Classic McEliece 양자 구현

장경배

https://youtu.be/hkpB7t7QRqE

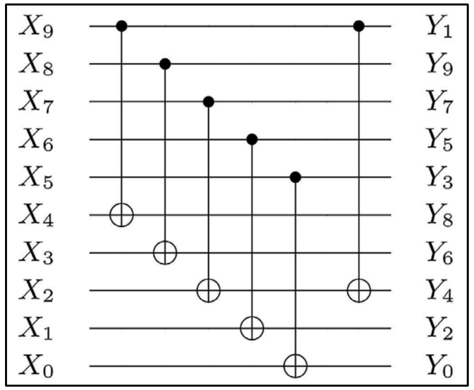
한성대학교
HANSUNG UNIVERSITY

CryptoCraft LAB

Code 기반 PQC 양자 구현

- **구현 타겟 (Classic McEliece 기준)**
  - **Key pair의 핵심 연산**

    → **Binary Field 산술** : 곱셈 및 역치 연산 (Itoh-Tsuji)
    양자 제곱(Squaring) 연산은 비교적 간단, 개선의 여지 거의 X,
    → 결국 **양자 곱셈기 (최적화 타겟)**



< $F_{2^8}/(x^{10}+x^3+1)$ Squaring 양자

  - **Encryption (Encoding)의 핵심 연산**

    → **행렬 벡터 곱** : 특이 기법 없이 단순 구현, **Toffoli 게이트만이 사용되어 구현됨**

  - **Decryption (Decoding)의 핵심 연산**

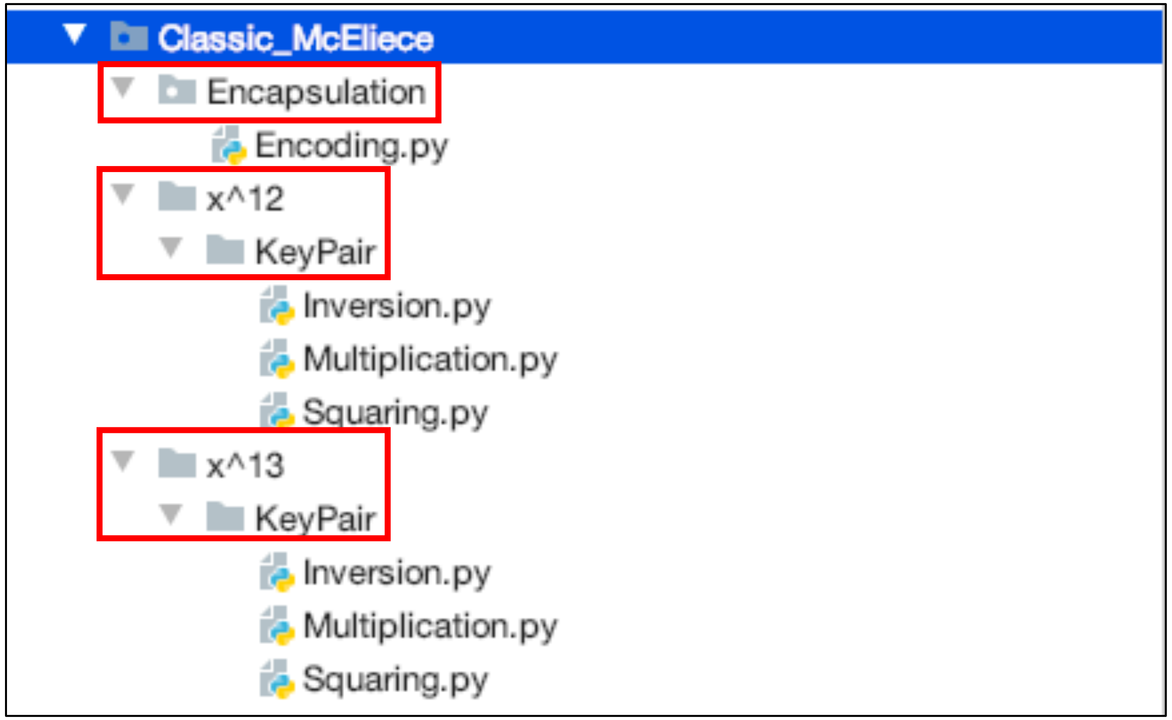
    → **신드롬 디코딩 알고리즘(Berlekamp–Massey)** : **양자로 구현 필요성 ??**
    (양자 컴퓨터를 사용한 복호화?, 양자 brute-force에도 필요 X, 양자 ISD에서도 필요 X)

