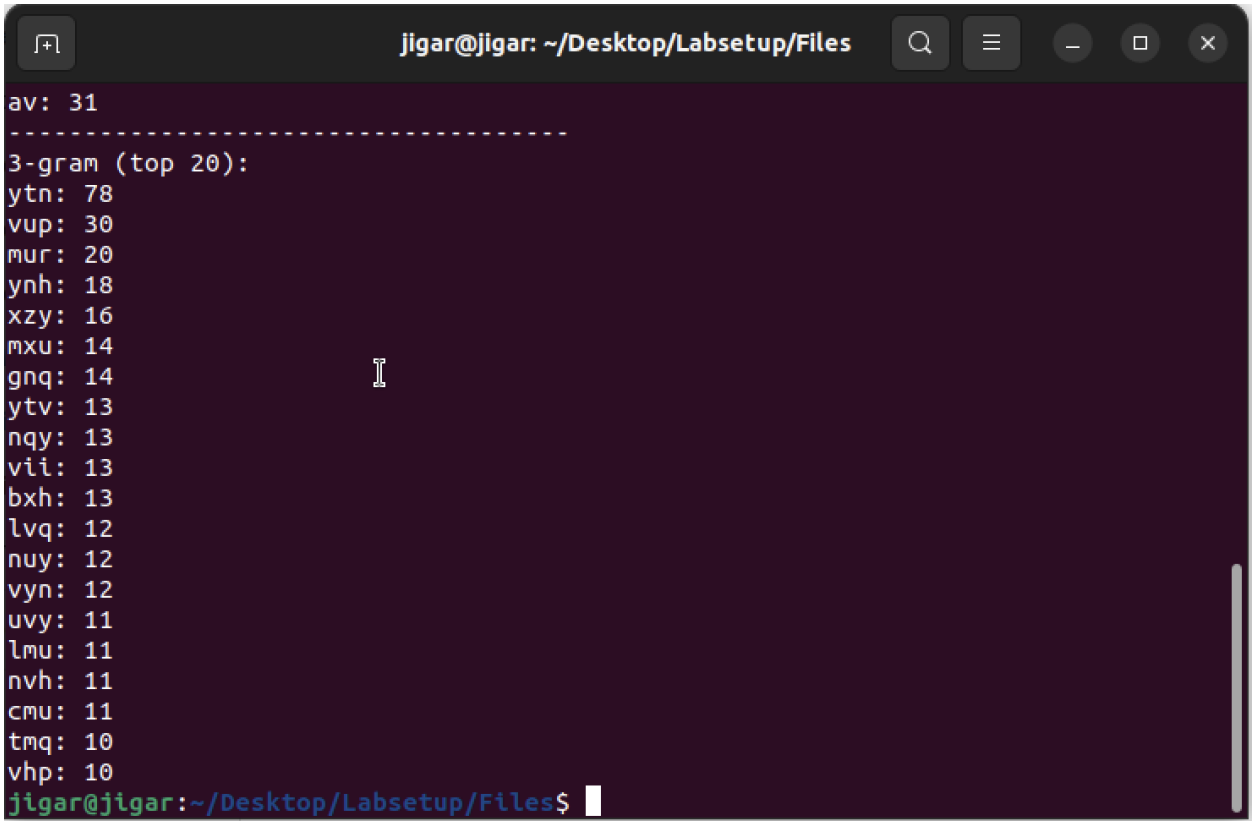
Task 1: Frequency Analysis

In this task monoalphabetic cipher is used for encryption and this cipher is not so secure. it can be easily subjected to frequency analysis.

We got single letter frequencies, bigram frequencies and trigram frequencies using freq.py file. With the help of difference resources on internet we can guess the most frequent letters and then we can guess other words.

Ex.:



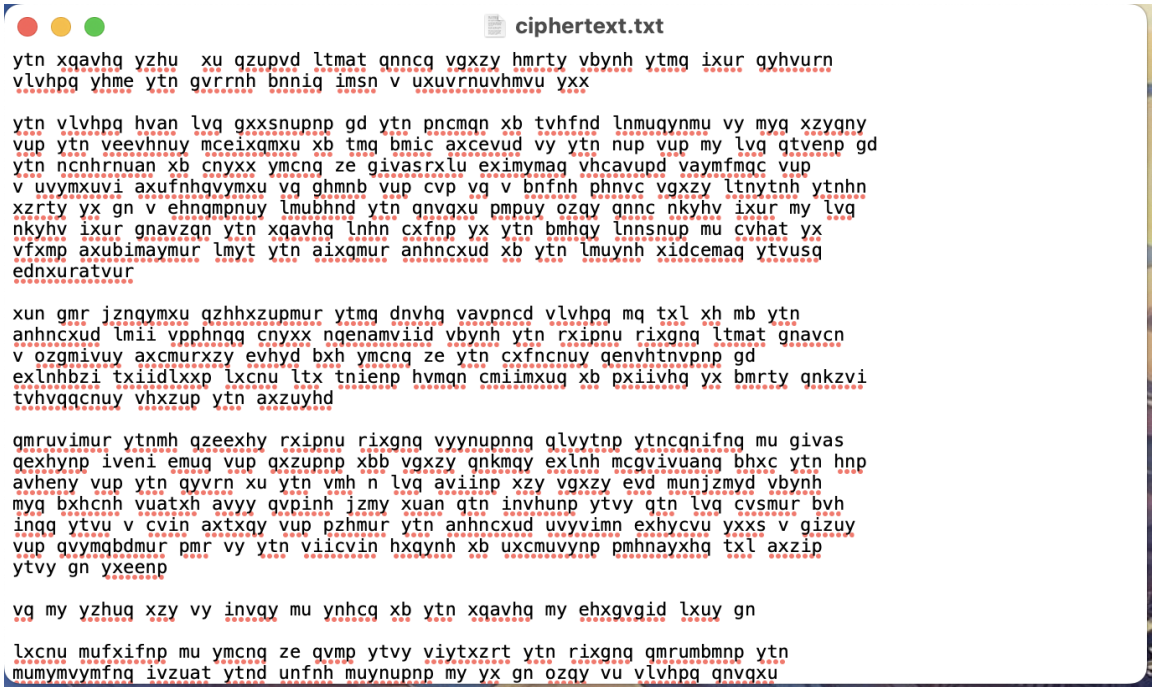
In ciphertext most frequent trigram letters are *ytn* and most frequent english word is *the* so ***y*** will be ***t***, ***t*** will be ***h*** and ***n*** wil be ***e***. Second most freq is *vup* and in english it's *and.* So that's how you can guess the correct letters.

