Tanmay Fadnavis

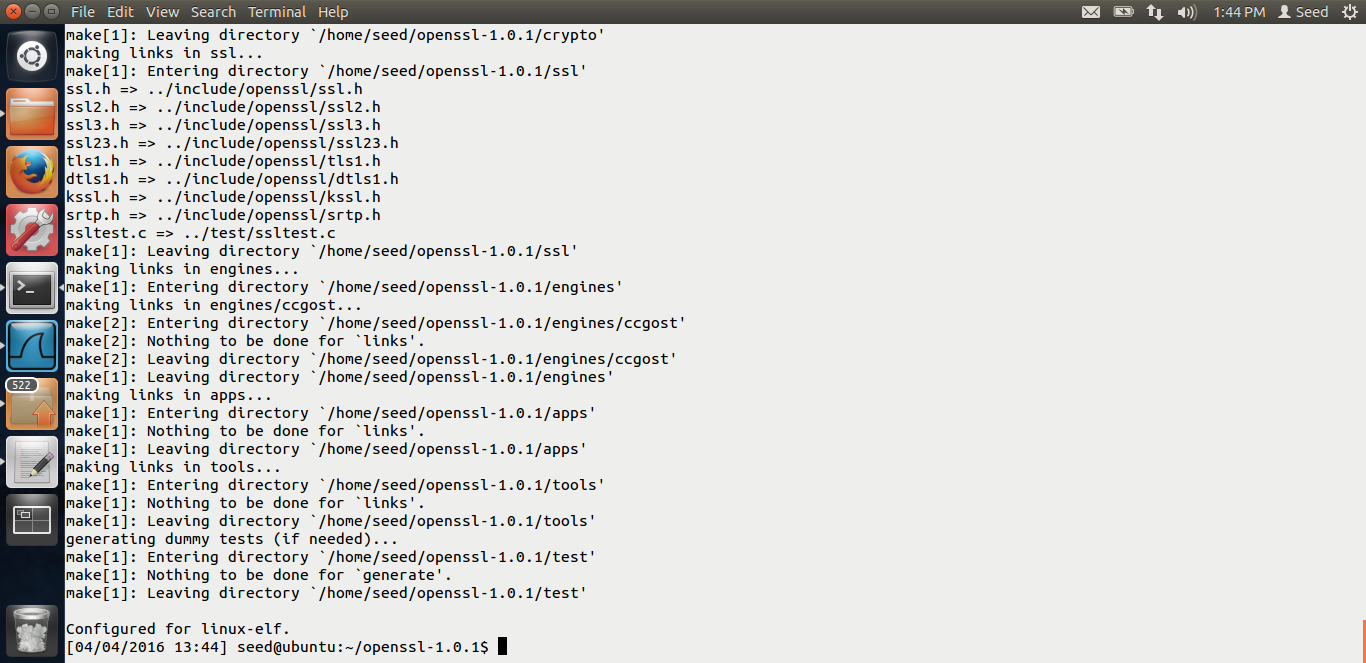
SUID: 971141760

INTERNET Security

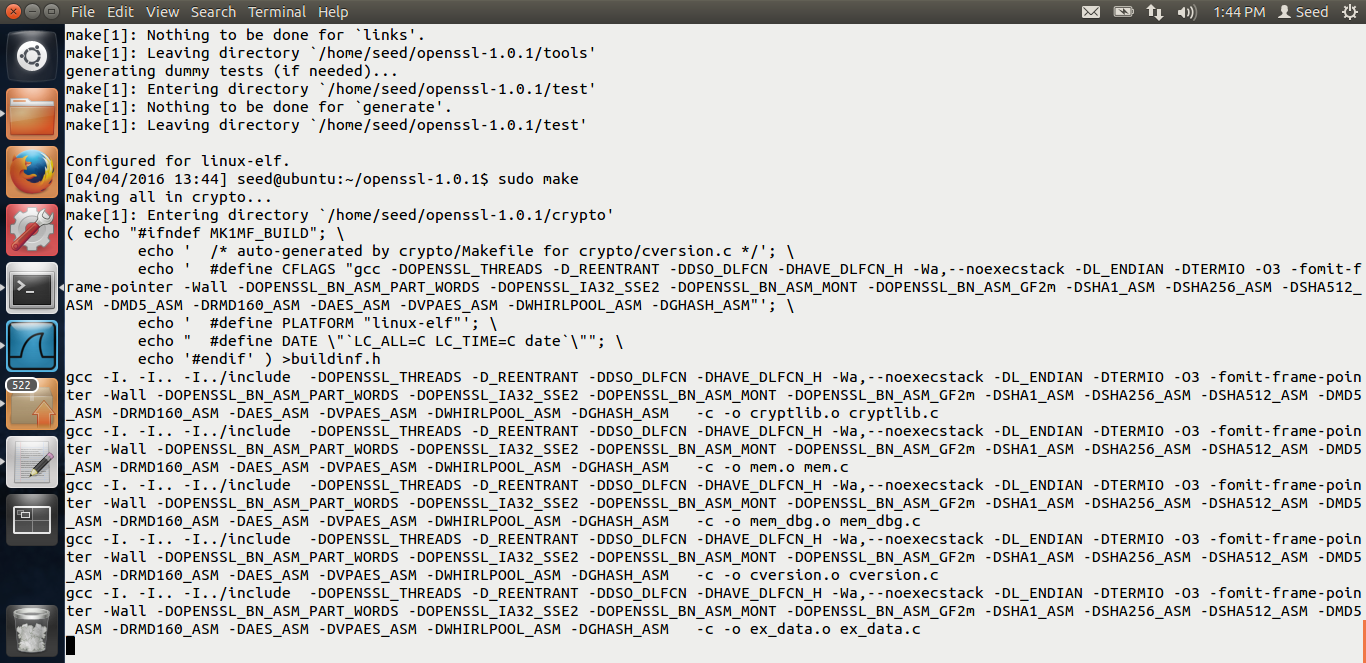
Lab - 7

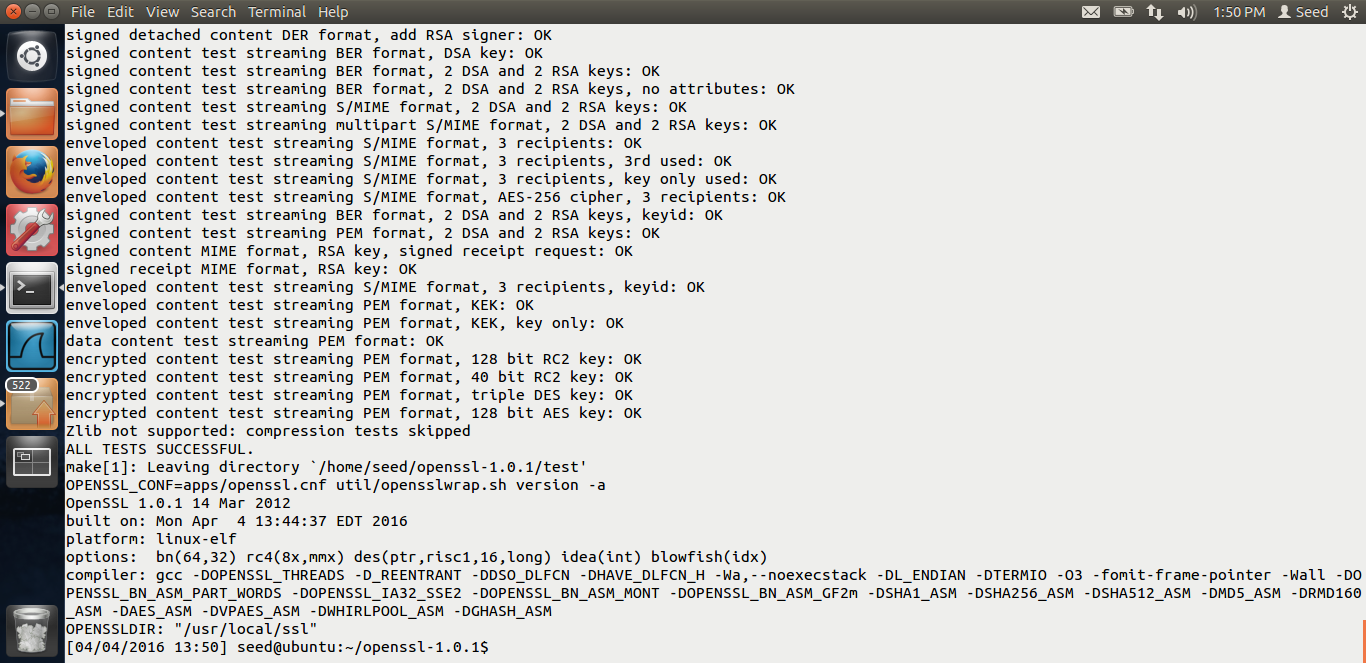
# ENVIRONMENT SETUP

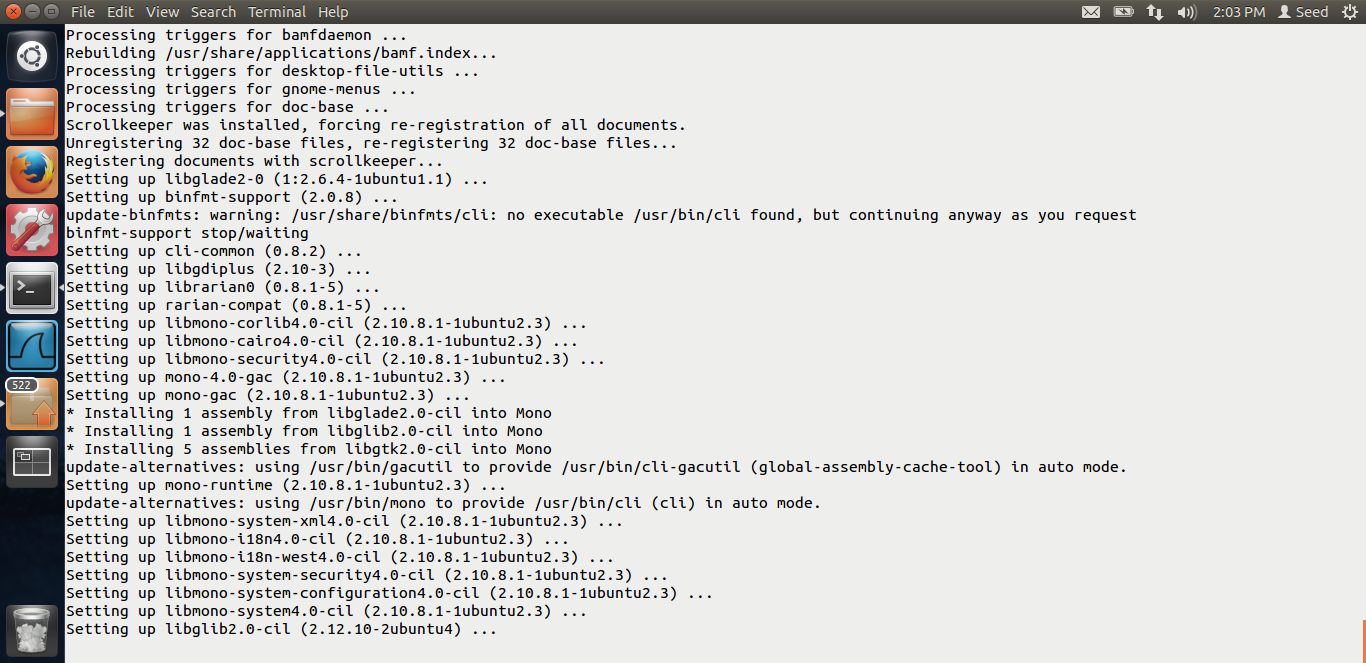
The screen shots for environment set-up are below.



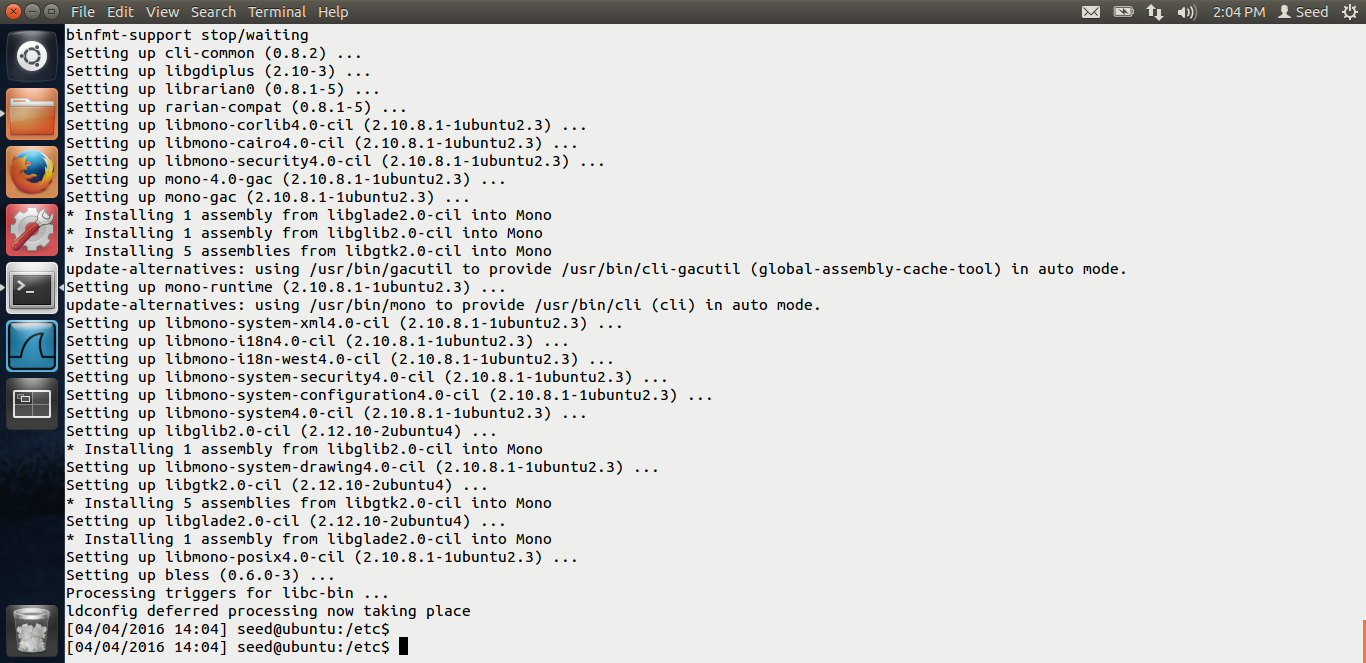
Installing the openssl libraries.







Testing the openssl libraries using the commands given in the lab description.

