CSE565 Lab 1

Name: Sri Charan Reddy Teegala

Email: teegala@buffalo.edu

UBID: teegala

UB Number: 50681752

Before You Start:

Please write a detailed lab report, with **screenshots**, to describe what you have **done** and what you have **observed**. You also need to provide **explanation** to the observations that you noticed. Please also show the important **code snippets** followed by explanation. Simply attaching code without any explanation will NOT receive credits.

After you finish, export this report as a **PDF** file and submit it on UBLearns.

Academic Integrity Statement:

I, Sri Charan Reddy Teegala, have read and understood the course academic integrity policy.

Task 1: Frequency Analysis

Steps Performed:

1. Firstly, I ran the python script in the Files folder *freq.py* to fetch the single-letter frequencies, bigram frequencies and trigram frequencies of the letters in the file *ciphertext.txt*

2. Then I compared the relative frequencies of English alphabets (single, bigram and trigram) from sources mentioned in the Lab file with the output of *freq.py,* assumed few mappings, replaced them, and created a new file *out.txt*

3. After every iteration, I checked the words formed in the *out.txt* file and compared them with common words in English to get new mappings that we can add in the next iteration.

4. For the actual words, I used capital letters to mark the substitutions, I repeated these steps until entire *out.txt* file was capitalized, to get cipher key for decrypting the monoalphabetic cipher.

Screenshots:

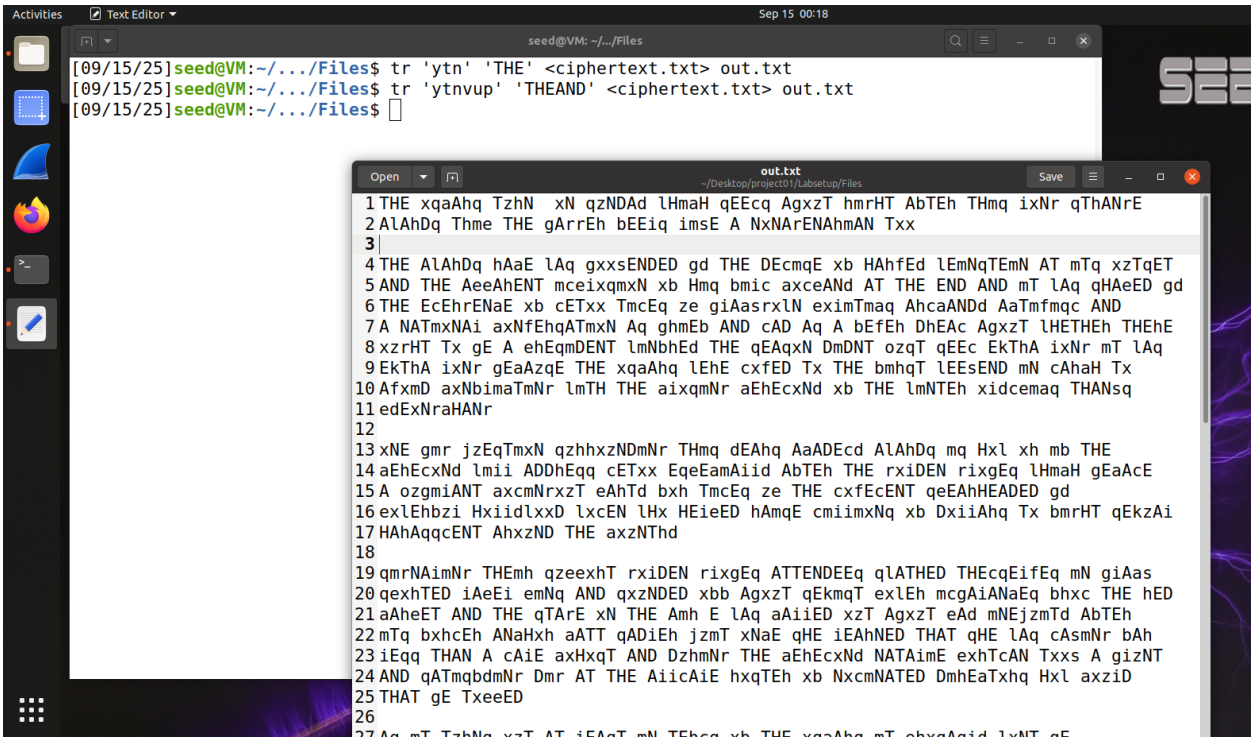


Figure (1.1)

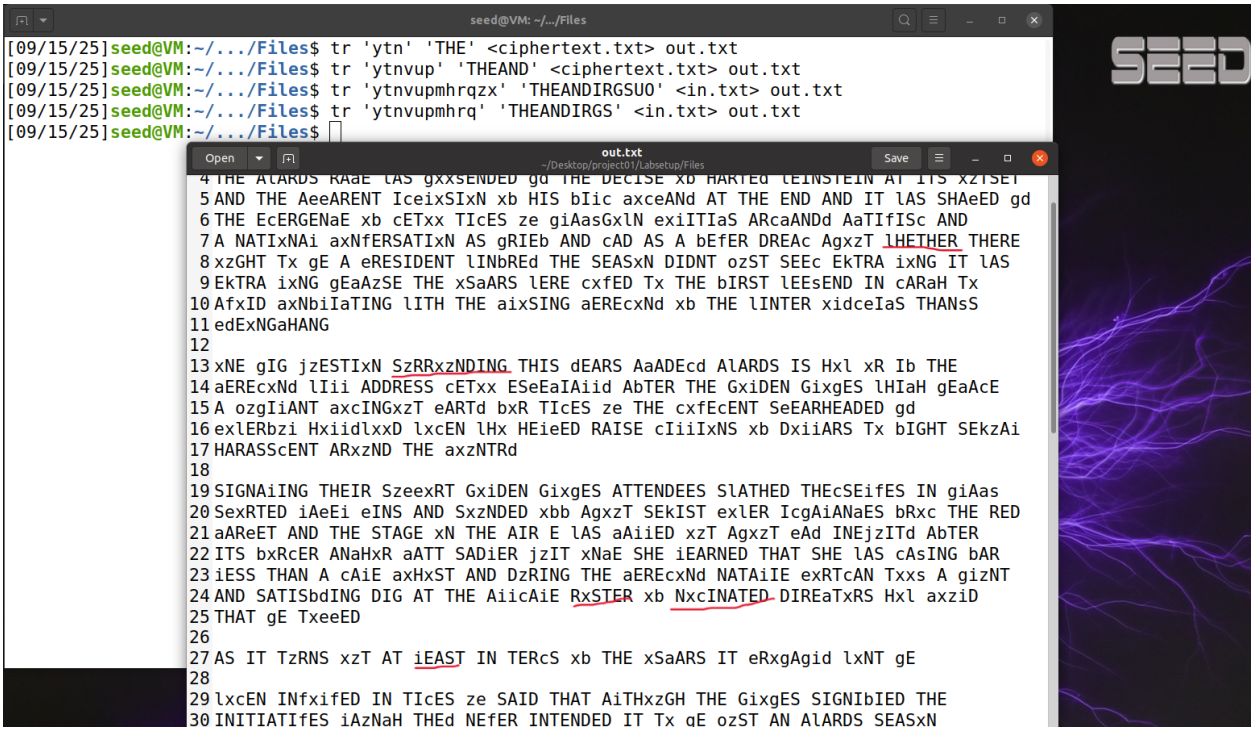


Figure (1.2)

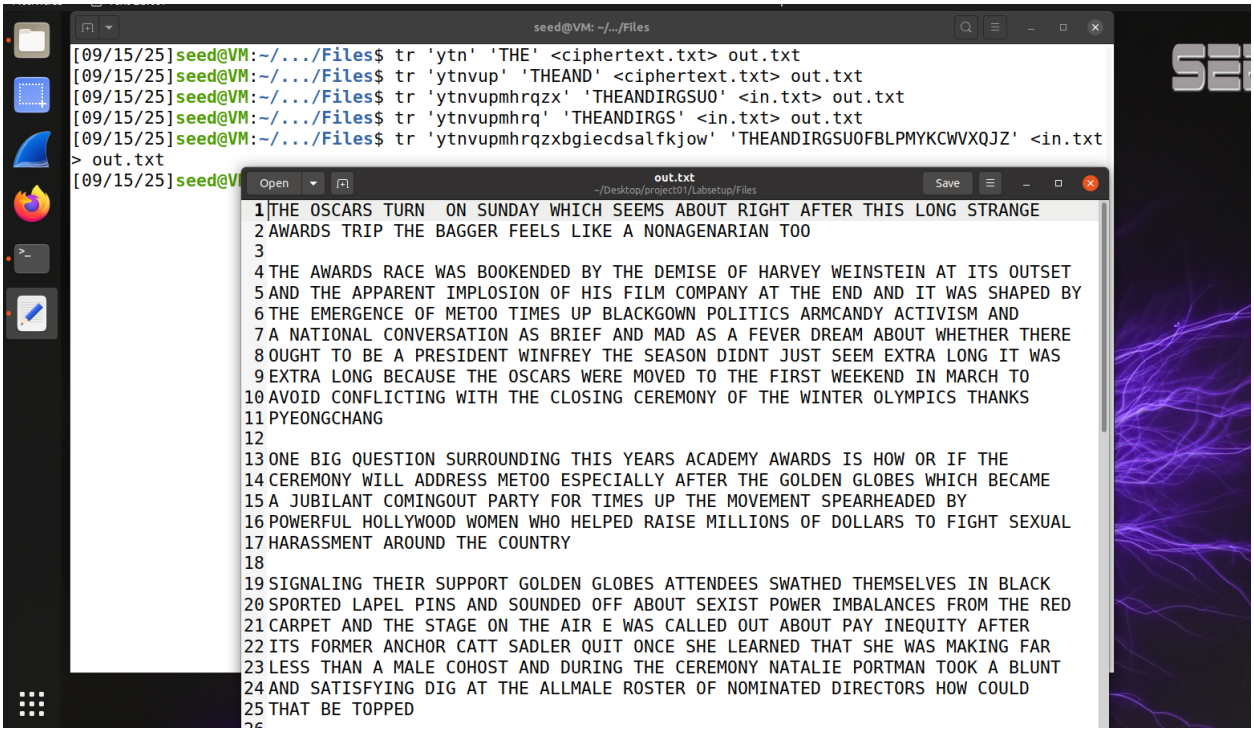


Figure (1.3)

Observations:

1. The most frequent trigram was *ytn* and the most frequent single letter was n from which we can conclude that *n -> e* and *ytn -> the*. Using tr command as shown in *Figure (1.1)* we marked the substitution.

2. After few iterations we could see some known words like '*SzRRxzNDING'* which could probably be *SURROUNDING* after decryption forming in *out.txt* in *Figure (1.2)* which we can use to find new mappings.

