Name: Deep Pawar(A20545137)
Professor: Yousef Elmehdwi
Institute: Illinois Institute of Technology

CS 458: Introduction to Information Security

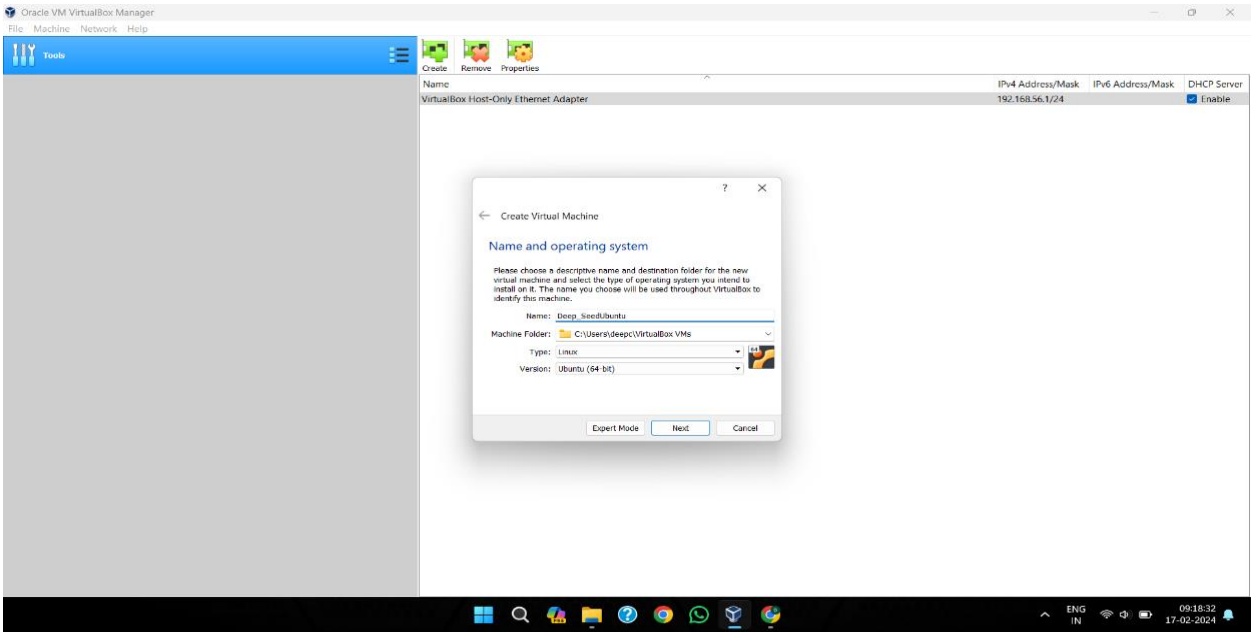
Spring 2024 – Lab 2 Secret-Key Encryption

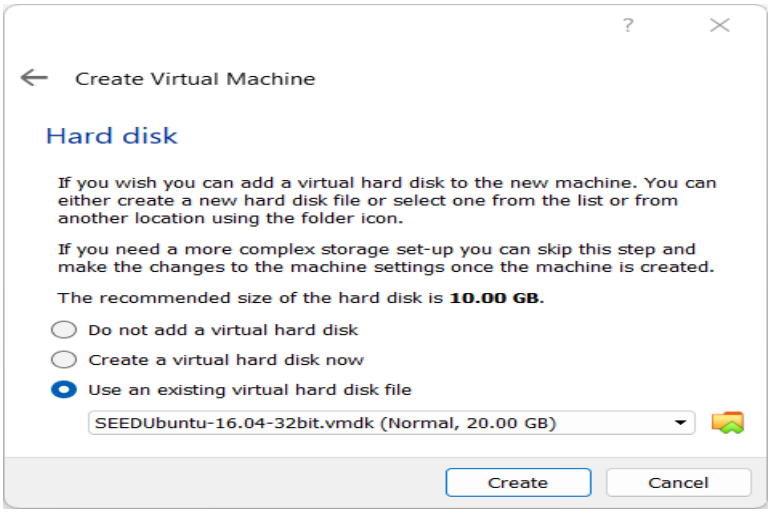
1. Overview

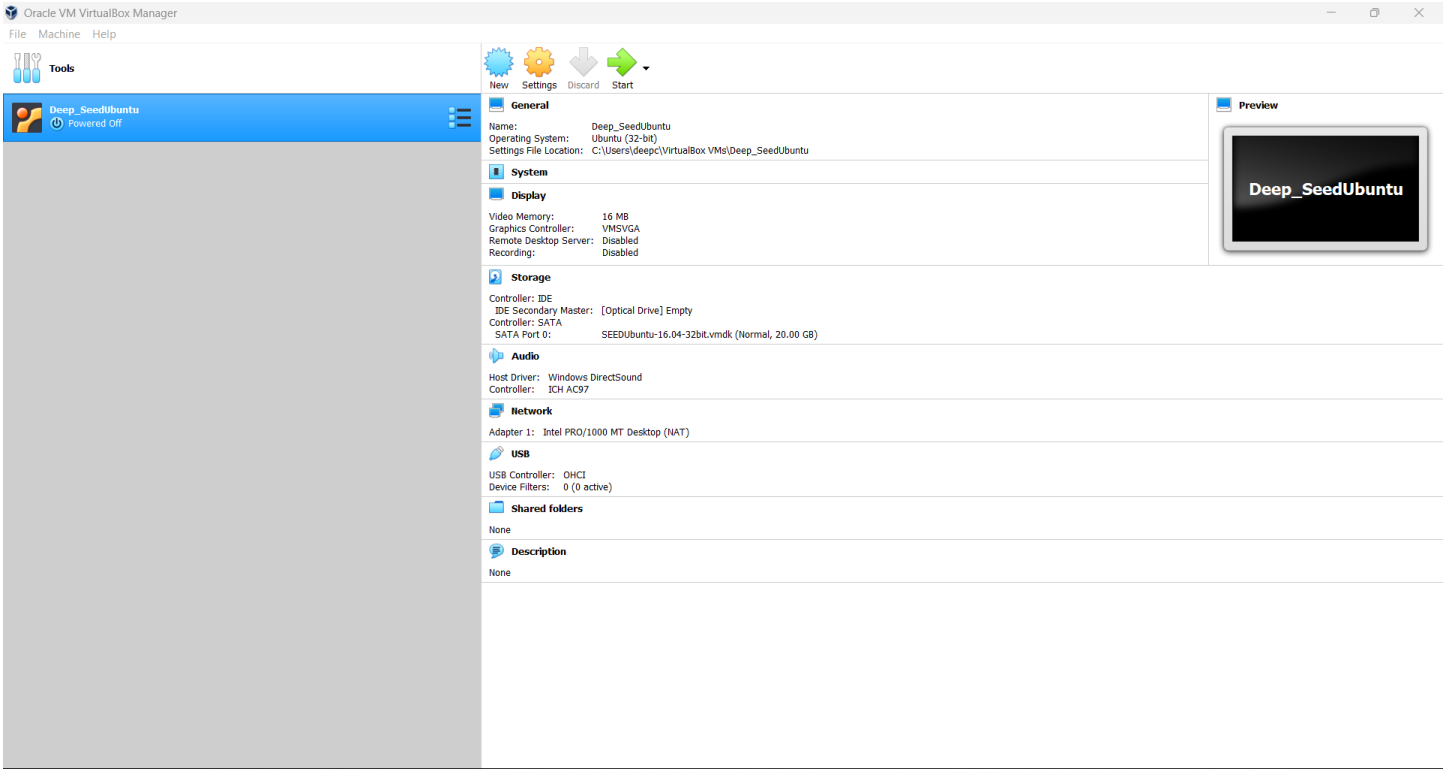
The learning objective of this lab is for students to get familiar with the concepts in the secret-key encryption. After finishing the lab, students should be able to gain a first-hand experience on encryption algorithms, encryption modes, and initial vector (IV). Moreover, students will be able to use tools and write programs to encrypt/decrypt messages. This lab covers the following topics:

- Secret-key encryption
- Substitution cipher and frequency analysis
- Encryption modes, IV, and paddings
- Common mistakes in using encryption algorithms.
- Programming using the crypto library.

- **Seed Ubuntu Installation:**






Oracle VM VirtualBox Manager
File Machine Help
Tools
Deep_SeedUbuntu
Powered Off
New Settings Discard Start
General
Name: Deep_SeedUbuntu
Operating System: Ubuntu (32-bit)
Settings File Location: C:\Users\deepc\VirtualBox VMs\Deep_SeedUbuntu
System
Display
Video Memory: 16 MB
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled
Storage
Controller: IDE
IDE Secondary Master: [Optical Drive] Empty
Controller: SATA
SATA Port 0: SEEDUbuntu-16.04-32bit.vmdk (Normal, 20.00 GB)
Audio
Host Driver: Windows DirectSound
Controller: ICH AC97
Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT)
USB
USB Controller: OHCI
Device Filters: 0 (0 active)
Shared folders
None
Description
None
Preview
Deep_SeedUbuntu


Deep_SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Ubuntu Desktop
9:37 AM
SEEDLABS
Right Ctrl

2. Lab Tasks

2.1 Task 1: Frequency Analysis

It is well-known that monoalphabetic substitution cipher (also known as monoalphabetic cipher) is not secure, because it can be subjected to frequency analysis. In this lab, you are given a cipher-text that is encrypted using a monoalphabetic cipher; namely, each letter in the original text is replaced by another letter, where the replacement does not vary (i.e., a letter is always replaced by the same letter during the encryption). Your job is to find out the original text using frequency analysis. It is known that the original text is an English article.

Your job is to use the frequency analysis to figure out the encryption key and the original plaintext given a ciphertext (`ciphertext.txt`). The file `ciphertext.txt` can be download from the course Blackboard.

Guidelines. Using the frequency analysis, you can find out the plaintext for some of the characters quite easily. For those characters, you may want to change them back to its plaintext, as you may be able to get more clues. It is better to use capital letters for plaintext, so for the same letter, we know which is plaintext and which is ciphertext. You can use the `tr` command to do this. For example, in the following, we replace letters a, e, and t in `in.txt` with letters X, G, E, respectively; the results are saved in `out.txt`.

```
$ tr 'aet' 'XGE' < in.txt > out.txt
```

There are many online resources that you can use. We list four useful links in the following:

- https://www.dcode.fr/frequency-analysis: This website can produce the statistics from a ciphertext, including the single-letter frequencies, bigram frequencies (2-letter sequence), and trigram frequencies (3-letter sequence), etc.
- https://en.wikipedia.org/wiki/Frequency analysis: This Wikipedia page provides frequencies for a typical English plaintext.
- https://en.wikipedia.org/wiki/Bigram: Bigram frequency.
- https://en.wikipedia.org/wiki/Trigram: Trigram frequency.

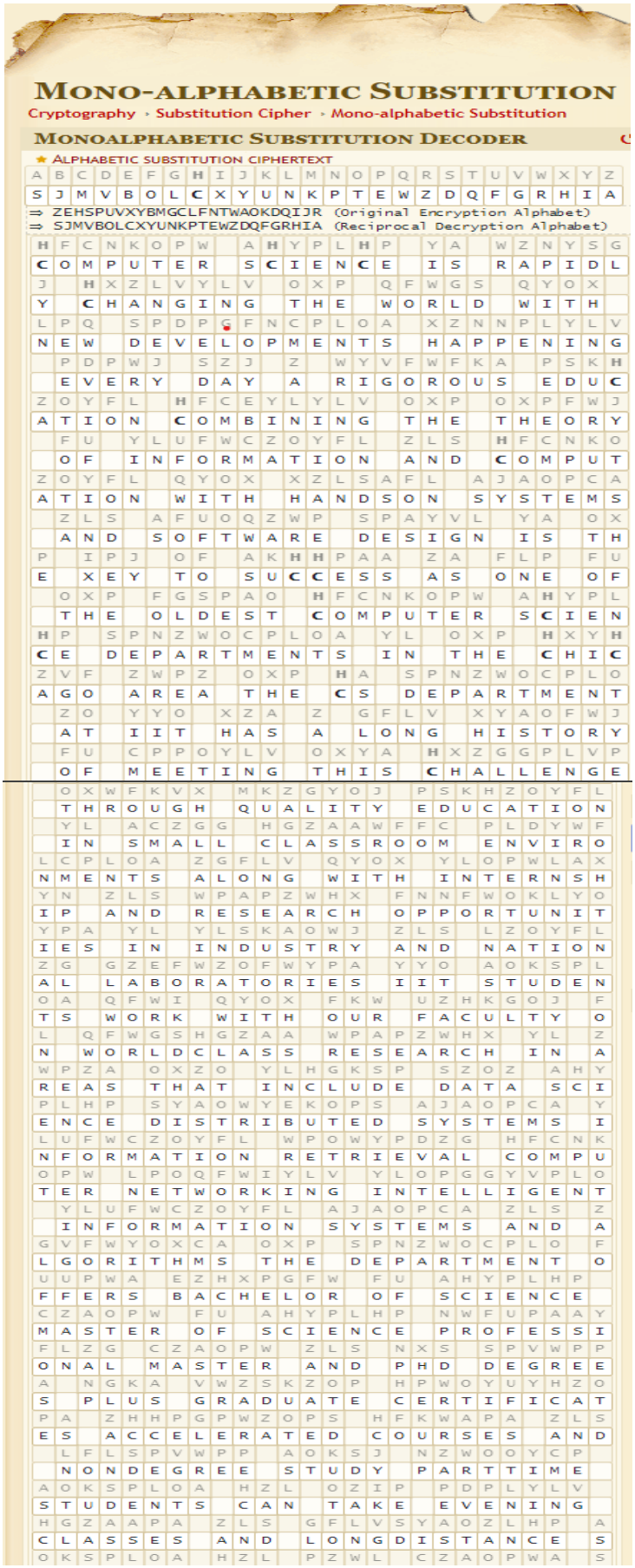
- **Ans:**

Here, the First task is to use the frequency analysis to figure out the encryption key and the original plaintext given a ciphertext. For that, I have used the first online resource mentioned above.

https://www.dcode.fr/frequency-analysis

In the **text to analyze textbox**, I entered the ciphertext given and when clicked on the **Launch Analysis** button I got frequencies of all the letters in the ciphertext in tabular form.

- After that, I opened the **Mono-alphabetic Substitution** link to decode the given ciphertext as follows:



- Using the above collected data I decoded the given ciphertext.txt into plaintext (plaintext.txt) as follows:

```
Terminal
[02/17/24]seed@VM:~$ gedit ciphertext.txt
[02/17/24]seed@VM:~$ cat ciphertext.txt
hfcnkopw ahyplhp ya wznysgj hxzlvylv oxp qfwgs qyox lpq spdpgfncploa xznnplylv p
dpwj szj z wyvfwfka pskhzoyfl hfceylylv oxp oxpfwj fu ylufwczoyfl zls hfcnkozoyf
l qyox xzlsafl ajaopca zls afuoqzwp spayvl ya oxp ipj of akhhpaa za flp fu oxp f
gspao hfcnkopw ahyplhp spnzwocploa yl oxp hxyhzvf zwpz oxp ha spnzwocplo zo yyo
xza z gflv xyaofwj fu cppoylv oxya hxzggplvp oxwfkvx mkzgyoj pskhzoyfl yl aczgg
hgzaawffc pldywflcploa zgflv qyox ylopwlaxyn zls wpapzwhx fnnfwoklyoypa yl ylska
owj zls lzoyflzg gzefwzofwypa
yyo aoksploa qfwi qyox fkw uzhkgoj fl qfwgshgzaa wpapzwhx yl zwpza oxzo ylhgksp
szoz ahyplhp syaowyekops ajaopca ylufwczoyfl wpowypdzg hfcnkopw lpoqfwiylv ylopg
gyvplo ylufwczoyfl ajaopca zls zgvfwyoxca
oxp spnzwocplo fuupwa ezhxpgfw fu ahyplhp czaopw fu ahyplhp nwfupaayflzg czaopw
zls nxs spvwppa ngka vwzskzop hpwoyuyhzopa zhhpgpwzops hfkwapa zls lflspvwpp aok
sj nzwooycp aoksploa hzl ozip pdplylv hgzaapa zls gflvsyaozlhp aoksploa hzl pzwl
 czaopwa spvwppa flgylp aoksploa wzop fkw opzhxylv za zcflv oxp epao zo oxp klyd
pwayoj zls fkw uzhkgoj xzdp qfl lkcpwfka opzhxylv zqzwsa
oxp aphwpo aploplhp ya vffs bfe vkja
[02/17/24]seed@VM:~$
```

- Now, by using the key I decoded the cipher text into plain text as follows:
- The key is: **SJMVBOLCKYUNQPTEWZDXFGRHIA**

```
Terminal
[02/17/24]seed@VM:~$ tr 'a-z' 'SJMVBOLCKYUNQPTEWZDXFGRHIA' <ciphertext.txt> plaintext.txt
[02/17/24]seed@VM:~$ cat plaintext.txt
COMPUTER SCIENCE IS RAPIDLY CHANGING THE WORLD WITH NEW DEVELOPMENTS HAPPENING EVERY DAY A
RIGOROUS EDUCATION COMBINING THE THEORY OF INFORMATION AND COMPUTATION WITH HANDSON SYSTEMS
 AND SOFTWARE DESIGN IS THE KEY TO SUCCESS AS ONE OF THE OLDEST COMPUTER SCIENCE DEPARTMENT
S IN THE CHICAGO AREA THE CS DEPARTMENT AT IIT HAS A LONG HISTORY OF MEETING THIS CHALLENGE
 THROUGH QUALITY EDUCATION IN SMALL CLASSROOM ENVIRONMENTS ALONG WITH INTERNSHIP AND RESEAR
CH OPPORTUNITIES IN INDUSTRY AND NATIONAL LABORATORIES
IIT STUDENTS WORK WITH OUR FACULTY ON WORLDCLASS RESEARCH IN AREAS THAT INCLUDE DATA SCIENC
E DISTRIBUTED SYSTEMS INFORMATION RETRIEVAL COMPUTER NETWORKING INTELLIGENT INFORMATION SYS
TEMS AND ALGORITHMS
THE DEPARTMENT OFFERS BACHELOR OF SCIENCE MASTER OF SCIENCE PROFESSIONAL MASTER AND PHD DEG
REES PLUS GRADUATE CERTIFICATES ACCELERATED COURSES AND NONDEGREE STUDY PARTTIME STUDENTS C
AN TAKE EVENING CLASSES AND LONGDISTANCE STUDENTS CAN EARN MASTERS DEGREES ONLINE STUDENTS
RATE OUR TEACHING AS AMONG THE BEST AT THE UNIVERSITY AND OUR FACULTY HAVE WON NUMEROUS TEA
CHING AWARDS
THE SECRET SENTENCE IS GOOD JOB GUYS
[02/17/24]seed@VM:~$
```

2.2 Task 2: Encryption using Different Ciphers and Modes

In this task, we will play with various encryption algorithms and modes. You can use the following openssl enc command to encrypt/decrypt a file. To see the manuals, you can type `man openssl` and `man enc`.

```
$ openssl enc ciphertype -e  -in plain.txt -out cipher.bin \
                    -K  00010203040506070809aabbccddeeff \
                    -iv 0a0b0c0d0e0f010203040506070809
```

Please replace the ciphertype with a specific cipher type, such as -aes-128-cbc, -aes-128-cfb, -bf-cbc, etc. In this task, you should try at least 3 different ciphers and three different modes. You can find the meaning of the command-line options and all the supported cipher types by typing `man enc`. We include some common options for the `openssl enc` command in the following:

```
-in <file>      input file
-out <file>     output file
-e              encrypt
-d              decrypt
-K/-iv          key/iv in hex is the next argument
-[pP]           print the iv/key (then exit if -P)
```

- To create a plaintext file, e.g., `plain.txt`, you can the following commands:

```
$ touch plain.txt
$ gedit plain.txt
```

- Since we did not cover the Cipher Feedback (CFB) and the Output Feedback (OFB) in class, you need to read Block cipher mode of operation.
- To observe the encrypted contents of a file, e.g., `cipher.bin`, in hexadecimal format, use the command line hex viewing tool `xxd`.

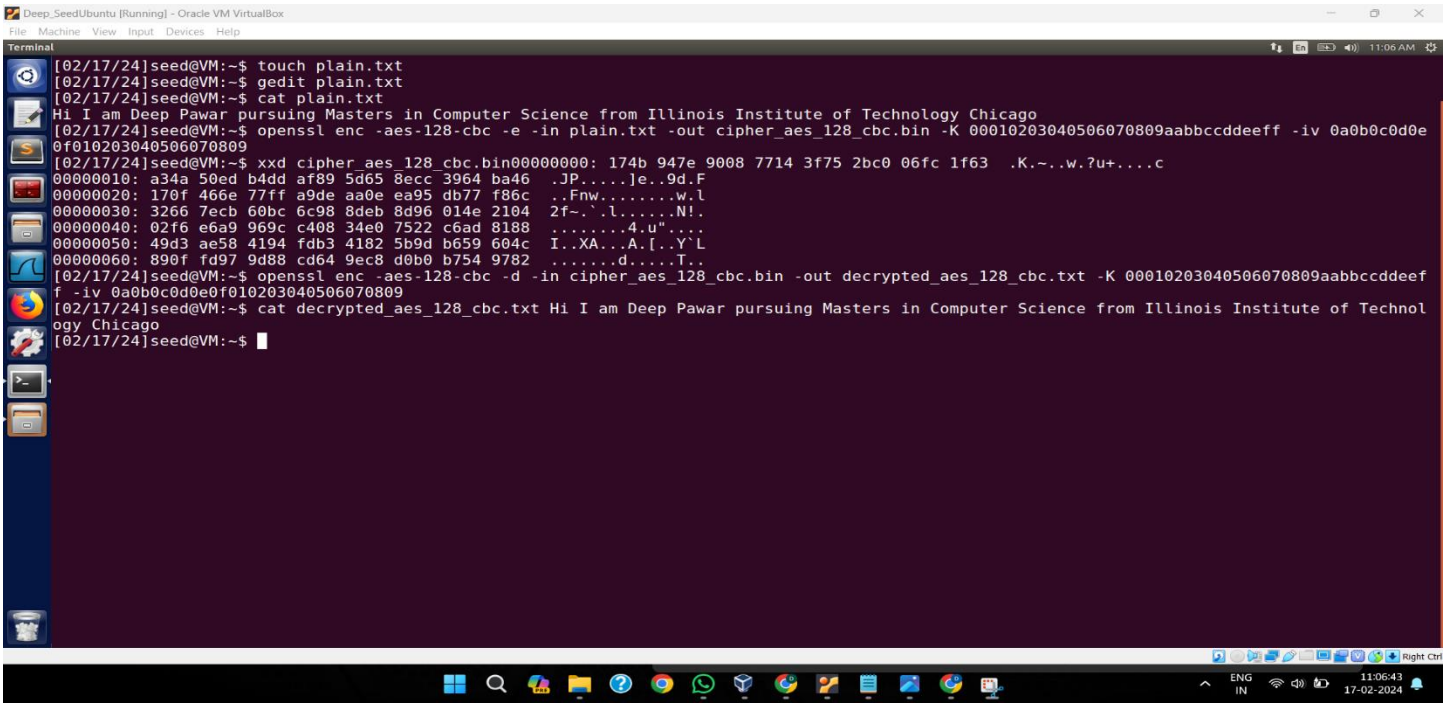
```
$ xxd cipher.bin
```

Ans:

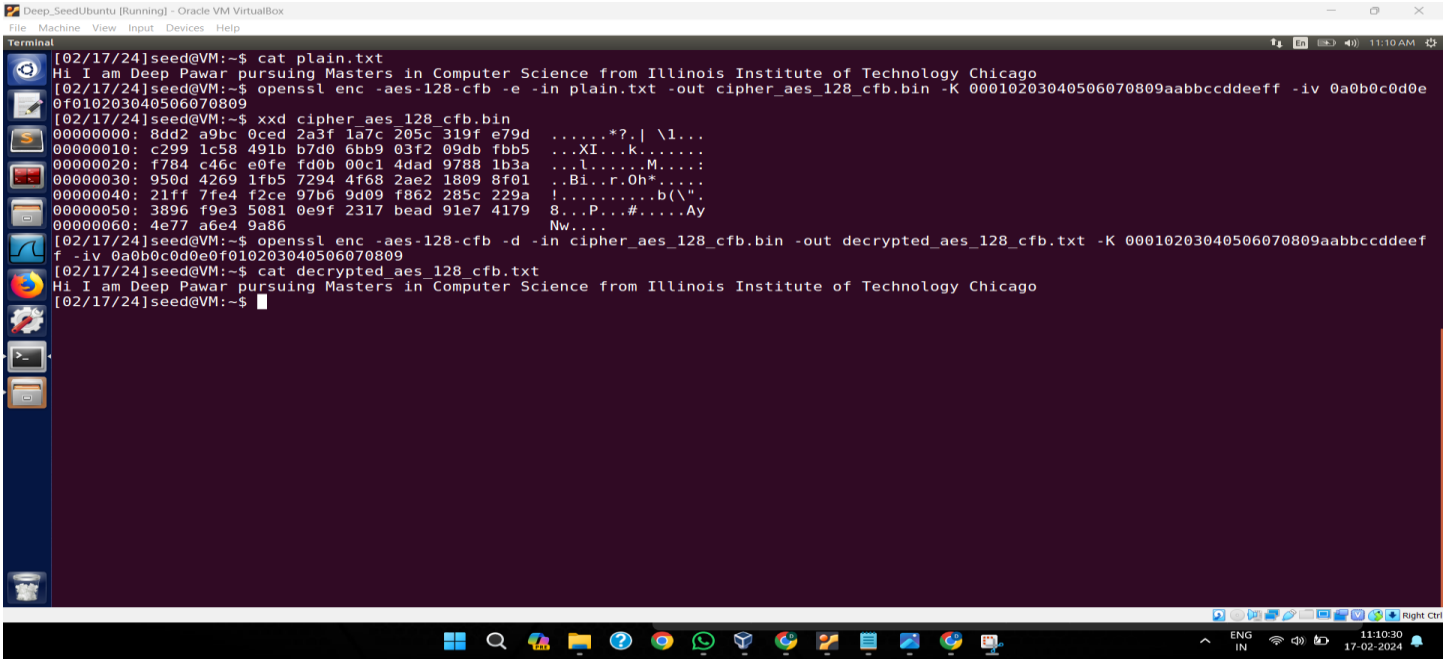
In this task, we need to use the openssl enc command with different cipher types and modes. The steps are as follows:

- Choose three different cipher types from the OpenSSL-supported ciphers, such as **AES-128, BF,** and **DES**
- Choose three different modes of operation for each cipher, such as **CBC, CFB,** and **OFB**.
- Generate a plaintext file (e.g., plain.txt) with some content.
- Encrypt the plaintext file using each cipher and mode combination.
- To observe the encrypted contents of a file in hexadecimal format, use the command line hex viewing tool **xxd**
- Decrypt the ciphertext file using each cipher and mode combination.

