PRESENT 마스킹 구현

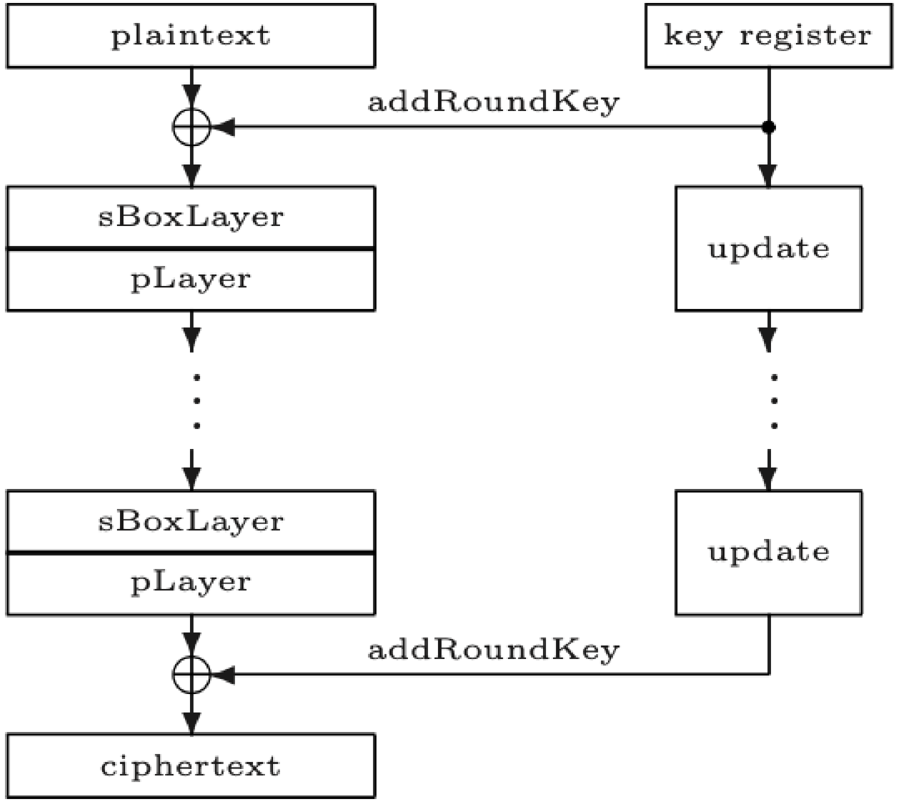
https://youtu.be/-KYuS23qwVY

HSU 한성대학교 HANSUNG UNIVERSITY

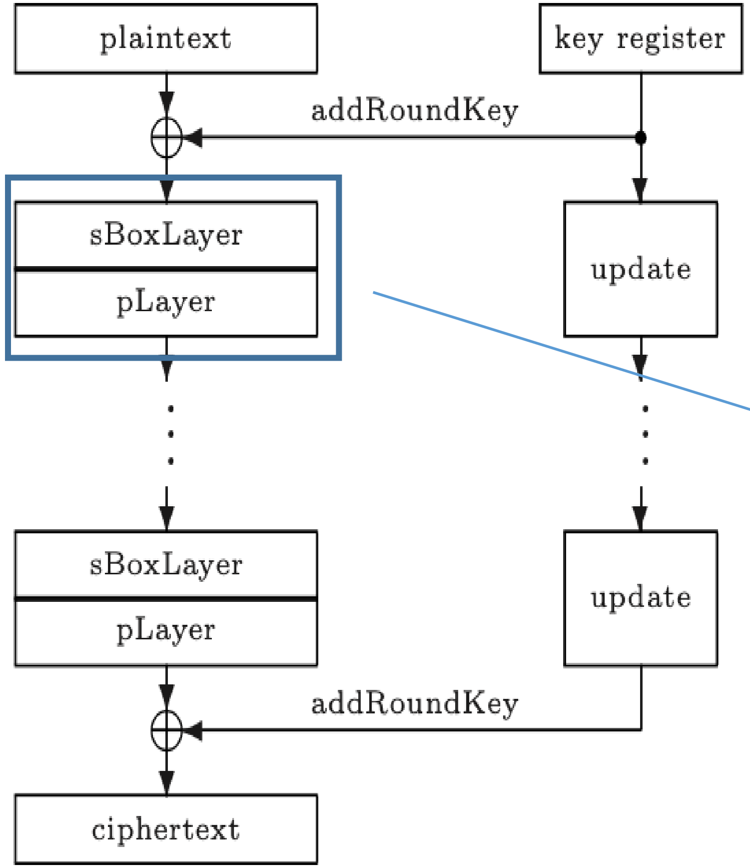
CryptoCraft LAB
https://crypto.modoo.at

PRESENT

- 2007년 제안된 SPN 구조의 경량 블록 암호
- 31 Round
- Block size : 64bit
- Key length : 80bit, 128bit