3. By substituting all the encrypted letters and adding the mappings to the *tr* command we could form cipher key '*ytnvupmhrqzxbgiecdsalfkjow'*

Code Snippets:

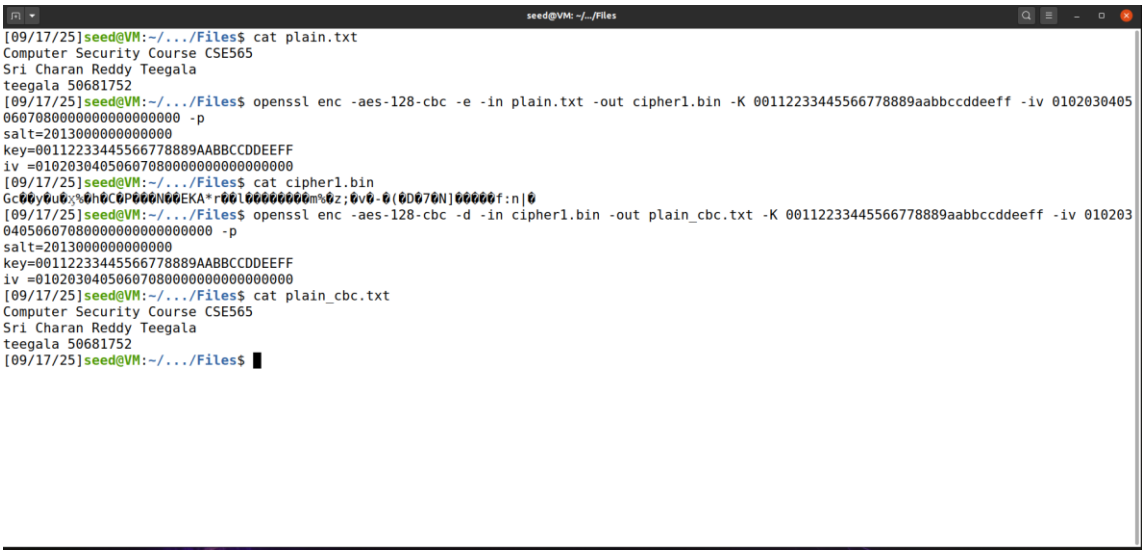
```
# Command for generating decrypted out.txt file from ciphertext.txt

tr 'ytnvupmhrqzxbgiecdsalfkjow' 'THEANDIRGSUOFBLPMYKCWVXQJZ' <ciphertext.txt> out.txt
```

Task 2: Encryption using Different Ciphers and Modes

Created a plain.txt file with some information and experimented with ***openssl enc*** command using different ciphers and modes.

1. **Encryption with AES-128-CBC**



Observations:

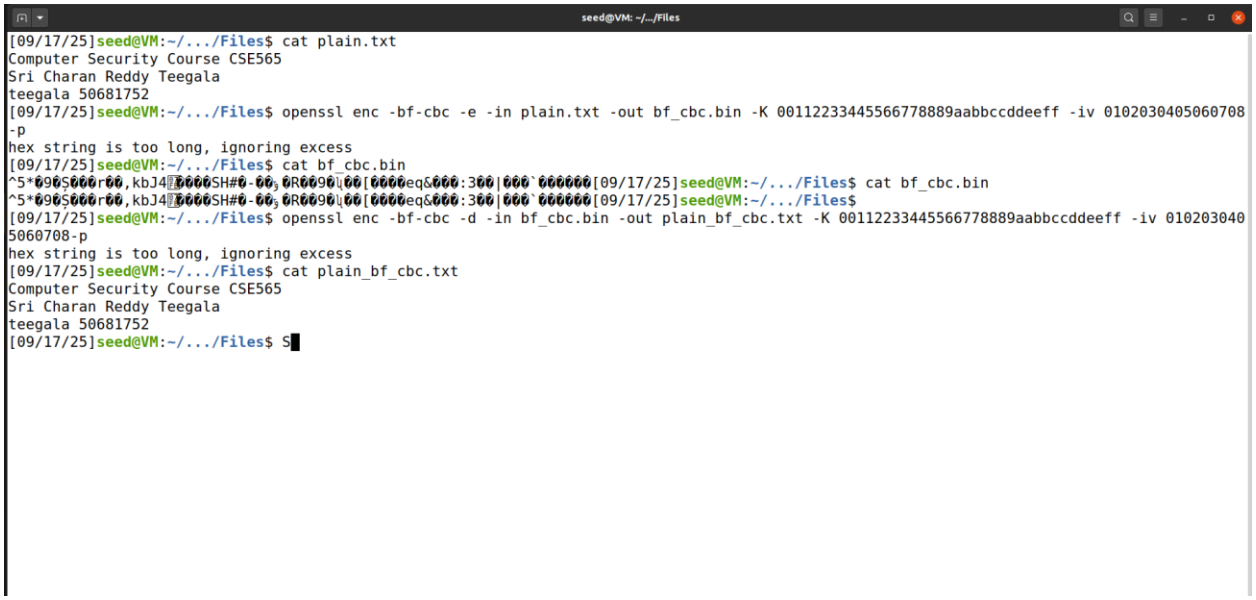
- AES-128-CBC requires both a 16-byte key and a 16-byte IV (padded IV with zeroes)
- Decryption restored the original file correctly into *plain_cbc.txt.*

2. **Encryption with AES-128-ECB**

Observations:

- AES-128-ECB uses a 16-byte key and does not require an IV.
- Decryption recovered the original content in plain.txt into plain_ecb.txt

3. Encryption with Blowfish (BF_CBC)



Observations:

- Blowfish uses an 8-byte IV that is the reason we see that error hex string is too long.
- Decryption restored the original plain.txt content into plain_bf_cbc.txt successfully.

Task 3: Encryption Mode – ECB vs. CBC

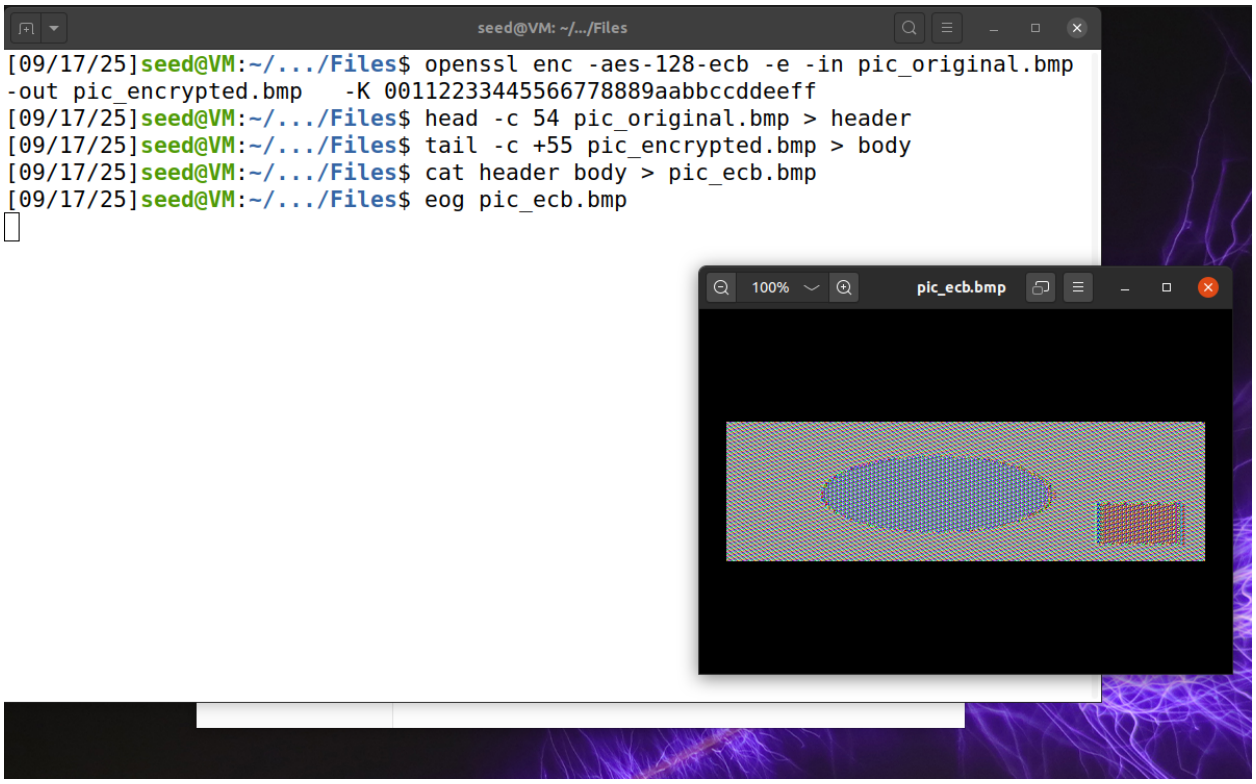
Steps Performed:

1. Encrypted the pic_original.bmp file using both ECB and CBC modes.
2. For the generated *pic_encrypted.bpm* file as mentioned we need to replace the header value with that of the original picture for retaining its properties.
3. Stored header from *pic_original.bpm* and the value of *pic_encrypted.bpm* without the header and combined them to get the *pic_ebc.bpm* and *pic_ecb.bpm* files for the modes ECB and CBC respectively.

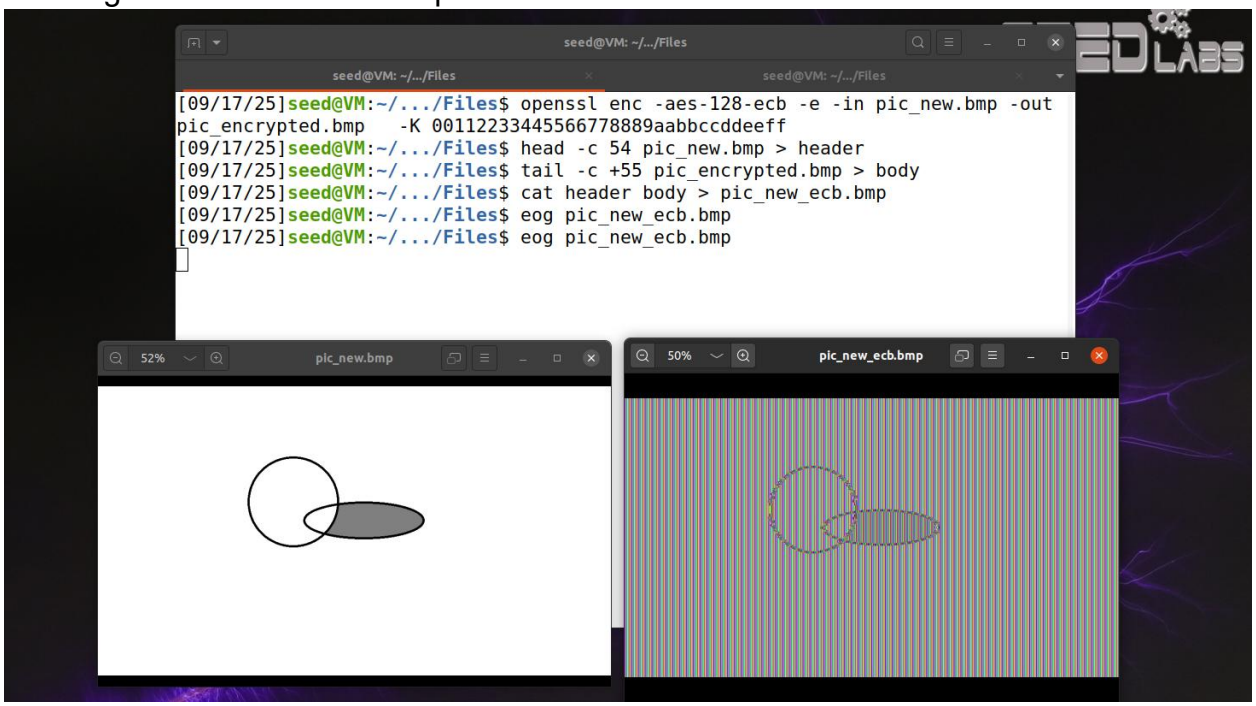
ECB Observations:

- The outline and the parameters of the original picture are still visible in the encrypted picture.