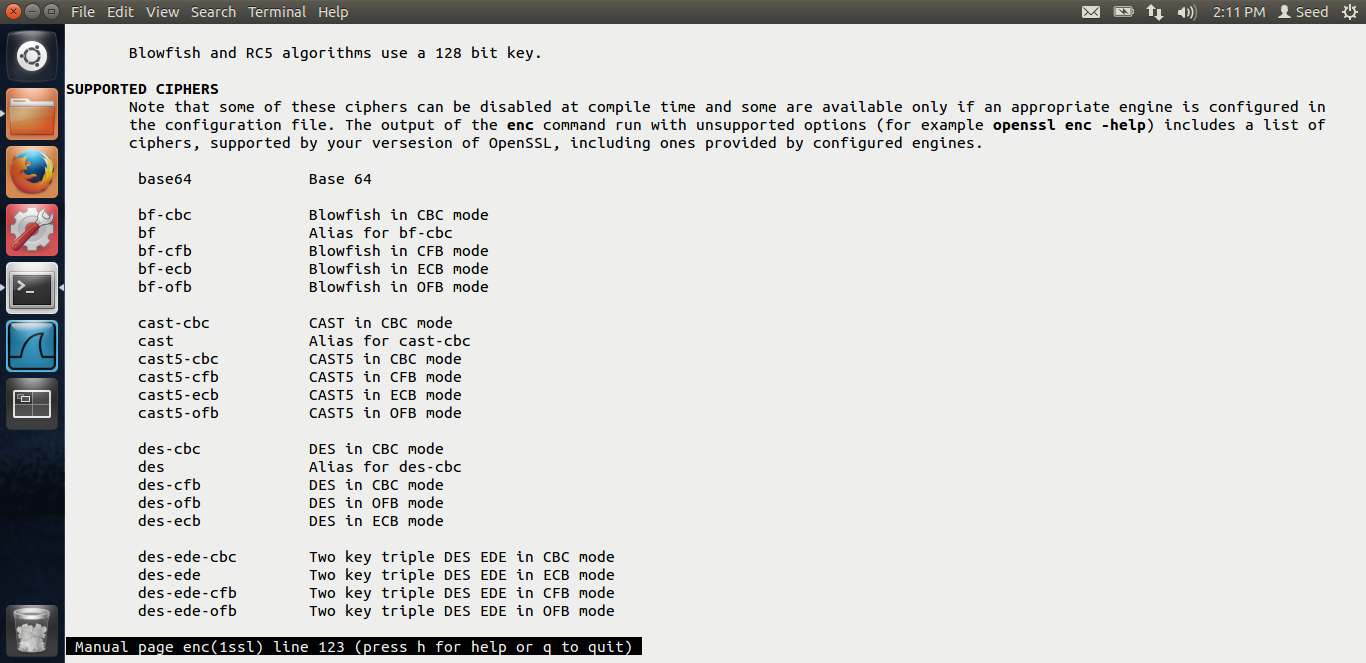


Installing the bless hex editor. The environment setup is done.

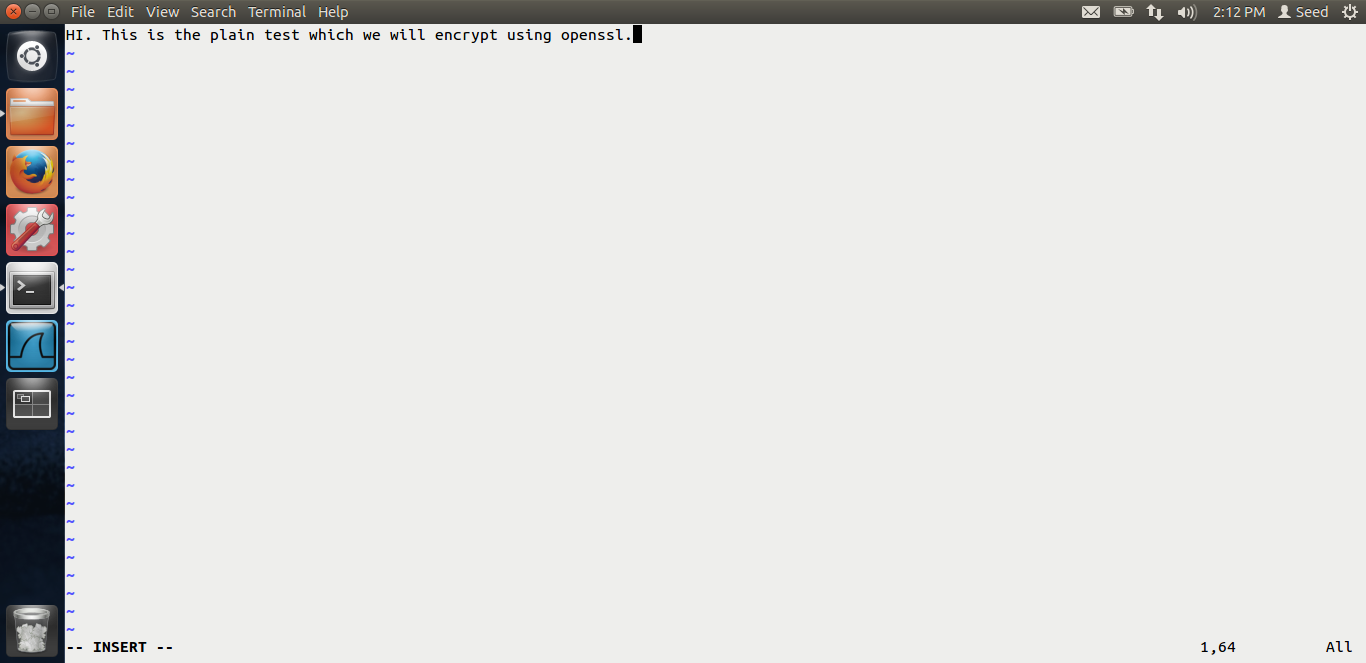
# Task 1:

The screen shots for task-1 are below.

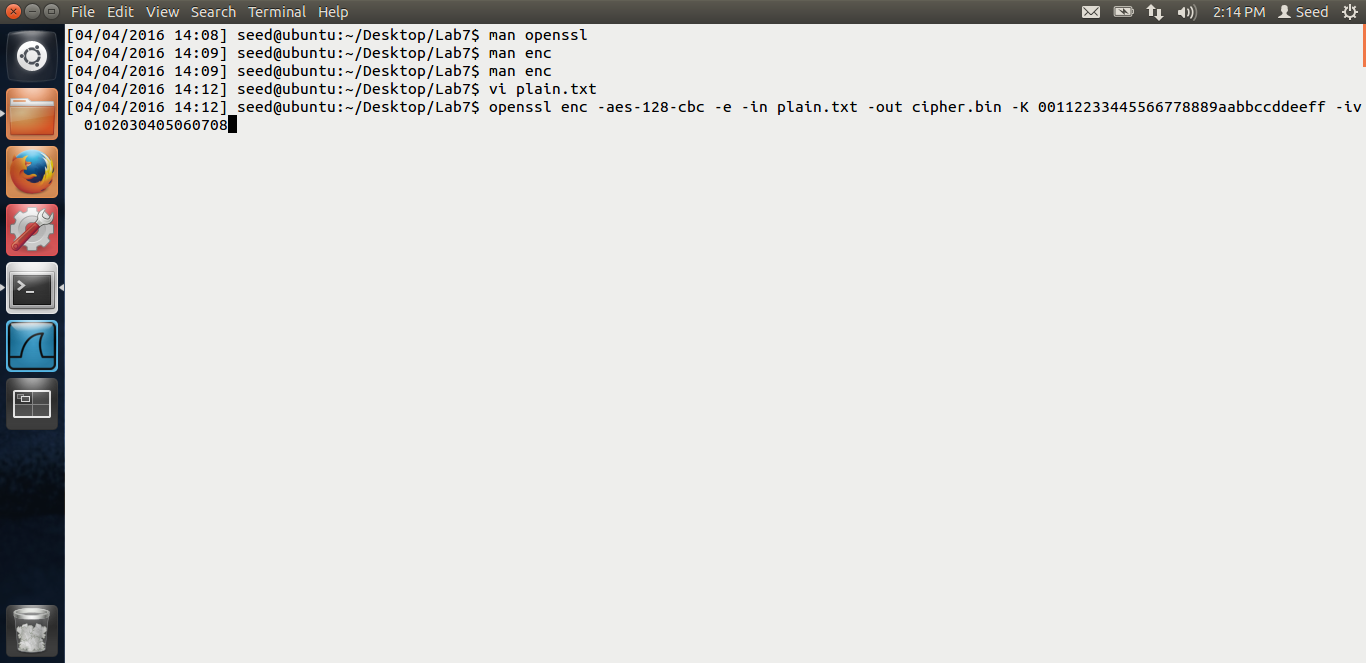
Task-1 is fairly simple, we just have to use the openssl and do the encryption using different ciphers and modes.



After doing the man enc commands, we can see that following ciphers supported by the open ssl libraries.



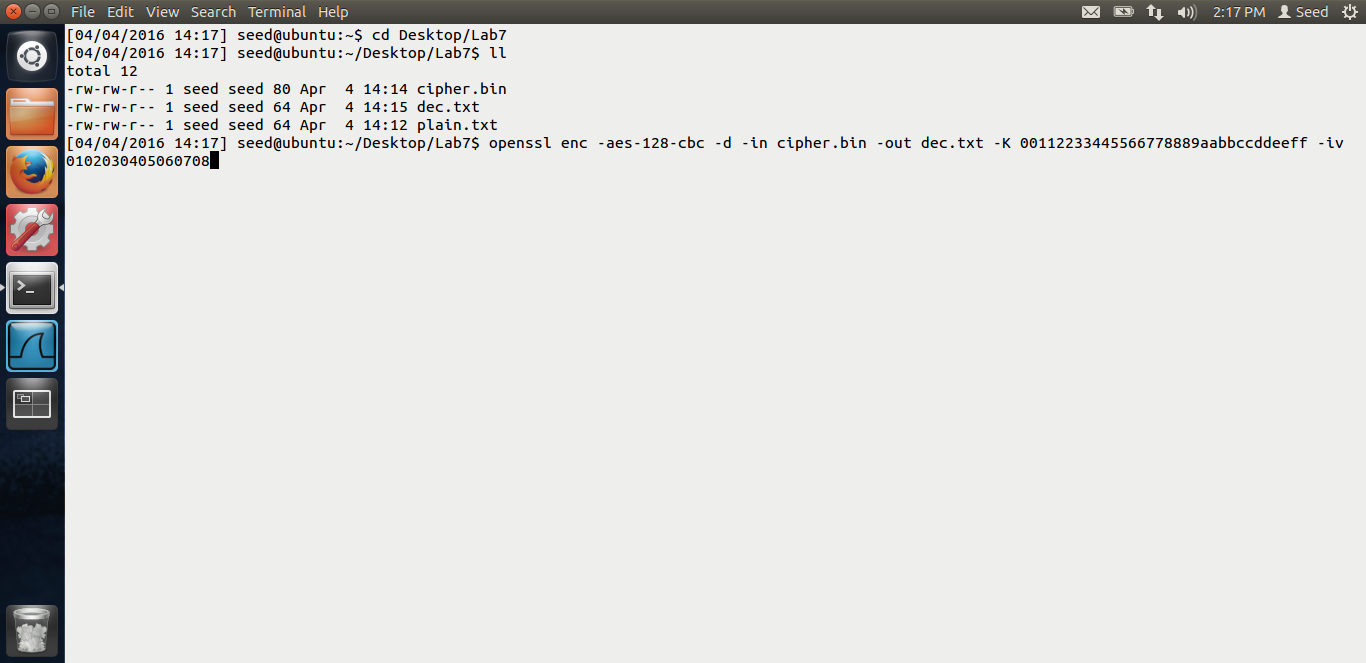
Above is the screen shot of the plain text which I am going to encrypt.



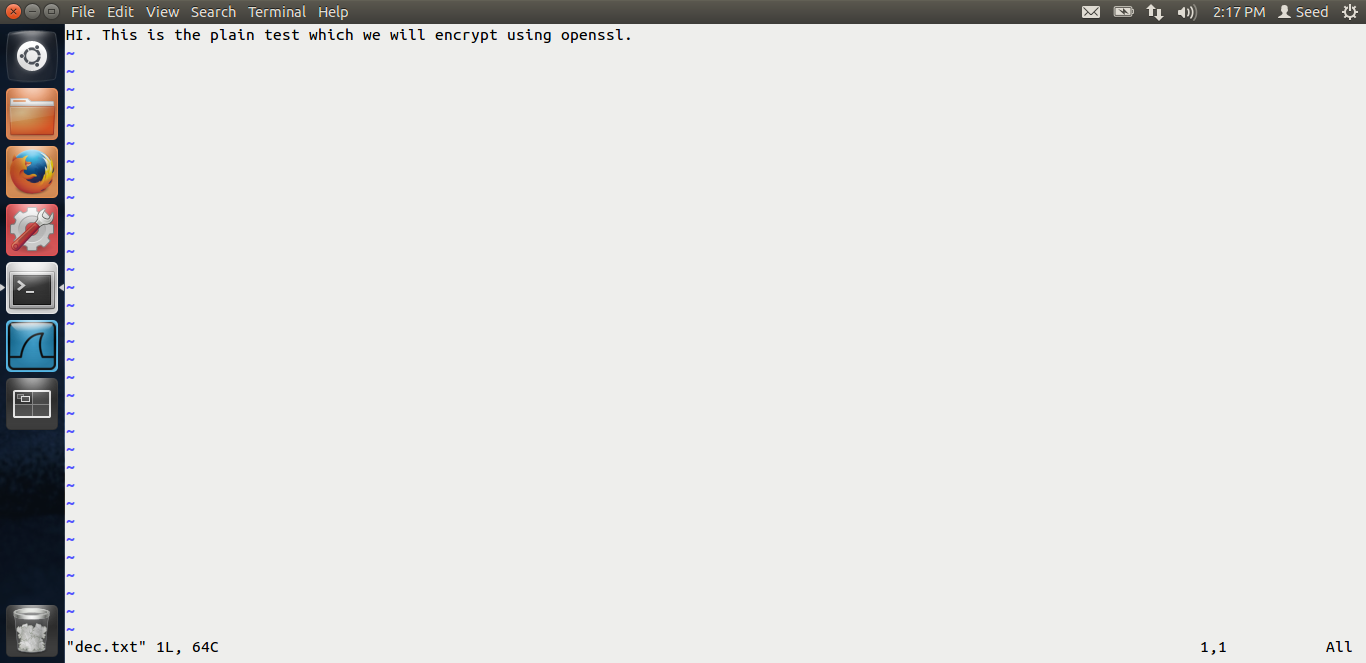
As you can see in the above command, I am using the aes-128-cbc, the encryption is aes-128 bit and the mode is cbc.



