Khanh Nguyen
525000335
CSCE465

Homework 3

Paper-and-Pencil problems:

2.2/ Random J. Protocol-Designer has been told to design a scheme to prevent messages from being modified by an intruder. Random J. decides to append to each message a hash of that message. Why doesn't this solve the problem? (We know of a protocol that uses this technique in an attempt to gain security.)
Anybody can generate and append a hash to any message. Intruder can modify the message and append the recomputed hash value. This modification leads to undetected at the receiving end. Therefore, the problem cannot be resolved.

2.3/ Suppose Alice, Bob, and Carol want to use secret key technology to authenticate each other. If they all used the same secret key K, then Bob could impersonate Carol to Alice (actually any of the three can impersonate the other to the third). Suppose instead that each had their own secret key, so Alice uses KA, Bob uses KB, and Carol uses KC. This means that each one, to prove his or her identity, responds to a challenge with a function of his or her secret key and the challenge. Is this more secure than having them all use the same secret key K? (Hint: what does Alice need to know in order to verify Carol's answer to Alice's challenge?)
Alice needs to know the secret key of Carol to check Carol's response to Alice's challenge. She also needs the secret key from Bob to verify the answer from Bob. To verify each other, they all need the secret keys of each other. Bob could therefore still impersonate Carol to Alice because he would have the secret key for Carol to respond to the challenges that Carol is facing. Even though this is more complex, it is not more secure.

2.4/ As described in §2.6.4 Downline Load Security, it is common, for performance reasons, to sign a message digest of a message rather than the message itself. Why is it so important that it be difficult to find two messages with the same message digest?
If there are 2 messages that have the same digest, then it's possible to forge the messages without any repercussions. This is because the fraudulent activities are hard to detect if there is more than one message with the same digest.

3.2/ Token cards display a number that changes periodically, perhaps every minute. Each such device has a unique secret key. A human can prove possession of a particular such device by entering the displayed number into a computer system. The computer system knows the secret keys of each authorized device. How would you design such a device?
All you need is a clock and some cryptographic scheme containing the key. At every tick of the clock, you generate a new number to display. The number could just be the secret key encryption of the time. The device's clock would have to not drift (or at least-drift in a predictable fashion).

3.3/ How many DES keys, on the average, encrypt a particular plaintext block to a particular ciphertext block?
DES has 56-bit keys, 64-bit plaintext blocks, and 64-bit ciphertext blocks. The number of ciphertext blocks equals the number of plaintext blocks. DES is a 1-1 mapping between ciphertext blocks and plaintext blocks. So 1 plaintext block is mapped to a given ciphertext block by any given key

Khanh Nguyen
525000335
CSCE465

3.5/ Suppose the DES mangler function mapped every 32-bit value to zero, regardless of the value of its input. What function would DES then compute?
With a mangler function that outputs zero always, therefore, after 16 rounds, the initial 64-bit word would be unchanged. All that would happen is the initial permutation, left and right swap and the final permutation. The net result is a permutation that interchanges consecutive even and odd bits. The result If the swap was not there, DES would do nothing at all.

4.2/ The pseudo-random stream of blocks generated by 64-bit OFB must eventually repeat (since at most 2^{64} different blocks can be generated). Will K{IV} necessarily be the first block to be repeated?
K{IV} will be the first block to be repeated.
Let b_i denote the i-fold encryption of IV. So the pad sequence is b_1, b_2, b_1, ..., where b_{i+1} is the encryption of b_i and b_i is the decryption of b_{i+1} . Let b_k be the first repeat element and let $b_k=b_j$ where $j < k$.

- If j=1 then b_j is now b_1 which is repeated)
- If $j > 1$ then $b_{j-1} = b_{k-1}$ (since $b_j = b_k$). So b_k is not the first repeated element. Then, b_j will be the next one repeated.

4.4/ What is a practical method for finding a triple of keys that maps a given plaintext to a given ciphertext using EDE? Hint: It is like the meet-in-the-middle attack of §4.4.1.2 Encrypting Twice with Two Keys.
Let p be the 64-bit block of plaintext and c be the 64-bit block of ciphertext. Set up the following loop to the table with n entries: select randomly a key ; encrypt the key with p. Continue if the result is in the table. Store the result with the key otherwise and proceed unless the table is full.

To find a triple of keys, pick a pair of keys at random, decrypt c with the second key, encrypt the result with the first key, and see if the result is in the table. If it is, a triple of keys is found (the key from the table entry together with the pair of keys just chosen). If not, continue doing it.

The probability that a randomly chosen pair of keys will produce a result in the table is $n/2^{64}$. So the expected number of picks it $2^{64}/n$. If we choose n to be 2^{32}, generating the table and searching for a pair that works will each take about 2^{32} steps, which is quite tractable.

4.6/ Consider the following alternative method of encrypting a message. To encrypt a message, use the algorithm for doing a CBC decrypt. To decrypt a message, use the algorithm for doing a CBC encrypt. Would this work? What are the security implications of this, if any, as contrasted with the "normal" CBC?
Yes, it would work in the sense of allowing encryption and decryption.

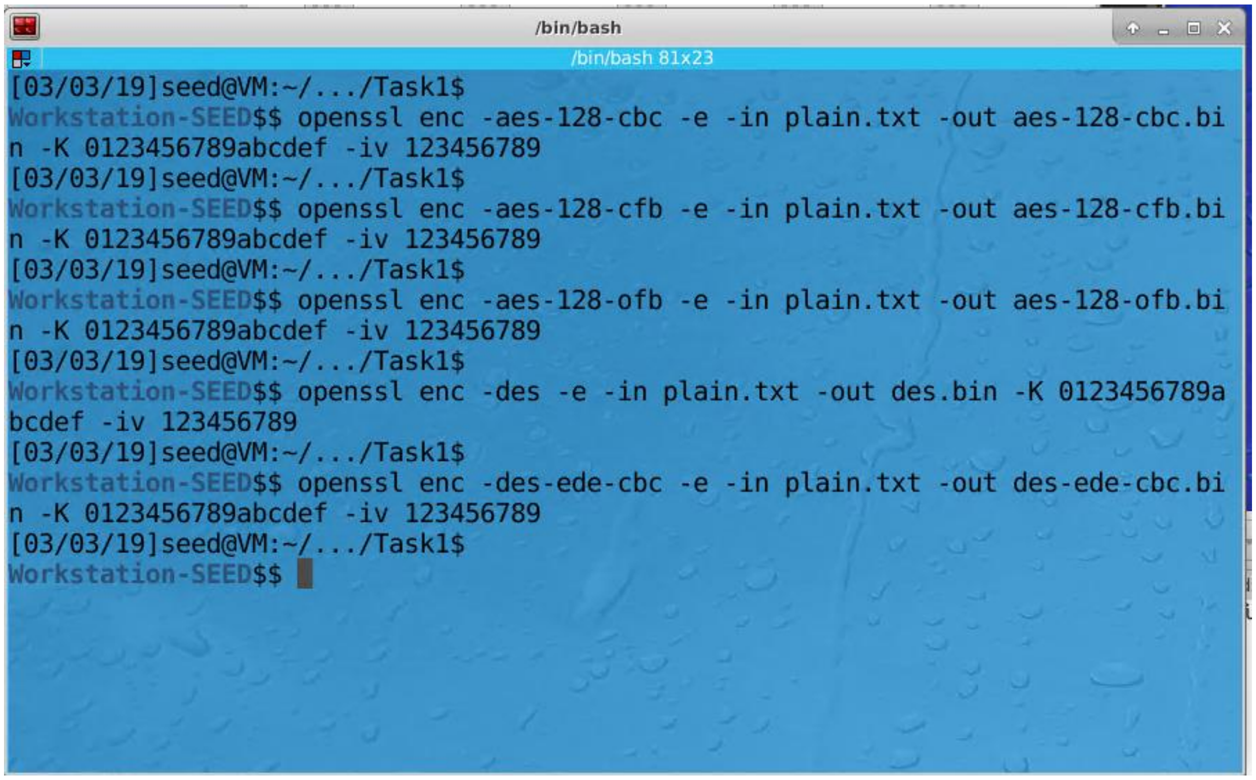
However, one problem with this is that if someone knows the plaintext and ciphertext for a set of messages, the blocks of those messages can be mixed and matched almost as easily as with ECB. Let D denotes the decryption algorithm with the secret key. Block n of plaintext XOR'ed with block n+1 of ciphertext is D of block n+1 of plaintext and once the attacker knows D of a desired block of plaintext, it can be XOR'ed with the plaintext of the previous block to produce correct ciphertext.

