

## SEED Labs – Secret-Key Encryption Lab

Brian Grigore

### Task 1:

We begin the task by running the frequency analysis on the ciphertext, using the Python script. The output is shown in the screenshot:

```
[09/13/24]seed@VM:~/.../Files$ freq.py ciphertext.txt
-----
1-gram (top 20):
t: 437
z: 340
w: 269
h: 260
l: 228
g: 224
u: 216
m: 207
s: 203
c: 140
a: 108
x: 90
v: 87
f: 87
e: 82
q: 67
k: 61
b: 60
o: 42
d: 41
```

```
2-gram (top 20):
zg: 135
gt: 128
tl: 70
lt: 54
um: 50
wm: 44
wl: 41
tm: 40
uz: 38
gw: 38
hl: 38
wz: 37
ct: 36
ma: 36
vt: 35
hm: 35
hf: 32
ta: 30
wc: 28
ts: 28
```

```
3-gram (top 20):
zgt: 98
wma: 25
vtl: 18
fhl: 17
bhs: 17
hsv: 17
svt: 17
jgw: 16
gwp: 16
wpu: 16
zgw: 16
tlt: 16
umk: 15
xgt: 15
puu: 14
uuz: 14
gtl: 14
etl: 14
gwz: 12
tls: 11
```

Looking at the Wikipedia pages, we can try to find words that are common in the English language within our ciphertext.

in English writing are ETAOINSHRDLU, and some common 2-gram and 3-gram words appear as their own words such as “the”, “and”, and “as”.

Using the 12 most frequent letters, we will use the TR command to see how closely our distribution follows the norm.

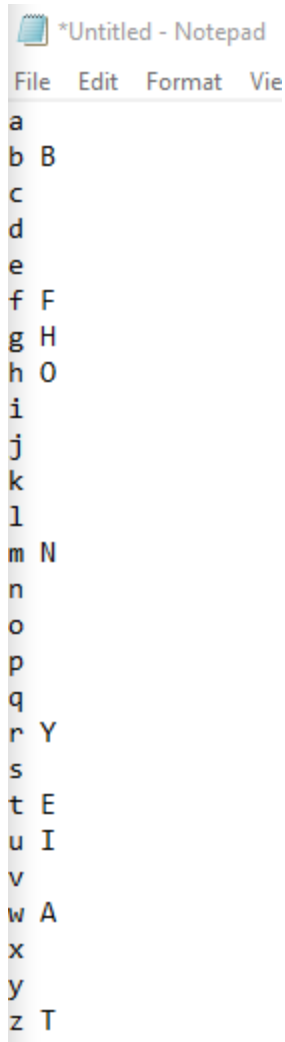
```
tr 'tzwhlgumscax' 'ETAOINSHRDLU' < ciphertext.txt > out1.txt
```

The resulting text is as follows:

```
vSiEL qHST TAUTSUR SH TNE fSdE rEAIR eAI  
d0DqvE OHE  
br U0LqR UADD0HqR  
TNE DEKSOHR U0qDL DEAIH fIOv TNE qHU0HdEHTSOHAD TAUTSUR qREL br TNE jNApSST SH  
TNE fSdE rEAIR eAI AKASHRT dADEHe00L S eAR RTATSOHEL AT TNE RoNSHiv0TN DEKSOH  
f0IT OH TNE b0ILEI HEAI LqHE AHL eSTHERREL vAHR Of TNE HOITNEIH RjSivSRNER  
fSIRTNAHL  
TNE eAI RTAITEL eSTN TNE ROUADDEL RDAqkNTEI Of TOIdAD TNE jNApSST UDASv  
TNAT TNE b0RvEI SHdALEL TNE USTr eSTN0qT oIOdOUATSOH AHL jSDDDEL OdeI A  
TN0qRAHL USTSyEHR bEf0IE bESHk LISdEH Off br IESHf0IUEvEHTR fIOv A HEAIbr  
pqHkDE TISbE TNE b0RvEI UDASv TNAT TNE ATTAUj eAR SH IETADSATSOH f0I jNApSTS|  
bAHLSTR eNO eEIE ATTAUjSHk e00L UAIAdAHR NEALEL f0I dADEHe00L  
SH TNE RoISHk Of E TNE eAI v0dEL UD0REI TO f0IT RoNSHiv0TN S eAR o0RTEL  
OH D00j0qT AHL RAe oAITR Of TNE UOHfDSUT S DATEI Ro0jE eSTN b0TN jNApSST AHL  
b0RvEI eNO f0qkNT SH TNE bATTDE AHL ST eSDD REIdE AR AH EiUEDDEHT EiAv0DE Of  
N0e TNE jNApSST qREL A vSiTqIE Of kIOqHL AHL TIEE qHSTR TO eSH TNE eAI  
TNE jNApSST beKAH TNE fSkNT SH AH qHqRqAD eAr br REHLSHK TIEEUqTTSBK TEAvR Of  
UATNArIANT AHL TNE fEAIROvE REHUNEIANT OI bATTDEUATR SHTO TNE 0qTRjSITR Of  
dADEHe00LR f0IERTR eNEH e0IL IEAUNEL TNE b0RvEI TNAT TIEER eEIE bESHk fEDEL  
ADDEKELDr A UISvE SH TNE RTIAHKe b0RvEIS IEDSKSOH A qHST Of AIUNEIR eEIE  
LSRoATUNEL fIOv DAIKEI UOHfDSUTR SH TNE R0qTN TNE b0RvEI eEIE TNqR k0ALEL
```

This initial guess does give us some more info that we can combine with our knowledge of common bigrams and trigrams from Wikipedia to refine our decoding.

We see TNE come up a lot as it's own word, and TNAT, so it is fair to assume that we should switch our N to an H. Since A appears on it's own and in the word TNAT, i believe it is also correct. Capital S should probably be switched to an I as it appears in locations where I is common. I'll write down what I think each letter maps to as I discover them, and re-run the tr command until the result is revealed. I will also guess br maps to BY as in this case the words seem to be in a short story format. Let's create a new tr command with the information gathered:



```
tr 'bfghmrtuwz' 'BFHONYEIAT' < ciphertext.txt > out2.txt
```

After running the command, I noticed a has to map to D to get the word AND, so i added it into the command and ran it again, giving us:

```
tr 'bfghmrtuwza' 'BFHONYEIATD' < ciphertext.txt > out2.txt
```

```
VIIED qNIT TAxTIXs IN THE FIde YEAlS eAl  
d0cqV E ONE  
BY x0Dqs xAccONqs  
THE cEkIONs x0qcD cEAlN Fl0v THE qNxONdENTIONAc TAxTIXs qSed BY THE jHApIIT IN  
THE FIde YEAlS eAl AkAINsT dAcENe00D I eAs sTATIONED AT THE soHINiv0TH cEkION  
FOlT ON THE BOlDEl NEAl DqNE AND eITNEssED vANY OF THE NOlTHElN sjIlvIsHEs  
FIlsTHAND
```

