# PIPO 고차마스킹

https://youtu.be/JhcVy-CtjdM

HANSUNG UNIVERSITY 한성대학교

CryptoCraft LAB
https://crypto.modoo.at

## 고차마스킹

- 암호화 알고리즘의 물리적 구현을 보호하기 위해 고차 마스킹 사용
- $x0 \perp x1 \perp \cdot \cdot \cdot \perp xd = x$
- 계산이 끝날 때 동일한 암호문 산출
- d 이하 중간 변수의 모든 튜플은 민감한 변수와 독립적
- 블록 암호에 대한 마스킹 체계를 설계하는 데 있어 가장 큰 어려움은 비선형 변환을 마스킹하는 것

  ex) ROR(X^m) = ROR(X)^ROR(m) , MIXCOLUMNS(X^m)= MIXCOLUMNS(X)^ MIXCOLUMNS(m)

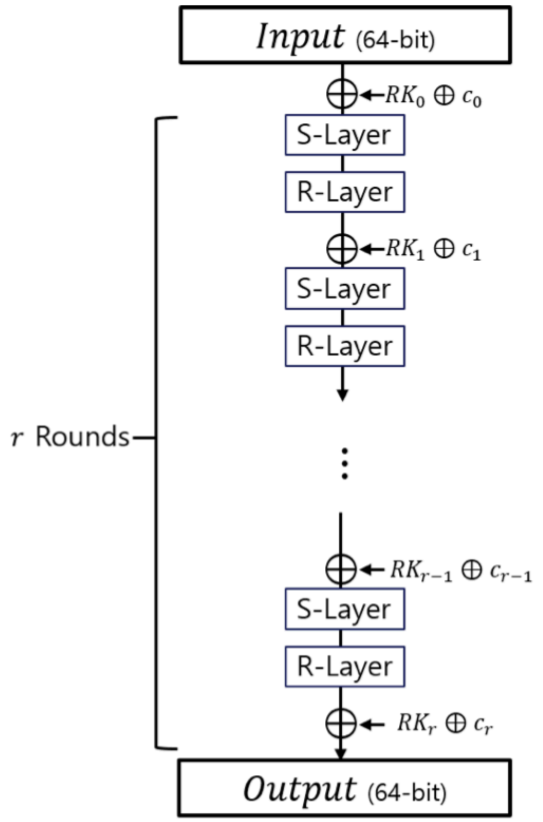
  SBOX(X^m) ≠ SBOX(X)^m ≠ SBOX(X)^SBOX(m)

# PIPO

- 효율적인 비트 슬라이싱 구현을 제공하는 새 경량 S- 박스를 개발하여 적용

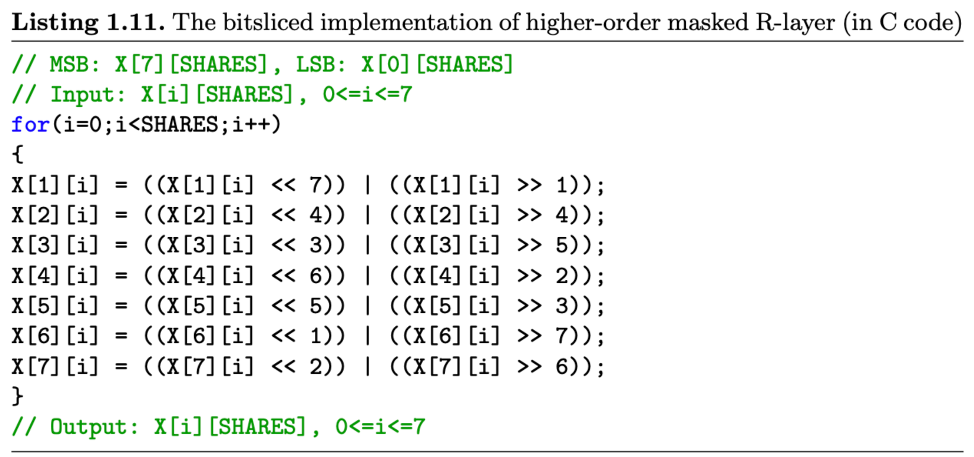
- 비트 슬라이싱 구현에 11 개의 비선형 비트 연산을 포함
  - ➢ 효율적인 고차 마스킹 가능

- 고차마스킹 레퍼런스 코드 X
  - ➢ 구현 필요
  - ➢ 비선형 레이어 S-Layer와 선형 레이어 R-Layer 가 반복
  - ➢ S-Layer의 비선형 변환 마스킹이 주요