  - **Encapsulation, Decapsulation의 핵심 연산**

    → **해시 함수 SHAKE256**

코드기반 암호의 안전성 분석에는, ISD 알고리즘 구현이 적합

Code 기반 PQC 양자 구현



- **Encapsulation**
  - **Encoding → Matrix × Vector**

- $\mathbb{F}_{2^{12}}/(\boldsymbol{x^{12}+x^{3}+1})$
  - **Keypair**
    - Multiplication
    - Squaring
    - Inversion (Multiplication + Squaring)

- $\mathbb{F}_{2^{13}}/(\boldsymbol{x^{13}+x^{4}+x^{3}+x^{1}+1})$
  - **Keypair**
    - Multiplication
    - Squaring
    - Inversion (Multiplication + Squaring)

Code 기반 PQC 양자 구현: Encoding

- **Quantum – Quantum**
  - **다수의 Toffoli gate로 구현**

- **Quantum – Classical**
  - **Naïve 버전**
    - 신드롬 값을 위한 Qubits 할당
    - Parity check matrix의 값에 따라 **CNOT 게이트 사용**

  - **PLU Decomposition 버전**
    - **이러한 선형 레이어는 선형 Matrix 구성 가능 → LUP 분해를 통해 In-Place 구현이 가능**
    - **추가 Qubit 없이 신드롬 값을 생성할 수 있음**

  - **Optimized 버전**
    - **선형 Matrix에 대한 최적화 (ex: 적은 XOR 연산, 적은 Depth)**
    - PLU는 최적화를 **고려하지** 않음

Code 기반 PQC 양자 구현 : Encoding (Q-Q)

```python
def Encoding_1(eng, h, e, col, row):

    syndrome = eng.allocate_qureg(row)

    for i in range(row):
        # Quantum - Quantum
        h_e_mul(eng, h[(col*i):(col*i)+col], e, syndrome[row-1-i], col)

    return syndrome
```

Toffoli 게이트가 사용됨

$$
\begin{array}{ccccc}
\textbf{Quantum} & & \textbf{Quantum} & & \textbf{Quantum} \\
\begin{pmatrix}
1&1&0&1&1&0&1&0\\
1&1&0&1&0&1&0&1\\
1&1&1&1&1&1&0&0\\
0&0&0&1&1&0&0&0\\
1&0&1&1&0&0&0&0\\
1&1&1&0&1&0&0&0\\
1&0&0&1&1&1&0&0\\
0&1&1&1&1&1&0&0
\end{pmatrix}
& \times &
\begin{bmatrix} 0\\0\\0\\0\\0\\0\\1\\1 \end{bmatrix}
& = &
\begin{bmatrix} 1\\1\\0\\0\\0\\0\\0\\0 \end{bmatrix}
\end{array}
$$

Code 기반 PQC 양자 구현: Encoding (C-Q, Naïve)

```python
def Encoding_2(eng, H, e, col, row):

    syndrome = eng.allocate_qureg(row)
    for i in range(row):
        for j in range(col):
            # Classic - Quantum
            if(H[col*i+j] == 1):
                CNOT | (e[col-1-j], syndrome[row-1-i])
```

추가 Qubit과 CNOT 게이트 사용

Classical · **Quantum** · **Quantum**

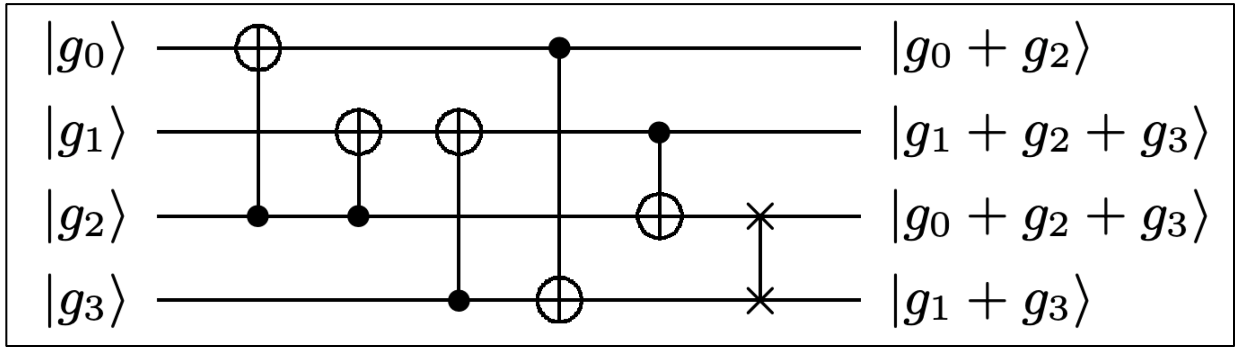
$$\begin{pmatrix} 1&1&0&1&1&0&1&0 \\ 1&1&0&1&0&1&0&1 \\ 1&1&1&1&1&1&0&0 \\ 0&0&0&1&1&0&0&0 \\ 1&0&1&1&0&0&0&0 \\ 1&1&1&0&1&0&0&0 \\ 1&0&0&1&1&1&0&0 \\ 0&1&1&1&1&1&0&0 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Code 기반 PQC 양자 구현: Encoding (C-Q, LUP)

