Information Security

2st Project

2015410039 이현건

STEP 1. 해시함수 분석

우선, output Y가 256bit이므로 이 해시의 preimage resistant는 2^{256}이며, brute force로는 평균적으로 2^{m-1}, 2^{255}만큼의 해시를 실행해야 collision을 찾을 수 있습니다.

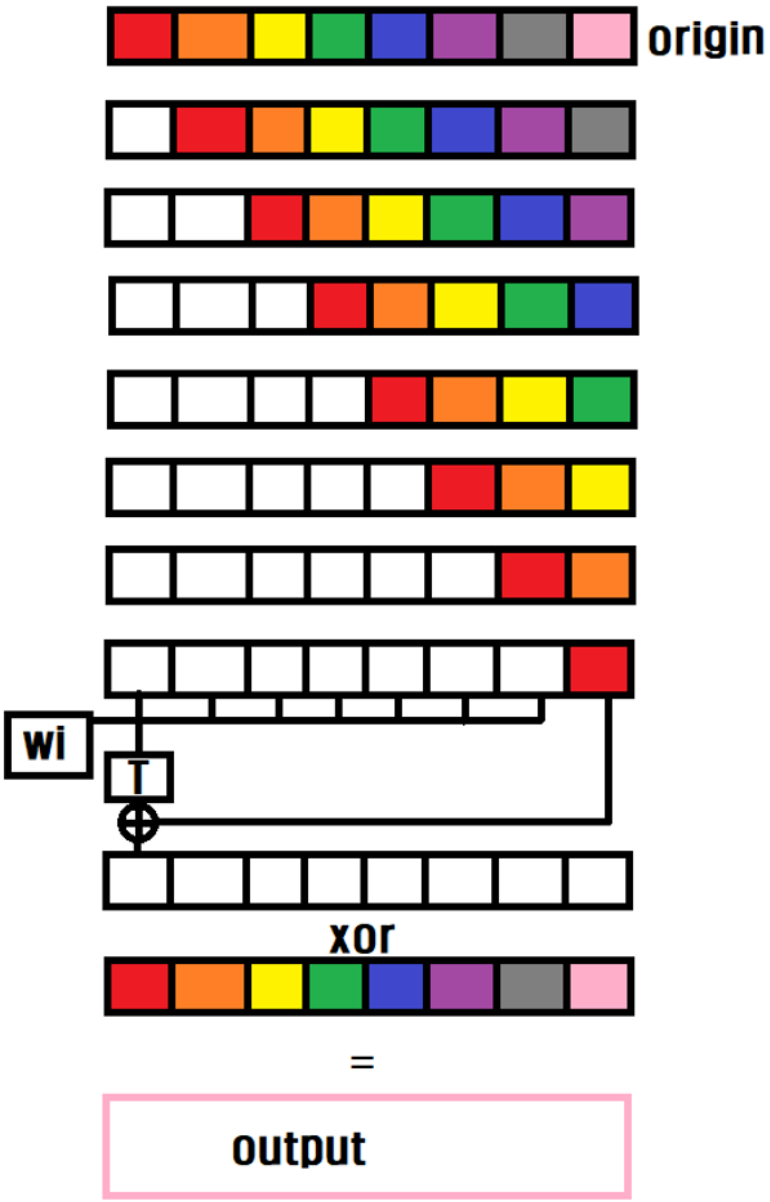
이 방법은 2^{253}내에 input X를 결정해야 한다는 조건을 만족하지 못하므로 사용할 수 없습니다.

Round function F를 보면, X_i의 A_i~G_i block은 X_{i+1}의 B_{i+1}~H_{i+1} block으로 이어지며, X_i의 H_i block과 W_i, A_i~G_i를 통해 계산된 T를 통해 H_{i+1}이 결정되는 구조입니다. 즉, output X_{i+1}만 주어진 상태에서 X_i를 계산하기 위해서는 H_i 또는 T의 값 둘 중 하나를 알아야 합니다.

