BIKE 양자 관련

https://youtu.be/pAqFpC4TbHQ

장경배

ClassicMcEliece

- **Classical McEliece**는 이름과 같이, **전통적인 코드 기반 암호의 성격**을 따름
  - 비효율적이라고 볼 수도 있는 오랜 전통의 **Goppa 코드** 사용

NIST

Search CSRC

≡ CSRC MENU

Information Technology Laboratory

COMPUTER SECURITY RESOURCE CENTER

CSRC

UPDATES 2022

PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates

PQC Fourth Round Candidate Key-Establishment Mechanisms (KEMs)

The following candidate KEM algorithms will advance to the fourth round:

Public-Key Encryption/KEMs
BIKE
Classic McEliece
HQC
SIKE

코드기반암호!!

아이소지니기반암호

BIKE와 HQC중 하나는 선정될 것으로 예상

Classic McEliece, SIKE는 지켜보아야 함

BIKE

- **BIKE**는 **QC-MDPC**(Quasi-Cyclic Moderate Density Parity Check) **코드** 사용
  - 생성한 코드가 **순환 이동에 닫혀있으며(Quasi-Cyclic)**,
    → 우측 Rotation된 모든 행들이 Field에 속함, **Why?,** Isomorphism: Circulant matrix 와 R

$$h = \begin{pmatrix} h_0 & h_1 & \dots & h_{r-2} & h_{r-1} \\ h_{r-1} & h_0 & \dots & h_{r-3} & h_{r-2} \\ h_{r-2} & h_{r-1} & \dots & h_{r-4} & h_{r-3} \\ & & \dots & & \\ h_1 & h_2 & \dots & h_{r-1} & h_0 \end{pmatrix}$$

NOTATION	
$\mathbb{F}_2$:	Binary finite field.
$\mathcal{R}$:	Cyclic polynomial ring $\mathbb{F}_2[X]/(X^r - 1)$.

- **각 열의 가중치가 선택되어 똑같은 밀도를 갖도록 함,** **Square matrix**
  → Moderate Density

- **첫 번째 행이 Square matrix를 대표함**
  - **키 사이즈를 줄일 수 있음**

BIKE: KeyGen

- **BIKE KeyGen**
  - h_0, h_1은 개인키, 랜덤 값의 Sparse vector(희소 벡터)를 사용
  - 공개키 $h = h_1 h_0^{-1}$, Field Inversion 사용

$\mathbf{KeyGen} : () \mapsto (h_0, h_1, \sigma), h$

Output: $(h_0, h_1, \sigma) \in \mathcal{H}_w \times \mathcal{M},\ h \in \mathcal{R}$

1: $(h_0, h_1) \xleftarrow{\$} \mathcal{H}_w$

2: $h \leftarrow h_1 h_0^{-1}$

3: $\sigma \xleftarrow{\$} \mathcal{M}$

Decoding 실패 시, 대체되는 쓰레기 값

```
shake256_init(seeds.s1.raw, ELL_SIZE, &h_prng_state);
res = generate_sparse_rep_keccak(h0, DV, R_BITS, &h_prng_state); CHECK_STATUS(res);
res = generate_sparse_rep_keccak(h1, DV, R_BITS, &h_prng_state); CHECK_STATUS(res);
```

```
status_t generate_sparse_rep_keccak(OUT uint8_t * r,
        IN  const uint32_t weight,
        IN  const uint32_t len,
        IN OUT shake256_prng_state_t *prf_state)
{
    uint32_t rand_pos = 0;
    status_t res = SUCCESS;
    uint64_t ctr      = 0;

    //Ensure r is zero.
    setZero(r, DIVIDE_AND_CEIL(len, 8ULL));

    do
    {
        res = get_rand_mod_len_keccak(&rand_pos, len, prf_state);
        CHECK_STATUS(res);


        if (!CHECK_BIT(r, rand_pos))
        {
            ctr++;
            //No collision set the bit
            SET_BIT(r, rand_pos);

        }
    } while(ctr != weight);

    EXIT:
    return res;
```

