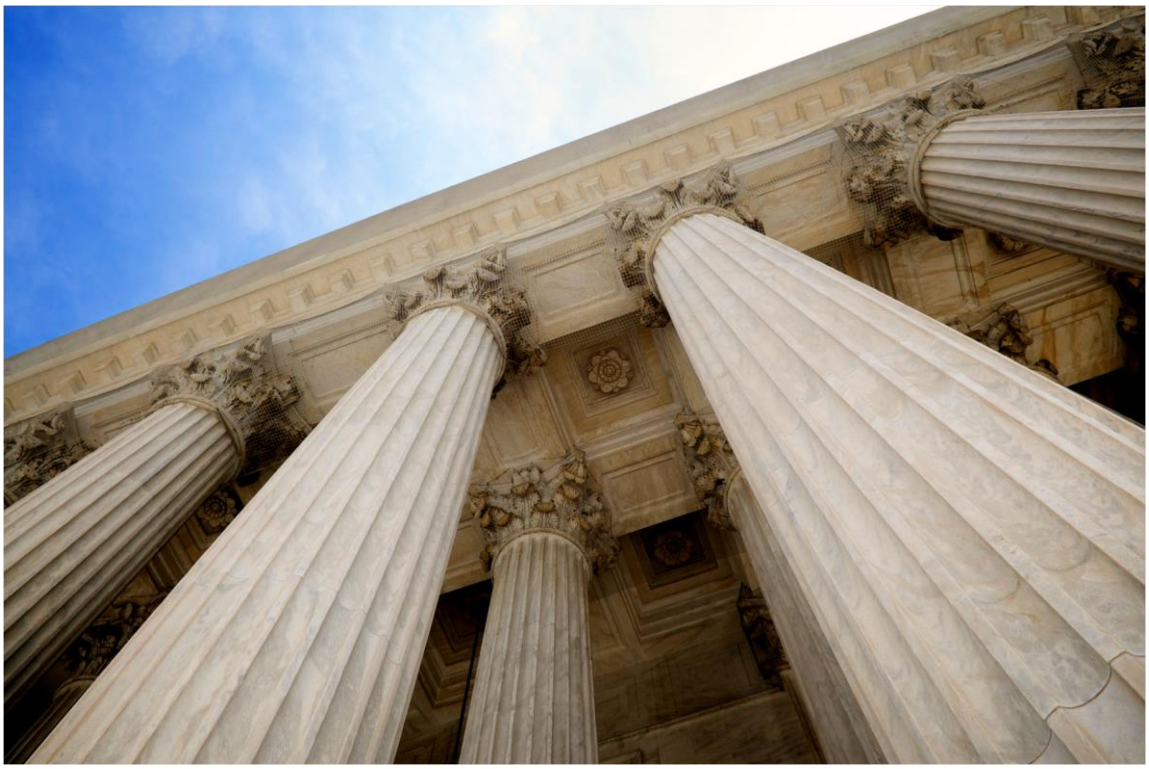


COMPUTER NETWORK SECURITY

F20CN – COURSEWORK 1

Jason Shawn D' Souza | H00202543 | 23/10/2017

Contents

Introduction

For this coursework we had been asked to carry out six tasks as follows

Task 1: Doing encryption

Task 2: Encryption modes

Task 3: Data Corruption

Task 4: Reflection on cipher and modes

Task 5: Hash sums

Task 6: Dictionary Attack

All this was done on Linux (CentOS 7 used via virtual box) and using Openssl. The overall goal of this coursework was to develop an understanding of cryptography as well as the various ciphers and modes.

Task 1: Doing Encryption

For this task we were tasked with encrypting a file with three different ciphers with each of three modes. I chose the following ciphers and the following modes:

Ciphers

1. aes-128
2. bf
3. camellia-128

Modes

1. ecb
2. cbc
3. ofb

Following is the required evidence

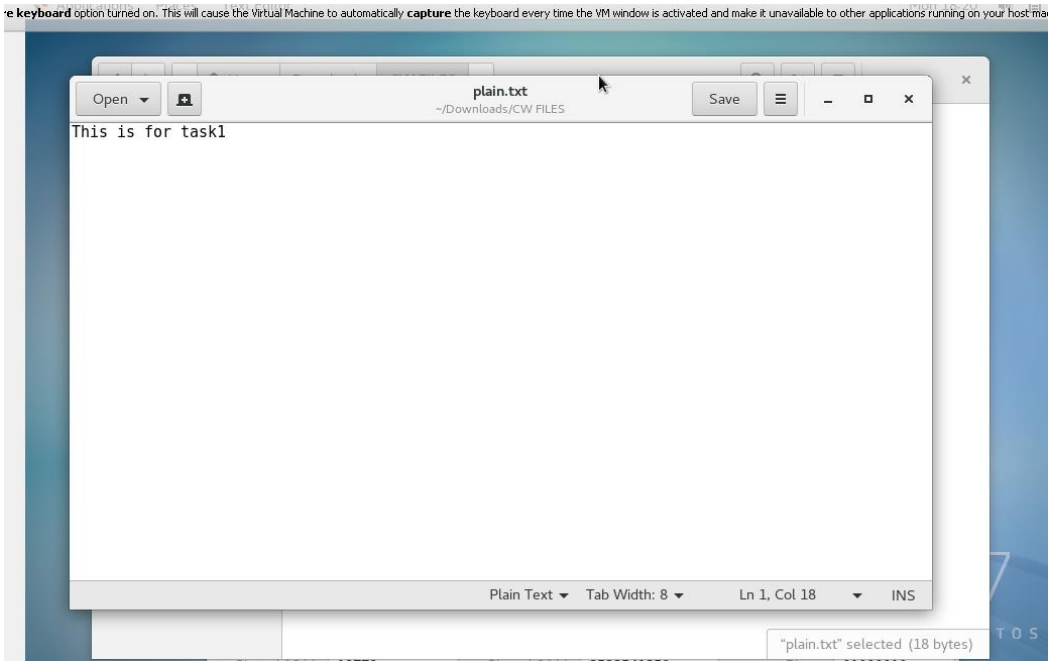


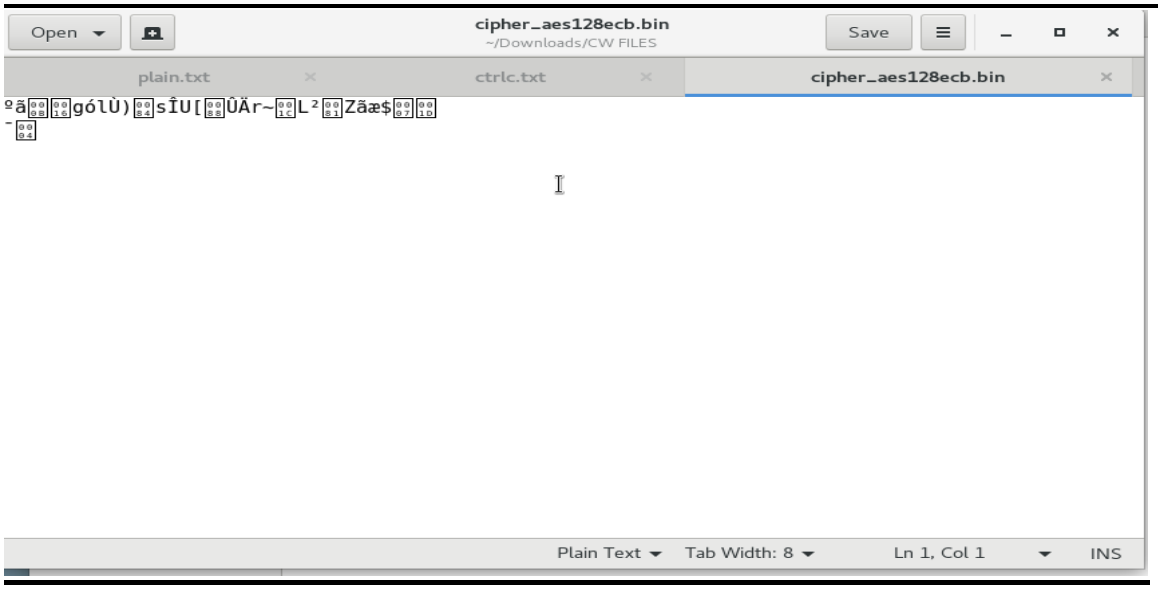
Figure 1.1

Figure 1.1 above shows the plain.txt containing some text

1. AES-128 Cipher

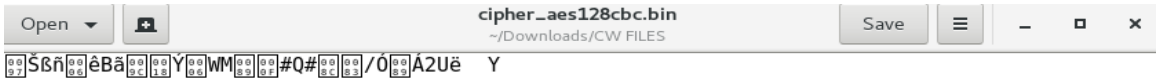
First done with ECB mode

```
[jasondsouza@localhost CW FILES]$ openssl enc -aes-128-ecb -e  -in plain.txt -ou
t cipher_aes128ecb.bin \-K  00112233445566778889aabbccddeeff \-iv 01020304050607
08
warning: iv not use by this cipher
[jasondsouza@localhost CW FILES]$
```



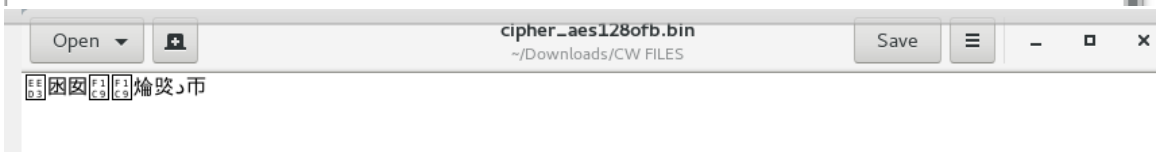
Next done using CBC mode

```
[jasondsouza@localhost CW FILES]$ openssl enc -aes-128-cbc -e  -in plain.txt-out
 cipher_aes128cbc.bin \-K  00112233445566778889aabbccddeeff \-iv 010203040506070
8
```



