Jeffrey Cho

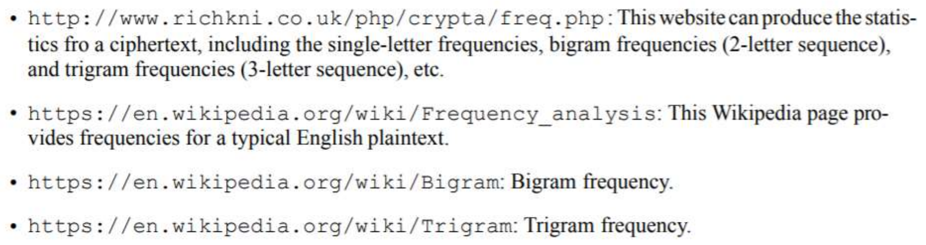
Professor Charles Palmer

COSC 55

September 27, 2020

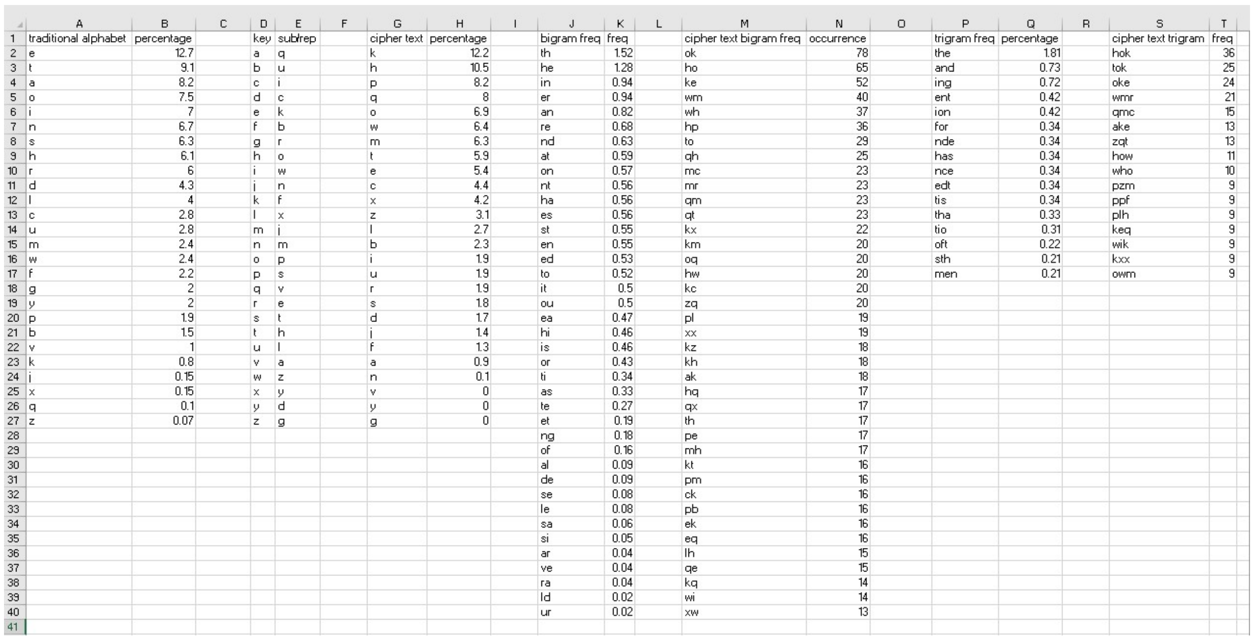
Lab 1: Lab Report

Task 1 Frequency Analysis: Given the provided online resources **(1.0)**, I have created an excel document **(1.1)** that highlights all the letter, bigram, trigram, and cipher text frequencies.



- `http://www.richkni.co.uk/php/crypta/freq.php`: This website can produce the statistics fro a ciphertext, including the single-letter frequencies, bigram frequencies (2-letter sequence), and trigram frequencies (3-letter sequence), etc.
- `https://en.wikipedia.org/wiki/Frequency_analysis`: This Wikipedia page provides frequencies for a typical English plaintext.
- `https://en.wikipedia.org/wiki/Bigram`: Bigram frequency.
- `https://en.wikipedia.org/wiki/Trigram`: Trigram frequency.

(1.0)

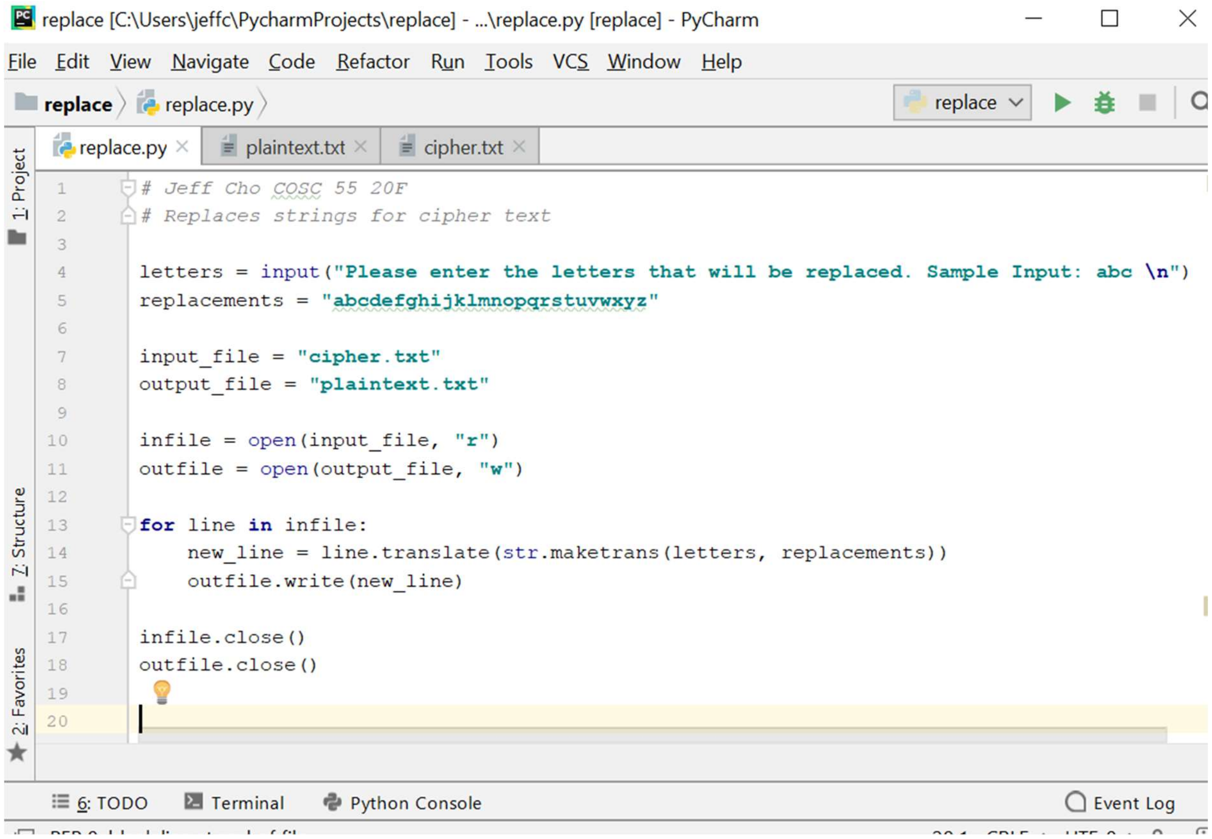


	A	B	C	D	E	F	G	H	I
1	traditional alphabet	percentage		key	sub/rep		cipher text	percentage	
2	e	12.7		a	q		k	12.2	
3	t	9.1		b	u		h	10.5	
4	a	8.2		c	i		p	8.2	
5	o	7.5		d	c		q	8	
6	i	7		e	k		o	6.9	
7	n	6.7		f	b		w	6.4	
8	s	6.3		g	r		m	6.3	
9	h	6.1		h	o		t	5.9	
10	r	6		i	w		e	5.4	
11	d	4.3		j	n		c	4.4	
12	l	4		k	f		x	4.2	
13	c	2.8		l	x		z	3.1	
14	u	2.8		m	j		l	2.7	
15	m	2.4		n	m		b	2.3	
16	w	2.4		o	p		i	1.9	
17	f	2.2		p	s		u	1.9	
18	g	2		q	v		r	1.9	
19	y	2		r	e		s	1.8	
20	p	1.9		s	t		d	1.7	
21	b	1.5		t	h		j	1.4	
22	v	1		u	l		f	1.3	
23	k	0.8		v	a		a	0.9	
24	j	0.15		w	z		n	0.1	
25	x	0.15		x	y		v	0	
26	q	0.1		y	d		y	0	
27	z	0.07		z	g		g	0	
28									
29									
30									
31									
32									
33									
34									
35									
36									
37									
38									
39									
40									
41									

	J	K	L	M	N	O	P	Q	R	S	T
1	bigram freq	freq		cipher text bigram freq	occurrence		trigram freq	percentage		cipher text trigram	freq
2	th	1.52		ok	78		the	1.81		hok	36
3	he	1.28		ho	65		and	0.73		tok	25
4	in	0.94		ke	52		ing	0.72		oke	24
5	er	0.94		wm	40		ent	0.42		wmr	21
6	an	0.82		wh	37		ion	0.42		qmc	15
7	re	0.68		hp	36		for	0.34		ake	13
8	nd	0.63		to	29		nde	0.34		zqt	13
9	at	0.59		qh	25		has	0.34		how	11
10	on	0.57		mc	23		nce	0.34		who	10
11	nt	0.56		mr	23		edt	0.34		pzm	9
12	ha	0.56		qm	23		tis	0.34		ppf	9
13	es	0.56		qt	23		tha	0.33		plh	9
14	st	0.55		kx	22		tio	0.31		keq	9
15	en	0.55		km	20		oft	0.22		wik	9
16	ed	0.53		oq	20		sth	0.21		kxx	9
17	to	0.52		hw	20		men	0.21		owm	9
18	it	0.5		kc	20						
19	ou	0.5		zq	20						
20	ea	0.47		pl	19						
21	hi	0.46		xx	19						
22	is	0.46		kz	18						
23	or	0.43		kh	18						
24	ti	0.34		ak	18						
25	as	0.33		hq	17						
26	te	0.27		qx	17						
27	et	0.19		th	17						
28	ng	0.18		pe	17						
29	of	0.16		mh	17						
30	al	0.09		kt	16						
31	de	0.09		pm	16						
32	se	0.08		ck	16						
33	le	0.08		pb	16						
34	sa	0.06		ek	16						
35	si	0.05		eq	16						
36	ar	0.04		lh	15						
37	ve	0.04		qe	15						
38	ra	0.04		kq	14						
39	ld	0.02		wi	14						
40	ur	0.02		xw	13						
41											

(1.1)

The process to figure out the plaintext and encryption key involved numerous testing. However, to create starting data, that will eventually be modified, the frequencies of letters in the cipher text and the traditional alphabet were compared and sorted in descending order next to each other. From there, the corresponding frequencies of letters in the traditional alphabet and the cipher text were determined, which can be seen in **col 1** and **col 3** in **(1.1)**. The testing portion would have to include trial and error of replacing such letters, and to minimize the time taken for testing, a python testing script has been written to make the testing much easier, which is shown in **(1.2)**.

