NIST Round 2 Code-based PQC

Crypto Craft Lab

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.

Contents

CryptoCraft LAB

코드 기반 암호

코드 기반 암호

코딩 이론

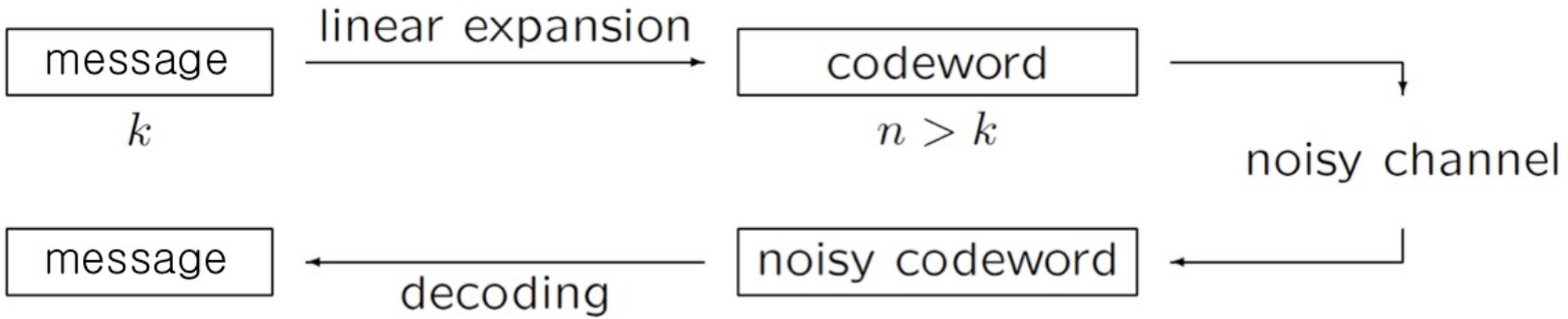
- 1948년 Claude Shannon 제안
- 불완전한 채널(noisy channel)에서 발생하는 잡음 제거
- 코드 기반 암호의 핵심 원리

