int32_t a) {  
    int32_t t;  
  
    t = (a + (1 << 22)) >> 23;  
    t = a - t*Q;    반올림하기 위해  
    return t;  
}
```

reduction in Dilithium

LBC 정수 연산 감산 방법



❖ 곱셈/제곱 연산에 대한 감산 방법 (Unsigned Short Montgomery 감산)

○ Standard 표현을 이용하여 연산 후, 감산 수행

○ final reduction 과정을 control bit을 이용하여 constant-time으로 동작하도록 함

- 감산 결과에 q 를 먼저 뺀 후, 음수가 된 경우 q 를 다시 더함 (양수인 경우 0을 더함)

```
static inline uint32_t
modp_montymul(uint32_t a, uint32_t b, uint32_t p, uint32_t p0i) {
    uint64_t z, w;
    uint32_t d;

    z = (uint64_t)a * (uint64_t)b;
    w = ((z * p0i) & (uint64_t)0x7FFFFFFF) * p;
    d = (uint32_t)((z + w) >> 31) - p;
    d += p & -(d >> 31);
    return d;
}
```

Falcon

❖ 덧셈/뺄셈 연산에 대한 감산

○ signed 자료형의 경우 lazy reduction을 통해 최종 결과에 대해서만 감산 수행

○ unsigned 자료형의 경우 constant-time 감산 수행

- q값을 뺀 후, 음수인 경우 (sign-bit 설정 시) 다시 q를 더함

```
static inline uint32_t
modp_add(uint32_t a, uint32_t b, uint32_t p) {
    uint32_t d;

    d = a + b - p;
    d += p & -(d >> 31);
    return d;
}
```

```
static inline uint32_t
modp_sub(uint32_t a, uint32_t b, uint32_t p) {
    uint32_t d;

    d = a - b;
    d += p & -(d >> 31);
    return d;
}
```

Falcon

❖Kyber/Dilithium에 적용된 Montgomery 도메인

○Montgomery 감산은 변환과정이 필요함

- NTT의 Twiddle factor는 Montgomery factor R 이 곱해진 형태임 → NTT 출력은 정수 도메인
- `basemul_acc_Montgomery` → R^{-1} 이 곱해진 상태
- `poly_tomont` → R^2 을 곱하여 Montgomery 감산 → 최종적으로 정수 도메인 값 출력

```
// indcpa_keypair
gen_a(a, publicseed);

for(i=0;i<KYBER_K;i++)
    poly_getnoise_eta1(&skpv.vec[i], noiseseed, nonce++);
for(i=0;i<KYBER_K;i++)
    poly_getnoise_eta1(&e.vec[i], noiseseed, nonce++);

polyvec_ntt(&skpv);
polyvec_ntt(&e);
for(i=0;i<KYBER_K;i++) {
    polyvec_basemul_acc_montgomery(&pkpv.vec[i], &a[i], &skpv);
    poly_tomont(&pkpv.vec[i]);
}
```

// 정수도메인
// 정수도메인

// R^{-1} 이 곱해져있는 형태
// R^2 을 곱하여 정수도메인으로 변환

❖Kyber/Dilithium에 적용된 Montgomery 도메인

○Montgomery 감산은 변환과정이 필요함

- NTT의 Twiddle factor는 Montgomery factor R 이 곱해진 형태임 → NTT 출력은 정수 도메인
- basemul_acc_Montgomery → R^{-1} 이 곱해진 상태
- invntt_tomont → invntt의 마지막 과정에서 R^2 을 곱하여 Montgomery 감산 → 정수 도메인

```
// indcpa_enc
gen_at(at, seed);
polyvec_ntt(&sp); // 정수도메인
for(i=0;i<KYBER_K;i++)
    poly_getnoise_eta1(sp.vec+i, coins, nonce++);
for(i=0;i<KYBER_K;i++)
    poly_getnoise_eta2(ep.vec+i, coins, nonce++);
for(i=0;i<KYBER_K;i++)
    polyvec_basemul_acc_montgomery(&b.vec[i], &at[i], &sp); //  $R^{-1}$ 이 곱해져있는 형태
polyvec_basemul_acc_montgomery(&v, &pkpv, &sp);

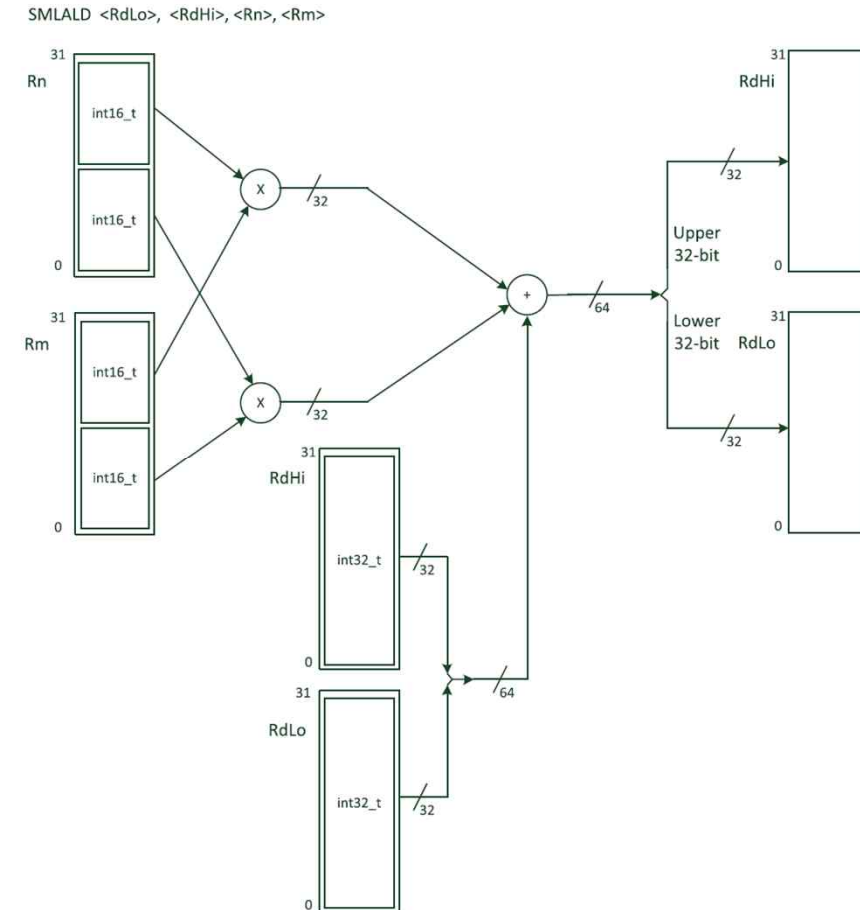
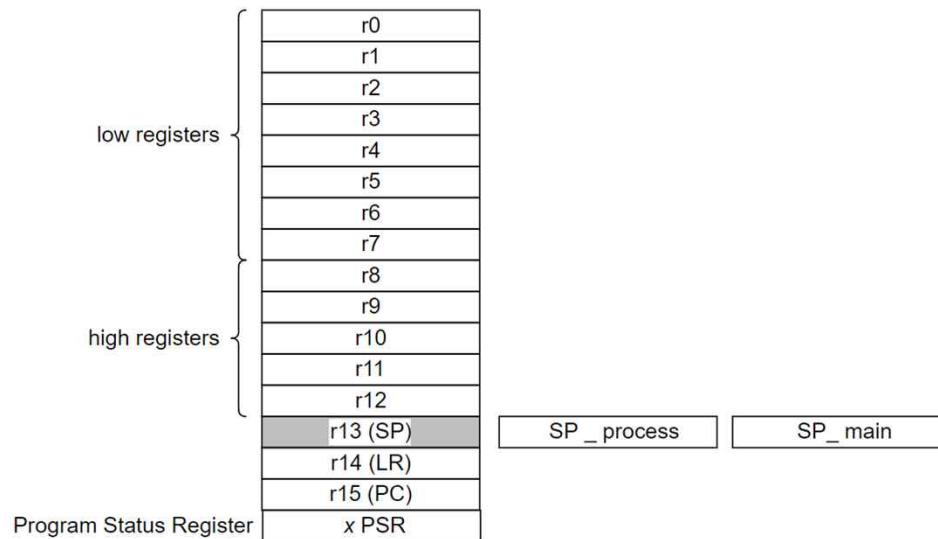
polyvec_invntt_tomont(&b); // INVNTT과정 마지막에서  $R^2$ 을 곱하여 정수도메인으로 변환
poly_invntt_tomont(&v);
```

LBC 운영환경별 최적화 사항 (Cortex-M4)



❖ 32-bit Cortex-M4 환경

- DSP 연산 (fact MAC, 제한된 SIMD) 제공
- Barrel Shifter 제공 (연산 명령과 Shift 연산을 함께 수행 가능)
- 나눗셈을 제외한 기본적인 연산명령어들은 고정클럭으로 동작



- 참고문헌: <https://developer.arm.com/documentation/100166/0001/Programmers-Model/Instruction-set-summary/Table-of-processor-instructions>

LBC 운영환경별 최적화 사항 (Cortex-M4)



❖32-bit Cortex-M4

- PQM4 프로젝트(<https://github.com/mupq/pqm4>)에 최신의 구현기법들이 통합되어 있는 상태
- 최신 구현 기법들은 NTT와 유한체 감산 연산을 Cortex-M4 상에서 asm을 이용하여 최적화함

알고리즘	적용된 최적화 기법
Kyber	<ul style="list-style-type: none">- NTT 기반 다항식 곱셈 연산을 ASM으로 처리 (7-4 레이어 병합, 3-1 레이어 병합)- 다항식 저장으로 인한 메모리 부하를 완화하기 위해 계수 단위 처리 방법 적용- m4fspeed 버전, m4fstack 버전 제공
Dilithium	<ul style="list-style-type: none">- NTT 기반 다항식 곱셈 연산을 ASM으로 처리 (7-6, 5-4, 3-2, 1-0 레이어 병합)
Falcon	<ul style="list-style-type: none">- C언어 레벨로만 구현되어 있음

- 참고문헌
 - pqm4 project (<https://github.com/mupq/pqm4>)
 - Faster Kyber and Dilithium on the Cortex-M4, ACNS 2022
 - Compact Dilithium Implementation on Cortex-M4 and Cortex-M4, TCHES 2021

LBC 운영환경별 최적화 사항 (Cortex-M4)



❖32-bit Cortex-M4

- 메모리 환경이 제한적이기 때문에 CPU 참조구현과 비교하여 다른 연산 순서를 적용함
- m4fspeed와 m4fstack 유사한 연산 구조를 가지나, m4fspeed가 추가적으로 Asymmetric multiplication과 Better accumulation 기법을 적용함

