Daniel Oliveros

A02093272

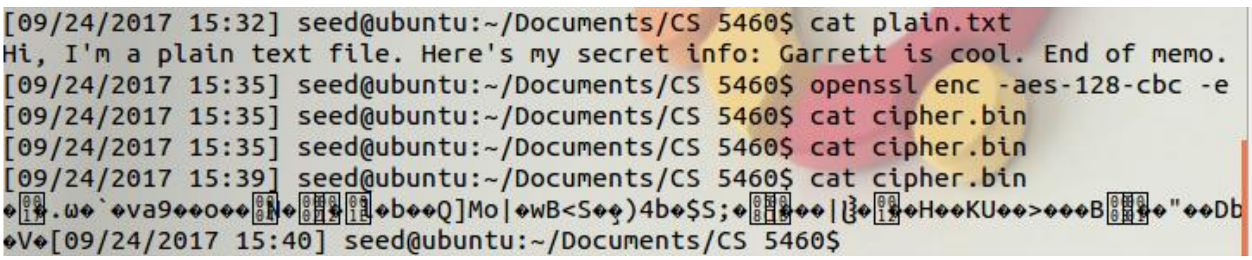
CS 5460

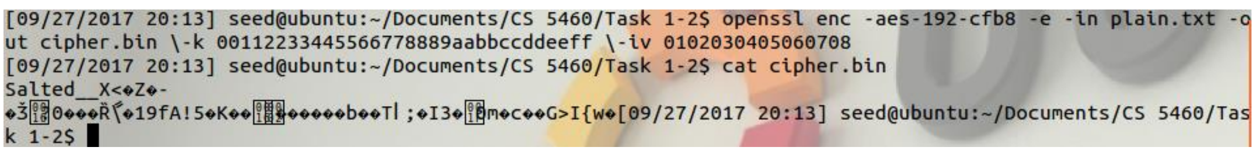
SEED Assignment

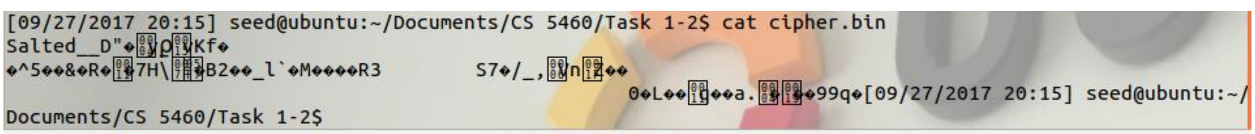
Originally, I considered using my raspberry pi for this assignment, but after looking more into it (and seeing how many programs I'd have to install), I decided to just use the virtual machine instead. So, I set that up and followed the other steps listed in the assignment description.

Task 1:

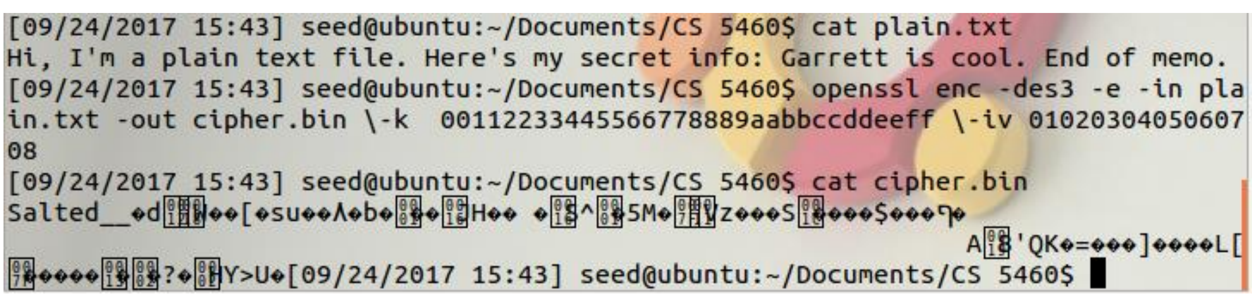
After figuring out some of the quirks of how to use the enc commands, I proceeded to use it to encrypt a file named "plain.txt" into a binary file named "binary.bin" using -aes-128-cbc, -aes-192-cfb8 and -aes-128-ctr encryption. The screenshots of the process are:

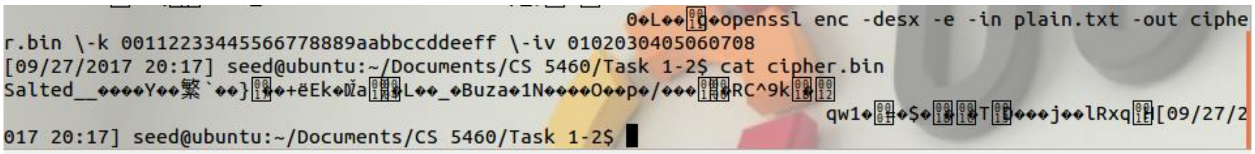


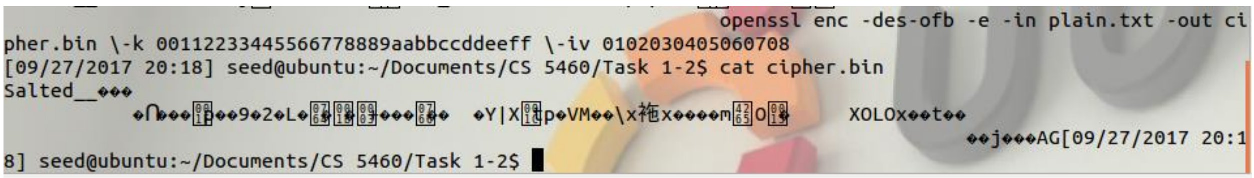


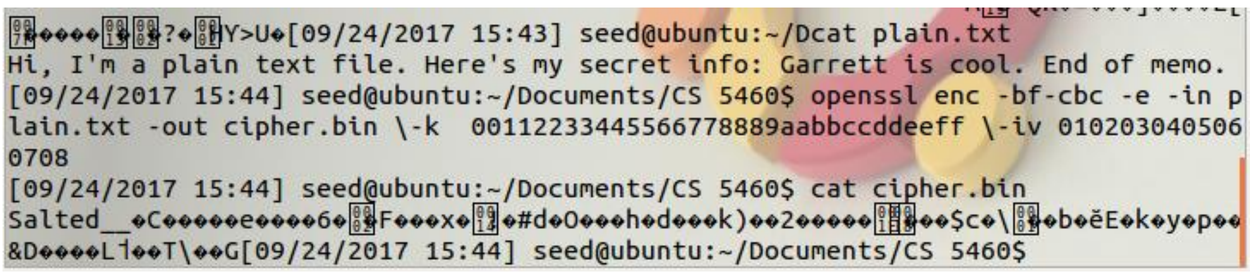


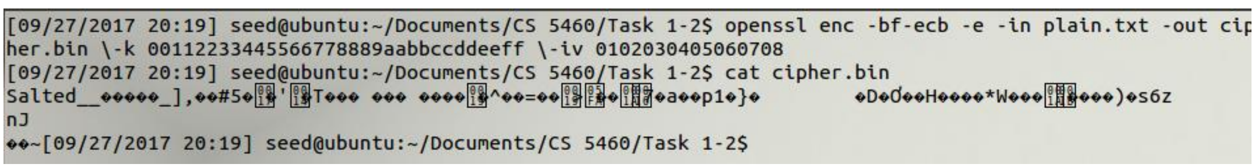
Afterwards, I did the exact same thing using the algorithms des3, -desx, -des-ofb, and bf-cbc, bf-ecb, and bf-ofb. Here are the screenshots.

