CPS 472/572 Fall 2024

Prof: Zhongmei Yao

Lab 1 Report

Dinesh Kumar Kala (kalad1@udayton.edu)

Student ID: 101745354

1. Overview:

From this lab, I learned about secret-key encryption, including different algorithms and modes, the importance of padding, and how IVs should be unpredictable. I practiced using encryption tools and writing programs to encrypt and decrypt data, and discovered common mistakes that can weaken encryption. The video provided by the professor also helped deepen my understanding of secret-key encryption.

Takeaways: This lab taught me about secret-key encryption, including encryption algorithms, modes, paddings, and IVs. I learned to use encryption tools, write programs for encryption and decryption, and recognize common mistakes that can weaken security.

2. Lab Environment

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ dcbuild
Building oracle-server
Step 1/8 : FROM handsonsecurity/seed-ubuntu:dev AS builder
 ---> 89212aee292b
Step 2/8 : COPY .  /oracle
 ---> Using cache
 ---> 4a5991b2afc2
Step 3/8 : WORKDIR /oracle
 ---> Using cache
 ---> b4d6a5aaaacb
Step 4/8 : RUN make
 ---> Using cache
 ---> 16019af9f4d2

Step 5/8 : FROM handsonsecurity/seed-ubuntu:small
 ---> 1102071f4a1d
Step 6/8 : COPY --from=builder /oracle/build /oracle
 ---> Using cache
 ---> 0d5e8a343bcd
Step 7/8 : WORKDIR /oracle
 ---> Using cache
 ---> f99f4810d361
Step 8/8 : CMD ./server
 ---> Using cache
 ---> dc9fa2e12ef6

Successfully built dc9fa2e12ef6
Successfully tagged seed-image-encryption:latest
```

To set up the lab environment, I downloaded and unzipped the Labsetup.zip file to my VM. Inside the Labsetup folder, I used the `**docker-compose.yml**` file to configure the container. For the initial setup, I ran `**docker-compose build**` to build the container images and `**docker-compose up**` to start the container. I also used `**docker-compose down**` to stop and remove the containers when needed. Since it was my first time setting up a SEEDlab environment using containers, I read the user manual for detailed instructions.

Takeaway: I learned how to effectively manage Docker containers for running the encryption oracle. To build the container, I used the docker-compose build command, which sets up the environment according to the specifications in the docker compose.yml file. To start the container and get the environment running, I used docker-compose-up. When I needed to stop and remove the container, docker-compose down was the command I used

3. Task1: Frequency Analysis

Step-1:

To do this task, first i need the Plain text so i went to google and got some text related to the network security and saved in a file called article.txt

```
[09/02/24]seed@VM:~$ cd Desktop
[09/02/24]seed@VM:~/Desktop$ cd Labsetup
[09/02/24]seed@VM:~/.../Labsetup$ cd Dinesh-Kala
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr [:upper:] [:lower:] < article.txt > lowercase.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr -cd '[a-z][\n][:space:]' < lowercase.txt > plaintext.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr 'abcdefghijklmnopqrstuvwxyz' 'wnmtgucrfhovjilxaqeskdpybz'
 < plaintext.txt > ciphertext.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat ciphertext.txt
dkvigqwnfvfsfge jwib igsplqo ebesgje rwdg vllxrlvge lu lig ofit lq wilsrgq srgeg dkvig
qwnfvfsfge jwb ngvlic sl srg ebesgj wvel fs rwe srg xleefnfvfsb ng ulqjgt nb srg i
gcvfcgimg lu srg jwiwcgjgis srgeg dkvigqwnfvfsfge mwi ng kegt nb rwmogqe sl tl dwqfl
ke wsswmoe fimvktfic srg xweeplqt tgsgmsfli fi wttfsfli eluspwqg wsswmo fe w mljjli
jgsrlt wjlic srgj rwmogqe rwdg wnfvfsb sl cwfi srg ekxgqkegq qfcrse lu srg mljxksgq fvvg
cwvvb wit mlisqlv ldgq fs mljxvgsgvb ngeftge srg lxgqwsflie lu ufvge fs mwi wvel mwxskqg fjwcge l
i srg tgeoslx wmakfqg srg xweeplqte lu vlse lu wxxvfmwsflie wit eljg lsrgq lxgqwsflie mvfgis ef
tg wit egqdgq eftg wqg srg spl sbxge lu srleg eluspwqg prgi rwmogqe wsswmo srgb pfv
v vlc fi kefic mvfgiseftg xqlcqwje jlnfvg srqgwsejwqsxrlig dfqkege xleg w srqgws sl mlqxlqw
sg igsplqoe wit xgqeliwv fiulqjwsfli uqlj srg xgqexgmsfdg lu sltwbe igsplqo tgdgvlxjgi
s sqgit wib gvgmsqlifm xqltkms jwb ng mliigmsgt sl srg igsplqo wit srg uwms srws s
rg igsplqo fe gdgqbprgqg erlpe srws wsswmoe wqg wvel gdgqbprgqg fi wttfsfli srgqg pf
vv ng vllxrlvge fi rwqtpwqg lxgqwsfic ebesgj igsplqo wmmgee gakfxjgis wit wxxvfmwsfli
ebesgj kefic srg eslqwcg exwmg lu jlnfvg tgdfmge srg vfdfic exwmg wit fimknwsfli xgqflt lu jwvfmf
lke mltg pfvv ng kixqgtfmswnvg gvgmsqljwcigsfm fisgqugqgimgrfcrdlvswcg pfqge qwtfl pwdg sqwie
jfssfic wisgiiwe jfmqlpwdg vfig rfcruqgakgimb gvgmsqlifm gakfxjgis wit el li pfvv xqlt
```

I opened the command the prompt and went to the desired location where to run, execute and save my files. For this i created separate folder with my name. After creating a .txt file, using

Tr [:upper:] [:lower:] <article.txt> lowercase.txt i convert the text into lowercase and using “-cd ‘[a-z] [\n] [:space:]’ i gave spacing to my text and saved in plaintext.txt. Now my text is in lowercase and with a spacing. After that i used .py file to generate a random sequence for the lowercase alphabets and i used that output here to replace with the original document. So if we compare a,b,c,d with the random output w,n,m,t which means
a -> w, b -> n, c -> m, d -> t. In the same way the original text will get replaced with the random output.

Step-2:

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$
[09/02/24]seed@VM:~/.../Dinesh-Kala$ freq.py
-------------------------------------
1-gram (top 20):
g: 186
s: 131
l: 107
f: 100
w: 97
i: 94
e: 89
q: 82
v: 58
m: 57
r: 54
t: 40
j: 39
x: 35
c: 28
k: 26
p: 25
u: 24
b: 22
d: 18
-------------------------------------
2-gram (top 20):
sr: 35
gq: 32
```

Now, after replacing the text, i used the freq.py to count the the repeated alphabets for single frequencies, bigram, trigram (ex: a, to, the). And the above the result.

