Symmetric Encryption and Message Confidentiality 실습

[문제 **1]** 각 블록 모드 작업을 사용하여 **DES** 암호 알고리즘을 작성하시오**.**

소스 코드

IS_HW3_201824439_김성현.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "DES.h"
#define BLOCK_MODE 4          /* 1: CBC, 2: CFB, 3: OFB, 4: CTR */
#define NONCE 661F98CD37A38B4B

// CBC
void DES_CBC_Enc(BYTE*, BYTE*, BYTE*, BYTE*, int);
void DES_CBC_Dec(BYTE*, BYTE*, BYTE*, BYTE*, int);
// CFB
void DES_CFB_Enc(BYTE*, BYTE*, BYTE*, BYTE*, int);
void DES_CFB_Dec(BYTE*, BYTE*, BYTE*, BYTE*, int);
// OFB
void DES_OFB_Enc(BYTE*, BYTE*, BYTE*, BYTE*, int);
void DES_OFB_Dec(BYTE*, BYTE*, BYTE*, BYTE*, int);
//CTR
void DES_CTR_Enc(BYTE*, BYTE*, BYTE*, UINT64, int);
void DES_CTR_Dec(BYTE*, BYTE*, BYTE*, UINT64, int);

int main()
{
    int i;
    BYTE p_text[128]={0,};
    BYTE key[9]={0,};
    BYTE IV[9]={0,};
    BYTE c_text[128]={0,};
    BYTE d_text[128]={0,};
    int msg_len;
    UINT64 ctr=0;

    /* 평문 입력 */
    printf("평문 입력: ");
        gets((char *)p_text);
    /* 비밀키 입력 */
    printf("비밀키 입력: ");
        scanf("%s", key);
    fflush(stdin);

    #if(BLOCK_MODE!=4)
    /* 초기화 벡터 입력 */
    printf("초기화 벡터 입력: ");
        scanf("%s", IV);
    #else
```

```
    /* 카운터 입력 */
    printf("ctr 입력: ");
        scanf("%u", &ctr);
    #endif

/* 메시지 길이 계산 */
    msg_len=(strlen((char *)p_text) % BLOCK_SIZE) ?
            ((strlen((char *)p_text) / BLOCK_SIZE +1)*8):
            strlen((char *)p_text);
    #if(BLOCK_MODE==1)
    DES_CBC_Enc(p_text, c_text, IV, key, msg_len);//DES-CBC 암호화
    #elif(BLOCK_MODE==2)
    DES_CFB_Enc(p_text, c_text, IV, key, msg_len);//DES-CFB 암호화
    #elif(BLOCK_MODE==3)
    DES_OFB_Enc(p_text, c_text, IV, key, msg_len);//DES-OFB 암호화
    #else
    DES_CTR_Enc(p_text, c_text, key, ctr, msg_len);//DES-CTR 암호화
    #endif

    /* 암호문 출력 */
    printf("\n암호문: ");
    for(i=0; i<msg_len; i++)
       printf("%c", c_text[i]);
    printf("\n");



    #if(BLOCK_MODE==1)
    DES_CBC_Dec(c_text, d_text, IV, key, msg_len);//DES-CBC 복호화
    #elif(BLOCK_MODE==2)
    DES_CFB_Dec(c_text, d_text, IV, key, msg_len);//DES-CFB 복호화
    #elif(BLOCK_MODE==3)
    DES_OFB_Dec(c_text, d_text, IV, key, msg_len);//DES-CFB 복호화
    #else
    DES_CTR_Dec(c_text, d_text, key, ctr, msg_len);//DES-CTR 복호화
    #endif

    /* 복호문 출력 */
    printf("\n복호문: ");
    for(i=0; i<msg_len; i++)
       printf("%c", d_text[i]);
    printf("\n");

    return 0;
}

// CBC
void DES_CBC_Enc(BYTE* p_text, BYTE* c_text, BYTE* IV, BYTE* key, int msg_len) {
        int i, j;
        BYTE* chain = IV;
        BYTE input_text[128] = {0,};

        for(i=0; i<msg_len/BLOCK_SIZE; i++) {
                for(j=0; j<BLOCK_SIZE; j++) {
                        input_text[i*BLOCK_SIZE+j] = p_text[i*BLOCK_SIZE+j] ^ chain[j];
                }
```

```
		DES_Encryption(input_text+(i*BLOCK_SIZE), c_text+(i*BLOCK_SIZE), key);
		chain = c_text+(i*BLOCK_SIZE);
	}
}

void DES_CBC_Dec(BYTE* c_text, BYTE* d_text, BYTE* IV, BYTE* key, int msg_len) {
	int i, j;
	BYTE* chain = IV;

	for(i=0; i<msg_len/BLOCK_SIZE; i++) {
		DES_Decryption(c_text+(i*BLOCK_SIZE), d_text+(i*BLOCK_SIZE), key);

		for(j=0; j<BLOCK_SIZE; j++) {
			d_text[i*BLOCK_SIZE+j] = d_text[i*BLOCK_SIZE+j] ^ chain[j];
		}

		chain = c_text+(i*BLOCK_SIZE);
	}
}

// CFB
void DES_CFB_Enc(BYTE* p_text, BYTE* c_text, BYTE* IV, BYTE* key, int msg_len) {
	int i, j;
	BYTE* chain = IV;

	for(i=0; i<msg_len/BLOCK_SIZE; i++) {
		DES_Encryption(chain, c_text+(i*BLOCK_SIZE), key);

		for(j=0; j<BLOCK_SIZE; j++) {
			c_text[i*BLOCK_SIZE+j]	=	c_text[i*BLOCK_SIZE+j]	^
p_text[i*BLOCK_SIZE+j];
		}

		chain = c_text+(i*BLOCK_SIZE);
	}
}

void DES_CFB_Dec(BYTE* c_text, BYTE* d_text, BYTE* IV, BYTE* key, int msg_len) {
	int i, j;
	BYTE* chain = IV;

	for(i=0; i<msg_len/BLOCK_SIZE; i++) {
		DES_Encryption(chain, d_text+(i*BLOCK_SIZE), key);

		for(j=0; j<BLOCK_SIZE; j++) {
			d_text[i*BLOCK_SIZE+j]	=	d_text[i*BLOCK_SIZE+j]	^
c_text[i*BLOCK_SIZE+j];
		}

		chain = c_text+(i*BLOCK_SIZE);
	}
}

// OFB
void DES_OFB_Enc(BYTE* p_text, BYTE* c_text, BYTE* IV, BYTE* key, int msg_len) {
```