- **Linear layer는 Matrix로 표현할 수 있음**

$$\boxed{\begin{matrix} |g_0 + g_2\rangle \\ |g_1 + g_2 + g_3\rangle \\ |g_0 + g_2 + g_3\rangle \\ |g_1 + g_3\rangle \end{matrix}} = \quad \Gamma = \begin{pmatrix} 1&0&1&0 \\ 0&1&1&1 \\ 1&0&1&1 \\ 0&1&0&1 \end{pmatrix} = P^{-1}LU = \underbrace{\begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{pmatrix}}_{\text{Permutation}} \underbrace{\begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&1&1&0 \\ 1&0&0&1 \end{pmatrix}}_{\text{Lower}} \underbrace{\begin{pmatrix} 1&0&1&0 \\ 0&1&1&1 \\ 0&0&1&0 \\ 0&0&0&1 \end{pmatrix}}_{\text{Upper}}$$

- **PLU 분해를 기반으로 한 In-place + 최적화 가능 → CNOT 게이트만이 사용됨**



Upper → Lower → Permutation 순서로 구현

Code 기반 PQC 양자 구현: Encoding (C-Q, LUP)

- **Sage에서 PLU 분해 가능**

SageMathCell

Type some Sage code below and press Evaluate.

```
M = Matrix (GF(2), [[1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1],
[0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1],
[0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1],
[0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0],
[0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0],
[0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0]])

print(M)
print(M.LU())
```

Evaluate

→

```
[1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1]
[0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1]
[0 0 1 0 0 0 0 0 1 1 1 1 1 0 1 1]
[0 0 0 1 0 0 0 0 0 1 0 1 1 1 0 1]
[0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 0]
[0 0 0 0 0 1 0 0 1 1 0 0 1 1 1 0]
[0 0 0 0 0 0 1 0 1 0 1 1 0 1 0 0]
[0 0 0 0 0 0 0 1 0 1 1 0 0 1 1 0]
([1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1], [1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1], [1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1]
[0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1]
[0 0 1 0 0 0 0 0 1 1 1 1 1 0 1 1]
[0 0 0 1 0 0 0 0 0 1 0 1 1 1 0 1]
[0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 0]
[0 0 0 0 0 1 0 0 1 1 0 0 1 1 1 0]
[0 0 0 0 0 0 1 0 1 0 1 1 0 1 0 0]
[0 0 0 0 0 0 0 1 0 1 1 0 0 1 1 0])
```

Target

Permutation

Lower

Upper

Code 기반 PQC 양자 구현: Encoding (C-Q, LUP)

```python
###### C-Q, PLU Decomposition ######
U = [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1,
     0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
     0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1,
     0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1,
     0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,
     0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0,
     0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0]

Apply_U(eng, vector_e, row, col, U)
```

```python
def Apply_U(eng, vector_e, row, col, U):
    for i in range(row):
        for j in range(col - 1 - i):
            if (U[(i * col) + 1 + i + j] == 1):
                CNOT | (vector_e[col - 2 - i - j], vector_e[col - 1 - i])
```

PLU Decomposition

Inner Mixing + Permutation
Only CNOT gates

Classical **Quantum** **Quantum**

$$\begin{pmatrix} 1&1&0&1&1&0&1&0 \\ 1&1&0&1&0&1&0&1 \\ 1&1&1&1&1&1&0&0 \\ 0&0&0&1&1&0&0&0 \\ 1&0&1&1&0&0&0&0 \\ 1&1&1&0&1&0&0&0 \\ 1&0&0&1&1&1&0&0 \\ 0&1&1&1&1&1&0&0 \end{pmatrix} \times \boxed{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Code 기반 PQC 양자 구현: Encoding (C-Q, Optimize)

- **Linear Layer 최적화 논문 (FSE'20)**

IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 0, No. 0, pp. 0–0.
DOI:10.13154/tosc.v0.i0.0-0

Optimizing Implementations of Linear Layers

Zejun Xiang[1], Xiangyong Zeng[1], Da Lin[1], Zhenzhen Bao[2] and Shasha Zhang[1]