Step-3

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr 'srg' 'THE' < ciphertext.txt > out.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr 'gq' 'ER' < out.txt > out1.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr 'l' 'O' <out1.txt> out2.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr 'igs' 'NET' <out2.txt> out3.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr 'f' 'I' <out3.txt> out4.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tr 'w' 'A' <out4.txt> out5.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat out5.txt
dkvNERAnIvITIEe jANb  NETpORo  ebeTEje  HAdE  vOOxHOvEe  Ou  ONE  oINt  OR  ANOTHER  THEeE  dkvNERAnIvI
eO  IT  HAe  THE  xOeeInIvITb  nE  uORjEt  nb  THE  NEcvIcENmE  Ou  THE  jANAcEjENT  THEeE  dkvNERAnIvI
tO  dARIOke  ATTAmoe  INmvktINc  THE  xAeepORt  tETEmTION  IN  AttITION  eOuTpARE  ATTAmo  Ie  A  mOjjC
 AnIvITb  TO  cAIN  THE  ekxERkeER RIcHTe Ou THE mOjxkTER IvvEcAvvb ANt mONTROv OdER IT mOjxvETEvb nEeI
O mAxTkRE IjAcEe ON THE tEeoTOx AmakIRE THE xAeepORte Ou vOTe Ou AxxvImATIONe ANt eOjE OTHER OxERATIONe
  THE  TpO  TbxEe  Ou  THOeE  eOuTpARE  pHEN  HAmoERe  ATTAmo  THEb pIvv vOc IN keINc mvIENTeItE xROcRA
OeE  A  THREAT  TO  mORxORATE  NETpORoe  ANt  xEReONAv  INuORjATION  uROj  THE  xERexEmTIdE  Ou  TOtAbe
EmTRONIm  xROtkmT  jAb  nE  mONNEmTEt  TO  THE  NETpORo  ANt  THE  uAmT  THAT  THE  NETpORo  Ie  EdERbp
 EdERbpHERE  IN  AttITION  THERE  pIvv  nE  vOOxHOvEe  IN  HARtpARE  OxERATINc  ebeTEj  NETpORo  AmmEee
eINc THE eTORAcE exAmE Ou jOnIvE tEdImEe THE vIdINc exAmE ANt INmknATION xERIOt Ou jAvImIOke mOtE pIvv
uERENmEHIcHdOvTAcE  pIREe  RAtIO  pAdE  TRANejITTINc  ANTENNAe  jImROpAdE  vINE  HIcHuREakENmb  EvEmTRC
tkmE  EvEmTROjAcNETIm  INTERuERENmE  eIcNAve  THEeE  EvEmTROjAcNETIm  INTERuERENmE eIcNAve pIvv tEeTROt
TIm iEtIki THke AuuEmTINc THE NETpORo eEmkRITb
```

Now, to check it write or wrong, i tried to replace it back for the highly repeated 1-gram,2-gram, 3-gram. To identify the changes i highlighted them as a uppercase. srg -> THE, gq -> ER, l -> O. and after performing some changes, it looks like it changes to ciphertext and changes back to original text.

```python
import random
s = "abcdefghijklmnopqrstuvwxyz"
list = random.sample(s, len(s))
key = ''.join(list)
print(key)
```

```
wnmtgucrfhovjilxaqeskdpybz
```

Python Code Used to Generate the random sequence of alphabets. I used this output to cipher the plain text.

Takeaway: From this task, I learned how to perform a substitution cipher encryption using a permutation of the alphabet. We first generate a random substitution key by shuffling the alphabet and then apply it to the plaintext after simplifying the text (converting to lowercase and removing non-letter characters while keeping spaces). By using the `tr` command, we substitute each letter in the plaintext with the corresponding letter from the shuffled alphabet to create ciphertext. To decode the message, we use frequency analysis to find common letters and patterns in the ciphertext, which helps us guess the substitution key and recover the original plaintext. This task shows how basic

encryption methods work and how analyzing letter frequencies can help break simple ciphers.

4. Task2: Encryption Using Different Cipher Modes

For this Task, i Created a .txt file and Encryted this file using the CBC, BF-CBC, CFB, ECB. all this modes use the KEYS and IV's but ECB not uses IV. To know why ECB not uses IV(Initailze Vector). I asked ChatGpt

Answer Given by ChatGpt:
"ECB (Electronic Codebook) mode does not use an IV (Initialization Vector) because it encrypts each block of plaintext independently. In ECB mode, each block of data is encrypted the same way every time with the same key, so there's no need for an IV to introduce variability. However, this also means that ECB mode can reveal patterns in the plaintext, as identical plaintext blocks will produce identical ciphertext blocks. This lack of variability makes ECB mode less secure compared to modes that use an IV to introduce randomness and obscure patterns."

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ echo -n "This is the original text" > plaintext.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in plaintext.txt -out ciphercb
c.bin \-K 00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -bf-cbc -e  -in plaintext.txt -out bfciphercbc.b
in \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cfb -e  -in plaintext.txt -out ciphercf
b.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ecb -e  -in plaintext.txt -out cipherec
b.bin \-K  00112233445566778889aabbccddeeff
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -d  -in ciphercbc.bin -out outciphe
rcbc.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708000000000000000
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat outciphercbc.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat outciphercbc.txt
This is the original text[09/02/24]seed@VM:~/.../Dinesh-Kala$
```

This Image shows all Encryptions are executed Successfully. And i did Decryption using CBC mode with same Key and IV but when decrypting a file using CBC mode, i added more 0's.
And after Decryption i got the output.

Takeaway: From this task, I learned how to use the `openssl enc` command to encrypt and decrypt files with different encryption algorithms and modes. By experimenting with various cipher types, I discovered how each one works and affects the encryption process. For example, I learned that specifying different ciphers, like AES-CBC, changes how the data is encrypted and decrypted, impacting both security and file format. Understanding these differences helps in choosing the right encryption method for specific needs.

