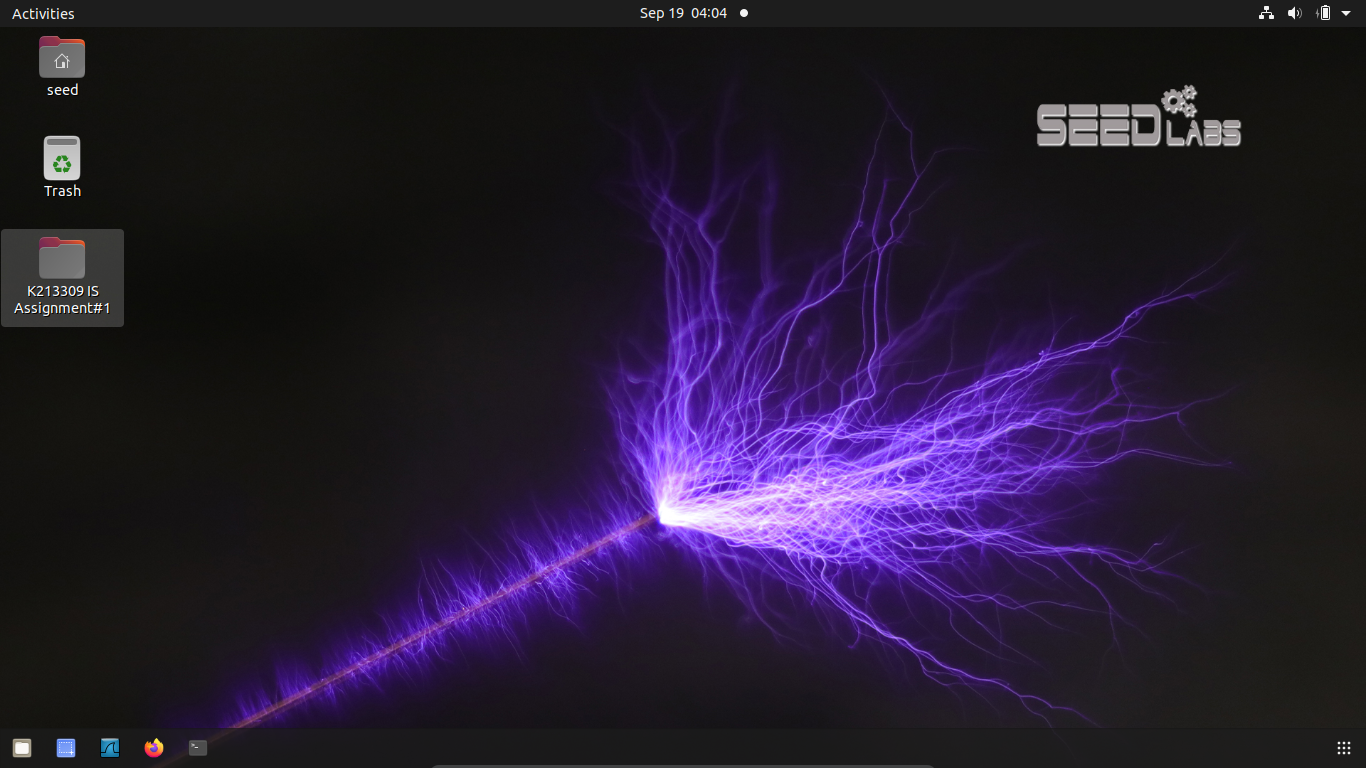
# Setup

Since we need a dedicated VM for this lab, we first need to download the zip file provided by the SEED project which is a pre-built SEED Ubuntu 20.04 VirtualBox image. You can find this file [here](https://seedsecuritylabs.org/labsetup.html). After that, you can follow the steps [here](https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md) to properly setup your VM on virtual box. Once that is done, you will have a custom ubuntu for this lab.



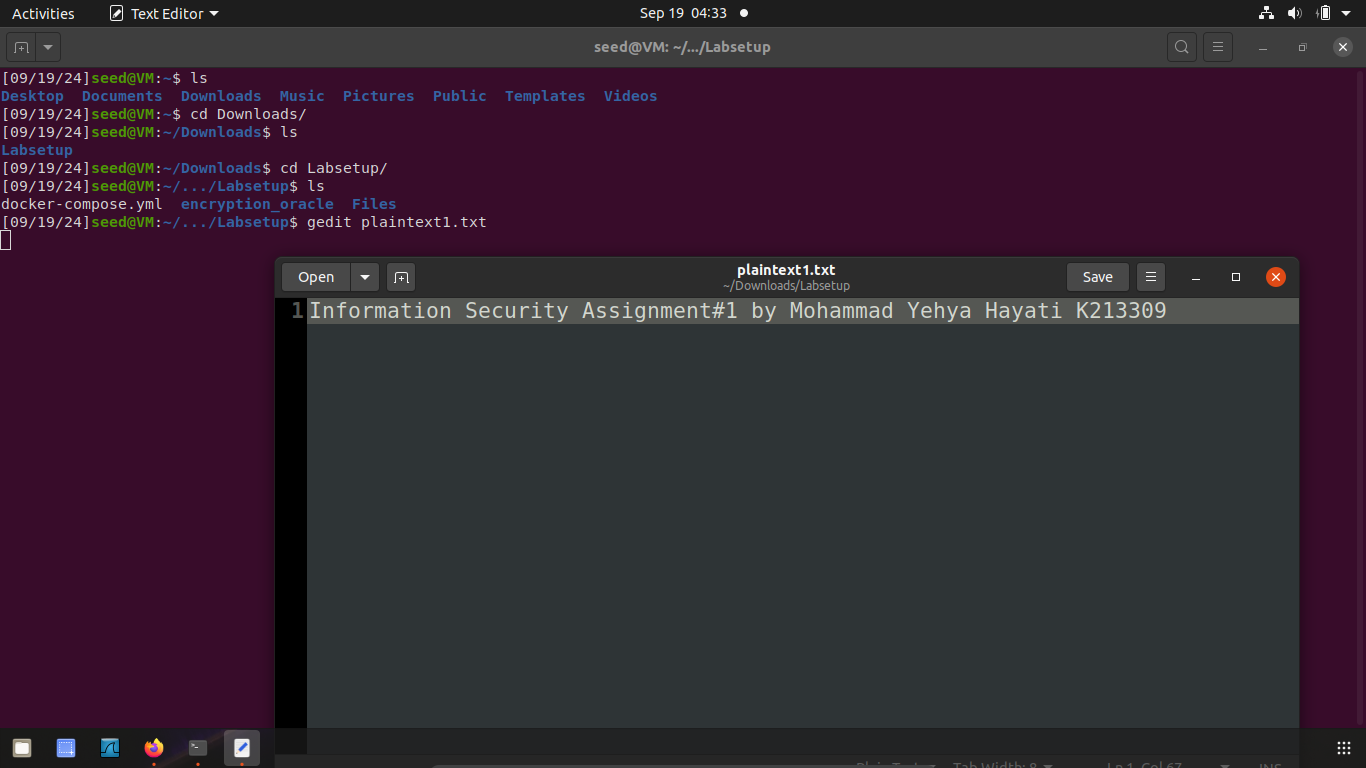


Next, we need to download the lab setup zip file to setup the environment for the lab and unzip it. That is the setup done.

# Task #1

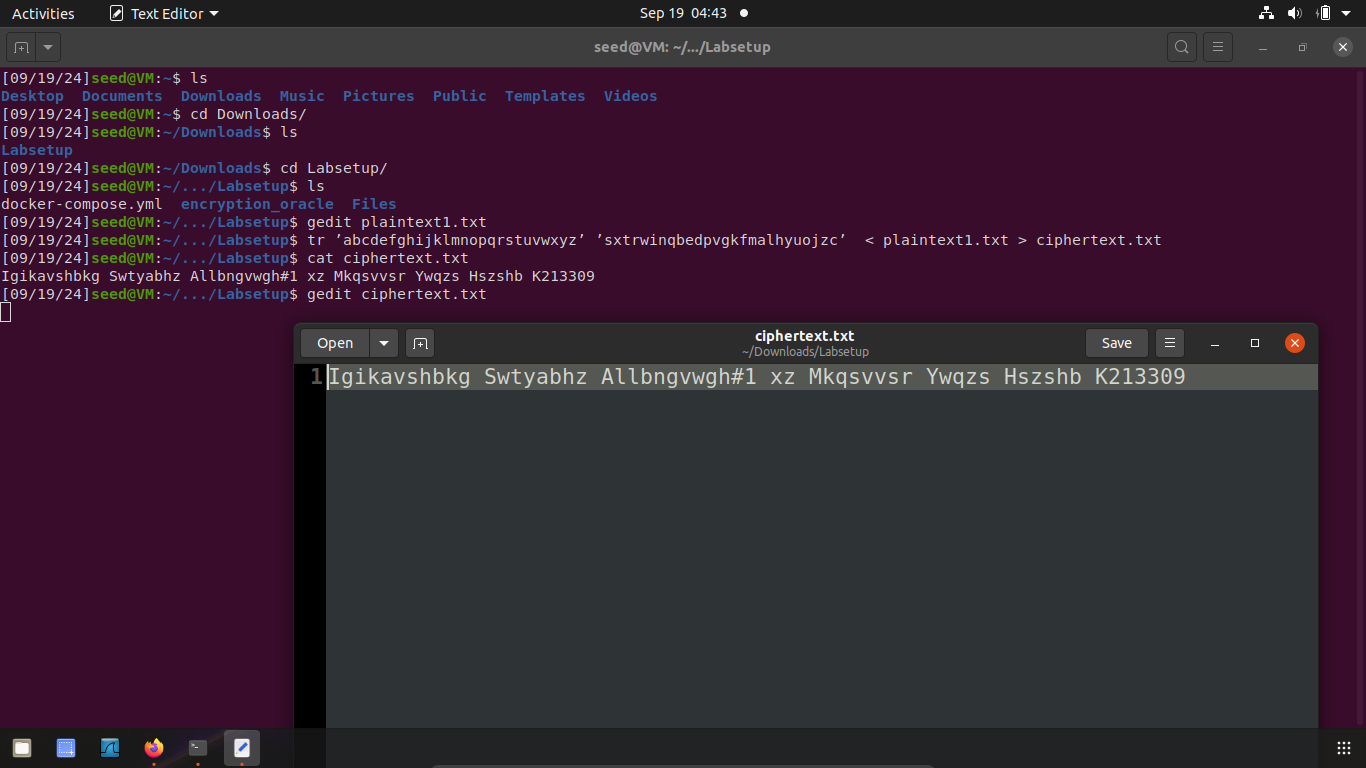
The first task is quite simple as all the commands and guidelines are given in the assignment document. Therefore we will follow the given commands.