```
	int i, j;
	BYTE chain[9] = {0,};

	for(j=0; j<9; j++) chain[i] = IV[i];

	for(i=0; i<msg_len/BLOCK_SIZE; i++) {
		DES_Encryption(chain, c_text+(i*BLOCK_SIZE), key);

		for(j=0; j<9; j++) chain[i] = c_text[i*BLOCK_SIZE+j];

		for(j=0; j<BLOCK_SIZE; j++) {
			c_text[i*BLOCK_SIZE+j] = c_text[i*BLOCK_SIZE+j] ^
p_text[i*BLOCK_SIZE+j];
		}
	}
}

void DES_OFB_Dec(BYTE* c_text, BYTE* d_text, BYTE* IV, BYTE* key, int msg_len) {
	int i, j;
	BYTE chain[9] = {0,};

	for(j=0; j<9; j++) chain[i] = IV[i];

	for(i=0; i<msg_len/BLOCK_SIZE; i++) {
		DES_Encryption(chain, d_text+(i*BLOCK_SIZE), key);

		for(j=0; j<9; j++) chain[i] = d_text[i*BLOCK_SIZE+j];

		for(j=0; j<BLOCK_SIZE; j++) {
			d_text[i*BLOCK_SIZE+j] = d_text[i*BLOCK_SIZE+j] ^
c_text[i*BLOCK_SIZE+j];
		}
	}
}

//CTR
void DES_CTR_Enc(BYTE* p_text, BYTE* c_text, BYTE* key, UINT64 ctr, int msg_len) {
	int i, j;
	BYTE chain[8] = {0,};

	for(i=0; i<msg_len/BLOCK_SIZE; i++) {
		for(j=7; j>=0 && ctr>0; j--) {
			chain[j] = ctr%256;
			ctr/=256;
		}

		DES_Encryption(chain, c_text+(i*BLOCK_SIZE), key);

		for(j=0; j<BLOCK_SIZE; j++) {
			c_text[i*BLOCK_SIZE+j] = c_text[i*BLOCK_SIZE+j] ^
p_text[i*BLOCK_SIZE+j];
		}

		ctr++;
	}
}
```

```
void DES_CTR_Dec(BYTE* c_text, BYTE* d_text, BYTE* key, UINT64 ctr, int msg_len) {
	int i, j;
	BYTE chain[8] = {0,};

	for(i=0; i<msg_len/BLOCK_SIZE; i++) {
		for(j=7; j>=0 && ctr>0; j--) {
			chain[j] = ctr%256;
			ctr/=256;
		}

		DES_Encryption(chain, d_text+(i*BLOCK_SIZE), key);

		for(j=0; j<BLOCK_SIZE; j++) {
			d_text[i*BLOCK_SIZE+j] = d_text[i*BLOCK_SIZE+j] ^
c_text[i*BLOCK_SIZE+j];
		}

		ctr++;
	}
}
```

DES.h

```
#pragma once

#include <iostream>
#include <bitset>

using namespace std;

#define BLOCK_SIZE 8
#define DES_ROUND 16

typedef unsigned char BYTE;
typedef unsigned int UINT;
typedef unsigned long long UINT64;

int ip[64] = { 58, 50, 42, 34, 26, 18, 10, 2,
		60, 52, 44, 36, 28, 20, 12, 4,
		62, 54, 46, 38, 30, 22, 14, 6,
		64, 56, 48, 40, 32, 24, 16, 8,
		57, 49, 41, 33, 25, 17, 9, 1,
		59, 51, 43, 35, 27, 19, 11, 3,
```

```
            61, 53, 45, 37, 29, 21, 13, 5,
            63, 55, 47, 39, 31, 23, 15, 7 };
int iip[64] = { 40, 8, 48, 16, 56, 24, 64, 32,
             39, 7, 47, 15, 55, 23, 63, 31,
             38, 6, 46, 14, 54 ,22, 62, 30,
             37, 5, 45, 13, 53, 21, 61, 29,
             36, 4, 44, 12, 52, 20, 60, 28,
             35, 3, 43, 11, 51, 19, 59, 27,
             34, 2, 42, 10, 50, 18, 58, 26,
             33, 1, 41, 9, 49, 17, 57, 25 };
int E[48] = { 32, 1, 2, 3, 4, 5, 4, 5,
                                  6, 7, 8, 9, 8, 9, 10, 11,
            12, 13, 12, 13, 14, 15, 16, 17,
            16, 17, 18, 19, 20, 21, 20, 21,
            22, 23, 24, 25, 24, 25, 26, 27,
            28, 29, 28, 29, 30, 31, 32, 1 };
/*
int E[48] = { 32, 6, 12, 16, 22, 28, 1, 7,
                                  13, 17, 23, 29, 2, 8, 12, 18,
            24, 28, 3, 9, 13, 19, 25, 29,
            4, 8, 14, 20, 24, 30, 5, 9,
            15, 21, 25, 31, 4, 10, 16, 20,
            26, 32, 5, 11, 17, 21, 27, 1 };
*/
int s_box[8][4][16] = {
                    {
                              { 14,  4, 13,  1,  2, 15, 11,  8, 3, 10,  6, 12,  5,  9,  0,  7 },
                              { 0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8 },
                              { 4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0 },
                              { 15, 12,  8,  2,  4,  9,  1,  7, 5, 11,  3, 14, 10,  0,  6, 13 },
                    },
             {
                       { 15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10 },
                              { 3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5 },
                              { 0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15 },
                              { 13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9 },
             },
             {
```

```
                { 10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8 },
                        { 13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,  1 },
                        { 13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,  7 },
                        { 1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2, 12 },
        },
        {
                { 7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15 },
                        { 13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,  9 },
                        { 10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,  4 },
                        { 3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2, 14 },
        },
        {
                { 2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9 },
                        { 14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,  6 },
                        { 4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0, 14 },
                        { 11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0,  9, 10,  4,  5,  3 },
        },
        {
                { 12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11 },
                        { 10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,  8 },
                        { 9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,  6 },
                        { 4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8, 13 },
        },
        {
                { 4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1 },
                        { 13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,  6 },
                        { 1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,  2 },
                        { 6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3, 12 },
        },
        {
                { 13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7 },
                        { 1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,  2 },
                        { 7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,  8 },
                        { 2,  1, 14,  7,  4, 10,  8, 13, 15, 12,  9,  0,  3,  5,  6, 11 },
        },
};
int P[32] = { 16, 7, 20, 21, 29, 12, 28, 17,
                        1, 15, 23, 26, 5, 18, 31, 10,
```

```
                                2, 8, 24, 14, 32, 27, 3, 9,
            19, 13, 30, 6, 22, 11, 4, 25 };
/*
int PC_1[56] = { 57, 49, 41, 33, 25, 17, 9,
                                        1, 58, 50, 42, 34, 26, 18,
                                        10, 2, 59, 51, 43, 35, 27,
              19, 11, 3, 60, 52, 44, 36,
              63, 55, 47, 39, 31, 23, 15,
              7, 62, 54, 46, 38, 30, 22,
              14, 6, 61, 53, 45, 37, 29,
              21, 13, 5, 28, 20, 12, 4 };
                                        */
int PC_1[56] = { 57, 1, 10, 19, 63, 7, 14, 21,
                                        49, 58, 2, 11, 55, 62, 6, 13,
                                        41, 50, 59, 3, 47, 54, 61, 5,
              33, 42, 51, 60, 39, 46, 53, 28,
              25, 34, 43, 52, 31, 38, 45, 20,
              17, 26, 35, 44, 23, 30, 37, 12,
              9, 18, 27, 36, 15, 22, 29 ,4};
int PC_2[48] = { 14, 17, 11, 24, 1, 5, 3, 28,
                                        15, 6, 21, 10, 23, 19, 12, 4,
                                        26, 8, 16, 7, 27, 20, 13, 2,
                                        41, 52, 31, 37, 47, 55, 30, 40,
                                        51, 45, 33, 48, 44, 49, 39, 56,
                                        34, 53, 46, 42, 50, 36, 29, 32 };

