Justin Gallagher

CSC 482

Secret-Key Encryption Lab

In this lab, we will experiment with encrypting and decrypting data with symmetric ciphers. We will examine how to apply encryption securely, focusing on issues where programmers frequently make mistakes, like the use of the wrong block cipher mode or predictable initialization vectors.

Task 1: Frequency Analysis

In this task, we are given a cipher-text that is encoded using letter substitution (that is, each letter has been matched to another letter in the alphabet and has replaced it). The job is to figure out what the original text says. To do this, I used the tr command from the command line along with a web application that can count the frequency of each individual word. I first took the file I downloaded from the Seed Labs website called 'ciphertext.txt' and ran it through the web application to find out the frequency of the individual letters. I then took the list of letters and organized it from most frequent to least frequent. I then used Wikipedia to find the frequency of letters within the English language. The two lists are given below:

```
ciphertext.txt Letter Frequency: nyvxuqmhtipaczlbgredfskjow
English Language Letter Frequency: etaoinshrdlcumwfgypbvkjxqz
```

I then used the following command that replaced each letter in the ciphertest.txt letter frequency list with the letter in the English Language letter frequency test that matches its place in the list (i.e. 'n' replaced with 'e', 'y' replaced with 't', ...) and then outputs it to a file called output1.txt. The command is listed below. I then used the less command to read the output1.txt file. Although it was still very unreadable, I was able to make out certain words along with letters that needed replacing. I used multiple tr commands in order to pick out certain letters and replace them with their appropriate letter. This was mostly a guessing game, but the initial command that used the English language frequency helped a lot in narrowing down the matches. The commands used are listed below along with the deciphered text after the multiple iterations of the process (saved as 'plaintext.txt').

```
tr 'nyvxuqmhtipaczlbgredfskjow' 'etaoinshrdlcumwfgypbvkjxqz' < ciphertext.txt
> output1.txt
less output1.txt
 tr 'rhi' 'HRN' < output1.txt > output2.txt
less output2.txt
tr 'sng' 'ISB' < output2.txt > output3.txt
less output3.txt
tr 'mteocau' 'UTEOCAM' < output3.txt > output4.txt
less output4.txt
tr 'lbwyf' 'DYWGF' < output4.txt > output5.txt
less output5.txt
tr 'dpkvqjxz' 'LPKVJXQZ' < output5.txt > output6.txt
less output6.txt
cp output6.txt plaintext.txt
```

