

MINI PROJECT 1: SECRET-KEY ENCRYPTION

Objective: The objective of this project is to get familiar with the concepts of the secret-key (Symmetric-Key) encryption such as encryption algorithms, encryption modes, paddings, and initial vector.

Task 1: Frequency Analysis:

As we know, Frequency Analysis helps in cracking monoalphabetic substitution ciphers. It counts the appearance of each character in the ciphertext to determine the frequency of each character.

To apply this, we ran a python program (freq.py) over a given text file (ciphertext.txt) and we got below results:

```
[02/18/23]seed@VM:~/.../Files$ ./freq.py
-----
1-gram (top 20): 2-gram (top 20): 3-gram (top 20):
n: 488          yt: 115          ytn: 78
y: 373          tn: 89          vup: 30
v: 348          mu: 74          mur: 20
x: 291          nh: 58          ynh: 18
u: 280          vh: 57          xzy: 16
q: 276          hn: 57          mxu: 14
m: 264          vu: 56          gnq: 14
h: 235          nq: 53          ytv: 13
t: 183          xu: 52          nqy: 13
i: 166          up: 46          vii: 13
p: 156          xh: 45          bxh: 13
a: 116          yn: 44          lvq: 12
c: 104          np: 44          nuy: 12
z: 95           vy: 44          vyn: 12
l: 90           nu: 42          uvy: 11
g: 83           qy: 39          lmu: 11
b: 83           vq: 33          nvh: 11
r: 82           vi: 32          cmu: 11
e: 76           gn: 32          tmq: 10
d: 59           av: 31          vhp: 10
```

After analysing above counts and the ciphertext.txt file, we concluded some common substitutions of the ciphers directly after seeing the top 20 trigrams and bigrams, such as 'ytn' is 'THE', 'v' is 'A' and so, 'vu' is 'AN', 'vy' is 'AT' and so on.

Hence, after few hits and trial, we were able to conclude the encryption key and converted it into a plaintext, as shown below.

Here is the command used to convert ciphertext into plaintext using encryption key:

```
[02/18/23]seed@VM:~/.../Files$ tr 'ytnvqbnhmrugipsxkjjzedaclofw' \
> 'THASFERIGNBLDKOXQUPYCMWJVZ' < ciphertext.txt > plaintext.txt
[02/18/23]seed@VM:~/.../Files$
```

The Required Plaintext:

```
plaintext.txt
Share ~/Share/Labsetup/Files

1 THE OSCARS TURN ON SUNDAY WHICH SEEMS ABOUT RIGHT AFTER THIS LONG STRANGE
2 AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO
3
4 THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET
5 AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY
6 THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND
7 A NATIONAL CONVERSATION AS BRIEF AND MAD AS A FEVER DREAM ABOUT WHETHER THERE
8 OUGHT TO BE A PRESIDENT WINFREY THE SEASON DIDNT JUST SEEM EXTRA LONG IT WAS
9 EXTRA LONG BECAUSE THE OSCARS WERE MOVED TO THE FIRST WEEKEND IN MARCH TO
10 AVOID CONFLICTING WITH THE CLOSING CEREMONY OF THE WINTER OLYMPICS THANKS
11 PYEONGCHANG
12
13 ONE BIG QUESTION SURROUNDING THIS YEARS ACADEMY AWARDS IS HOW OR IF THE
14 CEREMONY WILL ADDRESS METOO ESPECIALLY AFTER THE GOLDEN GLOBES WHICH BECAME
15 A JUBILANT COMINGOUT PARTY FOR TIMES UP THE MOVEMENT SPEARHEADED BY |
16 POWERFUL HOLLYWOOD WOMEN WHO HELPED RAISE MILLIONS OF DOLLARS TO FIGHT SEXUAL
17 HARASSMENT AROUND THE COUNTRY
18
19 SIGNALING THEIR SUPPORT GOLDEN GLOBES ATTENDEES SWATHED THEMSELVES IN BLACK
20 SPORTED LAPEL PINS AND SOUNDED OFF ABOUT SEXIST POWER IMBALANCES FROM THE RED
21 CARPET AND THE STAGE ON THE AIR E WAS CALLED OUT ABOUT PAY INEQUITY AFTER
22 ITS FORMER ANCHOR CATT SADLER QUIT ONCE SHE LEARNED THAT SHE WAS MAKING FAR
23 LESS THAN A MALE COHOST AND DURING THE CEREMONY NATALIE PORTMAN TOOK A BLUNT
24 AND SATISFYING DIG AT THE ALLMALE ROSTER OF NOMINATED DIRECTORS HOW COULD
25 THAT BE TOPPED
26
27 AS IT TURNS OUT AT LEAST IN TERMS OF THE OSCARS IT PROBABLY WONT BE
28
29 WOMEN INVOLVED IN TIMES UP SAID THAT ALTHOUGH THE GLOBES SIGNIFIED THE
30 INITIATIVES LAUNCH THEY NEVER INTENDED IT TO BE JUST AN AWARDS SEASON
31 CAMPAIGN OR ONE THAT BECAME ASSOCIATED ONLY WITH REDCARPET ACTIONS INSTEAD
```

Task 2: Encryption using different Ciphers and Modes:

In this task, we have used five different ciphers with different modes to encrypt and decrypt a text file. Ciphers used are **AES-128**, **Aria-128**, **BF** (Blowfish) and **DES**.