void PC2(UINT c, UINT d, BYTE *Key_Out);
UINT Cir_Shift(UINT n, int r);
void makeBit28(UINT *c, UINT *d, BYTE *Key_Out);
void PC1(BYTE *Key_In, BYTE *Key_Out);
void Key_Expansion(BYTE *key, BYTE round_key[16][6]);
void WtoB(UINT Left32, UINT Right32, BYTE *out);
void Swap(UINT *x, UINT *y);
UINT Permutation(UINT in);
UINT S_Box_Transfer(BYTE* in);
void EP(UINT Right32, BYTE* out);
UINT f(UINT Right32, BYTE* rKey);
void BtoW(BYTE *Plain64, UINT *Left32, UINT *Right32);
```

```
void IIP(BYTE *in, BYTE *out);
void IP(BYTE *in, BYTE *out);
void DES_Decryption(BYTE *c_text, BYTE *result, BYTE *key);
void DES_Encryption(BYTE *p_text, BYTE *result, BYTE *key);

void DES_Encryption(BYTE *p_text, BYTE *result, BYTE *key) {
	int i;
	BYTE data[BLOCK_SIZE] = { 0, };
	BYTE round_key[16][6] = { 0, };
	UINT L = 0, R = 0;

	/* 라운드 키 생성 */
	Key_Expansion(key, round_key);

	/* 초기 순열 */
	IP(p_text, data);

	/* 64bit 블록을 32bit로 나눔 */
	BtoW(data, &L, &R);

	/* DES Round 1~16 */
	for (i = 0; i<DES_ROUND; i++) {
		L=L^f(R, round_key[i]);

		/* 마지막 라운드는 Swap을 하지 않는다. */
		if (i != DES_ROUND - 1) {
			Swap(&L, &R);
		}
	}

	for (int i = 0; i < 8; i++)
		data[i] = 0;

	/* 32bit로 나누어진 블록을 다시 64bit 블록으로 변환 */
	WtoB(L, R, data);

	/* 역 초기 순열 */
	IIP(data, result);
```

```
}


void DES_Decryption(BYTE *c_text, BYTE *result, BYTE *key) {
	int i;
	BYTE data[BLOCK_SIZE] = { 0, };
	BYTE round_key[16][6] = { 0, };
	UINT L = 0, R = 0;

	/* 라운드 키 생성 */
	Key_Expansion(key, round_key);

	/* 초기 순열 */
	IP(c_text, data);

	/* 64bit 블록을 32bit로 나눔 */
	BtoW(data, &L, &R);

	/* DES Round 1~16 */
	for (i = 0; i<DES_ROUND; i++) {
		/* 암호화와 비교해서 라운드키를 역순으로 적용 */
		L = L ^ f(R, round_key[DES_ROUND - i - 1]);

		/* 마지막 라운드는 Swap을 하지 않는다. */
		if (i != DES_ROUND - 1) {
			Swap(&L, &R);
		}
	}

	for (int i = 0; i < 8; i++)
		data[i] = 0;

	/* 32bit로 나누어진 블록을 다시 64bit 블록으로 변환 */
	WtoB(L, R, data);

	/* 역 초기 순열 */
	IIP(data, result);
}
```

```
void IP(BYTE *in, BYTE *out) {
	int i;
	BYTE index, bit, mask=0x80;

	for(i=0; i<64; i++) {
		index=(ip[i]-1)/8;
		bit=(ip[i]-1)%8;

		if(in[index] & (mask>>bit)) {
			out[i/8]|=mask>>(i%8);
		}
	}
}

void IIP(BYTE *in, BYTE *out) {
	int i;
	BYTE index, bit, mask=0x80;

	for(i=0;i<64;i++) {
		index=(iip[i]-1)/8;
		bit=(iip[i]-1)%8;

		if(in[index] & (mask>>bit)) {
			out[i/8]|=mask>>(i%8);
		}
	}
}

void BtoW(BYTE *Plain64, UINT *Left32, UINT *Right32) {
	int i;

	for(i=0; i<8; i++) {
		if(i<4)
			*Left32|=(UINT)Plain64[i]<<(24-(i*8));
		else
			*Right32|=(UINT)Plain64[i]<<(56-(i*8));
```

```
        }
}

UINT f(UINT Right32, BYTE* rKey) {
        int i;
        BYTE data[6]={0,};      /* EP에 의한48 bit output 저장*/
        UINT out;

        EP(Right32, data);      /* 1. Expansion Permutation: EP-box */

        for(i=0; i<6; i++) {
                data[i]=data[i]^rKey[i];    /* 2. 48 bit XOR between data and rKey: S  -box */
        }
        /* 3 & 4. Straight permutation of 32-bit S-box output */
        out=Permutation(S_Box_Transfer(data));

        return out;
}

void EP(UINT Right32, BYTE* out) {
        int i;
        UINT bit8_Mask = 0x80, bit32_Mask = 0x80000000;
        for (i = 0; i<48; i++) {
                /* EP테이블이 나타내는 위치의 비트값을 & 연산과 시프트 연산을 이용하여 추출 */
                if (Right32 & (bit32_Mask >> (E[i] - 1))) {
                        /* 추출한 값을 배열의 상위 비트부터 저장 */
                        out[i / 8] |= (BYTE)(bit8_Mask >> (i % 8));
                }
        }
        return;
}

UINT S_Box_Transfer(BYTE* in) {
        int i, row, column, shift = 28;
        UINT temp = 0, result = 0, mask = 0x00000080;
        for (i = 0; i<48; i++) {
                /* 입력값의 상위 비트부터 1비트씩 차례로 추출하여 temp에 저장 */
                if (in[i / 8] & (BYTE)(mask >> (i % 8))) {
```

```
                        temp |= 0x20 >> (i % 6);
                } else
                        ;

                /* 추출한 비트가 6비트가 되면 */
                if ((i + 1) % 6 == 0) {
                        row = ((temp & 0x20) >> 4) + (temp & 0x01); /* 행의 값을 계산*/
                        column = (temp & 0x1E) >> 1; /* 열의 값을 계산*/
                                                                        /* 4비트의 결과
값을 result에 상위 비트부터 4비트씩 저장 */
                        result += ((UINT) s_box[i / 6][row][column] << shift);
                        shift -= 4;
                        temp = 0;
                }
        }
        return result;
}

UINT Permutation(UINT in) {
        int i;
        UINT out = 0, mask = 0x80000000;
        for (i = 0; i < 32; i++) {
                /* 순열 테이블이 나타내는 위치의 비트를 추출한 결과 값을 상위 비트부터 저장 */
                if (in&(mask >> (P[i] - 1))) {
                        out |= (mask >> i);
                } else
                        ; //do nothing
        }
        return out;
}

void Swap(UINT *x, UINT *y) {
        UINT temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