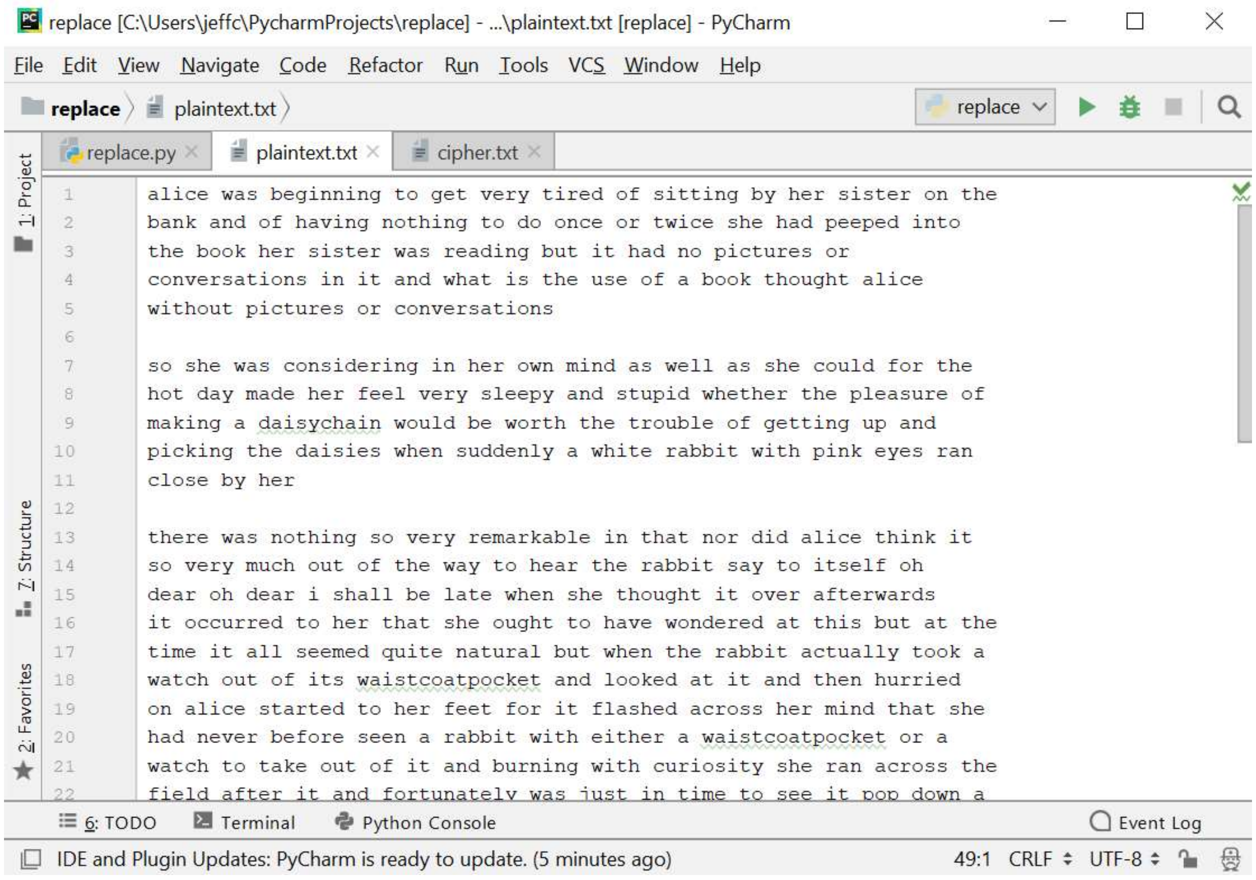

replace [C:\Users\jeffc\PycharmProjects\replace] - ...\replace.py [replace] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
replace
replace.py
replace.py
plaintext.txt
cipher.txt
1: Project
7: Structure
2: Favorites
6: TODO
Terminal
Python Console
Event Log

```python
# Jeff Cho COSC 55 20F
# Replaces strings for cipher text

letters = input("Please enter the letters that will be replaced. Sample Input: abc \n")
replacements = "abcdefghijklmnopqrstuvwxyz"

input_file = "cipher.txt"
output_file = "plaintext.txt"

infile = open(input_file, "r")
outfile = open(output_file, "w")

for line in infile:
    new_line = line.translate(str.maketrans(letters, replacements))
    outfile.write(new_line)

infile.close()
outfile.close()

```

(1.2)

The python script simply takes in the input as a string concatenated together with no spaces between. It also assumes that the first letter inputted is the sub (substitution) for a and so on. Then the python program takes in the cipher.txt as input and creates a new output file which tests the inputted encryption key. The encryption key is not fully complete of course until there are no spelling errors in the output file and the story makes sense. Soon after testing, the encryption key was recovered, which is in **col. 2** in **(1.1)**. The **'key'** part is the traditional alphabet, and the

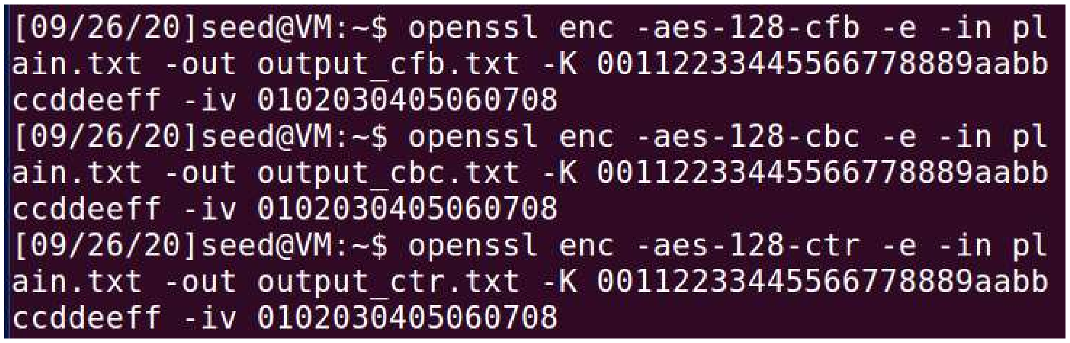
‘sub/rep’ is what the cipher text replaced the traditional alphabet letters with. The plaintext is pictured below **(1.3).**


replace [C:\Users\jeffc\PycharmProjects\replace] - ...\plaintext.txt [replace] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
replace plaintext.txt
replace.py plaintext.txt cipher.txt
1: Project
7: Structure
2: Favorites
6: TODO Terminal Python Console Event Log
IDE and Plugin Updates: PyCharm is ready to update. (5 minutes ago) 49:1 CRLF UTF-8

```
alice was beginning to get very tired of sitting by her sister on the
bank and of having nothing to do once or twice she had peeped into
the book her sister was reading but it had no pictures or
conversations in it and what is the use of a book thought alice
without pictures or conversations

so she was considering in her own mind as well as she could for the
hot day made her feel very sleepy and stupid whether the pleasure of
making a daisychain would be worth the trouble of getting up and
picking the daisies when suddenly a white rabbit with pink eyes ran
close by her

there was nothing so very remarkable in that nor did alice think it
so very much out of the way to hear the rabbit say to itself oh
dear oh dear i shall be late when she thought it over afterwards
it occurred to her that she ought to have wondered at this but at the
time it all seemed quite natural but when the rabbit actually took a
watch out of its waistcoatpocket and looked at it and then hurried
on alice started to her feet for it flashed across her mind that she
had never before seen a rabbit with either a waistcoatpocket or a
watch to take out of it and burning with curiosity she ran across the
field after it and fortunately was just in time to see it pop down a
```

(1.3)

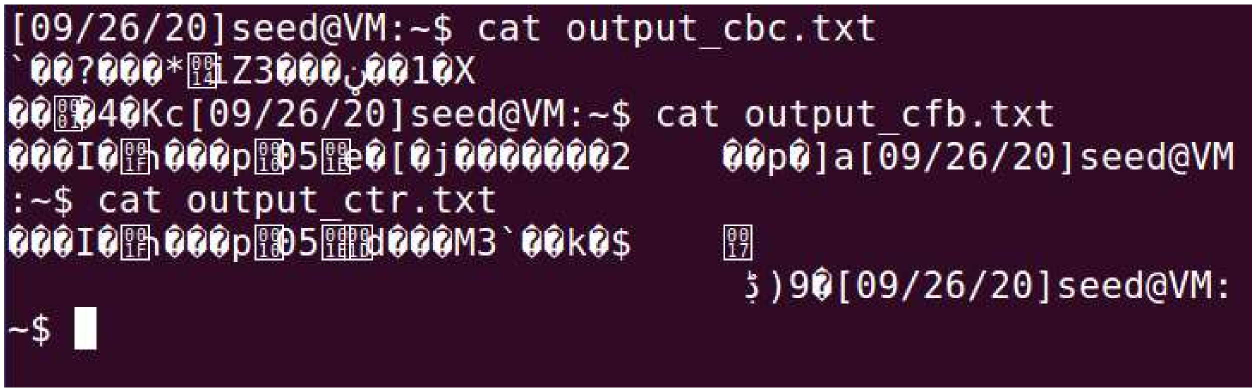
Task 2 Encryption using Different Ciphers and Modes: First in order to try out different ciphers, a **plain.txt** file in the VM was created. The contents strictly in the file are “*hello my name is Jeff. I am in cs 55.*” The file is strictly 38 bytes. The three ciphers used were “*aes-128-cbc, aes-128-cfb, aes-128-ctr*”, which can be seen in the command line arguments **(2.0)**.



```
[09/26/20]seed@VM:~$ openssl enc -aes-128-cfb -e -in pl
ain.txt -out output_cfb.txt -K 00112233445566778889aabb
ccddeeff -iv 0102030405060708
[09/26/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in pl
ain.txt -out output_cbc.txt -K 00112233445566778889aabb
ccddeeff -iv 0102030405060708
[09/26/20]seed@VM:~$ openssl enc -aes-128-ctr -e -in pl
ain.txt -out output_ctr.txt -K 00112233445566778889aabb
ccddeeff -iv 0102030405060708
```

(2.0)

Because of how small the file is, there was not noticeable time difference in speed for the commands. However, the encrypted contents were noticeably different, which can be seen in **(2.1).**