해밍 코드

선형 부호

코드 기반 암호

코딩 이론 구조



코드 기반 암호 - 해밍 코드

해밍 코드

- 오류 탐지 기능을 추가한 송신 코드

패리티 비트

- 정보 전달 과정에서 오류 발생 여부 확인을 위해 추가되는 비트

패리티 비트 공식

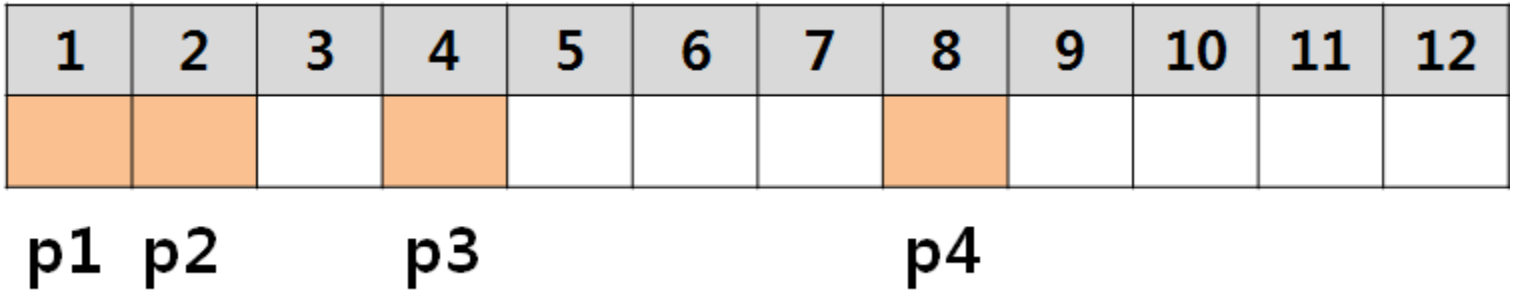
- d: 데이터 비트 수
- p: 패리티 비트

$$2^p+1 \geq d + p$$

코드 기반 암호 - 해밍 코드

패리티 비트

- 1부터 2^n 위치에 배치



코드 기반 암호 - 해밍 코드

ex) (7,4) 해밍 코드

- 4개의 비트 메시지 암호화
- 3개의 패리티 비트 사용

Message bit: x_0 x_1 x_2 x_3

Parity bit: P_1 P_2 P_4

코드 기반 암호 - 해밍 코드

(7,4) 해밍 코드

Message bit: x_0 x_1 x_2 x_3

Parity bit: P_1 P_2 P_4

Hamming code structure

1	2	3	4	5	6	7
p_1	p_2	x_3	p_4	x_2	x_1	x_0

코드 기반 암호 - 해밍 코드

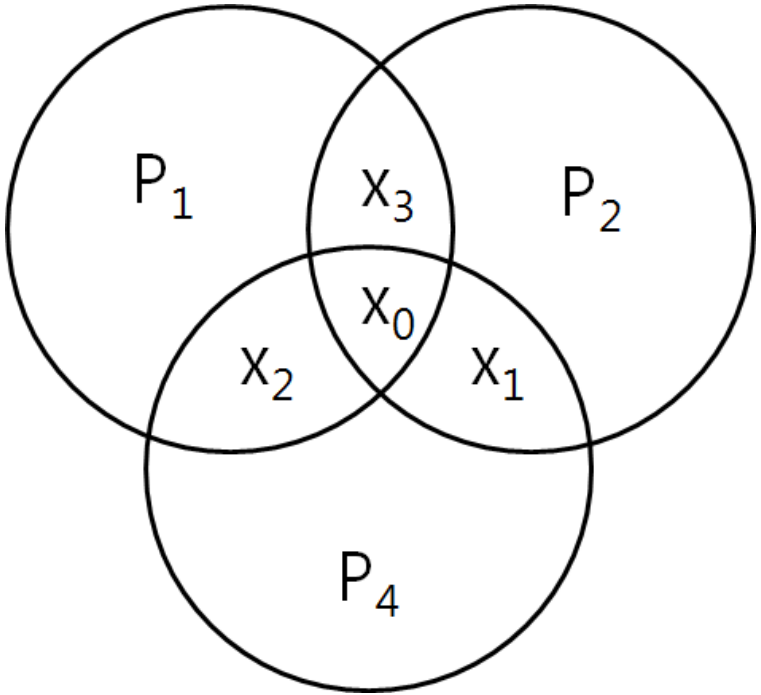
(7,4) 해밍 코드

Message bit: x_0 x_1 x_2 x_3

Parity bit: P_1 P_2 P_4

Hamming code structure

1	2	3	4	5	6	7
p_1	p_2	x_3	p_4	x_2	x_1	x_0



코드 기반 암호 - 해밍 코드

Hamming code structure

1	2	3	4	5	6	7
p_1	p_2	x_3	p_4	x_2	x_1	x_0

p1 1, 3, 5, 7 담당

P1 = X3 ⊕ x2 ⊕ x0

1	001
3	011
5	101
7	111

p2 2, 3, 6, 7 담당

P2 = X3 ⊕ x1 ⊕ x0

2	010
3	011
6	110
7	111

p4 4, 5, 6, 7 담당

P4 = X2 ⊕ x1 ⊕ x0

4	100
5	101
6	110
7	111

코드 기반 암호 - 해밍 코드

에러 체크

C1 = P1 ⊕ X3 ⊕ x2 ⊕ x0

C2 = P2 ⊕ X3 ⊕ x1 ⊕ x0

C4 = P4 ⊕ X2 ⊕ x1 ⊕ x0

→ 연산 결과가 0일시 정상

코드 기반 암호 - 해밍 코드

에러 체크

Hamming code example

1	2	3	4	5	6	7
p_1	p_2	x_3	p_4	x_2	x_1	x_0
1	1	0	0	1	1	0

1	0	0	0	1	1	0

코드 기반 암호 - 해밍 코드

에러 체크

Hamming code example

1	2	3	4	5	6	7
p_1	p_2	x_3	p_4	x_2	x_1	x_0
1	0	0	0	1	1	0

C1 = P1 ⊕ X3 ⊕ x2 ⊕ x0 = 0

C2 = P2 ⊕ X3 ⊕ x1 ⊕ x0 = 1

C4 = P4 ⊕ X2 ⊕ x1 ⊕ x0 = 0

010 = 2 → 2번째 자리에 에러

코드 기반 암호 - 선형 부호

선형 부호(Linear code)

- [n k] 코드를 갖고 진행
- n: 코드의 길이
- k: 차원

코드 기반 암호 - 선형 부호

$$G(\text{Generating Matrix}) = \overbrace{\left[\, I_k \mid P \,\right]}^{n}\Big\}_{k}$$

$$H(\text{Pairty Check Matrix}) = \underbrace{\left[\, -P^T \mid I_{n-k} \,\right]}_{n}\Big\}_{n-k}$$

I_k : 단위 행렬

P : 에러 수정 행렬

코드 기반 암호 - 선형 부호

G와 H 사이를 자유롭게 오갈 수(변경) 있음

ex)

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} I_k \mid P \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}_{\substack{k=2 \\ n=4}}$$

$$H = \begin{bmatrix} -P^T \mid I_{n-k} \end{bmatrix} = \begin{bmatrix} 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

T = 전위

코드 기반 암호 - 선형 부호

생성된 G와 H^T 곱은 0

$$G \qquad H^T$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

코드 기반 암호 - McEliece

- 1978, Robert J. McEliece에 의해 제안
- 40년 동안 다양한 방법으로 공격 시도
- 양자 컴퓨터 상에서도 안전하다고 여겨지는 암호
- 현존하는 다른 공개키 암호에 비해 키의 크기가 너무 큼
- 양자 컴퓨터 시대에 대비하는 움직임과 함께 다시 주목을 받음
- Goppa 코드 활용

코드 기반 암호 - McEliece

→ 키생성 : 주어진 파라미터를 이용한다.

G : 최소거리 $(d \geq 2t+1)$ 인 이진부호 g에대한 $k \times n$ 생성행렬 *Goppa* 부호

S : 임의의 $k \times k$ 이진 정칙 행렬

P : 임의의 $n \times n$ 순열행렬

G' : SGP

공개키 : (G', t)

비밀키 : $\left(S, D_g, P\right)$ 이때 D_g는 g에 대한 효율적인 복호 알고리즘이다.

