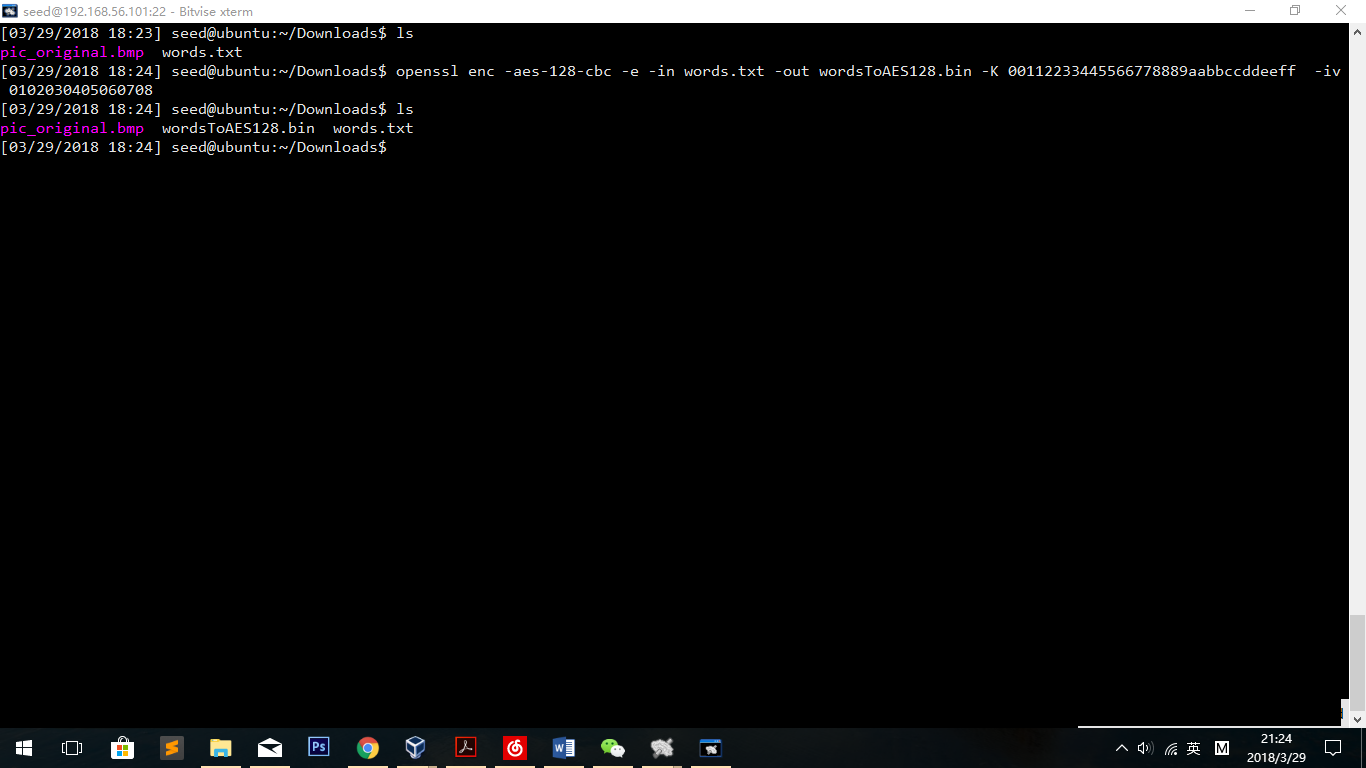
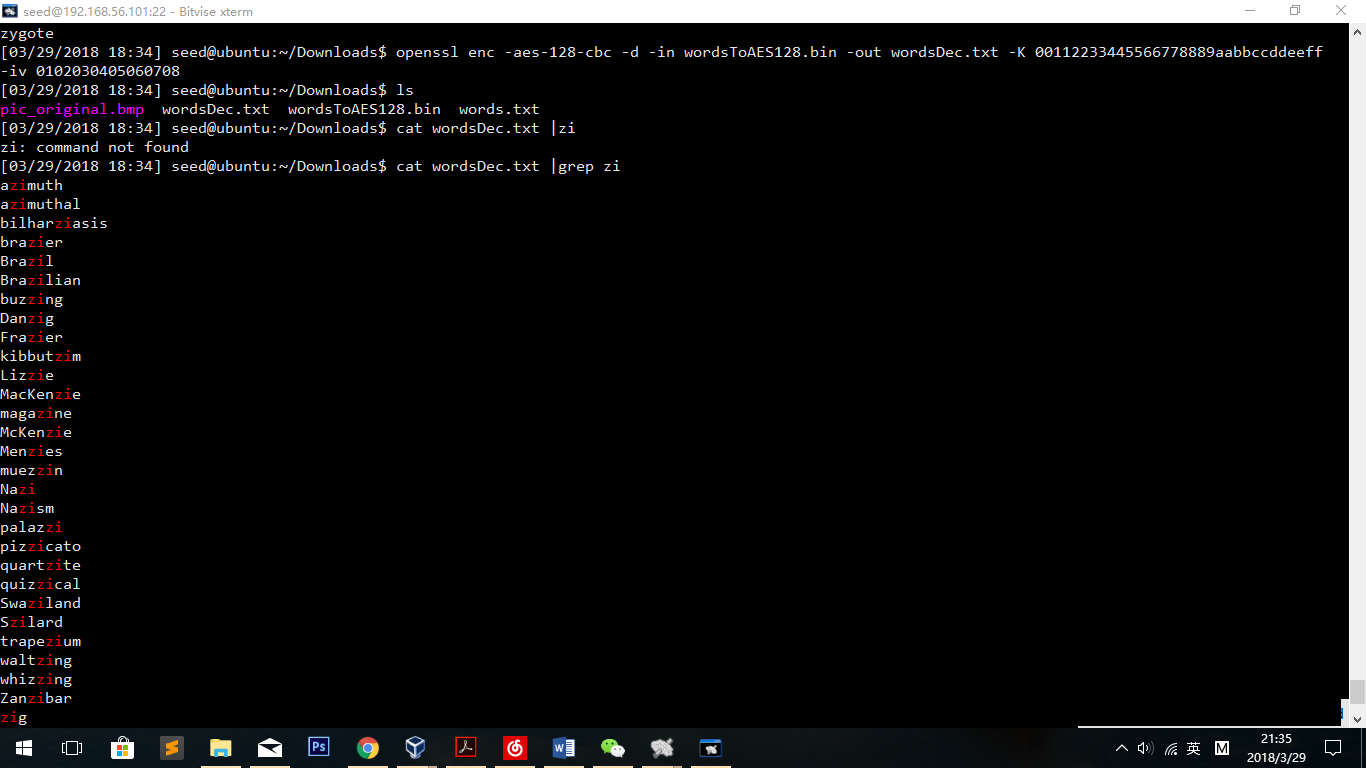
**Report of Lab 6**

**2.1 Task 1: Encryption using different ciphers and modes**

*openssl enc -aes-128-cbc -e -in words.txt -out wordsToAES128.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708*

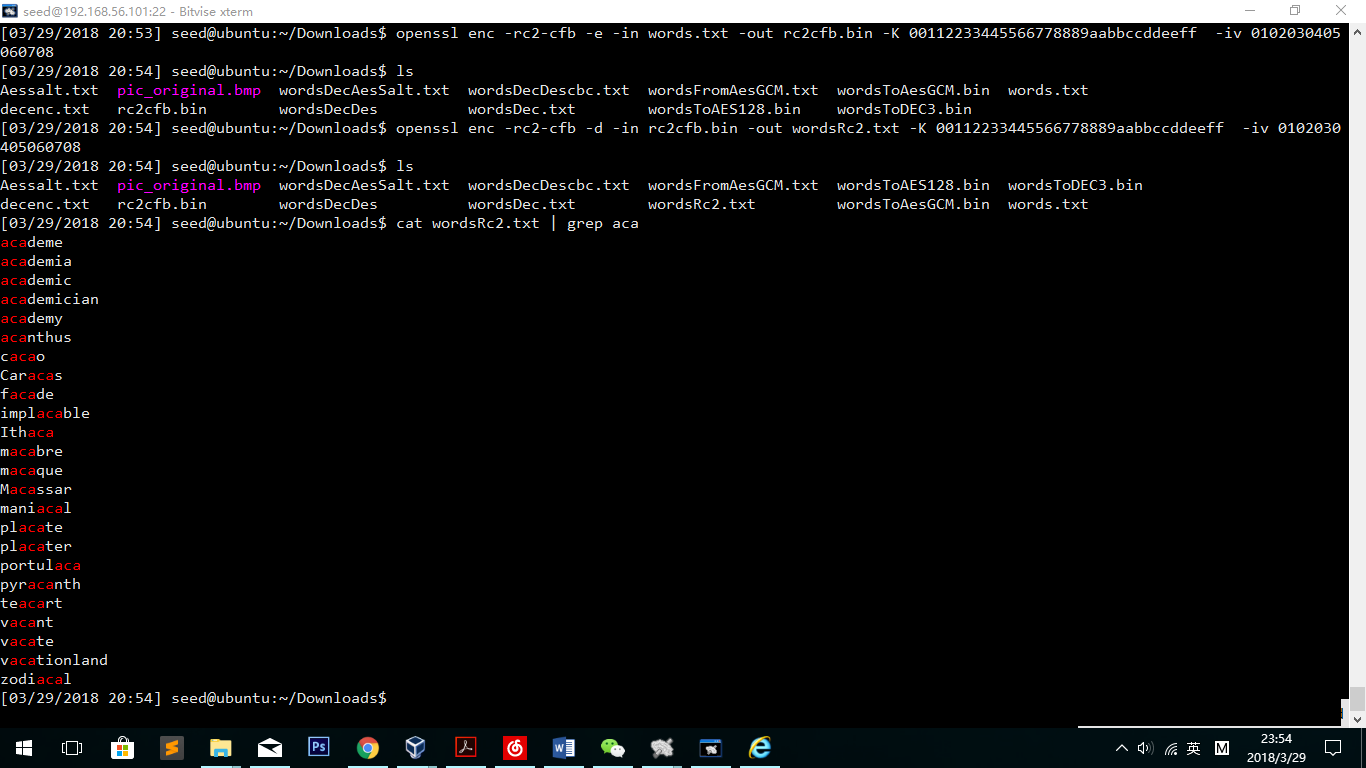


*openssl enc -aes-128-cbc -d -in wordsToAES128.bin -out wordsDec.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708*



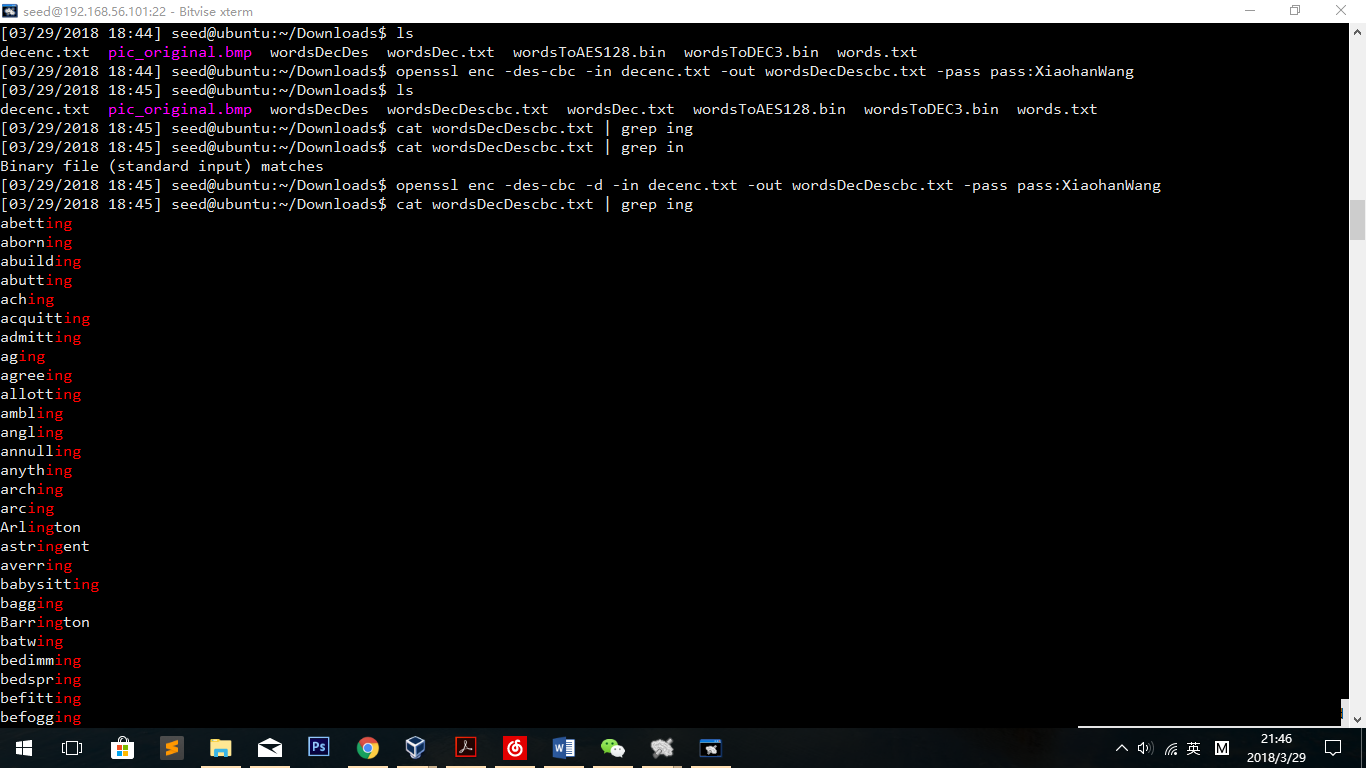
*openssl enc* -rc2-cfb *-e -in words.txt -out rc2cfb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708*

*openssl enc* -rc2-cfb *-d -in rc2cfb.bin -out wordsRc2.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708*



