CSE 565 Lab 1 Report
Secret-Key Encryption Lab

Notes: (IMPORTANT)

- ➔ It is required to use this report template.
- ➔ **Select <File> - <Make a copy> to make a copy of this report for yourself.**
- ➔ Report your work in each section. **Describe** what you have done and what you have observed. You should take screenshots to support your description. You also need to provide an explanation of the observations that are interesting or surprising. Please also list the important code snippets followed by an explanation.
- ➔ Simply attaching code or screenshots without any explanation will NOT receive credits.
- ➔ Do **NOT** claim anything you didn't do. If you didn't try on a certain task, leave that section blank. You will receive a ZERO for the whole assignment if we find any overclaim.
- ➔ Grading will be based on your **description** and the completion of each task.
- ➔ After you finish, export this report as a PDF file and submit it on UBLearns.

Your Full Name: Nazmus Saquib
UBITName: nsaquib2
Student Number: 50510460

I, nsaquib2 (UBITName), have read and understood the course academic integrity policy.

(Your report will not be graded without filling in the above AI statement)

---

1. Task 1: Frequency Analysis

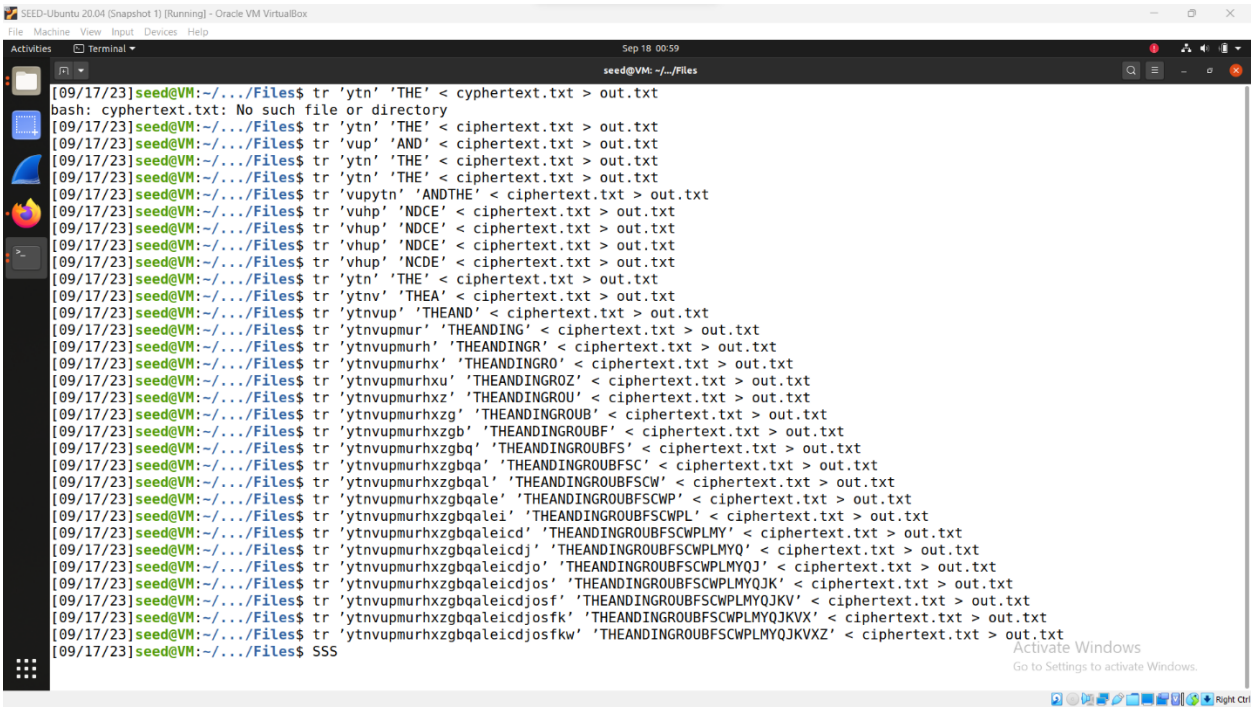
What I have done:

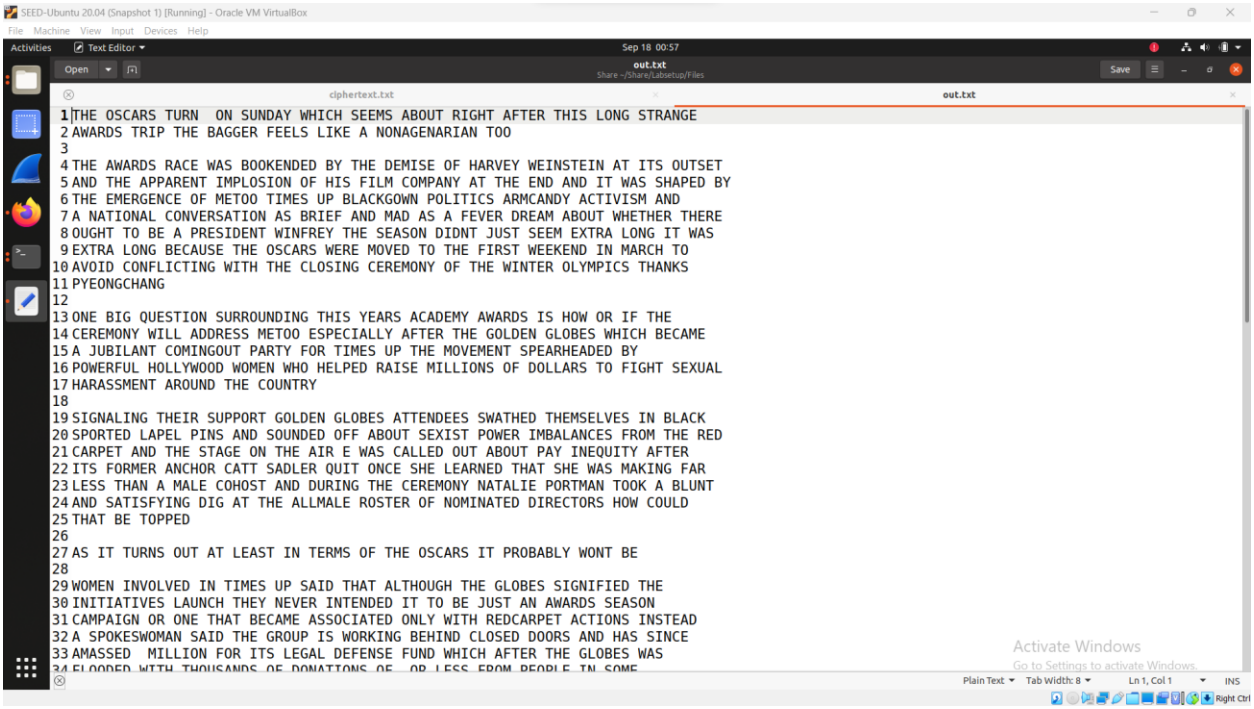
To solve this, the whole time I kept 3 tabs open: 1) output of "freq.py" 2) Bigram Frequency page from Wikipedia 3) Trigram Frequency page from Wikipedia.

Then I started replacing the cipher with upper case letter using "tr" command. I began with the most frequent trigrams "THE" and "AND" by replacing the top 2 trigrams in the cipher "ytn" and "vup". After seeing the output, I thought I could replace the letter "h" with "C" but it didn't make sense in the middle part of the paragraph. So, I decided to stick with decrypting the frequent

trigrams. Since the 3rd and 4th frequent diagrams has the same letters as “THE” and “AND”, I tried the 5th one which is “ING” and replaced the 3rd most frequent cipher trigram “mur”.

After doing this, when I looked at the output, some words started to make sense. So, I guessed some of the words and started brut forcing my guesses. While doing this, I was getting success on my guesses, and little by little it appeared to me easier and easier to guess and try. Then finally I decrypted all the letters successfully. My code and output are given below.