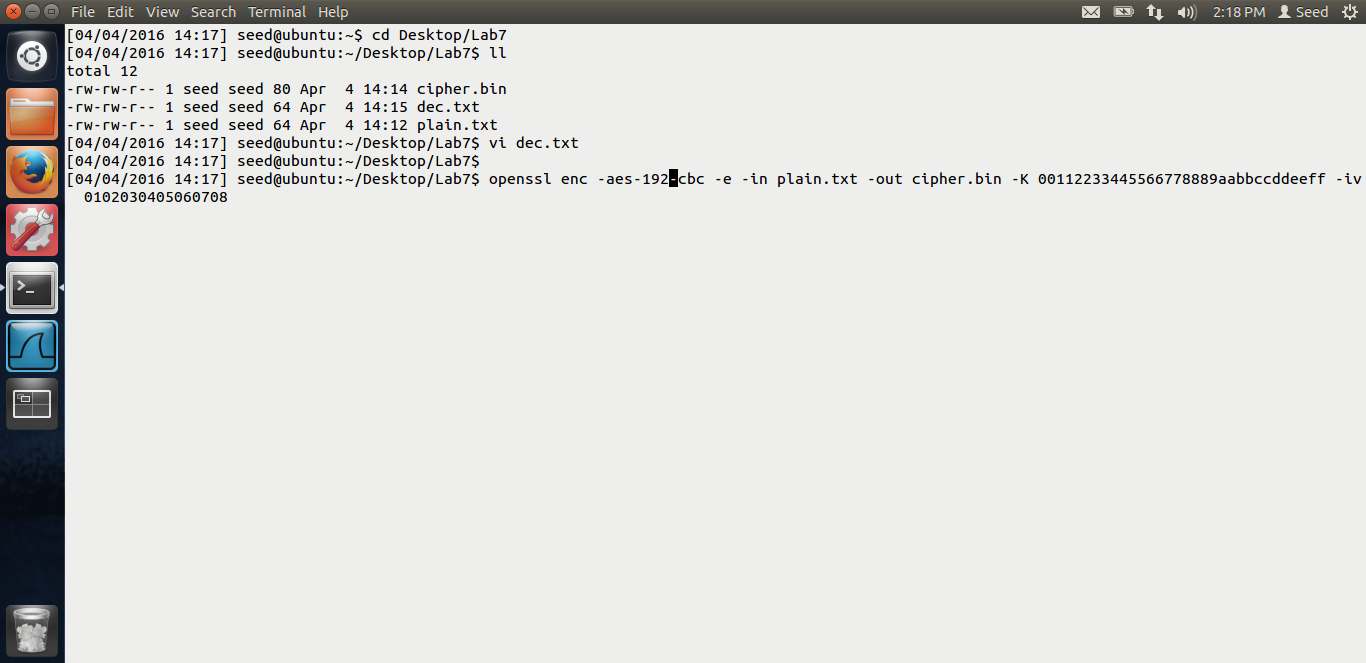
This is the output, cipher text which is being generated.



In the above screen shot, I am using the cbc mode to decrypt the cipher text using the same key and the iv.

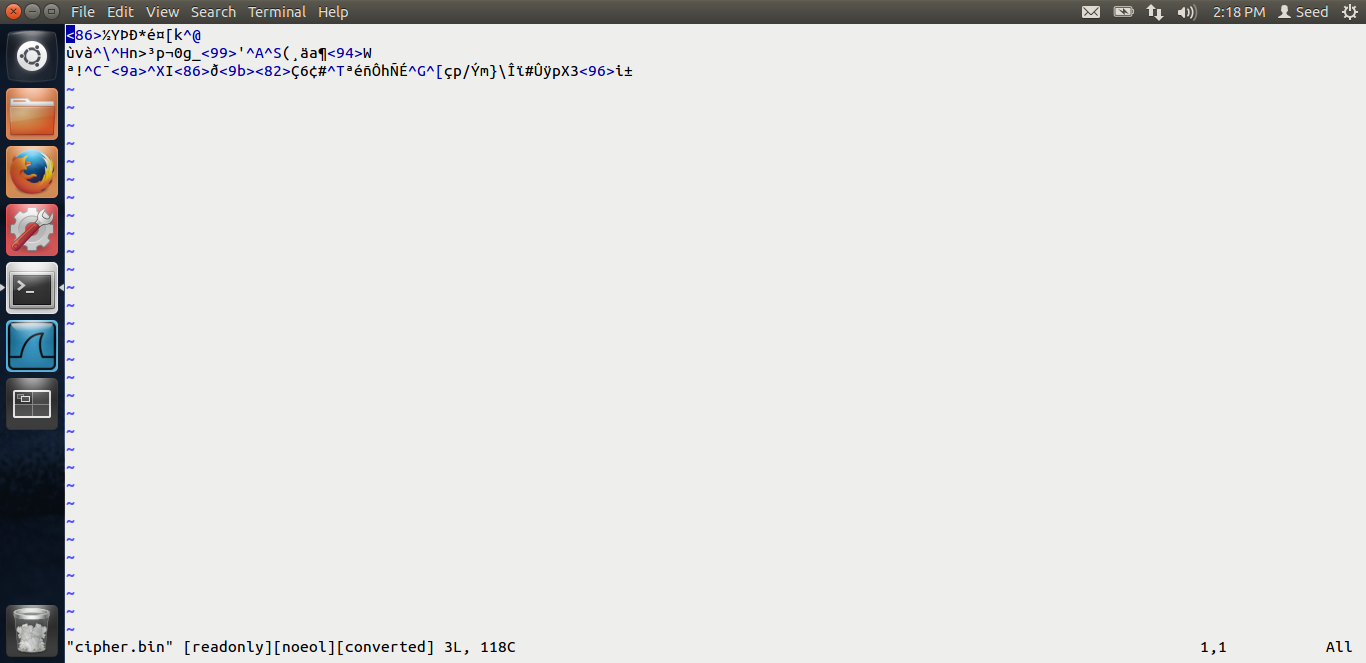


When I decrpty, We get the original text back.

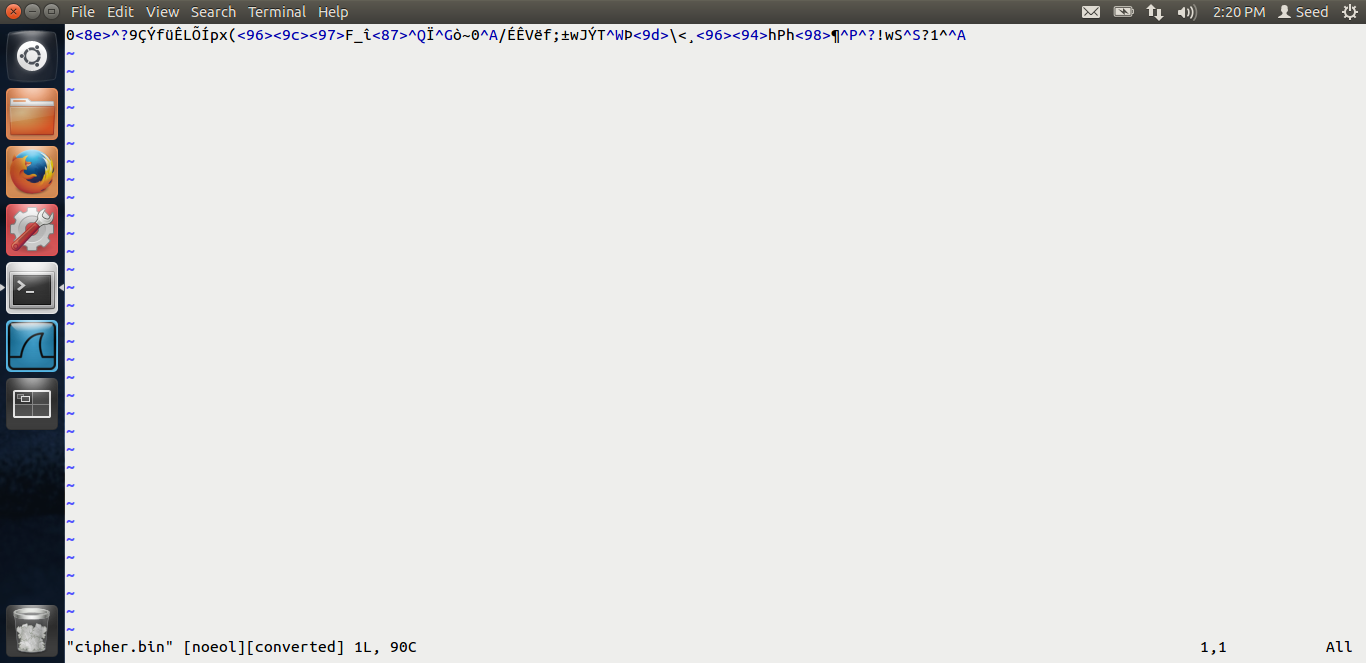


I am just using various encryption algorithms and different modes along with that.

I am using aes-192 bit algorithm in the above screen shot.



This is the cipher text.





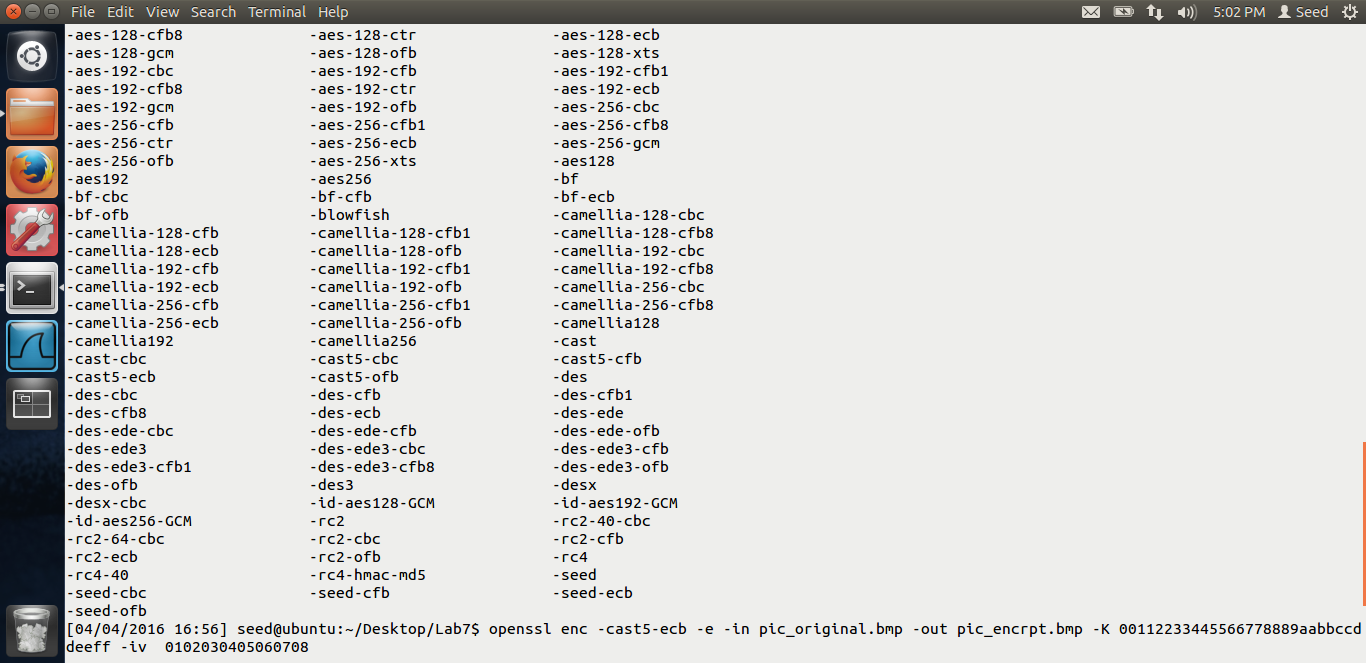


Above are various cipher texts using various algorithms and different encryption modes.

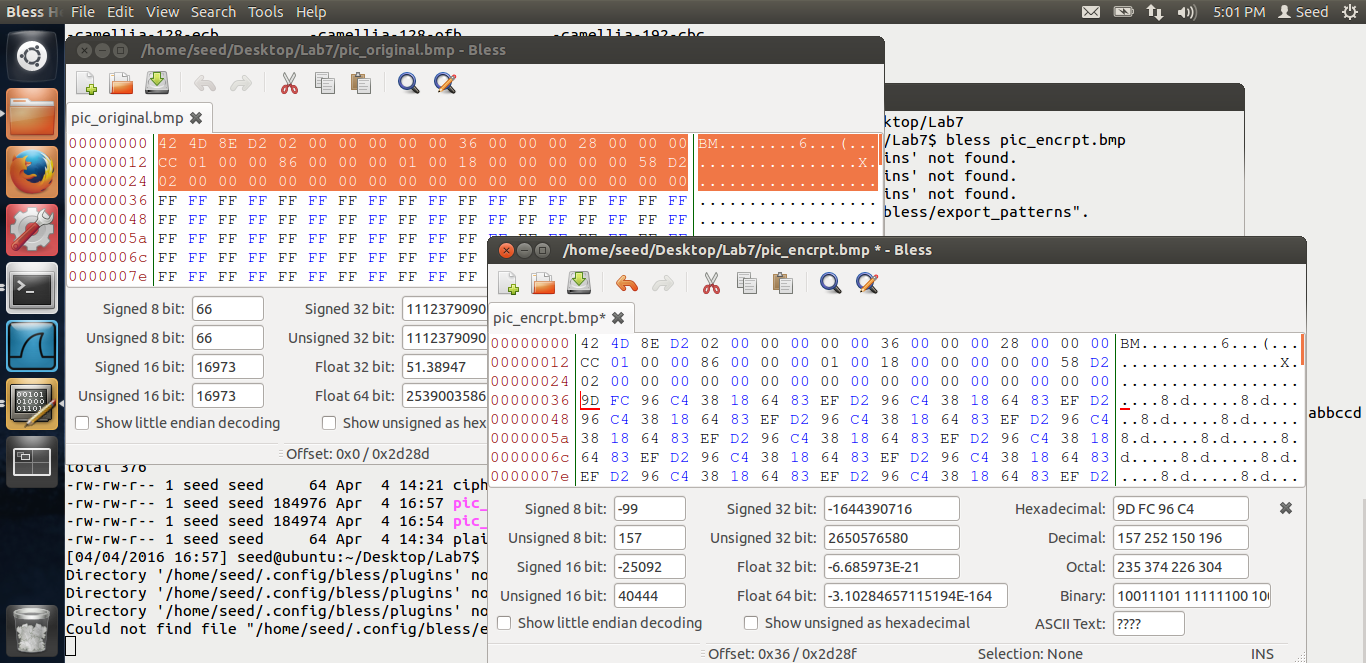
# Task 2:

The screen shots for task-2 are below.