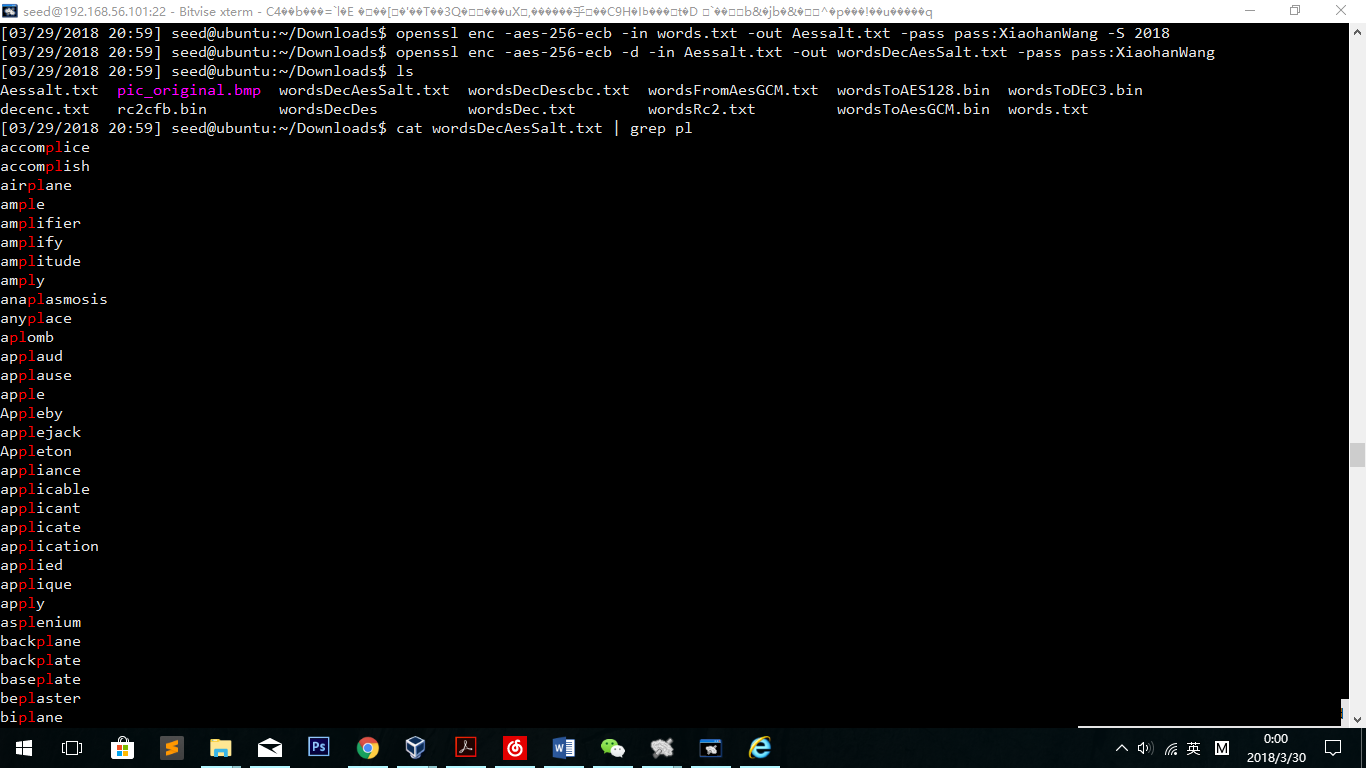
openssl enc -des-cbc -in words.txt -out decenc.txt -pass pass:XiaohanWang

openssl enc -des-cbc -d -in decenc.txt -out wordsDecDescbc.txt -pass pass:XiaohanWang

**

*openssl enc -aes-256-ecb -in words.txt -out Aessalt.txt -pass pass:XiaohanWang -S 2018*

openssl enc -aes-256-ecb -d -in Aessalt.txt -out wordsDecAesSalt.txt -pass pass:XiaohanWang



Observation:

Using different ciphers(aes-128,aes-256,des), modes(ebc, cbc and cfb) and different parameters, all encodes and decodes are successful, no matter whether IV or salt or other parameters given in the command. In the decode files, cat && grep commands show that each decode file is filled with words in the original file “words.txt”.

Explanation:

Without “-e”/”-d” command, the operation will be encryption by default; if salt is unset, salt will be automatically generated by the system; IV is the same as salt; K and pass are conflict, which means that if K is given, pass field will be ignored by the OS.

**2.2 Task 2: Encryption Mode – ECB vs. CBC**

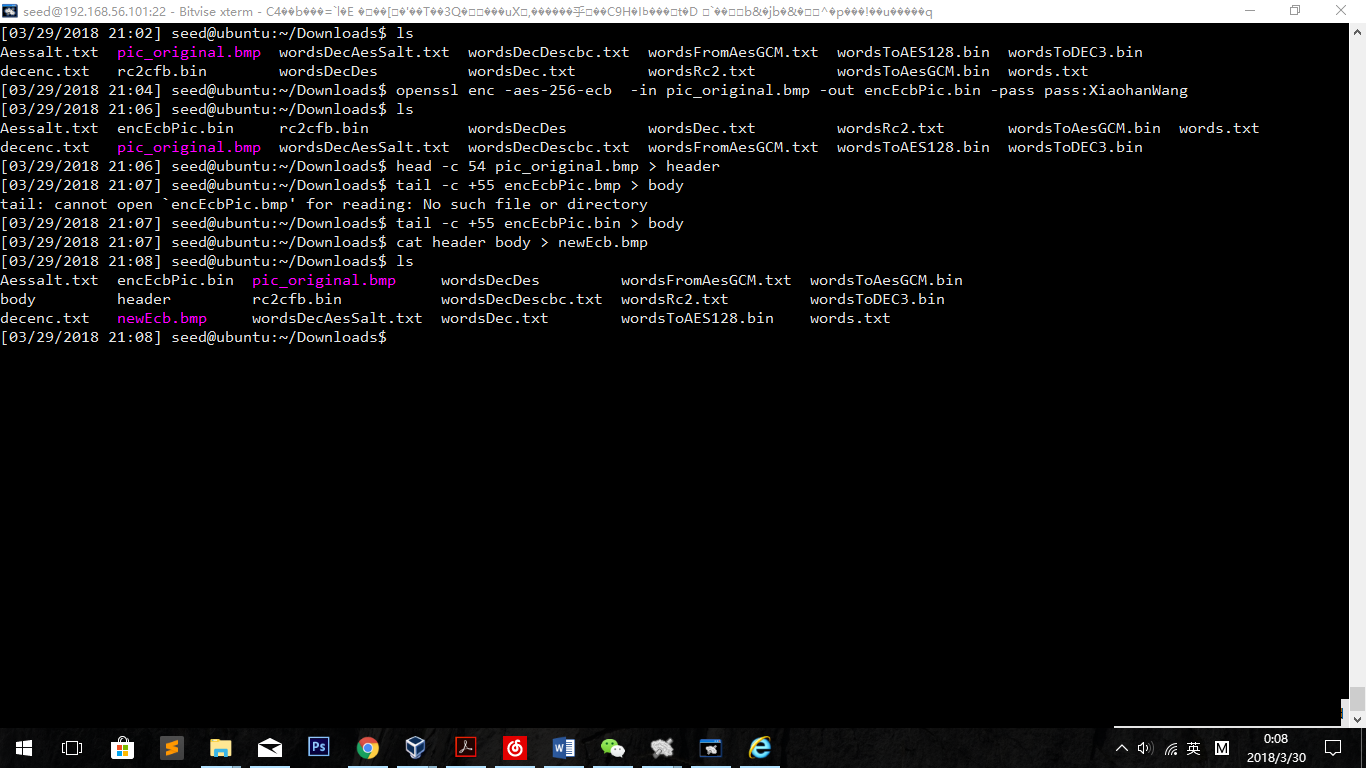
**ECB:**

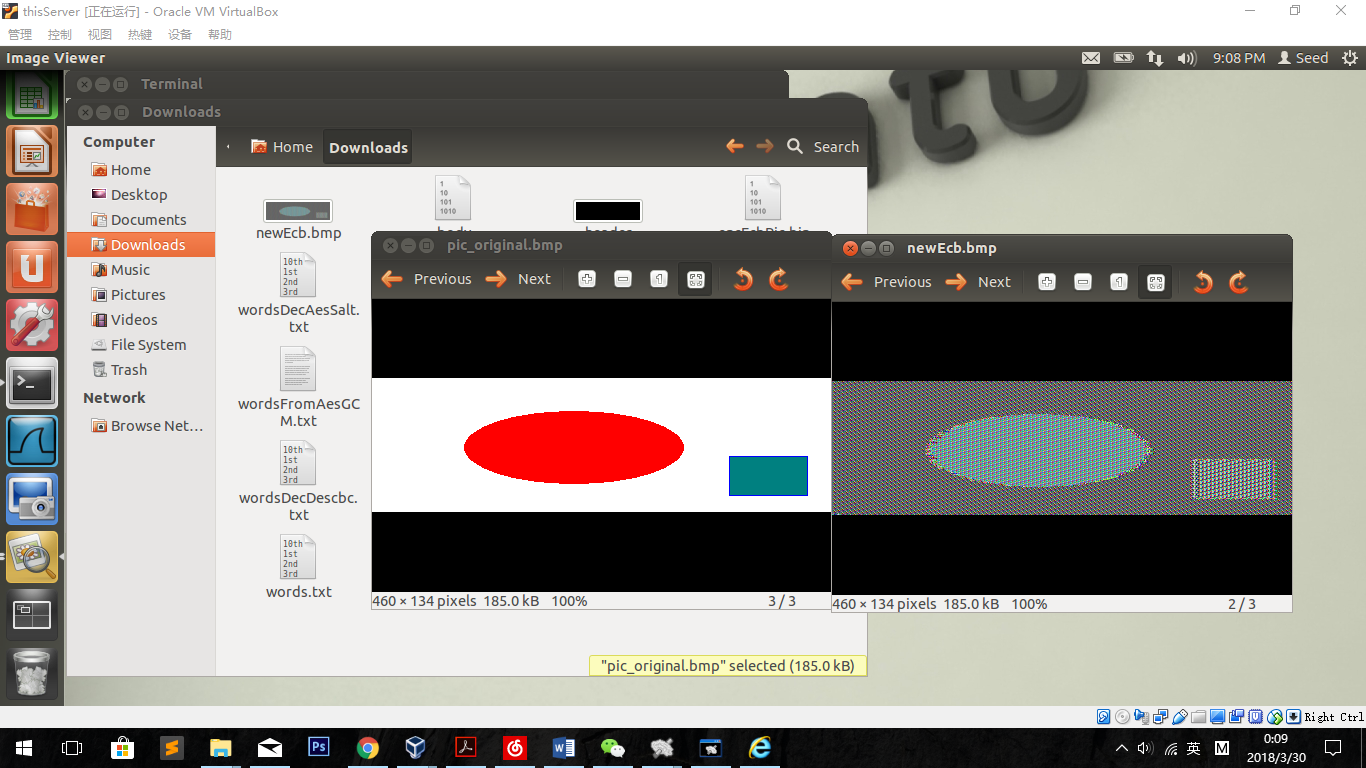
openssl enc *-aes-256-ecb*  -in pic\_original.bmp -out encEcbPic.bin -pass pass:XiaohanWang

