SKIP TASK 1, 6.2, 6.3, 7

Task 2)

First I created a plaintext text file that is to be encrypted and decrypted.

```
[10/21/21]seed@VM:~/.../Lab2$ echo "this is text file to be encrypted" > plaint
xt.txt
[10/21/21]seed@VM:~/.../Lab2$ ls
plaintext.txt
[10/21/21]seed@VM:~/.../Lab2$ cat plaintext.txt
this is text file to be encrypted
[10/21/21]seed@VM:~/.../Lab2$
```

First I try aes 128 cipher block chaining (cbc). I encrypt with a key 00112233445566778899aabbccddeeff. I encrypt with an initial vector of 0102030405060708. This is saved to a file called cbc_cipher.bin.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in plaintext.txt -out
 cbc_cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

Then I decrypt using the same key and IV into a file called cbc_plain.txt. I validate that it worked by running the "diff" command on our original file plaintext.txt and our decrypted file cbc_plain.txt. Nothing is returned by the "diff" command, which means the contents are identical.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -d -in cbc_cipher.bin -ou
t cbc_plain.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ diff plaintext.txt cbc_plain.txt
[10/21/21]seed@VM:~/.../Lab2$
```

Next I try AES 128 Cipher Feedback.

I go through the same process in the commands below. Diff once again does not return anything, which means the files are identical prior to encryption and after decryption.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cfb -e -in plaintext.txt -out
 cfb_cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cfb -d -in cfb_cipher.bin -ou
t cfb_plain.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ diff plaintext.txt cfb_plain.txt
[10/21/21]seed@VM:~/.../Lab2$
```

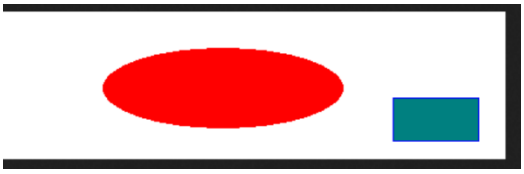
Next I try AES 128 Output Feedback.

I go through the same process in the commands below. Diff once again does not return anything, which means the files are identical prior to encryption and after decryption.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-ofb -e -in plaintext.txt -out
 ofb_cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-ofb -d -in ofb_cipher.bin -ou
t ofb_plain.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ diff plaintext.txt ofb_plain.txt
[10/21/21]seed@VM:~/.../Lab2$
```

Task 3)

First I save the picture from the website into our directory.



```
[10/21/21]seed@VM:~/.../Lab2$ ls
cbc_cipher.bin  cfb_cipher.bin  ofb_cipher.bin  pic_original.bmp
cbc_plain.txt   cfb_plain.txt   ofb_plain.txt   plaintext.txt
```

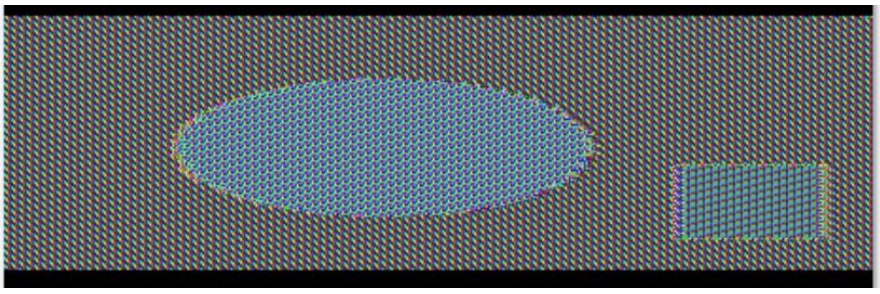
Then I encrypt the picture using ECB.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-ecb -e -in pic_original.bmp
out encrypted_pic.bin -K 00112233445566778889aabbccddeeff
```

Then I reset the header.

```
[10/21/21]seed@VM:~/.../Lab2$ head -c 54 pic_original.bmp > header
[10/21/21]seed@VM:~/.../Lab2$ tail -c +55 encrypted_pic.bin > body
[10/21/21]seed@VM:~/.../Lab2$ cat header body > full_cipher_pic.bmp
```

Then I view the picture.