Another problem is that, since block n+1 of ciphertext depends only on blocks n and n+1 of plaintext, patterns of ciphertext blocks indicate patterns in the plaintext, which provides a big clue for attackers. And if D of block n+1 of plaintext is known, it can be XOR'ed with block n+1 of ciphertext to get block n of plaintext.

Lab and Programming Tasks:

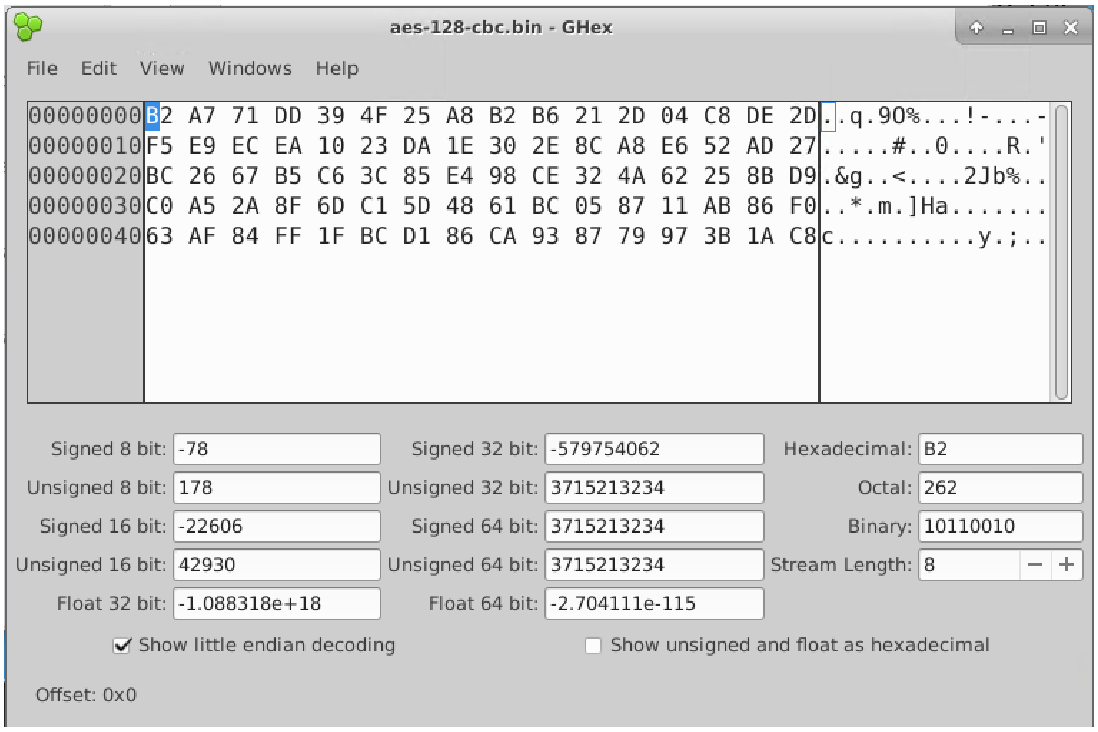
Task 1:

In this task, I created a plaintext file that contains the string " This is task 1 for homework 3 CSCE465. I dont want anyone to see this.". After that , I encrypted the file with 3 different ciphers (AES-128 bits , DES, DES-EDE using 2 keys) and the 3 different modes (CBC, CFB, OFB):



```
[03/03/19]seed@VM:~/.../Task1$
Workstation-SEED$$ openssl enc -aes-128-cbc -e -in plain.txt -out aes-128-cbc.bi
n -K 0123456789abcdef -iv 123456789
[03/03/19]seed@VM:~/.../Task1$
Workstation-SEED$$ openssl enc -aes-128-cfb -e -in plain.txt -out aes-128-cfb.bi
n -K 0123456789abcdef -iv 123456789
[03/03/19]seed@VM:~/.../Task1$
Workstation-SEED$$ openssl enc -aes-128-ofb -e -in plain.txt -out aes-128-ofb.bi
n -K 0123456789abcdef -iv 123456789
[03/03/19]seed@VM:~/.../Task1$
Workstation-SEED$$ openssl enc -des -e -in plain.txt -out des.bin -K 0123456789a
bcdef -iv 123456789
[03/03/19]seed@VM:~/.../Task1$
Workstation-SEED$$ openssl enc -des-ede-cbc -e -in plain.txt -out des-ede-cbc.bi
n -K 0123456789abcdef -iv 123456789
[03/03/19]seed@VM:~/.../Task1$
Workstation-SEED$$
```

Khanh Nguyen
525000335
CSCE465



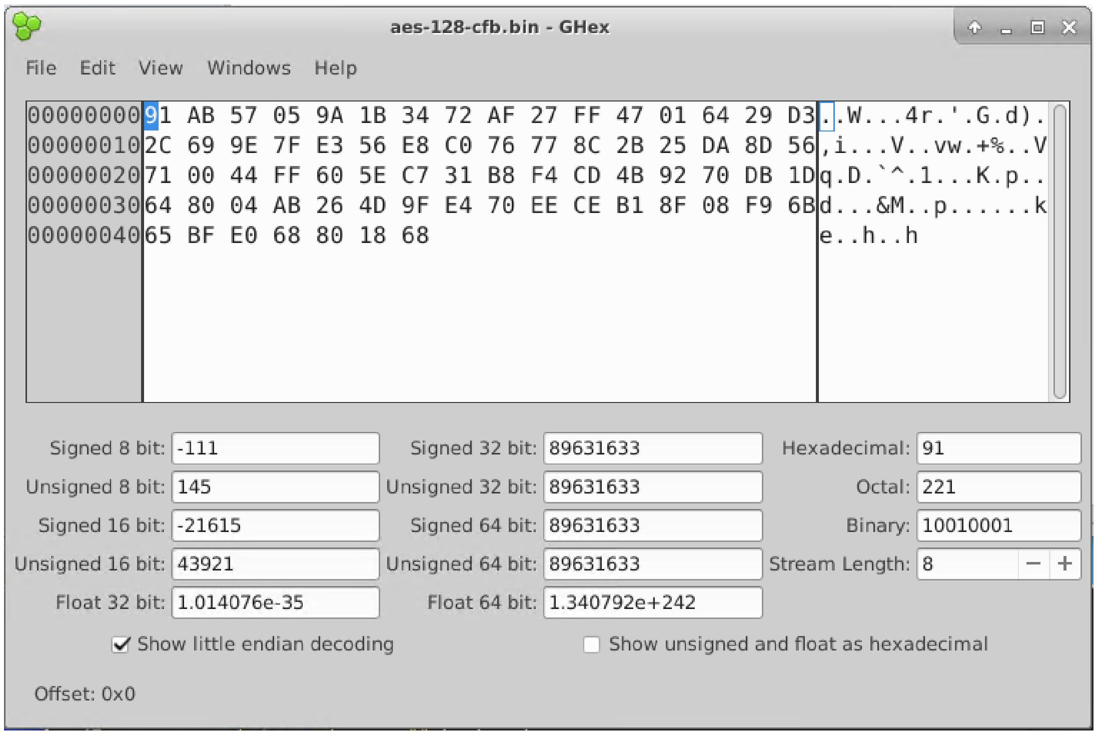
```
00000000 B2 A7 71 DD 39 4F 25 A8 B2 B6 21 2D 04 C8 DE 2D ..q.9O%...!-...-
00000010 F5 E9 EC EA 10 23 DA 1E 30 2E 8C A8 E6 52 AD 27 .....#..0....R.'
00000020 BC 26 67 B5 C6 3C 85 E4 98 CE 32 4A 62 25 8B D9 .&g..<....2Jb%..
00000030 C0 A5 2A 8F 6D C1 5D 48 61 BC 05 87 11 AB 86 F0 ..*.m.]Ha.......
00000040 63 AF 84 FF 1F BC D1 86 CA 93 87 79 97 3B 1A C8 c..........y.;..
```

```
Signed 8 bit: -78              Signed 32 bit: -579754062      Hexadecimal: B2
Unsigned 8 bit: 178            Unsigned 32 bit: 3715213234    Octal: 262
Signed 16 bit: -22606          Signed 64 bit: 3715213234      Binary: 10110010
Unsigned 16 bit: 42930         Unsigned 64 bit: 3715213234    Stream Length: 8
Float 32 bit: -1.088318e+18    Float 64 bit: -2.704111e-115

[✓] Show little endian decoding     [ ] Show unsigned and float as hexadecimal

Offset: 0x0
```