PRESENT



Algorithm 2. Our proposal for PRESENT encryption of

Input: A 64-bit block of plaintext B, a key K.
Output: A 64-bit block of ciphertext C.

1: $subkey = (subkey_1, subkey_2, ..., subkey_{32}) \leftarrow keySchedule($
2: $C \leftarrow B$
3: **for** $i = 1$ **to** 15 **do**
4: $\quad C \leftarrow C \oplus subkey_{2i-1}$
5: $\quad C \leftarrow P_0(C)$
6: $\quad C \leftarrow S_{BS}(C)$
7: $\quad C \leftarrow P_1(C)$
8: $\quad C \leftarrow C \oplus P(subkey_{2i})$
9: $\quad C \leftarrow S_{BS}(C)$
10: **end for**
11: $C \leftarrow C \oplus subkey_{31}$
12: $C \leftarrow P(C)$
13: $C \leftarrow S_{BS}(C)$
14: $C \leftarrow C \oplus subkey_{32}$
15: **return** C

```
void ENC_FUNC_16(u8* text, u16 (*sub_key)[4]){
    u16* text16 = (u16*)text;
    u32 i, j;

    for (i = 0; i < 15; i++) {
        for (j = 0; j < 4; j++) {
            text16[j] ^= (u16)(sub_key[2*i][j]);
        }

        PERMUTATE_FUNC_ZERO_16(&text16[0], &text16[1], &text16[2], &text16[3]);
        S_BOX_FUNC_16(&text16[0], &text16[1], &text16[2], &text16[3]);
        PERMUTATE_FUNC_ONE_16(&text16[0], &text16[1], &text16[2], &text16[3]);

        for (j = 0; j < 4; j++) {
            text16[j] ^= (u16)(sub_key[2*i + 1][j]);
        }
        S_BOX_FUNC_16(&text16[0], &text16[1], &text16[2], &text16[3]);

    }
    for (j = 0; j < 4; j++) {
        text16[j] ^= (u16)(sub_key[30][j]);
    }

    //P = P0 * P1
    PERMUTATE_FUNC_ONE_16(&text16[0], &text16[1], &text16[2], &text16[3]);
    PERMUTATE_FUNC_ZERO_16(&text16[0], &text16[1], &text16[2], &text16[3]);

    S_BOX_FUNC_16(&text16[0], &text16[1], &text16[2], &text16[3]);

    for (j = 0; j < 4; j++) {
        text16[j] ^= (u16)(sub_key[31][j]);
    }
}
```


Reis, T.B.; Aranha, D.F.; López, J. PRESENT runs fast. In Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems

PRESENT

PRESENT Runs Fast 649

The permutation P is specified by Eq. 1 below and moves the i-th bit of the state to the position $P(i)$:

$$P(i) = \begin{cases} 16i \mod 63, & \text{if } i \neq 63, \\ 63, & \text{if } i = 63. \end{cases} \quad (1)$$

From the definition of P, one can easily verify that $P^2 = P^{-1}$. By looking at Fig. 1, another interesting property of this permutation can be noticed: if the 64-bit state of the cipher is stored in four 16-bit registers, the application of the permutation P aligns the state in a way that the concatenation of the i-th bit of each of the four registers of the permuted state corresponds to 4 consecutive bits of the original state. These properties will be explored by the technique proposed later.