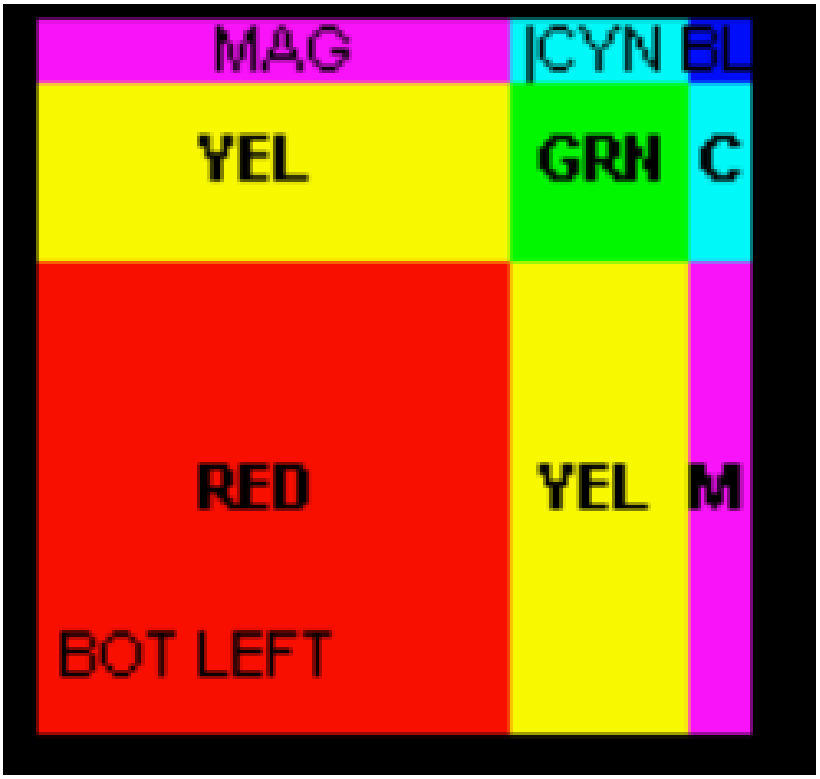
It seems similar to the original picture in some way. The colors are slightly different. Because we break the file into blocks of size 128 bit, and the use AES algorithm to encrypt each block. If two blocks are the same in the original picture, they will remain identical in the encrypted one

Next I try using CBC. First I encrypt, then I set the tail, then I cat the original picture's header to the tail and call it cbc_pic so that I can view it.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in pic_original.bmp
out cbc_pic.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ tail -c +55 cbc_pic.bin > body
[10/21/21]seed@VM:~/.../Lab2$ cat header body > full_cbc_pic.bmp
[10/21/21]seed@VM:~/.../Lab2$
```

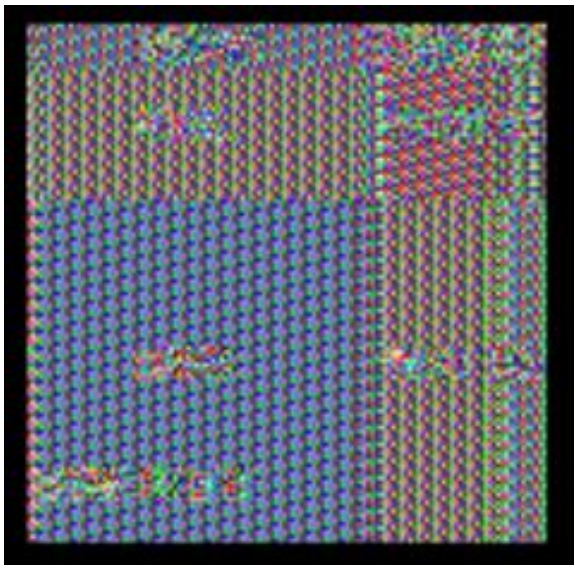
Now I view it. The picture is basically unviewable. This encryption scheme is very interesting and very successful.

Now, I try with my own picture of choice.



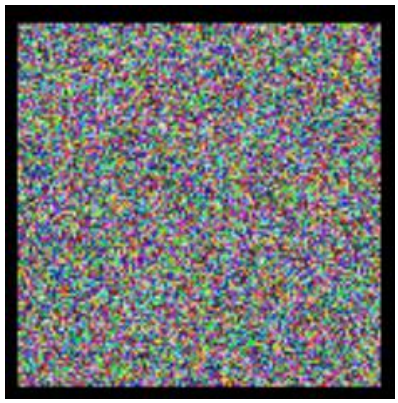
First I use ECB to encrypt and view. What is resulted is below.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-ecb -e -in pic.bmp -out ecb_p
ic.bmp -K 00112233445566778899aabbccddeeff
[10/21/21]seed@VM:~/.../Lab2$ head -c 54 pic.bmp > header
[10/21/21]seed@VM:~/.../Lab2$ tail -c +55 ecb_pic.bmp > body
[10/21/21]seed@VM:~/.../Lab2$ cat header body > full_ecb_pic1.bmp
```



Then I use CBC for the same image.