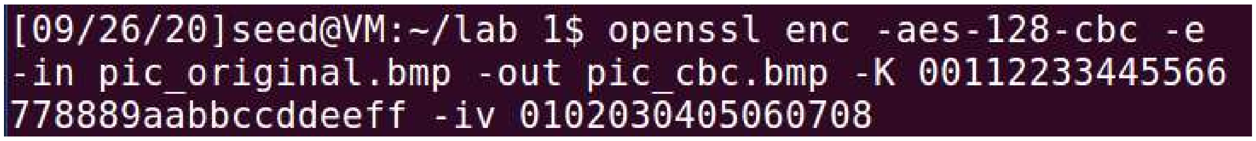
```
[09/26/20]seed@VM:~$ cat output_cbc.txt
`��?���*␔iZ3���ٻ��1�X
��␁�4�Kc[09/26/20]seed@VM:~$ cat output_cfb.txt
���I�␟h���p␛�5␞e�[�j�������2    ��p�]a[09/26/20]seed@VM
:~$ cat output_ctr.txt
���I�␟h���p␛�5␛␝d���M3`��k�$    ␗
                                   ڌ)9�[09/26/20]seed@VM:
~$ █
```

(2.1)

There is a clear difference in regard to spacing in addition to the wide variety of symbols used to encrypt the file. However, one other noticeable different was the sizes of the resulting encryptions. The file sizes respectively for "***output_cbc.txt***, ***output_cfb.txt***, and ***output_ctr.txt***" are **48 bytes**, **38 bytes**, and **38 bytes**. For some reason it seems as though the **aes-128-cbc** cipher specifically has a different encrypted file size for its content.

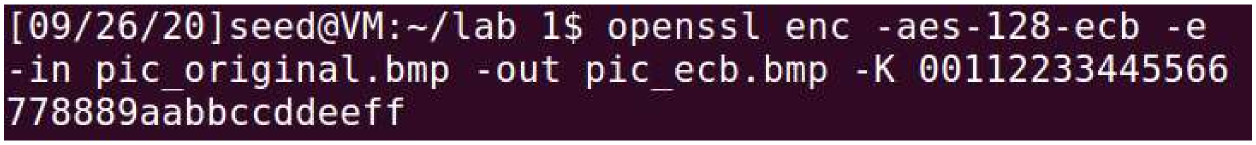
Task 3 Encryption Mode – ECB vs. CBC:

1) In order to encrypt the **pic_original.bmp** file, the following commands were used. The first command depicts **128-bit AES with CBC mode (3.0)**, and the second command depicts **128-bit AES with ECB mode (3.1)**.



```
[09/26/20]seed@VM:~/lab 1$ openssl enc -aes-128-cbc -e
-in pic_original.bmp -out pic_cbc.bmp -K 00112233445566
778889aabbccddeeff -iv 0102030405060708
```

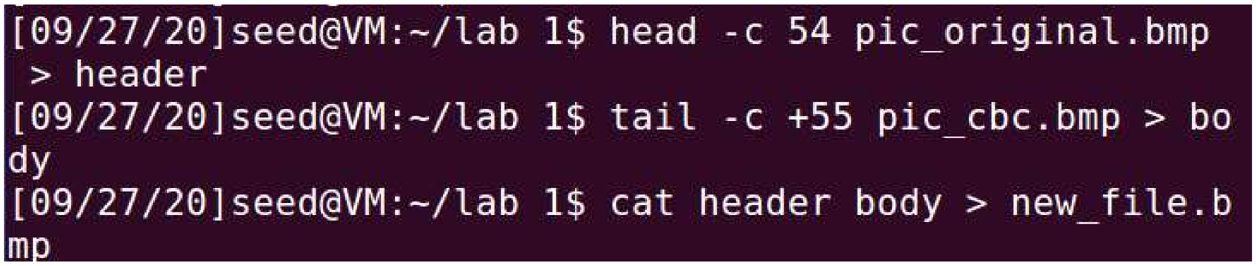
(3.0)



```
[09/26/20]seed@VM:~/lab 1$ openssl enc -aes-128-ecb -e
-in pic_original.bmp -out pic_ecb.bmp -K 00112233445566
778889aabbccddeeff
```

(3.1)

As you can see the **CBC** encrypted file was saved as **pic_cbc.bmp** file and the **ECB** encrypted file was saved as **pic_ecb.bmp** file. Using the following commands, the first 54 bytes of the encrypted pictures were replaced with the 54-byte header of the original picture (3.2).

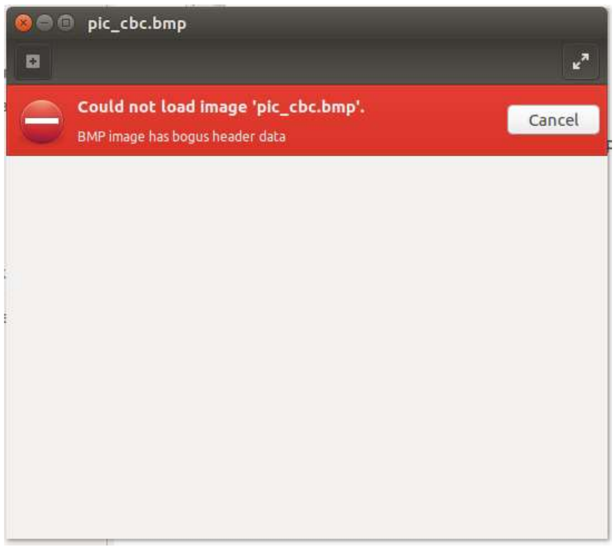


```
[09/27/20]seed@VM:~/lab 1$ head -c 54 pic_original.bmp
 > header
[09/27/20]seed@VM:~/lab 1$ tail -c +55 pic_cbc.bmp > bo
dy
[09/27/20]seed@VM:~/lab 1$ cat header body > new_file.b
mp
```

(3.2)

Here the first 54 bytes of the original picture was used as the header while the remaining bytes starting with 55 from the encrypted picture was used as the body. This led to the creation of the new_file.bmp, which is composed of the header and body.

2) The encrypted photos are depicted in **(3.3)** and **(3.4).** Here it is quite clear that there is no information that can be gained from the photos when comparing to the original photo in **(3.5)**.

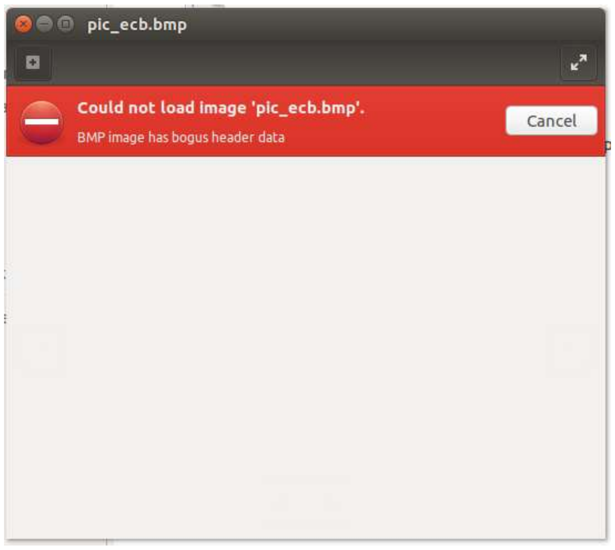


pic_cbc.bmp

Could not load image 'pic_cbc.bmp'.

BMP image has bogus header data

(3.3)



pic_ecb.bmp

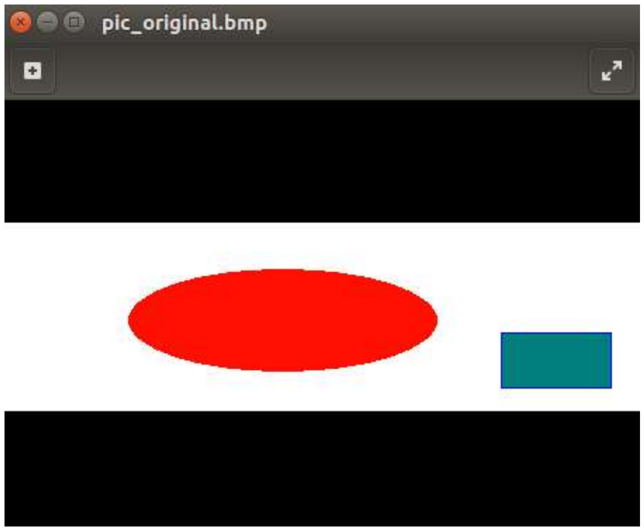
Could not load image 'pic_ecb.bmp'.

BMP image has bogus header data

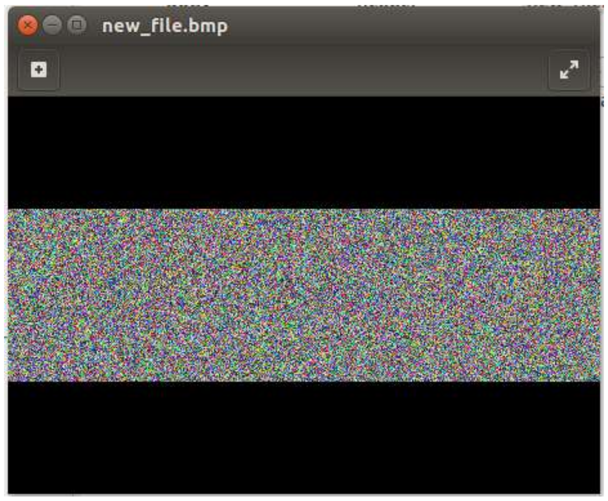
(3.4)

The only information we have for both the CBC and ECB encrypted photos is that there are bogus headers. However, when viewing the encrypted photos where the first 54 bytes from the original picture **(3.5)** replaced the bogus header, the ECB encrypted photo begins to appear **(3.7)**.

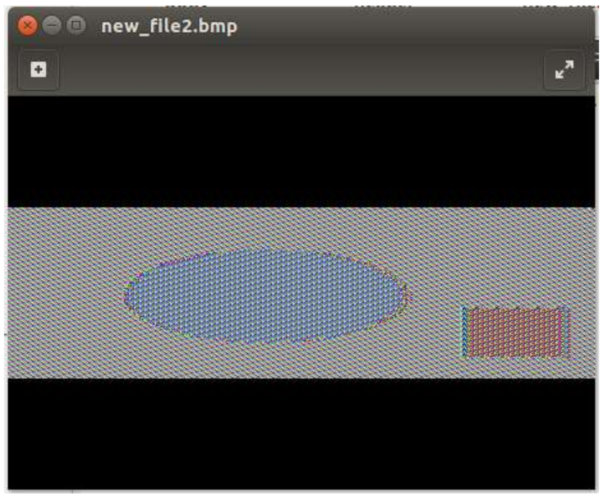
However, in the CBC encrypted photo **(3.6)**, there is still no clear indication of what the photo is; however, we can see a somewhat blurry version of the original picture in the ECB encrypted photo. Based on (3.7), we can get a basic shape and short preview of what the original picture could look like; however, for (3.6), it is still quite unclear as to what the original picture is.



