Lab 1 Report

by Eli K. Martin for CS 4173

Lab source: https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_Encryption/

Pre-Lab Setup

Prior to starting the lab, I set up the VM on my Windows laptop as per the in-class instructions. I created a shared folder to transport files between them. Finally, I downloaded and set up the container for the encryption oracle used in Task 6.

Task 1: Frequency Analysis

In this task, we are asked to crack a monoalphabetic cipher using frequency analysis. I started by running the provided program freq.py as shown below in Figure 1.1. This showed the 20 most frequent unigrams, bigrams, and trigrams in the document. Then, I compared the most frequent letters to the “Letter frequency” article on Wikipedia (Source 1). According to them, the most frequent letter in English texts is the letter ‘E’ at 13%. The ciphertext holds 3931 non-space characters. The most frequent was ‘n’ at 488, making up approximately 12.4% of the document. Thus, I concluded that ‘n’ represented E and replaced it in the ciphertext as shown in Figure 1.2. Checking the bigrams and trigrams (with ‘yt’ and ‘ytn’ as the top places respectively) made it clear that ‘ytn’ represented ‘the’ based on the frequencies in the Wikipedia articles for ‘Bigram’ and ‘Trigram’ mentioned in the lab documents. I then added these replacements and regenerated the plaintext file as seen in Figure 1.3. At first, I ran the command for only replacing ‘yt’ with ‘HE’, but discovered that using the same file for both input and output resulted in a blank file, and of course prior changes were overridden by running the command a second time. I then tried replacing all 20 unigrams based on pure frequency, but they didn’t agree with what I had already found. Based on pure unigram frequency, ‘h’ in the ciphertext would have become ‘H’ in the plaintext, but we already knew that ‘H’ was encrypted into ‘t’. So instead, I flipped ‘h’ and ‘t’ in the unigram list to become ‘R’ and ‘H’ respectively. I continued this method of using the unigram frequency as a first guess, and comparing with the bigram and trigram frequencies as well as reading the plaintext. For example, in Figure 1.4 we can see my guess for THE makes sense, since it’s so frequent. A in line 7 is also probably correct since it is a standalone letter. However, ‘OG’ in line 4 is clearly not a word. Since O and F are direct neighbors in frequency, and ‘OF’ is an English word, I make the educated guess that ‘F’ and ‘G’ have been flipped. Overall, I found the bigram frequency data the least useful. Because their frequencies are so similar, and many of them involve similar letter patterns, it was difficult to parse useful information out of it. However, the trigram data was useful for the first few reported, although it sharply drops off afterwards. Once I exhausted the usefulness of these, I switched entirely to reading the plaintext and making educated guesses, which are chronicled in Table 1.5. Finally, I decrypted the entire plaintext, as shown in Figure 1.6. One struggle I had was actually seeing where the encrypted letters were, as after a certain point I started “autocorrecting” the words so to speak and stopped noticing the mistakes. For example, I did not realize that the final letter ‘Z’ was still encrypted as ‘w’ until I checked the number of letters in my cipher. The final replacement command can be seen in Figure 1.7.

Appendix 1:

```
[09/17/22]seed@VM:~/.../Files$ ./freq.py
----------------------------------------
1-gram (top 20):
n: 488
y: 373
v: 348
x: 291
u: 280
q: 276
```

Fig 1.1 Running the freq.py program to get the frequency analysis of the ciphertext.

```
[09/17/22]seed@VM:~/.../Files$ tr 'n' 'E' < ciphertext.txt > plaintext.txt
```

Fig 1.2 Replacing all instances of ‘n’ in the ciphertext with ‘E’ and outputting to the plaintext file.

```
[09/17/22]seed@VM:~/.../Files$ tr 'ytn' 'THE' < ciphertext.txt > plaintext.txt
```

Fig 1.3 Replacing all instances of ‘ytn’ with ‘THE’.

```
1 THE ONCARN TMRI  OI NMILAB WHSCH NEEUN AFOMT RSYHT AGTER THSN
  DOIY NTRAIYE
2 AWARLN TRSP THE FAYYER GEEDN DSsE A IOIAYEIARSAI TOO
3
4 THE AWARLN RACE WAN FOOsEILEL FB THE LEUSNE OG HARfEB
  WESINTESI AT STN OMTNET
5 AIL THE APPAREIT SUPDONSOI OG HSN GSDU COUPAIB AT THE EIL AIL
  ST WAN NHAPEL FB
6 THE EUERYEICE OG UETOO TSUEN MP FDACsYOWI PODSTSCN ARUCAILB
  ACTSfSNU AIL
7 A IATSOIAD COIfERNATSOI AN FRSEG AIL UAL AN A GEfER LREAU
  AFOMT WHETHER THERE
```

Fig 1.4 An example of the partially decrypted plaintext.

Table 1.5 Educated Guesses Based on Plaintext	
Guess Based on Plaintext	**Resulting Change to Cipher**
HARfEB WEINSTEIN should be HARVEY WEINSTEIN	add f to V and switch B and Y
EkTRA should be EXTRA	add k to X

BOOsENDED should be BOOKENDED	add s to K
oUST should be JUST	add o to J
SMNDAY should be SUNDAY	swap M and U
jUESTION should be QUESTION	add j to Q

```
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
```

Fig 1.6 The fully decrypted plaintext.

```
[09/17/22]seed@VM:~/.../Files$ tr 'nyvxuqmhtipaczlgbredfskojw' 'ETAONSIRHLDCMUWBFGPYVKXJQZ' < ciphertext.txt > plaintext.txt
```

Fig 1.7 The completed replacement command with the full key.

Task 2: Encryption using Different Ciphers and Modes

For this task, we are supposed to encrypt and decrypt text using different ciphers and modes. I started by using the plaintext generated in the last task, but quickly realized it was too long to be useful, so I created the plaintexts shown in Figures 2.1 and 2.2. Next, I used the command

given in the lab to encrypt it using AES-126-CBC as shown in Figure 2.3. Then, I converted the binary file to a more readable hexadecimal format as shown in Figure 2.4 using the xxd command from Task 4.

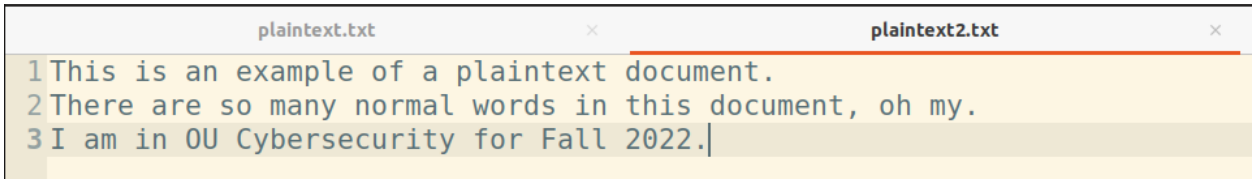

Fig 2.1 The plaintext I created for this task.