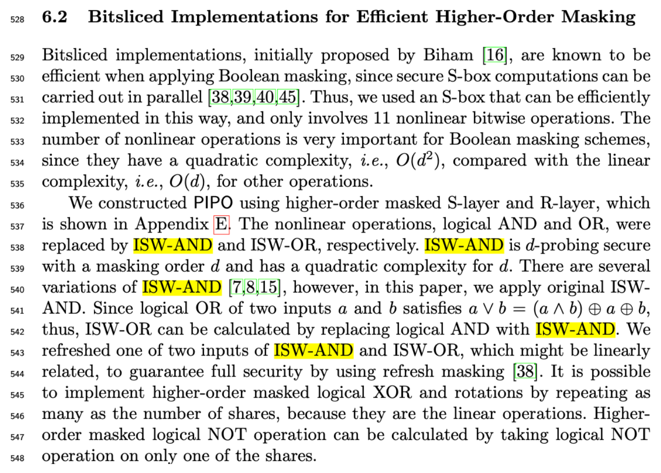
# PIPO 고차 마스킹

- 마스크 값 관리
- Mask_refreshing
- ISW_AND
- ISW_OR



**Listing 1.11.** The bitsliced implementation of higher-order masked R-layer (in C code)

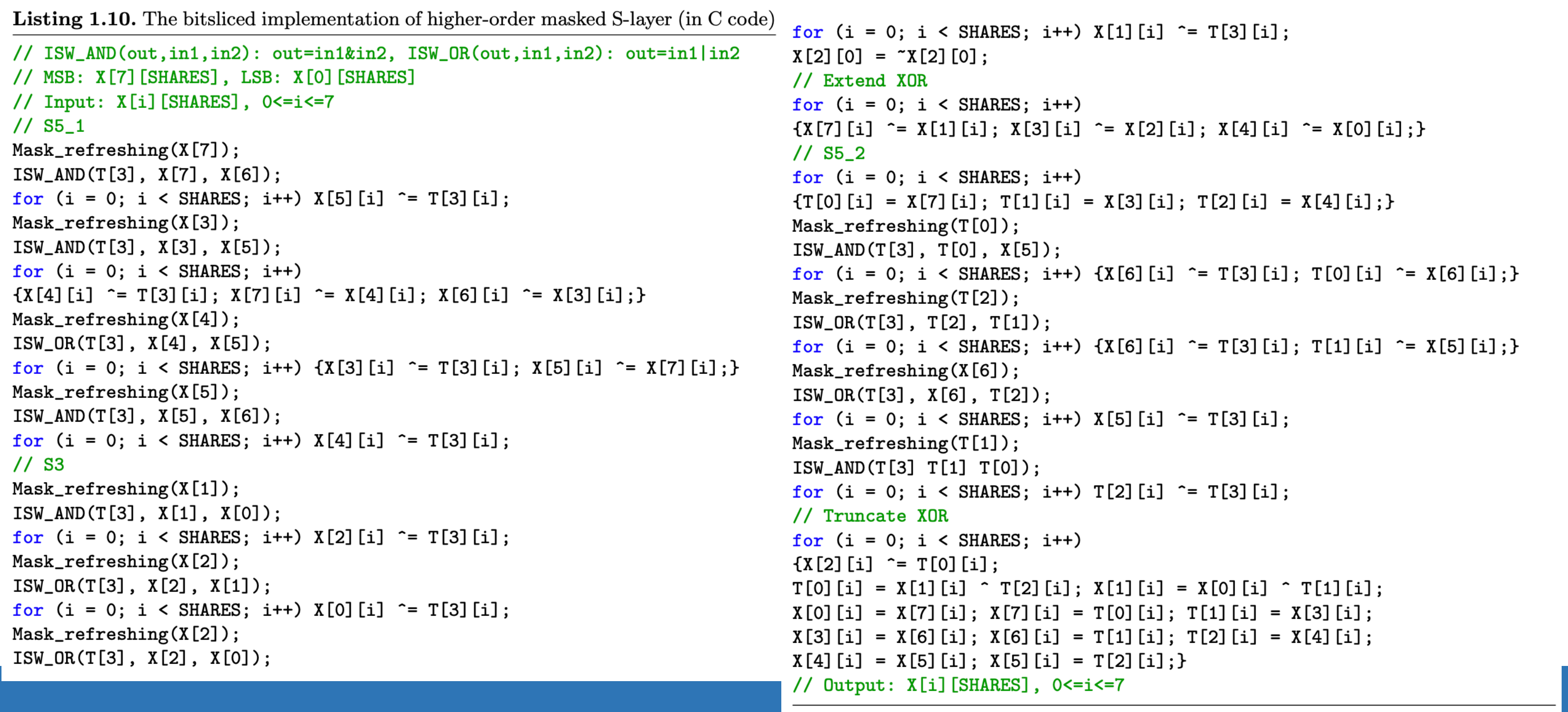
```
// MSB: X[7][SHARES], LSB: X[0][SHARES]
// Input: X[i][SHARES], 0<=i<=7
for(i=0;i<SHARES;i++)
{
X[1][i] = ((X[1][i] << 7)) | ((X[1][i] >> 1));
X[2][i] = ((X[2][i] << 4)) | ((X[2][i] >> 4));
X[3][i] = ((X[3][i] << 3)) | ((X[3][i] >> 5));
X[4][i] = ((X[4][i] << 6)) | ((X[4][i] >> 2));
X[5][i] = ((X[5][i] << 5)) | ((X[5][i] >> 3));
X[6][i] = ((X[6][i] << 1)) | ((X[6][i] >> 7));
X[7][i] = ((X[7][i] << 2)) | ((X[7][i] >> 6));
}
// Output: X[i][SHARES], 0<=i<=7
```



### 6.2 Bitsliced Implementations for Efficient Higher-Order Masking

Bitsliced implementations, initially proposed by Biham [16], are known to be efficient when applying Boolean masking, since secure S-box computations can be carried out in parallel [38,39,40,45]. Thus, we used an S-box that can be efficiently implemented in this way, and only involves 11 nonlinear bitwise operations. The number of nonlinear operations is very important for Boolean masking schemes, since they have a quadratic complexity, *i.e.*, $O(d^2)$, compared with the linear complexity, *i.e.*, $O(d)$, for other operations.

We constructed PIPO using higher-order masked S-layer and R-layer, which is shown in Appendix E. The nonlinear operations, logical AND and OR, were replaced by ISW-AND and ISW-OR, respectively. ISW-AND is $d$-probing secure with a masking order $d$ and has a quadratic complexity for $d$. There are several variations of ISW-AND [7,8,15], however, in this paper, we apply original ISW-AND. Since logical OR of two inputs $a$ and $b$ satisfies $a \vee b = (a \wedge b) \oplus a \oplus b$, thus, ISW-OR can be calculated by replacing logical AND with ISW-AND. We refreshed one of two inputs of ISW-AND and ISW-OR, which might be linearly related, to guarantee full security by using refresh masking [38]. It is possible to implement higher-order masked logical XOR and rotations by repeating as many as the number of shares, because they are the linear operations. Higher-order masked logical NOT operation can be calculated by taking logical NOT operation on only one of the shares.

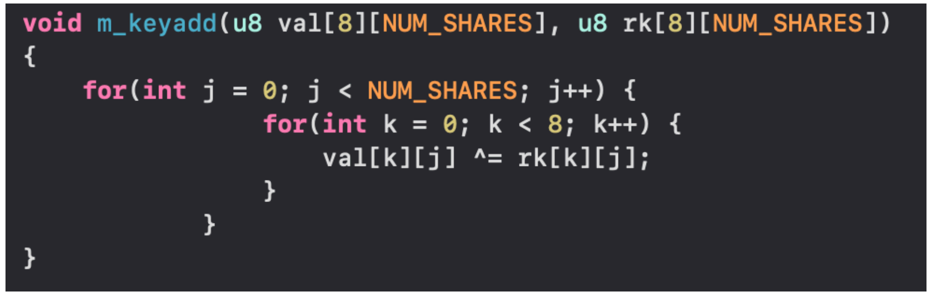


**Listing 1.10.** The bitsliced implementation of higher-order masked S-layer (in C code)

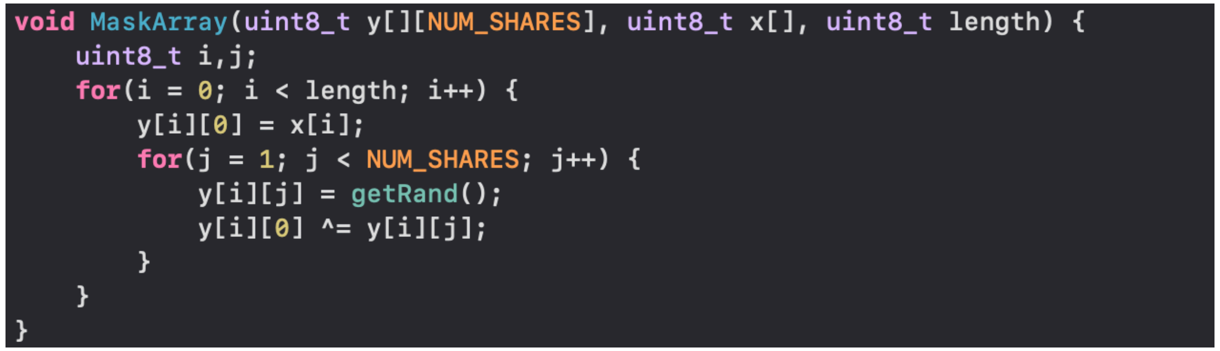
```
// ISW_AND(out,in1,in2): out=in1&in2, ISW_OR(out,in1,in2): out=in1|in2
// MSB: X[7][SHARES], LSB: X[0][SHARES]
// Input: X[i][SHARES], 0<=i<=7
// S5_1
Mask_refreshing(X[7]);
ISW_AND(T[3], X[7], X[6]);
for (i = 0; i < SHARES; i++) X[5][i] ^= T[3][i];
Mask_refreshing(X[3]);
ISW_AND(T[3], X[3], X[5]);
for (i = 0; i < SHARES; i++)
{X[4][i] ^= T[3][i]; X[7][i] ^= X[4][i]; X[6][i] ^= X[3][i];}
Mask_refreshing(X[4]);
ISW_OR(T[3], X[4], X[5]);
for (i = 0; i < SHARES; i++) {X[3][i] ^= T[3][i]; X[5][i] ^= X[7][i];}
Mask_refreshing(X[5]);
ISW_AND(T[3], X[5], X[6]);
for (i = 0; i < SHARES; i++) X[4][i] ^= T[3][i];
// S3
Mask_refreshing(X[1]);
ISW_AND(T[3], X[1], X[0]);
for (i = 0; i < SHARES; i++) X[2][i] ^= T[3][i];
Mask_refreshing(X[2]);
ISW_OR(T[3], X[2], X[1]);
for (i = 0; i < SHARES; i++) X[0][i] ^= T[3][i];
Mask_refreshing(X[2]);
ISW_OR(T[3], X[2], X[0]);
for (i = 0; i < SHARES; i++) X[1][i] ^= T[3][i];
X[2][0] = ~X[2][0];
// Extend XOR
for (i = 0; i < SHARES; i++)
{X[7][i] ^= X[1][i]; X[3][i] ^= X[2][i]; X[4][i] ^= X[0][i];}
// S5_2
for (i = 0; i < SHARES; i++)
{T[0][i] = X[7][i]; T[1][i] = X[3][i]; T[2][i] = X[4][i];}
Mask_refreshing(T[0]);
ISW_AND(T[3], T[0], X[5]);
for (i = 0; i < SHARES; i++) {X[6][i] ^= T[3][i]; T[0][i] ^= X[6][i];}
Mask_refreshing(T[2]);
ISW_OR(T[3], T[2], T[1]);
for (i = 0; i < SHARES; i++) {X[6][i] ^= T[3][i]; T[1][i] ^= X[5][i];}
Mask_refreshing(X[6]);
ISW_OR(T[3], X[6], T[2]);
for (i = 0; i < SHARES; i++) X[5][i] ^= T[3][i];
Mask_refreshing(T[1]);
ISW_AND(T[3] T[1] T[0]);
for (i = 0; i < SHARES; i++) T[2][i] ^= T[3][i];
// Truncate XOR
for (i = 0; i < SHARES; i++)
{X[2][i] ^= T[0][i];
T[0][i] = X[1][i] ^ T[2][i]; X[1][i] = X[0][i] ^ T[1][i];
X[0][i] = X[7][i]; X[7][i] = T[0][i]; T[1][i] = X[3][i];
X[3][i] = X[6][i]; X[6][i] = T[1][i]; T[2][i] = X[4][i];
X[4][i] = X[5][i]; X[5][i] = T[2][i];}
// Output: X[i][SHARES], 0<=i<=7
```