First, we created a dummy text file (task2_file.txt).

```
task2_file.txt
Share ~/Share/Labsetup/Files

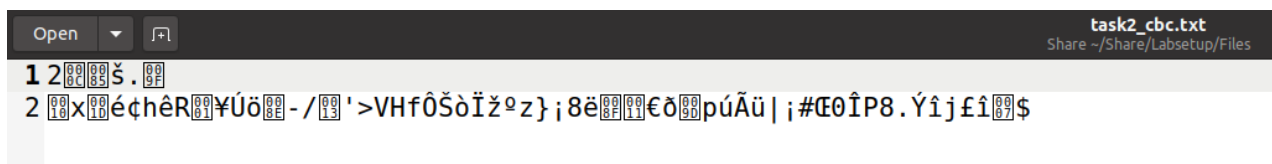
1 Hey! What do you know about Kerckhoff's Principle?
```

After that, we used AES-128 cipher with CBC mode first, using OPENSSL ENC command line tool, to encrypt and decrypt the above text file.

AES-128-CBC

```
[02/18/23]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e \
> -in task2_file.txt \
> -out task2_cbc.txt \
> -K 00112233445566778899aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
[02/18/23]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -d \
> -in task2_cbc.txt \
> -out task2_cbc_plaintext.txt \
> -K 00112233445566778899aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
[02/18/23]seed@VM:~/.../Files$
```

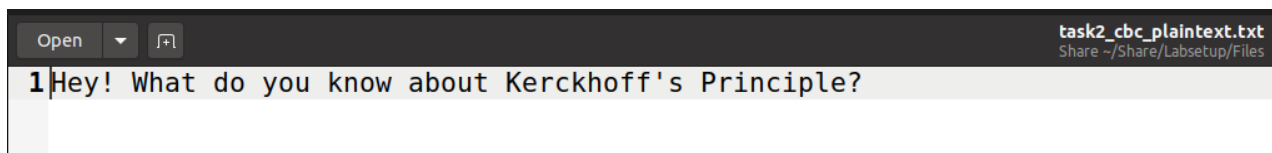
The encrypted file looked like below:



task2_cbc.txt
Share ~/Share/Labsetup/Files

```
1 2000š.00
2 00x00éçêR00YÜö00-/00'>VHfÔŠòİž°z};8ë0000€ö00púÄü|;#00ÎP8.Yîjfi00$
```

And the decrypted file gave our original text back:



task2_cbc_plaintext.txt
Share ~/Share/Labsetup/Files

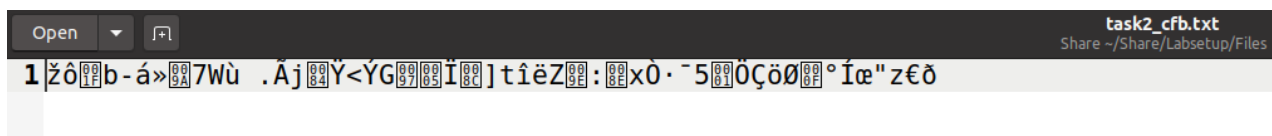
```
1 Hey! What do you know about Kerckhoff's Principle?
2
```

We then changed the cipher mode from CBC to CFB:

AES-128-CFB

```
[02/18/23]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e \
> -in task2_file.txt \
> -out task2_cfb.txt \
> -K 00112233444566778899aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
[02/18/23]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -d \
> -in task2_cfb.txt \
> -out task2_cfb_plaintext.txt \
> -K 00112233444566778899aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
[02/18/23]seed@VM:~/.../Files$
```

Encrypted File:



task2_cfb.txt
Share ~/Share/Labsetup/Files

```
1 žô00b-á»007Wù .Aj00Y<YG0000İ00]tîëZ00:00x0·~5000Çö00°Íæ"z€ð
2
```

Decrypted File:

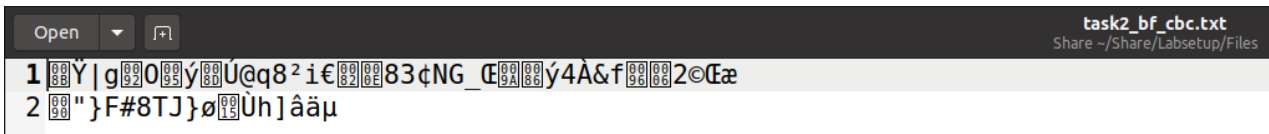


After AES, we tried BF (Blowfish) with CBC mode first:

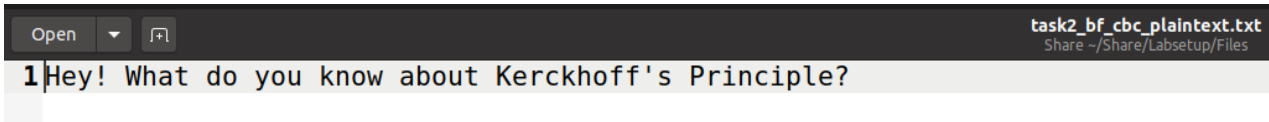
BF-CBC

```
[02/18/23]seed@VM:~/.../Files$ openssl enc -bf-cbc -e \
> -in task2_file.txt \
> -out task2_bf_cbc.txt \
> -K 00112233444556677889aabbccddeeff \
> -iv 0102030405060708
[02/18/23]seed@VM:~/.../Files$ openssl enc -bf-cbc -d \
> -in task2_bf_cbc.txt \
> -out task2_bf_cbc_plaintext.txt \
> -K 00112233444556677889aabbccddeeff \
> -iv 0102030405060708
[02/18/23]seed@VM:~/.../Files$
```

Encrypted file:



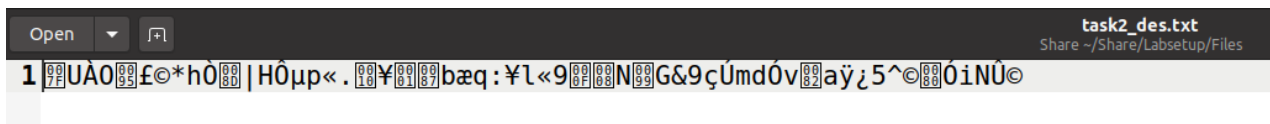
Decrypted file:



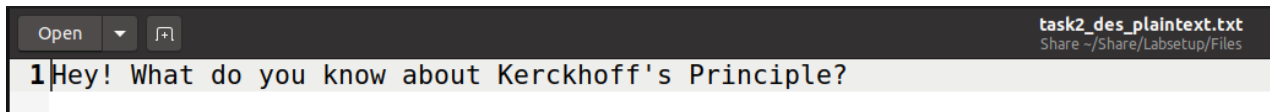
DES-CBC

```
[02/18/23]seed@VM:~/.../Files$ openssl enc -des-cbc -e \
> -in task2_file.txt \
> -out task2_des.txt \
> -K 00112233445566777899aabbccddeeff \
> -iv 0102030405060708
hex string is too long, ignoring excess
[02/18/23]seed@VM:~/.../Files$ openssl enc -des-cbc -d \
> -in task2_des.txt \
> -out task2_des_plaintext.txt \
> -K 00112233445566777899aabbccddeeff \
> -iv 0102030405060708
hex string is too long, ignoring excess
[02/18/23]seed@VM:~/.../Files$
```

Encrypted file:



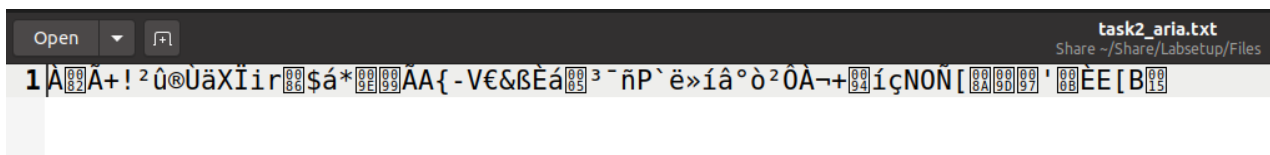
Decrypted file:



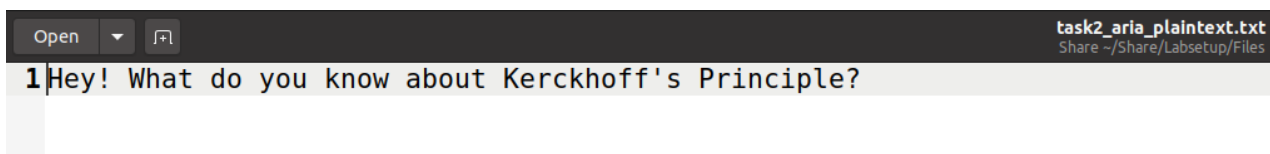
ARIA-128-CBC

```
[02/18/23]seed@VM:~/.../Files$ openssl enc -aria-128-cbc -e \  
> -in task2_file.txt \  
> -out task2_aria.txt \  
> -K 00111223344556677889aabbccddeeff \  
> -iv 0102030405060708  
hex string is too short, padding with zero bytes to length  
[02/18/23]seed@VM:~/.../Files$ openssl enc -aria-128-cbc -d \  
> -in task2_aria.txt \  
> -out task2_aria_plaintext.txt \  
> -K 00111223344556677889aabbccddeeff \  
> -iv 0102030405060708  
hex string is too short, padding with zero bytes to length  
[02/18/23]seed@VM:~/.../Files$
```

Encrypted file:



Decrypted file:



Task 3: Encryption Mode – ECB vs. CBC:

ECB (Electronic Code Book) and CBC (Cipher Block Chaining) are the two cipher modes with one major difference. In the CBC mode, we pass an “iv” (an initialization vector which is random and unique) before encryption just to prevent having the guessable plaintext result in the ciphertext whereas in the ECB mode, we do not pass any “iv” and therefore, one can see the guessable plaintext result in the ciphertext.

