**CS5460 Assignment #1 Crypto Lab – Secret Key Encryption**

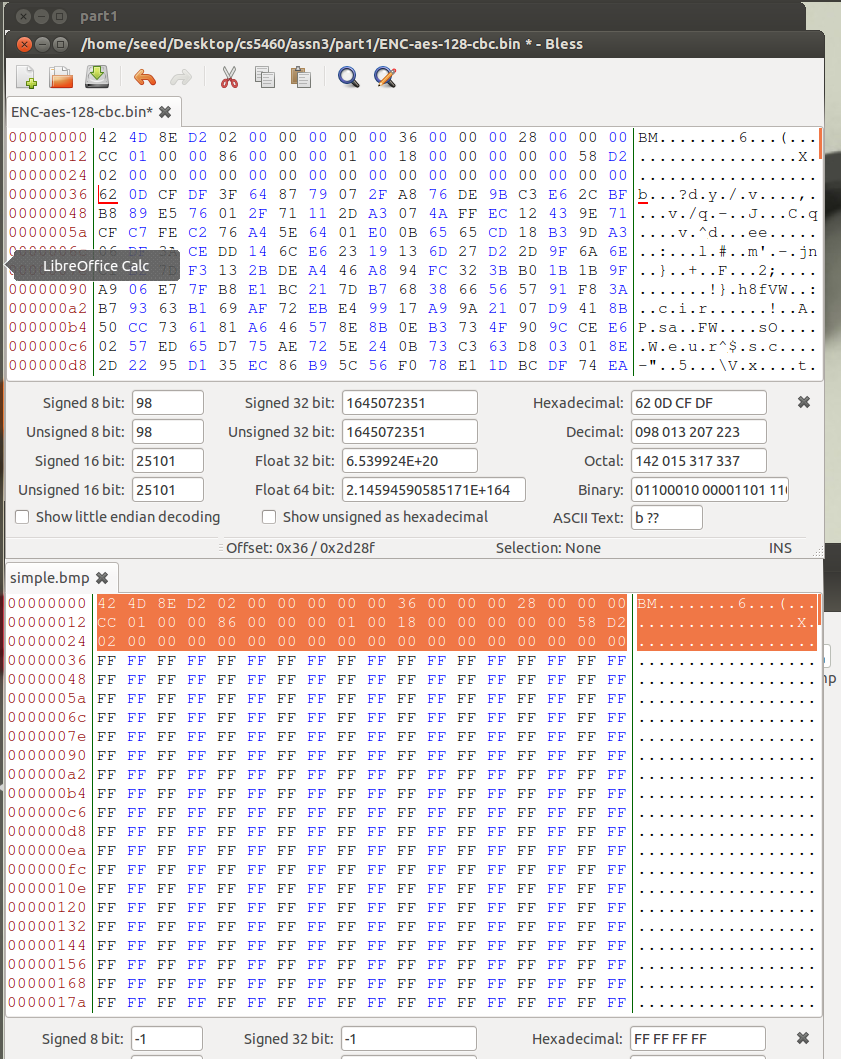
Austin Derbique A01967241 Fall 2017

https://github.com/aderbique/cs5460/tree/master/assn3

**3.1 Task 1: Encryption using different ciphers and modes**

During this task, I spent time getting familiarized with openssl and how to encrypt and decrypt files. I practiced on the README file that is found in the openssl directory on the seed image. These are a couple of the commands I used for this task. The files can be found in the Task 1 directory of the github repo.

seed@ubuntu:~/openssl-1.0.1$ sudo openssl enc -camellia-128-ecb -e -in README -out README\_rc2-128-cfb -K 00112233445566778889aabbccddeeff -iv 0102030405060708

seed@ubuntu:~/openssl-1.0.1$ sudo openssl enc -camellia-128-ecb -e -in README -out README\_camellia-128-ecb -K 00112233445566778889aabbccddeeff -iv 0102030405060708

**3.2 Task 2: Encryption Mode – ECB vs. CBC**

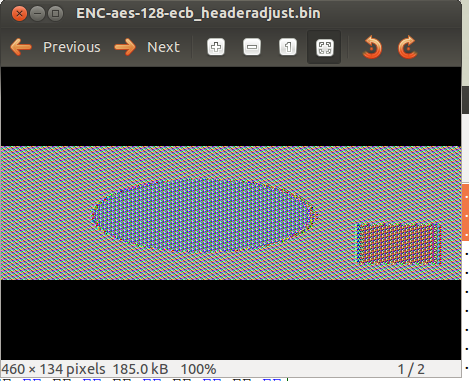
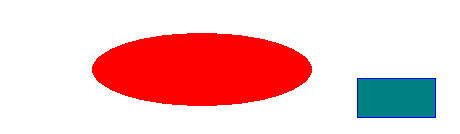
1. This task consisted of encrypting a given simple.bmp image file using ECB and CBC ciphers. To do this, the following commands are used:

seed@ubuntu:~/Desktop/cs5460/assn3/part1$ sudo openssl enc -aes-128-ecb -in simple.bmp -out ENC-aes-128-ecb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

seed@ubuntu:~/Desktop/cs5460/assn3/part1$ sudo openssl enc -aes-128-cbc -in simple.bmp -out ENC-aes-128-ccb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

The resulting files are encrypted binary files that must have the headers changed for them to be recognized as an image file. To do this, the first 54 bytes of the files are replaced with the first 54 bytes of the original image file using Bless. These files are available in the Task 2 section of the github repo.