- This is because each 16-byte block is encrypted independently so identical blocks are encrypted into similar ciphertext blocks therefore the patterns remain.

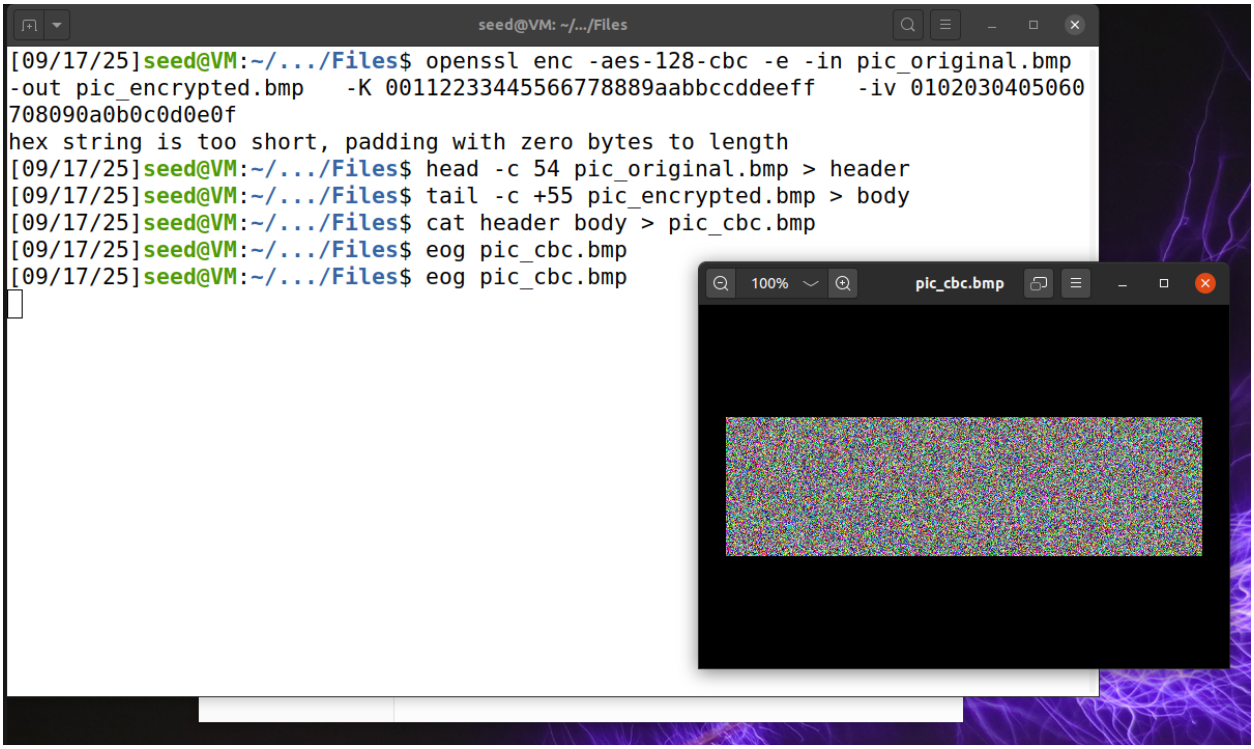


- Similarly, by taking a new picture and running the same steps to get encrypted image we can see that the patterns are retained.

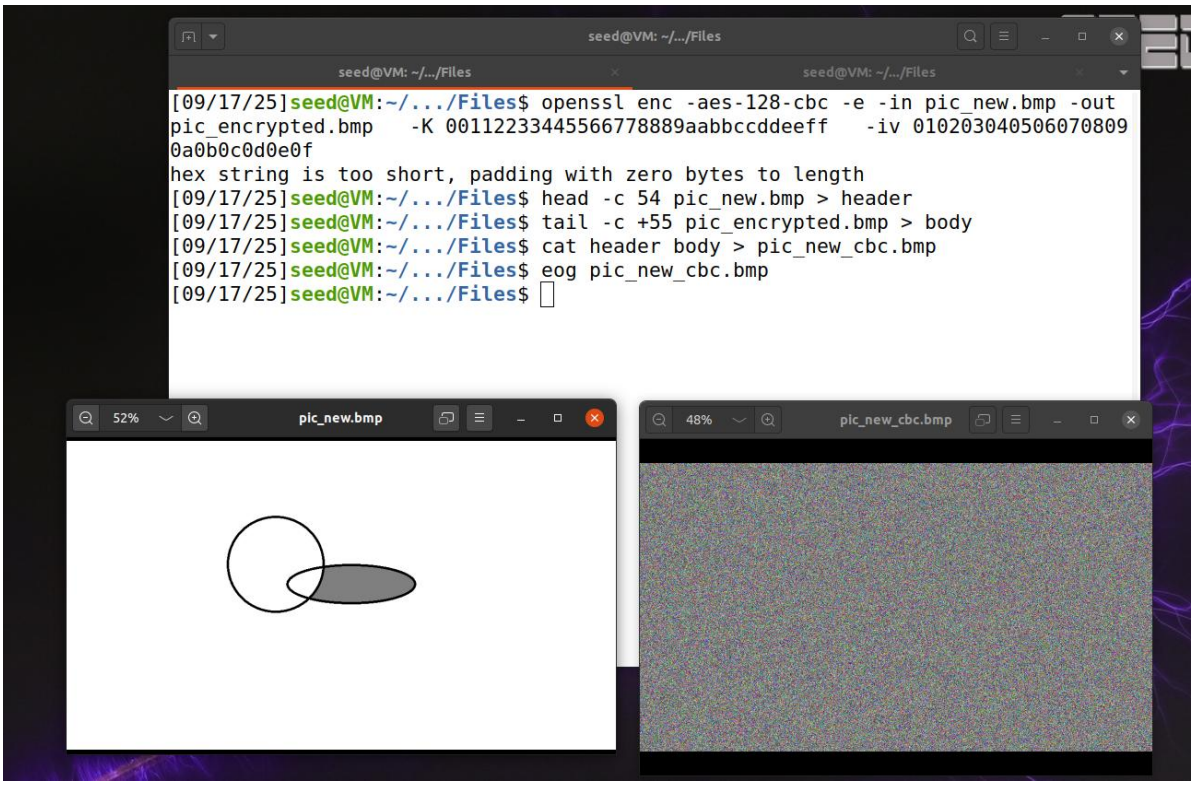


CBC Observations:

- The encrypted image looks completely random and noisy.
- No structures or patterns from the original picture are still visible.
- This is because CBC introduces sequential XOR therefore each ciphertext block will depend on the previous ciphertext block, so it will not repeat patterns.



- Similarly, by taking a new picture and running the same steps to get encrypted image using cbc we obtain an encrypted image with no repeated patterns.