$$B = \begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 & 06 & 07 & 08 & 09 & 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \end{bmatrix}$$

$$P(B) = \begin{bmatrix} 00 & 04 & 08 & 12 & 16 & 20 & 24 & 28 & 32 & 36 & 40 & 44 & 48 & 52 & 56 & 60 \\ 01 & 05 & 09 & 13 & 17 & 21 & 25 & 29 & 33 & 37 & 41 & 45 & 49 & 53 & 57 & 61 \\ 02 & 06 & 10 & 14 & 18 & 22 & 26 & 30 & 34 & 38 & 42 & 46 & 50 & 54 & 58 & 62 \\ 03 & 07 & 11 & 15 & 19 & 23 & 27 & 31 & 35 & 39 & 43 & 47 & 51 & 55 & 59 & 63 \end{bmatrix}$$

ex)

$P(2) = \underbrace{16 \times 2}_{32} \mod 63$

$P^2(2) = P(32) = 16 \times 32 \mod 63$

↳ 512

↳ 512(8×63+8)

$\equiv 8 \mod 63$

$P(8) = 16 \times 8 \mod 63$

↳ 128

↳ (2×63+2)

$\equiv 2 \mod 63$

$\therefore P^{-1}(2) = \underbrace{P^2(2)}_{P(32)}$

PRESENT

$$P(i) = \begin{cases} 16i \mod 63, & \text{if } i \neq 63, \\ 63, & \text{if } i = 63. \end{cases} \qquad (1)$$

$$P_1 \circ P_0 = P^2$$

$$B = \begin{bmatrix} 00 & 01 & 02 & 03 & 04 & 05 & 06 & 07 & 08 & 09 & 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \end{bmatrix},$$

$$P_0(B) = \begin{bmatrix} 00 & 16 & 32 & 48 & 04 & 20 & 36 & 52 & 08 & 24 & 40 & 56 & 12 & 28 & 44 & 60 \\ 01 & 17 & 33 & 49 & 05 & 21 & 37 & 53 & 09 & 25 & 41 & 57 & 13 & 29 & 45 & 61 \\ 02 & 18 & 34 & 50 & 06 & 22 & 38 & 54 & 10 & 26 & 42 & 58 & 14 & 30 & 46 & 62 \\ 03 & 19 & 35 & 51 & 07 & 23 & 39 & 55 & 11 & 27 & 43 & 59 & 15 & 31 & 47 & 63 \end{bmatrix},$$

$$P_1(B) = \begin{bmatrix} 00 & 01 & 02 & 03 & 16 & 17 & 18 & 19 & 32 & 33 & 34 & 35 & 48 & 49 & 50 & 51 \\ 04 & 05 & 06 & 07 & 20 & 21 & 22 & 23 & 36 & 37 & 38 & 39 & 52 & 53 & 54 & 55 \\ 08 & 09 & 10 & 11 & 24 & 25 & 26 & 27 & 40 & 41 & 42 & 43 & 56 & 57 & 58 & 59 \\ 12 & 13 & 14 & 15 & 28 & 29 & 30 & 31 & 44 & 45 & 46 & 47 & 60 & 61 & 62 & 63 \end{bmatrix}.$$

Algorithm 2. Our proposal for PRESENT encryption of one message block.

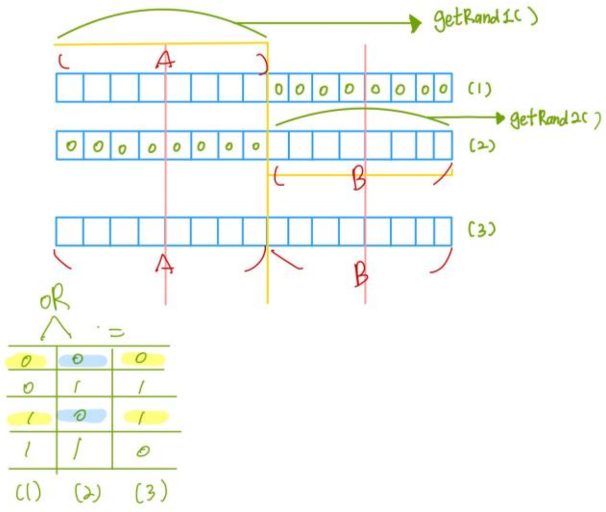
Input: A 64-bit block of plaintext B, a key K.
Output: A 64-bit block of ciphertext C.