Decrypted aes-128-cbc.bin opened in ghex



```
00000000 91 AB 57 05 9A 1B 34 72 AF 27 FF 47 01 64 29 D3 ..W...4r.'.G.d).
00000010 2C 69 9E 7F E3 56 E8 C0 76 77 8C 2B 25 DA 8D 56 ,i...V..vw.+%..V
00000020 71 00 44 FF 60 5E C7 31 B8 F4 CD 4B 92 70 DB 1D q.D.`^.1...K.p..
00000030 64 80 04 AB 26 4D 9F E4 70 EE CE B1 8F 08 F9 6B d...&M..p......k
00000040 65 BF E0 68 80 18 68                            e..h..h
```

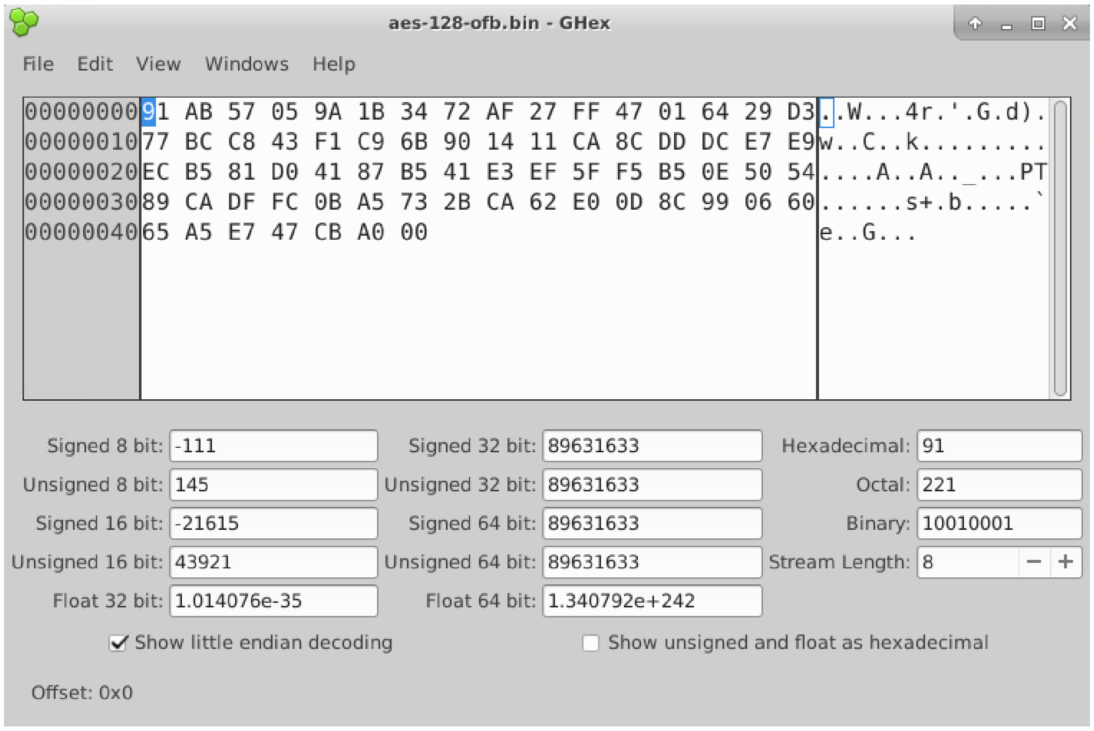
```
Signed 8 bit: -111             Signed 32 bit: 89631633        Hexadecimal: 91
Unsigned 8 bit: 145            Unsigned 32 bit: 89631633      Octal: 221
Signed 16 bit: -21615          Signed 64 bit: 89631633        Binary: 10010001
Unsigned 16 bit: 43921         Unsigned 64 bit: 89631633      Stream Length: 8
Float 32 bit: 1.014076e-35     Float 64 bit: 1.340792e+242

[✓] Show little endian decoding     [ ] Show unsigned and float as hexadecimal

Offset: 0x0
```

Decrypted aes-128-cfb.bin opened in ghex

Khanh Nguyen
525000335
CSCE465


aes-128-ofb.bin - GHex

File Edit View Windows Help

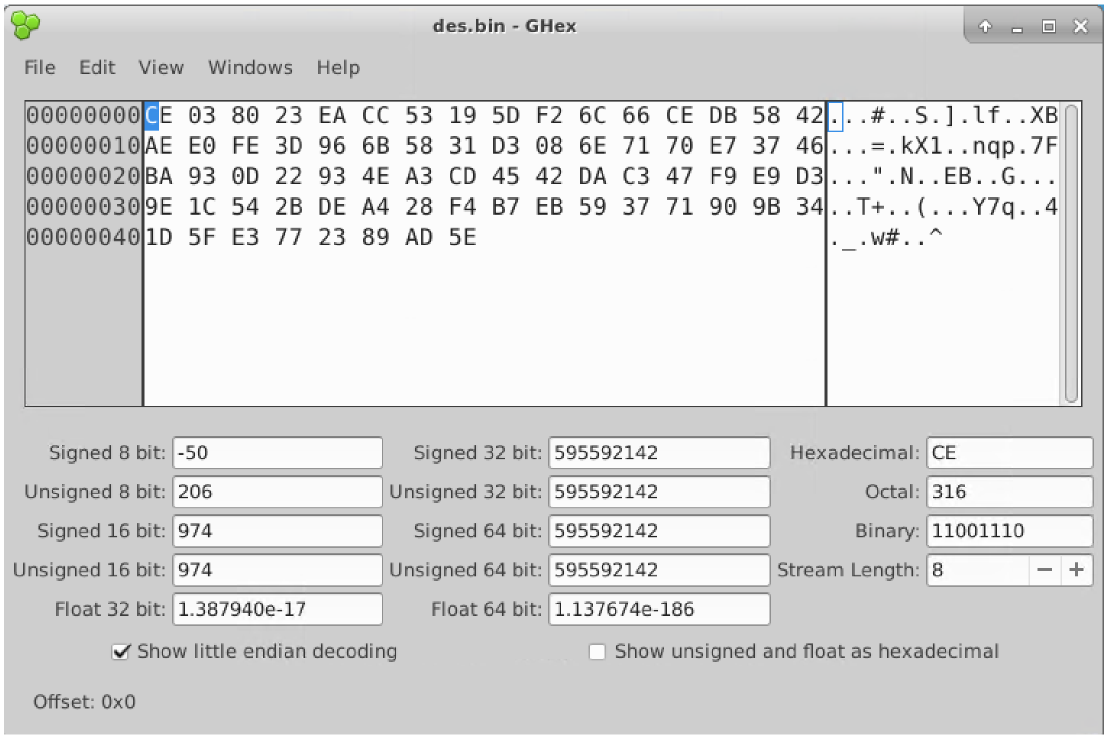
```
0000000091 AB 57 05 9A 1B 34 72 AF 27 FF 47 01 64 29 D3..W...4r.'.G.d).
0000001077 BC C8 43 F1 C9 6B 90 14 11 CA 8C DD DC E7 E9w..C..k.........
00000020EC B5 81 D0 41 87 B5 41 E3 EF 5F F5 B5 0E 50 54....A..A.._...PT
0000003089 CA DF FC 0B A5 73 2B CA 62 E0 0D 8C 99 06 60......s+.b.....`
0000004065 A5 E7 47 CB A0 00                           e..G...
```