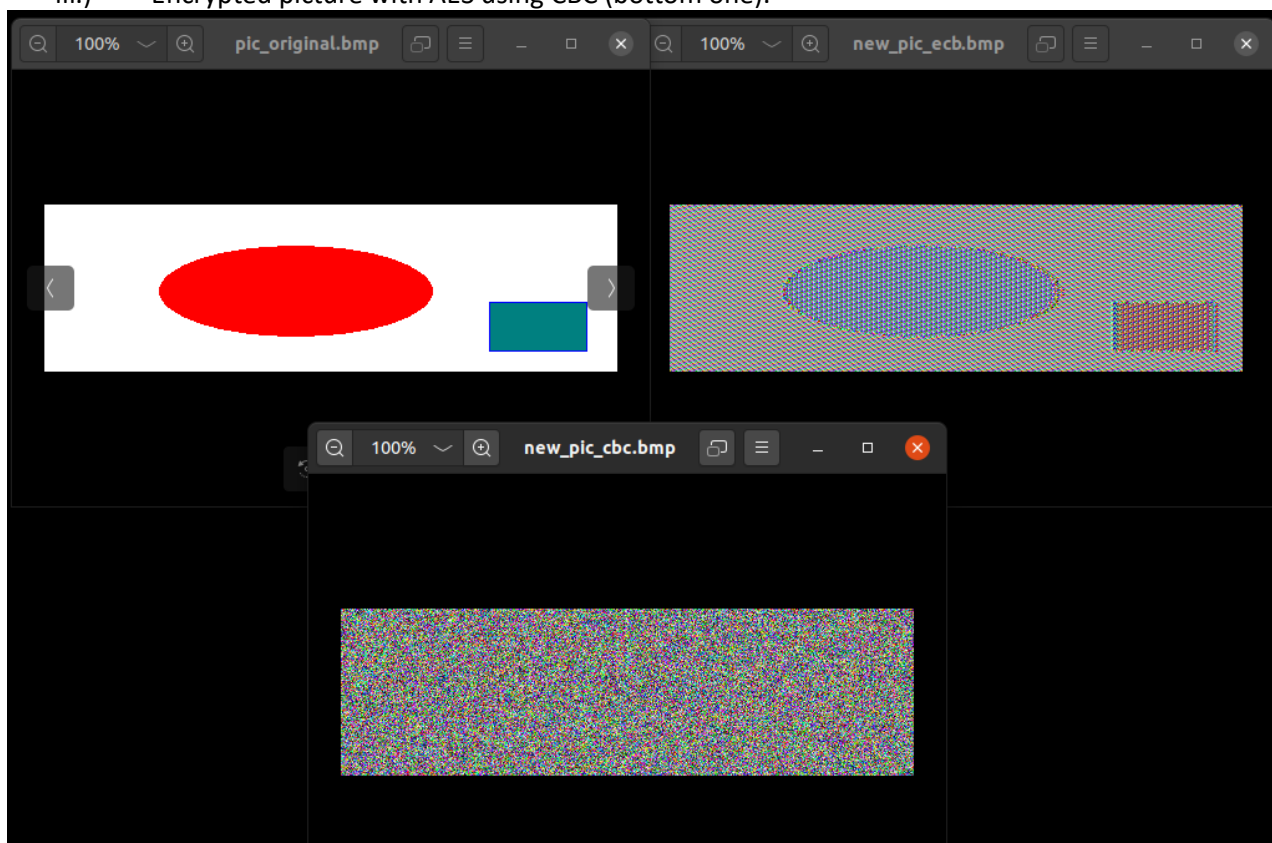
To show the difference, we encrypted a given picture using AES-128 cipher with CBC and ECB mode, as you can see in the below snapshot:


```
[02/18/23]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e \
> -in pic_original.bmp \
> -out pic_cipher_cbc.bmp \
> -K 00112233445566778889aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
[02/18/23]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e \
> -in pic_original.bmp \
> -out pic_cipher_ecb.bmp \
> -K 00112233445566778889aabbccddeeff
[02/18/23]seed@VM:~/.../Files$ head -c 54 pic_original.bmp > header
[02/18/23]seed@VM:~/.../Files$ tail -c +55 pic_cipher_cbc.bmp > body_cbc
[02/18/23]seed@VM:~/.../Files$ tail -c +55 pic_cipher_ecb.bmp > body_ecb
[02/18/23]seed@VM:~/.../Files$ cat header body_cbc > new_pic_cbc.bmp
[02/18/23]seed@VM:~/.../Files$ cat header body_ecb > new_pic_ecb.bmp
[02/18/23]seed@VM:~/.../Files$ █
```

And since it was a '.bmp' picture, we had to concatenate original picture header with encrypted picture body.

Below is the combined snapshot:

- i.) Original picture (on top left),
- ii.) Encrypted picture with AES using ECB (on top right),
- iii.) Encrypted picture with AES using CBC (bottom one).