And the third mode of choice OFB

```
[jasondsouza@localhost CW FILES]$ openssl enc -aes-128-ofb -e  -in  plain.txt -o
ut cipher_aes128ofb.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060
708
```



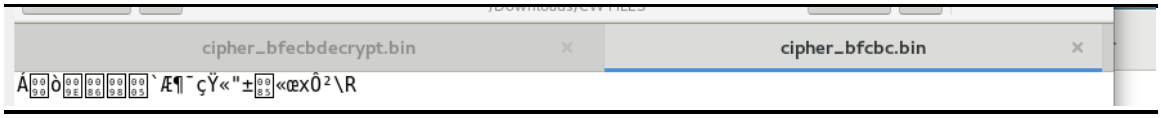
2. Bf Cipher

First done using the ECB mode

```
[jasondsouza@localhost CW FILES]$ openssl enc -bf-cbc -e  -in  plain.txt -out ci
pher_bfcbc.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
```

Next done using the CBC mode

```
[jasondsouza@localhost CW FILES]$ openssl enc -bf-cbc -e  -in  plain.txt -out ci
pher_bfcbc.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
```



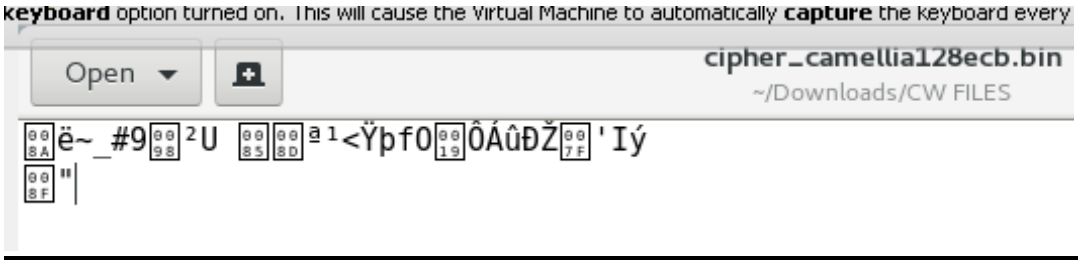
Finally done using the OFB mode

```
[jasondsouza@localhost CW FILES]$ openssl enc -bf-ofb -e  -in  plain.txt -out ci
pher_bfofb.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
[jasondsouza@localhost CW FILES]$
```

3. Camellia-128 Cipher

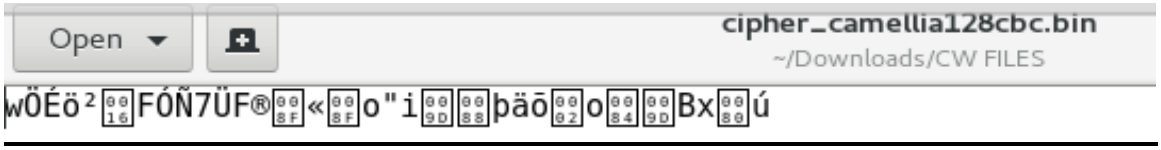
First done using the ECB mode

```
[jasondsouza@localhost CW FILES]$ openssl enc -camellia-128-ecb -e  -in  plain.t
xt -out cipher_camellia128ecb.bin \-K  00112233445566778889aabbccddeeff \-iv 010
2030405060708
warning: iv not use by this cipher
```



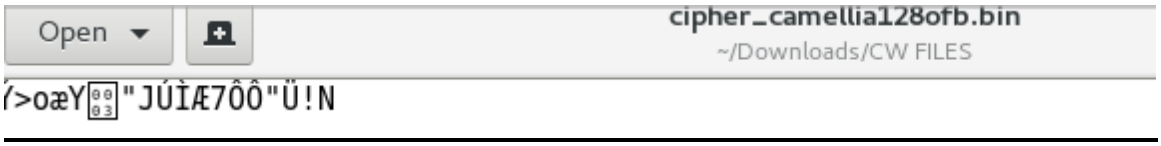
Next done using the CBC mode

```
[jasondsouza@localhost CW FILES]$ openssl enc -camellia-128-cbc -e  -in  plain.t
xt -out cipher_camellia128cbc.bin \-K  00112233445566778889aabbccddeeff \-iv 010
2030405060708
```



Finally done using the OFB mode

```
[jasondsouza@localhost CW FILES]$ openssl enc -camellia-128-ofb -e  -in  plain.t
xt -out cipher_camellia128ofb.bin \-K  00112233445566778889aabbccddeeff \-iv 010
2030405060708
```



Task 2: Encryption Modes

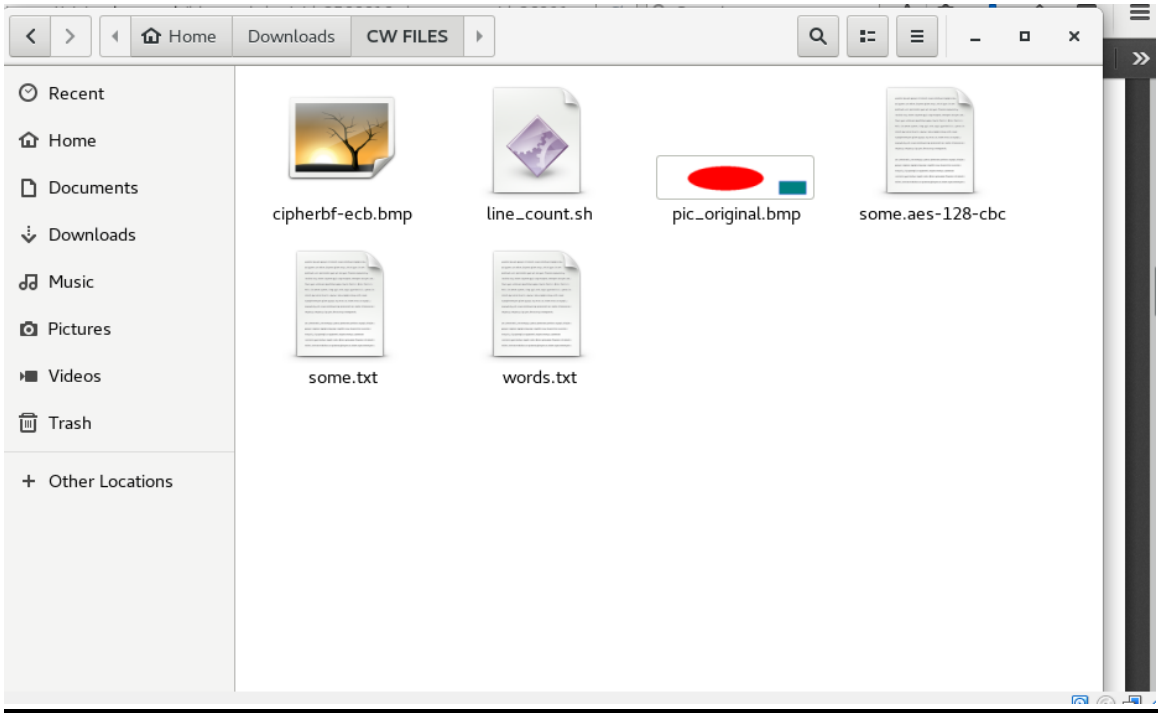
For this task we had to encrypt the following picture (named pic_original.bmp)

The picture above had to be encrypted using any cipher but only in the ECB and CBC (chain block cipher) so as to compare the two modes of encryption. **The cipher I chose was aes-128 with evidence as follows:**

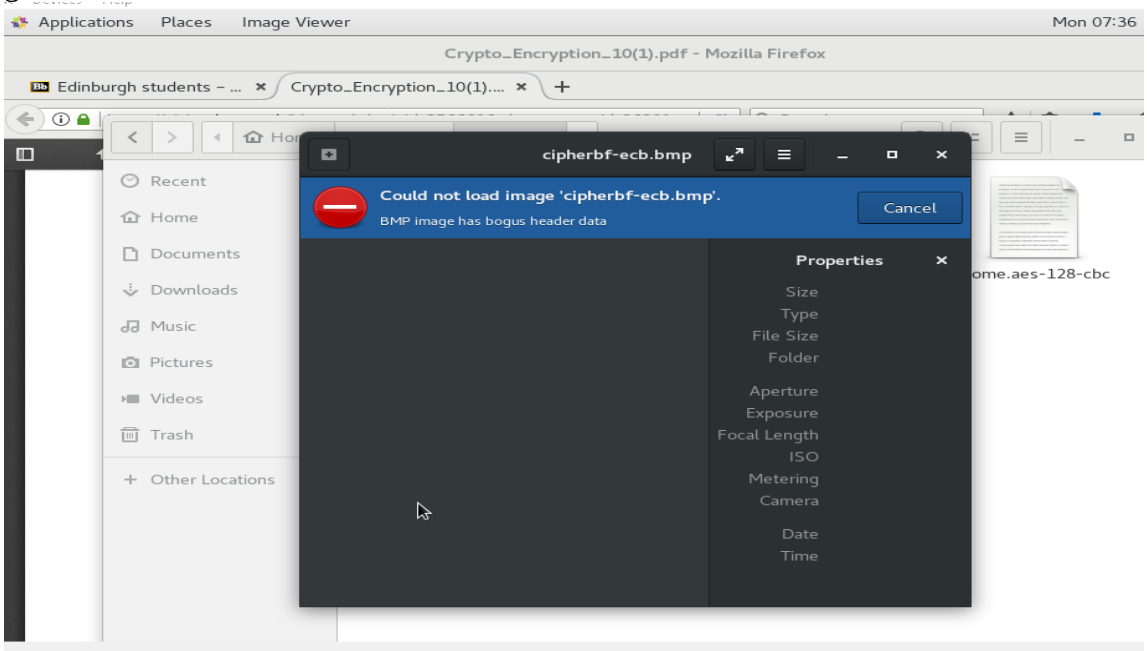
ECB

```
[jasondsouza@localhost CW FILES]$ openssl enc -aes-128-ecb -e  -in pic_original.
bmp -out cipher_aes128ecb.bmp \-K  00112233445566778889aabbccddeeff \-iv 0102030
405060708
```