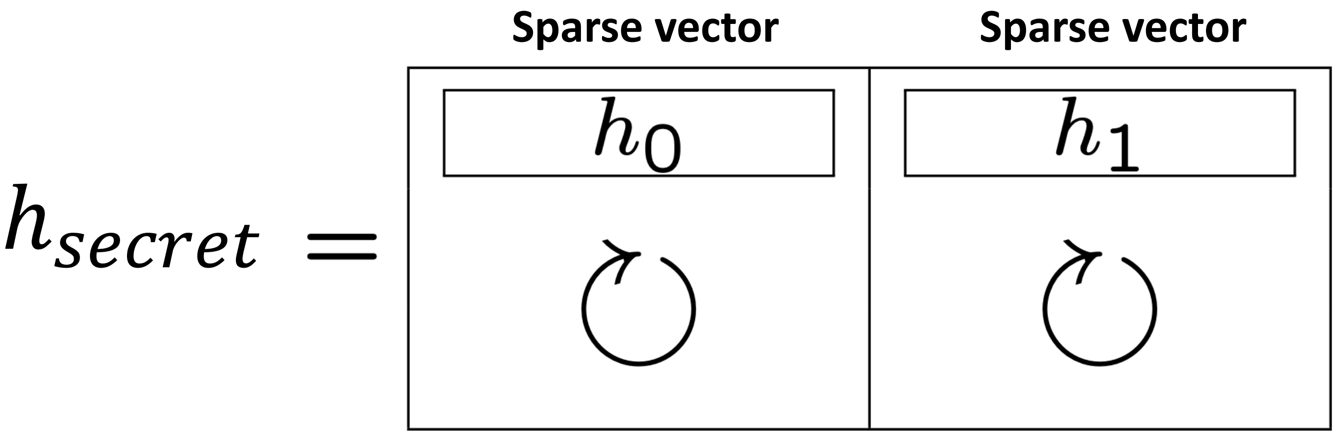
- BIKE-Level 1, 3, 5 파라미터 (Field size가 매우 큼)

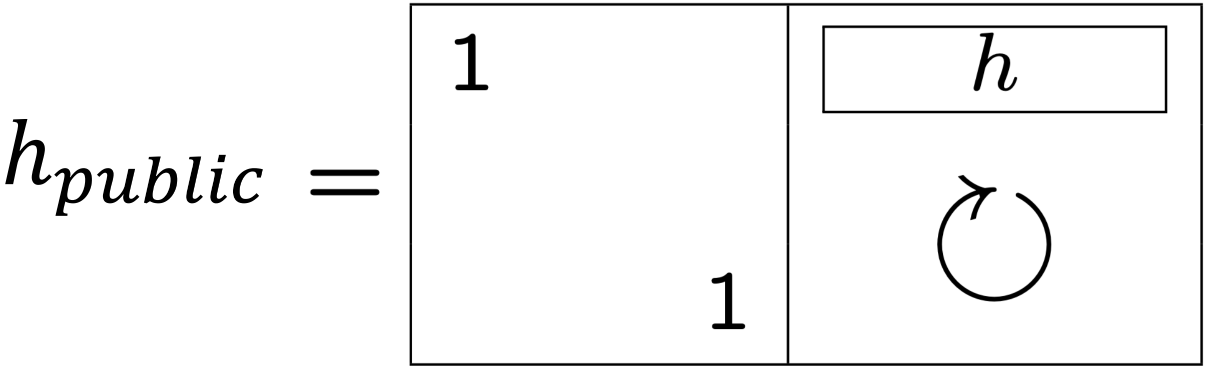
```
// LEVEL-1 security parameters:
#elif defined(PARAM64)
#define R_BITS 12323ULL
#define DV        71ULL
```
Field size
Sparse value

```
// LEVEL-3 Security parameters:
#elif defined(PARAM96)
#define R_BITS 24659ULL
#define DV       103ULL
```

```
// LEVEL-5 Security parameters:
#ifdef PARAM128
#define R_BITS 40973ULL
#define DV       137ULL
```

BIKE: KeyGen






```
// LEVEL-1 security parameters:
#elif defined(PARAM64)
#define R_BITS 12323ULL  Field size
#define DV     71ULL  Sparse value
```

```
// LEVEL-3 Security parameters:
#elif defined(PARAM96)
#define R_BITS 24659ULL
#define DV     103ULL
```

```
// LEVEL-5 Security parameters:
#ifdef PARAM128
#define R_BITS 40973ULL
#define DV     137ULL
```

Quantity	Size	Level 1	Level 3	Level 5
Private key	$\ell + w \cdot \lceil \log_2(r) \rceil$	2,244	3,346	4,640
Public key	r	12,323	24,659	40,973
Ciphertext	$r + \ell$	12,579	24,915	41,229

Table 5: Private Key, Public Key and Ciphertext sizes (in bits).