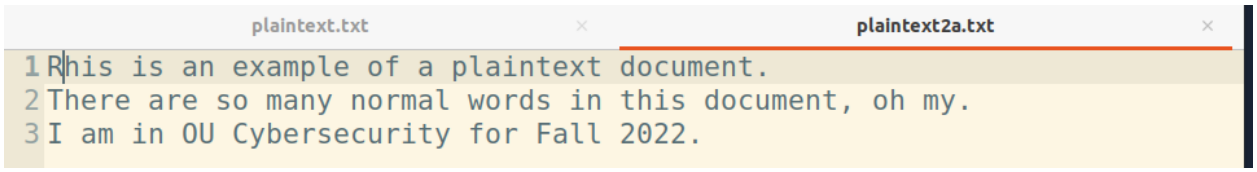

Fig 2.2 The second plaintext I created for this task. The only variation is the first letter, to easily show any avalanche effect.

```
[09/17/22]seed@VM:~/.../Files$  openssl enc -aes-128-cbc -e -in plaintex
t2.txt -out cipheraes.bin -K 00112233445566778889aabbccddeeff -iv 010203
0405060708
hex string is too short, padding with zero bytes to length
[09/17/22]seed@VM:~/.../Files$
```

Fig 2.3 The command to encrypt the plaintext with AES-128 CBC.

```
[09/18/22]seed@VM:~/.../Files$ xxd cipheraes.bin
00000000: 3587 a35a 7ee0 408a 51f8 7622 5f39 5c57  5..Z~.@.Q.v"_9\W
00000010: 5b25 881e 3e51 a5a3 26b4 3ef5 24ae cb43  [%..>Q..&.>.$..C
00000020: ab06 f77d ce2b 806a b6b5 54bb c5e3 2dad  ...}.+.j..T...-.
00000030: c4e2 04bb feb2 45d8 c95c bd86 7717 0d85  ......E..\..w...
00000040: 0147 150f 51e8 447b d413 b6d1 6fec cde2  .G..Q.D{....o...
00000050: 1667 2ec3 dad9 dc6c ec8a 0dff c7f9 a744  .g.....l.......D
00000060: 1544 2781 f606 13b7 b67c 2b7c b5ca 87e8  .D'......|+|....
00000070: 1c92 b636 b391 35f8 bebd 3063 30c1 17c3  ...6..5...0c0...
00000080: 2216 5ec4 9ed7 4938 6561 bf4b 79ee 78a4  ".^...I8ea.Ky.x.
[09/18/22]seed@VM:~/.../Files$ xxd cipheraesa.bin
00000000: 0d30 11e8 2c3a 8b8f a119 e632 98b3 daa6  .0..,:.....2....
00000010: e8d2 42e9 9d16 5aeb 6612 f0a1 fe9e 0e7b  ..B...Z.f......{
00000020: 976b 6373 1577 3ce0 2015 2827 1873 bffa  .kcs.w<. .('.s..
00000030: 2cf5 6155 e125 ec6b 58b1 4d28 3619 0b21  ,.aU.%.kX.M(6..!
00000040: c33a 6e1f 01a7 a141 4c14 fb9c 38f6 d356  .:n....AL...8..V
00000050: 5a32 138e b148 8cff 32d7 0f0e f65d 478c  Z2...H..2....]G.
00000060: d53e bef8 08ee 6093 1bc2 0bcb 5cb6 a5db  .>....`.....\...
00000070: 1b4f 9245 47e4 e533 3132 f520 06f1 1199  .O.EG..312. ....
00000080: 66a5 0b61 2ea2 256f c0e9 0242 60d5 b16c  f..a..%o...B`..l
```

Fig 2.4 The hexadecimal output from the encrypted .bin files.

```
[09/18/22]seed@VM:~/.../Files$ xxd plaintext2.txt
00000000: 5468 6973 2069 7320 616e 2065 7861 6d70  This is an examp
00000010: 6c65 206f 6620 6120 706c 6169 6e74 6578  le of a plaintex
00000020: 7420 646f 6375 6d65 6e74 2e0a 5468 6572  t document..Ther
00000030: 6520 6172 6520 736f 206d 616e 7920 6e6f  e are so many no
00000040: 726d 616c 2077 6f72 6473 2069 6e20 7468  rmal words in th
00000050: 6973 2064 6f63 756d 656e 742c 206f 6820  is document, oh
00000060: 6d79 2e0a 4920 616d 2069 6e20 4f55 2043  my..I am in OU C
00000070: 7962 6572 7365 6375 7269 7479 2066 6f72  ybersecurity for
00000080: 2046 616c 6c20 3230 3232 2e0a             Fall 2022..
[09/18/22]seed@VM:~/.../Files$ xxd plaintext2a.txt
00000000: 5268 6973 2069 7320 616e 2065 7861 6d70  Rhis is an examp
00000010: 6c65 206f 6620 6120 706c 6169 6e74 6578  le of a plaintex
00000020: 7420 646f 6375 6d65 6e74 2e0a 5468 6572  t document..Ther
00000030: 6520 6172 6520 736f 206d 616e 7920 6e6f  e are so many no
00000040: 726d 616c 2077 6f72 6473 2069 6e20 7468  rmal words in th
00000050: 6973 2064 6f63 756d 656e 742c 206f 6820  is document, oh
00000060: 6d79 2e0a 4920 616d 2069 6e20 4f55 2043  my..I am in OU C
00000070: 7962 6572 7365 6375 7269 7479 2066 6f72  ybersecurity for
00000080: 2046 616c 6c20 3230 3232 2e0a             Fall 2022..
[09/18/22]seed@VM:~/.../Files$
```

Fig 2.5 The hexadecimal output from the plaintext files, included for clarity and comparison.

Clearly, the AES encryption works. The encrypted files are drastically different from the original ones, and a small change in the plaintext does not translate to a small change in the ciphertext. Next, I tried DES encryption, as shown in Figure 2.6. These results were similar, with the ciphertext being drastically different than the plaintext, and not having an avalanche effect. I also tried varying the key by one bit as shown in Figure 2.7 and still got very different ciphertexts.

```
[09/18/22]seed@VM:~/.../Files$  openssl enc -des -e -in plaintext2.txt -
out cipherdes.bin -K 00112233445566778889aabbccddeeff -iv 01020304050607
08
hex string is too long, ignoring excess
[09/18/22]seed@VM:~/.../Files$  openssl enc -des -e -in plaintext2a.txt
-out cipherdesa.bin -K 00112233445566778889aabbccddeeff -iv 010203040506
0708
hex string is too long, ignoring excess
[09/18/22]seed@VM:~/.../Files$ xxd cipherdes.bin
00000000: f8a4 cc9e 6357 9ac3 ca05 1b00 8d62 7c4a  ....cW.......b|J
00000010: e6ac 96d5 80f8 b8d9 3a73 73ae 91a3 2aa1  ........:ss...*.
00000020: edb6 0d02 fb33 4b11 71f5 1a6f 1ecc 2b39  .....3K.q..o..+9
00000030: 8101 5c41 7d8d ff7c cfac d4b6 7369 3bb6  ..\A}..|....si;.
00000040: bf74 bb7a 4980 7a19 3260 bc76 cd0a 1378  .t.zI.z.2`.v...x
00000050: 46e2 cf4a 25a3 99c6 fc1a 70a1 0b4b fe72  F..J%.....p..K.r
00000060: 1275 6898 0b90 eb25 c64f ad2d 72f1 e1de  .uh....%.O.-r...
00000070: a995 5bd7 ce3b 13e8 0830 e57b 1eab 8cb1  ..[..;...0.{....
00000080: fe73 89ec 31d5 73a9 b281 c420 39dd 953c  .s..1.s.... 9..<
[09/18/22]seed@VM:~/.../Files$ xxd cipherdesa.bin
00000000: 1657 516a e5b0 b9e7 c775 3b1b f1a9 7cbe  .WQj.....u;...|.
00000010: 1615 2c50 8427 c3e2 ba8e a6ed 4ad7 a3bf  ..,P.'......J...
00000020: 2969 f170 1bcb 1ae7 3082 c305 d32e 8488  )i.p....0.......
00000030: 36d8 0edd 0973 fa72 f0d3 d3f7 125d 89b4  6....s.r.....]..
00000040: f1e1 a31a b4f6 170a b7e3 64f1 e95d df14  ..........d..]..
00000050: 8853 ef50 73f3 0e46 a027 113b dbeb dc1e  .S.Ps..F.';.....
00000060: 6426 950e 432d 8a11 bff7 8d03 9534 7aef  d&..C-.......4z.
00000070: 00a5 eb68 1e9a 480f bb56 f4bb 5764 14cc  ...h..H..V..Wd..
00000080: 636d e4fe 1675 7e3b 1106 941f 9475 ed16  cm...u~;.....u..
```