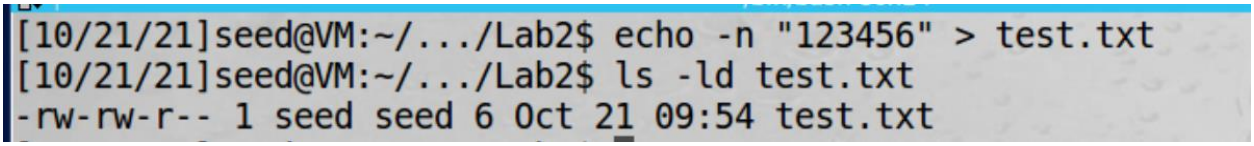
```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in pic.bmp -out cbc_p
ic1.bmp -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ tail -c +55 cbc_pic.bmp > body
tail: cannot open 'cbc_pic.bmp' for reading: No such file or directory
[10/21/21]seed@VM:~/.../Lab2$ tail -c +55 cbc_pic1.bmp > body
[10/21/21]seed@VM:~/.../Lab2$ cat header body > full_cbc_pic1.bmp
[10/21/21]seed@VM:~/.../Lab2$
```



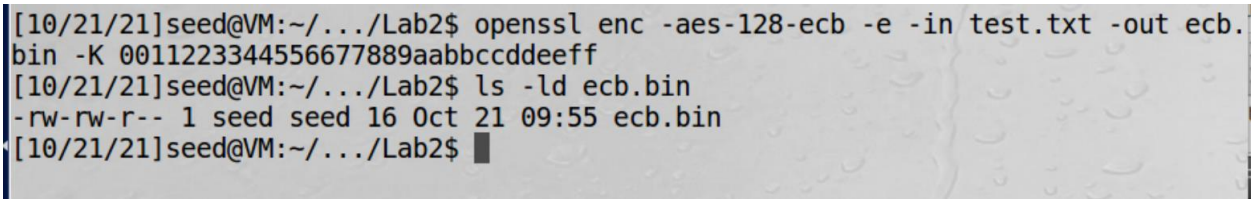
You can still see essences and principles of the original image in the ECB encrypted image. You can see the shapes, lines, and even some colors. The colors are slightly different. Because we break the file into blocks of size 128 bit, and the use AES algorithm to encrypt each block. If two blocks are the same in the original picture, they will remain identical in the encrypted one, which is why we can still see some of the original image in ECB. However, in the CBC encrypted image, it all looks like gibberish.

Task 4)

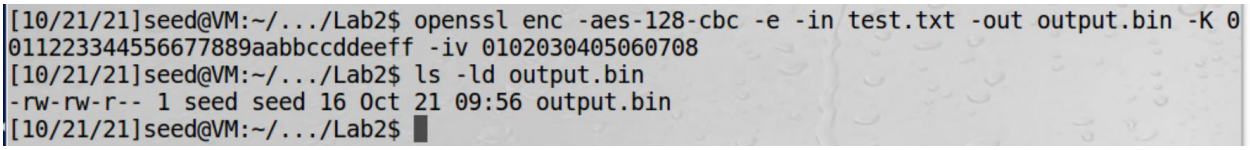
First I make a file to encrypt that has 6 bytes.



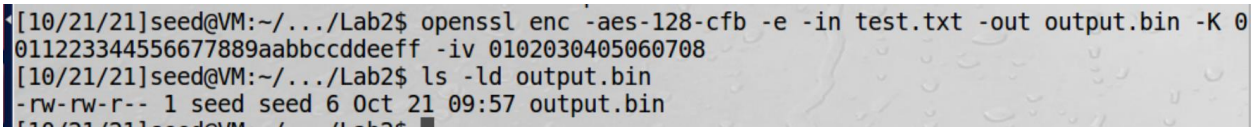
Then, I use ECB to encrypt and check to see if it has padding. The encrypted file has 16 bytes, so it must have padding.



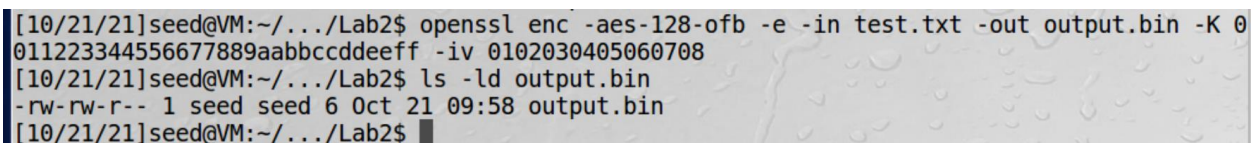
Then I try CBC. It also uses padding.