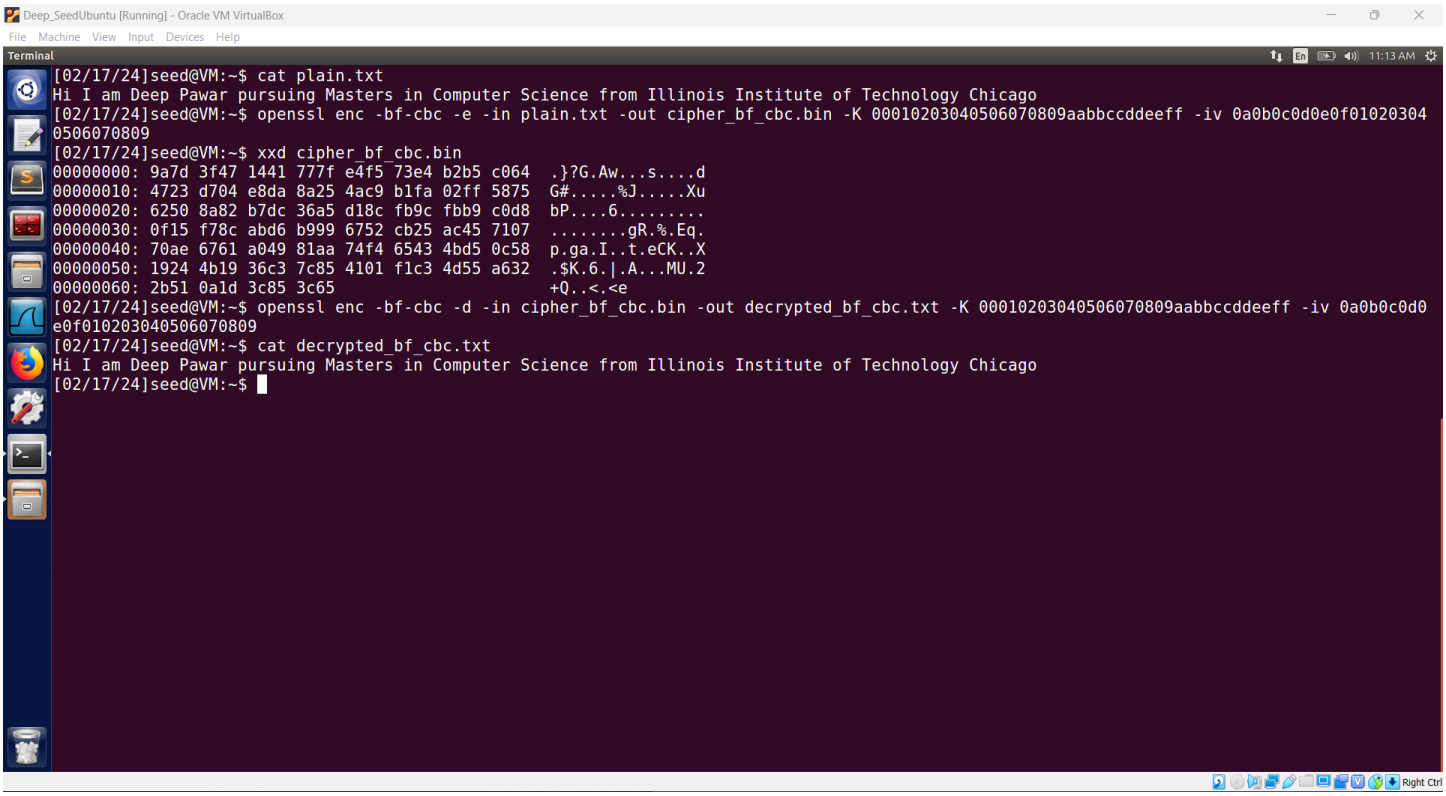
1. -aes-128- cbc



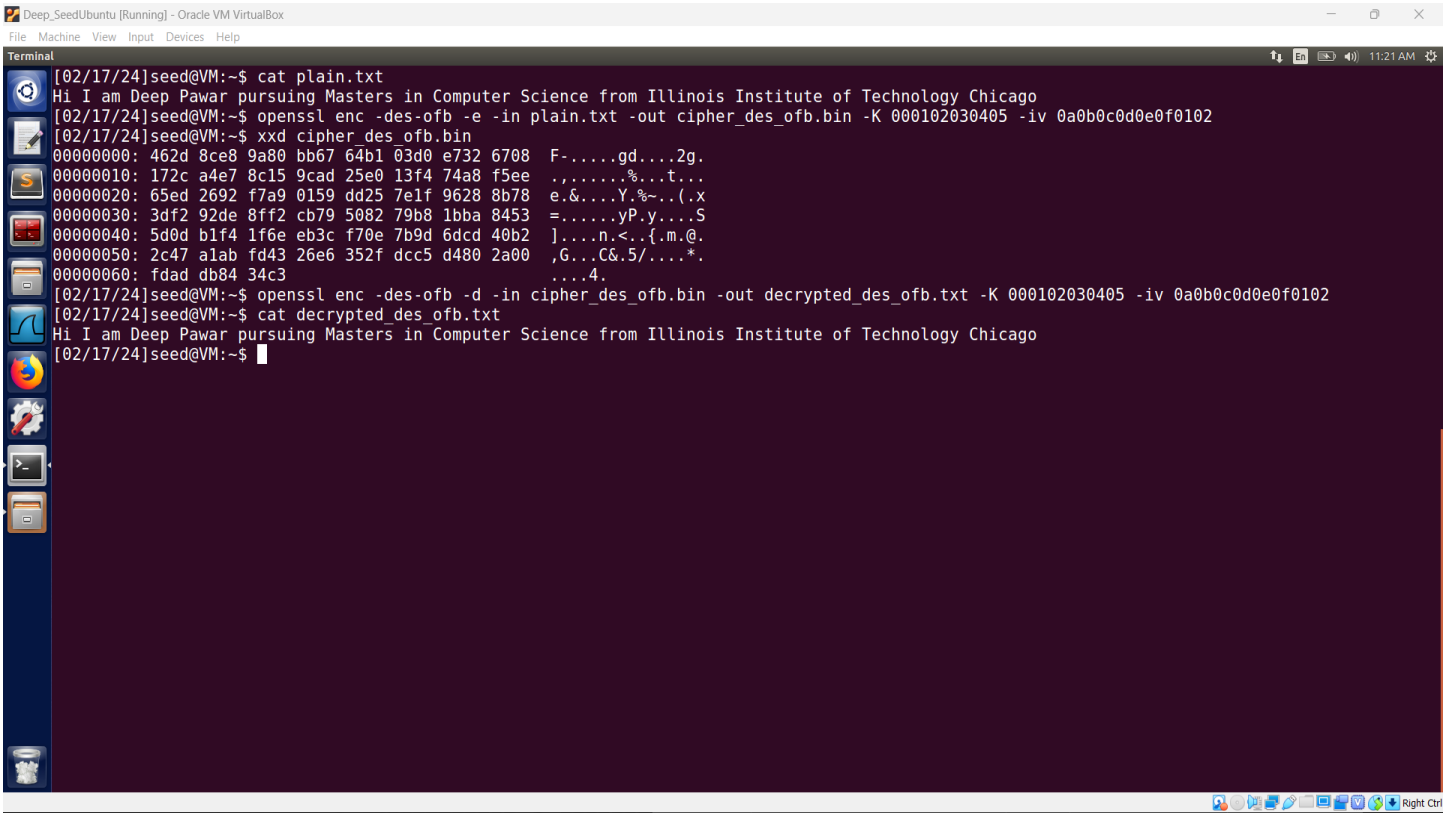
2. -aes-128-cfb



3. -bf-cbc



4. -des-ofb



2.3 Task 3: Encryption Mode – ECB vs. CBC

The file pic original.bmp can be downloaded from course Blackboard, and it contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. However, For the .bmp file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate .bmp file. We will replace the header of the encrypted picture with that of the original picture. You can use Bless a hex editor tool (already installed on our VM) to directly modify binary files. We can also use the following commands to get the header from p1.bmp, and the data from p2.bmp (from offset 55 to the end of the file), and then combine the header and data into a new file.

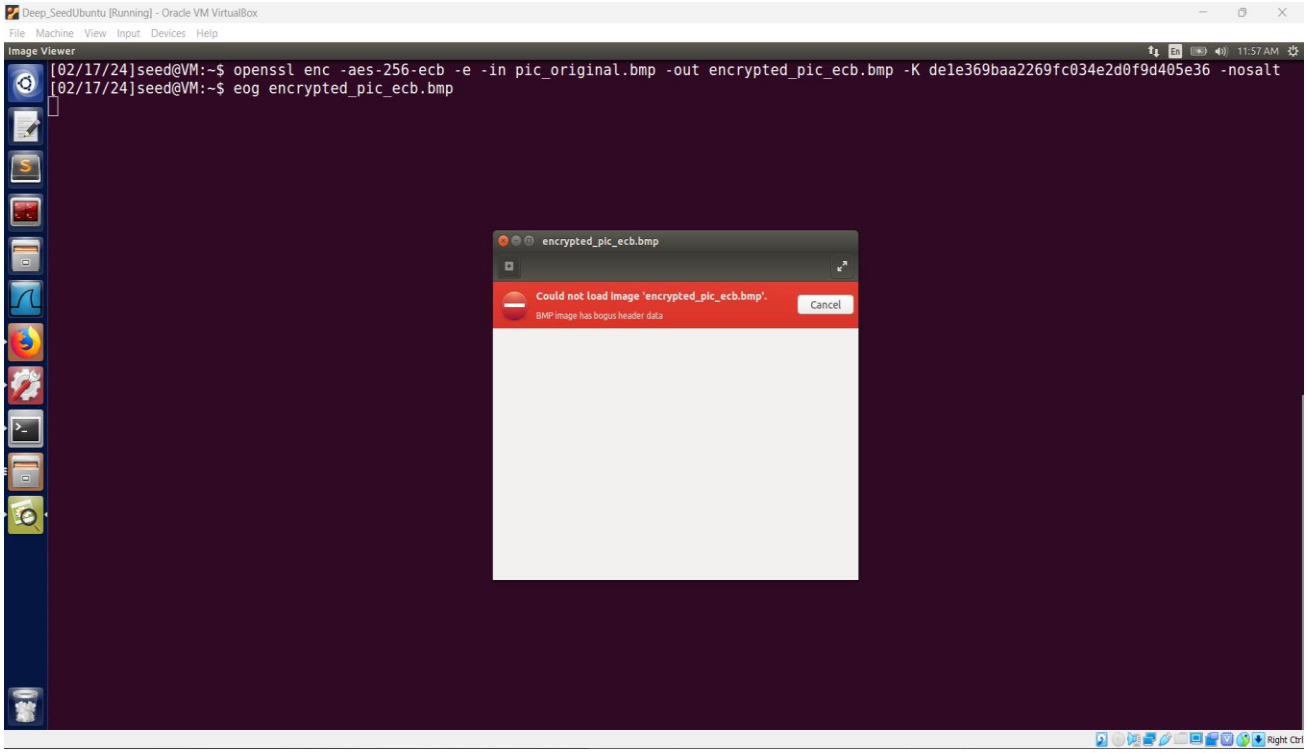
```
$ head -c 54 p1.bmp > header
$ tail -c +55 p2.bmp > body
$ cat header body > new.bmp
```

2. Display the encrypted picture using a picture viewing program (we have installed an image viewer program called eog on our VM). Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

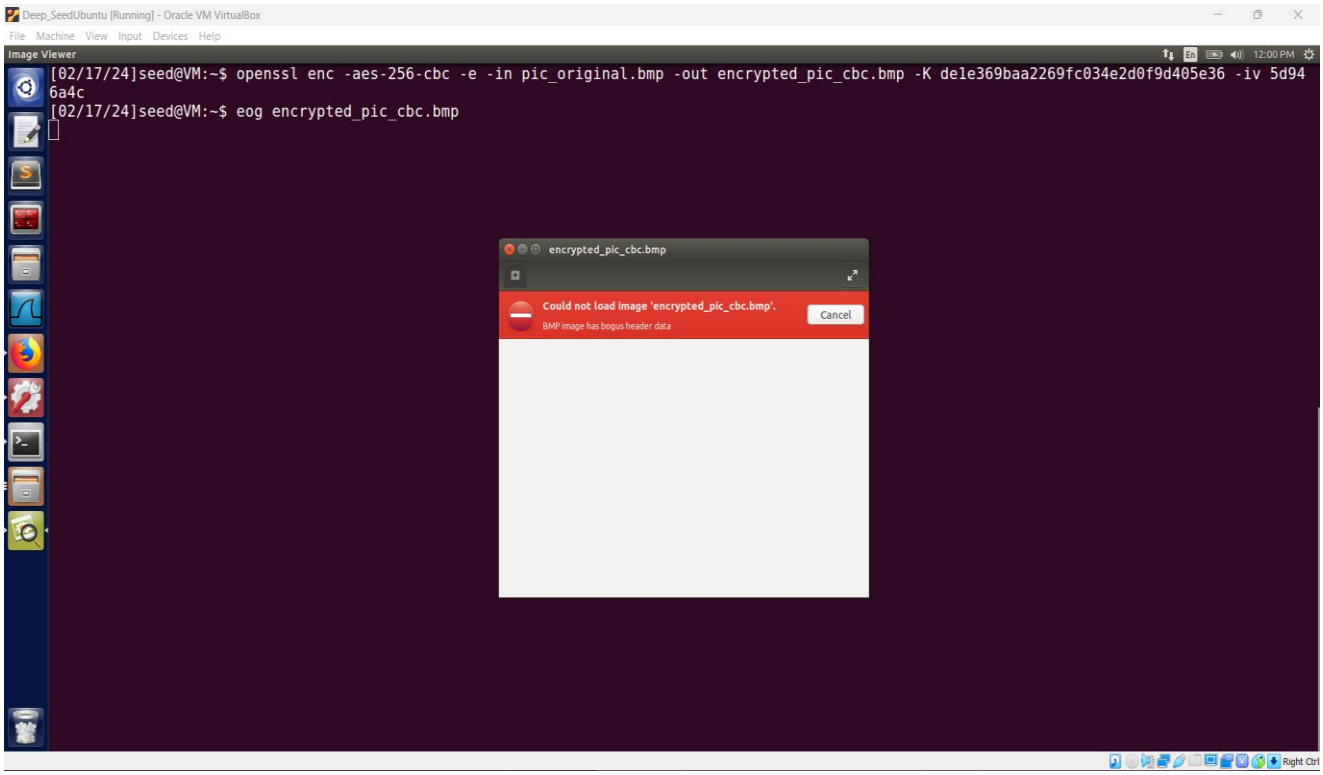
Ans:

- To complete this task, first I need to encrypt the original.bmp file using both ECB and CBC modes. Then, I'll replace the header of the encrypted pictures with the header of the original picture to make them viewable. And finally, we'll display the encrypted pictures and analyze any observable differences.
- The steps are as follows:

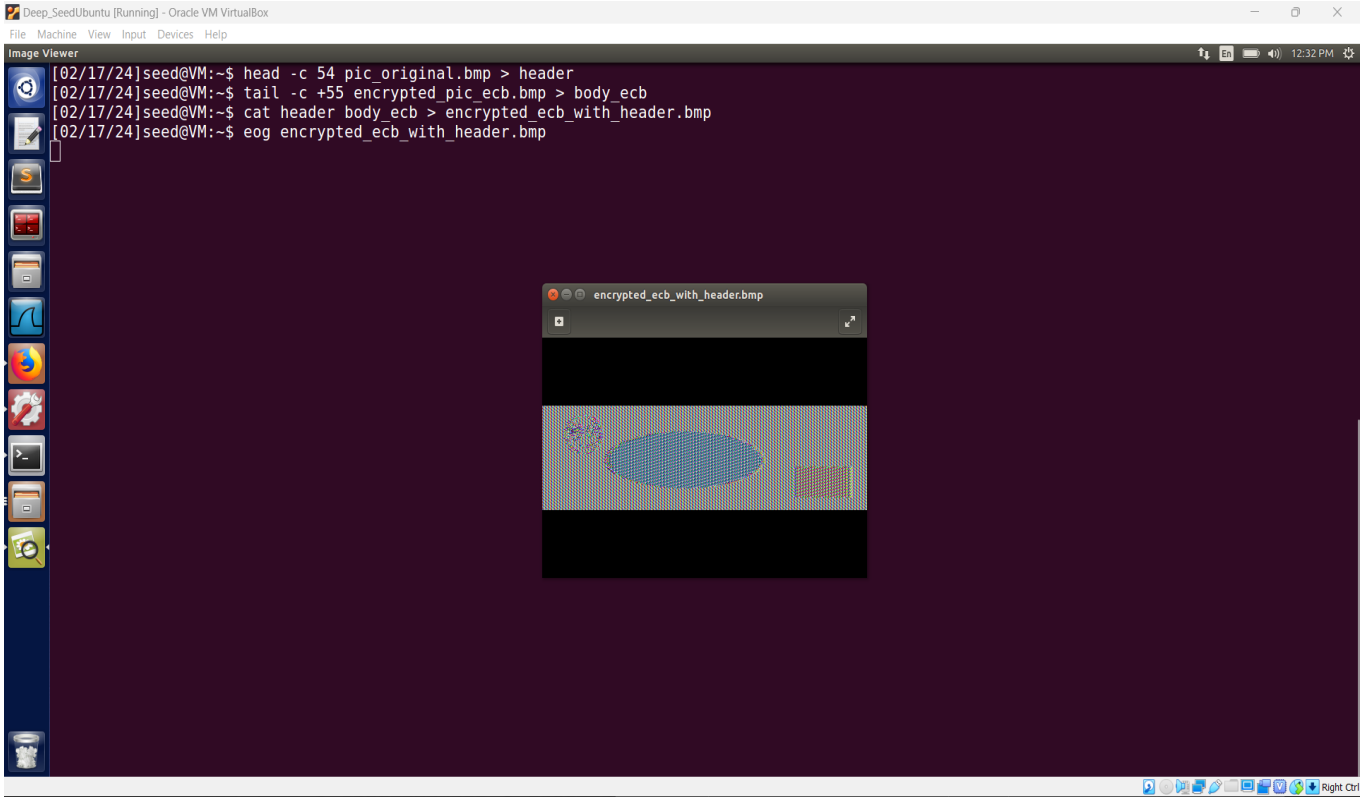
1. Encrypt the pic_original.bmp file using ECB mode and display the encrypted picture using an image viewer **eog** before replacing the header information to see if we can be able to see the encrypted image correctly or not.
2. As we can see we cannot able to see the encrypted image because the **.bmp** file contains the first 54 bytes in the header information about the picture which we need to replace from the original image to make it visible.



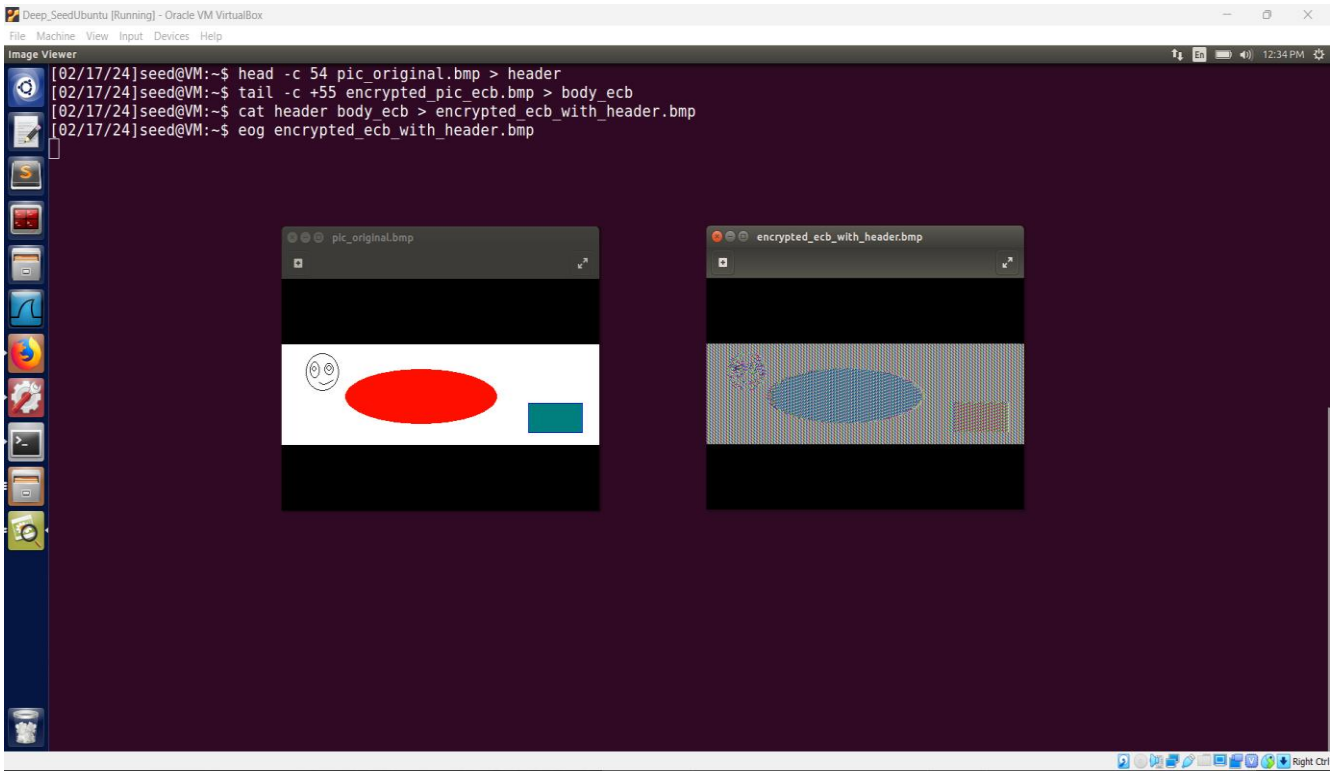
3. Repeat the above process for CBC mode:



4. Replace the header of the encrypted pictures with the header of the original picture for ECB mode to make it visible and display the encrypted pictures using an image viewer:



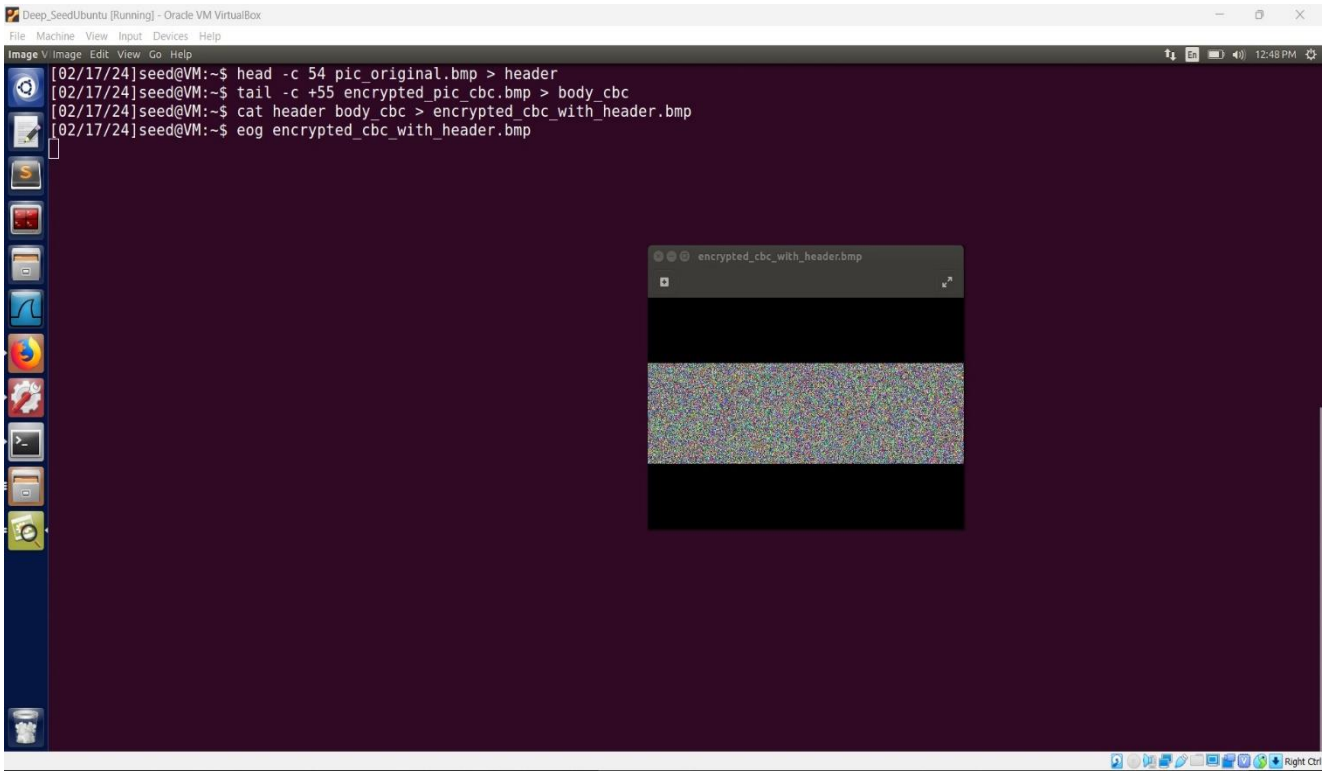
5. Compare the original image with the ECB encrypted image with the added header:



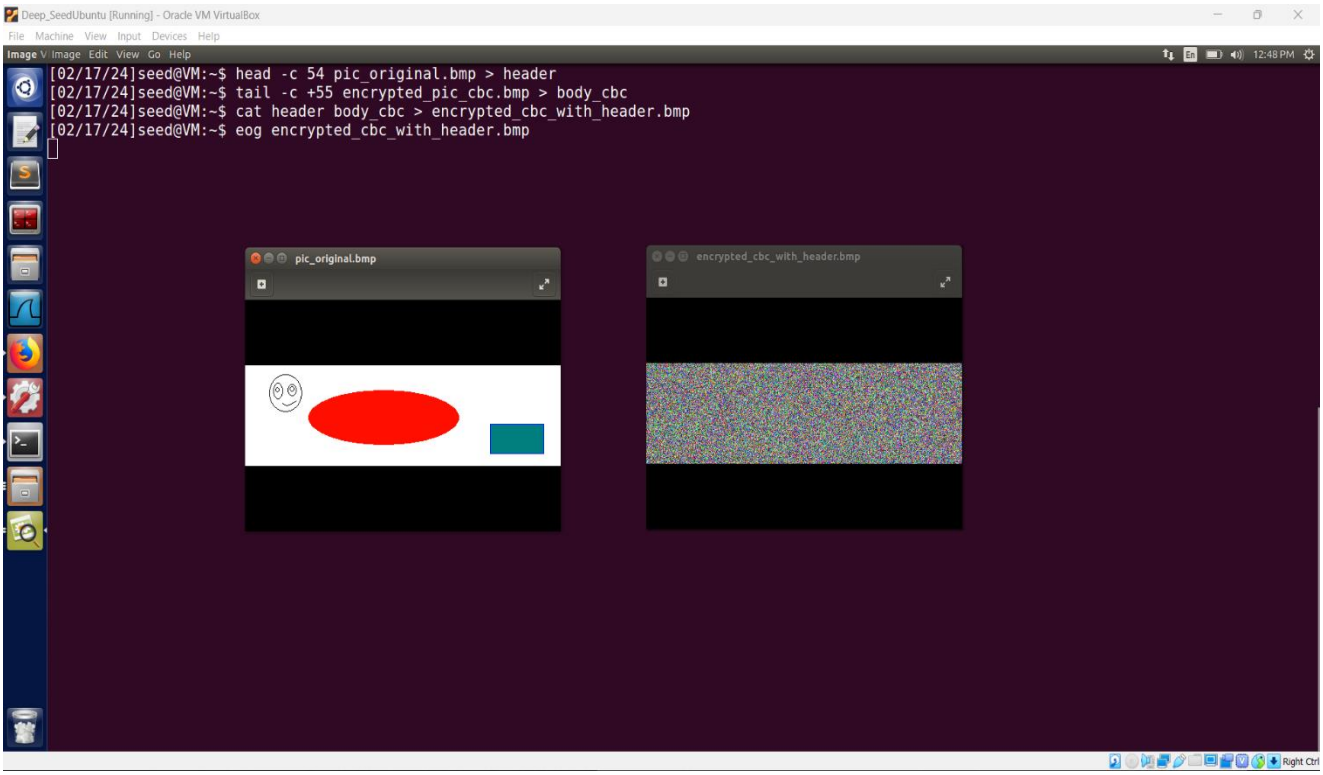
6. Observations:

- Since ECB mode encrypts each block of plaintext independently, identical blocks in the original image will result in identical blocks in the encrypted image.
- As the original image contains some repeating elements, some of these patterns of the original image are still visible in the encrypted image like the shapes and drawing of the face.
- As an observer, I can detect the presence of certain features or patterns in the encrypted image based on the repeating blocks or patterns in the original image.
- Since ECB mode lacks diffusion, slight modifications to the plaintext won't have a big impact on the ciphertext. Since identical blocks of plaintext will result in identical blocks of ciphertext, this lack of diffusion may cause weaknesses. This could potentially leak information about the content of the original image which is not recommended.

7. Replace the header of the encrypted pictures with the header of the original picture for CBC mode to make it visible and display the encrypted picture using an image viewer:



8. Compare the original image with the CBC encrypted image with the added header:

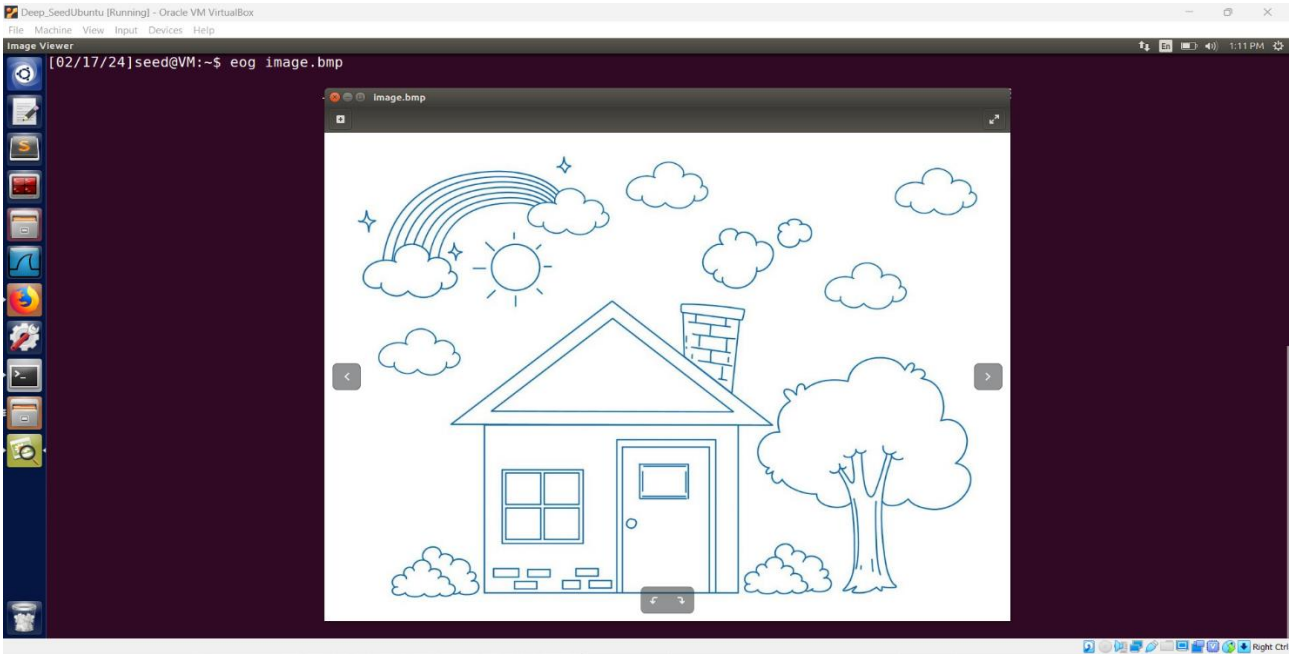


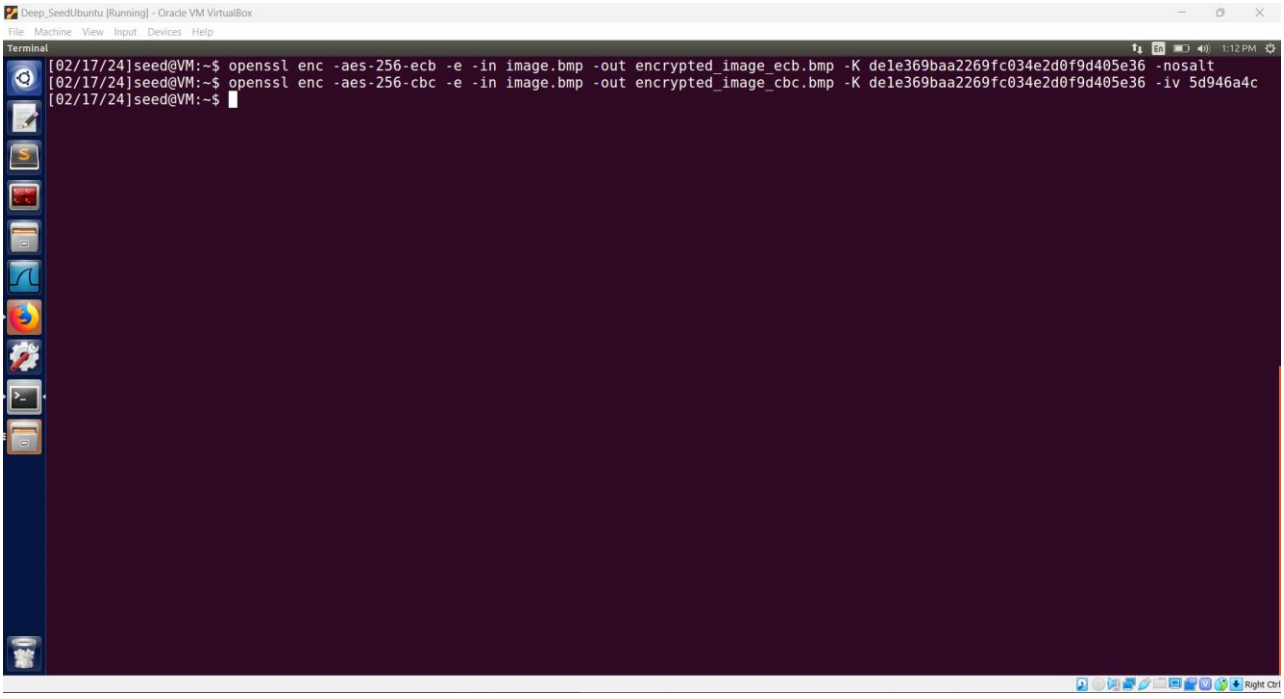
9. Observations:

- Before encryption, each plaintext block in CBC mode is XORed with the preceding ciphertext block to add randomness and diffusion to the encryption process.
- As an observer, I cannot be able to detect any information or patterns in the encrypted image of the original image which makes CBC more secure as compared to ECB.
- Compared to ECB, there is a lower chance that patterns from the original image will remain intact in the CBC encrypted image since randomness is introduced during the XOR operation with the prior ciphertext block. CBC mode offers more protection against disclosing specific details about the original image than ECB mode.

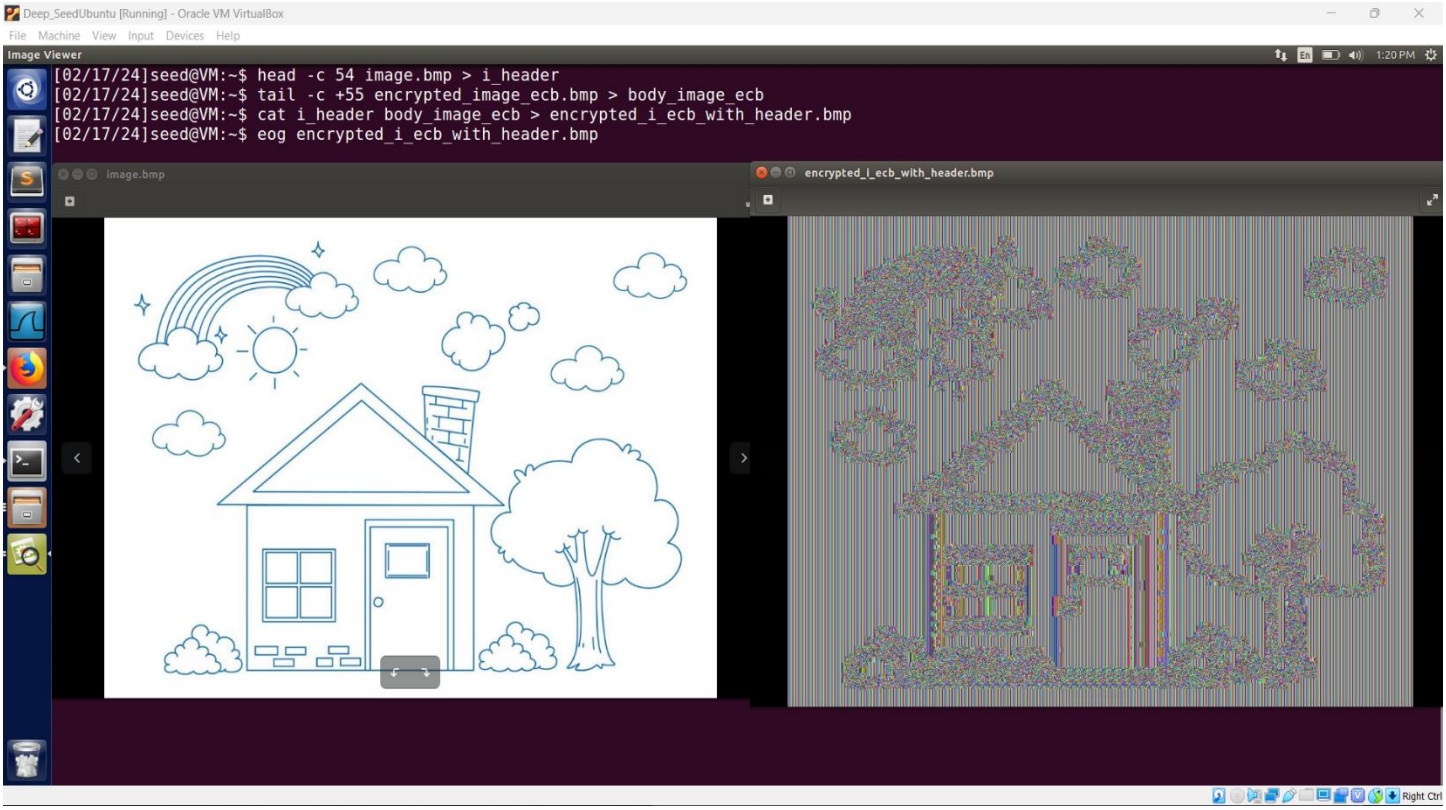
2.3.1 Select a picture of your choice, repeat the experiment above, and report your observations.

1. Encrypt the pic image.bmp file using both ECB and CBC mode.





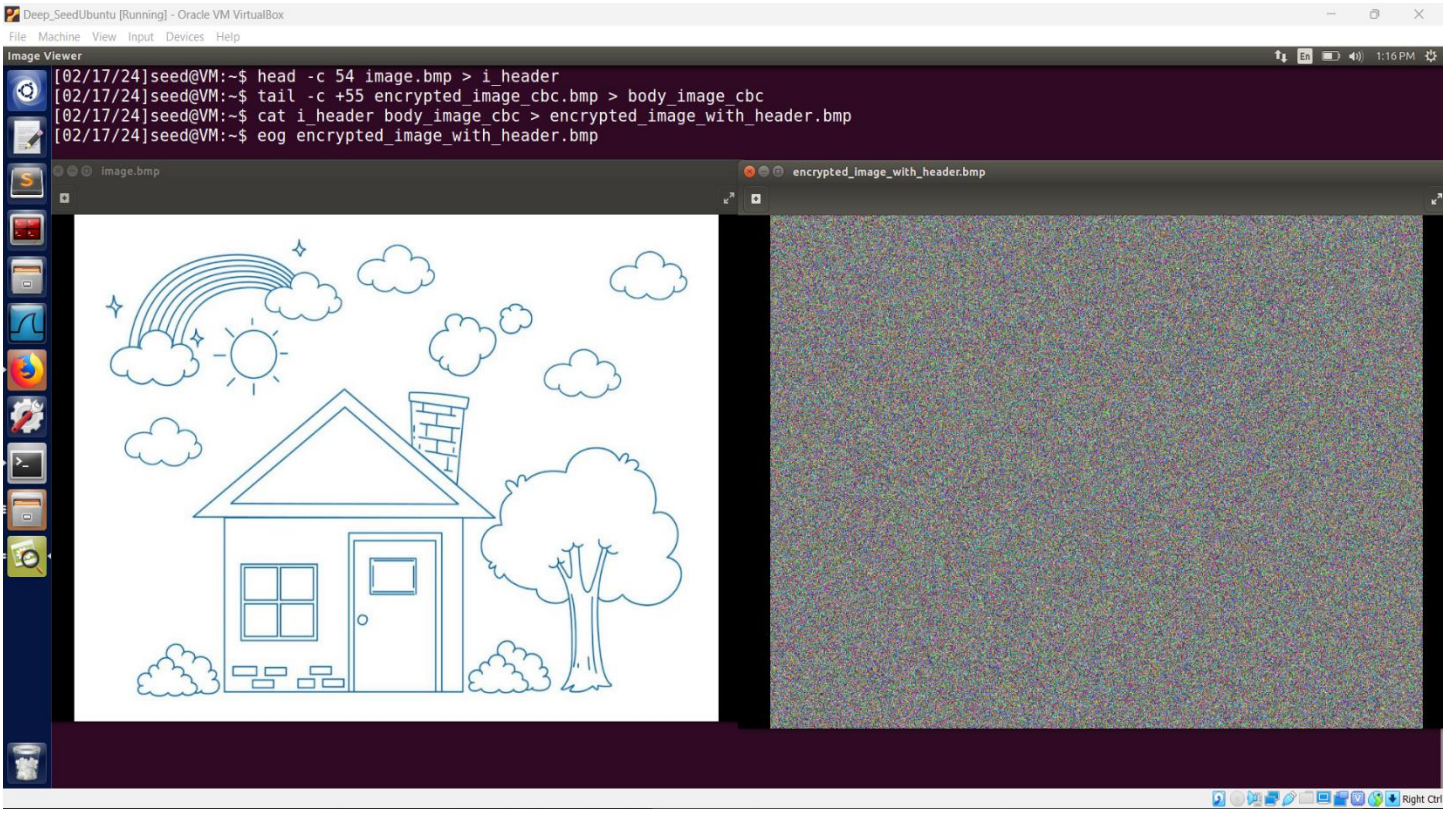
2. Replace the header of the encrypted pictures with the header of the original picture for ECB mode to make it visible and display the encrypted pictures using an image viewer:



3. Observations:

- Since ECB mode encrypts each block of plaintext independently, identical blocks in the original image will result in identical blocks in the encrypted image.
- As the original image contains some repeating elements, some of these patterns of the original image are still visible in the encrypted image like the shapes and pattern of the house and surroundings. As an observer, I can detect the presence of certain features or patterns in the encrypted image.
- Since ECB mode lacks diffusion, slight modifications to the plaintext won't have a big impact on the ciphertext. Since identical blocks of plaintext will result in identical blocks of ciphertext, this lack of diffusion may cause weaknesses. This could potentially leak information about the content of the original image which is not recommended.

4. Replace the header of the encrypted pictures with the header of the original picture for CBC mode to make it visible and display the encrypted picture using an image viewer:



5. Observations:

- Before encryption, each plaintext block in CBC mode is XORed with the preceding ciphertext block to add randomness and diffusion to the encryption process.
- As an observer, I cannot be able to detect any information or patterns in the encrypted image of the original image which makes CBC more secure as compared to ECB.
- Compared to ECB, there is a lower chance that patterns from the original image will remain intact in the CBC encrypted image since randomness is introduced during the XOR operation with the prior ciphertext block. CBC mode offers more protection against disclosing specific details about the original image than ECB mode.

2.4 Task 4: Padding

For block ciphers, when the size of a plaintext is not a multiple of the block size, padding may be required. All the block ciphers normally use PKCS#5 padding, which is known as standard block padding. We will conduct the following experiments to understand how this type of padding works:

1. Use ECB, CBC, CFB, and OFB modes to encrypt a file (you can pick any cipher). Please report which modes have paddings and which ones do not. For those that do not need paddings, please explain why.
2. Let us create three files, which contain 5 bytes, 10 bytes, and 16 bytes, respectively. We can use the following `echo -n` command to create such files. The following example creates a file `f1.txt` with length 5 (without the `-n` option, the length will be 6 because a newline character will be added by `echo`):

```
$ echo -n "12345" > f1.txt
```

We then use `openssl enc -aes-128-cbc -e` to encrypt these three files using 128-bit AES with CBC mode. Please describe the size of the encrypted files.

We would like to see what is added to the padding during the encryption. To achieve this goal, we will decrypt these files using `openssl enc -aes-128-cbc -d`. Unfortunately, decryption by default will automatically remove the padding, making it impossible for us to see the padding. However, the command does have an option called `-nopad`, which disables the padding, i.e., during the decryption, the command will not remove the padded data. Therefore, by looking at the decrypted data, we can see what data are used in the padding. Please use this technique to figure out what paddings are added to the three files.

It should be noted that padding data may not be printable, so you need to use a hex tool to display the content. The following example shows how to display a file in the hex format:

```
$ hexdump -C p1.txt
00000000 31 32 33 34 35 36 37 38 39 49 4a 4b 4c 0a.          |123456789IJKL.|
$ xxd p1.txt
00000000: 3132 3334 3536 3738 3949 4a4b 4c0a.                123456789IJKL.
```

Ans:

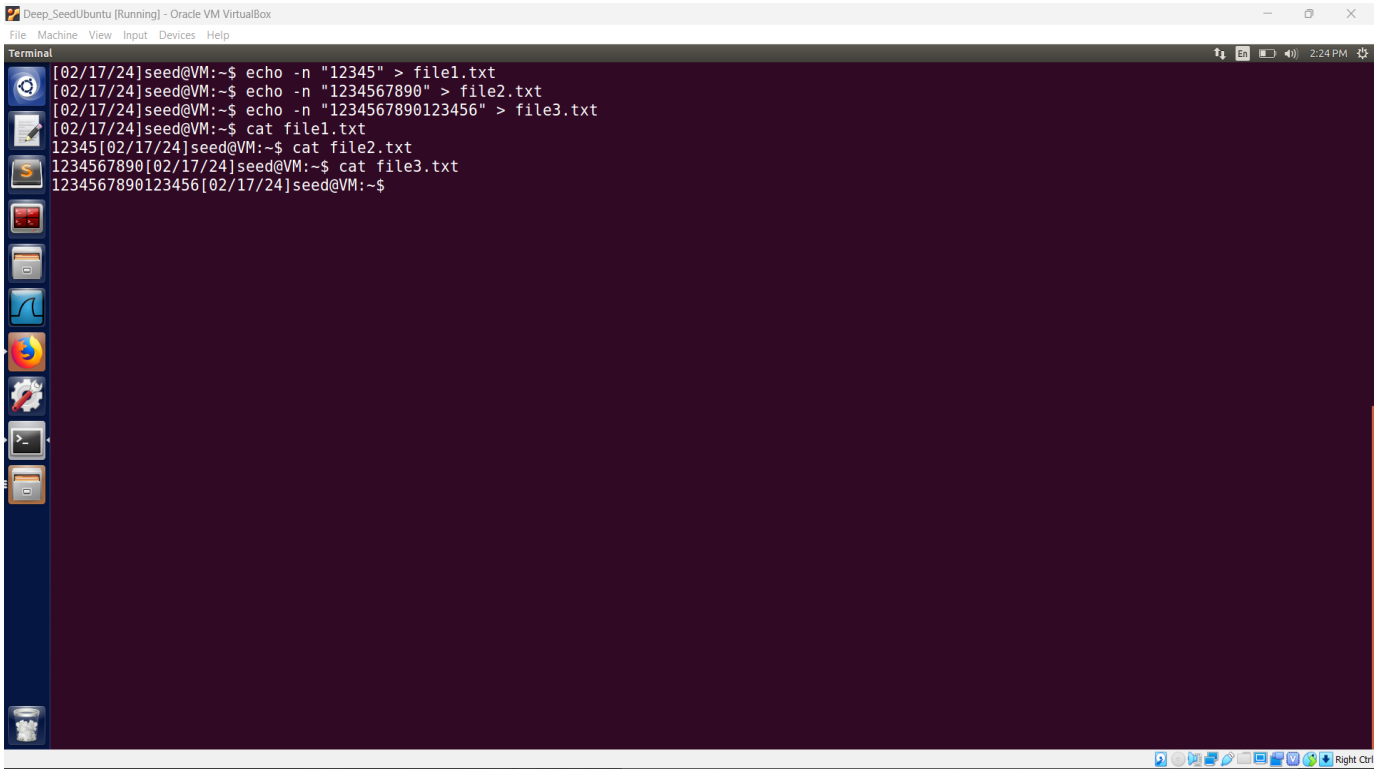
To conduct the experiments as described in the task, I will encrypt files using various block cipher modes (ECB, CBC, CFB, and OFB) and observe which modes require padding and which do not. Then, I will decrypt files encrypted using the above modes to examine the padding added during encryption.