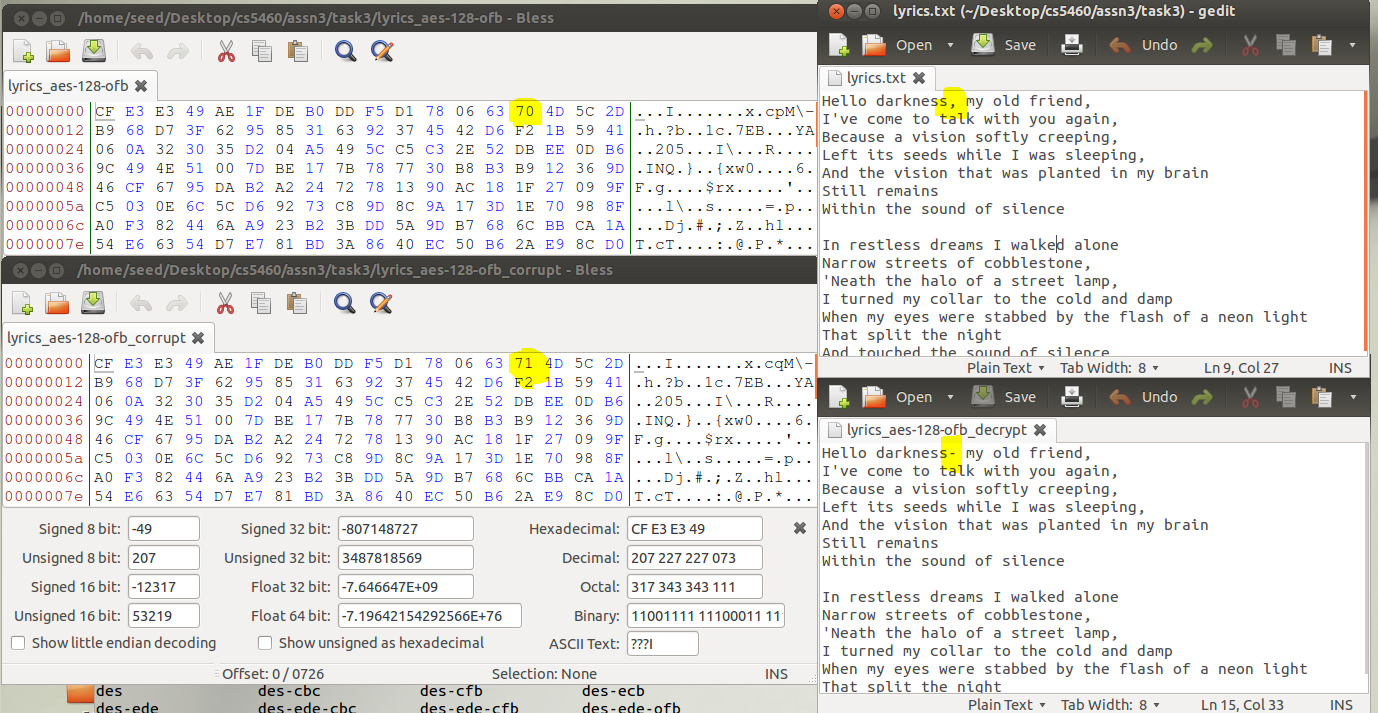
2. The CBC encryption completely scatters the image into a way that gives no information as to what the original image may have been. On the other hand, the ECB encryption shows a vague relation to the original image.



3. The task3.txt encrypted bmp image file is an encrypted version of the simple.bmp image using rc2-cbc cipher and a password of “cs5460”. This is available in the task 3 github repo.

**3.3 Task 3: Encryption Mode – Corrupted Cipher Text**

For this task, lyrics from a song were used and the input filename is lyrics.txt. The file was encrypted using AES-128-CBC cipher and saved as lyrics\_aes-128-cbc with the corrupted file as lyrics\_aes-128-cbc\_corrupt. The 30th bit the corrupted file had its value changed from 1 to 2. This corrupted file was then decrypted to a file named lyrics\_decrypt. A comparison of the before and after are shown below.