$$T = (W_i[0], W_i[1], W_i[2], W_i[3])$$
$$T = MDS(S(T[0] \oplus A_i[0]), S(T[1] \oplus A_i[1]), S(T[2] \oplus A_i[2]), S(T[3] \oplus A_i[3]))$$
$$T = MDS(S(T[0] \oplus B_i[0]), S(T[1] \oplus B_i[1]), S(T[2] \oplus B_i[2]), S(T[3] \oplus B_i[3]))$$
$$T = MDS(S(T[0] \oplus C_i[0]), S(T[1] \oplus C_i[1]), S(T[2] \oplus C_i[2]), S(T[3] \oplus C_i[3]))$$
$$T = MDS(S(T[0] \oplus D_i[0]), S(T[1] \oplus D_i[1]), S(T[2] \oplus D_i[2]), S(T[3] \oplus D_i[3]))$$
$$T = MDS(S(T[0] \oplus E_i[0]), S(T[1] \oplus E_i[1]), S(T[2] \oplus E_i[2]), S(T[3] \oplus E_i[3]))$$
$$T = MDS(S(T[0] \oplus F_i[0]), S(T[1] \oplus F_i[1]), S(T[2] \oplus F_i[2]), S(T[3] \oplus F_i[3]))$$
$$T = MDS(S(T[0] \oplus G_i[0]), S(T[1] \oplus G_i[1]), S(T[2] \oplus G_i[2]), S(T[3] \oplus G_i[3]))$$

T의 값을 결정했다면, W_i의 값 또한 계산할 수 있습니다. MDS는 inverse가 존재하는 행렬 곱이며, S-BOX또한 출력 값을 통해 입력 값을 찾을 수 있고, A_i~G_i는 X_{i+1}을 통해 얻을 수 있기 때문에 역연산이 가능하기 때문입니다.

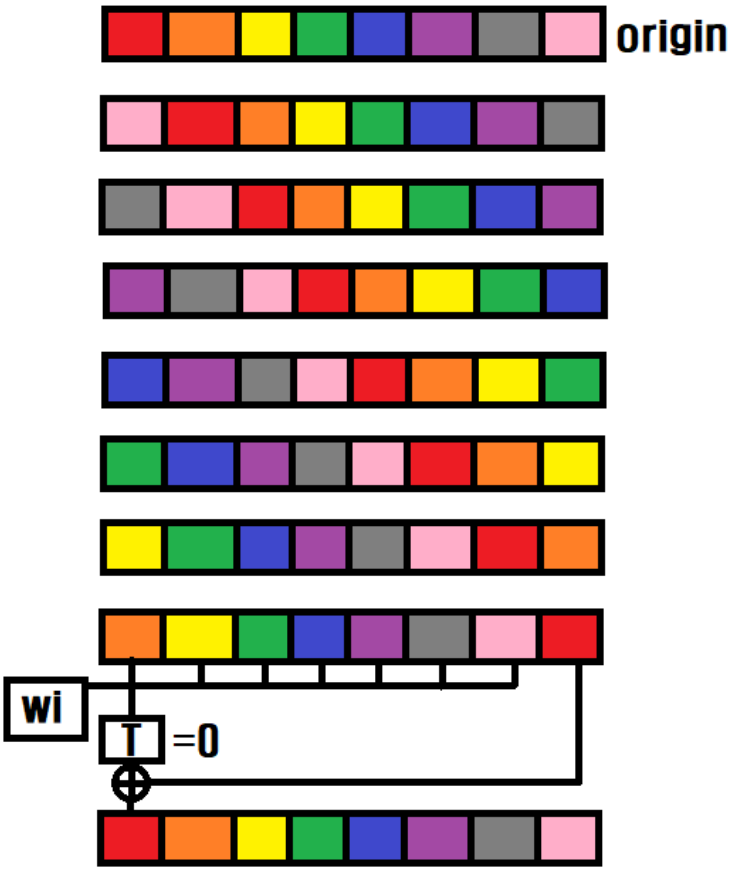
Round function F의 key는 $W_i(T)$ 또는 H_i이며 둘 다 32-bit이므로 F의 strength는 preimage attack의 경우에 2^{32}이며, 무작위로 시도할 경우 2^{32}회 시도해야 한 라운드를 통과할 수 있습니다.



또한 round함수의 다른 특징은, 이전 단계의 data들이 별다른 암호화(one-way)없이 shift된 채로 다음 단계로 넘어간다는 것입니다.