From this step there is more words we can figure out now, let's take a look:

qsED suggests q maps to U and s maps to S, and this is supported by S being at the end of many words. YEAls looks like YEARS, so l must map to R.

```
IN THE solINK OF E THE eAl vOdED xc0sEl TO F0lT soHINiv0TH I eAs o0sTED
ON c00j0qT AND sAe oAlTs OF THE x0NFcIxT I cATEl so0jE eITH BOTH jHApIIT AND
B0svEl eHO F0qkHT IN THE BATTcE AND IT eIcc sElde As AN EixEccENT EiAvocE OF
HOe THE jHApIIT qsED A vIiTqLE OF kl0qND AND TLEE qNITs TO eIN THE eAl
```

```
THE jHApIIT BEKAN THE FIKHT IN AN qNqsqAc eAY BY sENDINK TLEExqTTINK TEAvs OF
xATHAYLAHT AND THE FEAls0vE sENxHELAHT Ol BATTcExATs INTO THE 0qTsjiITs OF
dAcENe00Ds F0lEsTs eHEN e0lD lEAXHED THE B0svEl THAT TLEEs eELE BEINK FEccED
AccEKEDcY A xliVe IN THE sTLANKE B0svElI lEcIkION A qNIT OF AlxHEls eELE
DIsoATxHED Fl0v cAlkEl x0NFcIxTs IN THE s0qTH THE B0svEl eELE THqs k0ADED
INTO socITTINK THEIl F0lxEs INTO svAccEl kl0qos
```

```
THE B0svEl AlxHEls T00j qo o0sITIONs IN THE lEvAININK TLEEs eHOsE BLANxHEs
eELE NOe TeENTY Ol v0lE FEET AoAlT Acc0eINK s0vE cIkHT INTO THE F0lEsT Fc00l
THE B0svEl BENT THE lEvAININK TLEEs eITH THEIl vAkIxs INTO svAcc
F0lTIFIXATIONs Fl0v eHIxH TO FILE THEIl B0es
```

From this screenshot, we can see a lot more words that have obvious conclusions. o maps to P to make POSITIONS, k maps to G to make BEGAN, e maps to W to make TWENTY, c maps to L to make LATER, x and v map to C and M to make BECAME. Let's decipher with what we have now again to make the final letters easier to find and check for mistakes.

tr 'abcefgghklmoqrstuvwxyz' 'DBLWFHOGRNPUYSEIMACT' < ciphertext.txt > out3.txt

```
1 MIiED UNIT TACTICS IN THE FIdE YEARS WAR
2
3 dOLUME ONE
4
5 BY CODUS CALLONUS
6
7 THE LEGIONS COULD LEARN FROM THE UNCONDENTIONAL TACTICS USED BY THE jHApIIT IN
8 THE FIdE YEARS WAR AGAINST dALENWOOD I WAS STATIONED AT THE SPHINiMOTH LEGION
9 FORT ON THE BORDER NEAR DUNE AND WITNESSED MANY OF THE NORTHERN SjIRMISHES
0 FIRSHAND
.1
.2 THE WAR STARTED WITH THE SOCALLED SLAUGHTER OF TORdAl THE jHApIIT CLAIM
.3 THAT THE BOSMER INdADED THE CITY WITHOUT PR0dOCATION AND jILLED 0dER A
.4 THOUSAND CITIyENS BEFORE BEING DRIdEN OFF BY REINFORCEMENTS FROM A NEARBY
.5 pUNGLE TRIBE THE BOSMER CLAIM THAT THE ATTACj WAS IN RETALIATION FOR jHApITI
.6 BANDITS WHO WERE ATTACjing WOOD CARAdANS HEADED FOR dALENWOOD
.7
.8 IN THE SPRING OF E THE WAR MOded CLOSER TO FORT SPHINiMOTH I WAS POSTED
.9 ON LOOjOUT AND SAW PARTS OF THE CONFLICT I LATER SPOje WITH BOTH jHApIIT AND
.0 BOSMER WHO FOUGHT IN THE BATTLE AND IT WILL SERde AS AN EiCELLENT EiAMPLE OF
.1 HOW THE jHApIIT USED A MIiTURE OF GROUND AND TREE UNITS TO WIN THE WAR
```

This looks a lot closer to something legible. From here, we can figure out d maps to V, j maps to K, i maps to X to make EXCELLENT, p maps to J to make JUNGLE, y maps to Z to make DOZEN, and that leaves one letter, Q, for n.

Our final deciphering command is:

```
tr 'abcdefghijklmnopqrstuvwxyz' 'DBLVWFHOXKGRNQPJUYSEIMACZT' < ciphertext.txt > out4.txt
```

MIXED UNIT TACTICS IN THE FIVE YEARS WAR

VOLUME ONE

BY CODUS CALLONUS

THE LEGIONS COULD LEARN FROM THE UNCONVENTIONAL TACTICS USED BY THE KHAJIIT IN THE FIVE YEARS WAR AGAINST VALENWOOD I WAS STATIONED AT THE SPHINXMOOTH LEGION FORT ON THE BORDER NEAR DUNE AND WITNESSED MANY OF THE NORTHERN SKIRMISHES FIRSTHAND

THE WAR STARTED WITH THE SOCALLED SLAUGHTER OF TORVAL THE KHAJIIT CLAIM THAT THE BOSMER INVADDED THE CITY WITHOUT PROVOCATION AND KILLED OVER A THOUSAND CITIZENS BEFORE BEING DRIVEN OFF BY REINFORCEMENTS FROM A NEARBY JUNGLE TRIBE THE BOSMER CLAIM THAT THE ATTACK WAS IN RETALIATION FOR KHAJITI BANDITS WHO WERE ATTACKING WOOD CARAVANS HEADED FOR VALENWOOD

IN THE SPRING OF E THE WAR MOVED CLOSER TO FORT SPHINXMOOTH I WAS POSTED ON LOOKOUT AND SAW PARTS OF THE CONFLICT I LATER SPOKE WITH BOTH KHAJIIT AND BOSMER WHO FOUGHT IN THE BATTLE AND IT WILL SERVE AS AN EXCELLENT EXAMPLE OF HOW THE KHAJIIT USED A MIXTURE OF GROUND AND TREE UNITS TO WIN THE WAR

THE KHAJIIT BEGAN THE FIGHT IN AN UNUSUAL WAY BY SENDING TREECUTTING TEAMS OF CATHAYRAHT AND THE FEARSOME SENCHERAHT OR BATTLECATS INTO THE OUTSKIRTS OF VALENWOODS FORESTS WHEN WORD REACHED THE BOSMER THAT TREES WERE BEING FELLED ALLEGEDLY A CRIME IN THE STRANGE BOSMERI RELIGION A UNIT OF ARCHERS WERE DISPATCHED FROM LARGER CONFLICTS IN THE SOUTH THE BOSMER WERE THUS GOADED INTO SPLITTING THEIR FORCES INTO SMALLER GROUPS