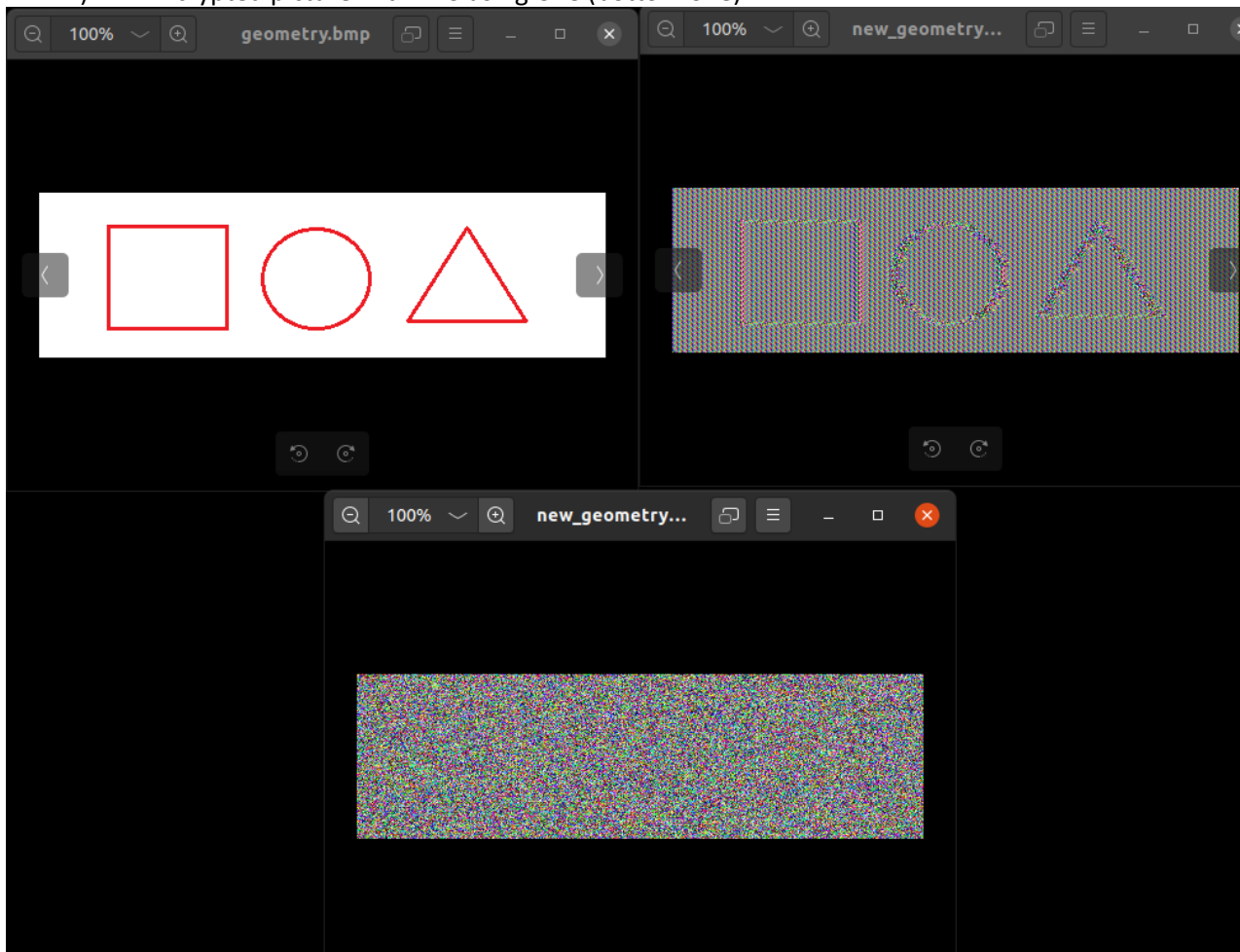
I again tried the same cipher with ECB and CBC mode on a custom image and got the same result, as you can see the below snapshot:

Commands to encrypt the image with AES cipher using CBC and ECB mode:

```
[02/18/23]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e \
> -in geometry.bmp \
> -out geometry_cbc.bmp \
> -K 00112233445566778899aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
[02/18/23]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e \
> -in geometry.bmp \
> -out geometry_ecb.bmp \
> -K 00112233445566778899aabbccddeeff
[02/18/23]seed@VM:~/.../Files$ head -c 54 geometry.bmp > geometry_header
[02/18/23]seed@VM:~/.../Files$ tail -c +55 geometry_cbc.bmp > geometry_cbc_body
[02/18/23]seed@VM:~/.../Files$ tail -c +55 geometry_ecb.bmp > geometry_ecb_body
[02/18/23]seed@VM:~/.../Files$ cat geometry_header geometry_cbc_body > new_geometry_cbc.bmp
[02/18/23]seed@VM:~/.../Files$ cat geometry_header geometry_ecb_body > new_geometry_ecb.bmp
[02/18/23]seed@VM:~/.../Files$
```

Below is the combined snapshot:

- i.) Original picture (on top left),
- ii.) Encrypted picture with AES using ECB (on top right),
- iii.) Encrypted picture with AES using CBC (bottom one).