Signed 8 bit: -111
Unsigned 8 bit: 145
Signed 16 bit: -21615
Unsigned 16 bit: 43921
Float 32 bit: 1.014076e-35
Signed 32 bit: 89631633
Unsigned 32 bit: 89631633
Signed 64 bit: 89631633
Unsigned 64 bit: 89631633
Float 64 bit: 1.340792e+242
Hexadecimal: 91
Octal: 221
Binary: 10010001
Stream Length: 8

[✓] Show little endian decoding
[ ] Show unsigned and float as hexadecimal

Offset: 0x0

Decrypted aes-128-ofb.bin opened in ghex


des.bin - GHex

File Edit View Windows Help

```
00000000CE 03 80 23 EA CC 53 19 5D F2 6C 66 CE DB 58 42...#..S.].lf..XB
00000010AE E0 FE 3D 96 6B 58 31 D3 08 6E 71 70 E7 37 46...=.kX1..nqp.7F
00000020BA 93 0D 22 93 4E A3 CD 45 42 DA C3 47 F9 E9 D3...".N..EB..G...
000000309E 1C 54 2B DE A4 28 F4 B7 EB 59 37 71 90 9B 34..T+..(...Y7q..4
000000401D 5F E3 77 23 89 AD 5E                        ._.w#..^
```

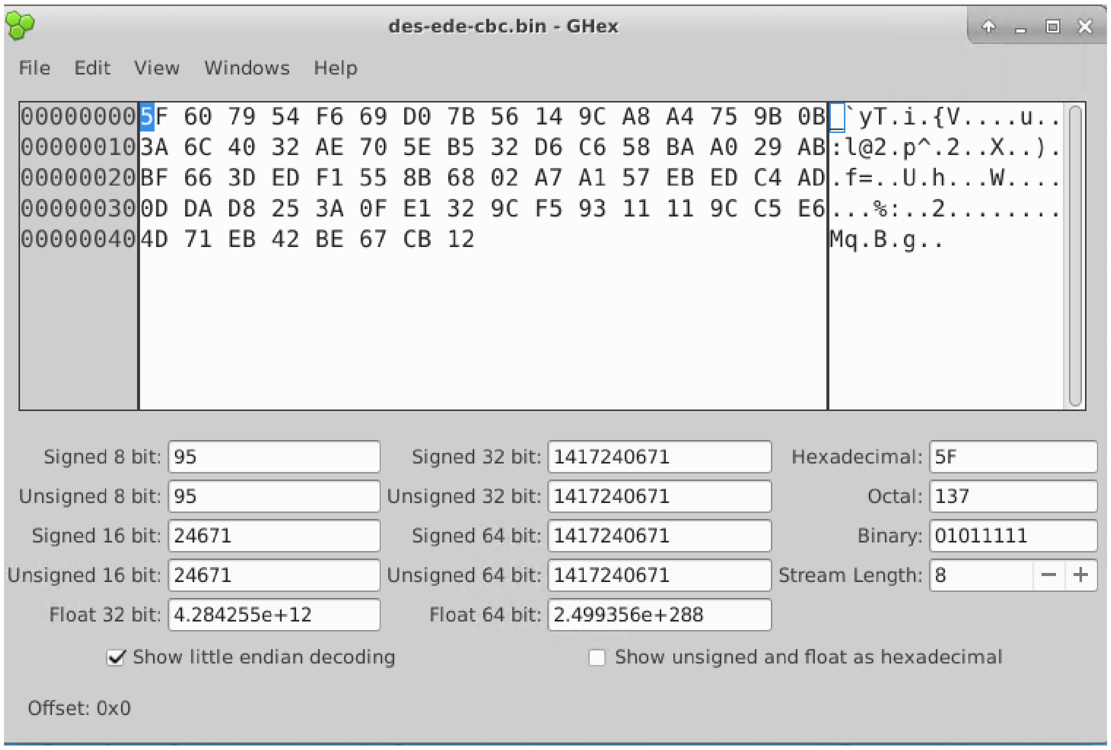
Signed 8 bit: -50
Unsigned 8 bit: 206
Signed 16 bit: 974
Unsigned 16 bit: 974
Float 32 bit: 1.387940e-17
Signed 32 bit: 595592142
Unsigned 32 bit: 595592142
Signed 64 bit: 595592142
Unsigned 64 bit: 595592142
Float 64 bit: 1.137674e-186
Hexadecimal: CE
Octal: 316
Binary: 11001110
Stream Length: 8

[✓] Show little endian decoding
[ ] Show unsigned and float as hexadecimal

Offset: 0x0

Decrypted des.bin opened in ghex

Khanh Nguyen
525000335
CSCE465



```
000000005F 60 79 54 F6 69 D0 7B 56 14 9C A8 A4 75 9B 0B_`yT.i.{V....u..
000000103A 6C 40 32 AE 70 5E B5 32 D6 C6 58 BA A0 29 AB:l@2.p^.2..X..).
00000020BF 66 3D ED F1 55 8B 68 02 A7 A1 57 EB ED C4 AD.f=..U.h...W....
000000300D DA D8 25 3A 0F E1 32 9C F5 93 11 11 9C C5 E6...%:..2........
000000404D 71 EB 42 BE 67 CB 12                        Mq.B.g..
```

Signed 8 bit: 95 | Signed 32 bit: 1417240671 | Hexadecimal: 5F
Unsigned 8 bit: 95 | Unsigned 32 bit: 1417240671 | Octal: 137
Signed 16 bit: 24671 | Signed 64 bit: 1417240671 | Binary: 01011111
Unsigned 16 bit: 24671 | Unsigned 64 bit: 1417240671 | Stream Length: 8
Float 32 bit: 4.284255e+12 | Float 64 bit: 2.499356e+288

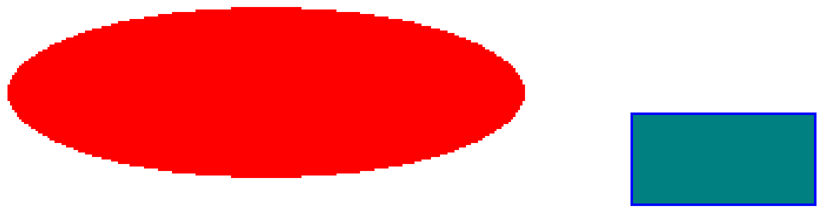
[✓] Show little endian decoding   [ ] Show unsigned and float as hexadecimal