## Task 2:

Running the command “man enc” reveals a large number of supported cipher types, including AES, ARIA, and Camellia.

```

aes-[128|192|256]-cfb 128/192/256 bit AES in 128 bit CFB mode
aes-[128|192|256]-cfb1 128/192/256 bit AES in 1 bit CFB mode
aes-[128|192|256]-cfb8 128/192/256 bit AES in 8 bit CFB mode
aes-[128|192|256]-ctr 128/192/256 bit AES in CTR mode
aes-[128|192|256]-ecb 128/192/256 bit AES in ECB mode
aes-[128|192|256]-ofb 128/192/256 bit AES in OFB mode

aria-[128|192|256]-cbc 128/192/256 bit ARIA in CBC mode
aria-[128|192|256] Alias for aria-[128|192|256]-cbc
aria-[128|192|256]-cfb 128/192/256 bit ARIA in 128 bit CFB mode
aria-[128|192|256]-cfb1 128/192/256 bit ARIA in 1 bit CFB mode
aria-[128|192|256]-cfb8 128/192/256 bit ARIA in 8 bit CFB mode
aria-[128|192|256]-ctr 128/192/256 bit ARIA in CTR mode
aria-[128|192|256]-ecb 128/192/256 bit ARIA in ECB mode
aria-[128|192|256]-ofb 128/192/256 bit ARIA in OFB mode

camellia-[128|192|256]-cbc 128/192/256 bit Camellia in CBC mode
camellia-[128|192|256] Alias for camellia-[128|192|256]-cbc
camellia-[128|192|256]-cfb 128/192/256 bit Camellia in 128 bit CFB mode
camellia-[128|192|256]-cfb1 128/192/256 bit Camellia in 1 bit CFB mode
camellia-[128|192|256]-cfb8 128/192/256 bit Camellia in 8 bit CFB mode
camellia-[128|192|256]-ctr 128/192/256 bit Camellia in CTR mode
camellia-[128|192|256]-ecb 128/192/256 bit Camellia in ECB mode
page enc(1) line 289 (press h for help or q to quit)

```

For our use, we will pick aes-128-cfb, aria-128-cbc, and camellia-128-cbc. The file we will encrypt is a copy of the ciphertext from the previous exercise called plain.txt. I've changed the -iv value to be 32 characters in length to satisfy the length requirements for aes-128.

Our commands are as follows:

```

openssl enc -aes-128-cfb -e -in plain.txt -out cipher.bin \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664

```

```

openssl enc -aria-128-cbc -e -in plain.txt -out cipher2.bin \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664

```

```

openssl enc -camellia-128-cbc -e -in plain.txt -out cipher3.bin \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664

```

Running the first command, we get:

```

[09/14/24]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e -in plain.txt -out cipher.bin -K 001122
33445566778889aabbccddeeff -iv 01020304050607086789647657865664
[09/14/24]seed@VM:~/.../Files$ █

```

Running the second;

Running the third:

```
[09/14/24]seed@VM:~/.../Files$ openssl enc -camellia-128-cbc -e -in plain.txt -out cipher3.bin \
> -K 00112233445566778889aabbccddeeff \
> -iv 01020304050607086789647657865664
```

We can now see the three encrypted files generated in the cwd.



Let's run a command to decrypt one of the files to verify our work. Instead of -e, we will use the -d option, and switch around the file names.

Our command is as follows:

```
openssl enc -aes-128-cfb -d -in cipher.bin -out newPlain.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

Running the command gives us:

```
[09/14/24]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -d -in cipher.bin -out newPlain.txt -K 001
12233445566778889aabbccddeeff -iv 01020304050607086789647657865664
```



```
nxhmdtmzuhmwc zwxzuxs qsta br zgt jgwpuuz um
ia u ews szwuhmta wz zgt sogumivhgz ctkuhm
izmtssta vwmr hf zgt mhlzgtlm sjulvusgts
```

```
scwqkgztl hf zhldwc zgt jgwpuuz xcwuv
```

And we can see that our original text file is back.

### Task 3:

We will begin this exercise by encrypting the two files using the same method (openssl command) as in the previous exercise. The output will be a .bmp instead of a .bin in this case. The K and IV values will be kept identical. We can find how to use AES in both CBC and ECB modes by viewing the enc manual page with the command “man enc”

```
aes-[128|192|256]-cbc 128/192/256 bit AES in CBC mode
aes[128|192|256]      Alias for aes-[128|192|256]-cbc
aes-[128|192|256]-cfb 128/192/256 bit AES in 128 bit CFB mode
aes-[128|192|256]-cfb1 128/192/256 bit AES in 1 bit CFB mode
aes-[128|192|256]-cfb8 128/192/256 bit AES in 8 bit CFB mode
aes-[128|192|256]-ctr 128/192/256 bit AES in CTR mode
aes-[128|192|256]-ecb 128/192/256 bit AES in ECB mode
aes-[128|192|256]-ofb 128/192/256 bit AES in OFB mode
```

For Image 1:

```
openssl enc -aes-128-ecb -e -in pic_original1.bmp -out pic1ecb.bmp \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -e -in pic_original1.bmp -out pic1cbc.bmp \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

For Image 2:

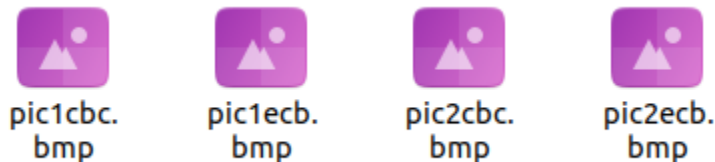
```
openssl enc -aes-128-ecb -e -in pic_original2.bmp -out pic2ecb.bmp \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -e -in pic_original2.bmp -out pic2cbc.bmp \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```



```
[09/14/24]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in pic_original1.bmp -out pic1ecb.bmp
\
> -K 00112233445566778889aabbccddeeff \
> -iv 01020304050607086789647657865664
warning: iv not used by this cipher
[09/14/24]seed@VM:~/.../Files$
[09/14/24]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in pic_original1.bmp -out pic1cbc.bmp
\
> -K 00112233445566778889aabbccddeeff \
> -iv 01020304050607086789647657865664
[09/14/24]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in pic_original2.bmp -out pic2ecb.bmp
\
> -K 00112233445566778889aabbccddeeff \
> -iv 01020304050607086789647657865664
warning: iv not used by this cipher
[09/14/24]seed@VM:~/.../Files$
[09/14/24]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in pic_original2.bmp -out pic2cbc.bmp
\
> -K 00112233445566778889aabbccddeeff \
> -iv 01020304050607086789647657865664
```

The ecb encryption method does not use the initialization vector, however that warning does not affect the operation of our command. The files now exist in our working directory.



We now need to bring back the header data so we can see the images with eog. To do so we will concatenate the data with the encrypted images into a new file, and repeat four times for all encryption modes.