$ head -c 54 pic\_original.bmp > header

$ tail -c +55 encEcbPic.bin > body

$ cat header body > newEcb.bmp





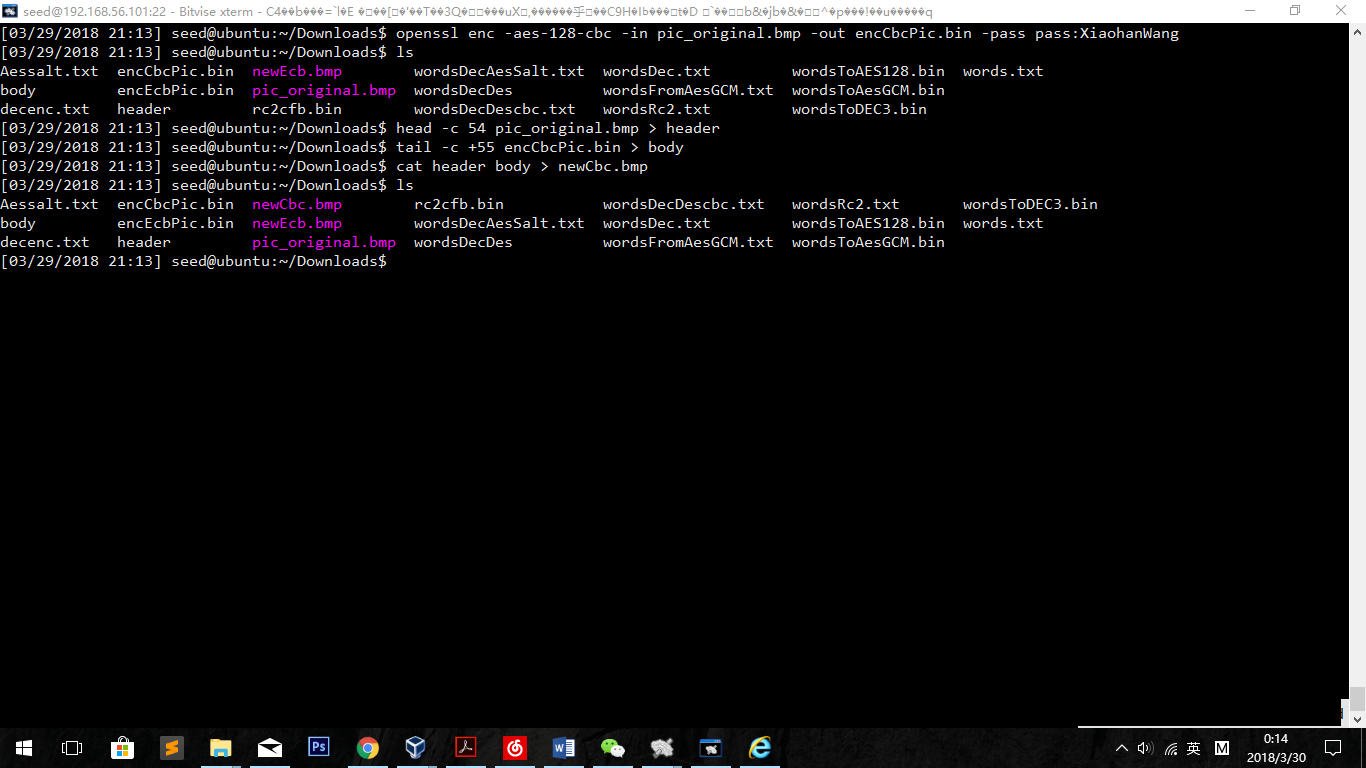
**CBC:**

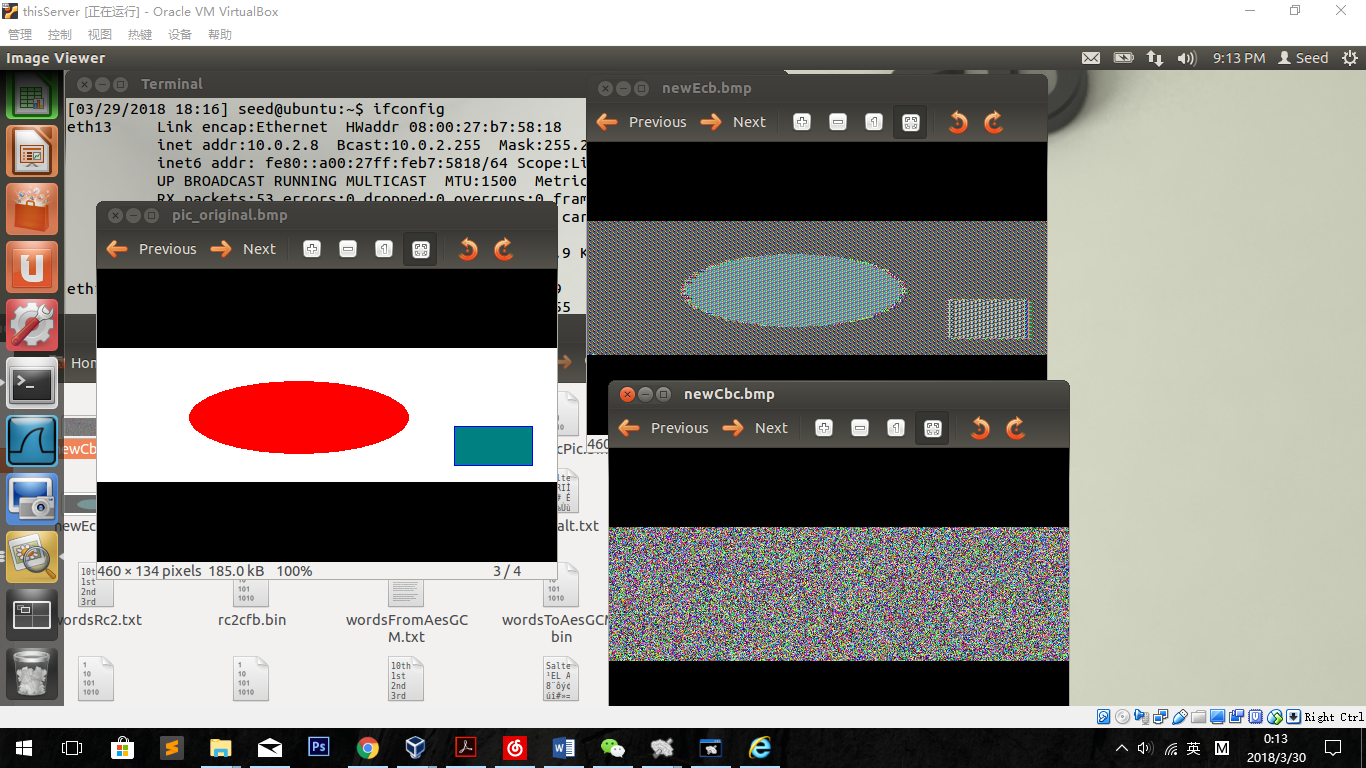
openssl enc -aes-128-cbc -in pic\_original.bmp -out encCbcPic.bin -pass pass:XiaohanWang

$ head -c 54 pic\_original.bmp > header

$ tail -c +55 encCbcPic.bin > body

$ cat header body > newCbc.bmp





Observation:

Obviously, in the ECB encryption, we can see the ellipse and the cube clearly. Although the color has changed and the outline seems to be fuzzy, but we can figure out the shape of the picture.

In the CBC encryption, we can’t get any useful information from the new picture. Comparing to the original one, it has no shape and no similar color information.

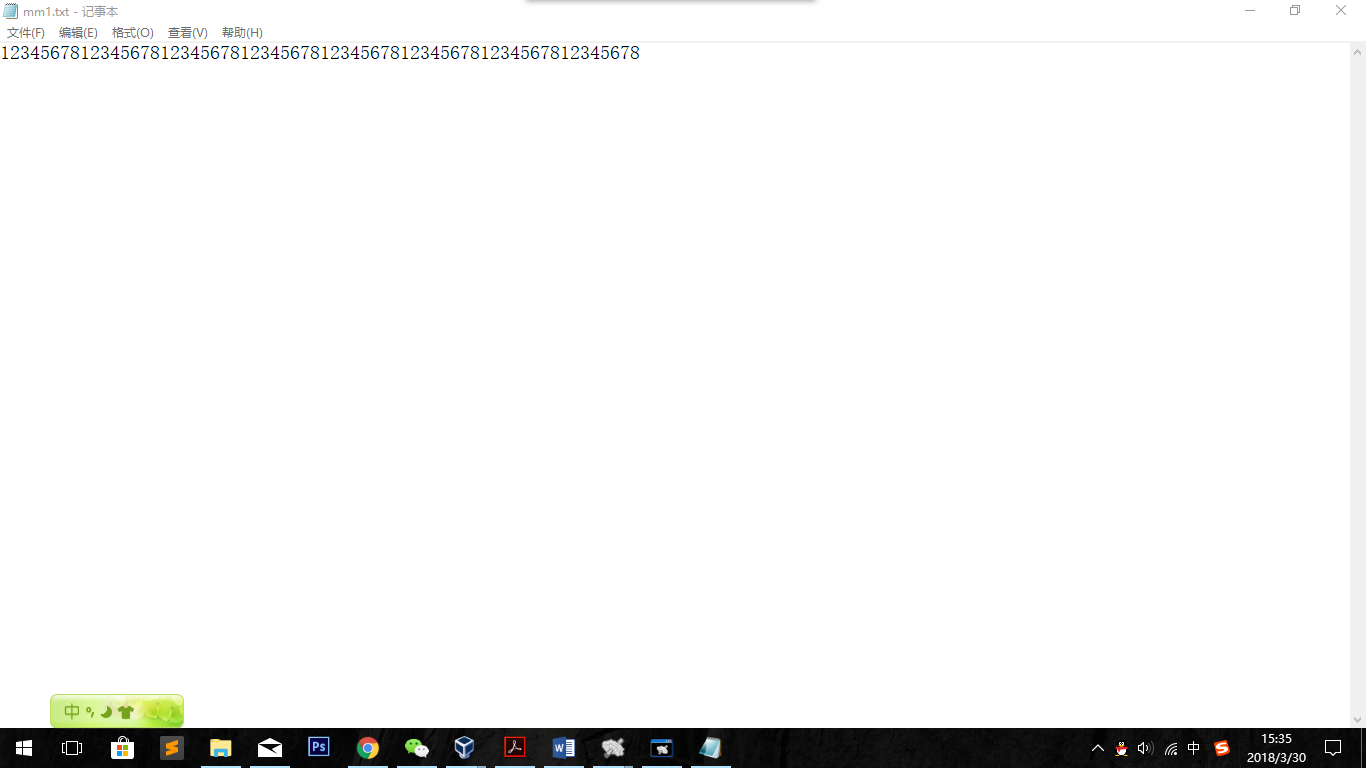
Explanation:

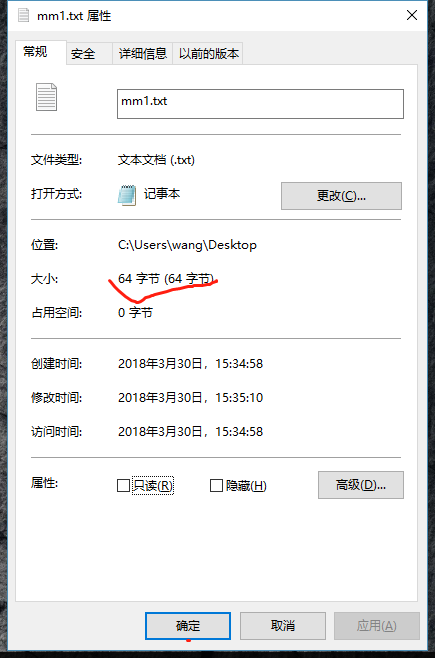
In the ECB encryption, each part of the data is encrypted separately. Although it is not all pixels encrypted separately, the long success same pixels occur many same plaintext. In this case, EBC will treat those same plaintext to the same result, and that is why we can clearly see the pixel changes so that we can see the shapes.

In the CBC encryption, on the other hand, has different result treating same plaintext, because the text is always influenced by formal calculations. Which means, we cannot judge long success same pixels in the encrypted picture.

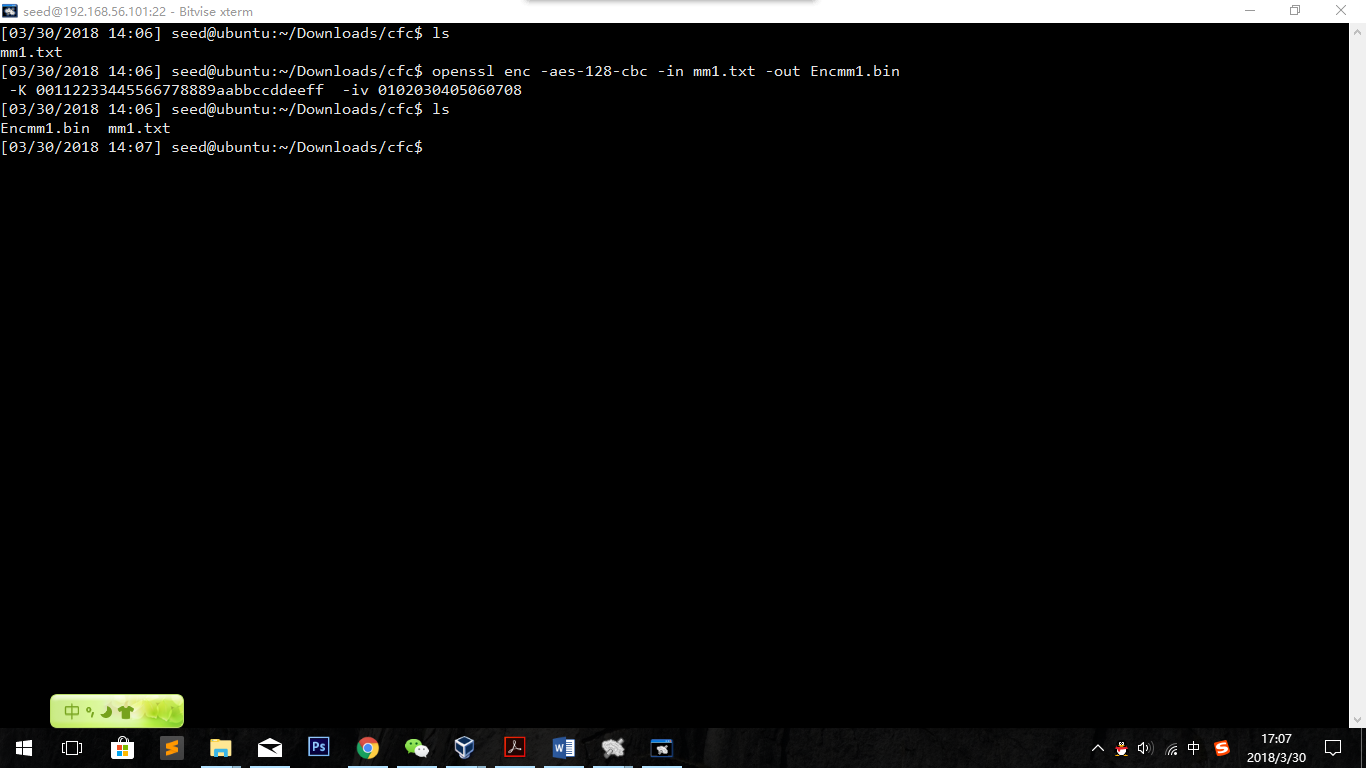
**2.3 Task 3: Encryption Mode – Corrupted Cipher Text**

Create a file of 64byte:

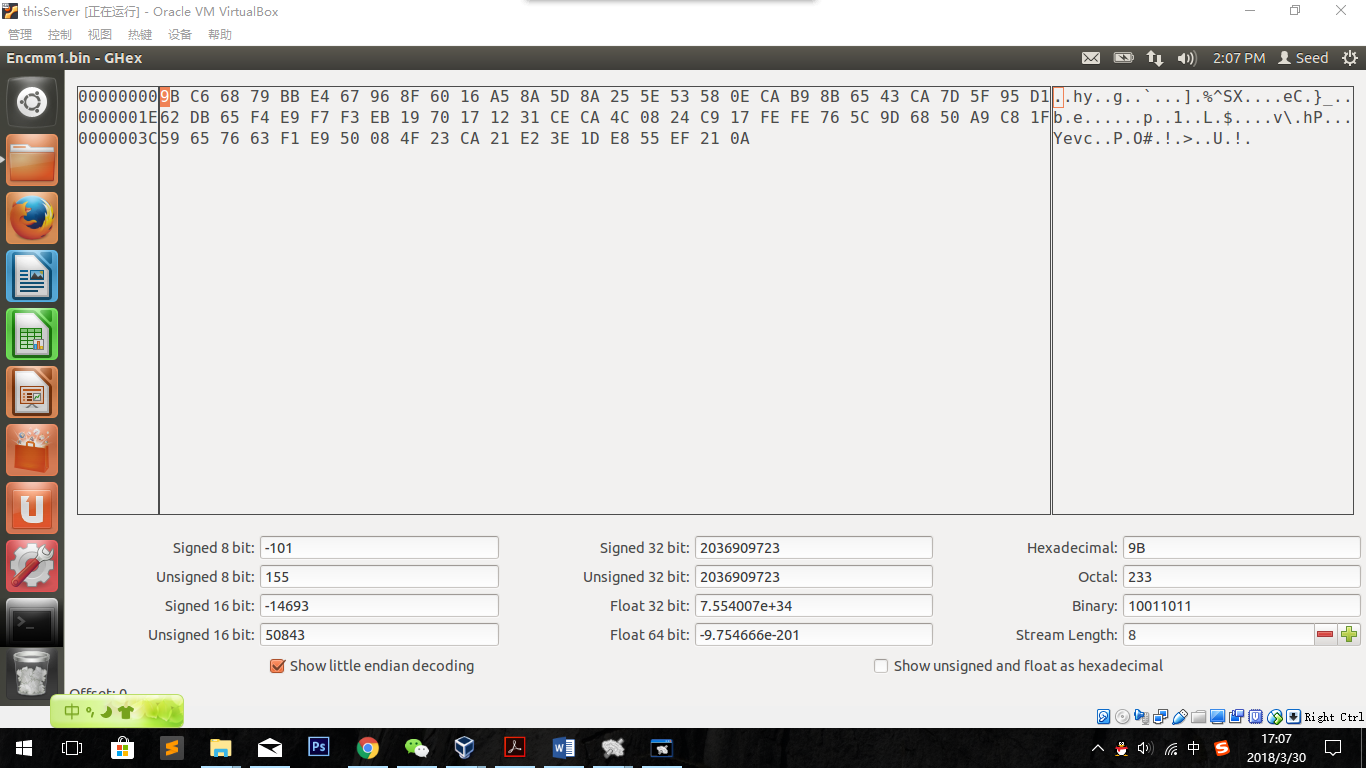




openssl enc -aes-128-cbc -in mm1.txt -out Encmm1.bin *-K 00112233445566778889aabbccddeeff -iv 0102030405060708*