First and foremost, we will create a plain text document.

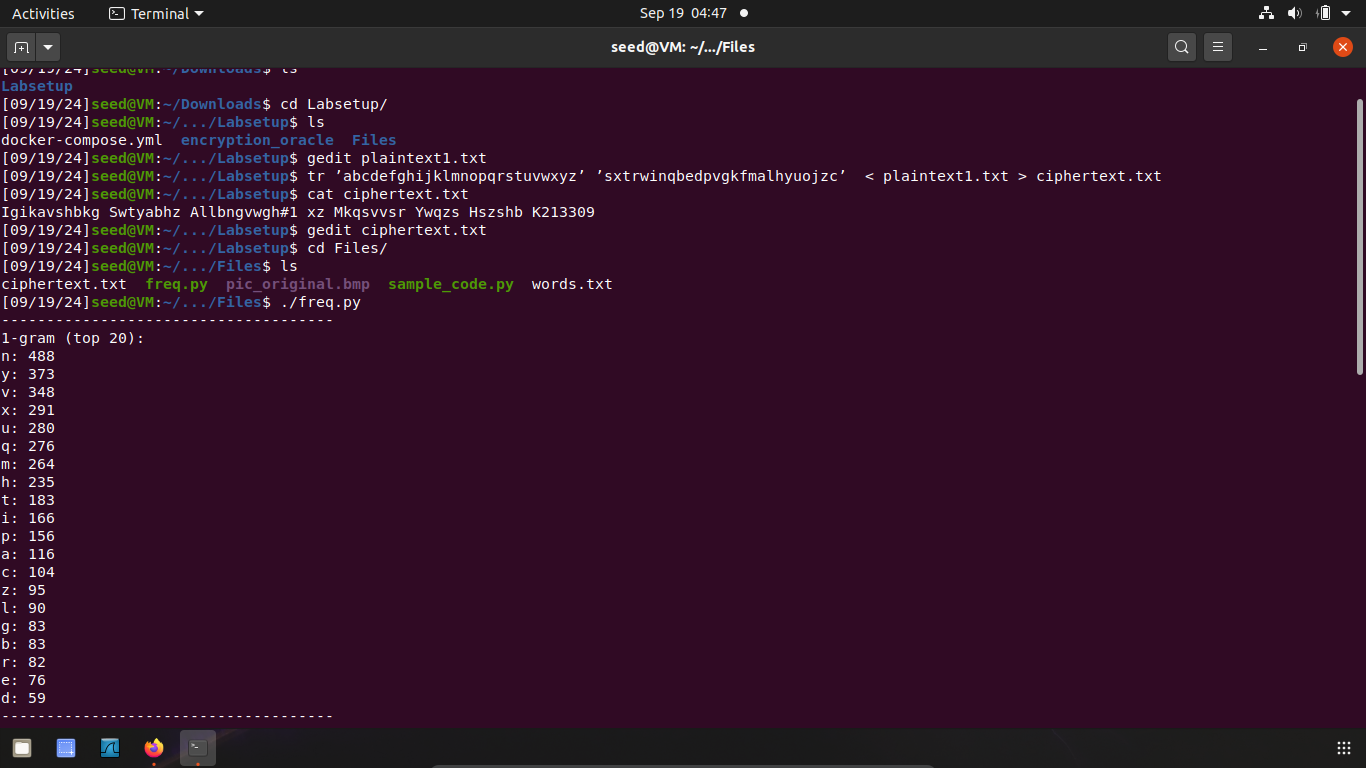


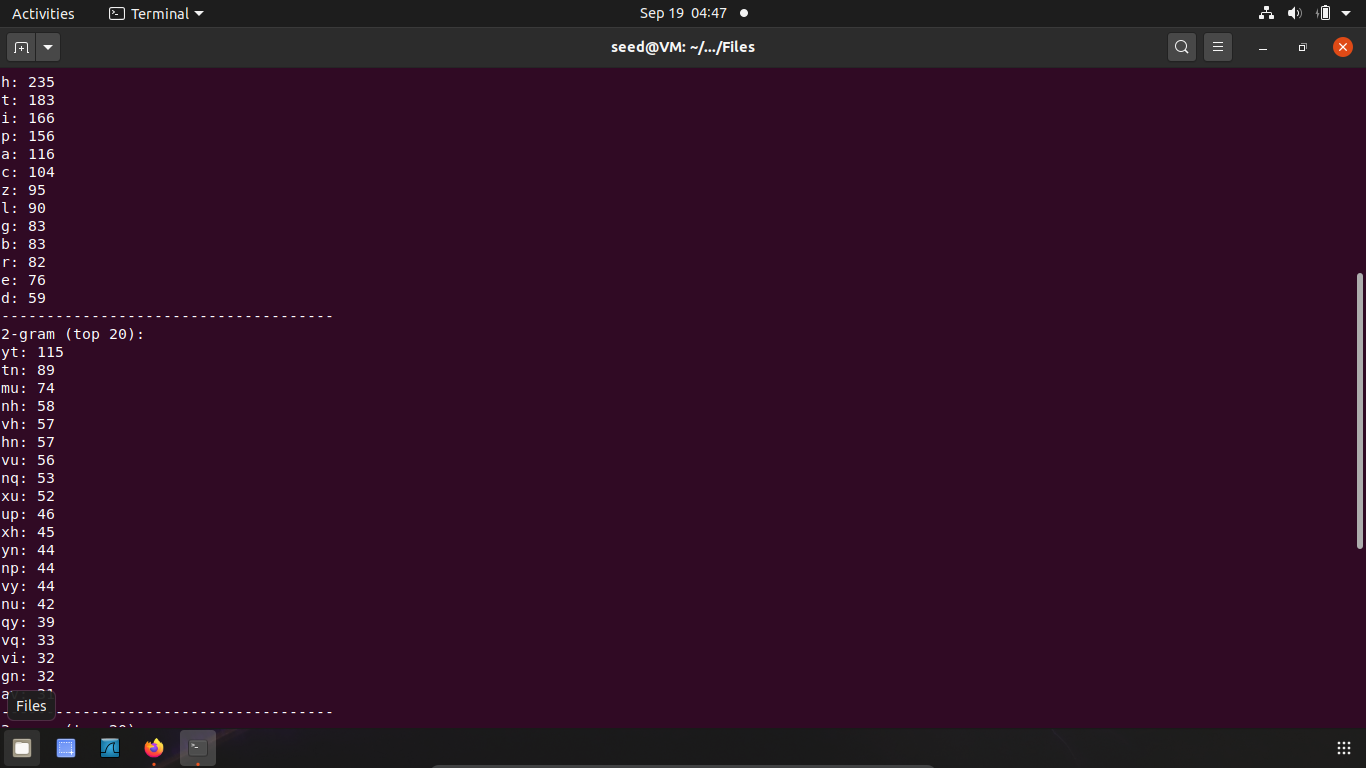
Then, as given in the document, we will run the following command:

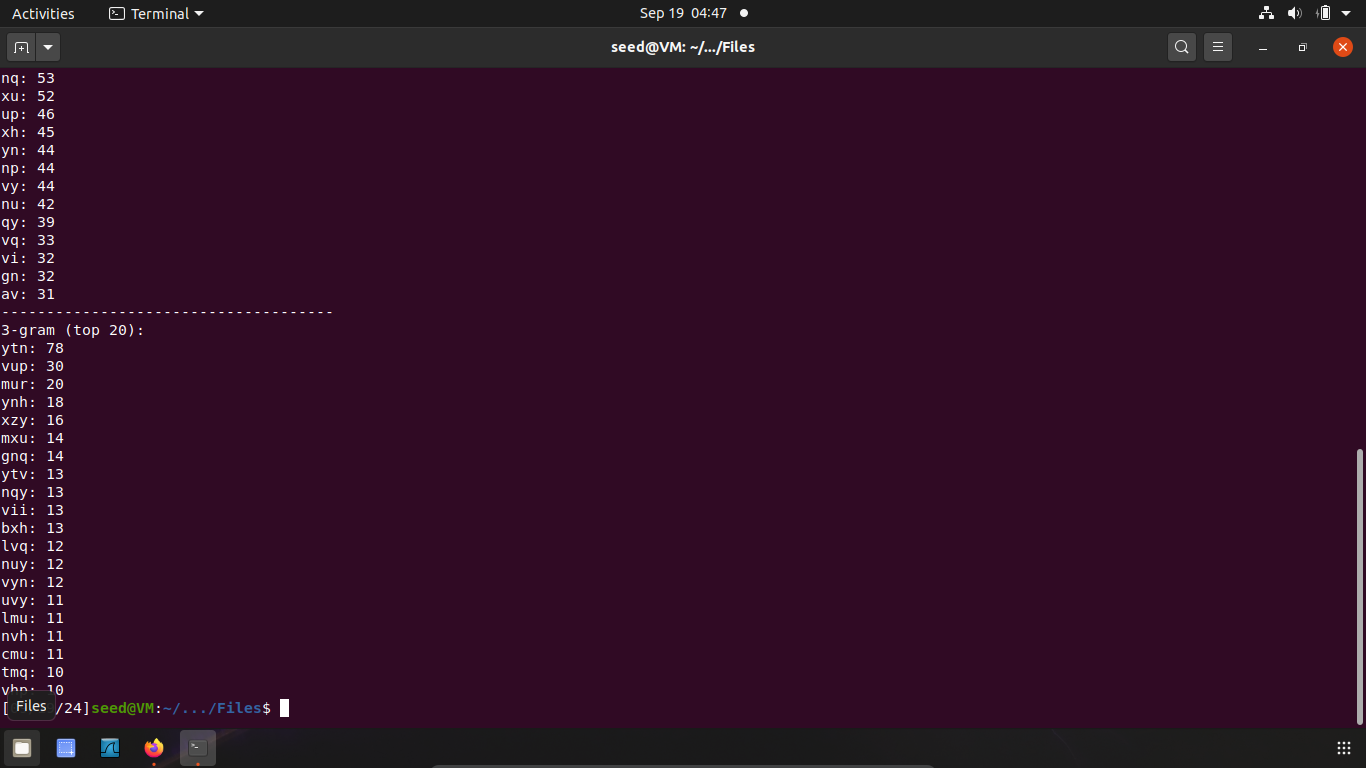
$ tr ’abcdefghijklmnopqrstuvwxyz’ ’sxtrwinqbedpvgkfmalhyuojzc’ < plaintext.txt > ciphertext.txt



Next, we will use the freq.py file, which was given in the lap setup files, to generate the frequency of the words in the plain text. The frequency will be given in the form of all possible n-grams which is basically a ‘n-numbered’ word.







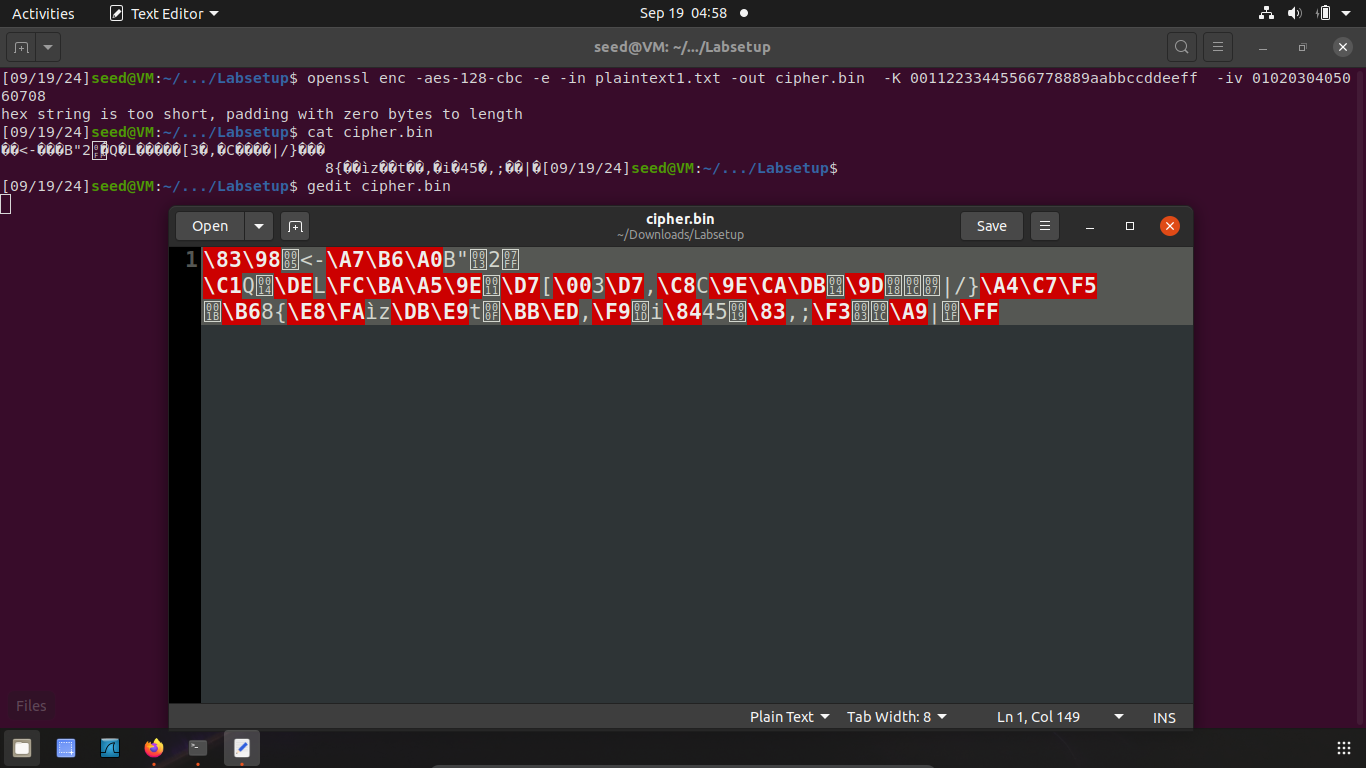
# Task #2

For this task, we will again be using the resources in the lap setup file. This time, we will specifically be using the different encryption algorithms. The basic format of the command is:

$ openssl enc -ciphertype -e -in plain.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

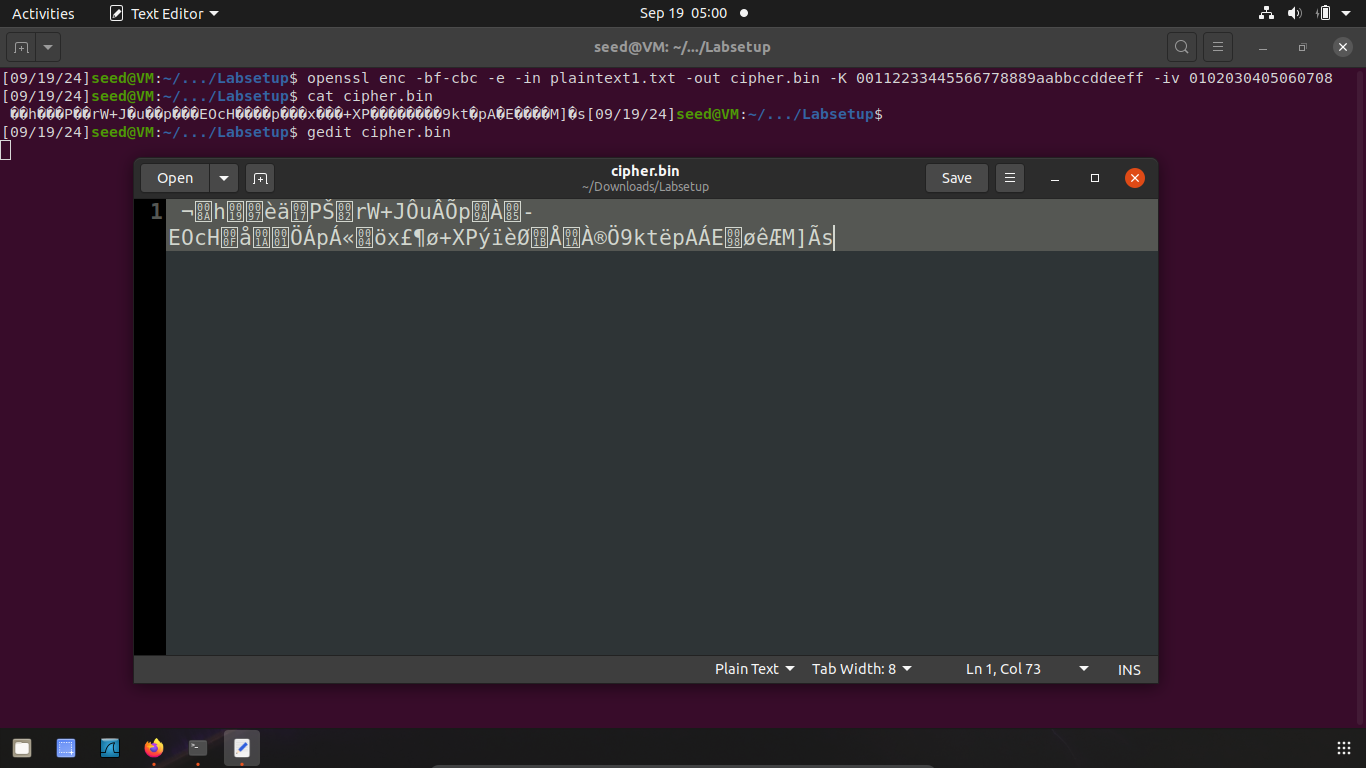
The first algorithm to try is -aes-128-cbc. So, the command with be:

$ openssl enc -aes-128-cbc -e -in plaintext1.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708



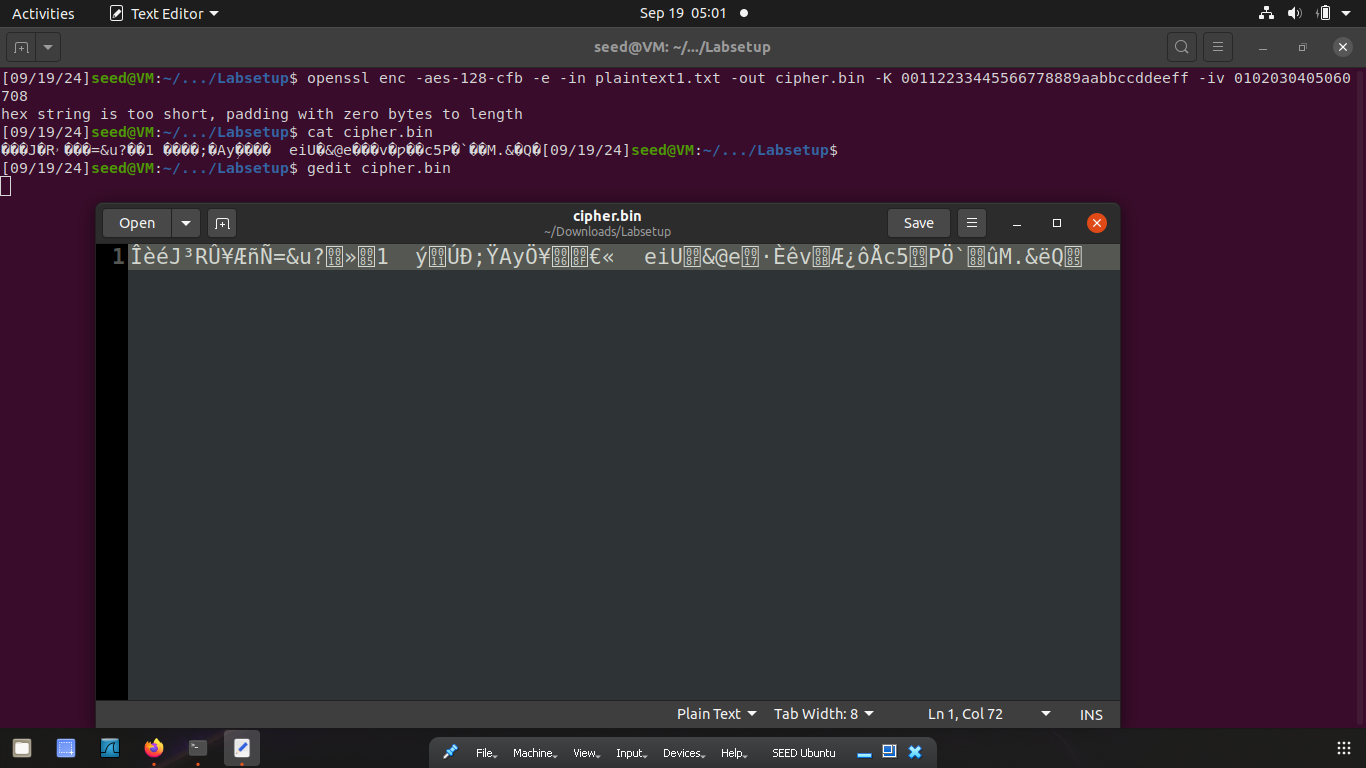
Next we will try -bf-cbc. So the command will be:

$ openssl enc -bf-cbc -e -in plaintext1.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708



Lastly, we will be using -aes-128-cfb. So the command will be:

$ openssl enc -aes-128-cfb -e -in plaintext1.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708



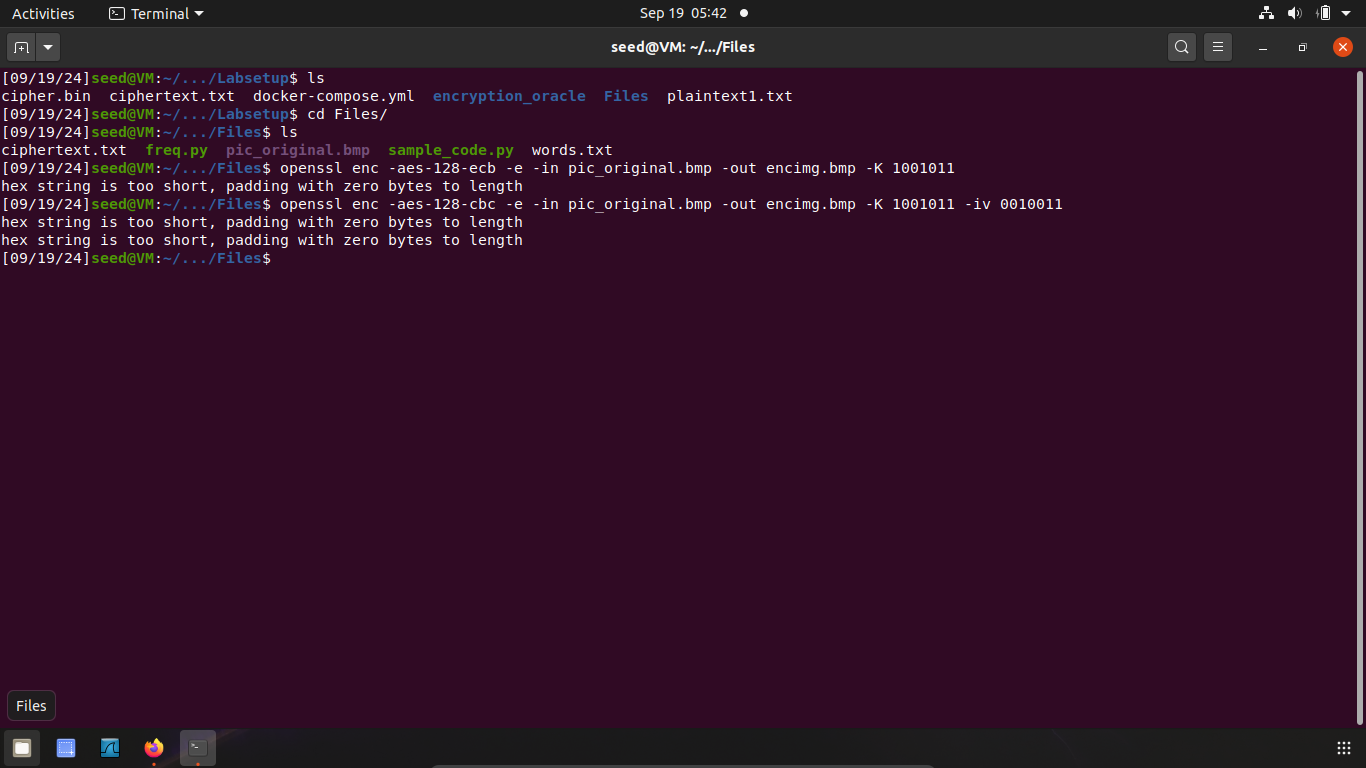
# Task #3

This task is similar to the previous, the only difference being that we will now be encrypting an image.

