HQC Reed-Solomon 인코딩 양자 회로 구현

임세진

https://youtu.be/O9SzEOEZAns

Contents

CryptoCraft LAB

01. Reed-Solomon 코드 분석

- HQC Encrypt에서 암호문 v를 구하는 연산 중 mG 연산이 Reed-Solomon 코드의 인코딩을 거쳐 수행됨

- $\mathsf{Setup}(1^\lambda)$: generates and outputs the global parameters $\mathsf{param} = (n, k, \Delta, w, w_{\mathbf{r}}, w_{\mathbf{e}})$.
- $\mathsf{KeyGen}(\mathsf{param})$: samples $\mathbf{h} \xleftarrow{\$} \mathcal{R}$, the generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n}$ of $\mathcal{C}$, $(\mathbf{x}, \mathbf{y}) \xleftarrow{\$} \mathcal{R}_w \times \mathcal{R}_w$, sets $\mathsf{sk} = (\mathbf{x}, \mathbf{y})$ and $\mathsf{pk} = (\mathbf{h}, \mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y})$, and returns $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Encrypt}(\mathsf{pk}, \mathbf{m})$: generates $\mathbf{e} \xleftarrow{\$} \mathcal{R}_{w_{\mathbf{e}}}$, $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2) \xleftarrow{\$} \mathcal{R}_{w_{\mathbf{r}}} \times \mathcal{R}_{w_{\mathbf{r}}}$, sets $\mathbf{u} = \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$ and $\mathbf{v} = \mathbf{mG} + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$, returns $\mathbf{c} = (\mathbf{u}, \mathbf{v})$.
- $\mathsf{Decrypt}(\mathsf{sk}, \mathbf{c})$: returns $\mathcal{C}.\mathsf{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y})$.

Figure 2: Description of our proposal HQC.PKE.

01. Reed-Solomon 코드 분석

- Reed-Solomon 코드에서 바이너리 필드 곱셈이 수행되는데, 2^{17668}에서 수행되는 대부분의 연산들과 달리 2^8에서 연산을 수행함 → 바이너리 필드 곱셈 구현 필요

In our case, we will be working in $\mathbb{F}_{2^m}$ with $m = 8$. To do so, we use the primitive polynomial $1 + x^2 + x^3 + x^4 + x^8$ of degree 8 to build this field (polynomial from [21]). We denote by $g_1(x)$, $g_2(x)$ and $g_3(x)$ the generator polynomials of RS-S1, RS-S2 and RS-S3 respectively, which are equal to the generator polynomials of Reed-Solomon codes RS-1, RS-2 and RS-3 respectively. We precomputed the generator polynomials $g_1(x)$, $g_2(x)$ and $g_3(x)$ of the code RS-S1, RS-S2 and RS-S3 and we included them in the file `parameters.h`. One can use the functions provided in the file `reed_solomon.h` to reconstruct the generator polynomials for those codes.

Generator polynomial of RS-1. $g_1(x) = 9 + 69x + 153x^2 + 116x^3 + 176x^4 + 117x^5 + 111x^6 + 75x^7 + 73x^8 + 233x^9 + 242x^{10} + 233x^{11} + 65x^{12} + 210x^{13} + 21x^{14} + 139x^{15} + 103x^{16} + 173x^{17} + 67x^{18} + 118x^{19} + 105x^{20} + 210x^{21} + 174x^{22} + 110x^{23} + 74x^{24} + 69x^{25} + 228x^{26} + 82x^{27} + 255x^{28} + 181x^{29} + x^{30}$.

hqc-128에서 사용되는 생성 다항식

```
#define RS_POLY_COEFS
    9,69,153,116,176,117,111,75,73,233,242,233,65,210,21,139,103,173,67,118,105,210
    ,174,110,74,69,228,82,255,181,1
```