# 고차 마스크 적용

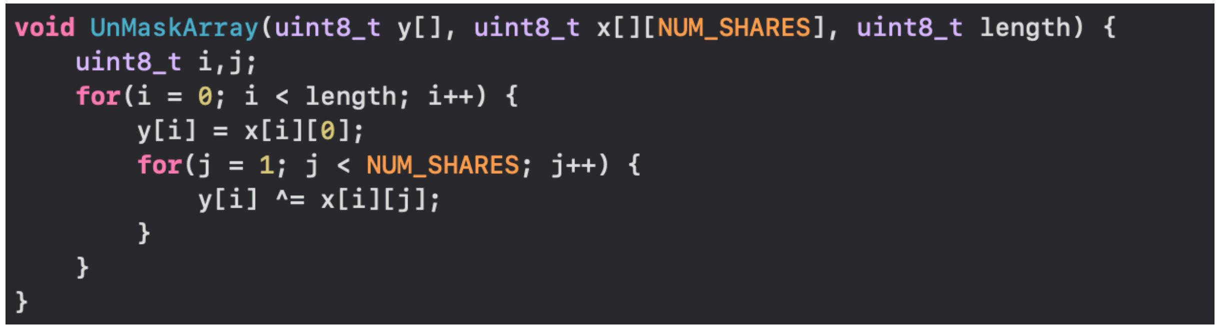
- MaskArray : 랜덤 마스크를 생성하여 적용하고 배열로 마스크 값 저장
- UnMaskArray : 배열에 저장된 마스크 값을 연산하여 마스크 값 제거



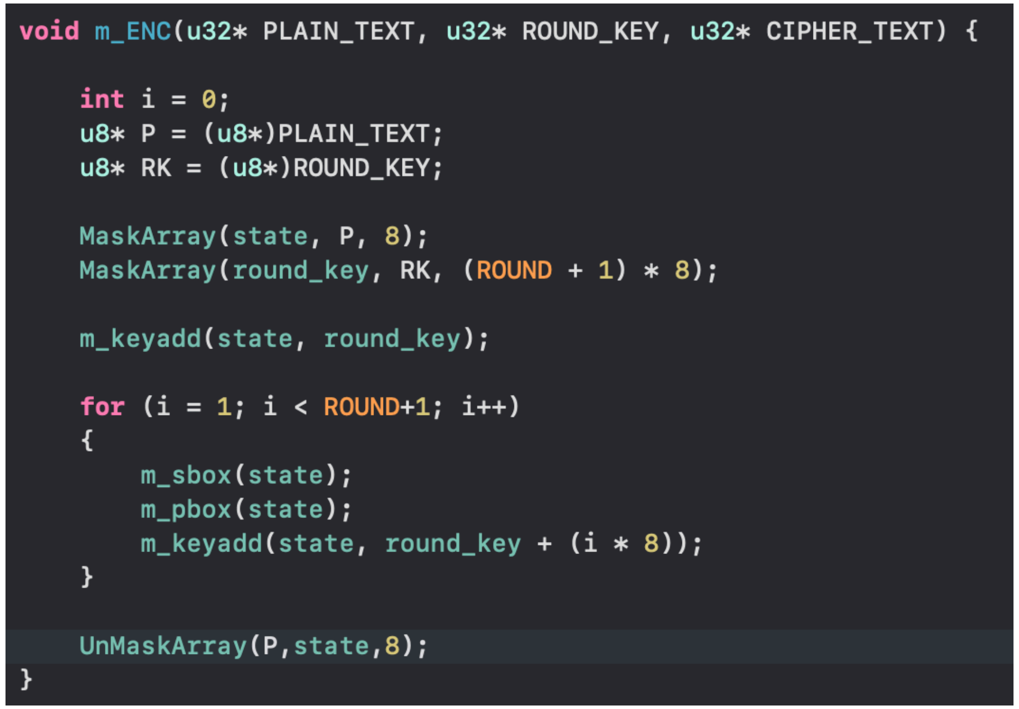
```
void m_keyadd(u8 val[8][NUM_SHARES], u8 rk[8][NUM_SHARES])
{
    for(int j = 0; j < NUM_SHARES; j++) {
                for(int k = 0; k < 8; k++) {
                    val[k][j] ^= rk[k][j];
                }
            }
}
```



```
void MaskArray(uint8_t y[][NUM_SHARES], uint8_t x[], uint8_t length) {
    uint8_t i,j;
    for(i = 0; i < length; i++) {
        y[i][0] = x[i];
        for(j = 1; j < NUM_SHARES; j++) {
            y[i][j] = getRand();
            y[i][0] ^= y[i][j];
        }
    }
}
```