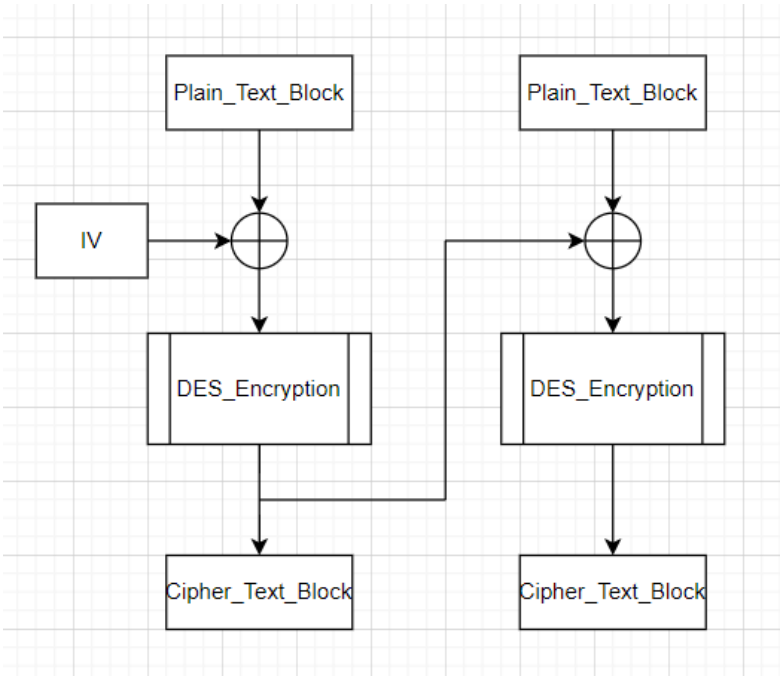
```
void WtoB(UINT Left32, UINT Right32, BYTE *out) {
	int i;
	UINT mask=0xff000000;

	for(i=0; i<8; i++) {
		if(i<4)
			out[i]|=(Left32 & (mask>>(i*8)))>>(24-(i*8));
		else
			out[i]|=(Right32 & (mask>>(i*8)))>>(56-(i*8));
	}
}

void Key_Expansion(BYTE *key, BYTE round_key[16][6]) {
	int i;
	BYTE pc1_result[7]={0,};
	UINT c=0, d=0;

	/* 키를  순열 선택1 테이블을이용해서재배치*/
	PC1(key, pc1_result);
	/* 56 비트의 데이터를28 비트로나누기*/
	makeBit28(&c, &d, pc1_result);

	/* 라운드키 생성*/
	for(i=0; i<16; i++) {
		c= Cir_Shift(c, i); /* 28비트 데이터를좌측으로순환 이동*/
		d=Cir_Shift(d, i);

		PC2(c, d, round_key[i]); /* 순열 선택2 테이블을이용해서재배치*/
	}
}

void PC1(BYTE *Key_In, BYTE *Key_Out) {
	int i, index, bit;
	UINT mask = 0x00000080;

	/* PC-1이 나타내는 위치를 계산하여 입력값으로부터 해당 위치의 비트를 추출하고 결과값을
저장할 배열에 상위 비트부터 저장 */
	for (i = 0; i < 56; i++) {
```

```
                index = (PC_1[i] - 1) / 8;
                bit = (PC_1[i] - 1) % 8;
                if (Key_In[index] & (BYTE)(mask >> bit)) {
                        // cout << "i / 8 :: " << i / 8 << "\t i % 8 :: " << i % 8 << endl;
                        Key_Out[i / 8] |= (BYTE)(mask >> (i % 8));
                }
                //bitset<8> x((int)Key_Out[i % 8]);
                //cout << "Key_Out[ " << i % 8 << " ] ::: " << Key_Out[i % 8] << " ||| " << x << endl;
        }
}

void makeBit28(UINT *c, UINT *d, BYTE *Key_Out) {
        int i;
        BYTE mask = 0x80;
        for (i = 0; i < 56; i++) {
                if (i < 28) {
                        if (Key_Out[i / 8] & (mask >> (i % 8))) {
                                *c |= 0x08000000 >> i;
                        } else
                                ; // do nothing
                } else {
                        if (Key_Out[i / 8] & (mask >> (i % 8))) {
                                *d |= 0x08000000 >> (i - 28);
                        } else
                                ; // do nothing
                }
        }
}

UINT Cir_Shift(UINT n, int r) {
        int n_shift[16] = { 1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1 };
        if (n_shift[r] == 1) {
                n = (n << 1) + (n >> 27); /* 28bit 유효 자릿수에 기반한 circulation shift */
        } else {
                n = (n << 2) + (n >> 26);
        }

        n &= 0x0FFFFFFF;
```

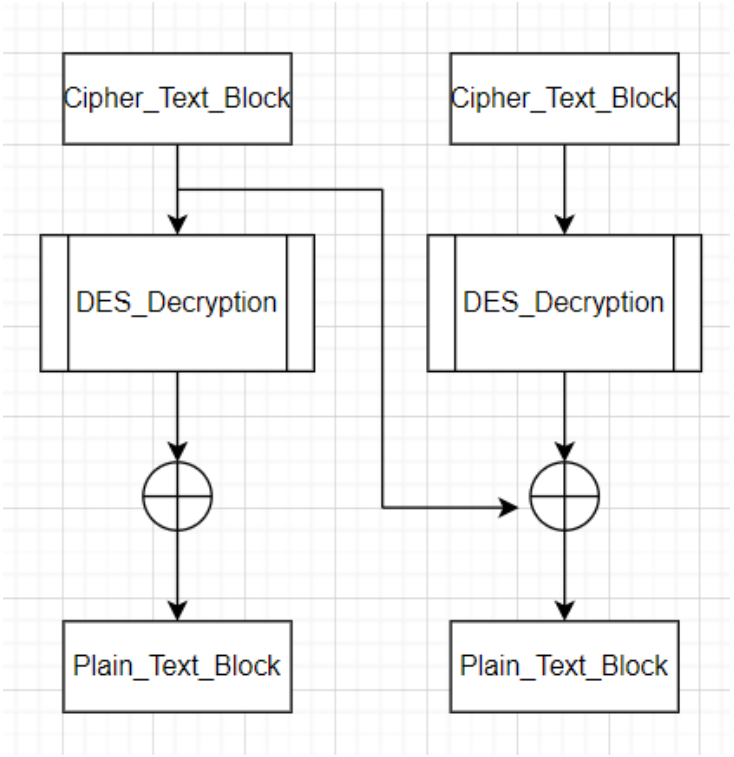
```
	return n;
}

void PC2(UINT c, UINT d, BYTE *Key_Out) {
	int i;
	UINT mask = 0x08000000;

	/* PC-2가 나타내는 위치를 계산하여 입력값으로부터 해당 위치의 비트를 추출하여 결과값을 저장할 배열에 상위 비트부터 저장 */
	for (i = 0; i<48; i++) {
		if (PC_2[i]<28) {
			if (c&(mask >> (PC_2[i] - 1))) {
				Key_Out[i / 8] |= 0x80 >> (i % 8);
			} else
				; // do nothing
		} else {
			if (d&(mask >> (PC_2[i] - 1 - 28))) {
				Key_Out[i / 8] |= 0x80 >> (i % 8);
			} else
				; // do nothing
		}
	}
}
```