이는 8 이하의 round R에서는 R개의 임의의 32 bit block을 X_R의 일부로서 가정하면 이를 통해 input 전체를 얻을 수 있으며, 동시에 X_{R-1} 단계의 H_{R-1} block을 input으로부터 이끌어 낼 수 있다는 뜻입니다. 따라서 2^{32}번의 무작위 해시 시행 없이도 각 round들의 W_i를 즉시 결정할 수 있습니다.

즉 8 이하의 round에서는 어떤 임의의 X_R(R*32 bit)을 가정하더라도 output과 collision을 발생시키는 input과 $W_{0\sim7}$을 무작위 대입 없이 구할 수 있습니다.



그리고 각 round에서 X_{i+1}이 주어졌을 때, T=0인 경우를 만들 수 있습니다. 이 경우, T=0을 만드는 W_i는 쉽게 구할 수 있으며, $H_{i+1}=H_i$ xor T에서 T가 0이므로 $H_{i+1} = H_i$가 됩니다. 즉 round을 진행하는 것은, 256 bit X_i를 32-bit left rotation 해 X_{i+1}을 만드는 연산이 됩니다.

이러한 round를 8번 반복할 경우, 32-bit rotation이 8회 일어나므로 X_i은 X_{i+8}과 같은 값을 지니게 됩니다.

이 특성과 앞장의 8 round 이하에선 반드시 collision을 찾을 수 있다는 점을 결합하면, 9~16 round에 대해서도 input X를 구할 수 있습니다.

예를 들어 16 round의 경우를 생각하면, 어떠한 input X에 대해 8 round에서 collision을 발생시키는 W_i들을 구해 $W_{8\sim15}$로 두고, 나머지 8 round에선 T=0을 만드는 W_i들을 구해 $W_{0\sim7}$로 두면 어떤 input X을 가정해도 collision이 발생하도록 하는 $W_{0\sim15}$를 구할 수 있습니다.

16 round 이후에서는 W_i block을 임의로 설정할 수 없고, 앞의 W_i들에 의해 계산되어 결정되기 때문에, 첫 페이지에서 계산한대로 2^{32}의 secure를 가지고, 2^{32}회 해시를 실행해야 적합한 결과를 찾을 수 있습니다.

따라서 2^{253}번의 해시 실행 내로 X를 찾을 수 있는 최대 round는, round당 2^{32}번 해시를 실행해야 하므로 16 + 253/32 = 23.90... 23round입니다.

STEP 2. 코드 작성

메인함수 전에 별도로 작성, 사용한 함수입니다.

```
unsigned int inv_sbox[256] = { 0x52, 0x9, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0
unsigned int inv_MDS_cal(unsigned int S1, unsigned int S2, unsigned int S3, unsigned int S4) {
    //S1*14 + S2*11 + S3*13 + S4*9
    return S1 << 3 ^ S1 << 2 ^ S1 << 1 ^ S2 << 3 ^ S2 << 1 ^ S2 ^ S3 << 3 ^ S3 << 2 ^ S3  ^  S4 << 3 ^ S4;
}
unsigned int inv_MDS(unsigned int S) {
    unsigned int S1 = (S & 0xFF000000) >> 24;
    unsigned int S2 = (S & 0xFF0000) >> 16;
    unsigned int S3 = (S & 0xFF00) >> 8;
    unsigned int S4 = (S & 0xFF);
    unsigned int temp_MDS = 0;
    temp_MDS |= inv_MDS_cal(S1, S2, S3, S4);
    temp_MDS = temp_MDS << 8;
    temp_MDS |= inv_MDS_cal(S2, S3, S4, S1);
    temp_MDS = temp_MDS << 8;
    temp_MDS |= inv_MDS_cal(S3, S4, S1, S2);
    temp_MDS = temp_MDS << 8;
    temp_MDS |= inv_MDS_cal(S4, S1, S2, S3);

    return temp_MDS;
}
```