→ 암호화: 암호문 c는 이렇게 생성된다.

$$c = mG' \oplus e$$

→ 복호화 : 먼저 $cP^{-1} = (mS)G \oplus eP^{-1}$를 계산한 후, 복호알고리즘 D_g을 적용하여 다음과 같이 계산한다.

$$mSG = D_g\left(cP^{-1}\right)$$

복호알고리즘 후 $m = mS \cdot S^{-1}$을 계산하면 원본 메세지인 m 을 획득할 수 있다.

코드 기반 암호 - 선형 이진 부호

Parameter: [*n, k, d*]

minimum weight가 *d*인 선형 이진 후보 [***n, k***]는 $t = \frac{d-1}{2}$ 의 오류를 수정할 수 있음

Definition:

[***n, k***] 는 F_2에서의 **k** 선형 독립한 벡터들로 표현 가능

해당 ***k*** 벡터들을 ***k*** x ***n***의 행렬로 사용 → Generator Matrix(***G***)

코드 내의 모든 벡터들을 generator matrix 행들의 선형조합으로 생성 가능

코드 기반 암호 - 이진 선형 부호

- 길이 ***k*** 의 메시지 ***m*** 을 암호화 하기 위해 Goppa code ***G*** 를 사용하여 길이 ***n*** 으로 선형확장 → 인코딩

(Message) x [Goppa code] = (codeword)

(***1*** x ***k***) (***k*** x ***n***) (***1*** x ***n***)

선형확장

(Linear expansion)

- 디코딩 과정에서는 ***G*** 에 해당하는 Parrity matrix ***H*** 가 사용됨 → Syndrome Decoding

codeword x [Parrity Check matrix.] = s, 오류가 없다면 s = 0

(***1*** x ***n***) (***n*** − ***k*** x ***n***)

코드 기반 암호 - 이진 선형 부호

- 해당 codeword ***c*** 에 오류가 추가 되어도 수정 가능
  - Goppa code가 오류수정 역할을 수행 → 공개키로 사용된다.
  - 송신자들은 자신의 메시지와 Goppa code를 사용하여 codeword를 생성, 그 뒤에 오류 ***e*** 를 임의로 추가하여 원본 메세지를 암호화 → ***mG*** + ***e*** = ***codeword*** (암호문)

- 해당 Goppa code (***G***)를 그대로 공개키로 사용하면 누구나 오류를 수정할 수 있음
  → ***G*** 를 숨기는 과정이 존재

- 수신자는 ***G*** 를 활용하여 수신된 암호문의 오류를 수정(Syndrome decoding)하여 원본 메세지를 획득

코드 기반 암호 - Key generation

개인키

- 비밀키로 사용되는 generator matrix (***G***)는 랜덤한 Goppa 코드 공간에서 뽑아냄.

  Ex) [7, 4, 3]

- *k* x *k* 의 가역행렬 ***S***와 *n* x *n* 의 순열행렬 ***P***를 랜덤하게 선택

$$G = \begin{matrix} 1\,0\,0\,0\,1\,1\,0 \\ 0\,1\,0\,0\,1\,0\,1 \\ 0\,0\,1\,0\,0\,1\,1 \\ 0\,0\,0\,1\,1\,1\,1 \end{matrix} \qquad S = \begin{matrix} 1\,1\,0\,1 \\ 1\,0\,0\,1 \\ 0\,1\,1\,1 \\ 1\,1\,0\,0 \end{matrix} \qquad P = \begin{matrix} 0\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,1 \\ 1\,0\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,1\,0\,0 \end{matrix}$$

공개키

- 개인키 ***G, S, P***를 사용하여 공개키 생성 → G를 비밀스럽게 숨기기 위해 Scramble 함

$G' = S \cdot G \cdot P$ → scramble 된 Goppa Matirx ***G'***를 공개키로 사용 (***S***는 가역, ***P***는 순열행렬)

코드 기반 암호 - Encoding & Encryption

- **k** - bit 의 메세지 ***m*** 을 암호화

$$c = G' \cdot m \oplus e$$

$$G' = S\,G\,P = \begin{matrix} 1\,1\,1\,1\,0\,0\,0 \\ 1\,1\,0\,0\,1\,0\,0 \\ 1\,0\,0\,1\,1\,0\,1 \\ 0\,1\,0\,1\,1\,1\,0 \end{matrix}$$

$$m = 1\ 1\ 0\ 1$$

$$e = 0\ 0\ 0\ 0\ 1\ 0\ 0$$

$$c = mG' \oplus e = 0\ 1\ 1\ 0\ \boxed{0}\ 1\ 0 \oplus 0\ 0\ 0\ 0\ 1\ 0\ 0 \qquad c = 0\ 1\ 1\ 0\ \boxed{1}\ 1\ 0$$

- 송신자는 암호문 전송

  (*e* 는 weight *t* 인 길이 n-bit 오류벡터)

코드 기반 암호 - Decoding & Decryption

- P의 역행렬을 암호문 c 우측에 곱해줌

$$cP^{-1} = mSG \oplus eP^{-1} = 1\ 0\ 0\ 0\ 1\ 1\ 1$$

- ***P*** 는 순열행렬이므로 오류벡터 e의 **weight 를 변경시키지 않음**
- 따라서 ***G*** 에 해당하는 Parrity matrix ***H*** 를 사용한 **Syndrome Decoding** 과정을 수행