동작 과정

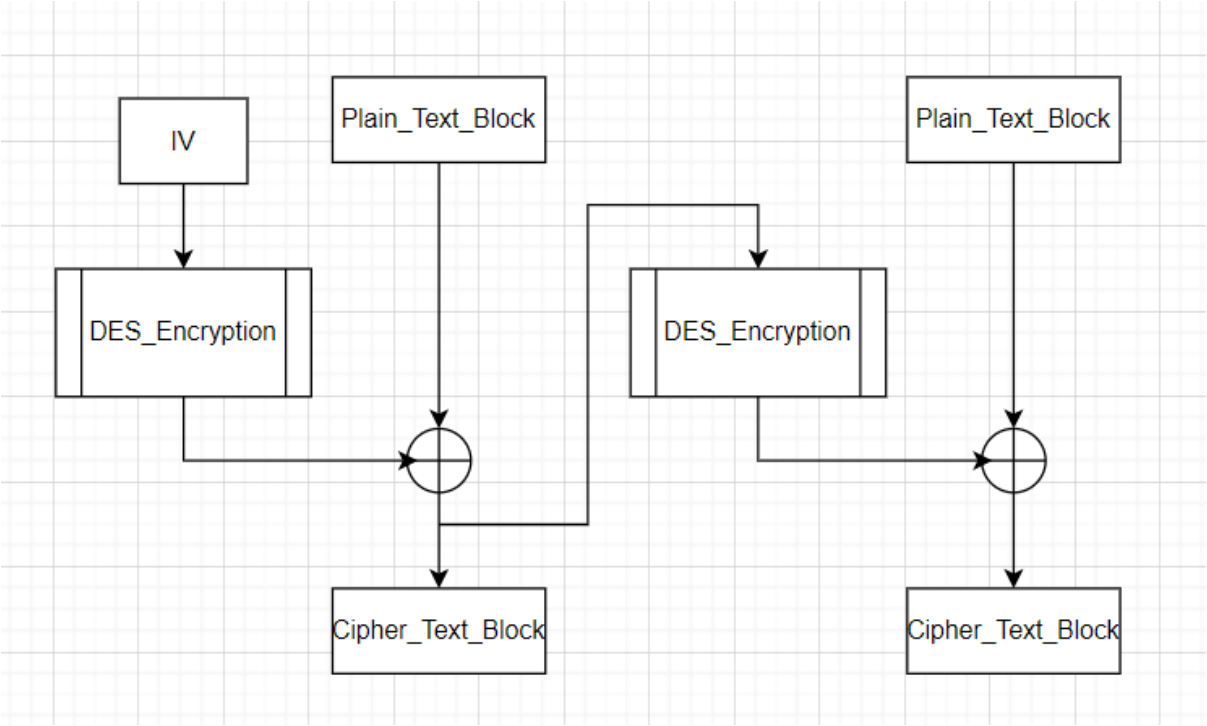
1. CBC 모드 암호화



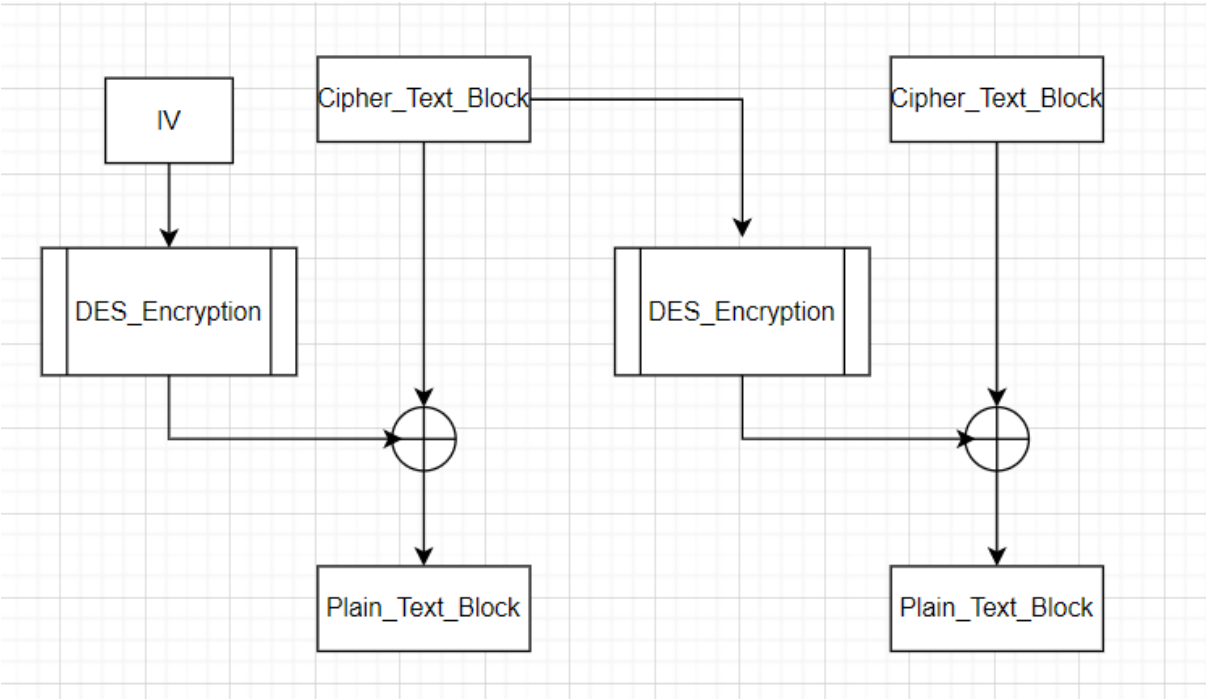
2. CBC 모드 복호화



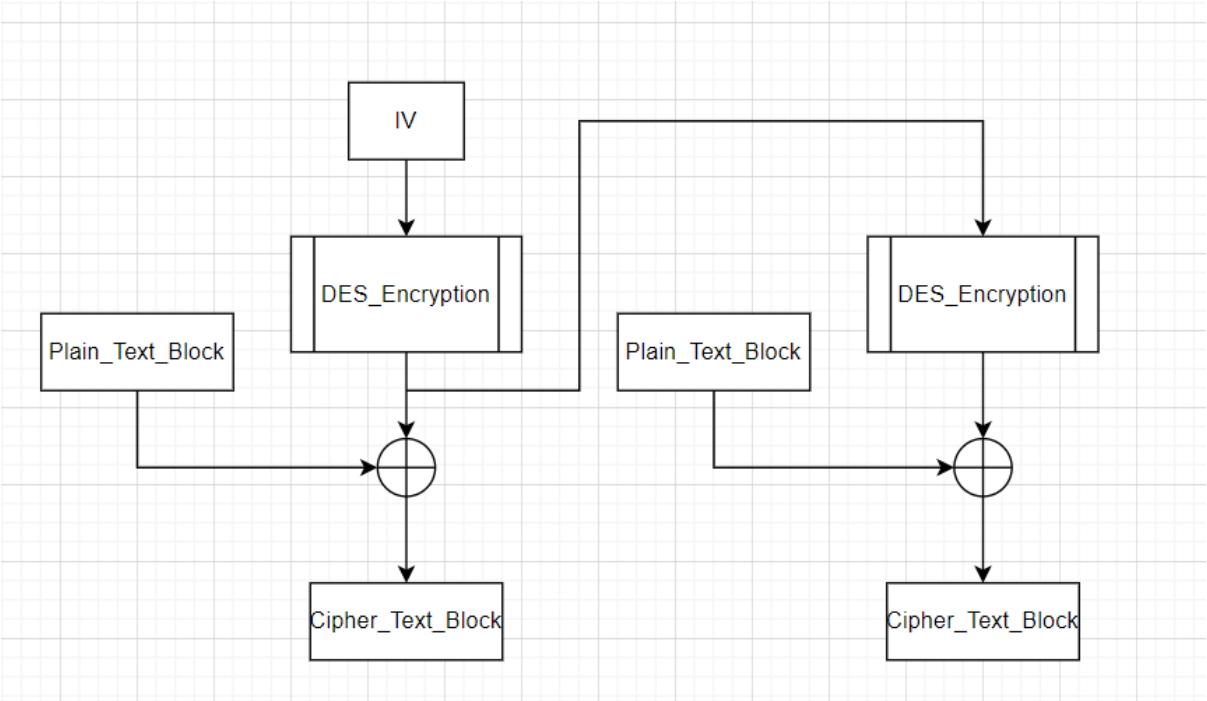
3. CFB 모드 암호화