Inverse_sbox는 계산된 값을 직접 256크기의 숫자 배열에 하드코드하였습니다. 32bit input S를 8bit 4개로 분리 후 inverse AES matrix multiplication 연산을 하는 코드도 작성했습니다.

```
unsigned int inv_SBOX_cal(unsigned int S) {
    return inv_sbox[(S & 0xFFFF0000) * 16 + (S & 0xFFFF)];
}

unsigned int reverse_SBOX(unsigned int S) {
    unsigned int S1 = (S & 0xFF000000) >> 24;
    unsigned int S2 = (S & 0xFF0000) >> 16;
    unsigned int S3 = (S & 0xFF00) >> 8;
    unsigned int S4 = (S & 0xFF);
    unsigned int temp_MDS = 0;

    temp_MDS |= inv_SBOX_cal(S1);
    temp_MDS = temp_MDS << 8;
    temp_MDS |= inv_SBOX_cal(S2);
    temp_MDS = temp_MDS << 8;
    temp_MDS |= inv_SBOX_cal(S3);
    temp_MDS = temp_MDS << 8;
    temp_MDS |= inv_SBOX_cal(S4);

    return temp_MDS;
}
```

위와 마찬가지로 SBox의 inverse또한 32bit input를 받아 8bit로 4개씩 나눈 후 4bit/4bit 각 bit를 통해 inv_sbox배열을 조회하여 반환하는 함수를 작성했습니다.

```
unsigned int circular_shift(unsigned int a, int shift) {
    unsigned int temp = a & 0x80000000;
    temp = temp >> 31;
    a = a << 1;
    a = a | temp;
    shift -= 1;
    if (shift == 0) {
        return a;
    }
    else return circular_shift(a, shift);
}
```

Right circular shift함수이며, 다른 w_i들로부터 다음, 혹은 이전의 w_i값을 계산하기 위해 작성했습니다.

$$W_i = (W_{i-3} <<< 1) \oplus (W_{i-8} <<< 6) \oplus (W_{i-14} <<< 11) \oplus W_{i-16}$$

다음 식으로부터, $W_{0\sim15}$로부터 $W_{16\sim31}$을 계산할 수 있지만, 반대로 $W_{16\sim31}$으로부터 $W_{0\sim15}$또한 계산할 수 있다는 것을 알 수 있습니다.

```
unsigned int output[8];
int R;  //round

//get output
for (int i = 0; i < 8; i++) {
    scanf("%u ", &output[i]);
}
scanf("%d ", &R);

//input(256)+wi(32*@) = X
unsigned int input[8] = { 0, };
unsigned int* wi = (unsigned int*)malloc(sizeof(unsigned int)*(R));
unsigned int Xi[8] = { 0, };
```

본문의 시작부분이며, output과 round는 외부 입력을 받도록 하였고, input, wi, Xi(중간 결과물)등의 변수를 선언했습니다.

```
int BF_R = R > 8 ? R - 8 : R;
if (R <= 16) {
    //setup init Xi and input
    //set random Xi

    srand(time(NULL) * 3 % 377);
    for (int i = 0; i < 8; i++) {
        Xi[i]=rand();
    }

    for (int i = 0; i < BF_R; i++) {
        input[i] = output[i] ^ Xi[i];
    }
    for (int i = BF_R; i < 8; i++) {
        Xi[i] = input[i - BF_R];
        input[i] = output[i] ^ Xi[i];
    }

    //first print
    printf("Each 32bit block is devided by blank\n");
    printf("Round %2d / X : ", R);
    for (int i = 0; i < 8; i++) {
        printf("%x ", Xi[i]);
    }printf("\n");
```

Round가 16이하인 경우와 이상인 경우를 분리하였습니다. 위 코드는 Xi를 설정하고 이를 통해 input값을 알아내며, 이를 첫 라운드로서 출력하는 코드입니다. Xi는 무작위 값을 대입하도록 했으나, 테스트에선 편의성을 위해 임시로 주석처리 하고 초기값인 0을 사용했습니다.

