KYBER 세미나 진행 상황 보고

24.01.23 류지은

Algorithm 1 Forward NTT of a polynomial $f = c_0 + c_1X \ldots c_{n-1}X^{n-1} \in \mathbb{Z}_q[X]/(X^n+1)$ with precomputed roots of unity $\zeta_k = \zeta^{\text{brv}(k)}$, $0 \le k < n$.

```
k ← 1
for l ← n/2; l > 0; l ← l/2 do
    for s ← 0; s < n; s ← j + l do
        for j ← s; j < s + l; j ← j + 1 do
            t ← ζ_k · c_{j+l}
            c_{j+l} ← c_j − t
            c_j ← c_j + t
        end for
        k ← k + 1
    end for
end for
```

$n = 256$

for $l \leftarrow 128; l \geq 1; l \leftarrow l/2$

$$X^{256}+1 = \prod_{i=0}^{127}(X^2 - \zeta^{2i+1}) = \prod_{i=0}^{127}(X^2 - \zeta^{2\text{br}_7(i)+1}),$$

Algorithm 8 NTT(f)

Computes the NTT representation $\hat{f}$ of the given polynomial $f \in R_q$.

Input: array $f \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the input polynomial

Output: array $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ the coefficients of the NTT of the input polynomial

1: $\hat{f} \leftarrow f$ ▷ will compute NTT in-place on a copy of input array
2: $k \leftarrow 1$
3: **for** ($len \leftarrow 128$; $len \geq 2$; $len \leftarrow len/2$) **FIPS 203에서 수정됨**
4: **for** ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5: $zeta \leftarrow \zeta^{\text{BitRev}_7(k)} \bmod q$
6: $k \leftarrow k + 1$
7: **for** ($j \leftarrow start$; $j < start + len$; j++)
8: $t \leftarrow zeta \cdot \hat{f}[j + len]$
9: $\hat{f}[j + len] \leftarrow \hat{f}[j] - t$
10: $\hat{f}[j] \leftarrow \hat{f}[j] + t$
11: **end for**
12: **end for**
13: **end for**
14: **return** $\hat{f}$

$$X^{256}+1 = \prod_{k=0}^{127}\left(X^2 - \zeta^{2\text{BitRev}_7(k)+1}\right).$$

▷ steps 8-10 done modulo q

Therefore, $R_q := \mathbb{Z}_q[X]/(X^{256}+1)$ is isomorphic to a direct sum of 128 quadratic extension fields of $\mathbb{Z}_q$, denoted T_q. Specifically, this ring has the structure

$$T_q := \bigoplus_{k=0}^{127} \mathbb{Z}_q[X]/\left(X^2 - \zeta^{2\text{BitRev}_7(k)+1}\right). \quad (4.10)$$

Uniform sampling (matrix A)