```
void indcpa_keypair(uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES],
                   uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES])
{
    unsigned int i;
    uint8_t buf[2*KYBER_SYMBYTES];
    const uint8_t *publicseed = buf;
    const uint8_t *noiseseed = buf+KYBER_SYMBYTES;
    uint8_t nonce = 0;
    polyvec a[KYBER_K], e, pkpv, skpv;

    randombytes(buf, KYBER_SYMBYTES);
    hash_g(buf, buf, KYBER_SYMBYTES);
```

다항식 18개

Kyber768 on liboqs

```
void indcpa_keypair(unsigned char *pk, unsigned char *sk) {
    polyvec skpv;
    poly pkp;
    unsigned char buf[2 * KYBER_SYMBYTES];
    unsigned char *publicseed = buf;
    unsigned char *noiseseed = buf + KYBER_SYMBYTES;
    int i;
    unsigned char nonce = 0;

    randombytes(buf, KYBER_SYMBYTES);
    hash_g(buf, buf, KYBER_SYMBYTES);
```

다항식 3개

Kyber768 stack on pqm4

LBC 운영환경별 최적화 사항 (Cortex-M4)



❖Kyber (m4fstack) on 32-bit Cortex-M4

```
void indcpa_keypair(unsigned char *pk, unsigned char *sk) {
```

```
    polyvec skpv;  
    poly pkp;
```

skpv는 비밀벡터 s 저장
pkp는 A 행렬에서 하나의 다항식 저장

```
    unsigned char buf[2 * KYBER_SYMBYTES];  
    unsigned char *publicseed = buf;  
    unsigned char *noiseseed = buf + KYBER_SYMBYTES;  
    int i;  
    unsigned char nonce = 0;
```

```
    randombytes(buf, KYBER_SYMBYTES);  
    hash_g(buf, buf, KYBER_SYMBYTES);
```

```
    for (i = 0; i < KYBER_K; i++)  
        poly_getnoise(skpv.vec + i, noiseseed, nonce++);
```

```
    polyvec_ntt(&skpv);
```

비밀벡터 s는 NTT 도메인으로 유지

```
    for (i = 0; i < KYBER_K; i++) {
```

```
        matacc(&pkp, &skpv, i, publicseed, 0);  
        poly_invntt(&pkp);  
  
        poly_addnoise(&pkp, noiseseed, nonce++);  
        poly_ntt(&pkp);  
  
        poly_tobytes(pk+i*KYBER_POLYBYTES, &pkp);
```

A[i]*skpv를 연산한 후
INVNTT 수행
(e 벡터는 on-the-fly
하게 더해짐)

```
    }  
    polyvec_tobytes(sk, &skpv);  
    memcpy(pk + KYBER_POLYVECBYTES, publicseed, KYBER_SYMBYTES);
```

```
void matacc(poly* r, const polyvec *b, unsigned char i, const unsigned char *seed, int transposed) {
```

```
    unsigned char buf[XOF_BLOCKBYTES+2];  
    xof_state state;  
    int16_t c[4];  
    int j = 0;
```

```
    if (transposed)  
        xof_absorb(&state, seed, i, j);  
    else  
        xof_absorb(&state, seed, j, i);
```

```
    xof_squeezeblocks(buf, 1, &state);  
    matacc_asm(r->coeffs, b->vec[j].coeffs, c, buf, zetas, &state);  
    for(j=1;j<KYBER_K;j++) {
```

```
        if (transposed)  
            xof_absorb(&state, seed, i, j);  
        else  
            xof_absorb(&state, seed, j, i);  
  
        xof_squeezeblocks(buf, 1, &state);
```

```
        matacc_asm_acc(r->coeffs, b->vec[j].coeffs, c, buf, zetas, &state);
```

matacc_asm_acc()
→ Rejection Sampling을 통해
반복문 한번 당 2개의 계수 생성

LBC 운영환경별 최적화 사항 (Cortex-M4)



❖ Kyber (m4fspeed) on 32-bit Cortex-M4

○ Asymmetric multiplication 기법

- As 연산 수행 시, 비밀벡터 s 는 고정임
- basemul 단계에서 s 와 twiddle factor의 곱은 반복됨 (이 연산 결과를 holding)
- skpv_prime에 연산 결과를 저장하여 활용

$$\begin{aligned}\widetilde{h}_{2i} + \widetilde{h}_{2i+1}X &= (\widetilde{a}_{2i} + \widetilde{a}_{2i+1}X)(\widetilde{b}_{2i} + \widetilde{b}_{2i+1}X) \bmod X^2 - \zeta^{2br_7(i)+1} \\ \rightarrow \widetilde{h}_{2i} &\leftarrow \widetilde{a}_{2i} \cdot \widetilde{b}_{2i} + \widetilde{a}_{2i+1} \cdot \widetilde{b}_{2i+1} \cdot \zeta^{2br_7(i)+1} \\ \rightarrow \widetilde{h}_{2i+1} &\leftarrow \widetilde{a}_{2i+1} \cdot \widetilde{b}_{2i} + \widetilde{a}_{2i} \cdot \widetilde{b}_{2i+1}\end{aligned}$$

○ Better accumulation 기법

- A의 행과 s 를 곱한 값을 계속 누적하여 마지막에 감산 수행
 - 32-bit 곱셈 결과에 대해서 감산하지 않고 누적 수정

```
polyvec_ntt(&skpv);
```

```
// i = 0
```

```
matacc_cache32(&pkp, &skpv, &skpv_prime, 0, publicseed, 0);  
poly_invntt(&pkp);
```

Asymmetric Mult

```
poly_addnoise(&pkp, noiseseed, nonce++);  
poly_ntt(&pkp);
```

```
poly_tobytes(pk, &pkp);
```

```
for (i = 1; i < KYBER_K; i++) {
```

```
    matacc_opt32(&pkp, &skpv, &skpv_prime, i, publicseed, 0);  
    poly_invntt(&pkp);
```

Better Accu

```
poly_addnoise(&pkp, noiseseed, nonce++);  
poly_ntt(&pkp);
```

```
poly_tobytes(pk+i*KYBER_POLYBYTES, &pkp);
```

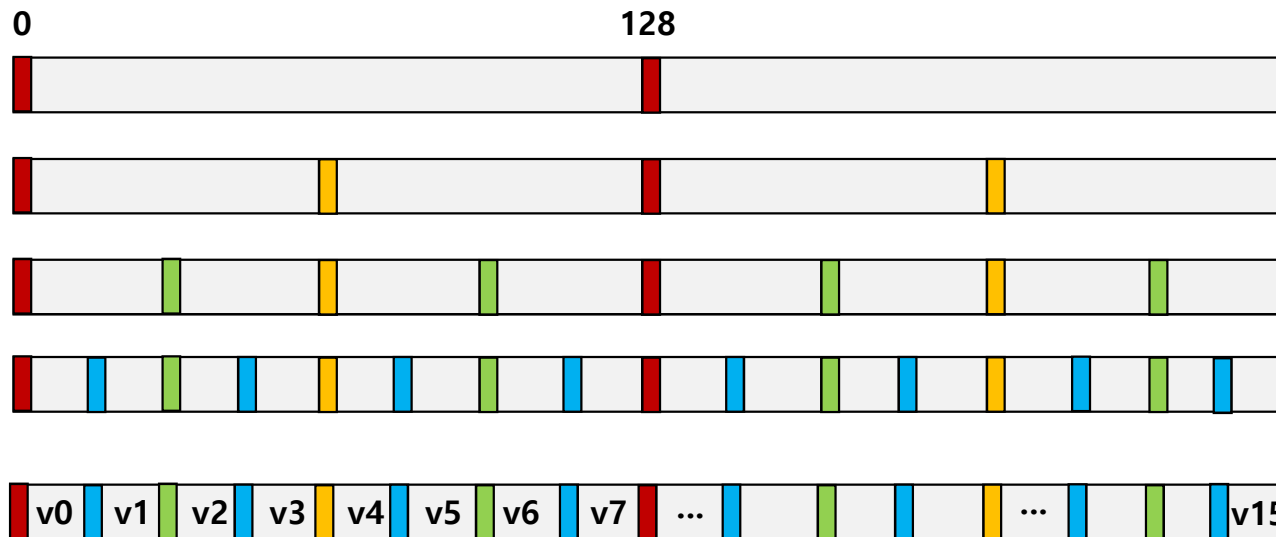
```
}
```

```
polyvec_tobytes(sk, &skpv);
```

```
memcpy(pk + KYBER_POLYVECBYTES, publicseed, KYBER_SYMBYTES);
```

❖NTT Layer Merging

- 임베디드 환경에서 메모리 접근 명령은 연산 명령어보다 부하가 큼
- 제한된 레지스터 공간을 효율적으로 이용하기 위해 NTT 연산 계층을 통합함



- NTT 연산 계층에서 연산결과가 의존적인 부분들을 식별하여, 해당 부분들을 레지스터에 로드
→ 추가적인 메모리 접근 없이 NTT 연산 수행 가능
- 참고문헌
Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1, TCHES 2022

❖ 감산 방법 최적화 (Montgomery 감산)

○ 32-bit Cortex-M4의 ASM 명령어를 적절히 활용하여 Montgomery, Barrett 감산 최적화

- 32-bit Signed Montgomery multiplication ($R = 2^{32}$)

```
smull a0, a1, a, b      // 곱셈  
mul    tmp, a0, Qinvt   →  $t = (\text{int64\_t})(\text{int32\_t})a * Q_{INV};$   
smlal a0, a1, tmp, Q   →  $t = (a - (\text{int64\_t})t * Q) \gg 32;$   
(Result in a1)
```

- 16-bit Signed Montgomery multiplication ($R = 2^{16}$)