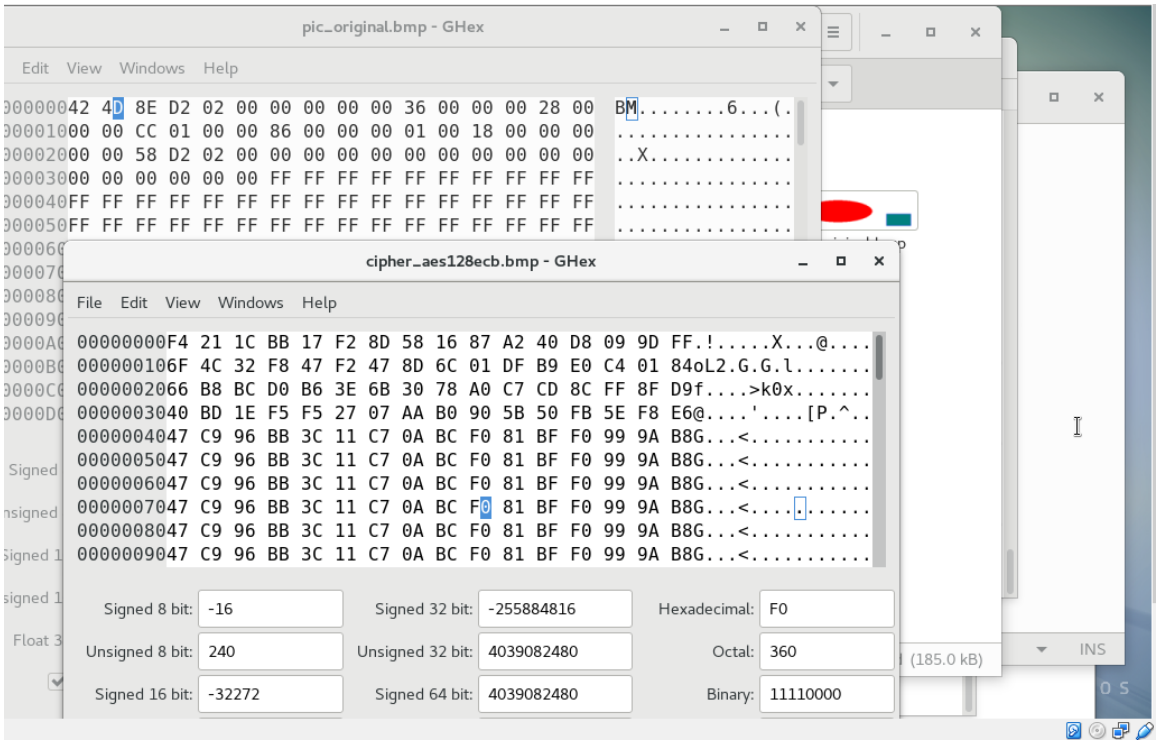
The encrypted file as shown in the directory



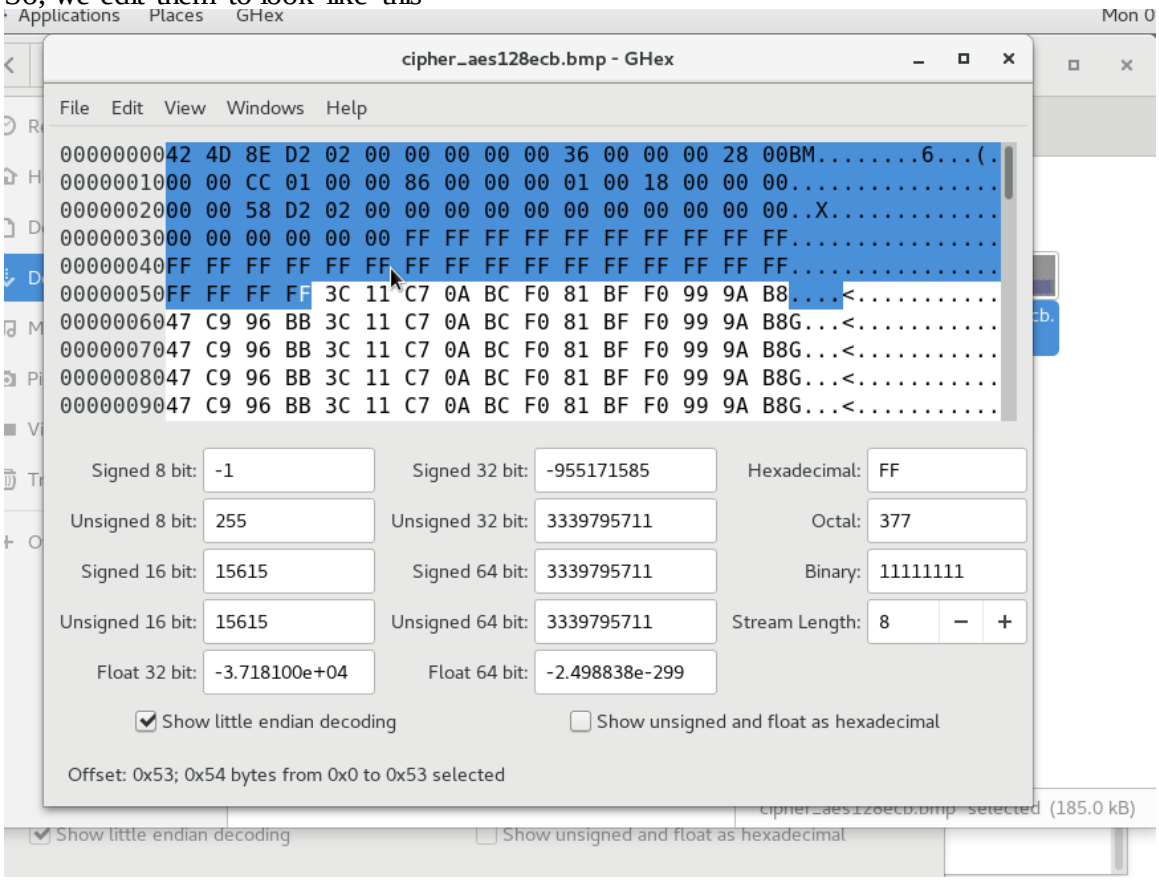
On opening the file, the following error appears as suggested in the coursework guidelines



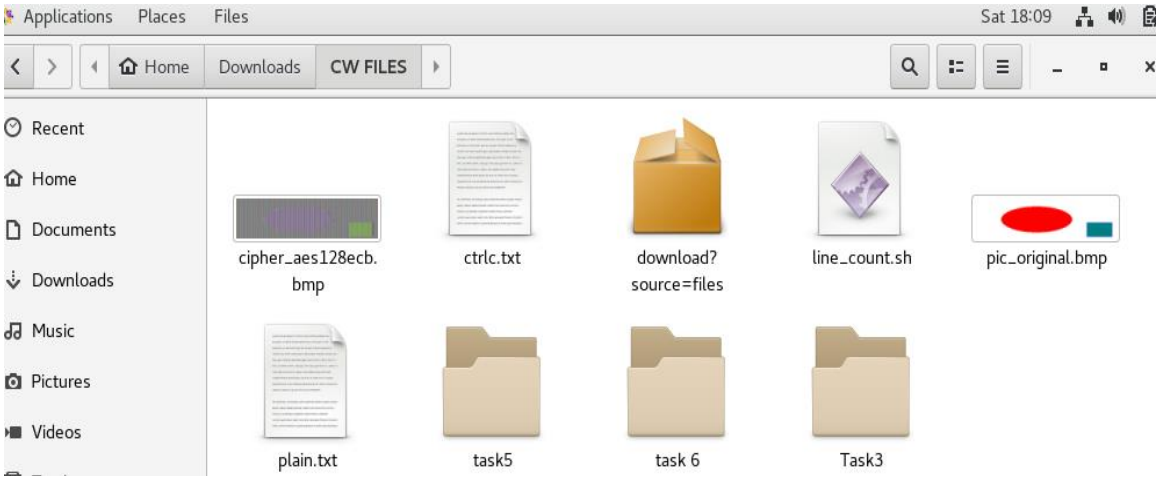
On comparing the original bmp and the encrypted file in ghex we observe the headers are different(first 54 bytes)



So, we edit them to look like this



Then the following is observed:



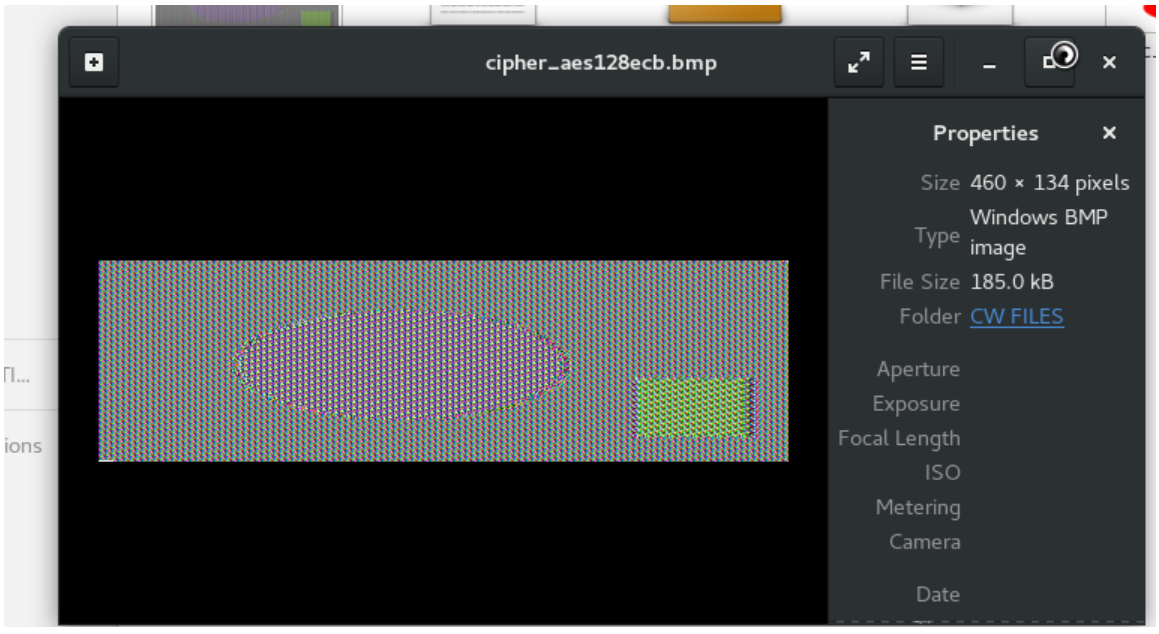
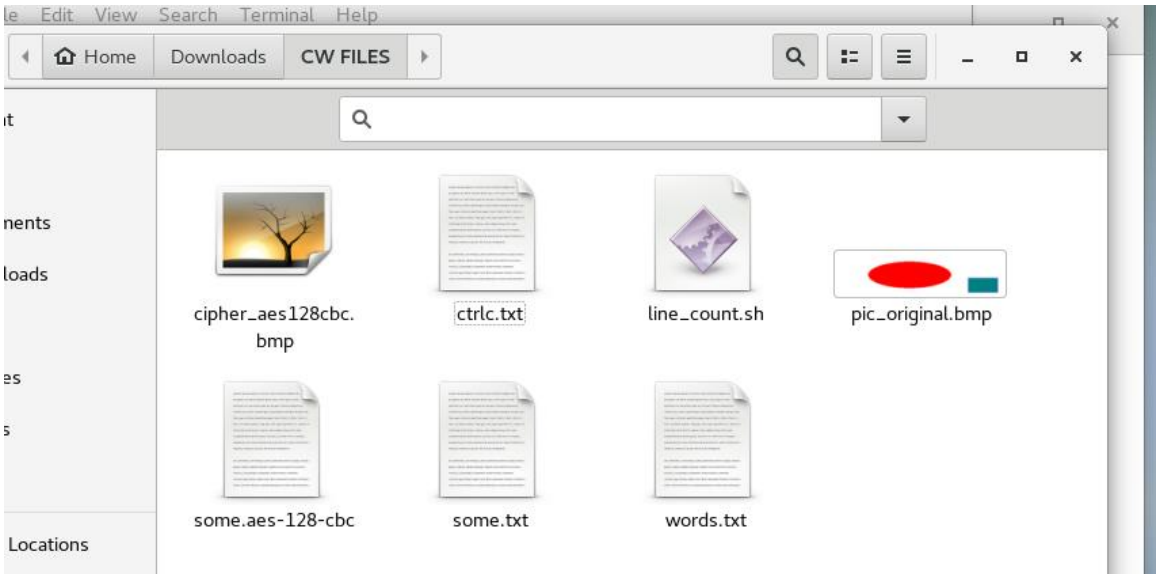


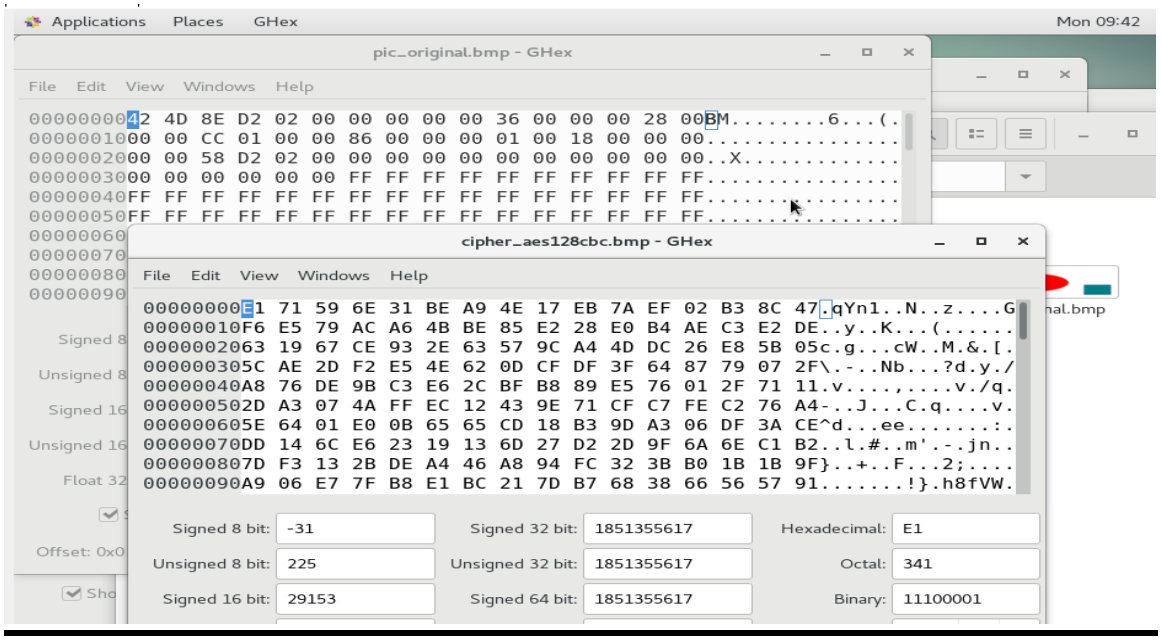
Figure 2.1

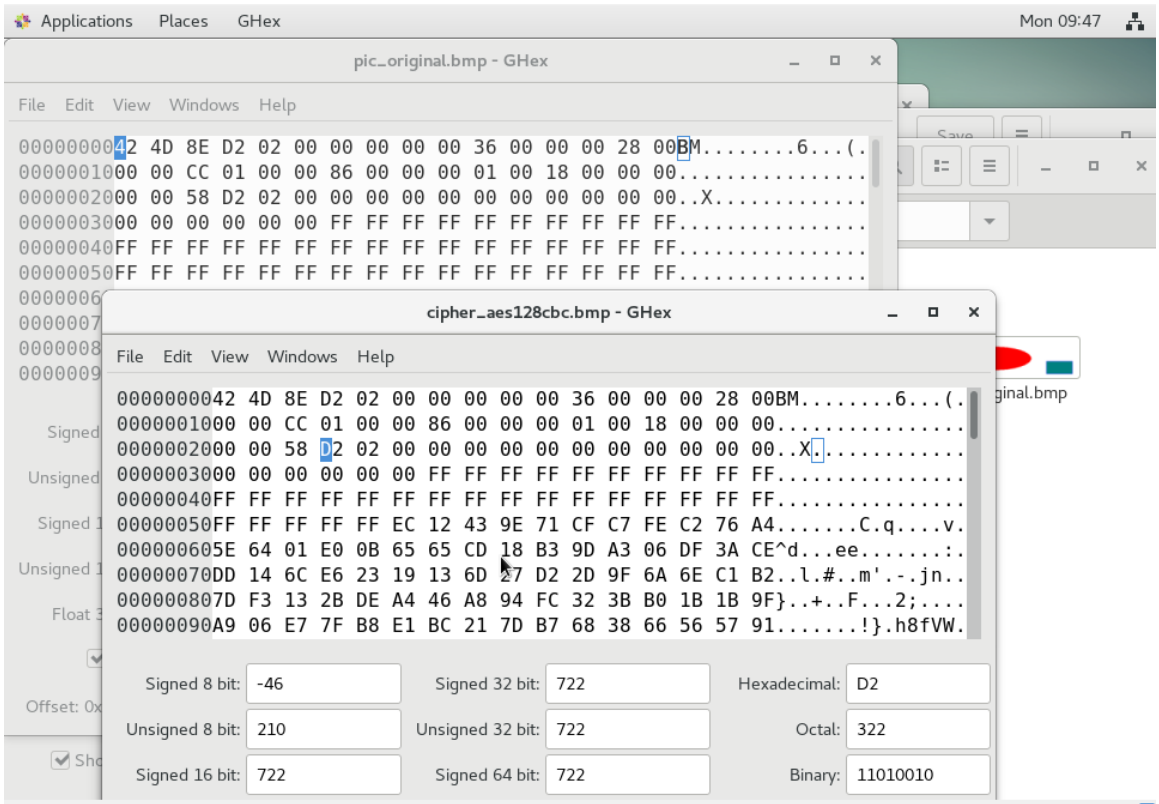
CBC

Similar steps as in the case of ecb

```
[jasondsouza@localhost CW FILES]$ openssl enc -aes-128-cbc -e  -in pic_original.
bmp -out cipher_aes128cbc.bmp \-K  00112233445566778889aabbccddeeff \-iv 0102030
405060708
[jasondsouza@localhost CW FILES]$
```