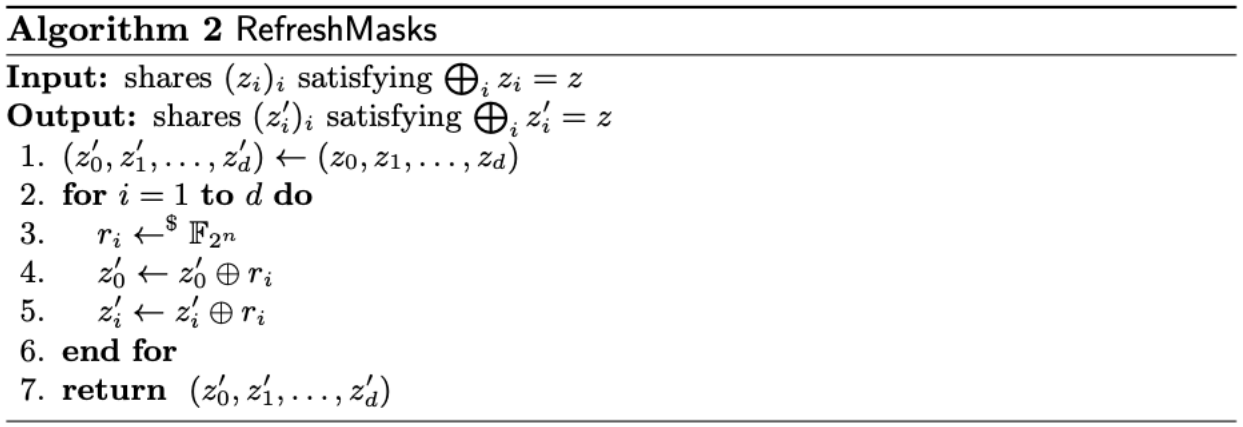


```
void UnMaskArray(uint8_t y[], uint8_t x[][NUM_SHARES], uint8_t length) {
    uint8_t i,j;
    for(i = 0; i < length; i++) {
        y[i] = x[i][0];
        for(j = 1; j < NUM_SHARES; j++) {
            y[i] ^= x[i][j];
        }
    }
}
```



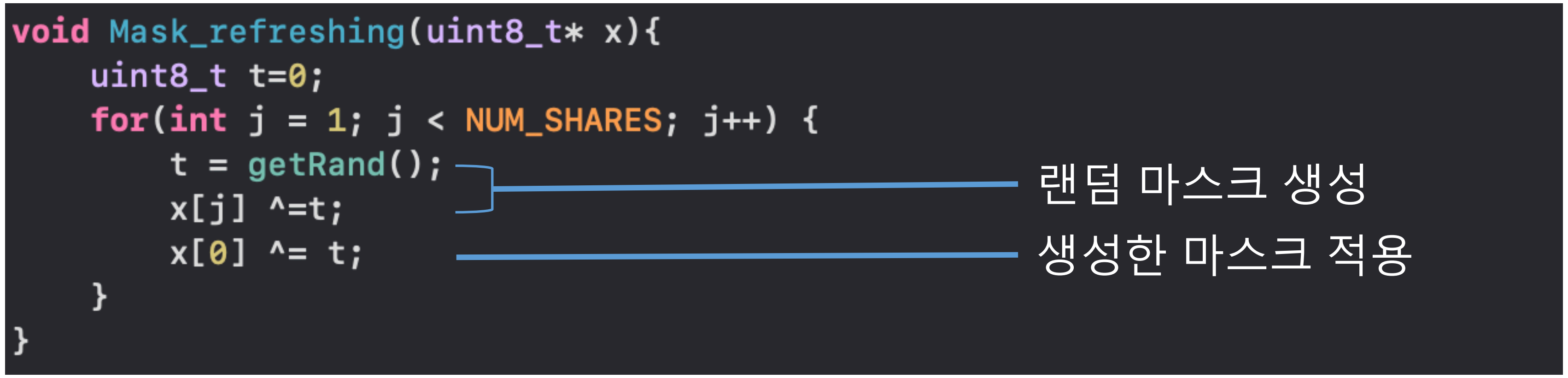
```
void m_ENC(u32* PLAIN_TEXT, u32* ROUND_KEY, u32* CIPHER_TEXT) {

    int i = 0;
    u8* P = (u8*)PLAIN_TEXT;
    u8* RK = (u8*)ROUND_KEY;

    MaskArray(state, P, 8);
    MaskArray(round_key, RK, (ROUND + 1) * 8);

    m_keyadd(state, round_key);

    for (i = 1; i < ROUND+1; i++)
    {
        m_sbox(state);
        m_pbox(state);
        m_keyadd(state, round_key + (i * 8));
    }

    UnMaskArray(P,state,8);
}
```

# RefreshMasks



**Algorithm 2** RefreshMasks

```
Input: shares $(z_i)_i$ satisfying $\bigoplus_i z_i = z$
Output: shares $(z'_i)_i$ satisfying $\bigoplus_i z'_i = z$
1. $(z'_0, z'_1, \ldots, z'_d) \leftarrow (z_0, z_1, \ldots, z_d)$
2. for $i = 1$ to $d$ do
3.     $r_i \xleftarrow{\$} \mathbb{F}_{2^n}$
4.     $z'_0 \leftarrow z'_0 \oplus r_i$
5.     $z'_i \leftarrow z'_i \oplus r_i$
6. end for
7. return $(z'_0, z'_1, \ldots, z'_d)$
```

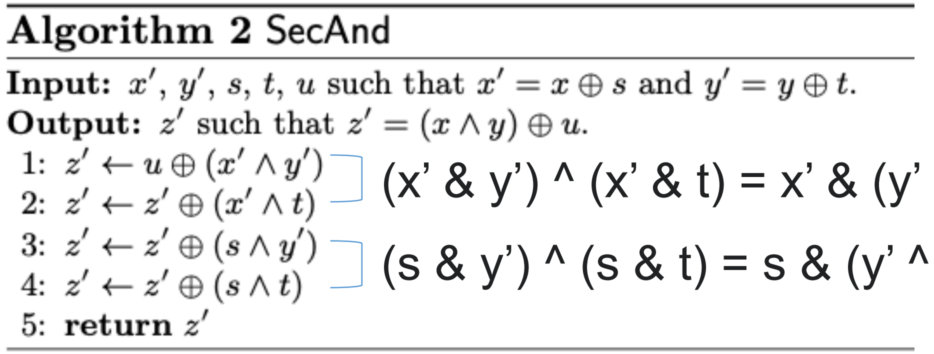


```c
void Mask_refreshing(uint8_t* x){
    uint8_t t=0;
    for(int j = 1; j < NUM_SHARES; j++) {
        t = getRand();
        x[j] ^=t;
        x[0] ^= t;
    }
}
```

랜덤 마스크 생성

생성한 마스크 적용

# SecAnd



**Algorithm 2** SecAnd

**Input:** $x'$, $y'$, $s$, $t$, $u$ such that $x' = x \oplus s$ and $y' = y \oplus t$.
**Output:** $z'$ such that $z' = (x \wedge y) \oplus u$.

```
1: $z' \leftarrow u \oplus (x' \wedge y')$
2: $z' \leftarrow z' \oplus (x' \wedge t)$
3: $z' \leftarrow z' \oplus (s \wedge y')$
4: $z' \leftarrow z' \oplus (s \wedge t)$
5: return $z'$
```