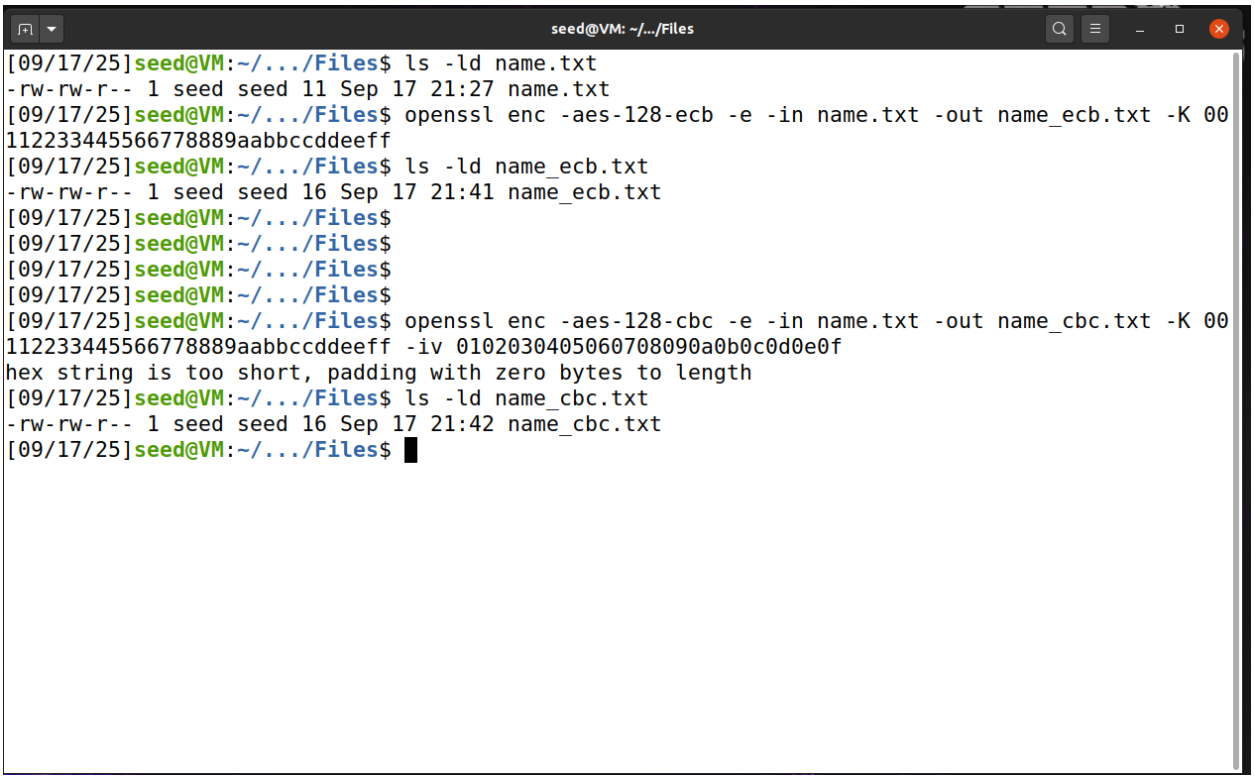


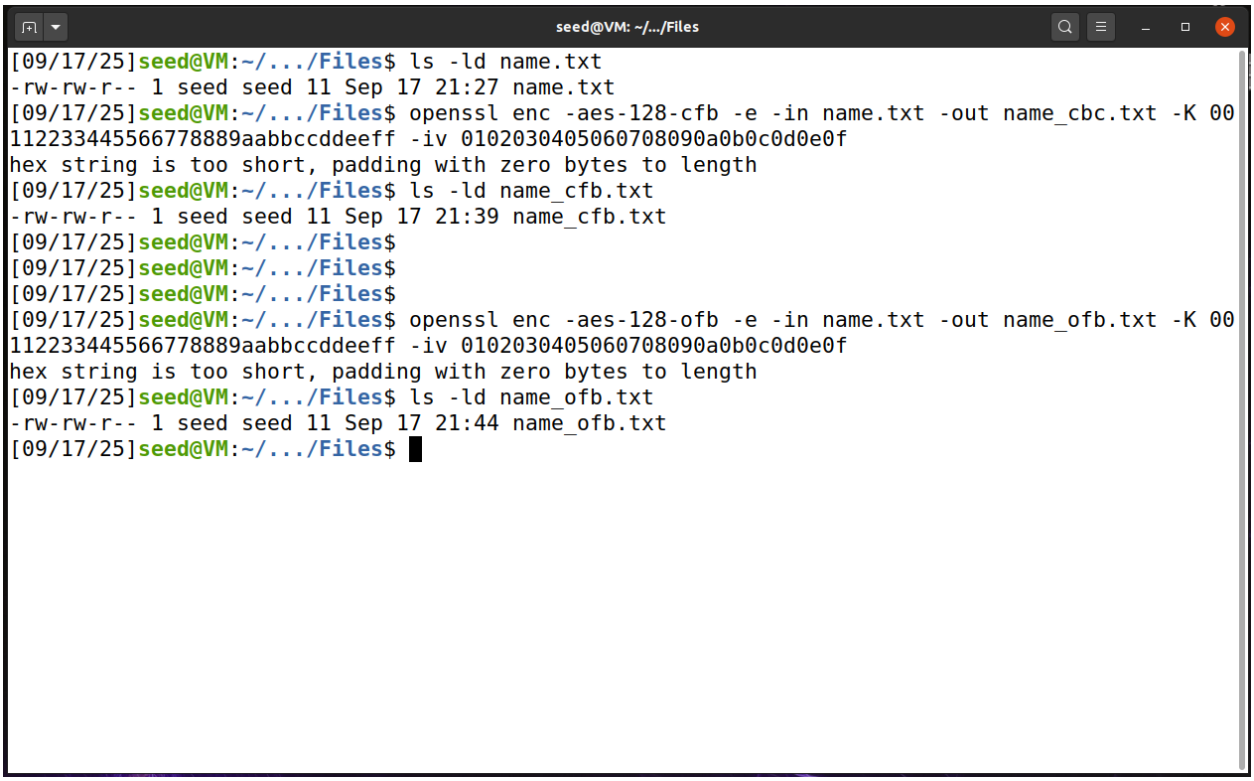
Task 4: Padding

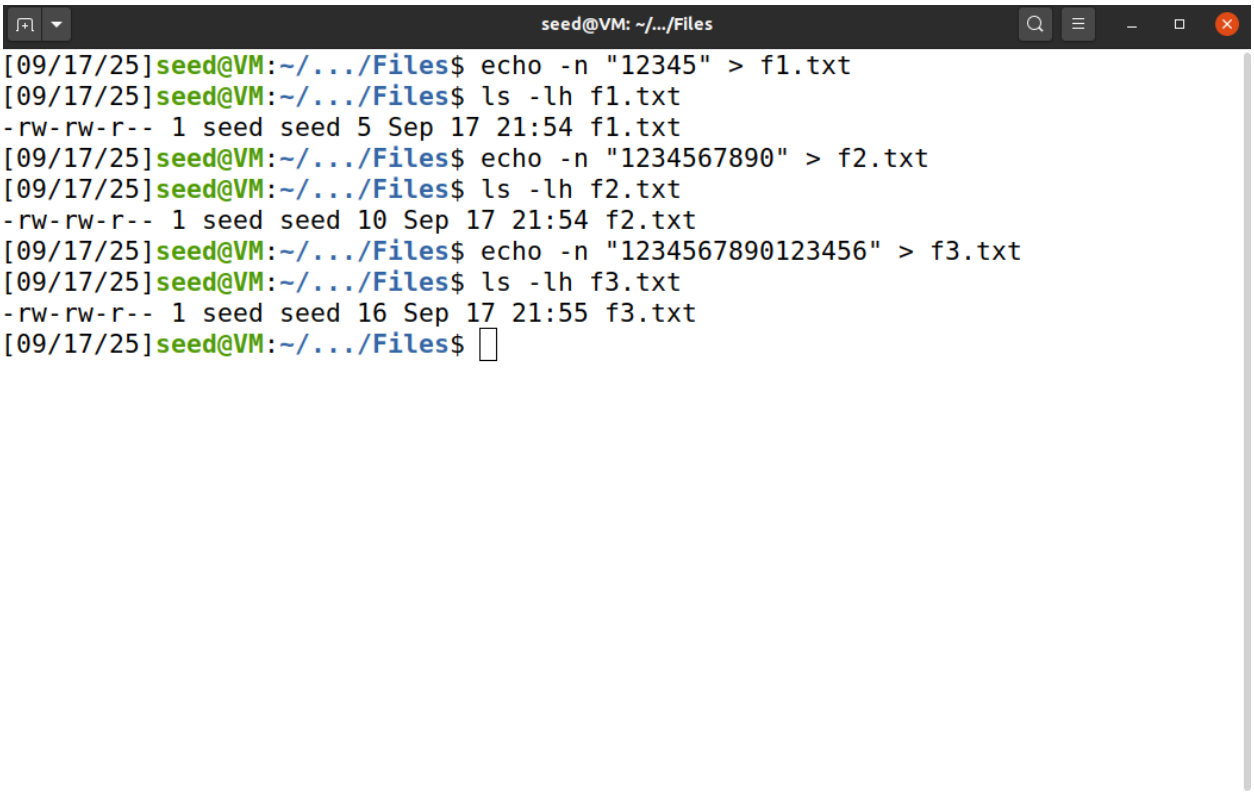
Steps Performed:

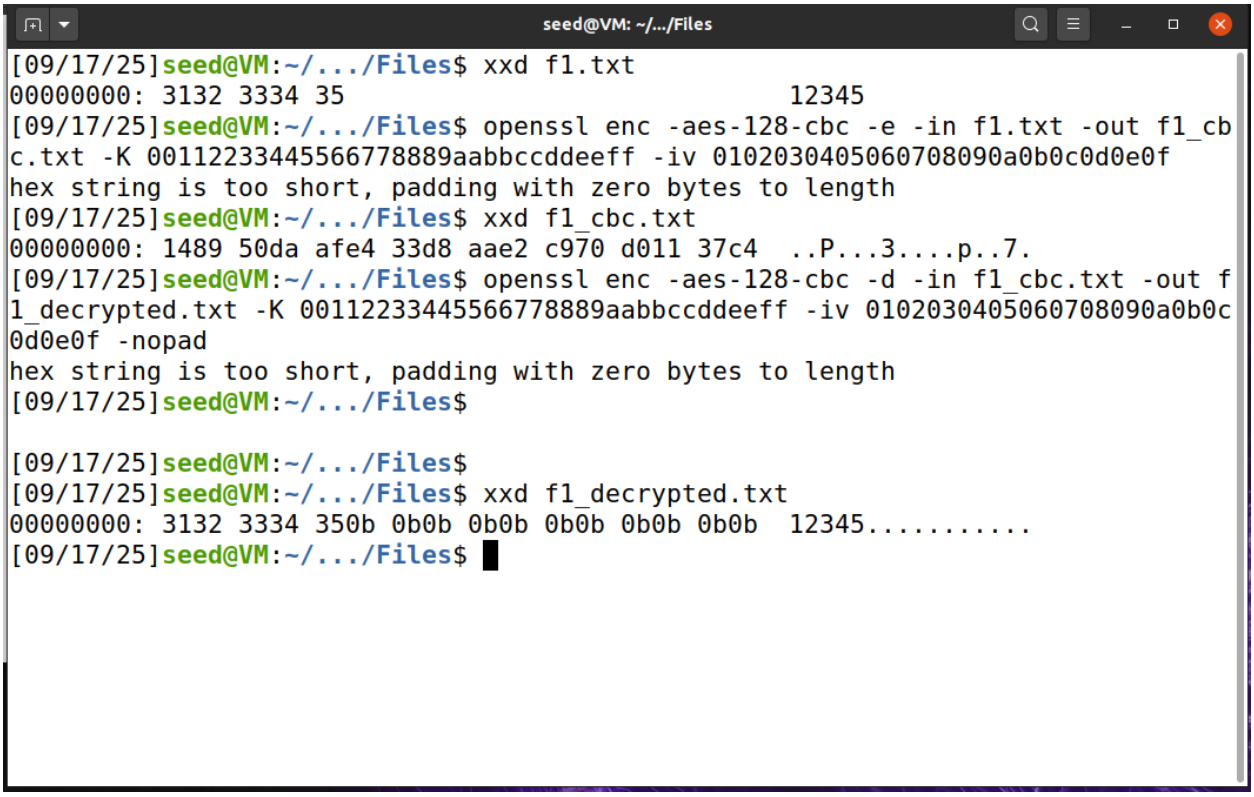
1. I have created a file *name.txt* and verified its size using ls -ld command and encrypted the file using all four modes ECB, CBC, CFB, and OFB.
2. Then, decrypted the file and checked the length of the new decrypted file to verify if there is any padding added
3. Created three files of sizes 5,10, and 16 bytes.
4. Encrypted all these files with AES-128-CBC cipher.
5. Decrypted files with -nopad to view padding
6. Used xxd command to inspect padded values.