Fig 2.6 Encrypting both plaintexts in DES and showing their results.

```
[09/18/22]seed@VM:~/.../Files$  openssl enc -des -e -in plaintext2a.txt
-out cipherdesb.bin -K 001122334455668f -iv 0102030405060708
[09/18/22]seed@VM:~/.../Files$ xxd cipherdesb.bin
00000000: 14f0 3c69 41a6 88fe 50d1 cc0a 20ac 2878  ..<iA...P... .(x
00000010: c20d 313d ea0b 3c87 6648 0dd0 bf1e e7cb  ..1=..<.fH......
00000020: 3d4c f6ba fd4a 5a31 1d19 2760 c4d9 926e  =L...JZ1..'`...n
00000030: 2f7e 30ea 7fe4 3ebe 37d7 a34f 1c40 af04  /~0...>.7..O.@..
00000040: d957 2247 159b f2d7 68ea b84e e368 19ec  .W"G....h..N.h..
00000050: 18e2 f039 73fe 2770 8c82 8092 0b12 31b9  ...9s.'p......1.
00000060: 9c94 9d99 9afb f14e 871d 1b70 e12e d8fc  .......N...p....
00000070: 47e6 88ed 1124 e8d9 7028 0fcc dd0b 74f5  G....$..p(....t.
00000080: 7a3d 2924 9e09 8f5b aa3f 3e03 036f 7b26  z=)$...[.?>..o{&
```

Fig 2.7 Encrypting the second plaintext with a slightly different key (the second-to-last digit is changed by 1).

Finally, I wanted to compare ciphers of two different sizes, so I chose AES-128-CBC again and this time compared it with AES-256-CBC as shown in Figure 2.8. They obviously take two different size keys, and their ciphertexts were very different. Encrypting the second ciphertext with AES-256-CBC also produced a very different result. Overall, all 3 ciphers seemed secure, did not have avalanche effects, and ran quickly on the machine.

```
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in plaintext
2.txt -out cipher128.bin -K 00112233445566778899aabbccddeeff -iv 0011223
3445566778899aabbccddeeff
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-256-cbc -e -in plaintext
2.txt -out cipher256.bin -K 00112233445566778899aabbccddeeff001122334455
66778899aabbccddeeff -iv 00112233445566778899aabbccddeeff
[09/18/22]seed@VM:~/.../Files$ xxd cipher128.bin
00000000: 354e b95c 4b68 f215 5578 6493 43e4 400a  5N.\Kh..Uxd.C.@.
00000010: ab16 550a ff03 e30b 5c6e 3cda 711d 996e  ..U.....\n<.q..n
00000020: e79d 3bb8 d306 95d6 ca06 ec3e a327 972a  ..;........>.'.*
00000030: e22c 3b6f 58de 88b4 57a0 6393 998b c3e9  .,;oX...W.c.....
00000040: aef9 f561 984f 0aa9 7775 14dc a284 5b2c  ...a.O..wu....[,
00000050: 1a38 72af a9d1 f5c5 cfc7 37c4 58d6 4b63  .8r.......7.X.Kc
00000060: c813 7209 8dcc c212 5518 0b40 8ad5 431f  ..r.....U..@..C.
00000070: 4793 f68e 4eaf e155 019e a793 8cf4 be22  G...N..U.......".
00000080: e54b 3dce f5c7 5d98 a594 f703 63d3 b9b6  .K=...].....c...
[09/18/22]seed@VM:~/.../Files$ xxd cipher256.bin
00000000: ae93 c7f1 9b36 fb2b 54d5 e808 09b4 ac5b  .....6.+T......[
00000010: 0ec1 0c33 d058 8e87 6f10 e37e a651 d18e  ...3.X..o..~.Q..
00000020: 1123 1a67 e8e9 e301 165e cce8 ad5e 9565  .#.g.....^...^.e
00000030: bc60 e9c9 90be bf43 3cd9 fbc1 a2e9 4151  .`.....C<.....AQ
00000040: 4fc4 bab9 1a51 ff86 7fac 2b37 d0ce 1c97  O....Q....+7....
00000050: aeee 4771 d65c cdd7 3ed2 675e 2cd5 6095  ..Gq.\..>.g^,.`.
00000060: c88f 2094 a408 0dce 40dd 6d19 d804 29b3  .. .....@.m...).
00000070: c41d 053d 1a04 1dc0 0492 435d 9fbe 84aa  ...=......C]....
00000080: ce6a 877a f013 b555 ff8e 1f21 9b21 332b  .j.z...U...!.!3+
```

Fig 2.8 Comparing the output of AES-128-CBC and AES-256-CBC.

```
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-256-cbc -e -in plaintext
2a.txt -out cipher256a.bin -K 00112233445566778899aabbccddeeff0011223344
5566778899aabbccddeeff -iv 00112233445566778899aabbccddeeff
[09/18/22]seed@VM:~/.../Files$ xxd cipher256a.bin
00000000: 83e5 7e4b 307c e7ed bd7b 2442 16ee e1d0  ..~K0|...{$B....
00000010: 9f69 2966 28f8 2f9c 4fa2 8f21 0bd1 dd53  .i)f(./.O..!...S
00000020: 5899 16d7 ed77 0935 f7fb 6fd7 dcec a1ca  X....w.5..o.....
00000030: 3733 f2f3 f0ce 5a77 8bef 1f69 5790 bce9  73....Zw...iW...
00000040: 0ecc ada3 0f22 456e b1f3 59ff f285 53a9  ....."En..Y...S.
00000050: 66e5 bd81 3be4 fc98 0062 2e13 48de f963  f...;....b..H..c
00000060: 8d0a b844 3f7e b0cb a458 43d3 479a 5b49  ...D?~...XC.G.[I
00000070: 0120 8e3a 9de1 cd3e fa6b 1f0a 534f c249  . .:...>.k..SO.I
00000080: e276 0237 6bf1 c720 0aea 8c05 3d99 5b00  .v.7k.. ....=.[.
```

Fig 2.9 Encrypting the second plaintext using AES-256-CBC.

Task 3: Encryption Mode – ECB vs. CBC

For this task, I started by encrypting the picture in both AES-128-ECB and AES-128-CBC as shown in Figure 3.1. The keys were the same, and I used an IV for the CBC encryption since it

is the only one that takes it. Then, I formatted the files as shown in Figure 3.2 to be able to view the pictures.

```
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in pic_origi
nal.bmp -out pic-ecb.bmp -K 00112233445566778899aabbccddeeff
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in pic_origi
nal.bmp -out pic-cbc.bmp -K 00112233445566778899aabbccddeeff -iv 0011223
3445566778899aabbccddeeff
```

Fig 3.1 Encrypting the picture in both ECB and CBC.

```
[09/18/22]seed@VM:~/.../Files$ head -c 54 pic_original.bmp > header
[09/18/22]seed@VM:~/.../Files$ tail -c +55 pic-cbc.bmp > body
[09/18/22]seed@VM:~/.../Files$ cat header body > pic-cbc-final.bmp
[09/18/22]seed@VM:~/.../Files$ tail -c +55 pic-ebc.bmp > body
tail: cannot open 'pic-ebc.bmp' for reading: No such file or directory
[09/18/22]seed@VM:~/.../Files$ tail -c +55 pic-ecb.bmp > body
[09/18/22]seed@VM:~/.../Files$ cat header body > pic-ecb-final.bmp
```

Fig 3.2 Formatting both encrypted pictures.