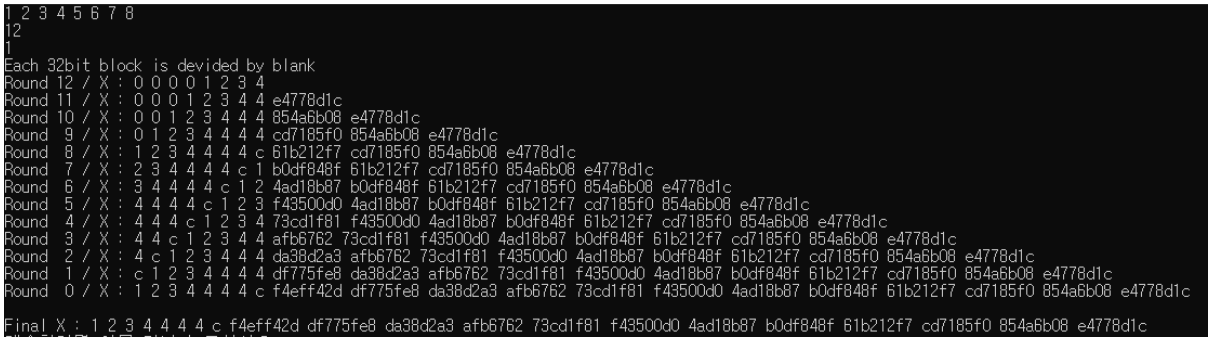
```
//common part of round 1~16
for (int i = 0; i < BF_R; i++) {
    unsigned int T = Xi[0] ^ input[(8-BF_R)+i];

    for (int j = 7; j >= 1; j--) {
        T = inv_MDS(T);
        T = reverse_SBOX(T);
        T = T ^ Xi[j]; // Gi~Ai
    }
    wi[BF_R-i-1] = T;


    for (int j = 0; j < 7; j++) {
        Xi[j] = Xi[j + 1];
    } Xi[7] = input[(8 - BF_R) + i];

    printf("Round %2d / X : ", R-1-i);
    for (int j = 0; j < 8; j++) {
        printf("%x ", Xi[j]);
    }
    for (int j = i; j >= 0; j--) {
        printf("%x ", wi[BF_R - j - 1]);
    }printf("\n");

}
```

8라운드 이하인 경우, input값의 특정 부분으로부터 현재 round의 H_i값을 획득하고 이를 바탕으로 T값을 확정하며, inverse MDS, inverse S-BOX, xor연산을 반복 시행하며 W_i 값을 구해가는 코드입니다. 9~16라운드 또한 이 과정은 동일하게 진행한 후 추후에 T=0이 나오도록 하는 8개의 round를 추가했습니다.

```
//if round is 9~16
if (R > 8) {
    //shift wi 8 block
    for (int i = 0; i < BF_R; i++) {
        wi[i + 8] = wi[i];
        wi[i] = 0;
    }
    for (int i = 7; i >=0; i--) {
        unsigned int T = 0;

        for (int j = 7; j >= 1; j--) {
            T = inv_MDS(T);
            T = reverse_SBOX(T);
            T = T ^ Xi[j]; // Gi~Ai
        }
        wi[i] = T;

        unsigned int temp = Xi[0];
        for (int j = 0; j < 7; j++) {
            Xi[j] = Xi[j + 1];
        } Xi[7] = temp;


        printf("Round %2d / X : ", i);
        for (int j = 0; j < 8; j++) {
            printf("%x ", Xi[j]);
        }
        for (int j = i; j <= 7 + BF_R; j++) {
            printf("%x ", wi[j]);
        }printf("\n");
    }

}
```

R이 9~16인 경우에 T=0으로 가정한 round를 8번 실행하는 부분입니다. T가 0이기 때문에 각 라운드 X_i는 한 block씩 left circular shift되는 연산만이 진행됩니다.

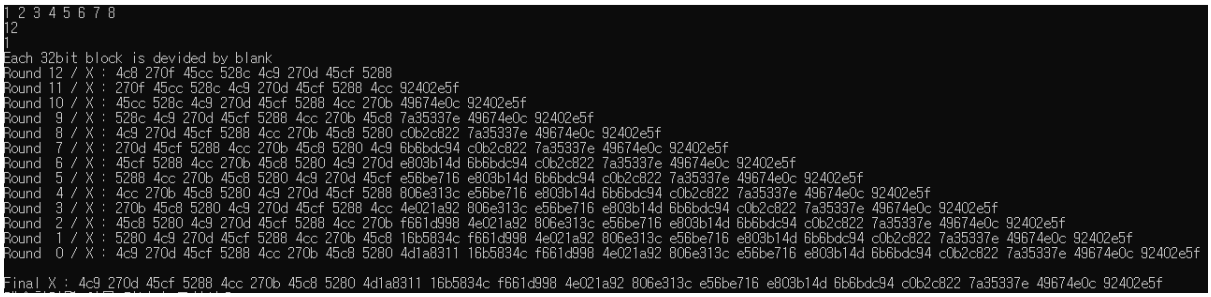
또한 각 라운드마다 현재 라운드의 X_i와 W_i를 출력하도록 했습니다.