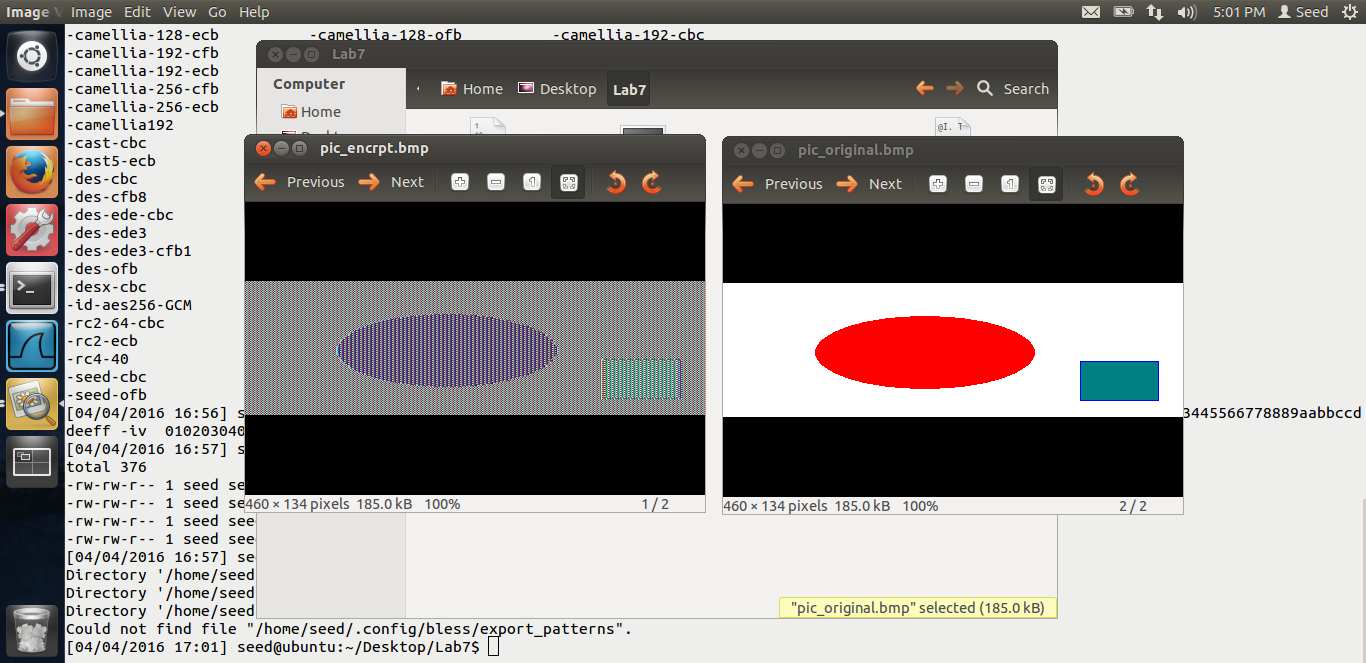
Encrypting the image.



In the above screen shot, we can see that I have downloaded the image and encrypting it using the ecb mode i.e. electronic code book.

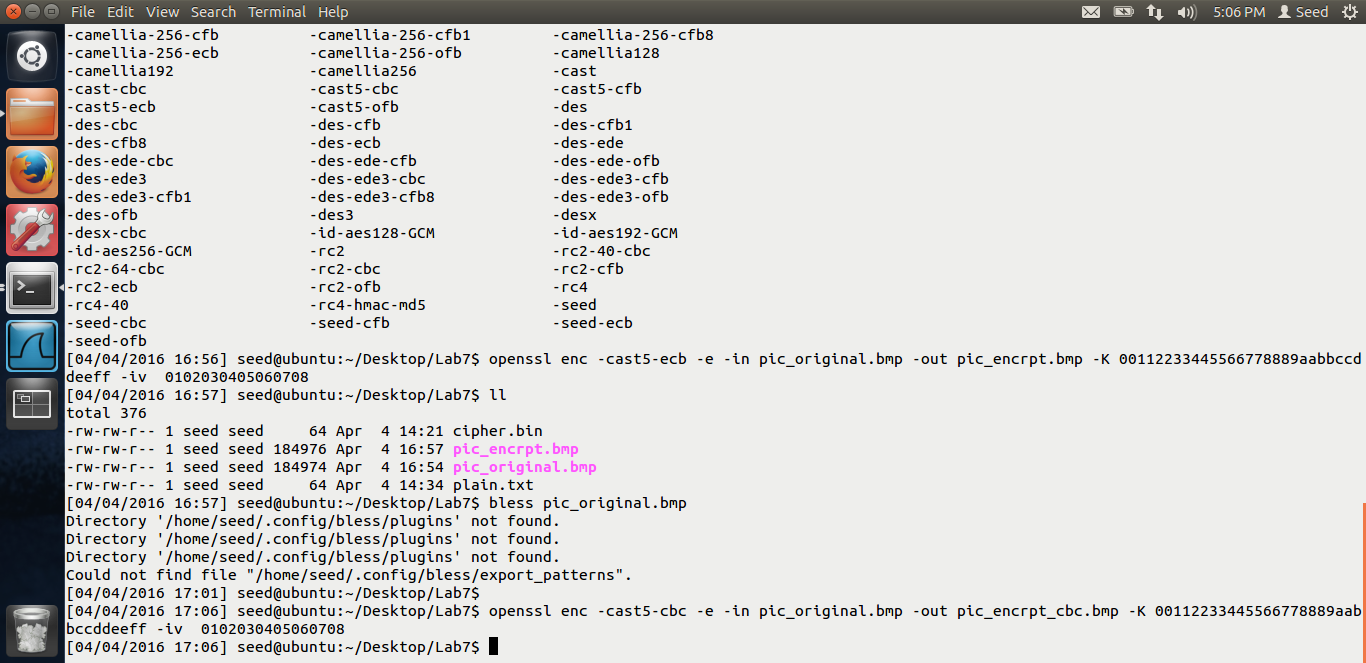


I am changing the first 54 bytes of the header as mentioned in the lab using the bless hex editor.

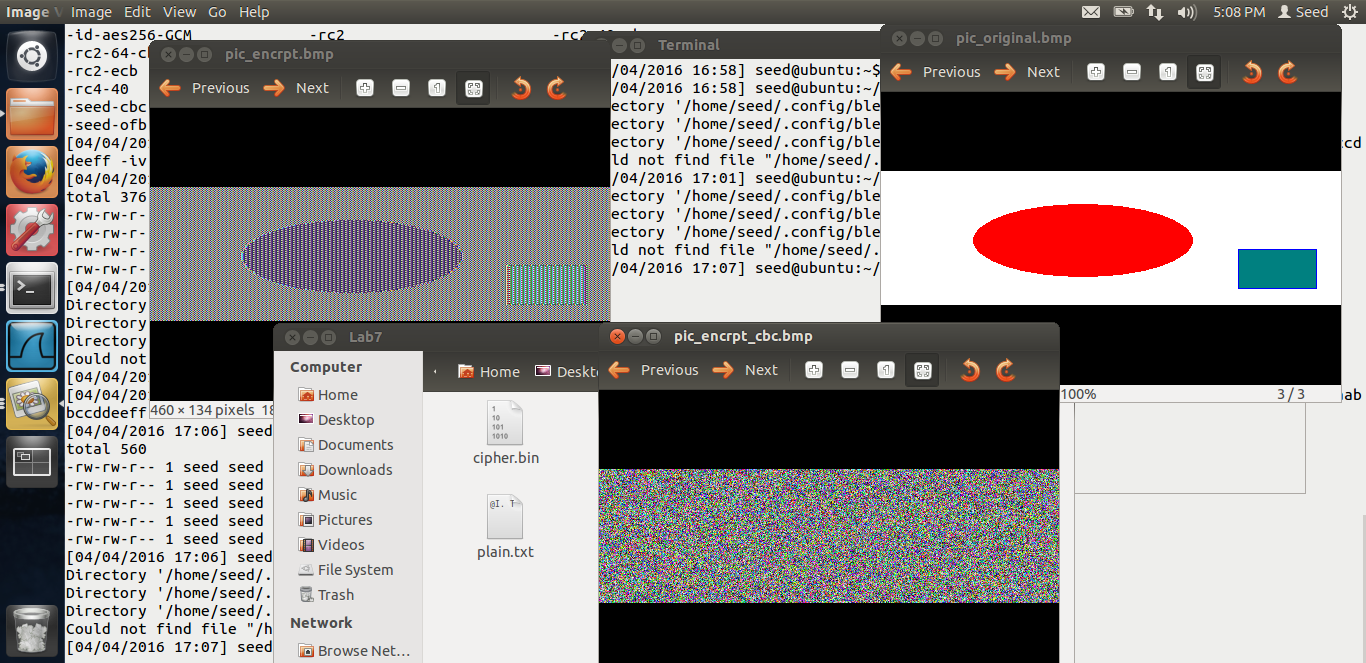


We can see the encrypted image. We can see that we can derive lot of information from the original image when we use the ecb mode.

Reason: Reason is ecb mode, for every block, the same key is used. So if there are many blocks with the same data, same encrypted text would be generated. Hence as images have many common blocks, using the ecb mode, we can derive lot of information.



Now, I am doing the encryption using the cbc mode, cipher block chaining.

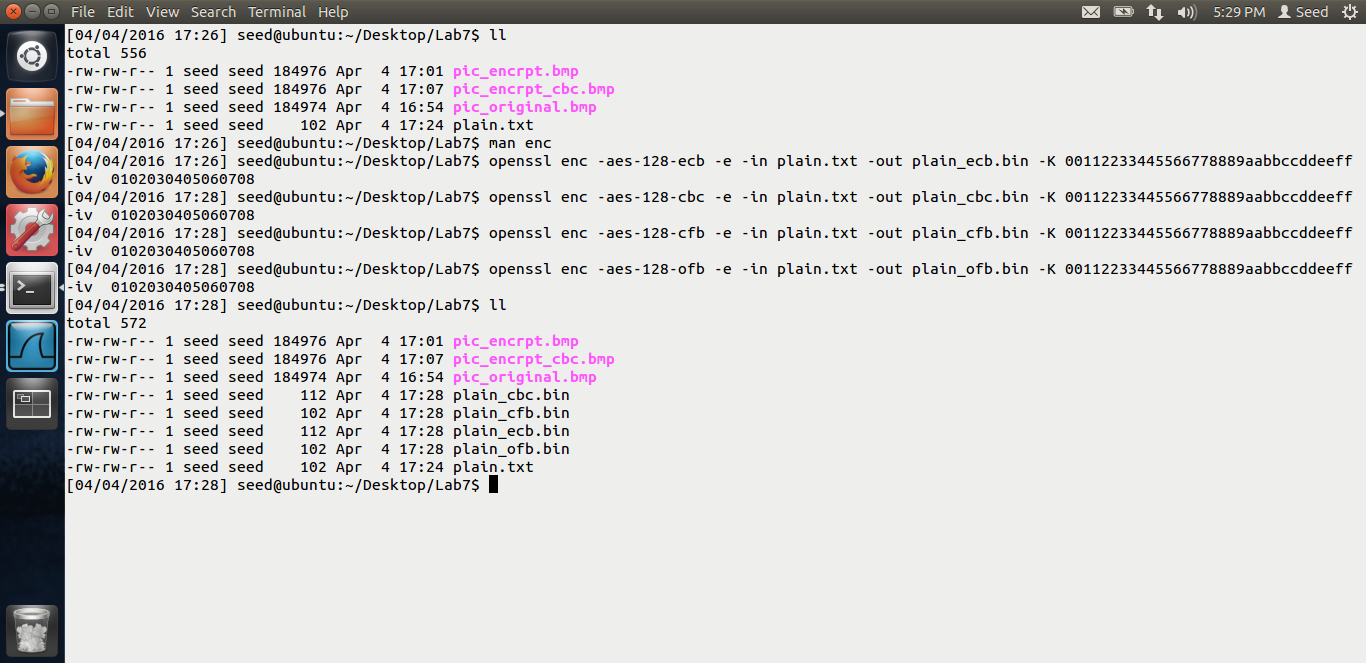


In the above screen shot, we can see that, the below image is the encrypted image from the cbc mode. Thus, we cannot derive any useful information if we use the cbc mode. The reason is, for the next block, the key is the same, but the input is XORed with the previously generated output. Thus, even though the consequtive blocks have the same value, the input to the algorithm is different generating various values. Hence, we get the encrypted data from which we cannot derive any information.

# Task 3:

The screen shots for task-3 are below.

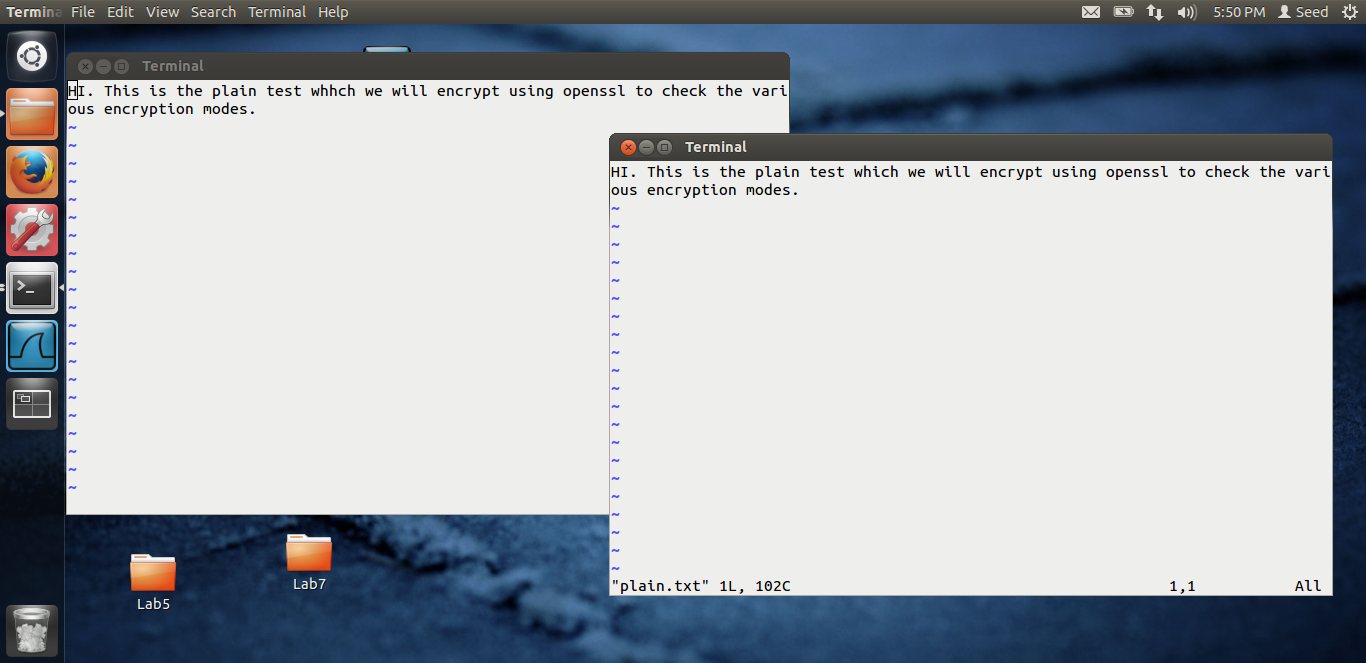
In this task, we are looking at what the output will be in the corrupted cipher text. We will use all the 4 modes using the AES-128 bit cipher.