THE OSCARS TURN ON SUNDAY WHICH SEEMS ABOUT RIGHT AFTER THIS LONG STRANGE AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO
THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND A NATIONAL CONVERSATION AS BRIEF AND MAD AS A FEVER DREAM ABOUT WHETHER THERE OUGHT TO BE A PRESIDENT WINFREY THE SEASON DIDNT JUST SEEM EXTRA LONG IT WAS EXTRA LONG BECAUSE THE OSCARS WERE MOVED TO THE FIRST WEEKEND IN MARCH TO AVOID CONFLICTING WITH THE CLOSING CEREMONY OF THE WINTER OLYMPICS THANKS PYEONGCHANG
ONE BIG QUESTION SURROUNDING THIS YEARS ACADEMY AWARDS IS HOW OR IF THE CEREMONY WILL ADDRESS METOO ESPECIALLY AFTER THE GOLDEN GLOBES WHICH BECAME A JUBILANT COMINGOUT PARTY FOR TIMES UP THE MOVEMENT SPEARHEADED BY POWERFUL HOLLYWOOD WOMEN WHO HELPED RAISE MILLIONS OF DOLLARS TO FIGHT SEXUAL HARASSMENT AROUND THE COUNTRY
SIGNALING THEIR SUPPORT GOLDEN GLOBES ATTENDEES SWATHED THEMSELVES IN BLACK SPORTED LAPEL PINS AND SOUNDED OFF ABOUT SEXIST POWER IMBALANCES FROM THE RED CARPET AND THE STAGE ON THE AIR E WAS CALLED OUT ABOUT PAY INEQUITY AFTER ITS FORMER ANCHOR CATT SADLER QUIT ONCE SHE LEARNED THAT SHE WAS MAKING FAR LESS THAN A MALE COHOST AND DURING THE CEREMONY NATALIE PORTMAN TOOK A BLUNT AND SATISFYING DIG AT THE ALLMALE ROSTER OF NOMINATED DIRECTORS HOW COULD THAT BE TOPPED
AS IT TURNS OUT AT LEAST IN TERMS OF THE OSCARS IT PROBABLY WONT BE
WOMEN INVOLVED IN TIMES UP SAID THAT ALTHOUGH THE GLOBES SIGNIFIED THE INITIATIVES LAUNCH THEY NEVER INTENDED IT TO BE JUST AN AWARDS SEASON CAMPAIGN OR ONE THAT BECAME ASSOCIATED ONLY WITH REDCARPET ACTIONS INSTEAD A SPOKESWOMAN SAID THE GROUP IS WORKING BEHIND CLOSED DOORS AND HAS SINCE AMASSED MILLION FOR ITS LEGAL DEFENSE FUND WHICH AFTER THE GLOBES WAS FLOODED WITH THOUSANDS OF DONATIONS OF OR LESS FROM PEOPLE IN SOME COUNTRIES
NO CALL TO WEAR BLACK GOWNS WENT OUT IN ADVANCE OF THE OSCARS THOUGH THE MOVEMENT WILL ALMOST CERTAINLY BE REFERENCED BEFORE AND DURING THE CEREMONY ESPECIALLY SINCE VOCAL METOO SUPPORTERS LIKE ASHLEY JUDD LAURA DERN AND NICOLE KIDMAN ARE SCHEDULED PRESENTERS
ANOTHER FEATURE OF THIS SEASON NO ONE REALLY KNOWS WHO IS GOING TO WIN BEST PICTURE ARGUABLY THIS HAPPENS A LOT OF THE TIME INARGUABLY THE NAILBITER NARRATIVE ONLY SERVES THE AWARDS HYPE MACHINE BUT OFTEN THE PEOPLE FORECASTING THE RACE SOCALLED OSCAROLOGISTS CAN MAKE ONLY EDUCATED GUESSES
THE WAY THE ACADEMY TABULATES THE BIG WINNER DOESNT HELP IN EVERY OTHER CATEGORY THE NOMINEE WITH THE MOST VOTES WINS BUT IN THE BEST PICTURE CATEGORY VOTERS ARE ASKED TO LIST THEIR TOP MOVIES IN PREFERENTIAL ORDER IF A MOVIE GETS MORE THAN PERCENT OF THE FIRSTPLACE VOTES IT WINS WHEN NO MOVIE MANAGES THAT THE ONE WITH THE FEWEST FIRSTPLACE VOTES IS ELIMINATED AND ITS VOTES ARE REDISTRIBUTED TO THE MOVIES THAT GARNERED THE ELIMINATED BALLOTS SECONDPLACE VOTES AND THIS CONTINUES UNTIL A WINNER EMERGES
IT IS ALL TERRIBLY CONFUSING BUT APPARENTLY THE CONSENSUS FAVORITE COMES OUT AHEAD IN THE END THIS MEANS THAT ENDOFSEASON AWARDS CHATTER INVARIABLY INVOLVES TORTURED SPECULATION ABOUT WHICH FILM WOULD MOST LIKELY BE VOTERS SECOND OR THIRD FAVORITE AND THEN EQUALLY TORTURED CONCLUSIONS ABOUT WHICH FILM MIGHT PREVAIL
IN IT WAS A TOSSUP BETWEEN BOYHOOD AND THE EVENTUAL WINNER BIRDMAN