What I have observed:

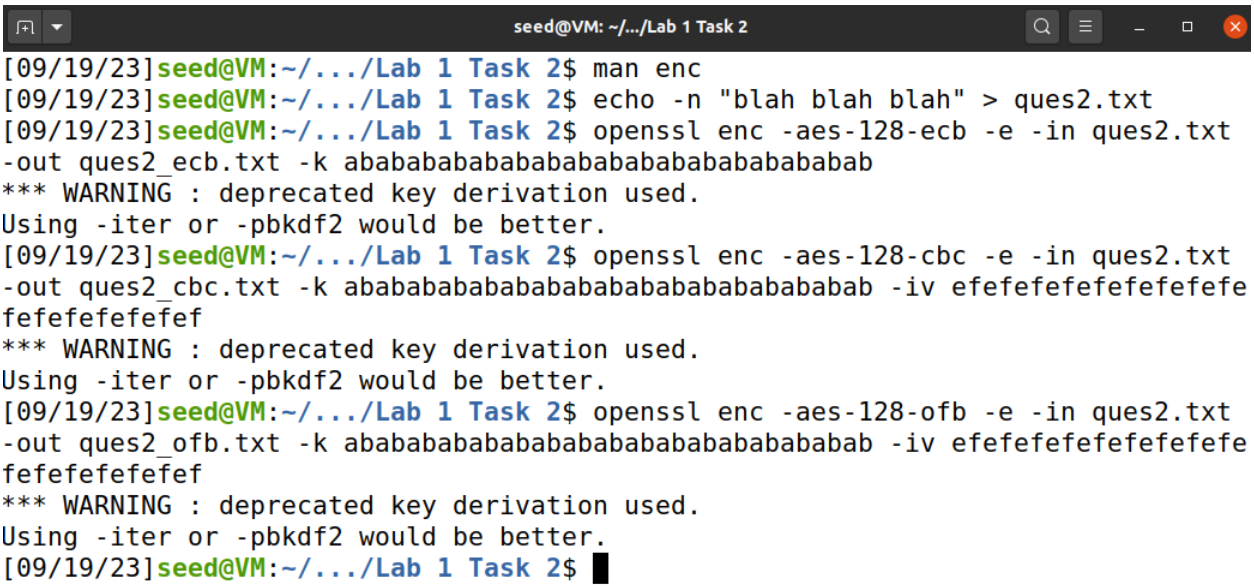
I observed that using the first 5 frequent trigrams when I decrypted the message partially, the whole cipher started to make sense to me little by little. So, I could just guess, try, and finally decrypt it easily.

My understanding is, decrypting a monoalphabetic cipher is easy if few of the characters can be decrypted by brut forcing. So, the security level of this cipher is low.

2. Task 2: Encryption using Different Ciphers and Modes

What I have done:

I first created a text file “ques2.txt” with random texts. Then I encrypted the file using 3 different ciphers aes-128-ecb, aes-128-cbc and aes-128-ofb, and observed the output files “ques2_ecb.txt”, “ques2_cbc.txt” and “ques2_ofb.txt” consecutively. My code is given below.



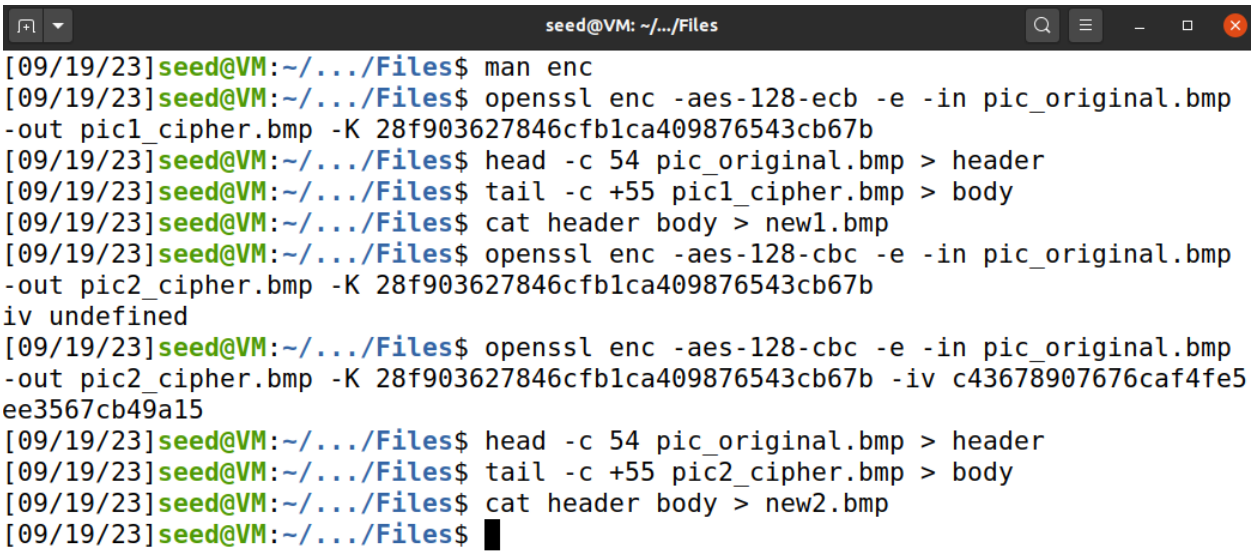
What I have observed:

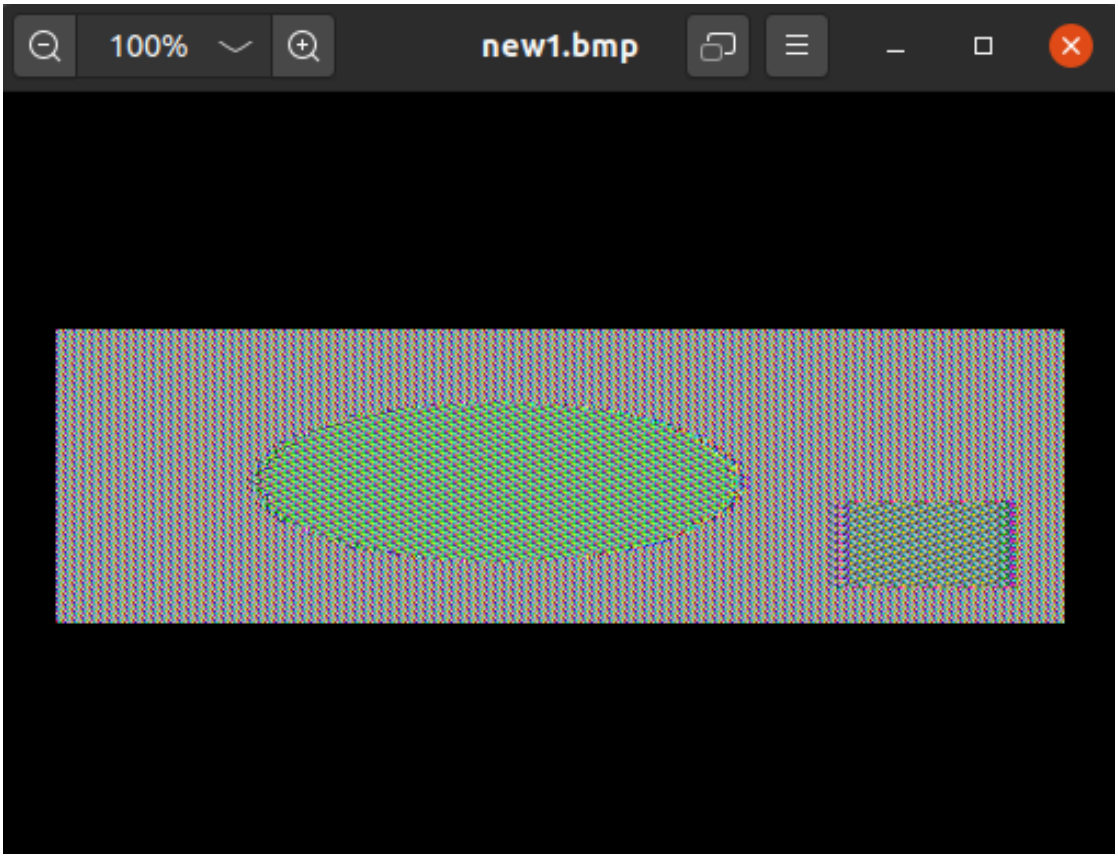
All the output files are encrypted and hard to read. They are not distinguishable. So, just by seeing the encrypted text in bare eyes, it is not possible to know which encryption method, key, iv was used. Also I came to know that ECB method do not accept iv. But CBC and OFB accepts iv for ciphering and deciphering.

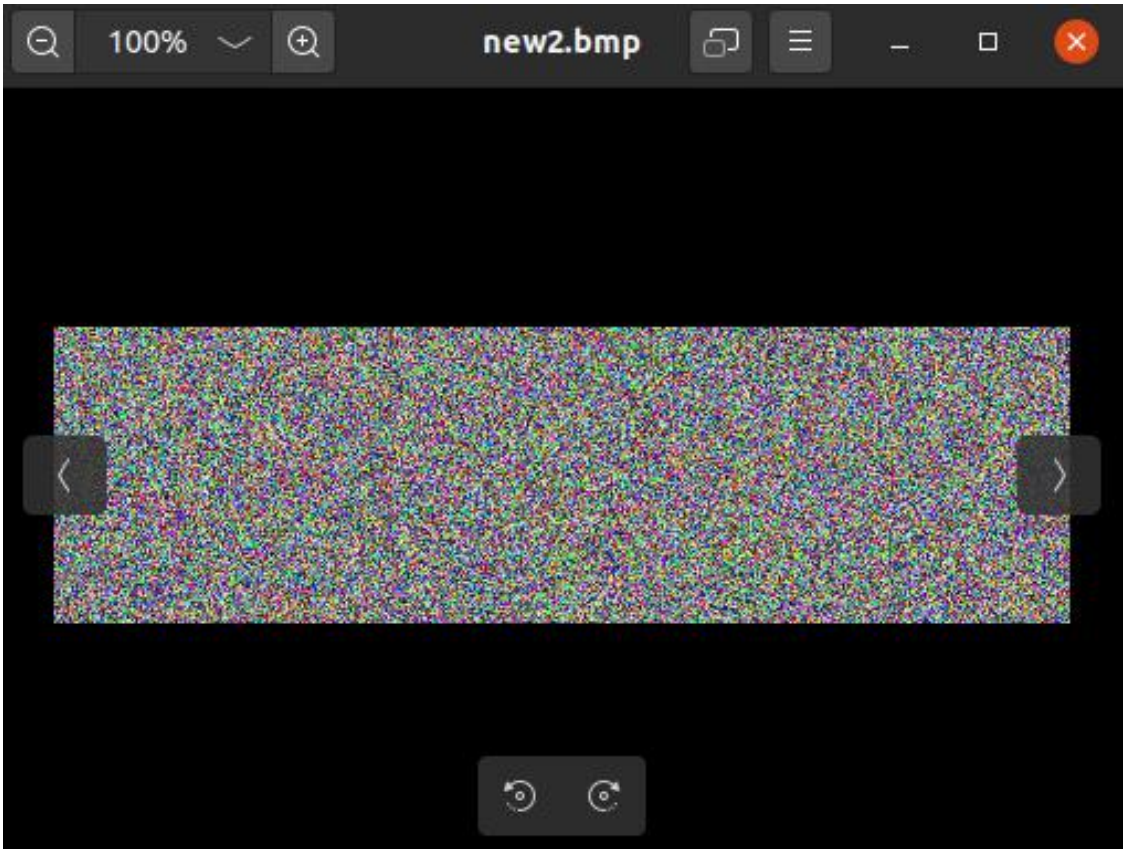
3. Task 3: Encryption Mode – ECB vs. CBC