```
smulbb t, a, b          // 곱셈  
smulbb tmp, t, Qinvt     →  $t = (\text{int16\_t})a * Q_{INV};$   
smlabb tmp, tmp, Q, t    →  $t = (a - (\text{int32\_t})t * KYBER\_Q)$   
asr tmp, tmp, #16       →  $t = (a - (\text{int32\_t})t * KYBER\_Q) \gg 16;$   
(Usually asr can be eliminated by merging with packing pkhbt)
```

```
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;  
}
```

```
int16_t montgomery_reduce(int32_t a)  
{  
    int16_t t;  
  
    t = (int16_t)a * QINV;  
    t = (a - (int32_t)t * KYBER_Q) >> 16;  
    return t;  
}
```

- 참고문헌: Cortex-M4 optimizations for {R, M}LWE schemes, TCHES 2020
Compact Dilithium Implementation on Cortex-M4 and Cortex-M4,, TCHES 2021

❖ 감산 방법 최적화 (Packed 연산)

○ 32-bit 레지스터에 2개의 계수를 로드 또는 저장하여 처리

Input : $a = (a_t \parallel a_b)$

Output: $c = (c_t \parallel c_b) \bmod^{\pm} q$

```
1 smlawb t0, -[232/q], a, 215
2 smlabt t0, q, t0, a
3 smlawt t1, -[232/q], a, 215
4 smulbt t1, q, t1
5 add t1, a, t1, lsl #16
6 pkhbt c, t0, t1, lsl #16
```

packed Barrett reduction

```
1.  $t_0 \leftarrow a_b \cdot (BC1) + BC2$ 
2.  $t_0 \leftarrow q \cdot t_0 + a$ 
3.  $t_1 \leftarrow a_t \cdot (BC1) + BC2$ 
4.  $t_1 \leftarrow q \cdot t_0 + a$ 
5.  $t_1 \leftarrow a + t_1 \ll 16$ 
6.  $c \leftarrow (t_1 \ll 16 \parallel t_0)$ 
```

```
.macro doublebutterfly tb, a0, a1, twiddle, tmp, tmp2, q, qinv
    smulb\tb \tmp, \a1, \twiddle // a1_b * twiddle_tb
    smult\tb \a1, \a1, \twiddle // a1_t * twiddle_tb
    montgomery \q, \qinv, \tmp, \tmp2 // reduce -> result in tmp2
    montgomery \q, \qinv, \a1, \tmp // reduce -> result in tmp
    pkhtb \tmp2, \tmp, \tmp2, asr#16 // combine results from above in one register
    usub16 \a1, \a0, \tmp2 // a0 - a1 * twiddle (a0, a1 contain 2 coeffs)
    uadd16 \a0, \a0, \tmp2 // a0 + a1 * twiddle (a0, a1 contain 2 coeffs)
.endm

.macro montgomery q, qinv, a, tmp
    smulbt \tmp, \a, \qinv
    smlabb \tmp, \q, \tmp, \a
.endm
```

- 참고문헌: Cortex-M4 optimizations for {R, M}LWE schemes, TCHES 2020
Compact Dilithium Implementation on Cortex-M4 and Cortex-M4,, TCHES 2021
Faster Kyber and Dilithium on the Cortex-M4, ACNS 2022

❖SIMD 환경에서의 LBC 최적화

○ 범용 CPU 환경에서 AVX2 명령어 집합 이용

○ 16개의 256-bit 벡터 레지스터

○ 32-bit ARMv7-A, 64-bit ARMv8 환경에서 NEON 명령어 집합 이용

○ 32-bit ARMv7-A: 16개의 128-bit 벡터 레지스터

○ 64-bit ARMv8: 32개의 128-bit 벡터 레지스터

○ 병렬화 가능 부분

▪ Sampling (Rejection, CBD)

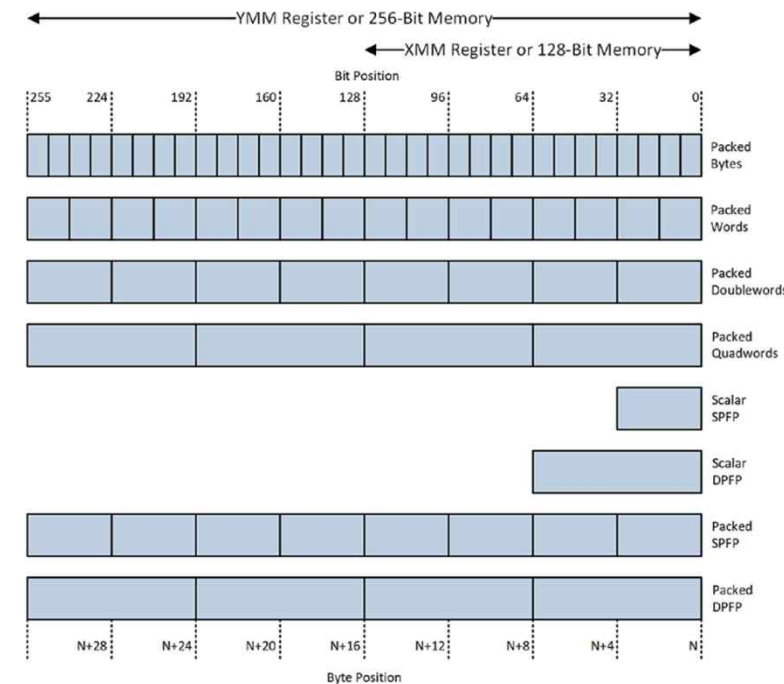
▪ SHA3 해시함수

- XKCP 프로젝트에서 SHA3에 대한 병렬 구현 제공

(<https://github.com/XKCP/XKCP/tree/master/lib/low>)

▪ NTT 기반 다항식 곱셈

▪ 기타 다항식 연산



❖ Kyber on AVX2

○ Kyber의 핵심 동작 과정을 AVX2를 이용하여 병렬화를 진행

- gen_a
- getnoise_eta1
- polyvec_ntt,
- polyvec_basemul_acc_montgomery
- polyvec_add/reduce

○ 에러 및 비밀벡터는 4개의 다항식을 동시에 샘플링

- Kyber768의 경우 더미연산이 존재

```
void PQCLEAN_KYBER768_AVX2_indcpa_keypair(uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES],
      uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES]) {
    unsigned int i;
    uint8_t buf[2 * KYBER_SYMBYTES];
    const uint8_t *publicseed = buf;
    const uint8_t *noiseseed = buf + KYBER_SYMBYTES;
    polyvec a[KYBER_K], e, pkpv, skpv;

    randombytes(buf, KYBER_SYMBYTES);
    hash_g(buf, buf, KYBER_SYMBYTES);

    gen_a(a, publicseed); rejection Sampling을 병렬적으로 구성

    PQCLEAN_KYBER768_AVX2_poly_getnoise_eta1_4x(skpv.vec + 0, skpv.vec + 1, skpv.vec + 2, e.vec + 0, noiseseed, 0, 1, 2, 3);
    PQCLEAN_KYBER768_AVX2_poly_getnoise_eta1_4x(e.vec + 1, e.vec + 2, pkpv.vec + 0, pkpv.vec + 1, noiseseed, 4, 5, 6, 7);

    PQCLEAN_KYBER768_AVX2_polyvec_ntt(&skpv);
    PQCLEAN_KYBER768_AVX2_polyvec_reduce(&skpv);
    PQCLEAN_KYBER768_AVX2_polyvec_ntt(&e); 더미 연산

    // matrix-vector multiplication
    for (i = 0; i < KYBER_K; i++) {
        PQCLEAN_KYBER768_AVX2_polyvec_basemul_acc_montgomery(&pkpv.vec[i], &a[i], &skpv);
        PQCLEAN_KYBER768_AVX2_poly_tomont(&pkpv.vec[i]);
    }

    PQCLEAN_KYBER768_AVX2_polyvec_add(&pkpv, &pkpv, &e);
    PQCLEAN_KYBER768_AVX2_polyvec_reduce(&pkpv);

    pack_sk(sk, &skpv);
    pack_pk(pk, &pkpv, publicseed);
}
```

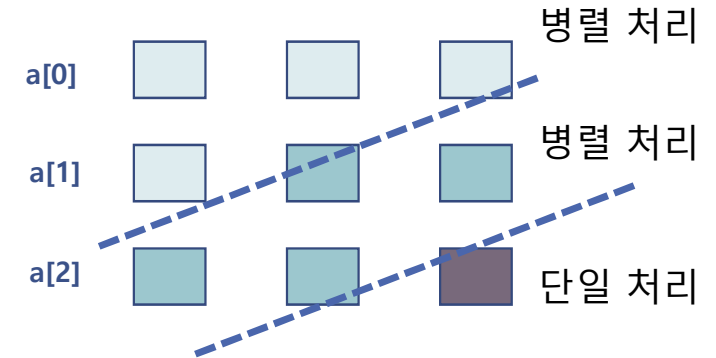
LBC 운영환경별 최적화 사항



❖Kyber on AVX2

○ gen_a(공개 행렬 A 생성과정)

- 공개 행렬 A의 다항식에 대해 4개 단위로 생성
- shake도 4개 단위로 병렬화



4개의 seed에 대한
shake 병렬 처리

```
PQCLEAN_KYBER768_AVX2_shake128x4_absorb_once(&state, buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, 34);  
PQCLEAN_KYBER768_AVX2_shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, REJ_UNIFORM_AVX_NBLOCKS, &state);
```

```
ctr0 = PQCLEAN_KYBER768_AVX2_rej_uniform_avx(a[0].vec[0].coeffs, buf[0].coeffs);  
ctr1 = PQCLEAN_KYBER768_AVX2_rej_uniform_avx(a[0].vec[1].coeffs, buf[1].coeffs);  
ctr2 = PQCLEAN_KYBER768_AVX2_rej_uniform_avx(a[0].vec[2].coeffs, buf[2].coeffs);  
ctr3 = PQCLEAN_KYBER768_AVX2_rej_uniform_avx(a[1].vec[0].coeffs, buf[3].coeffs);
```

각 다항식에 대한 rejection sampling 병렬처리
16개 계수를 동시에 샘플링하며, 총 4번 반복

```
while (ctr0 < KYBER_N || ctr1 < KYBER_N || ctr2 < KYBER_N || ctr3 < KYBER_N) {  
    PQCLEAN_KYBER768_AVX2_shake128x4_squeezeblocks(buf[0].coeffs, buf[1].coeffs, buf[2].coeffs, buf[3].coeffs, 1, &state);
```