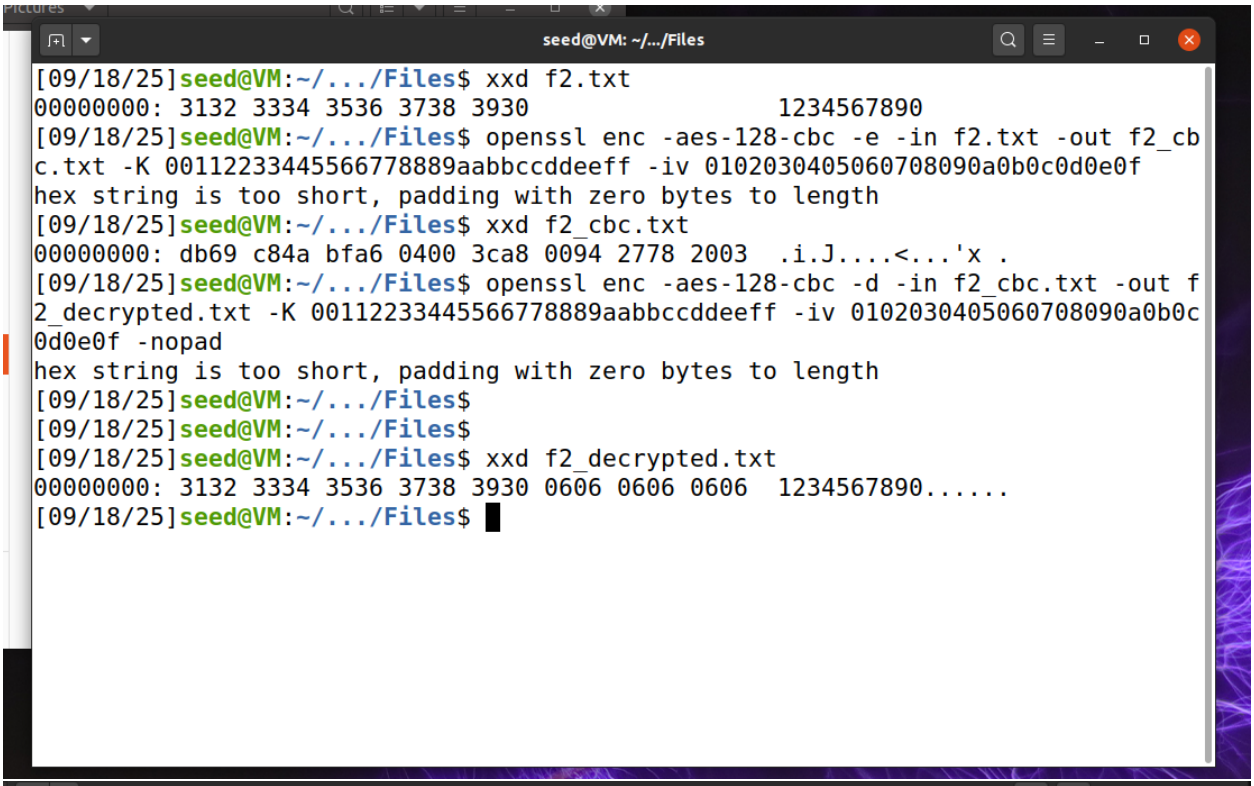
Screenshots:

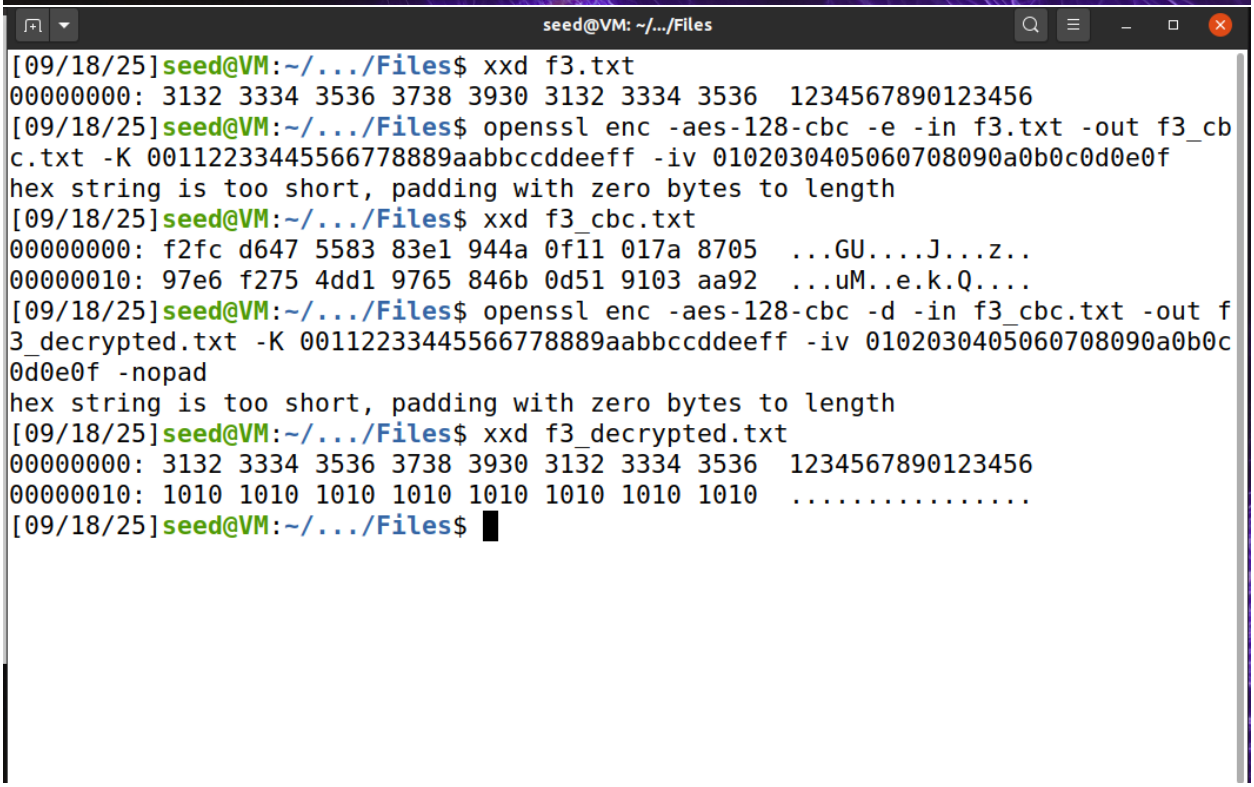

seed@VM: ~/.../Files
[09/17/25]seed@VM:~/.../Files$ ls -ld name.txt
-rw-rw-r-- 1 seed seed 11 Sep 17 21:27 name.txt
[09/17/25]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in name.txt -out name_ecb.txt -K 00
112233445566778889aabbccddeeff
[09/17/25]seed@VM:~/.../Files$ ls -ld name_ecb.txt
-rw-rw-r-- 1 seed seed 16 Sep 17 21:41 name_ecb.txt
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in name.txt -out name_cbc.txt -K 00
112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f
hex string is too short, padding with zero bytes to length
[09/17/25]seed@VM:~/.../Files$ ls -ld name_cbc.txt
-rw-rw-r-- 1 seed seed 16 Sep 17 21:42 name_cbc.txt
[09/17/25]seed@VM:~/.../Files$


seed@VM: ~/.../Files
[09/17/25]seed@VM:~/.../Files$ ls -ld name.txt
-rw-rw-r-- 1 seed seed 11 Sep 17 21:27 name.txt
[09/17/25]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e -in name.txt -out name_cbc.txt -K 00
112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f
hex string is too short, padding with zero bytes to length
[09/17/25]seed@VM:~/.../Files$ ls -ld name_cfb.txt
-rw-rw-r-- 1 seed seed 11 Sep 17 21:39 name_cfb.txt
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$ openssl enc -aes-128-ofb -e -in name.txt -out name_ofb.txt -K 00
112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f
hex string is too short, padding with zero bytes to length
[09/17/25]seed@VM:~/.../Files$ ls -ld name_ofb.txt
-rw-rw-r-- 1 seed seed 11 Sep 17 21:44 name_ofb.txt
[09/17/25]seed@VM:~/.../Files$


seed@VM: ~/.../Files
[09/17/25]seed@VM:~/.../Files$ echo -n "12345" > f1.txt
[09/17/25]seed@VM:~/.../Files$ ls -lh f1.txt
-rw-rw-r-- 1 seed seed 5 Sep 17 21:54 f1.txt
[09/17/25]seed@VM:~/.../Files$ echo -n "1234567890" > f2.txt
[09/17/25]seed@VM:~/.../Files$ ls -lh f2.txt
-rw-rw-r-- 1 seed seed 10 Sep 17 21:54 f2.txt
[09/17/25]seed@VM:~/.../Files$ echo -n "1234567890123456" > f3.txt
[09/17/25]seed@VM:~/.../Files$ ls -lh f3.txt
-rw-rw-r-- 1 seed seed 16 Sep 17 21:55 f3.txt
[09/17/25]seed@VM:~/.../Files$


seed@VM: ~/.../Files
[09/17/25]seed@VM:~/.../Files$ xxd f1.txt
00000000: 3132 3334 35 12345
[09/17/25]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_cb
c.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f
hex string is too short, padding with zero bytes to length
[09/17/25]seed@VM:~/.../Files$ xxd f1_cbc.txt
00000000: 1489 50da afe4 33d8 aae2 c970 d011 37c4 ..P...3....p..7.
[09/17/25]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -d -in f1_cbc.txt -out f
1_decrypted.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c
0d0e0f -nopad
hex string is too short, padding with zero bytes to length
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$
[09/17/25]seed@VM:~/.../Files$ xxd f1_decrypted.txt
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b 12345.....
[09/17/25]seed@VM:~/.../Files$


seed@VM: ~/.../Files
[09/18/25]seed@VM:~/.../Files$ xxd f2.txt
00000000: 3132 3334 3536 3738 3930 1234567890
[09/18/25]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in f2.txt -out f2_cb
c.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f
hex string is too short, padding with zero bytes to length
[09/18/25]seed@VM:~/.../Files$ xxd f2_cbc.txt
00000000: db69 c84a bfa6 0400 3ca8 0094 2778 2003 .i.J....<...'x .
[09/18/25]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -d -in f2_cbc.txt -out f
2_decrypted.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c
0d0e0f -nopad
hex string is too short, padding with zero bytes to length
[09/18/25]seed@VM:~/.../Files$
[09/18/25]seed@VM:~/.../Files$
[09/18/25]seed@VM:~/.../Files$ xxd f2_decrypted.txt
00000000: 3132 3334 3536 3738 3930 0606 0606 0606 1234567890......
[09/18/25]seed@VM:~/.../Files$


seed@VM: ~/.../Files
[09/18/25]seed@VM:~/.../Files$ xxd f3.txt
00000000: 3132 3334 3536 3738 3930 3132 3334 3536 1234567890123456
[09/18/25]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in f3.txt -out f3_cb
c.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f
hex string is too short, padding with zero bytes to length
[09/18/25]seed@VM:~/.../Files$ xxd f3_cbc.txt
00000000: f2fc d647 5583 83e1 944a 0f11 017a 8705 ...GU....J...z..
00000010: 97e6 f275 4dd1 9765 846b 0d51 9103 aa92 ...uM..e.k.Q....
[09/18/25]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -d -in f3_cbc.txt -out f
3_decrypted.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c
0d0e0f -nopad
hex string is too short, padding with zero bytes to length
[09/18/25]seed@VM:~/.../Files$ xxd f3_decrypted.txt
00000000: 3132 3334 3536 3738 3930 3132 3334 3536 1234567890123456
00000010: 1010 1010 1010 1010 1010 1010 1010 1010 ....
[09/18/25]seed@VM:~/.../Files$

Observations:

- Modes and Padding Requirement.
  ECB and CBC need padding (block-based)
  CFB and OFB need no padding (stream-based)
- File 1 (5 bytes), Encrypted size: 16 bytes, Padding: 11 bytes of 0x0B
- File 2 (10 bytes), Encrypted size: 16 bytes, Padding: 6 bytes of 0x06
- File 1 (16 bytes), Encrypted size: 32 bytes (extra block added), Padding: 16 bytes of 0x10

Explanations:

- ECB and CBC encrypt using 16-byte blocks which require padding when plain text length is not a multiple block size.
- CFB and OFB are stream-based processes byte-by-byte therefore no padding is required.
- So, the difference between plaintext length and next block boundary is added as padding. If the plaintext length is exactly a block size, then a full block of padding is added.

