Faculty of Engineering and Technology

Electrical & Computer Engineering Department

ENCS4320

Applied Cryptography

Cryptography Lab Report

Prepared by:

Al-Ayham Maree 1191408

Sara Ammar 1191052

Supervised by:

Dr. Ahmad Alsadeh

Section:

2

11-1-2023

Contents

Table of Figures:

Abstract

This lab was intended to familiarize users with many concepts in secret-key encryption and how an attacker can break the security of encryption, as well as to gain experience with encryption algorithms and encryption modes, padding, initial vector (IV), and the tasks in this lab helping users in being able to use the tools and write programs to encryption and decryption messages in different ways and modes.

Task 1: Frequency Analysis:

Step 1:

The encryption key was generated using permutation alphabet from (a-z), and the permuted alphabet was used as a key, using python code as shown below

Figure 1: Generating key using python code

After encryption, the mono-alphabetic substitution cipher was produced as shown below

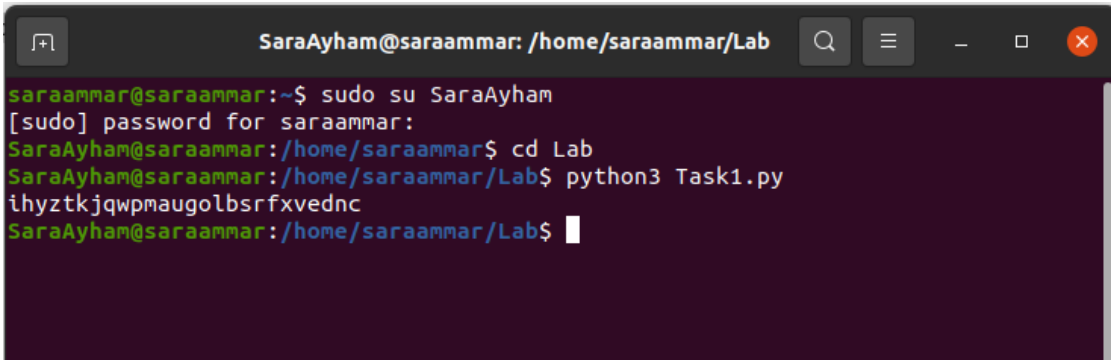

Figure 2: The mono-alphabetic substitution cipher

Step 2:

In the figure below, the plaintext (message) has been converted to lower case, and the non-alphabetic characters has been removed before producing the mono-alphabetic cipher

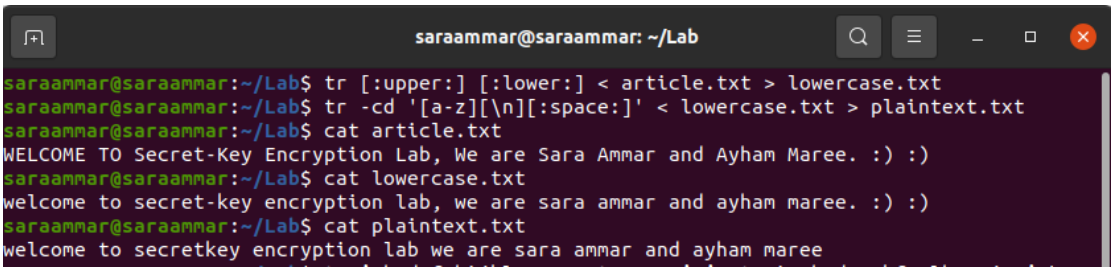

Figure 3: - Converting to lowercase and removing non-alphabetic characters

Step 3:

In this step, the encryption has been worked using the "tr" command, and the letters has been encrypted only, and the space and return characters has been left alone, as shown below:

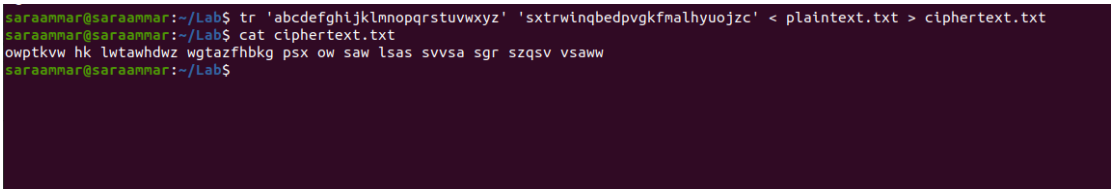

Figure 4: the encryption results

The "ciphertext.txt" has been read and the statistics of n-gram was produced, including the single letter frequencies, bigram frequencies, and trigram frequencies using the "freq.py" python program inside the "LabSetup/Files" folder, as shown in figures below:

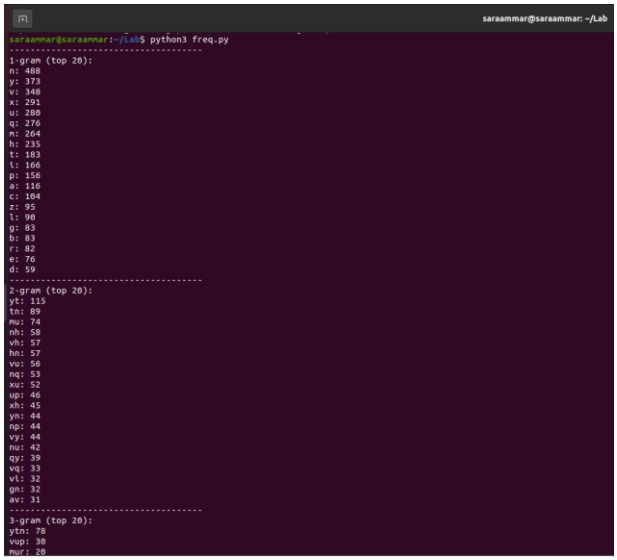

Figure 5: The frequencies of n-gram using freq.py

In this task, the cipher.txt was converted to plain.txt using the frequency analysis of letters, and we utilized Python to perform the letter conversion in order to get the original message that was sent (You can find it in the appendix1). First, the message 'THE' was produced from decryption the cipher 'ytn' from English, so every Y in the cipher is a T, every T is an H, and every N is an E.

The cipher was 'n,' and the most repeating letter in English is an 'E,' therefore that suggests we are on the right track to decryption. Alternatively, each letter that repeated numerous times was counted. We currently know the cipher's solution for the first three letters, and when we altered those letters and searched for another word, we discovered the following:

1. Since just the letter "v" was found in the cipher, only the letter "A" in English may be found alone. As a result, every "v" in the cipher was altered to an "A," and we then complied again.
2. What else might the word 'Tx' (capital letters indicate converted English letters) be if not 'TO'
3. The cipher contained the words "uO" and "Ou," and we therefore predicted that every "u" is an "N," leading to the creation of the letters "NO" and "ON."
4. What else could the word "ANp" be if not "AND"? Therefore, every "p" is a "D."
5. Since every letter has been altered except for the letter "g," the sentence "TO gE A" has been found. However, given the meaning of the sentence, we had expected that it would be "TO BE A," thus each "g" is now a "B."
6. Since the word "HOl" was found, every "l" is a "W," making it impossible for it to be anything else.
7. Every "h" is an "R" since the word "OTHEh" was discovered; what else could it be.
8. The word "Aii" was found; what else could it be except "ALL," thus each "I" is a "L."
9. The word "mT" was found, but since "A" was already present, it couldn't be "AT"; instead, it could only be "IT," meaning that every "m" is an "I."
10. Since we found a replacement for the letter "R," the word "WAq" couldn't be "WAR" and could only be "WAS," which means that every "q" is a "S."

11. When the word "RIrHT" was found, it could only mean "RIGHT," hence each r stands for "G."
12. Every 'z' is a 'U' because the word 'ABOzT' was found; what else could it be except 'ABOUT'.
13. When the word "AbTER" was found, it could only mean "AFTER," hence every "b" is a "F.
14. Since the word "SUNDAd" was found to be "SUNDAY," every "d" is a "Y."
15. Since the word "THANsS" was found, it could only mean "THANKS," hence each s represents a "K."
16. When the word "DREAc" was found, it could only mean "DREAM," hence each c stands for "M."
17. The word "RAaE" was found; hence, every "a" is a "C," and it could only be "RACE."
18. When the word "TRIe" was found, it could only be "TRIP," therefore each 'e' is a 'P'.
19. When the word "AfOID" was found, it could only mean "AVOID," hence each f stands for "V."
20. The word "oUST" was found; since it could only be "JUST," every "o" is a "J."
21. Since the word "EkSTRA" was found, every "k" is an "X," and there is no other possible meaning for it.
22. When the word "EjUALLY" was found, it could only mean "EQUALLY," therefore every "j" is a "Q."