- Syndrome : codeword 에 해당하는 패리티 검사 행렬을 계산한 값
- 암호문의 syndrome 값을 구하고 오류가 없으면 syndrome 값이 0임을 이용
- $cH^T = 0$ → 오류 없음

코드 기반 암호 - Decoding & Decryption

$cP^{-1} = (mS)G \oplus eP^{-1} = 1\ 0\ 0\ 0\ 1\ 1\ 1$ 의 syndrome 값을 계산하면

$$Hc'^{t} = H(msG + eP^{-1})^{T} = H(msG)^{T} + H(eP^{-1})^{T} = H(eP^{-1})^{T}$$

올바른 codeword 의 syndrome 값은 0 이므로, 오류벡터에 대한 syndrome 값만 남게 됨

G 는 scramble 되어 공개키에 사용되기 때문에 원본 ***G*** 에 해당하는 Parrity check matrix 는 수신자만 보유

$1\ 0\ 0\ 0\ 1\ 1\ 1 \cdot H^{T} = \boxed{0\ 0\ 1}$

$$H = \begin{matrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{matrix}$$

7번째 비트에 오류 발생

코드 기반 암호 - Decoding & Decryption

오류가 수정된 벡터는 1 0 0 0 1 1 0 → mSG

$\boldsymbol{G}$ 의 특성으로 mS 는 1 0 0 0 이 된다.

$$G = \begin{matrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{matrix}$$

마지막으로 S^{-1} 을 우측에 곱해줌으로써 원본메세지 m을 획득

$$1\ 0\ 0\ 0 \quad \text{x} \quad S^{-1} = \begin{matrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{matrix} \quad = \quad 1\ 1\ 0\ 1$$

m = 1 1 0 1

코드 기반 암호 - McEliece

- McEliece 초기 → n = 524, k = 1024 (취약점 발견)
- 현재 사이즈 → n = 4096, k = 3556

→ 효율성은 감소하였으나 공격에 방어 가능

코드 기반 암호 - MDPC

NIST Round 2 Code-based PQC

NIST Round 2

NIST Round 2 - Classic McEliece

NIST Round 2 - Classic McEliece

	Intel NUC			Raspberry Pii		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
mceliece348864	164.82	0.0696	23.88	1780.94	0.84	247.94
mceliece460896	339.86	0.15	56.6	3864.43	2.52	630.32
mceliece6688128	939.42	0.20	108.20	8987.07	2.74	9893.79
mceliece6960119	638.54	0.47	103.28	7003.93	6.64	1138.87
mceliece8192128	534.11	0.23	140.84	5738.65	3.05	1709.16

NIST Round 2 - BIKE

NIST Round 2 - BIKE

	Intel NUC			Raspberry Pi		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
BIKE-1 CPA(LEVEL1)	0.26	0.22	1.49	2.68	2.69	11.8
BIKE-1 CPA(LEVEL3)	0.66	0.57	3.99	7.10	6.99	2.94
BIKE-1 CPA(LEVEL5)	1.00	0.89	8.97	10.98	10.89	73.8
BIKE-2 CPA(LEVEL1)	1.74	0.129	1.41	30.79	1.62	10.83
BIKE-2 CPA(LEVEL3)	6.69	0.31	3.75	80.86	3.72	23.93
BIKE-2 CPA(LEVEL5)	11.41	0.48	8.56	131.81	6.26	66.51
BIKE-3 CPA(LEVEL1)	0.23	0.27	1.83	1.51	2.95	12.24
BIKE-3 CPA(LEVEL3)	0.41	0.64	4.17	4.07	8.04	31.37
BIKE-3 CPA(LEVEL5)	0.86	1.30	99.03	8.51	16.74	79.88
BIKE-1 CCA(LEVEL1)	0.37	0.28	1.85	3.40	3.46	19.14
BIKE-1 CCA(LEVEL3)	0.89	0.73	4.40	9.08	9.70	49.17
BIKE-1 CCA(LEVEL5)	1.74	1.60	9.45	20.53	20.97	113.41
BIKE-2 CCA(LEVEL1)	2.28	0.16	1.61	37.86	1.86	16.11

NIST Round 2 - BIKE

	Intel NUC			Raspberry Pi		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
BIKE-2 CCA(LEVEL3)	8.96	0.38	3.72	105.13	4.79	38.95
BIKE-2 CCA(LEVEL5)	18.62	0.83	8.02	231.88	10.79	93.15
BIKE-3 CCA(LEVEL1)	0.24	0.29	1.95	1.88	3.81	18.26
BIKE-3 CCA(LEVEL3)	0.62	8.09	4.51	5.21	10.66	50.94
BIKE-3 CCA(LEVEL5)	1.12	1.87	10.21	12.53	24.98	122.57

NIST Round 2 - NTS-KEM

NIST Round 2 - NTS-KEM

	Intel NUC			Raspberry Pi		
	KeyGen	Encap	Decap	KeyGen	Encap	Decap
nts_kem_12_64	55.96	0.12	1.71	291.66	1.28	9.8
nts_kem_13_80	159.37	0.28	3.39	958.46	2.99	17.80
nts_kem_13_136	277.88	0.37	6.80	2513.81	3.892	35.57