Task 5: Error Propagation – Corrupted Cipher Text

How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively?

- In ECB, all plaintext except the 4^{th} block (55^{th} bit is in 4^{th} block) can be recovered.
- In CBC, all plaintext except the 4^{th} block and 5^{th} block can be recovered.
- In CFB, ciphertext feed backs to generate future keystream i.e., single ciphertext pollutes future keystreams
- In OFB, only the bit won't be recovered everything else can be recovered.

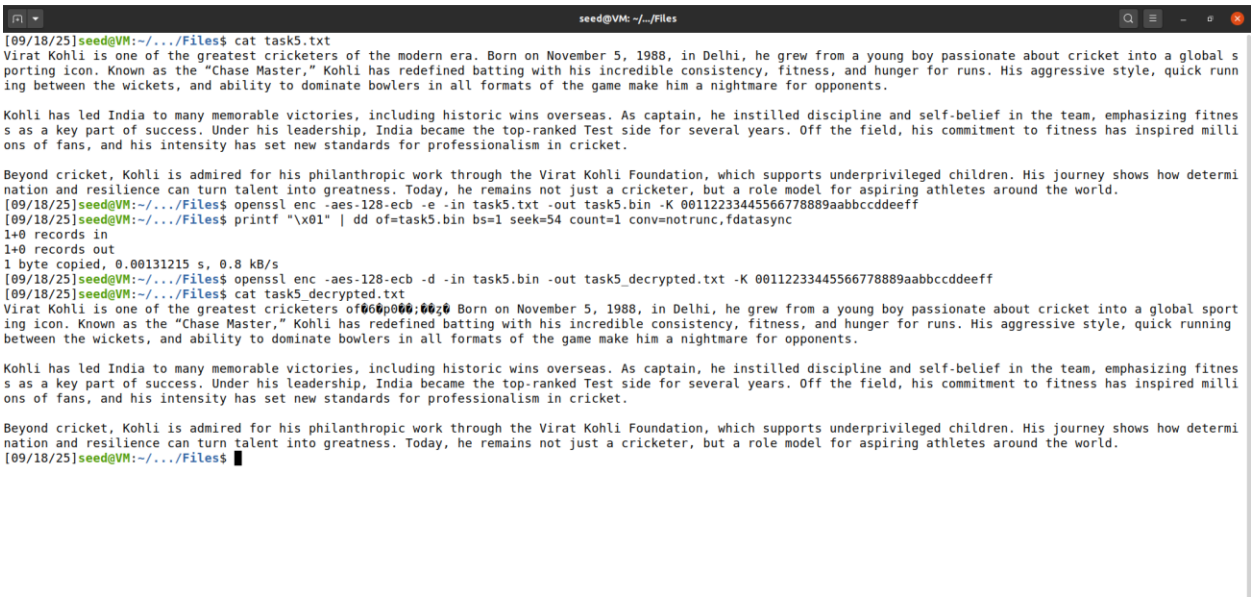
Steps Performed:

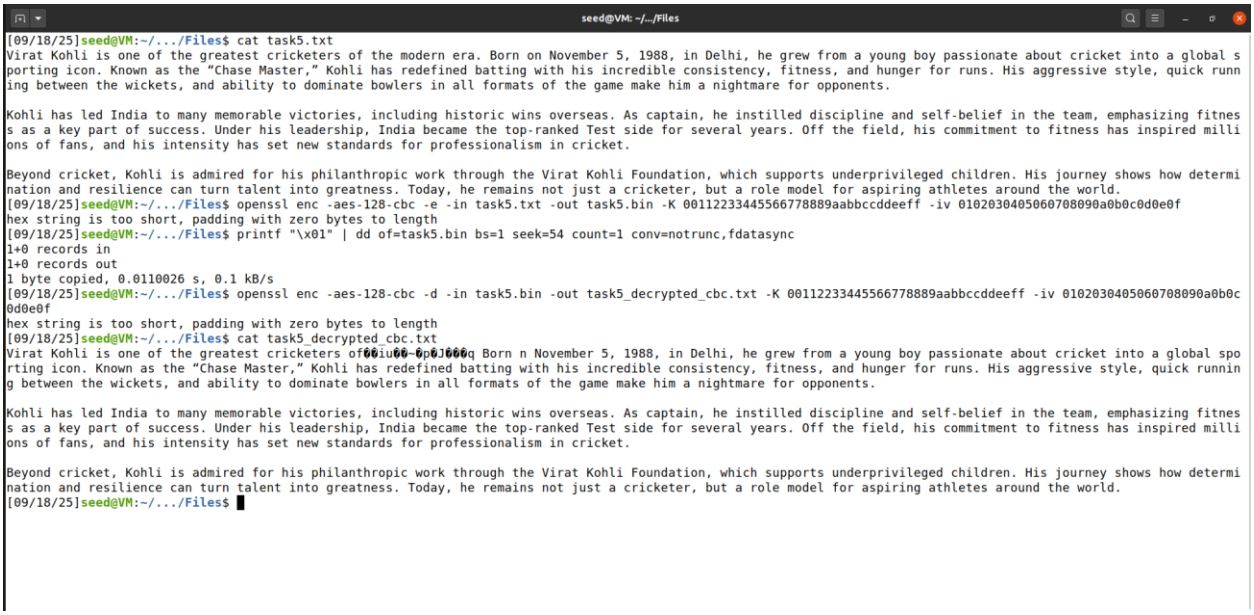
1. Created a plaintext file *task5.txt* and encrypted it using AES-128 in ECB, CBC, CFB, OFB modes.
2. Flipped one bit in the 55^{th} byte of ciphertext using dd (as bless editor was not working for me)

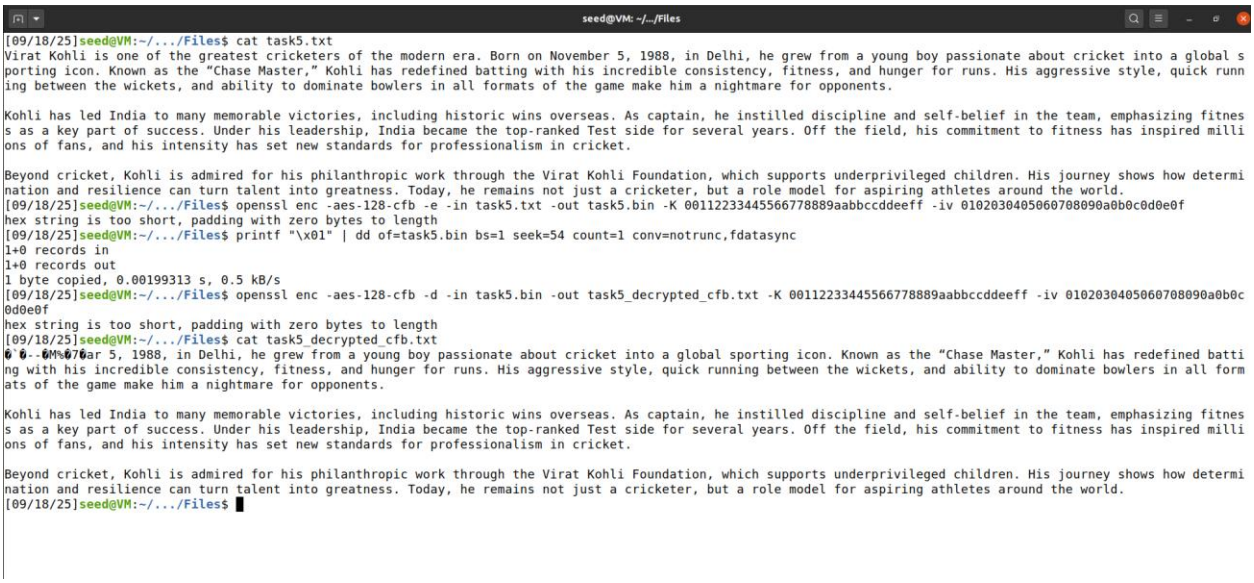
   printf '\x01' | dd of=file.enc bs=1 seek=54 count=1 conv=notrunc

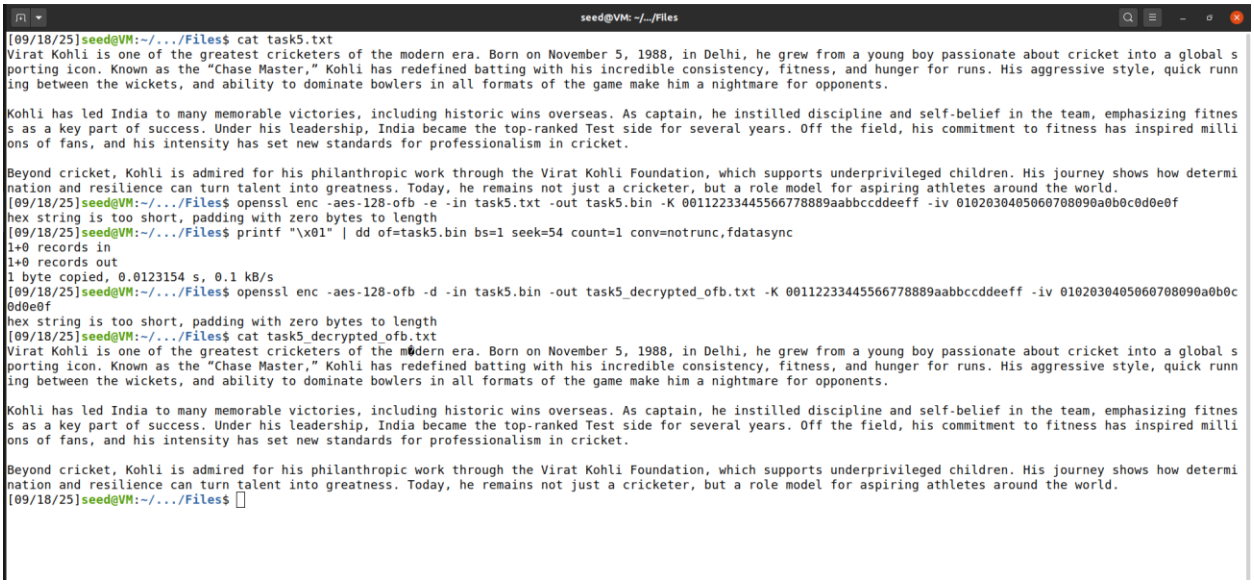
3. Decrypted the corrupted ciphertext with the same key/IV.
4. Printed the decrypted text and compared it with the original text.

