Mini Project 1: Secret-Key Encryption

Project Topic: Experiments on Symmetric-Key Cryptography Principles

Task 1: Frequency Analysis

Step 1: We generate an encryption key; we use the given python program to produce an encryption key. An encryption key is just shuffled alphabet, which is later used to for monoalphabetic substitution.

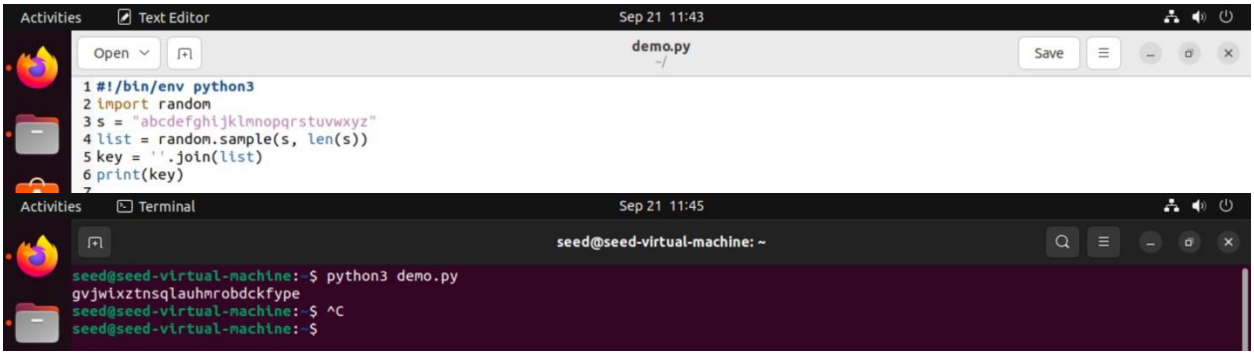


Fig 1.1

Fig 1.1 shows the code and the encryption key produced after executing the code.

Step 2: In this step we create a text file called as “article.txt” which contains an example sentence. The purpose of this step is to demonstrate simplification. This is achieved by converting all upper case characters to lower case characters, then removing all punctuations and numbers. In the real monoalphabetic cipher the spaces are removed but to keep the demonstration simple we keep the spaces.

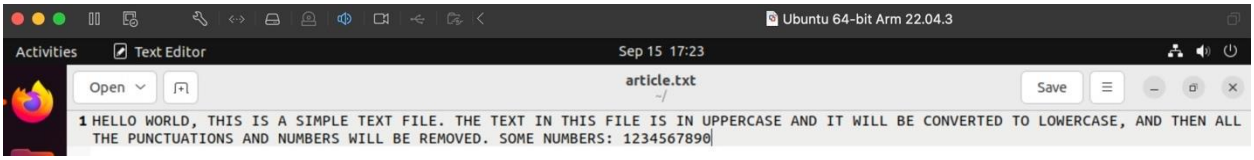


Fig 1.2.1

In Fig 1.2.1, we can see that the text file named “article.txt” has a sentence in all upper case letters for easy demonstration and it also contains number at the end.

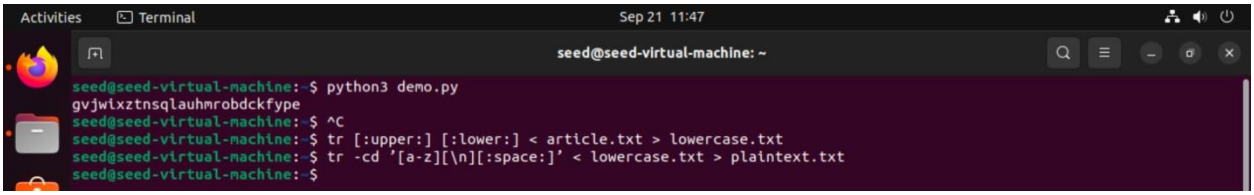


Fig 1.2.2

In Fig 1.2.2, we have executed two commands for simplification of the text present in “article.txt” file.

Command 1: tr [:upper:] [:lower:] < article.txt > lowercase.txt

With this command, all uppercase letters in the "article.txt" file are changed to their lowercase equivalents, and the transformed content is saved in a new file called "lowercase.txt." Numbers and punctuation are unaffected by this alteration of other characters. This can be seen in the Fig 1.2.3 given below.

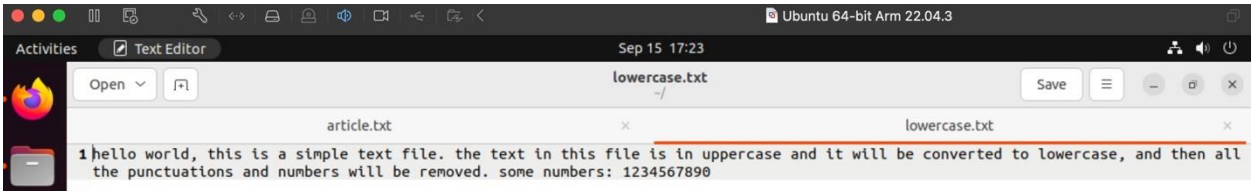


Fig 1.2.3

Command 2: tr -cd ’[a-z][\n][:space:]’ < lowercase.txt > plaintext.txt
This command processes the text from the "lowercase.txt" file. It removes all characters that are not lowercase letters, newline characters, or whitespace characters. The resulting filtered text, which does not contain any number, is then saved in a new file called "plaintext.txt." This can be seen in the Fig 1.2.4 given below.

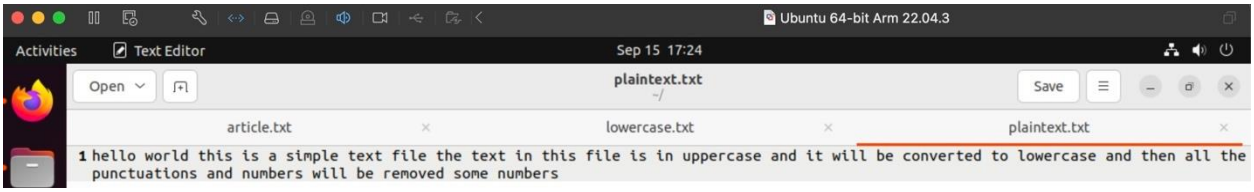


Fig 1.2.4

Step 3: In this step we encrypt the “plaintext.txt” file.

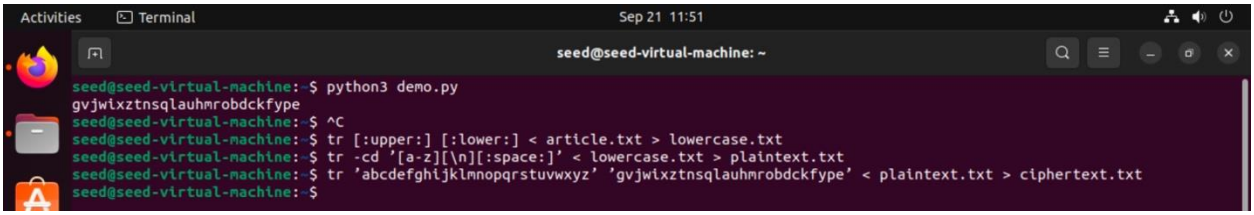


Fig 1.3.1

Fig 1.3.2 shows the command used to perform the encryption.

Command: tr ’abcdefghijklmnopqrstuvwxyz’ ’gvjwixztnsqlauhmrobdckfype’ < plaintext.txt > ciphertext.txt
This is the encryption key we got in the step 1: “gvjwixztnsqlauhmrobdckfype”.
This command substitutes characters in the "plaintext.txt" file based on a specified mapping. It replaces each letter from the alphabet 'a' to 'z' with the corresponding letter from the encryption key. The resulting transformed text is then saved in the "ciphertext.txt" file. The Fig 1.3.2 given below show the transformed text in the file “ciphertext.txt”.

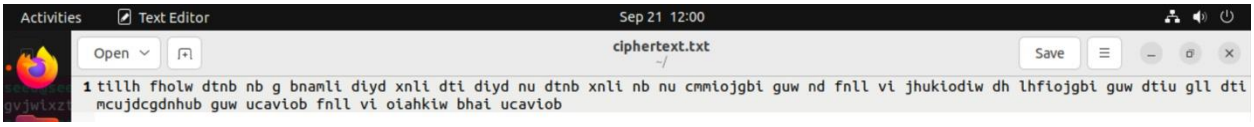


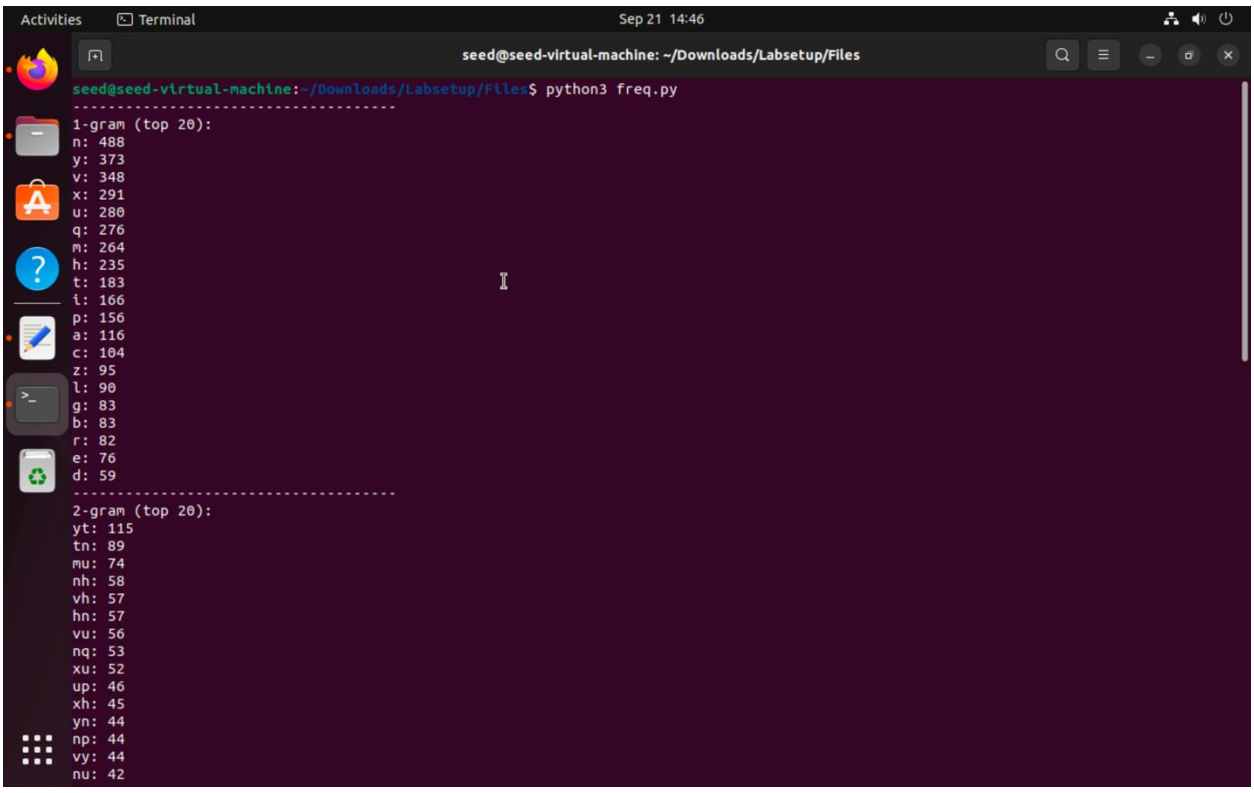
Fig 1.3.2

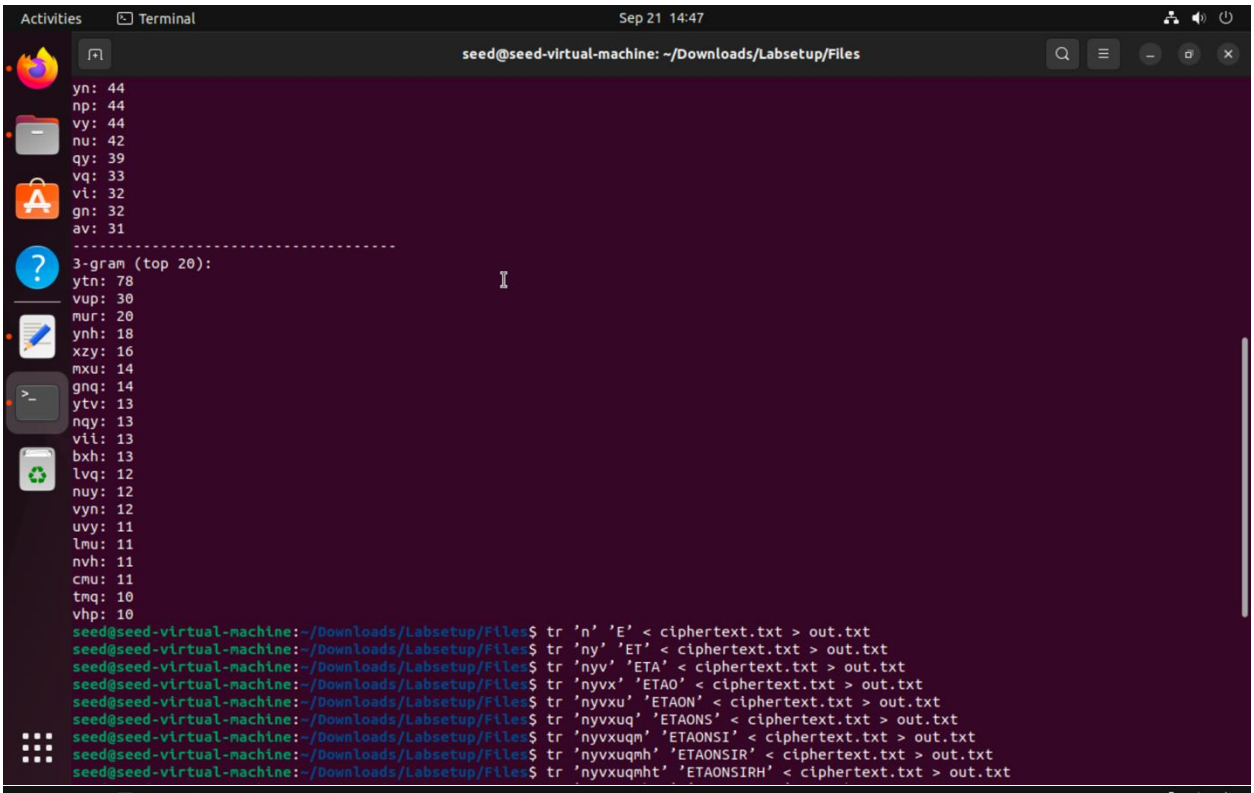
Now, to finish the Task 1, it is required to use Frequency Analysis to figure out the encryption key and the original plain text. The Python code provided in the file “freq.py” inside the Labsetup/Files folder.