5. Task3: Encryption Mode - ECB vs CBC

1. ECB Mode:

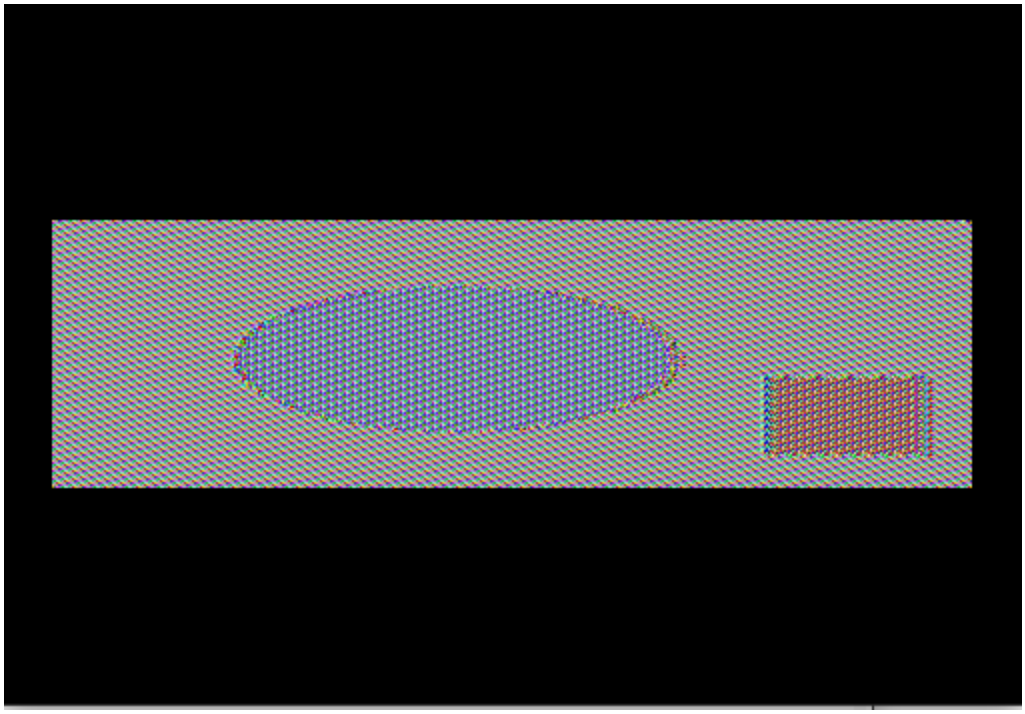
For this task, i used pic_original.bmp file which provided in the LabSetup. I used same image image for both encryption modes. Initially i enrcypted the image using the ECB mode.

Note: ECB mode do not require IV.

For bmp file the first 54 bytes are called as header information of the image. So now now ill replace the first 54 bytes of the encrypted file with the original header (54 bytes) for that i use the head and tail command and add them using the cat and save it in the new file. So, i encrypted the .bmp using the ECB mode. Taken First 54 bytes from original image to the head and from 55 bytes from the encrypted image. And merged using the “cat” command. To display the Output i used “eog”.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ecb -e  -in pic_origin
al.bmp -out encryptedecb.bmp \-K  00112233445566778889aabbccddeeff
[09/02/24]seed@VM:~/.../Dinesh-Kala$ head -c 54 pic_original.bmp > header
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tail -c +55 encryptedecb.bmp > ecb_body
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat header ecb_body > new_ecb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ eog new_ecb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$
```

Image Encryped using ECB mode and replacing Encrypted header with the Original Header

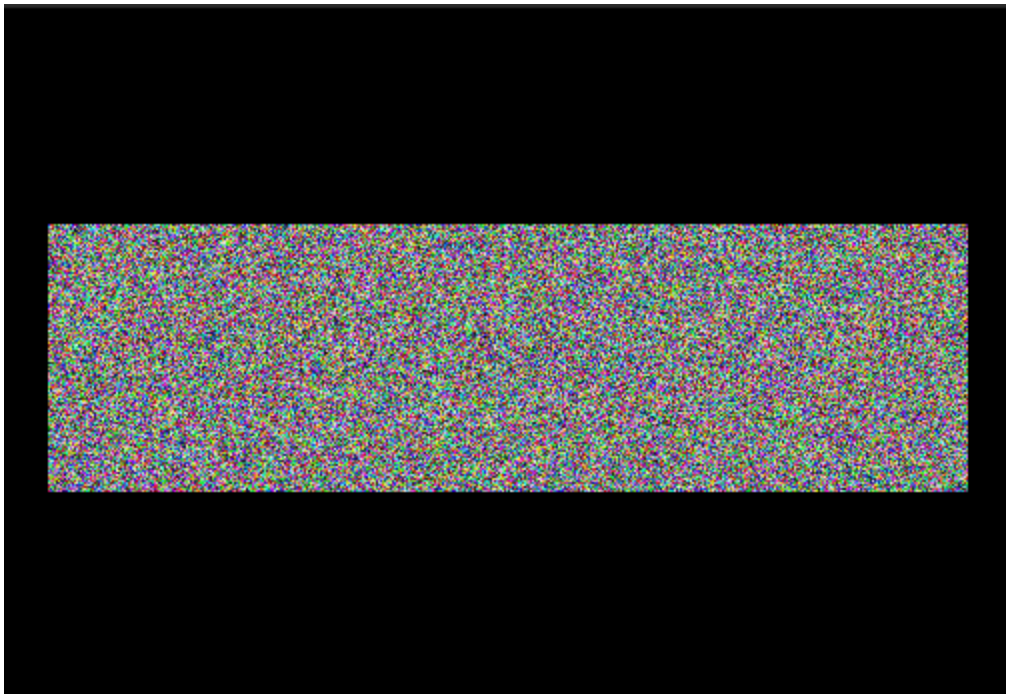


This is a Image After replacing the Header of a Encypted file

2. CBC Mode

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in pic_origin
al.bmp -out encryptedcbc.bmp \-K  00112233445566778889aabbccddeeff \-iv 01020304
05060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in pic_origin
al.bmp -out encryptedcbc.bmp \-K  00112233445566778889aabbccddeeff \-iv 01020304
0506070800000000000000000
[09/02/24]seed@VM:~/.../Dinesh-Kala$ head -c 54 pic_original.bmp > header
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tail -c +55 encryptedcbc.bmp > cbc_body
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat header cbc_body > new_cbc.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ eog new_cbc.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$
```

I Followed the Same steps that i followed for the ECB Mode but here i Encrypted the Image using the CBC Mode.



Output Image

Takeaway: From the experiment, I learned that encrypting .bmp images with ECB mode can reveal patterns in the encrypted picture because it processes each block of the image separately, making repeated patterns visible. On the other hand, CBC mode hides these patterns better because it links each block with the previous one, making the encrypted image look more random and secure. So, while ECB mode is less secure for visual data, CBC mode provides better protection by making the encrypted image harder to interpret.