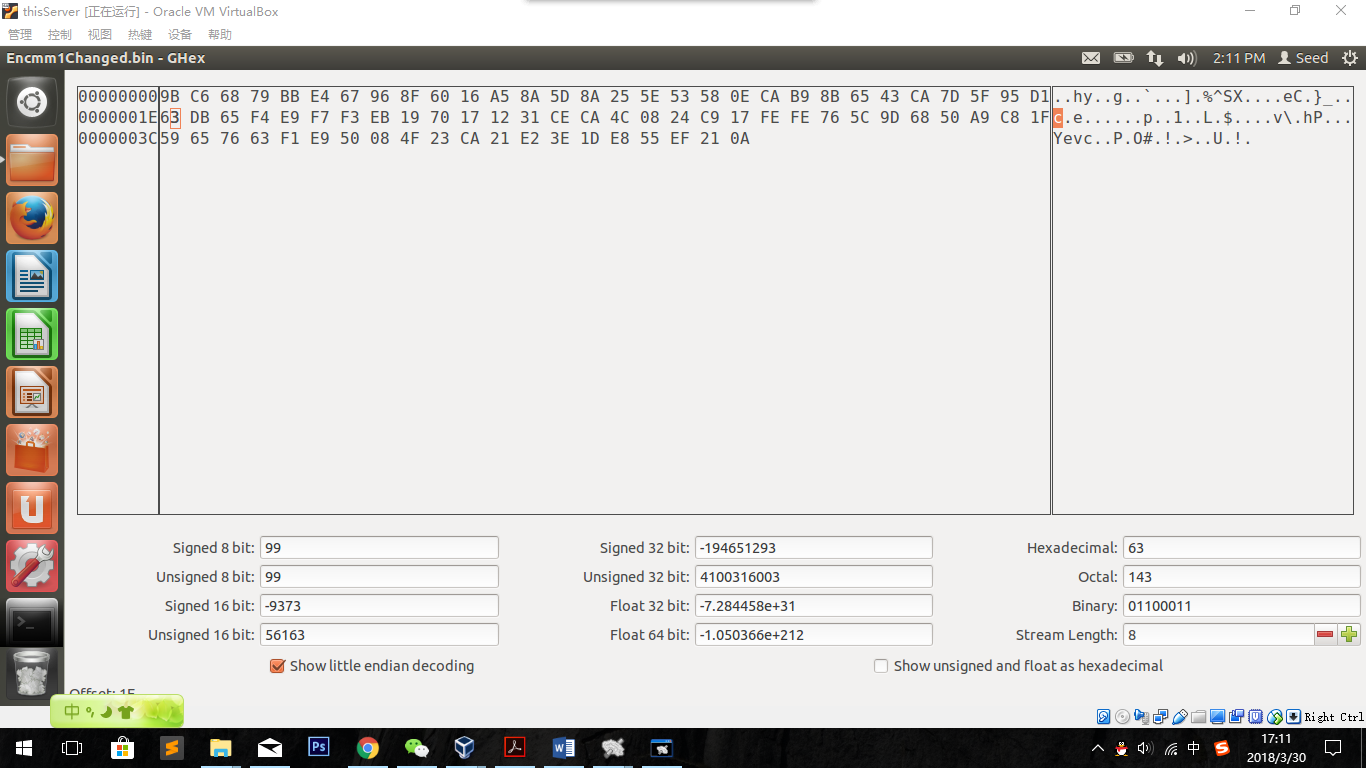


Hex file is like this：



Change the 30th byte file:

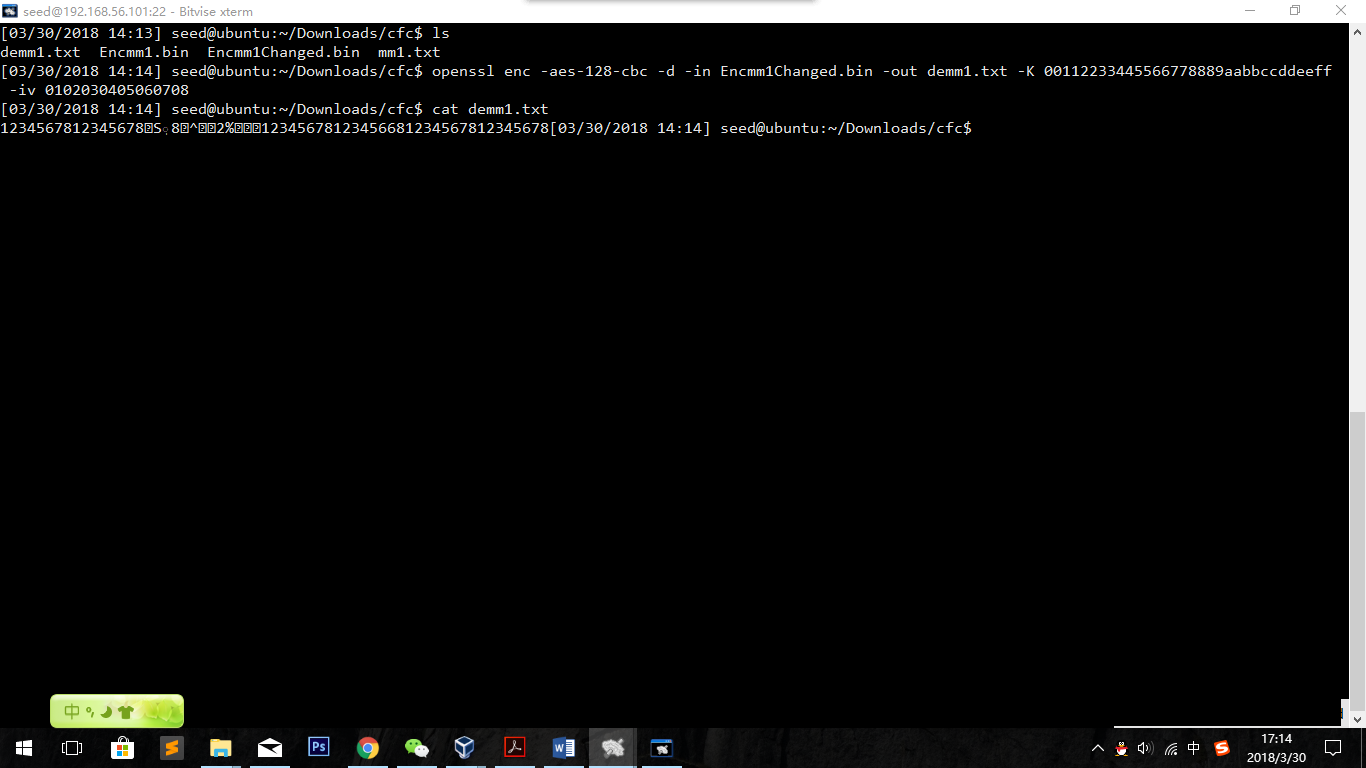
62->63



Save as Encmm1Changed.bin

Decrypt:

openssl enc -aes-128-cbc -d -in Encmm1Changed.bin -out demm1.txt -*K 00112233445566778889aabbccddeeff -iv 0102030405060708*



That is how it works as cbc.

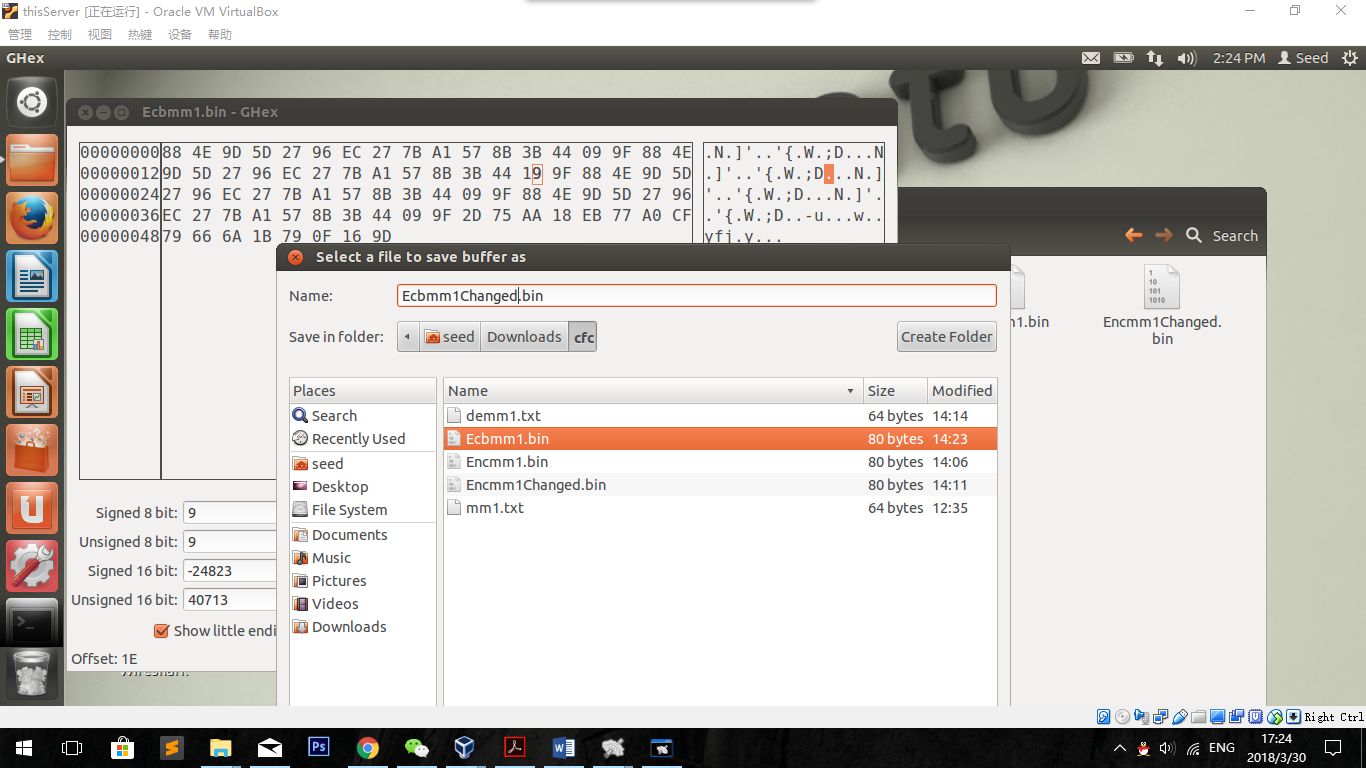
Results of other modes:

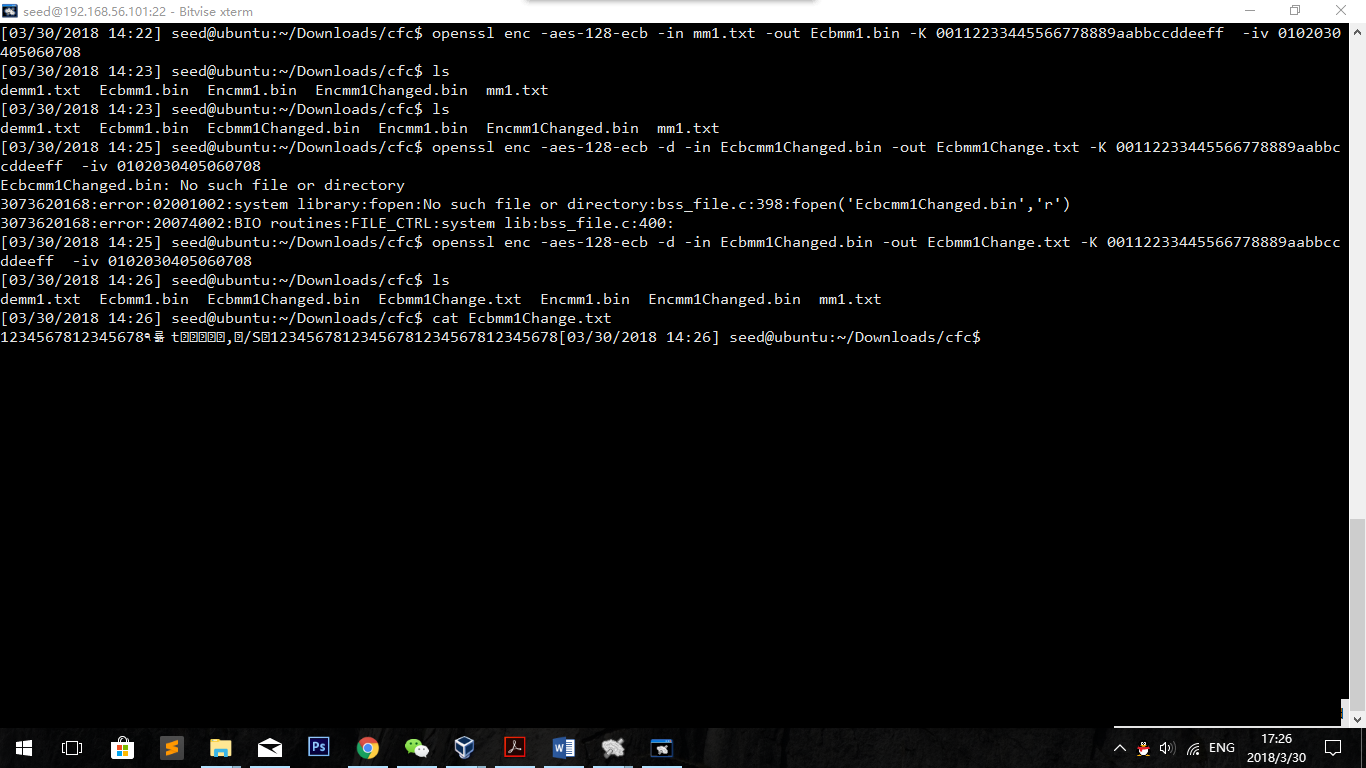
ECB:

openssl enc -aes-128-ecb -in mm1.txt -out Ecbmm1.bin -*K 00112233445566778889aabbccddeeff -iv 0102030405060708*

openssl enc -aes-128-ecb -d -in Ecbmm1Changed.bin -out Ecbmm1Change.txt -*K 00112233445566778889aabbccddeeff -iv 0102030405060708*