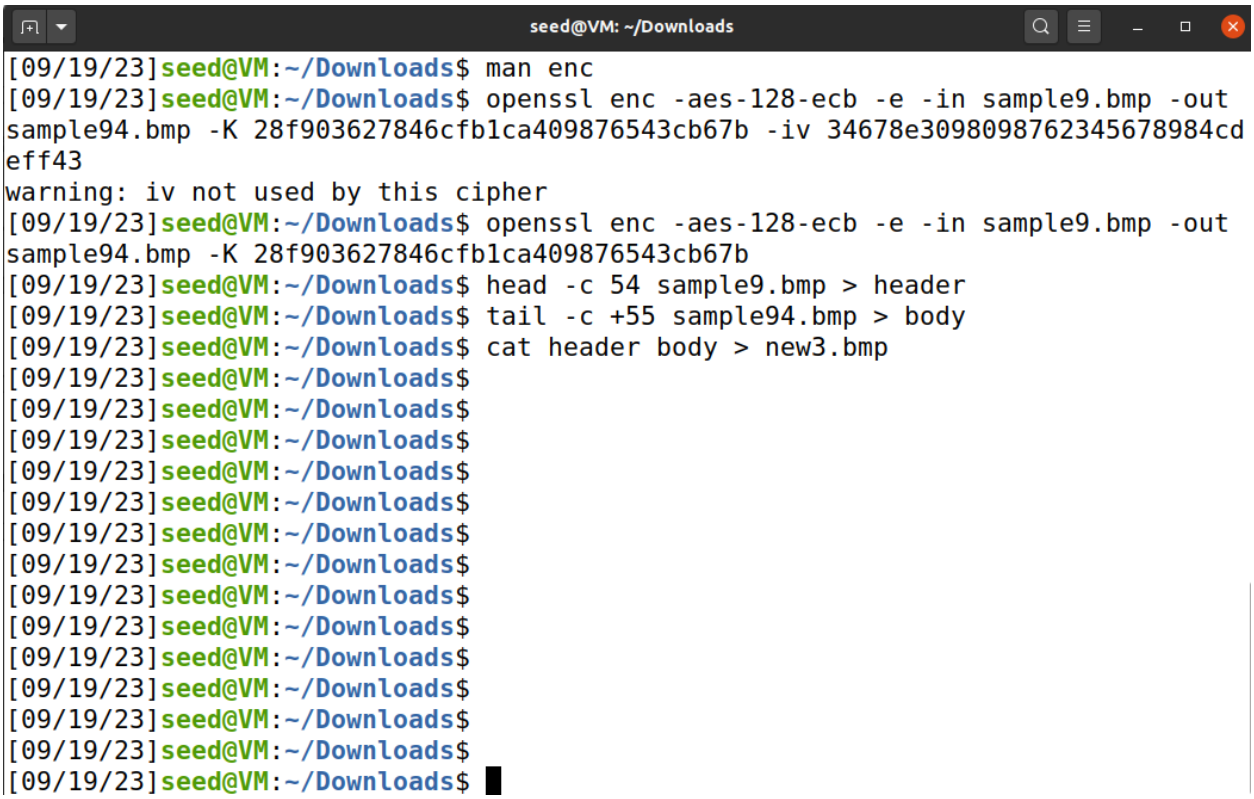
What I have done:

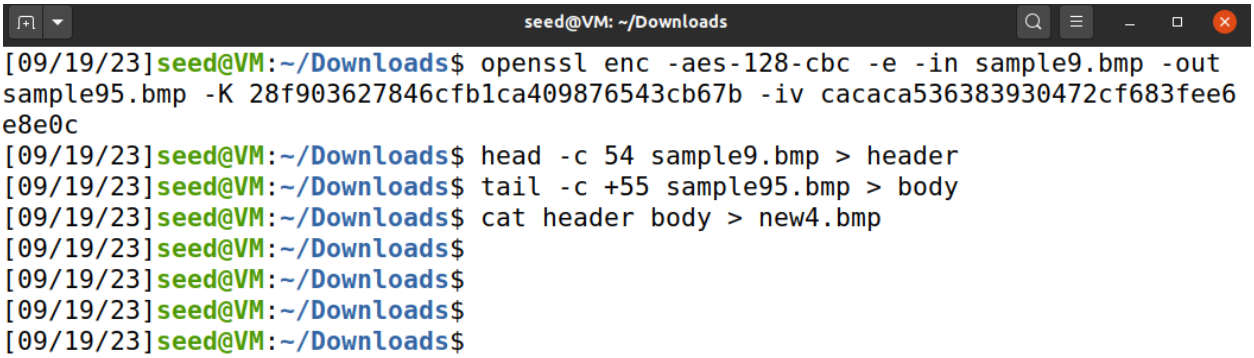
I encrypted both the pictures using ECB and CBC. Then took the header (first 54 bytes) of the original photo and added it with the tail (from 55th bytes to the rest) of encrypted file so the format ".bmp" stays intact and I can view the picture. Then I did the same thing with the picture downloaded by me. The code and outputs are below.

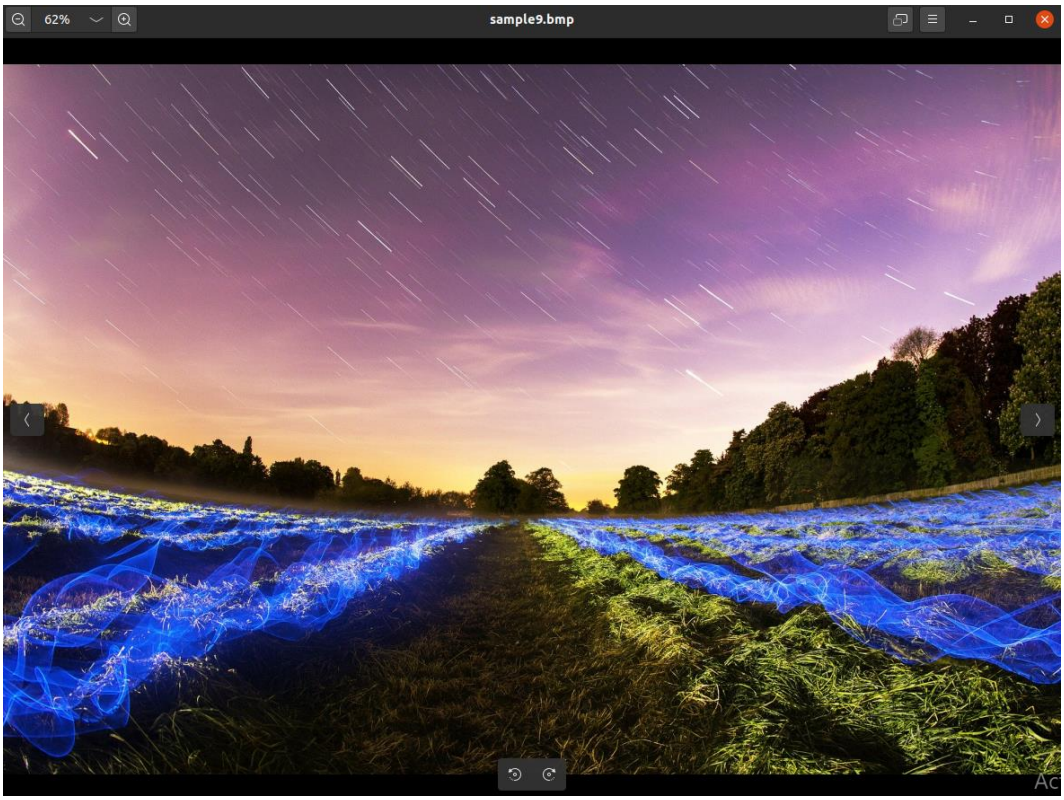



100%
new1.bmp


100%
new2.bmp


seed@VM: ~/Downloads
[09/19/23]seed@VM:~/Downloads$ man enc
[09/19/23]seed@VM:~/Downloads$ openssl enc -aes-128-ecb -e -in sample9.bmp -out
sample94.bmp -K 28f903627846cfb1ca409876543cb67b -iv 34678e3098098762345678984cd
eff43
warning: iv not used by this cipher
[09/19/23]seed@VM:~/Downloads$ openssl enc -aes-128-ecb -e -in sample9.bmp -out
sample94.bmp -K 28f903627846cfb1ca409876543cb67b
[09/19/23]seed@VM:~/Downloads$ head -c 54 sample9.bmp > header
[09/19/23]seed@VM:~/Downloads$ tail -c +55 sample94.bmp > body
[09/19/23]seed@VM:~/Downloads$ cat header body > new3.bmp
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$