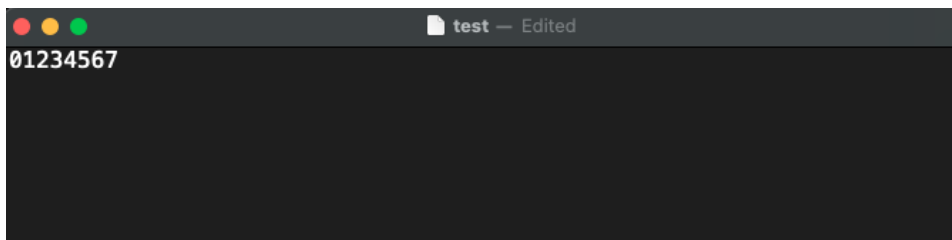


Our Observations:

- i.) ECB mode leaks data about the underlying message being encrypted. The reason this happens is because it produces identical ciphertext blocks after encrypting identical plaintext blocks.
- ii.) CBC mode eliminates the ECB issue. The plaintext of a block is combined with the ciphertext of the previous block via XOR operation and the result is encrypted.

Task 4: Padding:

We encrypted an 8 byte “test” file using these modes ECB, CBC, CFB, OFB:



```
dev@Devs-Air desktop % openssl enc -aes-128-ecb -e -in /Users/dev/Desktop/test -K "0011223344556677889aabbccddeeff" -out ~/desktop/test_encrypt_ecb
dev@Devs-Air desktop % openssl enc -aes-128-cbc -e -in /Users/dev/Desktop/test -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/test_encrypt_cbc
dev@Devs-Air desktop % openssl enc -bf-cfb -e -in /Users/dev/Desktop/test -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/test_encrypt_cfb
dev@Devs-Air desktop % openssl enc -bf-ofb -e -in /Users/dev/Desktop/test -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/test_encrypt_ofb
dev@Devs-Air desktop %
```

ECB and CBC mode add padding to the plaintext because the block size on which encryption will be done should be 16 bytes (128 bits). If it is shy of this size, then padding is done. CFB and OFB do not need padding because when you encrypt using Cipher-Feedback (CFB) or Output-Feedback (OFB), then the ciphertext is of the same size as the plaintext and this is Why padding is not required in these modes.

- i.) Creating 3 files of length 5, 10 and 16 bytes, respectively.

```
dev@Devs-Air desktop % echo -n "01234" > f1.txt
dev@Devs-Air desktop % echo -n "0123456789A" > f2.txt
dev@Devs-Air desktop % echo -n "0123456789ABCDEF" > f3.txt
dev@Devs-Air desktop %
```

- ii.) Encrypting f1, f2 and f3 using CBC.

```
dev@Devs-Air desktop % openssl enc -aes-128-cbc -e -in /Users/dev/Desktop/f1.txt -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/f1encrypted.txt
dev@Devs-Air desktop % openssl enc -aes-128-cbc -e -in /Users/dev/Desktop/f2.txt -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/f2encrypted.txt
dev@Devs-Air desktop % openssl enc -aes-128-cbc -e -in /Users/dev/Desktop/f3.txt -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/f3encrypted.txt
dev@Devs-Air desktop %
```

- iii.) The size of the encrypted files is 16, 16, and 32 bytes for f1, f2 and f3 respectively, which in bits is 128 bits, 128 bits and 2 blocks of 128 bits for f3. The size of a block in CBC mode is 128 bits.
- iv.) Decrypting the above encrypting files with “-nopad” command.

```
dev@Devs-Air desktop % openssl enc -aes-128-cbc -d -in /Users/dev/Desktop/f1encrypted.txt -nopad -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/f1decrypted.txt
dev@Devs-Air desktop % openssl enc -aes-128-cbc -d -in /Users/dev/Desktop/f2encrypted.txt -nopad -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/f2decrypted.txt
dev@Devs-Air desktop % openssl enc -aes-128-cbc -d -in /Users/dev/Desktop/f3encrypted.txt -nopad -iv "0102030405060708" -K "0011223344556677889aabbccddeeff" -out ~/desktop/f3decrypted.txt
```


- v.) The paddings which were added during the encryption process in all three files are shown below:

```
Desktop — -zsh — 83x14
dev@Devs-Air desktop % hexdump -C f1decrypted.txt
00000000  30 31 32 33 34 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b  |01234.....|
00000010
dev@Devs-Air desktop % hexdump -C f2decrypted.txt
00000000  30 31 32 33 34 35 36 37  38 39 41 05 05 05 05  |0123456789A....|
00000010
dev@Devs-Air desktop % hexdump -C f3decrypted.txt
00000000  30 31 32 33 34 35 36 37  38 39 41 42 43 44 45 46  |0123456789ABCDEF|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10  |.....|
00000020
dev@Devs-Air desktop %
```

Task 5: Error Propagation – Corrupted Cipher Text:

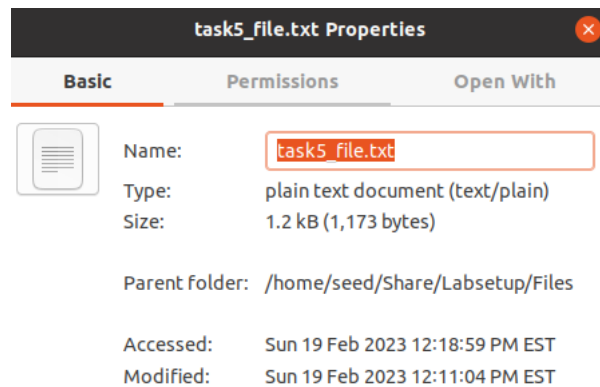
How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively?