The steps are as follows:

- Generate three files with lengths of 5 bytes, 10 bytes, and 16 bytes, respectively.
- Encrypt files using **ECB, CBC, CFB,** and **OFB** modes with a chosen block cipher (e.g., **AES**). Describe the size of the encrypted files. Determine which modes require padding and explain why.
- Decrypt the encrypted files using openssl enc -aes-128-cbc -d -nopad to disable automatic padding removal during decryption.
- Examine the decrypted data to determine the padding added during encryption, using a hex tool such as **hexdump** or **xxd**

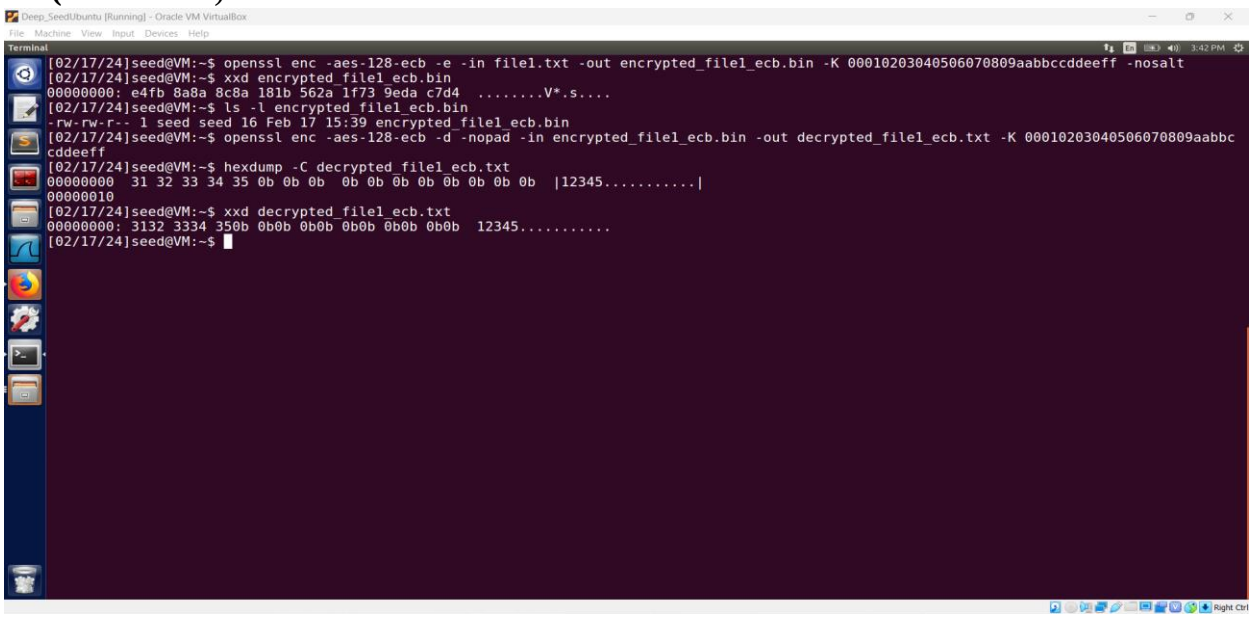
- **Steps Execution:**

1. Create three files of different lengths (5 bytes, 10 bytes, and 16 bytes) using **echo** command as follows:



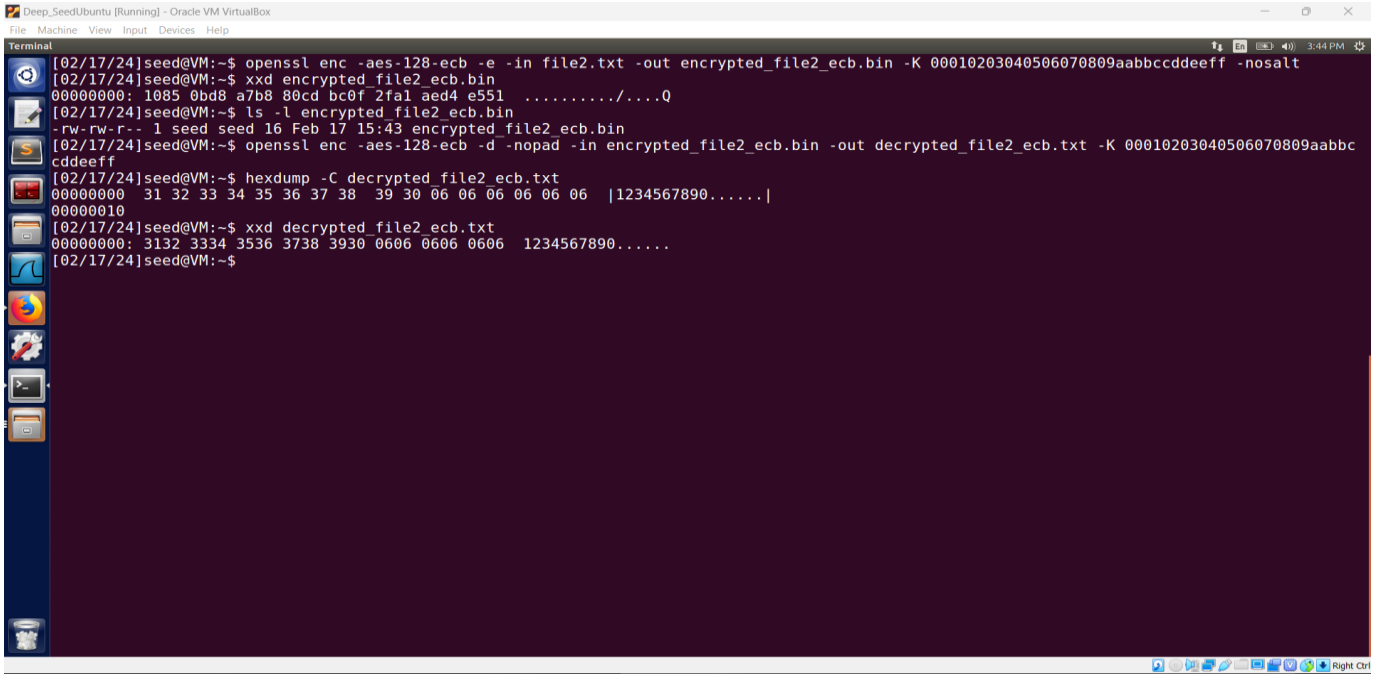
2. Encrypt file1.txt, file2.txt and file3.txt using **-aes-128-ecb** mode. Then, we'll describe the size of the encrypted files by using the **ls -l** command. Then, decrypt encrypted files using **-aes-128-ecb** with padding disabled using **-nopad** to disable automatic padding removal during decryption. And at last, examine the decrypted data to determine the padding added during encryption, using a hex tool such as **hexdump** as follows:

1. file.txt (ECB Mode)



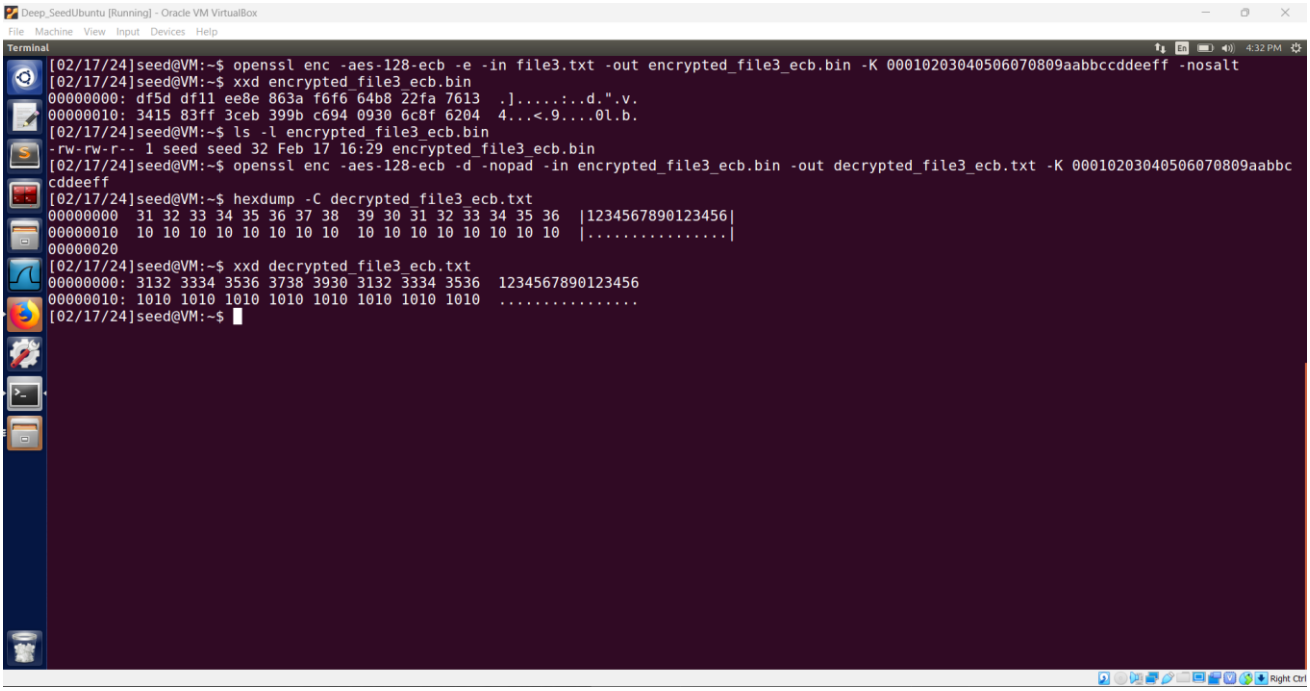
- **Observations:**
  - file1.txt size = 5 bytes
  - Size after encryption = 16 bytes
  - Padding added during encryption = 11 bytes

2. file2.txt (ECB Mode)



- **Observations:**
  - file2.txt size = 10 bytes
  - Size after encryption = 16 bytes
  - Padding added during encryption = 6 bytes

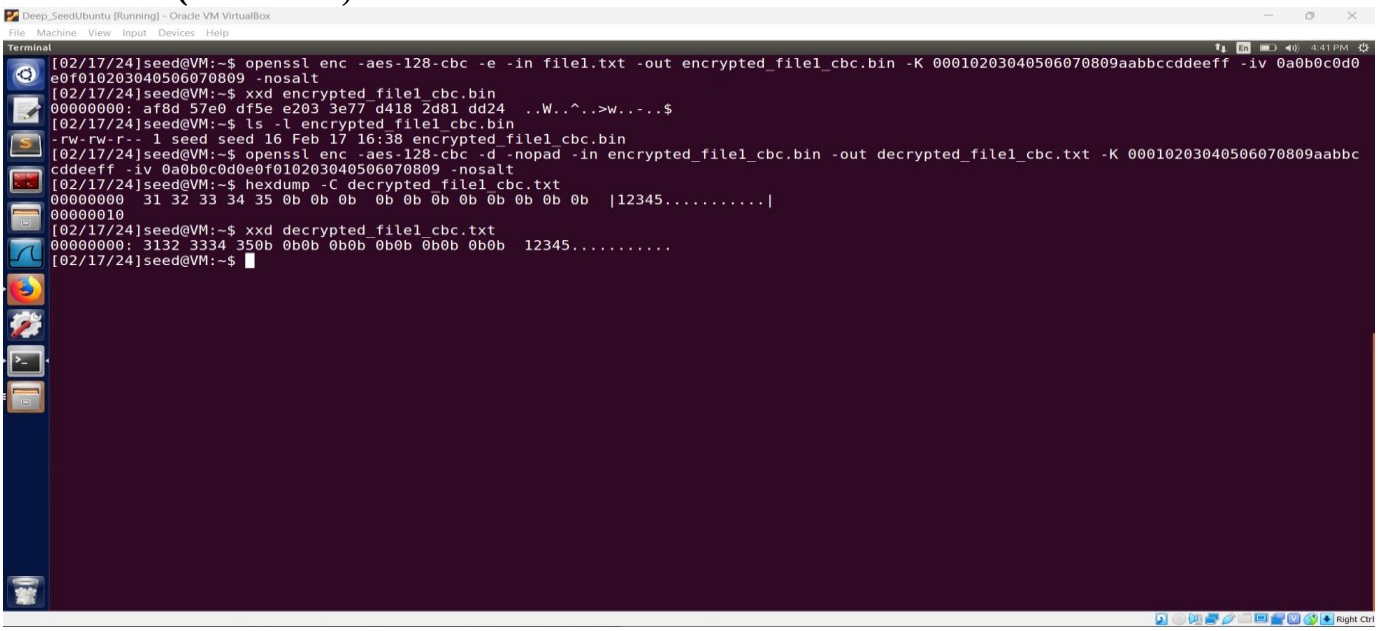
3. file3.txt (ECB Mode)



- **Observations:**
  - file3.txt size = 16 bytes
  - Size after encryption = 32 bytes
  - Padding added during encryption = 16 bytes

3. Encrypt file1.txt, file2.txt and file3.txt using **-aes-128-cbc** mode. Then, we'll describe the size of the encrypted files by using the **ls -l** command. Then, decrypt encrypted files using **-aes-128-cbc** with padding disabled using **-nopad** to disable automatic padding removal during decryption. And at last, examine the decrypted data to determine the padding added during encryption, using a hex tool such as **hexdump** as follows:

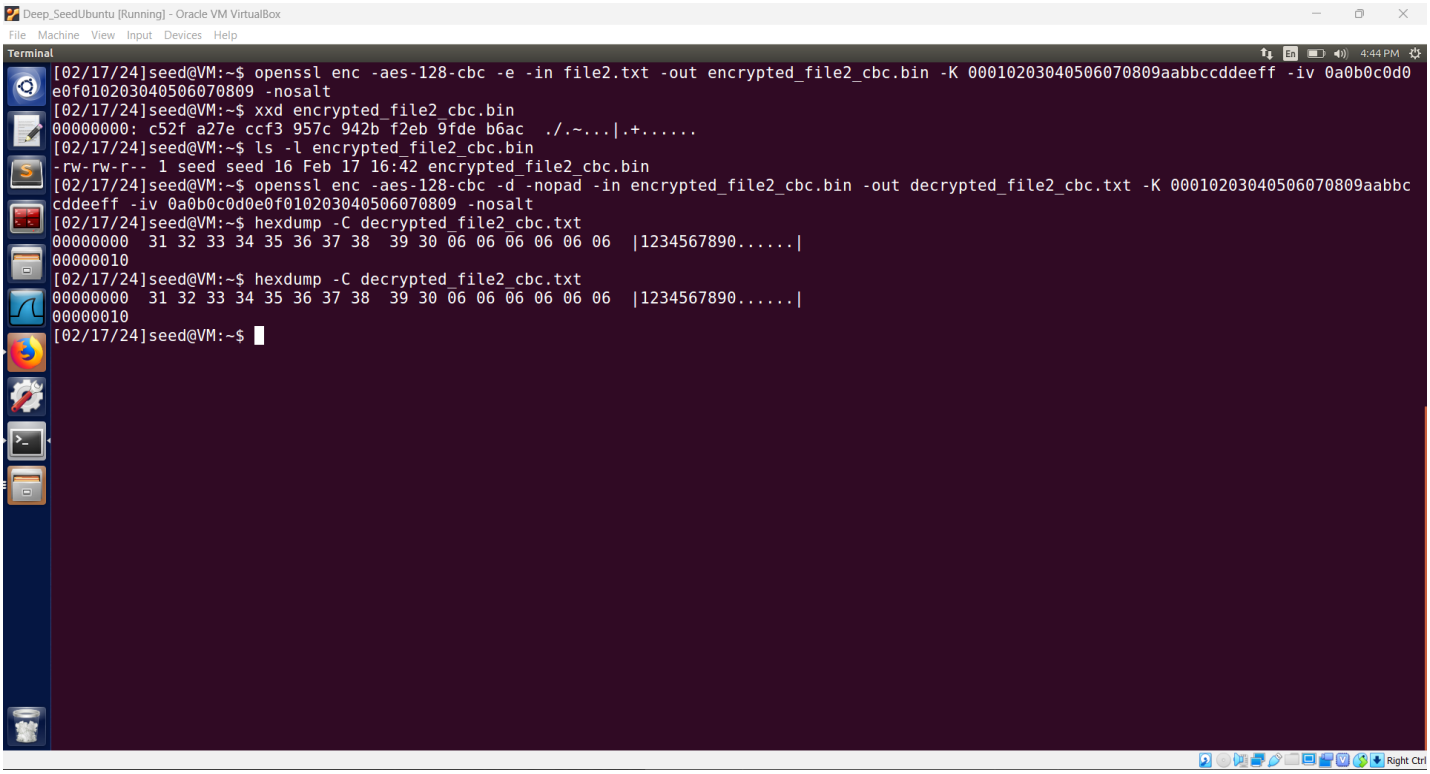
   3.1 file1.txt (CBC Mode)



- **Observations:**

  - file1.txt size = 5 bytes
  - Size after encryption = 16 bytes
  - Padding added during encryption = 11 bytes

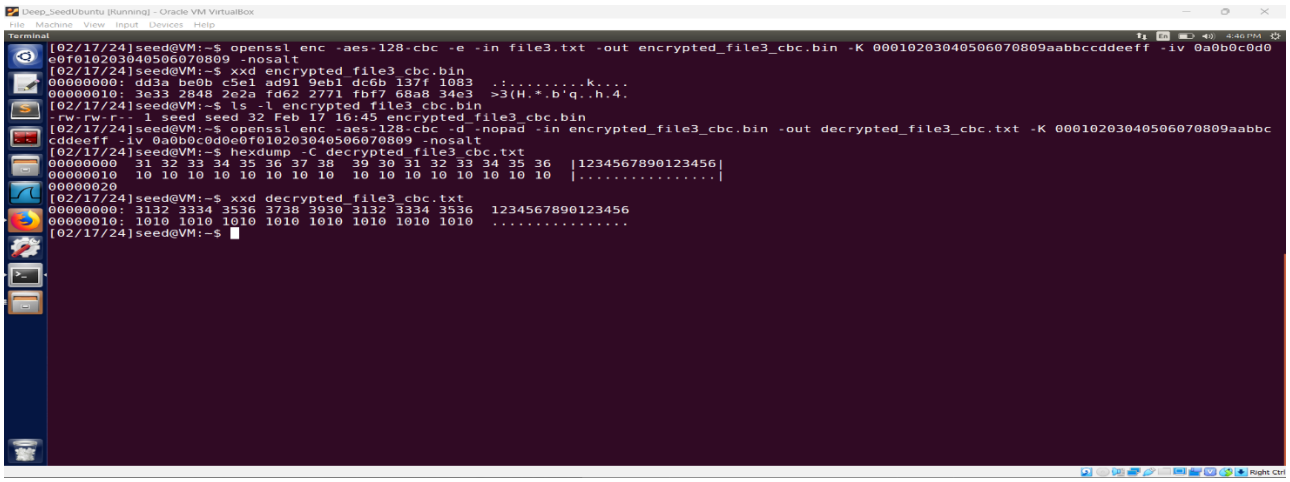
3.2 file2.txt (CBC Mode)



- **Observations:**

  - file2.txt size = 10 bytes
  - Size after encryption = 16 bytes
  - Padding added during encryption = 6 bytes

3.3 file3.txt (CBC Mode)

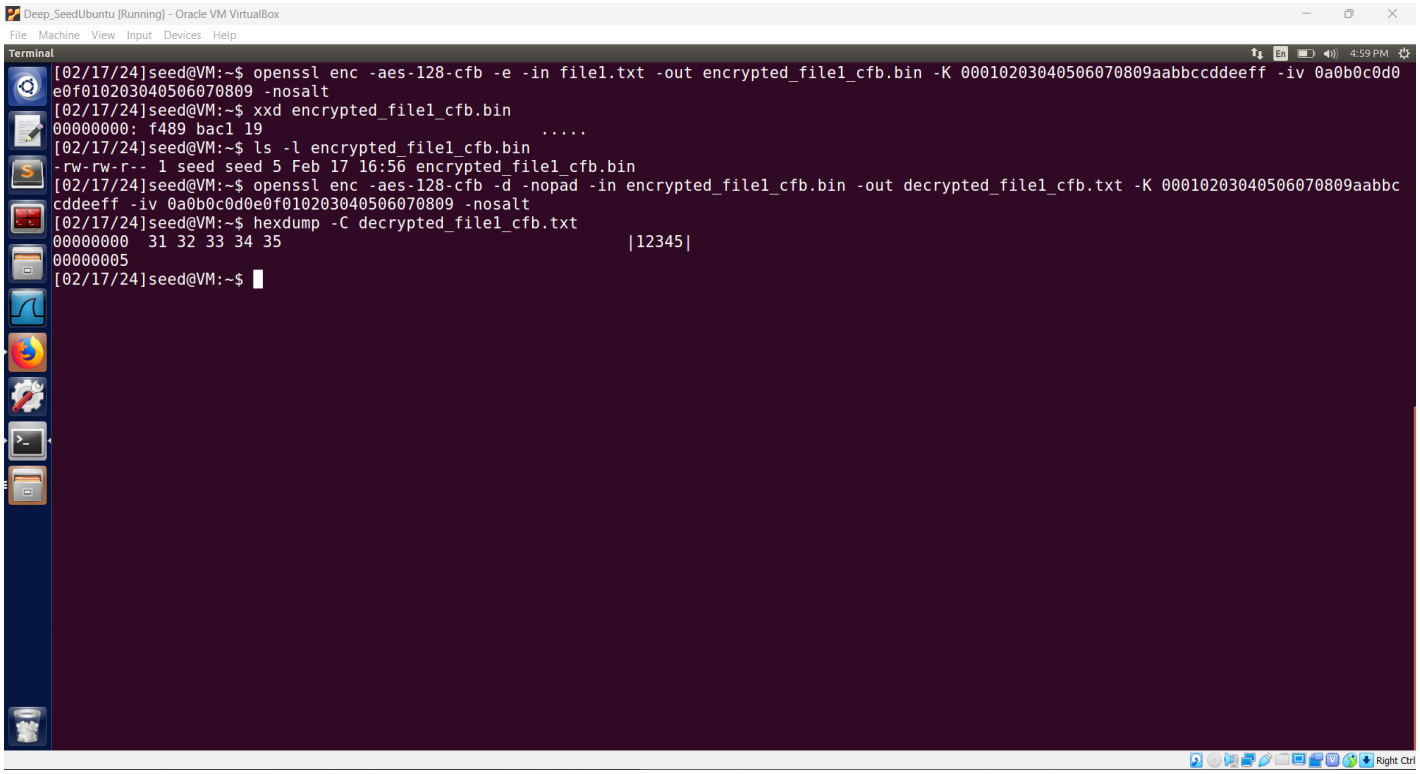


- **Observations:**

  - file3.txt size = 16 bytes
  - Size after encryption = 32 bytes
  - Padding added during encryption = 16 bytes

4. Encrypt file1.txt, file2.txt and file3.txt using **-aes-128-cfb** mode. Then, we'll describe the size of the encrypted files by using the **ls -l** command. Then, decrypt encrypted files using **-aes-128-cfb** with padding disabled using **-nopad** to disable automatic padding removal during decryption. And at last, examine the decrypted data to determine the padding added during encryption, using a hex tool such as **hexdump** as follows:

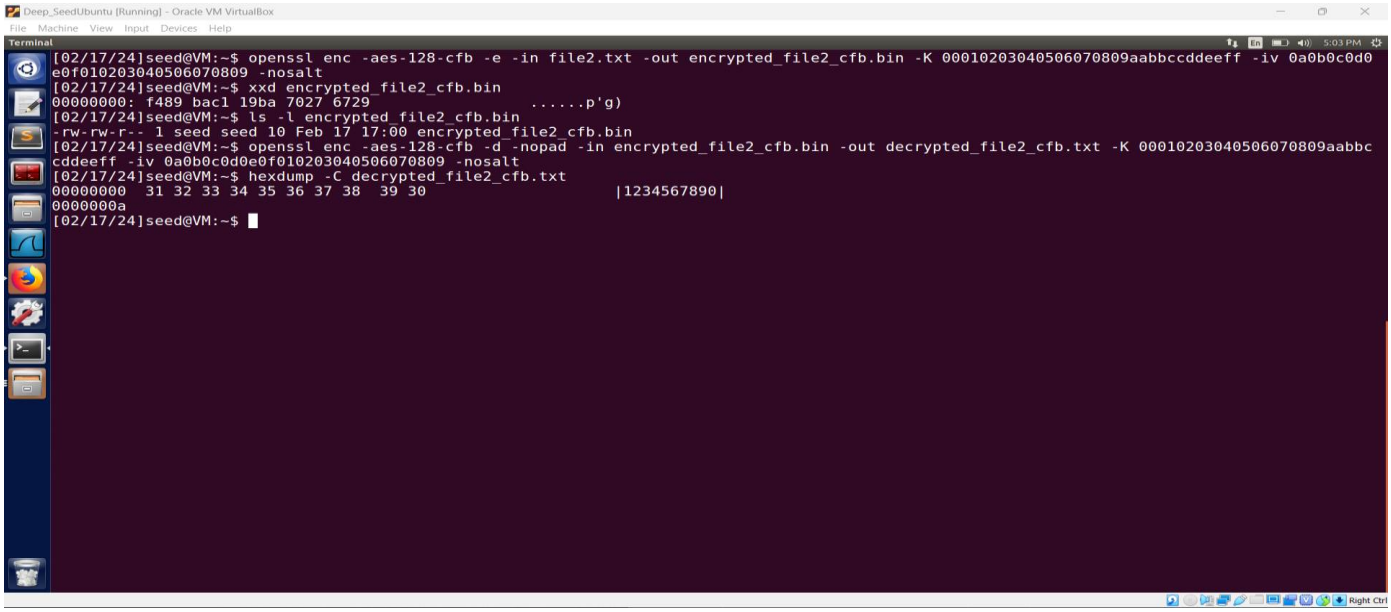
4.1 file1.txt (CFB Mode)