NIST Round 2 - HQC

NIST Round 2 - HQC

	Intel NUC			Raspberry Pi		
	KeyGen	Encap	Decap	KeyGen	Encap	Decap
hqc-128-1	3.96	0.76	1.24	50.65	9.02	15.30
hqc-192-1	0.93	1.83	2.82	11.06	24.12	37.09
hqc-192-2	1.01	1.94	2.93	12.68	25.94	39.97
hqc-256-1	1.30	2.56	3.92	16.64	35.57	51.61
hqc-256-2	1.59	3.11	4.73	17.85	34.61	52.12
hqc-256-3	1.80	3.60	5.46	19.13	40.	60.05

NIST Round 2 - RQC

NIST Round 2 - RQC

	Intel NUC			Raspberry Pi		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
rqc128	0.31	0.55	2.61	2.64	4.65	27.03
rqc192	0.46	0.91	5.75	4.19	8.50	64.91
rqc256	0.86	1.75	11.99	7.67	16.98	126.34

NIST Round 2 - ROLLO

NIST Round 2 - ROLLO

	Intel NUC			Raspberry Pi		
	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Rollo-I-128	0.84	0.17	0.53	8.19	1.21	4.27
Rollo-I-192	1.62	0.21	0.99	10.18	1.48	6.96
Rollo-I-256	1.70	0.31	1.55	18.94	2.48	12.46
Rollo-II-128	7.16	0.82	2.23	73.94	6.89	19.84
Rollo-II-192	7.70	0.92	2.62	89.42	7.97	24.07
Rollo-II-256	7.77	1.03	2.91	93.77	8.69	27.02
Rollo-III-128	0.16	0.30	0.49	1.67	2.62	3.98
Rollo-III-192	0.20	0.41	0.96	2.19	3.56	7.53
Rollo-III-256	0.37	0.73	1.65	3.46	5.94	12.66

NIST Round 2 - LEDAcrypt

NIST Round 2 - LEDAcrypt

	Intel NUC			Raspberry Pi		
	KeyGen	Encap	Decap	KeyGen	Encap	Decap
LEDAcrypt ephemeral keys 1-2	8.66	0.737	3.041	79.62	7.89	29.91
LEDAcrypt ephemeral keys 1-3	2.641	0.548	3.301	28.16	5.71	34.09
LEDAcrypt ephemeral keys 1-4	2.599	0.706	5.250	27.95	7.63	53.12
LEDAcrypt ephemeral keys 3-2	21.237	1.344	7.945	208.15	12.94	80.64
LEDAcrypt ephemeral keys 3-3	8.801	1.429	8.078.	86.61	15.07	81.22
LEDAcrypt ephemeral keys 3-4	7.907	1.863	12.023	81.02	17.13	122.37
LEDAcrypt ephemeral keys 5-2	43.430	2.675	13.400	417.12	29.05	134.20
LEDAcrypt ephemeral keys 5-3	25.234	2.778	17.188	256.30	29.97	173.21
LEDAcrypt ephemeral keys 5-4	18.651	3.350	17.561	180.91	36.33	179.01
LEDAcrypt Longterm keys	390.139	2.811	3.466	3911.10	30.22	36.07