```
    ctr0 += rej_uniform(a[0].vec[0].coeffs + ctr0, KYBER_N - ctr0, buf[0].coeffs, SHAKE128_RATE);  
    ctr1 += rej_uniform(a[0].vec[1].coeffs + ctr1, KYBER_N - ctr1, buf[1].coeffs, SHAKE128_RATE);  
    ctr2 += rej_uniform(a[0].vec[2].coeffs + ctr2, KYBER_N - ctr2, buf[2].coeffs, SHAKE128_RATE);  
    ctr3 += rej_uniform(a[1].vec[0].coeffs + ctr3, KYBER_N - ctr3, buf[3].coeffs, SHAKE128_RATE);
```

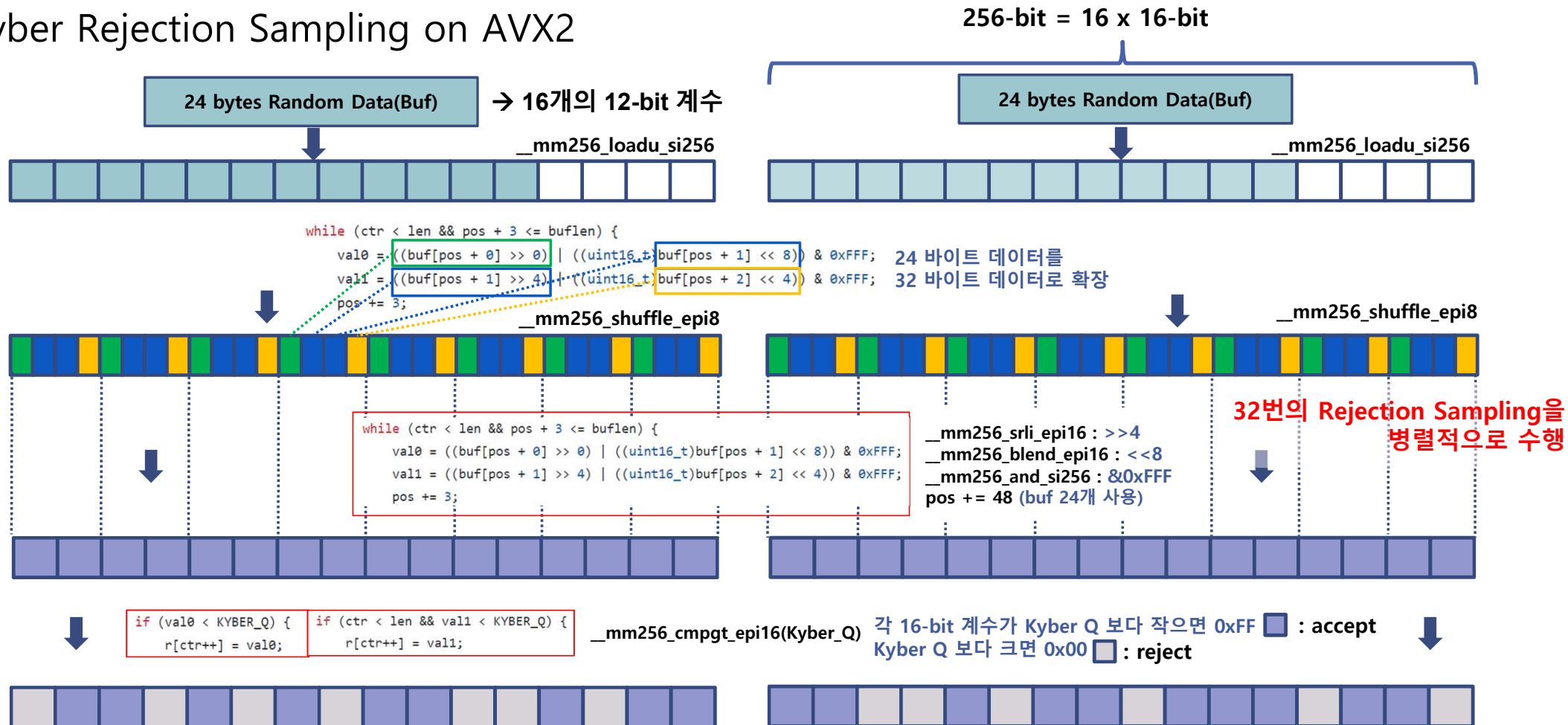
```
}
```

buf가 부족한 경우 해당 부분에 대한
rejection sampling 진행 (not AVX2)

LBC 운영환경별 최적화 사항



❖Kyber Rejection Sampling on AVX2

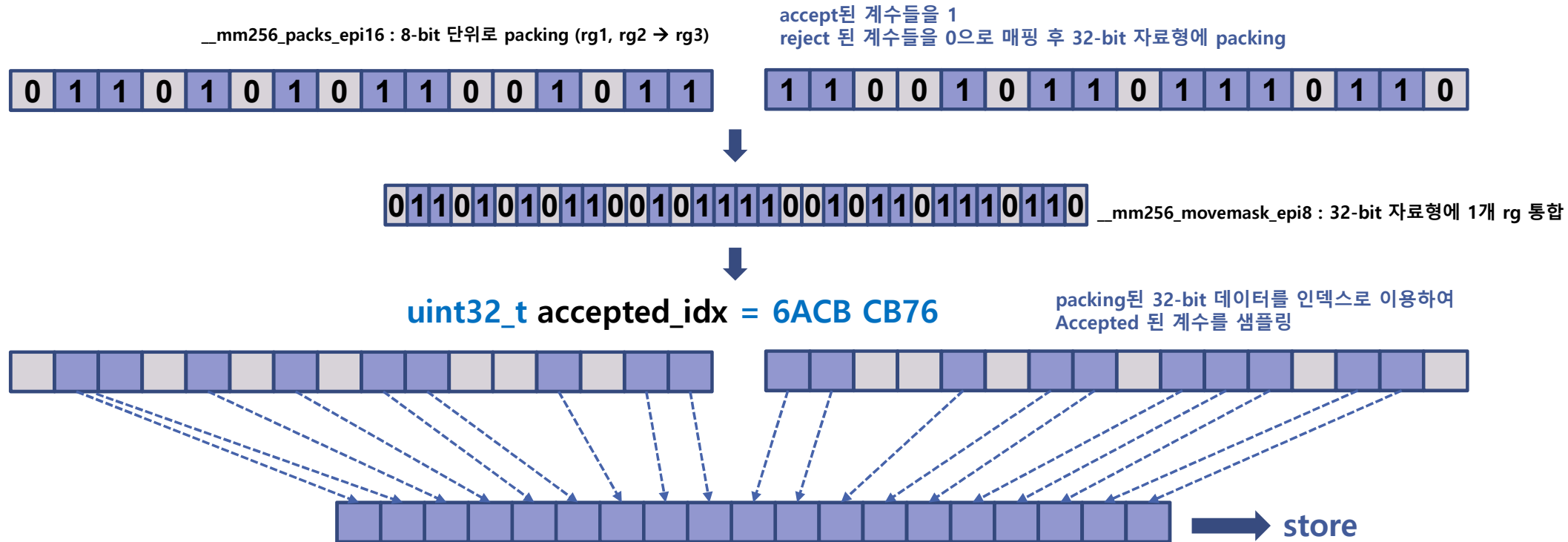


LBC 운영환경별 최적화 사항



❖Kyber Rejection Sampling on AVX2

rg : AVX2 register



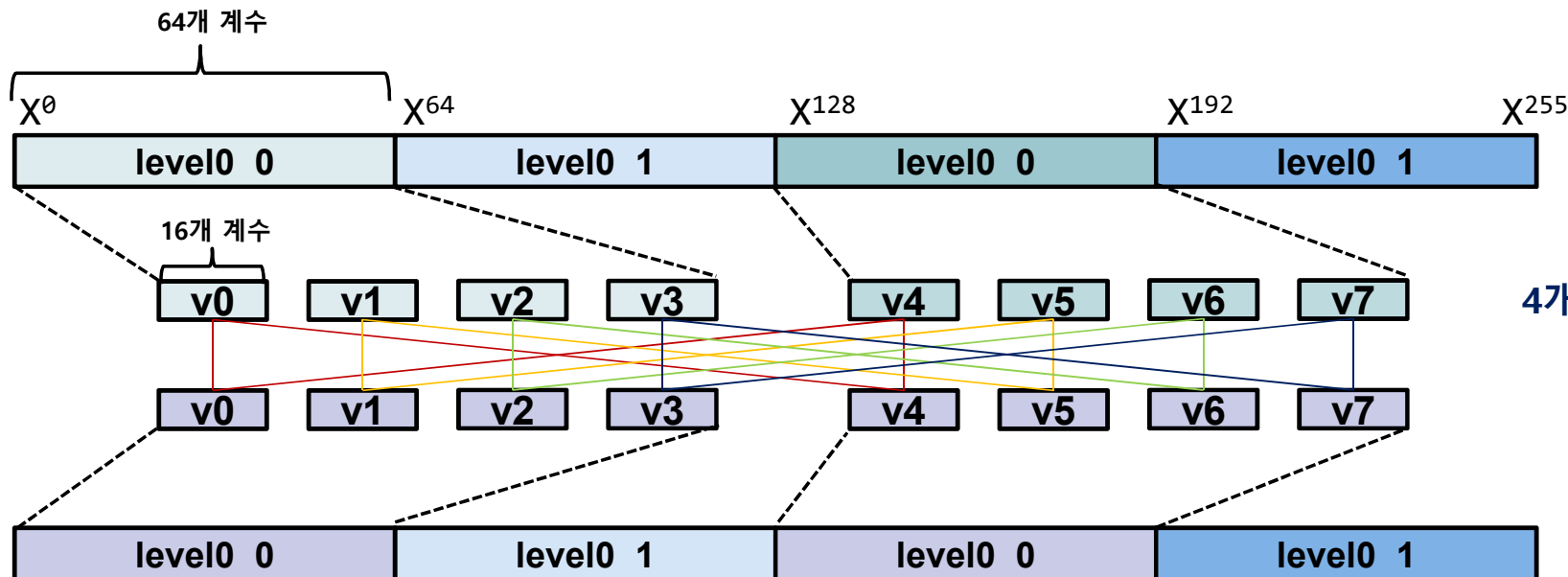
32개 계수 동시 Sampling (2개 레지스터 이용, avx2) → 16개 계수 동시 Sampling (1개 레지스터 이용, avx2) → 16개 계수 이하 남은 시는 단일 sampling (cpu)
샘플해야하는 계수가 32개 미만으로 남을 때 까지 동작 샘플해야하는 계수가 16개 미만으로 남을 때 까지 동작

LBC 운영환경별 최적화 사항



❖ Kyber NTT on AVX2

- 8개 레지스터 = $8 \times 256 = 16 \times 128 \rightarrow$ 한번에 128개의 계수에 대한 NTT를 수행
- 7개의 NTT Layer 중 0번째 레이어를 제외하고 1~6번째 레이어는 Merged
 - 하기 그림은 level0 offset : 0의 도식도
 - level0 offset : 1 은 동일한 논리를 이용하여 남은 계수에 대해 Butterfly 수행