*0->1*



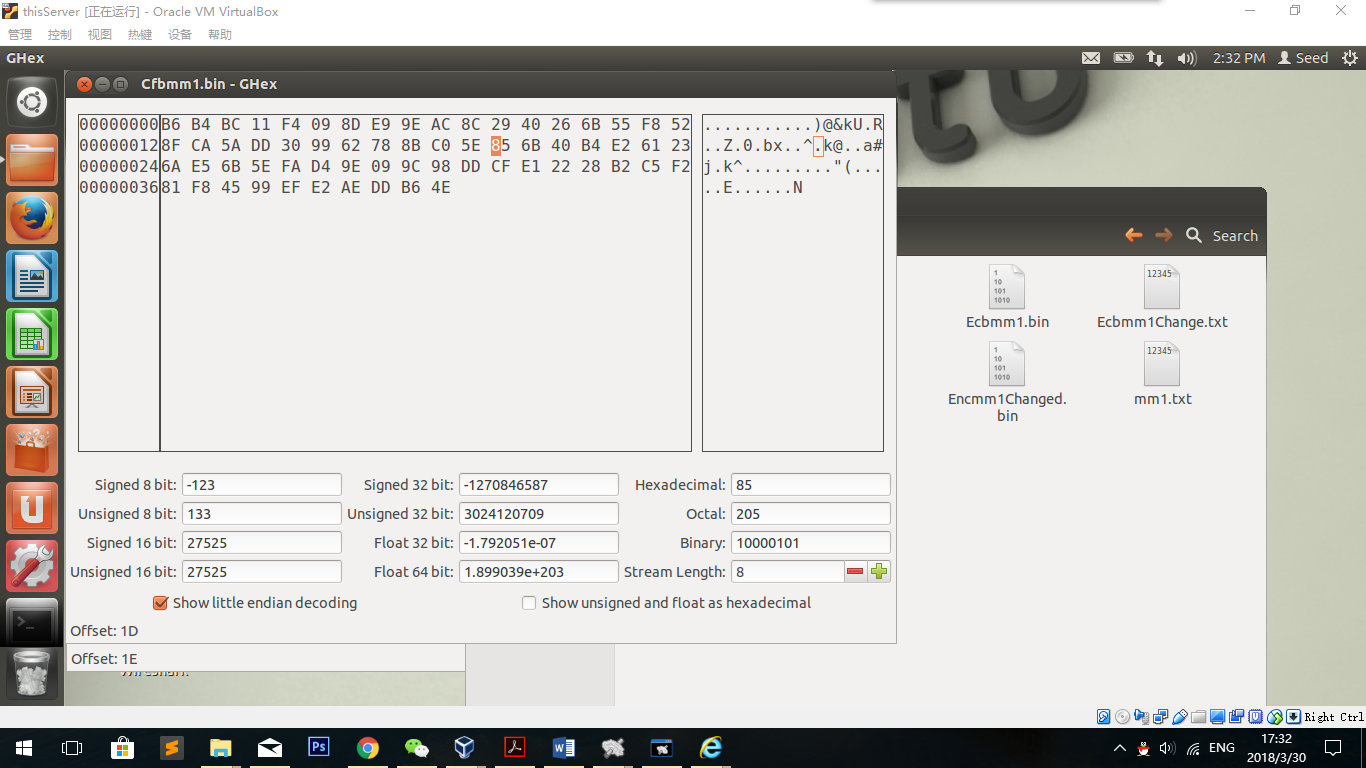


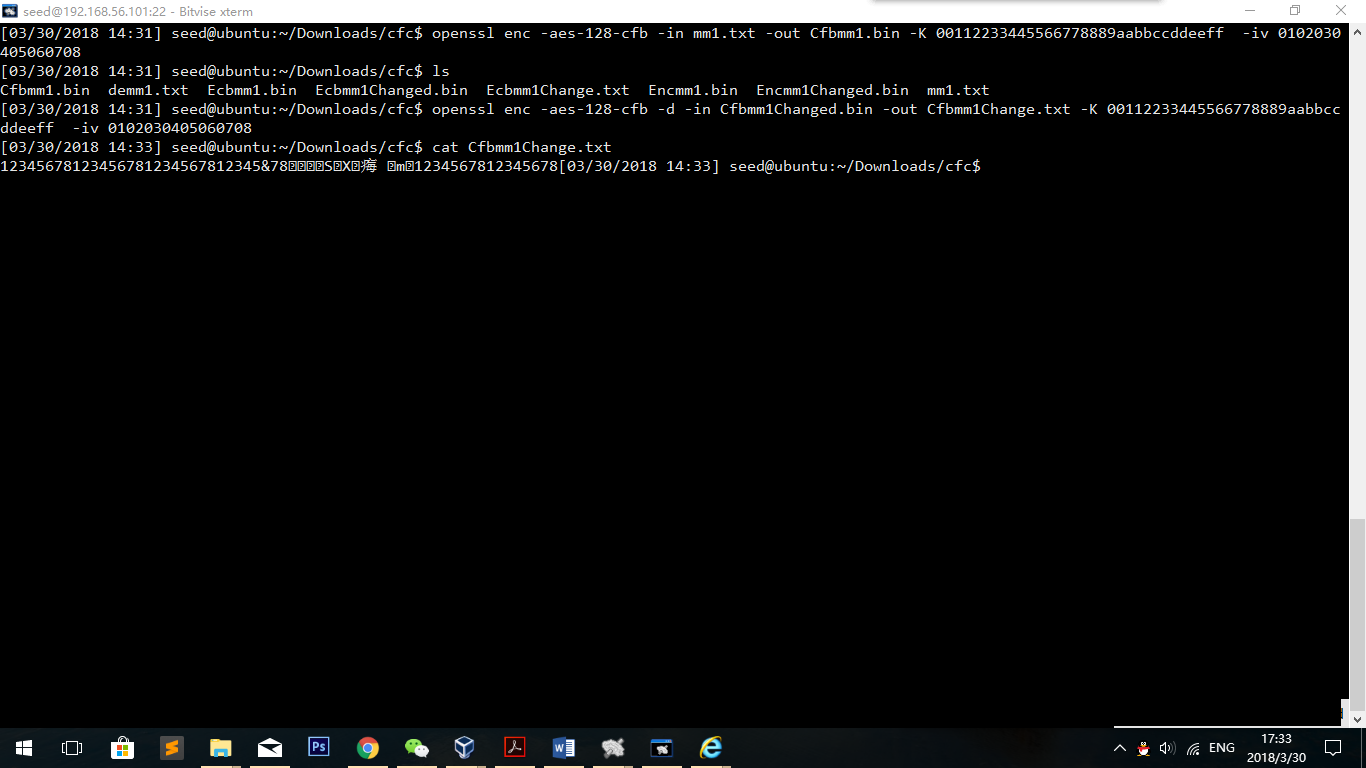
CFB:

openssl enc -aes-128-cfb -in mm1.txt -out Cfbmm1.bin -*K 00112233445566778889aabbccddeeff -iv 0102030405060708*

openssl enc -aes-128-cfb -d -in Cfbmm1Changed.bin -out Cfbmm1Change.txt -*K 00112233445566778889aabbccddeeff -iv 0102030405060708*

9->8



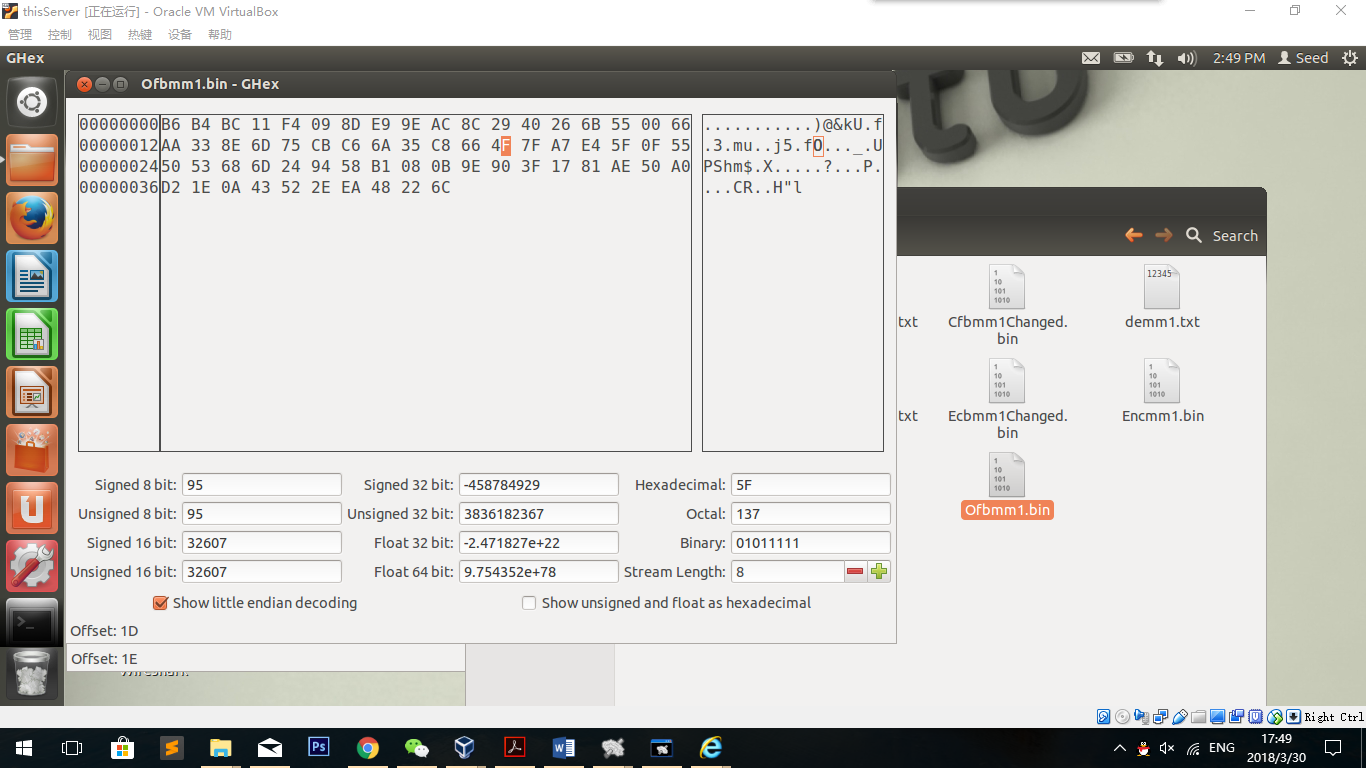


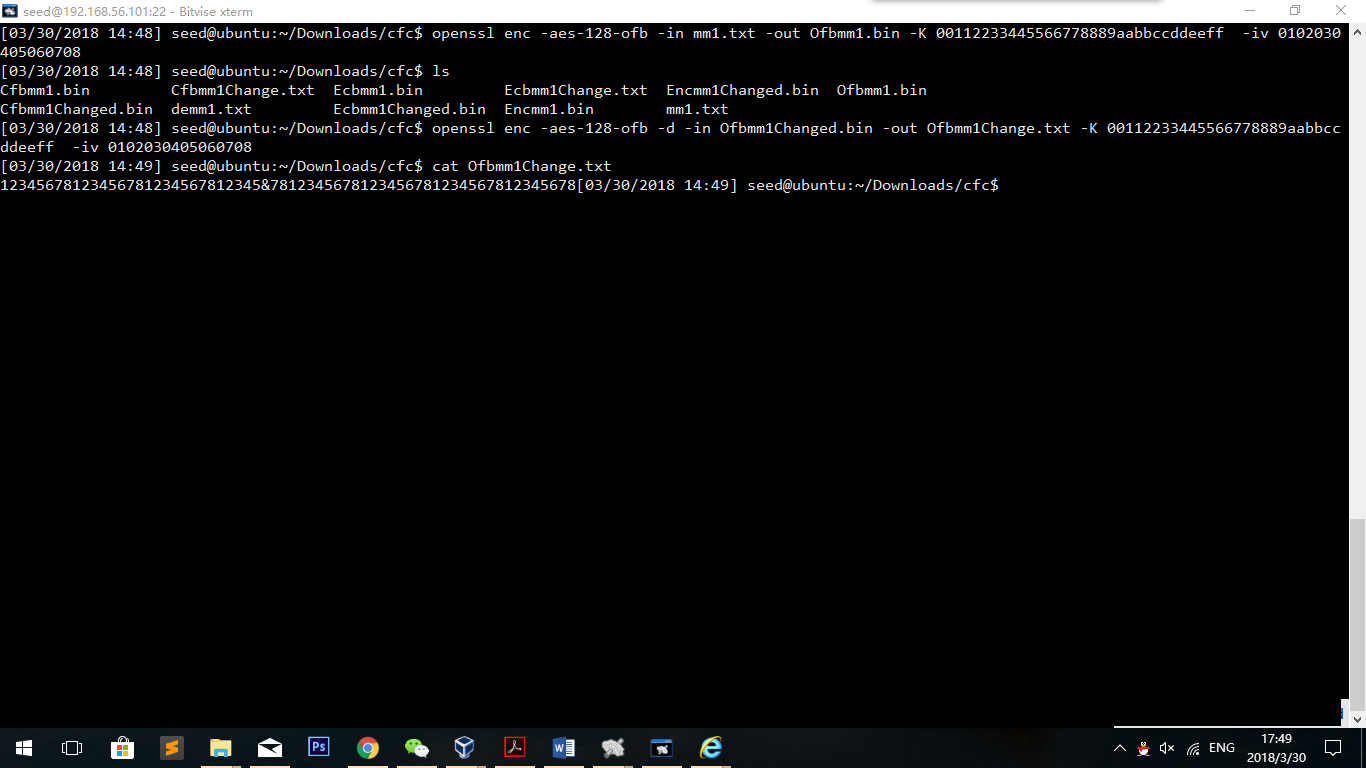
OFB:

openssl enc -aes-128-ofb -in mm1.txt -out Ofbmm1.bin -*K 00112233445566778889aabbccddeeff -iv 0102030405060708*

openssl enc -aes-128-ofb -d -in Ofbmm1Changed.bin -out Ofbmm1Change.txt -*K 00112233445566778889aabbccddeeff -iv 0102030405060708*

5->4





**Observation:**

CBC: After 16 bytes, there comes messy codes. It seems like after some bytes it comes to normal, but there is a string changed except for the 2 messed groups (12345678), which is: 12345678 after the messy codes has been changed to 12345668.