(3.5)



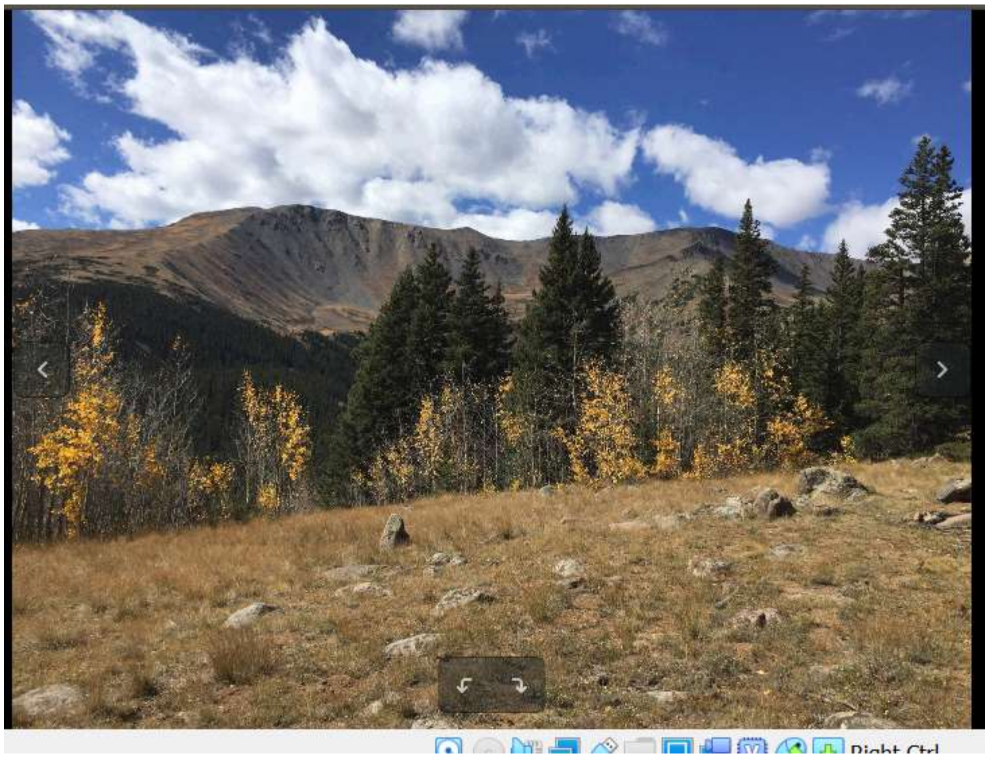
(3.6)



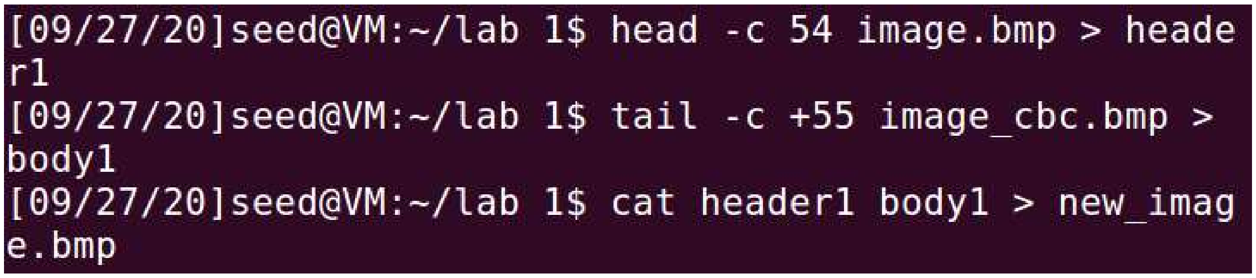
(3.7)

3) ***<u>Repeat using another picture</u>***

Another **.bmp** image was encrypted, and the image can be seen in **(3.8).** The terminal command lines are similar as seen in **(3.0)** and **(3.1)** with the only difference being image.bmp and not "**pic_original.bmp**."


(3.8)

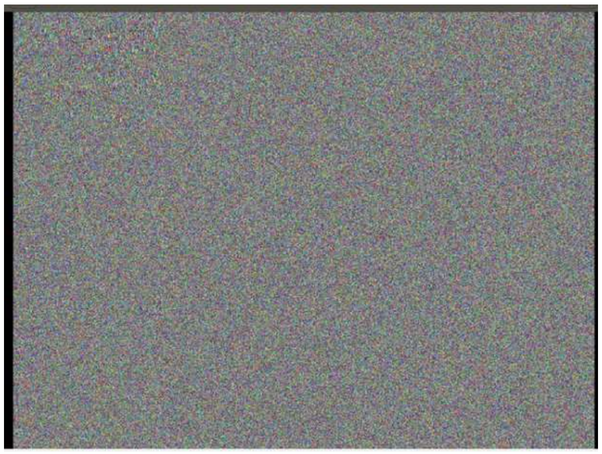
The similar methodology was used to encrypt the images with the first 54 bytes from the "**image.bmp**" and the remaining bytes from the encrypted images. We can see the command line arguments below **(3.9).** They are exactly the same as used previously.

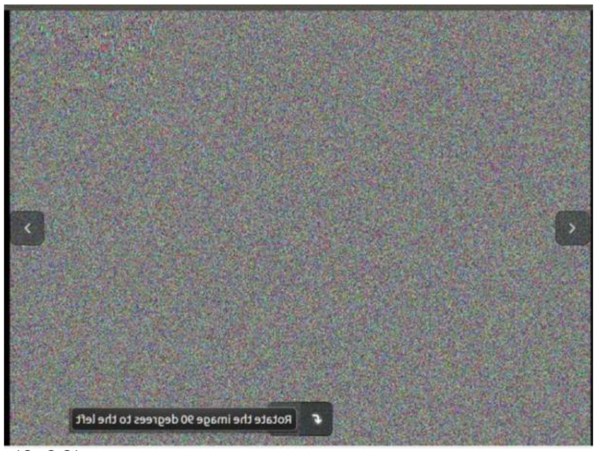


```
[09/27/20]seed@VM:~/lab 1$ head -c 54 image.bmp > heade
r1
[09/27/20]seed@VM:~/lab 1$ tail -c +55 image_cbc.bmp >
body1
[09/27/20]seed@VM:~/lab 1$ cat header1 body1 > new_imag
e.bmp
```

(3.9)

Below, we have the encrypted images (with the byte modifications). Unlike the previous trial with the "**pic_original.bmp**", here we see that even the "**image_ecb_encrypted.bmp**" is not as revealing as before for "**pic_original.bmp" (3.92)**. However, similar to the **cbc** image for "**pic_original.bmp**," the "**image_cbc_encrypted.bmp**" also shares the same encrypted screen where we get little to no information about the original image **(3.91).**

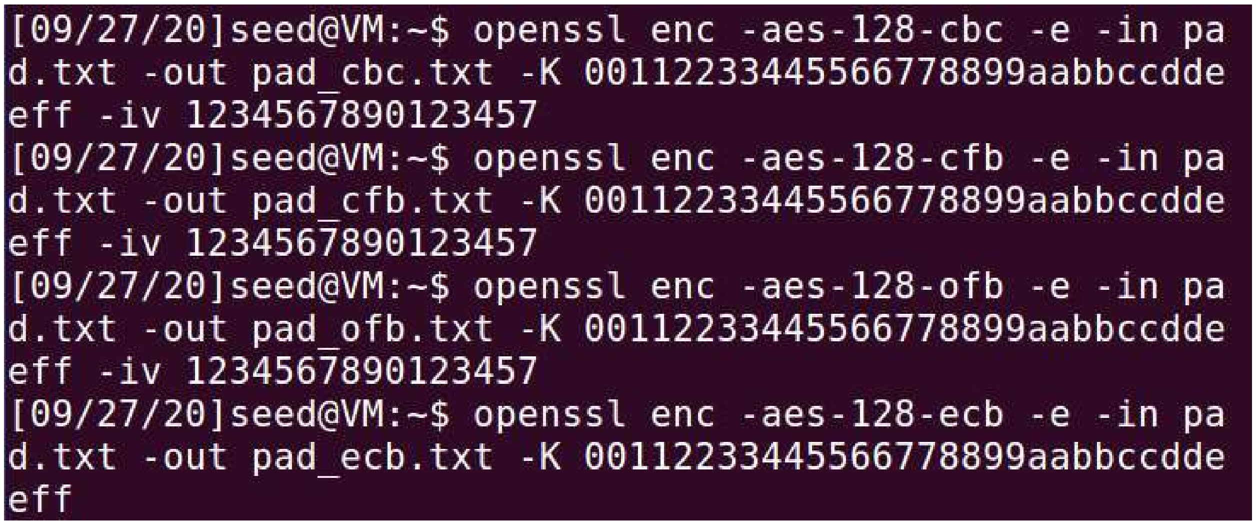

(3.91)



(3.92)

Task 4 Padding: Before encrypting the files to report for padding, a file was created. The file "**pad.txt**" was created and its contents are "*hello world today is a wonderful day with lots of sunshine*." The size of the file is 58 bytes.

1) For pad.txt, **128-bit AES with CBC, CFB, OFB, and ECB modes** were used. We can see the terminal commands in **(4.0)**. For example, on output files, if **ECB** mode

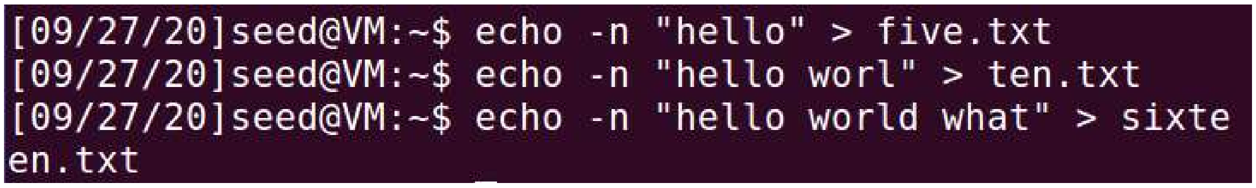


```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in pa
d.txt -out pad_cbc.txt -K 00112233445566778899aabbccdde
eff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-cfb -e -in pa
d.txt -out pad_cfb.txt -K 00112233445566778899aabbccdde
eff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-ofb -e -in pa
d.txt -out pad_ofb.txt -K 00112233445566778899aabbccdde
eff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-ecb -e -in pa
d.txt -out pad_ecb.txt -K 00112233445566778899aabbccdde
eff
```

(4.0)

was used, then the output file would be pad_ecb.txt. This goes for all the modes that were tested. After each output file was created, the sizes for the resulting documents for **pad_cbc.txt, pad_cfb.txt, pad_ofb.txt**, and **pad_ecb.txt** were 64 bytes, 58 bytes, 58 bytes, and 64 bytes respectively. Through the differences of the size for the original file (58 bytes), it is quite noticeable that the **CBC** and **ECB** modes do indeed use padding in contrast to the **CFB** and **OFB** encryption modes.