IN WITH LOTS OF EXPERTS BETTING ON THE REVENANT OR THE BIG SHORT THE
PRIZE WENT TO SPOTLIGHT LAST YEAR NEARLY ALL THE FORECASTERS DECLARED LA
LA LAND THE PRESUMPTIVE WINNER AND FOR TWO AND A HALF MINUTES THEY WERE
CORRECT BEFORE AN ENVELOPE SNAFU WAS REVEALED AND THE RIGHTFUL WINNER
MOONLIGHT WAS CROWNED
THIS YEAR AWARDS WATCHERS ARE UNEQUALLY DIVIDED BETWEEN THREE BILLBOARDS
OUTSIDE EBBING MISSOURI THE FAVORITE AND THE SHAPE OF WATER WHICH IS
THE BAGGERS PREDICTION WITH A FEW FORECASTING A HAIL MARY WIN FOR GET OUT
BUT ALL OF THOSE FILMS HAVE HISTORICAL OSCARVOTING PATTERNS AGAINST THEM THE
SHAPE OF WATER HAS NOMINATIONS MORE THAN ANY OTHER FILM AND WAS ALSO
NAMED THE YEARS BEST BY THE PRODUCERS AND DIRECTORS GUILDS YET IT WAS NOT
NOMINATED FOR A SCREEN ACTORS GUILD AWARD FOR BEST ENSEMBLE AND NO FILM HAS
WON BEST PICTURE WITHOUT PREVIOUSLY LANDING AT LEAST THE ACTORS NOMINATION
SINCE BRAVEHEART IN THIS YEAR THE BEST ENSEMBLE SAG ENDED UP GOING TO
THREE BILLBOARDS WHICH IS SIGNIFICANT BECAUSE ACTORS MAKE UP THE ACADEMYS
LARGEST BRANCH THAT FILM WHILE DIVISIVE ALSO WON THE BEST DRAMA GOLDEN GLOBE
AND THE BAFTA BUT ITS FILMMAKER MARTIN MCDONAGH WAS NOT NOMINATED FOR BEST
DIRECTOR AND APART FROM ARGO MOVIES THAT LAND BEST PICTURE WITHOUT ALSO
EARNING BEST DIRECTOR NOMINATIONS ARE FEW AND FAR BETWEEN

Task 2: Encryption using Different Ciphers and Modes

In this task, we explore the various cipher algorithms and modes using the openssl enc command to encrypt and decrypt files. I used the following modes: Cipher Block Chaining, Cipher Feedback, Output Feedback. Cipher Block Chaining takes each block of plaintext and XOR's it with the previous cipher block. Cipher Feedback takes the cipher text from the previous block and feeds it into the cipher for encryption and then takes that output and XOR's it with the plaintext to generate the cipher text. Finally, Output Feedback is like the Cipher Feedback except before the XOR is performed the data is fed into the next block. To demonstrate these modes, I used the plaintext.txt file from the previous task and encrypted and decrypted it 3 times using the modes describes above. The commands are shown below that were used (diff command was used to make sure decrypted files retained all data):

```
openssl enc -aes-128-cbc -e -in plaintext.txt -out cbc_cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cbc -d -in cbc_cipher.bin -out cbc_plain.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708
diff plaintext.txt cbc_plain.txt
openssl enc -aes-128-cfb -e -in plaintext.txt -out cfb_cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cfb -d -in cfb_cipher.bin -out cfb_plain.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708
diff plaintext.txt cfb_plain.txt
openssl enc -aes-128-ofb -e -in plaintext.txt -out ofb_cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-ofb -d -in ofb_cipher.bin -out ofb_plain.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708
diff plaintext.txt ofb_plain.txt
```

Task 3: Encryption Mode – ECB vs. CBC

In this task, we are given a picture and we are supposed to encrypt it using the Electronic Code Book and Cipher Block Chaining modes. To do this, I used the following commands:

```
openssl enc -aes-128-cbc -e -in pic_original.bmp -out cbc_cipher_pic.bmp -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-ecb -e -in pic_original.bmp -out ecb_cipher_pic.bmp -K
00112233445566778889aabbccddeeff
head -c 54 pic_original.bmp > header
tail -c +55 cbc_cipher_pic.bmp > body_cbc
tail -c +55 ecb_cipher_pic.bmp > body_ecb
cat header body_cbc > cbc_cipher_pic.bmp
cat header body_ecb > ecb_cipher_pic.bmp
```