```
head -c 54 pic_original1.bmp > header
tail -c +55 pic1cbc.bmp > body
cat header body > new1cbc.bmp
```

```
head -c 54 pic_original1.bmp > header
tail -c +55 pic1ecb.bmp > body
cat header body > new1ecb.bmp
```

```
head -c 54 pic_original2.bmp > header
tail -c +55 pic2cbc.bmp > body
cat header body > new2cbc.bmp
```

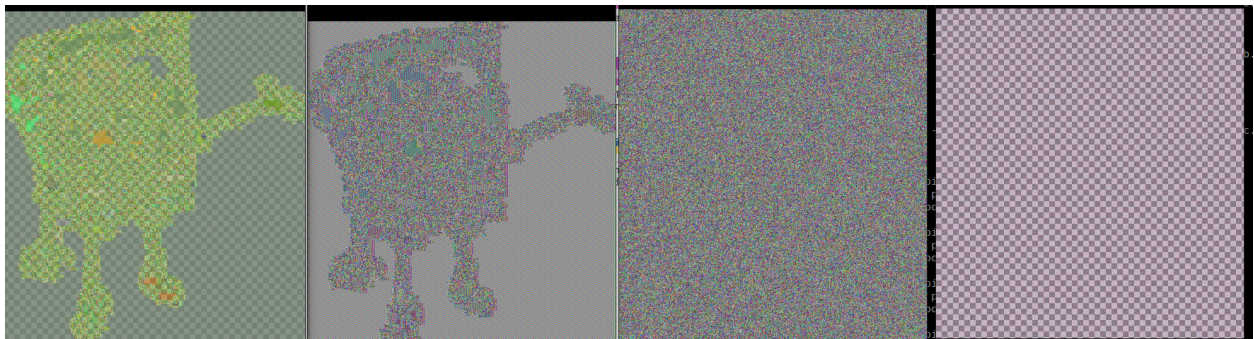
```
head -c 54 pic_original2.bmp > header
tail -c +55 pic2ecb.bmp > body
cat header body > new2ecb.bmp
```

```

[09/14/24]seed@VM:~/.../Files$ head -c 54 pic_original1.bmp > header
[09/14/24]seed@VM:~/.../Files$ tail -c +55 pic1cbc.bmp > body
[09/14/24]seed@VM:~/.../Files$ cat header body > new1cbc.bmp
[09/14/24]seed@VM:~/.../Files$
[09/14/24]seed@VM:~/.../Files$ head -c 54 pic_original1.bmp > header
[09/14/24]seed@VM:~/.../Files$ tail -c +55 pic1ecb.bmp > body
[09/14/24]seed@VM:~/.../Files$ cat header body > new1ecb.bmp
[09/14/24]seed@VM:~/.../Files$
[09/14/24]seed@VM:~/.../Files$ head -c 54 pic_original2.bmp > header
[09/14/24]seed@VM:~/.../Files$ tail -c +55 pic2cbc.bmp > body
[09/14/24]seed@VM:~/.../Files$ cat header body > new2cbc.bmp
[09/14/24]seed@VM:~/.../Files$
[09/14/24]seed@VM:~/.../Files$ head -c 54 pic_original2.bmp > header
[09/14/24]seed@VM:~/.../Files$ tail -c +55 pic2ecb.bmp > body
[09/14/24]seed@VM:~/.../Files$ cat header body > new2ecb.bmp

```

The files are as shown in the working directory:



I observe that when in ecb mode, you can derive the silhouette of the original image, in this case, the spongebob character from the encrypted image. In the images encrypted with cbc, it is not possible to determine the content of the original image.

Visually there are differences in the patterns and colours created in each of the encrypted images. The original 1 image is more monochromatic when encrypted and visually creates a large checked pattern, while the original 2 image creates what looks to be random multicolor noise. The reason they are different is that the original images contain different data, the original pic 1 has a file size of 1.9MB while the original image 2 has a file size of 1.4MB, so

when encrypted with the same algorithm and key, they will produce a different result. Since ECB encrypts each block in the same way, the original pattern of the image in the bitmap, i.e. the spongebob pattern is still discernable. The IV helps cbc to encrypt each block in a pseudorandom way, eliminating this problem.

#### **Task 4:**

For this task, I will use the ciphertext from assignment one, and the AES cipher. The commands to encrypt with ECB, CBC, CFB, and OFB are as follows:

```
openssl enc -aes-128-ecb -e -debug -in infile.txt -out cipher1.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -e -debug -in infile.txt -out cipher2.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cfb -e -debug -in infile.txt -out cipher3.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-ofb -e -debug -in infile.txt -out cipher4.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

We can check which modes have padding by decrypting the files with the -nopad option. The commands to do so are as follows:

```
openssl enc -aes-128-ecb -d -nopad -in cipher1.txt -out decr1.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -d -nopad -in cipher2.txt -out decr2.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cfb -d -nopad -in cipher3.txt -out decr3.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-ofb -d -nopad -in cipher4.txt -out decr4.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

ECB has padding:

1 alex|  
2 00000000000000000000000000000000

CBC has padding:

1 alex|  
2 00000000000000000000000000000000

CFB has no padding:

1 alex|

OFB has no padding:

1 alex|

Next I will make 3 files consisting of 5 ,10, and 16 bytes respectively. We can do this by running three commands:

```
echo -n 12345 > f1.txt  
echo -n 1234567890 > f2.txt  
echo -n 1234567890abcdef > f3.txt
```



Names: f1.txt, f2.txt, f3.txt  
Type: plain text document (text/plain)  
Contents: 3 items, totalling 31 bytes

We've created the 3 files with 5,10, and 16 bytes lengths respectively, totalling 31 bytes in the data.




Then, we will encrypt them with AES in CBC mode:

```
openssl enc -aes-128-cbc -e -in f1.txt -out enc1.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -e -in f2.txt -out enc2.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -e -in f3.txt -out enc3.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

	Name:	enc1.txt
	Type:	plain text document (text/plain)
	Size:	16 bytes
	Name:	enc2.txt
	Type:	plain text document (text/plain)
	Size:	16 bytes
	Name:	enc3.txt
	Type:	plain text document (text/plain)
	Size:	0 bytes

Enc 1 and 2 both show they've been padded to 16 bytes, while enc3 reports a size of 0 bytes, which may be a glitch. Listing the file size with ll shows enc3.txt has 32 bytes.

```
-rw-rw-r-- 1 seed seed 16 Sep 16 14:05 enc1.txt  
-rw-rw-r-- 1 seed seed 16 Sep 16 14:05 enc2.txt  
-rw-rw-r-- 1 seed seed 32 Sep 16 14:05 enc3.txt
```

And now we decrypt the files with the -nopad option:

```
openssl enc -aes-128-cbc -d -nopad -in enc1.txt -out denc1.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -d -nopad -in enc2.txt -out denc2.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -d -nopad -in enc3.txt -out denc3.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

-iv 01020304050607086789647657865664

Let's view each of the files with hexdump to see what padding was added:

hexdump -C denc1.txt

```
[09/16/24] seed@VM:~/.../Files$ hexdump -C denc1.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b  |12345.....|
00000010
```

hexdump -C denc2.txt

```
[09/16/24] seed@VM:~/.../Files$ hexdump -C denc2.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06  |1234567890.....|
00000010
```

hexdump -C denc3.txt