– Key for ciphertext is : cfmypvbrlqxwiejdsgkhnazotu

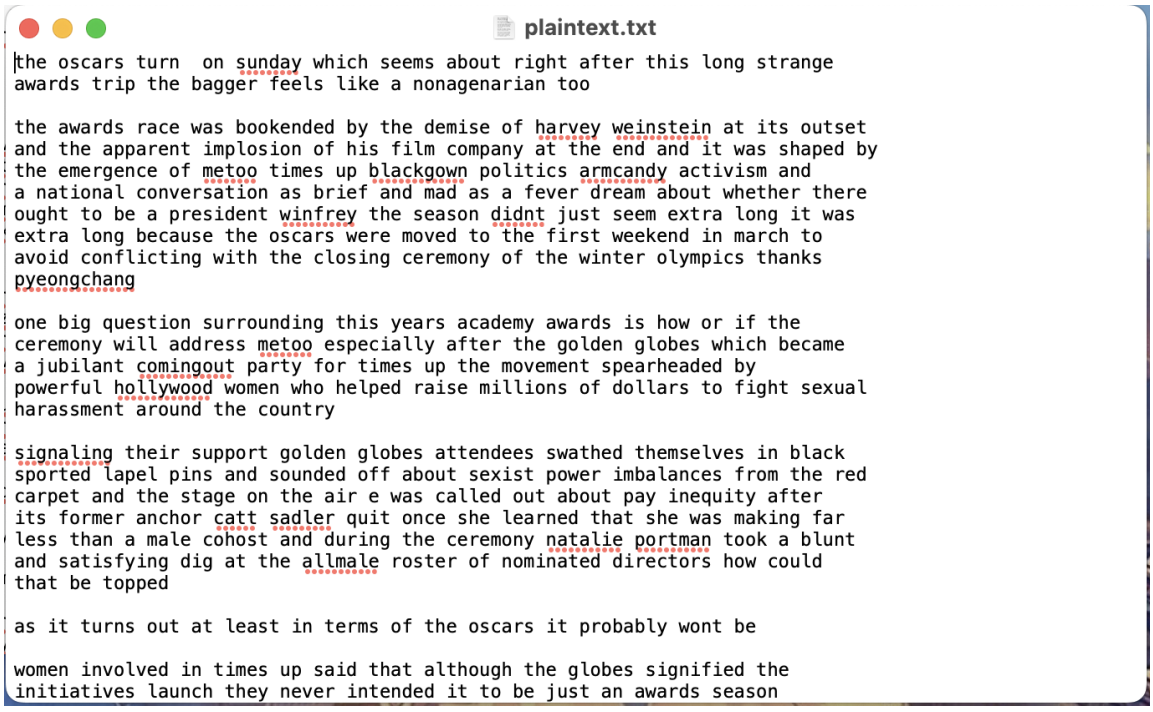
– Replace this key with abcdefghijklmnopqrstuvwxyz with the use of command 'tr'.

– $ tr 'abcdefghijklmnopqrstuvwxyz' 'cfmypvbrlqxwiejdsgkhnazotu' \
<ciphertext.txt> plaintext.txt

This command will create a new text file with name plaintext.txt and in that file it will replace all letter with key letter and we can get decrypted message back with the help of frequency analysis.