- 계수를 저장하는 배열 요소의 크기가 16비트지만, 계수는 모두 8비트로 표현 가능한 2^8 미만의 숫자들임
- 메시지 배열 요소의 크기도, gf_mul의 input인 gate_value도 8비트임
→ 8비트 X 8비트 바이너리 필드 곱셈 구현

```
void reed_solomon_encode(uint64_t *cdw, const uint64_t *msg) {
    size_t i, j, k;
    uint8_t gate_value = 0;

    uint16_t tmp[PARAM_G] = {0};
    uint16_t PARAM_RS_POLY [] = {RS_POLY_COEFS};

    uint8_t msg_bytes[PARAM_K] = {0};
    uint8_t cdw_bytes[PARAM_N1] = {0};

    memcpy(msg_bytes, msg, PARAM_K);

    for (i = 0; i < PARAM_K; ++i) { // 16
        gate_value = msg_bytes[PARAM_K - 1 - i] ^ cdw_bytes[PARAM_N1 - PARAM_K - 1]; // 29

        for (j = 0; j < PARAM_G; ++j) { // 31
            tmp[j] = gf_mul(gate_value, PARAM_RS_POLY[j]);
        }

        for(k = PARAM_N1 - PARAM_K - 1; k; --k) { // 46-16-1
            cdw_bytes[k] = cdw_bytes[k - 1] ^ tmp[k];
        }

        cdw_bytes[0] = tmp[0];
    }

    memcpy(cdw_bytes + PARAM_N1 - PARAM_K, msg_bytes, PARAM_K);
    memcpy(cdw, cdw_bytes, PARAM_N1);
}
```

RS-1 다항식의 계수와 메시지의 곱

01. Reed-Solomon 코드 분석

- Reed-Solomon_encode : 메시지 u와 생성 다항식 $g(x)$의 연산을 통해 코드 워드 $c(x)$를 구하는 과정
- 코드 워드는 데이터 (맨 뒤에서부터 k개)와 오류 정정 비트 (맨 앞에서부터 $n-k$개) 로 구성된 블록임

2.5.3 Encoding shortened Reed-Solomon codes

In the following we present the encoding of Reed-Solomon codes which can also be used to encode shortened Reed-Solomon codes. We denote by $u(x) = u_0 + \cdots + u_{k-1}x^{k-1}$ the polynomial corresponding to the message $u = (u_0, \cdots, u_{k-1})$ to be encoded and $g(x)$ the generator polynomial. We use the systematic form of encoding where the rightmost k elements of the code word polynomial are the message bits and the leftmost $n-k$ bits are the parity-check bits. Following [21], the code word is given by $c(x) = b(x)+x^{n-k}u(x)$, where $b(x)$ is the reminder of the division of the polynomial $x^{n-k}u(x)$ by $g(x)$. In consequence, the encoding in systematic form consists of three steps :

1. Multiply the message $u(x)$ by x^{n-k}.
2. Compute the remainder $b(x)$ by dividing $x^{n-k}u(x)$ by the generator polynomial $g(x)$.
3. Combine $b(x)$ and $x^{n-k}u(x)$ to obtain the code polynomial $c(x) = b(x) + x^{n-k}u(x)$.

```c
 */
void reed_solomon_encode(uint64_t *cdw, const uint64_t *msg) {
    size_t i, j, k;
    uint8_t gate_value = 0;

    uint16_t tmp[PARAM_G] = {0};
    uint16_t PARAM_RS_POLY [] = {RS_POLY_COEFS};

    uint8_t msg_bytes[PARAM_K] = {0};
    uint8_t cdw_bytes[PARAM_N1] = {0};

    memcpy(msg_bytes, msg, PARAM_K);

    for (i = 0; i < PARAM_K; ++i) { // 16
        gate_value = msg_bytes[PARAM_K - 1 - i] ^ cdw_bytes[PARAM_N1 - PARAM_K - 1]; // 29

        for (j = 0; j < PARAM_G; ++j) { // 31
            tmp[j] = gf_mul(gate_value, PARAM_RS_POLY[j]);
        }

        for(k = PARAM_N1 - PARAM_K - 1; k; --k) { // 46-16-1
            cdw_bytes[k] = cdw_bytes[k - 1] ^ tmp[k];
        }

        cdw_bytes[0] = tmp[0];
    }

    memcpy(cdw_bytes + PARAM_N1 - PARAM_K, msg_bytes, PARAM_K);
    memcpy(cdw, cdw_bytes, PARAM_N1);
}
```