```
.text
.global cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx)
.global _cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx)
cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx):
    _cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx):
        vmovdqa    _16XQ*2(%rsi),%ymm0

level0      0      128개 계수
level0      1      128개 계수

levels1t6   0      128개 계수
levels1t6   1      128개 계수

ret
offset
```

데이터 Load 시
4개의 레지스터(64개 계수)에 대해
Butterfly 수행

각 레지스터는 16개의 계수
에 대해 동시에 Butterfly
수행

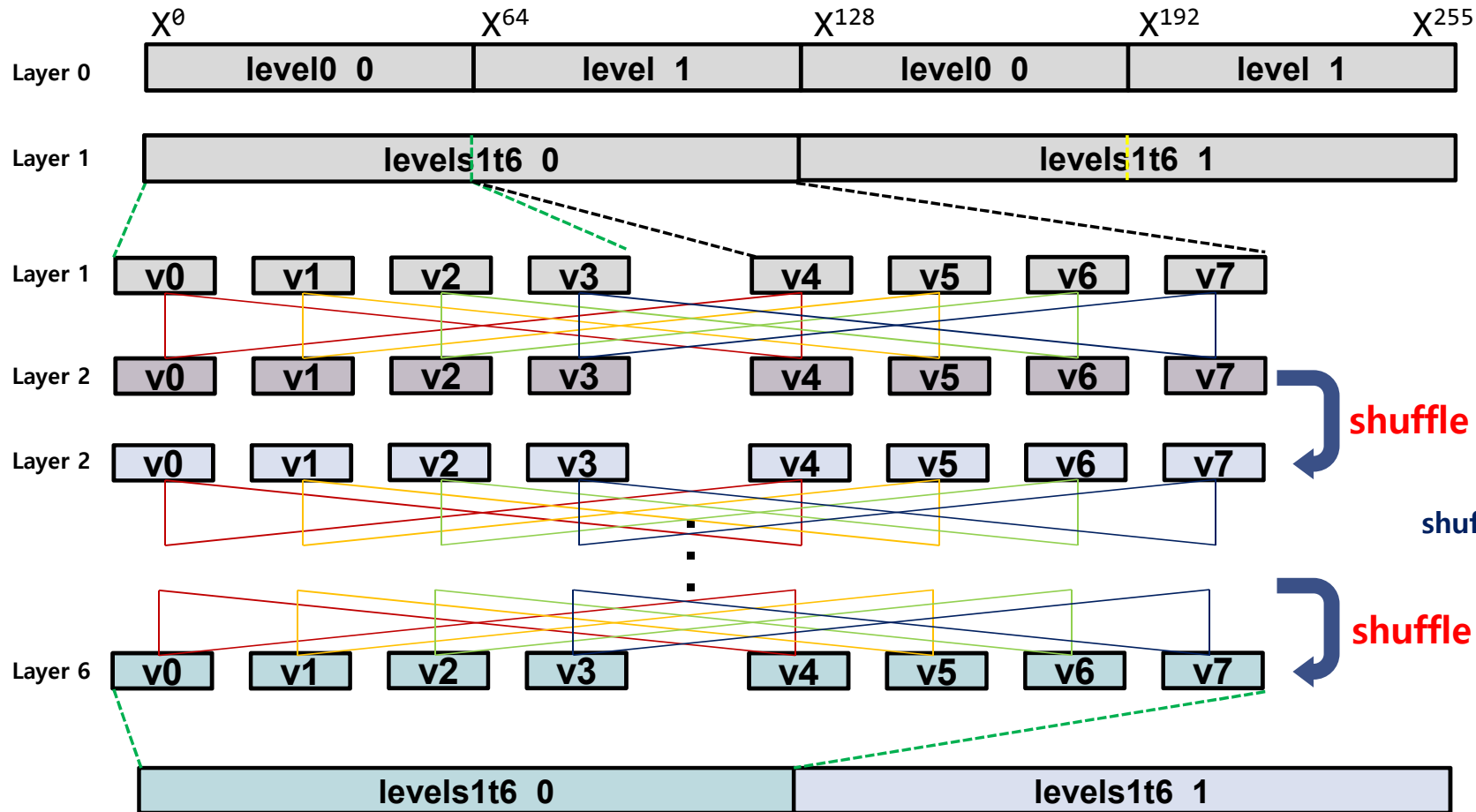
LBC 운영환경별 최적화 사항



❖ Kyber NTT on AVX2

○ Merged 된 1~6번째 Layer의 연산은 Layer당 재정렬 과정이 필요

- **Shuffle n** : 각 레지스터가 n개의 계수가 연속적으로 배열되도록 재정렬



```
.text
.global cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx)
.global _cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx)
cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx):
_cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx):
    vmovdqa    _16XQ*2(%rsi),%ymm0

level0 0
level0 1

levels1t6 0
levels1t6 1

ret
```

level0	0	128개 계수
level0	1	128개 계수
levels1t6	0	128개 계수
levels1t6	1	128개 계수

offset

shuffle 8 데이터의 접근은 1 Layer Load
6 Layer Store에서만 발생

shuffle 과정을 제외하고 각 Layer마다
연산논리를 동일하게 구성

shuffle 1 각 레지스터는 16개의 계수에
대해 동시에 Butterfly 수행

상기 그림은
levels1t6 offset: 0의 도식도

LBC 운영환경별 최적화 사항



❖ Kyber NTT on AVX2 세부사항 (NTT and Montgomery reduce)

○ Kyber의 7개(0~6) 레이어 중 1~6개 레이어를 병합

```

.text
.global cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx)
.global _cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx)
cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx):
_cdec1(PQCLEAN_KYBER768_AVX2_ntt_avx):
    vmovdqa    _16XQ*2(%rsi),%ymm0

level0      0
level0      1

levels1t6   0
levels1t6   1

ret
        
```

```

.macro level0 off
vpbroadcastq    (_ZETAS_EXP+0)*2(%rsi),%ymm15
vmovdqa         (64*\off+128)*2(%rdi),%ymm8
vmovdqa         (64*\off+144)*2(%rdi),%ymm9
vmovdqa         (64*\off+160)*2(%rdi),%ymm10
vmovdqa         (64*\off+176)*2(%rdi),%ymm11
vpbroadcastq    (_ZETAS_EXP+4)*2(%rsi),%ymm2

mul             8,9,10,11

vmovdqa         (64*\off+ 0)*2(%rdi),%ymm4
vmovdqa         (64*\off+ 16)*2(%rdi),%ymm5
vmovdqa         (64*\off+ 32)*2(%rdi),%ymm6
vmovdqa         (64*\off+ 48)*2(%rdi),%ymm7

reduce
update          3,4,5,6,7,8,9,10,11

vmovdqa         %ymm3,(64*\off+ 0)*2(%rdi)
vmovdqa         %ymm4,(64*\off+ 16)*2(%rdi)
vmovdqa         %ymm5,(64*\off+ 32)*2(%rdi)
vmovdqa         %ymm6,(64*\off+ 48)*2(%rdi)
vmovdqa         %ymm8,(64*\off+128)*2(%rdi)
vmovdqa         %ymm9,(64*\off+144)*2(%rdi)
vmovdqa         %ymm10,(64*\off+160)*2(%rdi)
vmovdqa         %ymm11,(64*\off+176)*2(%rdi)
.endm
        