BIKE: Encapsulation

- **BIKE Encapsulation**
  - 해시함수 (H, K, L)가 자주 사용되며,
  - 동일한 **인코딩(신드롬 계산)**이 사용됨 → $e_1 h$

Encaps : $h \mapsto K, c$

Input: $h \in \mathcal{R}$

Output: $K \in \mathcal{K}$, $c \in \mathcal{R} \times \mathcal{M}$

1: $m \xleftarrow{\$} \mathcal{M}$

2: $(e_0, e_1) \leftarrow \mathbf{H}(m)$

3: $c \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$

4: $K \leftarrow \mathbf{K}(m, c)$

2.5 The Functions $\mathbf{H}, \mathbf{K}, \mathbf{L}$

The functions $\mathbf{H}, \mathbf{K}, \mathbf{L}$ are modeled as random oracles. Their concrete instantiation is the following.

- $\mathbf{H}$ is instantiated as a pseudorandom expansion of a seed of length ℓ bits that is input to the function. It is generated by invoking Algorithm 3 with the appropriate parameters. The function uses the Extendable-Output Function SHAKE256 to generate a stream of pseudorandom bits.
- $\mathbf{K}$ is instantiated as the $\ell = 256$ least significant bits of the standard SHA3-384 hash digest of the input. The notation $\mathbf{K}(m, C)$ where $C = (c_0, c_1)$ (and similarly, $\mathbf{K}(m', C)$) refers to hashing an input of $\{0,1\}^{\ell+r+\ell}$ bits that is the concatenation of m, c_0 and c_1. Here, the bits of m are consumed (by SHA3-384) first, then the bits of c_0, and then the bits of c_1.
- $\mathbf{L}$ is instantiated as the $\ell = 256$ least significant bits of the standard SHA3-384 hash digest of the input. The notation $\mathbf{L}(e_0, e_1)$ (and similarly, $\mathbf{L}(e'_0, e'_1)$) refers to hashing an input of $\{0,1\}^{r+r}$ bits that is the concatenation of e_0 and e_1. Here, the bits of e_0 are consumed (by SHA3-384) first, and then the bits of e_1.

BIKE: Decapsulation

- **BIKE Decapsulation**
  - Decoder를 사용해 Encapsulation에서와의 동일한 세션키 K 생성
  - 디코딩 실패 시, 쓰레기 값 σ를 해시 함
  - 디코딩은 더 봐야함, **Black-Gray-Flip (BGP) Decoder**

Decaps : $(h_0, h_1, \sigma), c \mapsto K$

Input: $((h_0, h_1), \sigma) \in \mathcal{H}_w \times \mathcal{M}$, $c = (c_0, c_1) \in \mathcal{R} \times \mathcal{M}$

Output: $K \in \mathcal{K}$

1: $e' \leftarrow \texttt{decoder}(c_0 h_0, h_0, h_1)$ $\quad \triangleright\ e' \in \mathcal{R}^2 \cup \{\perp\}$

2: $m' \leftarrow c_1 \oplus \mathbf{L}(e')$ $\quad \triangleright$ with the convention $\perp = (0, 0)$

3: **if** $e' = \mathbf{H}(m')$ **then** $K \leftarrow \mathbf{K}(m', c)$ **else** $\boxed{K \leftarrow \mathbf{K}(\sigma, c)}$

BIKE: Future work

- **Encoding은 matrix, vector 곱셈**, → 동일한 Linear matrix 분해 후 양자 구현 이지만 **Circulant matrix에 따른 차이점**이 있을지?

- 곱셈은 Circulant matrix에 따른 차이점 없을 듯?
- 곱셈기는 확장해 둔 상태,
  - **필드 사이즈가 매우 커서** 시뮬레이션이 너무 오래걸림

- **BIKE에 대한 암호 분석 알고리즘?** (Ex: Information Set Decoding)

```
def Karatsuba_Toffoli_Depth_1(eng) :

    n = 12323
    a = eng.allocate_qureg(n)
    b = eng.allocate_qureg(n)

    if (resource_check != 1):
        Round_constant_XOR(eng, a, 0xabcdabcdabcdabcdabcdabcdab
        Round_constant_XOR(eng, b, 0xabcdabcdabcdabcdabcdabcdab

    result = []
    count = 0
    result, count = recursive_karatsuba(eng, a, b, n, count)

    print_state(eng, result, 2*n-1)
    print(count)

    Mul_test(0xabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabcdabc
```

감사합니다