[1 2 3 4 5 6 7 8]의 output과 12round를 input으로 테스트한 결과입니다.

12round의 경우 8+4로 나눠 실행하기 때문에 128bit의 Xi, A~D block을 초기값인 0으로 설정한 후, output과 xor연산하여 얻은 1,2,3,4를 Xi의 E~H block으로 설정하였으며, 이를 바탕으로 Round 8에선 [1 2 3 4 4 4 4 c]라는 결과를 얻었습니다.

초기 Xi인 [0 0 0 0 1 2 3 4]와 [1 2 3 4 4 4 4 c]를 xor연산하면 입력한 output인 [1 2 3 4 5 6 7 8]이 정상적으로 나오는 것을 알 수 있으며, round 8~0의 과정은 단순히 32bit씩 circular shift되며 T를 0으로 만드는 W_i를 구하는 것입니다.



X_i의 초기값을 랜덤하게 설정하면 랜덤 X_i에 해당하는 결과를 얻을 수 있습니다.

```
else {//R>16
    BF_R = R - 16;//round to brute-force
    unsigned int* T_temp = (unsigned int*)malloc(sizeof(unsigned int)*(BF_R));
    for (int i = 0; i < BF_R; i++) {
        T_temp[i] = 0;
    }
    while (T_temp[0] != 0xFFFFFFFF) {
        //create every T case for each round - brute-force
        for (int i = 1; i < BF_R; i--) {
            if (T_temp[BF_R-i] == 0xFFFFFFFF) {
                T_temp[BF_R - i] = 0;
                T_temp[BF_R - i - 1]++;
            }
        } T_temp[BF_R - 1]++;

        for (int i = 0; i < 8; i++) {
            Xi[i] = rand();
        }

        for (int i = 0; i < 8; i++) {
            input[i] = output[i] ^ Xi[i];
        }
```

Round가 16보다 큰 경우에 대해선, 16이후 round에서 사용할 $W_{i,}$ 정확히는 각round에서 사용할 32bit T(T_temp)에 대해 모든 경우의 수를 만들어가며 체크하도록 했으며, 이후 16-0 round를 진행하며 생성된 전체 W_i들과 생성규칙에 의해 생성된 W_i들을 비교하여 검증하도록 했습니다.

$$W_i = (W_{i-3} <<< 1) \oplus (W_{i-8} <<< 6) \oplus (W_{i-14} <<< 11) \oplus W_{i-16}$$

예를 들어 Round 18 output이 주어졌을 경우, Round 17,16에서 사용할 T는 각 32bit이므로 총 2^{64}만큼의 경우의 수가 있고, 각 T를 통해 서로 다른 $W_{17,16}$, $H_{17,16}$을 얻을 수 있습니다. 이후의 Round 16-0은 이전과 동일하게 진행, $W_{0\sim15}$를 생성할 수 있습니다. 또한 $W_{16,13,8,2}$로부터 W_0'을, $W_{17,14,9,3}$으로부턴 W_1'을 계산할 수 있습니다. 합당한 input값을 만들어내는 $W_{0,1}$와 W_i생성규칙을 만족하는 $W_{0,1}'$를 비교해 각 값이 전부 같다면, 모든 조건을 만족하며 collision을 발생하는 X(input + $W_{0\sim15}$)를 찾았다고 할 수 있는 것입니다.

```
//round after 16
for (int i = BF_R-1; i >= 0; i--) {
    unsigned int Hi = Xi[0] ^ T_temp[i]; // Hi+1 xor T

    unsigned int T = T_temp[i];
    for (int j = 7; j >= 1; j--) {
        T = inv_MDS(T_temp[i]);
        T = reverse_SBOX(T);
        T = T ^ Xi[j]; // Gi~Ai
    }wi[i + 16] = T;


    for (int j = 0; j < 7; j++) {
        Xi[j] = Xi[j + 1];
    } Xi[7] = Hi;


    printf("Round %2d / X : ", R+i);
    for (int j = 0; j < 8; j++) {
        printf("%x ", Xi[j]);
    }
    for (int j = BF_R-1; j >= i; j--) {
        printf("%x ", wi[j+16]);
    }printf("\n");
}
```