```
[09/16/24] seed@VM:~/.../Files$ hexdump -C denc3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10  |.....|
00000020
```



Names: denc1.txt, denc2.txt, denc3.txt  
Type: plain text document (text/plain)  
Contents: 3 items, totalling 64 bytes

After decryption, our total byte count for the 3 files is now 64.

From hexdump, we see that 11 bytes were added to f1, 6 bytes were added to f2, and 16 bytes were added to f3.

### Task 5:

For this task, I created a copy of the solved ciphertext from the first task, so that once the error is created, we can still read any recoverable data. This document has a size of 4093 bytes.



Name:

story.txt

Type:

plain text document (text/plain)

Size:

4.1 kB (4,093 bytes)

Before the task, I predict that the amount of recoverable information will vary by encryption mode. In ECB mode, since there is no XOR dependency on the next block, only the same block will be corrupted. In CBC mode, since the decryption result is XOR'ed with the ciphertext of the previous block, the current block will be corrupted and every block after will have a flipped bit. In CFB mode, because the feedback loop used to generate the keystream can be affected, I think the subsequent block will be affected. In OFB mode, the error will not propagate because, unlike CFB, the encrypted cipher is the feedback, not the plaintext.

We will start by encrypting story.txt with each mode:

```
openssl enc -aes-128-ecb -e -in story.txt -out cipher ECB.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

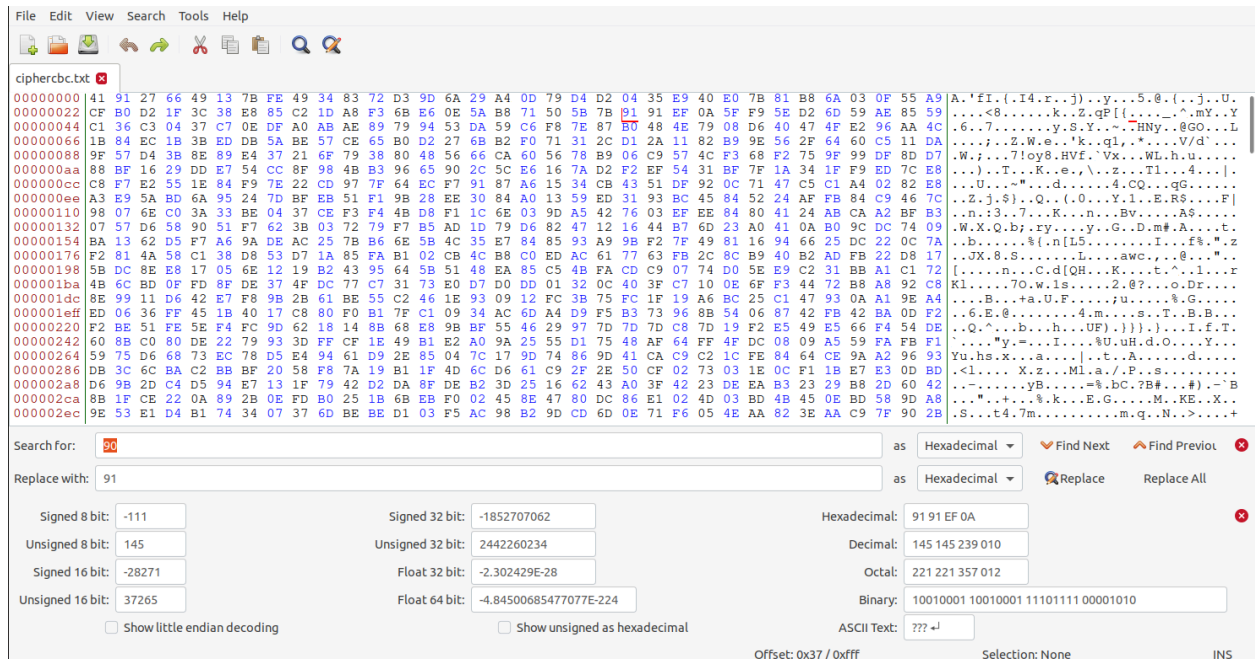
```
openssl enc -aes-128-cbc -e -in story.txt -out cipher CBC.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cfb -e -in story.txt -out cipher CFB.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-ofb -e -in story.txt -out cipher OFB.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

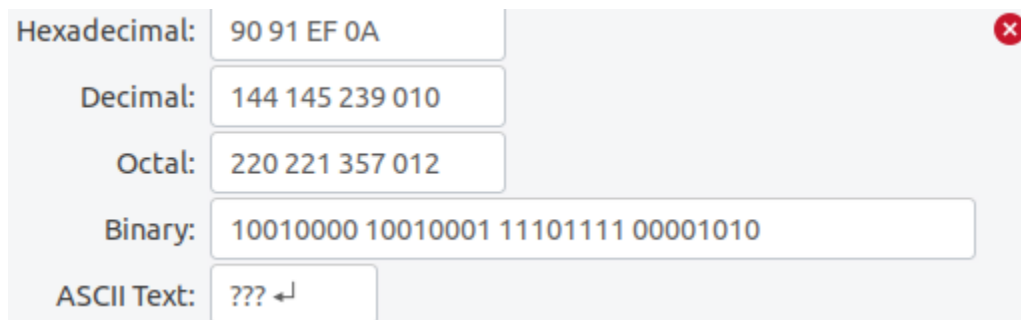
Now, we will flip one bit of the 55th byte in each of the encrypted files with a hex editor, in this case I am using Bless as it comes pre-installed in the SEED VM:





Starting with the cbc file our cursor is on the 55th byte, shown by the red underline on the 91 and verified by the offset indicator on the bottom of the screen, showing 0x37, or 55 in decimal. We will flip one bit by changing this 91 to a 90, changing it's binary representation from 10010001 to 10010000.

90 91



We can see that the first 91 has been flipped to a 90, changing one bit from 1 to 0 in the 55th byte. Now we can save and close our file.

In the second file, the cfb cipher, the 55th byte is a 47. We can flip one bit at the end of the byte from 1 to 0 to change it to a 46.

47 24



Hexadecimal:	46 24 AC 2E
Decimal:	070 036 172 046
Octal:	106 044 254 056
Binary:	01000110 00100100 10101100 00101110
ASCII Text:	F\$?.

In the third file, the ecb cipher, the 55th byte is a DE. We can flip one bit at the end of the byte from 0 to 1 to change it to a DF.

~~DE~~ A5

Hexadecimal:	DF A5 6C 7E
Decimal:	223 165 108 126
Octal:	337 245 154 176
Binary:	11011111 10100101 01101100 01111110
ASCII Text:	??!~

In the fourth file, the ofb cipher, the 55th byte is a 90. We can flip one bit at the end of the byte from 0 to 1 to change it to a 91.

90 D0

Hexadecimal:	91 D0 9A 73
Decimal:	145 208 154 115
Octal:	221 320 232 163
Binary:	10010001 11010000 10011010 01110011
ASCII Text:	???s

Now that all our files have been changed, we can decrypt them using the same keys and view what has been corrupted in the plaintext.

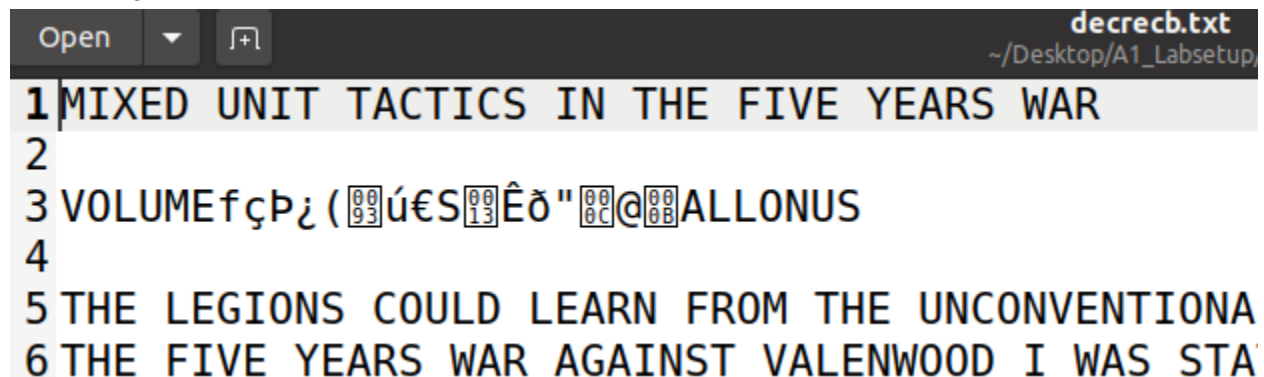
```
openssl enc -aes-128-ecb -d -in cipherecb.txt -out decrecb.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -d -in ciphercbc.txt -out decrcbc.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