This was ciphertext

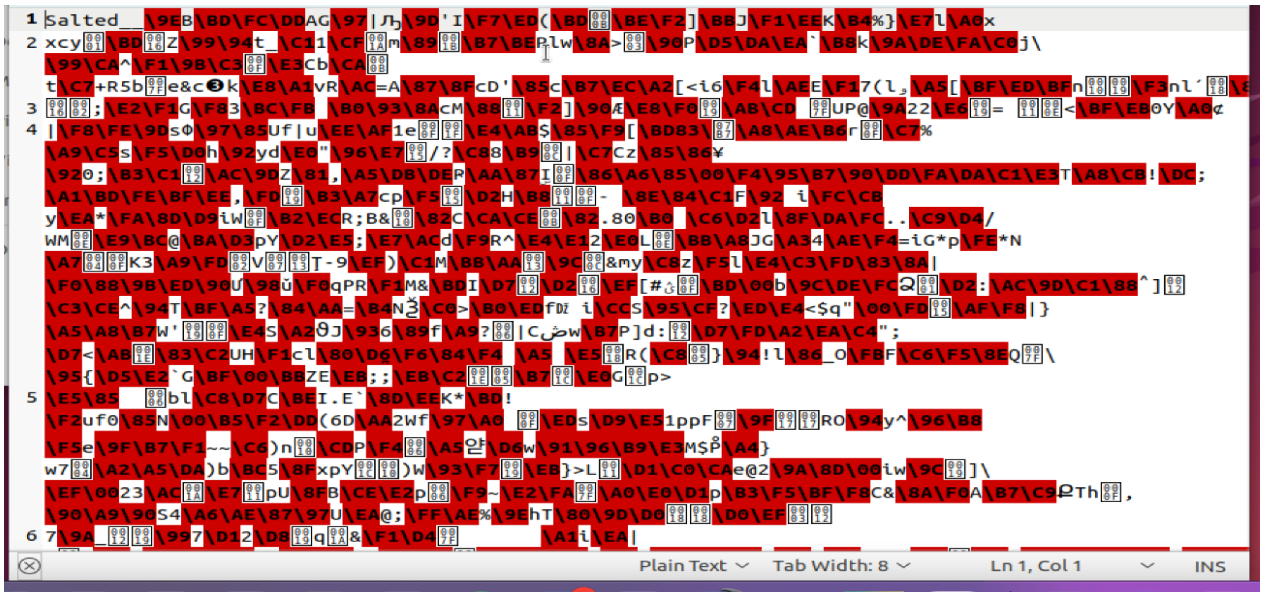


And this is plain text that we got from deciphering cipher text.

Task 2: Encryption using Different Ciphers and Modes

– For this task we used openssl command for encrypt and decrypt files.

```
[06/18/22]seed@VM:~/.../Task4$ openssl enc -aes-128-ecb -e -in ex.txt -out ecb.t
xt
enter aes-128-ecb encryption password:
Verifying - enter aes-128-ecb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[06/18/22]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -e -in ex.txt -out cbc.t
xt
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[06/18/22]seed@VM:~/.../Task4$ openssl enc -aes-128-cfb -e -in ex.txt -out cfb.t
xt
enter aes-128-cfb encryption password:
Verifying - enter aes-128-cfb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[06/18/22]seed@VM:~/.../Task4$ openssl enc -aes-128-ofb -e -in ex.txt -out ofb.t
xt
enter aes-128-ofb encryption password:
Verifying - enter aes-128-ofb encryption password:
*** WARNING : deprecated key derivation used.
```



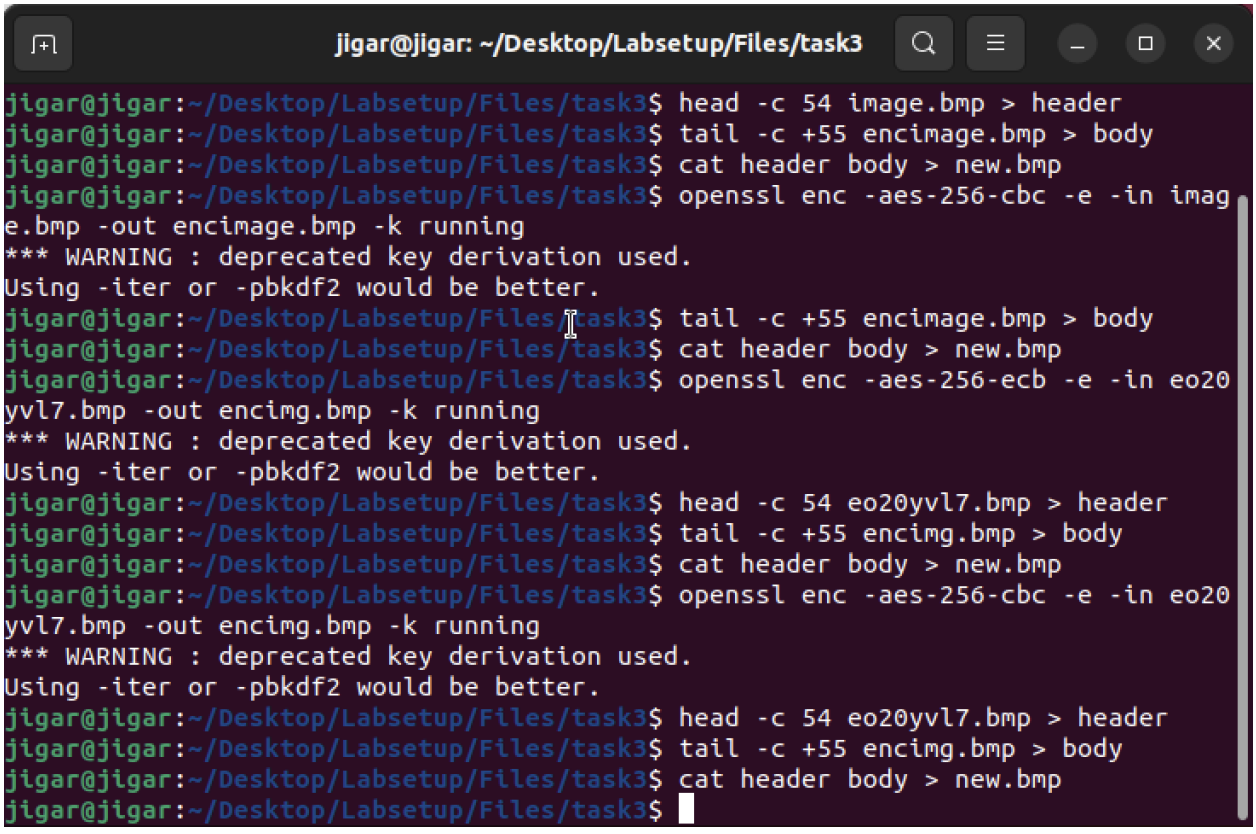
Encrypted cipher text.
(we used previous task's plaintext for this task)

– We used -aes-128-ecb, -aes-128-cbc, -aes-128-cfb, -aes-128-ofb, -aes-256, -aria-128-cbc modes for encryption.