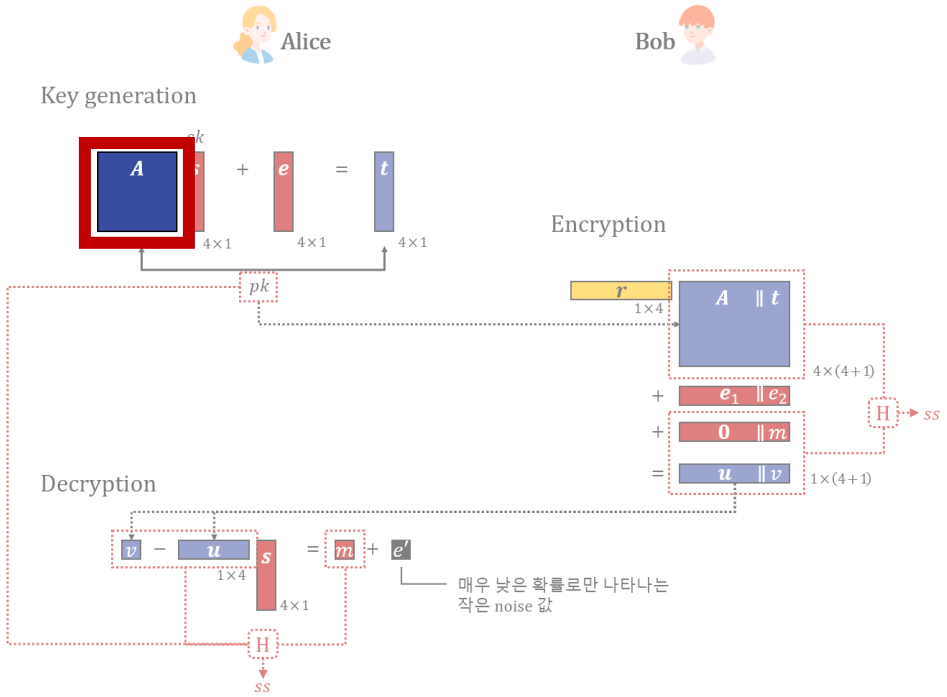
Uniform sampling in R_q. KYBER uses a deterministic approach to sample elements in R_q that are statistically close to a uniformly random distribution. For this sampling we use a function Parse: $\mathcal{B}^* \to R_q$, which receives as input a byte stream $B = b_0, b_1, b_2, \ldots$ and computes the NTT-representation $\hat{a} = \hat{a}_0 + \hat{a}_1 X + \cdots + \hat{a}_{n-1} X^{n-1} \in R_q$ of $a \in R_q$. Parse is described in Algorithm 1 (note that this description assumes that $q = 3329$).

Algorithm 1 Parse: $\mathcal{B}^* \to R_q^n$

Input: Byte stream $B = b_0, b_1, b_2 \cdots \in \mathcal{B}^*$
Output: NTT-representation $\hat{a} \in R_q$ of $a \in R_q$

```
i := 0
j := 0
while j < n do
    d1 := b_i + 256 · (b_{i+1} mod+ 16)
    d2 := ⌊b_{i+1}/16⌋ + 16 · b_{i+2}
    if d1 < q then
        â_j := d1
        j := j + 1
    end if
    if d2 < q and j < n then
        â_j := d2
        j := j + 1
    end if
    i := i + 3
end while
return â_0 + â_1 X + ··· + â_{n-1} X^{n-1}
```

랜덤한 point(어떤 값)를 직접 생성하고
이를 랜덤하게 생성한 다항식으로부터 NTT를 수행한 결과라고 여김



Uniform sampling (sk and error)

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:

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

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

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.

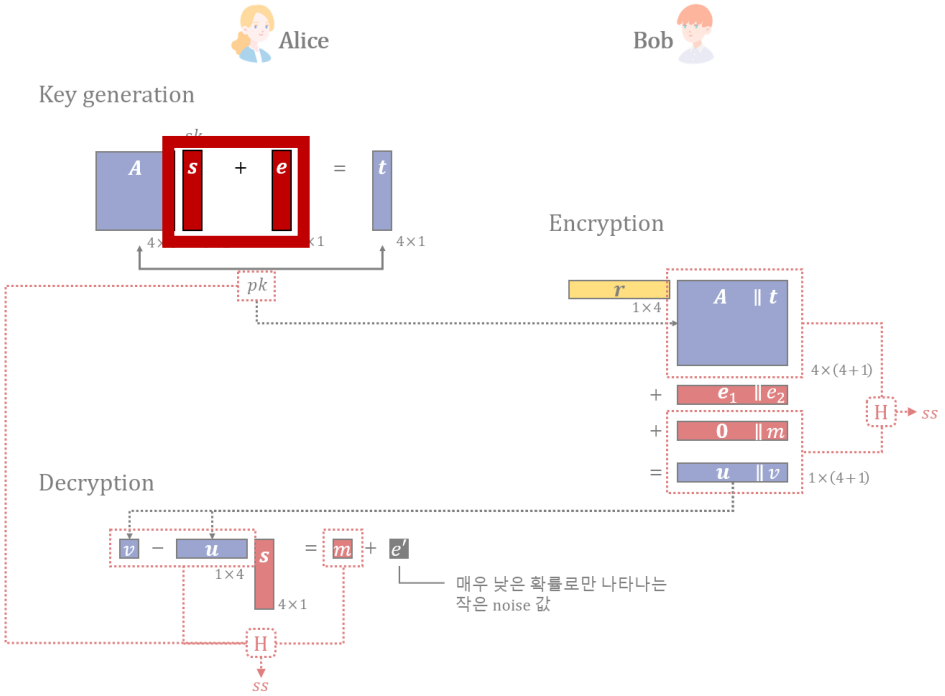
For the specification of KYBER we need to define how a polynomial $f \in R_q$ is sampled according to B_η deterministically from 64η bytes of output of a pseudorandom function (we fix $n = 256$ in this description). This is done by the function CBD (for "centered binomial distribution") defined as described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