The key found is therefore "VGAPNBRTMOSICUXEJHQYZFLKDW"

Following the decryption, the plain.txt can be found here (you can also find it in the appendix as text):

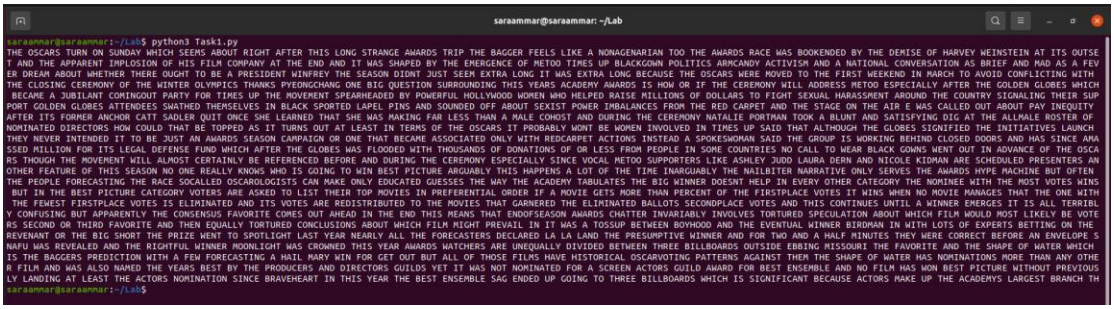

Figure 6: The decryption result

Task 2: Encryption using different ciphers and modes

In this task, various encryption algorithms and modes was used, “openssl end” command has been used to encrypt/decrypt file, AES 128 bits Cipher Block Chain, Blow Fish Cipher Block Chain, and Cipher Feedback were the three block ciphers we tried. The figure below illustrates how the various ciphers in the cipher texts differ:

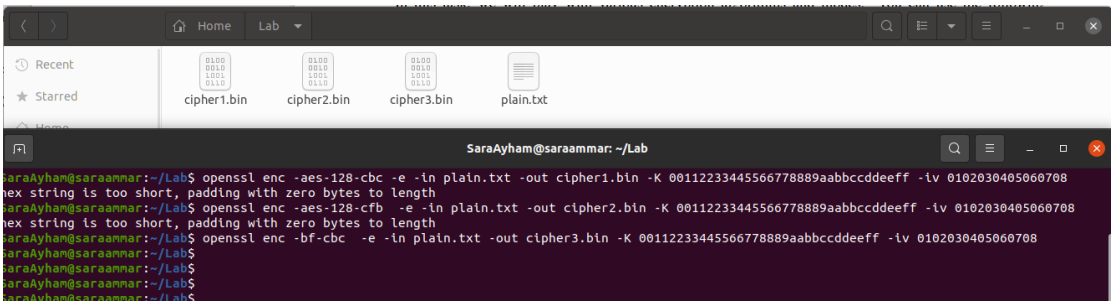

Figure 7: encryption using different cipher modes

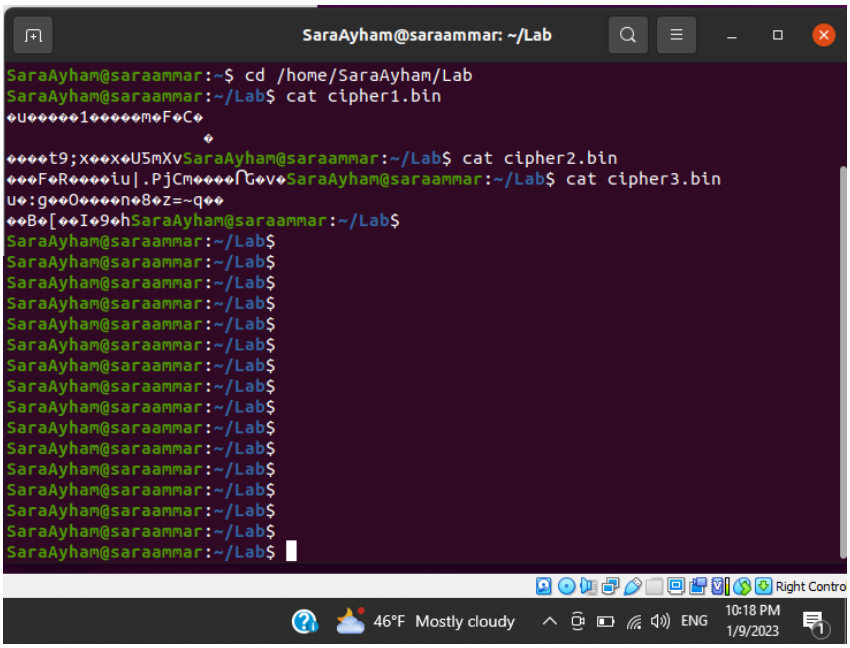

Figure 8: Encryption result and the cipher formats using different modes

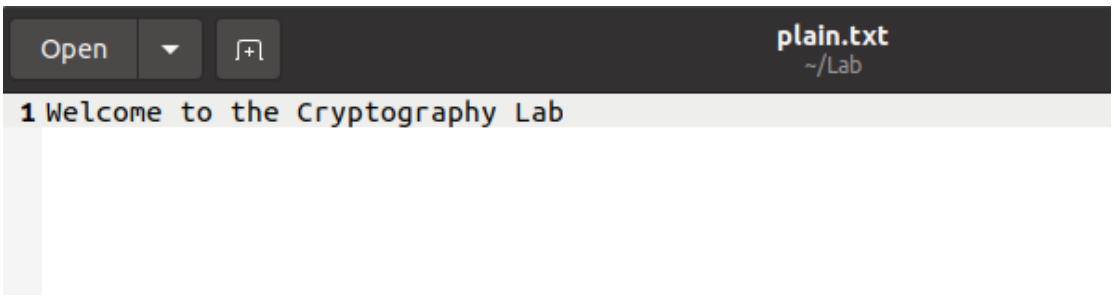

Figure 9: Plaintext used

Task 3: Encryption mode ECB vs CBC

The penguin image (.bmp) has been encrypted by three cipher modes: Electronic Codebook Mode and Cipher Block Chaining Mode with Initialization Vector (IV) and AES-128 Cipher Block Chaining .Therefore, the body of the image was encrypted and the original header has been put, the following figures show the encryption using three cipher modes:

1. Encryption image using ECB:

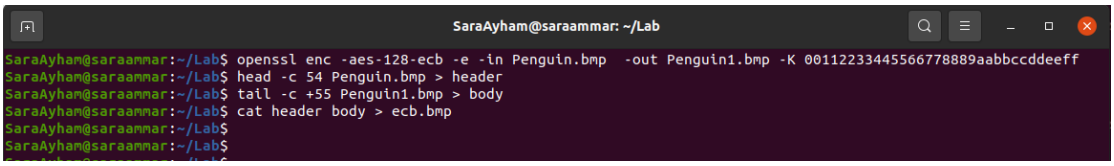

Figure 10: Encryption use ECB

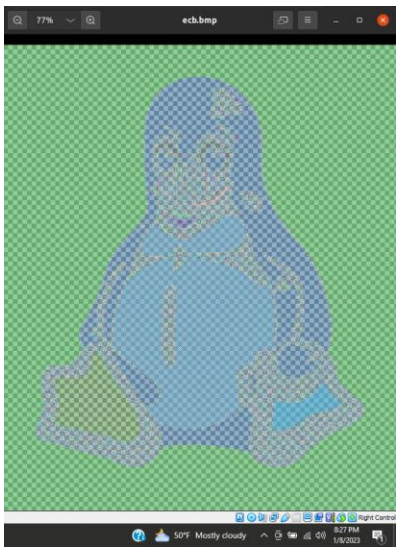

Figure 11: Encryption result using ECB

2. Encryption result using CBC mode with IV:

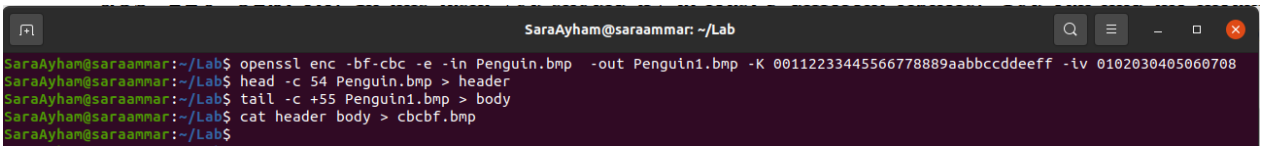

Figure 12: Encryption using CBC with IV

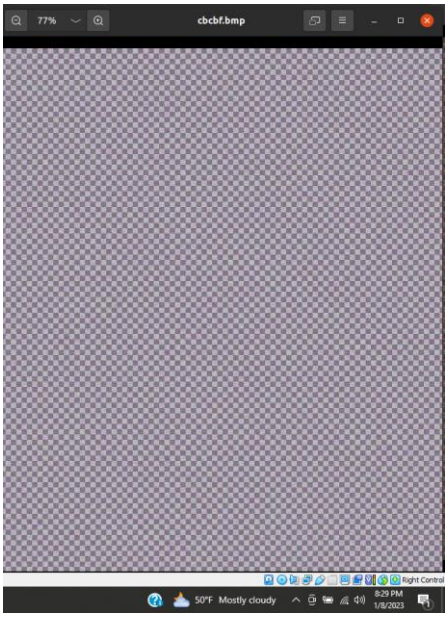

Figure 13: Encryption Result using CBC with IV

3. Encryption image using AES-128 CBC:

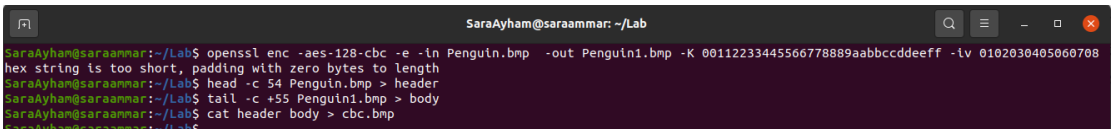

Figure 14: Encryption using AES-128 CBC

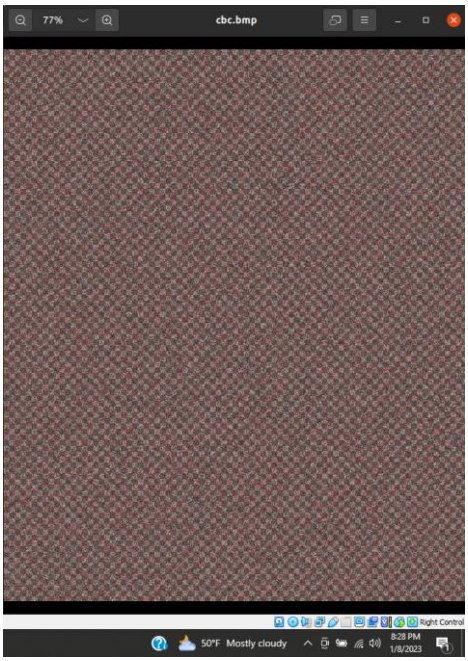

Figure 15: Encryption result of AES-128 CBC

Task 4: Padding

In this task, the three files with different lengths was encrypted using three ciphers: ECB, CBC and CFB. Firstly, the three files has been created, which contains f1 = 5 bytes, f2 =10 bytes, f3=16 bytes, files has been created using the command “echo -n” as shown in figure below:

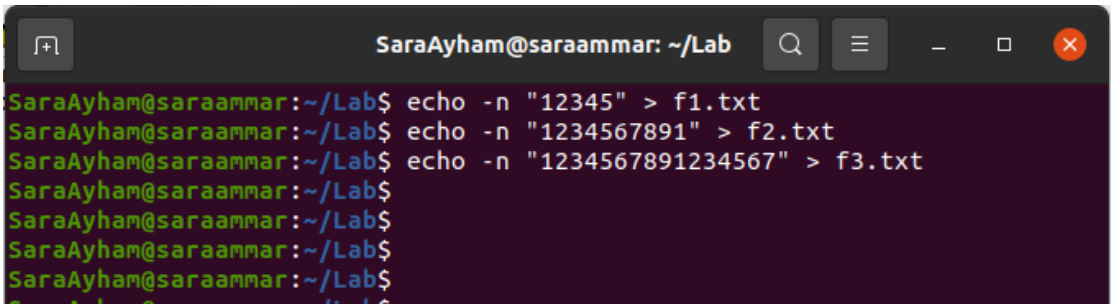

Figure 16: Creating three files

1. Encryption three files using ECB

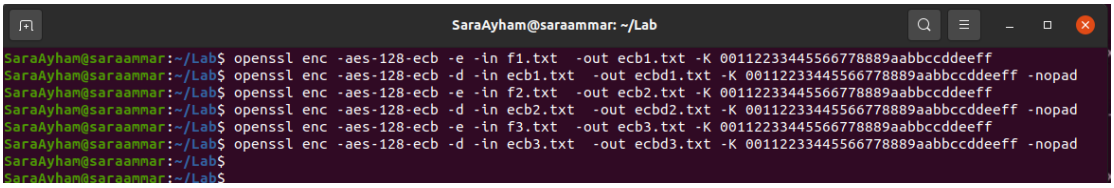

Figure 17: Encryption files using ECB

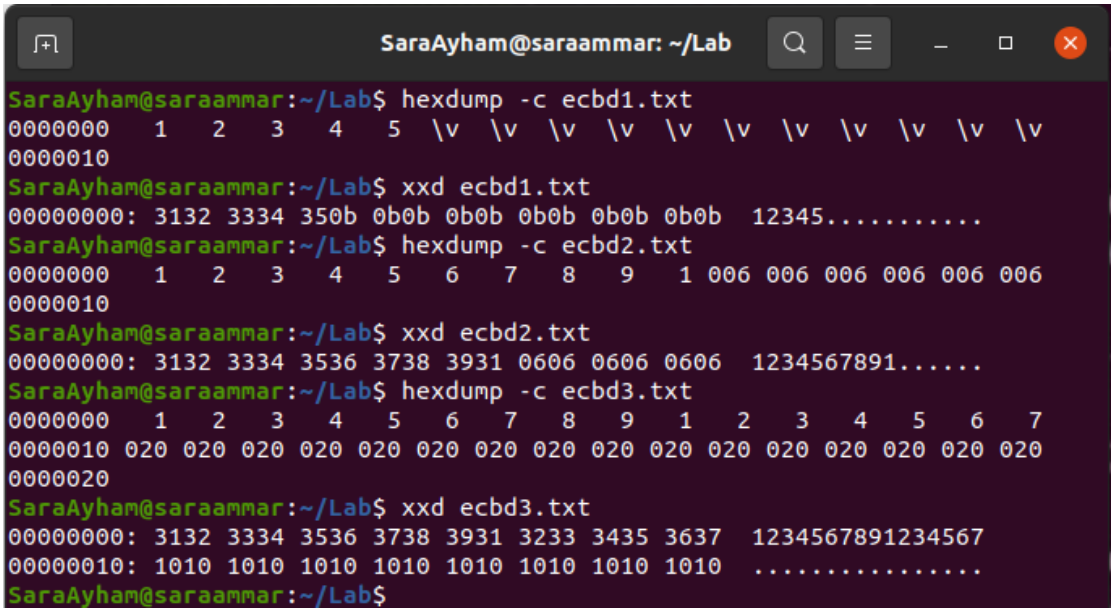

Figure 18: encrypted files result using ECB

2. Encryption files using CBC:

The first encrypted file in CBC included 11 bytes of 0X0B as padding data because the total block size in CBC is 16 bytes, and B in HEX equals 11 in decimal. As padding data, the second file adds 6 bytes of 0X06. The padding bytes for the third file are 16 bytes of 0X10 (10 in HEX=16 in decimal), as indicated in the figure below, because its length is greater than the block size of 16.

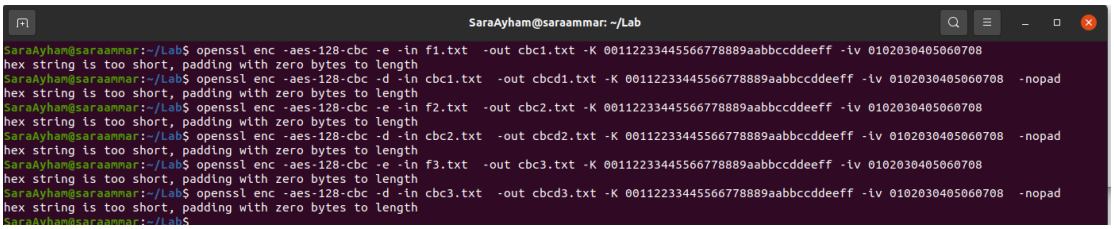

Figure 19: Encryption files using CBC

As shown in the figure below, the CBC is the same concept as padding the ECB.

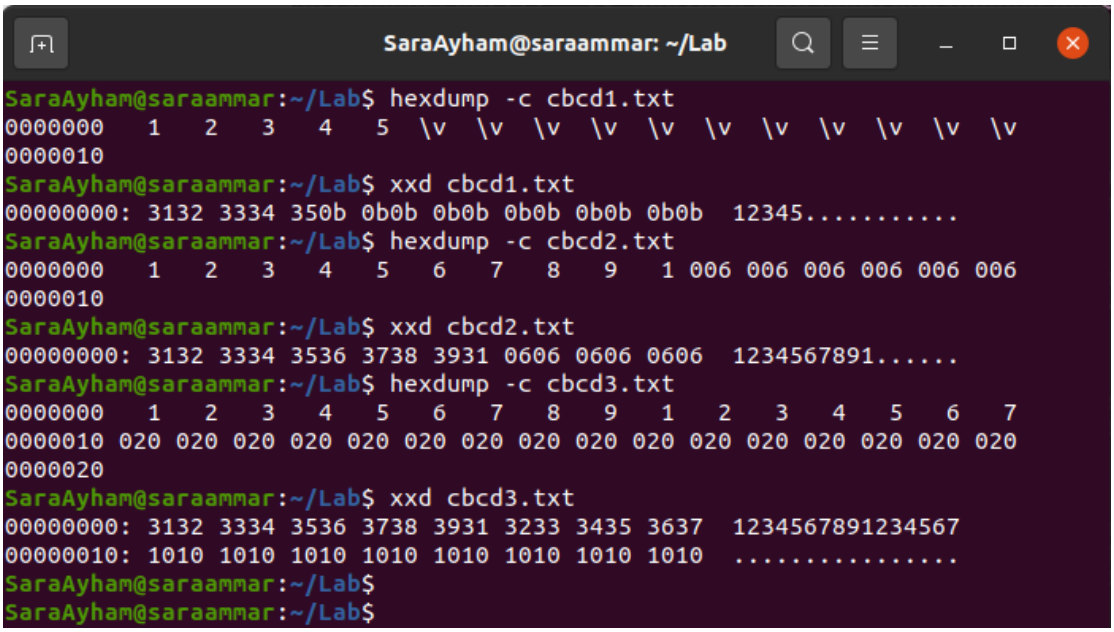

Figure 20: Encryption result of files using CBC and padding files

3. Encryption files using CFB:

Contrary to CBC and ECB, which require padding, CFB's cipher text size is the same as plain text size, as illustrated in the figure below:

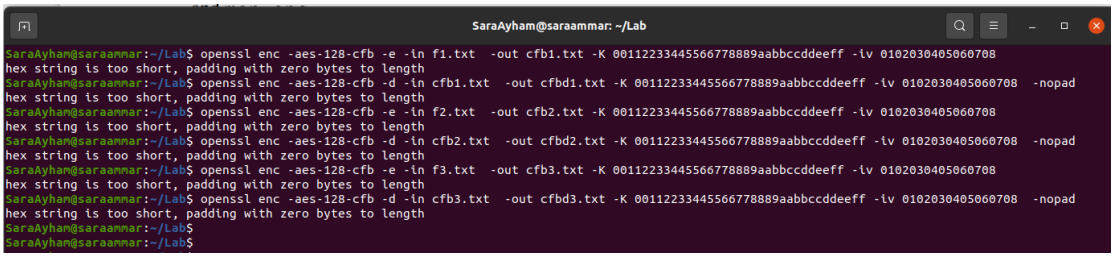

Figure 21: Encryption files using CFB

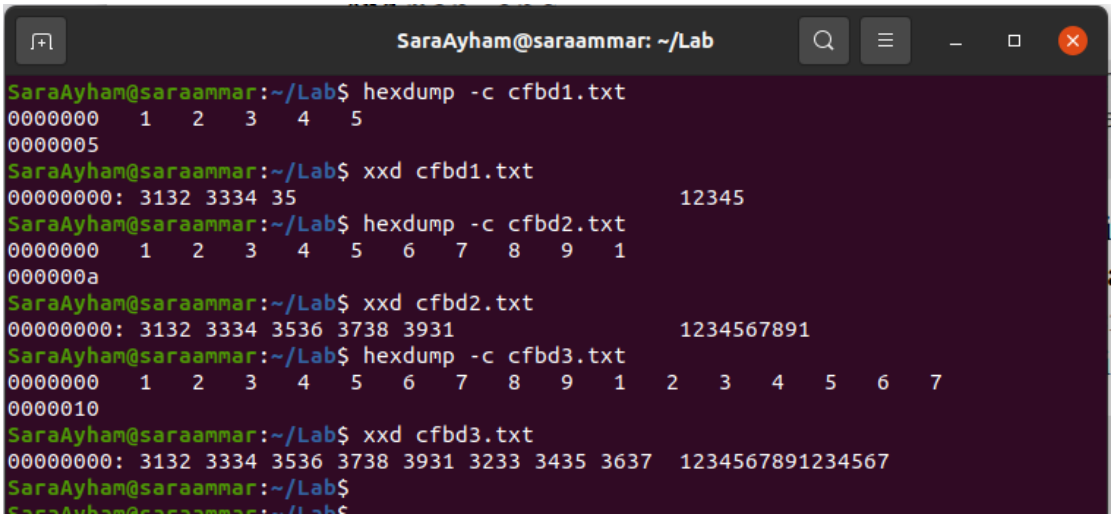

Figure 22: Encryption result using CFB

4. Encryption files using OFB:

Since OFB does not have padding, as illustrated in the figure below, it is the same concept as CFB in terms of padding.

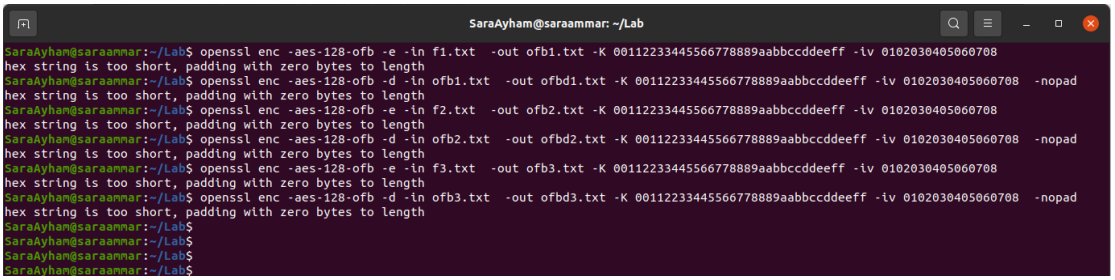

Figure 23: Encryption files using OFB

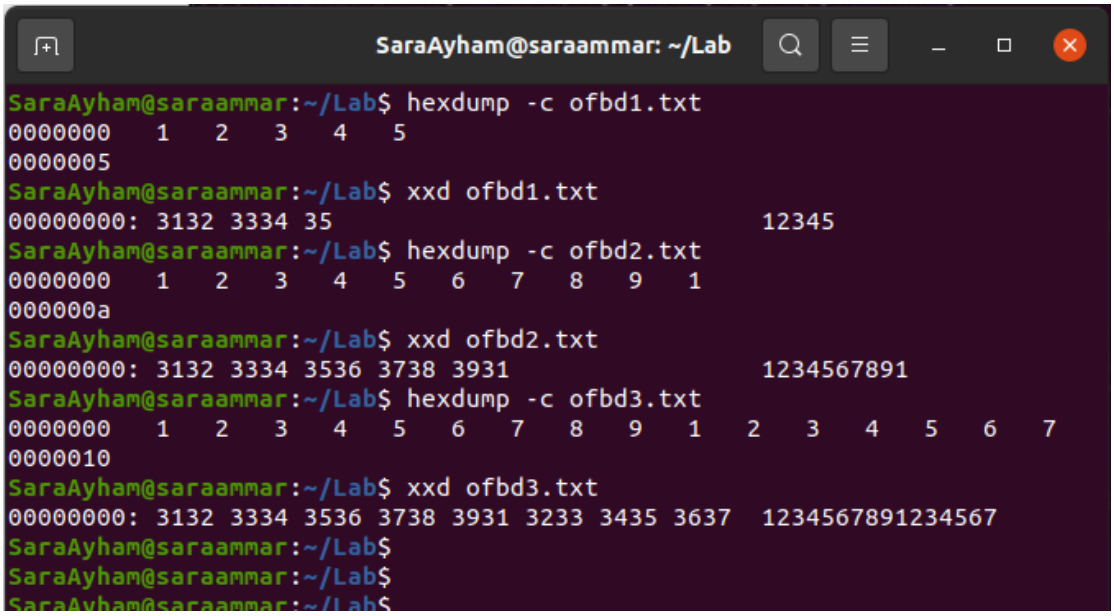

Figure 24: Encryption result using OFB

Task 5: Error Propagation – Corrupted Cipher Text

In this task, the error proration property of encryption modes have been understand using the following steps:

1. The text file was created with size 1.1 Kb, as shown below:

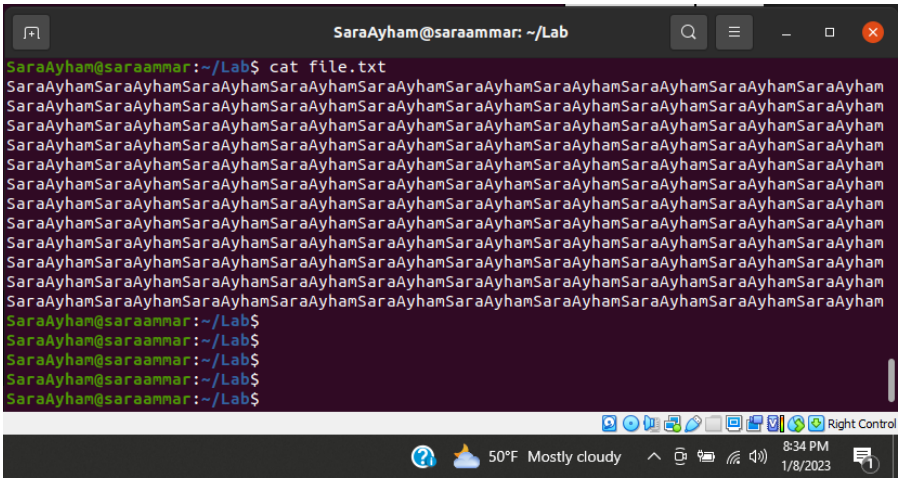

Figure 25: Creating file with 1.1 Kb size

2. The file has been encrypted using AES-128 cipher with four different modes, as shown below:

a. Using ECB:

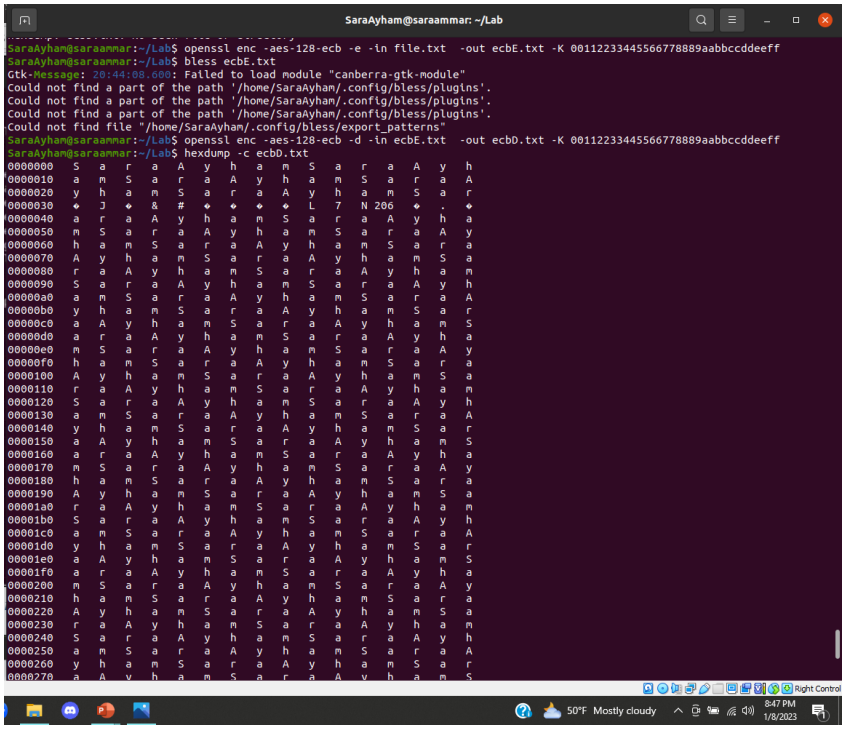

Figure 26: encryption and decryption using AES-128 ECB

After this step, the 55th byte in the encrypted file was edited, and the corrupted byte was achieved using bless hex editor, then the corrupted cipher text was decrypted using the correct key and IV, and the result was as shown in figure below: (note that the result of decryption is in the figure above). In ECB, the block size completely has been corrupted.