After formatting, I viewed both of the encrypted pictures. As expected, both pictures were clearly encrypted, but the ECB file retained signatures of the original data (Figure 3.3) while the CBC file did not (Figure 3.4). Because ECB encrypts in blocks, if two blocks contain the same data, they will be encrypted the same way. That's why the picture encrypted using ECB stil shows a circle. Even though the blocks in the circle were encrypted so they are no longer red, they were all encrypted the same way, so they all have a vague blue pattern. Similarly, the rectangle is no longer blue, but it is in the same place and is still visible because of the uniform yellow-ish encrypted blocks. CBC on the other hand chains each block together. Even though all the blocks in the circle contain red data, that data is mixed with data from the previous block, and thus appears more scrambled and obscured.

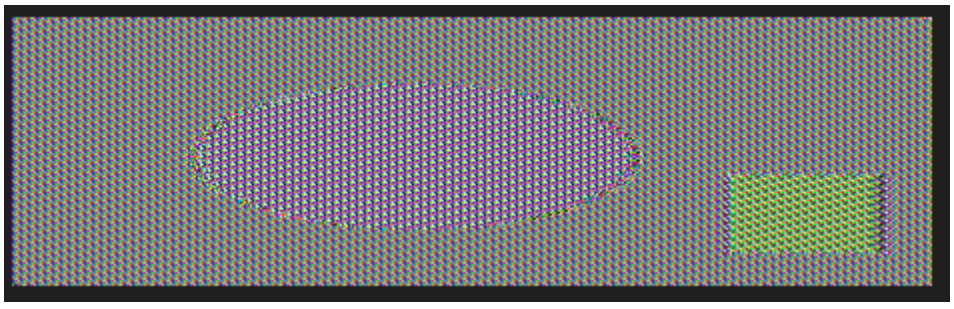

Fig 3.3 The image created using ECB encryption.

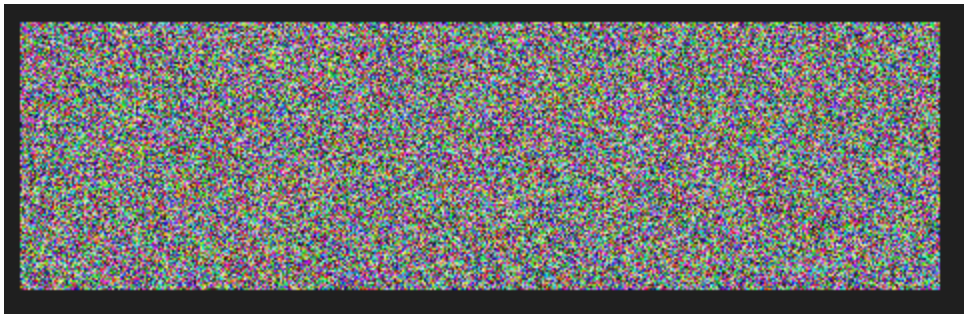

Fig 3.4 The image created using CBC encryption.

For the second image, I choose a photograph of a cat (Source 2) to see if the detail in the image would affect the ability to get information from the encrypted file. I used the same commands to encrypt and format the file, as shown in Figure 3.5. This time, it was much harder to distinguish which encrypted file was which, as we can see in Figures 3.6 and 3.7. The ECB file still has a pattern in the top left. This might be from the camera blur, making all the pixels appear similar, but I think it is more likely that the header on this picture was somehow different to the original picture given as an example. They were both in the .bmp filetype though, so I am unsure how to correct this issue if this is the case. Regardless, the rest of the file is still appears much more random than in the previous iteration of this experiment. Again, I think this is because of the higher level of detail and thus the greater difference in sequential pixels in the original picture. However, this does not imply that ECB is a safe way to transmit this file, as any identical portions of the output still correspond to identical portions of the input. It's just harder to visually see the information this way.

```
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in cat.bmp -
out cat-ecb.bmp -K 00112233445566778899aabbccddeeff
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in cat.bmp -
out cat-cbc.bmp -K 00112233445566778899aabbccddeeff -iv 0011223344556677
8899aabbccddeeff
[09/18/22]seed@VM:~/.../Files$ head -c 54 cat.bmp > header
[09/18/22]seed@VM:~/.../Files$ tail -c +55 cat-ecb.bmp > body
[09/18/22]seed@VM:~/.../Files$ cat header body > cat-ecb-final.bmp
[09/18/22]seed@VM:~/.../Files$ tail -c +55 cat-cbc.bmp > body
[09/18/22]seed@VM:~/.../Files$ cat header body > cat-cbc-final.bmp
```

Fig 3.5 Encrypting and formatting the cat pictures in both modes.

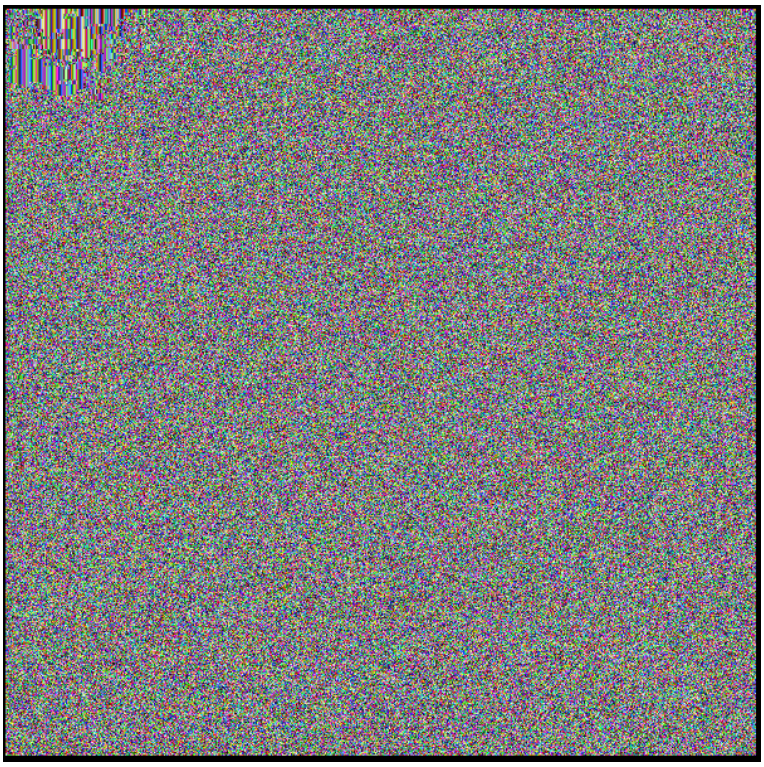

Fig 3.6 The ECB encrypted photograph.

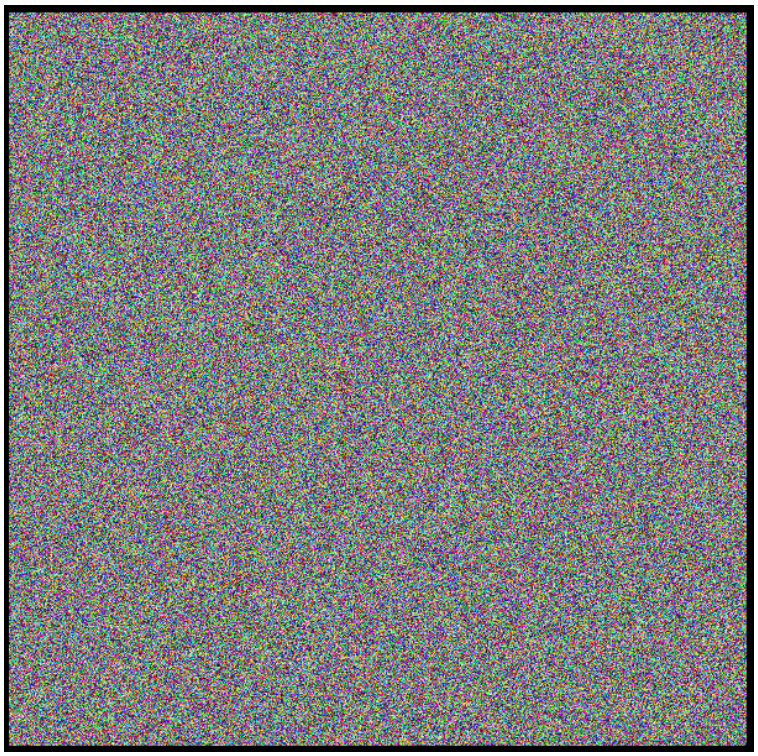

Fig 3.7 The CBC encrypted photograph.