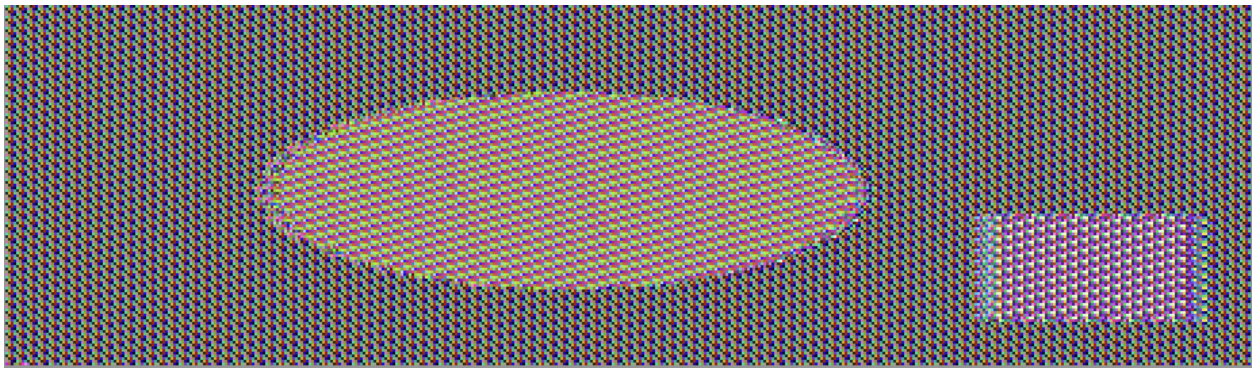
– In this process we used several commands for encryption and decryption..
  – **-in <file>** for input file
  – **-out <file>** for output file
  – **-k/-iv** for key/iv
  – **-e** for encryption
  – **-d** for decryption.

Task 3: Encryption Mode – ECB vs. CBC

In this task we encrypt images using ecb and cbc modes with the help of openssl command