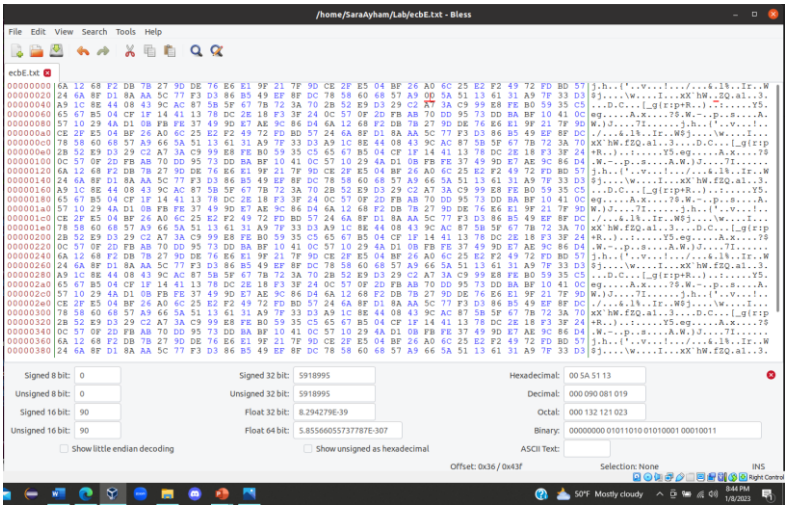

Figure 27: Edit 55th byte of the cipher text with ECB

b. Using CBC:

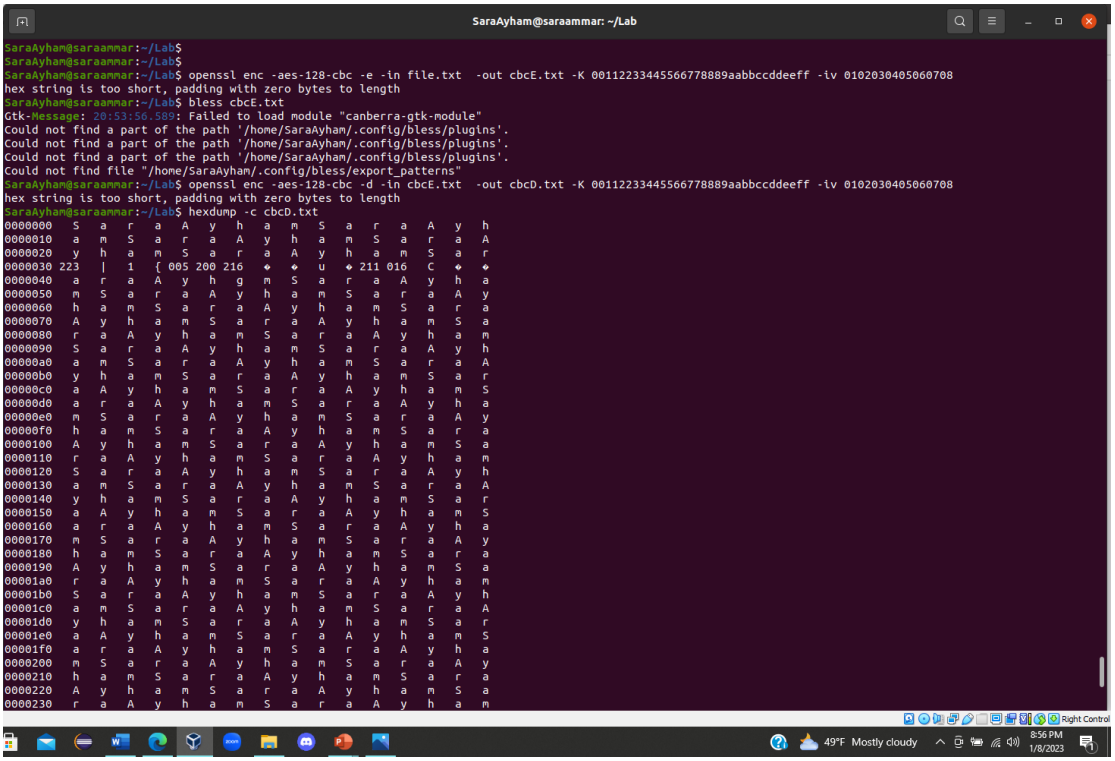

Figure 28: Encryption and decryption using AES-128 CBC

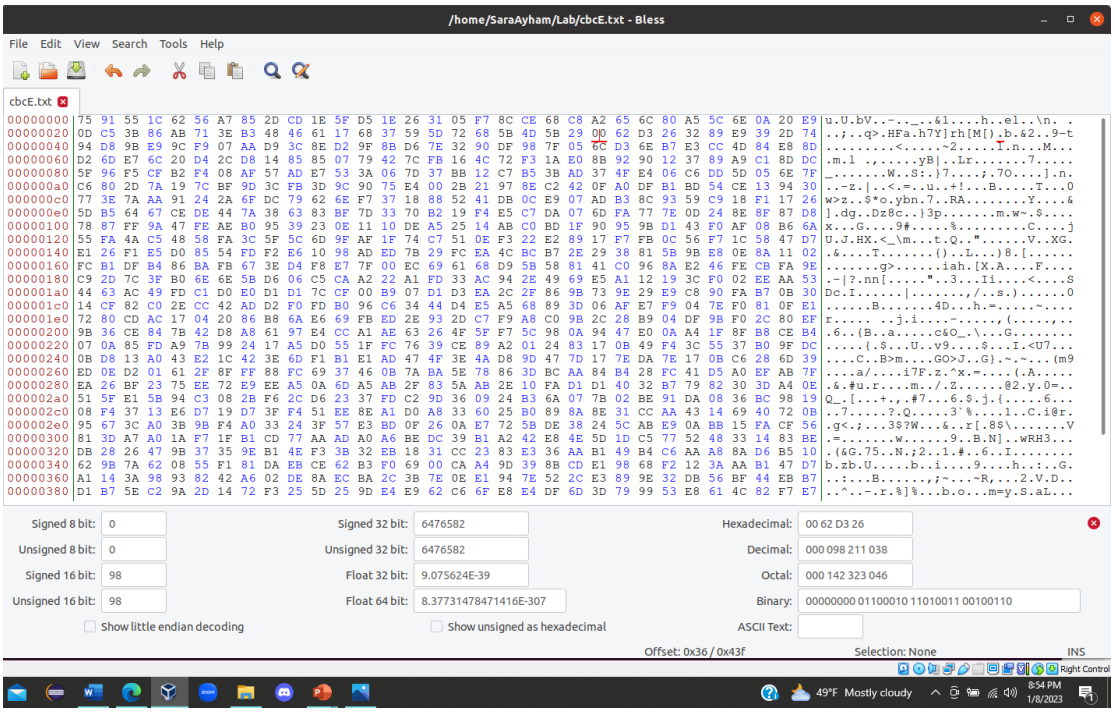

Figure 29: Edit 55th byte of the cipher text with CBC

c. Using CFB:

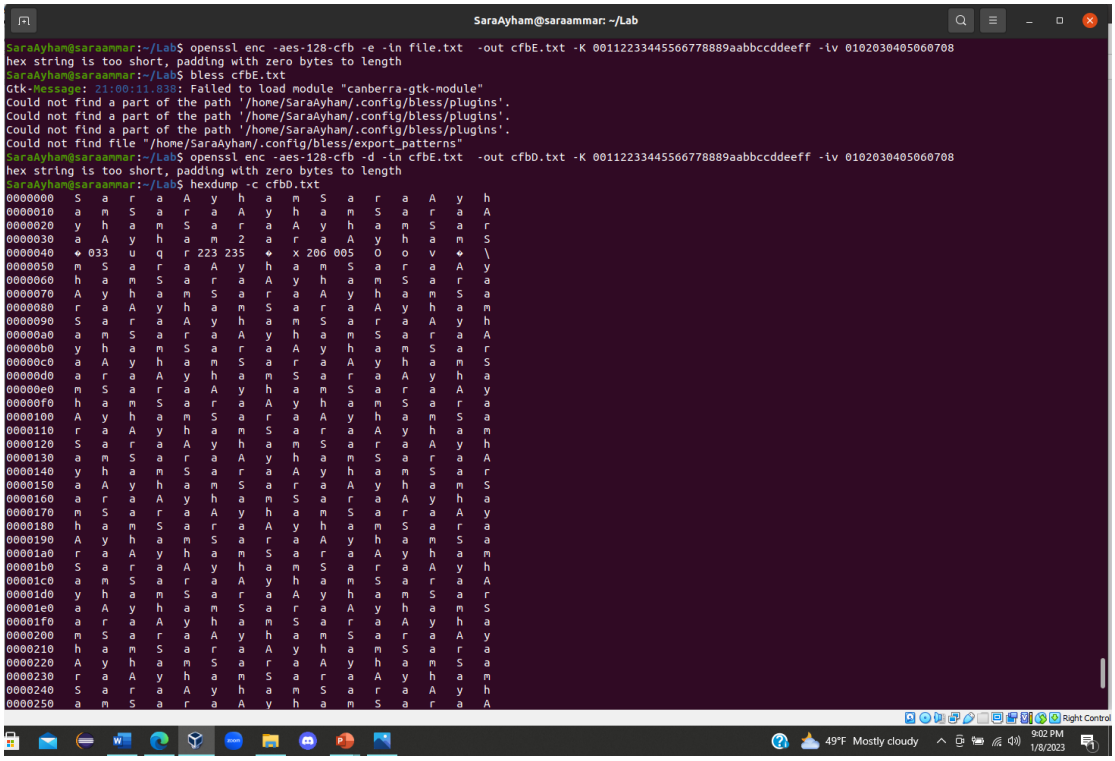

Figure 30: Encryption and decryption using AES-128 CFB

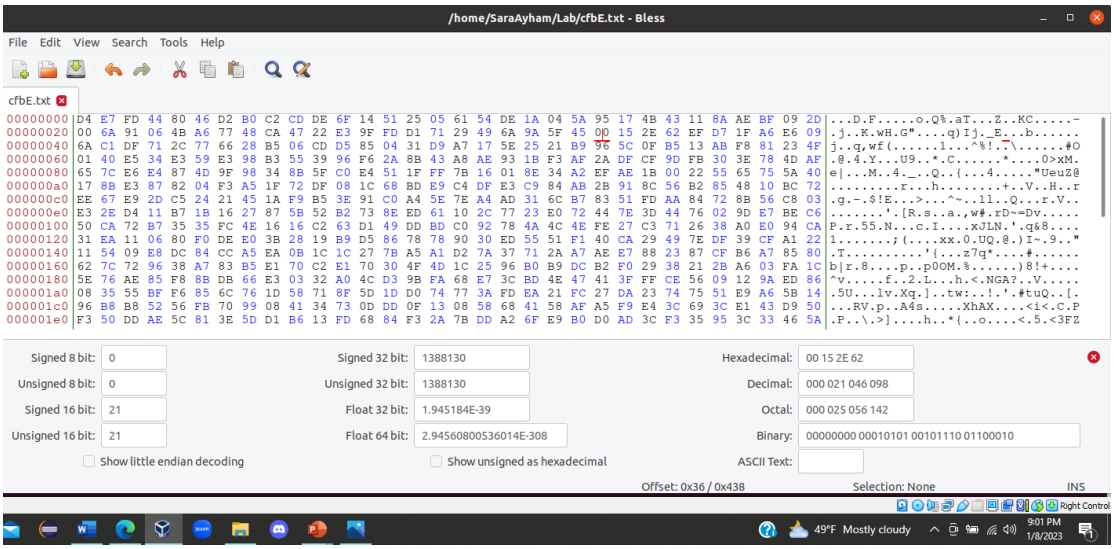

Figure 31: Edit 55th byte of the cipher text with CFB

Also in CBC and CFB, the block size (16 Bytes) has been corrupted.

d. Using OFB:

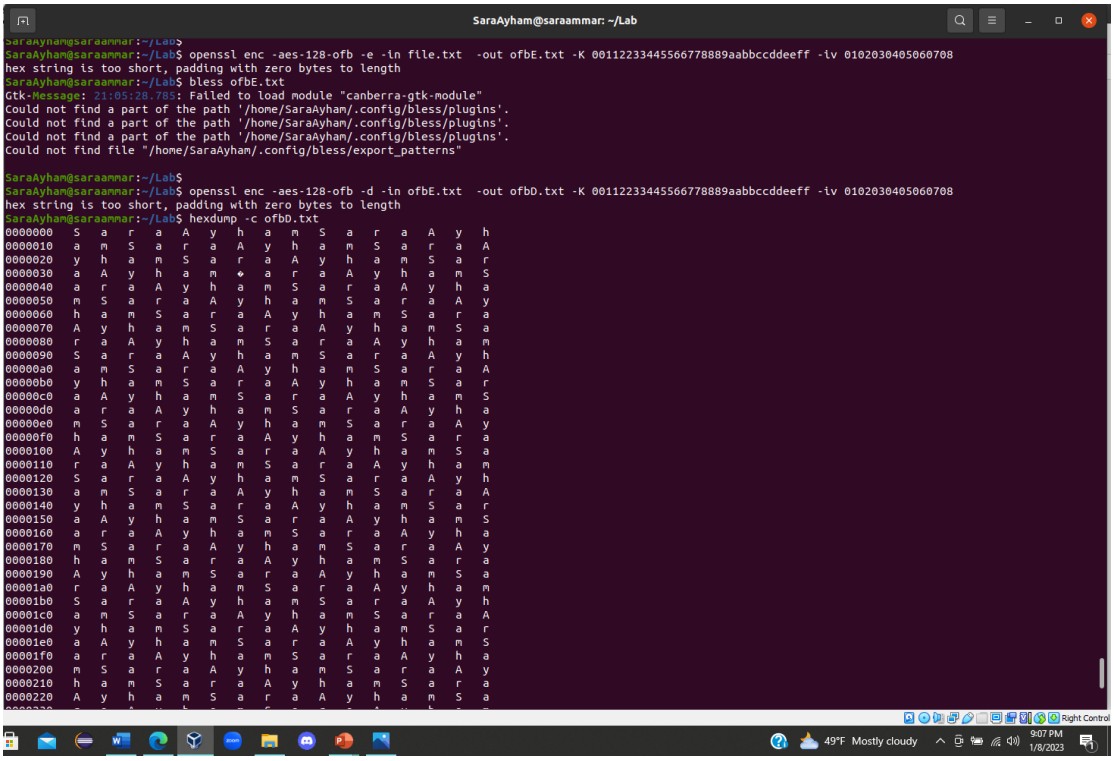

Figure 32: Encryption and decryption using AES-128 OFB

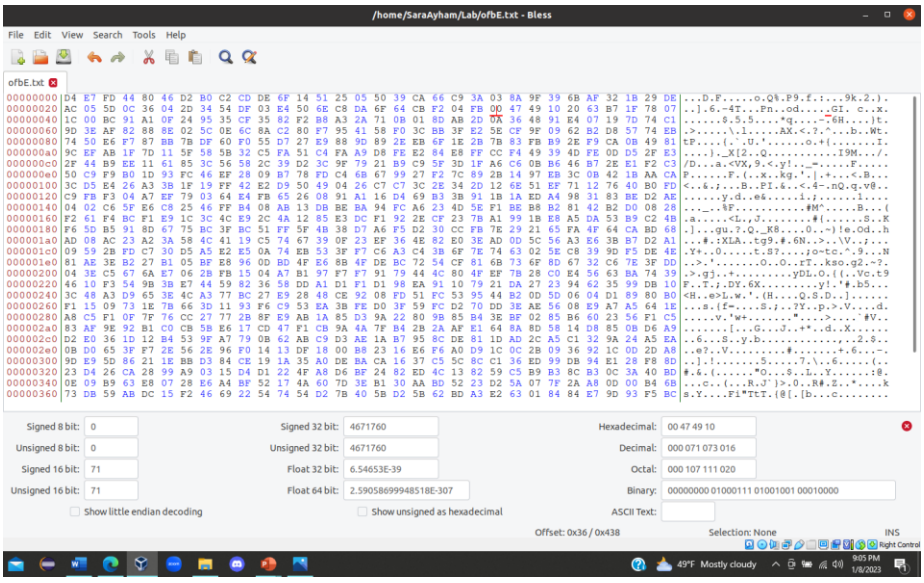

Figure 33: Edit 55th byte of the cipher text with OFB

In OFB, the corrupted cipher text is only affected by the corrupted byte and not by any other changes.

Task 6: Initial Vector (IV) and Common Mistakes

The objective in this task is to understand the problems if an IV is not selected properly.

6.1. IV Experiment

Encrypt the same plaintext using two different IVs:

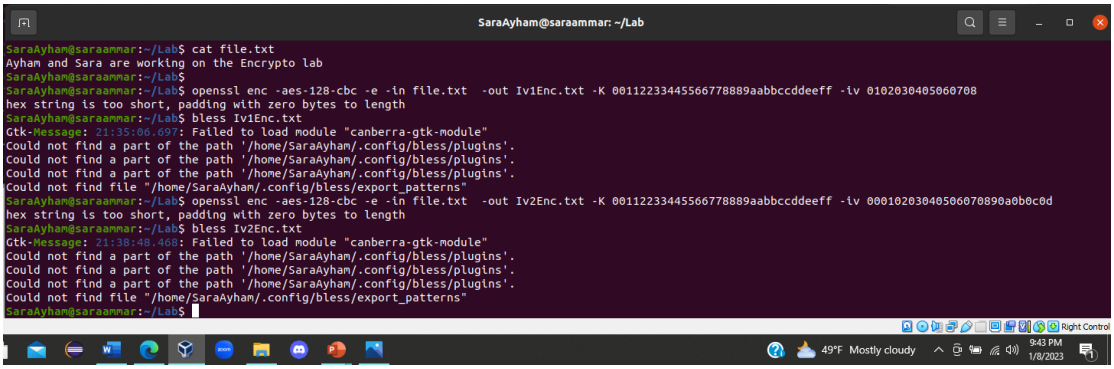

Figure 34: Encrypting the same plaintext using two different IVs

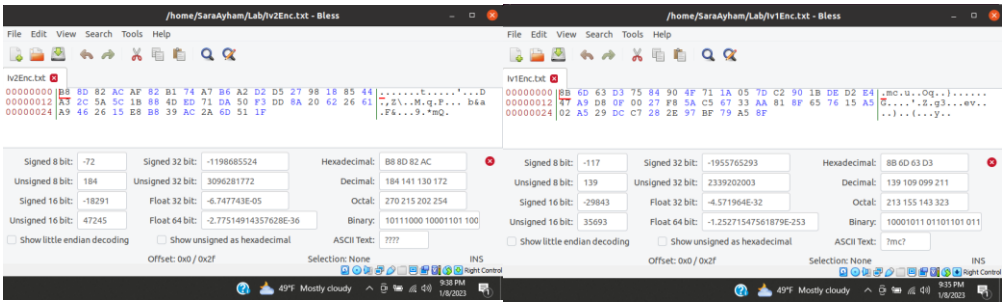

Figure 35: The resulting ciphertexts from using two different IVs

We note from the results mentioned above that the resulting encrypted texts differ from each other, although the encrypted texts are identical, and the reason for this difference is the difference in the IVs.