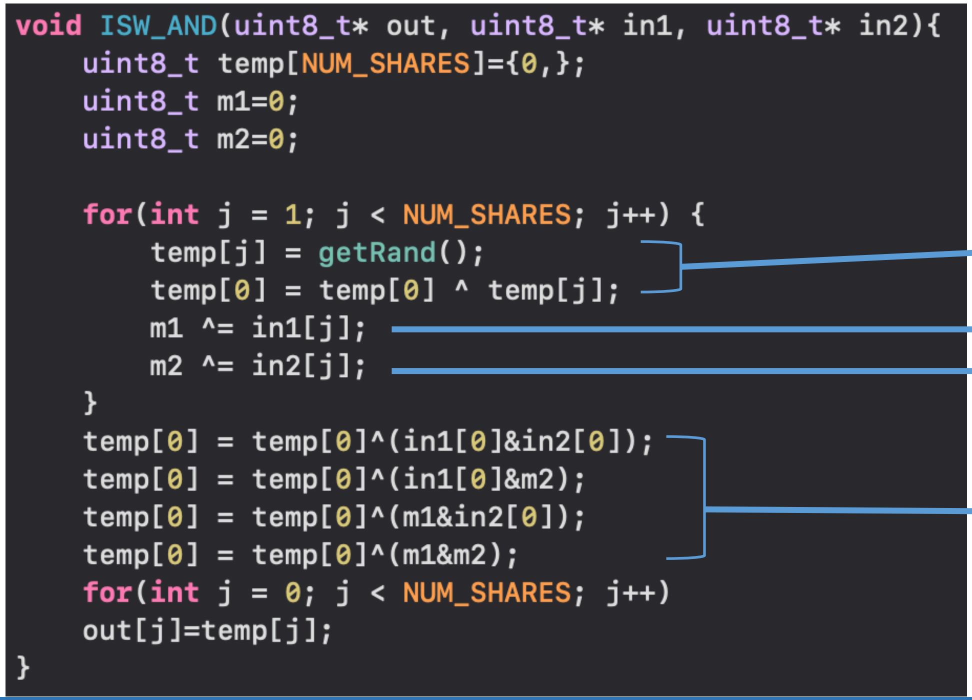
x ^ (y ^ z) = (x ^ y) ^ z

x & (y ^ z) = (x & y) ^ (x & z)

(x’ & y’) ^ (x’ & t) = x’ & (y’ ^ t) = x’ & y

(s & y’) ^ (s & t) = s & (y’ ^ t) = s & y

(x’ & y) ^ (s & y) = y & (x’ ^ s) = x & y



```c
void ISW_AND(uint8_t* out, uint8_t* in1, uint8_t* in2){
    uint8_t temp[NUM_SHARES]={0,};
    uint8_t m1=0;
    uint8_t m2=0;

    for(int j = 1; j < NUM_SHARES; j++) {
        temp[j] = getRand();
        temp[0] = temp[0] ^ temp[j];
        m1 ^= in1[j];
        m2 ^= in2[j];
    }
    temp[0] = temp[0]^(in1[0]&in2[0]);
    temp[0] = temp[0]^(in1[0]&m2);
    temp[0] = temp[0]^(m1&in2[0]);
    temp[0] = temp[0]^(m1&m2);
    for(int j = 0; j < NUM_SHARES; j++)
    out[j]=temp[j];
}
```

새로운 마스크 값 생성( u )

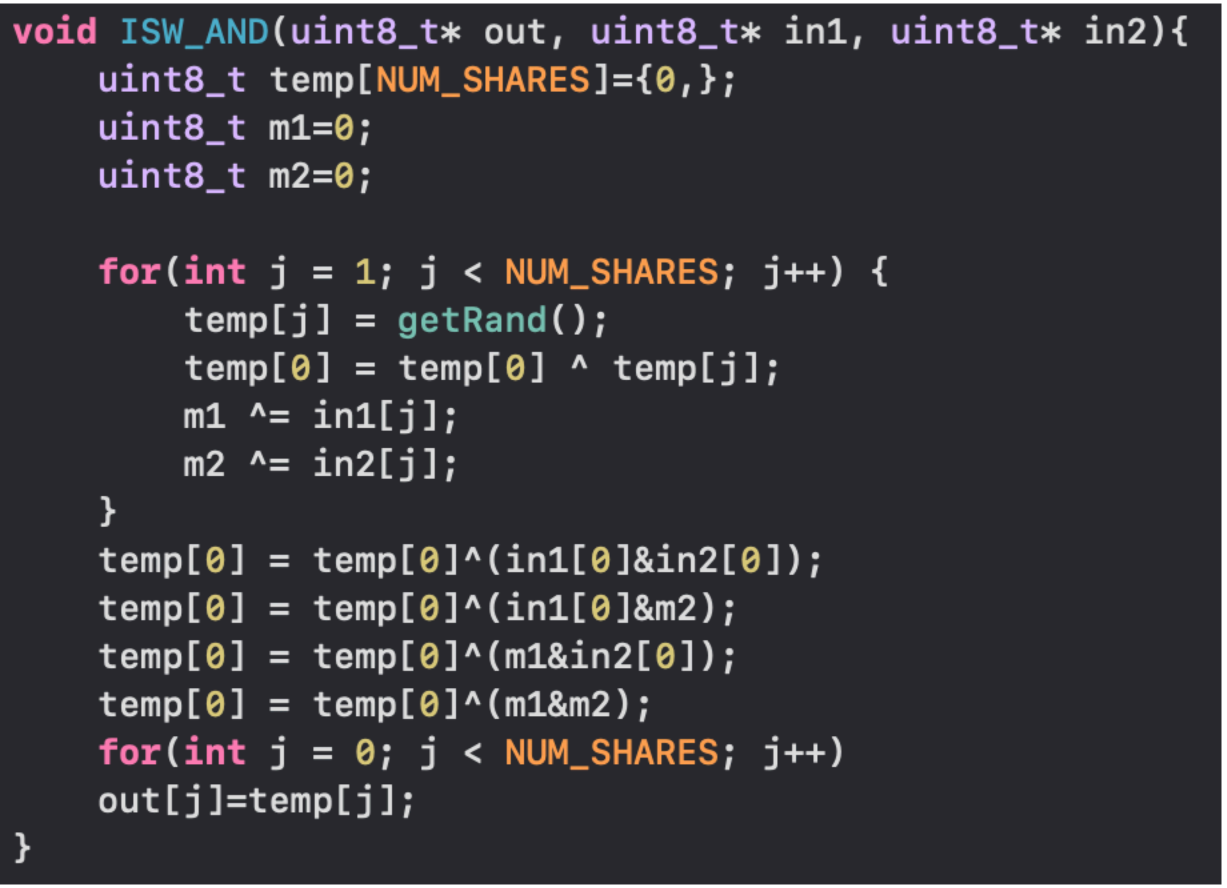
m1 : in1의현재 마스크 상태(s)

m2 : in2의현재 마스크 상태(t)