seed@VM: ~/Downloads
[09/19/23]seed@VM:~/Downloads$ openssl enc -aes-128-cbc -e -in sample9.bmp -out
sample95.bmp -K 28f903627846cfb1ca409876543cb67b -iv cacaca536383930472cf683fee6
e8e0c
[09/19/23]seed@VM:~/Downloads$ head -c 54 sample9.bmp > header
[09/19/23]seed@VM:~/Downloads$ tail -c +55 sample95.bmp > body
[09/19/23]seed@VM:~/Downloads$ cat header body > new4.bmp
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$
[09/19/23]seed@VM:~/Downloads$


62%
sample9.bmp


62%
new3.bmp

What I have observed:

ECB mode did not perform well in decrypting the given picture. Although encrypted, the shapes are clearly visible. On the other hand, CBC performed really well and encrypted the photo so perfectly that no insight is there to be found.

But for encrypting the photo I chose, both CBC and ECB performed well. I believe it's because the picture I chose has mixed colors with nice shades and doesn't have much clearly defined geometrical shapes unlike the given file.

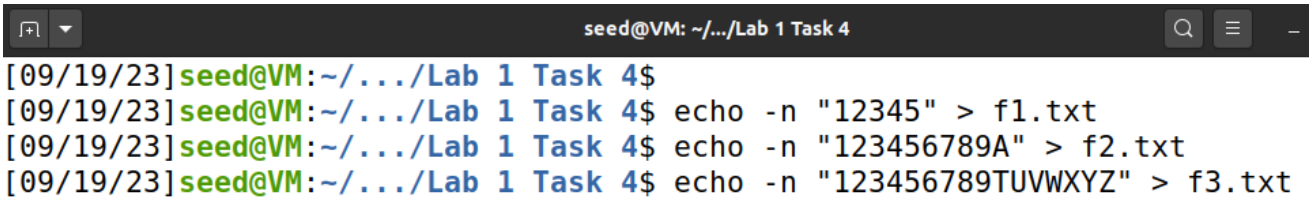
My conclusion is, although ECB can encrypt photos, it has clear limitations and does not perform well in every circumstance. But CBC is clearly a better mode of encrypting any photo.

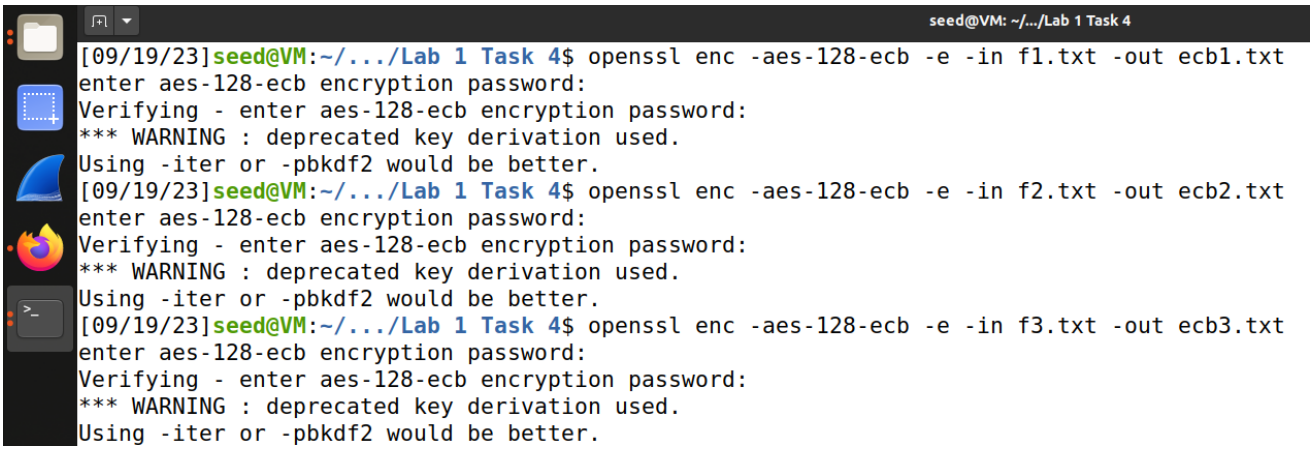
4. Task 4: Padding

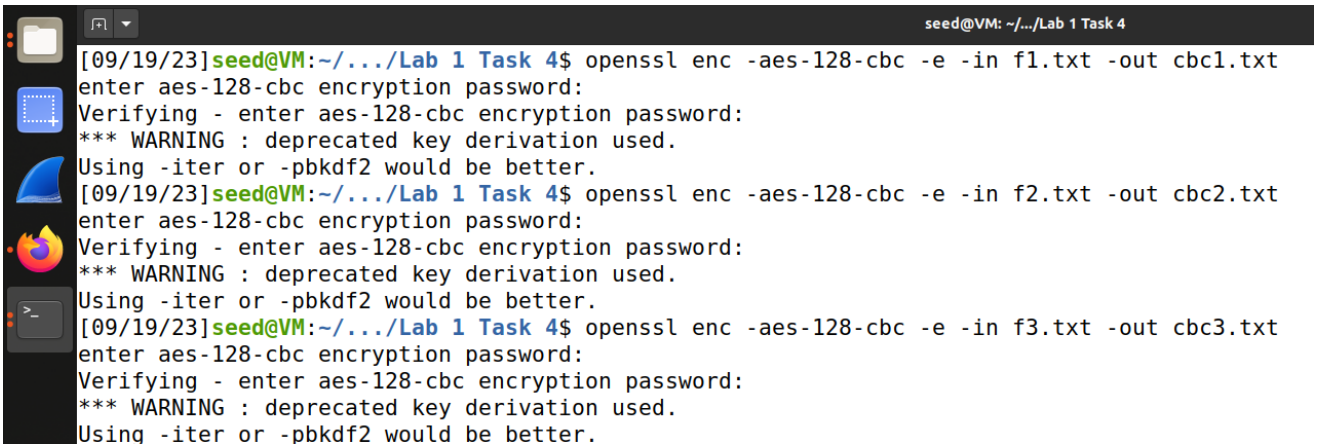
4.1)

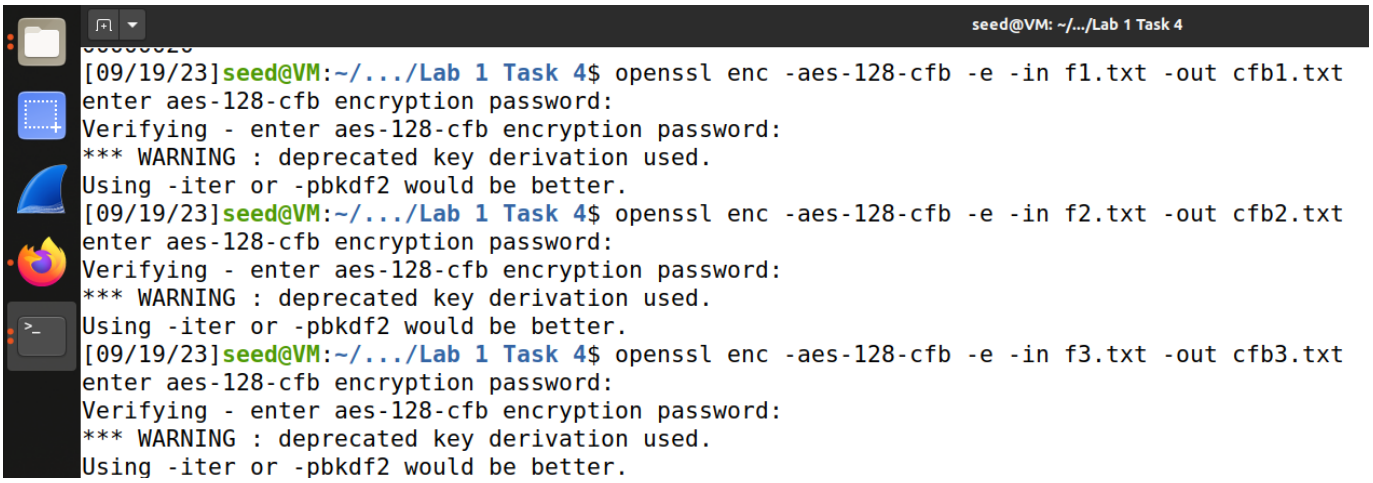
What I have done:

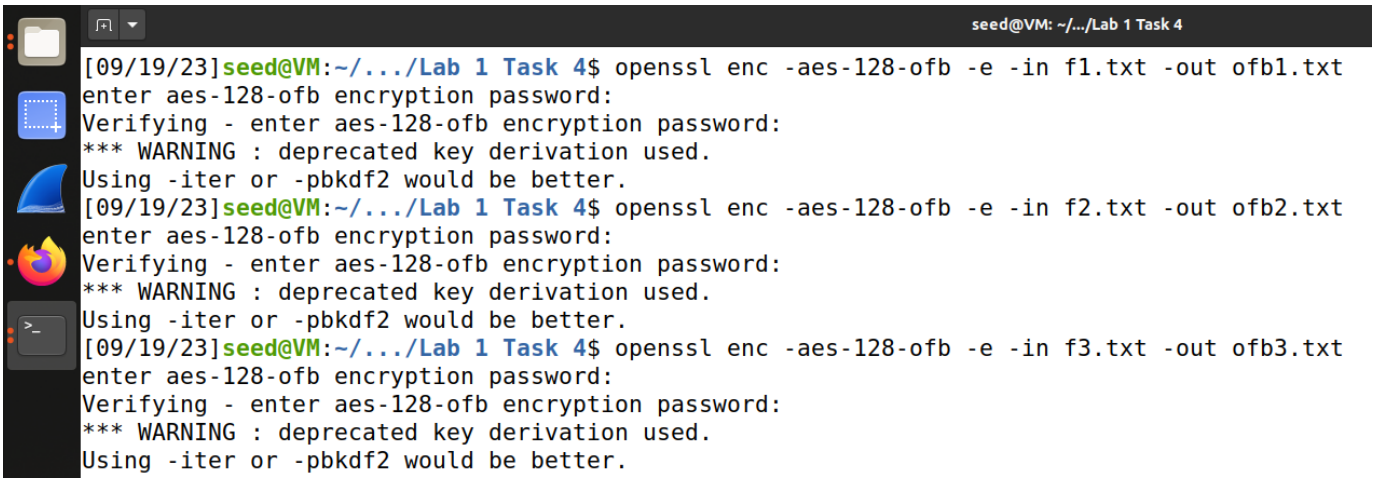
I created 3 files containing 5, 10, and 16 bytes for the next task (Task 4.2) and encrypted all of them using ECB, CBC, CFB, and OFB method of encryption. After encrypting I checked the size of all the files by running “l -ls”. I knew that after encrypting 16 bytes would be added to each of them due to the encryption. Code snippets are below.

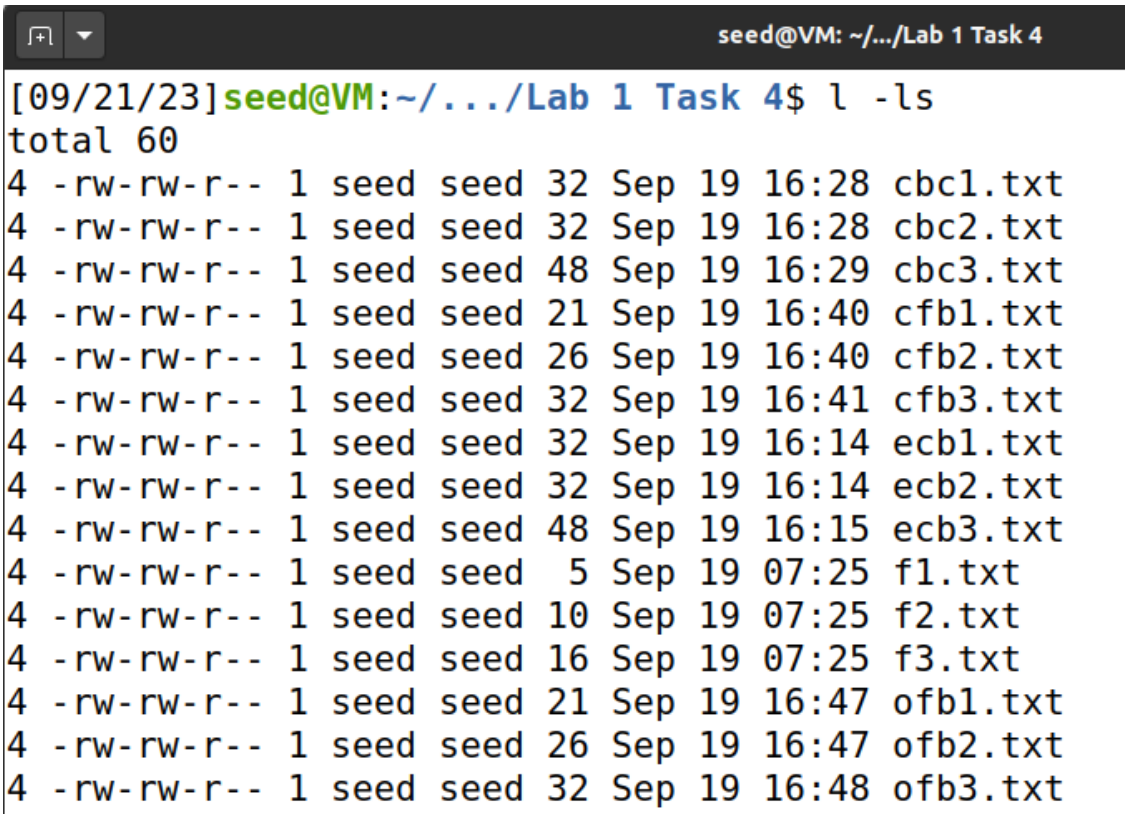







seed@VM: ~/.../Lab 1 Task 4
[09/19/23]seed@VM:~/.../Lab 1 Task 4$ openssl enc -aes-128-cfb -e -in f1.txt -out cfb1.txt
enter aes-128-cfb encryption password:
Verifying - enter aes-128-cfb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[09/19/23]seed@VM:~/.../Lab 1 Task 4$ openssl enc -aes-128-cfb -e -in f2.txt -out cfb2.txt
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.
[09/19/23]seed@VM:~/.../Lab 1 Task 4$ openssl enc -aes-128-cfb -e -in f3.txt -out cfb3.txt
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.


seed@VM: ~/.../Lab 1 Task 4
[09/19/23]seed@VM:~/.../Lab 1 Task 4$ openssl enc -aes-128-ofb -e -in f1.txt -out ofb1.txt
enter aes-128-ofb encryption password:
Verifying - enter aes-128-ofb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[09/19/23]seed@VM:~/.../Lab 1 Task 4$ openssl enc -aes-128-ofb -e -in f2.txt -out ofb2.txt
enter aes-128-ofb encryption password:
Verifying - enter aes-128-ofb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[09/19/23]seed@VM:~/.../Lab 1 Task 4$ openssl enc -aes-128-ofb -e -in f3.txt -out ofb3.txt
enter aes-128-ofb encryption password:
Verifying - enter aes-128-ofb encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.



What I have observed:

For ECB and CBC, I found out that, the input files of 5, 10, and 16 bytes has become 32, 32, and 48 bytes consecutively. That is because paddings have been added to each of them to make it multiple of a block size. Since ECB and CBC are block cipher modes, they encrypt data block by block and need padding.

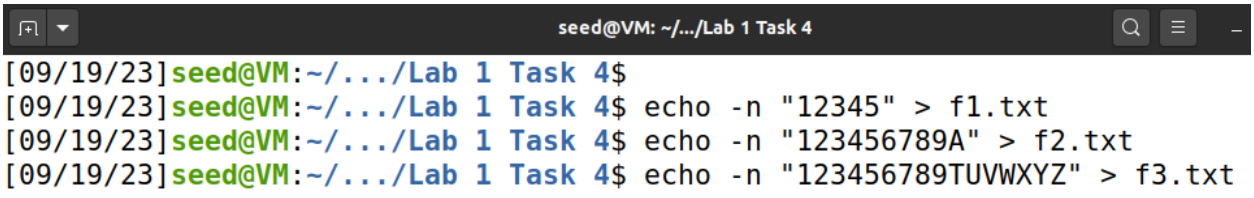
On the other hand, CFB and OFB are stream ciphers which encrypt bit by bit. As they do not need to match the length with any block size, no padding is required. So, after encrypting, the input files of 5, 10, and 16 bytes have become 21, 26, and 32 bytes consecutively by adding only 16 bytes for encryption. No padding was added.