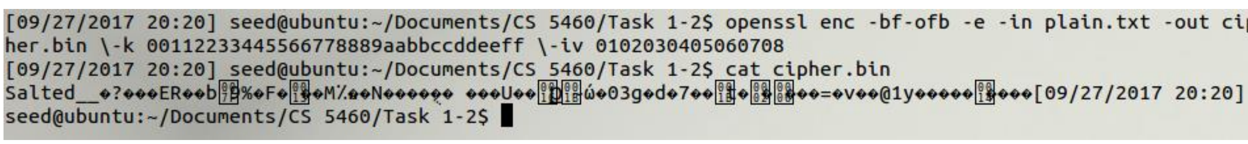










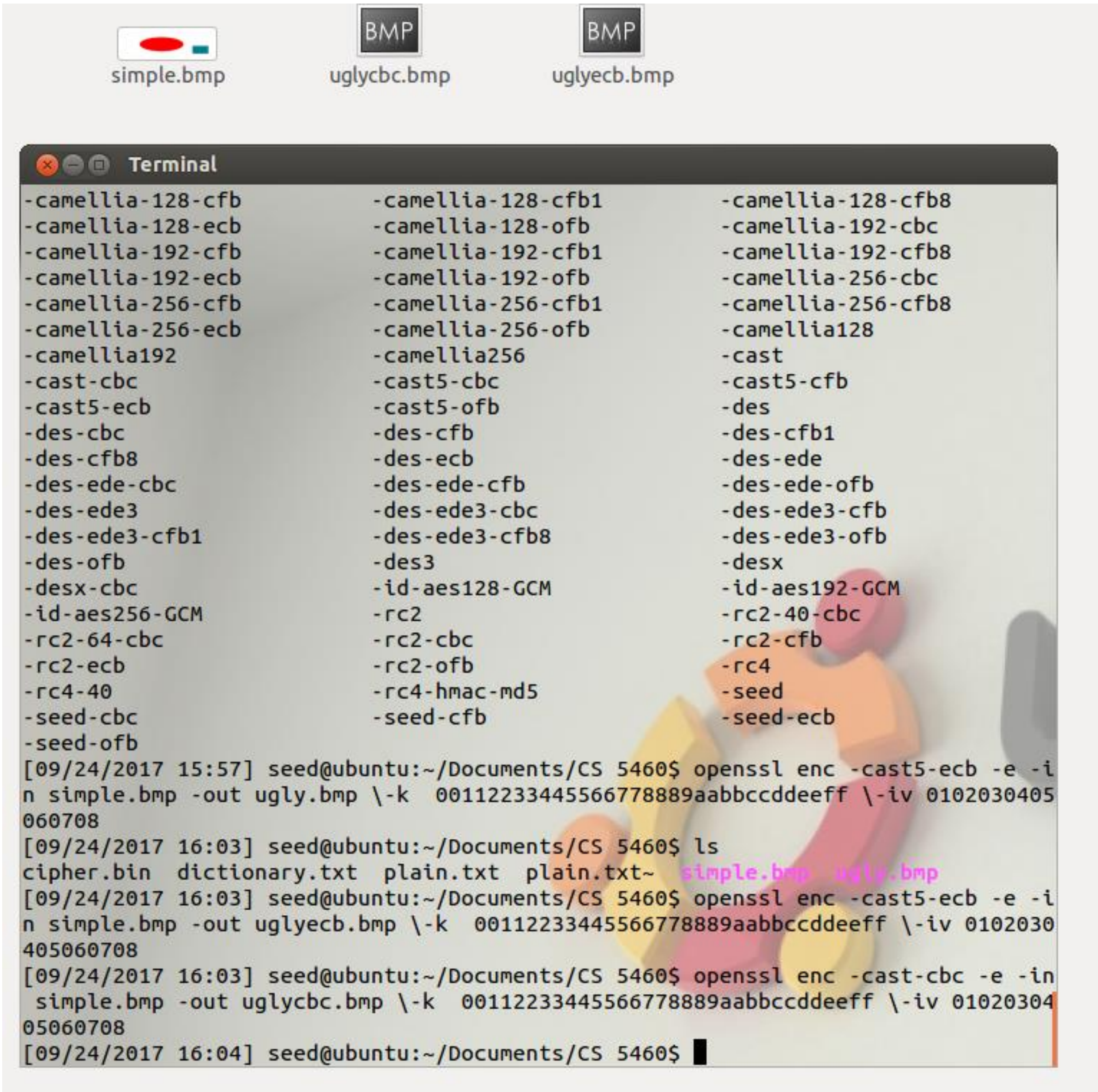


I found it much easier to use these commands after the first couple of times. Really made it clear how different the results from each algorithm really are.

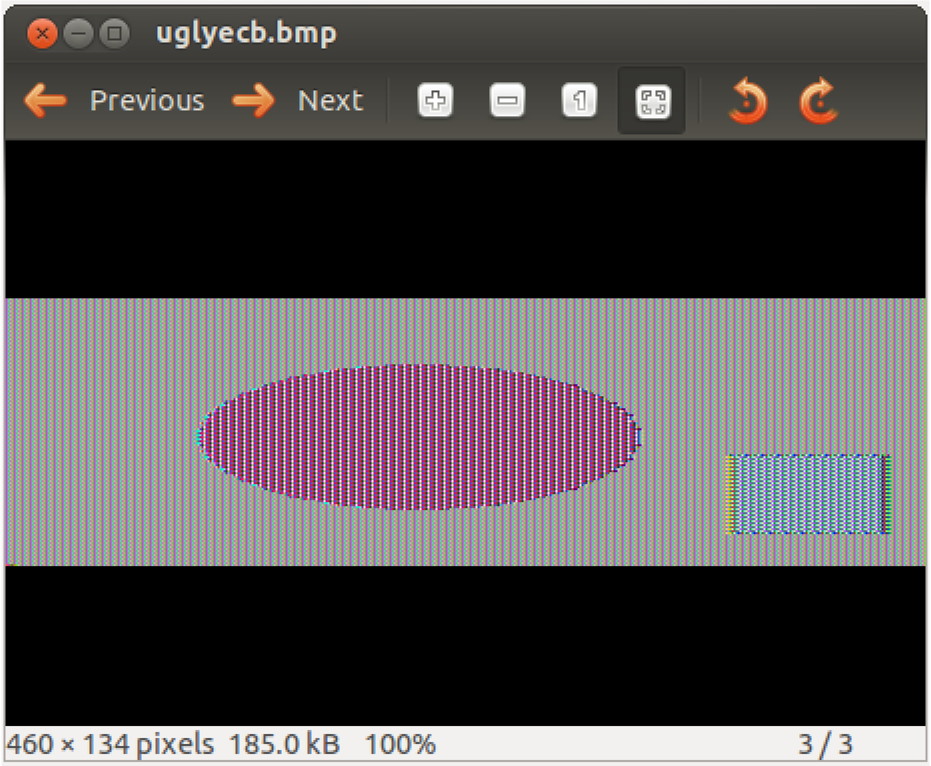
Task 2:

1. After googling what ECB and CBC were, I started working on the task afterward.

   Here is a screenshot of the first two steps. I encrypted the files using cast5-ecb for my ECB encryption, and used cast-cbc for my CBC encryption

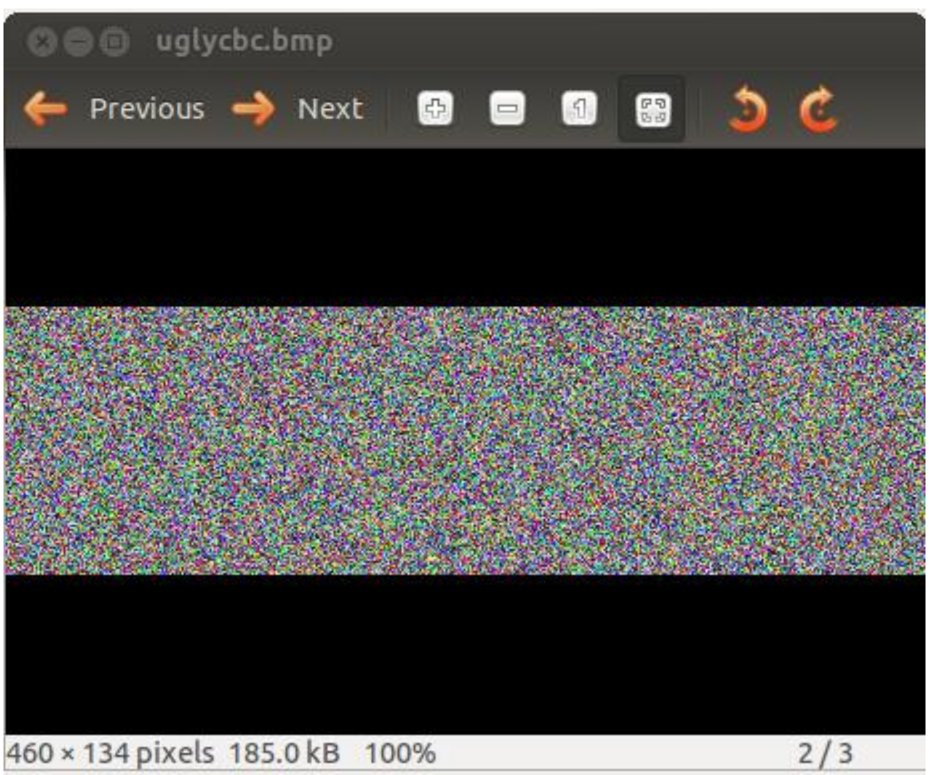


2. After making the changes to the encrypted files' headers, the one encrypted using ECB looked like this:



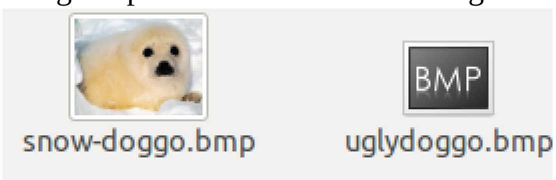
This image is strikingly similar to the original one, which allows me to easily have an idea of what the original may have looked like.

After performing the same changes to the cbc-encrypted file, this was the result:



This image is completely unreadable, I have no way of knowing what was originally in it from glancing at it. This seems much more useful for encrypting image files.