Offset: 0x0

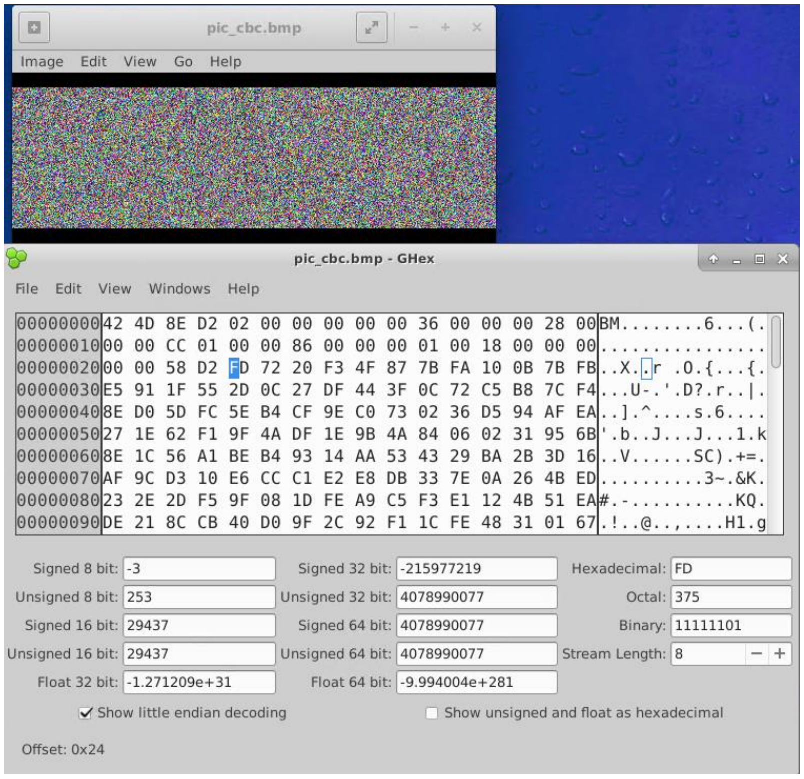
Decrypted des-ede-cbc.bin opened in ghex

Task 2:

In this task, I encrypted a bitmap image given below using AES-128 in botch ECB and CBC modes. After that, I copied heading 36 bits from the original image to the encrypted files for them to be in the right format:



Khanh Nguyen
525000335
CSCE465

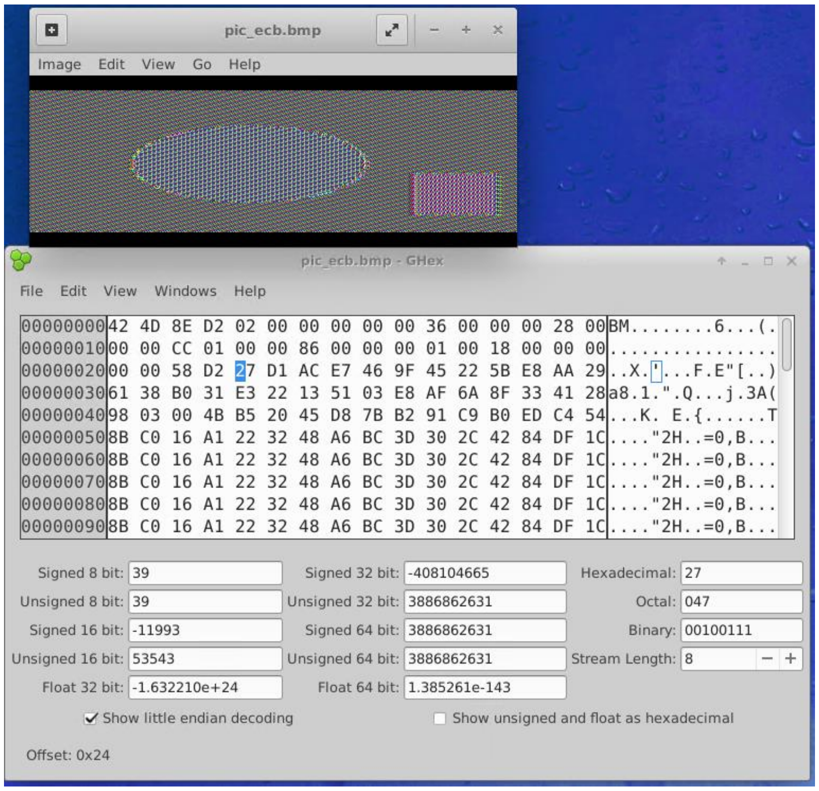


```
00000000 42 4D 8E D2 02 00 00 00 00 00 36 00 00 00 28 00 BM........6...(.
00000010 00 00 CC 01 00 00 86 00 00 00 01 00 18 00 00 00 ................
00000020 00 00 58 D2 FD 72 20 F3 4F 87 7B FA 10 0B 7B FB ..X..r .O.{...{.
00000030 E5 91 1F 55 2D 0C 27 DF 44 3F 0C 72 C5 B8 7C F4 ...U-.'.D?.r..|.
00000040 8E D0 5D FC 5E B4 CF 9E C0 73 02 36 D5 94 AF EA ..].^....s.6....
00000050 27 1E 62 F1 9F 4A DF 1E 9B 4A 84 06 02 31 95 6B '.b..J...J...1.k
00000060 8E 1C 56 A1 BE B4 93 14 AA 53 43 29 BA 2B 3D 16 ..V......SC).+=.
00000070 AF 9C D3 10 E6 CC C1 E2 E8 DB 33 7E 0A 26 4B ED ..........3~.&K.
00000080 23 2E 2D F5 9F 08 1D FE A9 C5 F3 E1 12 4B 51 EA #.-..........KQ.
00000090 DE 21 8C CB 40 D0 9F 2C 92 F1 1C FE 48 31 01 67 .!..@..,....H1.g
```

```
     Signed 8 bit: -3               Signed 32 bit: -215977219    Hexadecimal: FD
   Unsigned 8 bit: 253            Unsigned 32 bit: 4078990077          Octal: 375
    Signed 16 bit: 29437            Signed 64 bit: 4078990077         Binary: 11111101
  Unsigned 16 bit: 29437          Unsigned 64 bit: 4078990077  Stream Length: 8
     Float 32 bit: -1.271209e+31     Float 64 bit: -9.994004e+281
☑ Show little endian decoding        ☐ Show unsigned and float as hexadecimal

Offset: 0x24
```

Encrypted image with CBC



```
00000000 42 4D 8E D2 02 00 00 00 00 00 36 00 00 00 28 00 BM........6...(.
00000010 00 00 CC 01 00 00 86 00 00 00 01 00 18 00 00 00 ................
00000020 00 00 58 D2 27 D1 AC E7 46 9F 45 22 5B E8 AA 29 ..X.'...F.E"[..)
00000030 61 38 B0 31 E3 22 13 51 03 E8 AF 6A 8F 33 41 28 a8.1.".Q...j.3A(
00000040 98 03 00 4B B5 20 45 D8 7B B2 91 C9 B0 ED C4 54 ...K. E.{......T
00000050 8B C0 16 A1 22 32 48 A6 BC 3D 30 2C 42 84 DF 1C ...."2H..=0,B...
00000060 8B C0 16 A1 22 32 48 A6 BC 3D 30 2C 42 84 DF 1C ...."2H..=0,B...
00000070 8B C0 16 A1 22 32 48 A6 BC 3D 30 2C 42 84 DF 1C ...."2H..=0,B...
00000080 8B C0 16 A1 22 32 48 A6 BC 3D 30 2C 42 84 DF 1C ...."2H..=0,B...
00000090 8B C0 16 A1 22 32 48 A6 BC 3D 30 2C 42 84 DF 1C ...."2H..=0,B...
```

```
     Signed 8 bit: 39               Signed 32 bit: -408104665    Hexadecimal: 27
   Unsigned 8 bit: 39             Unsigned 32 bit: 3886862631          Octal: 047
    Signed 16 bit: -11993           Signed 64 bit: 3886862631         Binary: 00100111
  Unsigned 16 bit: 53543          Unsigned 64 bit: 3886862631  Stream Length: 8
     Float 32 bit: -1.632210e+24     Float 64 bit: 1.385261e-143
☑ Show little endian decoding        ☐ Show unsigned and float as hexadecimal

Offset: 0x24
```

Encrypted image using ECB

Khanh Nguyen
525000335
CSCE465

The ECB mode outputs the general shapes of the objects in the image. It's a significant information leaked. The attacker can use this to compromise the message. This happens because ECB encrypts every block independently. That means repeated blocks in plaintext will give identical blocks in ciphertext.