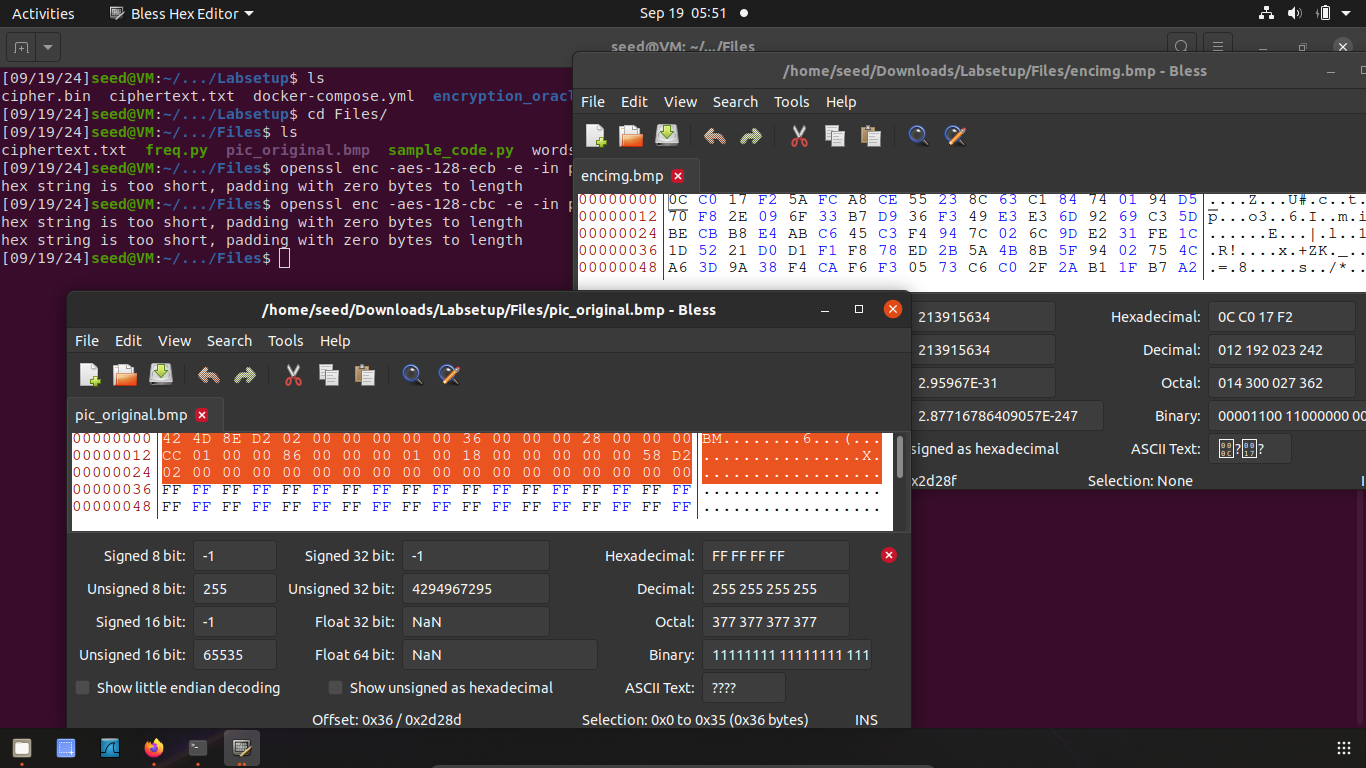
Firstly, we have to encrypt the image using ECB and CBC encryption modes. The command for this is:

$ openssl enc -aes-128-ecb -e -in pic\_original.bmp -out encimg.bmp -K 1001011 -iv 0010011

$ openssl enc -aes-128-cbc -e -in pic\_original.bmp -out encimg.bmp -K 1001011 -iv 0010011



Now to retain the property of a bitmap (.bmp) image, we need the original 54 bytes of the image in the encrypted to actually view the encrypted image. For this we will use ‘bless’ (as mentioned in the assignment document) to edit the first 54 bytes of the encrypted image.

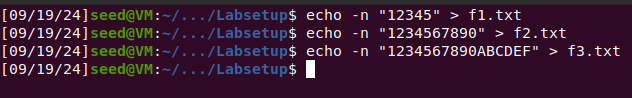


Now we can see the difference between the original image and the encrypted image.

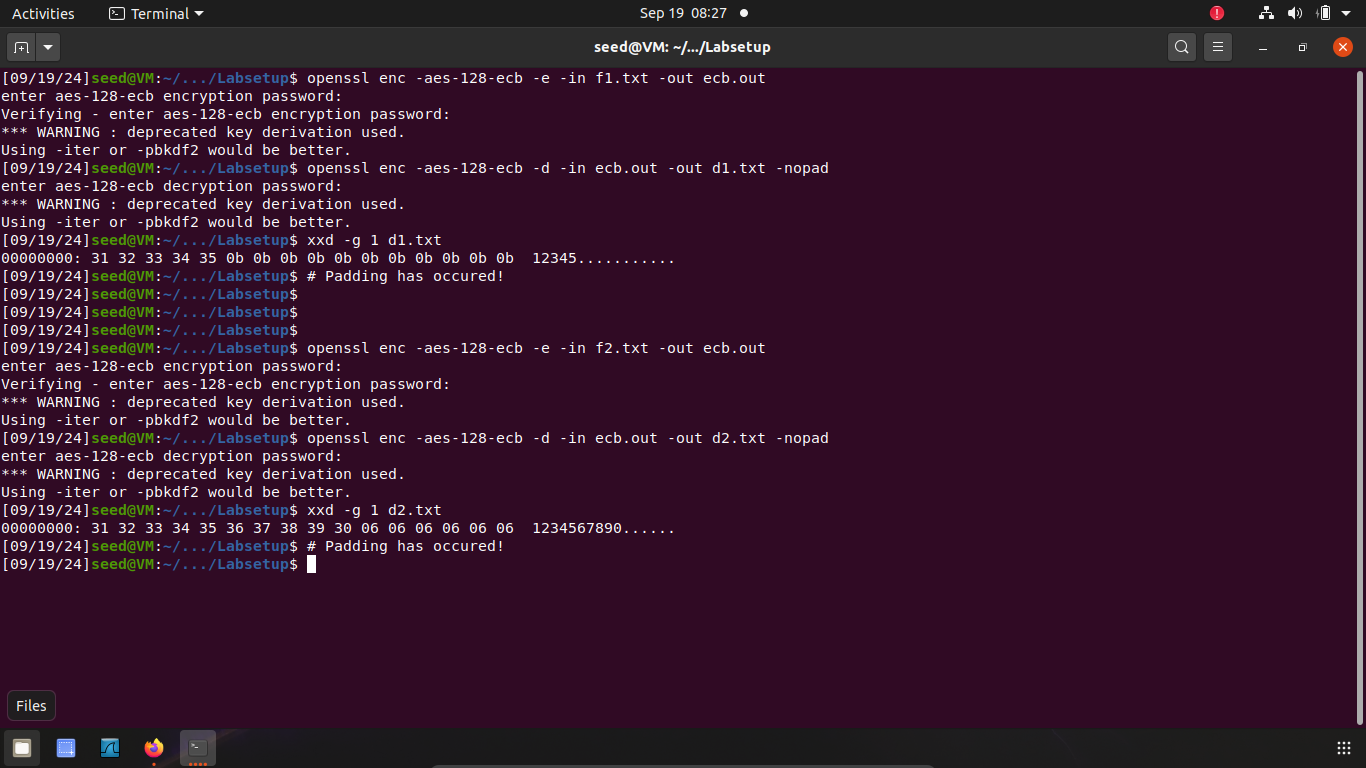
# Task #4

This task explores how different encryption algorithms use the concept of padding.

Firstly, we will generate the three files as mentioned in the document. Then, we will encrypt the files using the four given techniques: ECB, CBC, CFB, and OFB. After using each technique, we will check whether they use padding or not.

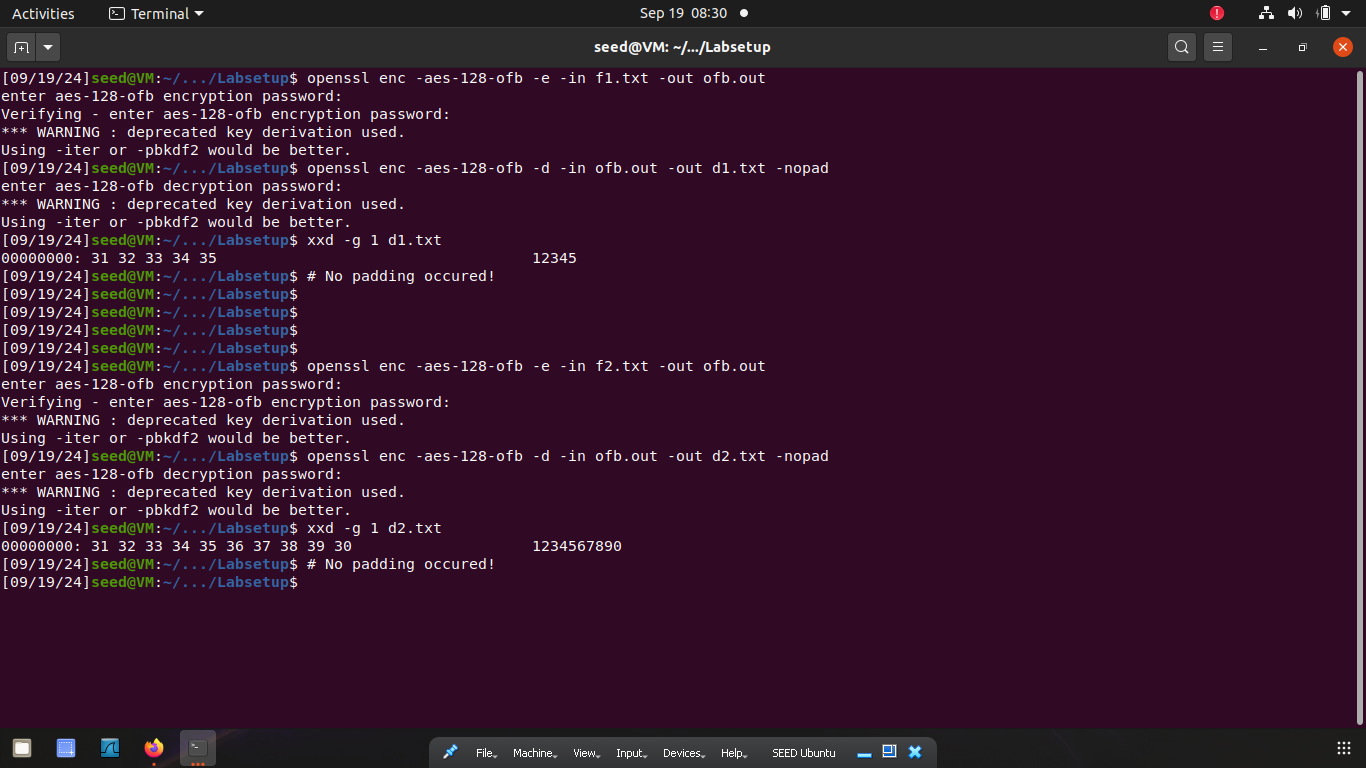


Here are the commands for ECB:



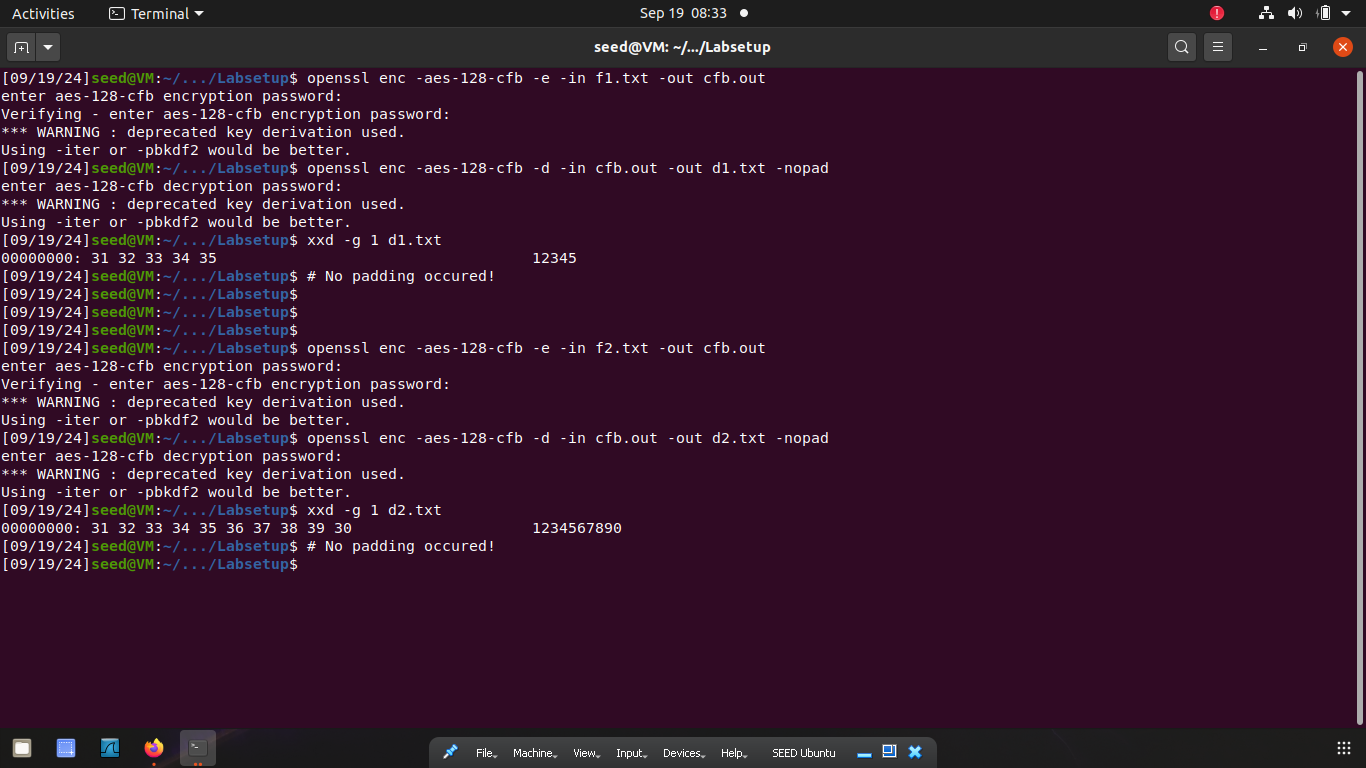
As you can see, ECB uses padding

Here are the commands for OFB:



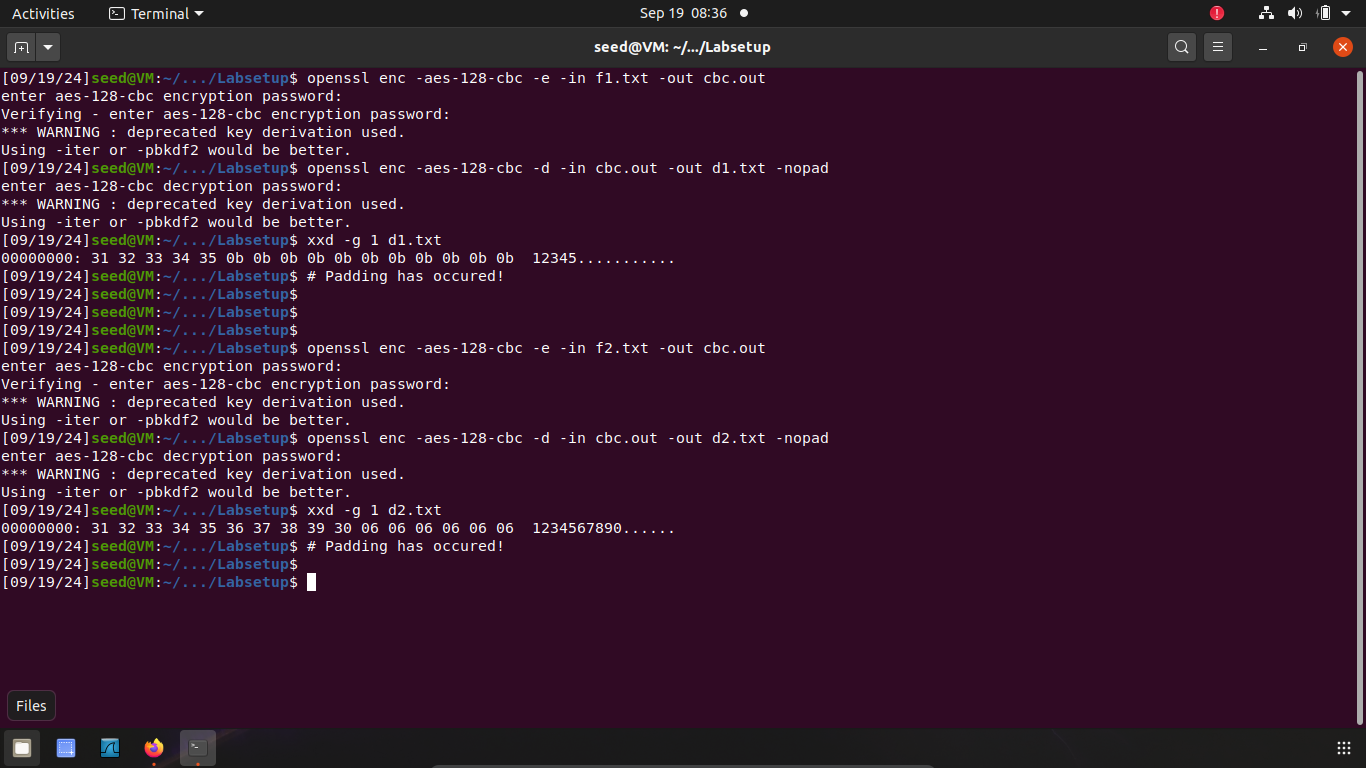
As you can see OFB does not use padding.

Here are the commands for CFB



As you can see CFB does not use padding.