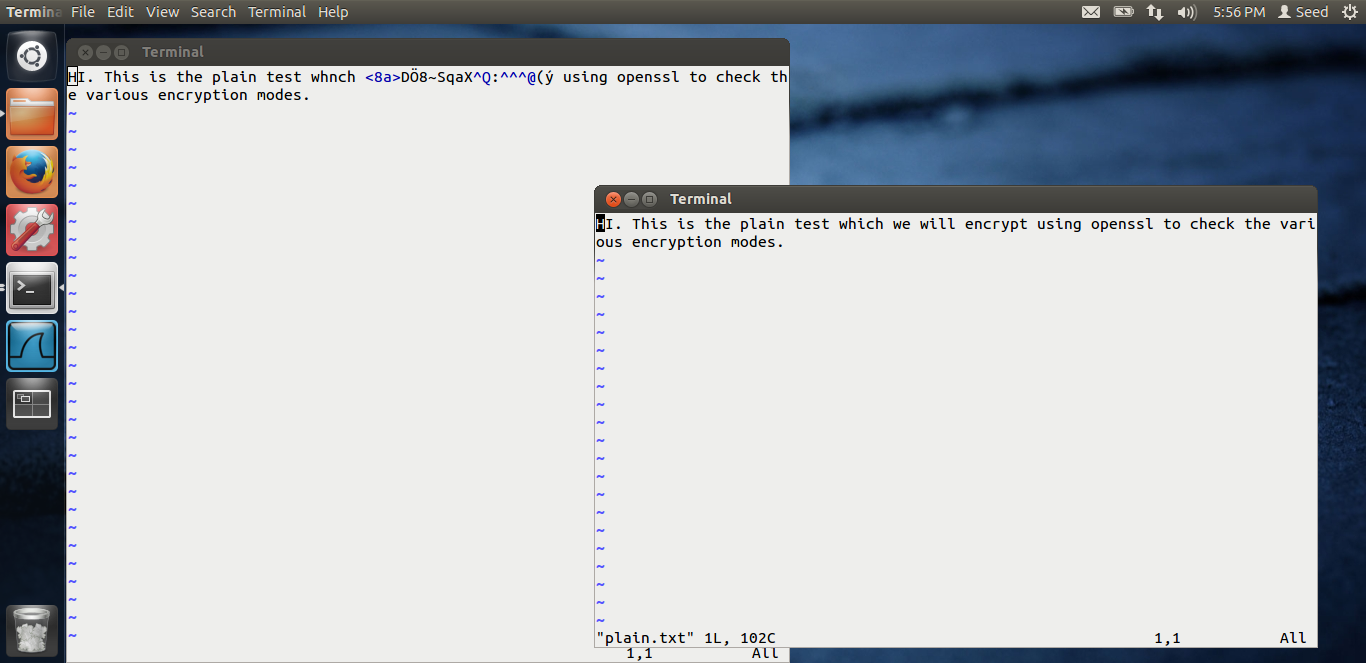
In the above screen shot, we can see that I have encrypted using all the 4 modes using the aes algorithm. Now let us decrypt and see what happens, I have also corrupted 1 bit of the 30th byte.



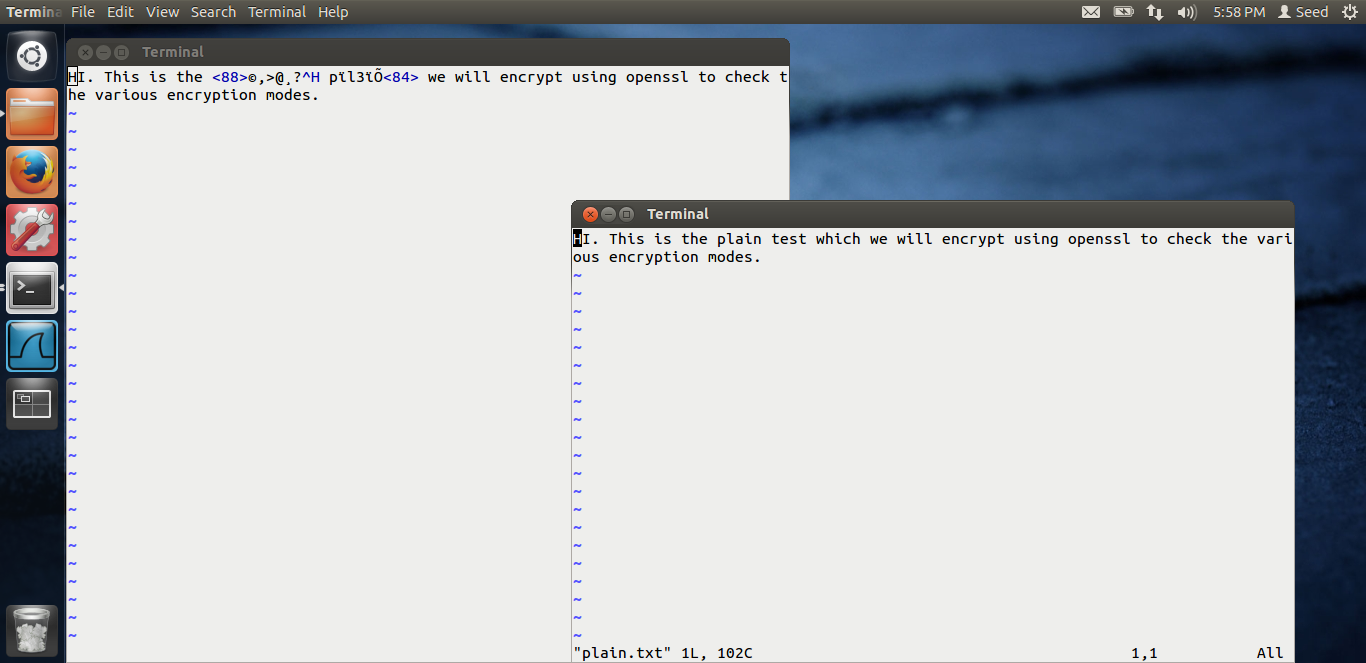
**CBC:** This is the output for the CBC mode. As we can see, the second block and 1 bit of the 3rd block is corrupted during the decryption. Rest, all the decryption is perfect.



**OFB:** Above is the output for the OFB decryption. We can see in the decrypted text, “whhch” is corrupted from the original “which” plain text. Just that 1 character is corrupted. Reason is, in OFB mode, the cipher text is XORed with plain text for every bit, as it’s a stream cipher, thus only that 1 bit is corrupted



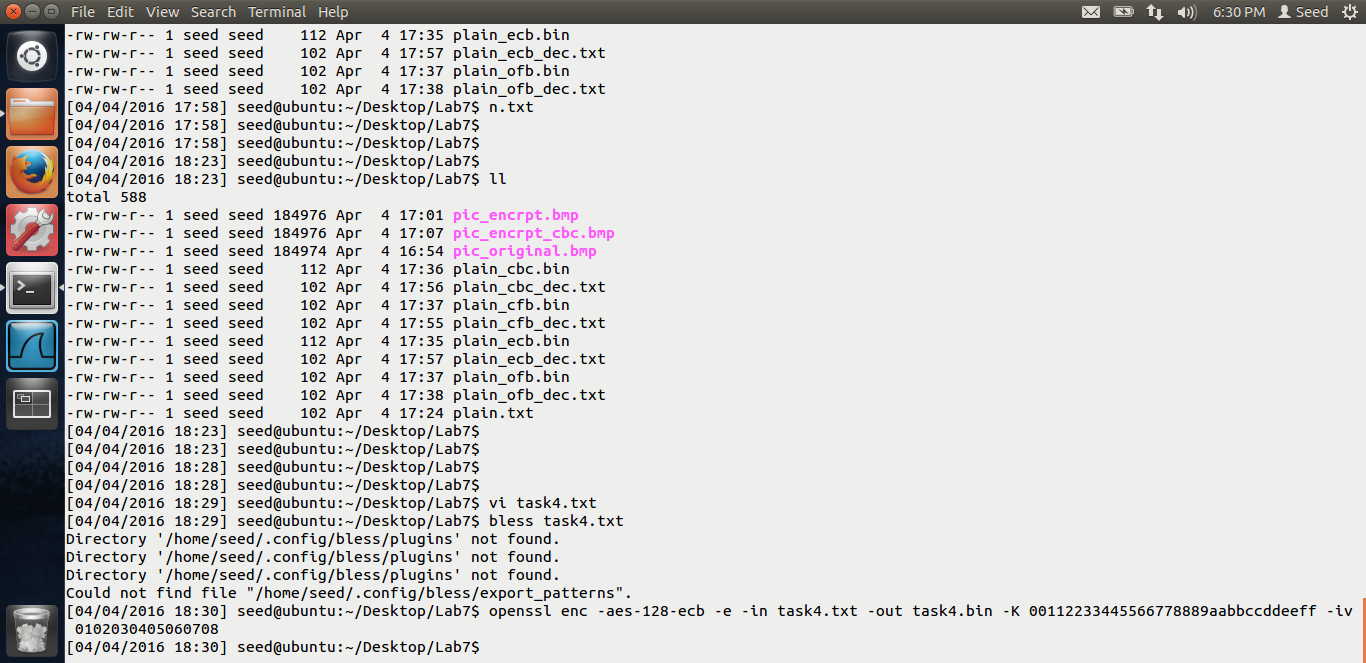
**CFB:** This is the output for the CFB mode. We can see that from the corrupted bit, that block and the next block is corrupted in the CFB mode. Reason is, as the output of the previous block, acts as the input to the next block, i.e. the cipher text XORed with the plain text, as the plain text is corrupted for the previous block, the next block also gets corrupted. Hence we get the above output.



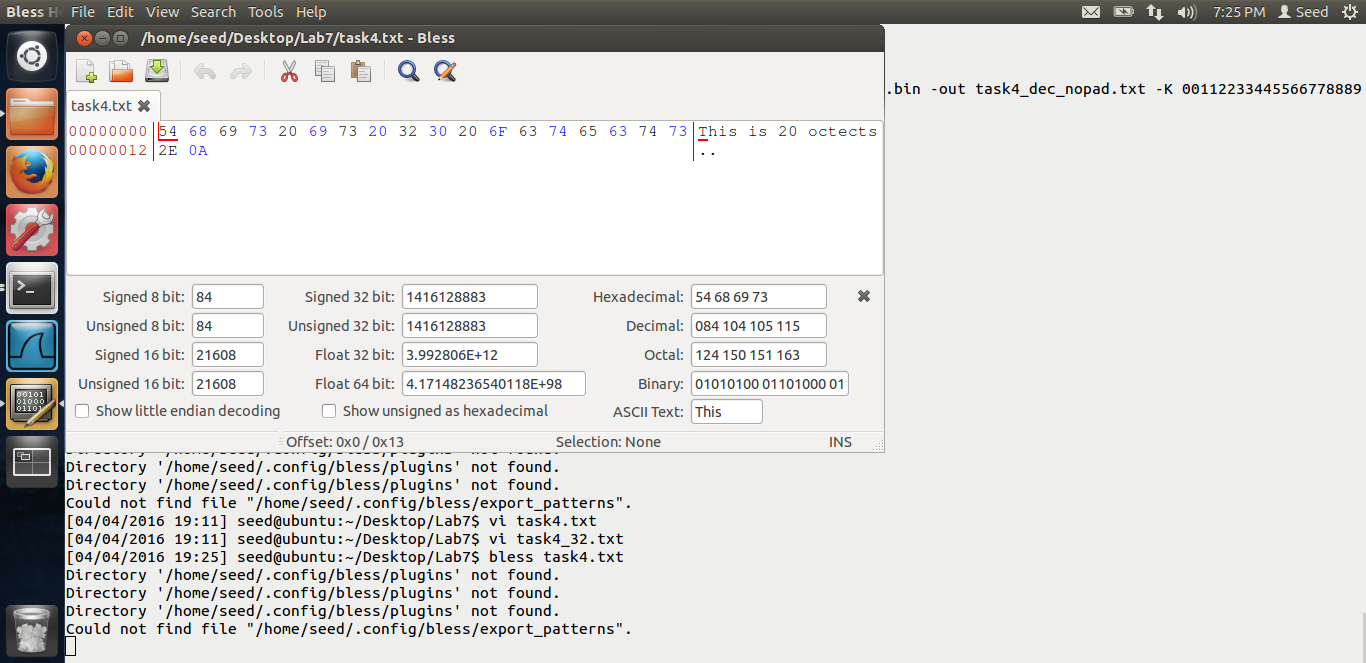
**ECB:** In the ecb mode, that entire block gets corrupted, nothing else. As all the blocks are encrypted individually, only that block is corrupted.