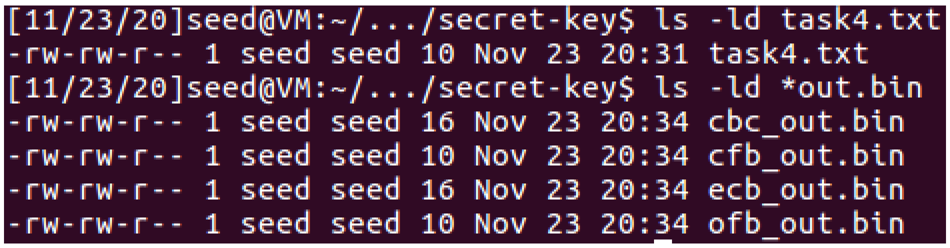
After creating the encrypted pictures (saved as 'cbc_cipher_pic.bmp' and 'ecb_cipher_pic.bmp') I looked at them to see if there were any differences. Using the ECB mode, we can still make out the shape of the objects on the page whereas the CBC mode completely scrambled everything so that no objects can be made out.

Task 4: Padding

In this task, we will explore the use of padding by block ciphers. The task is broken up into steps. The first step has us use the ECB, CBC, CFB, and OFB modes to encrypt a file. The file I used was called 'task4.txt' and it contains the string of characters "1234567890" which equates to the file being 10 bytes in total. After the encryption of this file using the mentioned modes, we can see which ones use padding and which ones don't. These are the commands I used to create the encrypted files:

```
openssl enc -aes-128-ecb -e -in task4.txt -out ecb_out.bin -K
00112233445566778889aabbccddeeff
openssl enc -aes-128-cbc -e -in task4.txt -out cbc_out.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cfb -e -in task4.txt -out cfb_out.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-ofb -e -in task4.txt -out ofb_out.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

I then used the ls command to show the files and their sizes. We can see below that the CBC and ECB add padding to the files because the encrypted files associated with them are 16 bytes instead of 10 like the task4.txt file. The CFB and OFB do not use padding because they take outputs from the previous block which has the be the same size as the cipher block.



```
[11/23/20]seed@VM:~/.../secret-key$ ls -ld task4.txt
-rw-rw-r-- 1 seed seed 10 Nov 23 20:31 task4.txt
[11/23/20]seed@VM:~/.../secret-key$ ls -ld *out.bin
-rw-rw-r-- 1 seed seed 16 Nov 23 20:34 cbc_out.bin
-rw-rw-r-- 1 seed seed 10 Nov 23 20:34 cfb_out.bin
-rw-rw-r-- 1 seed seed 16 Nov 23 20:34 ecb_out.bin
-rw-rw-r-- 1 seed seed 10 Nov 23 20:34 ofb_out.bin
```

The second step of this task has us create 3 files of 5 bytes, 10 bytes, and 16 bytes called f1.txt, f2.txt, and f3.txt, respectively. It then has us encrypt them using the CBC mode. I used these commands:

```
echo -n "12345" > f1.txt
echo -n "1234512345" > f2.txt
echo -n "1234512345123456" > f3.txt
openssl enc -aes-128-cbc -e -in f1.txt -out f1_out.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cbc -e -in f2.txt -out f2_out.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cbc -e -in f3.txt -out f3_out.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Once that is done, I compare the padding results. The first two files, f1 and f2 were padded to 16 bytes while the f3 file was padded to 32 bytes. We are then asked to decrypt each file using the –nopad option so that we may see the padding. The commands below are the ones I used to decrypt the files and then see the padding within:

```
xxd -g 1 f1.txt
openssl enc -aes-128-cbc -d -in f1_out.bin -out f1_plain.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad
xxd -g 1 f1_plain.txt
xxd -g 1 f2.txt
openssl enc -aes-128-cbc -d -in f2_out.bin -out f2_plain.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad
 xxd -g 1 f2_plain.txt
xxd -g 1 f3.txt
openssl enc -aes-128-cbc -d -in f3_out.bin -out f3_plain.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad
 xxd -g 1 f3_plain.txt
```