Task 4: Padding

For part 1, I encrypted short.txt, which is a text file containing only the character '1' that is 2 bytes long using AES-128 with no padding. As shown in Figure 4.1, the ECB and CBC modes both require padding to function while CFB and OFB do not. This is because CFB and OFB are both stream cipher modes, meaning they encrypt a single bit at a time instead of a block of fixed size like ECB or CBC.

```
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in short.txt
 -out task4ecb.bin -K 00112233445566778899aabbccddeeff -iv 0011223344556
6778899aabbccddeeff -nopad
warning: iv not used by this cipher
bad decrypt
139675235968320:error:0607F08A:digital envelope routines:EVP_EncryptFina
l_ex:data not multiple of block length:crypto/evp/evp_enc.c:431:
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in short.txt
 -out task4cbc.bin -K 00112233445566778899aabbccddeeff -iv 0011223344556
6778899aabbccddeeff -nopad
bad decrypt
140015304852800:error:0607F08A:digital envelope routines:EVP_EncryptFina
l_ex:data not multiple of block length:crypto/evp/evp_enc.c:431:
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e -in short.txt
 -out task4cfb.bin -K 00112233445566778899aabbccddeeff -iv 0011223344556
6778899aabbccddeeff -nopad
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-ofb -e -in short.txt
 -out task4ofb.bin -K 00112233445566778899aabbccddeeff -iv 0011223344556
6778899aabbccddeeff -nopad
```

Fig 4.1 Encrypting a text file 1 byte long without padding. ECB and CBC both threw "bad decrypt" errors as highlighted in the figure.

For part 2, I first created the files f1.txt, f2.txt, and f3.txt and verified they had 5, 10, and 16 bytes respectively. Then I encrypted them as shown in Figure 4.2. The resulting files were each 16 bytes, regardless of the starting size. Finally, I decrypted them with no padding and viewed them, as shown in Figure 4.3. As we can see, file 1 is padded with "0b" repeating and file 2 is padded with "06" repeating. I thought file 3 would be padded with nothing, but instead it has 16 bytes of "10" repeating. The padding appears to be the 2-digit hexadecimal number for the amount of padding required, repeated as many times as necessary. For example, f1 required 16 bytes for the encrypted message - 5 bytes for the original = 11 bytes of padding. 11 is B in hexadecimal, so its padding was "0b" repeated.

```
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in f1.txt -o
ut o1.txt -K 00112233445566778899aabbccddeeff -iv 00112233445566778899aa
bbccddeeff
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in f2.txt -o
ut o2.txt -K 00112233445566778899aabbccddeeff -iv 00112233445566778899aa
bbccddeeff
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in f3.txt -o
ut o3.txt -K 00112233445566778899aabbccddeeff -iv 00112233445566778899aa
bbccddeeff
```

Fig 4.2 Encrypting each of the files for part 2.

```
[09/18/22]seed@VM:~/.../Files$ xxd f1.txt
00000000: 3132 3334 35                                12345
[09/18/22]seed@VM:~/.../Files$ xxd d1.txt
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
[09/18/22]seed@VM:~/.../Files$ xxd f2.txt
00000000: 3132 3334 3536 3738 3961                    123456789a
[09/18/22]seed@VM:~/.../Files$ xxd d2.txt
00000000: 3132 3334 3536 3738 3961 0606 0606 0606  123456789a......
[09/18/22]seed@VM:~/.../Files$ xxd f3.txt
00000000: 3132 3334 3536 3738 3961 6263 6465 6630  123456789abcdef0
[09/18/22]seed@VM:~/.../Files$ xxd d3.txt
00000000: 3132 3334 3536 3738 3961 6263 6465 6630  123456789abcdef0
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
```

Fig 4.3 Side-by-side comparison of the original files (f1, f2, f3) with their decrypted counterparts (d1, d2, d3).

Task 5: Error Propagation – Corrupted Cipher Text

Prior to completing this task, these are my beliefs about data recovery when the encryption file becomes corrupted:
For ECB, you can recover everything except the block that was affected (so in this case, 16 bytes will be lost) because it encodes per-block, but does not use data from other blocks.
For CBC, you can recover all except two blocks (so in this case, 32 bytes) because all the data has already been generated. So for decryption, you only need the block you're working on and the one directly next to it.
For CFB, similar to CBC, you can recover all except two blocks, because it also requires the current block as well as the one next to it to decrypt.
For OFB, you can recover everything except one bit, because you are not depending on anything else besides the pad.

To show my process I will use images from the CBC cipher mode. First, I created a text file called "ann.txt" which contains the poem Annabel Lee (Source 3) and is 1,557 bytes. Next, I encrypted it with AES as shown in Figure 5.1. Then, I opened the encrypted file ann_e.txt in

Bless and selected the 55th byte, and changed it from 2 in hexadecimal to 3 as shown in Figures 5.3 and 5.4. Finally, I decrypted this corrupted file using the original key and IV.

```
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in ann.txt -
out ann_e.txt -K 00112233445566778899aabbccddeeff -iv 00112233445566778
99aabbccddeeff
```

Fig 5.1 Encrypting the text file using AES-128-CBC.

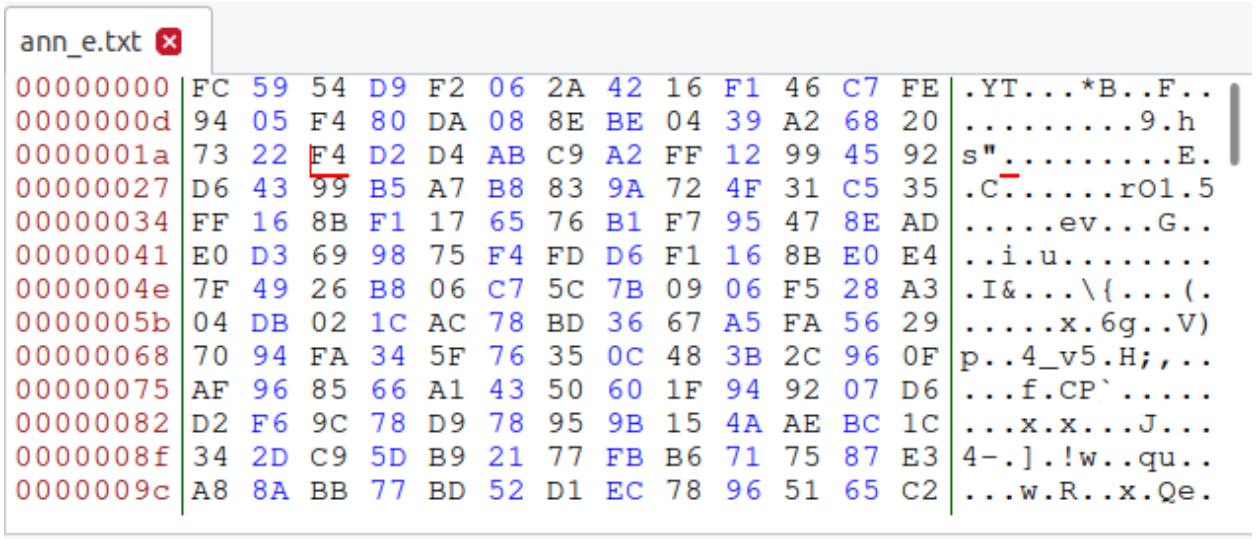

Fig 5.2 The original encrypted file in Bless.