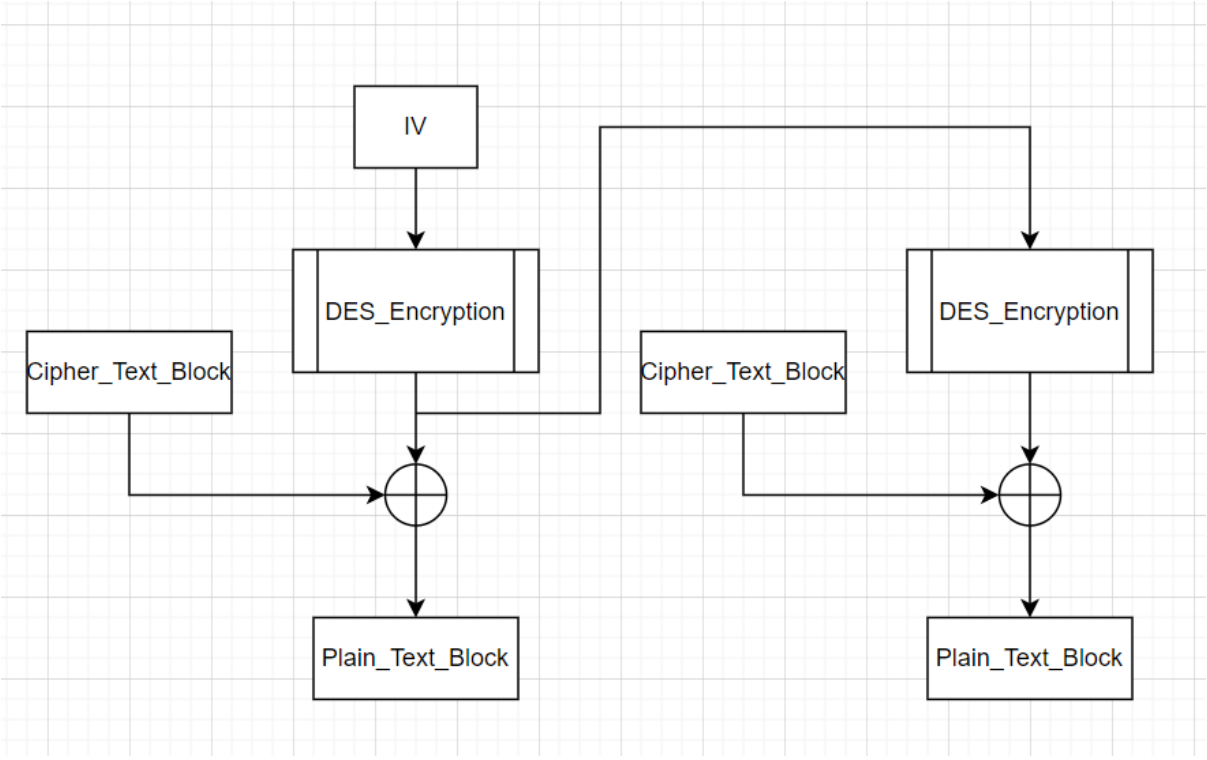
4. CFB 모드 복호화



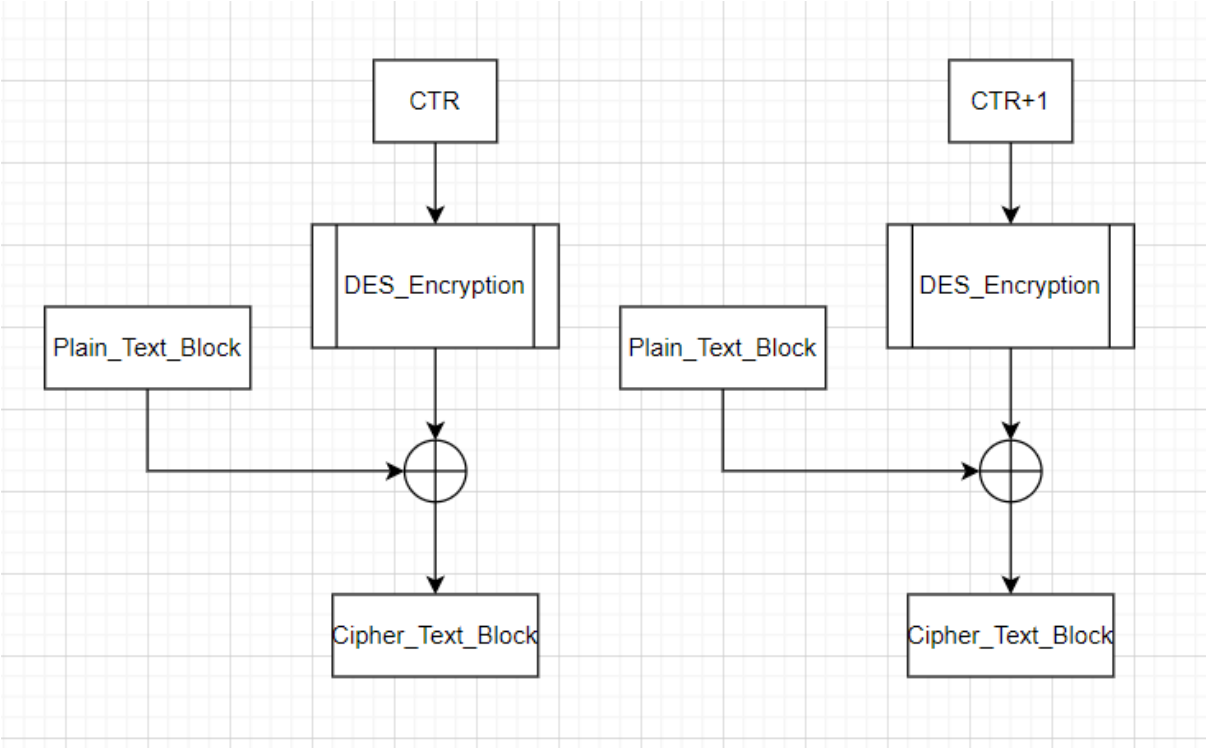
5. OFB 모드 암호화



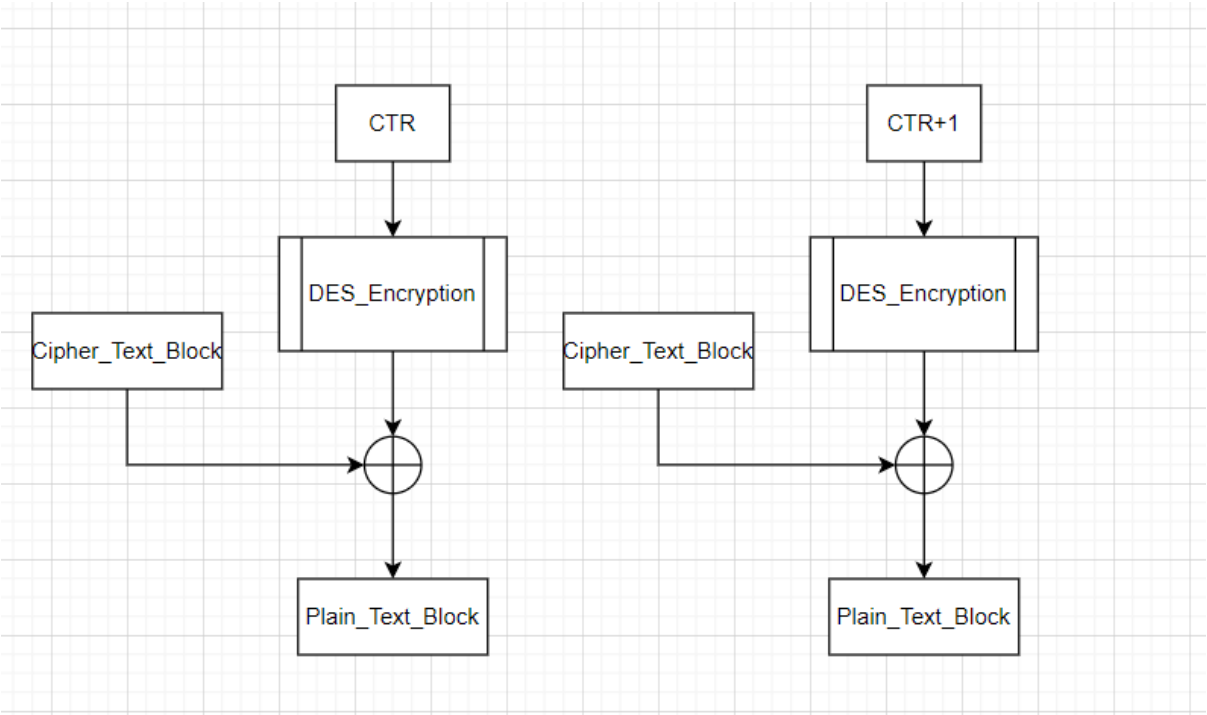
6. OFB 모드 복호화