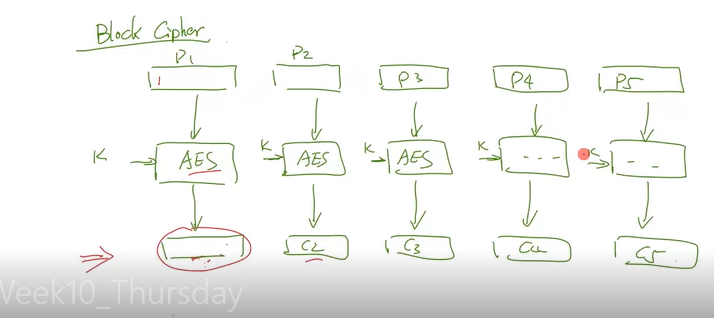
EBC: As well, after 16 bytes, 2 groups become messy codes. However, it does not change any of the following groups.

CFB: The first 29 bytes are not changed. The 30th byte has changed and next two groups have become messy codes.

OFB: Literally, there is only one byte changed, which is the 30th byte.

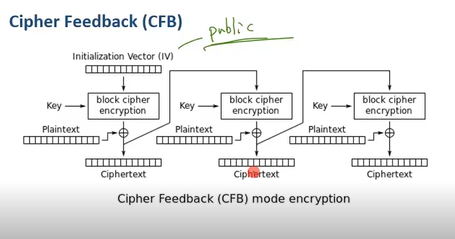
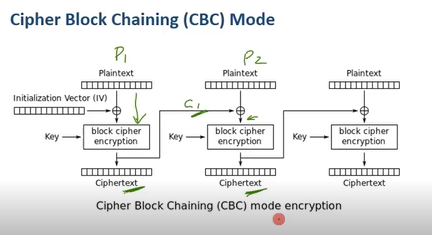
**Explanation:**

In EBC ,



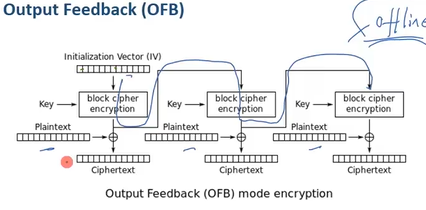
Calculations are only influenced in the current block (whose size is 16 bytes). That is why only those in the 3rd and 4th group of (1-8) are influenced.

In CFBand CBC,



Ciphers influenced the later calculation, that is why some later text become wrong or messy codes.

But in OFB,



Ciphers are not used for the later calculations, and the change only influences the xor operation, so it only effects one bit.

**(1) 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.**

Before: I think the first block of all decode text will not change. In ECB it influences only one block. And in CBC and CFB it influences all later bytes. OFB is only changed by one byte.

After: Result are above. I find most of the guesses are right, but I didn’t expect that in CBC and CFB the influence is such small.

**(2) Please explain why.**

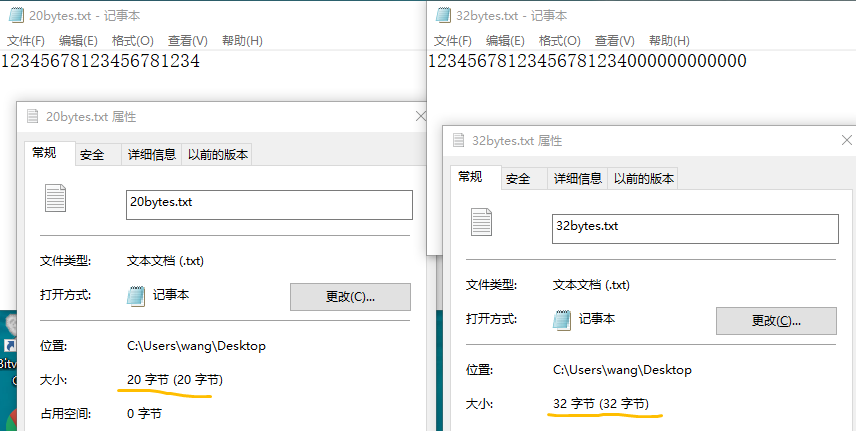
Explanations are above.

**(3) What are the implication of these differences?**

While transporting information from one place to another, mistakes cannot be avoided. In this case, the differences of those change of encryption can tell which packets are correct while finding a transportation mistake. It is important to know which ones can be used to decrease punishment of the mistake.

**2.4 Task 4: Padding**

Create 2 files with lengths of 20 bytes and 32 bytes.

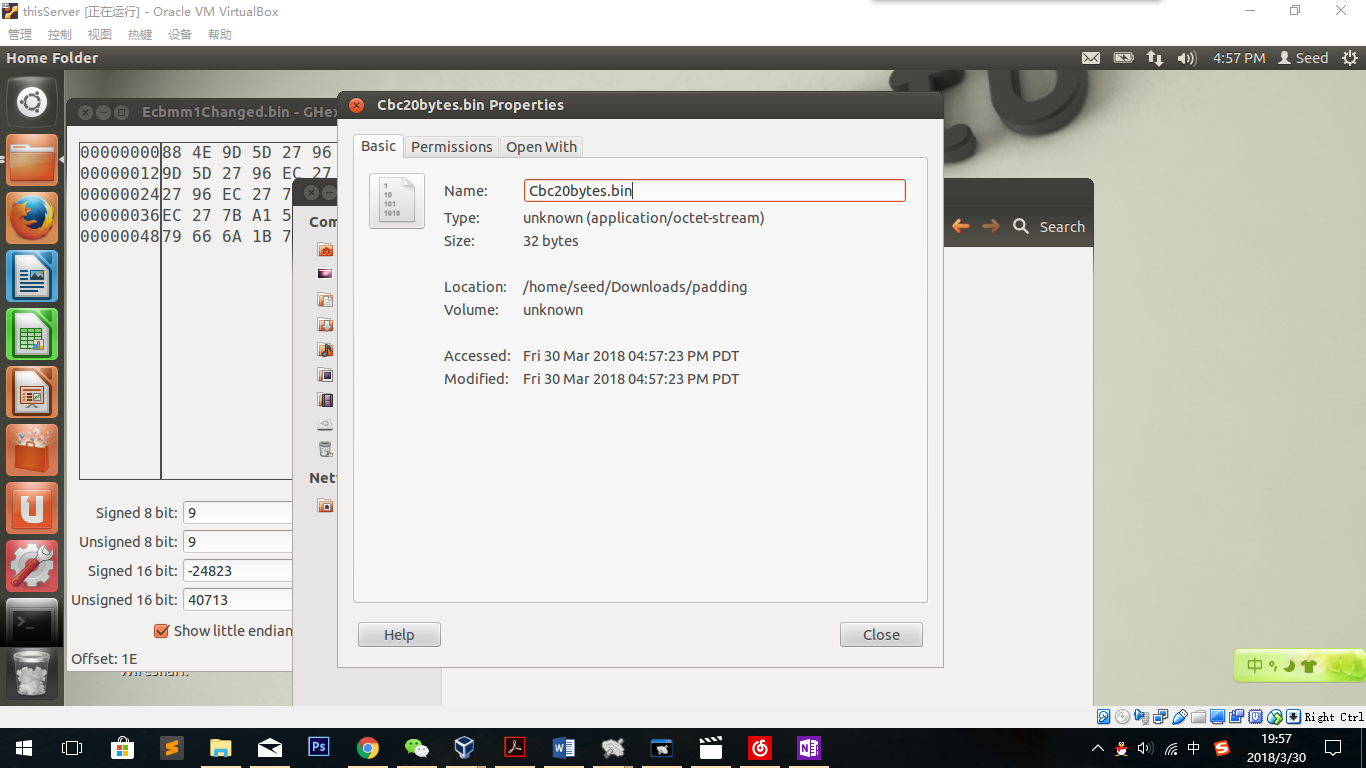


**Experiment design:**

I will use the characteristics to verify the PKC5. If a file’s length is not divided by the block size, then it will pad until it fits the block size; but if it is divided by the block size, the padding goes like append an extra space of block size.

In this case, I am going to test the two characteristics.

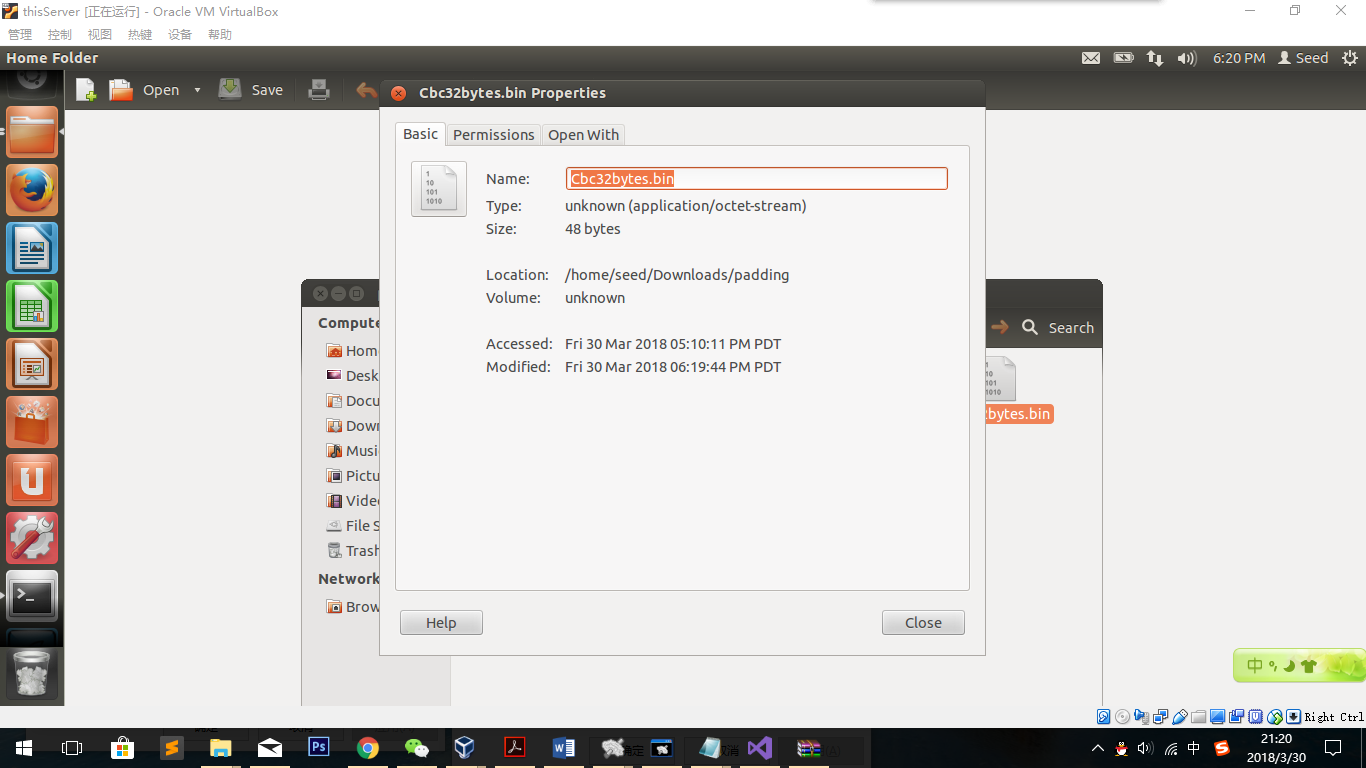
*openssl enc -aes-128-cbc -in 20bytes.txt -out Cbc20bytes.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708*



The padding is working! And we can also see the size of the cipher text is 32 bytes, which means the block size is 16 not 8. (If is 8, the size should be 24)

Then we will encrypt the 32 bytes file:

*openssl enc -aes-128-cbc -in 32bytes.txt -out Cbc32bytes.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708*



We find that the encrypt file size is 48 bytes, which means it appends an extra field because 16 divides 32. That means it uses PKC 5 as default.