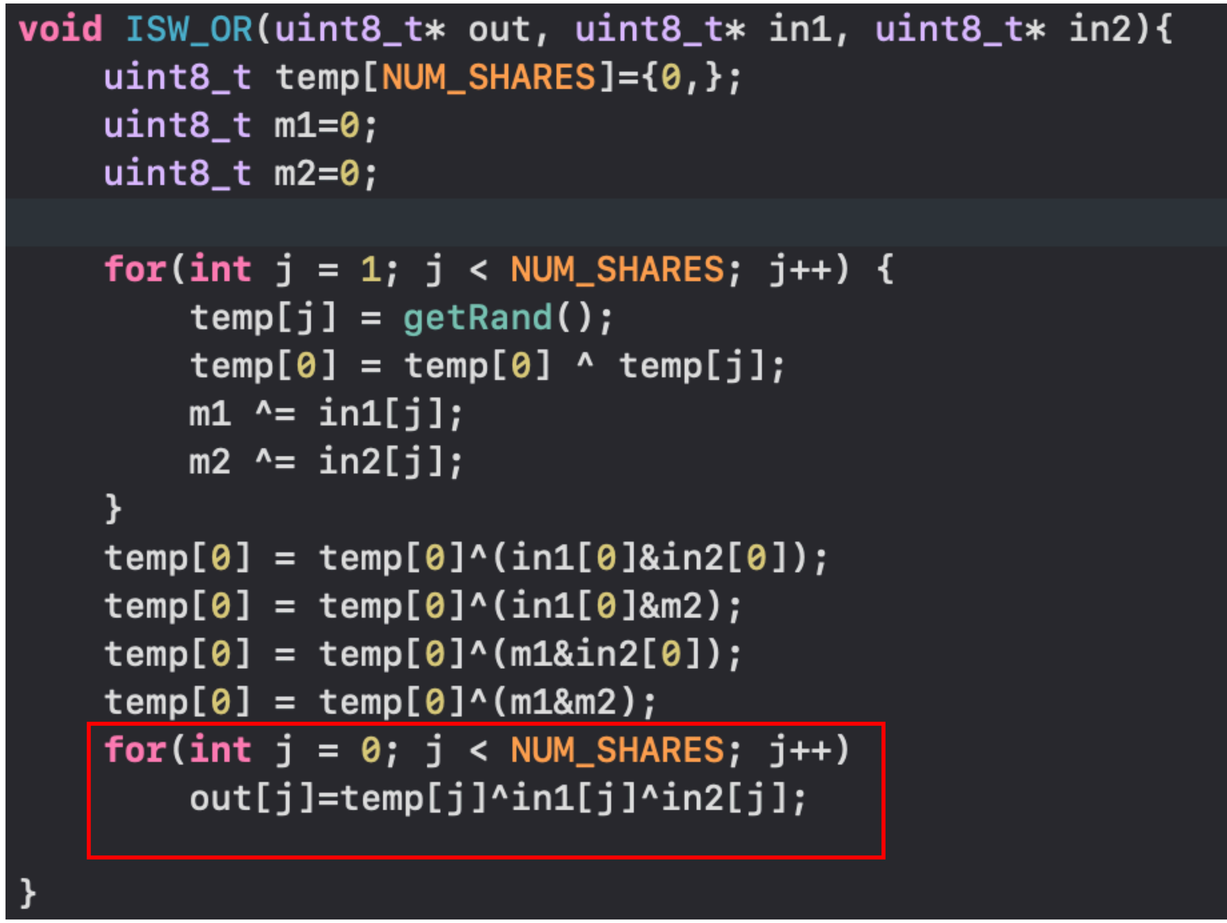
SecAnd 1,2,3,4

## ISW_OR

$$a \vee b = (a \wedge b) \oplus a \oplus b,$$



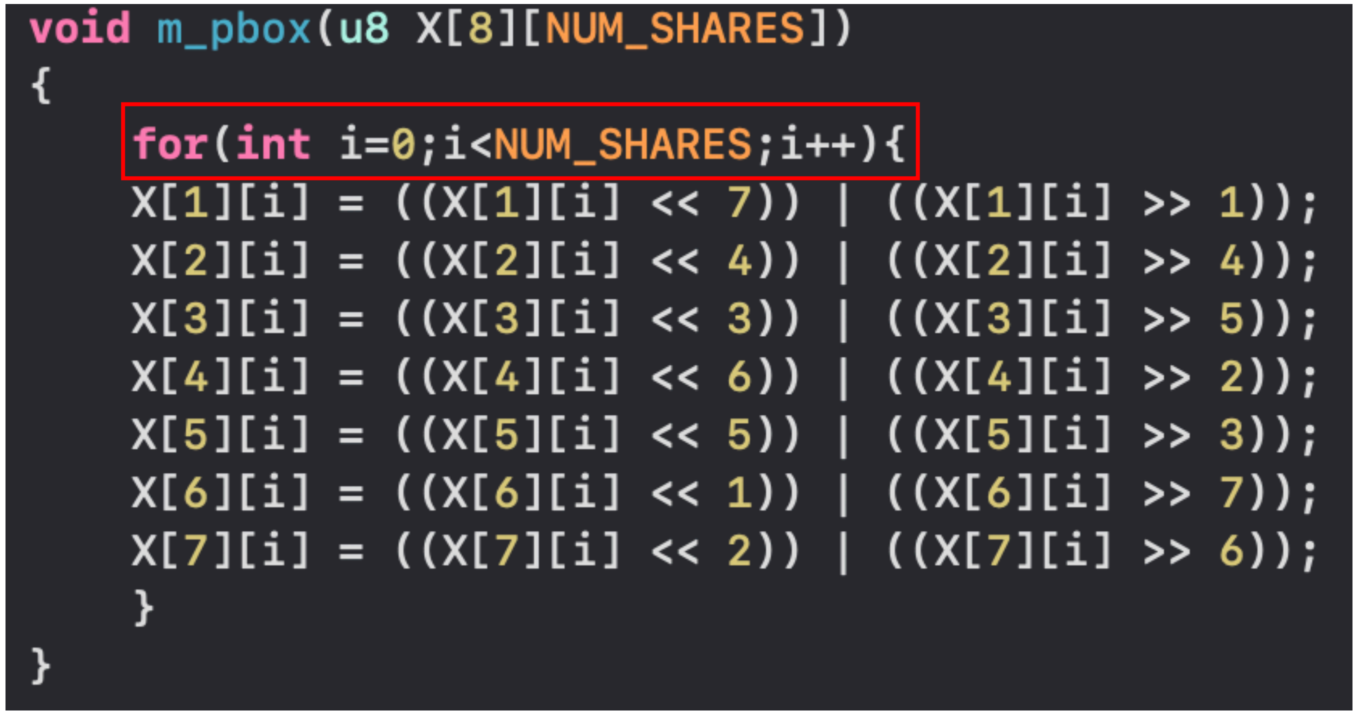
```c
void ISW_AND(uint8_t* out, uint8_t* in1, uint8_t* in2){
    uint8_t temp[NUM_SHARES]={0,};
    uint8_t m1=0;
    uint8_t m2=0;

    for(int j = 1; j < NUM_SHARES; j++) {
        temp[j] = getRand();
        temp[0] = temp[0] ^ temp[j];
        m1 ^= in1[j];
        m2 ^= in2[j];
    }
    temp[0] = temp[0]^(in1[0]&in2[0]);
    temp[0] = temp[0]^(in1[0]&m2);
    temp[0] = temp[0]^(m1&in2[0]);
    temp[0] = temp[0]^(m1&m2);
    for(int j = 0; j < NUM_SHARES; j++)
    out[j]=temp[j];
}
```



```c
void ISW_OR(uint8_t* out, uint8_t* in1, uint8_t* in2){
    uint8_t temp[NUM_SHARES]={0,};
    uint8_t m1=0;
    uint8_t m2=0;

    for(int j = 1; j < NUM_SHARES; j++) {
        temp[j] = getRand();
        temp[0] = temp[0] ^ temp[j];
        m1 ^= in1[j];
        m2 ^= in2[j];
    }
    temp[0] = temp[0]^(in1[0]&in2[0]);
    temp[0] = temp[0]^(in1[0]&m2);
    temp[0] = temp[0]^(m1&in2[0]);
    temp[0] = temp[0]^(m1&m2);
    for(int j = 0; j < NUM_SHARES; j++)
        out[j]=temp[j]^in1[j]^in2[j];

}
```