3. Finally, I found a BMP image online of a snow doggo. I proceeded to encrypt the image using the password cs5460. The images looked as follows:



From this exercise, I found how different algorithms vary wildly in their resulting encryptions, giving me a better idea of which type I should use in the future myself.

Task 3:

Here are my predictions on how the corrupted file decryption will go:

- ECB: It will only corrupt a small chunk of the text file; the rest should remain untouched.
- CBC: The whole file will be corrupted
- CFB: Only part of the file should be corrupted; the rest should be readable.
- OFB: I expect a great deal of the file to be corrupted

I encrypted the text file Facts.txt using CBC shown below:

```
better.txt ✖ | Facts.txt ✖
Look at my horse. Jk, that's a dead meme. It's all about that crippling depression nowadays I hear. Kids love that stuff.
```

In the resulting file, the memory in byte 30 was stored as:

5F

Which, of course, I changed to:

69

The file, better.txt, which resulted from decrypting this corrupted cypher looked like:

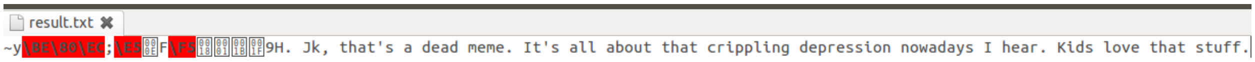
```
better.txt ✖ | Facts.txt ✖
²[009A][009C]B:ÊQFSÉPYÇÏ[000B]Í. Jk, that's W dead meme. It's all about that crippling depression nowadays I hear. Kids love that stuff.
```

In this file, only the starting bunch of data was lost due to the corrupted cypher. Everything else seemed mostly okay.

Now, I will do the same with ECB, CFB, and OFB.

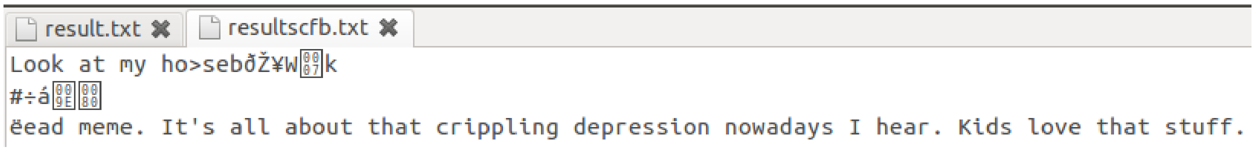
ECB:

After performing the same steps with ECB encryption, this was the resulting file:



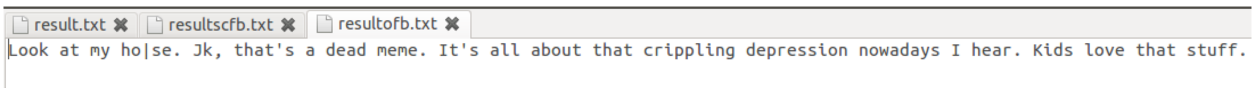
CFB:

This was the result from doing the same thing with CFB encryption:



OFB:

Doing the same thing with OFB gave this result:



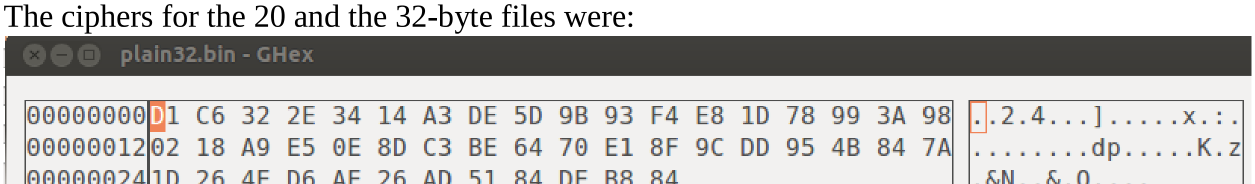
Analysis:

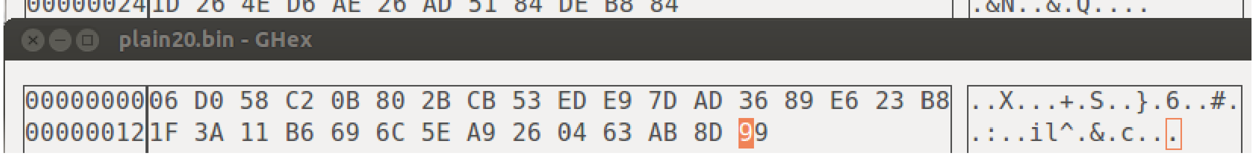
My assessments regarding CBC and OFB were incorrect. It seems OFB is extremely good at ensuring most of the text is correctly decrypted despite the cypher being corrupted. I assume this might be due to OFB reading in one byte at a time, though I could be wrong. This makes OFB a much better candidate for a situation where you'd worry about your files being corrupted.