Algorithm 2 $\mathsf{CBD}_\eta\colon \mathcal{B}^{64\eta} \to R_q$

Input: Byte array $B = (b_0, b_1, \ldots, b_{64\eta-1}) \in \mathcal{B}^{64\eta}$
Output: Polynomial $f \in R_q$

```
(β_0, ..., β_{512η−1}) := BytesToBits(B)
for i from 0 to 255 do
    a := Σ_{j=0}^{η−1} β_{2iη+j}
    b := Σ_{j=0}^{η−1} β_{2iη+η+j}
    f_i := a − b
end for
return f_0 + f_1 X + f_2 X^2 + ··· + f_255 X^255
```

Uniform distribution에서 sk와 e를 생성



Compression & Decompression

Compression and Decompression. We now define a function $\mathsf{Compress}_q(x, d)$ that takes an element $x \in \mathbb{Z}_q$ and outputs an integer in $\{0, \ldots, 2^d - 1\}$, where $d < \lceil \log_2(q) \rceil$. We furthermore define a function $\mathsf{Decompress}_q$, such that

$$x' = \mathsf{Decompress}_q(\mathsf{Compress}_q(x, d), d) \tag{1}$$

is an element close to x – more specifically

$$|x' - x \bmod^{\pm} q| \leq B_q := \left\lceil \frac{q}{2^{d+1}} \right\rfloor . \tag{2}$$

The functions satisfying these requirements are defined as:

$$\mathsf{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod^{+} 2^d ,$$
$$\mathsf{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor .$$

When $\mathsf{Compress}_q$ or $\mathsf{Decompress}_q$ is used with $x \in R_q$ or $\mathbf{x} \in R_q^k$, the proc coefficient individually.

The main reason for defining the $\mathsf{Compress}_q$ and $\mathsf{Decompress}_q$ functions is to low-order bits in the ciphertext which do not have much effect on the correctness thus reducing the size of ciphertexts.

The $\mathsf{Compress}_q$ and $\mathsf{Decompress}_q$ are also used for a purpose other than compression – namely to perform the usual LWE error correction during encryption and decryption. More precisely, in line 20 of the encryption procedure (Algorithm 5) the $\mathsf{Decompress}_q$ function is used to create error tolerance gaps by sending the message bit 0 to 0 and 1 to $\lceil q/2 \rfloor$. Later, on line 4 of the decryption procedure (Algorithm 6), the $\mathsf{Compress}_q$ function is used to decrypt to a 1 if $v - \mathbf{s}^T\mathbf{u}$ is closer to $\lceil q/2 \rfloor$ than to 0, and decrypt to a 0 otherwise.