## R-Layer

- 로테이션 연산 수행
- ROR(X^m) = ROR(X)^ROR(m)
- 배열에 저장된 마스크 값도 ROR 연산 수행



```
void m_pbox(u8 X[8][NUM_SHARES])
{
    for(int i=0;i<NUM_SHARES;i++){
    X[1][i] = ((X[1][i] << 7)) | ((X[1][i] >> 1));
    X[2][i] = ((X[2][i] << 4)) | ((X[2][i] >> 4));
    X[3][i] = ((X[3][i] << 3)) | ((X[3][i] >> 5));
    X[4][i] = ((X[4][i] << 6)) | ((X[4][i] >> 2));
    X[5][i] = ((X[5][i] << 5)) | ((X[5][i] >> 3));
    X[6][i] = ((X[6][i] << 1)) | ((X[6][i] >> 7));
    X[7][i] = ((X[7][i] << 2)) | ((X[7][i] >> 6));
    }
}
```

# 부채널 분석

- TVLA test 수행
- 고정 vs 랜덤
- 마스크 적용 전, 1차 마스크 적용, 1차 마스크 적용(Mask_refreshing 제거) TVLA test
- Chipwhisperer 8bit xmeaga, 암호화 과정의 파형 1000개 수집

## 마스크 적용 전

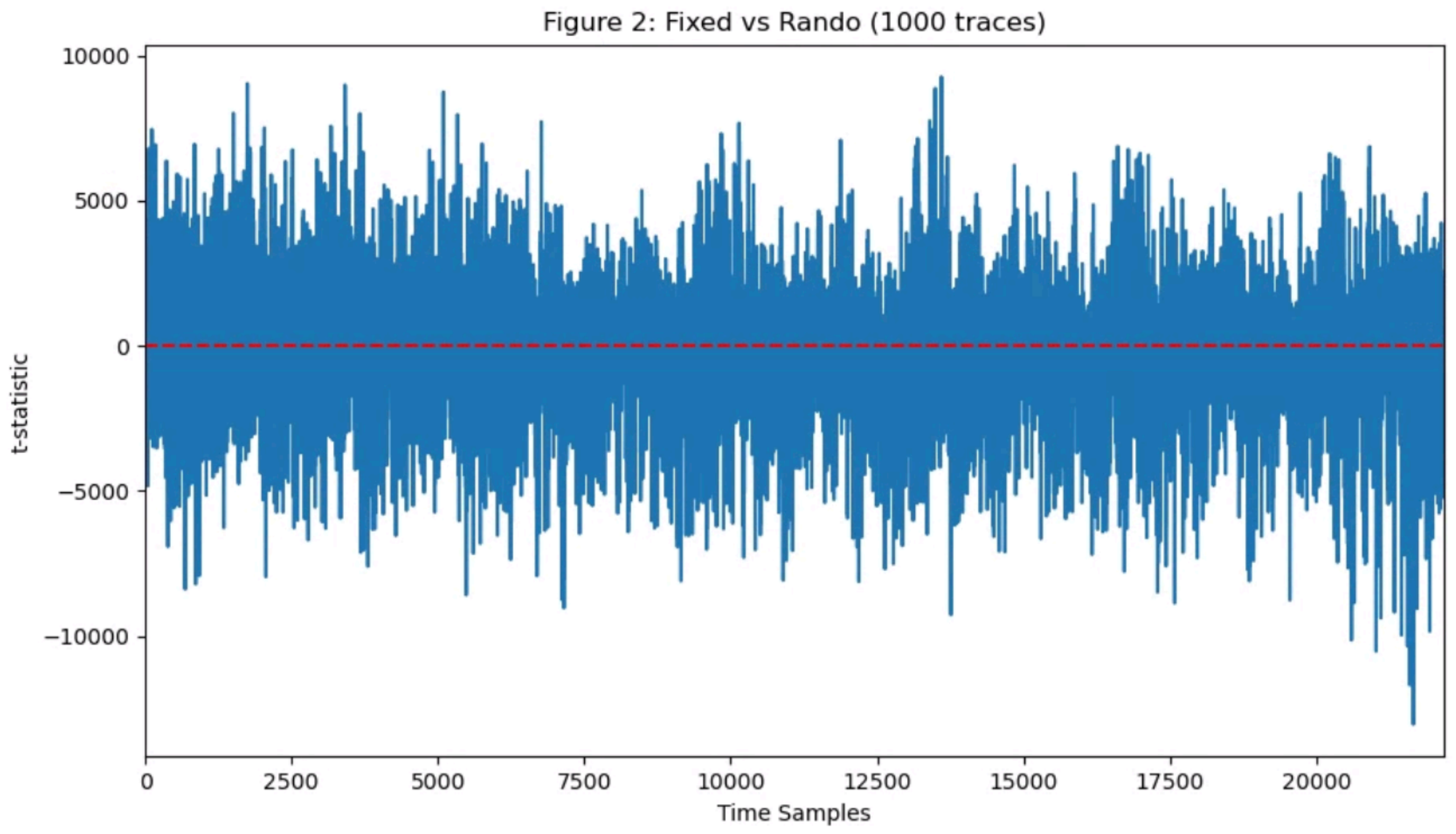


Figure 2: Fixed vs Rando (1000 traces)
10000
5000
0
−5000
−10000
t-statistic
0
2500
5000
7500
10000
12500
15000
17500
20000
Time Samples

## 마스크 적용 후

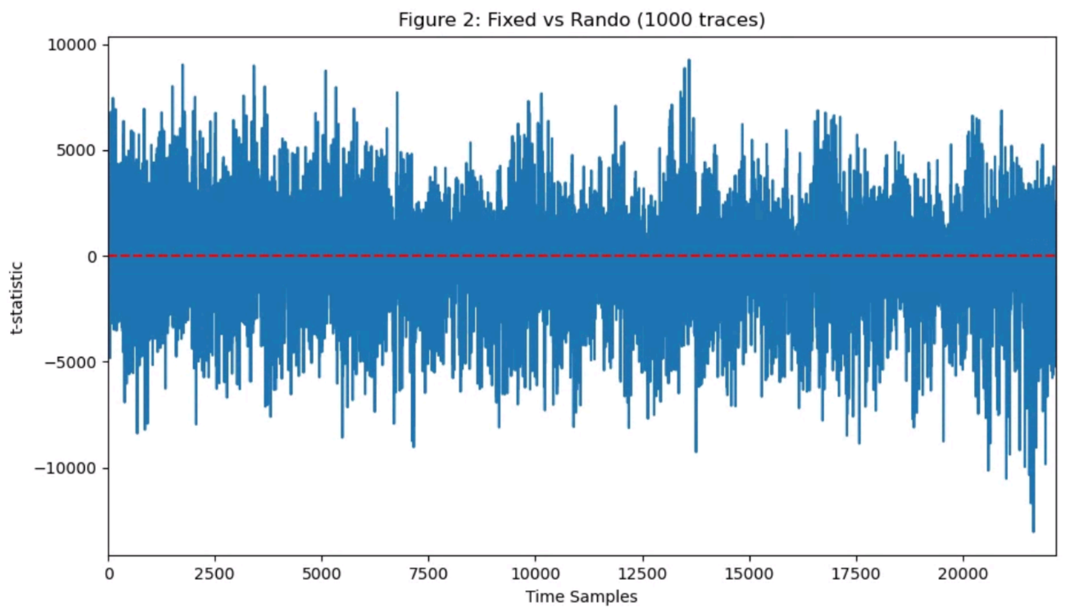


Figure 2: Fixed vs Rando (1000 traces)
t-statistic
Time Samples
10000
5000
0
−5000
−10000
0
2500
5000
7500
10000
12500
15000
17500
20000