02. Reed-Solomon 인코딩 양자 회로 구현

```python
result = combine(eng, new_a, new_b, new_r, n)

# modular
#######
CNOT | (result[8], result[4])
CNOT | (result[8], result[3])
CNOT | (result[8], result[2])
CNOT | (result[8], result[0])

CNOT | (result[9], result[5])
CNOT | (result[9], result[4])
CNOT | (result[9], result[3])
CNOT | (result[9], result[1])

CNOT | (result[10], result[6])
CNOT | (result[10], result[5])
CNOT | (result[10], result[4])
CNOT | (result[10], result[2])

CNOT | (result[11], result[7])
CNOT | (result[11], result[6])
CNOT | (result[11], result[5])
CNOT | (result[11], result[3])

CNOT | (result[12], result[7])
CNOT | (result[12], result[6])
CNOT | (result[12], result[3])
CNOT | (result[12], result[2])
CNOT | (result[12], result[1])

CNOT | (result[13], result[7])
CNOT | (result[13], result[2])
CNOT | (result[13], result[1])
CNOT | (result[13], result[0])

CNOT | (result[14], result[4])
CNOT | (result[14], result[1])
CNOT | (result[14], result[0])

return result
```

- 8비트 X 8비트 바이너리 필드 곱셈 modular 연산 구현
- 8비트 X 8비트 = 15비트 (x^{14}까지 처리해줘야 됨)
- $x^8 + x^4 + x^3 + x^2 + 1$

$$x^8 = x^4 + x^3 + x^2 + 1$$
$$x^9 = x^5 + x^4 + x^3 + x$$
$$x^{10} = x^6 + x^5 + x^4 + x^2$$
$$x^{11} = x^7 + x^6 + x^5 + x^3$$
$$x^{12} = \cancel{x^8} + x^7 + x^6 + \cancel{x^4} + \cancel{x^4} + x^3 + x^2 + 1 = x^7 + x^6 + x^3 + x^2 + 1$$
$$x^{13} = \cancel{x^8} + x^7 + \cancel{x^4} + \cancel{x^3} + x + \cancel{x^4} + \cancel{x^3} + x^2 + 1 = x^7 + x^2 + x + 1$$
$$x^{14} = \cancel{x^8} + \cancel{x^3} + \cancel{x^2} + x + x^4 + \cancel{x^3} + \cancel{x^2} + 1 = x^4 + x + 1$$

02. Reed-Solomon 인코딩 양자 회로 구현

- 양자 회로에서는 0,1로만 표현할 수 있으므로 ex) uint8_t 배열[16] → 큐비트 배열[16][8]
- C코드와 비교하여 실제로 사용되는 값만 큐비트 할당 → tmp[30]은 연산에 사용되지 않아서 tmp[30], PARAM_RS_POLY[30], gabage[29] 큐비트는 할당하지 않음 + 곱셈 연산 30회만 수행

```python
r_a_ = []
r_b_ = []
for i in range(30):
    r_a_.append(eng.allocate_qureg(int(n / 2)))
    r_b_.append(eng.allocate_qureg(int(n / 2)))

### Reed_solomon_encode ###
msg = []
for i in range(16):
    msg.append(eng.allocate_qureg(8))

cdw_bytes = []
for i in range(46):
    cdw_bytes.append(eng.allocate_qureg(8))

gate_value = []
for i in range(16):
    gate_value.append(eng.allocate_qureg(8))

PARAM_RS_POLY = []
for i in range(30):
    PARAM_RS_POLY.append(eng.allocate_qureg(8))

gabage = []  # gate_value 값 복사용 // tmp[29]까지만 쓰길래 gate_value도 1개 적게 구함
for i in range(29):
    gabage.append(eng.allocate_qureg(8))
```