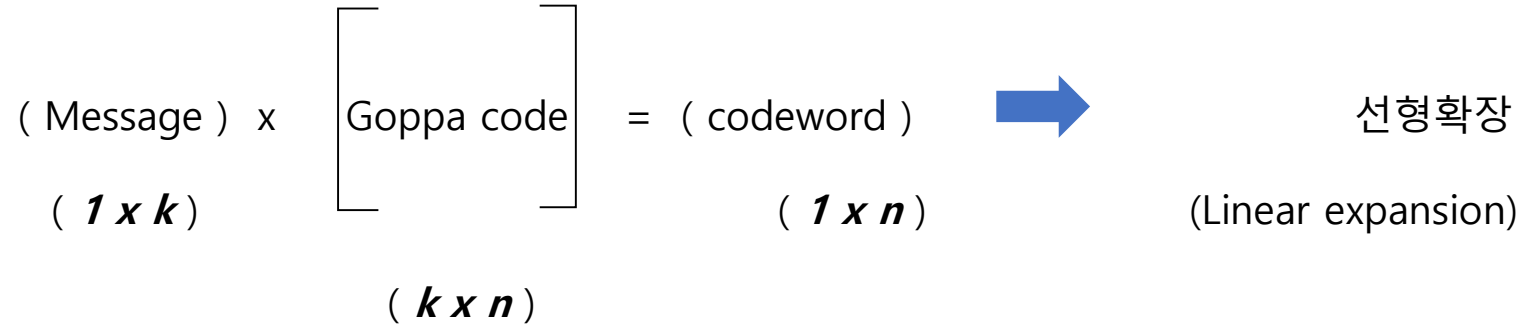
NIST Round 2 - LEDAcrypt

LEDAcrypt Longterm keys 1-2(2 ^-128)	581.784	3.905	3.746	5722.01	41.01	38.91
LEDAcrypt Longterm keys 3-2(2 ^-64)	926.896	4.602	8.505	9273.56	45.12	85.15
LEDAcrypt Longterm keys 3-2(2 ^-192)	2135.948	9.984	9.274	20059.0	101.16	95.65
LEDAcrypt Longterm keys 5-2(2 ^-64)	3665.445	7.818	14.634	38056.9	80.12	149.94
LEDAcrypt Longterm keys 5-2(2 ^-256)	4467.717	18.410	17.787	42680.2	186.55	178.07
LEDAcrypt PKC 1-2(2^-64)	302.381	2.885	3.283	3126.65	29.01	33.73
LEDAcrypt PKC 1-2(2^-128)	432.130	4.125	4.303	4123.01	43.21	44.81
LEDAcrypt PKC 3-2(2^-64)	926.896	4.602	8.505	8971.91	48.72	87.15
LEDAcrypt PKC 3-2(2^-192)	2135.948	9.984	9.274	2317.43	102.31	93.01
LEDAcrypt PKC 5-2(2^-64)	4415.511	7.954	16.393	42458.7	81.22	162.53
LEDAcrypt PKC 5-2(2^-256)	4555.365	18.819	16.856	45355.2	189.27	170.51

Information Set Decoding

McEliece 시스템 복습

- 길이 ***k*** 의 메시지 ***m*** 을 암호화 하기 위해 Goppa code ***G*** 를 사용하여 길이 ***n*** 으로 선형확장



- 여기서 중요한 것은 생성된 codeword ***c*** 에 오류가 추가 되어도 수정할 수 있다는 점
  - Goppa code가 그 오류수정 역할을 수행 → 공개키로 사용된다.
  - 송신자들은 자신의 메시지와 Goppa code를 사용하여 codeword를 생성, 그 뒤에 오류 ***e*** 를 임의로 추가하여 원본 메시지를 암호화 한다. → ***mG*** + ***e*** = ***codeword*** (암호문)

McEliece 시스템 복습

- 하지만 Goppa code ***G*** 를 그대로 공개키로 사용하면 누구나 오류를 수정할 수 있음
  - 때문에 G를 비밀스럽게 숨기는 과정이 존재

$$G' = S \cdot G \cdot P$$ → scramble 된 Goppa Matirx ***G′*** 를 공개키로 사용 (***S*** 는 가역, ***P*** 는 순열행렬)

mG′ + ***e*** = **codeword**(암호문)

- 마지막으로 수신자는 G를 활용하여 수신된 암호문의 오류를 수정(Syndrome decoding)하여 원본 메시지를 획득한다.

Infomation Set Decoding Attack

이러한 구조가 의미하는 것은 ***G′*** 로 생성한 **codeword**의 오류수정을 ***G*** 가 수행한다는 것.

이 구조 때문에 **Information Set Decoding Attack** 이 가능
→핵심은 원본코드 ***G*** 가 아닌 동일한 오류수정이 가능한 다른 ***G″*** 를 찾아내는 것

Information Set Decoding Attack

- 이러한 구조가 의미하는 것은 G' 로 생성한 **codeword**의 오류수정을 G 가 수행한다는 것.

- 이 구조 때문에 **Information Set Decoding Attack** 이 가능
  - 핵심은 원본 Goppa code G 가 아닌 동일한 오류수정이 가능한 다른 Goppa Matirx 를 찾아내는 것

Information set decoding

- Syndrome decoding → $cH^T = s$, codeword c에 G의 Parritycheck 행렬을 곱하여 syndrome 값을 획득
  - Codeword의 오류위치를 찾아주는 과정이다.
    - 오류의 개수만큼 weight 가 결정되고, 오류가 존재하지 않는다면 s 값은 0

- $cH^T = s \iff c'H'^T = s'$ where $\begin{bmatrix} H' = UHP \\ S' = SU^T \\ c' = cp \end{bmatrix}$

U = non singular matrix
P = any permutation matrix

Information Set Decoding Attack

Information set decoding

- $cH^T = s \iff c'H'^T = s'$ where $\begin{bmatrix} H' = UHP \\ s' = sU^T \\ c' = cp \end{bmatrix}$

U = any non singular matrix (invertible)
P = any permutation matrix

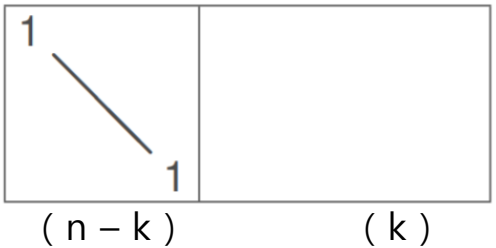
Proof

- $c'H'^T = (cP)(UHP)^T$
  $= (cP)P^T H^T U^T$ → 순열행렬과 전치행렬의 곱은 단위행렬
  $= cH^T U^T$
  $= sU^T$
  $= s'$

- 이 두가지 Syndrome decoding 계산은 동등함을 뜻한다.
- $CSD(H, s, w) \equiv CSD(UHP, SU^T, w)$ 가 동등한 것에 기반하여 **하나를 풀면 다른 한가지도 풀린다**
  → 코딩이론