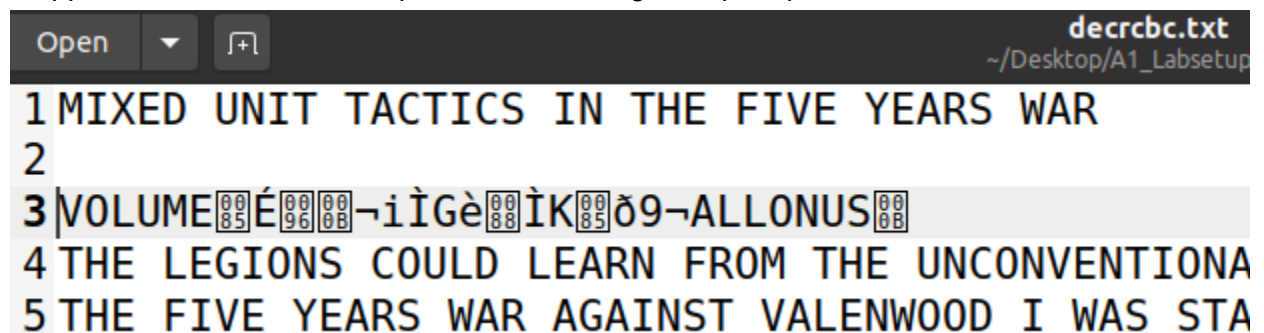
```
openssl enc -aes-128-cfb -d -in ciphercfb.txt -out decrcfb.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-ofb -d -in cipherofb.txt -out decrofb.txt \
-K 00112233445566778889aabbccddeeff \
-iv 01020304050607086789647657865664
```

In ECB mode, we can see that only the same block the error occurred in has been corrupted, with a length of 16 characters.



In CBC mode, We can see that the current block was corrupted, and the subsequent block has a flipped bit, due to the XOR operation containing corrupted plaintext.



In CFB mode, We can see that the subsequent block was affected because of the keystream being part of the XOR operation. We can also see the BX indicating our original flipped bit.

# MIXED UNIT TACTICS IN THE FIVE YEARS WAR

## VOLUME ONE

BX CODUS C\94\DC6\C2\00\80\F2~a\AF\E8"\A8ION!  
TACTICS USED BY THE KHAJIIT IN

In OFB mode, we can see that only the flipped bit was affected, and the error did not propagate because the feedback loop does not rely on the corrupted information.

## MIXED UNIT TACTICS IN T

## VOLUME ONE

BX CODUS CALLONUS

THE LEGIONS COULD LEARN

I was able to verify most of my initial predictions, with the exception of CBC. In CBC mode only the current block was corrupted and a subsequent bit, when I expected the entirety of the subsequent block to be affected, since the XOR relies on the present plaintext block.

### Task 6:

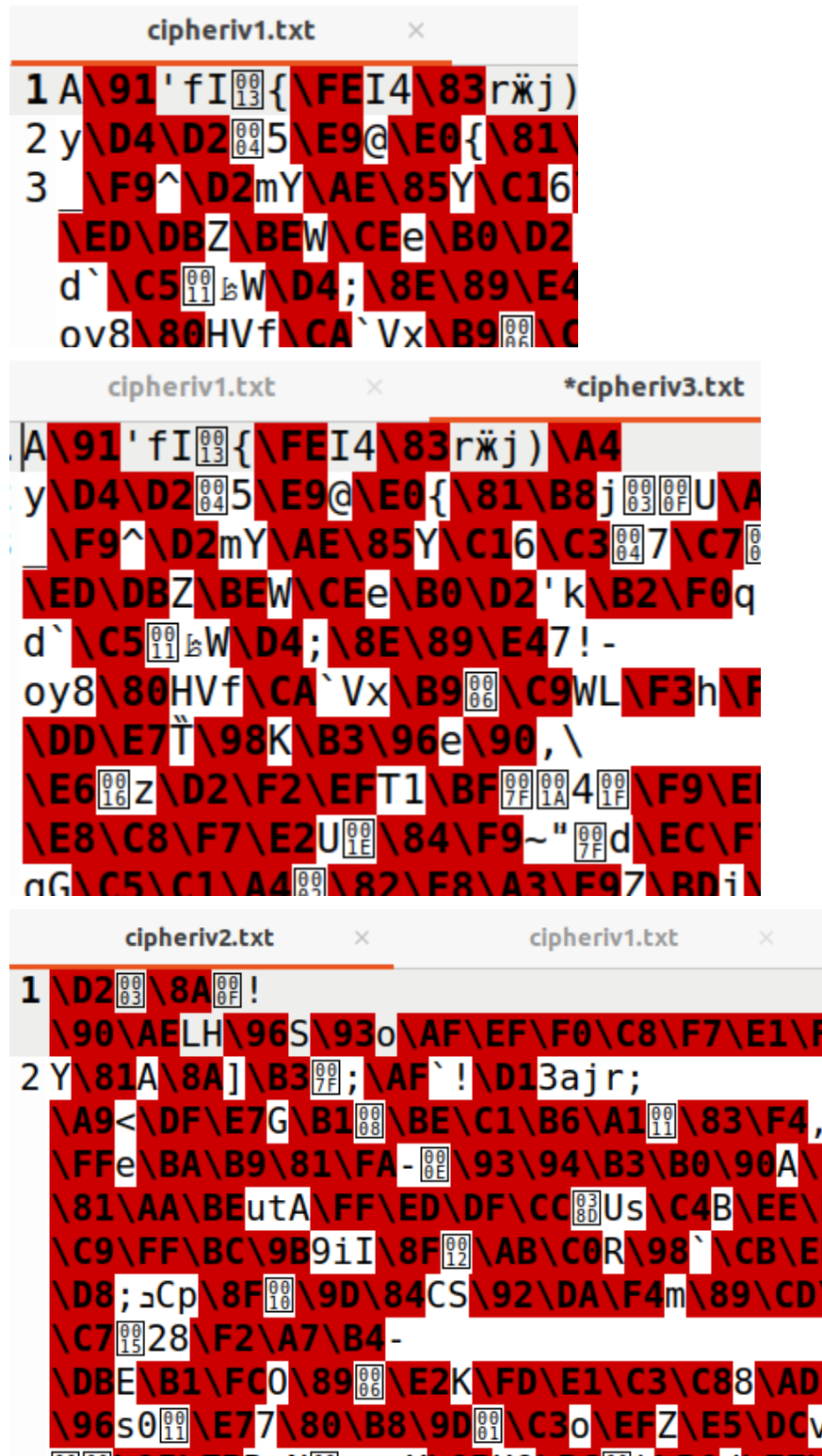
To begin the task, we can start by encrypting the story.txt from the previous task in CBC mode, using the same key with both a unique IV and the same IV for a total of 3 files. In our case, cipheriv1 and 3 have the same IV and cipheriv2 has a different IV.

```
openssl enc -aes-128-cbc -e -in story.txt -out cipheriv1.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