Screenshots:









Observations:

- ECB: Entire 4th block of plaintext corrupted; all other blocks intact.
- CBC: 4th block completely corrupted, next block has some flipped bytes, rest unchanged.
- CFB: 4th block has bit flips, all subsequent blocks unreadable.
- OFB: Only flipped bit in 4th block changed, rest entirely intact.

Explanations:

- ECB: Every block independently → corruption confined to a single block.
- CBC: P_i = D(C_i) ⊕ C_{i-1}, so corruption of C_i destroys block i. P_{i+1} depends on C_i, so same bit positions flipped. Beyond that, ciphertext fine.
- CFB: Keystream based on ciphertext feedback. Corruption of C_i changes all subsequent keystreams → all subsequent blocks destroyed.
- OFB: Keystream independent of ciphertext. Flipping a ciphertext bit flips only the corresponding plaintext bit.

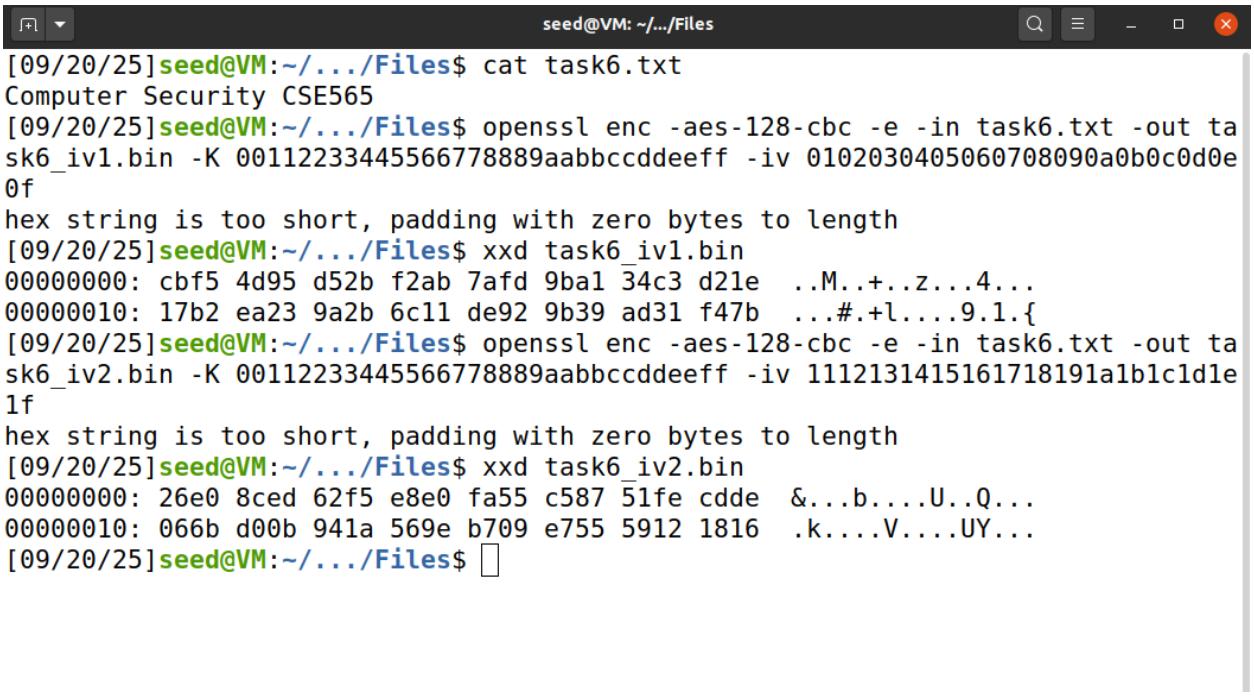
Task 6: Initial Vector (IV) and Common Mistakes

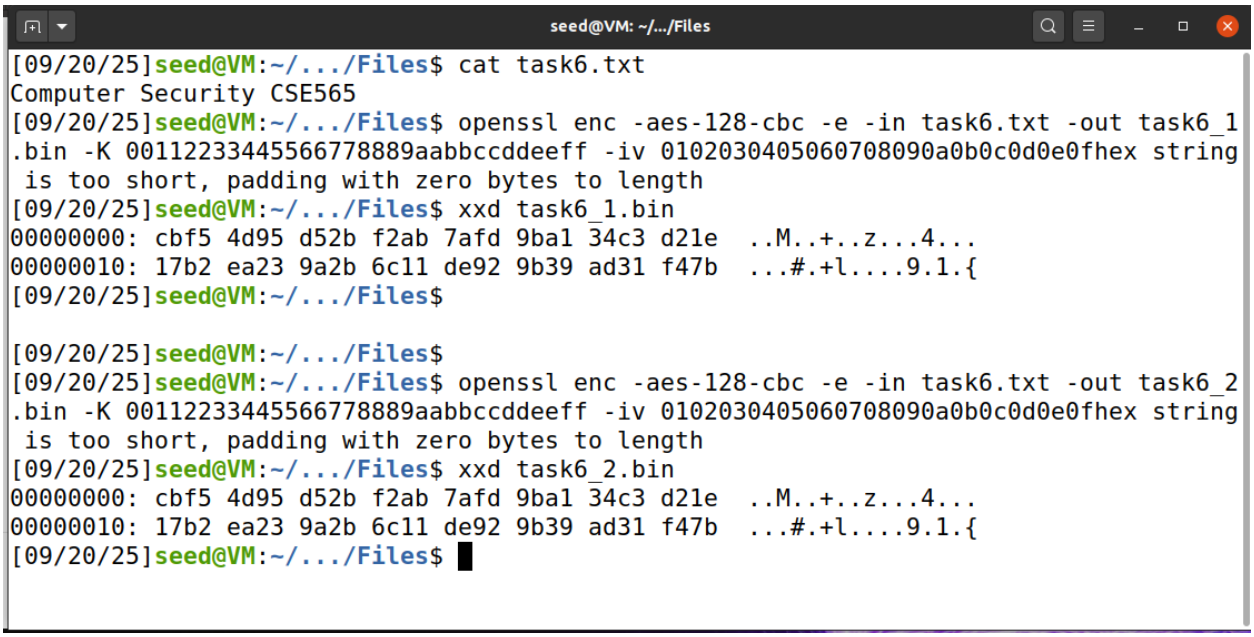
Task 6.1. IV Experimented

Steps Performed:

1. Created a file task6.txt to store a small plain text.
2. Encrypted task6.txt with aes-128-cbc mode that uses an IV, once with IV1 and once with IV2.
3. I repeated the same by encrypting the same plain text with same key but reused same IV value for both encryptions and observed results using xxd.

Screenshots:





Observations & Explanation:

- Using different IVs ($IV_1 \neq IV_2$) it produces different ciphertexts. IV's purpose is to randomize the first block so identical plaintexts do not result in identical ciphertexts.

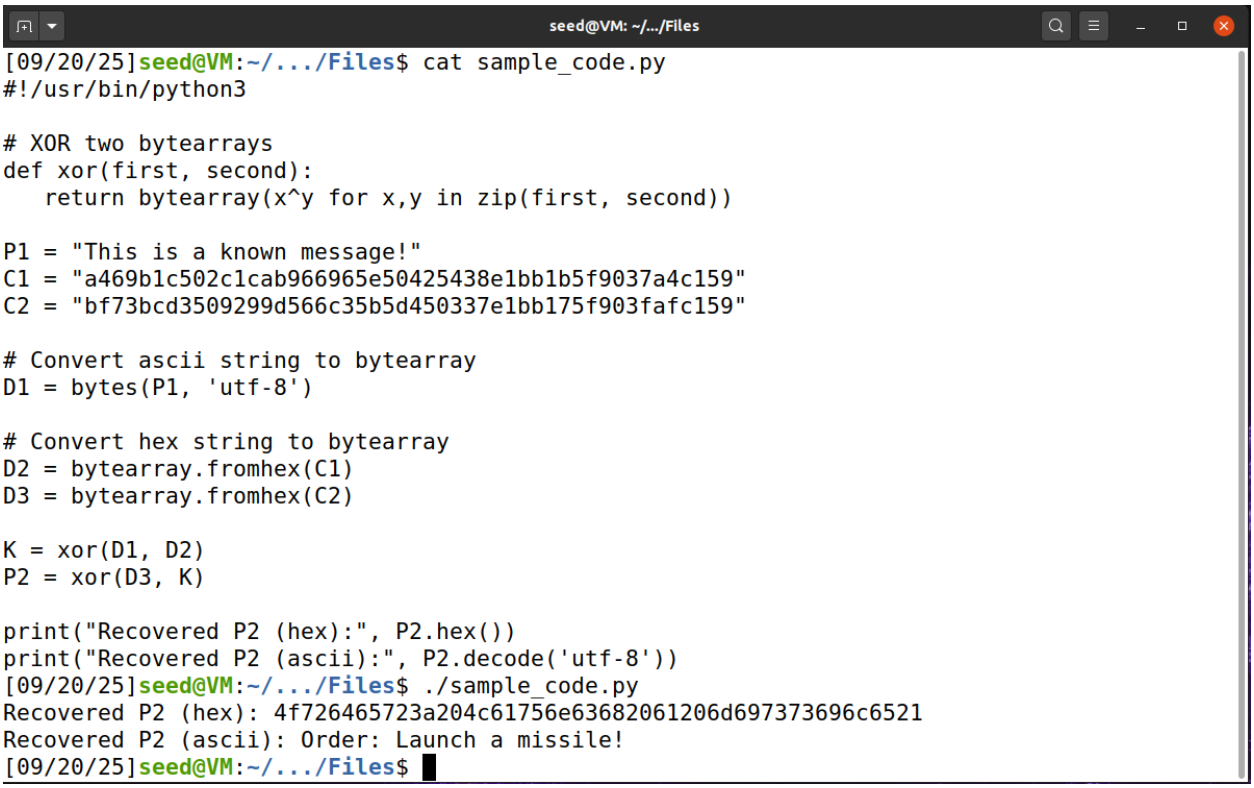
- With the same IV and with the same key and same plaintext, we obtain the same ciphertexts (for deterministic modes like CBC) or you will reuse the same keystream again (for stream/stream-like modes like OFB/CTR) i.e. the ciphertexts are identical or reveal relationships between plaintexts.

Task 6.2. Common Mistake: Use the Same IV

Steps Performed:

1. Converted the given Plaintext P1 and Ciphertexts (C1 & C2) into byte arrays
2. XOR C1 and P1 to get the key stream.
3. Since C2 was encrypted with same IV(key), we can XOR the same key stream with C2 to get P2
4. All the implementation is updated in the sample_code.py file and executed it for the output

Screenshots:



Observations:

- If the same IV were used in OFB, the entire plaintext P2 could be constructed by P1 and C1.
- In CFB, only part of P2 would be revealed since the errors carry over through the feedback.