It depends on several factors such as the encryption mode used, the percentage of corruption in the file, and many more.

- If a file has been encrypted using ECB mode, each block of data in the file is encrypted independently of the other blocks. This means that if some blocks of the file are corrupted, it may still be possible to recover the remaining uncorrupted blocks.
- If a file has been encrypted using CBC mode, each block of data is XORed with the previous block before encryption, which introduces some randomness into the encryption process. This means that if some blocks of the file are corrupted, it may still be possible to recover the remaining uncorrupted blocks, but only if the previous blocks are also uncorrupted. If a corrupted block is encountered, it will typically cause the decryption process to fail for the remaining blocks.
- If a file has been encrypted using CFB mode, each block of data is encrypted using the previous block as input to a feedback function, which generates a pseudo-random stream of data. This stream is then XORed with the plaintext to produce the ciphertext. If some blocks of the file are corrupted, it may still be possible to recover the remaining uncorrupted blocks, but only if the feedback function can be reconstructed based on the available information. If a corrupted block is encountered, it will typically cause the decryption process to fail for the remaining blocks.
- If a file has been encrypted using OFB mode, each block of data is encrypted using a pseudo-random stream generated from an IV (initialization vector) and a secret key. This stream is then XORed with the plaintext to produce the ciphertext. If some blocks of the file are corrupted, it may still be possible to recover the remaining uncorrupted blocks, but only if the IV and secret key are available, and the feedback function can be reconstructed based on the available information. If a corrupted block is encountered, it will typically cause the decryption process to fail for the remaining blocks.

To accomplish Task-5, I created a dummy text file (task5_file.txt) of 1173 bytes and encrypted it with AES-128 cipher with CBC, CFB, ECB and OFB modes, as you can see in the below snapshots:

Dummy text file properties:



Commands to encrypt the above text file with AES cipher using different modes:

```
[02/19/23]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e \  
> -in task5_file.txt \  
> -out task5_cbc.txt \  
> -K 00111233445566778899aabbccddeeff \  
> -iv 0102030405060708  
hex string is too short, padding with zero bytes to length  
[02/19/23]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e \  
> -in task5_file.txt \  
> -out task5_cfb.txt \  
> -K 00112233445566778899aabbccddeeff \  
> -iv 0102030405060708  
hex string is too short, padding with zero bytes to length  
[02/19/23]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e \  
> -in task5_file.txt \  
> -out task5_ecb.txt \  
> -K 00112233444566778899aabbccddeeff \  
>  
[02/19/23]seed@VM:~/.../Files$ openssl enc -aes-128-ofb -e \  
> -in task5_file.txt \  
> -out task5_ofb.txt \  
> -K 001122334455566778899aabbccddeeff \  
> -iv 0102030405060708  
hex string is too short, padding with zero bytes to length  
hex string is too long, ignoring excess  
[02/19/23]seed@VM:~/.../Files$ █
```

Now, I have corrupted the 55th byte of all the ciphertext files using the hex editor. After that, I decrypted all the different corrupted ciphertexts one by one, shown in the below snapshots:

ECB:

```
[02/19/23]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -d \  
> -in task5_ecb_corrupt.txt \  
> -out task5_ecb_corrupt_dec.txt \  
> -K 00112233444566778899aabbccddeeff \  
>  
bad decrypt  
139764271232320:error:0606506D:digital envelope routines:EVP_DecryptFinal_ex:wrong final block length:crypto/evp/evp_enc.c:572:  
[02/19/23]seed@VM:~/.../Files$
```

Result:

task5_ecb_corrupt_dec.txt
Share ~/Share/Labsetup/Files

```

1 The BMW M3 has a madness inside "$\F8\F0\F8\C2\E1!\cD\B0"C\F0\Hm\8C\9B\h\D4\
  \E8\98\FC\DEj\9A\B6\D9\CB\D6^E7X\89\F5\E7S\DA\F6
2 s\C5Z}\A9\D0"}EB\95/\D9[\F9\9E\B4\FC<\FD\F1\A2}auQ]\98\F4\F7d\EF\86U8\AA]\A7\A
3 Ei\F4'J3\j$F9D\A1CæHn<FF\EA\GK\F7\BD\E8(\F7{\C3W\99H\B9\C7b\C3J\DD\
  \AC\C73T)6cGC\A7\DE\D2\D9\A1=B2\BE\C2\F7lc\C9u\A0k\F06\AB\87t\6\81\91\8E\
  g\BA{\99\8E4\AC$z\9B)6\CEr\B3\82s\CA0\D3}Ar\91:4\DAC7u\F5y0\IWI\BA
4 \E0f\8E\B6R\CD3=S>\C7\FAE\97H4\AE9\ECjN\C8\C4~\98\D1\CBvB\87GPS\8A3\B8\F9&\85

```

Observation:

Surprisingly, we found our assumption partial true as it was able to recover few portion of the plaintext but there were many uncorrupted blocks and since it works independently, it should have recovered more portion of the plaintext as per our assumption.