This is the image before encryption

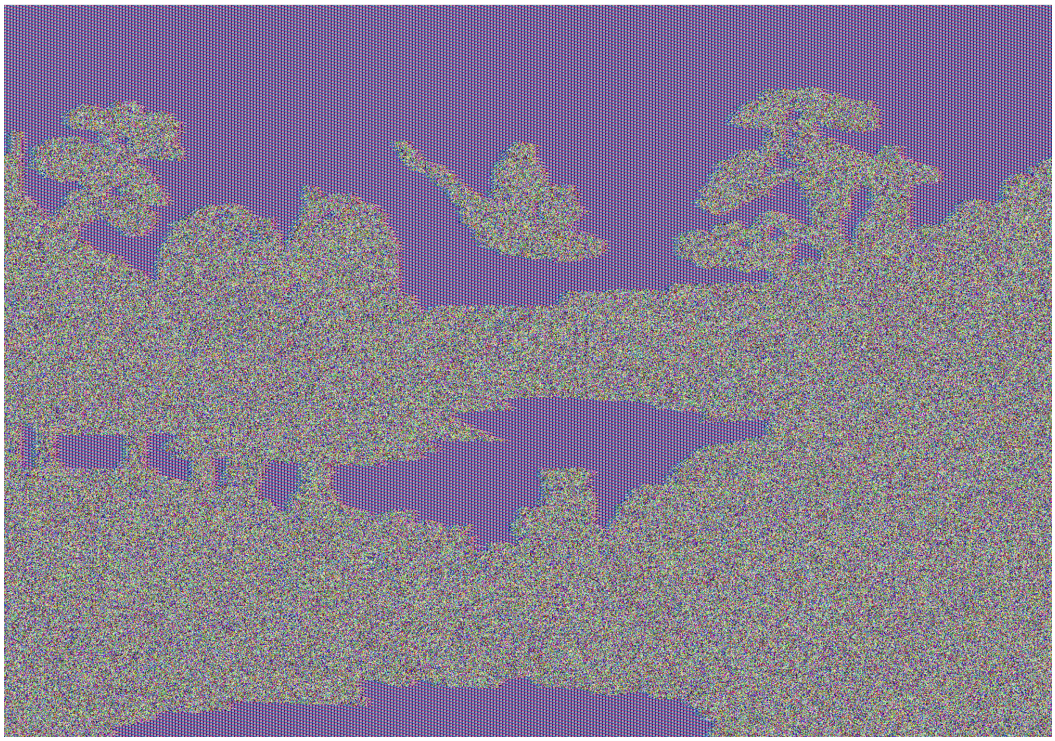


This is the image after encryption with ECB mode

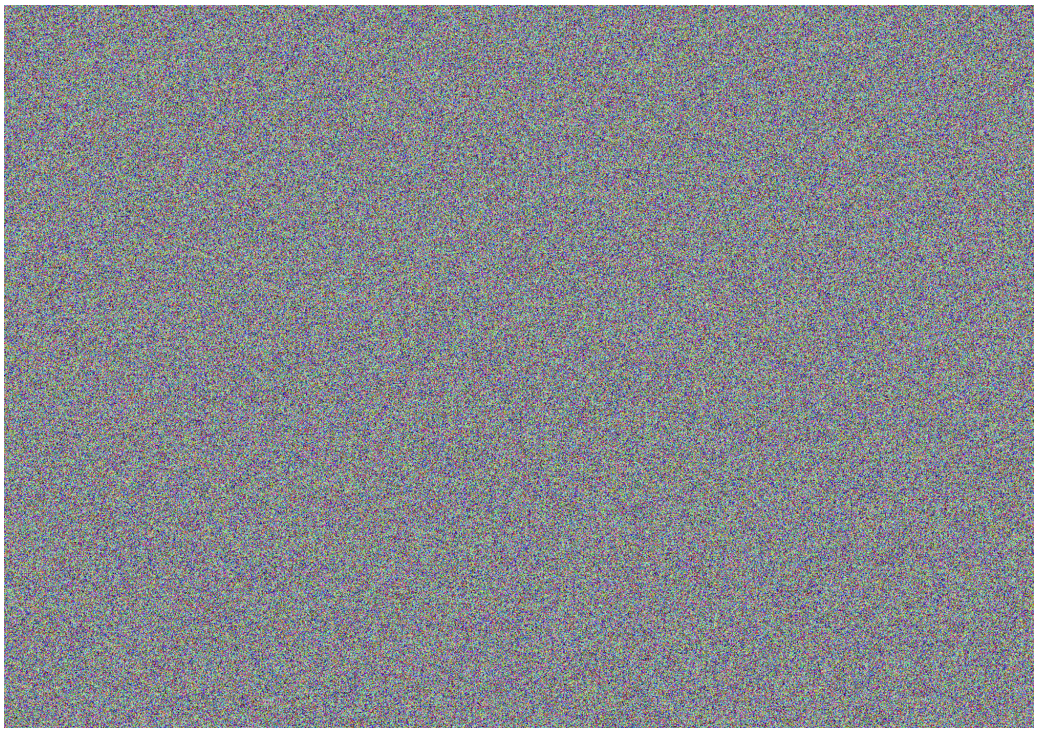
This is the image after encryption with CBC mode

– After observing encrypted images we can tell that with ecb mode we can still get some information about the real image and with cbc we can not get any information about the real image. This same thing we observe with another image.

This is the image before encryption



This is the image after encryption with ECB mode



This is the image after encryption with CBC mode

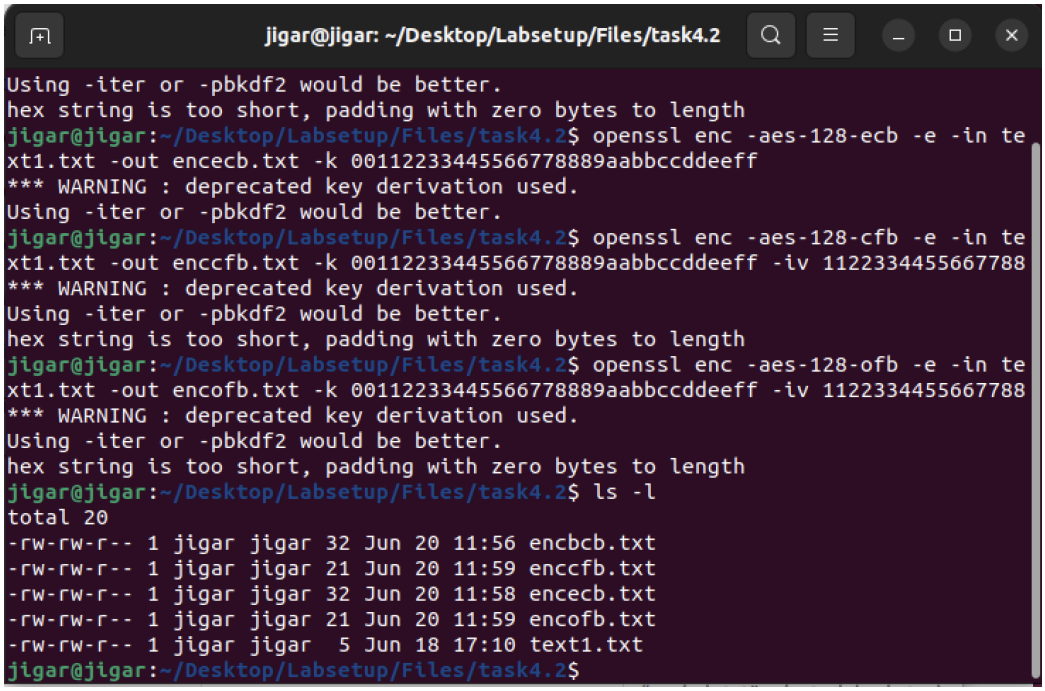
– In this image also we can guess the shape of real image from encrypted image with ecb and can not get any information about real image with cbc.

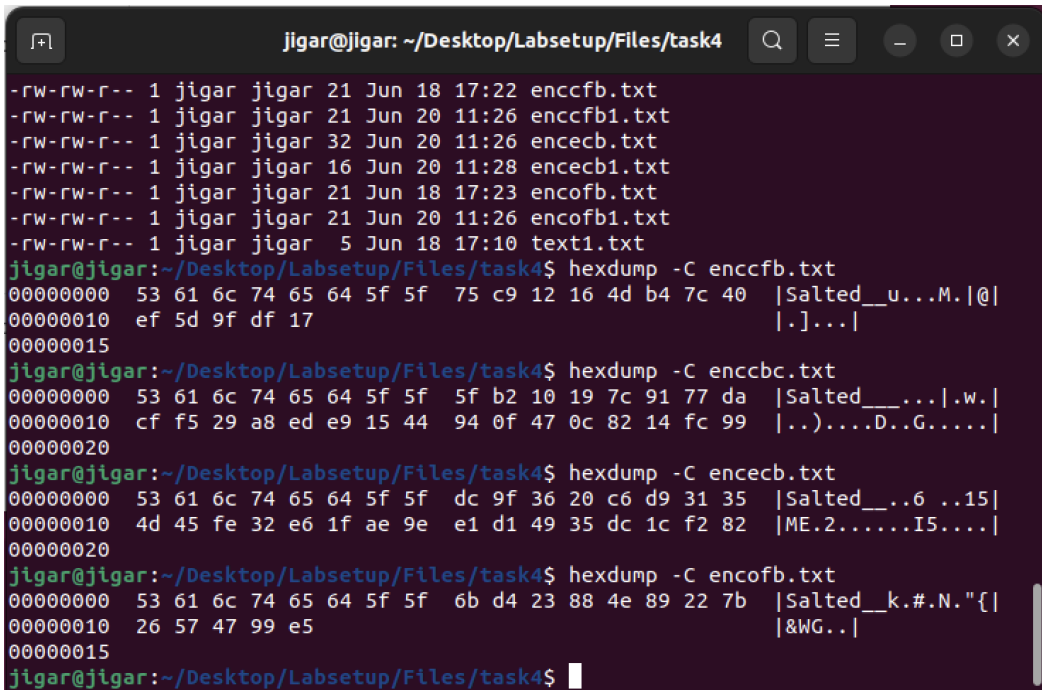
– In conclusion we can say that using CBC mode for encryption is more safe and make it hard to derive any information about real image than ECB mode.