And here is the output of the 3 files with the padding:

```
[11/23/20]seed@VM:~/.../secret-key$ xxd -g 1 f2.txt00000000: 31 32 33 34 35
31 32 33 34 35                    1234512345
[11/23/20]seed@VM:~/.../secret-key$ xxd -g 1 f1_plain.txt00000000: 31 32 33
34 35 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b  12345...........
[11/23/20]seed@VM:~/.../secret-key$ xxd -g 1 f2_plain.txt
00000000: 31 32 33 34 35 31 32 33 34 35 06 06 06 06 06 06  1234512345......
[11/23/20]seed@VM:~/.../secret-key$ xxd -g 1 f3_plain.txt
```

```
00000000: 31 32 33 34 35 31 32 33 34 35 31 32 33 34 35 36  1234512345123456
00000010: 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10  ................
```

Task 5: Error Propagation – Corrupted Cipher Text

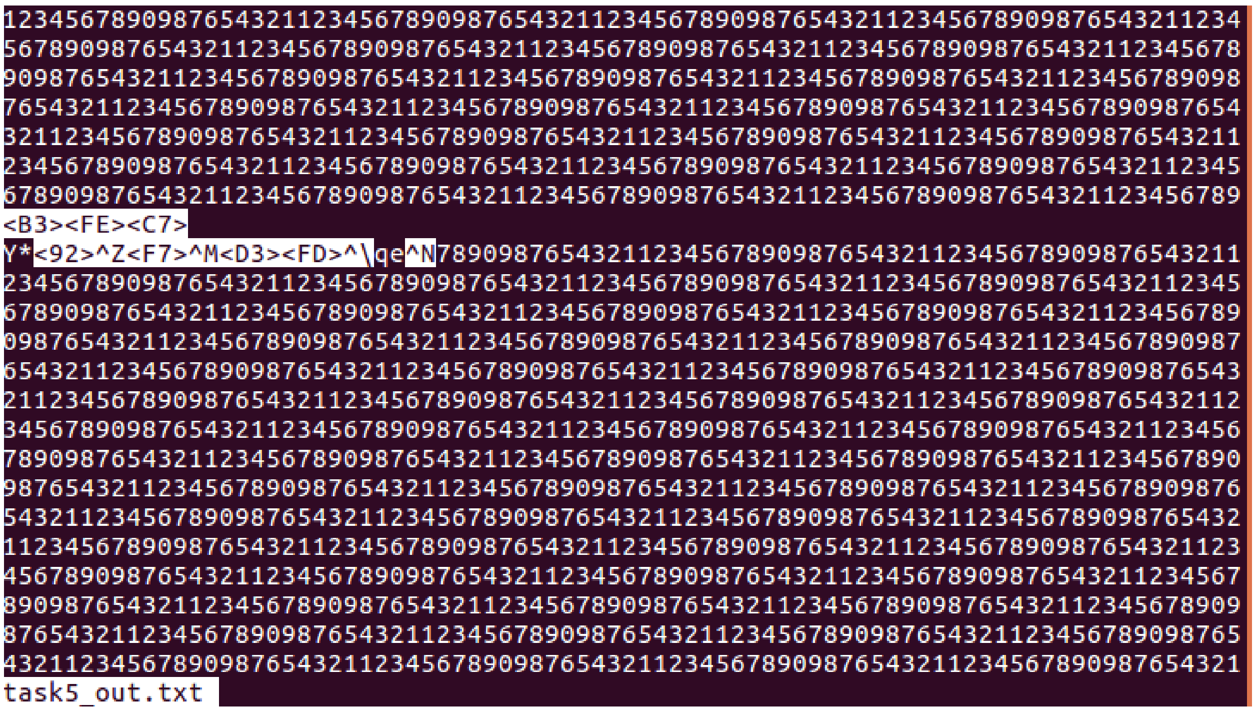
In this task, we will learn about the error propagation property of various encryption modes. This task is split into steps. The first step has us create a text file that is at least 1000 bytes in length. The second step has us encrypt using the AES-128 cipher. I used python to create the file and save it as 'task5.txt' using this command along with encrypting it:

```
python -c "print '1234567890987654321'*100" > task5.txt
openssl enc -aes-128-ecb -e -in task5.txt -out task5_cipher.bin -K
00112233445566778889aabbccddeeff
```