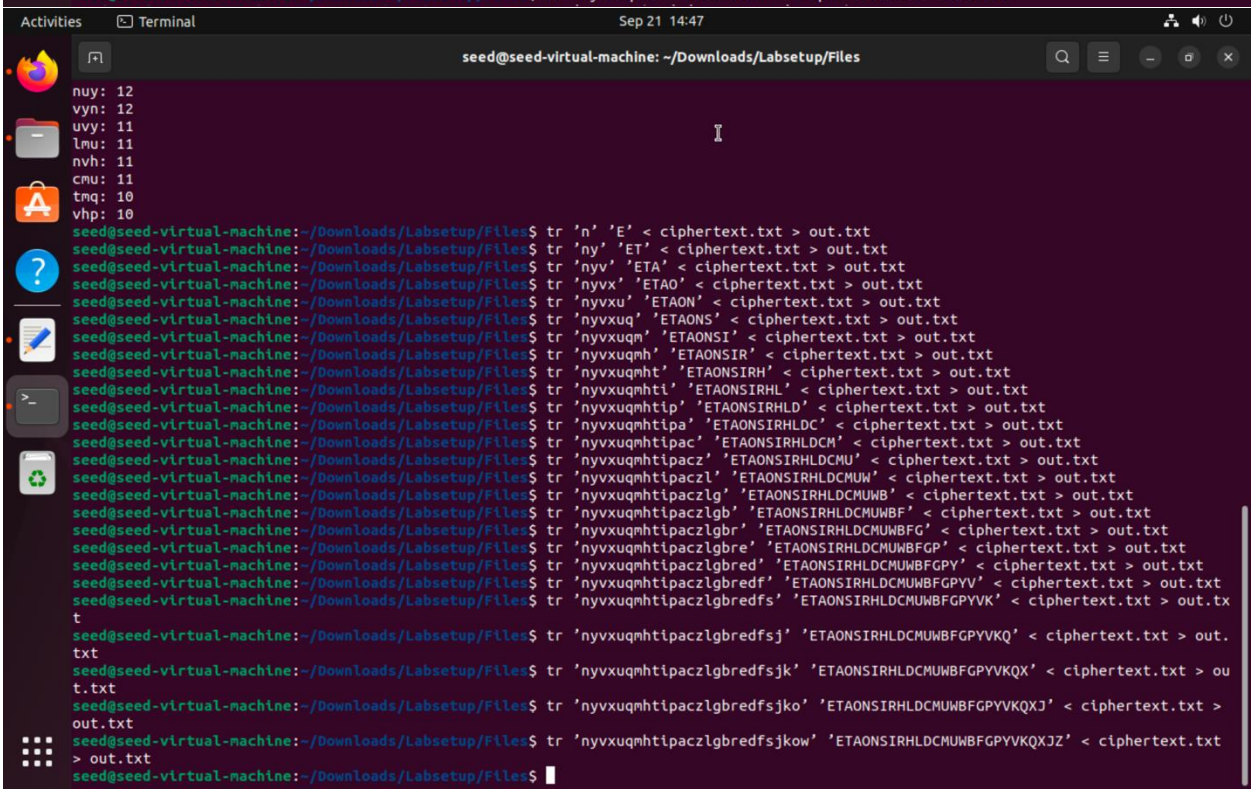
- By running the provided “freq.py” file we generated statistics for n-grams in the ciphertext. This helped us identify common letter sequences in English.
- We started with single-letter frequencies (1-gram) to identify the most common letters in the ciphertext. Match these with their English counterparts which we found here (single-letter frequency graph) (e.g., if 'n' is the most common letter, assume it corresponds to 'e').
- Then we used the identified substitutions to update the ciphertext using the “tr” command to get more clues about the key and plaintext.
- Then we used the bigram (2-gram) and trigram (3-gram) frequencies to find common letter pairs and triplets. This helped us refine our substitutions. (Bigram frequencies and Trigram frequencies)
- We continued this process iteratively, making educated guesses based on frequency analysis, updating the ciphertext, and refining our substitutions.
- Eventually, you were able to decode most of the text, which will give us more clues about the key and made it easier to decrypt the remaining portions.
- We also kept track of your substitutions and progress and used capital letters for the plaintext to distinguish it from the ciphertext.

The encryption key is “vgapnbrtmosicuxejhqyzflkdw”.

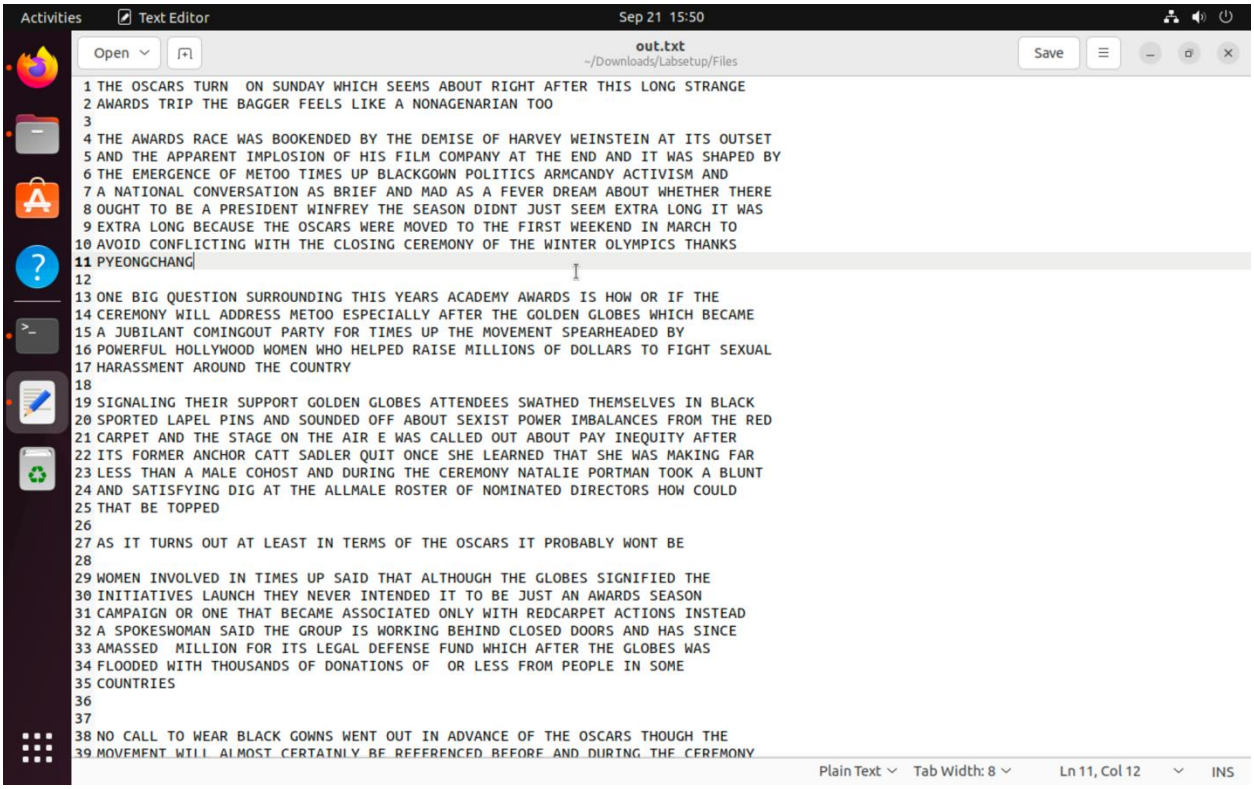
Below are the screenshots of the frequency statistics for 1-gram, 2-gram, and 3-gram along with the commands used to replace the characters and automatically store them in a new file called “out.txt”.




Activities
Terminal
Sep 21 14:47
seed@seed-virtual-machine: ~/Downloads/Labsetup/Files
yn: 44
np: 44
vy: 44
nu: 42
qy: 39
vq: 33
vi: 32
gn: 32
av: 31
3-gram (top 20):
ytn: 78
vup: 30
mur: 20
ynh: 18
xzy: 16
mxu: 14
gnq: 14
ytv: 13
nqy: 13
vii: 13
bxh: 13
lvq: 12
nuy: 12
vyn: 12
uvy: 11
lmu: 11
nvh: 11
cmu: 11
tmq: 10
vhp: 10
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'n' 'E' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'ny' 'ET' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyv' 'ETA' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvx' 'ETAO' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxu' 'ETAON' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuq' 'ETAONS' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqm' 'ETAONSI' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmh' 'ETAONSIR' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmht' 'ETAONSIRH' < ciphertext.txt > out.txt


Activities
Terminal
Sep 21 14:47
seed@seed-virtual-machine: ~/Downloads/Labsetup/Files
nuy: 12
vyn: 12
uvy: 11
lmu: 11
nvh: 11
cmu: 11
tmq: 10
vhp: 10
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'n' 'E' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'ny' 'ET' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyv' 'ETA' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvx' 'ETAO' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxu' 'ETAON' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuq' 'ETAONS' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqm' 'ETAONSI' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmh' 'ETAONSIR' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmht' 'ETAONSIRH' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhti' 'ETAONSIRHL' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtip' 'ETAONSIRHLD' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipa' 'ETAONSIRHLDC' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipac' 'ETAONSIRHLDCM' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipacz' 'ETAONSIRHLDCMU' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczl' 'ETAONSIRHLDCMUW' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlg' 'ETAONSIRHLDCMUWB' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgb' 'ETAONSIRHLDCMUWBF' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbr' 'ETAONSIRHLDCMUWBFG' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbre' 'ETAONSIRHLDCMUWBFGP' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbred' 'ETAONSIRHLDCMUWBFGPY' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbredf' 'ETAONSIRHLDCMUWBFGPYV' < ciphertext.txt > out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbredfs' 'ETAONSIRHLDCMUWBFGPYVK' < ciphertext.txt > out.tx
t
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbredfsj' 'ETAONSIRHLDCMUWBFGPYVKQ' < ciphertext.txt > out.
txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbredfsjk' 'ETAONSIRHLDCMUWBFGPYVKQX' < ciphertext.txt > ou
t.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbredfsjko' 'ETAONSIRHLDCMUWBFGPYVKQXJ' < ciphertext.txt >
out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$ tr 'nyvxuqmhtipaczlgbredfsjkow' 'ETAONSIRHLDCMUWBFGPYVKQXJZ' < ciphertext.txt
> out.txt
seed@seed-virtual-machine:~/Downloads/Labsetup/Files$

The screenshot below shows the original plain text that is stored on a text file called “out.txt”. Every time the “tr” command is executed the out.txt file gets updated with the substituted alphabet.



To conclude, we can say that this task highlights the effectiveness of frequency analysis and systematic decryption techniques in deciphering monoalphabetic substitution ciphers, showcasing the importance of pattern recognition and iterative decryption in the process.

Task 2: Encryption using Different Ciphers and Modes

1. AES (aes-192-ctr)

- In this example, we use Advanced Encryption Standard (AES) which uses a 192-bit key for encryption.
- When used, CTR mode converts a block cipher into a stream cipher by encrypting a counter and XORing the result with the plaintext. This style is renowned for being parallelizable and efficient.

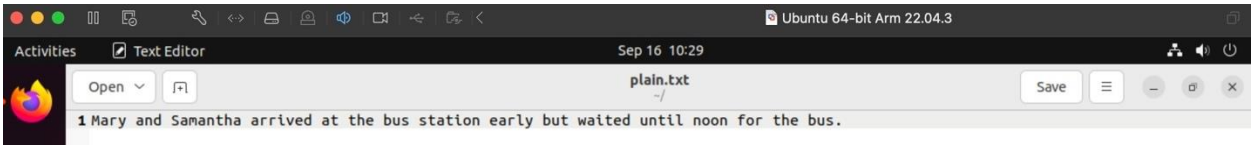


Fig: The plain text.

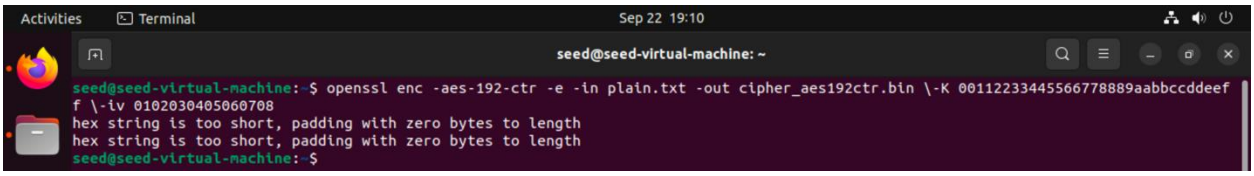


Fig: The plain text encrypted using AES.

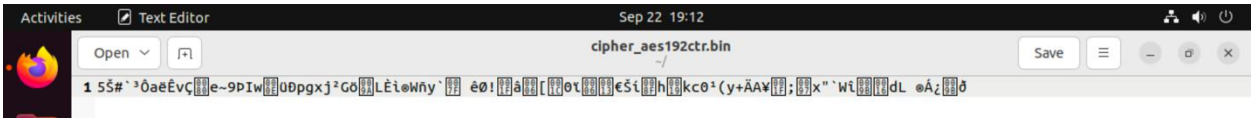


Fig: The encrypted text.

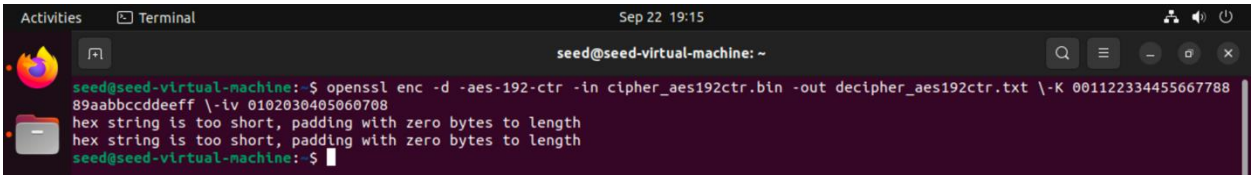


Fig: The decryption command.

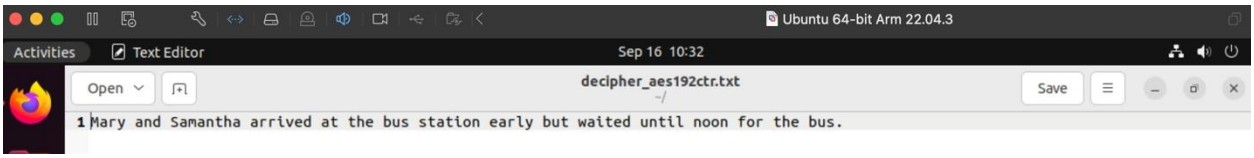


Fig: The original text after decryption.