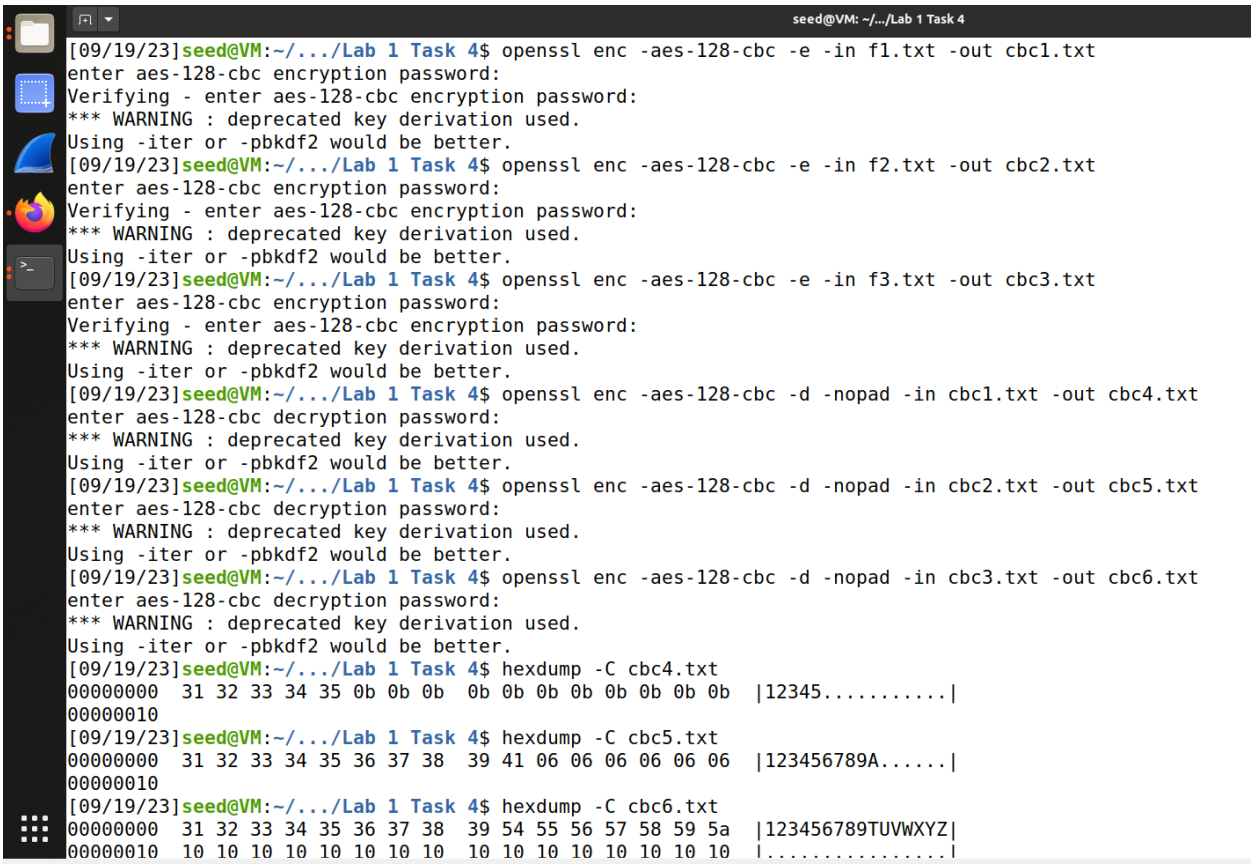
4.2)

What I have done:

I created 3 files containing 5, 10, and 16 bytes in advance during the previous task (Task 4.1) and encrypted all of them using ECB, CBC, CFB, and OFB method of encryption. After encrypting I checked the size of all the files by running “l -ls” and mentioned my finding in the above section.

For this task I encrypted them using CBC once again and decrypted those 3 encrypted files. I used “-nopad” in the command line to make the padding visible in the decrypted files. Then I viewed the decrypted files using “hexdump”. Code snippets are below.





What I have observed:

In the above section (Task 4.1) I viewed the file sizes after CBC encryption and understood that CBC mode adds padding during encryption.

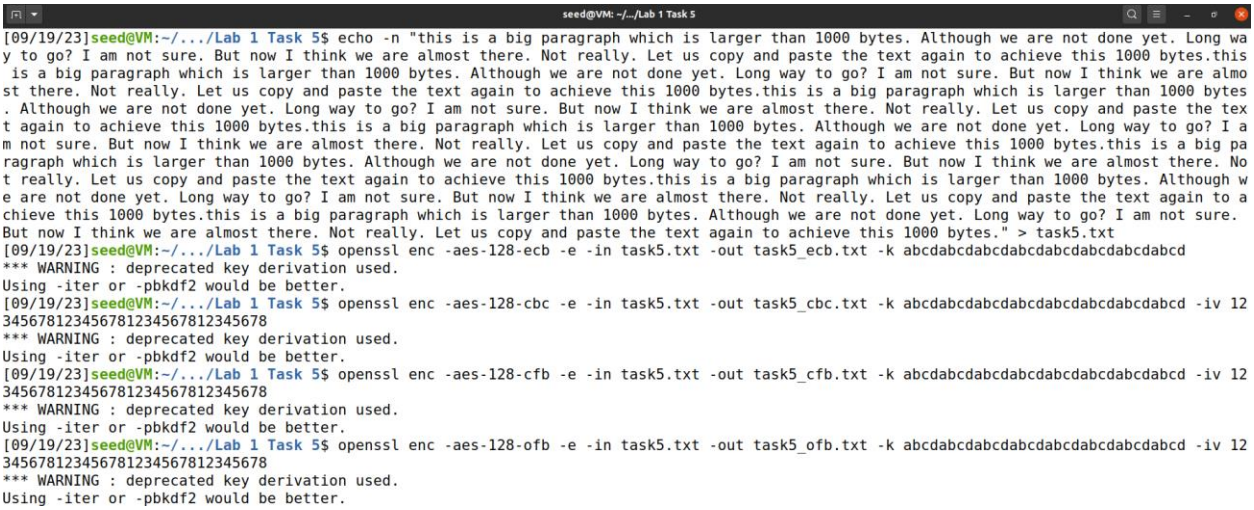
In this task, after decrypting and viewing them using “hexdump”, I can see the paddings clearly. Suppose for the first file, 11 bytes of padding was added to fill up the block size of 16 bytes. Since 11 bytes were added, all the bytes were there as “0b” which means “11” in hexadecimal. This fact seems interesting to me. Also, it added “06” for every byte of padding where required

padding was 6 bytes. And for the 3rd file, as the block size was full, it added a whole new block of 16 bytes padding using “10” for every bytes which means “16” in hexadecimal.

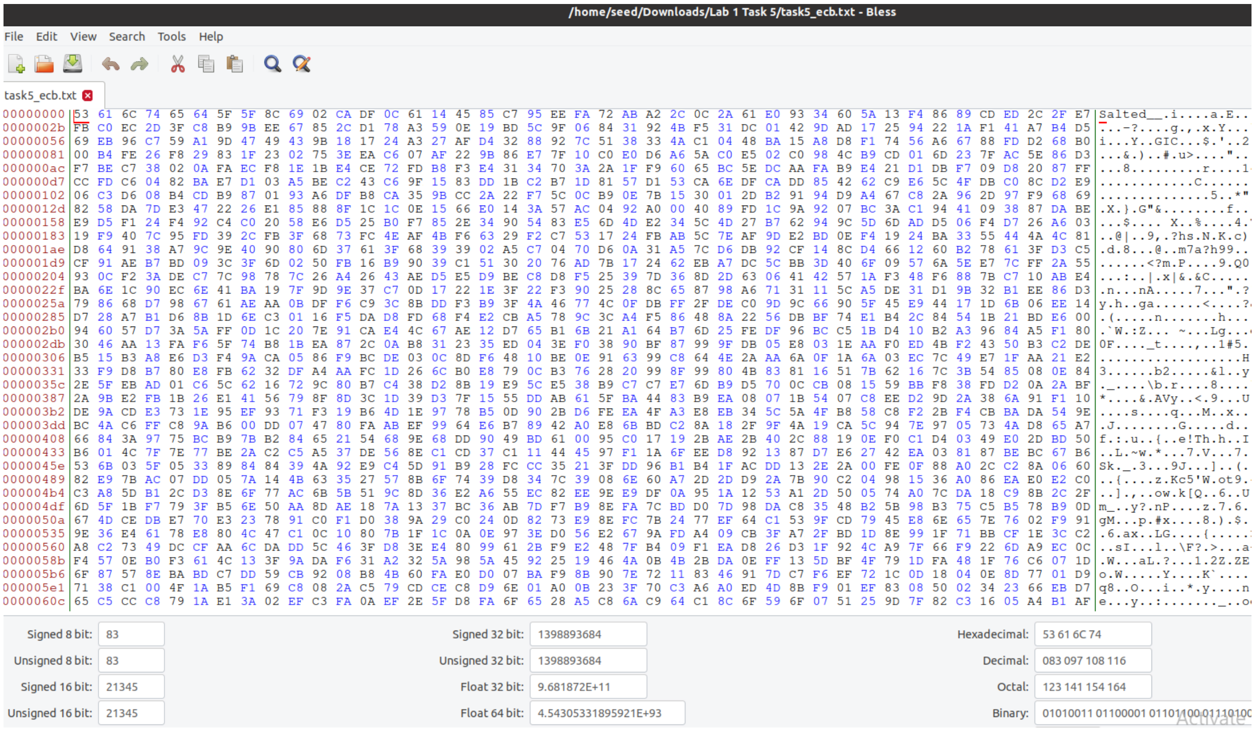
5. Task 5: Error Propagation – Corrupted Cipher Text

What I have done:

Created a text file bigger than 1000 bytes. Encrypted the file using AES-128 cypher of 4 modes: CBC, ECB, CFB, and OFB. Then I opened each of the encrypted files on “bless” hex editor and changed the 55th byte for all of them to make them corrupted. Then decrypted the corrupted files again to observe what happened. Code snippets and snapshot of input / output files are below.