Task 4: Padding

(1)

- We encrypted a 5 byte file with cbc, cfb, ecb,ofb modes

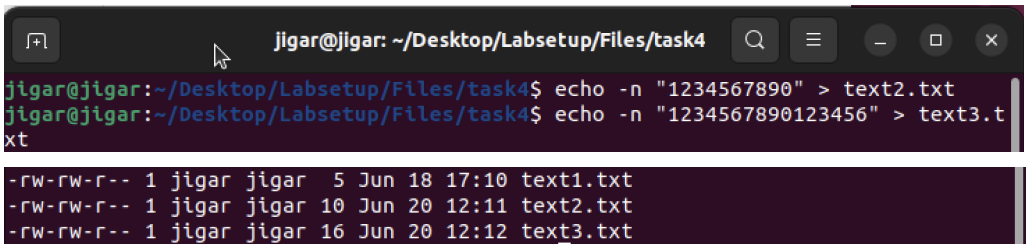




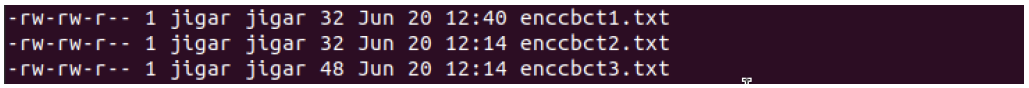
– Padding is use for filling the space that are remaining in the block after filling it with plaintext. After observing HAX formet of text file we can say that **ECB and CBC modes need padding. CFB and OFB do not need padding** because the encryption and decryption functions are identical, and it doesn't require the plaintext data to be padded (i.e. the ciphertext is the same length as the plaintext).

(2)

- created 3 text files with 5 bytes, 10 bytes, 16 bytes



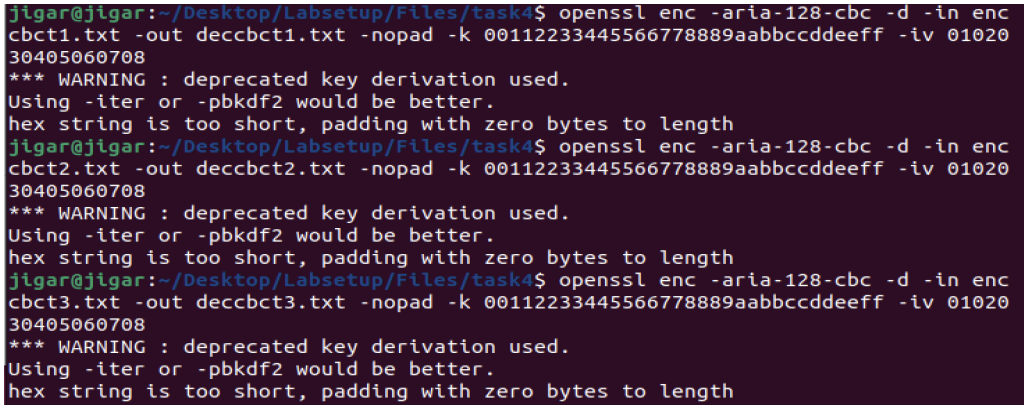
- After encryption of all files with -aes-128-cbc

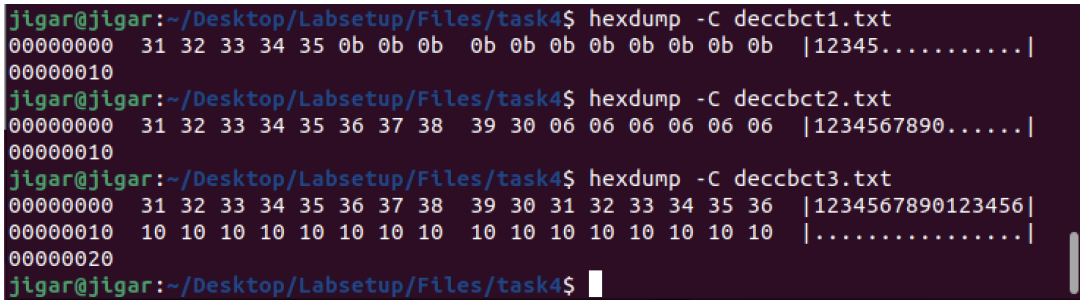


– Size of files after encryption is

Size of plaintext	Size of ciphertext
5 bytes	32 bytes
10 bytes	32 bytes
16 bytes	48 bytes

- Decryption with -nopad command





– In 5 bytes plaintext, 11 byte of padding was added and padding was 0B.

– In 10 bytes plaintext, 6 bytes of padding was added and padding was 06.