Applications Places GHex Mon 09:47
pic_original.bmp - GHex
File Edit View Windows Help
cipher_aes128cbc.bmp - GHex
File Edit View Windows Help
0000000042 4D 8E D2 02 00 00 00 00 00 36 00 00 00 28 00BM.....6...(.
0000001000 00 CC 01 00 00 86 00 00 00 01 00 18 00 00 00.....
0000002000 00 58 D2 02 00 00 00 00 00 00 00 00 00 00 00..X.....
0000003000 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF.....
00000040FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF.....
00000050FF FF FF FF FF EC 12 43 9E 71 CF C7 FE C2 76 A4.....C.q....v.
000000605E 64 01 E0 0B 65 65 CD 18 B3 9D A3 06 DF 3A CE^d...ee.....:.
00000070DD 14 6C E6 23 19 13 6D 27 D2 2D 9F 6A 6E C1 B2..l.#..m'.-.jn..
000000807D F3 13 2B DE A4 46 A8 94 FC 32 3B B0 1B 1B 9F}..+..F...2;....
00000090A9 06 E7 7F B8 E1 BC 21 7D B7 68 38 66 56 57 91.....!}.h8fVW.
Signed 8 bit: -46
Signed 32 bit: 722
Hexadecimal: D2
Unsigned 8 bit: 210
Unsigned 32 bit: 722
Octal: 322
Signed 16 bit: 722
Signed 64 bit: 722
Binary: 11010010

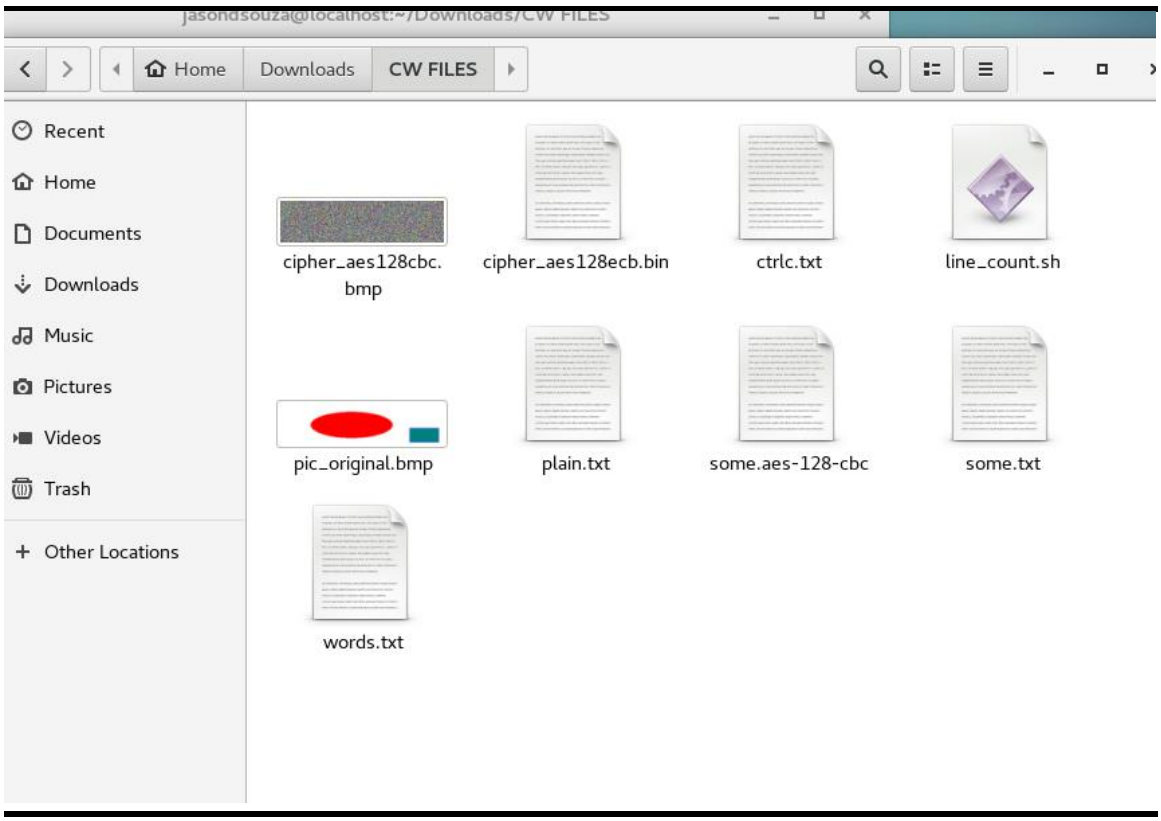

jasondsouza@localhost:~/Downloads/CW FILES
Home Downloads CW FILES
Recent
Home
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
cipher_aes128cbc.bmp
cipher_aes128ecb.bin
ctrlc.txt
line_count.sh
pic_original.bmp
plain.txt
some.aes-128-cbc
some.txt
words.txt

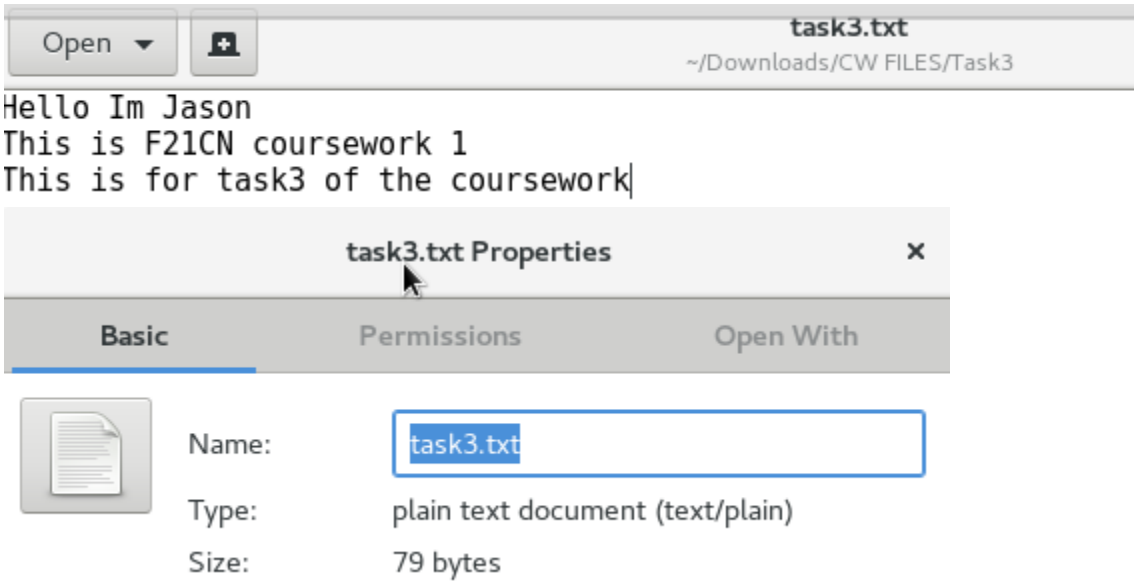
Figure 2.2

Observations

As can be inferred from the Figures 2.1 and 2.2 more of the picture is visible in the encrypted ECB image whereas nothing can be seen in the CBC image (all scrambled). From an image point of view ECB is better but from an encryption standpoint CBC is better (In the case of ECB it can be understood that the original picture must contain a square and a circle while this is not the case for the CBC image)

Task 3: Data Corruption

For this task we were asked to create a text file which was at least 64 bytes. Following is the file I made



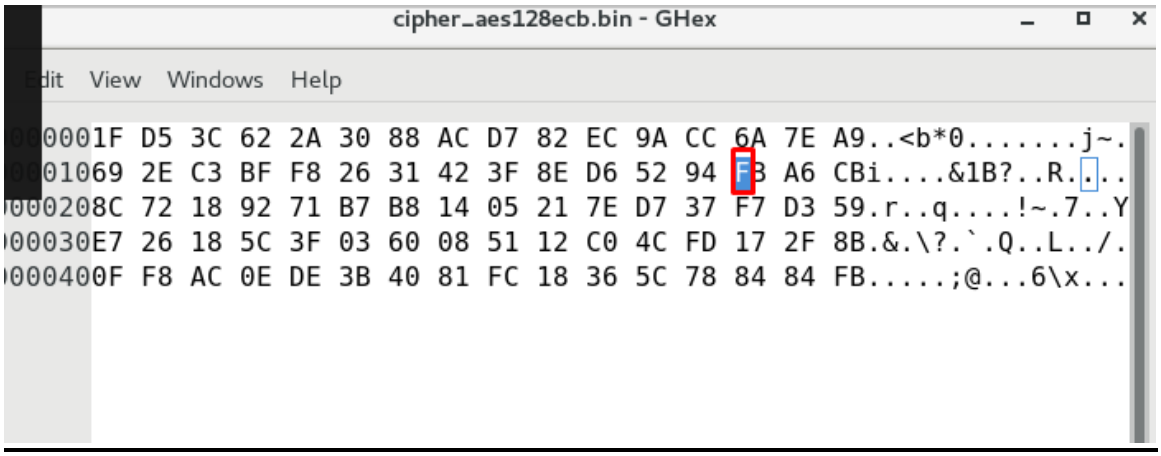
We were then asked to encrypt this using the AES-128 cipher with the ECB, CBC, CFB and OFB modes as shown:

```
[jasondsouza@localhost Task3]$ openssl enc -aes-128-ecb -e  -in task3.txt -out c
ipher_aes128ecb.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
warning: iv not use by this cipher

[jasondsouza@localhost Task3]$ openssl enc -aes-128-cbc -e  -in task3.txt -out c
ipher_aes128cbc.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708

[jasondsouza@localhost Task3]$ openssl enc -aes-128-ofb -e  -in task3.txt -out c
ipher_aes128ofb.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
[jasondsouza@localhost Task3]$

[jasondsouza@localhost Task3]$ openssl enc -aes-128-cfb -e  -in task3.txt -out c
ipher_aes128cfb.bin \-K  00112233445566778889aabbccddeeff \-iv 0102030405060708
```

