**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

**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 264 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 bi denote the i-fold encryption of IV. So the pad sequence is b1, b2, b1, ....., where bi+1 is the encryption of bi and bi is the decryption of bi+1 . Let bk be the first repeat element and let bk=bj where j < k.

* If j=1 then bj is now b1 which is repeated)
* If j > 1 then bj-1 = bk-1 (since bj = bk). So bk is not the first repeated element. Then, bj 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**](file:///C:\Volumes\Hackintosh%20HDD%201\CSCE%20465\Network-Security-Textbook-CSCE-465\ch04lev1sec4.html#ch04lev3sec4) **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/264. So the expected number of picks it 2­­64/n. If we choose n to be 232, generating the table and searching for a pair that works will each take about 232­ 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.

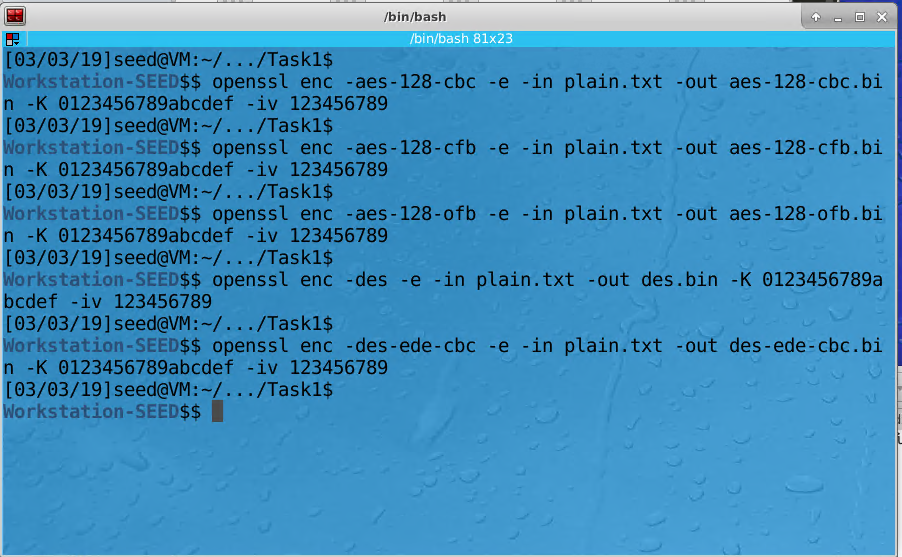
Another problem is that, since block n+1 of ciphertext depends only on blocks n and n+1 of plaintext, patterns of ciphertext blocks indicate patterns in the plaintext, which provides a big clue for attackers. And if D of block n+1 of plaintext

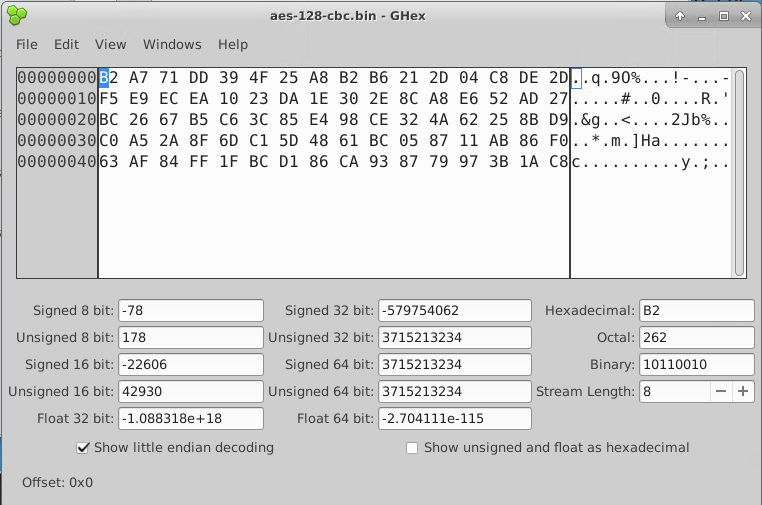
t is known, it can be XOR’ed with block n+1 of ciphertext to get block n of plaintext.