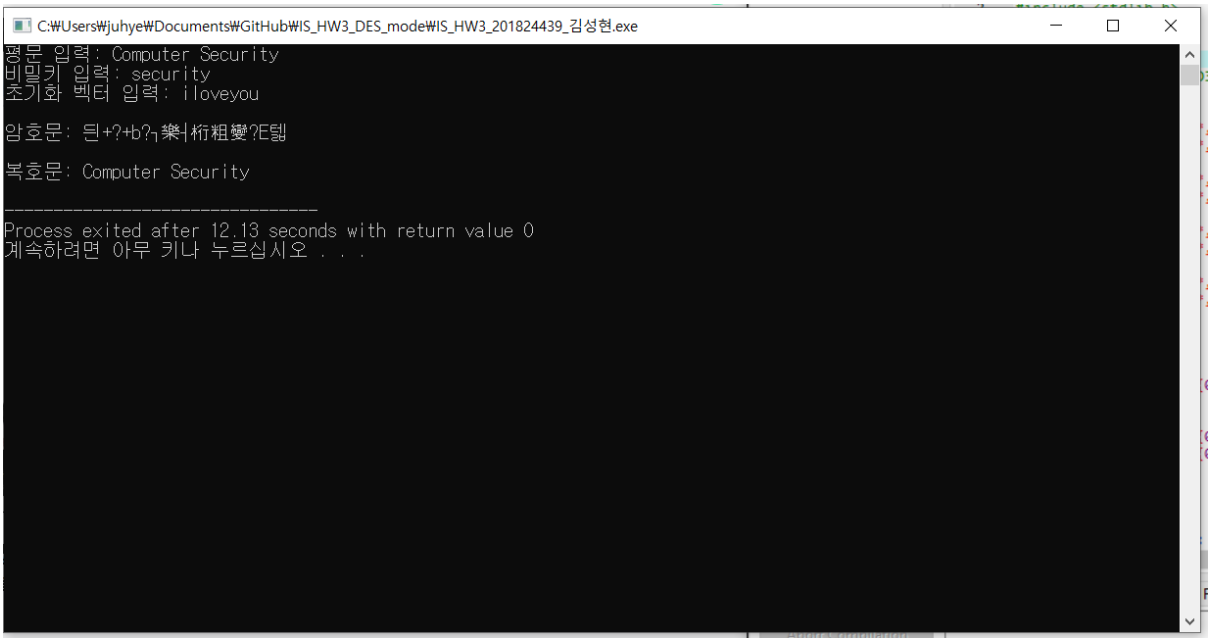
7. CTR 모드 암호화



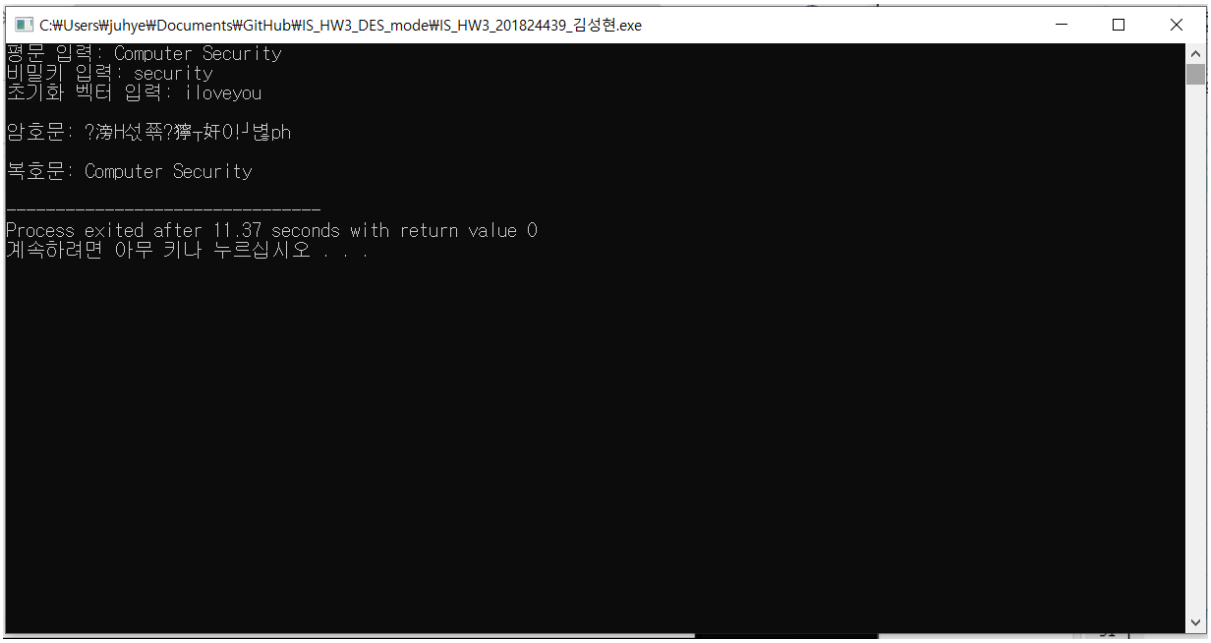
8. CTR 모드 복호화



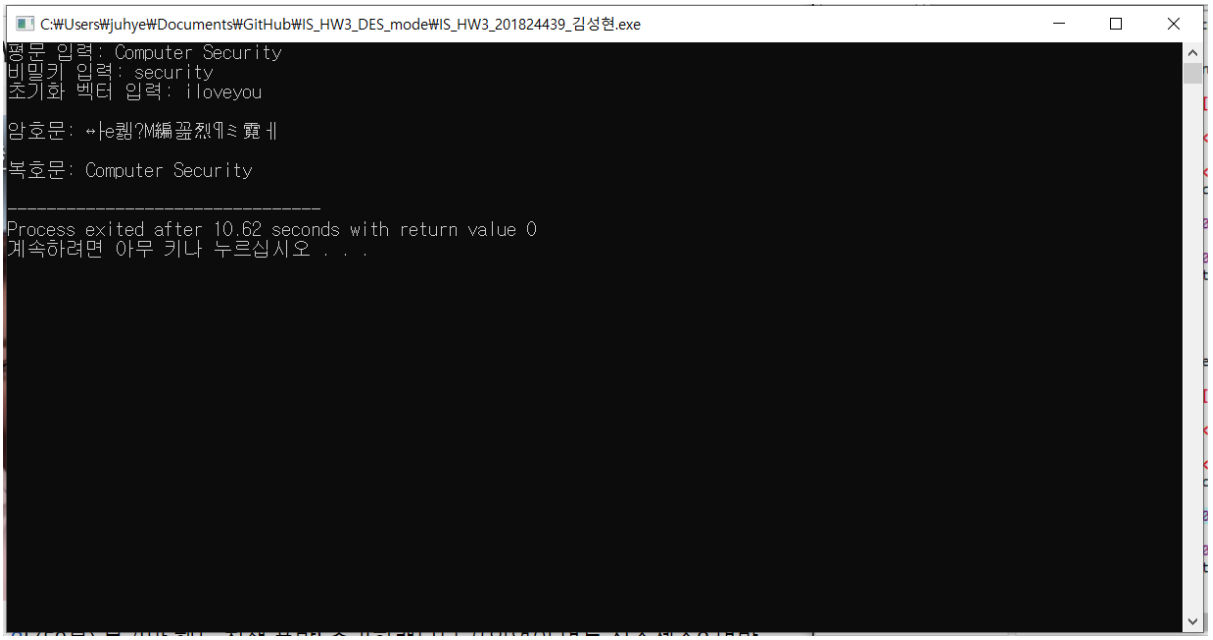
결과 화면