먼저 16round 이후의 round들에 대해선, 위 코드에서 생성한 각 round의 T를 적용해 해당하는 H_i block과 W_i block을 생성, 이전 round의 X_i를 구하였습니다.

```
//round 16->8
for (int i = 0; i < 8; i++) {
    unsigned int T = Xi[0] ^ input[i]; // Hi+1 xor Hi

    for (int j = 7; j >= 1; j--) {
        T = inv_MDS(T);
        T = reverse_SBOX(T);
        T = T ^ Xi[j]; // Gi~Ai
    }
    wi[15-i] = T;


    for (int j = 0; j < 7; j++) {
        Xi[j] = Xi[j + 1];
    } Xi[7] = input[(8 - BF_R) + i];

    printf("Round %2d / X : ", R - 1 - i);
    for (int j = 0; j < 8; j++) {
        printf("%x ", Xi[j]);
    }
    for (int j = 0; j <= i; j++) {
        printf("%x ", wi[15 - j]);
    }printf("\n");

}
```

```
//Round 8->0
for (int i = 7; i >= 0; i--) {
    unsigned int T = 0;

    for (int j = 7; j >= 1; j--) {
        T = inv_MDS(T);
        T = reverse_SBOX(T);
        T = T ^ Xi[j]; // Gi~Ai
    }
    wi[i] = T;

    unsigned int temp = Xi[0];
    for (int j = 0; j < 7; j++) {
        Xi[j] = Xi[j + 1];
    } Xi[7] = temp;


    printf("Round %2d / X : ", i);
    for (int j = 0; j < 8; j++) {
        printf("%x ", Xi[j]);
    }
    for (int j = i; j <= 16 + BF_R; j++) {
        printf("%x ", wi[j]);
    }printf("\n");
}
```

그 후 16round부터 0round까지는 동일하게 진행하였으며, 이를 통해 $W_{0\sim15}$ block들을 계산하였습니다.

```
//compute & compare Wi
unsigned int* wi_com = (unsigned int*)malloc(sizeof(unsigned int)*(BF_R));
for (int i = 0; i < BF_R; i++) {
    wi_com[i] = wi[i + 16] ^ circular_shift(wi[i + 13],1) ^ circular_shift(wi[i + 8], 6) ^
        circular_shift(wi[i + 2], 11);
}
bool check = true;
for (int i = 0; i < BF_R; i++) {
    //if wi computed by other wi == wi used to make input data, then input & wi is valid
    if (wi_com[i] != wi[i]) {
        check = false; break;
    }
}

//if valid, print & end
if (check == true) {
    printf("\nFinal X : ");
    for (int j = 0; j < 8; j++) {
        printf("%x ", Xi[j]);
    }
    for (int j = 0; j < 16; j++) {
        printf("%x ", wi[j]);
    }printf("\n");

    break;
}
```

그 후 round수에 따른 W′ block들을 계산하였고, 계산된 W′들과 위의 W들을 비교하여 각 값들이 전부 일치하면 적합한 X로 결정하고 출력하도록 하였습니다.

16round이상에서는 코드의 복잡도가 $2^{32*(R-16)}$으로 크게 증가하였고, round마다 값을 출력하는 overhead또한 크기 때문에 직접 실행해서 값을 확인해 볼 수는 없었습니다.

참고자료

Inverse S-Box

https://en.wikipedia.org/wiki/Rijndael_S-box

Inverse Matrix multiplication

https://en.wikipedia.org/wiki/Rijndael_MixColumns

One-way compression function

https://en.wikipedia.org/wiki/One-way_compression_function