[1] Faculty of Mathematics and Statistics, Hubei Key Laboratory of Applied Mathematics, Hubei University, Wuhan, China.
{xiangzejun,xzeng}@hubu.edu.cn, linda@stu.hubu.edu.cn, amushasha@163.com
[2] Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore.
zzbao@ntu.edu.sg

Abstract. In this paper, we propose a new heuristic algorithm to search efficient implementations (in terms of XOR count) of linear layers used in symmetric-key cryptography. It is observed that the implementation cost of an invertible matrix is related to its matrix decomposition if sequential-XOR (s-XOR) metric is considered, thus reducing the implementation cost is equivalent to constructing an optimized matrix decomposition. The basic idea of this work is to find various matrix decompositions for a given matrix and optimize those decompositions to pick the best implementation. In order to optimize matrix decompositions, we present several matrix multiplication rules over $\mathbb{F}_2$, which are proved to be very powerful in reducing the implementation cost. We illustrate this heuristic by searching implementations of several matrices proposed recently and matrices already used in block ciphers and Hash functions, and the results show that our heuristic performs equally good or outperforms Paar's and Boyar-Peralta's heuristics in most cases.

Keywords: Linear Layer · Implementation · XOR Count · AES

Table 3: An implementation of AES MixColumns with 92 XOR operations.

No.	Operation	No.	Operation	No.	Operation
0	$x_{23}=x_{23}+x_{31}$	31	$x_{20}=x_{20}+x_{27}$	62	$x_{14}=x_{14}+x_{21}$
1	$x_{31}=x_{31}+x_{15}$	32	$x_{20}=x_{20}+x_{19}$	63	$x_{6}=x_{6}+x_{5}$
2	$x_{12}=x_{12}+x_{4}$	33	$x_{27}=x_{27}+x_{31}$	64	$x_{22}=x_{22}+x_{21}$
3	$x_{13}=x_{13}+x_{21}$	34	$x_{12}=x_{12}+x_{15}$	65	$x_{5}=x_{5}+x_{29}[y_{29}]$
4	$x_{17}=x_{17}+x_{9}$	35	$x_{27}=x_{27}+x_{3}$	66	$x_{21}=x_{21}+x_{28}$
5	$x_{11}=x_{11}+x_{27}$	36	$x_{3}=x_{3}+x_{11}$	67	$x_{29}=x_{29}+x_{21}[y_{13}]$
6	$x_{4}=x_{4}+x_{28}$	37	$x_{11}=x_{11}+x_{2}$	68	$x_{21}=x_{21}+x_{13}[y_{21}]$