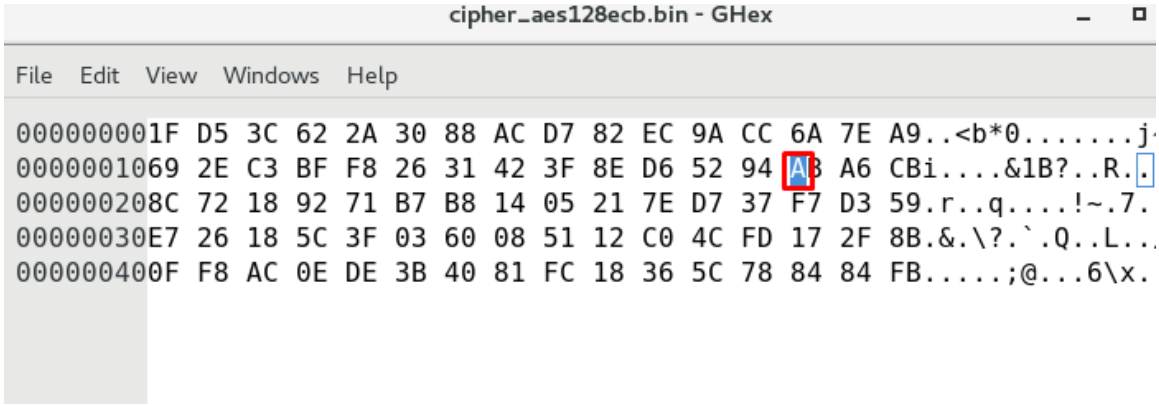
Then we were asked to change a single bit of the 30th byte to corrupt the encrypted file

EFB

This is the original encrypted file

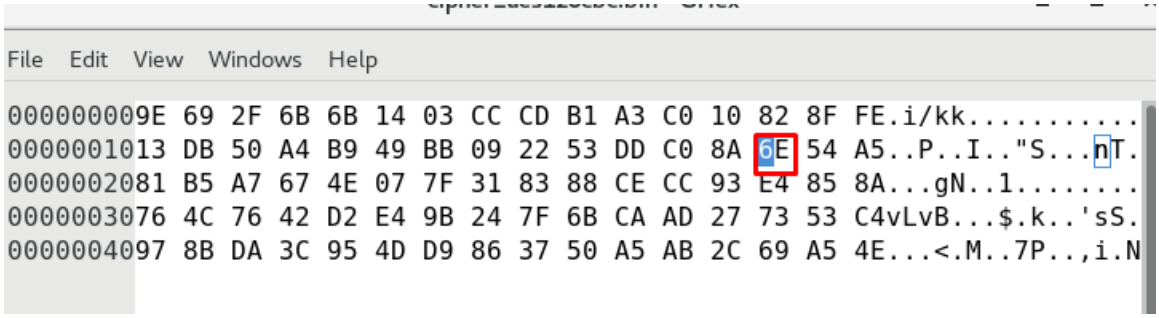


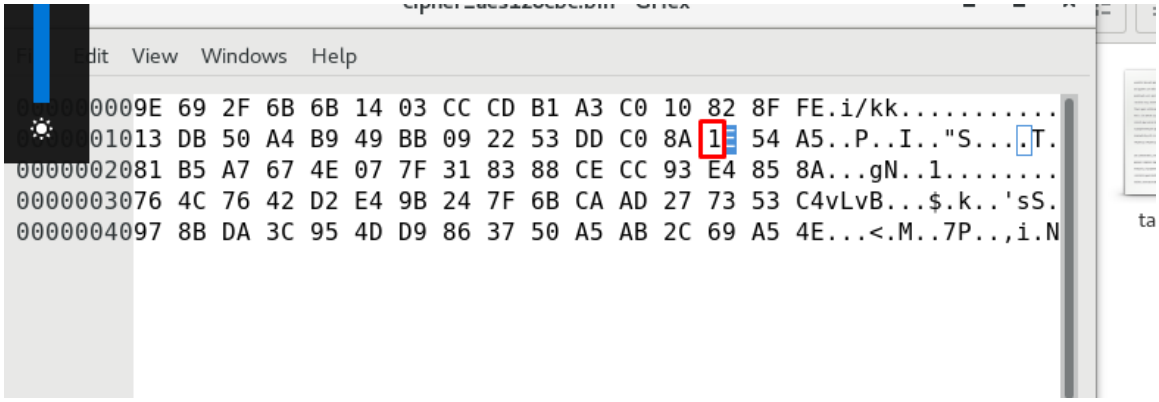
This is after changing the single digit of the 30th byte



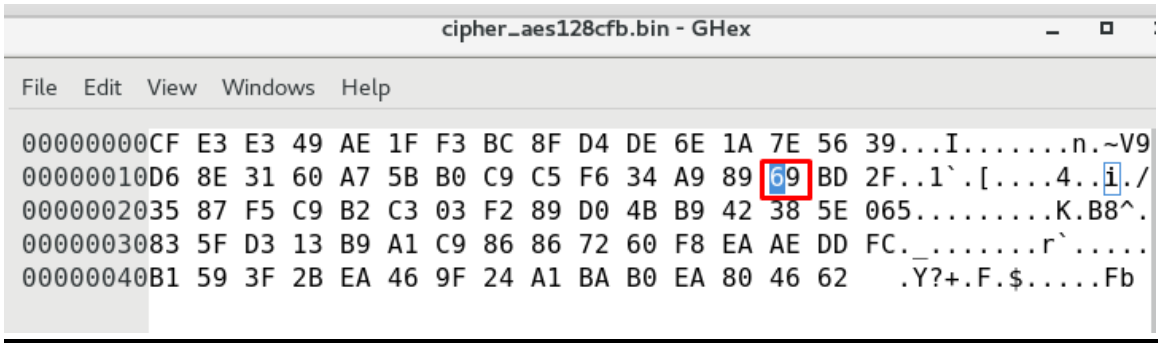
CBC

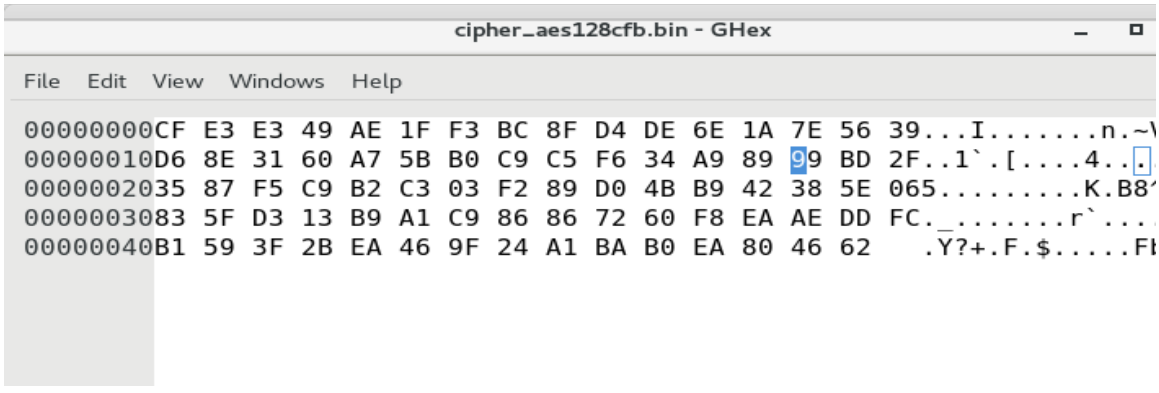
Just like in the case of ECB the following is done for CBC, CFB and OFB.



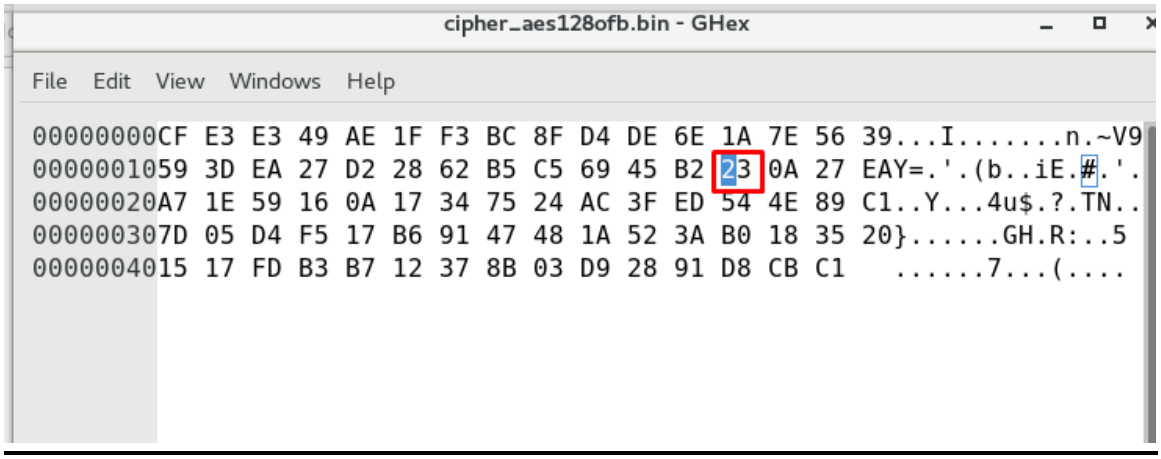

Edit View Windows Help
0000000 9E 69 2F 6B 6B 14 03 CC CD B1 A3 C0 10 82 8F FE .i/kk..........
0000010 13 DB 50 A4 B9 49 BB 09 22 53 DD C0 8A 1E 54 A5 ..P..I.."S....T.
0000020 81 B5 A7 67 4E 07 7F 31 83 88 CE CC 93 E4 85 8A ...gN..1........
0000030 76 4C 76 42 D2 E4 9B 24 7F 6B CA AD 27 73 53 C4 vLvB...$.k..'sS.
0000040 97 8B DA 3C 95 4D D9 86 37 50 A5 AB 2C 69 A5 4E ...<.M..7P..,i.N