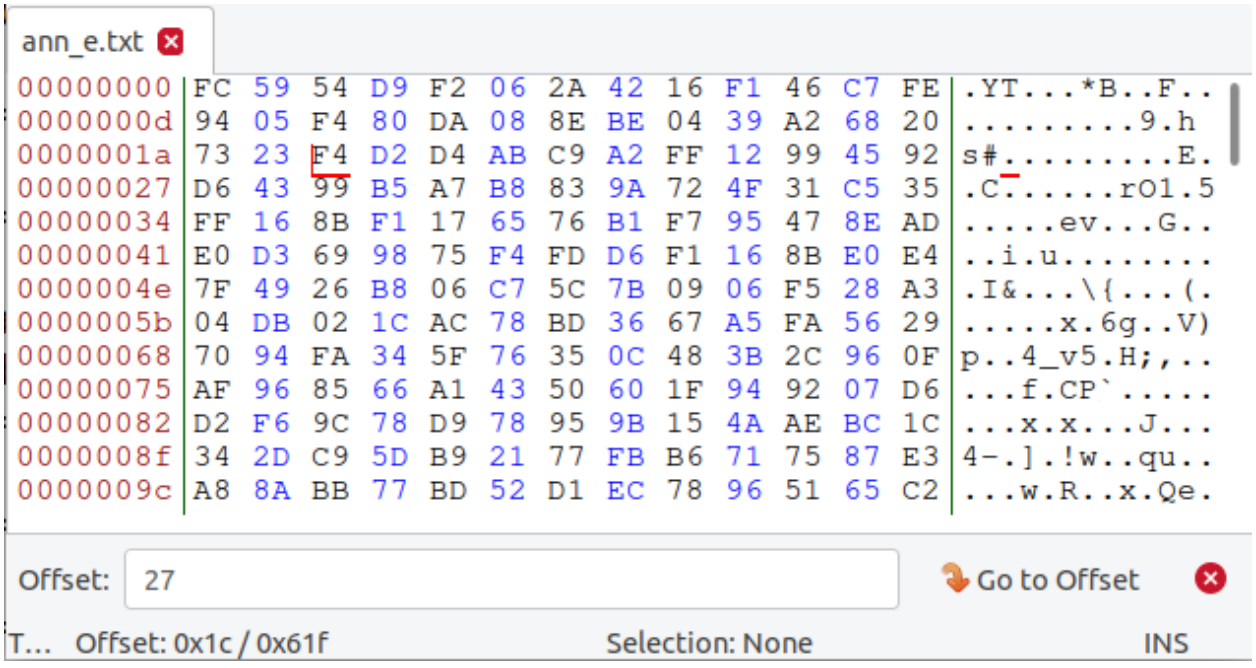

Fig 5.3 The encrypted file in Bless with the 55th byte corrupted.

Fig 5.4 Decrypting the corrupted ciphertext file.

To compare results, I used an online text comparator to find differences (Source 4).
For ECB, I was correct in stating that one block would be affected. The second set of 128 bits was incorrect once decrypted.
For CBC, I was also correct. The same block as the ECB file was corrupted, as well as a single bit in the next byte.
For CFB, I was also correct. The same blocks as CBC were corrupted, but both blocks were almost unreadable, instead of the single bit in the second block in CBC.
For OFB, I was also correct. There was only one bit corrupted, although it was not the same bit that I thought it would be. Again, I believe this is due to the scrambling function.

One interesting note is that every file except the OFB file also had corrupted spacing at the end of some lines in addition to the corrupted parts mentioned. However, the issue was consistent regardless of where it was repeated in the file, so I think it was due to a formatting issue because of the corruption and not aligned with where the actual corruption was happening.

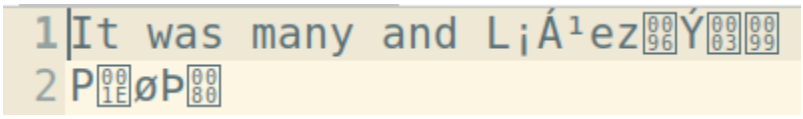

Fig 5.5 The corrupted portion of the ECB decrypted file.

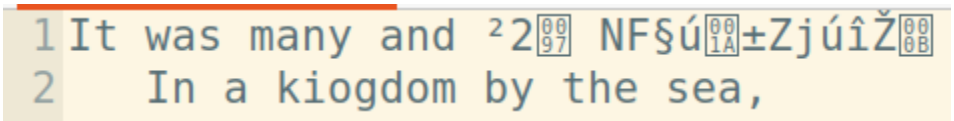

Fig 5.6 The corrupted portion of the CBC decrypted file.

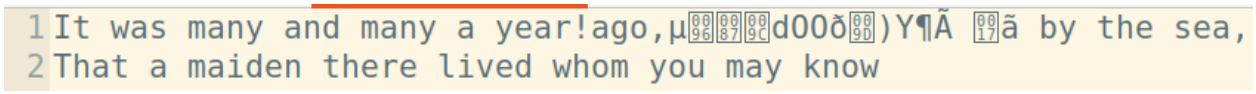

Fig 5.7 The corrupted portion of the CFB decrypted file.

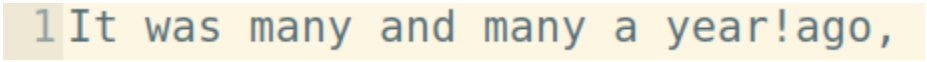

Fig 5.8 The corrupted portion of the OFB decrypted file.

Task 6: Initial Vector

For part 1, I encrypted the same plaintext as task 2. The first IV I chose was the same as the IV for task 5, and the second IV has the last byte different by 1, as shown in Figure 6.1. As we can see in Figure 6.2, choosing the same key AND IV results in the same ciphertext, while the same key but a different IV results in a completely different ciphertext.

```
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-ofb -e -in plaintext
2.txt -out plaintext2_1.txt -K 00112233445566778899aabbccddeeff -iv 0011
2233445566778899aabbccddeeff
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-ofb -e -in plaintext
2.txt -out plaintext2_2.txt -K 00112233445566778899aabbccddeeff -iv 0011
2233445566778899aabbccddeeff
[09/18/22]seed@VM:~/.../Files$ openssl enc -aes-128-ofb -e -in plaintext
2.txt -out plaintext2_3.txt -K 00112233445566778899aabbccddeeff -iv 0011
2233445566778899aabbccddeefe
```