2. ARIA (aria-256-cfb)

- In this example, we use ARIA, a South Korean encryption algorithm. It uses a 256-bit key for encryption.
- The CFB mode of operation enables the encryption of single bytes as opposed to whole blocks. In this mode, the following plaintext byte is encrypted using the preceding ciphertext block.

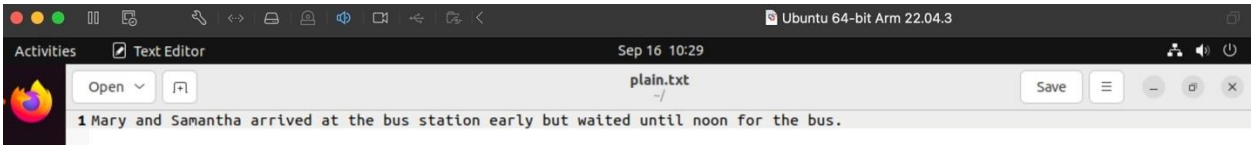


Fig: The plain text.

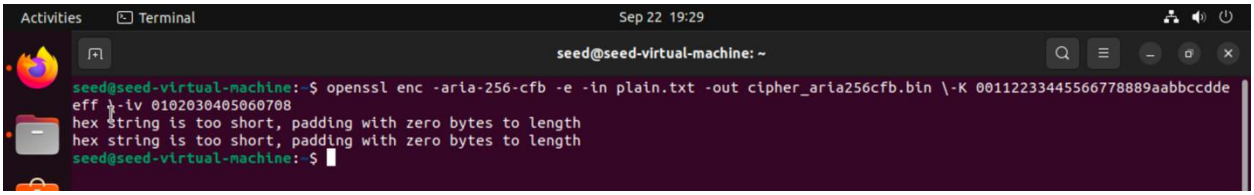


Fig: The plain text encrypted using ARIA.

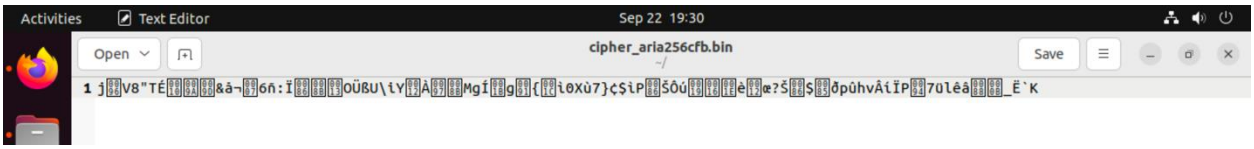


Fig: The encrypted text.

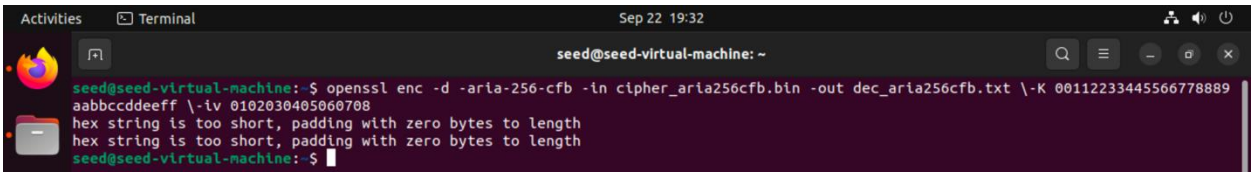


Fig: The decryption command.

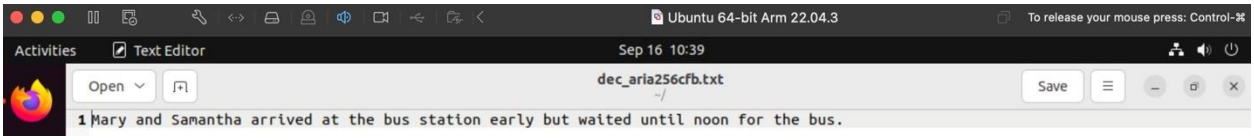


Fig: The original text after decryption.

3. CAMELLIA (camellia-128-cbc)

- In this example, we use Camellia, which is a symmetric key block cipher and uses a 128-bit key for encryption.
- Each block of plaintext in CBC mode is encrypted after being XORed with the block of ciphertext that came before it. It offers discretion and is frequently used for secure conversations.

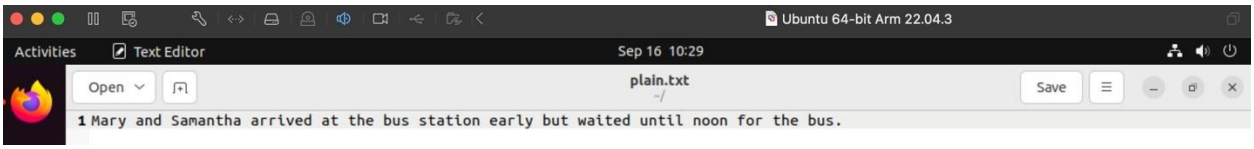


Fig: The plain text.

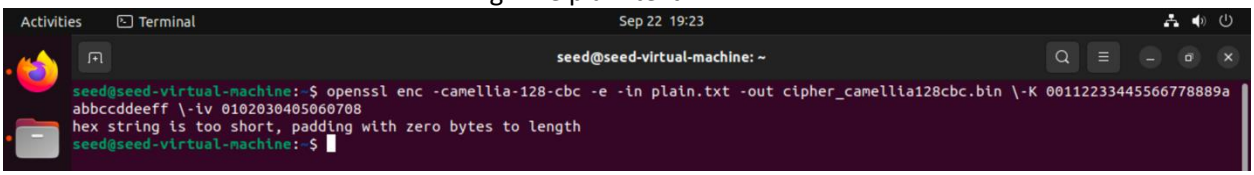


Fig: The plain text encrypted using CAMELLIA.

Fig: The encrypted text.

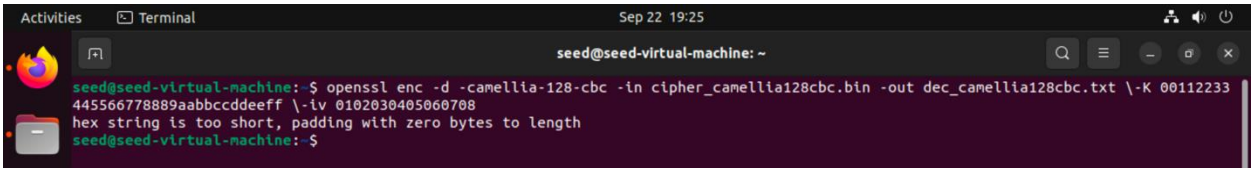


Fig: The decryption command.

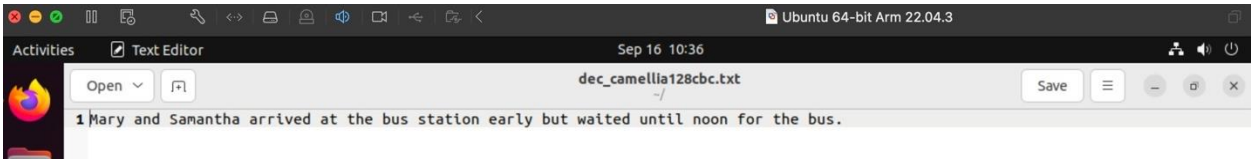


Fig: The original text after decryption.

Observation:
In all of these experiments the following text came up "hex key too short, padding with zero bytes to length". This indicates that the key specified for encryption is shorter than what is expected by the encryption algorithm for example, AES-192 requires a 192-bit key, which is 24 bytes, and AES-256 requires a 256-bit key, which is 32 bytes. When this happens, the encryption tool (in this case, OpenSSL) will automatically pad the provided key with zero bytes to meet the required key length for the chosen encryption algorithm.

Since both the key and IV are supplied in the command, OpenSSL doesn't need to interactively ask the user for an encryption password. This can be useful in scenarios where you want to automate encryption processes and don't want to rely on manual input.

Task 3: Encryption Mode – ECB vs. CBC

Encrypting an image file using different encryption modes (ECB and CBC) and then analyzing the results is an interesting experiment to understand the differences between these modes and their impact on image encryption. Below are the steps to perform this experiment using the provided image file (pic_original.bmp) and some additional observations with a picture of my choice.

Step 1: Encrypt the Image

Fig: pic_original.bmp

This is the original pic that we will encrypt now using the following commands.

Encrypt using ECB mode:
$ openssl enc -aes-256-ecb -e -in pic_original.bmp -out enc_ecb.bmp

Encrypt using CBC mode:
$ openssl enc -aes-256-cbc -e -in pic_original.bmp -out enc_cbc.bmp

This will create two encrypted image files: enc_ecb.bmp and enc_cbc.bmp.

To make the encrypted files appear as legitimate BMP images, we need to replace their headers with the headers from the original image. We can use the commands you provided:

Extract header from pic_original.bmp.
$ head -c 54 pic_original.bmp > header

Extract data from encrypted_ecb.bmp (skip the first 54 bytes).
$ tail -c +55 enc_ecb.bmp > body_ecb

Extract data from encrypted_cbc.bmp (skip the first 54 bytes).
$ tail -c +55 enc_cbc.bmp > body_cbc

Combine headers and bodies to create new image.
$ cat header body_ecb > new_ecb.bmp
$ cat header body_cbc > new_cbc.bmp

Step 3: Display the Encrypted Images

Now, let's view the encrypted images using an image viewer like “eog”
$ eog new_ecb.bmp
$ eog new_cbc.bmp

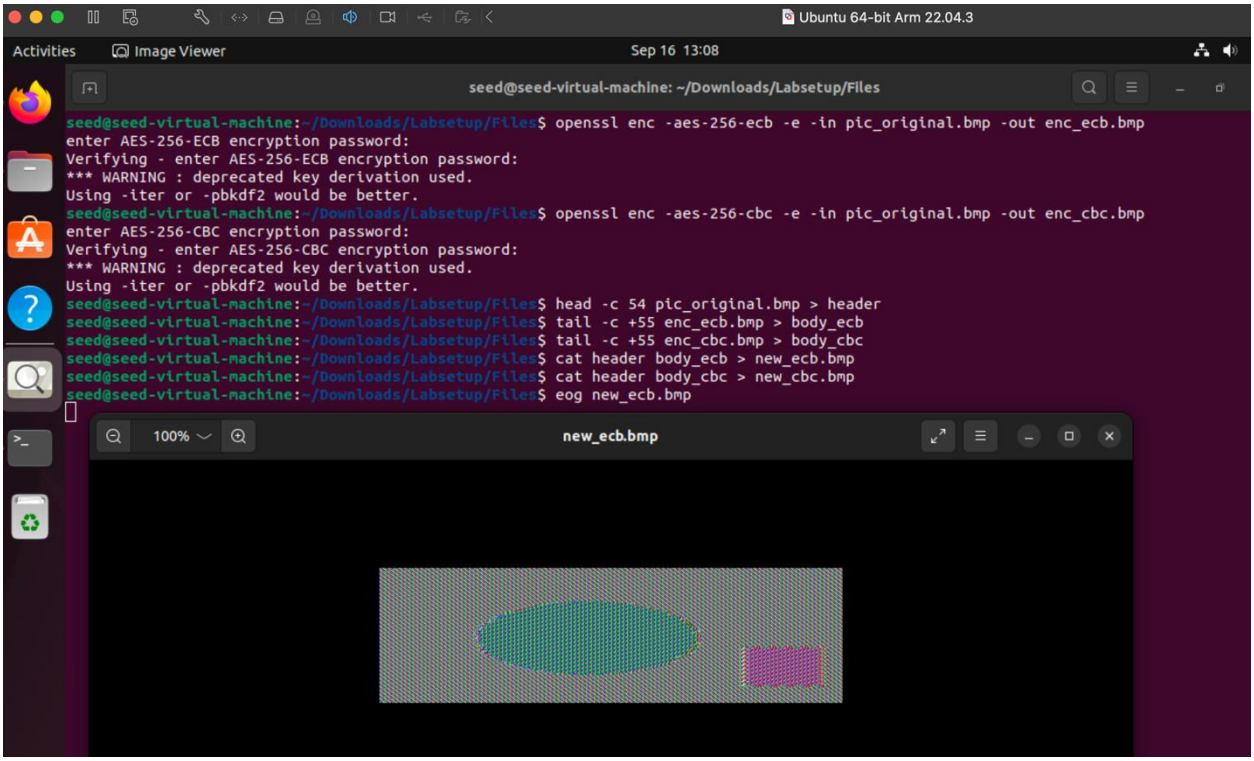


Fig: Encrypted using ECB mode

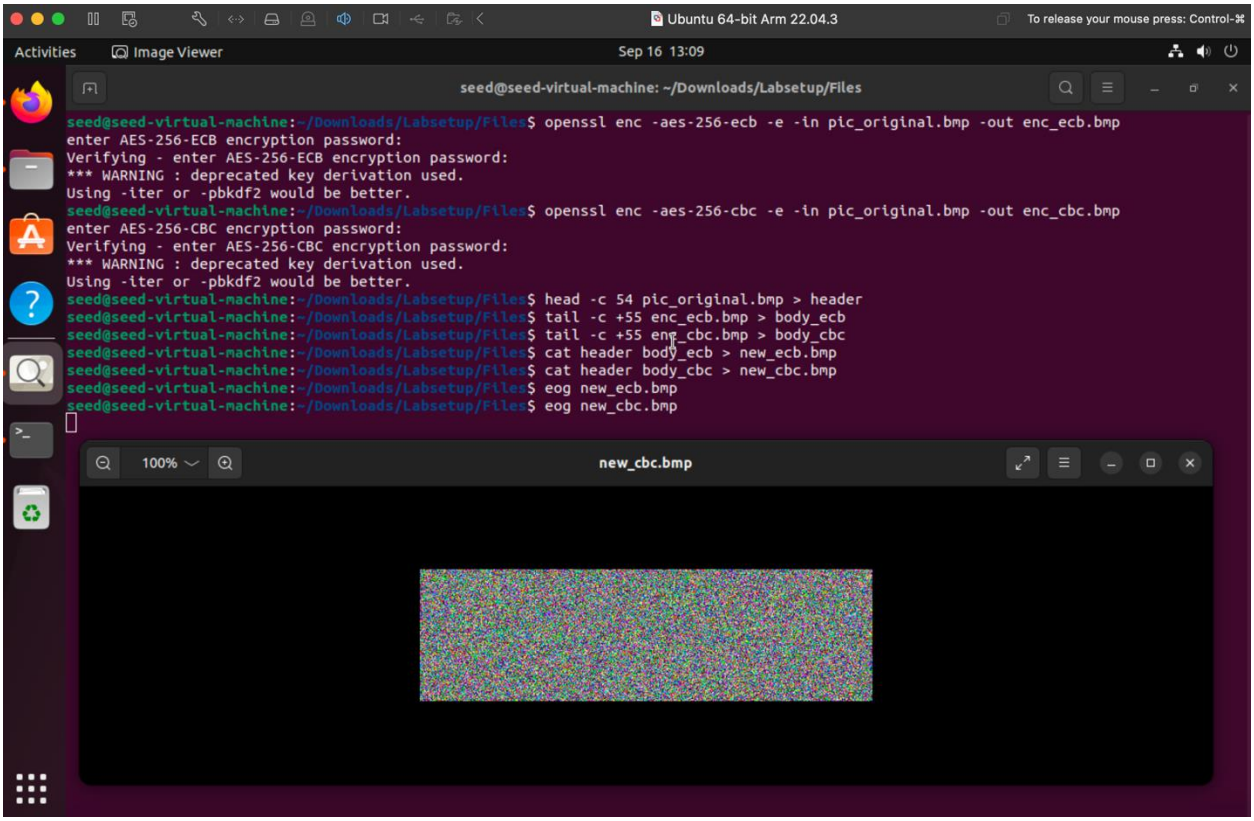


Fig: Encrypted using CBC mode

Observations:
ECB Mode: There is probably some repetition and visual organization in the encrypted picture when you examine “new_ecb.bmp”. The ECB mode separately encrypts each block of data, which might result in patterns that are easy to see, especially in pictures with distinctive patterns or colors. You could see sections or blocks that resemble different areas of the original image, depending on the content of the image.