The CBC mode gives a much better encrypted image which cannot be recognized. This is because in CBC mode, the previous block in ciphertext will be XOR'ed with plaintext before the encrypting. As a result, identical blocks in plaintext won't produce identical ciphertext blocks.

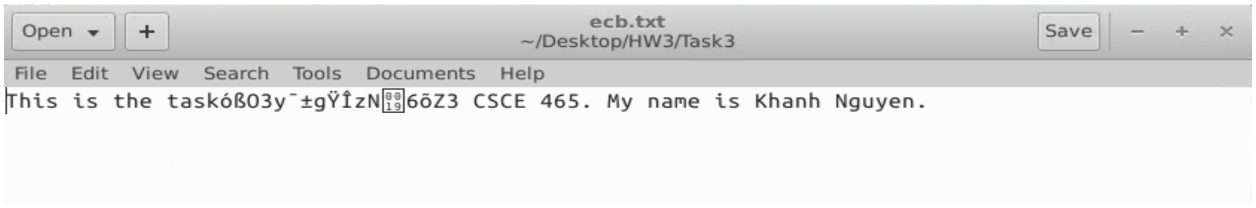
Task 3:

For this task, I created a text file that contains a string at least 64 bytes long. The string I created is "This is the task 3 for homework 3 CSCE 465. My name is Khanh Nguyen."

After that I encrypted the file using 4 different modes: CBC, CFB, ECB and OFB. Then, I corrupted the 30th byte in each of the encrypted file and then decrypted them back.

These are the results I got:

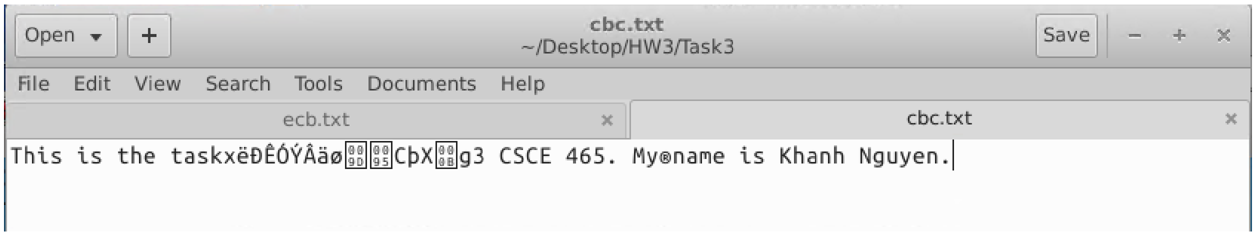
<u>ECB mode:</u>



```
This is the taskóßO3y¯±gŸÎzN6õZ3 CSCE 465. My name is Khanh Nguyen.
```

In ECB, all blocks are encrypted/decrypted independently. The corrupted bit only affects the block that it belongs to. Since the 30th bit is in second block, the error only propagates the second block.

<u>CBC mode:</u>

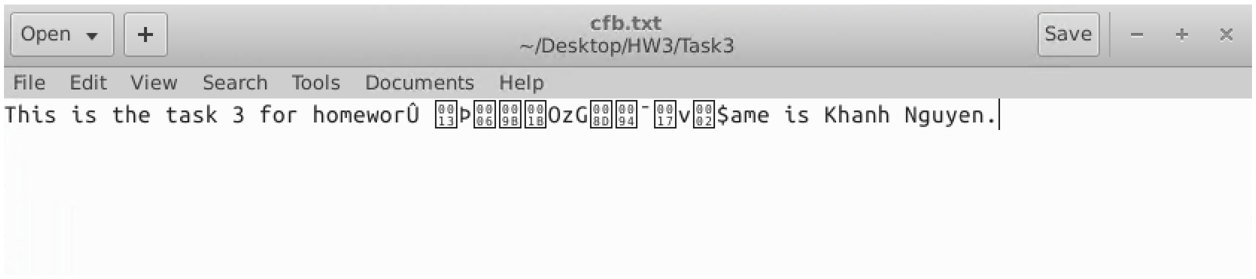


```
This is the taskxëÐÊÓÝÂäøCþXg3 CSCE 465. My®name is Khanh Nguyen.
```

In CBC, the corrupted bit in ciphertext block affects the next plaintext block (They are XOR'ed with each other). The previous corrupted ciphertext will XOR with next ciphertext (which went through decryption process) to make plaintext. Therefore, the result is as expected, the second and third blocks are affected.

Khanh Nguyen
525000335
CSCE465

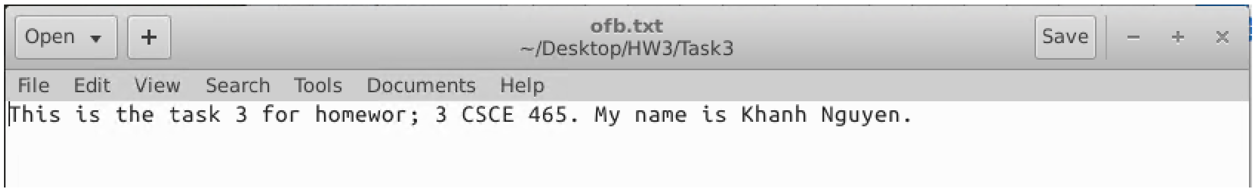
<u>CFB mode:</u>



```
This is the task 3 for homeworÛ þOzG¯v$ame is Khanh Nguyen.
```

In CFB mode, the error propagation is the worst since the corrupted ciphertext will go into the encryption algorithm before XOR with the next ciphertext.

<u>OFB mode:</u>



```
This is the task 3 for homewor; 3 CSCE 465. My name is Khanh Nguyen.
```

In OFB mode, the error doesn’t propagate much, only one 30th bit is affected. That is because in the decryption process, the 30th corrupted bit will be XOR’ed with the output of encryption algorithm to make plaintext.