7	$x_{21}=x_{21}+x_{5}$	38	$x_{19}=x_{19}+x_{18}$	69	$x_{12}=x_{12}+x_{27}[y_{28}]$
8	$x_{0}=x_{0}+x_{24}$	39	$x_{11}=x_{11}+x_{10}$	70	$x_{27}=x_{27}+x_{26}$
9	$x_{15}=x_{15}+x_{7}$	40	$x_{10}=x_{10}+x_{18}$	71	$x_{28}=x_{28}+x_{20}[y_{20}]$
10	$x_{9}=x_{9}+x_{1}$	41	$x_{18}=x_{18}+x_{2}$	72	$x_{20}=x_{20}+x_{4}[y_{12}]$
11	$x_{14}=x_{14}+x_{6}$	42	$x_{10}=x_{10}+x_{9}[y_{2}]$	73	$x_{26}=x_{26}+x_{1}$
12	$x_{24}=x_{24}+x_{16}$	43	$x_{2}=x_{2}+x_{9}$	74	$x_{14}=x_{14}+x_{30}[y_{6}]$
13	$x_{6}=x_{6}+x_{22}$	44	$x_{18}=x_{18}+x_{17}[y_{10}]$	75	$x_{4}=x_{4}+x_{12}[y_{4}]$
14	$x_{16}=x_{16}+x_{31}$	45	$x_{17}=x_{17}+x_{25}$	76	$x_{3}=x_{3}+x_{19}[y_{19}]$
15	$x_{24}=x_{24}+x_{8}$	46	$x_{1}=x_{1}+x_{17}$	77	$x_{19}=x_{19}+x_{27}[y_{11}]$
16	$x_{18}=x_{18}+x_{26}$	47	$x_{25}=x_{25}+x_{24}$	78	$x_{1}=x_{1}+x_{25}$
17	$x_{22}=x_{22}+x_{30}$	48	$x_{9}=x_{9}+x_{8}$	79	$x_{0}=x_{0}+x_{24}[y_{24}]$
18	$x_{26}=x_{26}+x_{10}$	49	$x_{24}=x_{24}+x_{15}[y_{0}]$	80	$x_{1}=x_{1}+x_{0}[y_{25}]$
19	$x_{8}=x_{8}+x_{23}$	50	$x_{11}=x_{11}+x_{15}[y_{3}]$	81	$x_{2}=x_{2}+x_{26}[y_{18}]$
20	$x_{30}=x_{30}+x_{13}$	51	$x_{8}=x_{8}+x_{0}[y_{16}]$	82	$x_{25}=x_{25}+x_{9}[y_{17}]$
21	$x_{13}=x_{13}+x_{29}$	52	$x_{15}=x_{15}+x_{23}$	83	$x_{15}=x_{15}+x_{7}[y_{7}]$
22	$x_{5}=x_{5}+x_{13}$	53	$x_{17}=x_{17}+x_{16}$	84	$x_{7}=x_{7}+x_{23}[y_{15}]$
23	$x_{29}=x_{29}+x_{4}$	54	$x_{16}=x_{16}+x_{0}$	85	$x_{6}=x_{6}+x_{14}[y_{22}]$
24	$x_{4}=x_{4}+x_{11}$	55	$x_{0}=x_{0}+x_{31}$	86	$x_{9}=x_{9}+x_{17}[y_{9}]$
25	$x_{11}=x_{11}+x_{19}$	56	$x_{16}=x_{16}+x_{23}[y_{8}]$	87	$x_{23}=x_{23}+x_{31}[y_{31}]$
26	$x_{13}=x_{13}+x_{12}[y_{5}]$	57	$x_{23}=x_{23}+x_{6}$	88	$x_{26}=x_{26}+x_{18}[y_{26}]$
27	$x_{19}=x_{19}+x_{23}$	58	$x_{31}=x_{31}+x_{7}$	89	$x_{22}=x_{22}+x_{6}[y_{30}]$
28	$x_{4}=x_{4}+x_{31}$	59	$x_{31}=x_{31}+x_{22}[y_{23}]$	90	$x_{17}=x_{17}+x_{0}[y_{1}]$
29	$x_{12}=x_{12}+x_{20}$	60	$x_{30}=x_{30}+x_{6}[y_{14}]$	91	$x_{27}=x_{27}+x_{11}[y_{27}]$
30	$x_{28}=x_{28}+x_{12}$	61	$x_{7}=x_{7}+x_{14}$		