After the file had been encrypted, I used the bless editor to search for the 55th byte in the encrypted file. I did this by using the find tool inside the editor to search for the hexadecimal value of 0x37 which gave the position of the 55th byte. I changed it to '00' and then saved the changes. I used the following commands to decrypt the file and then view it:

```
openssl enc -aes-128-ecb -d -in task5_cipher.bin -out task5_out.txt -K
00112233445566778889aabbccddeeff
less task5_out.txt
```

Looking at the output and comparing it with the original, we see that the file has now been corrupted and the text is messed up. A picture is shown below:



```
12345678909876543211234567890987654321123456789098765432112345678909876543211234
56789098765432112345678909876543211234567890987654321123456789098765432112345678
90987654321123456789098765432112345678909876543211234567890987654321123456789098
76543211234567890987654321123456789098765432112345678909876543211234567890987654
32112345678909876543211234567890987654321123456789098765432112345678909876543211
23456789098765432112345678909876543211234567890987654321123456789098765432112345
67890987654321123456789098765432112345678909876543211234567890987654321123456789
<B3><FE><C7>
Y*<92>^Z<F7>^M<D3><FD>^\qe^N7890987654321123456789098765432112345678909876543211
23456789098765432112345678909876543211234567890987654321123456789098765432112345
67890987654321123456789098765432112345678909876543211234567890987654321123456789
09876543211234567890987654321123456789098765432112345678909876543211234567890987
65432112345678909876543211234567890987654321123456789098765432112345678909876543
21123456789098765432112345678909876543211234567890987654321123456789098765432112
34567890987654321123456789098765432112345678909876543211234567890987654321123456
78909876543211234567890987654321123456789098765432112345678909876543211234567890
98765432112345678909876543211234567890987654321123456789098765432112345678909876
54321123456789098765432112345678909876543211234567890987654321123456789098765432
11234567890987654321123456789098765432112345678909876543211234567890987654321123
45678909876543211234567890987654321123456789098765432112345678909876543211234567
89098765432112345678909876543211234567890987654321123456789098765432112345678909
87654321123456789098765432112345678909876543211234567890987654321123456789098765
43211234567890987654321123456789098765432112345678909876543211234567890987654321
task5_out.txt
```