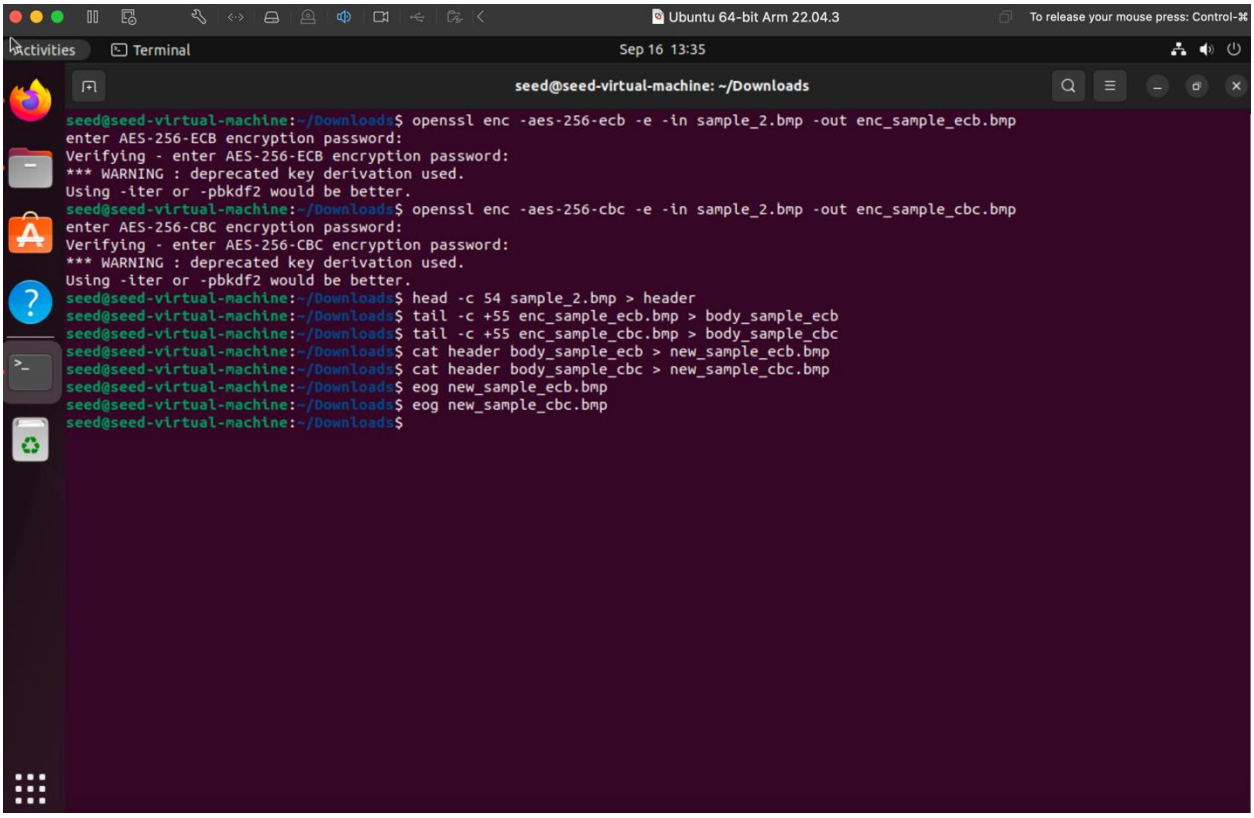
CBC Mode: If you look at “new_cbc.bmp”, you'll see that the encrypted picture looks less organized and lacks the distinct patterns that you may observe in ECB mode. This is so that patterns may be hidden, and greater security can be provided, as CBC mode encrypts the current block using feedback from earlier blocks.

Additional Experiment with a Different Image:

For an additional experiment, we repeated the same process with a different image. Encrypt the image using both ECB and CBC modes, replace the headers, and observe the differences. The new image in the same .bmp format is shown below.

Fig: sample_2.bmp

The screenshot below shows the same set of commands being executed for the sample_2.bmp image file.



This is the “new_sample_ecb.bmp” file which is the which is encrypted using ECB mode.

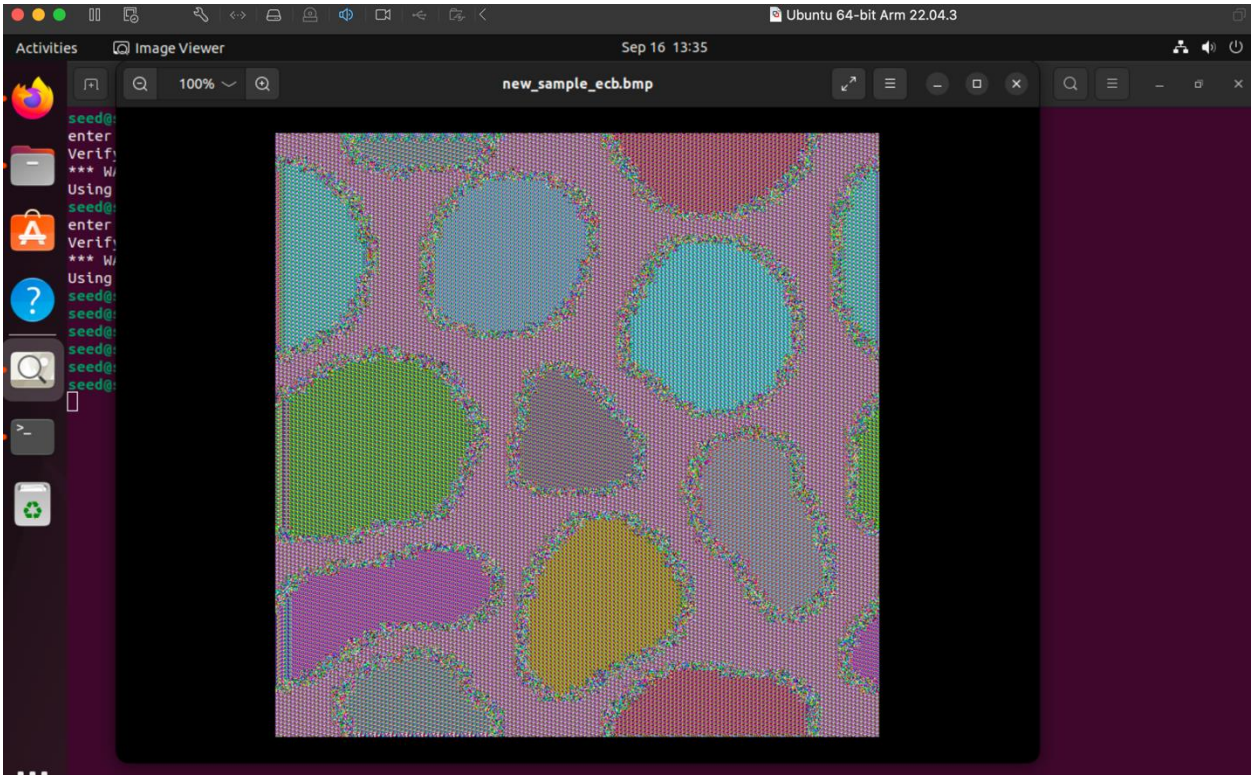


Fig: New image encrypted using ECB mode

This is the “new_sample_cbc.bmp” file which is the which is encrypted using CBC mode.

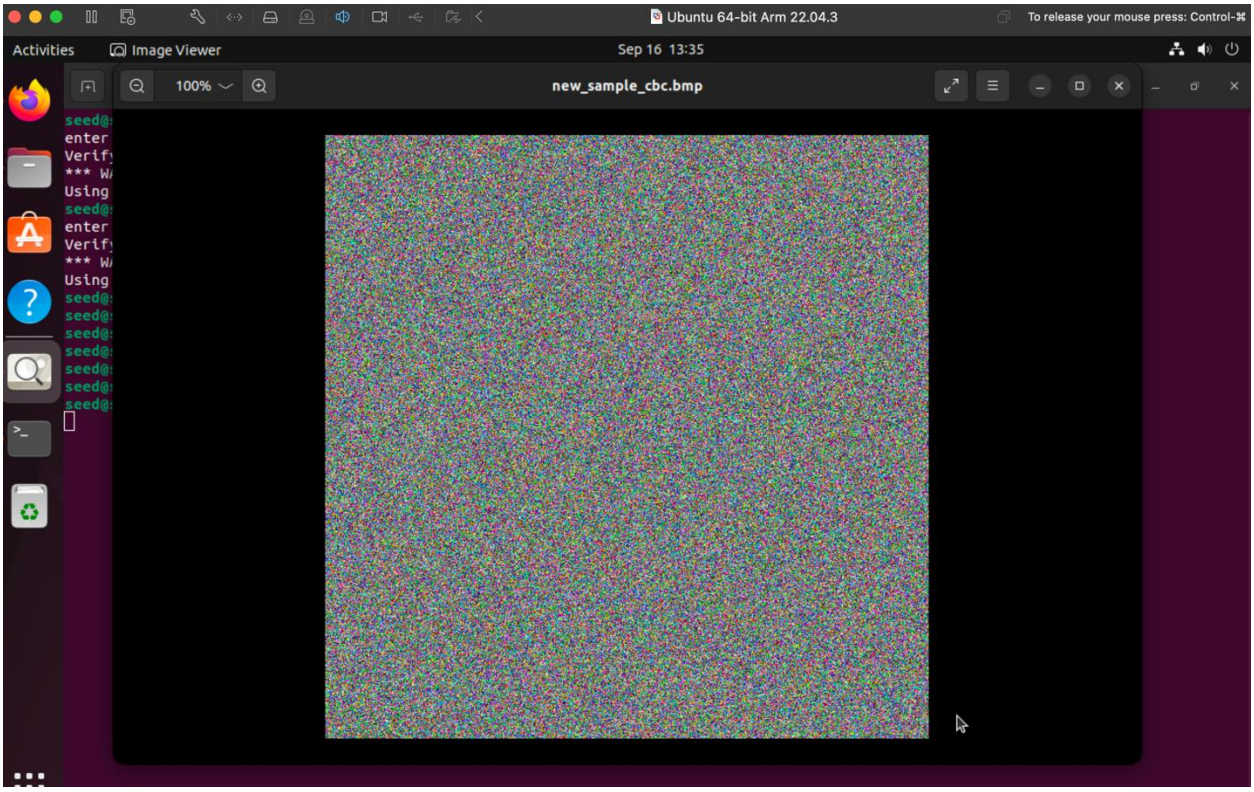
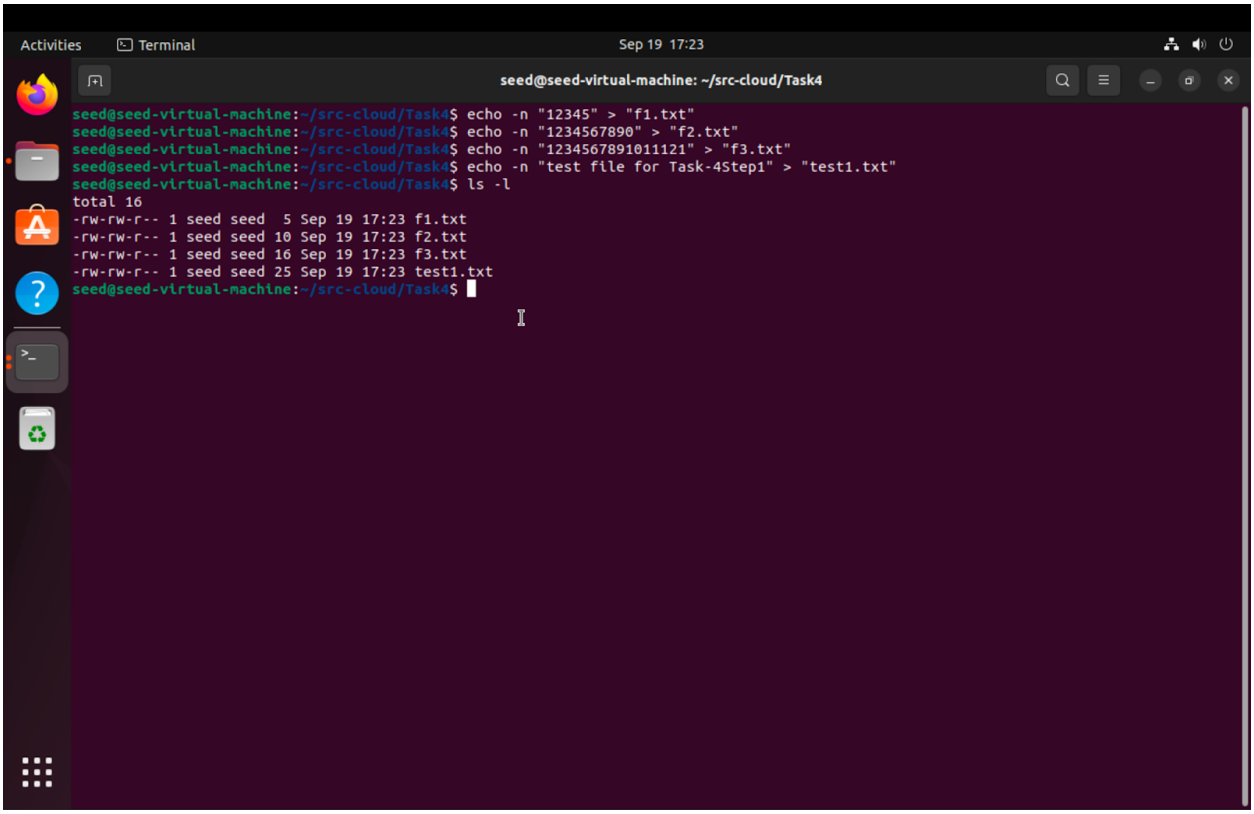


Fig: New image encrypted using CBC mode

Observation:
We found that the same observations hold true: ECB mode reveals patterns in the encrypted image, while CBC mode produces a more randomized result. This experiment helps highlight the importance of using appropriate encryption modes for different types of data and the impact it has on the security of encrypted content.