**Lab and Programming Tasks:**

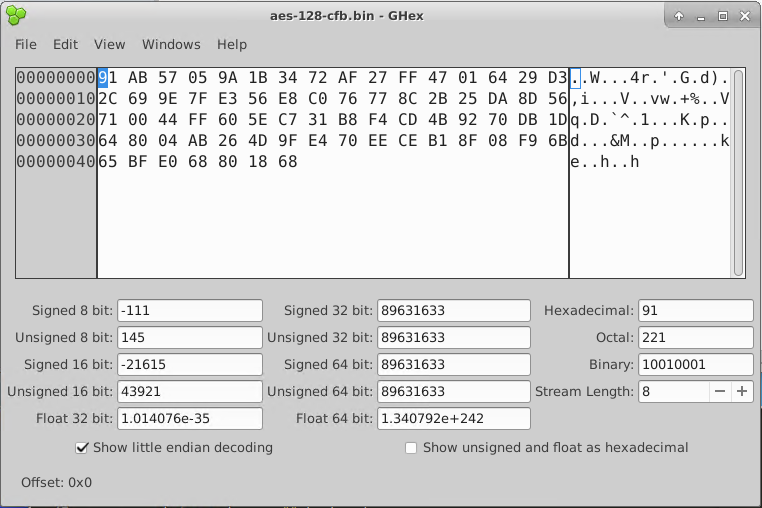
**Task 1:**

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):

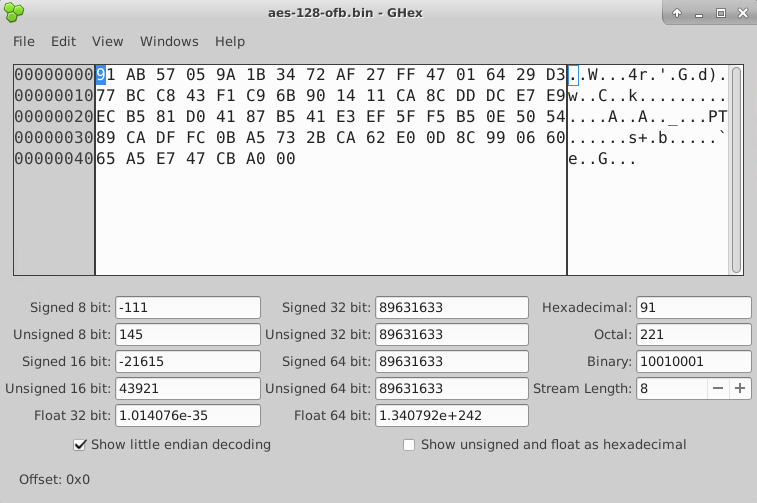




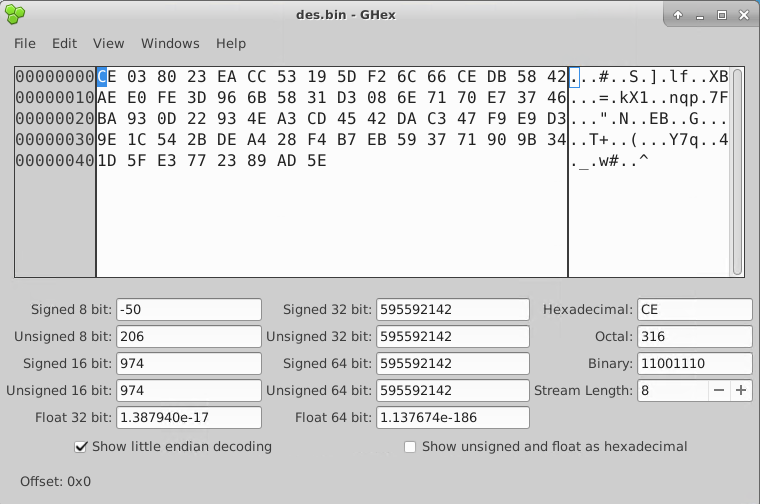
Decrypted aes-128-cbc.bin opened in ghex



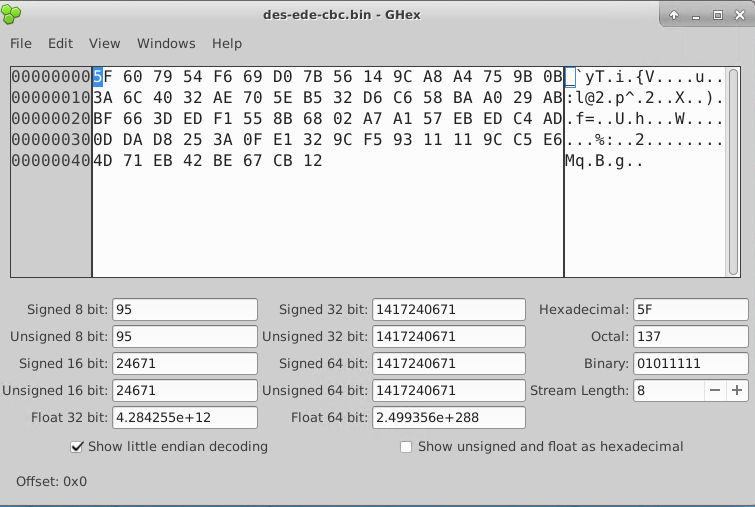
Decrypted aes-128-cfb.bin opened in ghex