- **Observations:**

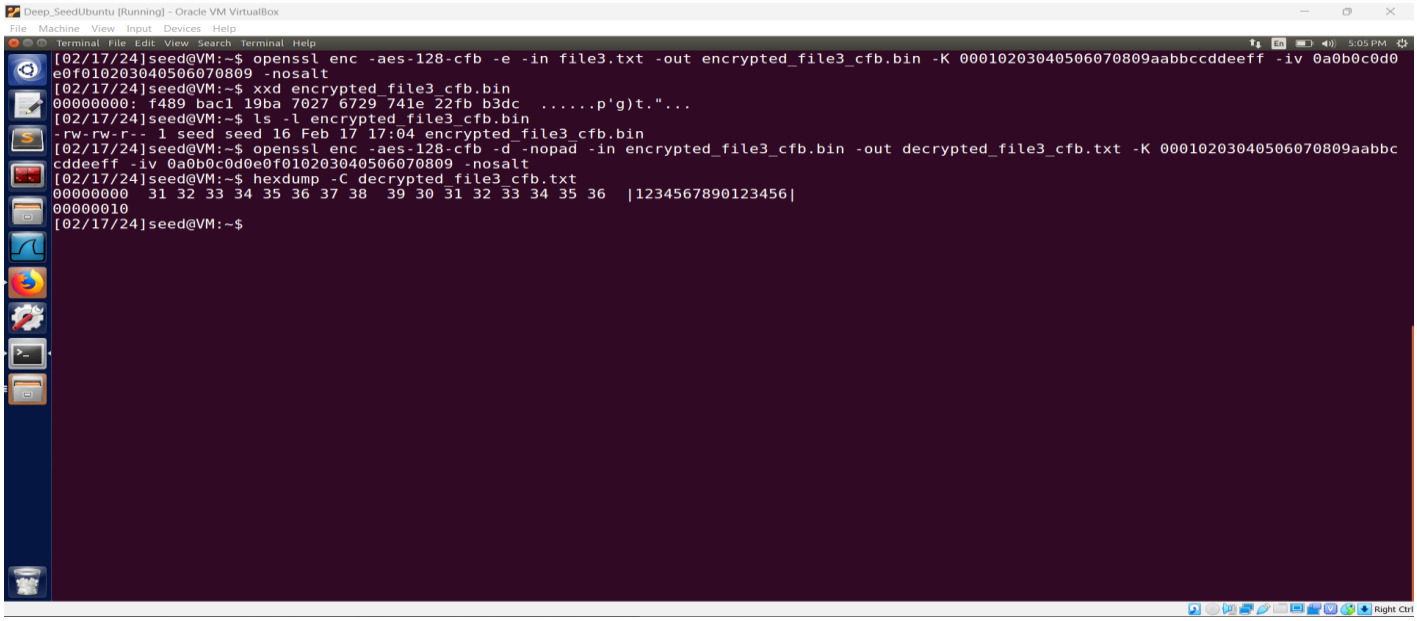
  - file1.txt size = 5 bytes
  - Size after encryption = 5 bytes
  - Padding added during encryption = 0 bytes

4.2 file2.txt (CFB Mode)



- **Observations:**
  - file2.txt size = 10 bytes
  - Size after encryption = 10 bytes
  - Padding added during encryption = 0 bytes

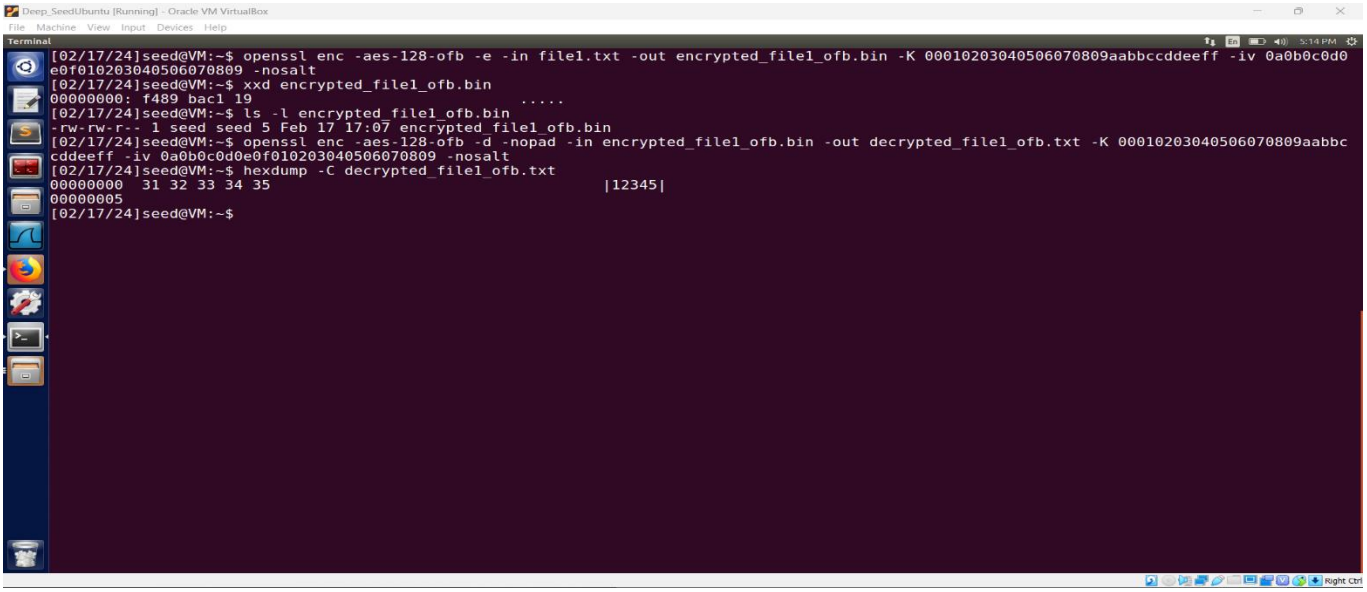
4.3 file3.txt (CFB Mode)



- **Observations:**
  - file3.txt size = 16 bytes
  - Size after encryption = 16 bytes
  - Padding added during encryption = 0 bytes

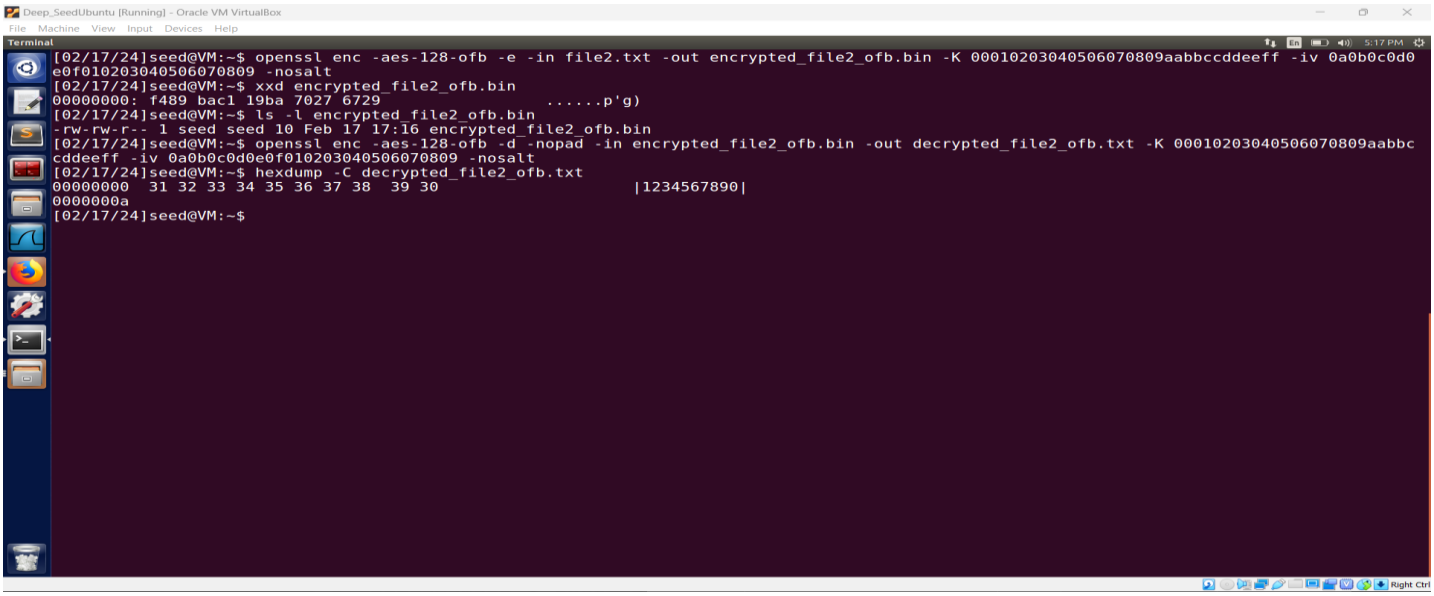
5. Encrypt file1.txt, file2.txt and file3.txt using **-aes-128-ofb** mode. Then, we'll describe the size of the encrypted files by using the **ls -l** command. Then, decrypt encrypted files using **-aes-128-ofb** with padding disabled using **-nopad** to disable automatic padding removal during decryption. And at last, examine the decrypted data to determine the padding added during encryption, using a hex tool such as **hexdump** as follows:

5.1 file1.txt (OFB Mode)



- **Observations:**
  - File1.txt size = 5 bytes
  - Size after encryption = 5 bytes
  - Padding added during encryption = 0 bytes

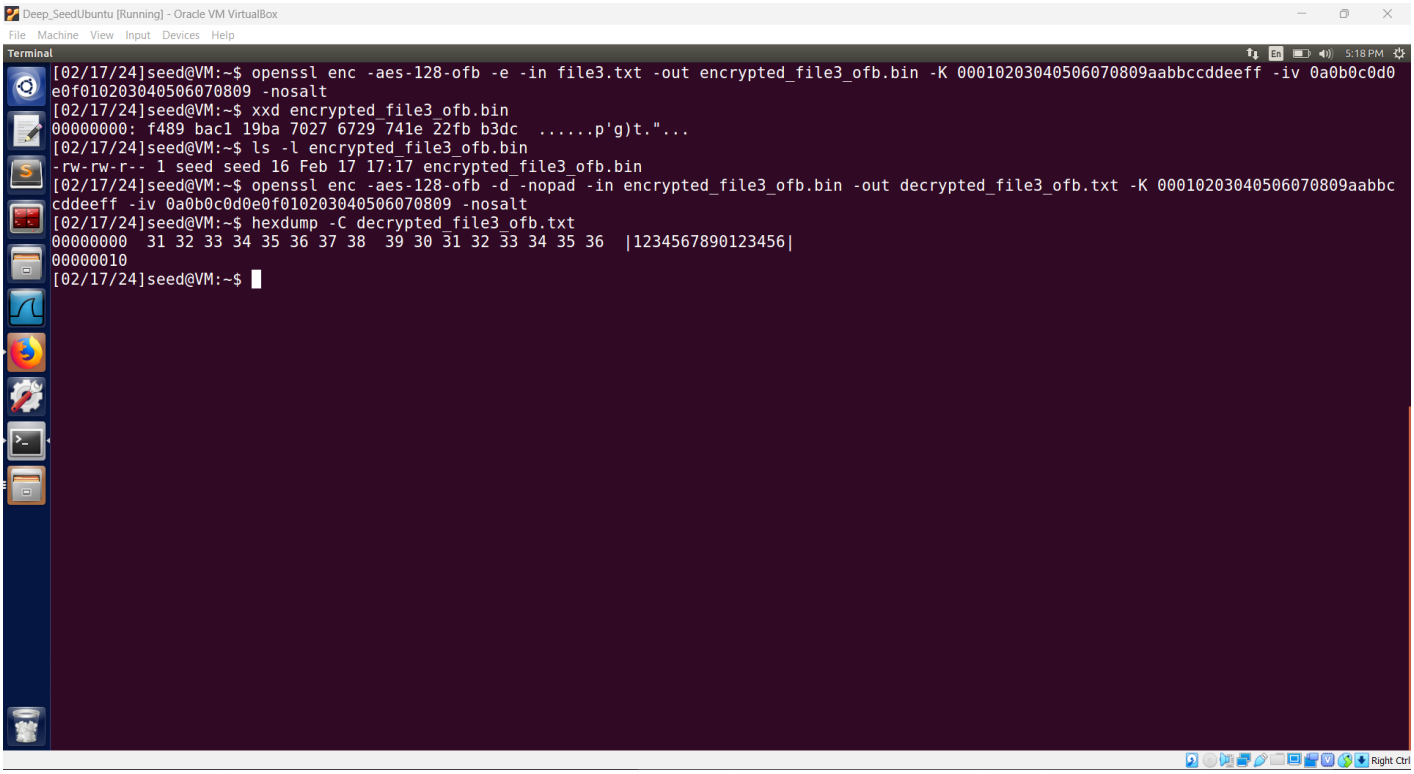
5.2 file2.txt (OFB Mode)



- **Observations:**

  - File2.txt size = 10 bytes
  - Size after encryption = 10 bytes
  - Padding added during encryption = 0 bytes

5.3 file3.txt (OFB Mode)



- **Observations:**

  - File3.txt size = 16 bytes
  - Size after encryption = 16 bytes
  - Padding added during encryption = 0 bytes

6. Final Verdict:

6.1 AES Mode: Padding added during encryption.

- The Electronic Codebook (ECB) mode of operation is a block cipher in which every plaintext block is individually encrypted with the same key. Every plaintext block is split up into fixed-size blocks, which are usually the same as the block size of the encryption algorithm in use (for example, 128 bits for AES). Padding is needed to guarantee that each plaintext block may be encrypted when the plaintext's size is not an exact multiple of the block size. Because ECB mode works with fixed-size blocks and cannot accommodate incomplete plaintext blocks, padding is necessary.

- PKCS#5 padding, sometimes referred to as PKCS#7 padding, is the most widely used padding method for block ciphers, including ECB mode. It involves padding the plaintext with bytes so that its length is multiples of the block size. The number of bytes added is indicated by the padding bytes themselves. For

instance, PKCS#5 padding will add extra bytes to the plaintext, each of which will have the value of the number of padding bytes added, if the plaintext size is not a multiple of the block size. As a result, the decryption process can decide how many padding bytes to discard once decoding is complete.

6.2 CBC Mode: Padding added during encryption.

- Before encrypting plaintext, CBC (Cipher Block Chaining) mode adds padding to make it match the preset block size required by the encryption algorithm—128 bits for AES, for example. When the length of the plaintext is not a perfect multiple of the block size, padding is required to ensure that each block may be encrypted separately.

- Before encryption in CBC mode, each plaintext block is XORed with the preceding ciphertext block; therefore, the encryption process requires a full block to begin. Padding ensures there is enough data for the XOR operation even if the plaintext doesn't fit a block completely. Additionally, padding guarantees that for every encryption operation, the initialization vector (IV) used in CBC mode stays unique. If padding is not used, it could be unclear what is in the last block, especially if the plaintext size is the same as the block size.

6.3 CFB Mode: Padding does not add during encryption.

- In the Cipher Feedback (CFB) mode, a block cipher is transformed into a stream cipher in which every block of plaintext is encrypted separately. Because of the encryption's independence, CFB mode doesn't require padding. The Initialization Vector (IV) for the first block or the previous ciphertext block is encrypted in CFB mode instead of the plaintext itself. This keystream is then XORed with the plaintext to create the ciphertext. Because of this procedure, plaintexts of any length can be handled by CFB mode without the need for padding to align the plaintext with a set block size.

- Additionally, CFB mode does not require padding to satisfy certain block size requirements because of its stream cipher behavior, which enables it to work on individual bits instead of fixed-size blocks. This stream cipher behavior guarantees that padding is not required during encryption, as does the feedback mechanism built into CFB mode.

6.4 OFB Mode: Padding does not add during encryption.

- When it comes to encryption, the OFB (Output Feedback) mode functions differently than block cipher modes like ECB or CBC. In contrast to these modes, OFB creates a pseudo-random bit keystream, which is XORed with the plaintext to create the ciphertext. The plaintext is not divided into fixed-size blocks, but rather is processed bit by bit. Consequently, since OFB mode does not operate with data blocks, padding is not required. Incomplete blocks are not a problem because every bit of the keystream is created independently based on the encryption key and initialization vector.

- Furthermore, padding is not required because OFB mode is self-synchronizing and independent of plaintext blocks. Since each bit in the keystream is created independently, block boundaries are not a concern for efficiently encrypting data streams of any length. Further eliminating the need for padding to maintain synchronization, this mode's self-synchronization characteristic guarantees that the sender and receiver can independently create the identical keystream as long as they utilize the same initialization vector and key.

4. Task 5: Error Propagation – Corrupted Cipher Text

To understand the error propagation property of various encryption modes, we would like to do the following exercise:

1. Create a text file that is at least 1000 bytes long.
2. Encrypt the file using the AES-128 cipher.
3. Unfortunately, a single bit of the 55th byte in the encrypted file got corrupted. You can achieve this corruption using the bless hex editor.
4. Decrypt the corrupted ciphertext file using the correct key and IV.

Please answer the following question: How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Please answer this question before you conduct this task, and then find out whether your answer is correct or wrong after you finish this task. Please provide justification.

Ans:

Before conducting the task, let's analyze how much information can we recover by decrypting the corrupted file using different encryption modes: In each encryption mode, the amount of information that can be recovered from the corrupted file depends on the mode's error propagation characteristics.

- **ECB (The Electronic Codebook):**

Every plaintext block is separately encrypted in ECB mode. As a result, just the associated block of plaintext is impacted if one bit in the ciphertext is corrupted. Therefore, all data may be recovered in ECB mode with the exception of the particular block containing the damaged byte. Every other block is still in place.

- **CBC (Cipher Block Chaining):**

Before encryption in CBC mode, every plaintext block is XORed with the preceding ciphertext block. Because of the XOR operation, a mistake in one ciphertext block will therefore damage the associated plaintext block and also the blocks that come after. As a result, you can only retrieve data up to the corrupted block in CBC mode. All blocks after the tainted block will be permanently changed.

- **CFB (Cipher Feedback):**

In CFB mode, the ciphertext is created by XORing the result of the encryption process with the plaintext. Because of the feedback process, a corrupted bit in the ciphertext will therefore impact the equivalent bit in the decoded plaintext and possibly a few more bits. You can therefore retrieve the data up to the corrupted bit in CFB mode. It is possible to change the corrupted bit as well as one or more succeeding bits.

- **OFB (Output Feedback):**

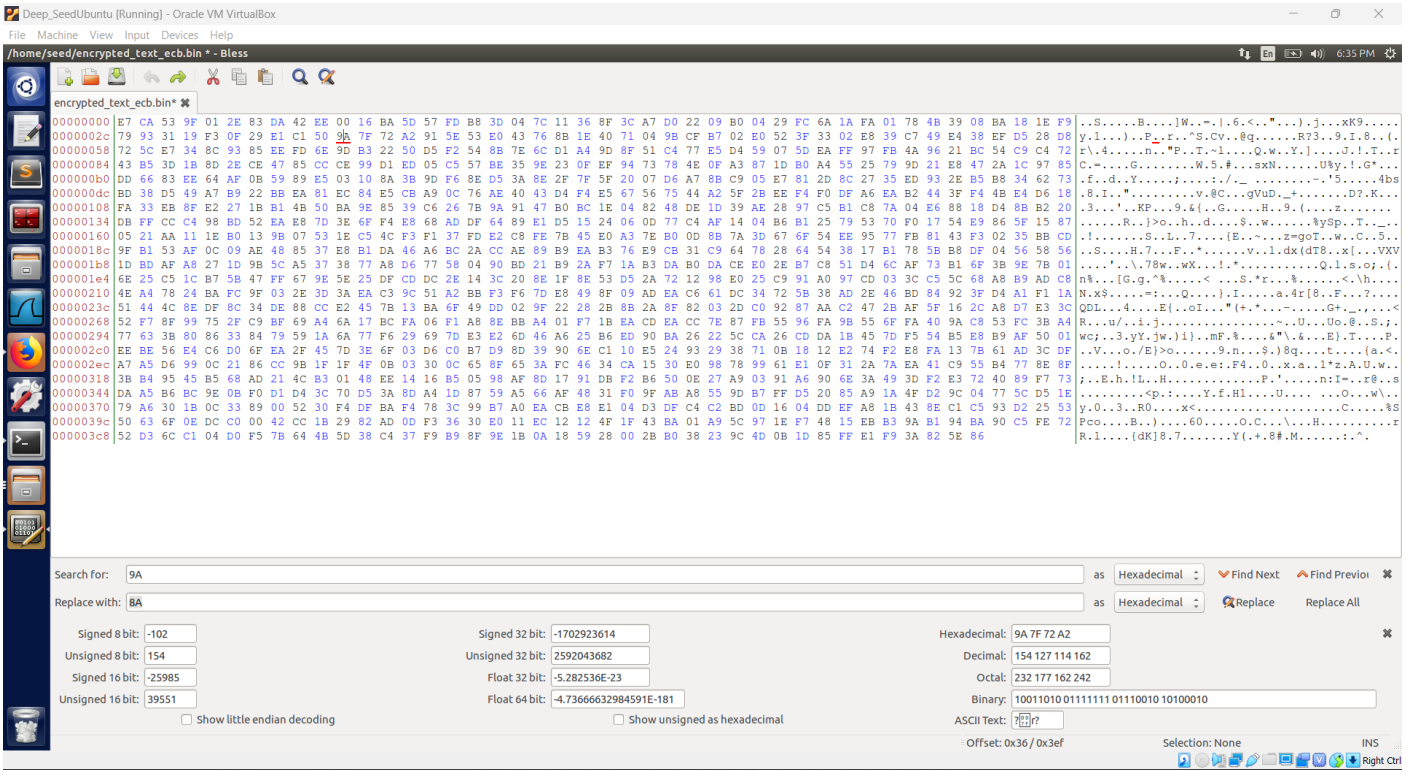
A keystream is created independently of the plaintext in OFB mode. The ciphertext is then created by XORing the plaintext and the keystream. Consequently, a mistake in the ciphertext has no direct impact on the plaintext that has been deciphered. As a result, all information can be recovered in OFB mode with the exception of the particular ciphertext bit that is corrupted. The matching bit of the decoded plaintext may be impacted by the corrupted bit.

- **Task process step by step:**

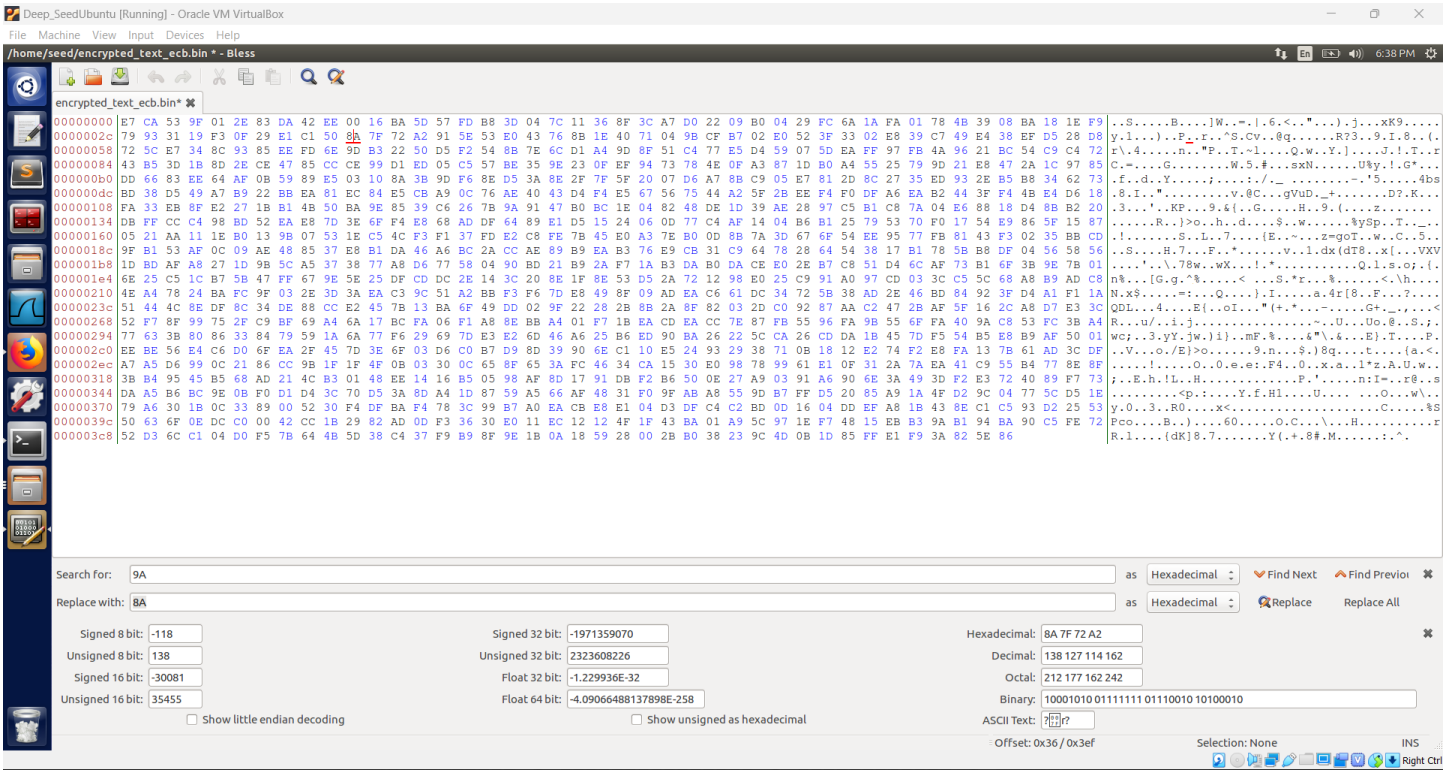
1. Generate a text file **plain_text.txt** that is at least 1000 bytes long. This file will serve as our plaintext input for encryption.