# Task 4:

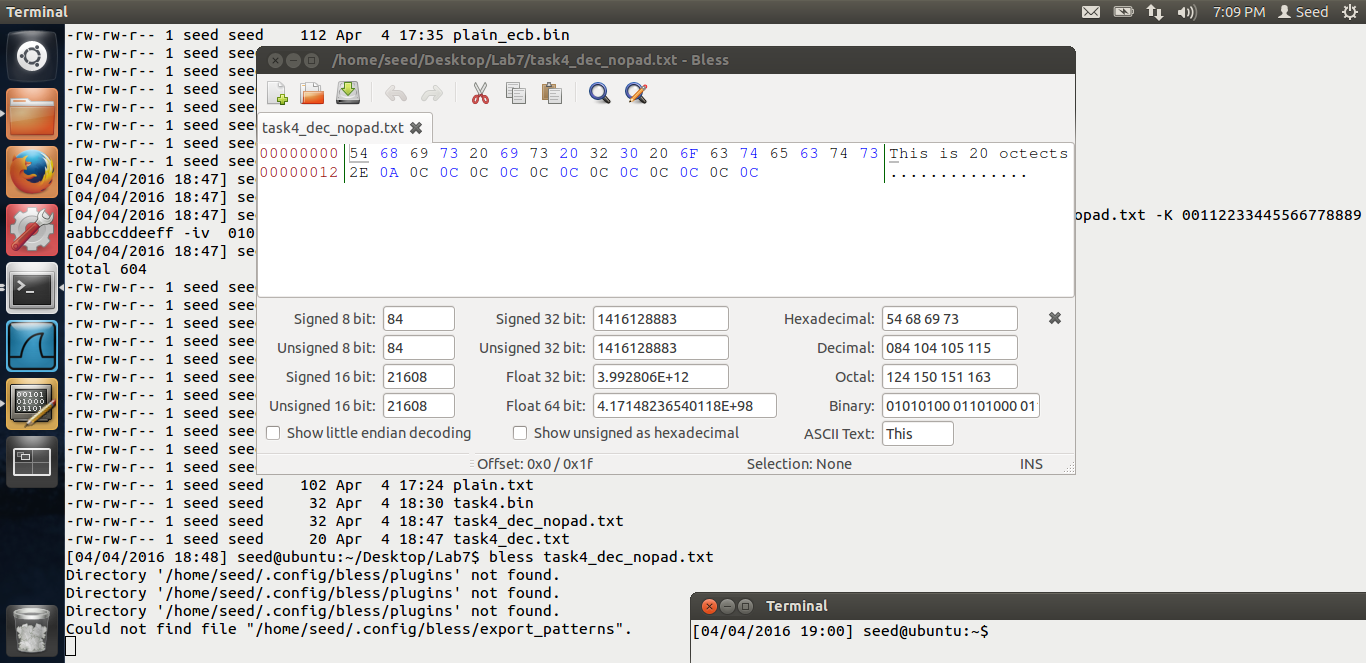
The screen shots for task-4 are below. This is the padding task.



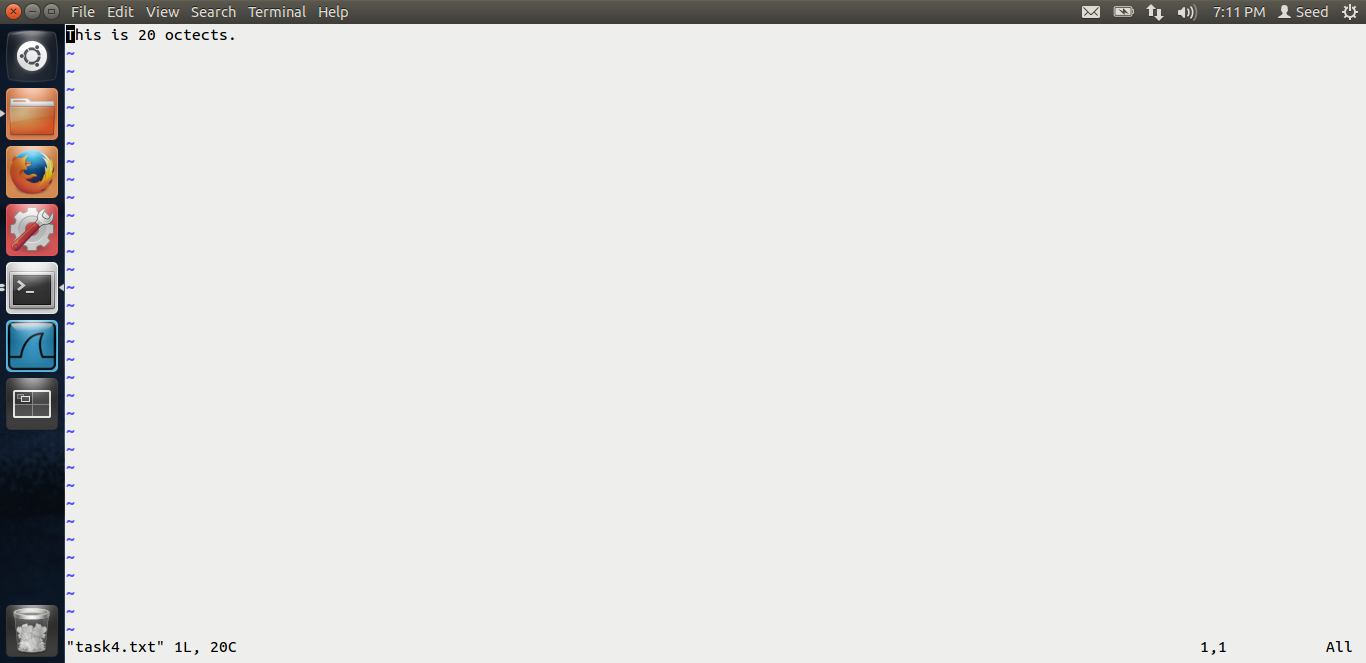
First, I am encrypting a 20 octet block using the ECB mode.



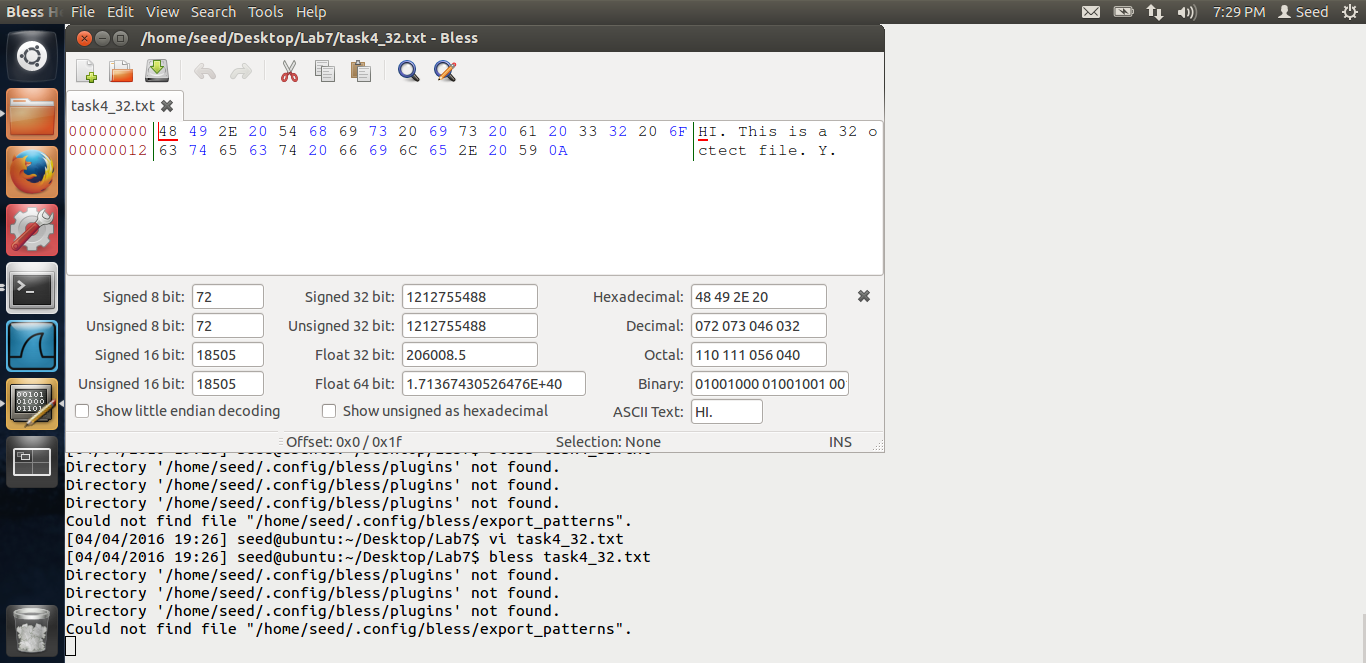
In the above screen shot, we can see that the plain text is 20 octet.



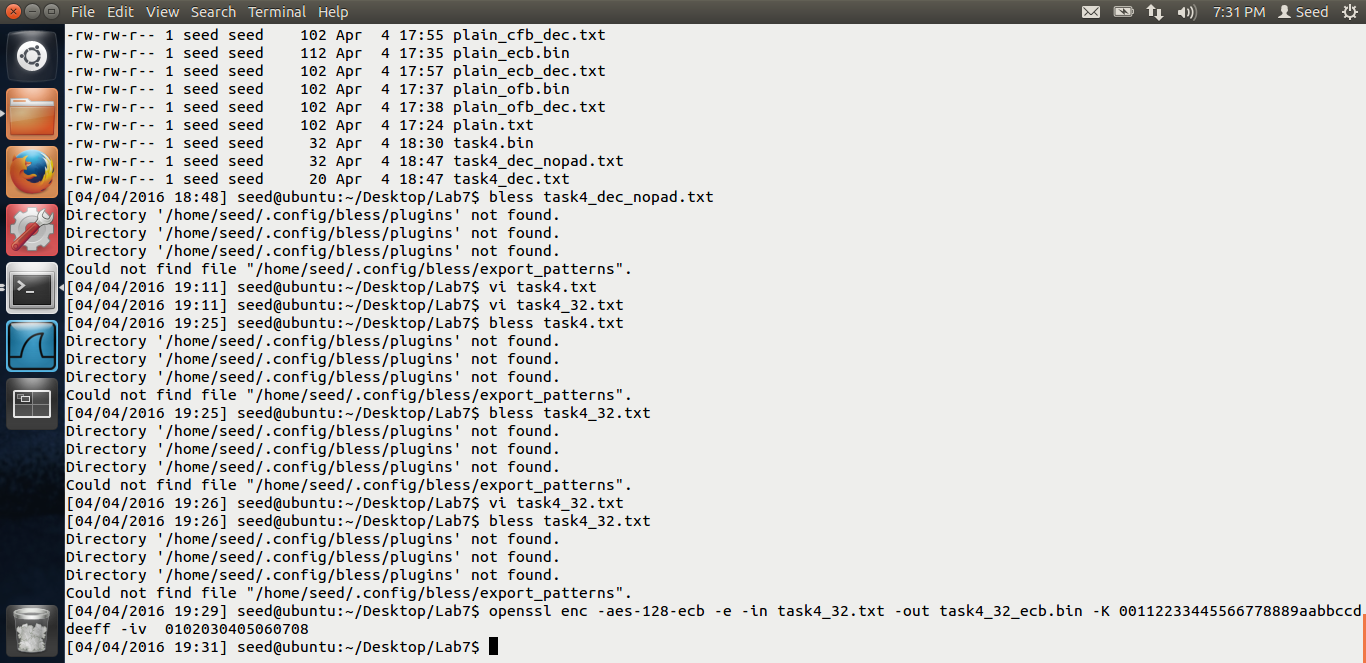
Once I decrypt that, using the –nopad option, we see whether padding was used. This is the screen shot after decryption. We see that the padding “OC” was used. Means 12 octets, i.e. 12 octets were used for padding. Thus, padding was used in the ECB mode.



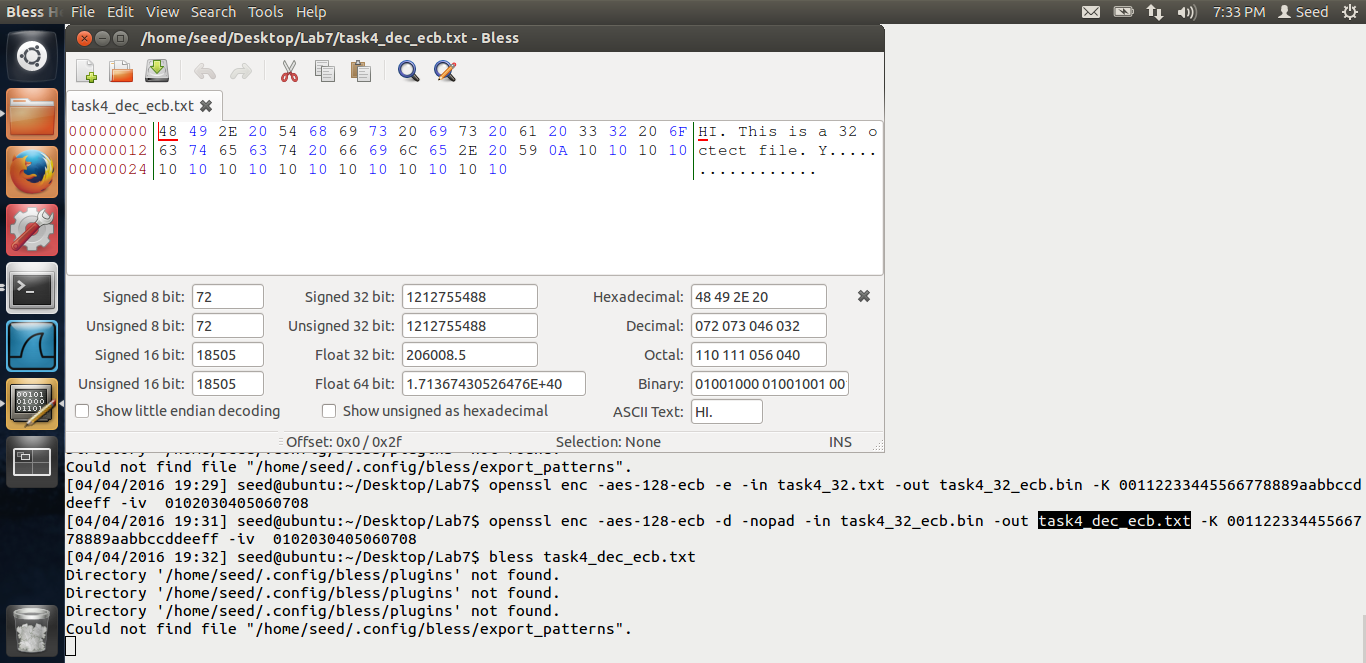
This is the screen shot for the plain text which I used.