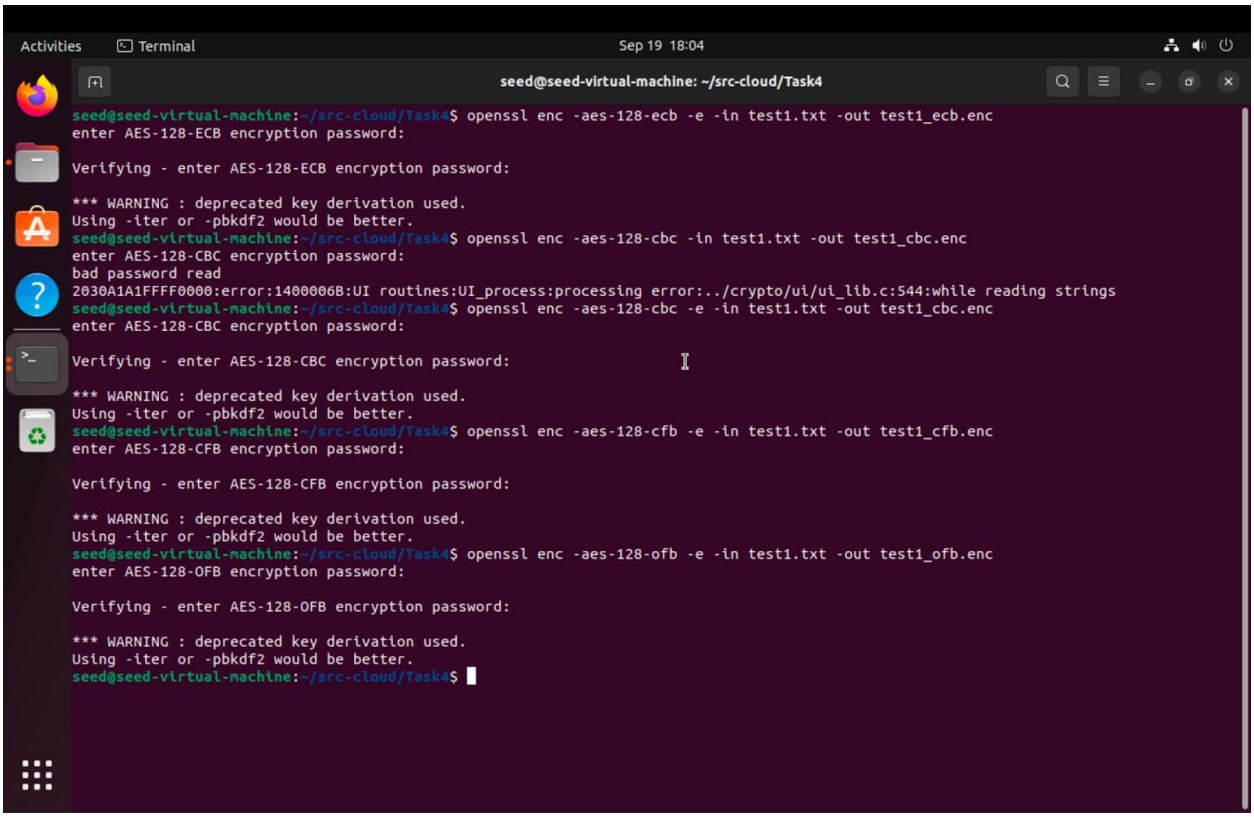
Task 4: Padding

Making the necessary files

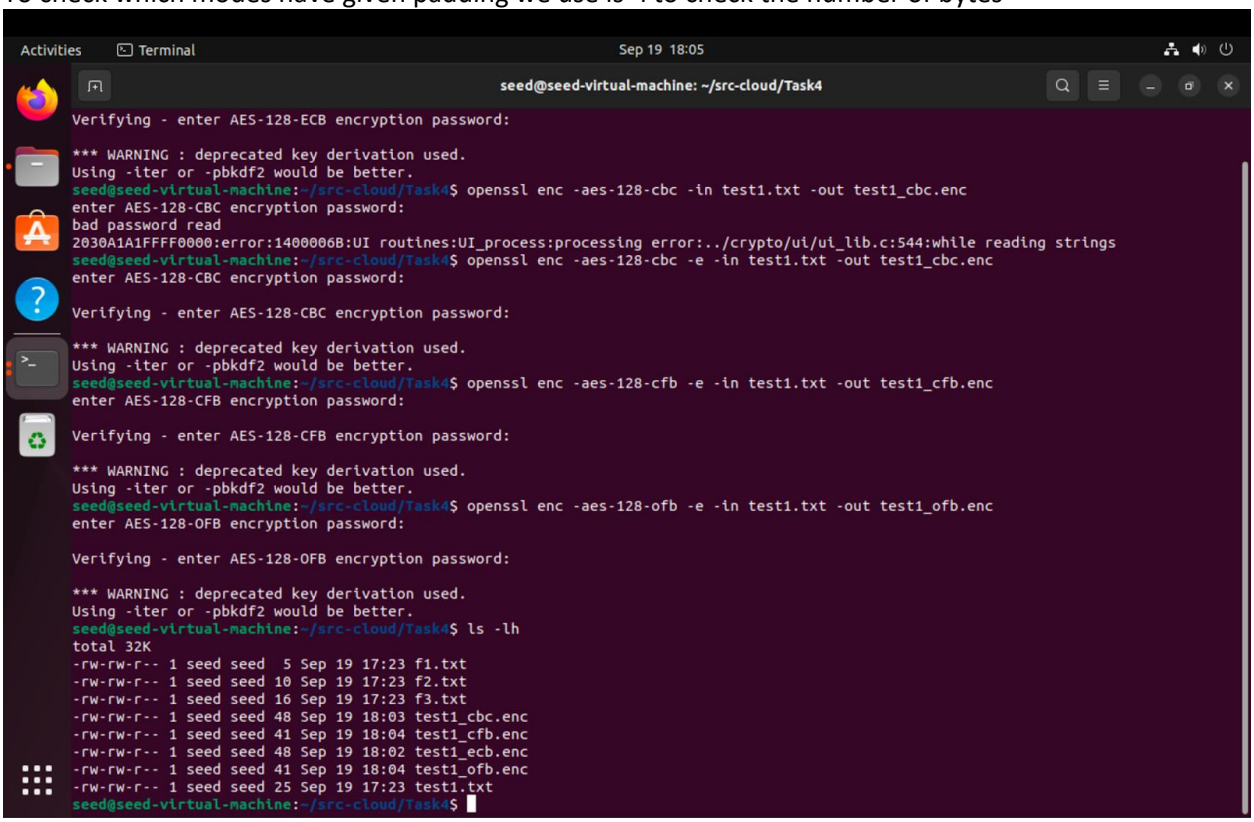


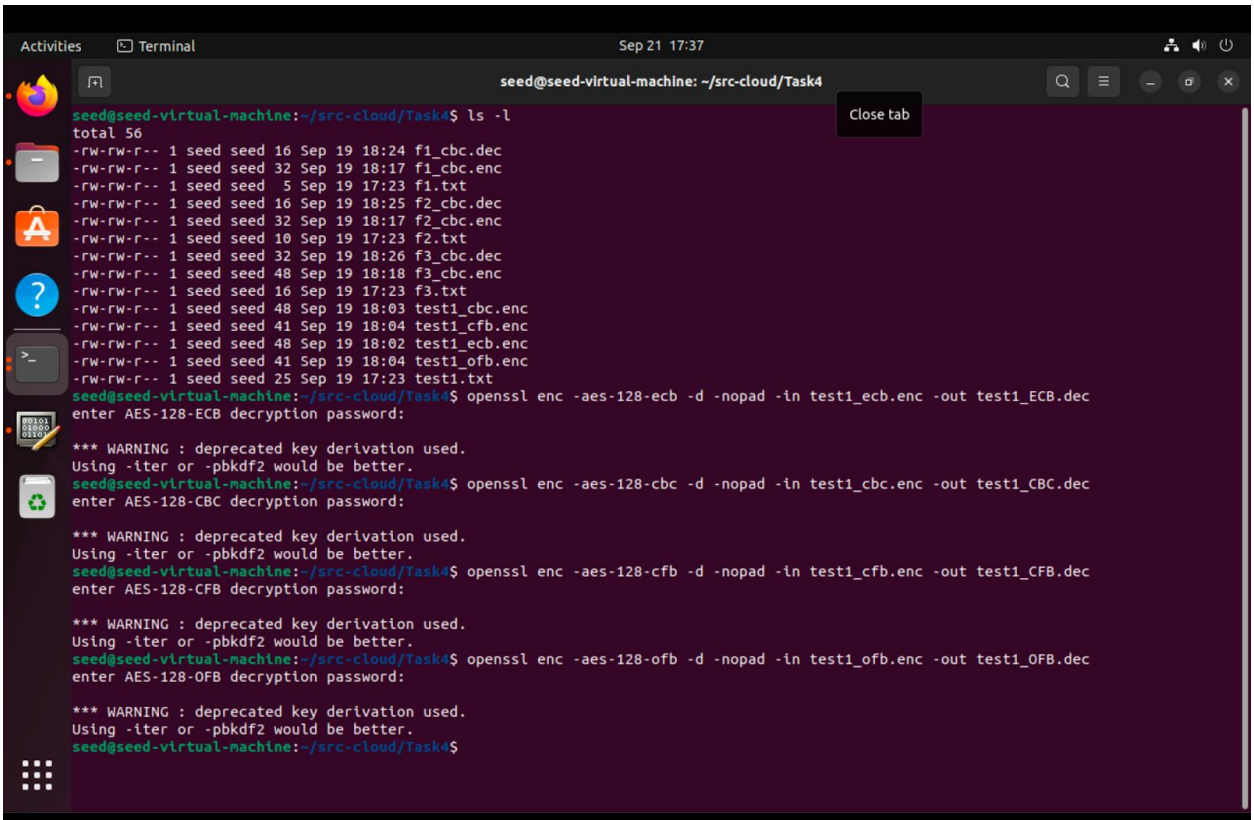
Here we can see that we have made 4 files f1.txt, f2.txt, f3.txt for the 2) step in the task which are 5,10,16 bytes respectively and test1.txt which is our own file of size 25 bytes

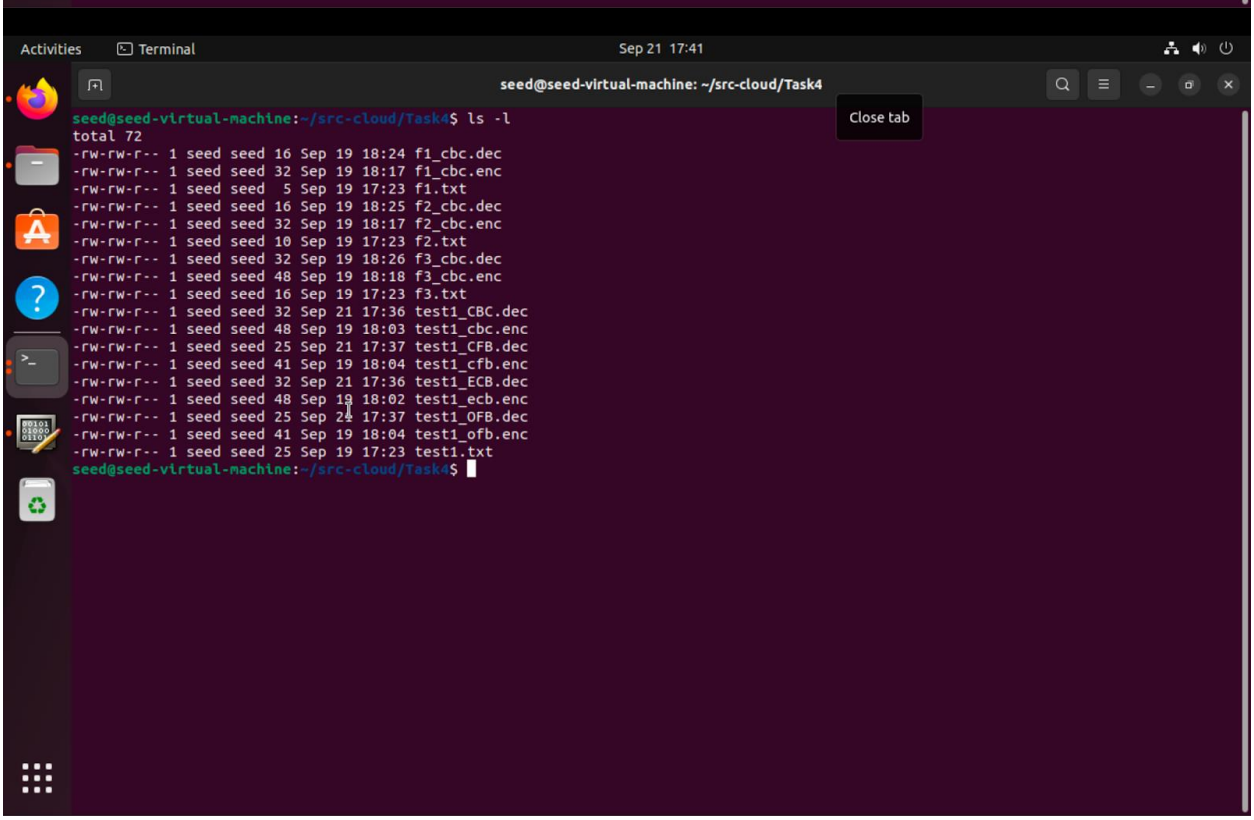
1)
We have to use ECB, CBC, CFB and OFB to encrypt the file test1.txt
Done as follows:
Password: 2537167



To check which modes have given padding we use ls -l to check the number of bytes







According to this :
We can see that ECB and CBC will pad the 25 byte file with 7 bytes to make it 32 bytes.

CFB and OFB will not give any padding as it turns a block cipher into a stream cipher therefore it doesn't need any padding

ECB Mode: Each block is encrypted separately from the others in ECB mode.Padding is a method used to make sure that the plaintext message to be encrypted is a multiple of the block size before encryption occurs in the ECB (Electronic Codebook) style of encryption.In our example padding is done.When padding is applied in ECB mode, additional bytes are often added to the plaintext to increase its size to a multiple of the block size.