```

다항식 절반씩 계산
(0과 1은 offset을 의미)

Algorithm 3 Signed Montgomery reduction

Require: $0 < q < \frac{\beta}{2}$ odd, $-\frac{\beta}{2}q \leq a = a_1\beta + a_0 < \frac{\beta}{2}q$ where $0 \leq a_0 < \beta$

Ensure: $r' \equiv \beta^{-1}a \pmod{q}$, $-q < r' < q$

1: $m \leftarrow a_0q^{-1} \bmod \pm\beta$

Zetas x Qinvs → precomputed

2: $t_1 \leftarrow \lfloor \frac{mq}{\beta} \rfloor$

3: $r' \leftarrow a_1 - t_1$

- **level0** : 0번째 레이어의 분할
- **levels1t6** : 1~6번째 레이어 분할 (Merged)
- **mul** : $m = a_0 \times q^{-1} \times \text{zetas}$ 와 a_1 을 생성
- **reduce** : $t_1 = \lfloor mq/\beta \rfloor$
- **reduce** : $a_1 - t_1$ 및 Butterfly의 산술연산 수행

LBC 운영환경별 최적화 사항



Input: $a_0 \dots a_{15}$ Input: $b_0 \dots b_{15}$

mul

$$t0_0 \dots t0_{15} \leftarrow b_0 \dots b_{15} \times zl \dots zl$$

$$t0 \leftarrow b \times zl \bmod^{\pm} \beta \text{ with vpmullw}$$

$$t1_0 \dots t1_{15} \leftarrow b_0 \dots b_{15} \times zh \dots zh$$

$$t1 \leftarrow \left\lfloor \frac{b \times zh}{\beta} \right\rfloor \text{ with vpmulhw}$$

reduce

$$t0_0 \dots t0_{15} \leftarrow t0_0 \dots t0_{15} \times q \dots q$$

$$t0 \leftarrow \left\lfloor \frac{t0 \times q}{\beta} \right\rfloor \text{ with vpmulhw}$$

output:

$$a_0 \dots a_{15} \leftarrow t1_0 \dots t1_{15} - t0_0 \dots t0_{15}$$

$$a = m_{ont}(a + bz) \leftarrow a + t1 - t0 \text{ with vpaddw, vpsubw}$$

output:

$$b_0 \dots b_{15} \leftarrow t1_0 \dots t1_{15} + t0_0 \dots t0_{15}$$

$$b = m_{ont}(a - bz) \leftarrow a - t1 + t0 \text{ with vpsubw, vpaddw}$$

update

• AVX2 Signed Montgomery multiplication ($\beta = 2^{16}$)

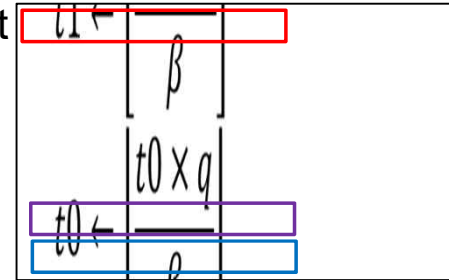
• input : $a = a_1\beta + a_0$: 32-bit

• output : $r = m_{ont}(a)$: 16-bit

$$m \leftarrow a_0 q^{-1} \bmod^{\pm} \beta$$

$$t \leftarrow \left\lfloor \frac{m q}{\beta} \right\rfloor$$

$$r \leftarrow a_1 - t$$



• AVX2 Butterfly with Montgomery multiplication ($\beta = 2^{16}$)

• input : a, b, zh, zl : 16-bit

(zl is precomputed q^{-1})

$t0, t1$

• output : $m_{ont}(a + bz, a - bz)$ (16-bit)

$$t0 \leftarrow b \times zl \bmod^{\pm} \beta \quad \text{vpmullw}$$

$$t1 \leftarrow \left\lfloor \frac{b \times zh}{\beta} \right\rfloor \quad \text{vpmulhw}$$

$$t0 \leftarrow \left\lfloor \frac{t0 \times q}{\beta} \right\rfloor \quad \text{vpmulhw}$$

$$m_{ont}(a + bz) \leftarrow a + t1 - t0 \quad \text{vpsubw, vpaddw}$$

$$m_{ont}(a - bz) \leftarrow a - t1 + t0 \quad \text{vpaddw, vpsubw}$$

❖Kyber Barret Reduction on AVX2

○ AVX2의 ASM 명령어를 활용하여 Barrett 감산 최적화

PQClean reference Code

```
int16_t PQCLEAN_KYBER768_CLEAN_barrett_reduce(int16_t a) {
    int16_t t;
    const int16_t v = ((1 << 26) + KYBER_Q / 2) / KYBER_Q;

    t = ((int32_t)v * a + (1 << 25)) >> 26;
    t *= KYBER_Q;
    return a - t;
}
```

for AVX2 system

Presented by Seiler*

Algorithm 5 General reduction for signed one word integers

Require: $0 \leq q < \frac{\beta}{2}$, $-\frac{\beta}{2} \leq a < \frac{\beta}{2}$

Ensure: $r \equiv a \pmod{q}$ with $0 \leq r \leq q$

- 1: $v \leftarrow \left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1} \beta}{q} \right\rfloor$ ▷ precomputed
- 2: $t \leftarrow \left\lfloor \frac{av}{2^{\lfloor \log(q) \rfloor - 1} \beta} \right\rfloor$ ▷ signed high product and arithmetic right shift
- 3: $t \leftarrow tq \bmod \beta$ ▷ signed low product
- 4: $r \leftarrow a - t$

참고문헌

Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography, Gregor Seiler, eprint 2018

input : (a : ymm/r)
 (v : ymm1 = floor($2^{26}/q + 0.5$)) $\left\lfloor \frac{2^{\lfloor \log(q) \rfloor - 1} \beta}{q} \right\rfloor$
 (t : tmp)

output : (r : ymm/r)

PQClean AVX2 Code

```
.macro red16 r,rs=0,x=12
    vpmulhw    %ymm1,%ymm\r,%ymm\x    t = av : signed high product
    vpsraw     $10,%ymm\x,%ymm\x      t =  $\left\lfloor \frac{av}{2^{\lfloor \log(q) \rfloor - 1} \beta} \right\rfloor$  : arithmetic right shift
    vpmullw     %ymm0,%ymm\x,%ymm\x    t = tq mod β: signed low product
    vpsubw      %ymm\x,%ymm\r,%ymm\r   r = a - t
.endm
```

16-bit right shift는 high product (vpmulhw)에서 수행되었음

Implementation

AVX2 register : 256-bit
16개의 계수를 동시에 Reduction

❖ Dilithium on AVX2

○ Kyber와 동일하게 핵심 연산에 대해 AVX2를 이용하여 병렬화를 진행

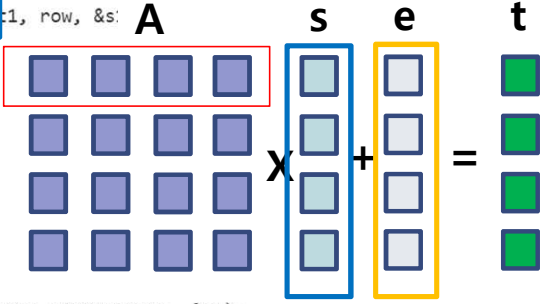
- Kyber의 PQM4와 동일하게 계수단위 구현
 - AVX2이므로 8개의 계수($256 = 8 \times 32$)
 - ref(CPU)에는 적용이 되어있지 않음
- 공개 행렬 A를 행단위로 생성
- Dilithium에 필요한 세부연산들을 AVX2 기반으로 구성
 - AVX2_power2round
 - AVX2_caddq 등

```
int PQCLEAN_DILITHIUM2_AVX2_crypto_sign_keypair(uint8_t *pk, uint8_t *sk) {
    :
    :
    /* Sample short vectors s1 and s2 */
    PQCLEAN_DILITHIUM2_AVX2_poly_uniform_eta_4x(&s1.vec[0], &s1.vec[1], &s1.vec[2], &s1.vec[3], rhoprime, 0, 1, 2, 3);
    PQCLEAN_DILITHIUM2_AVX2_poly_uniform_eta_4x(&s2.vec[0], &s2.vec[1], &s2.vec[2], &s2.vec[3], rhoprime, 4, 5, 6, 7);
    :
    :
    for (i = 0; i < K; i++) {
        /* Expand matrix row */
        polyvec_matrix_expand_row(&row, rowbuf, rho, i);
        /* Compute inner-product */
        PQCLEAN_DILITHIUM2_AVX2_polyvec1_pointwise_acc_montgomery(&t1, row, &s);
        PQCLEAN_DILITHIUM2_AVX2_poly_invntt_tomont(&t1);
        /* Add error polynomial */
        PQCLEAN_DILITHIUM2_AVX2_poly_add(&t1, &t1, &s2.vec[i]);
        /* Round t and pack t1, t0 */
        PQCLEAN_DILITHIUM2_AVX2_poly_caddq(&t1);
        PQCLEAN_DILITHIUM2_AVX2_poly_power2round(&t1, &t0, &t1);
        PQCLEAN_DILITHIUM2_AVX2_polyt1_pack(pk + SEEDBYTES + i * POLYT1_PACKEDBYTES, &t1);
        PQCLEAN_DILITHIUM2_AVX2_polyt0_pack(sk + 3 * SEEDBYTES + (L + K) * POLYETA_PACKEDBYTES + i * POLYT0_PACKEDBYTES, &t0);
    }