6. Task4 : Padding

1.

First, I encrypted the .txt file using all 4 modes. with the same key and IV. but ECB does not require IV.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in plaintext.
txt -out f1_out_cbc.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060
708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ecb -e  -in plaintext.
txt -out f1_out_ecb.txt \-K  00112233445566778889aabbccddeeff
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cfb -e  -in plaintext.
txt -out f1_out_cfb.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060
708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ofb -e  -in plaintext.
txt -out f1_out_ofb.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060
708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$
```

This the encrypted commands for all the 4 modes (ECB, CBC, CFB, OFB).

Source: Padding (cryptography) - Wikipedia
“ECB and CBC modes require padding because they operate on fixed-size blocks of data. CFB and OFB modes do not require padding as they effectively convert the block cipher into a stream cipher, allowing them to handle data of any length without the need for padding”.

Takeaway: After going through the slides, i came up with the following. ECB encrypts data in fixed-size chunks, so if the data isn’t perfectly sized, extra padding is added to make it fit. CBC also works with fixed-size chunks, so padding is required if the data doesn’t fit exactly into these chunks. CFB encrypts data in smaller parts and can handle data of any length without needing extra padding. OFB works similarly to CFB and can process data of any size directly without adding padding.

2.
In this Task i need to create 3 .txt files with size 5, 10 and 16 bytes. I created the 5 bytes .txt files using echo and used -n (new line). And used “stat -c%s” to check the file size and encrypted using the CBC mode.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ echo -n "12345" > f1.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s f1.txt
5
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in f1.txt -ou
t f1_out.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s f1.txt
5
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s f1_out.txt
16
[09/02/24]seed@VM:~/.../Dinesh-Kala$
```

This Image shows encrypting of 5 bytes file using CBC Mode.

After Encryption the file size changed to 16. Which means padding is added. Now if i decrypted the image and check the size of the decrypted file its shows 5 bytes which is same as Original file. But we dont know how much padding is adding while encrypting.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -d  -in f1_out.txt
 -out decr_f1_out.txt \-K  00112233445566778889aabbccddeeff \-iv 010203040506070
8
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s decr_f1_out.txt
5
[09/02/24]seed@VM:~/.../Dinesh-Kala$
```

Decrypting the File

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -d -nopad -in f1_o
ut.txt -out decr_f1_out.txt \-K  00112233445566778889aabbccddeeff \-iv 010203040
5060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s decr_f1_out.txt
16
[09/02/24]seed@VM:~/.../Dinesh-Kala$ hexdump -c decr_f1_out.txt
0000000   1   2   3   4   5  \v  \v  \v  \v  \v  \v  \v  \v  \v  \v  \v
0000010
[09/02/24]seed@VM:~/.../Dinesh-Kala$ hexdump -C decr_f1_out.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
[09/02/24]seed@VM:~/.../Dinesh-Kala$ xxd decr_f1out.txt
xxd: decr_f1out.txt: No such file or directory
[09/02/24]seed@VM:~/.../Dinesh-Kala$ xxd decr_f1_out.txt
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
```

Decrypting the file without removing the Padding

So, this time i used “-nopad” command which means, It does not remove the padding while decrypting it. After the decryption the file size is 16 bytes which means padding is

not removed. To check the where the padding is used i used “xxd”, which shows the hexa decimal format and by this we can check the where padding is used.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ echo -n "1234567890" > f2.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s f2.txt
10
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in f2.txt -ou
t f2_out.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s f2_out.txt
16
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -d -nopad -in f2_o
ut.txt -out decr_f2_out.txt \-K  00112233445566778889aabbccddeeff \-iv 010203040
5060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ xxd decr_f2_out.txt
00000000: 3132 3334 3536 3738 3930 0606 0606 0606  1234567890......
[09/02/24]seed@VM:~/.../Dinesh-Kala$ hexdump -C decr_f2_out.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
[09/02/24]seed@VM:~/.../Dinesh-Kala$
```

Encrypting the 10 bytes size file

This time i created a 10 bytes size file and encrypted it with the CBC mode. After Encrypting, its size increased to 16 bytes. I used “-nopad”while decrypting to check where the padding is added.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ echo -n "1234567890abcdef" > f3.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s f3.txt
16
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in f3.txt -ou
t f3_out.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ stat -c%s f3_out.txt
32
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -d -nopad  -in f3_
out.txt -out decr_f3_out.txt \-K  00112233445566778889aabbccddeeff \-iv 01020304
05060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ hexdump -C decr_f3_out.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
[09/02/24]seed@VM:~/.../Dinesh-Kala$ xxd decr_f3_out.txt
00000000: 3132 3334 3536 3738 3930 6162 6364 6566  1234567890abcdef
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
```

Encrypting the 16 bytes File

This time i created the 16 byte file and encrypted it with the CBC, But this time after encryption its size increased to 32 bytes. And i followed the same steps to check where the padding is added.

Takeaway: I learned that when encrypting files using AES-CBC, padding is added to ensure that the file size matches the block size required for encryption (16 bytes for AES). The amount of padding depends on the original file size. To see the padding, you can use the `-nopad` option during decryption, which keeps the padding data intact. By examining the decrypted output with a hex viewer, you can view the padding bytes, which are not normally visible in plain text. This process helps understand how padding adjusts the file size to fit the encryption requirements.

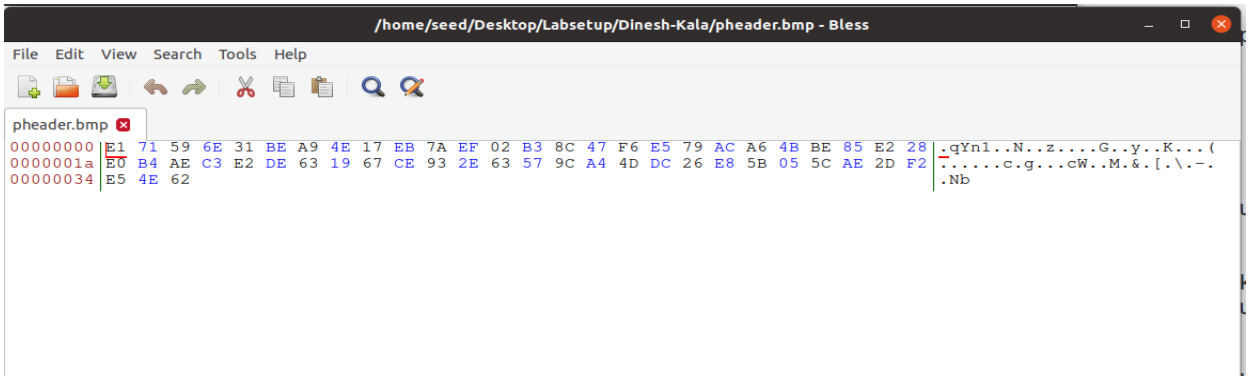
7. Task5: Error Propagation - Corrupted Cipher Text

Before the task, I think CBC and CFB will not generate image because, CBC used chain connecrion, so if one block is corrupted remaining every blocks will get corrupted and decryption will not happen. And coming to the CFB, CFB is closely similar to CBC. so i think if it get corrupted at one block, all blocks will be corrupted. ECB will decrypted easily because blockas are not connected.