Fig 6.1 Encrypting for part 1.

```
[09/18/22]seed@VM:~/.../Files$ xxd plaintext2_1.txt
00000000: 369e 10cd 0b99 aa11 0570 23f9 db21 76c2  6........p#..!v.
00000010: 82c1 6fc3 53fc ca25 be20 40c1 01e0 ec29  ..o.S..%. @....)
00000020: f6ae 8ce3 a419 396f 0a16 952b 5172 44f2  ......9o...+QrD.
00000030: ee43 5fa7 c81b df0c ac68 3e0a c855 ea22  .C_......h>..U."
00000040: b58f b4bf fe31 98c3 ad46 ec59 c334 c92a  .....1...F.Y.4.*
00000050: eb4a 86f2 f1a8 306c 9884 a44c a399 4b43  .J....0l...L..KC
00000060: 0758 e813 0426 5838 ae86 6c98 5bbc 2a42  .X...&X8..l.[.*B
00000070: 2ec5 e2de aac0 1e64 e1e1 3ef3 a852 229b  .......d..>..R".
00000080: e58f cb2a 9ca6 2b4d 82c4 ba08            ...*..+M....
[09/18/22]seed@VM:~/.../Files$ xxd plaintext2_2.txt
00000000: 369e 10cd 0b99 aa11 0570 23f9 db21 76c2  6........p#..!v.
00000010: 82c1 6fc3 53fc ca25 be20 40c1 01e0 ec29  ..o.S..%. @....)
00000020: f6ae 8ce3 a419 396f 0a16 952b 5172 44f2  ......9o...+QrD.
00000030: ee43 5fa7 c81b df0c ac68 3e0a c855 ea22  .C_......h>..U."
00000040: b58f b4bf fe31 98c3 ad46 ec59 c334 c92a  .....1...F.Y.4.*
00000050: eb4a 86f2 f1a8 306c 9884 a44c a399 4b43  .J....0l...L..KC
00000060: 0758 e813 0426 5838 ae86 6c98 5bbc 2a42  .X...&X8..l.[.*B
00000070: 2ec5 e2de aac0 1e64 e1e1 3ef3 a852 229b  .......d..>..R".
00000080: e58f cb2a 9ca6 2b4d 82c4 ba08            ...*..+M....
[09/18/22]seed@VM:~/.../Files$ xxd plaintext2_3.txt
00000000: c58c 1e9c bdb3 f0f8 0673 e9a0 282d 5d9d  .........s..(-].
00000010: 1033 b516 a8fc 779c 4751 b8cd 929d 87cd  .3....w.GQ......
00000020: 702c 1a71 f991 aa1e 8f02 7bf3 ad41 b954  p,.q......{..A.T
00000030: d04a 85c2 4e60 5fc2 64a6 fedf 1cfc e1fe  .J..N`_.d.......
00000040: f825 a42a 607c 3f70 6bed 7d35 f0c5 4734  .%.*`|?pk.}5..G4
00000050: f0d3 994a a4a9 5bec 8b8f 1977 5f13 d033  ...J..[....w_..3
00000060: 37de 5e18 4f4a 1fe4 c8f5 6d90 18ff beb3  7.^.OJ....m.....
00000070: d487 28e4 e753 5fb7 a5c5 055c 07b6 25f4  ..(..S_....\..%.
00000080: 6638 68f7 ced1 89ff 8f12 99ad            f8h.........
[09/18/22]seed@VM:~/.../Files$
```

Fig 6.2 The ciphertexts resulting from Figure 6.1.

For part 2, I used the sample code provided to XOR the texts. I replaced “MSG” with the known message, HEX_1 with C1 and HEX_2 with C2. I XOR’d the message with the first cyphertext to get the key, and then XOR’d that key with the unknown ciphertext. Finally, I decoded the

bytestream back to regular characters to get the message: “Order: Launch a missile!” The program is inserted below. If this were to be a CFB cipher instead of an OFB cipher, we would only be able to parse 2 blocks of the text instead of the whole thing because the blocks would be chained together instead of parallel and separate.

```
#!/usr/bin/python3

# XOR two bytearrays
def xor(first, second):
   return bytearray(x^y for x,y in zip(first, second))

MSG   = "This is a known message!"
HEX_1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159"
HEX_2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc159"

# Convert ascii string to bytearray
D1 = bytes(MSG, 'utf-8')

# Convert hex string to bytearray
D2 = bytearray.fromhex(HEX_1)
D3 = bytearray.fromhex(HEX_2)

r1 = xor(D1, D2) # should be the key?
r2 = xor(r1, D3) # should be the message?
r3 = xor(D2, D2)
print(r1.hex())
print(r2.decode('utf-8'))
print(r3.hex())
```

For the final part, I started by setting up the container and trying a few plaintexts in the Oracle. At first, I thought I might be able to spot a pattern, but it wasn’t so simple. To know Bob’s message, we want to be able to produce the same ciphertext as him. We know his ciphertext will be produced by encrypting the XOR of his message and his IV. We know our cipher will be made by encrypting the XOR of our message and our IV. If we choose our plaintext to be the XOR of both IVs and and what we think his message is, our encryption will be done on ((our guess) XOR (his IV) XOR (our IV)) XOR our IV. Since XOR cancels out if done again, the XOR of our IV cancels and we get just the encryption of (his IV) XOR (our guess). Thus, if that ciphertext matches his ciphertext, we will know we are right. However, after setting up a Python program to test for this, I was not getting any matches with Bob. I’m unsure why it doesn’t work. I believe I tried all permutations of the order of operations, although I don’t think it matters in this case because XOR is commutative. Below is my Python program to generate plaintexts and then some of my attempts at guessing the cipher. I also tried putting “yes” and “no” in lowercase letters, but that didn’t seem to matter either.

```
#!/usr/bin/python3
```

```
# XOR two bytearrays
def xor(first, second):
   return bytearray(x^y for x,y in zip(first, second))

MSG   = "Yes" # Our guess for his message
OUR_IV = "b5f7348634e2ae3f2d03d4a138f9bfc9"
HIS_IV = "2aff70c033e2ae3f2d03d4a138f9bfc9"

# Convert ascii string to bytearray
M = bytes(MSG, 'utf-8')

# Convert hex string to bytearray
O = bytearray.fromhex(OUR_IV)
H = bytearray.fromhex(HIS_IV)

r1 = xor(M, H)
r2 = xor(r1, O)
print(r2.hex())
```

```
[09/19/22]seed@VM:~$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 1b5b9ffb6afbe1e3fa62791ee031a86a
The IV used    : 2aff70c033e2ae3f2d03d4a138f9bfc9

Next IV        : b927381e34e2ae3f2d03d4a138f9bfc9
Your plaintext : ddb7
Your ciphertext: 22aa3d941411c14f85a3e9bc1971cd60

Next IV        : b5f7348634e2ae3f2d03d4a138f9bfc9
Your plaintext : c66d37
Your ciphertext: cc226a370deba2b46b109245e4d3128c

Next IV        : 901e11e834e2ae3f2d03d4a138f9bfc9
Your plaintext :
```

Fig 6.3 My attempt at guessing the plaintext. The program above shows my second guess "Yes", and previously I tried "No".

```
[09/19/22]seed@VM:~$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 40a1b3652296118678cc42535c10cce9
The IV used    : 039c4b5eb9de63988a487ff238479b72

Next IV        : 43dfc6abb9de63988a487ff238479b72
Your plaintext : 596573
Your ciphertext: 262015773a593e3523553908839058ce

Next IV        : 77303efeb9de63988a487ff238479b72
Your plaintext : 4e6f
Your ciphertext: a73c9780dbaaf50b7050837d1172a785

Next IV        : 0907c23bbade63988a487ff238479b72
Your plaintext :
```

Fig 6.4 Guessing “Yes” and then “No” directly as the plaintexts did not yield any useful information either.