CFB


cipher_aes128cfb.bin - GHex
File Edit View Windows Help
00000000 CF E3 E3 49 AE 1F F3 BC 8F D4 DE 6E 1A 7E 56 39 ...I.......n.~V9
00000010 D6 8E 31 60 A7 5B B0 C9 C5 F6 34 A9 89 69 BD 2F ..1`.[....4..i./
00000020 35 87 F5 C9 B2 C3 03 F2 89 D0 4B B9 42 38 5E 06 5.........K.B8^.
00000030 83 5F D3 13 B9 A1 C9 86 86 72 60 F8 EA AE DD FC ._.......r`.....
00000040 B1 59 3F 2B EA 46 9F 24 A1 BA B0 EA 80 46 62 .Y?+.F.$.....Fb


cipher_aes128cfb.bin - GHex
File Edit View Windows Help
00000000 CF E3 E3 49 AE 1F F3 BC 8F D4 DE 6E 1A 7E 56 39 ...I.......n.~
00000010 D6 8E 31 60 A7 5B B0 C9 C5 F6 34 A9 89 99 BD 2F ..1`.[....4..
00000020 35 87 F5 C9 B2 C3 03 F2 89 D0 4B B9 42 38 5E 06 5.........K.B8
00000030 83 5F D3 13 B9 A1 C9 86 86 72 60 F8 EA AE DD FC ._.......r`...
00000040 B1 59 3F 2B EA 46 9F 24 A1 BA B0 EA 80 46 62 .Y?+.F.$.....F

OFB



My answer before task

This task asked to answer which of the corrupted encrypted files i.e. ECB, CBC, CFB or OFB would be best at recovering the corrupted information. I feel OFB (Output Feedback Mode) would be the better mode out of the 4 mentioned as it works as a stream cipher and has the advantages of a stream cipher (as soon as byte arrives it can be sent instead of waiting for blocks).

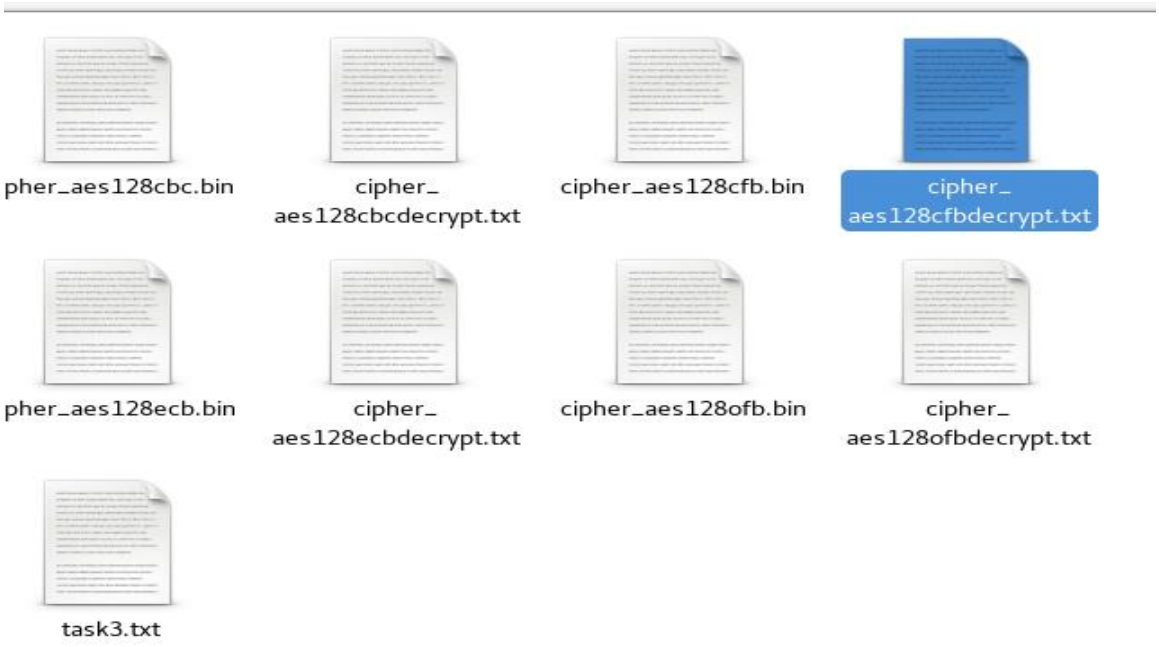
The Task

Decryption of the corrupted encrypted file was carried out

```
[jasondsouza@localhost Task3]$ openssl enc -aes-128-ecb -d -in cipher_aes128ecb.
bin -out cipher_aes128ecbdecrypt.txt \-K 00112233445566778889aabbccddeeff \-iv 0
102030405060708
warning: iv not use by this cipher
```

```
[jasondsouza@localhost Task3]$ openssl enc -aes-128-cfb -d -in cipher_aes128cfb.
bin -out cipher_aes128cfbdecrypt.txt \-K 00112233445566778889aabbccddeeff \-iv 0
102030405060708
```

Similar commands for cbc and ofb were used and the following files are produced (the decrypt.txts for the four modes)



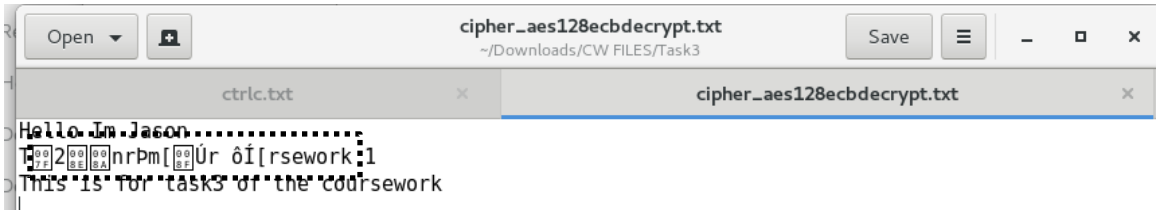


Figure 3.1

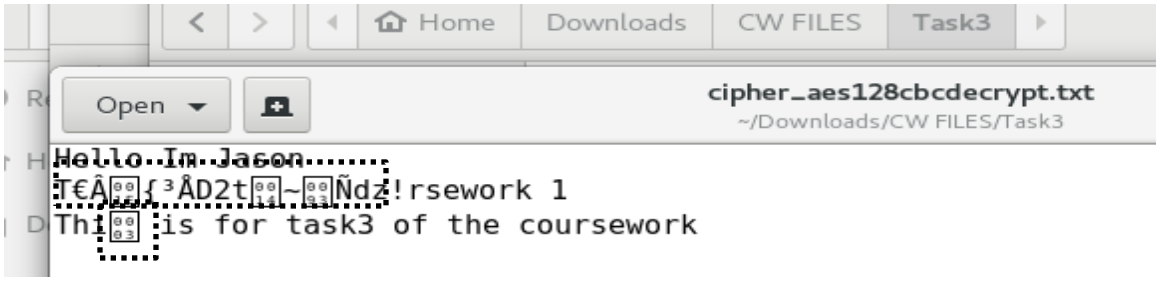


Figure 3.2

Figure 3.3

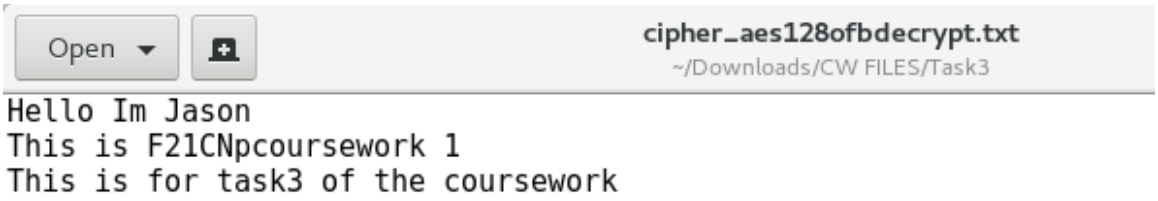


Figure 3.4

In the ECB file only one block of data in the second line is affected (ECB encrypts each block independently) as can be seen in figure 3.1.

In the CBC file seen in figure 3.2, two blocks are affected one in the second line and one in the third line of my file.

As can be seen in figure 3.4, OFB has recovered most of the information from the correct file. Also, the most errors are observed in the CFB decrypted file shown in figure 3.3.

Task 4: Reflection on cipher and modes

In this task we are asked to reflect on the results of Tasks 2 and 3 in more detail.

Task 2 reflection

As done in task2, Figures 2.1 and 2.2 show the final encrypted images obtained using ECB and CBC modes respectively. Since we use encryption to keep data hidden, the ECB mode would not be preferred to the CBC mode. Main problem here is that blocks of plaintext that are identical are encrypted onto blocks of the ciphertext that are also identical which is why the data is not well hidden. We were asked to specifically answer which mode would be better to hide the highest degree of information and the answer is CBC because XOR is done before the encryption of each ciphertext block.