C코드

```c
void reed_solomon_encode(uint64_t *cdw, const uint64_t *msg) {
    size_t i, j, k;
    uint8_t gate_value = 0;

    uint16_t tmp[PARAM_G] = {0};
    uint16_t PARAM_RS_POLY [] = {RS_POLY_COEFS};

    uint8_t msg_bytes[PARAM_K] = {0};
    uint8_t cdw_bytes[PARAM_N1] = {0};

    memcpy(msg_bytes, msg, PARAM_K);

    for (i = 0; i < PARAM_K; ++i) { // 16
        gate_value = msg_bytes[PARAM_K - 1 - i] ^ cdw_bytes[PARAM_N1 - PARAM_K - 1]; // 29

        for (j = 0; j < PARAM_G; ++j) { // 31
            tmp[j] = gf_mul(gate_value, PARAM_RS_POLY[j]);
        }

        for(k = PARAM_N1 - PARAM_K - 1; k; --k) { // 46-16-1
            cdw_bytes[k] = cdw_bytes[k - 1] ^ tmp[k];
        }

        cdw_bytes[0] = tmp[0];
    }

    memcpy(cdw_bytes + PARAM_N1 - PARAM_K, msg_bytes, PARAM_K);
    memcpy(cdw, cdw_bytes, PARAM_N1);
}
```

02. Reed-Solomon 인코딩 양자 회로 구현

- 반복문 한 번 당 수행되는 30번의 곱셈을 동시에 병렬로 수행 → Toffoli-depth 1
  - gate_value 값을 29개의 gabage 큐비트에 복사 (복사 시 depth를 줄이기 위해 아래와 같이 구현)

```python
for i in range(16):
    for j in range(8):  # gate_value
        CNOT | (cdw_bytes[29][j], gate_value[i][j])
        CNOT | (msg[15 - i][j], gate_value[i][j])

    # gate_value Copy
    with Compute(eng):
        Copy(eng, gate_value[i], gabage[0], 8)

        Copy(eng, gate_value[i], gabage[1], 8)
        Copy(eng, gabage[0], gabage[2], 8)

        Copy(eng, gate_value[i], gabage[3], 8)
        Copy(eng, gabage[0], gabage[4], 8)
        Copy(eng, gabage[1], gabage[5], 8)
        Copy(eng, gabage[2], gabage[6], 8)

        Copy(eng, gate_value[i], gabage[7], 8)
        Copy(eng, gabage[0], gabage[8], 8)
        Copy(eng, gabage[1], gabage[9], 8)
        Copy(eng, gabage[2], gabage[10], 8)
        Copy(eng, gabage[3], gabage[11], 8)
        Copy(eng, gabage[4], gabage[12], 8)
        Copy(eng, gabage[5], gabage[13], 8)
        Copy(eng, gabage[6], gabage[14], 8)
```

```python
Copy(eng, gate_value[i], gabage[15], 8)
Copy(eng, gabage[0], gabage[16], 8)
Copy(eng, gabage[1], gabage[17], 8)
Copy(eng, gabage[2], gabage[18], 8)
Copy(eng, gabage[3], gabage[19], 8)
Copy(eng, gabage[4], gabage[20], 8)
Copy(eng, gabage[5], gabage[21], 8)
Copy(eng, gabage[6], gabage[22], 8)
Copy(eng, gabage[7], gabage[23], 8)
Copy(eng, gabage[8], gabage[24], 8)
Copy(eng, gabage[9], gabage[25], 8)
Copy(eng, gabage[10], gabage[26], 8)
Copy(eng, gabage[11], gabage[27], 8)
Copy(eng, gabage[12], gabage[28], 8)
```

```c
void reed_solomon_encode(uint64_t *cdw, const uint64_t *msg) {
    size_t i, j, k;
    uint8_t gate_value = 0;

    uint16_t tmp[PARAM_G] = {0};
    uint16_t PARAM_RS_POLY [] = {RS_POLY_COEFS};

    uint8_t msg_bytes[PARAM_K] = {0};
    uint8_t cdw_bytes[PARAM_N1] = {0};

    memcpy(msg_bytes, msg, PARAM_K);

    for (i = 0; i < PARAM_K; ++i) { // 16
        gate_value = msg_bytes[PARAM_K - 1 - i] ^ cdw_bytes[PARAM_N1 - PARAM_K - 1]; // 29

        for (j = 0; j < PARAM_G; ++j) { // 31
            tmp[j] = gf_mul(gate_value, PARAM_RS_POLY[j]);
        }

        for(k = PARAM_N1 - PARAM_K - 1; k; --k) { // 46-16-1
            cdw_bytes[k] = cdw_bytes[k - 1] ^ tmp[k];
        }

        cdw_bytes[0] = tmp[0];
    }

    memcpy(cdw_bytes + PARAM_N1 - PARAM_K, msg_bytes, PARAM_K);
    memcpy(cdw, cdw_bytes, PARAM_N1);
}
```