$subkey = (subkey_1, subkey_2, ..., subkey_{32}) \leftarrow keySchedule(K)$
$C \leftarrow B$
for $i = 1$ **to** 15 **do**
  $C \leftarrow C \oplus subkey_{2i-1}$
  $C \leftarrow P_0(C)$
  $C \leftarrow S_{BS}(C)$
  $C \leftarrow P_1(C)$
  $C \leftarrow C \oplus P(subkey_{2i})$
  $C \leftarrow S_{BS}(C)$
end for
$C \leftarrow C \oplus subkey_{31}$
$C \leftarrow P(C)$
$C \leftarrow S_{BS}(C)$
$C \leftarrow C \oplus subkey_{32}$
return C

Reis, T.B.; Aranha, D.F.; López, J. PRESENT runs fast. In Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems

PRESENT 마스킹 구현

- getRand : 마스크 값 생성을 위해 rand() 사용
- MaskArray : 랜덤 마스크를 생성 후, 적용하여 배열로 마스크 값 저장
- UnMaskArray : 배열에 저장된 마스크 값 연산하여 마스크 값 제거




```
uint16_t getRand(void){
    u16 t1 = rand()%256;
    u16 t2 = rand()%256;
    t1 = t1 << 8;
    t2 = t1|t2;
    return t2;
}

void MaskArray(uint16_t y[][NUM_SHARES], uint16_t x[], uint16_t length){
    uint16_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(uint16_t y[], uint16_t x[][NUM_SHARES], uint16_t length){
    uint16_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];
        }
    }
}
```

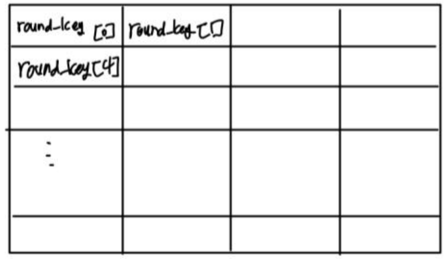
PRESENT 마스킹 구현

```c
void ENC_FUNC_16(u8* text, u16 (*sub_key)[4]){
    u16* text16 = (u16*)text;
    u32 i, j;

    for (i = 0; i < 15; i++) {
        for (j = 0; j < 4; j++) {
            text16[j] ^= (u16)(sub_key[2*i][j]);
        }

        PERMUTATE_FUNC_ZERO_16(&text16[0], &text16[1], &text16[2], &text16[3]);
        S_BOX_FUNC_16(&text16[0], &text16[1], &text16[2], &text16[3]);
        PERMUTATE_FUNC_ONE_16(&text16[0], &text16[1], &text16[2], &text16[3]);

        for (j = 0; j < 4; j++) {
            text16[j] ^= (u16)(sub_key[2*i + 1][j]);
        }
        S_BOX_FUNC_16(&text16[0], &text16[1], &text16[2], &text16[3]);

    }

    for (j = 0; j < 4; j++) {
        text16[j] ^= (u16)(sub_key[30][j]);
    }

    //P = P0 * P1
    PERMUTATE_FUNC_ONE_16(&
    PERMUTATE_FUNC_ZERO_16(&

    S_BOX_FUNC_16(&text16[0

    for (j = 0; j < 4; j++)
        text16[j] ^= (u16)(
    }
}
```

① 기존 sub_key 구조

→ 이런식으로 보기 쉽게 나열해둠.

② masking된 sub_key → static round_key로 변경.




```c
void m_ENC_FUNC_16(u8* text, u16 (*sub_key)[4]){
    u16* text16 = (u16*)text;
    u32 i, j, m;

    MaskArray(state, text16, 4);

    printf("₩nMasking Plaintext₩n");
    for(int k=0;k<4;k++){
        printf("%04X ",state[k][0]);
    }
    printf("₩n₩n");
    MaskArray(round_key, sub_key, 32*4);
        for (i = 0; i < 15; i++) {

            for (j = 0; j < 4; j++) {
                for(m = 0; m < NUM_SHARES; m++){
                state[j][m] ^= round_key[(i*8)+j][m];

                }
            }
            m_PERMUTATE_FUNC_ZERO_16(state[0], state[1], state[2], state[3]);
            m_S_BOX_FUNC_16(state[0], state[1], state[2], state[3]);
            m_PERMUTATE_FUNC_ONE_16(state[0],state[1], state[2], state[3]);

            for (j = 0; j < 4; j++) {
                for(m = 0; m < NUM_SHARES; m++){
                    state[j][m] ^= round_key[(2*4*i) + j + 4][m];
                }
            }
            m_S_BOX_FUNC_16(state[0], state[1], state[2], state[3]);

        }
```