2. Use **AES-128** encryption with **ECB** mode to encrypt the text file and produce a ciphertext file.

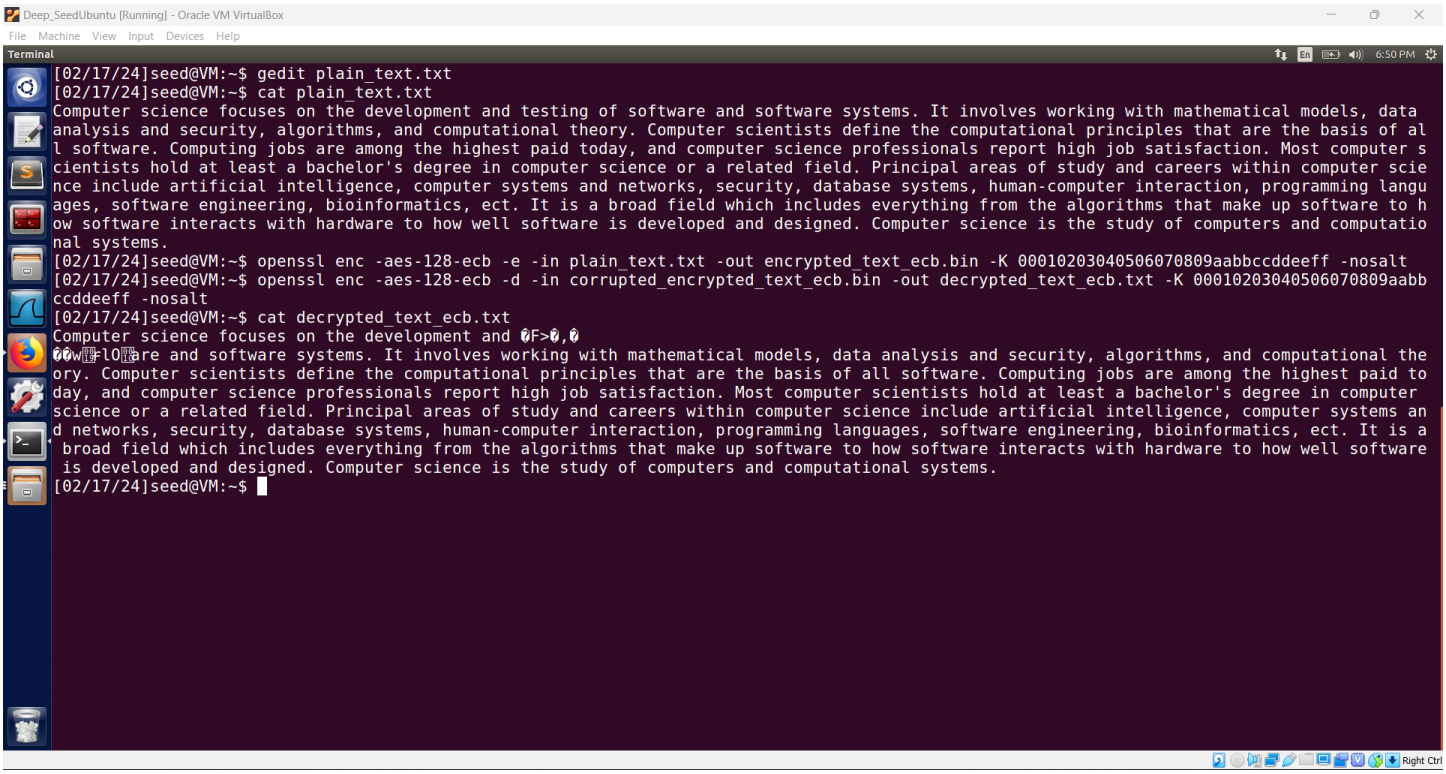
3. Using a hex editor tool **Bless text editor**, manually modify a single bit in the 55th byte of the encrypted ciphertext file to introduce corruption.



- In the encrypted file, the 55th byte is **9A**. I am corrupting the encrypted file by changing the 55th byte to **8A** as follows:



4. Decrypt the corrupted ciphertext file using the correct key and initialization vector (IV) used during encryption. Examine the decrypted plaintext file to determine how the corruption in the ciphertext has affected the decrypted content.

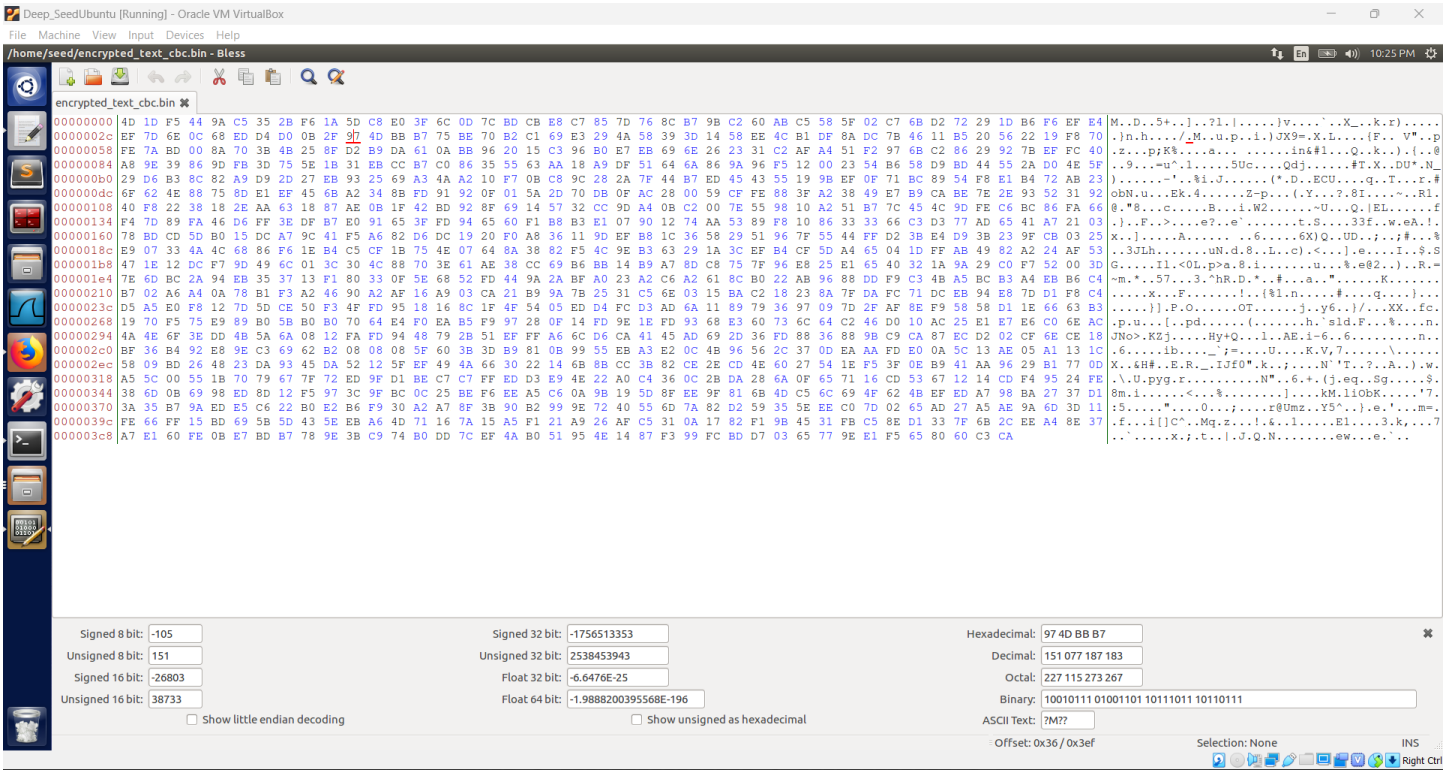


- **Observations:**

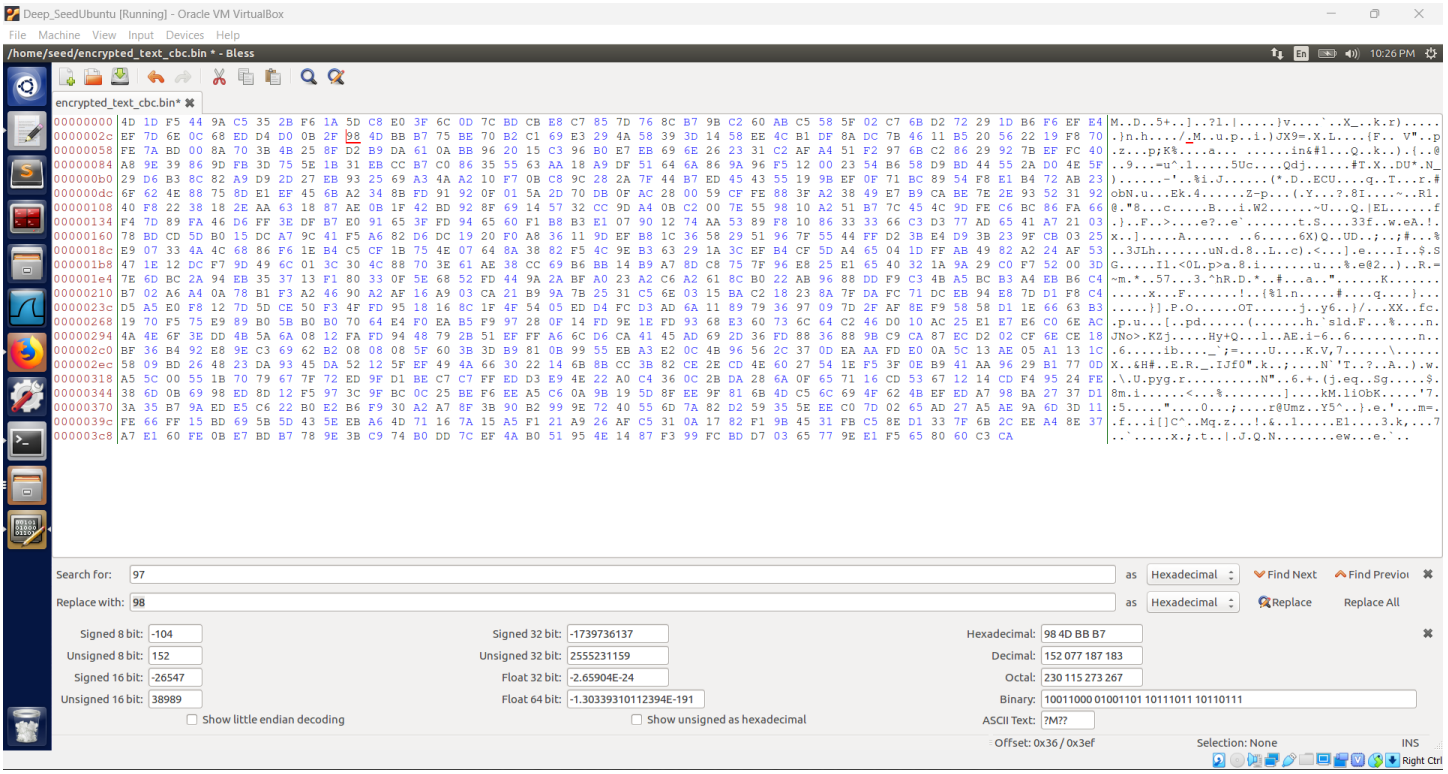
As discussed earlier, just the associated corrupted block of plaintext along with some data from both before and after the corrupted block is impacted as one bit in the ciphertext is corrupted. All the data after the corrupted part is recovered in ECB mode and every other block is still in place.

5. Use **AES-128** encryption with **CBC** mode to encrypt the text file and produce a ciphertext file.

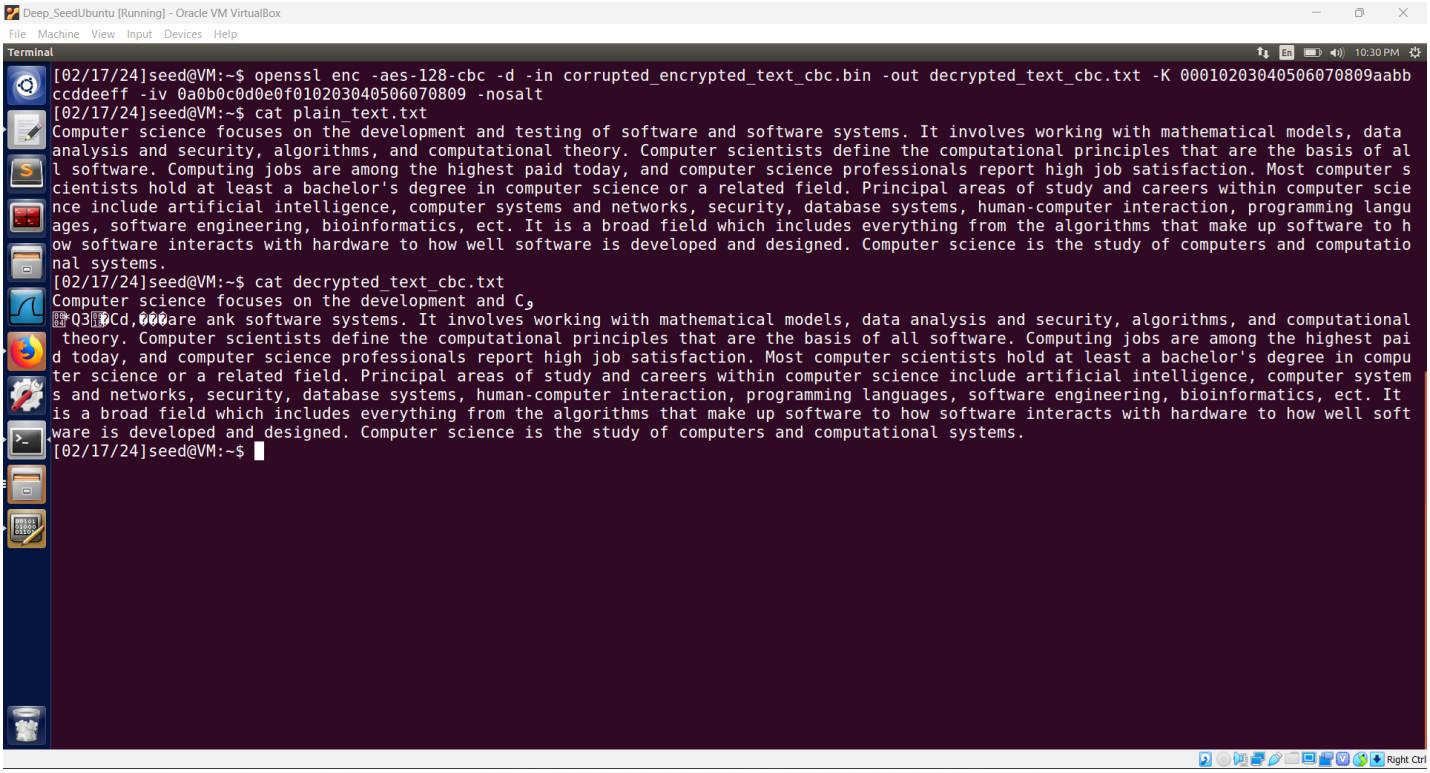
6. Using a hex editor tool **Bless text editor**, manually modify a single bit in the 55th byte of the encrypted ciphertext file to introduce corruption.



- In the encrypted file, the 55th byte is **97**. I am corrupting the encrypted file by changing the 55th byte to **98** as follows:



7. Decrypt the corrupted ciphertext file using the correct key and initialization vector (IV) used during encryption. Examine the decrypted plaintext file to determine how the corruption in the ciphertext has affected the decrypted content.



- **Observations:**

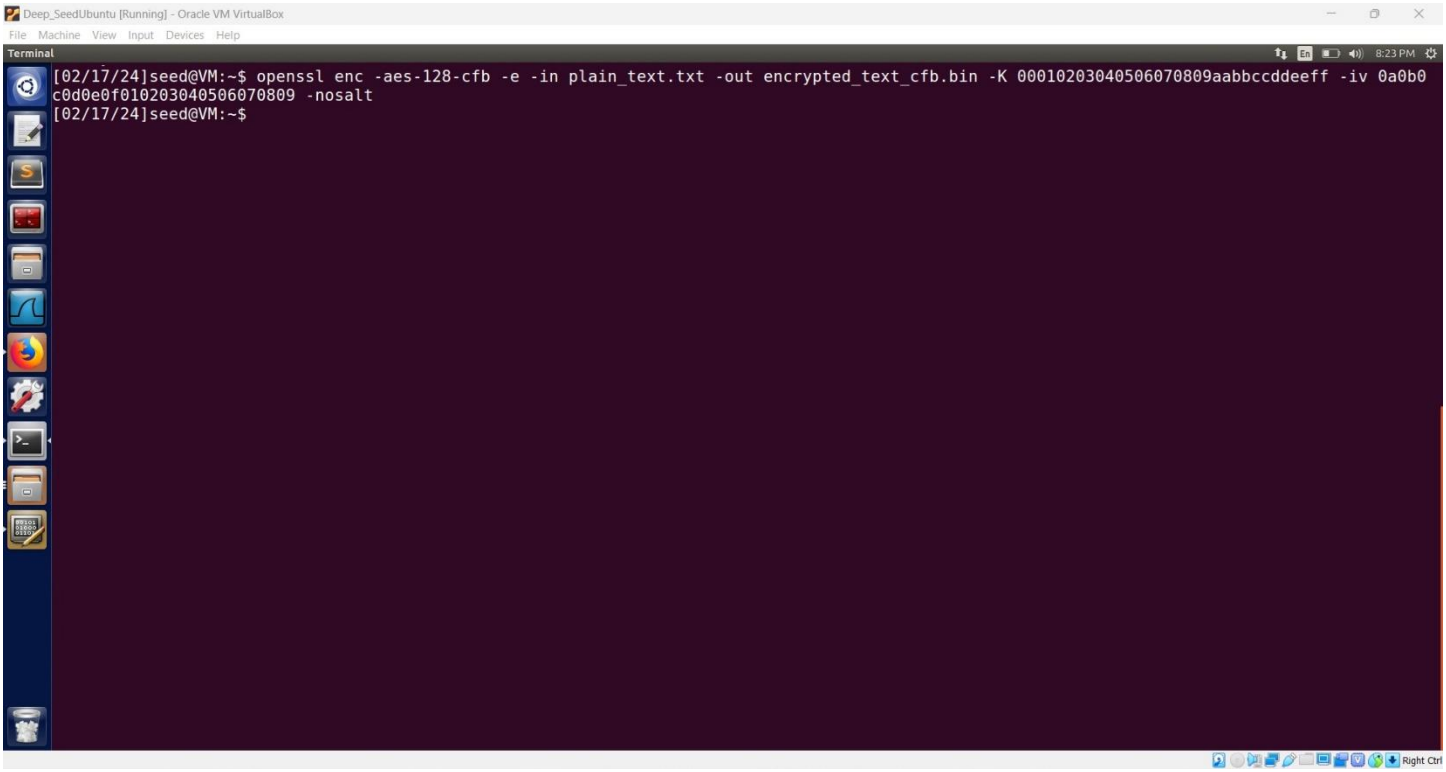
As discussed earlier, you can only retrieve data up to the corrupted block in CBC mode. All blocks after the corrupted block will be permanently changed. It is considered wrong because in this case, I can recover all the blocks after the corrupted block. The reason for this is as follows:

Because CBC mode uses an XOR technique to combine ciphertext blocks, a single-bit corruption in one of the ciphertext blocks corrupts the corresponding plaintext block and influences the decryption of the following blocks. This implies that every future plaintext block will be changed, as well as the corrupted block itself.

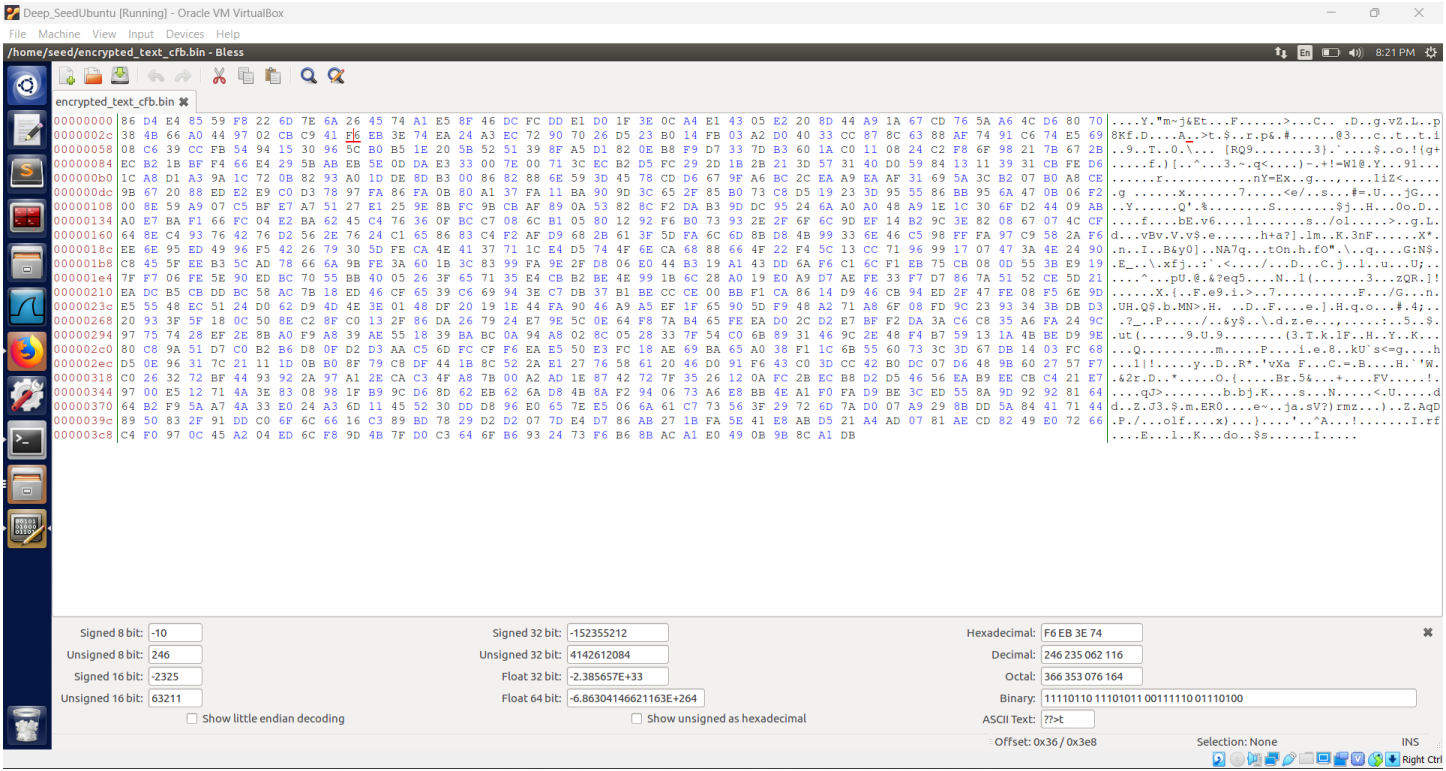
But there are situations in which the effect on later blocks may not be as bad as if they were irreversibly altered, particularly if the corruption just affects one bit. This is because, although the XOR operation spreads the fault across later blocks, its contents could not be entirely obliterated.

For example, the corresponding bit in the decrypted plaintext block will be impacted if a single bit in a ciphertext block is compromised. The XOR operation will then have an impact on the following block due to this modified bit. The remaining data in succeeding blocks may still be recoverable, although slightly corrupted because just one bit is changed.

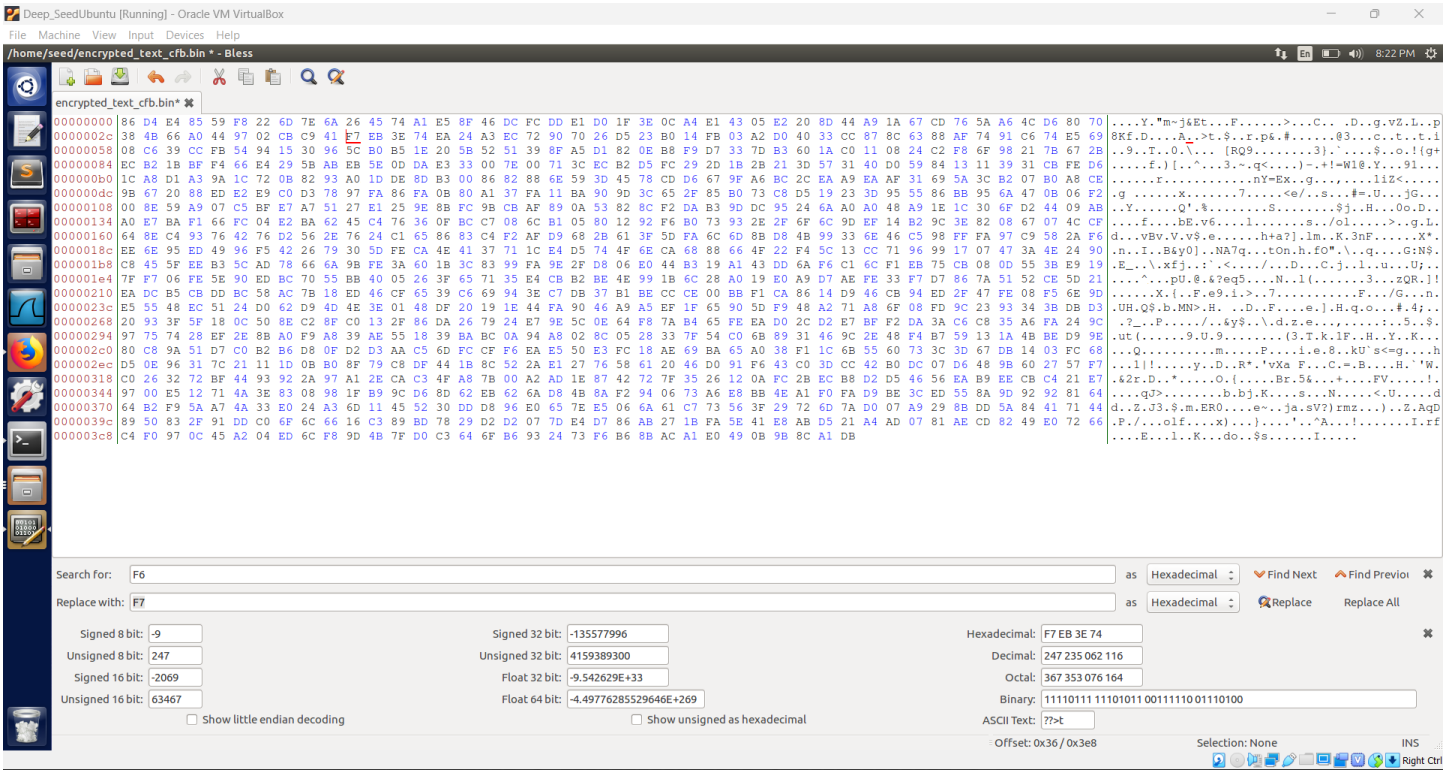
8. Use **AES-128** encryption with **CFB** mode to encrypt the text file and produce a ciphertext file.