02. Reed-Solomon 인코딩 양자 회로 구현

- 반복문 한 번 당 수행되는 30번의 곱셈을 동시에 병렬로 수행 → Toffoli-depth 1
  - gate_value 값을 29개의 gabage 큐비트에 복사 (복사 시 depth를 줄이기 위해 아래와 같이 구현)
  - 곱셈 연산에 필요한 ancilla 큐비트는 독립적으로 사용
  - gabage 큐비트는 reverse 연산을 통해 다음 반복문에서 재활용 가능

```python
    tmp_mul = []
    tmp = Karatsuba_8_Toffoli_Depth_1(eng, gate_value[i], PARAM_RS_POLY[0], r_a_[0], r_b_[0])
    tmp_mul.append(tmp)
    for j in range(29):  ### gf_mul
        tmp = Karatsuba_8_Toffoli_Depth_1(eng, gabage[j], PARAM_RS_POLY[j+1], r_a_[j+1],
            r_b_[j+1])
        tmp_mul.append(tmp)

    for j in range(29, 0, -1):
        # cdw_bytes[k] = cdw_bytes[k - 1] ^ tmp[k];
        Copy(eng, cdw_bytes[j - 1], cdw_bytes[j],8)
        Copy(eng, tmp_mul[j], cdw_bytes[j],8)

    Copy(eng, tmp_mul[0], cdw_bytes[0],8)

    ### reverse (gate_value copy 부분만) ###
    Uncompute(eng)

for i in range(16):  # memcpy(cdw_bytes + PARAM_N1 - PARAM_K, msg_bytes, PARAM_K);
    for j in range(8):
        CNOT | (msg[i][j], cdw_bytes[i + 30][j])
```

```c
void reed_solomon_encode(uint64_t *cdw, const uint64_t *msg) {
    size_t i, j, k;
    uint8_t gate_value = 0;

    uint16_t tmp[PARAM_G] = {0};
    uint16_t PARAM_RS_POLY [] = {RS_POLY_COEFS};

    uint8_t msg_bytes[PARAM_K] = {0};
    uint8_t cdw_bytes[PARAM_N1] = {0};

    memcpy(msg_bytes, msg, PARAM_K);

    for (i = 0; i < PARAM_K; ++i) { // 16
        gate_value = msg_bytes[PARAM_K - 1 - i] ^ cdw_bytes[PARAM_N1 - PARAM_K - 1]; // 29

        for (j = 0; j < PARAM_G; ++j) { // 31
            tmp[j] = gf_mul(gate_value, PARAM_RS_POLY[j]);
        }

        for(k = PARAM_N1 - PARAM_K - 1; k; --k) { // 46-16-1
            cdw_bytes[k] = cdw_bytes[k - 1] ^ tmp[k];
        }

        cdw_bytes[0] = tmp[0];
    }

    memcpy(cdw_bytes + PARAM_N1 - PARAM_K, msg_bytes, PARAM_K);
    memcpy(cdw, cdw_bytes, PARAM_N1);
}
```

02. Reed-Solomon 인코딩 양자 회로 구현

- 자원 측정 결과 → 사용된 게이트 수에 비해 Toffoli-depth와 Full-depth가 낮게 최적화 되어 구현됨

Field	Arithmetic	Qubits	CNOT gates	Toffoli gates	Toffoli depth	Full depth
$\mathbb{F}_{2^8}$	Multiplication	81	164	27	1	26

Reed-Solomon	Qubits	CNOT gates	Toffoli gates	Toffoli depth	Full depth
hqc-128	28,696	94,320	12,960	16	545

감사합니다