Code 기반 PQC 양자 구현: Encoding (C-Q, Optimize)

- **Out-of-place 선택지도 있음**

IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2019, No. 1, pp. 84–117. DOI:10.13154/tosc.v2019.i1.84-117

Constructing Low-latency Involutory MDS Matrices with Lightweight Circuits

Shun Li[1,2,4], Siwei Sun[1,2,4]✉, Chaoyun Li[3], Zihao Wei[1,2,4] and Lei Hu[1,2,4]

[1] State Key Laboratory of Information Security (SKLOIS), Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{lishun,sunsiwei,hulei,weizihao}@iie.ac.cn

[2] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing 100093, China

[3] imec - Computer Security and Industrial Cryptography (COSIC) research group, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium
chaoyun.li@esat.kuleuven.be

[4] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

...plementation of G with 80 XOR gates and depth 4, where $(x_0, \cdots, x_{31})$ are ...$y_0, \cdots, y_{31})$ are output signals, and t_i's are intermediate signals.

	...on	Depth	No.	Operation	Depth	No.	Operation	Depth
	... x_9	1	28	$t_{28} = x_{31} + t_{16}$	2	55	$t_{55} = x_4 + t_{38}$	3
	... x_8	1	29	$t_{29} = x_7 + t_{28}$ $[y_7]$	3	56	$t_{56} = t_{40} + t_{55}$ $[y_4]$	4
	... t_1	2	30	$t_{30} = x_7 + x_{19}$	1	57	$t_{57} = x_5 + x_{29}$	1
	... $+ t_2$	2	31	$t_{31} = x_7 + x_{26}$	1	58	$t_{58} = t_6 + t_{57}$ $[y_5]$	2
	... x_{30}	1	32	$t_{32} = x_8 + t_{30}$	2	59	$t_{59} = x_9 + t_{34}$	3
	... x_{22}	1	33	$t_{33} = x_{29} + t_{32}$ $[y_{29}]$	3	60	$t_{60} = t_{36} + t_{59}$ $[y_9]$	4
	... x_{27}	1	34	$t_{34} = x_{14} + t_{31}$	2	61	$t_{61} = x_{10} + t_7$	2
	... x_{18}	1	35	$t_{35} = x_{20} + t_{34}$ $[y_{20}]$	3	62	$t_{62} = t_8 + t_{61}$ $[y_{10}]$	3
	... $+ t_7$	2	36	$t_{36} = x_{24} + t_{22}$	2	63	$t_{63} = x_{11} + t_{32}$	3
	... t_9 $[y_{21}]$	3	37	$t_{37} = x_0 + t_{36}$ $[y_0]$	3	64	$t_{64} = t_{38} + t_{63}$ $[y_{11}]$	4
	... $+ t_1$	2	38	$t_{38} = x_{28} + t_2$	2	65	$t_{65} = x_{12} + t_{11}$	3
	...11 $[y_{30}]$	3	39	$t_{39} = x_{22} + t_{38}$ $[y_{22}]$	3	66	$t_{66} = t_{13} + t_{65}$ $[y_{12}]$	4
	... $+ t_3$	3	40	$t_{40} = x_{21} + t_4$	3	67	$t_{67} = x_{13} + x_{21}$	1
	...13 $[y_{23}]$	4	41	$t_{41} = x_{31} + t_{40}$ $[y_{31}]$	4	68	$t_{68} = t_5 + t_{67}$ $[y_{13}]$	2
	... x_{22}	1	42	$t_{42} = x_{12} + x_{23}$	1	69	$t_{69} = x_{17} + t_{17}$	3
	... $+ x_{16}$	1	43	$t_{43} = x_{24} + t_{21}$	2	70	$t_{70} = t_{19} + t_{69}$ $[y_{17}]$	4
	... $+ t_{15}$	2	44	$t_{44} = x_{15} + t_{43}$ $[y_{15}]$	3	71	$t_{71} = x_{18} + t_{43}$	3
	...17 $[y_{14}]$	3	45	$t_{45} = x_{30} + t_{42}$	2	72	$t_{72} = t_{45} + t_{71}$ $[y_{18}]$	4
	... $+ t_6$	3	46	$t_{46} = x_6 + t_{45}$ $[y_6]$	3	73	$t_{73} = x_{19} + t_{26}$	3
	...19 $[y_{24}]$	4	47	$t_{47} = t_4 + t_5$	3	74	$t_{74} = t_{28} + t_{73}$ $[y_{19}]$	4
21	$t_{21} = x_5 + x_{23}$	1	48	$t_{48} = x_{16} + t_{47}$ $[y_{16}]$	4	75	$t_{75} = x_{25} + t_{45}$	3
22	$t_{22} = x_{14} + x_{17}$	1	49	$t_{49} = x_1 + t_{24}$	3	76	$t_{76} = t_{47} + t_{75}$ $[y_{25}]$	4
23	$t_{23} = x_6 + x_{25}$	1	50	$t_{50} = t_{26} + t_{49}$ $[y_1]$	4	77	$t_{77} = x_{26} + t_{15}$	2
24	$t_{24} = x_{15} + t_8$	2	51	$t_{51} = x_2 + t_{32}$	3	78	$t_{78} = t_{16} + t_{77}$ $[y_{26}]$	3
25	$t_{25} = x_{28} + t_{24}$ $[y_{28}]$	3	52	$t_{52} = t_{34} + t_{51}$ $[y_2]$	4	79	$t_{79} = x_{27} + t_{21}$	2
26	$t_{26} = x_{16} + t_{23}$	2	53	$t_{53} = x_3 + t_9$	3	80	$t_{80} = t_{22} + t_{79}$ $[y_{27}]$	3
27	$t_{27} = x_8 + t_{26}$ $[y_8]$	3	54	$t_{54} = t_{11} + t_{53}$ $[y_3]$	4			

Code 기반 PQC 양자 구현: Encoding (C-Q, Optimize)

- **Encoding (Matrix X Vector) 자원 비교**
  - 8 x 16 Matrix 대상
  - 시나리오는 Classical – Quantum이 적합
  - **선형 연산에 대한 최적화가 중요**

Method	Qubits	CNOT	Toffoli	Full Depth
Q-Q	152	·	128	147
C-Q (Naïve)	24	45	·	14
C-Q (PLU Decomposition)	**16**	**37**	·	**13**
C-Q (Optimized)	**?**	**?**	·	**?**

Code 기반 PQC 양자 구현: Key pair (Multiplication)

- x^{12}, x^{13}의 곱셈에 대해서는 WISA22 곱셈 적용
  - 이 부분도 Modular reduction (선형 연산) 최적화 가능