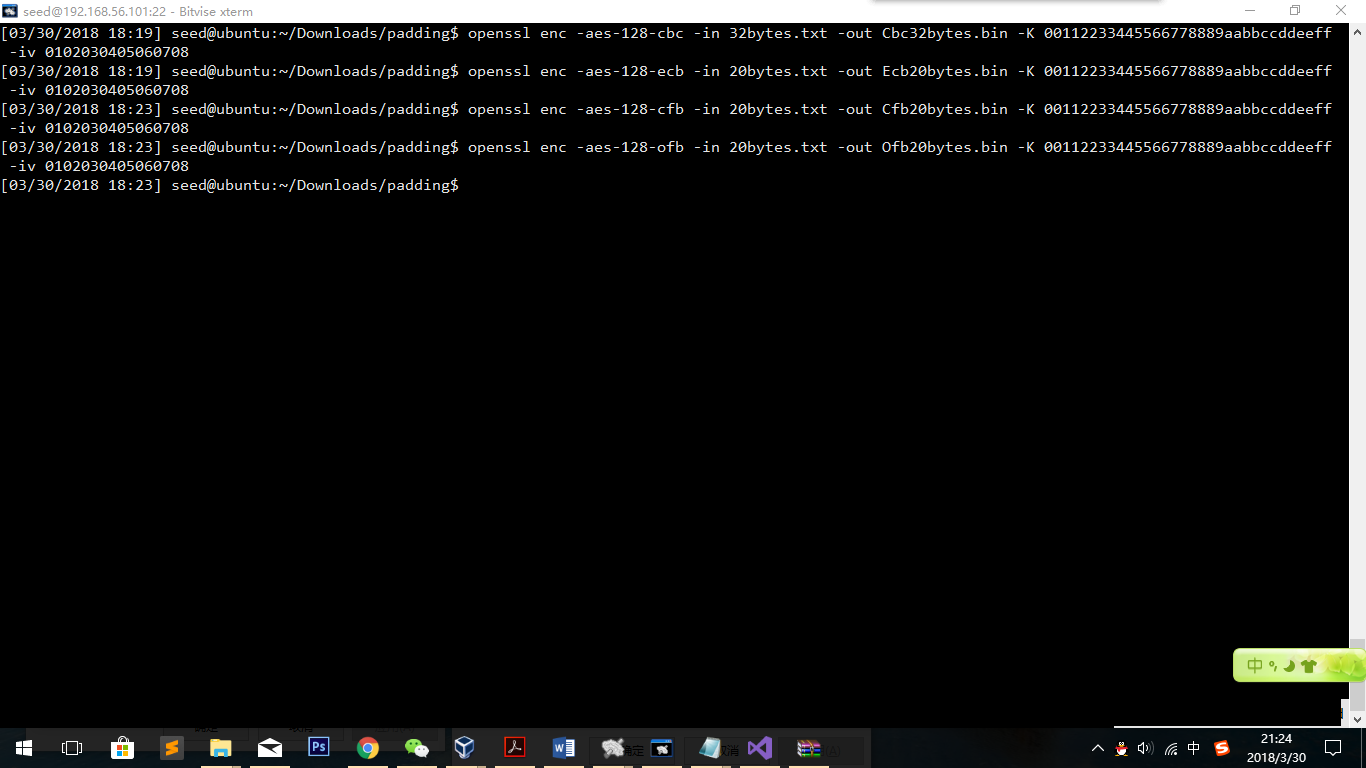
Different modes(Encrypt 20-byte file):

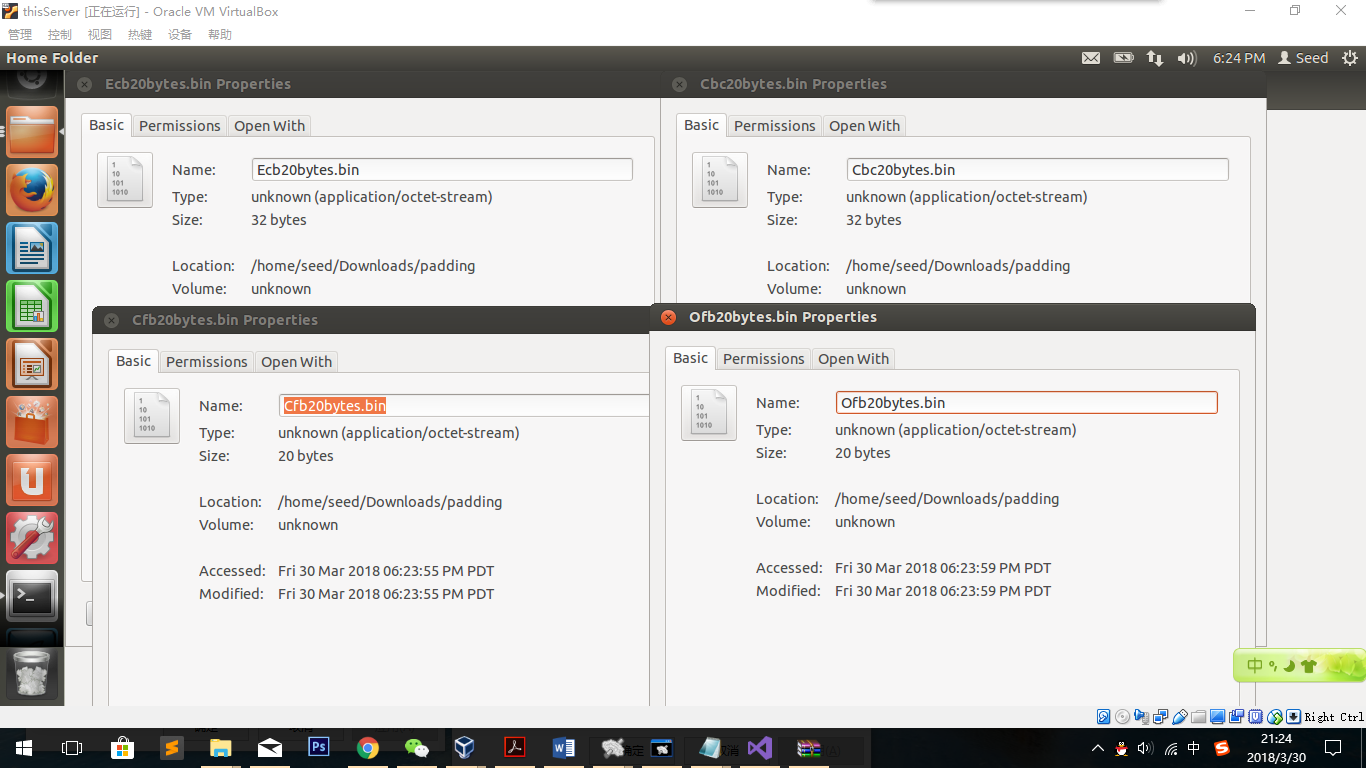
*openssl enc -aes-128-ecb -in 20bytes.txt -out Ecb20bytes.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708*

*openssl enc -aes-128-cbc -in 20bytes.txt -out Cbc20bytes.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708*

*openssl enc -aes-128-cfb -in 20bytes.txt -out Cfb20bytes.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708*

*openssl enc -aes-128-ofb -in 20bytes.txt -out Ofb20bytes.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708*



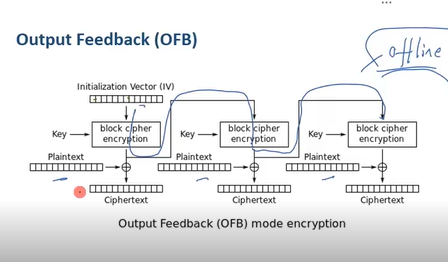
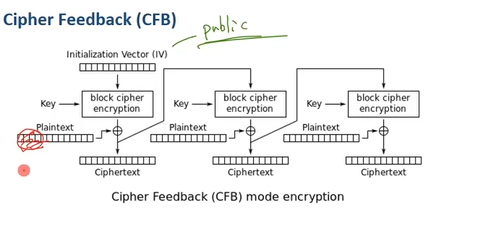


**Observation:**

ECB and CBC files have become 32 bytes and the others are still 20 bytes, which means cfb and ofb don’t have padding.

**Explanation:**

In cfb and ofb:

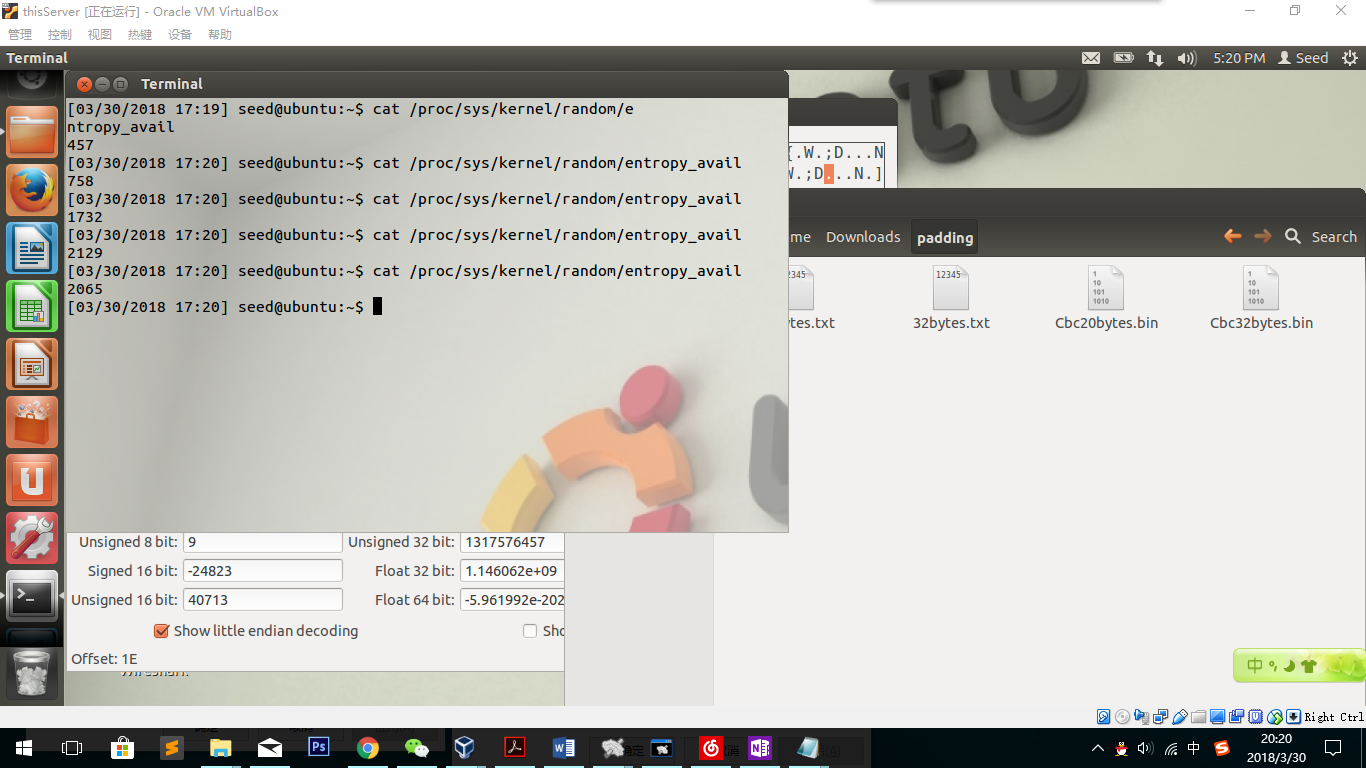


Plaintexts are only used to xor to ciphers without doing the encryption itself, so they don’t need padding.

2.5 Task 5: Pseudo Random Number Generation

Task 5.A: Measure the Entropy of Kernel

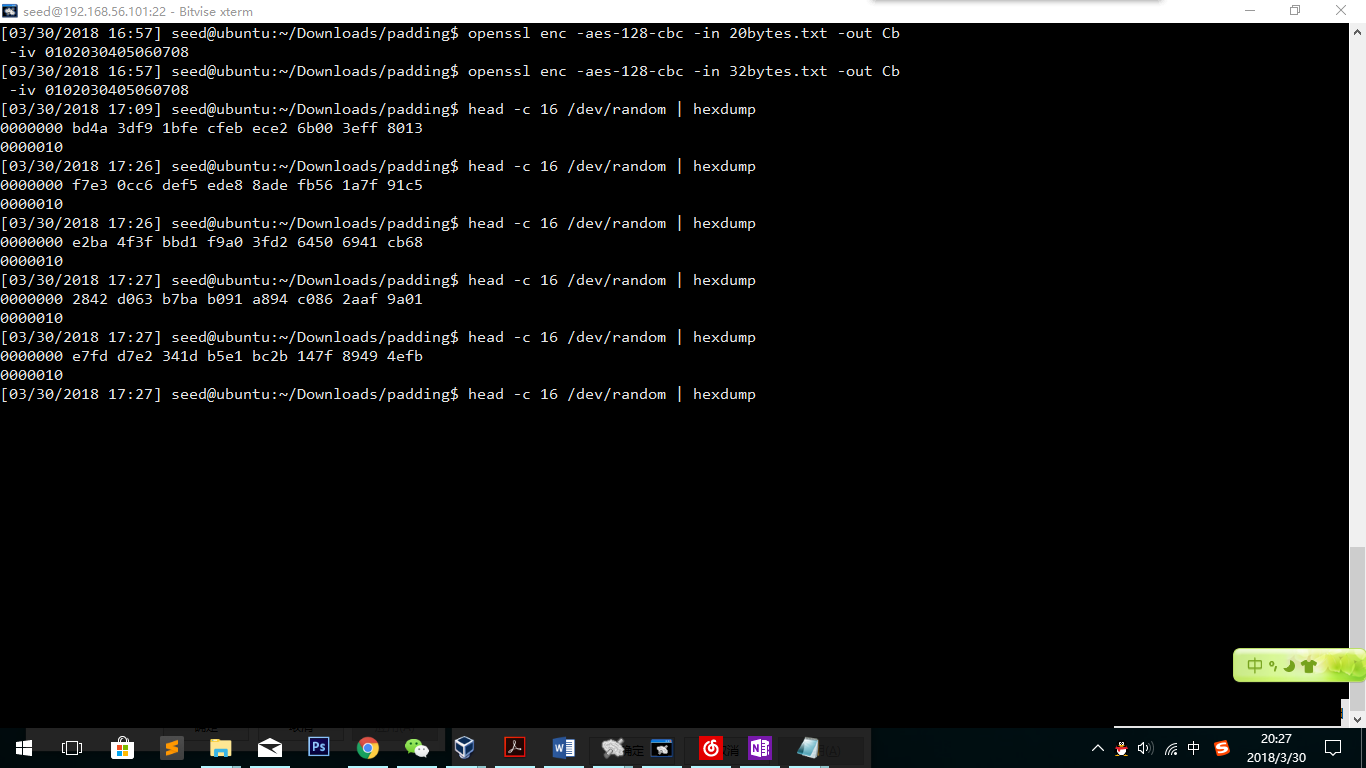
*cat /proc/sys/kernel/random/entropy\_avail*