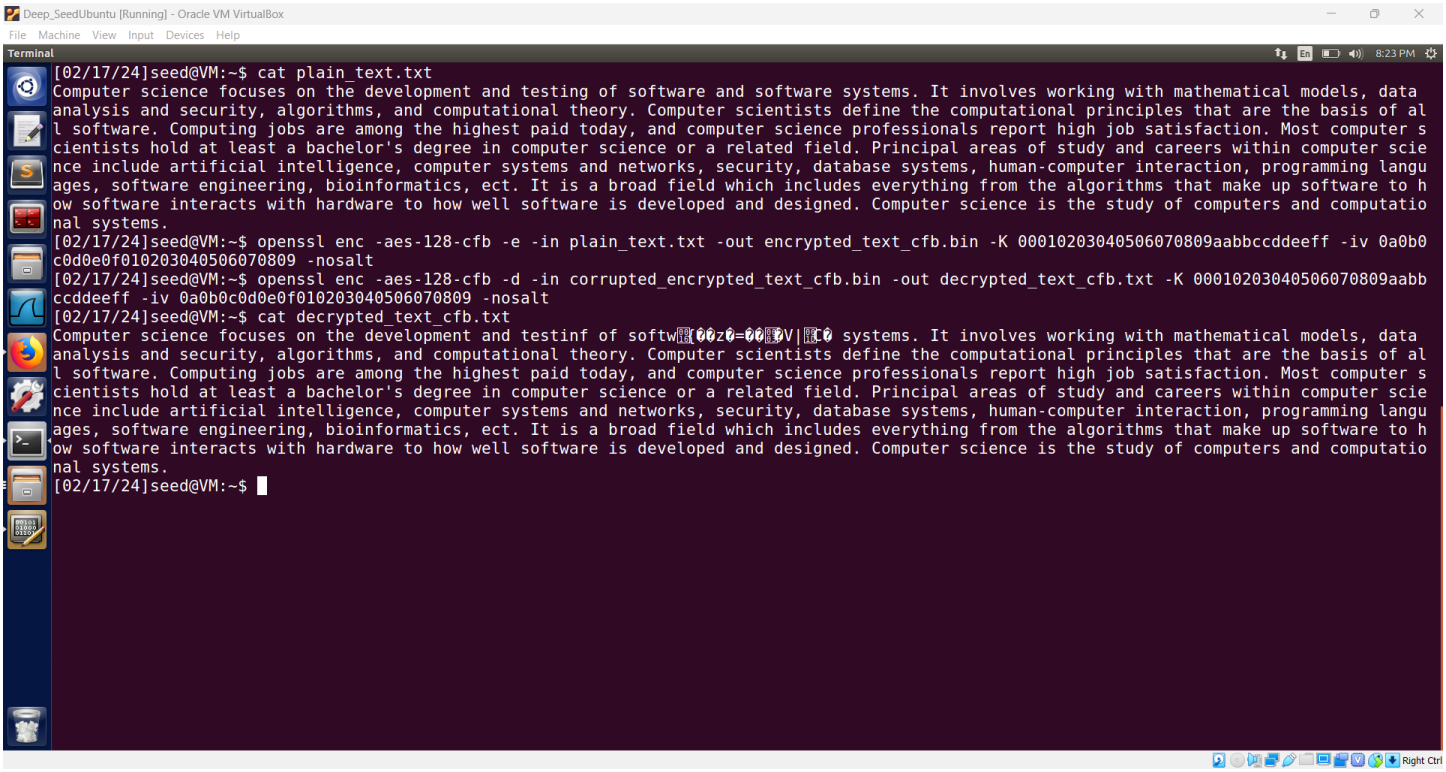
9. Using a hex editor tool **Bless text editor**, manually modify a single bit in the 55th byte of the encrypted ciphertext file to introduce corruption.



- In the encrypted file, the 55th byte is **F6**. I am corrupting the encrypted file by changing the 55th byte to **F7** as follows:



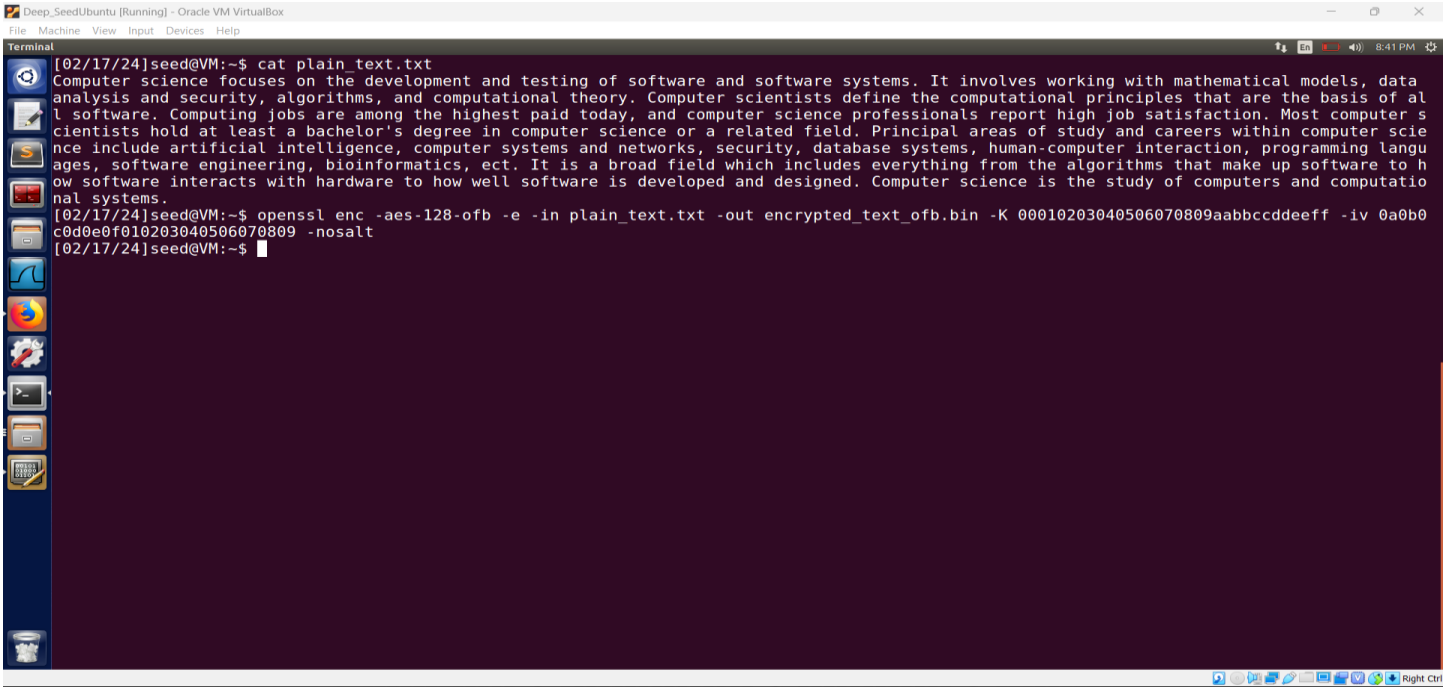
10. Decrypt the corrupted ciphertext file using the correct key and initialization vector (IV) used during encryption. Examine the decrypted plaintext file to determine how the corruption in the ciphertext has affected the decrypted content.



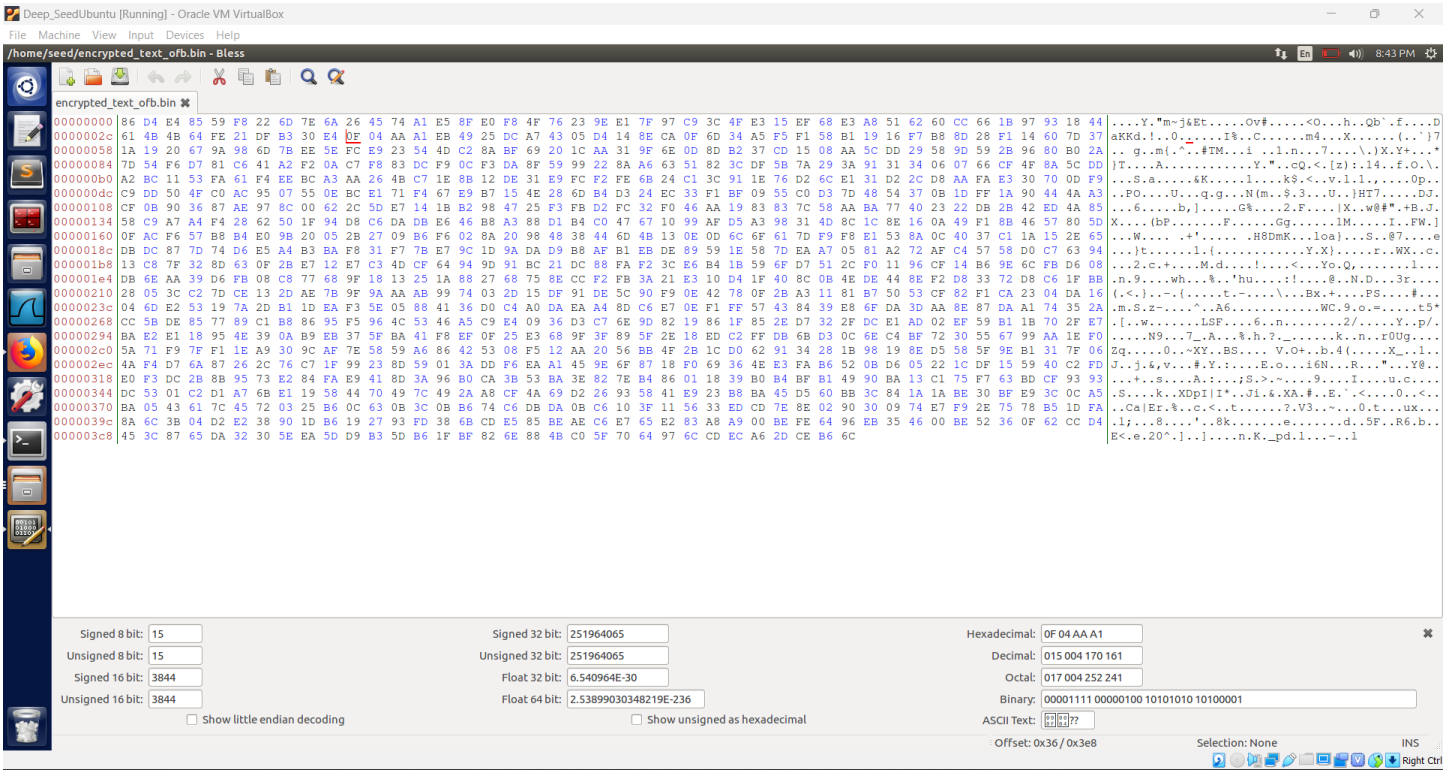
- **Observations:**

As discussed earlier, it affects the corresponding bit in the decrypted plaintext and potentially a few subsequent bits due to the feedback mechanism as we corrupt one bit in the ciphertext. All the data after the corrupted part is recovered in CFB mode and every other block is still in place.

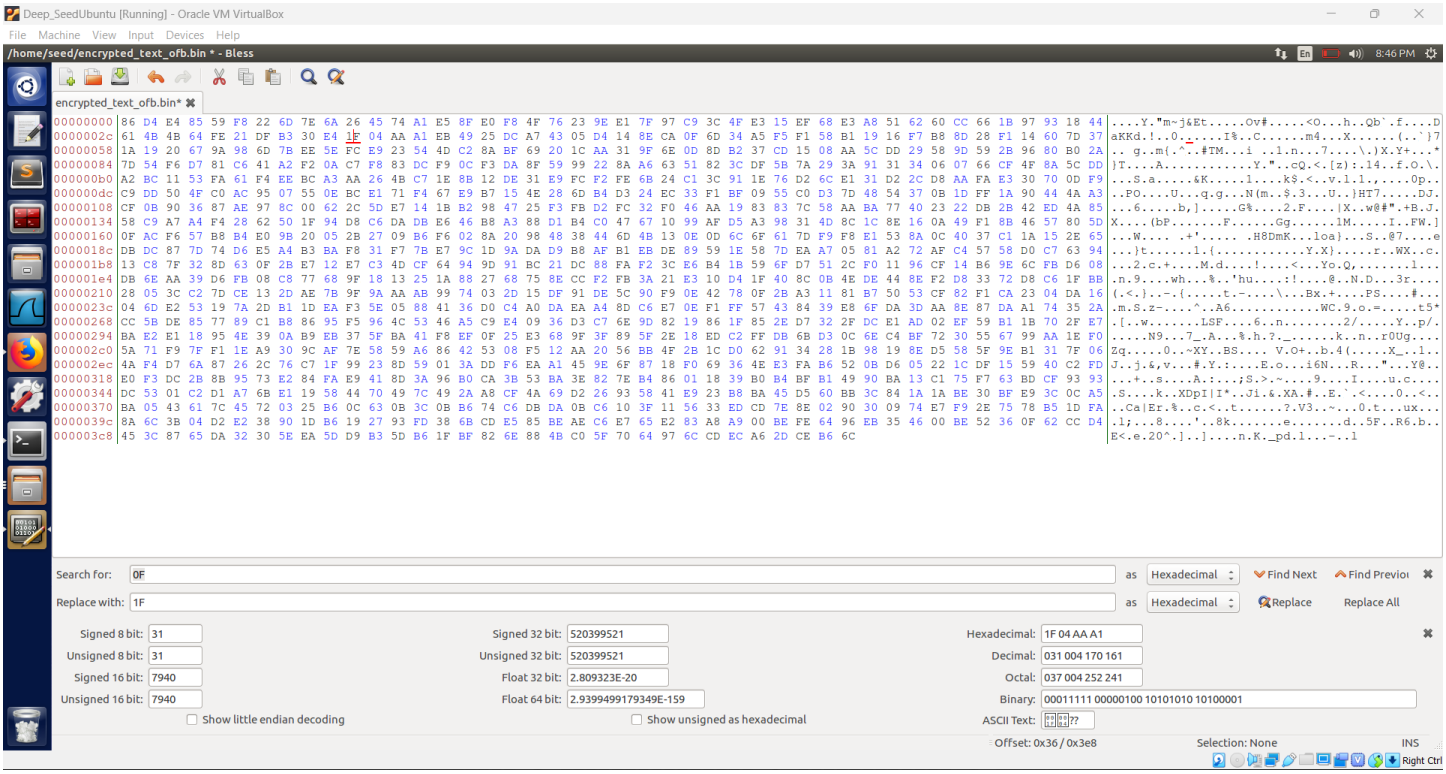
11. Use **AES-128** encryption with **OFB** mode to encrypt the text file and produce a ciphertext file.



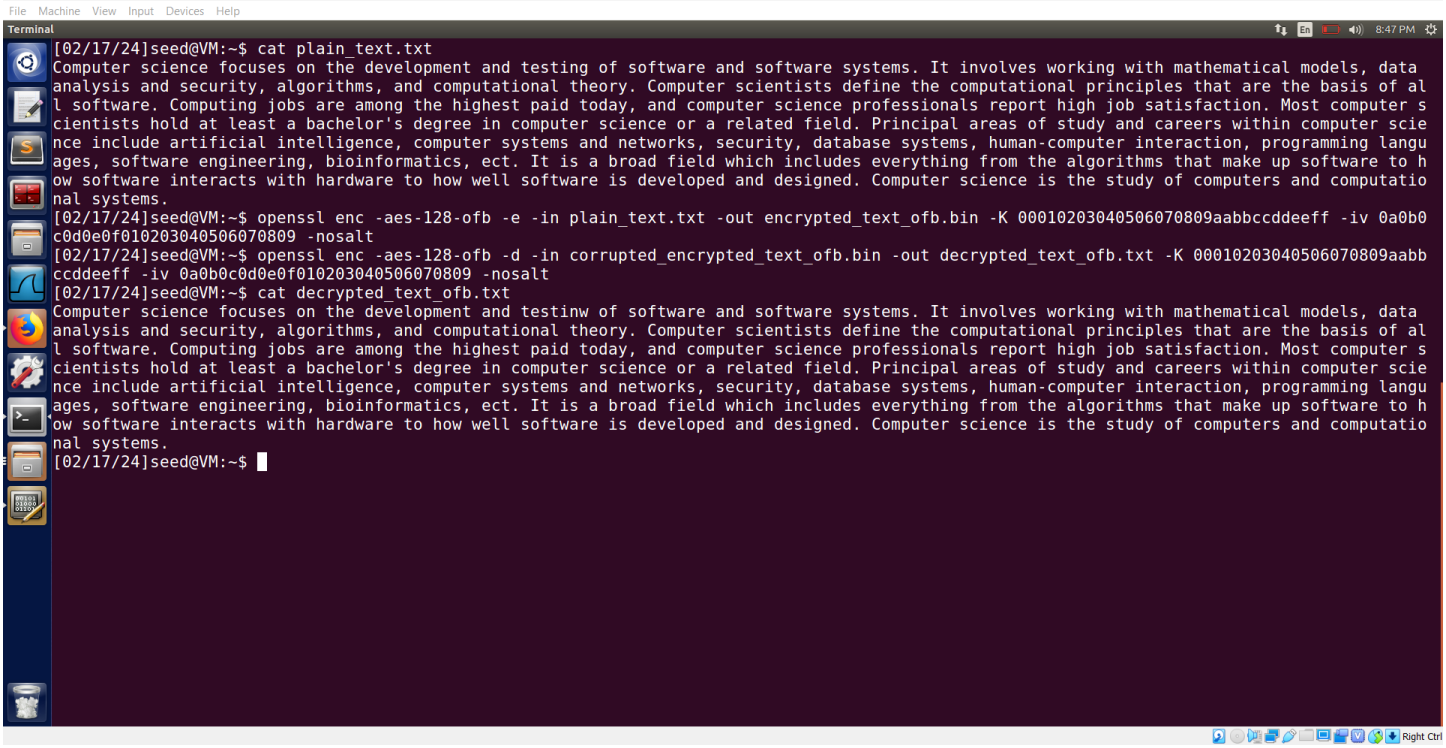
12. Using a hex editor tool **Bless text editor**, manually modify a single bit in the 55th byte of the encrypted ciphertext file to introduce corruption.



- In the encrypted file, the 55th byte is **0F**. I am corrupting the encrypted file by changing the 55th byte to **1F** as follows:



13. Decrypt the corrupted ciphertext file using the correct key and initialization vector (IV) used during encryption. Examine the decrypted plaintext file to determine how the corruption in the ciphertext has affected the decrypted content.



- **Observations:**

As discussed earlier, in CFB mode, you can recover all information except for just the specific bit that is corrupted in the ciphertext. All the data after and before the corrupted part is successfully recovered in CFB mode and every other block is still in place.

5. Task 6: Initial Vector (IV)

Most of the encryption modes require an initial vector (IV). Properties of an IV depend on the cryptographic scheme used. If we are not careful in selecting IVs, the data encrypted by us may not be secure at all, even though we are using a secure encryption algorithm and mode. The objective of this task is to help students understand the problems if an IV is not selected properly. Please do the following experiments:

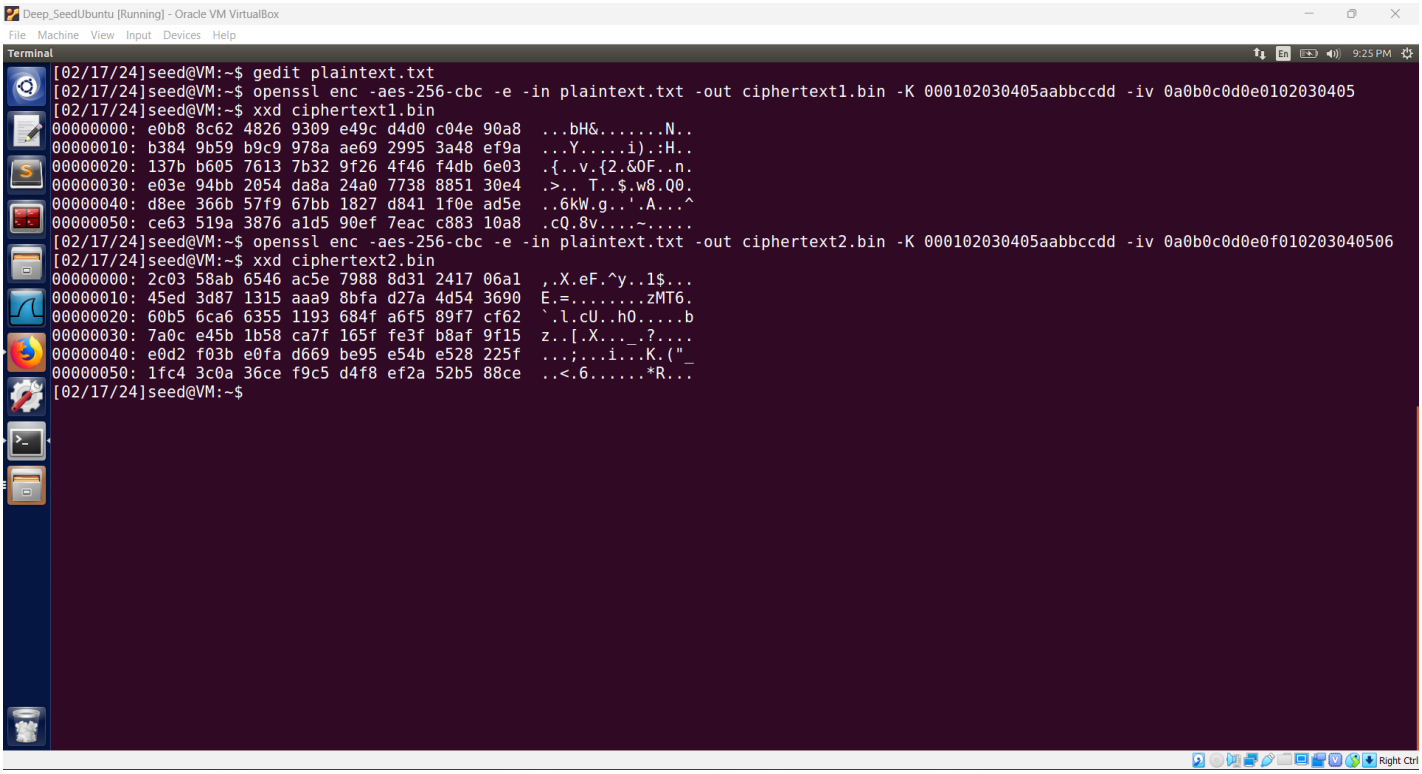
2.6.1 Task 6.1. Uniqueness of the IV

A basic requirement for IV is *uniqueness*, which means that no IV may be reused under the same key. To understand why, please encrypt the same plaintext using (1) two different IVs, and (2) the same IV. Please describe your observation, based on which, explain why IV needs to be unique.

Ans:

- **Steps to be followed:**

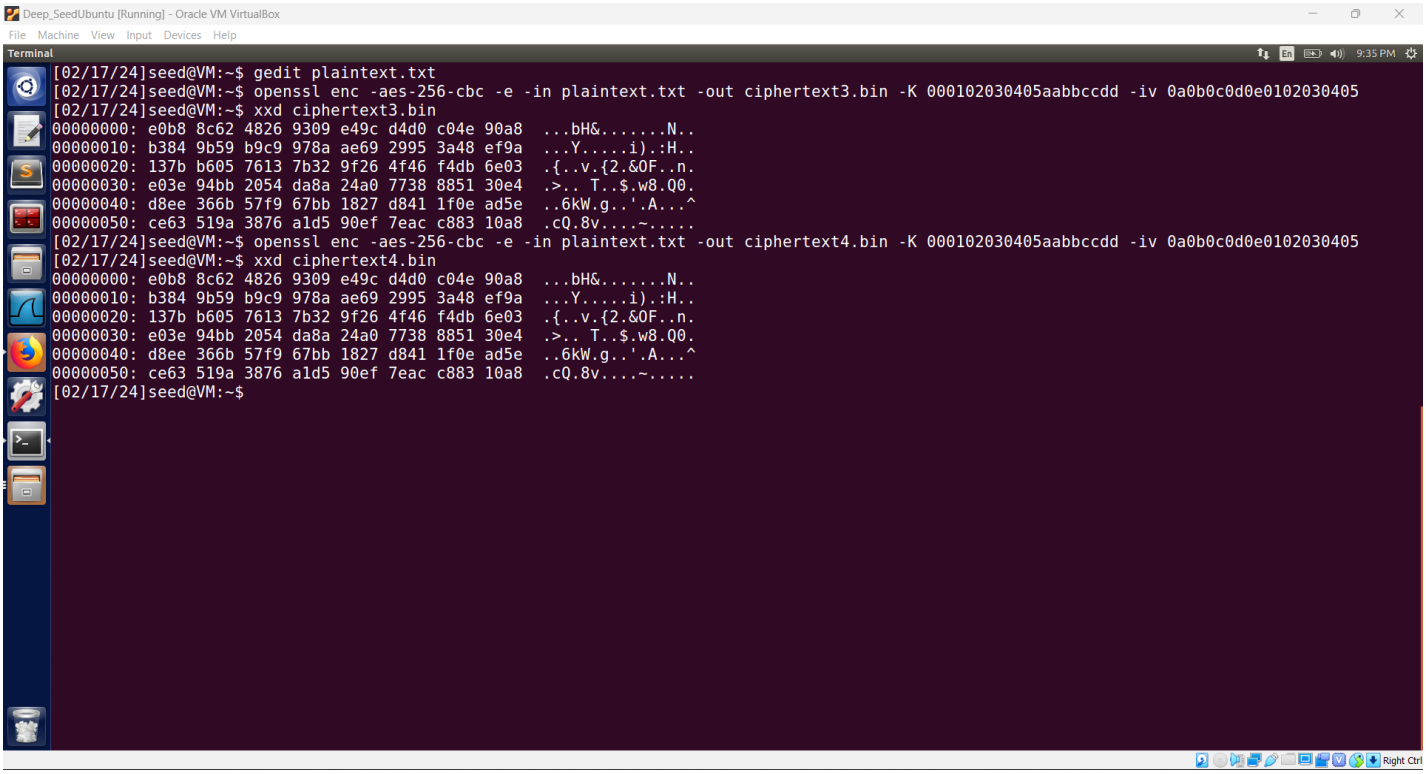
1. Encryption with Different IVs:



- **Observations:**

The ciphertexts that are produced when the identical plaintext is encrypted with various IVs will differ as well as we can see in the above output. This is because the IV influences how the encryption process proceeds since it serves as the encryption algorithm's first input. A slight alteration in the IV will result in a whole distinct ciphertext, guaranteeing the uniqueness of the encrypted data.