CBC Mode: In this mode before encryption, each block in CBC mode is XORed with the block of ciphertext before it. This tells us that during the encryption the ciphertext of the previous block must be known in order for the block to be encrypted. Padding is necessary to make the plaintext length a multiple of the block size in order to guarantee that the final block can be encrypted.Padding is done according to the PKCS#5 Padding scheme which is commonly used.
To make sure that the plaintext message to be encrypted is a multiple of the block size before encryption occurs, the CBC technique of encryption uses padding. Similar to ECB mode, CBC mode necessitates the division of the plaintext into blocks, with each block having a size that corresponds to the block size of the encryption method (for example, 128 bits for AES-128).

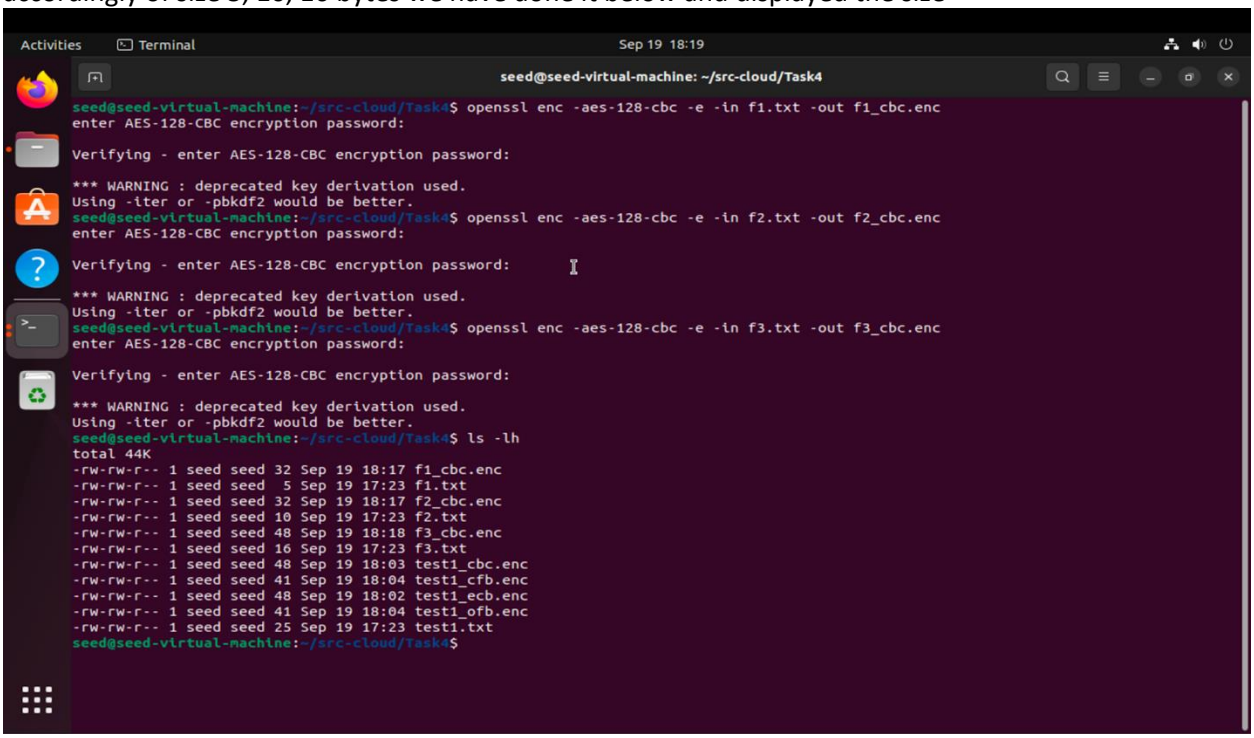
CFB and OFB Mode: CFB and OFB does not add padding as it turns the block cipher into a stream cipher so it can handle a plaintext file with any length without any padding necessary.
In our example we can see no padding is done.Padding is often employed differently in CFB and OFB encryption modes than it is in block cipher modes like CBC or ECB. Both the CFB and OFB stream cipher modes work bit-by-bit or byte-by-byte rather than splitting the plaintext into predetermined blocks.

In conclusion, padding is needed when the plaintext size is not a multiple of the block size for CBC and ECB forms of encryption. This is necessary since both modes work with fixed-size blocks, and padding provides appropriate alignment if the plaintext is not aligned with the block size. In both CBC and ECB modes, padding is required because the encryption of each block depends on the ciphertext block that came before it.

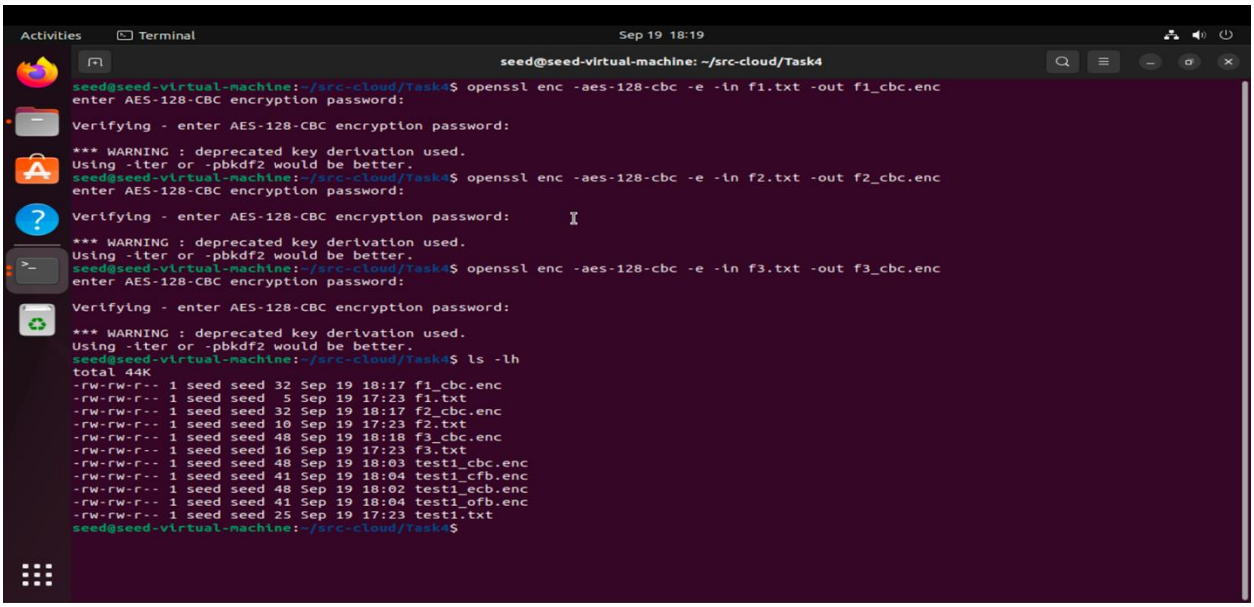
To accomplish block alignment, however, the CFB and OFB stream cipher modes do not have the same severe requirements.

2)

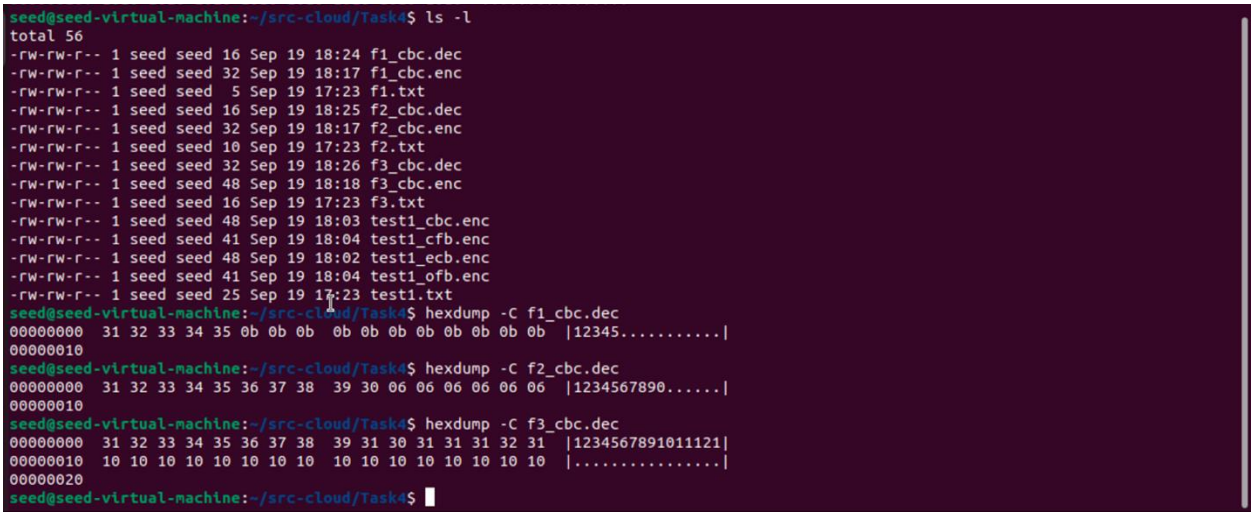
Here we do enc to the files f1.txt , f2.txt , f3.txt to find out the padding that takes place for the files accordingly of size 5, 10, 16 bytes we have done it below and displayed the size



Doing decryption along with the -nopad option.



Here we can see for different files the padding is done as follows:



f1.txt padding is 11 bytes from 5 bytes to get a total of 16 bytes
f2.txt padding is 6 bytes from 10 bytes to get a total of 16 bytes
F3.txt padding is 16 bytes from 16 bytes to get a total of 32 bytes

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?

Information Recovered According to different Modes are as Follows:

- **ECB:** Every Block except the corrupted block can be recovered. ie the block with the 55th byte is not recoverable
  ECB we can get all the information recovered except the corrupted Block. This is because ECB encrypts each block separately and independently. If a bit in a block gets corrupted, then only that block gets corrupted and not recoverable.

- **CBC:** The block containing the corrupted byte and the next block is unrecoverable.
  In CBC mode, if a byte in the encrypted file is changed, the associated block of plaintext is corrupted as well as the subsequent block of plaintext, which is completely altered and corrupted. This occurs as a result of the fact that every block of plaintext is combined with the block before it in the ciphertext, and thus altering only one byte in the ciphertext can lead to a series of mistakes throughout the decryption process. The remaining plaintext is not altered.

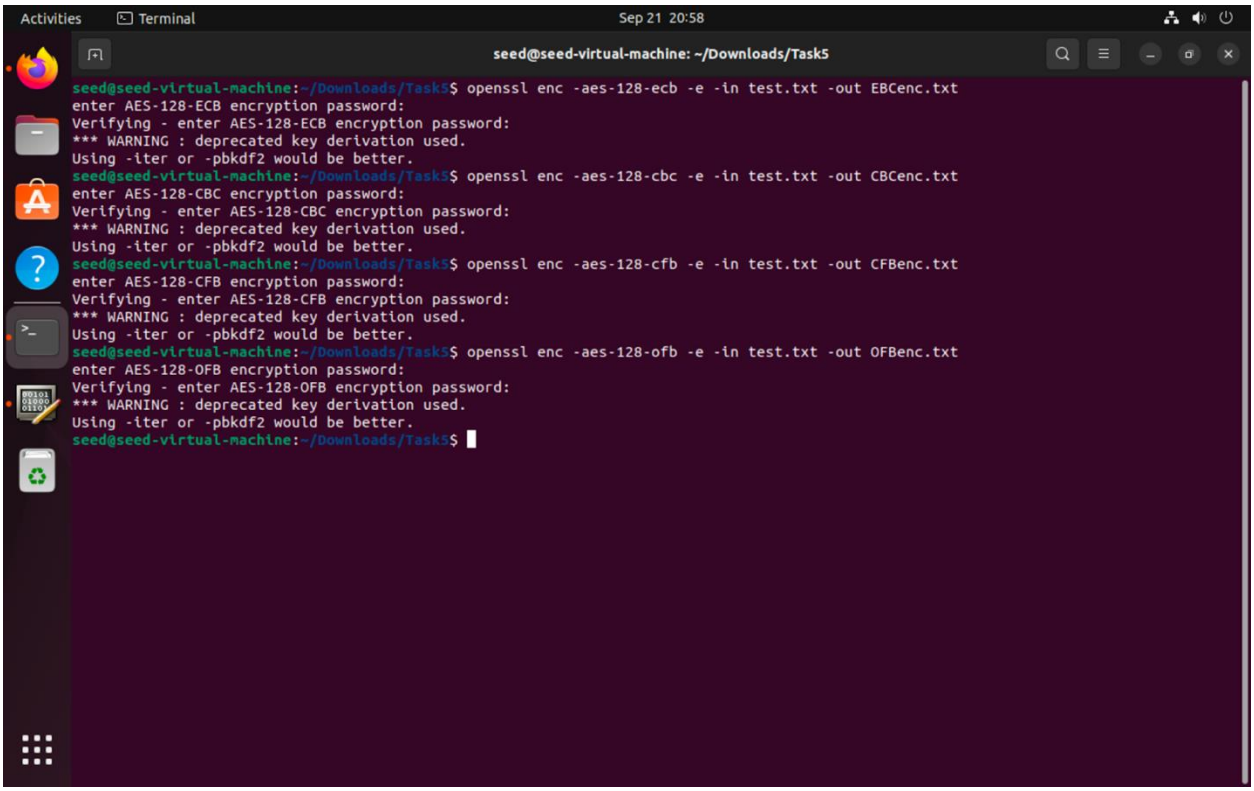
- **CFB:** In CFB mode generally if one block gets corrupted that affects several other blocks. Since they are dependent on the shift register . So, a corrupted byte in one block might affect subsequent blocks. In CFB Due to the fact that each block's decryption depends on the ciphertext from the previous stage and the shift register, if one byte in the encrypted block is

incorrect, it may potentially cause mistakes to spread to following blocks.The first block of plaintext will be impacted by any errors in the first block of ciphertext. Furthermore, because the damaged ciphertext block updates the shift register, the keystream used for next blocks will likewise be tainted. This implies that unless the shift register is emptied and the proper keystream creation starts, mistakes may spread over succeeding blocks.

- **OFB:** Like ECB Every Block except the corrupted block can be recovered .ie the block with the 55th byte is not recoverable. But in OFB it is generally resilient to errors changing one byte only effects that specific byte in a block.
  If a byte in the encrypted file changes in OFB (Output Feedback) mode, a different byte will in fact show up in the same spot in the decrypted plaintext block. This is so that faults in the encrypted data only impact the corresponding bits in the decrypted data for that block as OFB mode creates a keystream separately for each block. In contrast to certain other modes, such as CBC, OFB mode does not result in a cascade of mistakes that spread to consecutive blocks when a ciphertext byte is changed. The remaining plaintext is unchanged, and just the block where the modification took place is altered.