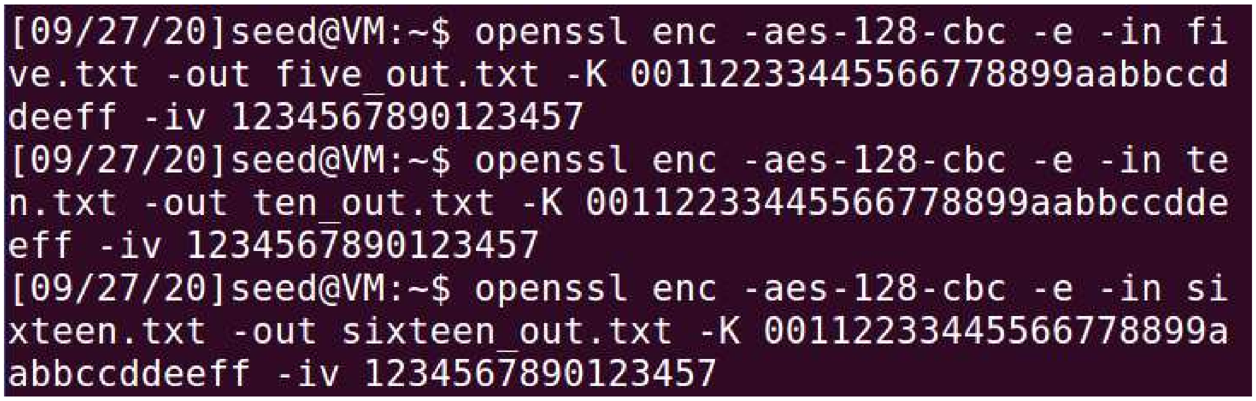
2) Three files were created. The files for the 5 bytes, 10 bytes, and 16 bytes are "**five.txt**, **ten.txt**, and **sixteen.txt**" respectively and were created through terminal command lines as shown in **(4.1).**



```
[09/27/20]seed@VM:~$ echo -n "hello" > five.txt
[09/27/20]seed@VM:~$ echo -n "hello worl" > ten.txt
[09/27/20]seed@VM:~$ echo -n "hello world what" > sixte
en.txt
```

(4.1)

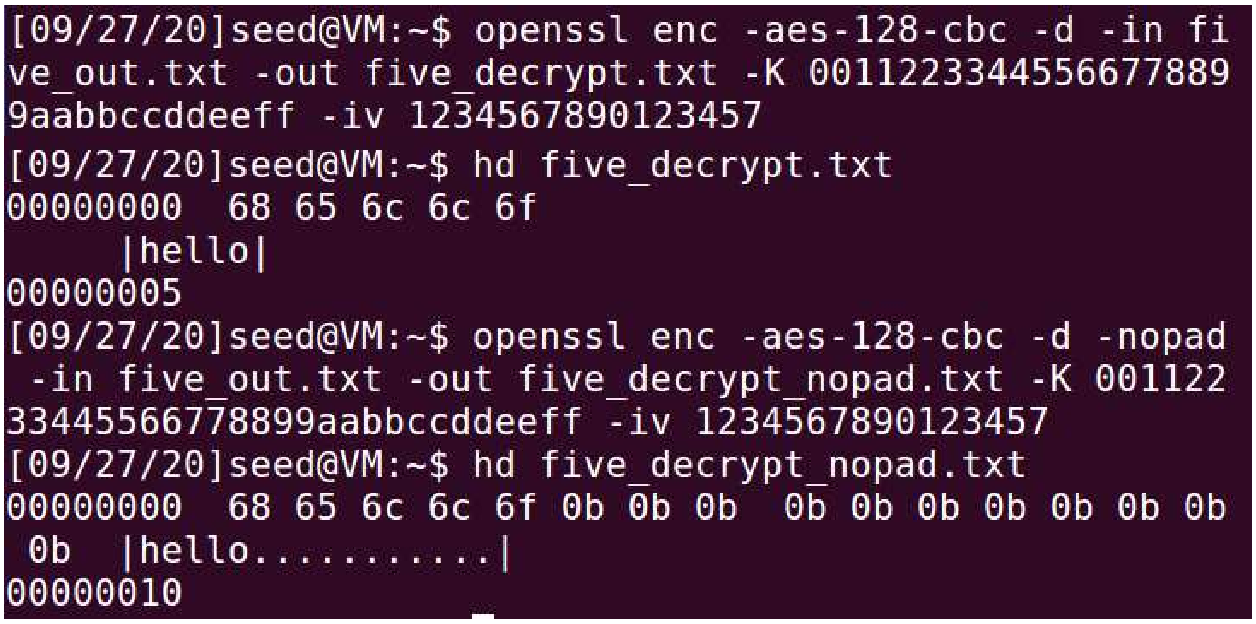
The three files were then encrypted using **128-bit AES with CBC mode** through the terminal command lines as shown in **(4.2).**



```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in fi
ve.txt -out five_out.txt -K 00112233445566778899aabbccd
deeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in te
n.txt -out ten_out.txt -K 00112233445566778899aabbccdde
eff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in si
xteen.txt -out sixteen_out.txt -K 00112233445566778899a
abbccddeeff -iv 1234567890123457
```

(4.2)

The sizes of the resulting encryption files for "**five_out.txt**, **ten_out.txt**, and **sixteen_out.txt**" are 16 bytes, 16 bytes, and 32 bytes respectively. This provides evidence for padding in the **CBC** mode. Now the files are to be decrypted while using the "*-nopad*" option, which will not remove padded data in the decrypted data. We can see how this is achieved through the terminal command lines as shown in **(4.3)**.

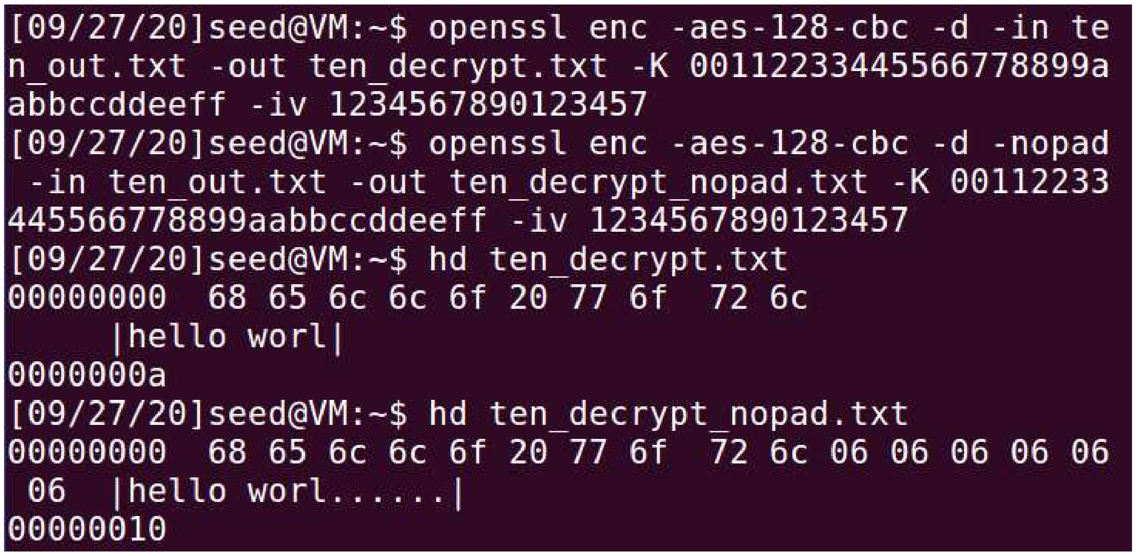


```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -d -in fi
ve_out.txt -out five_decrypt.txt -K 00112233445566778899aabbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ hd five_decrypt.txt
00000000  68 65 6c 6c 6f
     |hello|
00000005
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -d -nopad
 -in five_out.txt -out five_decrypt_nopad.txt -K 001122
33445566778899aabbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ hd five_decrypt_nopad.txt
00000000  68 65 6c 6c 6f 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b
 0b  |hello...........|
00000010
```

(4.3)

We can see the padding in the "**five_decrypt_nopad.txt**" when comparing it with "**five_decrypt.txt**." The difference in hex is the padding which is "*0b 0b …. 0b*" as shown in **(4.3)**. We can also see the padding when decrypting "**ten.txt**" shown in **(4.4).**

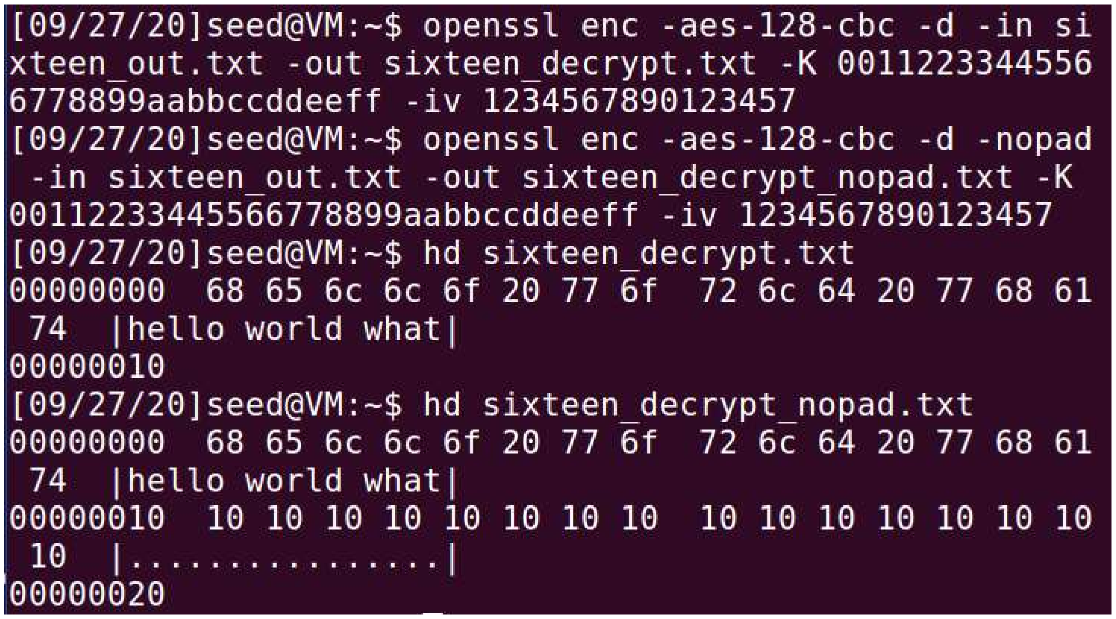
We can see the padding in the "**five_decrypt_nopad.txt**" when comparing it with "**five_decrypt.txt**." The difference in hex is the padding which is "*0b 0b .... 0b*" as shown in **(4.3)**. We can also see the padding when decrypting "**ten.txt**" shown in **(4.4).**



```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -d -in te
n_out.txt -out ten_decrypt.txt -K 00112233445566778899a
abbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -d -nopad
 -in ten_out.txt -out ten_decrypt_nopad.txt -K 00112233
445566778899aabbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ hd ten_decrypt.txt
00000000  68 65 6c 6c 6f 20 77 6f  72 6c
     |hello worl|
0000000a
[09/27/20]seed@VM:~$ hd ten_decrypt_nopad.txt
00000000  68 65 6c 6c 6f 20 77 6f  72 6c 06 06 06 06 06
 06  |hello worl......|
00000010
```