Decrypted aes-128-ofb.bin opened in ghex



Decrypted des.bin opened in ghex

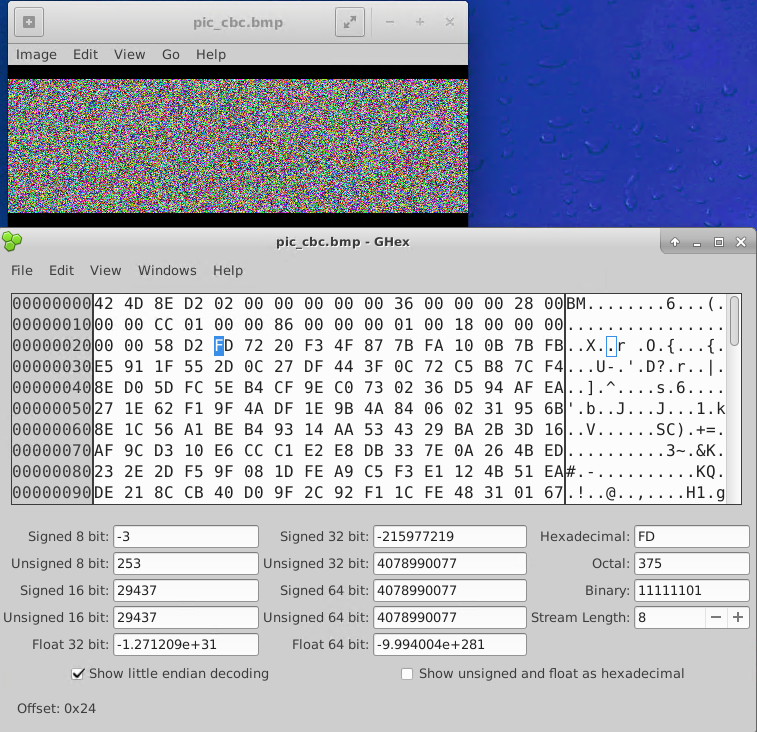


Decrypted des-ede-cbc.bin opened in ghex

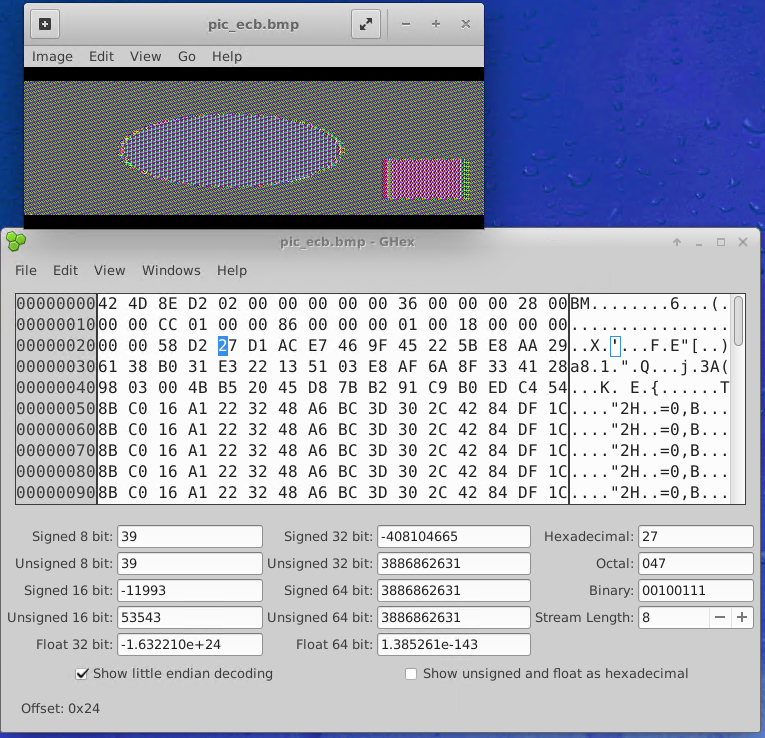
**Task 2:**

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:





Encrypted image with CBC



Encrypted image using ECB

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.

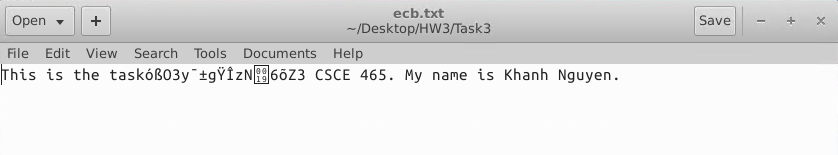
**Task 3:**

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.