```
openssl enc -aes-128-cbc -e -in story.txt -out cipheriv2.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304056732487612534548762315
```

```
openssl enc -aes-128-cbc -e -in story.txt -out cipheriv3.txt \  
-K 00112233445566778889aabbccddeeff \  
-iv 01020304050607086789647657865664
```

We can observe that in CBC mode, when using the same key and IV, the two encrypted texts are identical. Thus if someone has the original plaintext and the ciphertext, they could use that to decode another plaintext encrypted with the same IV and key. This problem does not occur with a different IV.

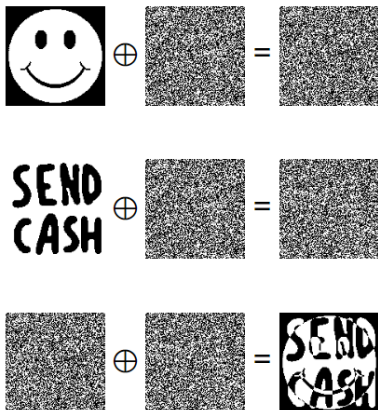


```
Plaintext (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

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

To figure out P2 based on the other 3 knowns, we can perform the operations described in the lecture slides to figure out the plaintext from a reused keystream.

## KEYSTREAM REUSE ILLUSTRATED



We will do this by XORing C1 and C2, to get both plaintexts, then XOR the result with the byte stream of P1 to get P2. This solution can be implemented by editing the sample python code to produce our result.

```
1#!/usr/bin/python3
2
3# XOR two bytearrays
4def xor(first, second):
5    return bytearray(x^y for x,y in zip(first, second))
6
7MSG = "This is a known message!"
8HEX_1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159"
9HEX_2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc159"
10
11# Convert ascii string to bytearray
12D1 = bytes(MSG, 'utf-8')
13
14# Convert hex string to bytearray
15D2 = bytearray.fromhex(HEX_1)
16D3 = bytearray.fromhex(HEX_2)
17
18r2 = xor(D2, D3)
19r3 = xor(r2, D1)
M20 print(r3.decode('utf-8'))
```

I've changed the sample code to first convert the plaintext "msg" into bytes on line 12, and denote it as D1. I then convert the two hex encoded ciphers into bytes, and set them to variables D2 and D3. I then perform the calculations described above, r2 is the XOR of the two ciphers, and r3 is the XOR of the combined cipher and the original plaintext. The script then converts r3 into plaintext encoded in utf-8 and displays it on the screen.

```
[09/17/24]seed@VM:~/.../Files$ sample_code.py
Order: Launch a missile!
```

We can now see the desired plaintext, which is Order: Launch a missile!

For the next part of this task, we need to simulate a chosen plaintext attack, in which we know the next IV, lowering the security of the algorithm. We will create a python script to take in information from the oracle and produce a result. Bob's secret message will be the result of Bob's plaintext message being XOR'd with the IV generated. If we know the upcoming IV, we can choose a plaintext, in this case Yes or No, that is equal to Bob's IV XOR our IV XOR our guess. Therefore, if our guess is a match for Bob's secret message, Then the encrypted blocks will be the same, indicating our guess is the same as the plaintext in Bob's secret message.

We'll start by running the oracle, obtaining the cipher, current IV, and next IV from Bob.

```
[09/17/24]seed@VM:~/.../Files$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertext: 13710afaa52b411f2461a7ce180ba509
The IV used      : 71b5a27f704bbcbcb15d25f4a71d0e122

Next IV         : 8d4ea4b2704bbcbcb15d25f4a71d0e122
Your plaintext : 
```

Our guesses will be Yes and No, padded to match the length of Bob's cipher. Since Bob is using AES, we will use PKCS#7 padding. We can do this easily with an online tool.

```
Block size:      16 bytes 128 bits
PKCS7 Pad:       5965730d0d0d0d0d0d0d0d0d0d0d0d0d
5965730d0d0d0d0d0d0d0d0d0d0d0d0d
```

For No, we will do the same.

```
Block size:      16 bytes 128 bits
PKCS7 Pad:       4e6f0e0e0e0e0e0e0e0e0e0e0e0e0e0e
```

With these Values, lets construct our code breaker in Python to do the XOR operations.

First, we will copy the xor method from sample\_code.py. Then, we will get our needed values in hex and convert them to bytes using the bytearray.fromhex() function. We then create Eve's plaintext based on the xor operations we described earlier, covert the result to hex with the .hex() function, and output to console. The code looks as follows:

```

1#!/usr/bin/python3
2
3# XOR two bytearrays
4def xor(first, second):
5    return bytearray(x^y for x,y in zip(first, second))
6
7Yes = bytearray.fromhex("5965730d0d0d0d0d0d0d0d0d0d0d")
8No  = bytearray.fromhex("4e6f0e0e0e0e0e0e0e0e0e0e0e0e")
9
10 CBob  = "0fb022ce4fb28284308747ad5c9913e0"
11 IvBob = bytearray.fromhex("2a0759df9aa0853c80652ad42a29ff39")
12 IvEve = bytearray.fromhex("3ef558219ba0853c80652ad42a29ff39")
13
14#XOR Bob's IV with Ours then XOR the Guess
15 GEve = xor(xor(IvBob, IvEve), Yes)
16
17print("Our Plaintext: " + GEve.hex())
18print("Bob's Ciphertext: " + CBob)