Explanation:

- Using a common IV in OFB is insecure as it produces the same keystream; a single plain text reveals all other messages encrypted with it.
- IVs should be different to maintain confidentiality.
- In CFB, the feedback mechanism makes the information about one plaintext only reveal partial information in subsequent ciphertexts limiting the damage.

Task 6.3. Common Mistake: Use a Predictable IV

Steps performed:

1. Connected to the oracle with:

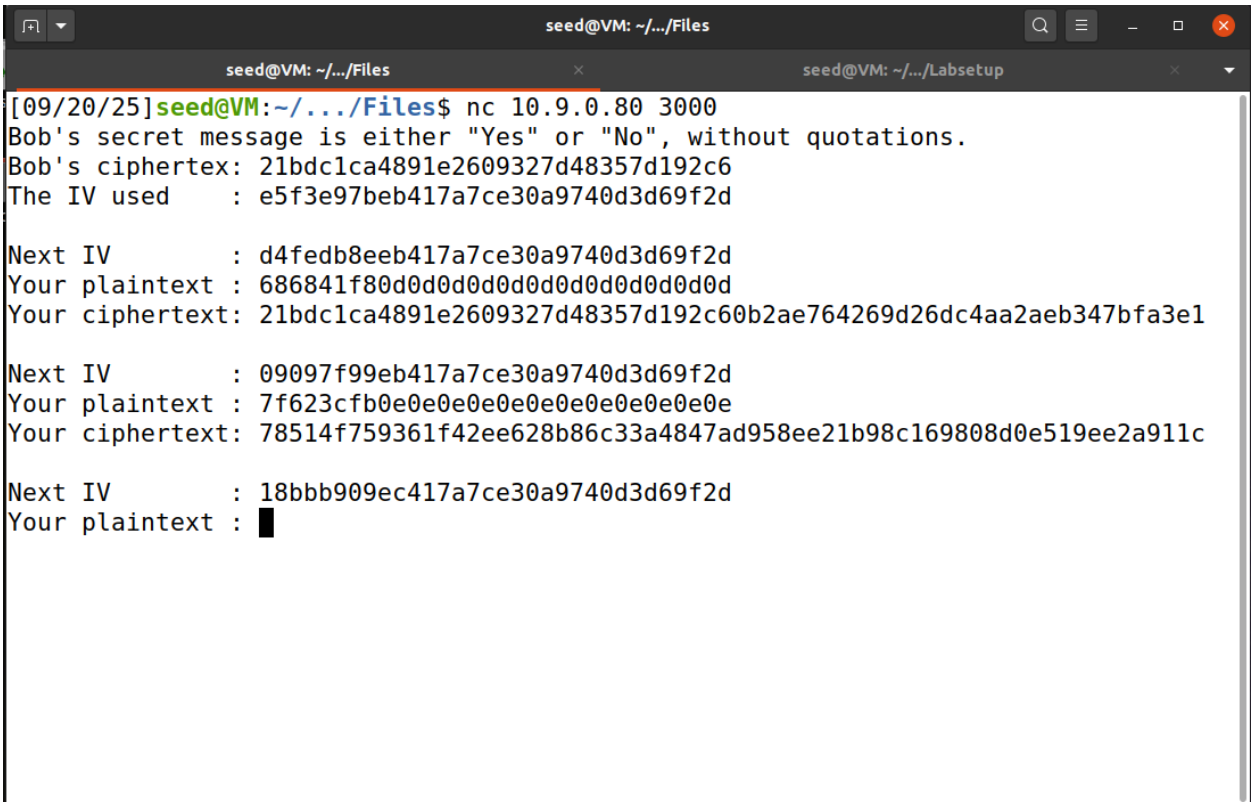
   nc 10.9.0.80 3000

2. Got the Bob's ciphertext, IV used, and next IV printed out by the oracle.
3. Constructed a plaintext block P_send with:

   $$P\_send = IV\_next \oplus IV\_secret \oplus pad(guess)$$

4. where guess was "Yes" or "No" which were converted into bytes and padding used PKCS#7.
5. Passed the constructed plaintext to the oracle and received the ciphertext.
6. Compared the ciphertext of the oracle with the target ciphertext of Bob to ensure the secret.

Screenshots:



Observations:

- Since the IVs were predictable, a compromised plaintext could be encrypted to reveal whether Bob's secret message was "Yes" or "No."
- By comparing the generated hex code to the given, I have found that the Bob's secret message was "Yes".
- The oracle returned ciphertext that was the same as the secret upon correct guess.

Explanation:

- Exploitable IVs make AES-CBC vulnerable to a chosen-plaintext attack.
- While AES is secure, knowing the next IV allows an attacker to dictate encryption and obtain secret information.
- IVs must be random and unpredictable in CBC mode.

Task 7: Programming using the Crypto Library

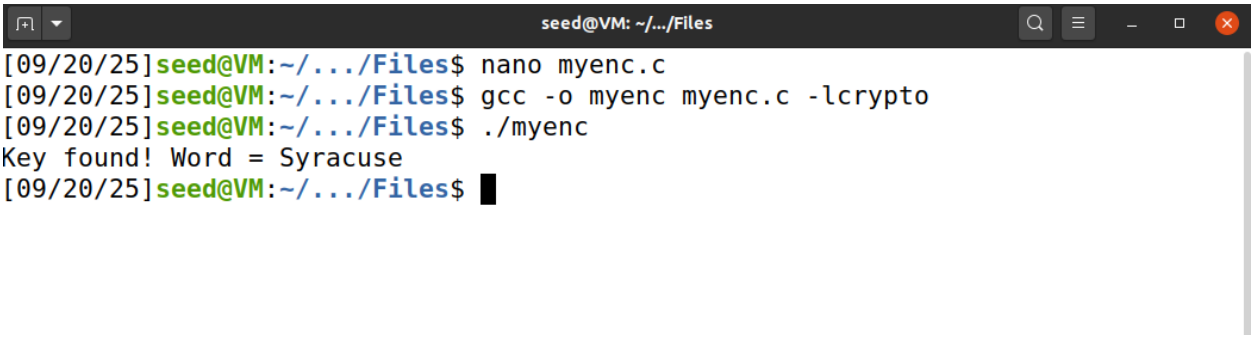
Steps Performed:

1. Saved the configuration plaintext "This is a top secret." in a file to ensure exact length.
2. Read an English word dictionary from words.txt.
3. For each word:
    - Padded it with # to 16 bytes to form the AES-128 key.
    - Used the OpenSSL crypto library (EVP API) to encrypt the plaintext with the key and given IV.

    - Verified the resulting ciphertext against the provided target ciphertext.

4. Shutdown when a match was found, displaying the correct key.

Screenshots:



Code Snippet:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <openssl/evp.h>
#include <openssl/aes.h>

int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *key,
            unsigned char *iv, unsigned char *ciphertext) {
    EVP_CIPHER_CTX *ctx;
    int len, ciphertext_len;

    if(!(ctx = EVP_CIPHER_CTX_new())) return -1;

    EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);

    EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len);
    ciphertext_len = len;

    EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
    ciphertext_len += len;

    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_len;
}

int main() {
    unsigned char *iv = (unsigned char
*)"\xaa\xbb\xcc\xdd\xee\xff\x00\x99\x88\x77\x66\x55\x44\x33\x22\x11";

FILE *fp = fopen("task7.txt", "rb");
if (!fp) { perror("task7.txt"); exit(1); }

fseek(fp, 0, SEEK_END);
```

```c
long filesize = ftell(fp);
rewind(fp);

unsigned char *plaintext = malloc(filesize);
fread(plaintext, 1, filesize, fp);
fclose(fp);




unsigned char target_cipher[] = {
        0x76,0x4a,0xa2,0x6b,0x55,0xa4,0xda,0x65,
        0x4d,0xf6,0xb1,0x9e,0x4b,0xce,0x00,0xf4,
        0xed,0x05,0xe0,0x93,0x46,0xfb,0x0e,0x76,
        0x25,0x83,0xcb,0x7d,0xa2,0xac,0x93,0xa2
    };

    FILE *f = fopen("words.txt", "r");
    if (!f) { perror("words.txt"); exit(1); }

    char word[64];
    while (fgets(word, sizeof(word), f)) {
        word[strcspn(word, "\n")] = 0;
        int wlen = strlen(word);
        if (wlen == 0 || wlen > 16) continue;

        unsigned char key[16];
        memset(key, 0, 16);
        memcpy(key, word, wlen);
        for (int i = wlen; i < 16; i++) key[i] = '#';

        unsigned char ciphertext[128];
        int clen = encrypt(plaintext, filesize,key, iv, ciphertext);

        if (clen == sizeof(target_cipher) &&
            memcmp(ciphertext, target_cipher, clen) == 0) {
            printf("Key found! Word = %s\n", word);
            break;
        }
    }

    fclose(f);
    return 0;
}
```

Code Explanation:

- The program uses fopen() to open the dictionary file (words.txt) in read mode and fgets() to read a word by a time line by line. The newline characters are stripped off using strcspn() to correctly pad the key.
- Candidate words with a size of under 16 bytes are padded with # to form a valid 16-byte AES-128 key, saved in an array unsigned char key[16].
- The encrypt() function encapsulates AES-128-CBC encryption using the OpenSSL EVP API.
- EVP_CIPHER_CTX_new() sets up a new encryption context.
- EVP_EncryptInit_ex() sets the context to use the AES-128-CBC cipher, the key created, and the provided IV.
- EVP_EncryptUpdate() encrypts the plaintext bytes and stores the intermediate ciphertext.
- EVP_EncryptFinal_ex() finishes encryption, performing padding and appending any remaining bytes to the ciphertext.
- EVP_CIPHER_CTX_free() releases memory held by the context.
- memcmp() is used to byte-by-byte compare the produced ciphertext with the destination ciphertext. If they match, the correct key has been found.
- strlen() is used to determine the plaintext length, and memcpy()/memset() are used to copy and initialize the array of the key.
- The application goes through all dictionary words, encrypting and comparing, using a dictionary attack successfully without attempting all 128-bit keys by brute force.

Observations:

- A single dictionary word, padded and used as the key, produced ciphertext indistinguishable from the target.
- The program could recover the encryption key without trying all 128-bit possibilities using brute-force.

Explanation:

- AES-128-CBC encryption is key and IV-deterministic.
- By trying all likely English words (padded to 16 bytes), the correct key can be decided efficiently.
- Keys derived from low-entropy or predictable sources (like words in a dictionary) are vulnerable to dictionary attacks.

Resources:

- Week-3_Class-1-2_Cryptography.pptx
- EVP_EncryptInit - OpenSSL Documentation
- https://medium.com/@amit.kulkarni/encrypting-decrypting-a-file-using-openssl-evp-b26e0e4d28d4
- https://www.tutorialspoint.com/cryptography/cipher_feedback_mode.htm
- https://www.geeksforgeeks.org/c/basics-file-handling-c/
- https://linuxcommand.org/lc3_man_page_index.php