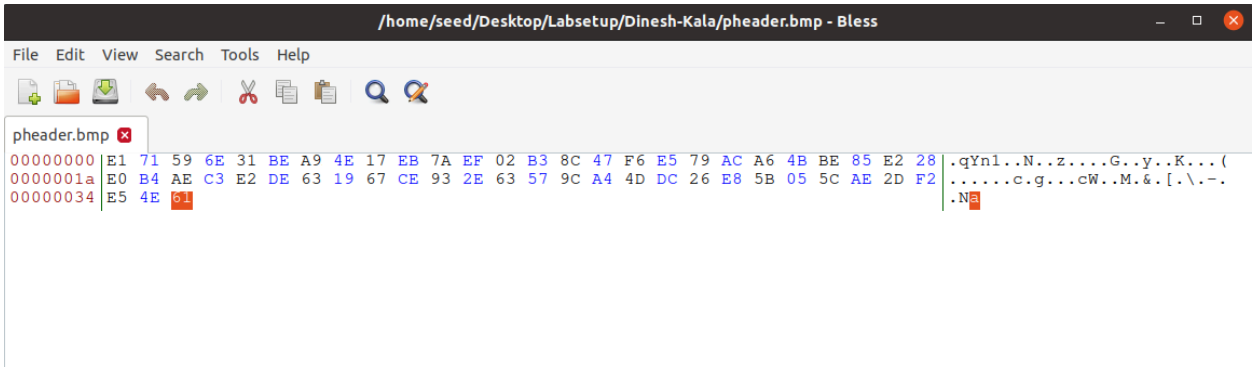
1. Using CBC mode:

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in pic_origin
al.bmp -out outputcbcimage.bmp \-K  00112233445566778889aabbccddeeff \-iv 010203
0405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ head -c 55 outputcbcimage.bmp > pheader.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tail -c +56 outputcbcimage.bmp > pbody.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat pheader.bmp pbody.bmp > finalcbc.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -d  -in finalcbc.b
mp -out imagecbc.bmp \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ eog imagecbc.bmp
```

I taken image given in the Labsetup and Encrypted it using the aes-128-CBC mode. I manually corrupted the encrypted file at 55th Byte using the Bless. Before Corrupting the Encrypted file, i divide the Bytes into Two, Header (first 55 bytes) and Tail (remaining bytes). Below the Screenshot before corrupting the header where last byte is 62.

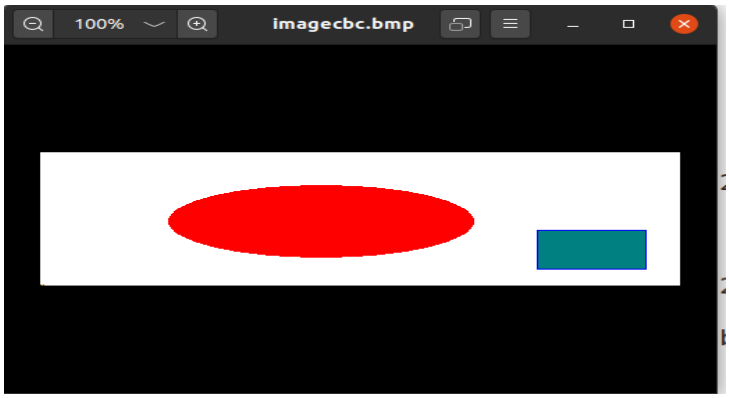


Before corrupted



After Corruption

Now i changed the last digit 2 to 1. And merged the corrupted header with the body using the cat command and in another file. Now i decrypted the corrupted file using the same mode CBC, Key and IV. and with the decrypted file. I run the eog command to display the output of decrypted file. And the got the image.



Output Image

2. ECB Mode:

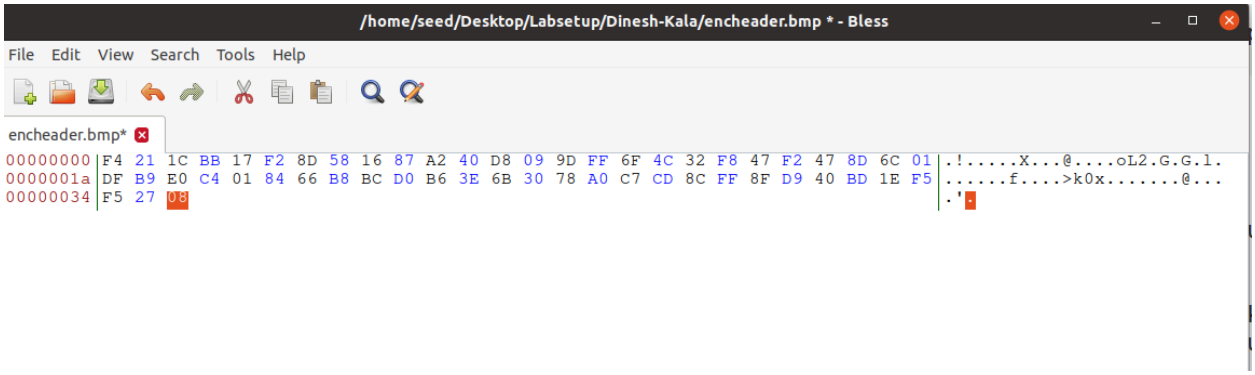
I followed the Same steps that i followed for CBC but this time i used ECB mode. I divided the encrypted file into 2 parts encrypted the 55th byte by changing the last digit and merged it with the body and then decrypted using the same Mode, Key, IV.

Note: ECB do not Require the IV.

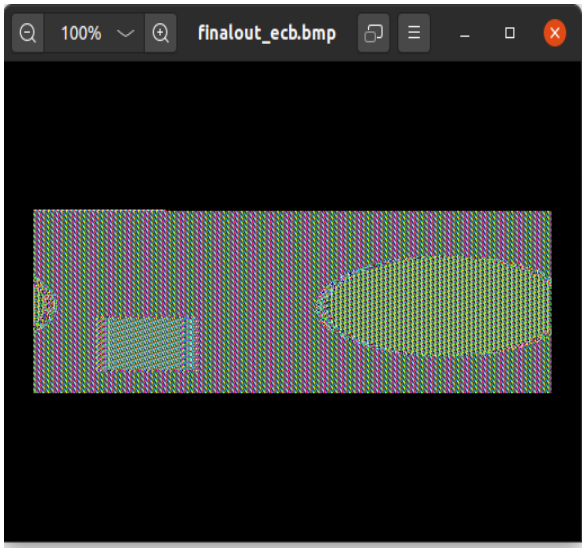
```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ecb -e  -in pic_origin
al.bmp -out enc_img_ecb.bmp \-K  00112233445566778889aabbccddeeff
[09/02/24]seed@VM:~/.../Dinesh-Kala$ head -c 55 enc_img_ecb.bmp > encheader.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tail +56 enc_img_ecb.bmp > encbody_ecb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat head encbody_ecb.bmp > finalencr_ecb.bm
p
cat: head: No such file or directory
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat encheader.bmp encbody_ecb.bmp > finalen
cr_ecb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ecb -d  -in finalencr_
ecb.bmp -out finalout_ecb.bmp \-K  00112233445566778889aabbccddeeff
bad decrypt
139736599954752:error:0606506D:digital envelope routines:EVP_DecryptFinal_ex:wro
ng final block length:crypto/evp/evp_enc.c:572:
[09/02/24]seed@VM:~/.../Dinesh-Kala$ eog finalout_ecb.bmp
```

Commands used to encrypt and decrypt the files

When i decrypted the file, it effected some blocks as we can see bad decrypt. But we can see by the output image that, some blocks are effected because of the corruption.



This above image shows the Corrupted byte in the Header. And imanually encrypted in Bless.

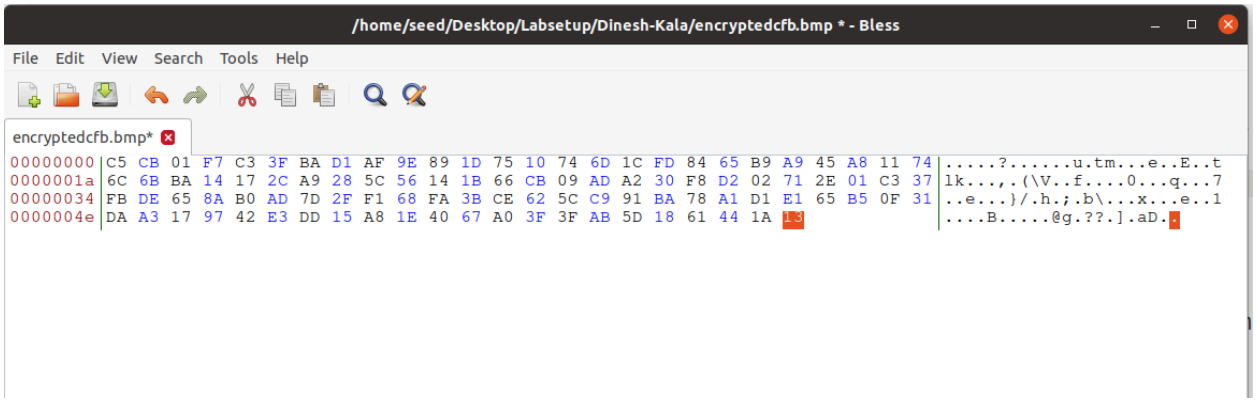


Output image of Corrupted File.

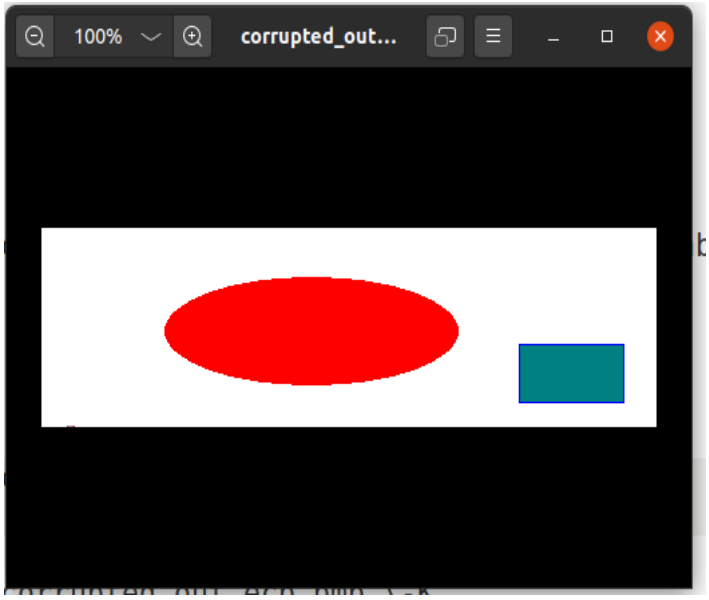
3. Using CFB mode:

I followed the same procedure for this Mode as well but this time i manually corrupted the 100th byte of header file. And merged with the body and decrypted the file and displayed the output using the eog Command.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cfb -e  -in pic_origin
al.bmp -out cfbimage.bmp \-K  00112233445566778889aabbccddeeff \-iv 010203040506
0708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ head -c 100 cfbimage.bmp > encryptedcfb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tail -c +101 cfbimage.bmp > bodycfb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat encryptedcfb.bmp bodycfb.bmp > corrupte
dcfb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ penssl enc -aes-128-cfb -d  -in corruptedcf
b.bmp -out corrupted_out_cfb.bmp \-K  00112233445566778889aabbccddeeff \-iv 0102
030405060708

Command 'penssl' not found, did you mean:

  command 'openssl' from deb openssl (1.1.1f-1ubuntu2)

Try: sudo apt install <deb name>

[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cfb -d  -in corruptedc
fb.bmp -out corrupted_out_cfb.bmp \-K  00112233445566778889aabbccddeeff \-iv 010
2030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ eog corrupted_out_cfb.bmp
```



Showing which byte is corrupted.