(4.4)

It is quite clear that for "**ten_decrypt_nopad.txt**" that the padding is hex "*06 06 ... 06*" as shown in **(4.4)**. For "**sixteen_out.txt**" we also see hex padding in "**sixteen_decrypt_nopad.txt**," as shown in **(4.5)** along with terminal command lines.



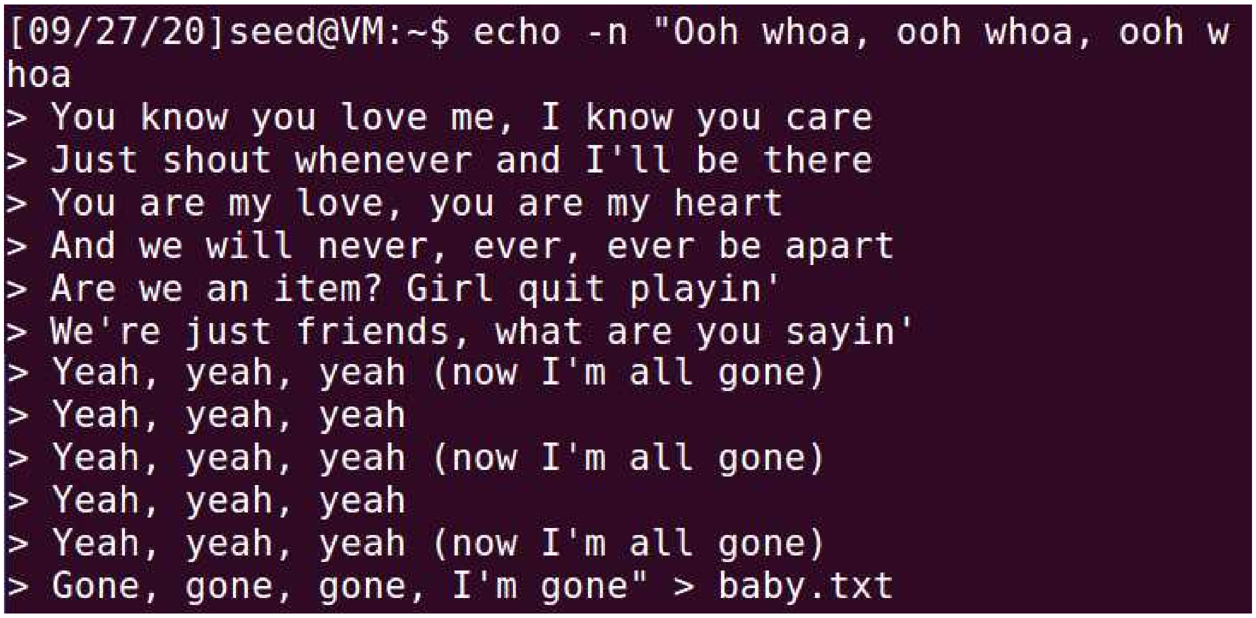
```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -d -in si
xteen_out.txt -out sixteen_decrypt.txt -K 0011223344556
6778899aabbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -d -nopad
 -in sixteen_out.txt -out sixteen_decrypt_nopad.txt -K
00112233445566778899aabbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ hd sixteen_decrypt.txt
00000000  68 65 6c 6c 6f 20 77 6f  72 6c 64 20 77 68 61
 74  |hello world what|
00000010
[09/27/20]seed@VM:~$ hd sixteen_decrypt_nopad.txt
00000000  68 65 6c 6c 6f 20 77 6f  72 6c 64 20 77 68 61
 74  |hello world what|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10
 10  |................|
00000020
```

(4.5)

We see that the hex padding is "*10 10 10 … 10*" when we compare "**sixteen_decrypt_nopad.txt**" with "**sixteen_decrypt.txt**."

Task 5 Error Propagation – Corrupted Cipher Text: A text file of at least 1000 bytes was created and it was named "**baby.txt**." The contents of this file are ironically enough the song lyrics to the hit song by Justin Bieber "*Baby*."

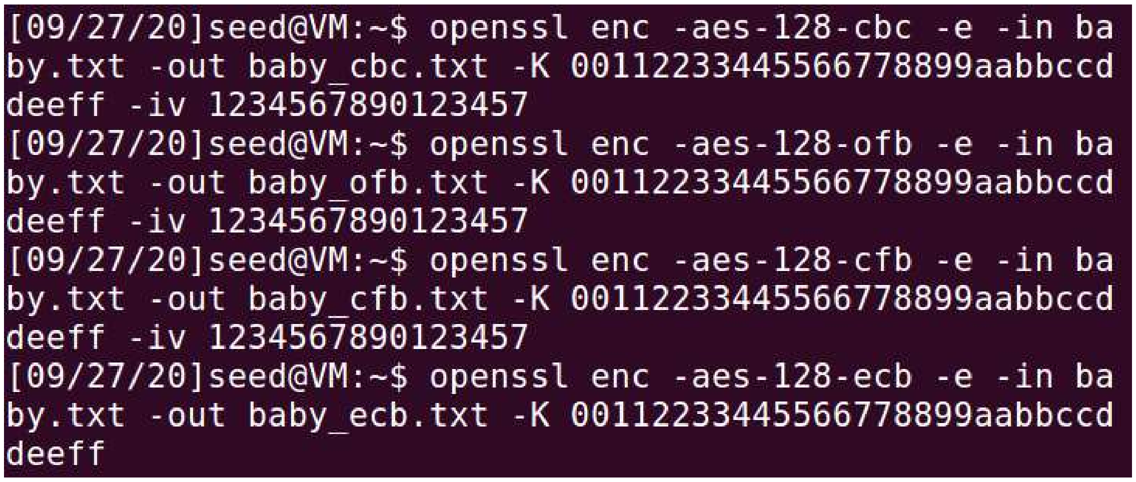
1. The text file creation in the terminal using command lines is shown in **(5.0).**



```
[09/27/20]seed@VM:~$ echo -n "Ooh whoa, ooh whoa, ooh w
hoa
> You know you love me, I know you care
> Just shout whenever and I'll be there
> You are my love, you are my heart
> And we will never, ever, ever be apart
> Are we an item? Girl quit playin'
> We're just friends, what are you sayin'
> Yeah, yeah, yeah (now I'm all gone)
> Yeah, yeah, yeah
> Yeah, yeah, yeah (now I'm all gone)
> Yeah, yeah, yeah
> Yeah, yeah, yeah (now I'm all gone)
> Gone, gone, gone, I'm gone" > baby.txt
```

(5.0)

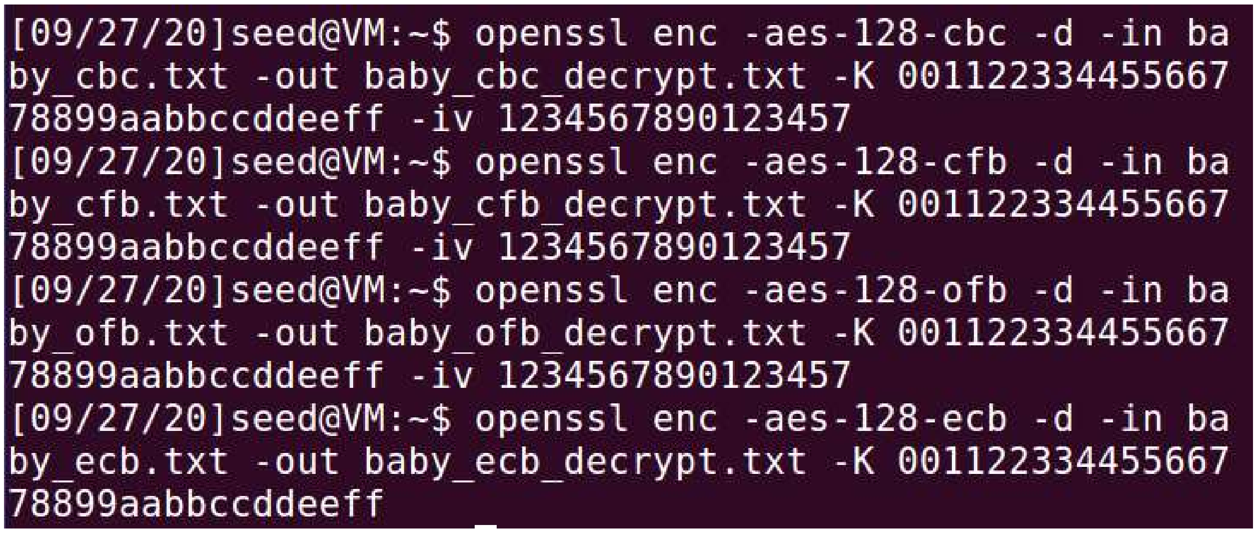
2. Using the commands shown in **(5.1)**, the file "**baby.txt**" was encrypted using **128-bit AES with ECB, CBC, CFB, and OFB modes**, and their respective files were created.



```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in ba
by.txt -out baby_cbc.txt -K 00112233445566778899aabbccd
deeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-ofb -e -in ba
by.txt -out baby_ofb.txt -K 00112233445566778899aabbccd
deeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-cfb -e -in ba
by.txt -out baby_cfb.txt -K 00112233445566778899aabbccd
deeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-ecb -e -in ba
by.txt -out baby_ecb.txt -K 00112233445566778899aabbccd
deeff
```

(5.1)