Task 4:

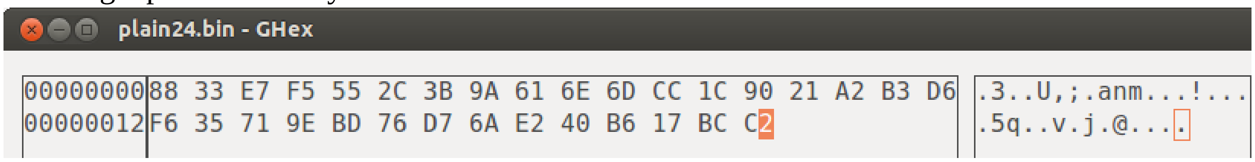
1. To test for this, I first went for a 20-octet long file. I read up on it, and it seems PKCS5 adds the number of octets needed to reach a multiple of 8 octets as the padding. So, I'm going to see if the length of the cipher is different for a 20-byte file and a 32-byte one. If that's the case, then I'll use a 24-byte file as well. In theory, if the length of the encrypted 24-byte file is the same as any of the other two, then the padding is not 8-bytes. Depending on how long the encrypted file is, we can determine exactly the length of the padding. My results went, as follows:

   The ciphers for the 20 and the 32-byte files were:



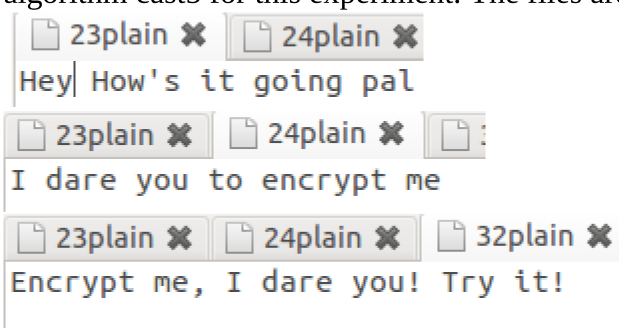


The 20-byte's resulting cipher was 32 bytes long. The one for the 32-byte one was 48 bytes long. This means that the padding is most likely 16, this is because, if it was 8, the cipher for our 20-byte file would be 24 bytes long, and the one for our 32-byte file would be 40 bytes long due to the way PCKS5 works. Just to be even more sure, let's look at the resulting cipher of a 24-byte file:



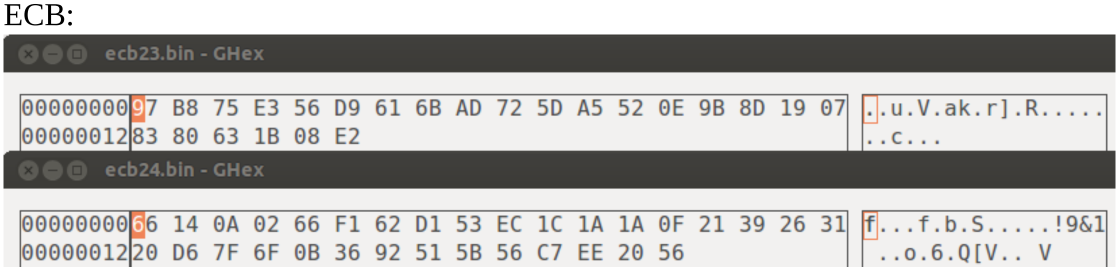
This cipher is just as long as the 20-byte file's, meaning that the 24 is definitely not divisible by our cipher. Which means that the padding must be 16-octets, which does not correspond with the way PCKS5 works.

2. For this, I will simply use a 23-byte long file, a 24-byte long file, and a 32-byte long one. I'll then simply measure the length of the resulting binary files to determine how long the padding is for each algorithm. In the cases where there is no difference between the 23 and the 24-byte files, I will use the 32-byte one to check. I used various versions of the algorithm cast5 for this experiment. The files are:



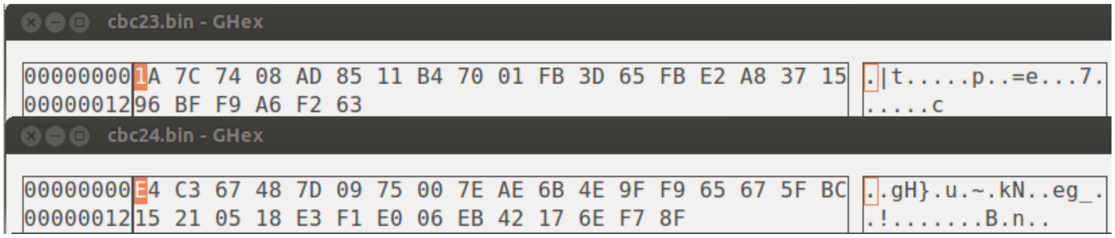
So, my results for the different encryption methods were as follows:

- ECB:



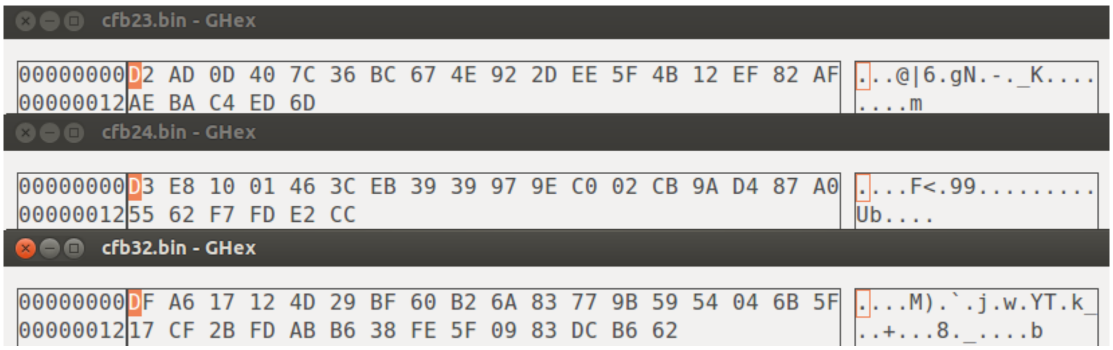
  The cipher for the 24-byte file was 8 bytes longer than the one for the 23-byte one. From this, I can assume that the padding for this method is 8-bytes.

- CBC:



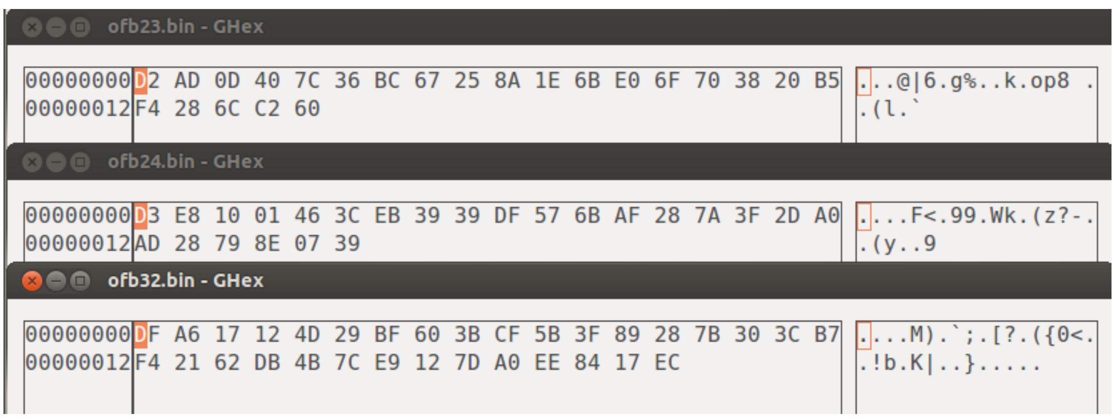
The cipher for the 24-byte file was 8 bytes longer than the one for the 23-byte one. From this, I can assume that the padding for this method is 8-bytes.

- CFB:



For this method, it seems there is no padding. The encrypted files were all just as long as the ones they were based off. I assume this method encrypts one byte at a time.

- OFB:



Just like CFB, this method seems to have no padding. I assume it encrypts one byte at a time

Task 5: This task took me more than all of the other ones combined. Saying that, it's unfortunately incomplete. I did not manage to get working code for this assignment. I used plenty of resources from a wiki page to even know where to begin.
https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption#Setting_it_up

I encountered issues from every aspect of this project. From setting up a working Makefile file; to getting used to coding on Linux; to formatting inputs and outputs. In the end, I have a general idea as to why my bugs are happening, but I have no drive left to solve them.

I think my code is breaking due to the way it's computing the decryption. I've determined plenty of the issues take place due to some things in it needing to be in hexadecimal. I'm extremely tired now however, and I fear I will break my code if I get close to it at this point.