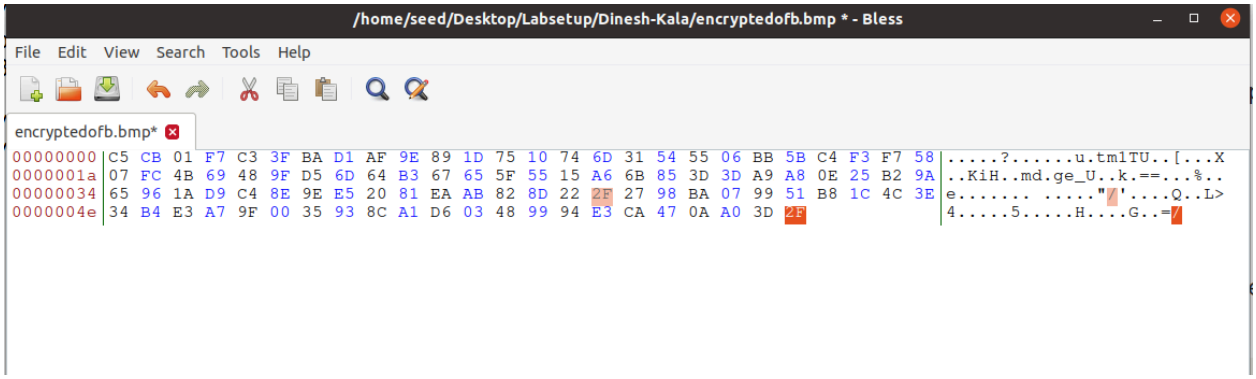
Output image of a Corrupted file

4. OFB Mode:

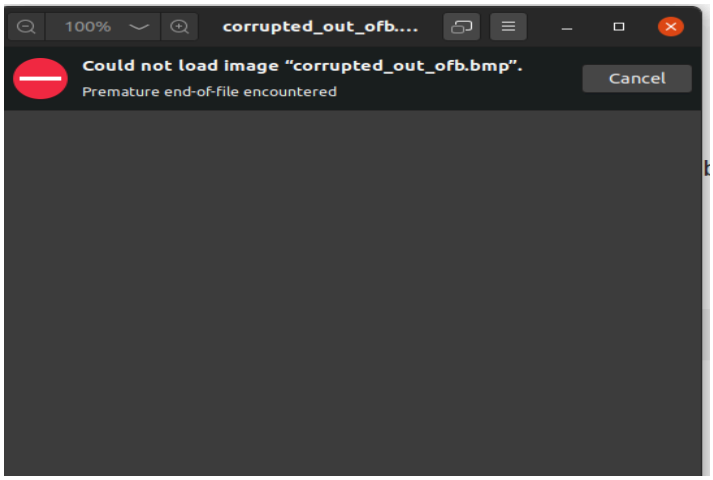
Followed the same steps as CFB and corrupted the encrypted file at 100th byte using the bless and Displayed the output using the eog Command.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ofb -e  -in pic_origin
al.bmp -out ofbimage.bmp \-K  00112233445566778889aabbccddeeff \-iv 010203040506
0708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ head -c 100 ofbimage.bmp > encryptedofb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ tail -c +101 ofbimage.bmp > bodyofb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ cat encryptedofb.bmp bodyofb.bmp > corrupte
dofb.bmp
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cfb -d  -in corruptedo
fb.bmp -out corrupted_out_ofb.bmp \-K  00112233445566778889aabbccddeeff \-iv 010
2030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ eog corrupted_out_ofb.bmp
```

Enrycption and Decryption of a file using OFB mode.



Showing which byte got corrupted(highlighted byte)



Output of a Corruptef file (OFB).

Takeaways: From this task, I learned how different encryption modes handle errors. In OFB mode, even a small error can make the entire data unreadable, as I couldn’t recover the image at all. In CFB and CBC modes, I could still see the images, meaning these modes are better at handling errors, though some data still got corrupted. In ECB mode, the image was blurry because errors in one block didn’t affect others, but the pattern became hard to recognize. This shows why choosing the right encryption mode is important, as it affects how much data can be recovered if something goes wrong.

8. Intail Vector (IV) and Common Mistakes

8.1 Task6.1 : IV Experiment

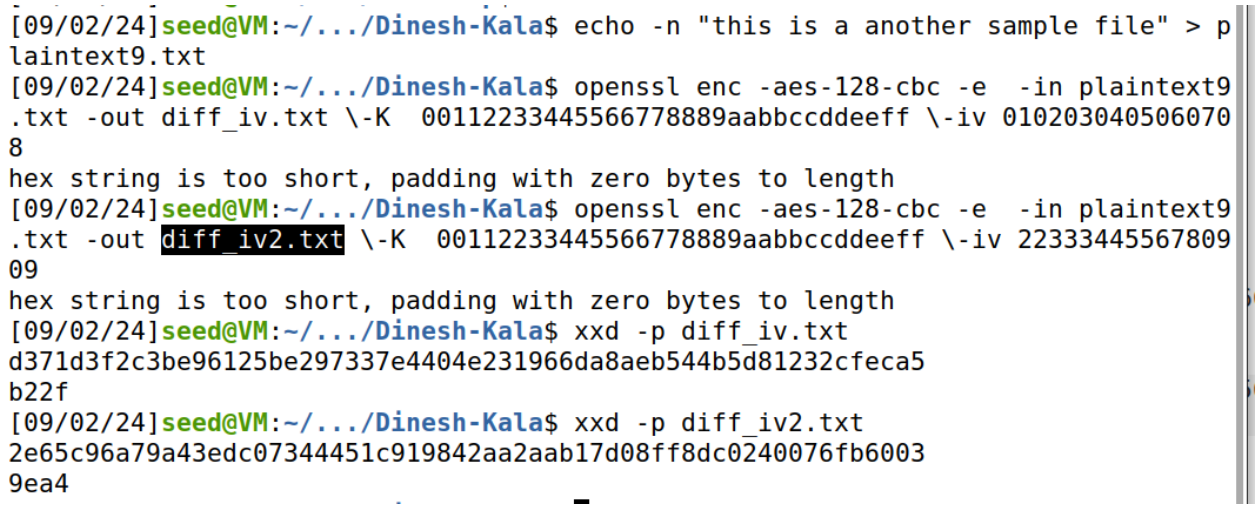
1. Same IV:

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ echo -n "this is same iv text" > iv_test.tx
t
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in iv_test.tx
t -out same_iv.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-cbc -e  -in iv_test.tx
t -out same_iv2.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ xxd -p same_iv.txt
d8aa22e0627b82cf2ede8b5df131cfa5dacdd014f394ab85ebb6d3cafe42
c642
[09/02/24]seed@VM:~/.../Dinesh-Kala$ xxd -p same_iv2.txt
d8aa22e0627b82cf2ede8b5df131cfa5dacdd014f394ab85ebb6d3cafe42
c642
```

In the above screenshot, i created a .txt file and encrypted it with the aes-128 using cbc mode. Then i encrypted the same input file using the same IV. but if we compare both

hexadecimals of ciphertexts are same. Which means, we need to use the different IV's even if it is a same .txt file.