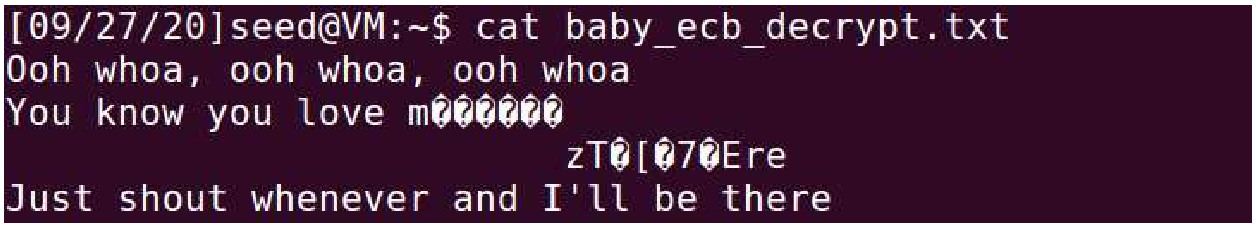
3. Use the bless hex editor to simulate the corruption ("flip") of a single bit of the 55th byte in each encrypted file. Flip the same bit in each file. Using bless hex editor to simulate corruption, a single bit of the 55th byte in each encrypted file was "flipped." For "**baby_cbc.txt**" the **8A** hex was changed to **8B**. For "**baby_cfb.txt**" the **37** hex was changed to **36** hex. In "**baby_ofb.txt**" the **1F** hex was changed to **1E** hex, and lastly, in "**baby_ecb.txt**" the **F8** hex was changed to **F9**.
4. Using the same key and IV when encrypting the files, the files were decrypted after they were "*corrupted*." The terminal commands can be seen in **(5.2).**



```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -d -in ba
by_cbc.txt -out baby_cbc_decrypt.txt -K 001122334455667
78899aabbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-cfb -d -in ba
by_cfb.txt -out baby_cfb_decrypt.txt -K 001122334455667
78899aabbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-ofb -d -in ba
by_ofb.txt -out baby_ofb_decrypt.txt -K 001122334455667
78899aabbccddeeff -iv 1234567890123457
[09/27/20]seed@VM:~$ openssl enc -aes-128-ecb -d -in ba
by_ecb.txt -out baby_ecb_decrypt.txt -K 001122334455667
78899aabbccddeeff
```

(5.2)

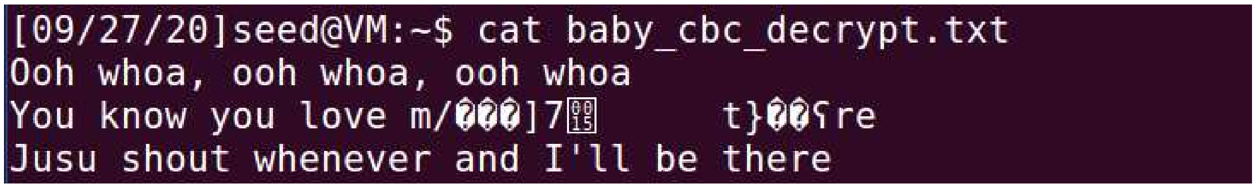
We now can take a look at how the corrupted files changed after they were decrypted as shown in **(5.3, 5.4, 5.5, and 5.6).**



```
[09/27/20]seed@VM:~$ cat baby_ecb_decrypt.txt
Ooh whoa, ooh whoa, ooh whoa
You know you love m������
                     zT�[�7�Ere
Just shout whenever and I'll be there
```

(5.3)

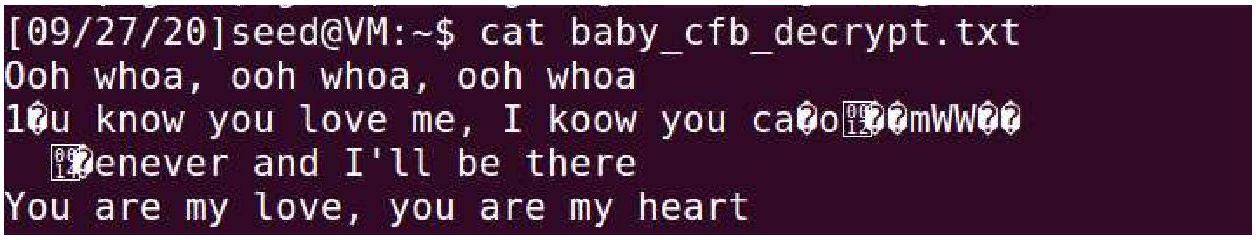
We can see here how the **ECB** mode specifically decrypted the corrupted document **(5.3)**. There is clearly a chunk of "**baby.txt**" that is still not decrypted, which can be explained by the corrupt "*flipping*." However, we know that the corruption affects the while block the corrupted bit is in, meaning that if the whole block is unable to be decrypted, then by the result we can see that the whole block will appear decrypted even if some parts of the block may not be "*corrupted*."



```
[09/27/20]seed@VM:~$ cat baby_cbc_decrypt.txt
Ooh whoa, ooh whoa, ooh whoa
You know you love m/���]7␕      t}��ʕre
Jusu shout whenever and I'll be there
```

(5.4)

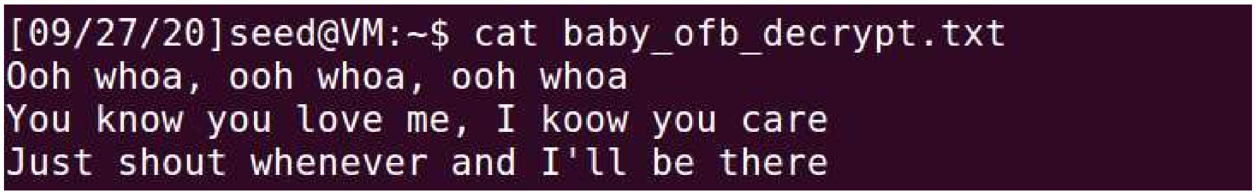
Here, we see how the **CBC** mode specifically decrypted the corrupted file **(5.4)**. There seems to be some plaintext missing, which is quite similar to how the ECB mode decryption turned out. **CBC** mode also involves block ciphering, and since the corrupted bit was in a block, it is quite like the **ECB** in that the block cipher is not wholly decrypted because of the "*flipped*" bit.



```
[09/27/20]seed@VM:~$ cat baby_cfb_decrypt.txt
Ooh whoa, ooh whoa, ooh whoa
1�u know you love me, I koow you ca�o␒��mWW��
  ␔�enever and I'll be there
You are my love, you are my heart
```

(5.5)

We can see how the **CFB** mode specifically decrypts the file with the "*flipped*" bit as shown above **(5.5).** However, unlike the **CBC** or even the **ECB** there are some words that are not correct and are simply out of place and missing. **CFB** mode however is still quite similar to **CBC**, but **CFB** mode is quite particular in that one corrupted bit could throw off the whole block decryption, which is evident when "*know*" is spelled incorrectly and parts of the verse "*Just shout whenever*" are still encrypted.



```
[09/27/20]seed@VM:~$ cat baby_ofb_decrypt.txt
Ooh whoa, ooh whoa, ooh whoa
You know you love me, I koow you care
Just shout whenever and I'll be there
```

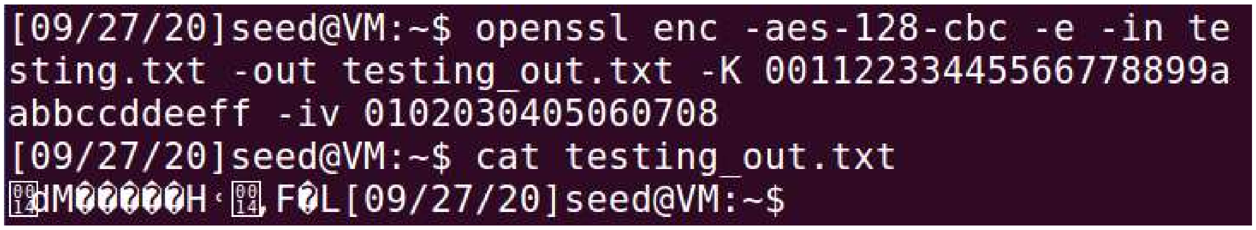
(5.6)

Based on the decryption using **OFB** mode, even with the single corrupted bit, we see that there are minimal errors here **(5.6)**. The only error lies in the 55th byte, which is where we "*flipped*" the bit. This show how **OFB** functions with errors, which is that it should be able to continue to correctly decrypt the rest of the block(s) even with the one bit "*flipped*" in the corrupted file.

Task 6 Initial Vector (IV) and Common Mistakes:

Task 6.1: A file "**testing.txt**" was created that contains one world "*Yes*."

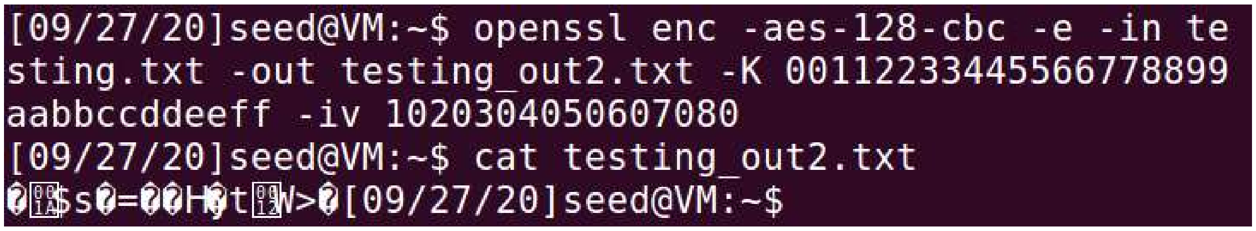
A. We can see below the terminal commands to encrypt "**testing.txt**" using the **iv: 0102030405060708**, which printed out the cipher text underneath the command **(6.0)**.



```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in te
sting.txt -out testing_out.txt -K 00112233445566778899a
abbccddeeff -iv 0102030405060708
[09/27/20]seed@VM:~$ cat testing_out.txt
dM�����H·,F�L[09/27/20]seed@VM:~$
```

(6.0)