- $\mathbb{F}_{2^{12}}/(x^{12}+x^3+1)$, $\mathbb{F}_{2^{13}}/(x^{13}+x^4+x^3+x^1+1)$ 곱셈 양자 자원

```
Gate counts:
    Allocate : 162
    CX : 653
    Deallocate : 162
    H : 108
    T : 162
    T^\dagger : 216

Depth : 37.
```

```
Gate counts:
    Allocate : 198
    CX : 834
    Deallocate : 198
    H : 132
    T : 198
    T^\dagger : 264

Depth : 54.
```

Code 기반 PQC 양자 구현: Key pair (Squaring)

- **Squaring 후의 Modular reduction 또한 선형 연산**
  - 비용이 매우 적은 연산이지만, 조금의 최적화 가능
  - 현재 구현은 그냥 손으로 노가다? 한 In-Place 구현

- $\mathbb{F}_{2^{12}}/(\boldsymbol{x}^{12}+\boldsymbol{x}^{3}+\mathbf{1})$ **,** $\mathbb{F}_{2^{13}}/(\boldsymbol{x}^{13}+\boldsymbol{x}^{4}+\boldsymbol{x}^{3}+\ \boldsymbol{x}^{1}+\mathbf{1})$ **곱셈 양자 자원**

```
Gate counts:
    Allocate : 12
    CX : 7
    Deallocate : 12

Depth : 2.
```

```
Gate counts:
    Allocate : 13
    CX : 23
    Deallocate : 13

Depth : 14.
```

Code 기반 PQC 양자 구현: Key pair (Inversion)

- **Fermat's Little Theorem(FLT) → Itoh-Tsujii 기반의 Inversion**

```
# Itoh-Tsuji Inversion
Copy(eng, input, out, n)
out = Squaring(eng, out)

temp_11 = []
temp_11 = Karatsuba_12_Toffoli_Depth_1(eng, out, input, r_a, r_b, rr_a, rr_b, rrr, c0)

temp_11_copy = eng.allocate_qureg(n)
Copy(eng, temp_11, temp_11_copy, n)

temp_11 = Squaring(eng, temp_11)
temp_11 = Squaring(eng, temp_11)

temp_1111 = []
temp_1111 = Karatsuba_12_Toffoli_Depth_1(eng, temp_11, temp_11_copy, r_a, r_b, rr_a, rr_b, rrr, c1)

temp_1111_copy = eng.allocate_qureg(n)
Copy(eng, temp_1111, temp_1111_copy, n)

temp_1111 = Squaring(eng, temp_1111)
temp_1111 = Squaring(eng, temp_1111)
temp_1111 = Squaring(eng, temp_1111)
temp_1111 = Squaring(eng, temp_1111)

result_temp0 = []
result_temp0 = Karatsuba_12_Toffoli_Depth_1(eng, temp_1111, temp_1111_copy, r_a, r_b, rr_a, rr_b, rrr, c2)

result_temp0 = Squaring(eng, result_temp0)
result_temp0 = Squaring(eng, result_temp0)

result_temp1 = []
result_temp1 = Karatsuba_12_Toffoli_Depth_1(eng, result_temp0, temp_11_copy, r_a, r_b, rr_a, rr_b, rrr, c3)

result_temp1 = Squaring(eng, result_temp1)

result = []
result = Karatsuba_12_Toffoli_Depth_1(eng, result_temp1, input, r_a, r_b, rr_a, rr_b, rrr, c4)

result = Squaring(eng, result)
```

Toffoli depth: 5

- **곱셈과 Squaring의 조합으로 구현됨**
- **매우 낮은 Toffoli depth, Full depth**

Code 기반 PQC 양자 구현: Key pair (Inversion)

- $\mathbb{F}_{2^{12}}/(x^{12}+x^3+1)$, $\mathbb{F}_{2^{13}}/(x^{13}+x^4+x^3+x^1+1)$ **Inversion 양자 자원**

```
Inversion result:
Result : 010000101111

Inversion Check (input * inversion) = 1? :
Result : 000000000001

Gate class counts:
    AllocateQubitGate : 402
    CXGate : 4218
    DaggeredGate : 1080
    DeallocateQubitGate : 402
    HGate : 540
    TGate : 810

Depth : 194.
```

```
Inversion result :
Result : 0110011001001

Inversion Check (input * inversion) = 1? :
Result : 0000000000001

Gate class counts:
    AllocateQubitGate : 422
    CXGate : 4460
    DaggeredGate : 1056
    DeallocateQubitGate : 422
    HGate : 528
    TGate : 792

Depth : 369.
```

Code 기반 PQC 양자 구현: Key pair (Inversion)