Information Set Decoding Attack

- 어떠한 U와 P를 사용해서라도 $CSD(H, s, w) \equiv CSD(UHP, sU^{T}, w)$

$H' = UHP =$ 

and $s' = sU^{T} =$ 

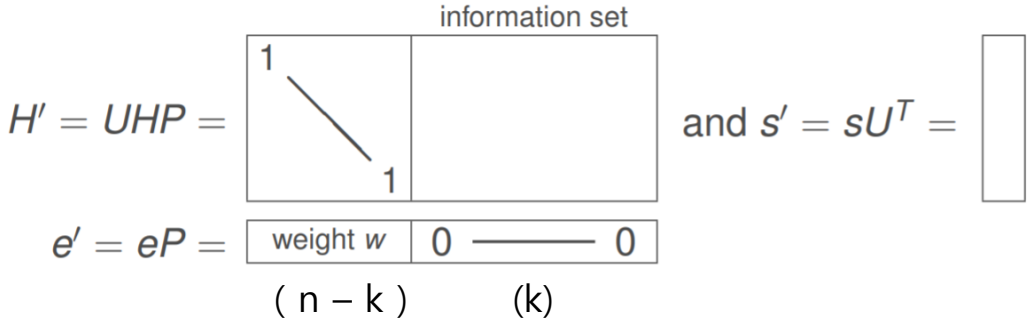
- Gaussian elimination(가우스 소거법) 을 사용하여 H에서 위의 H′ 형태의 행렬을 형성한다.

- 위의 과정을 성공할 때 까지 P 와 U 를 변경하며 계산한다.
  - 왼쪽의 (n – k) 행렬이 Linear independent(선형독립)하다면 행렬 뒤의 k 열은 information set을 형성한다.

참조: https://www.youtube.com/watch?v=2GKESu5atVQ

Information Set Decoding Attack

Step.



- 운이 좋다면 오류 위치는 information set 밖에 존재한다.
  - e′ = weight w, 그리고 s′ 의 weight 도 w 이다.

- $s\mathrm{U}^{\mathrm{T}}$ 의 weight 가 w 라면 성공
  - ($s\mathrm{U}^{\mathrm{T}}$, 0) P^{-1}를 반환 → Original Syndrome decoding 에 사용될 수 있다.

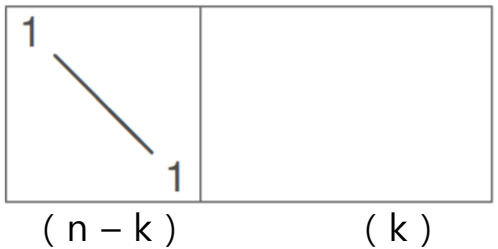
Information Set Decoding Attack

Algorithm

- input : $H \in \{0,1\}^{(n-k)\times n}$, $s \in \{0,1\}^{n-k}$, integer $w > 0$
- output : $e \in \{0,1\}^n$ such that $eH^T = s$ and $\mathrm{wt}(e) = w$

Repeat :

choose a permutation matrix P

$H' = UHP =$ 

and $s' = sU^T =$  (Gaussian elimination)

if weight (sU^T) = w, return (sU^T, 0) P^{-1}

Information Set Decoding Attack

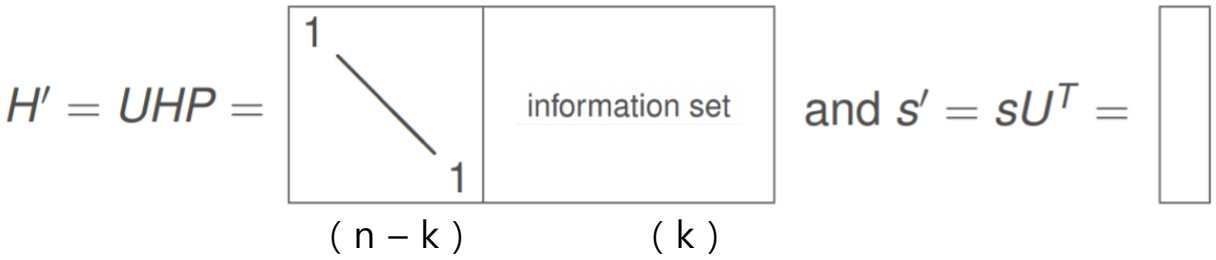
- Information set decoding 의 목표는 주어진 $eH^T = s$ 에서 w weight 의 e를 찾아내는 것
- 다시말해서 해결하고자 하는 문제는 n개의 변수를 가지고있는 n-k개의 방정식의 선형 시스템에 대하여 해를 찾는 것이며, 여기서 무게 조건 때문에 해가 독특하다.

- Information set decoding 은 여러 변형 버전이 있음.

- Information set decoding 공격이 공격법 중 가장 효율적인 것 뿐이지, 확실한 공격법은 아님
  - Scramble 된 G'에서 secret G를 찾아내는 구조 공격(Structural attack) 또한 대표적, 하지만 훨씬 느림
  - Syndrome Decoding 에서 Brute force 또한 적용가능, 하지만 비현실적
- Information set decoding은 어찌 보면 효율적인 Brute force attack

Information Set Decoding Attack

- Information set decoding 의 목표는 주어진 $eH^T = s$ 에서 w weight 의 e를 찾아내는 것
- 다시말해서 해결하고자 하는 문제는 n개의 변수를 가지고있는 n-k개의 방정식의 선형 시스템에 대하여 해를 찾는 것이며, 여기서 무게 조건 때문에 해가 독특하다.
- 주어진 k 열의 오류 벡터가 zero 라면 error position은 남아있는 n-k 에 존재하게 된다.
  다시 말해서 k 에 해당하는 변수들이 선형시스템에 포함되지 않는다면, n-k 개의 변수를 가지고 있는 n-k 방정식의 선형 시스템을 해결함으로써 오류 벡터를 찾아낼 수 있다.