```

The output is as follows:

```

[09/17/24]seed@VM:~/.../Files$ python3 chsn_txt_atk.py
Our Plaintext: 4d9772f30c0d0d0d0d0d0d0d0d0d0d0d
Bob's Ciphertext: 0fb022ce4fb28284308747ad5c9913e0

```

Let's put our plaintext into the oracle:

```

Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertext: 0fb022ce4fb28284308747ad5c9913e0
The IV used      : 2a0759df9aa0853c80652ad42a29ff39

Next IV         : 3ef558219ba0853c80652ad42a29ff39
Your plaintext  : 4d9772f30c0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: 0fb022ce4fb28284308747ad5c9913e07a785d366cc46c605
3

Next IV        : e40c5e6d9ba0853c80652ad42a29ff39

```

We can see that once the plaintext is encrypted, the first block matches Bob's ciphertext, which means that our guess is correct, and Bob's secret message is "Yes"

### Task 7:

To begin this task, I consulted the openssl wiki to find out the setup for decrypting a message. [https://wiki.openssl.org/index.php/EVP\\_Symmetric\\_Encryption\\_and\\_Decryption](https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption) . I copied over the decrypt method and changed the error handling to not kill the program when an incorrect key is entered by replacing `handleErrors()` with `return -1`. I then downloaded a list of 10,000 english words from MIT <https://mit.edu/~ecprice/wordlist.10000> and imported it into an array

in my C program using fgets. I setup the elements of decryption, the ciphertext and initialization vector, by converting them from text to an unsigned char array in hex, and created a buffer for the decrypted text output. To process the text and pad it out with #s, I first tokenize the string to remove the newline character, and then pad it using the for loop in the pad\_word method. I then convert the characters to unsigned characters, since it is expected from the decrypt method. I then run the decrypt method, print the results to the terminal, and look in the output for the key and decrypted message.

The code is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>

#define MAX_WORD_LENGTH 16

void pad_word(char* word) {
    size_t len = strlen(word);
    if (len < MAX_WORD_LENGTH) {
        for (size_t i = len; i < MAX_WORD_LENGTH; i++) {
            word[i] = '#';
        }
        word[MAX_WORD_LENGTH] = '\0'; // null termination
    }
}

int decrypt(unsigned char *ciphertext, int ciphertext_len, unsigned char *key,
            unsigned char *iv, unsigned char *plaintext)
{
    EVP_CIPHER_CTX *ctx;

    int len;

    int plaintext_len;

    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        return -1;

    /*
     * Initialise the decryption operation. IMPORTANT - ensure you use a key
```



```

    * and IV size appropriate for your cipher
    * In this example we are using 256 bit AES (i.e. a 256 bit key). The
    * IV size for *most* modes is the same as the block size. For AES this
    * is 128 bits
    */
    if(1 != EVP_DecryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv))
        return -1;

    /*
    * Provide the message to be decrypted, and obtain the plaintext output.
    * EVP_DecryptUpdate can be called multiple times if necessary.
    */
    if(1 != EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len))
        return -1;
    plaintext_len = len;

    /*
    * Finalise the decryption. Further plaintext bytes may be written at
    * this stage.
    */
    if(1 != EVP_DecryptFinal_ex(ctx, plaintext + len, &len))
        return -1;
    plaintext_len += len;

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return plaintext_len;
}

int main (void)
{
    unsigned char ciphertext[] = {
        0x30, 0x43, 0xa4, 0x86, 0xeb, 0x13, 0xc2, 0x8a, 0xd3, 0x73, 0xf0, 0x97, 0xe4, 0x54, 0x86,
        0xdd,
        0xb0, 0x78, 0xed, 0x1e, 0xc1, 0x26, 0x66, 0xa7, 0x72, 0xba, 0x56, 0x86, 0xe5, 0xa8,
        0xac, 0x85,
        0x87, 0xee, 0x21, 0x23, 0xf7, 0x15, 0x02, 0x20, 0xba, 0x04, 0x1c, 0xb0, 0x1c, 0xec, 0x87,
        0x2e,
        0x88, 0xc6, 0x40, 0x6a, 0x4a, 0x8a, 0x09, 0xbd, 0x96, 0xb5, 0x5d, 0x1e, 0xaf, 0xae, 0xb1,
        0x70};
    unsigned char iv[] = {0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x00, 0x99, 0x88, 0x77, 0x66, 0x55,
        0x44, 0x33, 0x22, 0x11};

```

```

/* Buffer for the decrypted text */
unsigned char decryptedtext[128];

FILE *file = fopen("words.txt", "r");
if (file == NULL) {
    perror("Error opening file");
    return 1;
}

char word[256];
unsigned char key[MAX_WORD_LENGTH];
while (fgets(word, sizeof(word), file)) {
    strtok(word, "\n");

    if (strlen(word) <= MAX_WORD_LENGTH) {
        pad_word(word);
        for(int i = 0; i < 16; i++){
            key[i] = (unsigned char)word[i];
        }
        int decrypted_len = decrypt(ciphertext, sizeof(ciphertext), key, iv, decryptedtext);
        if (decrypted_len > 0){
            printf("Key: %s\n", word);
            printf("Decrypted message: %s\n", decryptedtext);

        }
    }

}

fclose(file);
return 0;
}

```

The output of the program is as follows:

```

Key: require#####
Decrypted message: Why don't programmers like nature? It has too many bugs.
Key: reseller#####
Decrypted message: [0d0000m00j_00"      0^0E0.00^Dx7"000|Rg0g 0V*6005x03k8c:03
Key: resulting#####
Decrypted message: i0S0>0_
3
o010w0F0A0:0qlM00
00      0}^0$00,y0#00&V000G0X0
Key: right#####
0p0rpted message: 6e10000000æ0AdE_00^0V0i/000m
0n:0l)B0X@0*00000@"0
Key: shift#####
Decrypted message: [+000h00=0t000U00009204X0X00000=U000^300      00009N000q
Key: specifications##
Decrypted message: y0]Eq0Wp0+^oE0\0&00[0$000l0'4n000000400$喊 000000&0Qt
Key: stars#####
Decrypted message: 0000*muB0جC00E%:fC0000000짚 0000000
Q000=l-0d000~
Key: take#####
Decrypted message: uo0700A0I00'00\0'0v0~u0000y>0000M0@0zc\90000000W/0Qz<00
Key: unauthorized####
Decrypted message: g00i0Y00;Cruvj0گ00ض01000Qc00I100~0br30WQH00]0=G0000000R
Key: via#####
Decrypted message: t00"000[]00h00e0-MbI0p0.kcw00
Key: voyuer#####
00%d0tNB00AH0saghg0')0Q10l1<D0b0y0u!Z z000W0000i0
Key: yukon#####
Decrypted message: 00i0ISC00W0YX0n000z0Ej0<0T00f000000wP
=====

```

By looking, we can see one of our keys, require, revealed the plaintext message: Why don't programmers like nature? It has too many bugs.