PRESENT 마스킹 구현

4.1 Secure Computation of AND

Since Algorithm 1 contains AND operations, we first show how to secure the AND operation against first-order attacks. The technique is essentially the same as in [ISW03]. With $x = x' \oplus s$ and $y = y' \oplus t$ for two independent random masks s and t, we have for any u:

$$\boxed{(x \wedge y) \oplus u = \big((x' \oplus s) \wedge (y' \oplus t)\big) \oplus u = (x' \wedge y') \oplus (x' \wedge t) \oplus (s \wedge y') \oplus (s \wedge t) \oplus u}$$

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'

We see that the SecAnd algorithm requires 8 Boolean operations. The following Lemma shows that the SecAnd algorithm is secure against first-order attacks.

```
void ISW_AND(uint16_t* output, uint16_t* input1, uint16_t* input2, u16 m){
    uint16_t temp[NUM_SHARES] = {0, };
    uint16_t m1 = 0;
    uint16_t m2 = 0;


    for(int i = 1; i < NUM_SHARES; i++){
        temp[i] = m;
        temp[0] = temp[0] ^ temp[i];
        m1 ^= input1[i];
        m2 ^= input2[i];
    }
    temp[0] = temp[0] ^ (input1[0] & input2[0]);
    temp[0] = temp[0] ^ (input1[0] & m2);
    temp[0] = temp[0] ^ (m1 & input2[0]);
    temp[0] = temp[0] ^ (m1 & m2);

    for(int i = 0; i < NUM_SHARES; i++){
        output[i] = temp[i];
    }
}
```

PRESENT 마스킹 구현

Listing 1.2. Efficient implementation in C of the permutations P_0 and P_1 of our proposal for PRESENT encryption.

```c
/* The following macros permute two 64-bit blocks
 * simultaneously, using an auxiliary variable t
 * and storing one block on the high 16-bit word
 * of the 32-bit variables X0, X1, X2 and X3, and
 * the other block on the low 16-bit word of the
 * same variables.
 */

#define PRESENT_PERMUTATION_P0(X0,X1,X2,X3) \
    t = (X0^(X1>>1)) & 0x55555555;          \
    X0 = X0^t; X1 = X1^(t<<1);              \
    t = (X2^(X3>>1)) & 0x55555555;          \
    X2 = X2^t; X3 = X3^(t<<1);              \
    t = (X0^(X2>>2)) & 0x33333333;          \
    X0 = X0^t; X2 = X2^(t<<2);              \
    t = (X1^(X3>>2)) & 0x33333333;          \
    X1 = X1^t; X3 = X3^(t<<2);              \

#define PRESENT_PERMUTATION_P1(X0,X1,X2,X3) \
    t = (X0^(X1>>4)) & 0x0F0F0F0F;          \
    X0 = X0^t; X1 = X1^(t<<4);              \
    t = (X2^(X3>>4)) & 0x0F0F0F0F;          \
    X2 = X2^t; X3 = X3^(t<<4);              \
    t = (X0^(X2>>8)) & 0x00FF00FF;          \
    X0 = X0^t; X2 = X2^(t<<8);              \
    t = (X1^(X3>>8)) & 0x00FF00FF;          \
    X1 = X1^t; X3 = X3^(t<<8);              \
```

```c
//16-bit //16*4
void PERMUTATE_FUNC_ZERO_16(u16* x0_in, u16* x1_in, u16* x2_in, u16* x3_in) {
    u16 X0, X1, X2, X3;
    u16 t;

    X3 = *x0_in;
    X2 = *x1_in;
    X1 = *x2_in;
    X0 = *x3_in;

    t = (X0 ^ (ROR_u16(X1,1))) & 0x5555;
    X0 = X0 ^ t;
    X1 = X1 ^ (ROL_u16(t,1));

    t = (X2 ^ (ROR_u16(X3, 1))) & 0x5555;
    X2 = X2 ^ t;
    X3 = X3 ^ (ROL_u16(t, 1));

    t = (X0 ^ (ROR_u16(X2, 2))) & 0x3333;
    X0 = X0 ^ t;
    X2 = X2 ^ (ROL_u16(t, 2));

    t = (X1 ^ (ROR_u16(X3, 2))) & 0x3333;
    X1 = X1 ^ t;
    X3 = X3 ^ (ROL_u16(t, 2));

    *x0_in = X3;
    *x1_in = X2;
    *x2_in = X1;
    *x3_in = X0;
}
```

```c
for(int i = 0; i < NUM_SHARES; i++){
    t[i] = (X0[i]^ (ROR_u16(X1[i],1)));
}
ISW_AND(t, t, &k[0]);

for(int i = 0; i <NUM_SHARES; i++){
    X0[i] ^= t[i];
}

for(int i = 0; i < NUM_SHARES; i++){
    X1[i] ^= ROL_u16(t[i], 1);
}

for(int i = 0; i < NUM_SHARES; i++){
    t[i] = (X2[i]^(ROR_u16(X3[i], 1)));
}
ISW_AND(t, t, &k[0]);
```