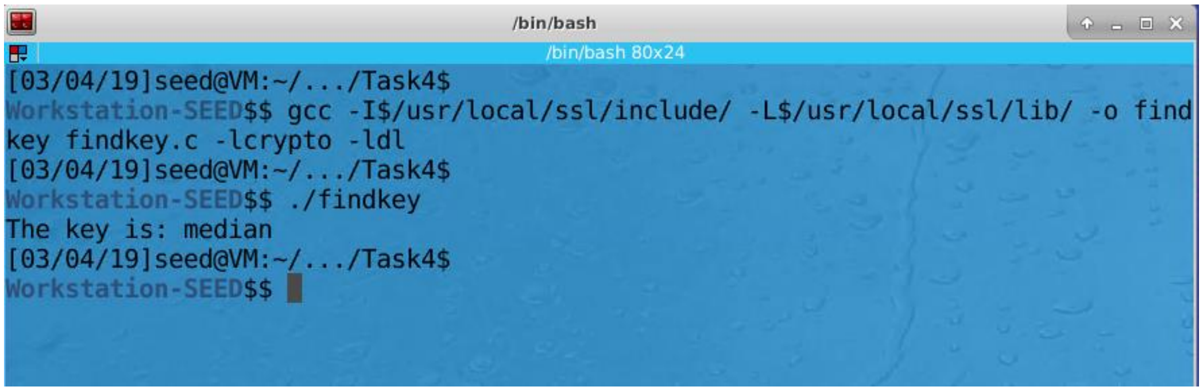
Task 4:

In this task, I was given the plaintext, ciphertext, list of keys (an English word that less than 16 characters), the IV is all zeros and the AES-128-CBC was used. I had write an program that can detect that correct key from the list. I basically encrypted the plaintext given with every key from the dictionary and compared the result with the ciphertext until they match. Since the word may be less than 16 characters, I had to pad it. I also had to convert them into hex forms in order to compare them with the ciphertext given. I used the EVP library to implement this program. These below are the functions that I used for encryption process:

- EVP_CIPHER_CTX_init: initializes cipher context.
- OPENSSL_assert: used to validate key and initialize vector.
- EVP_CipherInit_ex: this takes the arguments that specify the cipher context such as mode, key, integer for encryption/ decryption (1 for encryption).
- EVP_CipherUpdate: encrypts/ decrypts the plaintext/ciphertext.
- EVP_CipherFinal_ex: encrypts/ decrypts the last block.

Khanh Nguyen
525000335
CSCE465

I got the word “median” as the result:



```
[03/04/19]seed@VM:~/.../Task4$
Workstation-SEED$$ gcc -I$/usr/local/ssl/include/ -L$/usr/local/ssl/lib/ -o find
key findkey.c -lcrypto -ldl
[03/04/19]seed@VM:~/.../Task4$
Workstation-SEED$$ ./findkey
The key is: median
[03/04/19]seed@VM:~/.../Task4$
Workstation-SEED$$
```

Here is the code:

```
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>


// Given info
unsigned char *plaintext = "This is a top secret.";
unsigned char ciphertext[] =
{0x8d,0x20,0xe5,0x05,0x6a,0x8d,0x24,0xd0,0x46,0x2c,0xe7,0x4e,0x49,0x04,0xc1,0xb5,
0x13,0xe1,0x0d,0x1d,0xf4,0xa2,0xef,0x2a,0xd4,0x54,0x0f,0xae,0x1c,0xa0,0xaa,0xf9};
unsigned char iv[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

int compare(unsigned char *cipher, int cipher_length, unsigned char *buff, int buff_length);
int check_key(unsigned char *key);

int main(){
    //creates buffer for file pointer
    unsigned char buff[256];

    // opens file words
    FILE *fp = fopen("words.txt", "r");

    if (fp == 0){
            printf("File not opened\n");
            exit(1);
    }

    // reads the file line by line
    if(fp != 0){
        while( fgets (buff,sizeof(buff), fp) != 0) {
            //sets the last character to 0
            buff[strlen(buff) -1] = 0;
            //checks each key from buffer
            int k = check_key(buff);
            if (k){
                printf("The key is: %s\n", buff);
                break;
            }
        }
        fclose(fp);
    }

    return 0;
}

//this is the helper function to compare two ciphertexts
int compare(unsigned char *cipher, int cipher_len, unsigned char *buff, int buff_len){
    //check if they are comparable
    if (cipher_len <= 0 || buff_len <=0 || cipher_len != buff_len){
        return 0;
    }

    // compare two ciphers
    for (int i = 0; i < cipher_len; i++){
        if (cipher[i] != buff[i]){
            return 0;
        }
    }

    //if cipher == buff is true returns 1
    return 1;
}

//This function checks the key from words.txt by encrypting the plaintext with the key
int check_key(unsigned char *key){
    //allocates space for the output
    unsigned char output[1024 + EVP_MAX_BLOCK_LENGTH];
    int output_length = 0;

    int plaintext_length = strlen(plaintext);
    int key_length = strlen(key);
    int buff_output_length;

        //creates the cipher context (ctx)
    EVP_CIPHER_CTX ctx;
```

Khanh Nguyen
525000335
CSCE465

```
    int i;
    //pads all words to make sure they have the same length of 16
    if(key_length < 16){
            for (i = key_length; i < 16; i++){
             key[i] = 0x20;
        }
        }

    // initiallizes cipher context
    EVP_CIPHER_CTX_init(&ctx);

    //initializes encryption type of aes-128-cbc
    EVP_CipherInit_ex(&ctx, EVP_aes_128_cbc(), NULL, NULL, NULL, 1);

    OPENSSL_assert(EVP_CIPHER_CTX_key_length(&ctx) == 16);
    OPENSSL_assert(EVP_CIPHER_CTX_iv_length(&ctx) == 16);

    //initializes key and iv to be relevant to the context
    EVP_CipherInit_ex(&ctx, NULL, NULL, key, iv, 1);

    //update buffer output
    EVP_CipherUpdate(&ctx, output, &buff_output_length, plaintext, plaintext_length);
    output_length = buff_output_length;

    //finalizes the output
    EVP_CipherFinal_ex(&ctx, output + buff_output_length, &buff_output_length);
    output_length += buff_output_length;

    //free function
    EVP_CIPHER_CTX_cleanup(&ctx);

    //now we compare the output with ciphertext
    int check = compare(ciphertext, sizeof(ciphertext), output, output_length);

    return check;
}
```