Encrypt the same plaintext using same IV:

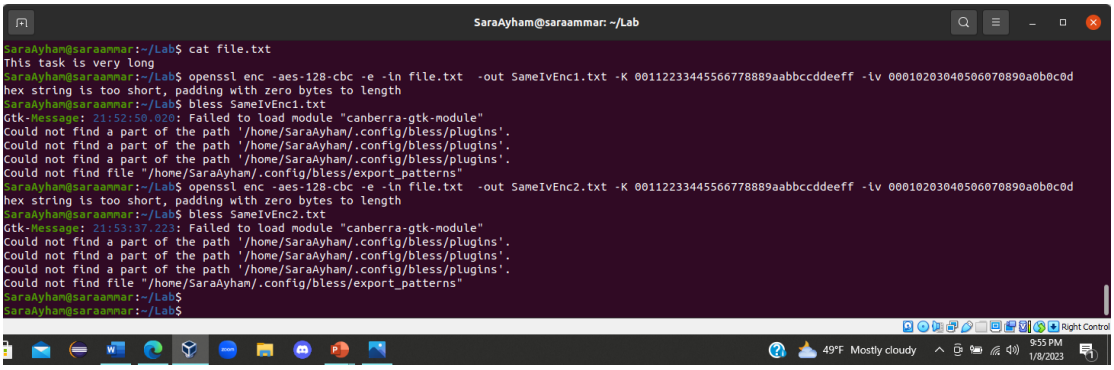

Figure 36: Encrypting the same plaintext using the same IV

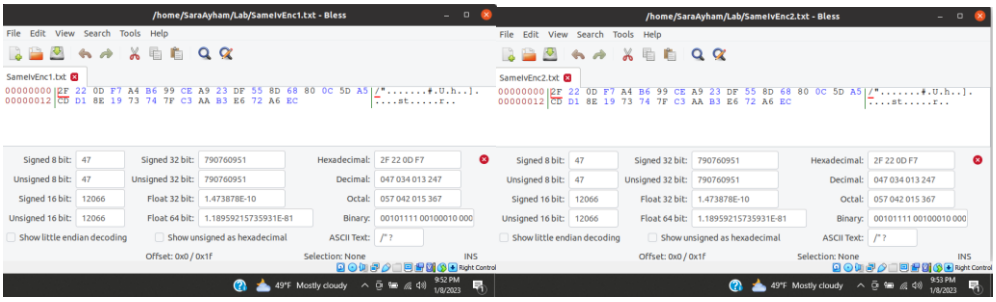

Figure 37: The resulting ciphertexts from using the same IV

We note from the above results that the resulting ciphertexts are similar, and the reason for this is that the same IV is used in the cipher process for the same text.

When the same IV is used with the same plaintext, they yield the same cipher text. Once a partis cracked, it makes it easier to follow the other plain text. This is a major flaw to security. So, same IV's must be avoided.

6.2. Common Mistake: Use the Same IV

Here we will discuss the issue that if the plaintext is not repeated, is it safe to use the same IV?

We assumed that the attacker got plaintext (P1), ciphertext (C1) and ciphertext (C2), and the encryption process was done using the same IV:

Plaintext (P1): This is a known message!

Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

Plaintext (P2): (unknown to you)

Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159

By formula,

$$P1 \text{ XOR } P2 = C1 \text{ XOR } C2$$

$$So, P2 = C1 \text{ XOR } C2 \text{ XOR } P1$$

We have been able to find out the actual content of P2 based on C2, P1 and C1.

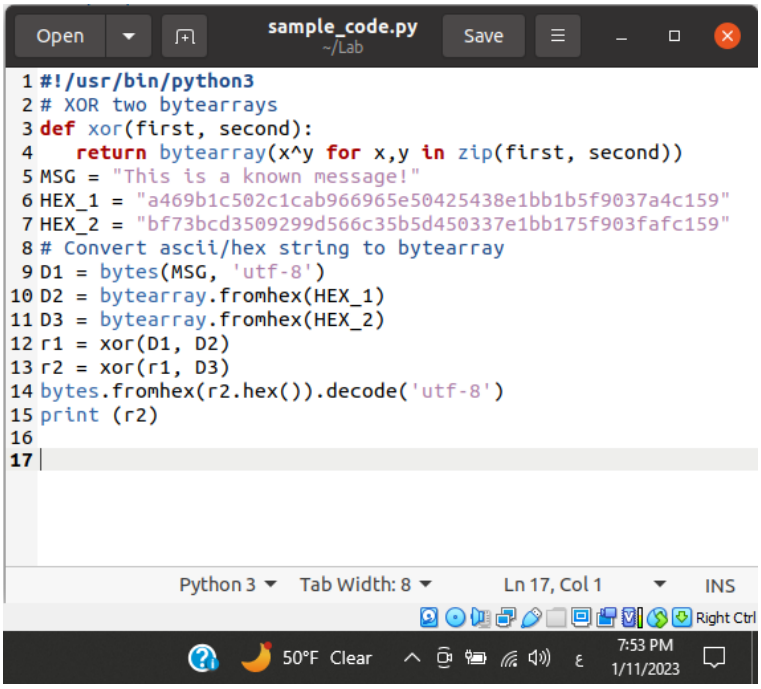

Figure 38: Sample_code to find P2 content

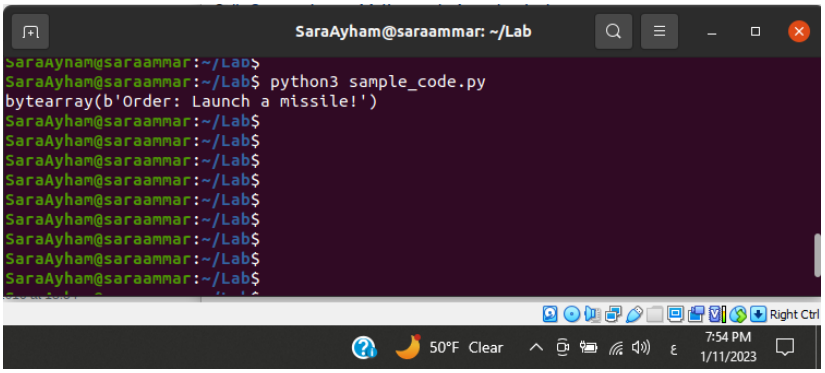

Figure 39: P2 Content

P2 String =Order: Launch a missile!

So, if the plaintext is not repeated, it is not safe to use the same IV.

6.3. Common Mistake: Use a Predictable IV

This task is more inclined towards chosen-plaintext attack than known-plaintext attack. The IV that are used are different but predictable.

By formula,

P2Bob= (IV2 xor IV1 xor P1Bob)

Since, we know the IV1, IV2 and the P1 of Bob, the formula can be used to find the P2 that would be sent by Bob since the IV is predictable.

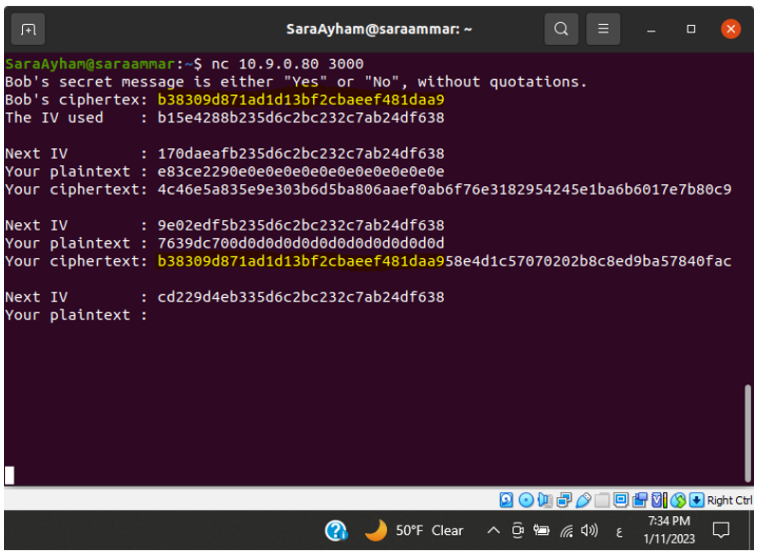

Figure 40: P2 Bob message

From the above results, it turns out that Bob's first message is yes, because the ciphertext output for (IV1 xor IVnext xor P(yes)) is the same ciphertext output for the first message.

So, using predictable IV is not secure.

Task 7: Programming using the Crypto Library

In this task, we have a given plaintext and a ciphertext, our job is to find the key that is used for the encryption. We know that the aes-128-cbc cipher is used for the encryption, and the key used to encrypt this plaintext is an English word shorter than 16 characters from a typical English dictionary.