PRESENT 마스킹 최적화

```c
uint16_t getRand(void){
    u16 t1 = rand()%256;
    u16 t2 = rand()%256;
    t1 = t1 << 8;
    t2 = t1|t2;
    return t2;
}

void MaskArray(uint16_t y[][NUM_SHARES], uint16_t x[], uint16_t length){
    uint16_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];
        }
    }
}
```

```c
void MaskArray(uint16_t y[][NUM_SHARES], uint16_t x[], uint16_t length){
    uint16_t i, j;
    u16 m2[4] = {0xE8F0, 0x3C95, 0x1864, 0xFAAD};

    for(i = 0; i < length; i++){
        y[i][0] = x[i];
        for(j = 1; j < NUM_SHARES; j++){
            for(int k=0;k<4;k++){
                y[i][j] = m2[k];
            }
            y[i][0] ^=y[i][j];
        }
    }
}
```

PRESENT 마스킹 최적화

- Permutate_func_zero_mask_opt

```c
for(int i = 0; i < NUM_SHARES; i++){
    t[i] = (X0[i]^ (ROR_u16(X1[i],1)));
}
ISW_AND(t, t, &k[0]);

for(int i = 0; i <NUM_SHARES; i++){
    X0[i] ^= t[i];
}

for(int i = 0; i < NUM_SHARES; i++){
    X1[i] ^= ROL_u16(t[i], 1);
}

for(int i = 0; i < NUM_SHARES; i++){
    t[i] = (X2[i]^(ROR_u16(X3[i], 1)));
}
ISW_AND(t, t, &k[0]);
```

```c
u16 m1[4] = {0x3957, 0x1664, 0x41B9, 0x3515};

for(int i =0; i<NUM_SHARES; i++){
    X3[i] = x0_in[i];
    X2[i] = x1_in[i];
    X1[i] = x2_in[i];
    X0[i] = x3_in[i];
}

//t[0] ->orginal
//t[1]->masking
//t[0] already masked state / t[1]

for(int i = 0; i < NUM_SHARES; i++){
    t[i] = (X0[i]^ (ROR_u16(X1[i],1)));
}
ISW_AND(t, t, &k[0], m1[0]);//m1[0] : X0 of mask
```

```c
void ISW_AND(uint16_t* output, uint16_t* input1, uint16_t* input2, u16 m){
    uint16_t temp[NUM_SHARES] = {0, };
    uint16_t m1 = 0;
    uint16_t m2 = 0;

    for(int i = 1; i < NUM_SHARES; i++){
        temp[i] = m;
        temp[0] = temp[0] ^ temp[i];
        m1 ^= input1[i];
        m2 ^= input2[i];
    }
    temp[0] = temp[0] ^ (input1[0] & input2[0]);
    temp[0] = temp[0] ^ (input1[0] & m2);
    temp[0] = temp[0] ^ (m1 & input2[0]);
    temp[0] = temp[0] ^ (m1 & m2);

    for(int i = 0; i < NUM_SHARES; i++){
        output[i] = temp[i];
    }
}
```

PRESENT 마스킹 최적화

- PRESENT 마스킹 최적화
  - 마스크 값 고정 (총 8개의 마스크 값 사용)
  - 마스킹된 비트슬라이싱 PRESENT에 비해 1.15배 속도 향상

enc_10,000번 동작	time	CPU cycle
비트슬라이스로 구현된 PRESENT(reference)	0.139344	4485
마스킹된 PRESENT	2.172269	934558
마스킹된 PRESENT 최적화	1.884144	396106

Q & A