- **ECC 양자 공격 논문(CHES)의 Table 2(Inversion 비용)는, Toffoli 게이트가 분해되기 전의 Depth**
  - **Full depth가 아니기 때문에 실제론 훨씬 더 큼**
  - **WISA 구현을 활용하면 Qubit 오버헤드가 감소(Qubit 재활용)하면서, Full Depth가 매우 낮음**

```
Inversion result:
Result : 010000101111

Inversion Check (input * inversion) = 1? :
Result : 000000000001

Gate class counts:
    AllocateQubitGate : 402
    CXGate : 4218
    DaggeredGate : 1080
    DeallocateQubitGate : 402
    HGate : 540
    TGate : 810

Depth : 194.
```

$n = 12$

Table 2: Comparison of various instances of division Algorithms 1 and 2. Field polynomials from Table 1. Depths and gate count are upper bounds since a generic algorithm is used rather than optimizing for specific fields.

n	GCD				FLT			
	TOF	CNOT	qubits	depth	TOF	CNOT	qubits	depth
8	3,641	1,516	67	4113	243	2,212	56	1314
16	10,403	5,072	124	12,145	1,053	10,814	144	5968
127	277,195	227,902	903	378,843	50,255	502,870	1,778	203,500
163	442,161	375,738	1,156	612,331	83,353	906,170	1,956	451,408
233	827,977	743,136	1,646	1,172,733	132,783	1,486,464	3,029	640,266
283	1,202,987	1,088,400	1,997	1,708,863	236,279	2,708,404	3,962	1,434,686
571	4,461,673	4,266,438	4,014	6,494,306	814,617	10,941,536	9,136	6,151,999

Conclusion

- **Encoding (Matrix X Vector)는 선형 연산 최적화 기술을 적용하여 효율적 구현 가능**
  - **PLU, Linear Optimization paper..**

- **CM의 Key pair, Decoding에서 사용되는 binary Field 산술 구현**
  - **Multiplication, Squaring, Inversion**

- **WISA 곱셈기를 활용하면 Qubit 수가 증가하지만 Depth가 매우 낮음**
  - **Time-efficient**
  - **Space-inefficient**

- **Decoding (Berkem-Massey)까지 구현?**

Classic McEliece : Decoding

LFSR Synthesis Algorithm (Berlekamp Iterative Algorithm):

1) $1 \to C(D)$ $\quad 1 \to B(D)$ $\quad 1 \to x$
   $0 \to L$ $\quad 1 \to b$ $\quad 0 \to N$
2) If $N = n$, stop. Otherwise compute

$$\boxed{d = s_N + \sum_{i=1}^{L} c_i s_{N-i}.}$$

3) If $d = 0$, then $x + 1 \to x$, and go to 6).
4) If $d \neq 0$ and $2L > N$, then
   $C(D) - d\, b^{-1} D^x B(D) \to C(D)$
   $x + 1 \to x$
   and go to 6).
5) If $d \neq 0$ and $2L \leq N$, then
   $\boxed{C(D) \to T(D) \text{ [temporary storage of } C(D)]}$
   $C(D) - \boxed{d\, b^{-1}}\boxed{D^x}\boxed{B(D)} \to C(D)$
   $N + 1 - L \to L$
   $\boxed{T(D) \to B(D)}$
   $d \to b$
   $\boxed{1 \to x.}$
6) $N + 1 \to N$ and return to 2).

```
for (N = 0; N < 2 * SYS_T; N++)
{
    d = 0;

    for (i = 0; i <= min(N, SYS_T); i++)
        d ^= gf_mul(C[i], s[ N-i]);

    mne = d; mne -= 1;    mne >>= 15; mne -= 1;
    mle = N; mle -= 2*L; mle >>= 15; mle -= 1;
    mle &= mne; //mle = 1111 1111 1111 1111 아니면 0

    for (i = 0; i <= SYS_T; i++)
        T[i] = C[i];

    f = gf_frac(b, d);

    for (i = 0; i <= SYS_T; i++)
        C[i] ^= gf_mul(f, B[i]) & mne;

    L = (L & ~mle) | ((N+1-L) & mle);

    for (i = 0; i <= SYS_T; i++)
        B[i] = (B[i] & ~mle) | (T[i] & mle);
                  (4)                (5)

    b = (b & ~mle) | (d & mle);

    for (i = SYS_T; i >= 1; i--) B[i] = B[i-1];
    B[0] = 0;
}

for (i = 0; i <= SYS_T; i++)
    out[i] = C[ SYS_T-i ];          → End
}
```

*

```
B[1] = C[0] = 1;
```

*

```
B[1] = C[0] = 1;
```

감사합니다