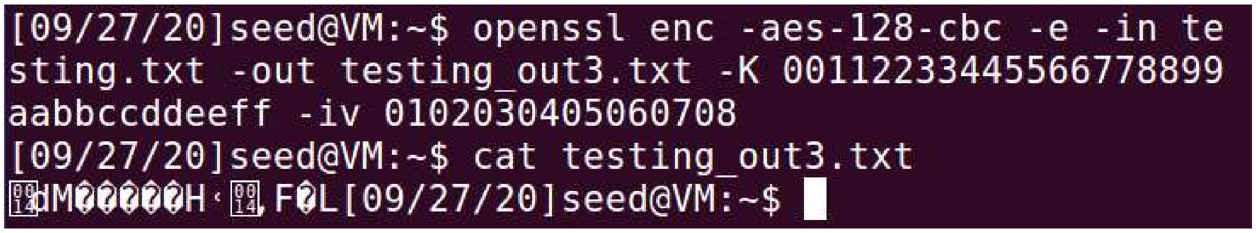
B. Below, we see the terminal commands to encrypt "**testing.txt**" using a different **iv: 1020304050607080**, which prints out a different cipher text seen in **(6.1)**.



```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in te
sting.txt -out testing_out2.txt -K 00112233445566778899
aabbccddeeff -iv 1020304050607080
[09/27/20]seed@VM:~$ cat testing_out2.txt
�$s�=��H�tW>�[09/27/20]seed@VM:~$
```

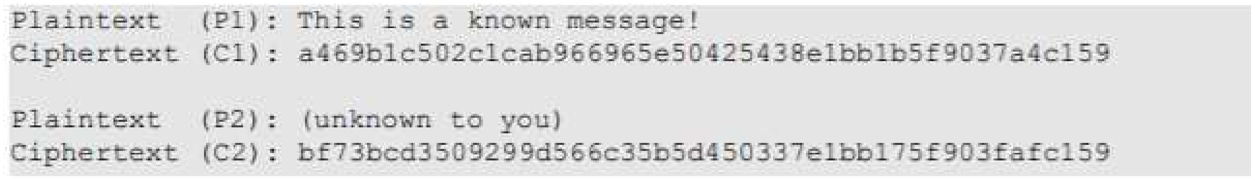
(6.1)

C. We can see below the terminal commands to encrypt "**testing.txt**" using the iv used in part A. Using **iv: 0102030405060708**, we see that the cipher text is identical to the cipher text from part A **(6.2)**.


```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -in te
sting.txt -out testing_out3.txt -K 00112233445566778899
aabbccddeeff -iv 0102030405060708
[09/27/20]seed@VM:~$ cat testing_out3.txt
dM������H‹ F�L[09/27/20]seed@VM:~$
```
(6.2)

Overall, based on parts A, B, and C, we can see that using the same IV or even a predictable IV can make it easier to figure out the cipher text and the message behind it. For example, if someone knows plaintext 1, and for some reason a plaintext 2 exists that the person does not know the message. It is quite easy to figure out the message if the same IV is used and the cipher texts are exactly the same. Therefore, not using a unique IV would make it much easier to possibly decrypt the cypher text as opposed to a unique IV, which we see in part B creates a whole different cypher text than the ones from part A and C.

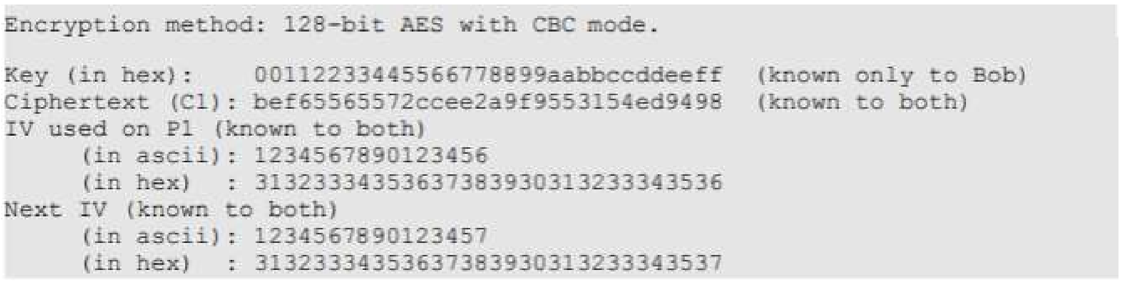
Task 6.2 Common Mistake: Use the Same IV: If an attacker gets hold of a plaintext (P1) and the corresponding ciphertext (C1), he/she should be able to decrypt other encrypted messages if the IV is always the same. For example: Given the following information **(6.3)**, we can figure out P2.


```
Plaintext  (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

Plaintext  (P2): (unknown to you)
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```
(6.3)

To get the IV, we can XOR the plaintext 1 and cipher text 1. (P1 XOR C1). → We get IV to be *f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a478* in hex. We then can use this IV and

XOR it with C2 to get the value of the message in P2. We then can get in ASCII for P2 the value to be *bf73bcd3509299d566c35b5d450337e1bb175f903fafc159*. If that is converted to text, we can get the message P2, which is quite alarming: "***<u>Order: Launch a missile!</u>***"

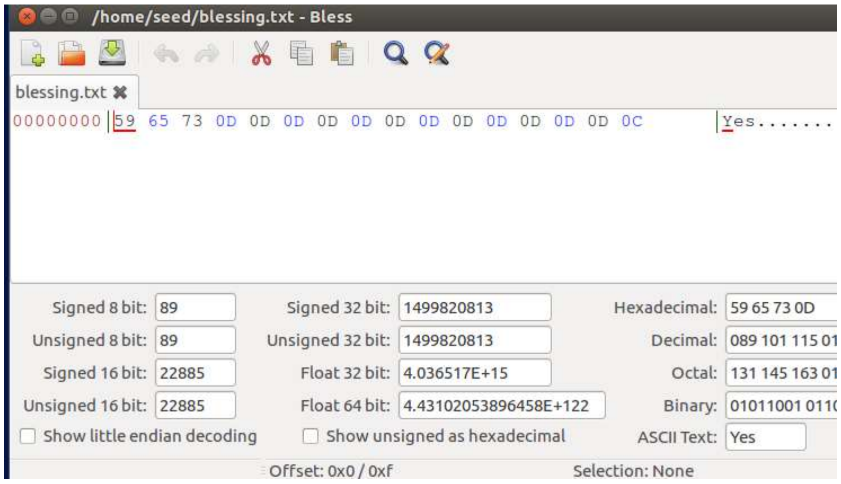
Task 6.3. Common Mistake: Use a Predictable IV: Given the information in **(6.4)**, we know **C1, IV1, and IV2**. After converting "**Yes**" from text to hex, we get the value to be "*5965730d0d0d0d0d0d0d0d0d0d0d0d0d*." It originally was "*5965730*;" however, in order to **XOR** the value with **IV1**, the extra padding needed to be added.



```
Encryption method: 128-bit AES with CBC mode.

Key (in hex):     00112233445566778899aabbccddeeff  (known only to Bob)
Ciphertext (C1): bef65565572ccee2a9f9553154ed9498  (known to both)
IV used on P1 (known to both)
     (in ascii): 1234567890123456
     (in hex)  : 31323334353637383930313233343536
Next IV (known to both)
     (in ascii): 1234567890123457
     (in hex)  : 31323334353637383930313233343537
```

(6.4)

After **"Yes" XOR IV1**, we get the hex "*68574039383b3a35343d3c3f3e39383b*," which is then used to **XOR with IV2**. When we finish that computation, we find out that the resulting value is very similar to "**Yes**" in that the hex is "*5965730d0d0d0d0d0d0d0d0d0d0d0d0c*" with minor padding differences. Using the bless editor in VM, we edited a blank text file and proceeded to put in the hex value of "*5965730d0d0d0d0d0d0d0d0d0d0d0d0c*" that we found earlier **(6.5).** We do this so we can send this message to Bob who will then encrypt it for us.

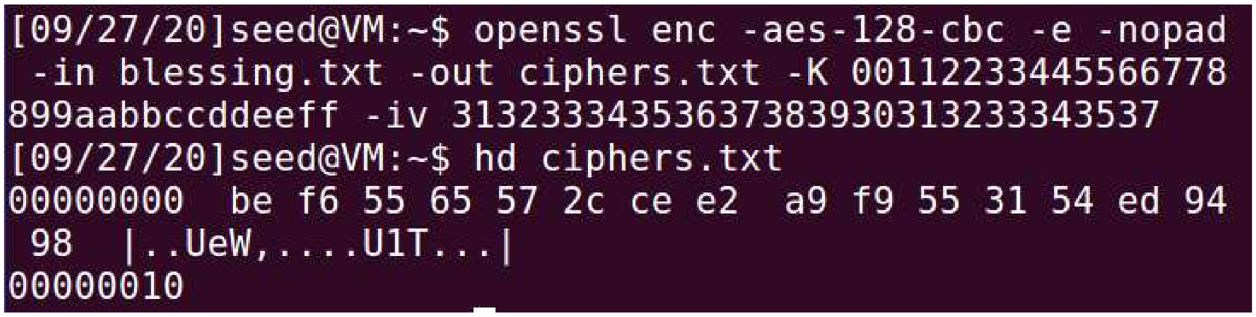


```
00000000 59 65 73 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0C   Yes.......
```

```
Signed 8 bit: 89              Signed 32 bit: 1499820813        Hexadecimal: 59 65 73 0D
Unsigned 8 bit: 89            Unsigned 32 bit: 1499820813      Decimal: 089 101 115 01
Signed 16 bit: 22885          Float 32 bit: 4.036517E+15       Octal: 131 145 163 01
Unsigned 16 bit: 22885        Float 64 bit: 4.43102053896458E+122   Binary: 01011001 011
☐ Show little endian decoding   ☐ Show unsigned as hexadecimal   ASCII Text: Yes
Offset: 0x0 / 0xf             Selection: None
```

(6.5)

We can see the terminal command line from Bob's encryption in **(6.6).** We also see that when we receive the ciphertext from Bob that it very much so resembles the ciphertext provided in C1, meaning that Bob's **P1** message was indeed "**Yes**."



```
[09/27/20]seed@VM:~$ openssl enc -aes-128-cbc -e -nopad
 -in blessing.txt -out ciphers.txt -K 00112233445566778
899aabbccddeeff -iv 31323334353637383930313233343537
[09/27/20]seed@VM:~$ hd ciphers.txt
00000000  be f6 55 65 57 2c ce e2  a9 f9 55 31 54 ed 94
 98  |..UeW,....U1T...|
00000010
```

(6.6)

Overall Thoughts: I was particularly interested and surprised by Task 6.2 and 6.3. Both tasks for me were quite aggravating, resulting in numerous trial and error attempts, but the overall aspect of being able to deduce and figure out messages given only so much information is something that I find hard to wrap my head around, and in the end those tasks were the most rewarding. However, the idea that I can find out information with only a certain amount of info is something that very much interesting and something I would never know or learn if not for these challenges. Even though these tasks are primarily exposition to the overall subject of Secret-Key Encryption, it makes me think of what I could actually do if I learned much more on the subject (not in a harmful or terrible way of course). It is quite amazing what can be done with only so much information and very specific information that makes me rethink all my saved passwords and secure messages/emails while also making me double check security measure and what I can do to not let some future issues arise.