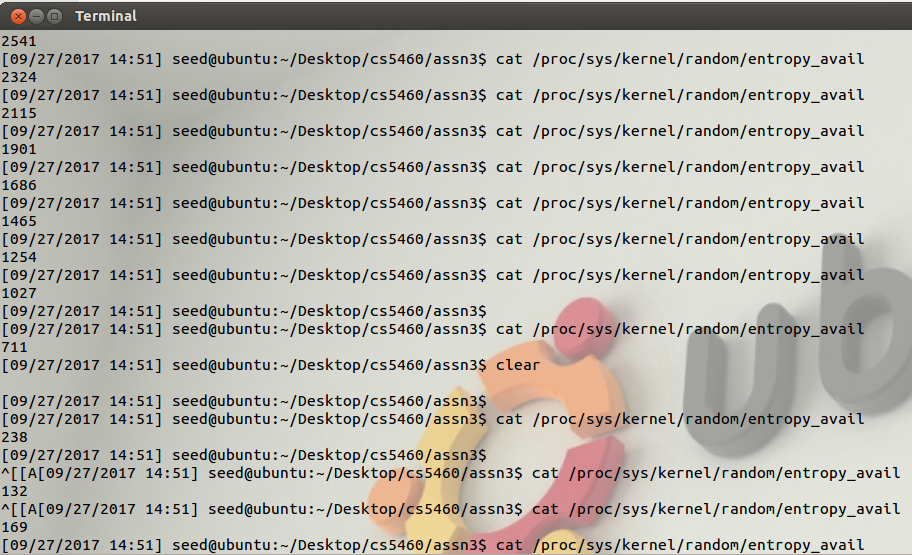
To answer the questions asked in the lab document, a majority of the information is recoverable. The only information that is changed is what came before the corrupted bit. I predict that this will change depending on the cipher used. For example, I predict that the CFB cipher will make this completely unrecoverable. To my surprise, the information in every document is mostly recoverable. There are slight differences between the files, but ultimately all are recoverable.

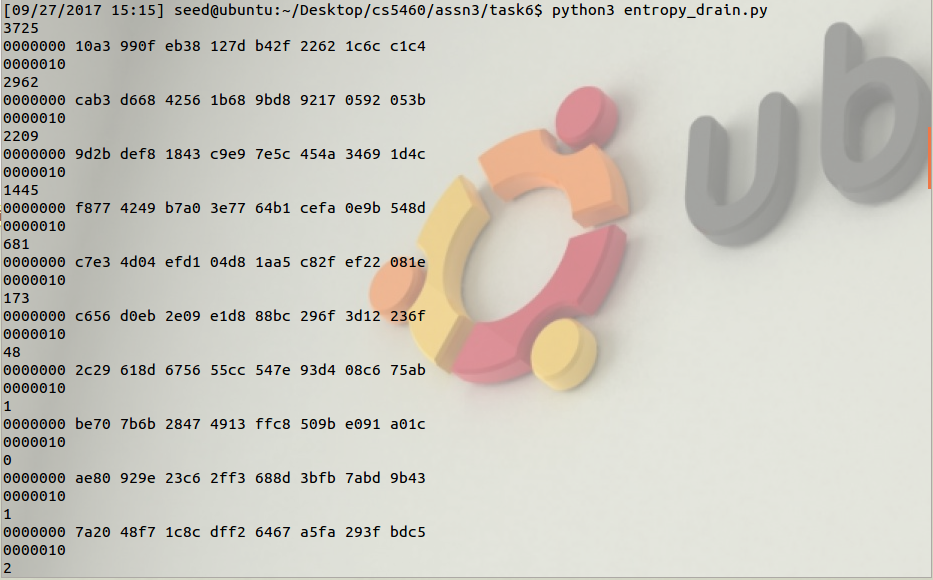
**3.4 Task 4 : Padding**

For this task, I created two text files, one containing 20 bytes and the other containing 32 bytes. The contents of these files were q’s. Next, the two files were encrypted using aes ciphers with ECB, CBC, CFB, and OFB modes. Using Bless, I attempted to see how much padding there was for each file. An assumption was made that the shorter files would have more padding than the file with 32 bytes. Unfortunately, I could not determine if the files contained padding or note, despite the fact that the length of the new file was now larger, meaning padding must have been added. I believe that all of modes I tried used padding.

**3.5 Task 5: Programming using the Crypto Library**

**3.6 Task 6: Pseudo Random Number Generation**

For Task 6.A, I moved the mouse and typed keys slowly and did not see a change in availability. This is when I realized that the value would only change when things happened fast. Decreasing the time in between calls led to a reduced availability. Below is a picture showing the time stamps and value recorded of entropy available.

For Task 6.B, I found the contrary to be true. Writing a script entropy\_drain.py, I attempt to get random numbers as fast as possible. It is evident that the entropy drops to single digit numbers, but it is rarely zero. It appears to hang until a random number is generated and eventually gives a random number. You can see that the first few calls occur very fast while entropy is high but the entropy quickly drains and that is when the calls start slowing down. The figure below shows entropy dropping quickly to a point where the pseudo random numbers can no longer collect random numbers fast enough.

For the last part of Task 6, a script urandom.py is available in the GitHub repo to call “head -c 1600 /dev/urandom | hexdump” 100 times as fast as possible. It is observable that there is no wait time for these random numbers which can be explained by the background information given in the assignment description.

**Conclusion**

The purpose of this lab is to get familiar with the concepts of secret key encryption. The six different tasks associated with this lab give a general understanding to the different ciphers and modes of encryption as well as how to generate randomness for good security. While completing this lab, I have learned more about openssl and encryption protocols as a whole.