2. Different IV's



If we see the above screenshot, i encrypted the same .txt file twice but this time i used different IV's. if we compare the both cipher texts they look different this time.

Takeaway: From this task, I observed that when you use two different IVs to encrypt the same plaintext, you get different ciphertexts, which makes it harder for attackers to figure out the original message. However, if you use the same IV, the ciphertexts are identical, revealing a pattern that could help attackers decode the message. This shows why it's crucial to always use a unique IV to keep the encryption strong and protect the data.

8.2 Task6.2: Common Mistake: Use Same IV

I created a .txt file with the message using "echo" command and then i encrypted it with the aes-128 and with mode ofb with the key and IV. using xxd -p. I got the hexa decimal for my ciphertext file. I repeated the same steps and and created 2nd .txt file and encrypted using the same encryption mode and got the hexadecimal for my 2nd ciphertext. Now i used thos two cipher codes in the .py file and generated the output. Note: i used the same IV to encrypt the both .txt files

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ echo "this is a first message" >k1.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ofb -e  -in k1.txt -ou
t cipher_k1.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ xxd -p cipher_k1.txt
f3eee656e156c9f1cebed9740763284d5c31ea74da3c27f9
[09/02/24]seed@VM:~/.../Dinesh-Kala$ echo "this is a second message" >k2.txt
[09/02/24]seed@VM:~/.../Dinesh-Kala$ openssl enc -aes-128-ofb -e  -in k2.txt -ou
t cipher_k2.txt \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
hex string is too short, padding with zero bytes to length
[09/02/24]seed@VM:~/.../Dinesh-Kala$ xxd -p cipher_k2.txt
f3eee656e156c9f1cebecc78167f32091139fc74c83a2596fd
[09/02/24]seed@VM:~/.../Dinesh-Kala$ sample_code.py
87868f25c13fbad1af9ebf1d75105c6d31549907bb5b42
74686973206973206120736565f6e64206d6573736167
00000000000000000000000000000000000000000000000000
[09/02/24]seed@VM:~/.../Dinesh-Kala$
```

Above are the commands that i used to encrypt the .txt files.

```python
#!/usr/bin/python3

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

MSG   = "this is a first message"
HEX_1 = "f3eee656e156c9f1cebed9740763284d5c31ea74da3c27f9"
HEX_2 = "f3eee656e156c9f1cebecc78167f32091139fc74c83a2596fd"

# Convert ascii string to bytearray
D1 = bytes(MSG, 'utf-8')

# Convert hex string to bytearray
D2 = bytearray.fromhex(HEX_1)
D3 = bytearray.fromhex(HEX_2)

r1 = xor(D1, D2)
r2 = xor(r1, D3)
r3 = xor(D2, D2)
print(r1.hex())
print(r2.hex())
print(r3.hex())
```

I run the .py code first (default code). Then i made changes to it to get a desired result. I did some changes in the code, I replaced D2 with the r1 to get the desired output

2nd Plaintext in Bless

Now i opened 2nd txt file in Bless to check the Bytes and i compared with the output that generated by the code. And both are similar to each other which means i get to know that if we use same IV to encrypt the message. Its easy for hackers to hack. And this is attack is known as known plaintext attack.

Changing OFB to CFB: if we change OFB to CFB only certain text can be leaked, but not all the data will be revealed.

Takeaways: From this experiment, I learned that using the same IV with known plaintext and ciphertext can make encryption vulnerable, especially in OFB mode. If an attacker knows one plaintext and its matching ciphertext, they might be able to uncover other encrypted messages. This shows why it's important to avoid reusing the same IV to keep encryption secure.

8.3 Task6.3: Common Mistake: Use a Predictable IV

I guess Bob's plaintext is "Yes". To verify that, I have written a code in .py using the hints given by the professor and ChatGPT. At first, i wrote code then i gave Bob used IV and Next_IV in the code. I guessed Bob plain text should be Yes. so i chosen my plain text as a Yes.

```python
#!/usr/bin/python3
def xor(a, b):
    return bytearray(x ^ y for x, y in zip(a, b))

# Given values
IV_BOB = "5a0b03ed789e171273a5426ce689c8eb"  # The IV used for Bob's ciphertext
IV_NEXT = "e40c9c1f799e171273a5426ce689c8eb"  # The next IV provided by the oracle

# Constructing guesses
YES = b"Yes" + bytes("\x0d"*13, 'utf-8')  # Padding for "Yes"
NO  = b"No"  + bytes("\x0e"*14, 'utf-8')  # Padding for "No"

# Guessing "Yes"
temp  = xor(YES, bytearray.fromhex(IV_BOB))  # XOR guess with Bob's IV
guess = xor(temp, bytearray.fromhex(IV_NEXT))  # XOR with the next IV
print(guess.hex())  # Output the result
```

6.3.1Python code

After running that code in command prompt, it generated the hexa decimal for my text and i used that hexa decimal as my input and gave it to oracle. Below is the Cipher text i got for my plain text.
If we compare my cipher text with the bob text we can clearly see that first 16 bytes are same which means the plain text is “Yes”.

```
[09/02/24]seed@VM:~/.../Dinesh-Kala$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 4536d96f09c8b0c7f70ae6495ddf81fd
The IV used    : 5a0b03ed789e171273a5426ce689c8eb

Next IV        : e40c9c1f799e171273a5426ce689c8eb
Your plaintext : e762ecff0c0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: 4536d96f09c8b0c7f70ae6495ddf81fdeb6b7874bbd42c24a1a672e7e2f8f022

Next IV        : ed533679799e171273a5426ce689c8eb
Your plaintext :
```

To do this task, i used Docker. I installed the Docker during the Lab Environment and used
nc 10.9.0.80 3000 to connect Oracle. Initially, it didn’t work, i used the Docker commands to down and up the server. Dcup (to up the server)
Dcdown (to down the server).

Takeaways: From this task, I learned that IVs (Initialization Vectors) must be unpredictable to keep encryption secure. If IVs are predictable, even strong encryption like AES can be at risk, as attackers might figure out the message by crafting specific inputs. This shows why it's crucial to use random IVs to protect the encrypted information.