복호화 과정에서 생성되는 error를 control

FIPS 203에서 변화 없음

Compression and decompression. Recall that $q = 3329$, and note that the bit length of q is 12. For $d < 12$, define

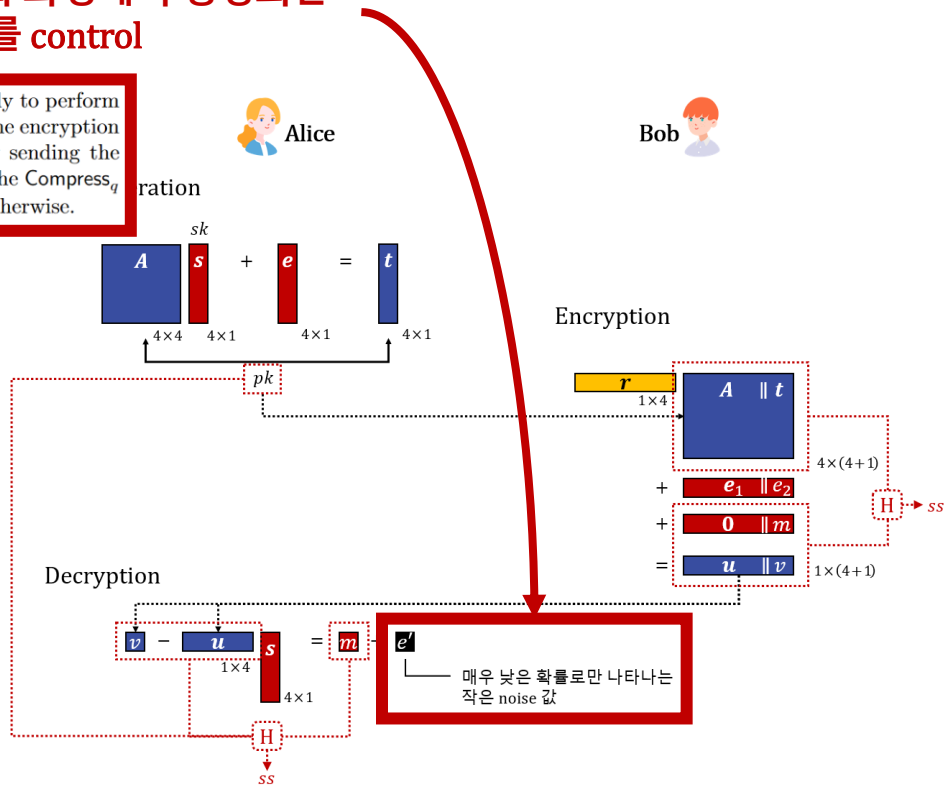
$$\mathsf{Compress}_d : \mathbb{Z}_q \longrightarrow \mathbb{Z}_{2^d} \tag{4.5}$$
$$x \longmapsto \lceil (2^d/q) \cdot x \rfloor .$$
$$\mathsf{Decompress}_d : \mathbb{Z}_{2^d} \longrightarrow \mathbb{Z}_q \tag{4.6}$$
$$y \longmapsto \lceil (q/2^d) \cdot y \rfloor .$$

Note that the input and output types of these functions are integers modulo m (see discussion of types in Section 2.4). Division and rounding in the computation of the above functions are performed in the set of rational numbers. Floating-point computations **should not** be used.

Informally, Compress discards low-order bits of the input, and Decompress adds low-order bits set to zero. These algorithms satisfy two important properties. First, decompression followed by compression preserves the input, that is, $\mathsf{Compress}_d(\mathsf{Decompress}_d(y)) = y$ for all $y \in \mathbb{Z}_q$ and all $d < 12$. Second, if d is large (i.e., close to 12) — meaning that the number of discarded bits is small — compression followed by decompression does not significantly alter the value. Specifically,

$$[\mathsf{Decompress}_d(\mathsf{Compress}_d(x)) - x] \bmod^{\pm} q \ \leq \lceil q/2^{d+1} \rfloor \tag{4.7}$$

for all $x \in \mathbb{Z}_q$ and all $d < 12$.



감사합니다