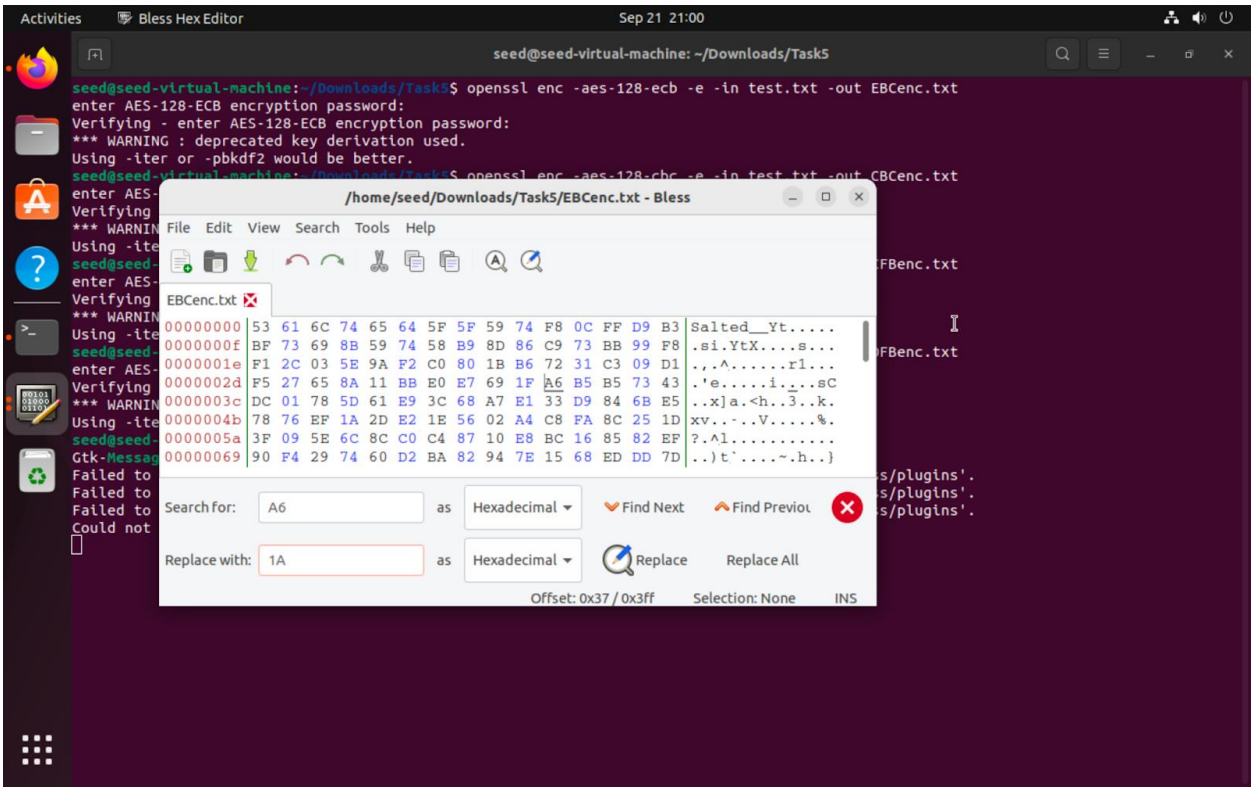
Made a file called text.txt of size 1000 bytes

Encrypting the files in different modes using Password: 2537167

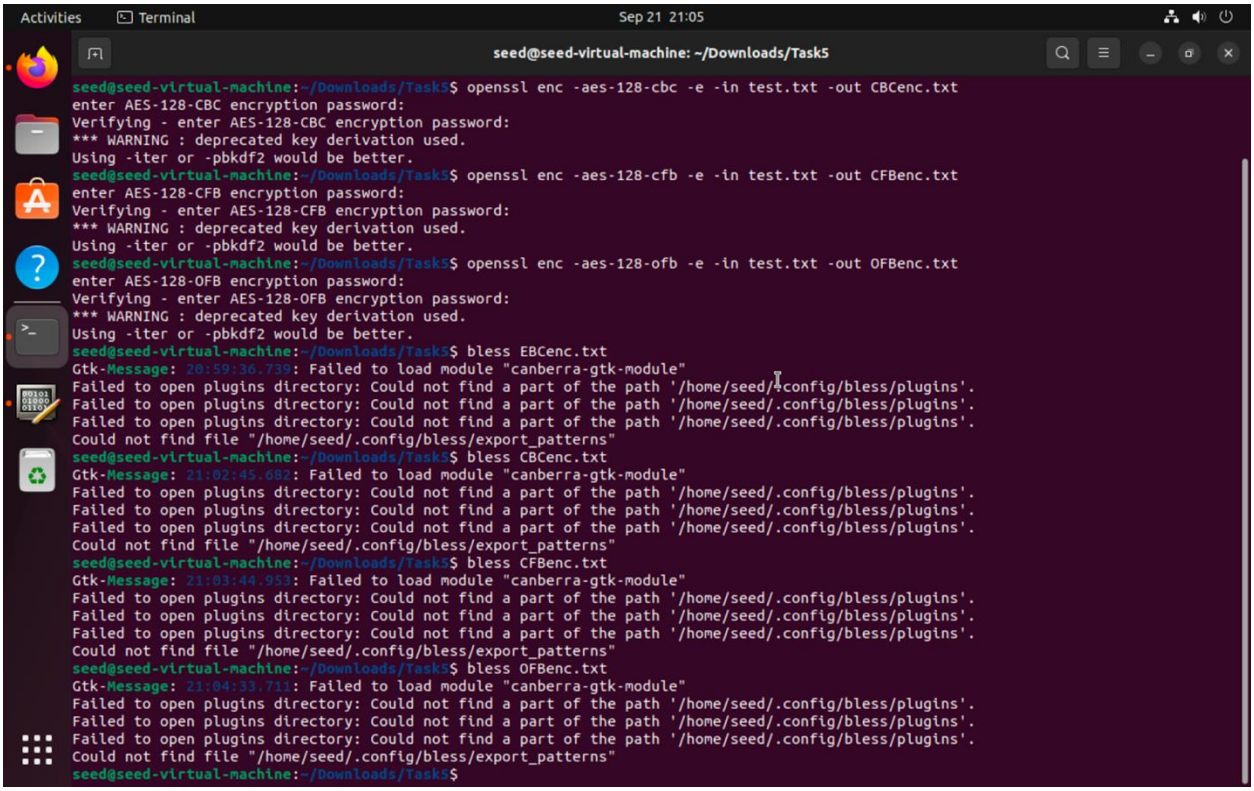


Using Bless to change the 55 bytes in the encrypted text. Here 55th Byte is 37 in Hexadecimal so looking at the 37 positions in Bless.

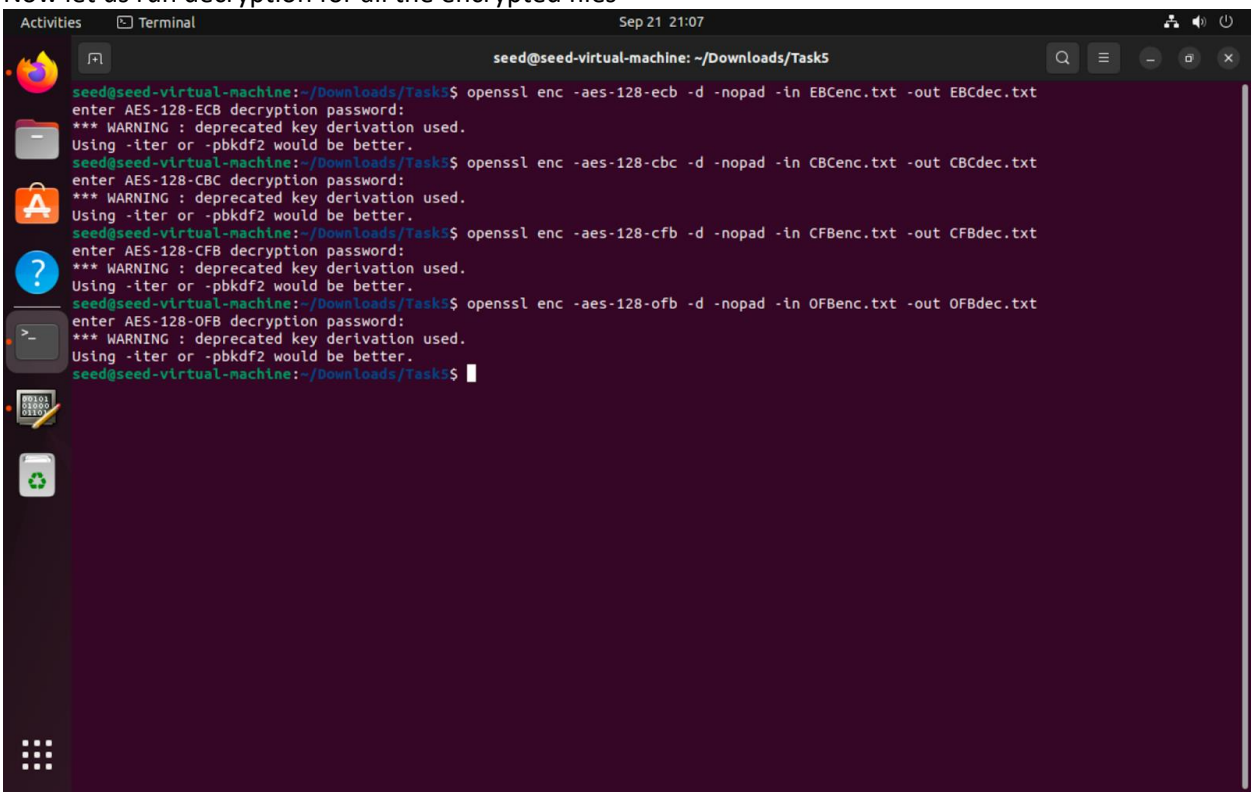
Here the 55 byte is A6. So, I will corrupt it and replace it with a different character 1A.



Similarly Doing it for all the modes.
All the enc files are corrupted with their 55 byte corrupted to 1A.

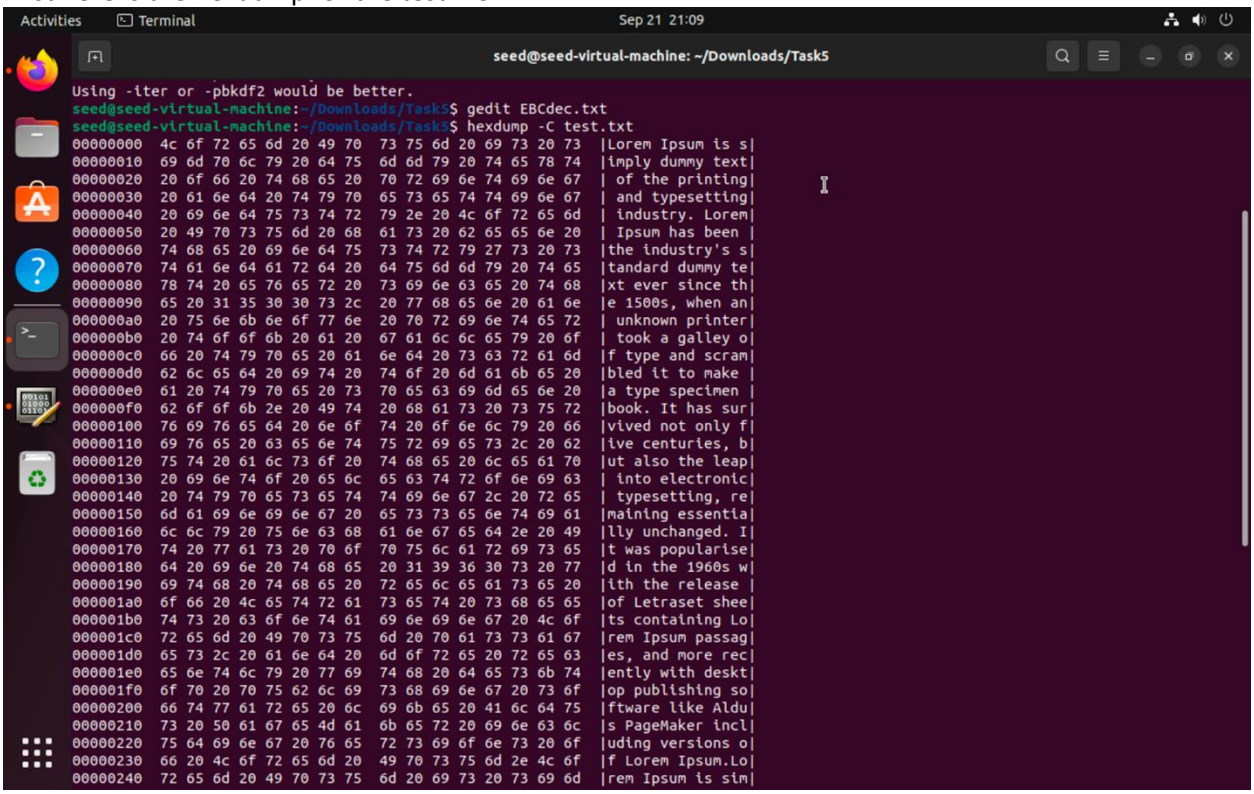


Now let us run decryption for all the encrypted files

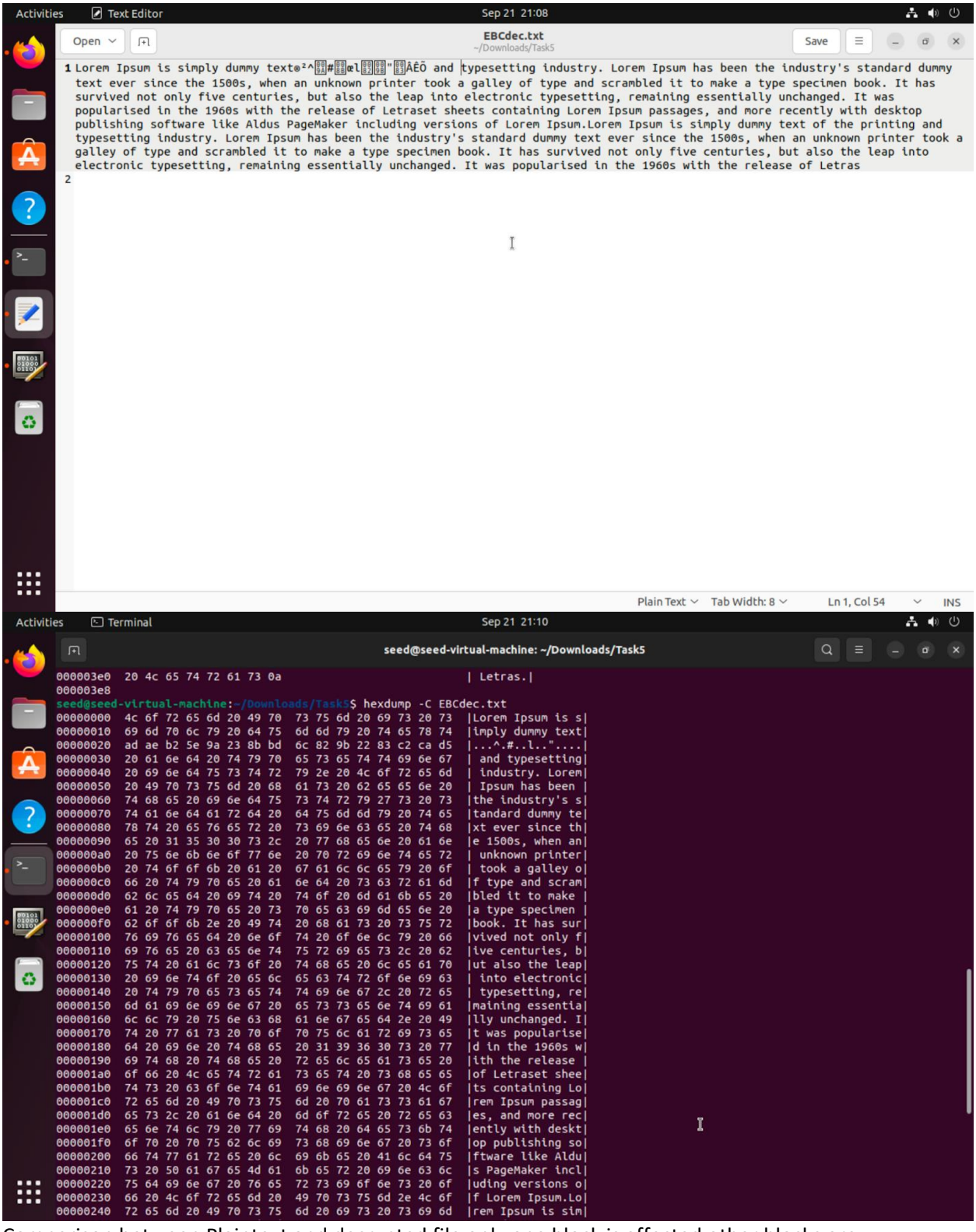


Now let us check the differences in the file text.txt and the decrypted text in different modes