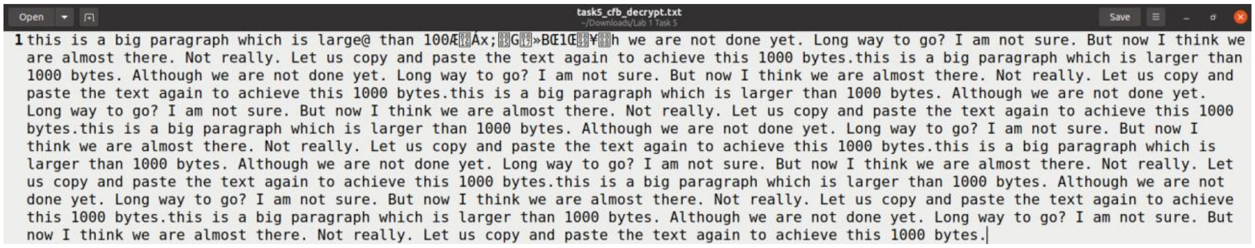
```
[09/19/23]seed@VM:~/.../Lab 1 Task 5$ bless task5_ecb.txt
Gtk-Message: 18:08:09.845: Failed to load module "canberra-gtk-module"
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find file "/home/seed/.config/bless/export_patterns"
[09/19/23]seed@VM:~/.../Lab 1 Task 5$ bless task5_cbc.txt
Gtk-Message: 18:09:55.815: Failed to load module "canberra-gtk-module"
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find file "/home/seed/.config/bless/export_patterns"
[09/19/23]seed@VM:~/.../Lab 1 Task 5$ bless task5_cfb.txt
Gtk-Message: 18:11:56.914: Failed to load module "canberra-gtk-module"
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find file "/home/seed/.config/bless/export_patterns"
[09/19/23]seed@VM:~/.../Lab 1 Task 5$ bless task5_ofb.txt
Gtk-Message: 18:13:10.436: Failed to load module "canberra-gtk-module"
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find a part of the path '/home/seed/.config/bless/plugins'.
Could not find file "/home/seed/.config/bless/export_patterns"
```

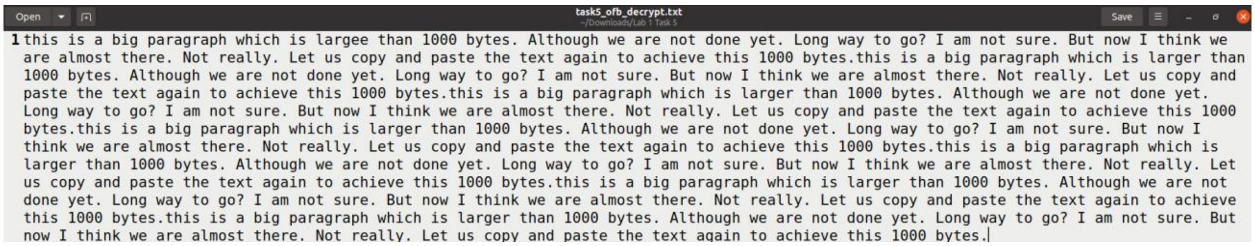



```
[09/19/23]seed@VM:~/.../Lab 1 Task 5$ openssl enc -aes-128-ecb -d -in task5_ecb.txt -out task5_ecb_decrypt.txt -k abcdabcdabcdabcdabcdabcdabc
dabcd
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[09/19/23]seed@VM:~/.../Lab 1 Task 5$ openssl enc -aes-128-cbc -d -in task5_cbc.txt -out task5_cbc_decrypt.txt -k abcdabcdabcdabcdabcdabcdabc
dabcd -iv 12345678123456781234567812345678
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[09/19/23]seed@VM:~/.../Lab 1 Task 5$ openssl enc -aes-128-cfb -d -in task5_cfb.txt -out task5_cfb_decrypt.txt -k abcdabcdabcdabcdabcdabcdabc
dabcd -iv 12345678123456781234567812345678
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[09/19/23]seed@VM:~/.../Lab 1 Task 5$ openssl enc -aes-128-ofb -d -in task5_ofb.txt -out task5_ofb_decrypt.txt -k abcdabcdabcdabcdabcdabcdabc
dabcd -iv 12345678123456781234567812345678
*** WARNING : deprecated key derivation used.
```

1 this is a big paragraph which isÕõ~l,ÕH_Zçµ(+0 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.

1 this is a big paragraph which isp2%òïœí`Úá÷fô0 byte7. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.this is a big paragraph which is larger than 1000 bytes. Although we are not done yet. Long way to go? I am not sure. But now I think we are almost there. Not really. Let us copy and paste the text again to achieve this 1000 bytes.





What I have observed:

For ECB and CBC, I found out that the whole block that contained 55th byte got corrupted in the output files as well. The parts that were decrypted from 49th to 64th byte of the cipher got no meaning in the output.

After that, for ECB the rest of the next blocks were okay.

But for CBC, the next block of the corrupted block was also a bit corrupted since, for CBC decrypting the next block depends on the current block like a chain effect. So, the error propagates. Since the previous block was corrupted, the next block was also partially corrupted. But after that, the error was mitigated and gave correct output.

As CFB and OFB are stream cyphers, they encrypt and decrypt byte by byte. So, a corrupted byte doesn't change the whole block of the corrupted byte.

For OFB, only the corrupted 55th byte was faulty in the decrypted output. But no other byte from before or after were affected.

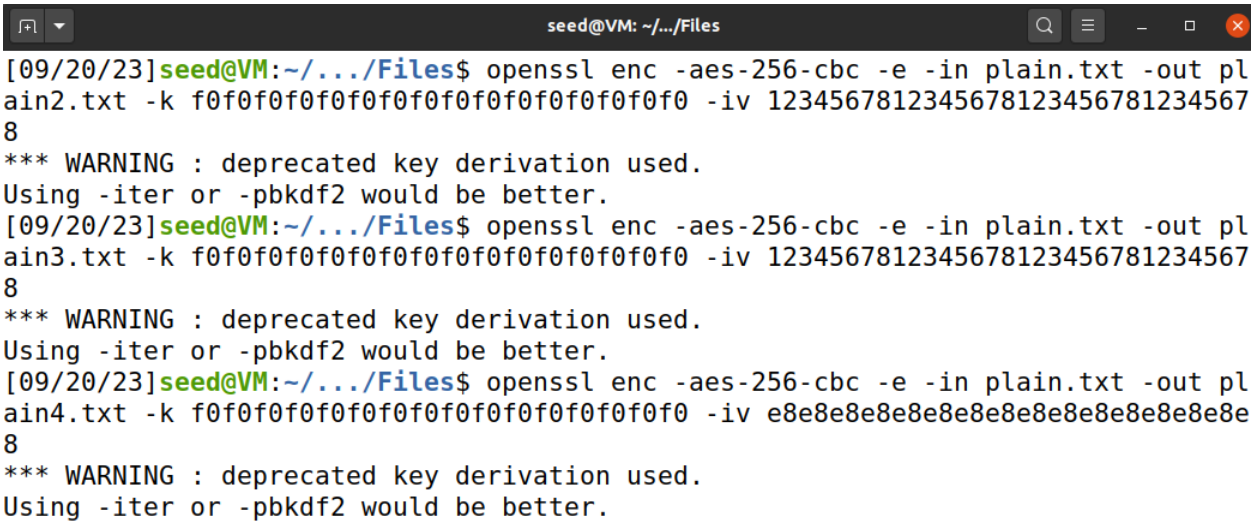
For CFB, the output of the corrupted 55th byte was faulty, and other characters of that block were okay. However, the next block totally got spoiled, because in CFB, during decryption there is an XOR relation with the previous block. So, if there is some error in the previous block, then the whole block got corrupted. But from the next blocks it gets okay again.