Then I try CFB. It does not have padding. The encrypted file is 6 bytes, as well. This doesn't need padding because they take outputs of the previous blocks, which must be the same size as the block size, as inputs to their encryption schemes.



Then I try OFB. It also does not have padding. This doesn't need padding because they take outputs of the previous blocks, which must be the same size as the block size, as inputs to their encryption schemes.



Then, to test how padding works with CBC, we make three files with 5,10,16 bytes respectively. We see that their sizes are indeed 5,10,16 bytes respectively.

```
[10/21/21]seed@VM:~/.../Lab2$ echo -n "12345" > f1.txt
[10/21/21]seed@VM:~/.../Lab2$ echo -n "1234567890" > f2.txt
[10/21/21]seed@VM:~/.../Lab2$ echo -n "1234567890abcdef" > f3.txt
[10/21/21]seed@VM:~/.../Lab2$ ls -ld f*.txt
-rw-rw-r-- 1 seed seed  5 Oct 21 10:03 f1.txt
-rw-rw-r-- 1 seed seed 10 Oct 21 10:03 f2.txt
-rw-rw-r-- 1 seed seed 16 Oct 21 10:03 f3.txt
```

Then we encrypt the files using CBC and print the sizes of the encrypted files. The two files that began with less than 16 bytes are padded to become 16 bytes. The 16-byte long original message is padded to be 32 bytes now.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in f1.txt -out f1.bin -K 0011223
344556677889aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in f2.txt -out f2.bin -K 0011223
344556677889aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in f3.txt -out f3.bin -K 0011223
344556677889aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ ls -ld f*.bin
-rw-rw-r-- 1 seed seed 16 Oct 21 10:04 f1.bin
-rw-rw-r-- 1 seed seed 16 Oct 21 10:04 f2.bin
-rw-rw-r-- 1 seed seed 32 Oct 21 10:04 f3.bin
[10/21/21]seed@VM:~/.../Lab2$
```

Next, I decrypt the files with the -nopad option. This will preserve the padding that the encryption scheme put on the file so that we can see in plaintext exactly what is getting used to pad the files.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -d -in f1.bin -out f1_plain.txt -K 0
01122334456677889aabbccddeeff -iv 0102030405060708 -nopad
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -d -in f2.bin -out f2_plain.txt -K 0
01122334456677889aabbccddeeff -iv 0102030405060708 -nopad
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -d -in f3.bin -out f3_plain.txt -K 0
01122334456677889aabbccddeeff -iv 0102030405060708 -nopad
[10/21/21]seed@VM:~/.../Lab2$
```

Now we inspect the files.

```
[10/21/21]seed@VM:~/.../Lab2$ xxd f1_plain.txt
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
[10/21/21]seed@VM:~/.../Lab2$ xxd f2_plain.txt
00000000: 3132 3334 3536 3738 3930 0606 0606 0606  1234567890......
[10/21/21]seed@VM:~/.../Lab2$ xxd f3_plain.txt
00000000: 3132 3334 3536 3738 3930 6162 6364 6566  1234567890abcdef
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
[10/21/21]seed@VM:~/.../Lab2$
```

You can see that the padding appears to be decrypted as periods. The 5 byte file had 11 periods to pad the size to 16 total. The 10 byte file had 6 periods to pad the file to 16 total bytes. And the 16 byte file had 16 more periods to pad it to a size of 32.

Task 5)

First I create a massive file containing more than 1000 bytes. It ends up being 1001 bytes.

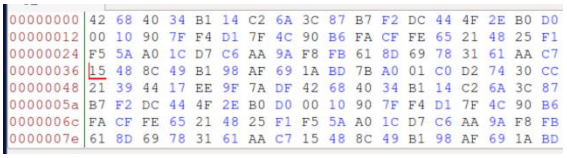
```
[10/21/21]seed@VM:~/.../Lab2$ python -c "print '1234567890'*100" > big_file.txt
[10/21/21]seed@VM:~/.../Lab2$ cat big_file.txt
12345678901234567890123456789012345678901234567890123456789012345678901234567890
23456789012345678901234567890123456789012345678901234567890123456789012345678901
34567890123456789012345678901234567890123456789012345678901234567890123456789012
45678901234567890123456789012345678901234567890123456789012345678901234567890123
56789012345678901234567890123456789012345678901234567890123456789012345678901234
67890123456789012345678901234567890123456789012345678901234567890123456789012345
78901234567890123456789012345678901234567890123456789012345678901234567890123456
89012345678901234567890123456789012345678901234567890123456789012345678901234567
90123456789012345678901234567890123456789012345678901234567890123456789012345678
01234567890123456789012345678901234567890123456789012345678901234567890123456789
12345678901234567890123456789012345678901234567890123456789012345678901234567890
[10/21/21]seed@VM:~/.../Lab2$ ld big_file.txt
ld:big_file.txt: file format not recognized; treating as linker script
ld:big_file.txt:1: syntax error
[10/21/21]seed@VM:~/.../Lab2$ ls -ld big_file.txt
-rw-rw-r-- 1 seed seed 1001 Oct 21 10:12 big_file.txt
[10/21/21]seed@VM:~/.../Lab2$
```

Then I encrypt it using ECB.