– In 16 bytes plaintext, 16 bytes of padding was added and padding was 10.

Task 5: Error Propagation – Corrupted Cipher Text

The file contains a series of "**1234567890**".(100 times)

Our Initial assumptions were:-

1. CBC: We expected that 1 entire block of 8 bytes would be affected and a single byte in next block will be affected
2. CFB: We expected only a single bit error.
3. OFB: We expected only a single bit error just like CFB.
4. ECB: We expect an entire block of error, making the corrupted byte's block completely useless.

After conducting the practical we gain that:

Files :

- decb - dichpered ECB
- dcfb - dichpered EFB
- dofb - dichpered OFB
- dcbc - dichpered CBC

– Final Observations drawn:

1. CBC: Entire block of corrupted bytes
2. CFB: Entire block of corrupted bytes with 8bytes of usable data in between
3. OFB: One corrupted byte
4. ECB: Entire block of corrupted bytes

Justification:

1. CBC : the way CBC deciphers data, by XORing previous ciphertext and block deciphering using key and block, it in turn makes the block completely useless with random bits.
2. CFB : the way CFB deciphers data, the affected bit in our case 55th will be corrupted as well as the next block of data because of XORing and key block deciphering.
3. OFB : the least affected deciphering method, as 1byte stream decipher method provides lowest loss of usable data.
4. ECB : As one by one byte of data is deciphered using individual keys, we get an entire block of corrupted data.

Task 6:Initial Vector (IV) and Common Mistakes

– 6.1 IV Experiment:

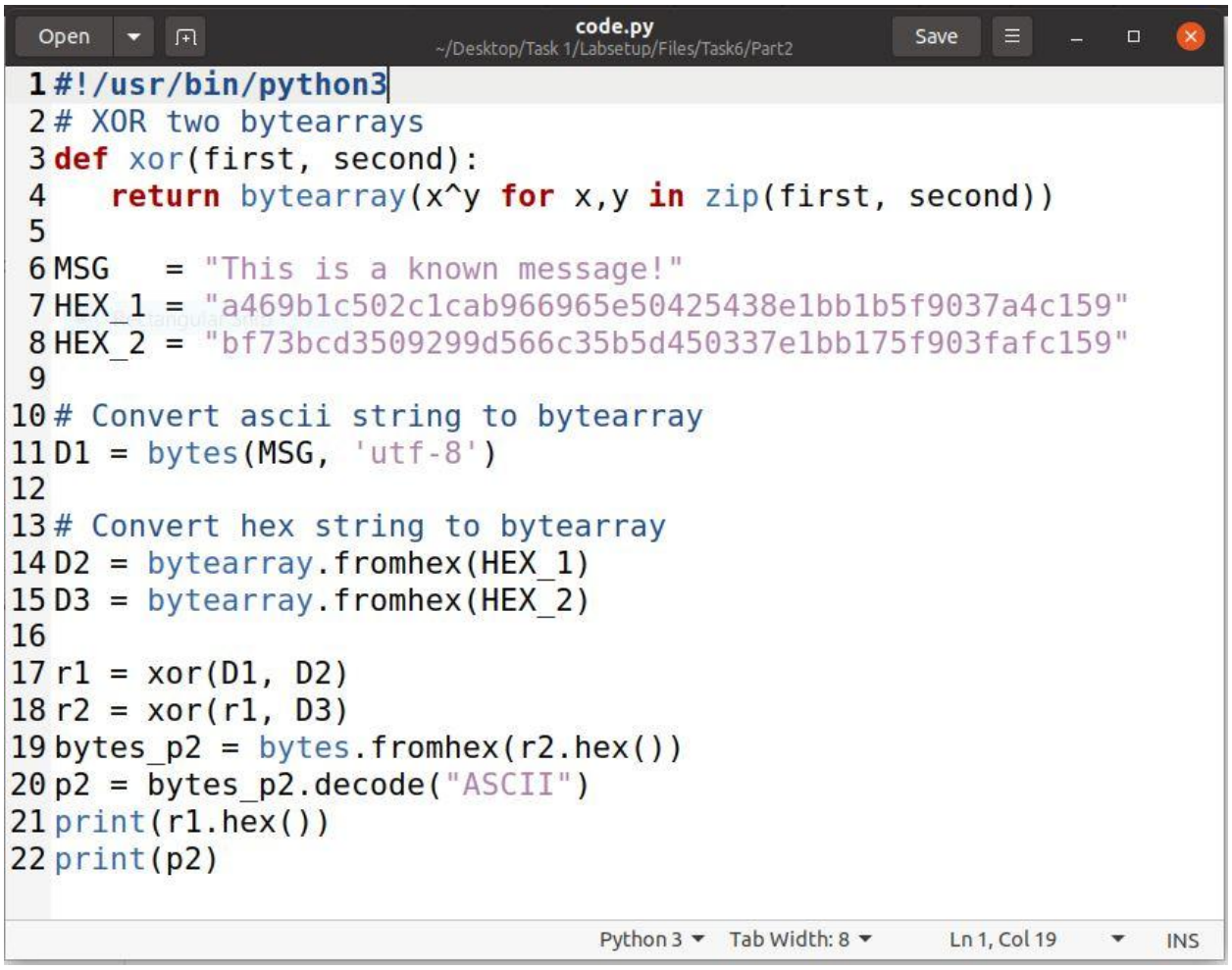
The image we obtain after using same IV's and different IV's:

Justification :

If the same key and IV are used, then you get the same stream, so you have the conditions of the (in)famous two-times pad. **Without knowing the key, you can still compute the XOR of any two messages, which is often enough to crack them, by exploiting their internal structure.**

– 6.2 Common Mistake: Use the Same IV:

After readjusting the code to gain our intended output:



The obtained output:

```
[06/19/22]seed@VM:~/.../Task6$ python3 code.py
f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a478
Order: Launch a missile!
```

The hidden plaintext2 is : **Lanch a missile!**

Part2:

OFB is an improved version of CFB. The basic difference among the two is that in OFB, the output of block cipher encryption is fed as a feedback to calculate the next block. The first block is calculated using an initialization vector (normally a string of random digits and alphabets). **If the IV is reused, the whole key stream will be reproduced again.**

– 6.3 Common Mistake: Use a Predictable IV :

Our initial nc 10.9.0.80 3000 :

```
[06/21/22]seed@VM:~/.../encryption_oracle$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: acad1c3e77e0e64589b89c95d387ac75
The IV used    : 5af0b23d5c9d5edf09fa20b3fbeeef74

Next IV        : b6cca05e5c9d5edf09fa20b3fbeeef74
Your plaintext : 11223344aabbccdd
Your ciphertext: 596892191b41f50d654ddc8c3a737ae4

Next IV        : 09b2f5715c9d5edf09fa20b3fbeeef74
Your plaintext : ae5c666baabbccdd
```

With the provided IV's we applied Xor's and got the right answer 2nd Your Plaintext.

Excluding all the padding, we were on right track.

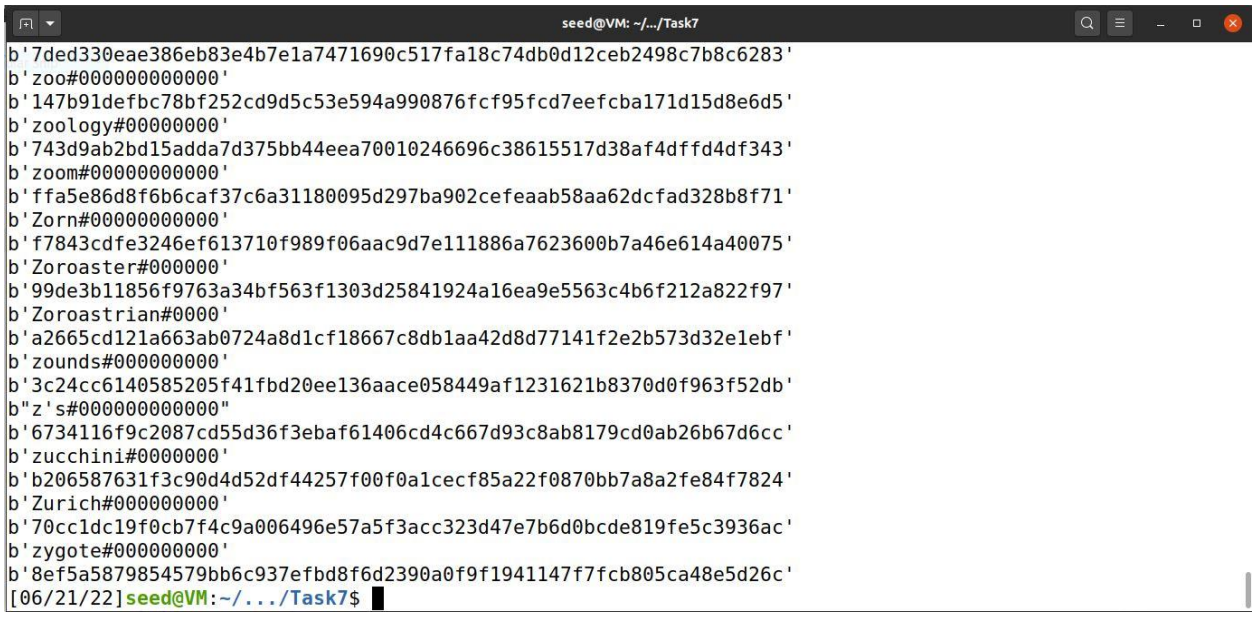
To find out Bob's initial plaintext we carried on with same method to give us:

```
[06/21/22]seed@VM:~/.../Part3$ python3 xor.py
b55961630000000000000000000000000
[06/21/22]seed@VM:~/.../Part3$ python3 xor.py
a25312630000000000000000000000000
```

1. XOR result for "Yes", which does not match with Bob's first ciphertext
2. XOR result for "No", which does match with Bob's first ciphertext, excluding all the padding and salt values.

Task 7:Programming using the Crypto Library

We didn't manage to find the key using below provided code



Code of python file:

```
from Crypto.Cipher import AES

from Crypto.Util.Padding import pad,unpad

from binascii import hexlify, unhexlify



fpx = open("/home/seed/Desktop/words.txt","r")

for fp in fpx:

      if(len(fp) <= 16):

            x = 16 - len(fp)

            a = ""

            for _ in range(0,x):

                  a += "0"

            key_str = fp+a

            key = str.encode(key_str)
```

```
            key_str = key_str.replace("\n","0")

            print(key)

            IV_hex = "aabbccddeeff00998877665544332211"

            IV_hex_bytes = bytes.fromhex(IV_hex)

            text_str = "This is a secret message."

            text = str.encode(text_str)

            mode = AES.MODE_CBC

            encryptor = AES.new(key, mode, IV = IV_hex_bytes)

            ciphertext = encryptor.encrypt(pad(text,AES.block_size))

            match_hex =
b'764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2'

            guess_hex = hexlify(ciphertext)

            print(guess_hex)

            if(guess_hex == match_hex):

                  print("Key found :" + fp)

                  decryptor = AES.new(key, mode, IV=IV_from_hex_bytes)

                  plain = unpad(decryptor.decrypt(ciphertext),AES.block_size)

                  print(plain)

                  break
```