Task 3 reflection

As shown under task 3, OFB would be the best mode to use in case of fault during data transmission as almost all the text file was almost recovered.

Task 5: Hash sums

For this task we were asked to use openssl and check the SHA-1 checksum of three files (words.txt, some.txt, some.aes-128-cbc).

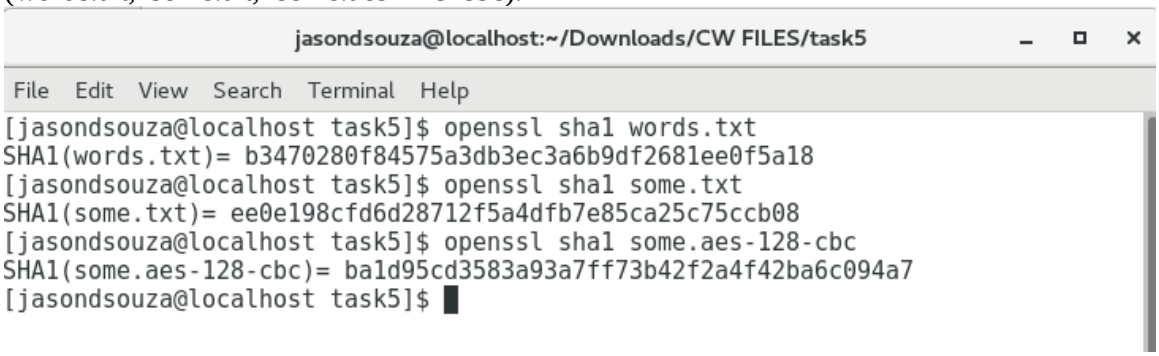


Figure 5.1

Figure 5.1 shows the commands used to check the checksum of the files.
Following are the answers to the questions asked

1. Comparing these values to those given in the coursework guidelines, the checksum of some.txt is different.
2. The cryptographic concept here is that of data integrity.
3. Even if a few bytes are different/tampered with the checksum will be different so yes it can be used to check if files from origin are the same or been tampered with
4. The checksums are of same length as whatever the input size the output is of fixed length (MD5's is 128 while SHA-1's is 160 bytes)

Task 6: Dictionary Attack

For this task we were asked to write a bash script that provides the key used to encode a file. The shell script I wrote was named task6.sh and requires 3 arguments to run on the shell

./task6.sh <file with possible keys> <any plaintext file> <Decrypted file>

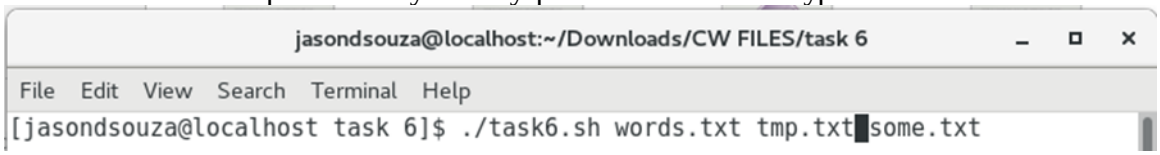


Figure 6.1

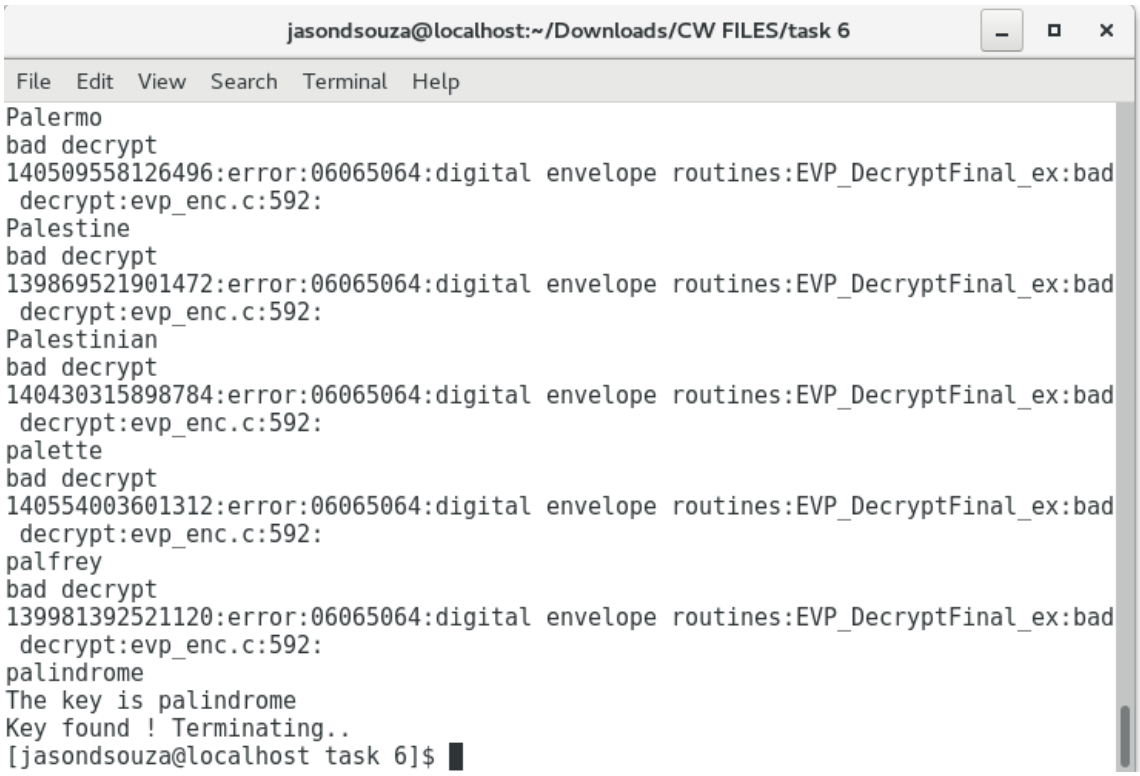


Figure 6.2

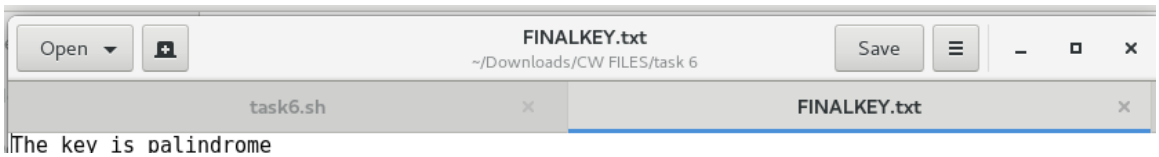


Figure 6.3

Figures 6.1, 6.2 and 6.3 show the working of my script and following I have pasted my script code

I used

```
openssl enc  -aes-128-cbc -d -in some.aes-128-cbc -out $compare -nosalt
-k $LINE
```

Instead of

```
openssl enc  -aes-128-cbc -d -in some.aes-128-cbc -out $compare -nosalt
-K $LINE
```

This was done keeping in mind the coursework guidelines (iv NOT specified)

Note: The shell script copied from my linux machine and pasted in the following page was beautified using the website - http://hilite.me/

```
#!/bin/bash
# Simple line count example, using bash provided by Hans Wolfgang-Loidl
# Edits made to be compatible with coursework 1 to find the decryption key
by Jason Shawn D'Souza H00202543
# with the scenarios of the coursework guidelines
# Bash tutorial: http://linuxconfig.org/Bash_scripting_Tutorial#8-2-read-
file-into-bash-array
# My scripting linked:
http://www.macs.hw.ac.uk/~hwloidl/docs/index.html#scripting
#
# Usage: ./task6.sh <file with possible keys> <a file to compare decrypted
text with possible key> <the actual decrypted file>
#eg: task6.sh words.txt compare.txt some.txt   ( NOTE: compare.txt has to
be plaintext file and some.txt should contain the decrypted text
# ------------------------------------------------------------------------
-----

# Link filedescriptor 10 with stdin
exec 10<&0
# stdin replaced with a file supplied as a first argument
exec < $1
in=$1
compare=$2
some=$3
key="FINALKEY.txt"
while read LINE
do
          echo $LINE
      SSL=`openssl enc  -aes-128-cbc -d -in some.aes-128-cbc -out $compare
-nosalt  -k $LINE`
#echo SSL
#Nested if loop this first if condition is to check the exit status of the
above command
#Using line_count.sh as reference this condition was modified( -eq instead
of -ne)
#Since the exit status would be 0 for a successfull decryption
if [ $? -eq 0 ]; then
#This condition compares the decryption with key from a line of the
inputted file to that
#of the already provided decrypted file on vision
 if [[ $(<$compare) == $(<$some) ]] ;then
         echo "The key is $LINE"
        #echo $LINE > keystmp.txt
        echo "The key is $LINE" > $key
        echo "Key found ! Terminating.."
        exit 1
fi
fi
done
echo "The key is `cat ${key}`"




# restore stdin from filedescriptor 10
# and close filedescriptor 10
exec 0<&10 10<&-
```

Summary

As mentioned in the Introduction, a set of six tasks had to be completed. Each task was completed with ease the exception being task 6 where various iterations of shell scripts were tested. Throughout this coursework, I learnt about the diverse types of ciphers and modes available. The results I expected before I did a task was the same after I did a task, this was due to the good teaching and lecture material made available to me. The only real problems were in task 6 where I had problems where the program would accept multiple keys (this was before I added the exit 1 and during the testing phase) but this was solved in the end with evidence being the code I have provided. **To conclude, all tasks of this coursework have been completed successfully and the required evidence has been provided.**