CBC:

```

seed@VM: ~/Files
[02/19/23]seed@VM:~/Files$ openssl enc -aes-128-cbc -d \
> -in task5_cbc_corrupt.txt \
> -out task5_cbc_corrupt_dec.txt \
> -K 00111233445566778899aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
bad decrypt
140312406570304:error:0606506D:digital envelope routines:EVP_DecryptFinal_ex:wrong final block length:crypto/evp/evp_enc.c:572:
[02/19/23]seed@VM:~/Files$

```

Result:

task5_cbc_corrupt_dec.txt
Share ~/Share/Labsetup/Files

```

1 The BMW M3 has a madness inside q\86
2
3 \E3\F7\DD\D8Pd\F3\9Av\E3@C1S~C2s\B6\A8\E3!\87\F0L}
  \93\C8\FF\B3yS\A2\9D\E9\DBR\EDxgI\94:m\A3\8A\ED\84\FAU\E6"m#K\95.qs\F4t\

```

Observation:

We found our assumption true as it was able to recover some portion of the plaintext but not all as it works sequentially and depends on the previous ciphertext block. So, once it encountered the corrupted ciphertext, the decryption process failed.

CFB:

```

[02/19/23]seed@VM:~/Files$ openssl enc -aes-128-cfb -d \
> -in task5_cfb_corrupt.txt \
> -out task5_cfb_corrupt_dec.txt \
> -K 00112223445566778899aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
[02/19/23]seed@VM:~/Files$

```

Result:

```
task5_cfb_corrupt_dec.txt
task5_cbc_corrupt_dec.txt
1 The BMW M3 has a madness inside it. IJ \D9\3Hw\4>N \D9\B39\BBb9\D7\E9e\EC\0\F9k\
\FEn\E9\ADGHq\DEr\FNT2" \87\C6\FE\S\BB\F8) [ \A0\995\D3\BB\82\EF$\80\Nt\AB\94\I
\DF\ufd\96\96\D3\D8\FB\B8\96\EF(\B3\E4\A1
2 \9A\Y\BF\AD\B5\FE\A9\7WN\9F\85\96u\FCD\8B\A1\A3\D0\Op\C6
3 \C4\EANb\ ] \D0o\84\ \00\D3~Ge\A0(
4 S\F0\A2\8E)8\F1\B5k\A0\EBq\B9\9B\9D\DE- \99\E0^\B7\E8;Q\A6\86P\9At\C1(\H\A:
\B0R\EF\F2\00* \B6\A6\B8\A9\FA '=0\847\ E0. \u\B5\9B" \87\11\95\91\A8\A7_ \E0\
```

Observation:

We found our assumption true as it is similar to CBC mode and it was able to recover a portion of the plaintext until it encountered the corrupted ciphertexts.

OFB:

```
[02/19/23] seed@VM:~/.../Files$ openssl enc -aes-128-ofb -d \
> -in task5_ofb_corrupt.txt \
> -out task5_ofb_corrupt_dec.txt \
> -K 001122334455566778899aabbccddeeff \
> -iv 0102030405060708
hex string is too short, padding with zero bytes to length
hex string is too long, ignoring excess
[02/19/23] seed@VM:~/.../Files$
```

Result:

```
task5_ofb_corrupt_dec.txt
1 The BMW M3 has a madness inside it. I\CCMF\D6uz\F0\8C1:t\94;\C7\E9\A8@99Dzmr\8
2 !\A1\D7\EAW\D6\ED\FBzW:\F9
3 HR8\BFK\FCg7\C1u\B5\F9\F0K4\EA\E2(\EA\BC\A1\AF\F0\C0$ \BD\B14\F5T\FD\A50\EE
\C0A\91)\AD\C7D"&\99\h[\chI\FF\C15eM\8BU\]r\AE3;\DDT\00\F7\E0\A0\BE\AC\DE
4 \DEM\ADR]\A8\A3\91yb\A3\90\C0d\EE;E%\B1\FE\81\C5\BAf\D4z\95\E6\DE\EFgU]0\A1\
\A4\85\C3\EC\F7k\B6Zy\B6\ee\EE\B7S R\F1\9A\8D0m\BEd\~)\ED\8D\CCWWC
5 \BF\9D\9A\E4\91G\EF\9A\DC\C3\F8\B6\D6\E3cv\AF\A5\BC%
\AD\FC5\FC\C6\B0\E2k\9Bd\CFjL\86e\D0\F8\CEw\8F\B7\FB\803\AE\84k\A0\FF\9C\F
\86
```

Observation:

We found our assumption true as it was able to recover some portion of the plaintext and since this mode is similar to CFB, it didn't able to recover the remaining portion of plaintext after it encountered corrupted ciphertext.

Conclusion:

After successfully doing each task, I learned how to use different secret key encryptions, their algorithms, used different ciphers to encrypt and decrypt texts and pictures, paddings, and error propagation. I also learned about pros and cons of different modes.