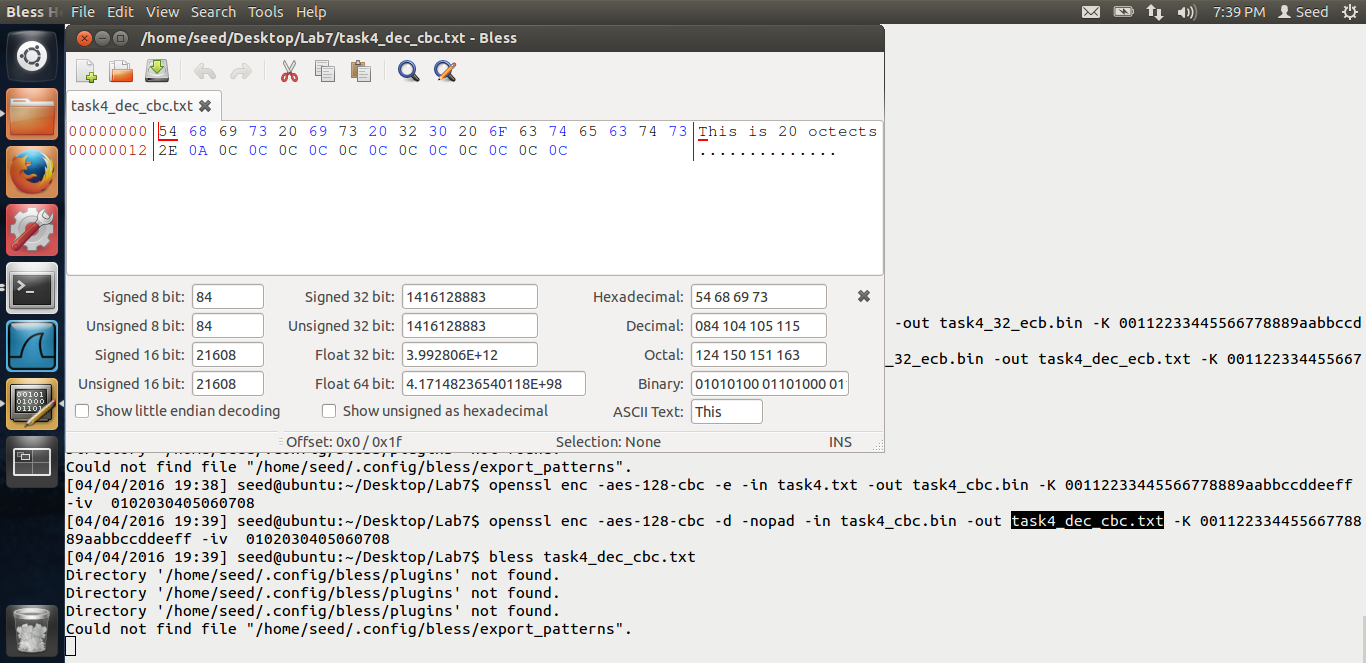
This is the screen shot for a 32 octet file. Lets see whether padding is used for this file. I am again using the ECB mode.



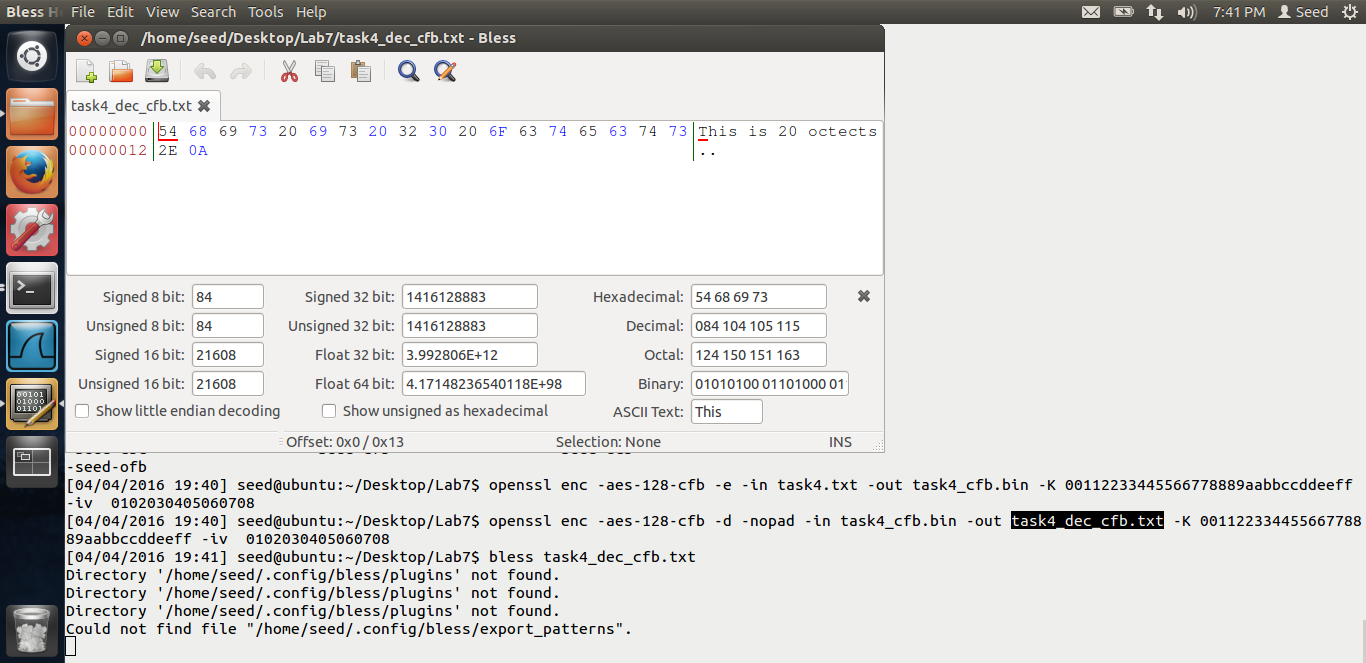
Above is the screen shot for the same.



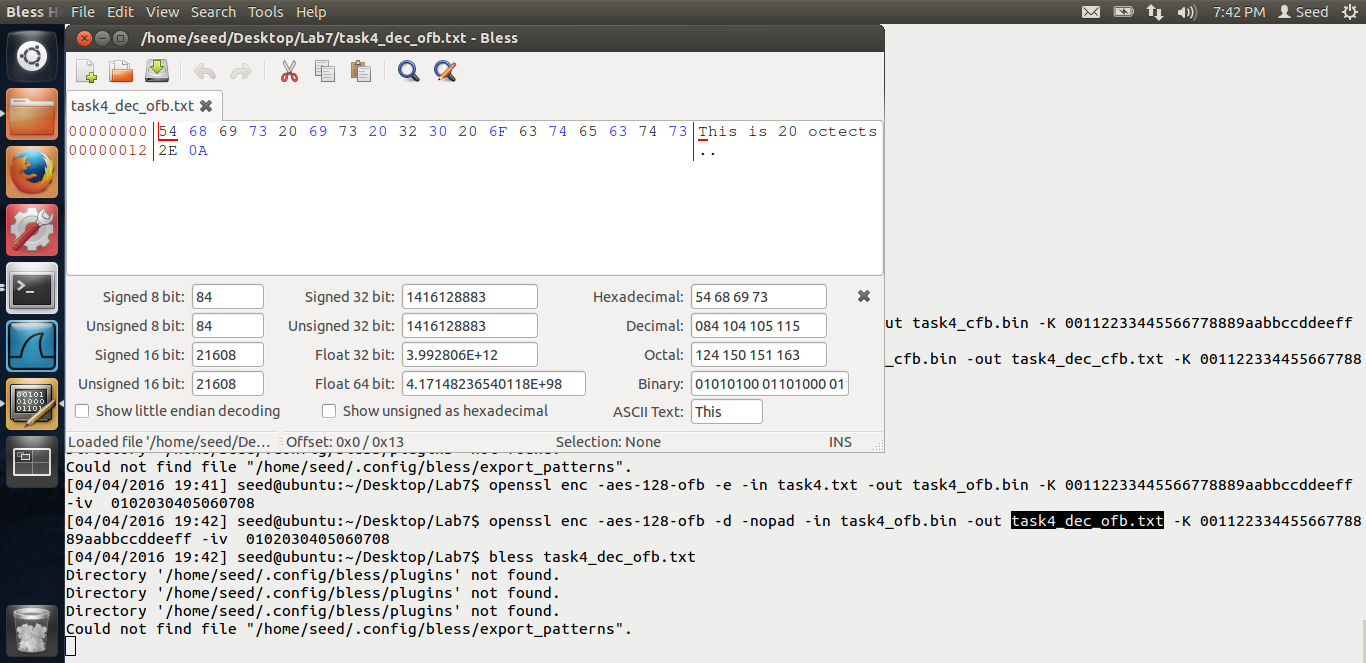
Thus, we see the padding clearly, “10” i.e. 16 octet padding was used. Thus, the PKCS5 scheme is used and the ECB mode uses padding.



**CBC mode:** Above is the output for the cbc mode. Thus, this is a 20 octet file. As we can see that both the above ECB AND CBC mode are block ciphers, they both use padding.



**CFB:** Above is the screen shot for the CFB mode. We can see from the output that no padding was used Reason is, CFB is a stream cipher, thus it does not require any padding and as the input comes, it gets encrypted.

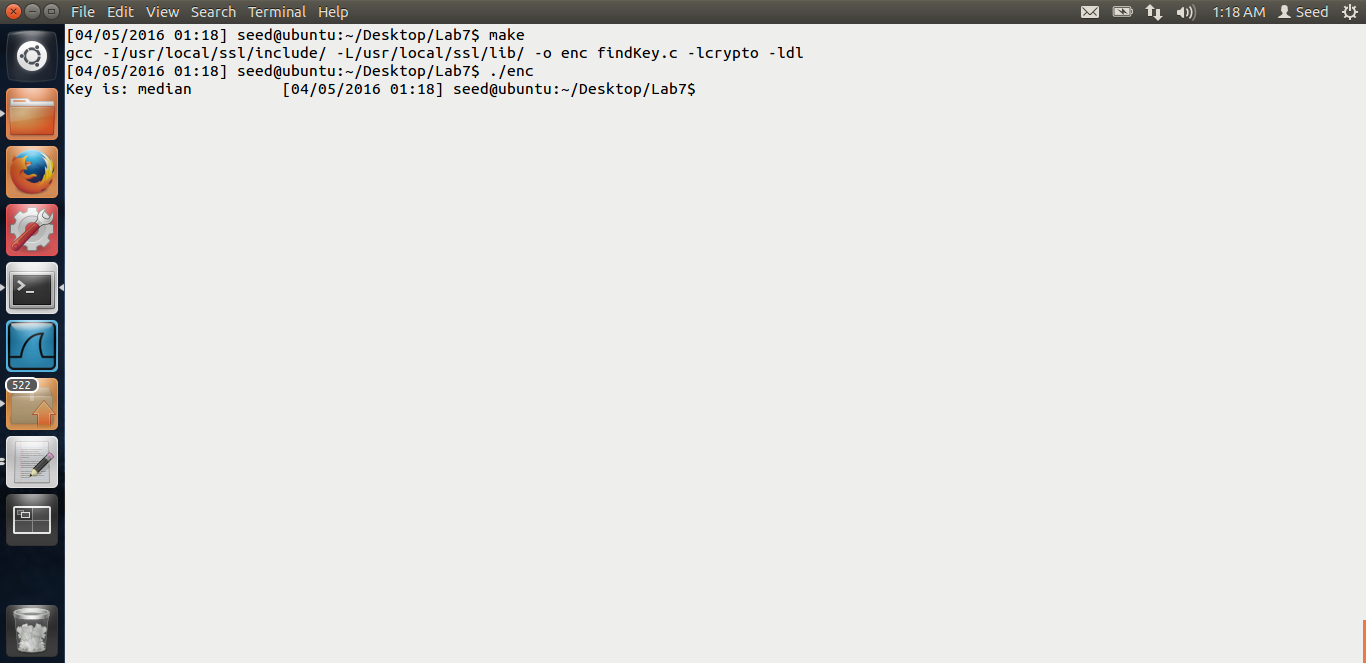


**OFB:** Above is the output for OFB mode. Again no padding is used. Reason is the same, as it’s a stream cipher, it does not require padding.

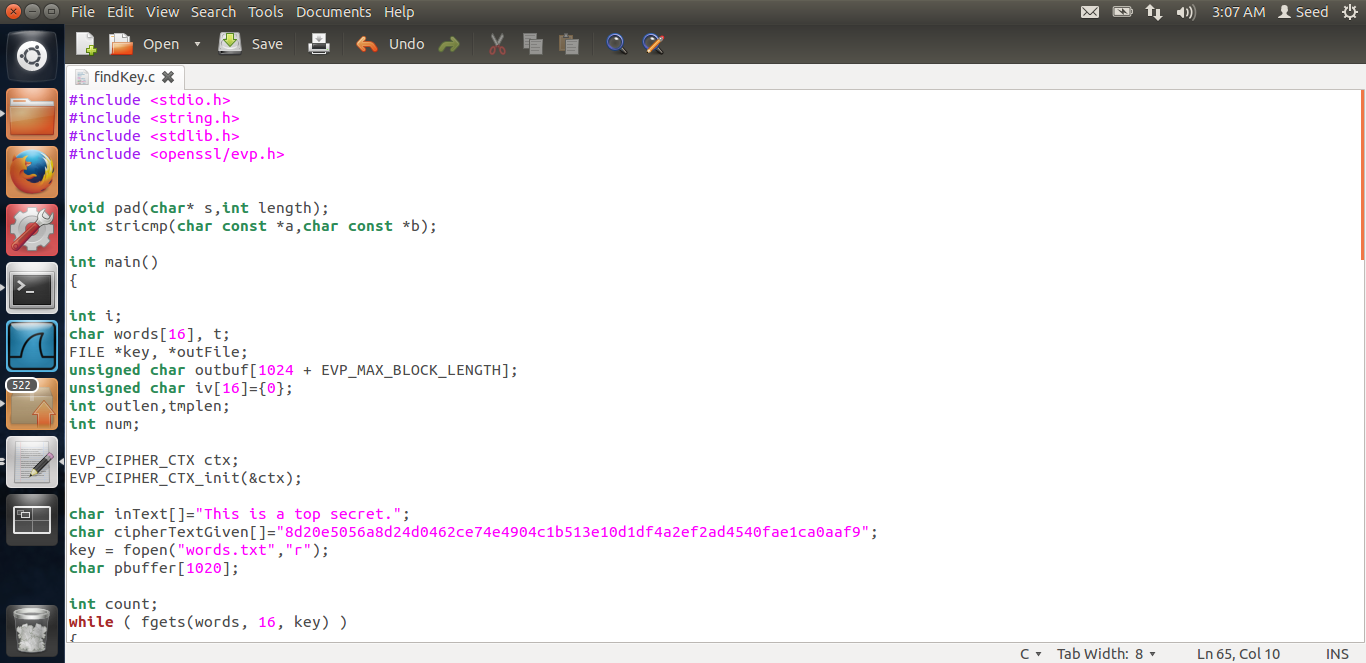
# Task 5:

Screen shots for task-5 are below.

This task is for coding using the crypto library. Here, we are given the plain and cipher text and we need to get the key. The iv is set as 0. First I will show you the output of the program and then the code.