Here are the commands for CBC

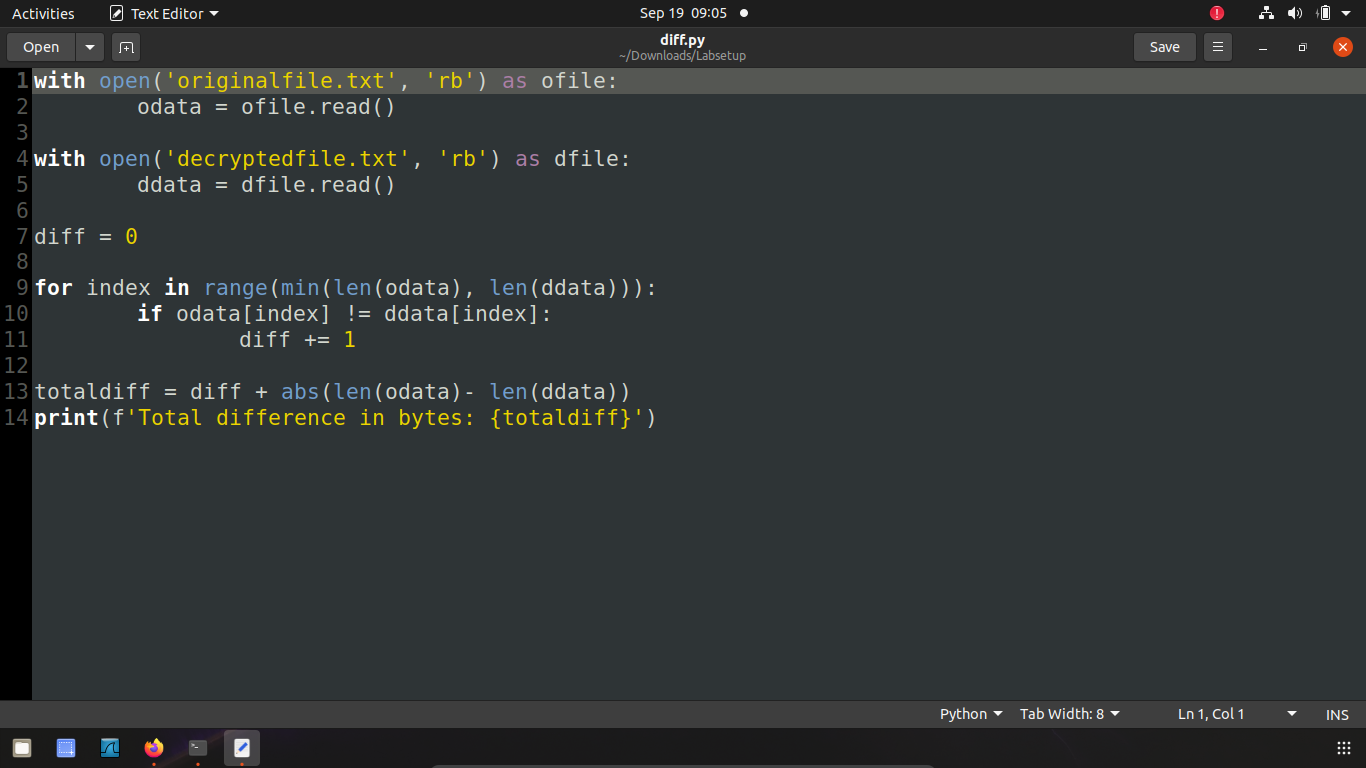


As you can see CBC uses padding

# Task #5

The aim of this task is to see the difference between original and decrypted texts if the encrypted text in between was corrupted. Each mode will behave differently since they use different techniques. OFB will only lose 1 byte as the keystream is generated independently of the cipher text. CFB will lose the equivalent of two chunks which depends on the block size, typically 16-32 bytes. ECB will lose 16 bytes because blocks are encrypted independently. CBC will lose 32 bytes due to the chaining of blocks.

Firstly, we need a function to find the difference in bytes between the two files:

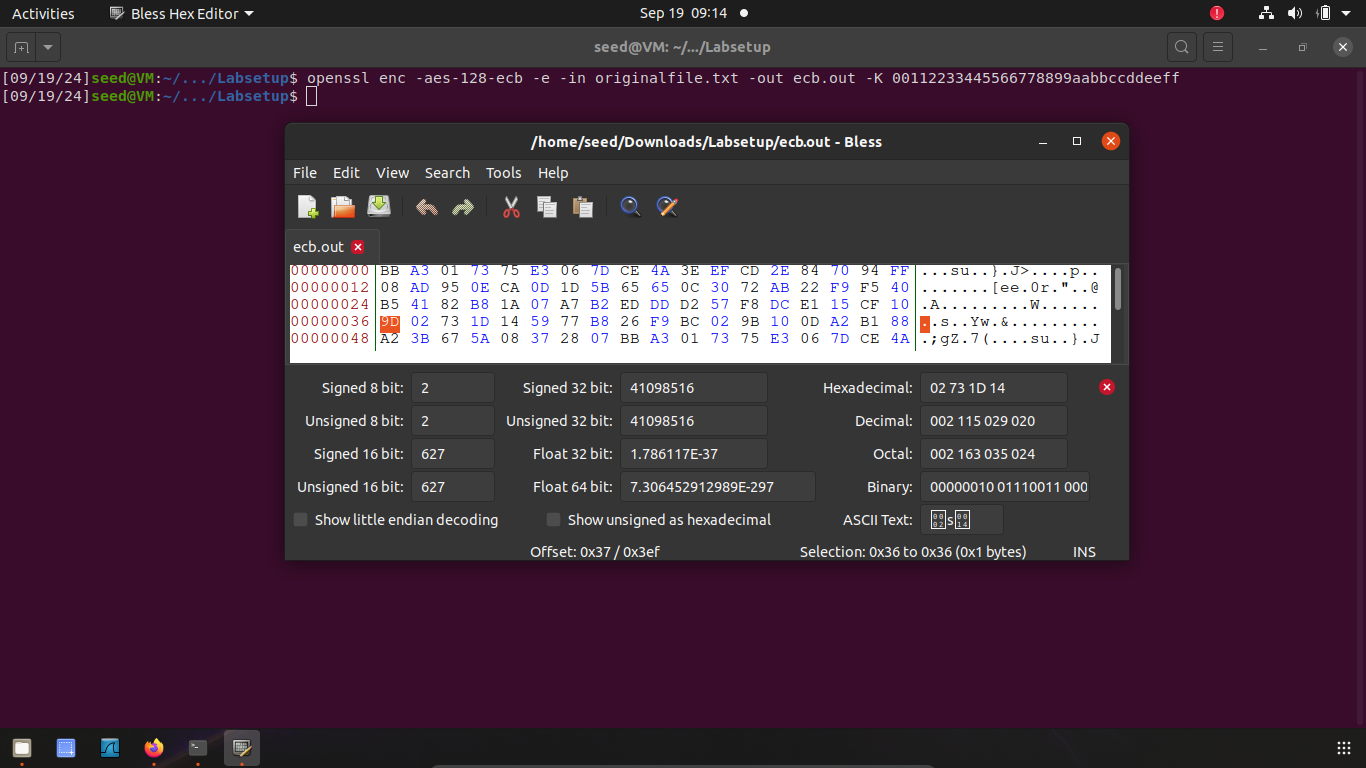


Also, we need a file of 1000 characters, so here is the command:



Now we can perform the experiments. We need to first encrypt a file, and then purposely corrupt it by changing the 55th byte. We then decrypt the corrupted file and see the difference in bytes. Each mode will have a different result due to using different techniques. We will then compare the answers I gave above of each mode will the actual result.

For ECB

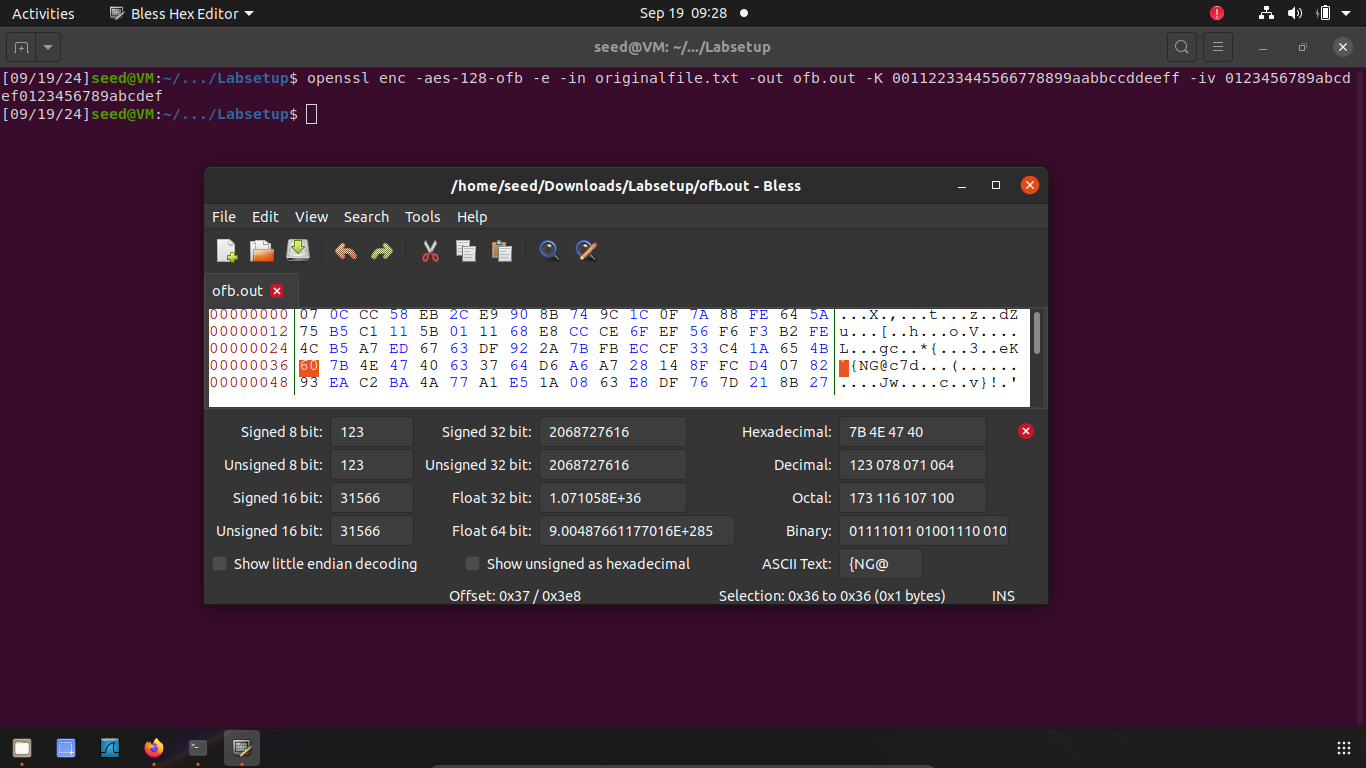


Let’s change to 00, and then decrypt to see the difference in bytes.