Task 6:

A. After running the sequence once, this was my output:

```
[09/27/2017 19:50] seed@ubuntu:~/Documents/CS 5460$ cat /proc/sys/kernel/random/entropy_avail
1126
```

I ran this again after a few minutes of solving other problems, this was the output:

```
[09/27/2017 19:52] seed@ubuntu:~/Documents/CS 5460$ cat /proc/sys/kernel/random/entropy_avail
1244
```

B. This is a snippet of the script working:

```
[09/27/2017 19:50] seed@ubuntu:~/Documents/CS 5460$ head -c 16 /dev/random | hexdump
0000000 7d7a 7017 45e5 e8f5 94f0 279e c955 23b9
0000010
```

And then, I decided I didn't want it to work anymore:

```
[09/27/2017 19:54] seed@ubuntu:~/Documents/CS 5460$ head -c 1200 /dev/random | hexdump
^C
```

As you can tell, I had to interrupt it. It stopped working for about a minute so I deciced to give it something more doable:

```
[09/27/2017 19:55] seed@ubuntu:~/Documents/CS 5460$ head -c 64 /dev/random | hexdump
```

Since it'd ran out of entropy, however, I had to wait a bit for this to work. Which it did, eventually. I had to click my mouse around a whole bunch.

```
[09/27/2017 19:55] seed@ubuntu:~/Documents/CS 5460$ head -c 64 /dev/random | hexdump
0000000 a653 d482 4317 7b39 7afa cf1e cfdc 4aa5
0000010 4b9b eba2 6eba fe83 a569 5a60 dec6 d767
0000020 7522 d9c2 4a3b 770e f732 f94c b60c 2794
0000030 fe4c 3d91 3223 c66a 4e6c 51e4 9ad3 0e47
0000040
[09/27/2017 19:57] seed@ubuntu:~/Documents/CS 5460$
```

C. As the problem stated, I ran both commands, these were the results:

- /dev/urandom:
  I decided to push it a bit and used the command "head -c 160000 /dev/urandom | hexdump". It worked just fine and printed out tons of randomly generated numbers. Proof here:

```
Terminal
0026fa0 c366 0f5b ce95 a4d4 f48a 91c2 e640 0d09
0026fb0 6e02 c494 1e7a 4237 b472 6109 0d34 3dcd
0026fc0 1713 1151 fd4a 2410 7d2e 904c 92fc dbeb
0026fd0 5e1d bcf1 d603 26e2 f93f 42cb 6109 2b60
0026fe0 f4de 341b 2907 a60d adf0 e8d8 af93 9f0b
0026ff0 acb3 8bf9 20d8 54d1 d86f c624 c6ef d648
0027000 6d7d 7646 cef0 4239 6cbb a761 a39d 832d
0027010 fc52 5414 fe99 34b1 d80b 671f 4c03 9ef9
0027020 3822 83fb 307e 8d39 1a0f ddd7 e72a a775
0027030 23c7 1e34 6b9c 3210 8560 ac51 693d 8b12
0027040 c16d 9eb5 f00d 0acf 2f97 1d9f 11cb 1bfc
0027050 e306 aad3 70e0 c408 0d75 cf21 75b7 2eb9
0027060 30b9 f388 c1b4 c44c 92b3 e858 77d8 aa23
0027070 62d1 e359 f487 f78d d9df 784d 313f 90a0
0027080 24ed 90ee c37b 8c0a ea55 28be 7658 01e8
0027090 ed73 4774 4680 5936 e413 a8c1 7eff 8311
00270a0 ed54 1ed3 8577 5daa 328a 3bd8 33c8 21ef
00270b0 da1f 7300 8701 a723 c24b fd14 ee11 a26d
00270c0 856d f5ee 38d1 97f5 adf2 f53a dc67 3e38
00270d0 4ef8 b642 56cc 4714 edc5 8a54 c0b6 bbc1
00270e0 3b78 d994 122f 86dc e286 328d e5a2 a179
00270f0 d5b5 615d 122f 943c 3f1a 4c67 0776 f69a
0027100
[09/26/2017 16:50] seed@ubuntu:~$ head -c 160000 /dev/urandom | hexdump
```

- /dev/random:
  It seems /dev/random didn't take my request so well. It sat there for a long while (I went to buy something and it was still running when I came back after running this command):

```
00270e0 3b78 d994 122f 86dc e286 328d e5a2 a179
00270f0 d5b5 615d 122f 943c 3f1a 4c67 0776 f69a
0027100
[09/26/2017 16:50] seed@ubuntu:~$ head -c 1600 /dev/random | hexdump
```