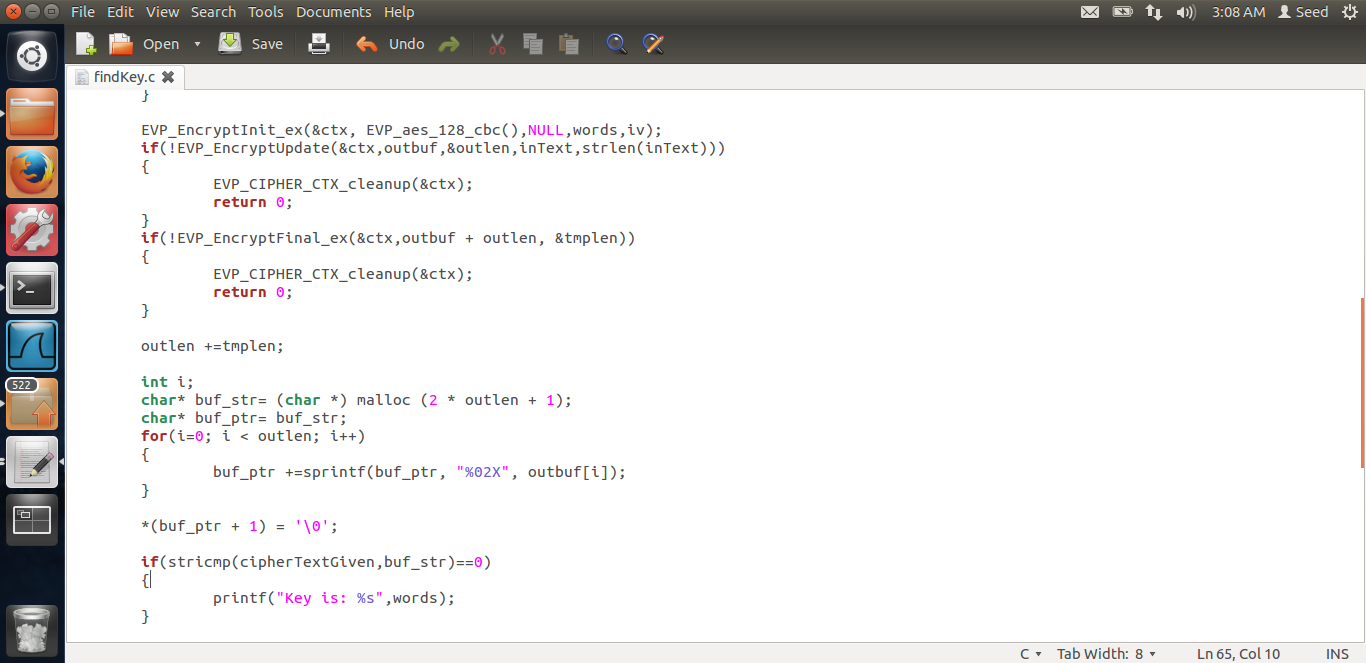


In the above screen shot, I do the make command. And then I run the exe. After it runs, it shows that the key is **median.** Thus median word from the English dictionary was used as the key. Now let us look at the program. The sample program given by Dr. Du I have modified it.

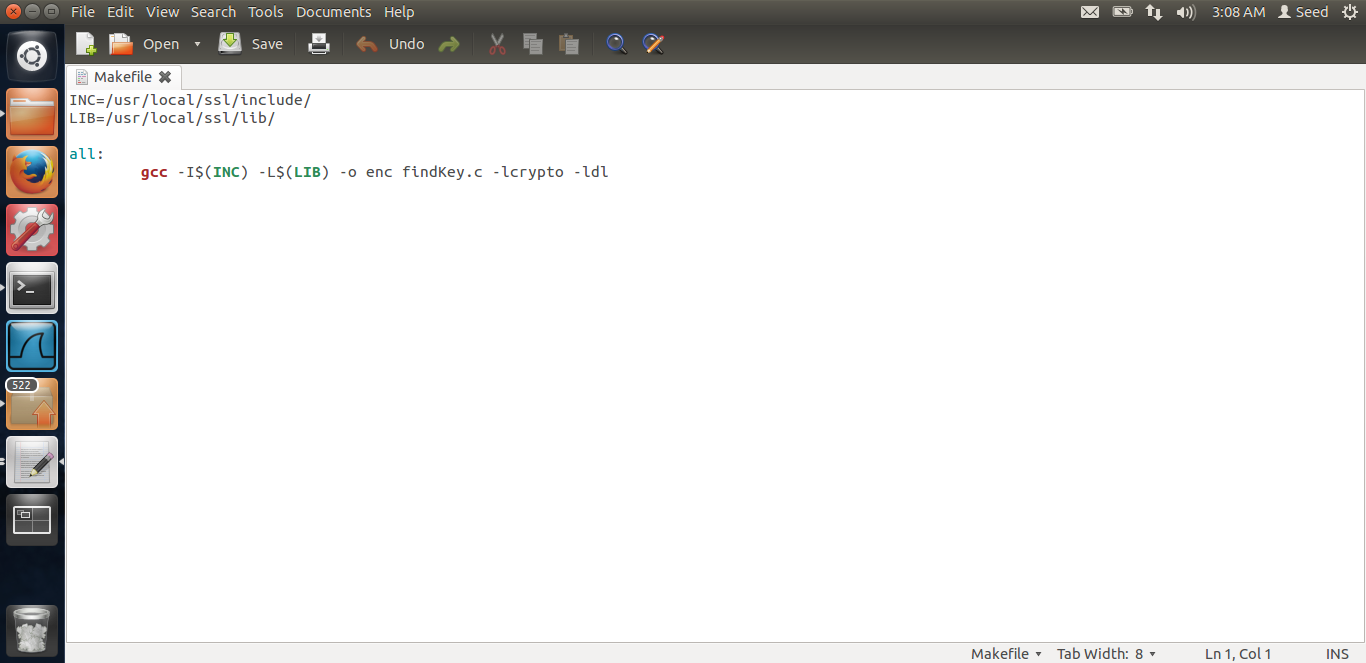


First, I have downloaded the words.txt file from the website. Then I hardcode the plain text and the ciphertext. IV I set to 0.

Then,I do the encryption using each word as key and check if the cipher text is matched. If yes, I print out the key.



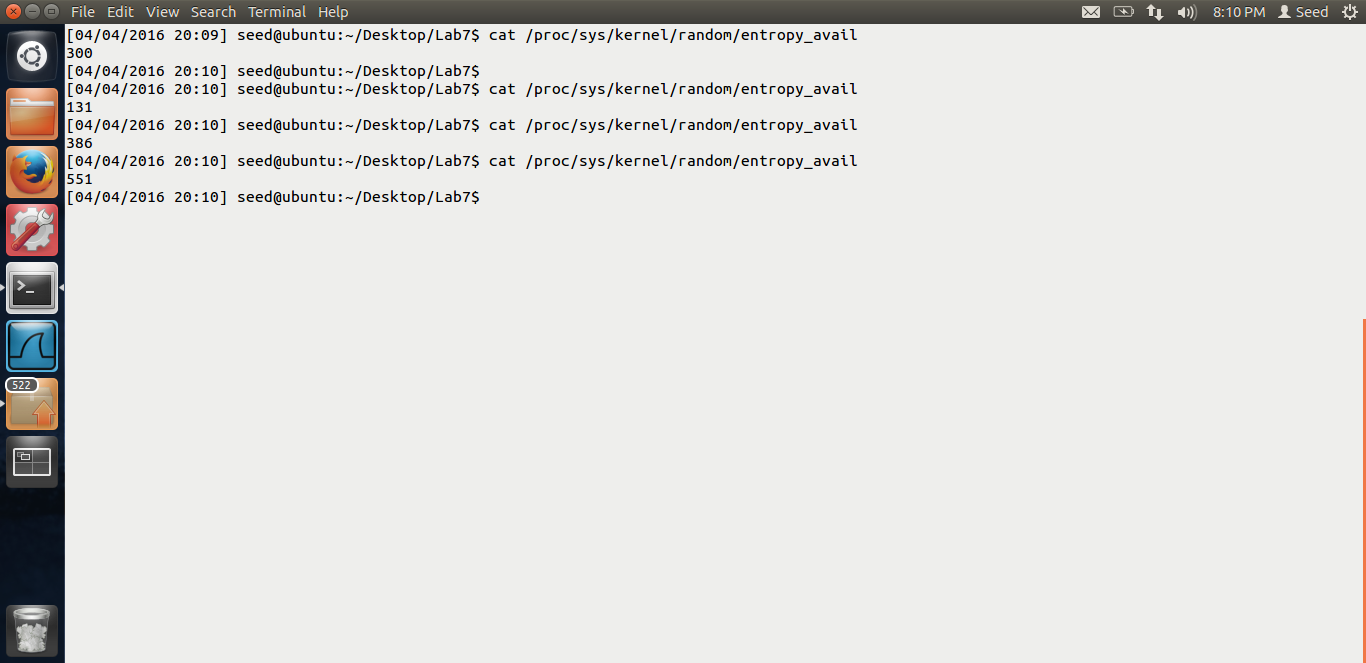
Thus, a very simple if check does the job. Below is the make file.



This make file is given by Dr Du. Thus after running the program, I found out that the key used was “median”.

# Task 6:

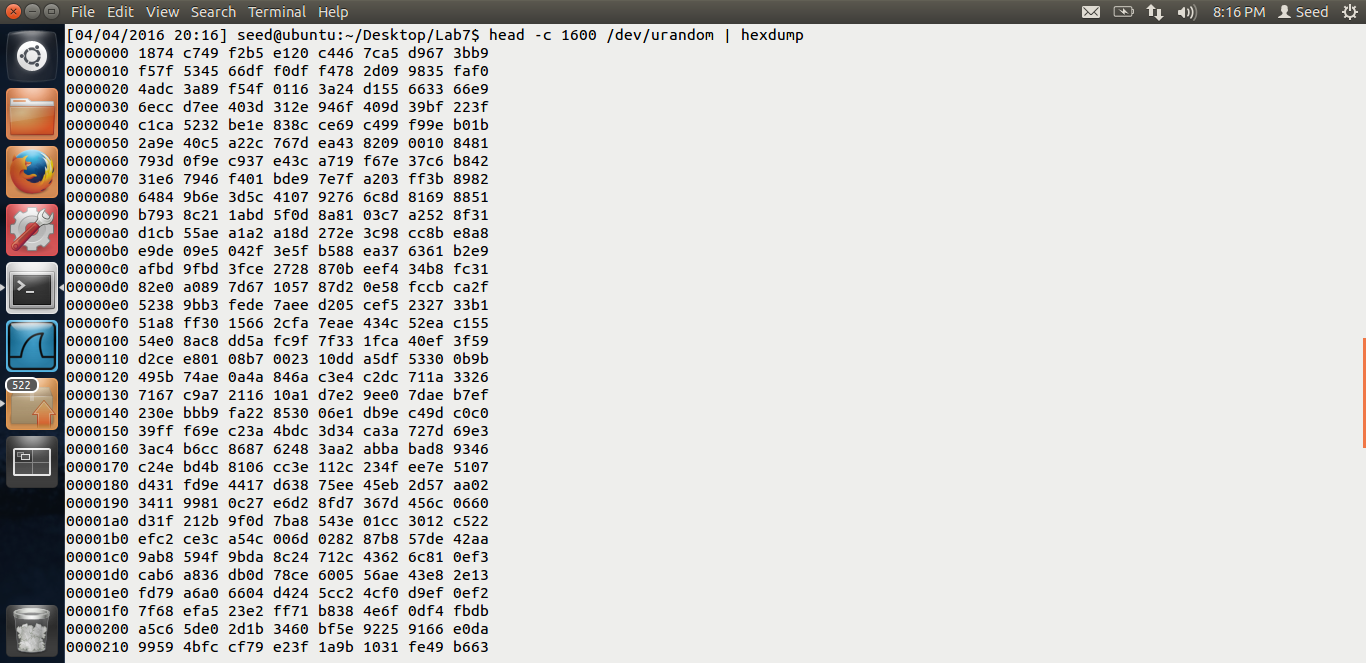
The screen shots for task-6 are below. This task is on random number generation.



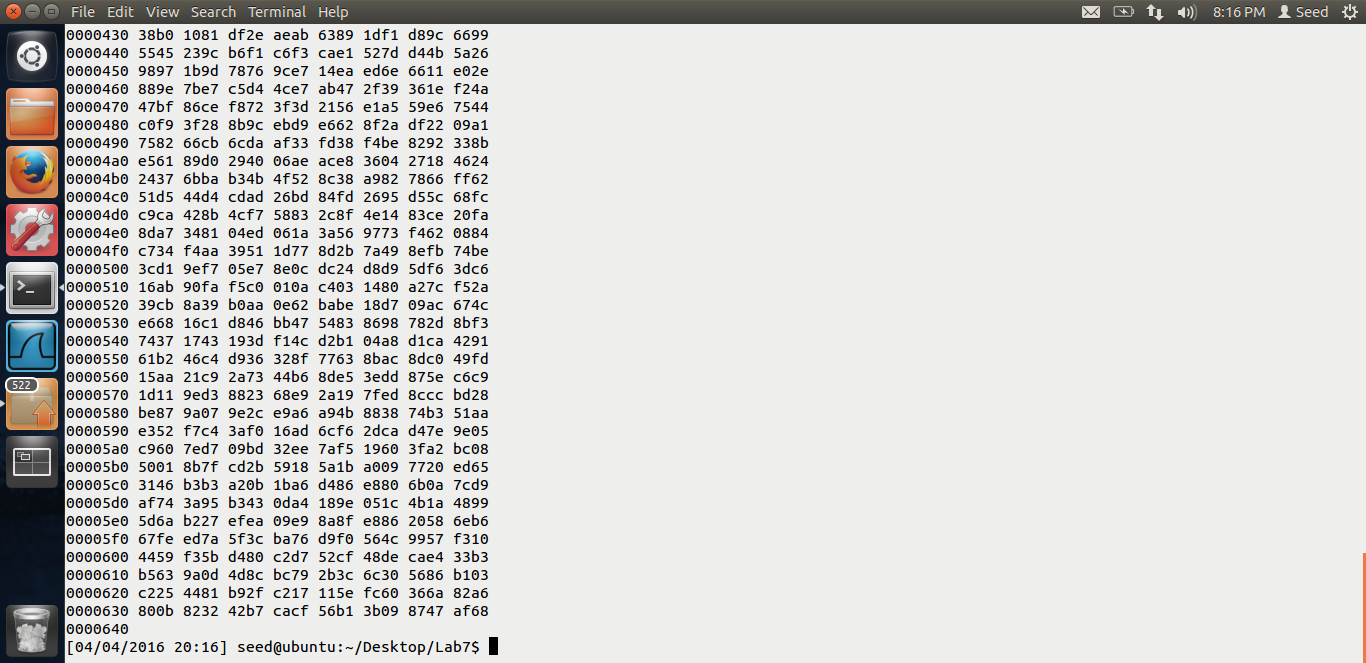
In the above screen shot, we see the entropy of kernel. First, the entropy was 300. Then I waited for sometime, it became 131. Then I moved the mouse. It became 381, then 551. Thus, the entropy increases as the randomness increased.



Above is the screen shot for /dev/random. Printing 16 bytes. First it printed quickly. Next time, I waited for 1 min as seen from screen shot, I moved the mouse then it printed after I got the randomness. Thus, this was blocking.



Above is the screen shot for /dev/urandom. This does not block and hence we could quickly see the output for the 1600 bytes.



Above is the same output.

End of Lab-7.