2. Encryption with the Same IV:



- **Observations:**

  The ciphertexts that are produced when the same IV is used to encrypt the same plaintext more than once will be the same. This could be a security problem since an attacker could be able to decipher the underlying plaintext by examining the patterns in the ciphertexts. Reusing the same IV makes the encryption system less secure since it makes it easier for attackers to find patterns in plaintext and maybe take advantage of weaknesses.

3. Why IV needs to be unique:

The security of encryption methods relies heavily on the Initialization Vector's (IV) uniqueness. Employing a different IV for every encryption operation is essential when encrypting data, especially when employing block cipher modes like CBC (Cipher Block Chaining), to avoid patterns forming in the ciphertext. When an IV is utilized for several encryptions with the same key, patterns in the ciphertext can appear that could provide details about the plaintexts. Attackers can use this weakness to match ciphertexts with known plaintexts in dictionary attacks or to infer relationships between plaintexts. Moreover, distinct IVs are necessary to preserve semantic security, guaranteeing that the ciphertext only discloses its length and not any other information about the plaintext. Encryption techniques can effectively resist chosen-plaintext attacks and protect the secrecy and integrity of encrypted data by enforcing the uniqueness of the IV, thus supporting overall security. Here's why IV needs to be unique:

- **Preventing Pattern Formation:** The ciphertext may start to show patterns if the same IV is used to encrypt several plaintexts with the same key. Even in the absence of the encryption key, attackers can use these patterns to deduce details about the plaintexts. For instance, if the same plaintext is encrypted twice using the same IV, the ciphertexts that are produced would be the same, which would indicate patterns or similarities in the plaintext.
- **Preserving Similarity:** Even when the same plaintext is encrypted more than once, the IV's uniqueness guarantees that every encryption process generates a distinct ciphertext. Semantic security requires that the ciphertext not reveal any information about the plaintext other than its length, and this characteristic is essential to achieving this goal.
- **Defense Against Chosen-Plaintext Attacks:** By reusing the IV, an adversary may be able to identify repeated plaintexts, deduce relationships between plaintexts, or conduct traffic analysis to obtain information about the system if they are able to watch how chosen plaintexts are encrypted and examine the corresponding ciphertexts.

2.6.2 Task 6.2. Common Mistake: Use the Same IV

One may argue that if the plaintext does not repeat, using the same IV is safe. Let us look at the Output Feedback (OFB) mode. Assume that the attacker gets hold of a plaintext (P1) and a ciphertext (C1), can he/she decrypt other encrypted messages if the IV is always the same? You are given the following information, please try to figure out the actual content of P2 based on C2, P1, and C1.

```
Plaintext (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c15913

Plaintext (P2): (unknown to you)
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc15913
```

If we replace OFB in this experiment with CFB (Cipher Feedback), how much of P2 can be revealed? You only need to answer the question; there is no need to demonstrate that.
The attack used in this experiment is called the known-plaintext attack, which is an attack model for cryptanalysis where the attacker has access to both the plaintext and its encrypted version (ciphertext). If this can lead to the revealing of further secret information, the encryption scheme is not considered as secure.

Ans:

Here,

Plaintext (P1): This is a known message!
(hex representation: 54686973206973206120b6e6f776e206d65737361676521)

Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c15913

Plaintext (P2): (unknown)

Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc15913

- **Code:**

```
P1 = "54686973206973206120b6e6f776e206d65737361676521"  # Hex representation of "This is a known message!"
C1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c15913"
C2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc15913"

# Convert hex strings to bytes
P1_bytes = bytearray.fromhex(P1_hex)
C1_bytes = bytearray.fromhex(C1_hex)
C2_bytes = bytearray.fromhex(C2_hex)

# XOR the known plaintext (P1) with C1 to obtain the keystream
keystream = bytearray([x ^ y for x, y in zip(P1_bytes, C1_bytes)])

# XOR the keystream with C2 to obtain P2
P2_bytes = bytearray([x ^ y for x, y in zip(keystream, C2_bytes)])

# Convert bytes to string
P2 = P2_bytes.decode("utf-8")

print("Recovered plaintext (P2) : ", P2)
```

- **Code Explanation:**

Let's break down the code:

- **Initialization:**

  P1: Represents the hexadecimal representation of the known plaintext P1, "This is a known message!", in ASCII format.
  C1: Represents the hexadecimal representation of the ciphertext corresponding to P1, denoted as C1.
  C2: Represents the hexadecimal representation of the ciphertext corresponding to the unknown plaintext P2, denoted as C2.

- **Conversion to Bytes:**

  P1_bytes, C1_bytes, C2_bytes: Convert the hexadecimal strings (representing P1, C1, and C2) into byte arrays for processing.

- **Keystream Generation:**

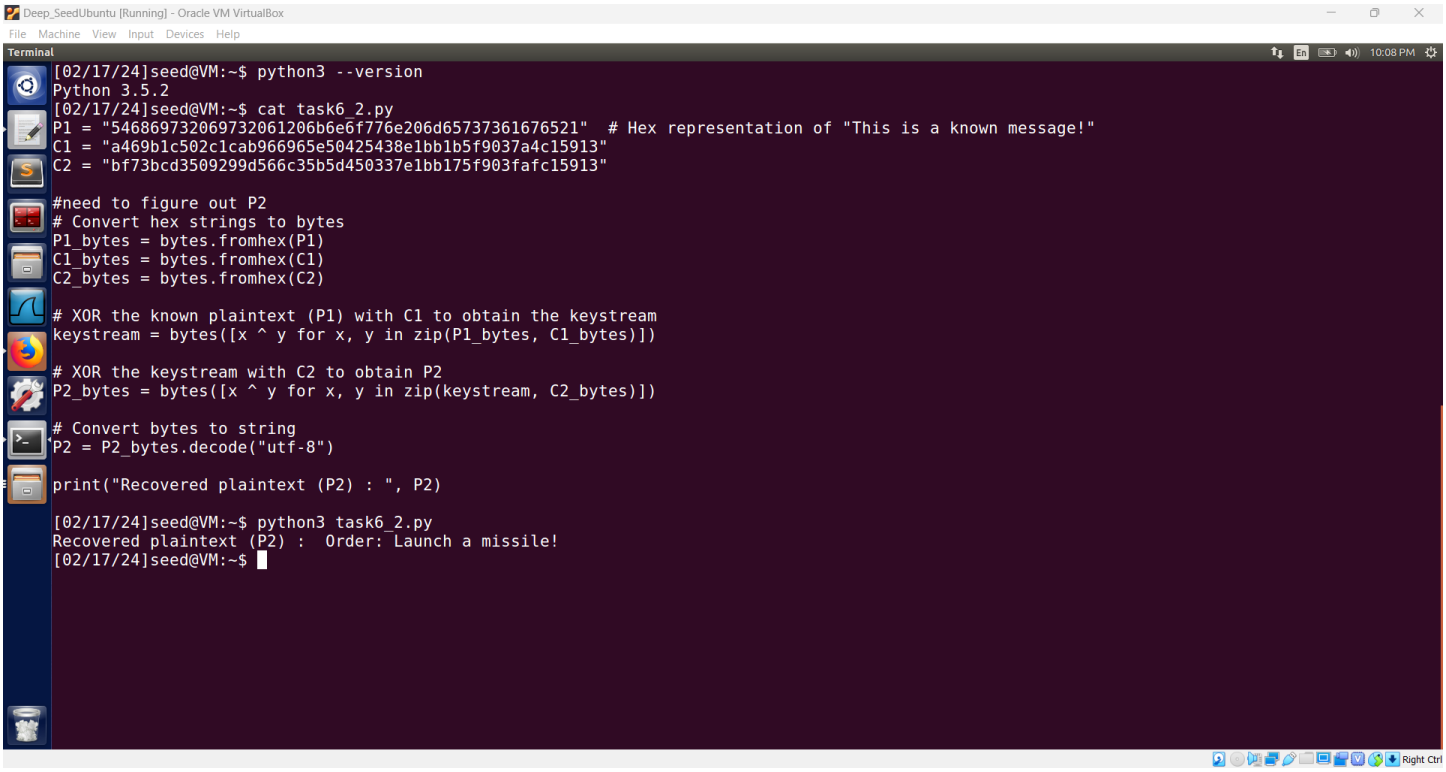
  keystream: XOR operation between the known plaintext (P1) and the ciphertext (C1). This results in the generation of the keystream, which is used to encrypt the plaintext.

- **Plaintext Recovery:**

  P2_bytes: XOR operation between the obtained keystream and the ciphertext C2. This operation effectively decrypts C2 using the keystream, revealing the unknown plaintext P2.
  P2: Decodes the obtained bytes into a UTF-8 string, representing the recovered plaintext P2.

- **Output:** The code prints the recovered plaintext P2 to the console.

- **Output: Order: Launch a missile!**



- **If we replace OFB in this experiment with CFB (Cipher Feedback), how much of P2 can be revealed?**

Even when an attacker uses the same IV in OFB mode to encrypt both plaintexts P1 and P2, they will not be able to decrypt additional encrypted communications once they have obtained the ciphertext corresponding to P1 (C1). This is because OFB mode encrypts plaintext without regard to earlier ciphertext blocks, therefore the IV has no effect on the encryption process after the initial setup.

In CFB (Cipher Feedback) mode, an attacker may be able to infer details about P2 based on their understanding of P1 and the associated ciphertexts C1 and C2, provided that the same IV is used for P1 and P2. The length of the plaintexts, the block size, and the encryption algorithm being used are some of the variables that affect how much of P2 can be exposed. Generally, P2 may disclose some information, but full decryption may not be achievable without further data or cryptanalysis.

As a result, if the same IV is used in OFB mode, a known-plaintext attack may be able to uncover further secret information, possibly even portions of P2. In contrast to OFB mode, the amount of information disclosed about P2 would be substantially less in CFB mode.

2.6.3 Task 6.3. Common Mistake: Use a Predictable IV

From the previous tasks, we now know that IVs cannot repeat. Another important requirement on IV is that IVs need to be unpredictable for many schemes, i.e., IVs need to be randomly generated. In this task, we will see what is going to happen if IVs are predictable.
Assume that Bob just sent out an encrypted message, and Eve knows that its content is either `Yes` or `No`; Eve can see the ciphertext and the IV used to encrypt the message, but since the encryption algorithm AES is quite strong, Eve has no idea what the actual content is. However, since Bob uses predictable IVs, Eve knows exactly what IV Bob is going to use next. The following summarizes what Bob and Eve know:

```
Encryption method: 128-bit AES with CBC mode.

Key (in hex): 00112233445566778899aabbccddeeff (known only to Bob)

Ciphertext (C1): bef65565572ccee2a9f9553154ed9498 (known to both)
IV used on P1 (known to both)
        (in ascii): 1234567890123456
        (in hex)  : 31323334353637383930313233343536
Next IV (known to both)
        (in ascii): 1234567890123457
```

A good cipher should not only tolerate the known-plaintext attack described previously, it should also tolerate the chosen-plaintext attack, which is an attack model for cryptanalysis where the attacker can obtain the ciphertext for an arbitrary plaintext. Since AES is a strong cipher that can tolerate the chosen-plaintext attack, Bob does not mind encrypting any plaintext given by Eve; he does use a different IV for each plaintext, but unfortunately, the IVs he generates are not random, and they can always be predictable.

Your job is to construct a message `P2` and ask Bob to encrypt it and give you the ciphertext. Your objective is to use this opportunity to figure out whether the actual content of `P1` is `Yes` or `No`.

Ans:

Eve can create a chosen-plaintext attack to take advantage of the predicted IVs and ascertain if P1's true content is "Yes" or "No". When utilizing predictable IVs, Eve can deduce information about P1 based on the change in ciphertext induced by a change in plaintext by creating a specific plaintext message (P2) and watching the related ciphertext (C2).

- **Steps to be followed by Eve:**

1. Creating the Message in Plaintext (P2):

Since Eve is aware of Bob's predicted IV, which he used for the subsequent message, she can build P2 so that it discloses details about P1. Eve can select "Yes" or "No" for P2, switching back and forth between the two choices to see how the ciphertext is affected by the change in plaintext.

2. Calling for Bob to Encrypt P2:

   Eve asks Bob to use the predicted IV to encrypt the carefully constructed plaintext message P2.

3. Observing the Ciphertext (C2):

   Eve checks the ciphertext C2, which corresponds to P2, with the ciphertext C1, which is known.
   If C1 and C2 differ significantly, it suggests that P1's actual content differs from P2's.

- **Steps to follow:**

Given data,

We need to use 128-bit AES with CBC mode encryption method.

Key (in hex): 00112233445566778899aabbccddeeff (known only to Bob)
Ciphertext (C1): bef65565572ccee2a9f9553154ed9498 (known to both) IV used on P1 (known to both)

(in ascii): 1234567890123456
(in hex): 31323334353637383930313233343536

Next IV (known to both)
(in ascii): 1234567890123457
(in hex): 31323334353637383930313233343537

We need to find out P2.
If C1 = C2, then it is safe to say that P1 can be ‘Yes or ‘No’. To find P2, we need to satisfy one condition, which is below:

$$IV1 \oplus P1 = IV2 \oplus P2$$

Also, the above can be written as,

$$P2 = IV1 \oplus IV2 \oplus P1$$

Now substituting the known values in the above equation,

P2 = 31323334353637383930313233343536 XOR 31323334353637383930313233343537 XOR 596573
P2 = Yes.

Now, we will find out the encrypted message for P2 using IV2.

C2 = 3fdff24f43d61f27e9539b963a3bf847 ((in hex): 31323334353637383930313233343537)

Now we know that,

C1 = bef65565572ccee2a9f9553154ed9498 ((in hex): 31323334353637383930313233343536) which is not equal to C2.
C1 ≠ C2, hence the plaintext of the P1 = No.

6. Additional Readings

There are more advanced cryptanalysis on IV that is beyond the scope of this lab. Students can read the article posted in this URL: https://defuse.ca/cbcmodeiv.htm. Because the requirements on IV really depend on cryptographic schemes, it is hard to remember what properties should be maintained when we select an IV. However, we will be safe if we always use a new IV for each encryption, and the new IV needs to be generated using a good pseudo random number generator, so it is unpredictable by adversaries. Students can read this Wikipedia page for ideas: Initialization vector (https://en.wikipedia.org/wiki/Initialization vector).

7. Task 7: Programming using the Crypto Library - Extra Credit 5%

So far, we have learned how to use the tools provided by `openssl` to encrypt and decrypt messages. In this task, we will learn how to use `openssl`'s crypto library to encrypt/decrypt messages in programs.

In this task, you are given a plaintext and a ciphertext, and your job is to find the key that is used for the encryption. You do know the following facts:

- The `aes-128-cbc` cipher is used for the encryption.
- The key used to encrypt this plaintext is an English word shorter than 16 characters; the word can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), pound signs (#: hexadecimal value is `0x23`) are appended to the end of the word to form a key of 128 bits.

Your goal is to write a program to find out the encryption key. You can download an English word list from the Internet or form course Blackboard (`words.txt`). The plaintext, ciphertext, and IV are listed in the following:

```
Plaintext (total 21 characters): This is a secret tool
Ciphertext (in hex format): ece6753e938f8f903cabbbe12d395bf5f7eae38ad918a2d3e1c3a832476d5c7a
IV (in hex format):                 0102030405060708090000a0b0c0d0e0f
```

You need to pay attention to the following issues:

- If you choose to store the plaintext message in a file, and feed the file to your program, you need to check whether the file length is 21. If you type the message in a text editor, you need to be aware that some editors may add a special character to the end of the file. The easiest way to store the message in a file is to use the following command (the `-n` flag tells `echo` not to add a trailing newline):

```
$ echo -n "This is a secret tool" > file
```

- In this task, you are supposed to write your own program to invoke the crypto library. No credit will be given if you simply use the `openssl` commands to do this task. Sample code can be found from the following URL: OpenSSL: EVP Encrypt Int
- When you compile your code using `gcc`, do not forget to include the `-lcrypto` flag, because your code needs the `crypto` library. See the following example:

```
$ gcc -o myenc myenc.c -lcrypto
```

Ans:

To accomplish this task, I'll write a C program that attempts to decrypt the given ciphertext using keys generated from words in an English word list (words.txt).

- **Steps to be performed:**

1. Read the plaintext, ciphertext, and IV provided in the task.
2. Load the English word list from a file.
3. Iterate through each word in the word list.
4. Generate a 128-bit key by appending pound signs (#) to the end of each word until the key length is 128 bits.
5. Attempt to decrypt the ciphertext using the AES-128-CBC cipher and the generated key.
6. If the decryption is successful and matches the provided plaintext, print the key and exit the program.
7. If no key is found after iterating through all words, print a message indicating that the key could not be found.

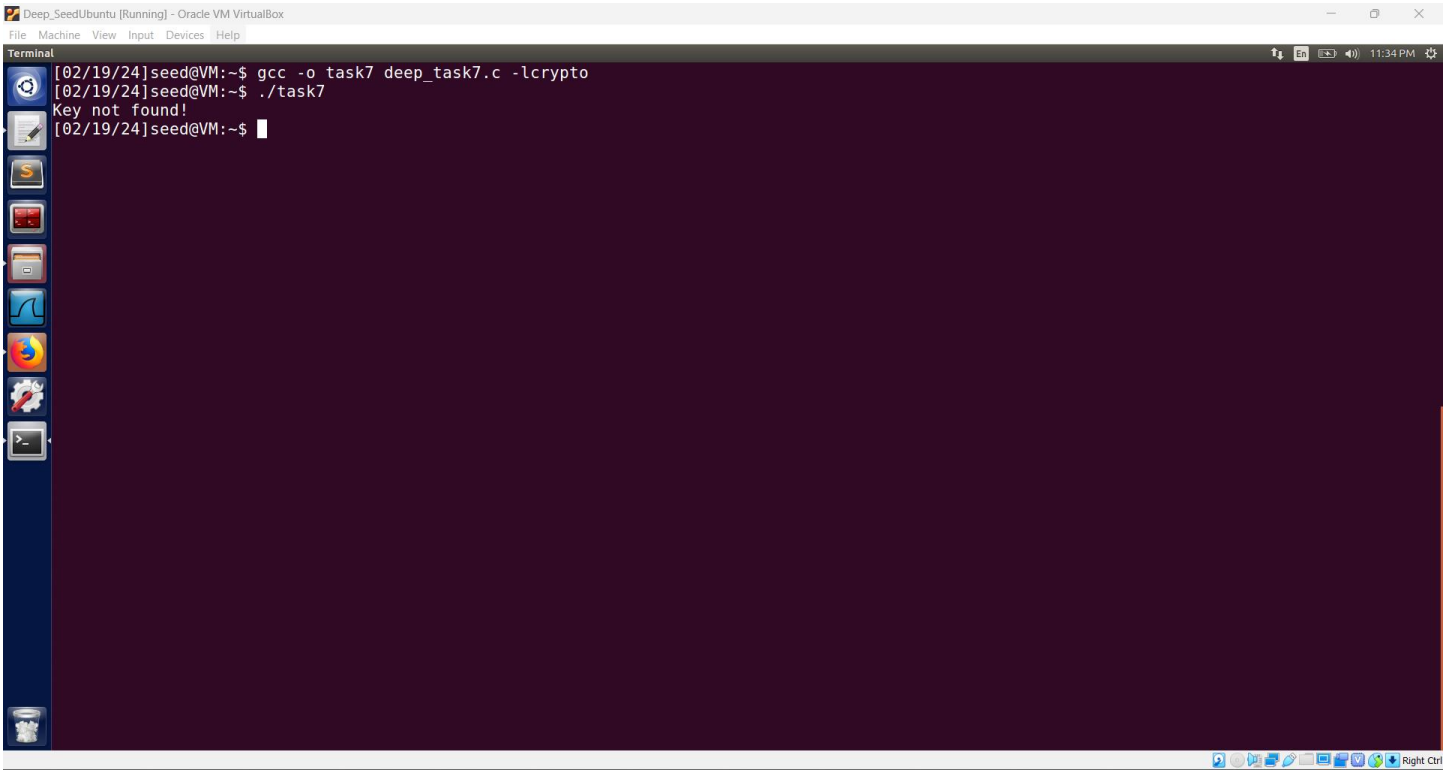
- **Code:**

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

// Function to convert a hexadecimal string to binary data
void hex_string_to_binary(const char* hex_string, unsigned char* binary_data, size_t length)
{
    for (size_t i = 0; i < length; i++)
    {
        sscanf(hex_string + 2 * i, "%2hhx", &binary_data[i]);
    }
}

int main()
{
    // Define the ciphertext and IV in hexadecimal format
    const char* ciphertext_hex =
"ece6753e938f8f903cabbbe12d395bf5f7eae38ad918a2d3e1c3a832476d5c7a";
    const char* iv_hex = "0102030405060708090a0b0c0d0e0f";

    // Convert the ciphertext and IV to binary data
    unsigned char ciphertext[128]; // AES-128 bits
    unsigned char iv[16];
    hex_string_to_binary(ciphertext_hex, ciphertext, strlen(ciphertext_hex) / 2);
    hex_string_to_binary(iv_hex, iv, strlen(iv_hex) / 2);

    // Open the English word list file
    FILE *file = fopen("words.txt", "r");
    if (file == NULL)
    {
        perror("Error opening file");
        return 1;
```

```
}

// Define variables for word processing
char word[16]; // Max word length is 16 characters
size_t word_length;
size_t plaintext_length = 21;
unsigned char plaintext[plaintext_length + 1]; // Null terminator
memset(plaintext, 0, sizeof(plaintext)); // Initialize plaintext buffer

// Read the plaintext from file or console
if (fgets((char*)plaintext, sizeof(plaintext), file) == NULL)
{
   perror("Error reading plaintext");
   fclose(file);
   return 1;
}

// Remove the newline character from plaintext
plaintext_length = strlen((char*)plaintext);
plaintext[plaintext_length - 1] = '\0';

// Loop through the words and try to decrypt the ciphertext
while (fgets(word, sizeof(word), file))
{
   word_length = strlen(word);
   word[word_length - 1] = '\0'; // Remove the newline character

   // Append pound signs (#) to create 128-bit keys
   char key[16]; // AES-128 key size
   size_t key_index = 0;
   while (key_index < sizeof(key))
   {
      key[key_index++] = word[key_index % word_length];
   }

   // Use AES-128-CBC decryption and compare results
   EVP_CIPHER_CTX *ctx;
   int len;
   int plaintext_len;
   unsigned char decrypted[128];

   if (!(ctx = EVP_CIPHER_CTX_new()))
   {
      perror("Error creating new cipher context");
      fclose(file);
      return 1;
   }

   if (EVP_DecryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, (unsigned char*)key, iv) != 1)
```

```
        {
            perror("Error initializing decryption");
            fclose(file);
            return 1;
        }

        if (EVP_DecryptUpdate(ctx, decrypted, &len, ciphertext, sizeof(ciphertext)) != 1)
        {
            perror("Error decrypting ciphertext");
            fclose(file);
            return 1;
        }
        plaintext_len = len;

        if (EVP_DecryptFinal_ex(ctx, decrypted + len, &len) != 1)
        {
            // Ignore "Error finalizing decryption" as it may occur if padding is incorrect
            EVP_CIPHER_CTX_cleanup(ctx);
            continue;
        }
        plaintext_len += len;

        decrypted[plaintext_len] = '\0'; // Ensure null termination

        EVP_CIPHER_CTX_free(ctx);

        // If decryption is successful, print the key and the decrypted plaintext
        if (strncmp((char*)plaintext, (char*)decrypted, plaintext_length) == 0)
        {
            printf("Key found: %s\n", key);
            printf("Decrypted plaintext: %s\n", decrypted);
            fclose(file);
            return 0; // Exit successfully after finding the key
        }
    }

    fclose(file);
    printf("Key not found!\n");
    return 0;
}
```

- **Output:**



3. DES/AES explained as a Flash animation

- DES: https://www.youtube.com/watch?v=Vcld7CMAnNs
- AES: https://www.youtube.com/watch?v=gP4PqVGudtg

4. Submission

You need to submit 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 are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

Your report must be a .pdf file that is uploaded to Blackboard for