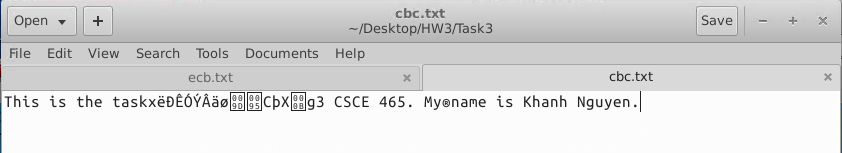
These are the results I got:

ECB mode:



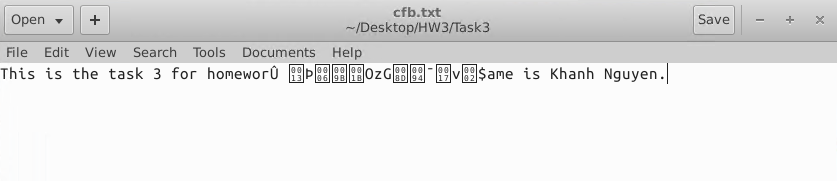
In ECB, all blocks are encrypted/decrypted independently. The corrupted bit only affects the block that it belongs to. Since the 30th bit is in second block, the error only propagates the second block.

CBC mode:



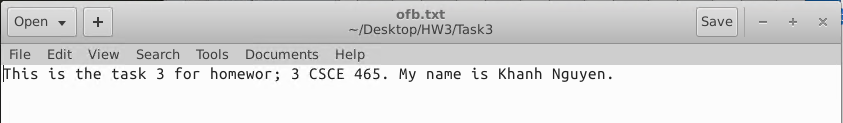
In CBC, the corrupted bit in ciphertext block affects the next plaintext block (They are XOR’ed with each other). The previous corrupted ciphertext will XOR with next ciphertext (which went through decryption process) to make plaintext. Therefore, the result is as expected, the second and third blocks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

CFB mode:



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.

OFB mode:



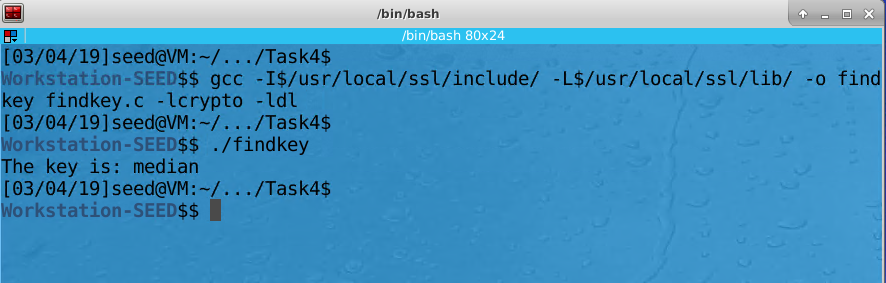
In OFB mode, the error doesn’t propagate much, only one 30th bit is affected. That is because in the decryption process, the 30th corrupted bit will be XOR’ed with the output of encryption algorithm to make plaintext.

**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.

I got the word “median” as the result:



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;

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;

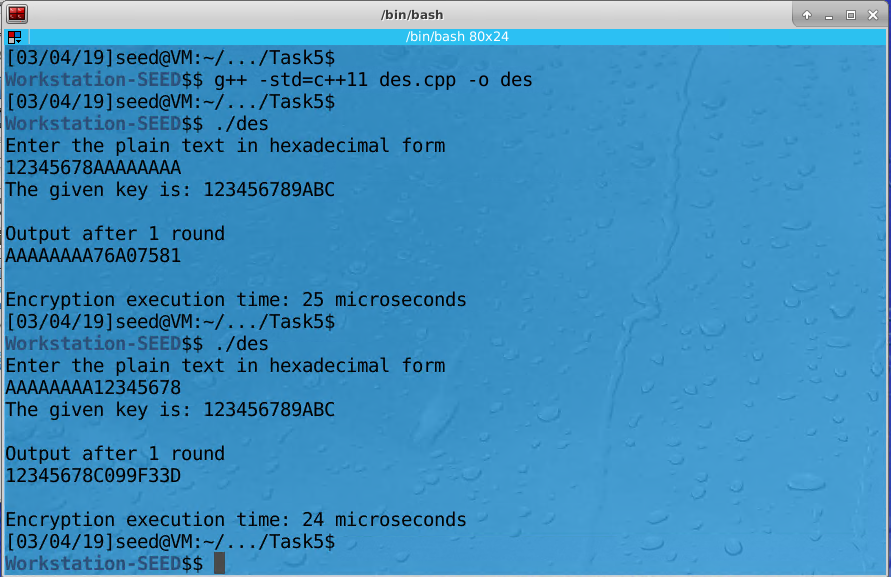
}

**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:



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.

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;

}

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 << "**\n**Encryption execution time: " << duration << " microseconds" << endl;

**return** 0;

}