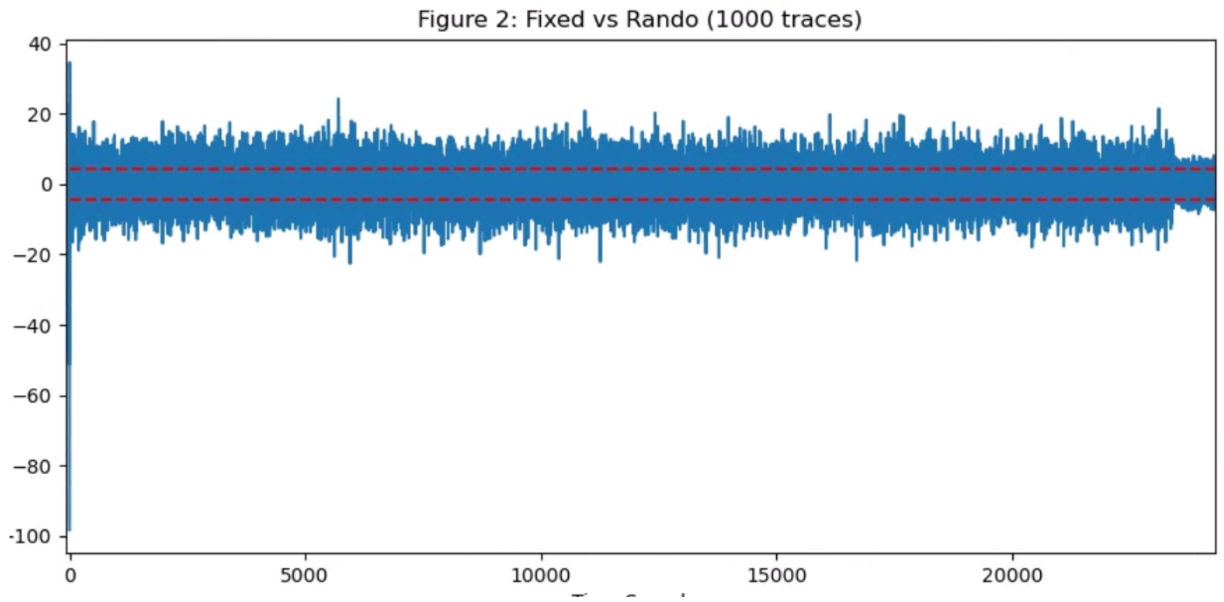
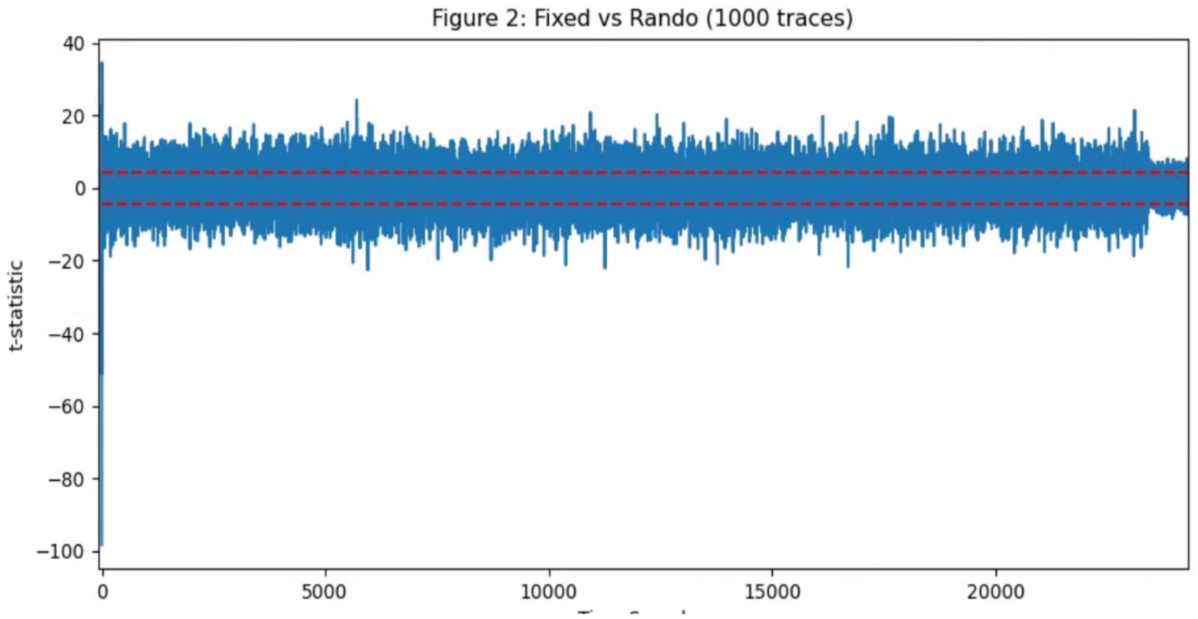
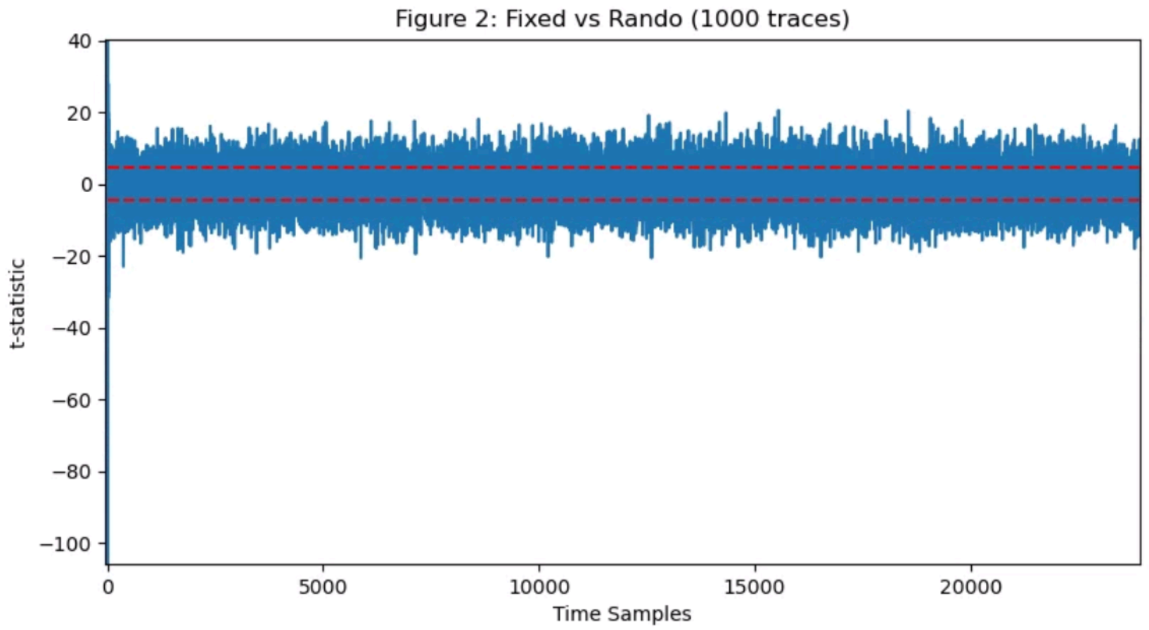


Figure 2: Fixed vs Rando (1000 traces)
40
20
0
−20
−40
−60
−80
-100
0
5000
10000
15000
20000

# 마스크 적용 후(Mask_refreshing 제거)





## 결론

- TVLA test 수행 결과 마스킹 적용 성공
- 속도차이가 100배 이상 차이
- 8bit 프로세서에 최적화된 암호이므로 최적화된 1~3차의 빠른 마스킹 구현 필요
- 테이블 구현 마스킹과 비교

# Q & A