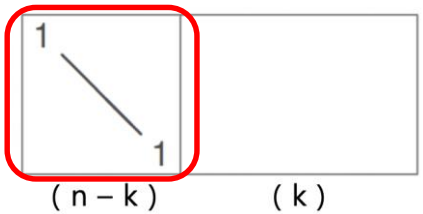
- 어려운 부분은 k – set 을 찾아내는 것이다.
  최종적으로 Solving (n-k) equations, (n-k) variables and returning 1 iff error vector has weight w

Information Set Decoding Attack

step1. Gaussian Elimination

H에서 랜덤하게 n-k개의 열을 선택한다. → (n-k, n-k) 의 subset이 생겼다. → 선택된 열 ℓ subset에 대하여 행들의 선형조합을 통해 Gaussian Elimination 수행

*선형조합 : 벡터들을 스칼라 배와 벡터 덧셈을 조합하여 새로운 벡터를 얻는 연산

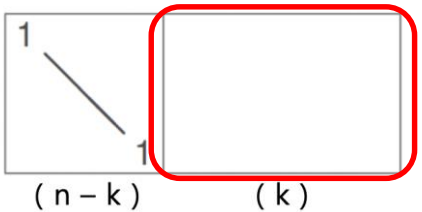


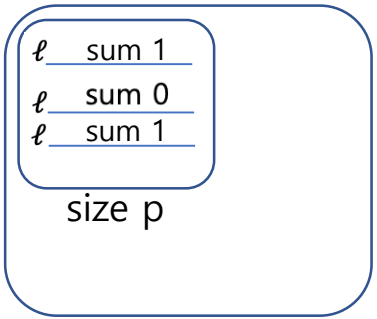
Information Set Decoding Attack

step2. Indexing

step1 에서 선택되지 않은 열의 index를 랜덤하게 쪼갠다. → X그룹과 Y그룹으로(same size)

X의 모든 size-p subset A에 대하여 sum(mod 2)을 ℓ 행에 매김으로서 ℓ-bit 벡터 $\pi(A)$ 획득
Y에도 동일하게 수행하여 $\pi(B)$ 획득





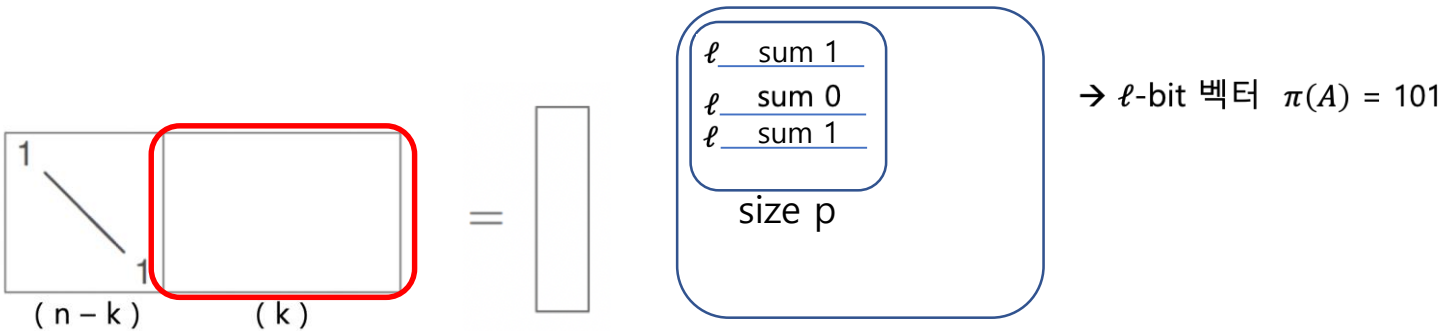
→ ℓ-bit 벡터 $\pi(A)$ = 101

Information Set Decoding Attack

step3. collision

각 $\pi(A)$ = $\pi(B)$ 에 대하여 A U B 의 2p 열들의 sum을 계산한다. → 이 합은 (n – k) bit 벡터가 됨

만약 이 sum의 Weight 가 w-2p 라면 이 A U B는 weight w의 codeword 를 형성



Quantum Information Set Decoding

- Grover 알고리즘은 데이터베이스 검색 뿐 아닌, 함수의 해를 찾는 분야에도 사용 된다.
- 이 관점에서 보아 Grover 알고리즘을 Information set decoding 에 적용하여 보자

  *Information set decoding 의 목표는 선형 시스템에 대하여 해를 찾는 것

- Grover 알고리즘은 size – *k* set 을 찾는데 사용된다.
  - 정확히는 size – *k* set 이 올바른지 검사하는 오라클에 적용된다.
  - 즉 선형 시스템의 n-k 다항식, n-k 변수를 푸는 것과, weight t 의 오류 벡터를 찾아 낸다.

감사합니다