Observation:

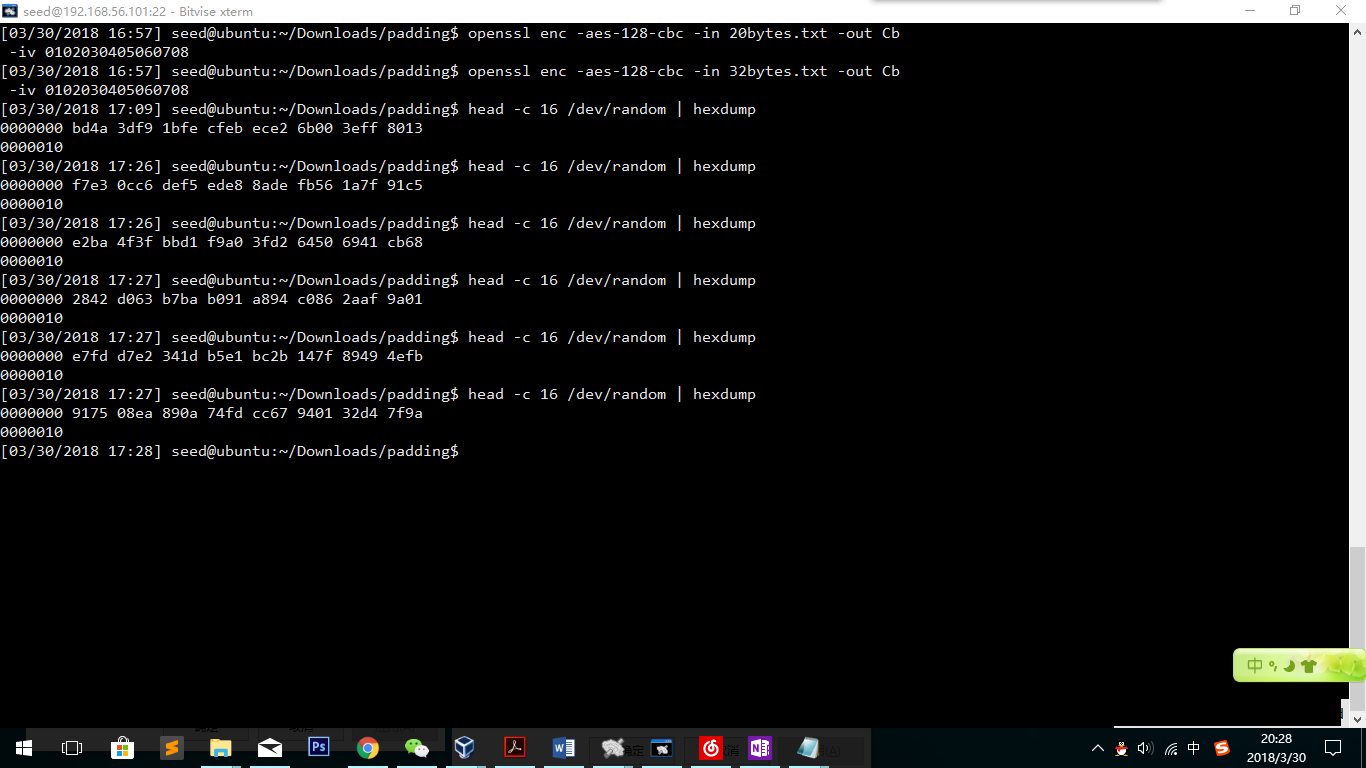
Every time I make some movement and redo the instruction again, the values it returns are different. This is because my mouse movement is creating randomness for the system. I also notice that the value does not always gets larger and it gets smaller for some reason. I guess the value is collected in the specific time period. Which means, if I slow down my movement and do less other operations like click, the value may become smaller.

**Task 5.B: Get Pseudo Random Numbers from /dev/random**



After running the command several times, the terminal seems blocked. Because the pool of randomness is decreased to 0.

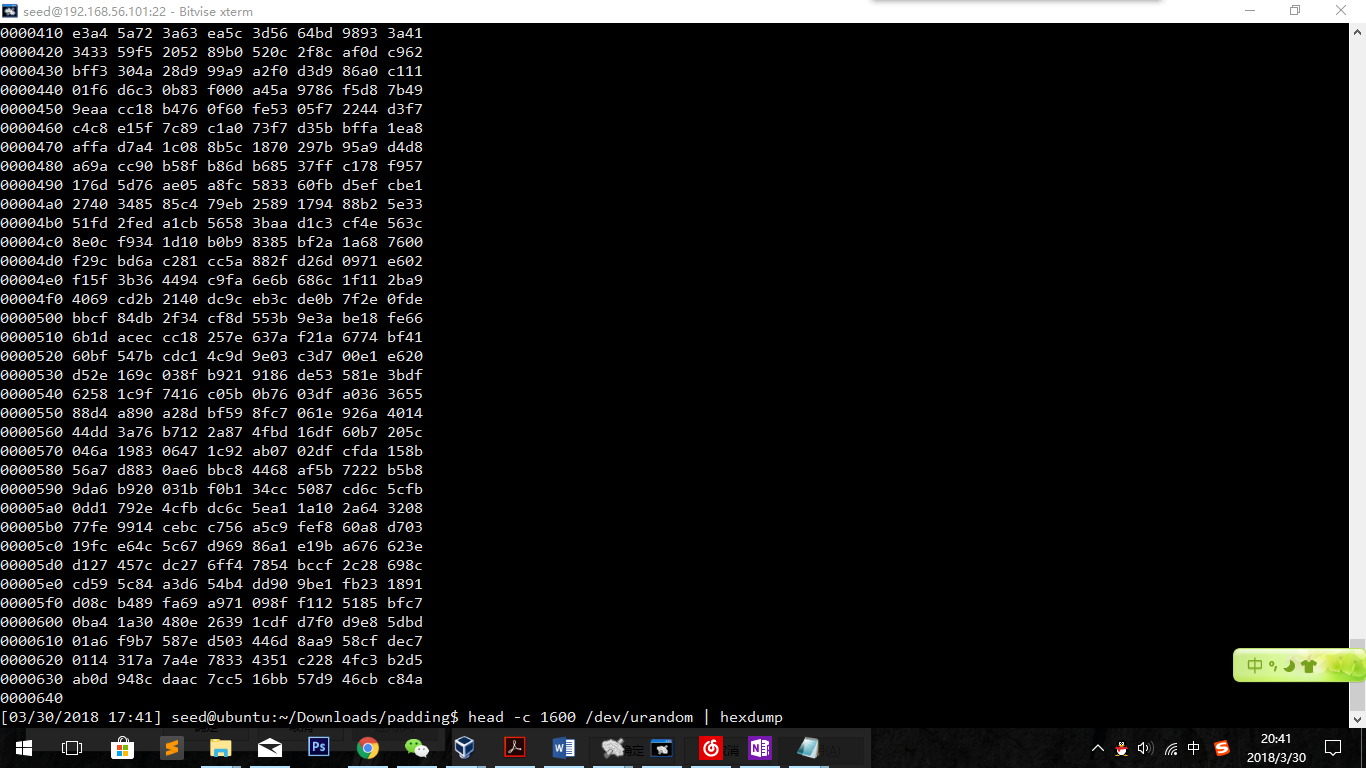
But when I switch back to the virtual machine window and make movements of my mouse, I find it gets unblock quickly.



This is because the randomness is increased and there are enough random numbers for output.

**Task 5.C: Get Random Numbers from /dev/urandom**

head -c 1600 /dev/urandom | hexdump



**Observation:**

I can get enough random numbers via urandom. I tried several times and it does not block once.

**Task 6: Programming using the Crypto Library**

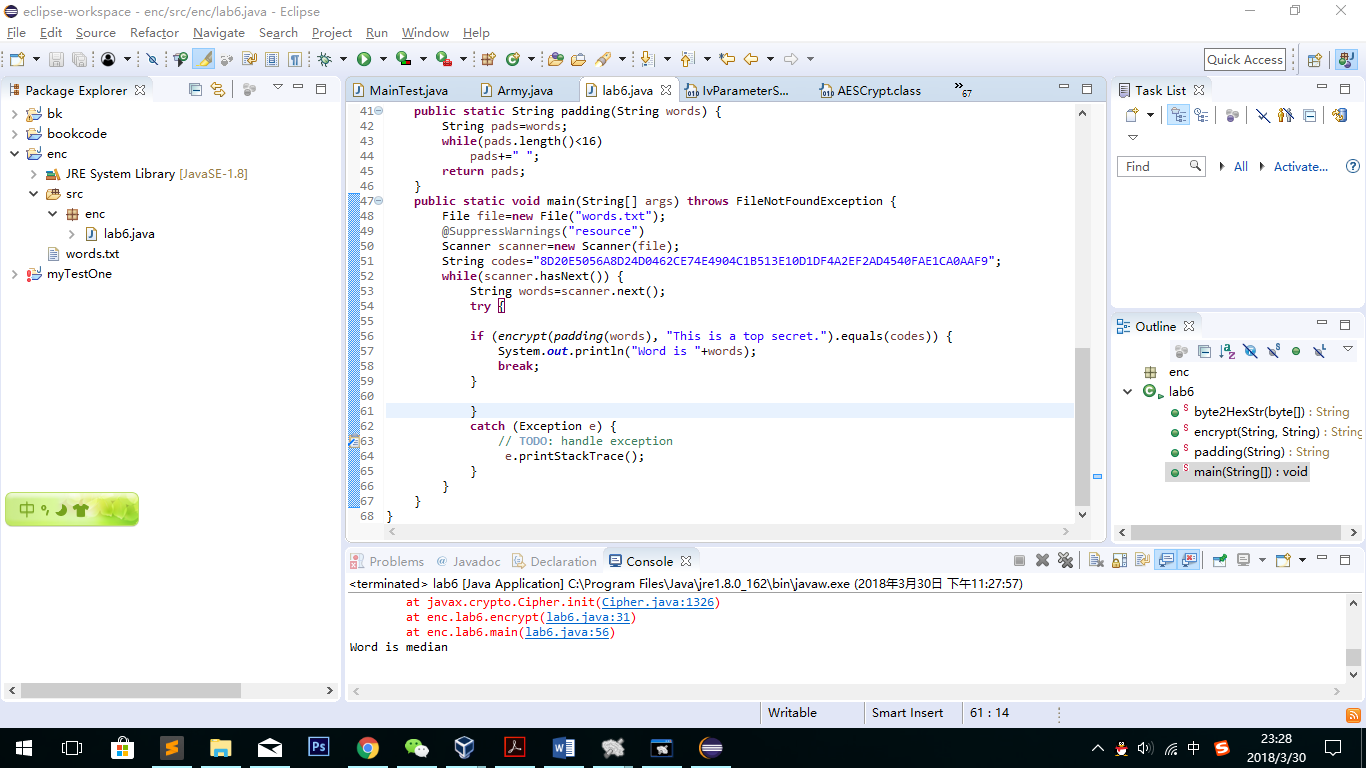
Goal is to write a program to find out the key

First, we generate the hex file.

The design is like this:

I will use different words in the words list as the key to encrypt the codes. If one solution equals to the final result, the word should be the key.

The key is median.



Codes and codes explanations are as follows.

Codes:

**package** enc;

**import** javax.crypto.Cipher;

**import** javax.crypto.spec.IvParameterSpec;

**import** javax.crypto.spec.SecretKeySpec;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.Scanner;

**public** **class** lab6 {

**public** **static** String byte2HexStr(**byte**[] b) {

String hs = "";

String stmp = "";

**for** (**int** n = 0; n < b.length; n++) {

stmp = (Integer.*toHexString*(b[n] & 0XFF));

**if** (stmp.length() == 1)

hs = hs + "0" + stmp;

**else**

hs = hs + stmp;

}

**return** hs.toUpperCase();

}

**public** **static** String encrypt(String key, String value) {

**try** {

**byte**[] zeros= {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

IvParameterSpec iv = **new** IvParameterSpec(zeros);

SecretKeySpec skeySpec = **new** SecretKeySpec(key.getBytes("UTF-8"), "AES");

Cipher cipher = Cipher.*getInstance*("AES/CBC/PKCS5PADDING");

cipher.init(Cipher.***ENCRYPT\_MODE***, skeySpec, iv);

**byte**[] encrypted = cipher.doFinal(value.getBytes());

**return** *byte2HexStr*(encrypted);

} **catch** (Exception ex) {

ex.printStackTrace();

}

**return** "";

}

**public** **static** String padding(String words) {

String pads=words;

**while**(pads.length()<16)

pads+=" ";

**return** pads;

}

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

File file=**new** File("words.txt");

@SuppressWarnings("resource")

Scanner scanner=**new** Scanner(file);

String codes="8D20E5056A8D24D0462CE74E4904C1B513E10D1DF4A2EF2AD4540FAE1CA0AAF9";

**while**(scanner.hasNext()) {

String words=scanner.next();

**try** {

**if** (*encrypt*(*padding*(words), "This is a top secret.").equals(codes)) {

System.***out***.println("Word is "+words);

**break**;

}

}

**catch** (Exception e) {

// **TODO**: handle exception

e.printStackTrace();

}

}

}

}

Codes Explanation:

Firstly, we need to pad the key with space to fill 16 bytes. That is what in the padding function. Although there are some words in the word-list that are more than 16 bytes, but we have already know the answer should be 16 bytes, so I write a try-catch structure to avoid the program being stopped. After getting the key, can use “getBytes” function to transform to byte array. The encrypt function of “aes-128-cbc” can be used as *Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");* After getting the encodes, we need to transform to string so that we can compare the two. That is the reason to write “byte2HexStr” function. “While” structure and Scanner tool ensure that each word should be tried. For the IV, just write a byte array as *byte[] zeros= {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};* Finally while testing the word median, the program finds output equals to the target, so print out “median” as the result.