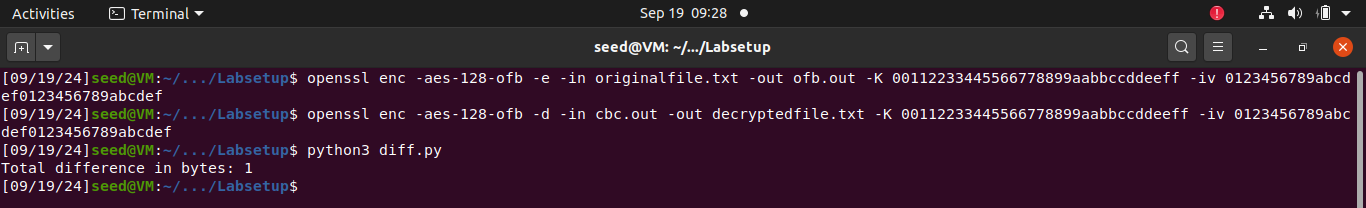


As you can see the difference is 16 bytes for ECB.

For OFB



Let’s change to 00, and then decrypt to see the difference in bytes.

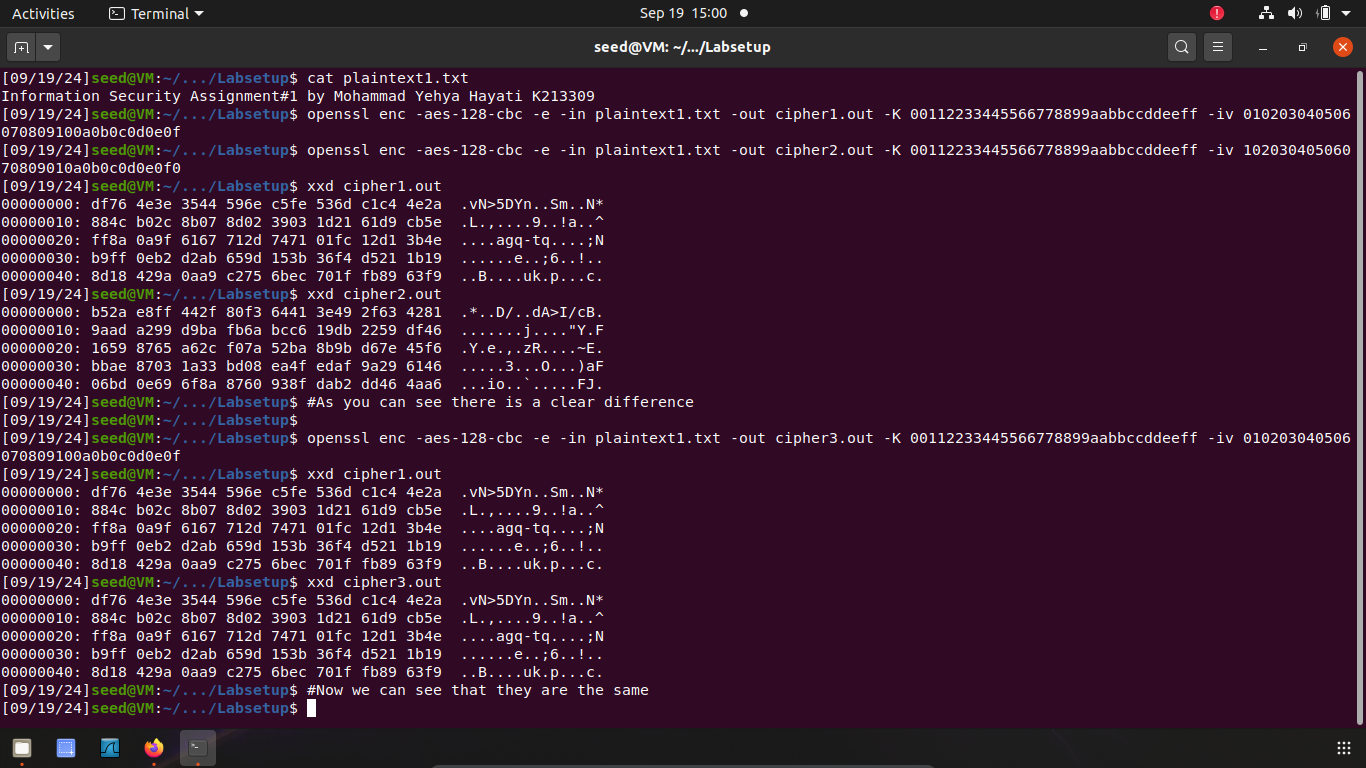


As you can see the difference is 1 byte for OFB.

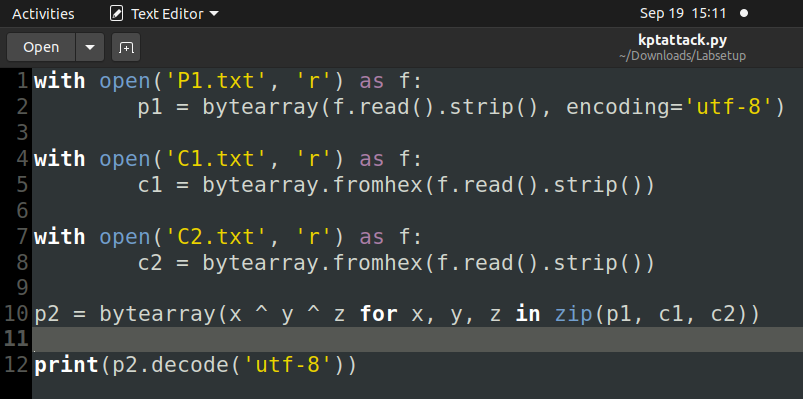
Similar results can be seen for CFB and CBC.

# Task #6

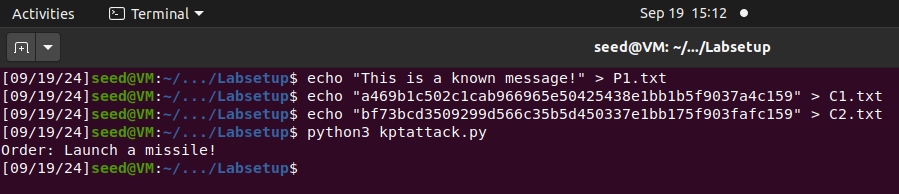
The first part of this task demonstrates the purpose of an IV. We will encrypt a plaintext using the different IV and same IV.



In the second part of this task, we have to perform a *known-plaintext attack.* Using the hint given in the manual about XOR strings, we can perform this easily. First, we need to write a python script to perform the XOR operation.



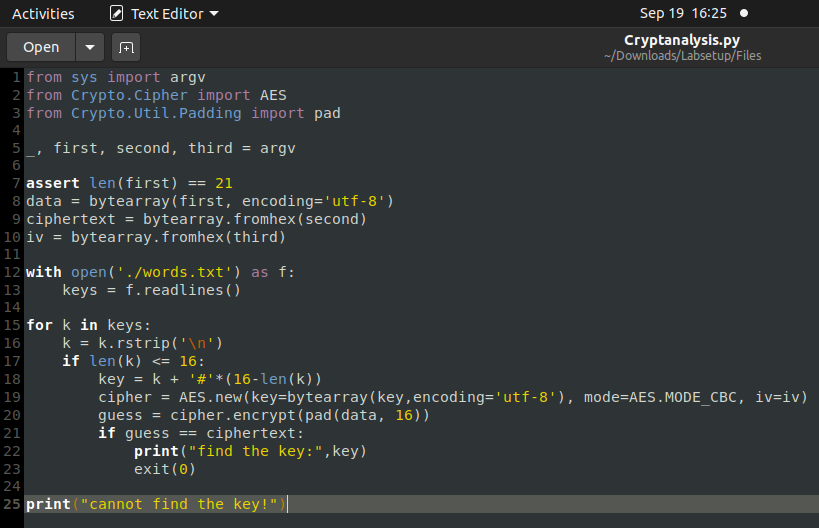
Next, we will use the python script to generate the message P2.



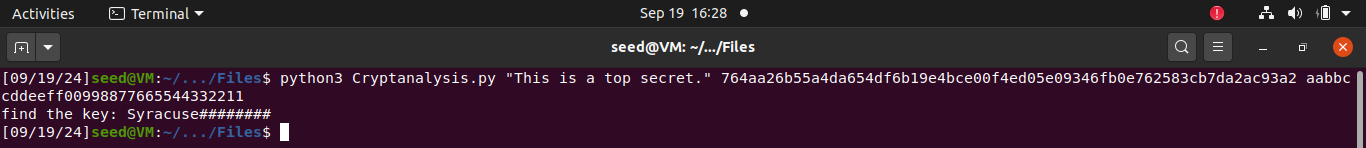
As you can see, we have successfully performed a *known-plaintext attack.*

# Task #7

In this task we need to perform a cryptanalysis to find out the encryption key. Firstly, we need a python script that can perform the analysis. This code is taken as a reference.



Now we run it.



We now have the key.