First here is the hexdump for the test file

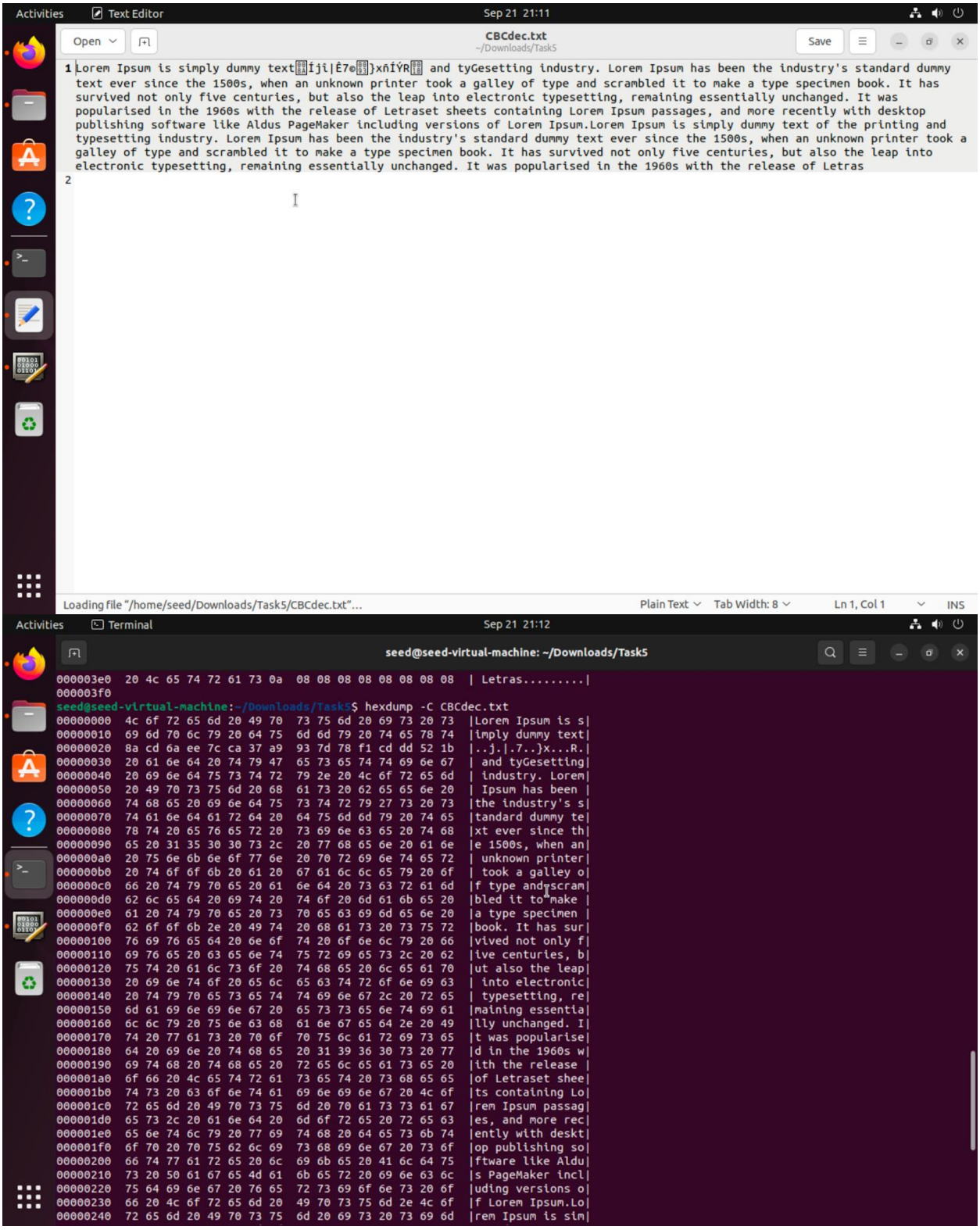


1. ECB:



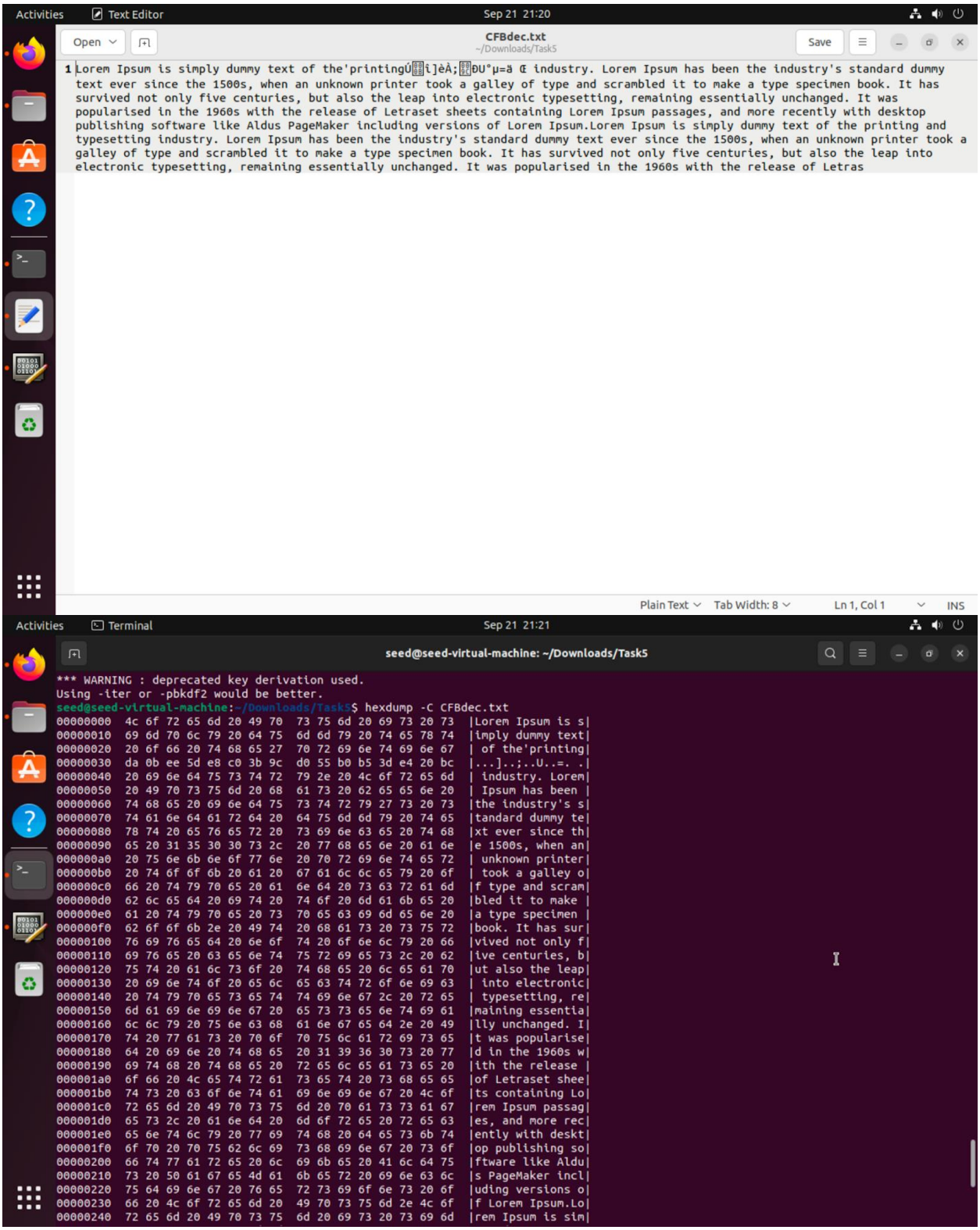
Comparison between Plaintext and decrypted file only one block is affected other blocks are same.

2. CBC :



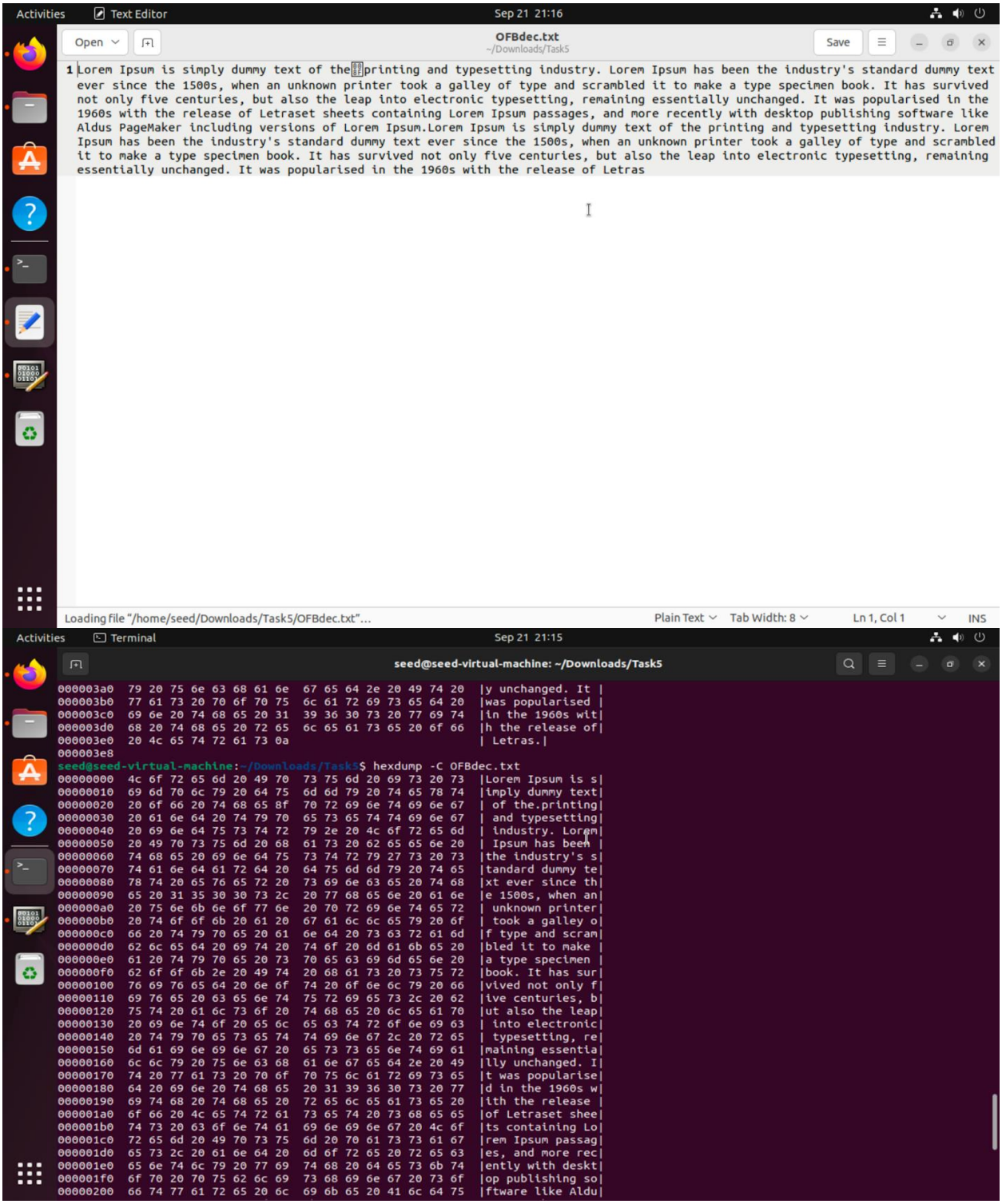
Comparison between Plaintext and decrypted file 2 blocks is affected other blocks are same.

3. CFB :



Comparison between Plaintext and decrypted file two blocks is affected other blocks are same.

4. OFB :



Comparison between Plaintext and decrypted file only one block is affected other blocks are same.(specifically 1 bit)

Summary:

A different byte will appear in the same location in the plaintext block if a byte in the encrypted file changes in OFB mode, as opposed to the plaintext, where the rest of the plaintext is unaffected.

In EBC mode in contrast to the plaintext, if a byte in the encrypted file is changed, the resulting block of plaintext is entirely different, while the remainder of the plaintext is unaffected.

In CBC mode, a single byte change in the ciphertext of a block results in a corrupted block of plaintext at that position and another corrupted block in the next position, while the rest of the plaintext is unaffected. Data integrity and authenticity checks are crucial in cryptographic systems utilizing the CBC mode because of this error propagation feature.

In conclusion, unless the shift register is synced once again, flaws in one block of ciphertext might actually spread to succeeding blocks when using CFB mode. Since any inaccuracy in the ciphertext might possibly cause a chain reaction of faults in the decrypted data, data integrity is essential in CFB mode.