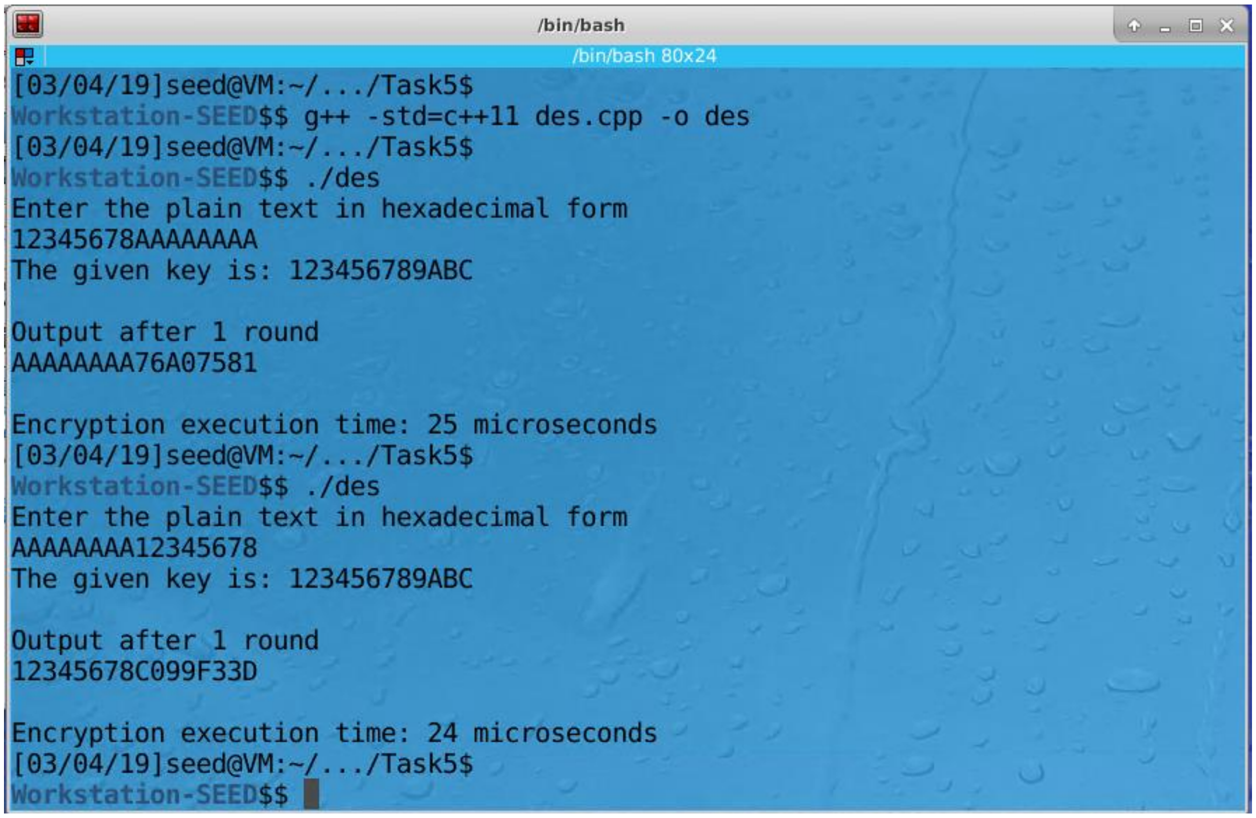
Khanh Nguyen
525000335
CSCE465

Task 5:

For this task I implemented a single DES encryption program with the round key given. The program first takes the user's input, then converts it binary form. Then it will split the input into 2 halves, left and right halves. The right half will be the left half of the output. The right half will also scramble with the mangler function:

- The 32 bit right half will go through the expansion box to get extended to 48 bits.
- After that it XOR with the 48 bit key given.
- The 48 bit result will then go through the S-box given to become 32 bits.
- Then that 32 bits will be permutated.

The 32 bits resulted from the process above will be XOR'ed with left half 32 bit input to produce right half of the output. Here below is my result:



```
[03/04/19]seed@VM:~/.../Task5$
Workstation-SEED$$ g++ -std=c++11 des.cpp -o des
[03/04/19]seed@VM:~/.../Task5$
Workstation-SEED$$ ./des
Enter the plain text in hexadecimal form
12345678AAAAAAAA
The given key is: 123456789ABC

Output after 1 round
AAAAAAAA76A07581

Encryption execution time: 25 microseconds
[03/04/19]seed@VM:~/.../Task5$
Workstation-SEED$$ ./des
Enter the plain text in hexadecimal form
AAAAAAAA12345678
The given key is: 123456789ABC

Output after 1 round
12345678C099F33D

Encryption execution time: 24 microseconds
[03/04/19]seed@VM:~/.../Task5$
Workstation-SEED$$
```

Notice that I entered 64 bits in hexadecimal form input. The right half of the input is always the left half of the output. This is because I did one round DES without initial permutation and final permutation as per requirements. The execution time is 25 microseconds. That is not slow since I only round DES. However, if I made a full-round DES with initial and final permutations, that would be significantly slower since full-round DES is much complicated and contains 16 rounds like this.

Khanh Nguyen
525000335
CSCE465

Here below is the code:

```
#include<bits/stdc++.h>
#include <stdio.h>
#include <fstream>
#include <string.h>
#include <iostream>
#include <stdlib.h>
#include <chrono>

using namespace std;
string hex2bin(string p)//hexadecimal to binary
{
    string ap="";
    int l=p.length();
    for(int i=0;i<l;i++)
    {
        string st="";
        if(p[i]>='0'&&p[i]<='9')
        {
            int te=int(p[i])-48;
            while(te>0)
            {
                st+=char(te%2+48);
                te/=2;
            }
            while(st.length()!=4)
                st+='0';
            for(int j=3;j>=0;j--)
                ap+=st[j];
        }
        else
        {
            int te=p[i]-'A'+10;
            while(te>0)
            {
                st+=char(te%2+48);
                te/=2;
            }
            for(int j=3;j>=0;j--)
                ap+=st[j];
        }
    }
    return ap;
}
```

Khanh Nguyen
525000335
CSCE465

```
int main()
{
        string p,l,r,ap="",ke,kp,rtem;
        pre:;
        //Input of the plain text and the key
        cout<<"Enter the plain text in hexadecimal form \n";
        cin>>p;
        if(p.length()!=16)
        {
            cout<<"enter all the bits\n";
            goto pre;
        }
        for(int i=0;i<16;i++)
        {
            if((p[i]>='0'&&p[i]<='9')||(p[i]>='A'&&p[i]<='F'))
                ;
            else
            {
                cout<<"Not a valid hexadecimal string\n";
                goto pre;
            }
        }
        int key[48];
        //This is the given key
        ke="123456789ABC";
        cout<<"The given key is: "<<ke<<endl;
        //Input is completed
        p=hex2bin(p);
        kp=hex2bin(ke);
        for(int i=0;i<48;i++){
            key[i]=kp[i]-'0';
        }
        cout<<endl;

     // beginning clock time to measure
    chrono::high_resolution_clock::time_point t1 = chrono::high_resolution_clock::now();

    int i,t=1,row,col,temp,round=1;
    l=p.substr(0,32);
    r=p.substr(32,32);
    //permutation after sbox
    int per[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};

 //sbox configuration 6 bit to 4 bit
 int s[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
    }};
    //DES Encryption
        while(round--)
        {
            rtem=r;
            t=1;
            string ep="",xorout="",sout="",soutt;
            //the expansion P box
            ep+=r[31];
            for(i=0;i<32;i++)
            {
                if((t+1)%6==0)
                {
                        ep+=r[4*((t+1)/6)];
                        t++;
                }
                if(t%6==0&&i!=0)
                {
                    ep+=r[4*(t/6)-1];
                    t++;
                }
                ep=ep+r[i];
                t++;
            }
            ep+=r[0];
            //Key xor with output of expansion p box
            for(i=0;i<48;i++)
                xorout+=char(((int(ep[i])-48)^key[i])+48);
            //sbox compression 48bit to 32 bit
            for(i=0;i<48;i+=6)
            {
                    row=(int(xorout[i+5])-48)+(int(xorout[i])-48)*2;
                    col= (int(xorout[i+1])-48)*8+(int(xorout[i+2])-48)*4+(int(xorout[i+3])-
48)*2+(int(xorout[i+4])-48);
                    temp=s[i/6][row][col];
                    soutt="";
                    while(temp>0)
                    {
                        soutt+=char(temp%2+48);
                        temp/=2;
                    }
                    while(soutt.length()!=4)
                        soutt+='0';
                    for(int j=soutt.length()-1;j>=0;j--)
                        sout+=soutt[j];
            }
            //permutation of the sbox output
            char pc[32];
            for(i=0;i<32;i++)
                pc[i]=sout[per[i]-1];
            r="";
            for(i=0;i<32;i++)
                r+=char(((int(pc[i])-48)^(int(l[i])-48))+48);
            l=rtem;
            cout<<"Output after "<<1-round<<" round"<<endl;
            string cip="";
            for(i=0;i<32;i+=4)
            {
                int te;
                te=(int(l[i])-48)*8+(int(l[i+1])-48)*4+(int(l[i+2])-48)*2+(int(l[i+3])-48);
                if(te<10)
                    cip+=char(te+48);
                else
                    cip+=char(te+55);
            }
            for(i=0;i<32;i+=4)
            {
                int te;
                te=(int(r[i])-48)*8+(int(r[i+1])-48)*4+(int(r[i+2])-48)*2+(int(r[i+3])-48);
                if(te<10)
                    cip+=char(te+48);
                else
                    cip+=char(te+55);
            }
            cout<<cip<<endl;
        }
    // Ending clock time
    chrono::high_resolution_clock::time_point t2 = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(t2 - t1).count();

    cout << "\nEncryption execution time: " << duration << " microseconds" << endl;
```

```
	return 0;
}
```