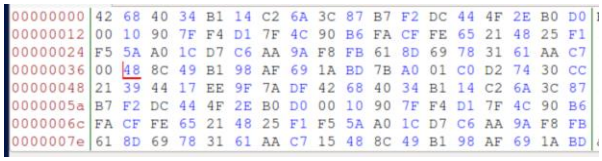
```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-ecb -e -in big_file.txt -out big_file.bi
n -K 00112233445566778899aabbccddeeff
```

Oops, I "accidentally" corrupt the 55th byte, this is element 54, so its offset is 0x36. It was 15, but I will make it 00.

Before:



After:



Now we decrypt it.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-ecb -d -in big_file.bin -out ecb_big_fil
e.txt -K 00112233445566778899aabbccddeeff
```

I found the following python script online. It calculates the number of different bytes between two files. I'll use it to calculate the number of different bytes between the original big_file.txt and the decrypted corrupted file ecb_big_file.txt

```
with open('big_file.txt', 'rb') as f:
    f1 = f.read()
with open('ecb_big_file.txt', 'rb') as f:
    f2 = f.read()
res = 0
for i in range(min(len(f1), len(f2))):
    if f1[i] != f2[i]:
        res += 1
print("diff bytes: "+str(res+abs(len(f1)-len(f2))))
```

The output from this python script is below. 17 bytes were different for ECB.

```
[10/21/21]seed@VM:~/.../Lab2$ python3 diff.py
diff bytes: 17
```

I do the following for ECB, CBC, CFB, and OFB and get 17, 17, 17, 1 respectively.

To answer the question **"how much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively?"**

ECB can recover all but one corrupted block. That is why the difference is 17. The block size is 16, plus one corrupted byte. It only uses the current block during encryption, so only the current block will be corrupted.

CBC can recover all blocks prior to the corrupted byte. This operation mode uses the previous block during encryption, as the cipher text from the last block is used as the IV to the next block. That is why all blocks after the corrupted byte will be corrupted.

CFB can recover all blocks prior to the corrupted byte. This operation mode uses the previous block during encryption, as the cipher text from the last block is used as the IV to the next block. That is why all blocks after the corrupted byte will be corrupted.

OFB can recover all but 1. It only uses the current block during encryption, so only the current block will be corrupted.

Task 6.1)

I'll use the 5-bit text file I created earlier, f1.txt. I encrypt it using IV 0102030405060708 twice and show the output. As you can see, the two outputted encrypted files are identical. I encrypt one and called it f1_a1.bin and the other f1_a2.bin

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_a1.bin -K 0011
223344556677889aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_a2.bin -K 0011
223344556677889aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ xxd f1_a1.bin
00000000: 8dd1 b019 2c2e c490 cddc 6b29 ed33 211d  ....,.....k).3!.
[10/21/21]seed@VM:~/.../Lab2$ xxd f1_a2.bin
00000000: 8dd1 b019 2c2e c490 cddc 6b29 ed33 211d  ....,.....k).3!.
```

Next, I use IV 0102030405060708 once, and then I use IV 1234567812345678. I call the one encrypted with IV=0102030405060708 f1_a.bin. Then, I call the one encrypted with IV=1234567812345678 f1_b.bin. As you can see, when I hexdump the binaries, they are very different.

```
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_a.bin -K 00112
23344556677889aabbccddeeff -iv 0102030405060708
[10/21/21]seed@VM:~/.../Lab2$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_b.bin -K 00112
23344556677889aabbccddeeff -iv 1234567812345678
[10/21/21]seed@VM:~/.../Lab2$ xxd f1_a.bin
00000000: 8dd1 b019 2c2e c490 cddc 6b29 ed33 211d  ....,.....k).3!.
[10/21/21]seed@VM:~/.../Lab2$ xxd f1_b.bin
00000000: f469 2921 603a fe3f 0189 6d78 e5ed c81f  .i)!`:.?..mx....
[10/21/21]seed@VM:~/.../Lab2$
```

When plaintexts are identical, the IV cannot be identical for because it will lead to the same ciphertexts.