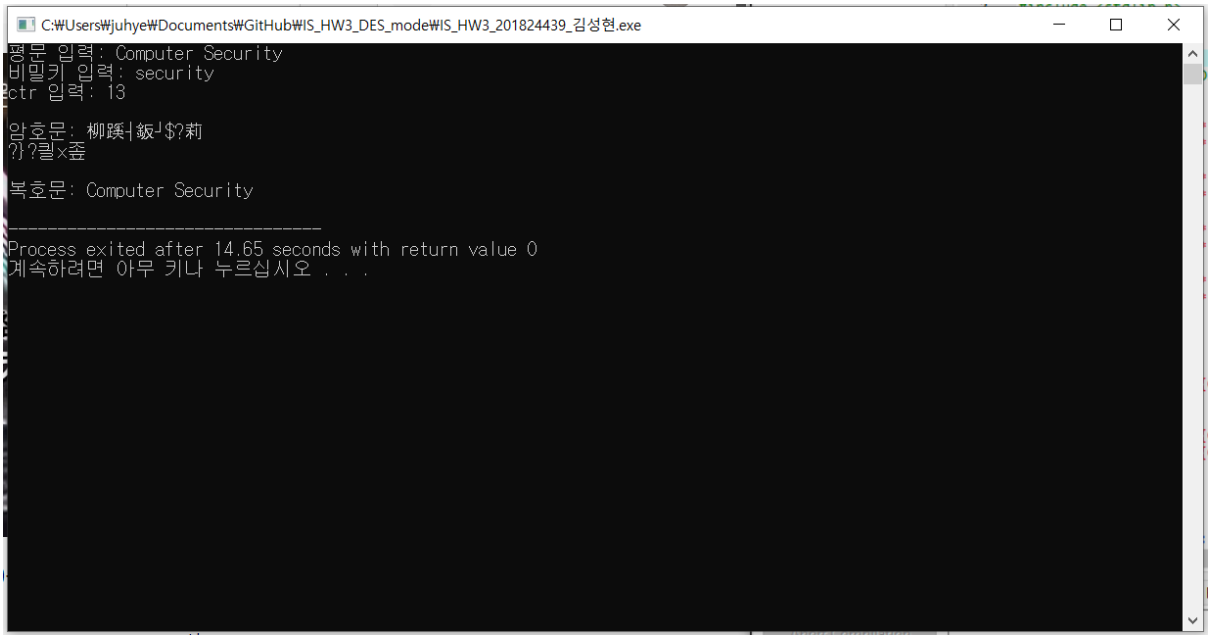
캡처 1. DES_CBC 모드 암호화/복호화 결과 화면



캡처 2. DES_CFB 모드 암호화/복호화 결과 화면



캡처 3. DES_OFB 모드 암호화/복호화 결과 화면



캡처 4. DES_CTR 모드 암호화/복호화 결과 화면