Task 6: Initial Vector (IV) and Common Mistakes

In this task, we explore the initial vector and the role it plays in the security of the data being encrypted. This task is also broken up into parts. The first part has us encrypting a plaintext file using 2 different IV's and then the same IV. To do this, I used the following commands to create the plaintext file and then encrypt it 3 times, 2 of the times I use the same IV value:

```
echo "1234567890" > task6.txt
openssl enc -aes-128-cbc -e -in task6.txt -out task6_cipher1.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cbc -e -in task6.txt -out task6_cipher2.bin -K
00112233445566778889aabbccddeeff -iv 1020304050607080
openssl enc -aes-128-cbc -e -in task6.txt -out task6_cipher3.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

I the used the diff command to compare the results between the three encrypted files. Encrypted files 1 and 3 share the same IV, and when they are compared, we see that they are the exact same cipher, whereas the 2nd output file contains differences when compared to the others. This tells us that using the same IV value for the same plaintext file will result in all the ciphertexts being the same.

Next in this task, we look at OFB and how using the same IV for non-repeating plaintexts can result in a vulnerability. Given a plaintext document P1 and its ciphertext C1, along with plaintext P2 and its ciphertext C2, assuming we do not know the message of P2, but we do know P1, C1, and C2, could we find the message contained within P2? The answer is yes, and we will prove that in this step. To start, we create messages for P1 and P2 and then I encrypt them using the OFB mode. I then wrote out the xor.py program from the book so that we can xor the hex strings of the ciphertexts and the plain texts. Once I had the program written, I found the hex strings for P1, C1, C2. I then used the xor.py program to xor the hex strings from P1 and C1. The output from that, I then xor'd with C2. Once that was done, I piped an echo command with the hex string from the xor.py program to an xxd command which translates hex strings to normal strings. After doing so, the message from P2 appeared meaning we are successful. The command used are listed below:

```
echo -n "This is the known message!" > P1
echo -n "Here is the secret message" > P2
openssl enc -aes-128-ofb -e -in P1 -out C1 -K
00112233445566778899AABBCCDDEEFF -iv 00000000000000000000000000000000
openssl enc -aes-128-ofb -e -in P2 -out C2 -K
00112233445566778899AABBCCDDEEFF -iv 00000000000000000000000000000000
xxd -p P1
xxd -p C1
xxd -p C2
xor.py 546869732069732074686520b6e6f776e206d657373616765521
a98c92dd6a6093009b9f47b6f4edec5c81f8ae13bec9d9bd6c4f
xor.py b58189cb6a6093009b9f47b6ece6e0598aace31ba8c9cbbb6e0b
fde4fbae4a09e020eff722969f83832befd8c376cdbab8da096e
echo -n "4865726520697320746865207365637265742064657373616765" | xxd -r -p
```

If we had used CBC we will have the same situation where we can get the plaintext using XOR but if the key is unknown, the ciphertext cannot be decrypted. The last part of this task has us looking into the use of predictable IV values. I used the experiment from the book to show how this works. I started by creating a message that imitates a vote that Bob has cast for John Smith. Eve wants to find out who Bob voted for but does not know the contents of the encrypted message . When Bob casts his vote (P1_2), the voting machine creates an IV value for it. The encrypted vote is stored in C1_2. If Eve knows the value of C1_2, she can calculate the next IV the machine will use. Eve then makes a guess as to what Bob's message contains (P1_2_guessed) and converts it to a hex string. We then xor P1_2_guessed with the IV that was assigned to Bob's message. After that, we take the result and xor it with the next IV that Eve obtained. We then convert the result from a hex string to normal and store it in P2_2. We then encrypt P2_2, and then compare the result, C2_2 with C1_2. If they are the same, then that means Eve's guess was right and she was able to figure out the contents of Bob's message. I compare them, and they do match, proving that predictable IV's can also cause trouble. The commands used are listed below:

```
echo -n "John Smith......" > P1_2
openssl enc -aes-128-cbc -e -in P1_2 -out C1_2 -K
00112233445566778899AABBCCDDEEFF -iv 4ae71336e44bf9bf79d2752e234818a5
echo -n "4ae71336e44bf9bf79d2752e234818a5" | xxd -r -p > IV_bob
md5sum IV_bob
echo -n "John Smith......" > P1_2_guessed
xxd -p P1_2_guessed
xor.py 4a6f686e20536d6974682e2e2e2e2e2e 4ae71336e44bf9bf79d2752e234818a5
xor.py 00887b58c41894d60dba5b000d66368b 398d01fdf7934d1292c263d374778e1a
echo -n "39057aa5338bd9c49f7838d37911b891" | xxd -r -p > P2_2
openssl enc -aes-128-cbc -e -in P2_2 -out C2_2 -K
00112233445566778899AABBCCDDEEFF -iv 398d01fdf7934d1292c263d374778e1a
xxd -p C1_2
xxd -p C2_2
```