    /* Compute H(rho, t1) and store in secret key */
    shake256(sk + 2 * SEEDBYTES, SEEDBYTES, pk, PQCLEAN_DILITHIUM2_AVX2_CRYPTO_PUBLICKEYBYTES);
}
```

rejection Sampling을 병렬적으로 구성 (Kyber와 동일)

행단위로 공개행렬 A 생성



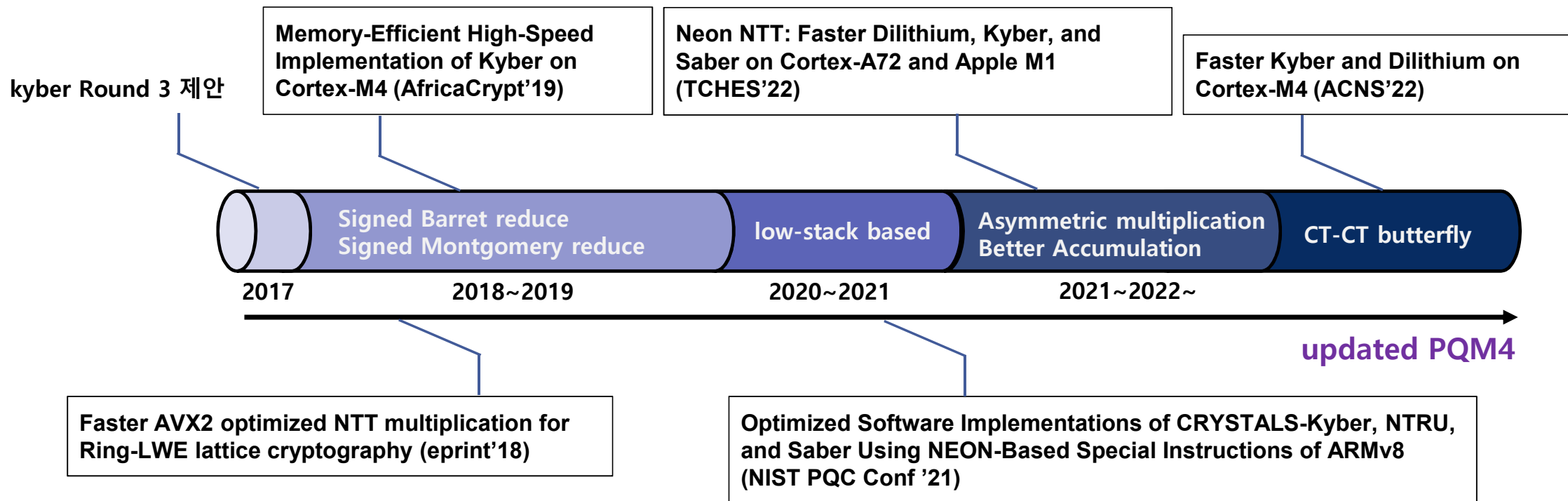
LBC 운영환경별 최적화 사항



❖ Crystals-Kyber 구현동향

CRYSTALS

Cryptographic Suite for Algebraic Lattices



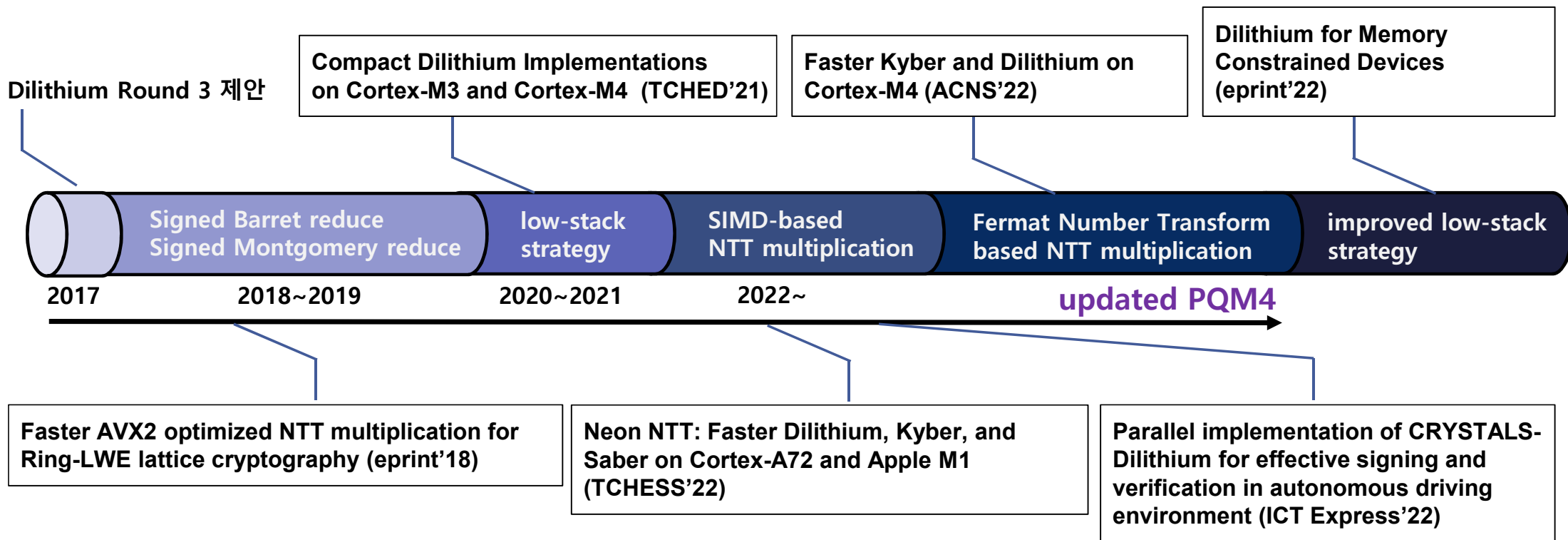
LBC 운영환경별 최적화 사항



❖ Crystals-Dilithium 구현동향

CRYSTALS

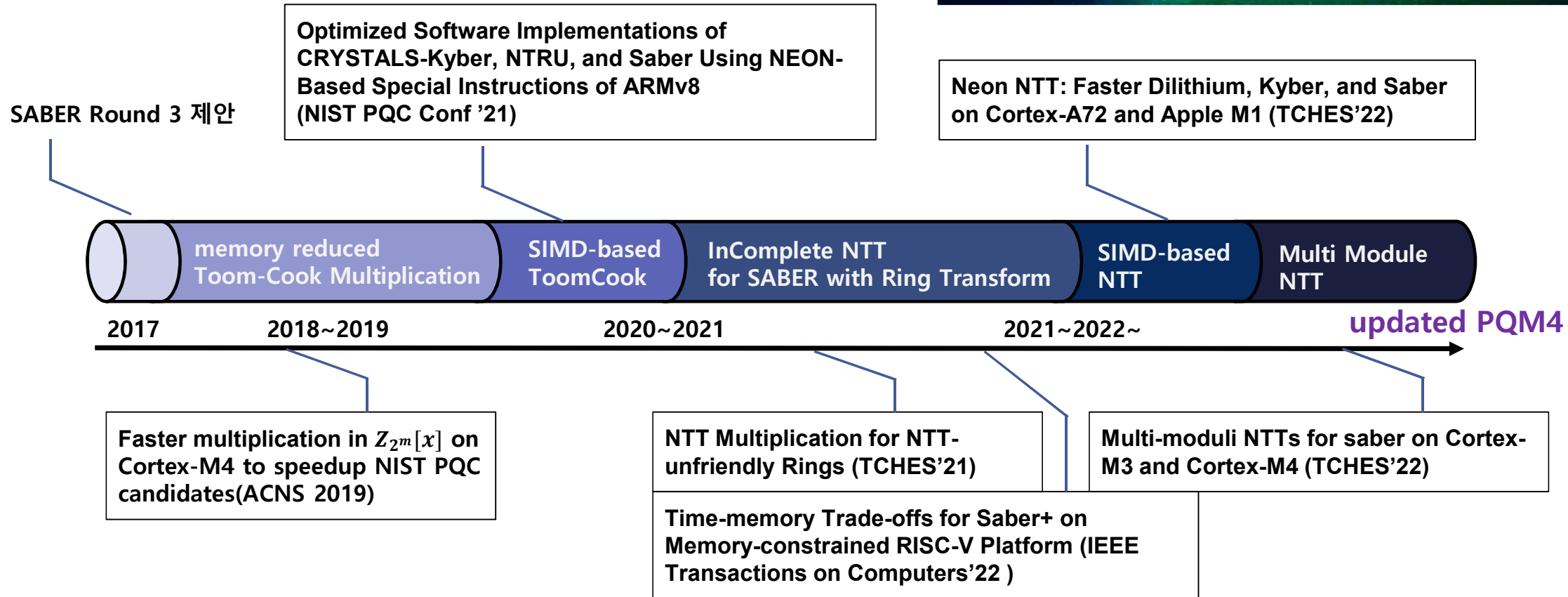
Cryptographic Suite for Algebraic Lattices



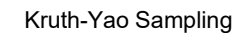
LBC 운영환경별 최적화 사항



❖ SABER 구현동향



- SVP문제 대상의 격자 구성요소 및 LWE/LWR의 공개행렬 및 에러의 원소는 범위를 가지고 있음
 - LWE 문제의 경우 보안성은 에러 e 가 가우시안 분포(정규분포)에서 추출했다는 것에 기인
- 이를 해결하기 위해 가우시안 분포를 따르는 특정 샘플러를 사용하여 구현



❖타겟분포 : (완벽한) 범위가 정해진 이산 가우시안 분포"

○이산 가우시안 분포 $D_{Z,\sigma}$: 평균이 0이고 표준편차가 σ 인 정규분포

○확률질량함수 $\rho_\sigma(x) = e^{-\frac{x^2}{2\sigma^2}}$ for integer $x \in Z$ 을 사용 $\rightarrow D_{Z,\sigma}$ 의 확률은 약 $\frac{\rho_\sigma(x)}{\sqrt{2\pi}\sigma}$

○샘플러가 사용하는 파라미터

- σ : 샘플러의 표준 편차로 평균에서의 데이터 분산을 조정하는 변수
 - Ex) $D_{Z,2.6}$: 평균이 0인 표본들에 대해 샘플러를 사용한다면 약 65%의 확률로 $(-2.6, 2.6)$ 의 범위내 수가 선택됨
- λ : 정밀도 매개변수로, 타겟 분포(이산 가우시안 분포)와 샘플러의 분포의 통계적 거리
 - 일반적으로, λ 에 대해 타겟 분포와의 통계적거리가 $2^{-\lambda}$ 보다 작게 만들
- τ : Tail-Cut 매개변수로 선택한 총 분포에서 무시하고 싶은 분포를 나타냄, 일반적으로 정의역을 제한하는 변수
 - LWE의 에러의 범위 등을 고려
 - 통계적 거리가 $2^{-\lambda}$ 인 경우, $|x| \in \{0, \infty\}$ 대신 $|x| \in \{0, \sigma\tau\}$ 에서 샘플을 추출

❖ “Rejection Sampling”

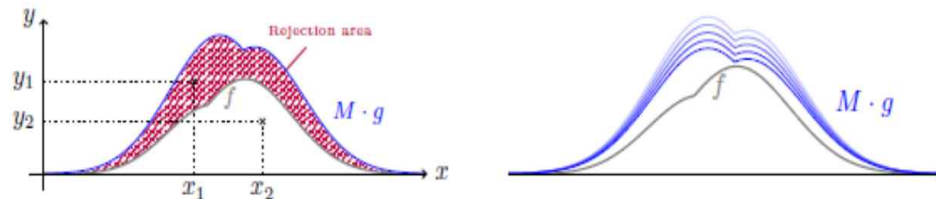
○ 샘플링을 하고싶은 분포 X (타겟 분포)의 확률 질량함수가 $f(x)$ 일 때 확률질량함수가 $g(x)$ 인 다른분포 Y (제안분포) 에서

샘플 x 를 추출하고 이를 $\frac{f(x)}{M g(x)}$ 의 확률을 통해 f 의 샘플로 인정

▪ $M \geq 1$, for all x $f(x) \leq M g(x)$, $|x| \in \{0, \sigma\}$

○ 확률질량함수 $\rho_\sigma(x) = e^{-\frac{x^2}{2\sigma^2}}$ 확률로 accept하여 이산 가우시안의 분포에서 샘플 가능

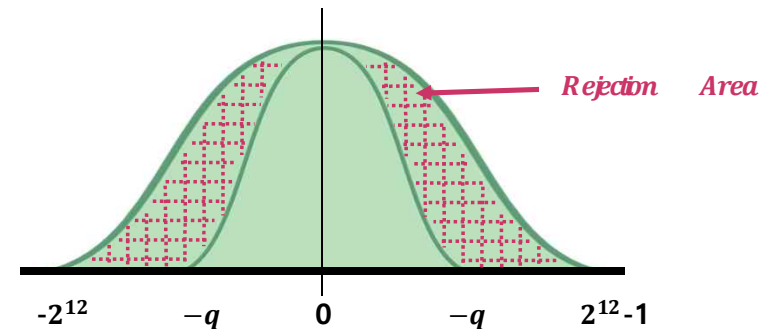
▪ 평균적으로 $\frac{2\tau}{\sqrt{2\pi}}$ 번의 시도를 해야 sampling이 가능 \rightarrow 이를 보완하기 위해 베르누이 샘플링이 제안됨



(a) (x_i, y_i) is sampled uniformly in the area under $M \cdot g$, and accepted when $y_i \leq f(x_i)$

(b) M can be reduced when g is better adapted to f

Rejection Sampling 개요



LWE/LWR에서의 Rejection Sampling

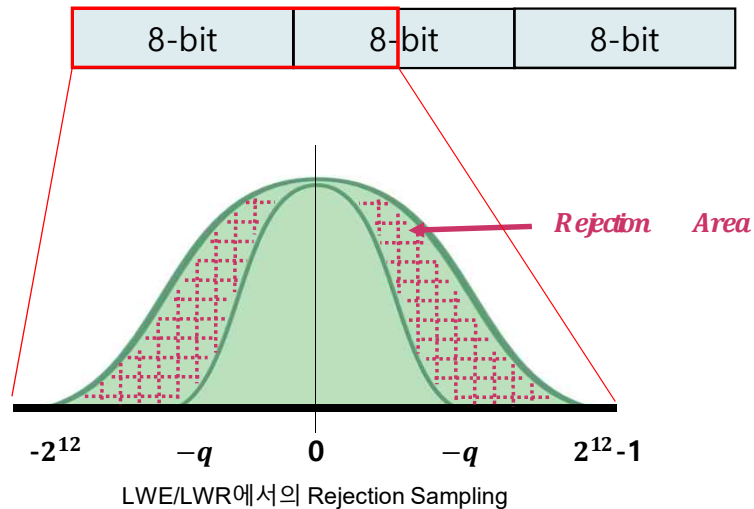
❖ “Rejection Sampling”

○Crystals-Kyber의 경우 q 는 약 12비트

- $R = Z_q[X] / \langle X^{256} + 1 \rangle, q = 3329$

○랜덤 값들을 12비트 단위로 분할 후 q 보다 작으면 → Accept

- 1개 다항식 (256개 계수)를 생성하기 위해 최소, 384 bytes가 필요