Task 7: Programming using the Crypto Library

I was unable to find the correct cipher key and finish this task, but I will outline how I attempted to complete it. First, I created a program to scan each word in from the English words list and printed each one out, as shown in Figure 7.1. Next, I encrypted the given plaintext using a random key to determine the correct length. The handout says a length 16, but I was getting errors that way, so I also tried a length of 32. Neither seemed to work correctly in the code, and while using the in-line cipher commands, it took a 30 byte key. However, when I checked the file for this output, it said I only had 32 bytes instead of the 65 bytes that I had for the given ciphertext file. I thought maybe this could be resolved by encrypting more than once, so I attempted to write a program that would allow me to use each of the padded keys to encrypt the given plaintext. Then, I could check the produced ciphertext against the given ciphertext. If they were the same, I would know that the key used to produce that ciphertext was the one used to produce the original. However, this did not pan out. I continuously got the incorrect cipher length, as well as strange characters, as seen in Figure 7.3 and 7.4. I tried using the strcmp() function to compare the ciphertext from each potential key to the given ciphertext, but they did not seem to match up well when I used the xxd command (see Figure 7.5). I believe this issue was likely due to me not treating the information as the correct type, but I’m not sure how else I could have treated them. I tried printing the produced cipher in hexadecimal, but that gave me the same result every time (Figure 7.6). I will note that on the documentation page for the cipher functions (Source 5), it mentions that we should use a binary buffer to write to an output file because that is what the cipher is in. However, if I output binary data, I can’t read that data back in a useful way that’s not directly back into a character buffer, which is the problem I am trying to avoid.

```
scanned zoo
scanned zoology
scanned zoom
scanned Zorn
scanned Zoroaster
scanned Zoroastrian
scanned zounds
scanned z's
scanned zucchini
scanned Zurich
scanned zygote
[09/19/22]seed@VM:~/.../Files$
```

Fig 7.1 The first step of creating the program was to read in each word correctly.

```
[09/19/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in 7.t
xt -out transship_out.txt -K 01234567890abcdef01234567890abcd -iv
aabbccddeeff00998877665544332211
[09/19/22]seed@VM:~/.../Files$
```

Fig 7.2 Trying out a random key to see if I have the correct length.

```
scanned zucchini
padded zucchini#########################
*curr_length: 16 | strlen(curr): 20 | curr: ��� 1@
                                                </����+_+
strcmp(curr, goal): 195
scanned Zurich
padded Zurich##########################
*curr_length: 16 | strlen(curr): 20 | curr: gNO@S���5O9@ū%@_+
strcmp(curr, goal): 50
scanned zygote
padded zygote##########################
*curr_length: 16 | strlen(curr): 20 | curr: /@8@@@@@@%AƐ@_@_+
strcmp(curr, goal): -6
```

Fig 7.3 Despite multiple rewrites of the program, I kept getting nonsense characters like this.

```
scanned yawn
padded yawn############################
CHAOS! Cipher is the wrong length.
*curr_length: 16 | strlen(curr): 4 | curr: @.@
strcmp(curr, goal): 135
```

Fig 7.4 Another issue was ciphertexts of the incorrect size, which I chose to flag with “CHAOS!” to make them easier to spot in the output.

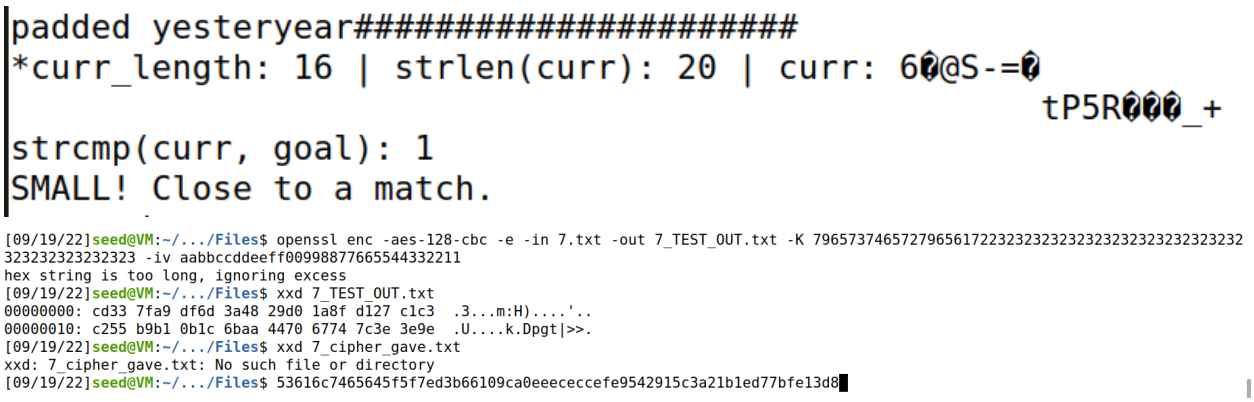

Fig 7.5 The top screencapture shows that the string comparison states that the current cipher and goal cipher are very close, while in fact they are quite different when we compare the xxd output of the one created with the same key with the hexadecimal we are given.

```
padded Zorn############################
*curr_length: 16 | strlen(curr): 20 | curr: 98A87420
strcmp(curr, goal): 183
scanned Zoroaster
padded Zoroaster######################
*curr_length: 16 | strlen(curr): 20 | curr: 98A87420
strcmp(curr, goal): 73
scanned Zoroastrian
padded Zoroastrian####################
*curr_length: 16 | strlen(curr): 20 | curr: 98A87420
strcmp(curr, goal): 163
scanned zounds
padded zounds#########################
*curr_length: 16 | strlen(curr): 20 | curr: 98A87420
strcmp(curr, goal): 163
scanned z's
padded z's############################
*curr_length: 16 | strlen(curr): 20 | curr: 98A87420
strcmp(curr, goal): 180
scanned zucchini
padded zucchini#######################
*curr_length: 16 | strlen(curr): 20 | curr: 98A87420
strcmp(curr, goal): 195
scanned Zurich
padded Zurich#########################
*curr_length: 16 | strlen(curr): 20 | curr: 98A87420
strcmp(curr, goal): 50
scanned zygote
padded zygote#########################
*curr_length: 16 | strlen(curr): 20 | curr: 98A87420
strcmp(curr, goal): -6
[09/19/22]seed@VM:~/.../Files$
```

Fig 7.6 By treating the variable curr as hexadecimal using %X instead of a string with %s I am able to get the hexadecimal characters I expect, but they are not the correct length AND they are all displaying the same, regardless of the key and string comparison result.

Below I will enclose a copy of the version of code that I think is closest to working. If it ran the way I thought it would, it would print out all keys until one produced a ciphertext matching the given one, at which point it would print out that key and the program would halt.

```
#include <openssl/evp.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
```

```
int main()
{
   printf("Program starting.\n");

   /* starting variables */
   char iv[] = "aabbccddeeff00998877665544332211";
   char intext[] = "oucybersecurity";
   char goal[] =
"53616c7465645f5f7ed3b66109ca0eeeceeccefe9542915c3a21b1ed77bfe13d8";
   EVP_CIPHER_CTX *ctx;
   FILE *in, *out;
   char buff[16];

   // where the current cipher will be stored
   int * curr_length;
   char curr[1000];

   // file to get the potential keys from
   in = fopen("words.txt", "r");

   //25143 words in the file
   for (int i = 1; i <= 25143; i++)
   {
     // read the word in
     fscanf(in, "%s", buff);

     printf("scanned %s\n", buff); //TESTING

     // pad the word to create the key
     while (strlen(buff) < 32)
     {
          char pad[] = "#";
          strcat(buff, pad);
     }

     printf("padded %s\n", buff); //TESTING

     // store and use it as the key
     ctx = EVP_CIPHER_CTX_new();
     EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, buff, iv);

     // encrypt using current key
     EVP_EncryptUpdate(ctx, curr, curr_length, intext, 16);
     EVP_EncryptFinal_ex(ctx, curr, curr_length);
```

```
      if (strcmp(curr, goal) == 0)
      {
            printf("The cipher key is: %s", buff);
            return 0;
      }

   }

   return 0;
}
```

References

1. Letter Frequency: https://en.wikipedia.org/wiki/Letter_frequency
2. Cat Picture: https://animalscene.ph/2018/06/19/panther-the-celebrity-cat/
3. Annabel Lee by Edgar Allan Poe: https://www.poetryfoundation.org/poems/44885/annabel-lee
4. Text Comparator: https://countwordsfree.com/comparetexts
5. Function documentation: https://www.openssl.org/docs/man1.1.1/man3/EVP_CipherInit.html