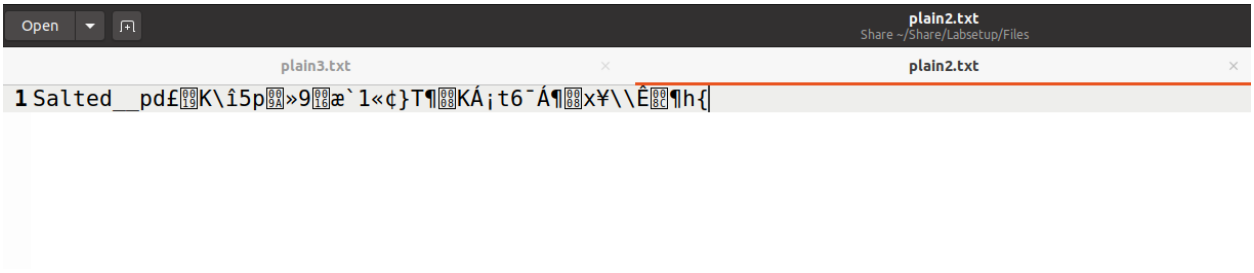
6. Task 6: Initial Vector (IV) and Common Mistakes

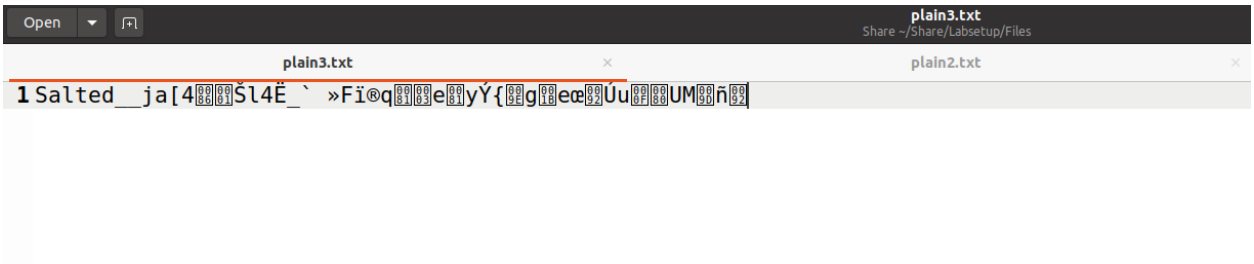
Task 6.1. IV Experiment

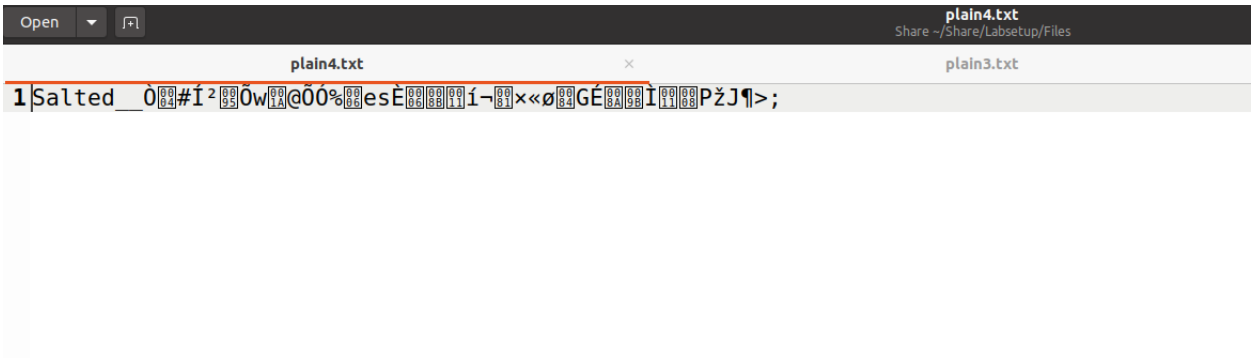
What I have done:

Here I encrypted a plaintext once with different IV and the next time with same IV. The key was same.









What I have observed:

I observed that whether I used the same IV or different IV to encrypt the same plain text while having the same key, the encrypted text looked difficult to understand for both. So, using same IV for small plain text doesn't seem less secured in naked eyes.

Task 6.2. Common Mistake: Use the Same IV

What I have done:

Here I took two ciphertext (that use same IV) and one known text as inputs. Did XOR to the known plain text with the known cipher text. after converting them to bytes. Then run XOR operation with the output and the ciphertext from which I want to find out the plain text.

In [51]:
```
#!/usr/bin/python3
# XOR two bytearrays

# Given Ciphertext 1
c1 = 'a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159'

# Given Ciphertext 2
c2 = 'bf73bcd3509299d566c35b5d450337e1bb175f903fafc159'

def xor(first, second):
    return bytearray(x^y for x,y in zip(first, second))

MSG = "This is a known message!"

# Convert ascii/hex string to bytearray
D1 = bytes(MSG, 'utf-8')
D2 = bytearray.fromhex(c1)
D3 = bytearray.fromhex(c2)

r1 = xor(D1, D2)
r2 = xor(r1, D3)

print(r2)
```

```
bytearray(b'Order: Launch a missile!')
```

What I have observed:

Found out the unknown plain text easily since I knew the other cipher's IV and for this encryption also the IV was the same. Therefore, using the same IV for two encryptions is dangerous and makes the hacker's work easy.

Task 6.3. Common Mistake: Use a Predictable IV

What I have done:

Here the server worked as BOB. I did XOR operations like the previous task to find a plain message by knowing the IV. We XORed the both used IV and then XORed the result with the next IV to find the plain text. As we knew two possible plain texts before, we used them to brut force. And when we found their hexadecimal plain text, we inputed them in the server and got the cipher, by mathcing with we found out that the plain text is “Yes”.

In [53]:

```
Bobs = "91eb247d9e7d3a4ded1fb2977db3c4a4"
IV_used = bytearray.fromhex("adc46ac62ede44ffbb1ee245fd108f1a")
IV_next = bytearray.fromhex("ac8235302fde44ffbb1ee245fd108f1a")

def xor(first, second):
    return bytearray(x^y for x,y in zip(first, second))

#Hex of Yes + padding = "5965730d0d0d0d0d0d0d0d0d0d0d0d0d"
Yes = bytearray.fromhex("5965730d0d0d0d0d0d0d0d0d0d0d0d0d")

#Hex of No + padding = "4E6F0e0e0e0e0e0e0e0e0e0e0e0e0e0e"
No = bytearray.fromhex("4E6F0e0e0e0e0e0e0e0e0e0e0e0e0e0e")

a = xor(IV_used, Yes)
b = xor(IV_used, No)

from binascii import hexlify

c = xor(a, IV_next)
#print(c)

d = xor(b, IV_next)
#print(d)

cc = hexlify(c)
dd = hexlify(d)

print(cc)
print(dd)
#x xor IV_next = a
```

```
b'58232cfb0c0d0d0d0d0d0d0d0d0d0d0d'
b'4f2951f80f0e0e0e0e0e0e0e0e0e0e0e'
```

What I have observed:

Knowing IV of a cipher is dangerous, even if it is not the same with any previous IV. It can be found out by using brut force attacks.

7. Task 7: Programming using the Crypto Library

What I have done:

Although all the steps are given in the comments of the code, I explained here why I did them.

At first, I installed pycryptodome to import packages for building AES encryption/decryption models that work using key, iv, and input. Also converted all hexadecimals to bytes for using them inside the decryption function.

I wrote a function called check_key() to brut force all the possible keys from the given file (words.txt).

Before that, I added paddings to the possible keys inside the file to make it equal to a complete block. Then converted all of them to hexadecimal. Then iterated though all the values and used them as possible keys one by one using check_key(). In the end found the actual key in the output which is “foundation”.

In [4]:
```
#installed pycryptodome to import functions from
pip install pycryptodome
```

```
Collecting pycryptodome
  Downloading pycryptodome-3.19.0-cp35-abi3-win_amd64.whl (1.7 MB)
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.19.0
Note: you may need to restart the kernel to use updated packages.
```

In [43]:
```
# AES is required to build the cipher decryptor
# Unhexlify is required to convert the hexadecimal values to bytes

from Crypto.Cipher import AES
from binascii import unhexlify
```

In [52]:
```
# Used unhexlify function to convert the hexadecimal values to bytes

iv = unhexlify('0102030405060708090a0b0c0d0e0f')
ciphertext = unhexlify('76d4a3a35b32dc5a3de44593f659c8713919425a1352e7bcb087952e2b791911')
```

In [56]:
```
# Imported word.txt file in the variable "file"

file=open("G:\CSE 565 labs\Lab 1 task 7\words.txt","r")

# Wrote this function to find out our desired key
# Here we took all the possible keys as the parameter of this function.
# This function decrypts the encrypted string and compare with our expected string

def check_key(hexa):
    key = unhexlify(hexa)
    decrypt_cipher = AES.new(key, AES.MODE_CBC, iv)
    plain_text = decrypt_cipher.decrypt(ciphertext)

    c = b'This is a top secret.'
    b = plain_text[:21]
    if c == b:
        print(key)

# This for loop is to iterate every possible key from words.txt file
# Added '#' to make the length 16 bytes
#Then converted it to hexa decimal

for i in file:
    a = len(i)
    j = i[:a-1]
    i = "".join([j,"################"])
    i = i[:16]
    st = i.encode('utf-8')
    hexa_key = st.hex()
    check_key(hexa_key)

# Finally found out desired key "foundation######"
```

```
b'foundation######'
```

What I have observed:

From this experiment I understood that we should never set a word of a language as a key. Keys should be random letters and symbols to make it secure by making it hard to guess.