```

/*****
 * Name:      rej_uniform
 *
 * Description: Run rejection sampling on uniform random bytes to generate
                uniform random integers mod q
 *
 * Arguments: - int16_t *r:      pointer to output buffer
 *             - unsigned int len: requested number of 16-bit integers
 *                               (uniform mod q)
 *             - const uint8_t *buf: pointer to input buffer
 *                               (assumed to be uniform random bytes)
 *             - unsigned int buflen: length of input buffer in bytes
 *
 * Returns number of sampled 16-bit integers (at most len)
 *****/
static unsigned int rej_uniform(int16_t *r,
                                unsigned int len,
                                const uint8_t *buf,
                                unsigned int buflen)
{
    unsigned int ctr, pos;
    uint16_t val0, val1;

    ctr = pos = 0;
    while(ctr < len && pos + 3 <= buflen) {
        val0 = ((buf[pos+0] >> 0) | ((uint16_t)buf[pos+1] << 8)) & 0xFFF;
        val1 = ((buf[pos+1] >> 4) | ((uint16_t)buf[pos+2] << 4)) & 0xFFF;
        pos += 3;

        if(val0 < KYBER_Q)
            r[ctr++] = val0;
        if(ctr < len && val1 < KYBER_Q)
            r[ctr++] = val1;
    }

    return ctr;
}
    
```

❖ Centered Binomial Distribution (CBD)

○ 샘플링을 하고 싶은 분포 X 확률질량함수

(타겟 분포 : $p_\sigma(x) = p_x = e^{-\frac{x^2}{2\sigma^2}}$, where $N = \tau\sigma \rightarrow D_{Z,\sigma}$ 인 이산 가우시안 분포)

○ $e^{-\frac{x^2}{2\sigma^2}}$ 를 계산하지 않고 사전 테이블을 생성할 필요가 없는 근사 가우스 샘플러

- $\sigma = \sqrt{8}$ 을 가우시안 분포의 표준편차라 한다면 이항분포를 특정 상수로 매개변수화 시킬 수 있음
→ 통계적 거리 λ 를 무시하고 샘플링 가능

○ $\{0, 1\}$ 에서 균일하게 $2 \cdot k$ 의 랜덤 한 값($a_1, \dots, a_k, b_1, \dots, b_k$)을 추출하고, $\sum_{i=1}^k (a_i, b_i)$ 로 샘플링 값을 결정

- $k = 2^\sigma$ 로 선택되므로, 표준 편차가 큰 전자서명에 이항 분포를 사용하는 것은 비효율적
- PKE/KEM의 경우 일반적으로 $\sigma = 2 \sim 3$ 의 작은 값
- DSA의 경우 일반적으로 $\sigma = 215$ 의 큰 값

○ PKE/KEM의 경우 이산 가우스 샘플링을 중심 이항 분포로 대체할 수 있으나,
DSA의 경우는 해당 표준편차에 대해 큰 k 를 선택해야 하므로 비효율적

LBC 정수 연산 감산 방법

❖ Centered Binomial Distribution (CBD)

○ 2η 만큼의 랜덤 비트를 생성하고 조합

- 4-bit값을 조합하여 1개의 계수를 선택해야함
- $4 = 2\eta = 2k = 2^{\sigma+1}$, where $(k = 2^{\sigma})$
- $\eta = 2, \sigma = 1$

○ Dilithium의 LWE 보안의 Reduction은 σ 은

최대 $2^7 = 128$ 이므로 CBD를 이용하는 것은 비효율적

- Rejection Sampling을 이용하여 8-bit로 최소 2개의 계수를 샘플링



바이트 배열을 비트 배열로 변환

2비트 (CBD_2) 또는 3비트 (CBD_3) 단위 씩 그룹핑하여 a 와 b 계산

$f_i = a - b$ 를 연산하여 저장

Sampling from a binomial distribution. Noise in KYBER is sampled from a centered binomial distribution B_η for $\eta = 2$ or $\eta = 3$. We define B_η as follows:

Sample $(a_1, \dots, a_\eta, b_1, \dots, b_\eta) \leftarrow \{0, 1\}^{2\eta}$

and output $\sum_{i=1}^{\eta} (a_i - b_i)$.

When we write that a polynomial $f \in R_q$ or a vector of such polynomials is sampled from B_η , we mean that each coefficient is sampled from B_η .

For the specification of KYBER we need to define how a polynomial $f \in R_q$ is sampled according to B_η deterministically from 64 η bytes of output of a pseudorandom function (we fix $n = 256$ in this description). This is done by the function CBD (for “centered binomial distribution”) defined as described in Algorithm 2.

Algorithm 2 $CBD_\eta: \mathcal{B}^{64\eta} \rightarrow R_q$

Input: Byte array $B = (b_0, b_1, \dots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$

Output: Polynomial $f \in R_q$

$(\beta_0, \dots, \beta_{512\eta-1}) := \text{BytesToBits}(B)$

for i from 0 to 255 **do**

$a := \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$

$b := \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$

$f_i := a - b$

end for

return $f_0 + f_1X + f_2X^2 + \dots + f_{255}X^{255}$

```
static void cbd2(poly *r, const uint8_t buf[2*KYBER_N/4])
```

```
{
```

```
    unsigned int i,j;
```

```
    uint32_t t,d;
```

```
    int16_t a,b;
```

```
    for(i=0;i<KYBER_N/8;i++) {
```

```
        t = load32_littleendian(buf+4*i);
```

```
        d = t & 0x55555555;
```

```
        d += (t>>1) & 0x55555555;
```

```
        for(j=0;j<8;j++) {
```

```
            a = (d >> (4*j+0)) & 0x3;
```

```
            b = (d >> (4*j+2)) & 0x3;
```

```
            r->coeffs[8*i+j] = a - b;
```

32-bit의 홀/짝수 번째 비트들을 분할
각 홀/짝 비트들은 더 해짐 (Sigma)

하나의 계수를 생성하기 위해
총 4-bit가 필요 (ai - bi)

❖LWE/LWR 환경에서의 샘플러 비교

- Rejection Sampling : 느리지만, 테이블을 생성하지않고 엔트로피가 높음 → 임베디드 장치에서 효과적
 - Crystals-Kyber의 공개행렬, Crystals-Dilithium의 공개행렬과 비밀벡터/에러 추출에 사용
 - SABER의 경우 2^{13} 의 modulus를 가지므로 Rejection Sampling을 사용하지 않음
- CBD: 빠르고, 테이블을 생성하지 않지만 전자서명에는 적합하지 않음 → Crystals-Kyber, SABER의 비밀 벡터/에러 추출에 사용
- CDT, Knuth-Yao, Bernoulli, Ziggurat 샘플러는 모두 테이블을 사용하고, 특정 환경/스킴에 특화 되어 있음
 - 테이블 사용은 추후 부채널 분석(타이밍 공격)에 취약할 수 있음

LWE/LWR 환경에서의 샘플러 간의 적합성 비교

Sampler	Speed	FP exp()	Table Size	Table Lookup	Entropy	Features
Rejection	slow	10	0	0	$45+10\log_2\sigma$	suitable for constrained devices
Ziggurat	flexible	flexible	flexible	flexible	flexible	suitable for encryption requires high precision FP arithmetic not suitable for HW implementation
CDT	fast	0	$\sigma\tau\lambda$	$\log_2(\tau\sigma)$	$2.1+\log_2\sigma$	suitable for digital signature easy to implement
Knuth-Yao	fastest	0	$1/2\sigma\tau\lambda$	$\log_2(\sqrt{2\pi e}\sigma)$	$2.1+\log_2\sigma$	not suitable for digital signature
Bernoulli	fast	0	$\lambda\log_2(2.4\tau\sigma^2)$	$\approx \log_2\sigma$	$\approx 6 + 3\log_2\sigma$	suitable for all schemes
Binomial	fast	0	0	0	$4\sigma^2$	not suitable for digital signature

Q&A