- Plaintext (total 21 characters): This is a top secret.
- Ciphertext (in hex): 764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2
- IV (in hex): aabbccddeeff00998877665544332211

We have written a C program to find the encryption key. See Appendix2

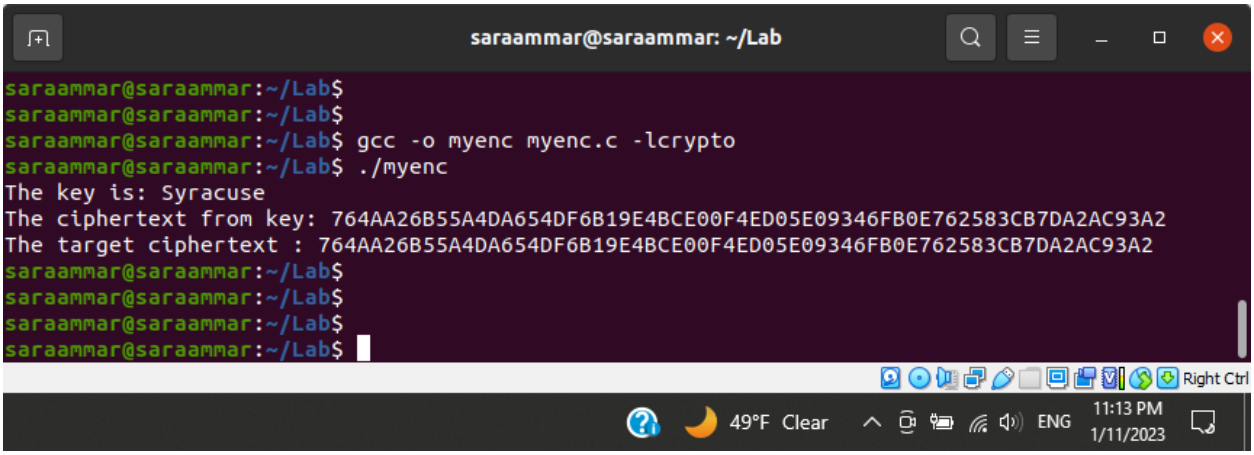

Figure 41: The result of the key

The key is: Syracuse (has number 22287 in the dictionary).

Conclusion

in conclusion, we learned how apply the theoretical information about encryption and decryption in secure-key encryption to get familiar with these concepts, there various ways used to learn the difference and security between each other , we faced an small problems in the last task ; because the task wasn't easy to be understood, after the task was understanded and analyzed the task the solution was easy, this lab was very important for our, cause we gained an small expertise of how encrypt and decrypt the messages with unique keys.

Appendix

1. The plaintext before decryption:

```
#!/bin/env python3

Cipher = []
repetition = {}

fileName = open('/home/saraammar/Lab/ciphertext.txt', '+r')
for line in fileName:
    for element in line.split(' '):
        Cipher.append(element.strip())
        Cipher = list(filter(None, Cipher))

buffer = {}

for word in Cipher:
    for character in word:
        if character not in buffer.keys():
            buffer[character] = 1
        else:
            buffer[character] += 1

repetition = dict(sorted(buffer.items(), key=lambda x:x[1], reverse=True))


k = 0
for word in Cipher:
    freq = list(word)
    j = 0
    for character in word:
        if character == 'n':
            freq[j] = 'E'
        elif character == 't':
            freq[j] = 'H'
        elif character == 'y':
            freq[j] = 'T'
        elif character == 'v':
             freq[j] = 'A'
        elif character == 'x':
            freq[j] = 'O'
        elif character == 'u':
            freq[j] = 'N'
        elif character == 'b':
            freq[j] = 'F'
        elif character == 'p':
            freq[j] = 'D'
        elif character == 'g':
```

```
            freq[j] = 'B'
        elif character == 'h':
            freq[j] = 'R'
        elif character == 'm':
            freq[j] = 'I'
        elif character == 'i':
            freq[j] = 'L'
        elif character == 'q':
            freq[j] = 'S'
        elif character == 's':
            freq[j] = 'K'
        elif character == 'f':
            freq[j] = 'V'
        elif character == 'z':
            freq[j] = 'U'
        elif character == 'd':
            freq[j] = 'Y'
        elif character == 'c':
            freq[j] = 'M'
        elif character == 'r':
            freq[j] = 'G'
        elif character == 'e':
            freq[j] = 'P'
        elif character == 'a':
            freq[j] = 'C'
        elif character == 'l':
            freq[j] = 'W'
        elif character == 'k':
            freq[j] = 'X'
        elif character == 'o':
            freq[j] = 'J'
        elif character == 'j':
            freq[j] = 'Q'
        elif character == 'w':
            freq[j] = 'Z'
        else:
            freq[j] = character
        j += 1
    tempString = "".join(freq)
    Cipher[k] = tempString
    k += 1

for i in Cipher:
    print(i, end =" ")
```

2. C code:

```
// main.cpp
#include <openssl/aes.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>

#define charMaxLeng 20


unsigned char AscToHex(unsigned char Char){
        int aChar = (int)Char;

   if((aChar>=0x30)&&(aChar<=0x39))
      aChar -= 0x30;
   else if((aChar>=0x41)&&(aChar<=0x46))
      aChar -= 0x37;
   else if((aChar>=0x61)&&(aChar<=0x66))
      aChar -= 0x57;
   else
      aChar = 0xff;

   return aChar;
}


unsigned char HexToAsc(unsigned char aHex){
   if((aHex>=0)&&(aHex<=9))
      aHex += 0x30;
   else if((aHex>=10)&&(aHex<=15))//A-F
      aHex += 0x37;
   else
      aHex = 0xff;

   return aHex;
}

unsigned char* str2hex(char *str) {
   unsigned char *ret = NULL;
   int str_len = strlen(str);
   int i = 0;
   // printf("%d \n", str_len);
   // printf("%s\n", str);
   assert((str_len%2) == 0);
   ret = (char *)malloc(str_len/2);
   for (i =0;i < str_len; i = i+2 ) {
      sscanf(str+i,"%2hhx",&ret[i/2]);
```

```
    }
    return ret;
}


char *padding_buf(char *buf,int size, int *final_size) {
    char *ret = NULL;
    int pidding_size = AES_BLOCK_SIZE - (size % AES_BLOCK_SIZE);
    int i;

    *final_size = size + pidding_size;
    ret = (char *)malloc(size+pidding_size);
    memcpy( ret, buf, size);
    if (pidding_size!=0) {
        for (i =size;i < (size+pidding_size); i++ ) {
            ret[i] = pidding_size;
        }
    }

    return ret;
}

void printf_buff(char *buff,int size) {
    int i = 0;
    for (i=0;i<size;i ++ ) {
        // printf( "%02X ", (unsigned char)buff[i] );
                    printf( "%02X", (unsigned char)buff[i] );
        if ((i+1) % 8 == 0) {
            // printf("\n");
        }
    }

    printf("\n");
}

void encrpyt_buf(char *raw_buf, char **encrpy_buf, int len, char* kkey) {
    AES_KEY aes;
    unsigned char *key = str2hex(kkey);
    unsigned char *iv = str2hex("aabbccddeeff00998877665544332211");
    AES_set_encrypt_key(key,128,&aes);
    AES_cbc_encrypt(raw_buf,*encrpy_buf,len,&aes,iv,AES_ENCRYPT);

    free(key);
    free(iv);
}

void decrpyt_buf(char *raw_buf, char **encrpy_buf, int len, char* kkey) {
    AES_KEY aes;
    unsigned char *key = str2hex(kkey);
```

```
    unsigned char *iv = str2hex("aabbccddeeff00998877665544332211");
    AES_set_decrypt_key(key,128,&aes);
    AES_cbc_encrypt(raw_buf,*encrpy_buf,len,&aes,iv,AES_DECRYPT);

    free(key);
    free(iv);
}

int main(int argc, char* argv[]) {
        char*
target="764AA26B55A4DA654DF6B19E4BCE00F4ED05E09346FB0E762583CB7DA2AC93A2";
        FILE* p=NULL;
        if((p=fopen("words.txt","r"))==NULL)
        {
                printf("ERROR");
        }

        char buffer[charMaxLeng];
        char buf2[charMaxLeng];
        int flag=0;

        while (!feof(p)){
                int i=0;
                memset(buffer,'\0', charMaxLeng * sizeof(char));
                memset(buf2,'\0', charMaxLeng * sizeof(char));
                fgets(buffer, charMaxLeng, p);
                while(i<charMaxLeng){
                        buf2[i]=buffer[i];
                        i+=1;
                }
                size_t len = strlen(buffer);
                if (len == 1)        continue;
                // printf("%s", buffer);


            char *raw_buf = NULL;
            char *after_padding_buf = NULL;
            int padding_size = 0;
            char *encrypt_buf = NULL;
            char *decrypt_buf = NULL;

            i=0;
            unsigned char* key=NULL;
            key=(unsigned char*)malloc(33);

            while(i < strlen(buffer)){
                    unsigned char letter = buffer[i];

                    key[2*i] = HexToAsc(letter/0x10);
```

```
            key[2*i+1] = HexToAsc(letter%0x10);
            // printf("%d\n", i);

            ++i;
            if(i==0x0f || buffer[i] < 0x20)
                        break;
    }
    while(i < 0x10){
            key[2*i] = '2';
            key[2*i+1] = '3';
            ++i;
    }
    key[0x20]='\0';
   //  printf("%s\n", key);
    raw_buf = (char *)malloc(21);
    memcpy(raw_buf,"This is a top secret.",21);
    after_padding_buf = padding_buf(raw_buf,21,&padding_size);

    encrypt_buf = (char *)malloc(padding_size);
    encrpyt_buf(after_padding_buf,&encrypt_buf, padding_size, key);

    // printf("%d\n", strlen(target));
    i=0;
    char temp='\0';
    flag=1;
    while(i<padding_size){
            temp = HexToAsc((unsigned char)encrypt_buf[i]/0x10);
            if(temp!=target[2*i]){
                        flag=0;
                        break;
                        }
            temp = HexToAsc((unsigned char)encrypt_buf[i]%0x10);
            if(temp!=target[2*i+1]){
                        flag=0;
                        break;
                        }
            i+=1;
    }
    if(flag==0){
                        continue;
            }
            printf("The key is: %s", buf2);
            printf("The ciphertext from key: ");
            printf_buff(encrypt_buf,padding_size);
            printf("The target ciphertext : %s\n", target);

    free(raw_buf);
    free(after_padding_buf);
    free(encrypt_buf);
```

```
            free(decrypt_buf);
            // printf("%02X\n", (unsigned char)('T'));
            break;
        }
    fclose(p);
    return 0;
}
```