ECE 471 Lab 1

Secret-Key Encryption Lab

Mitchell Dzurick

2/17/2020

Github with all documentation - `https://www.github.com/mitchdz/ECE471`

Contents

Secret Key Encryption Lab

1 Overview

The learning objective of this lab is for students to get familiar with the concepts in the secret-key encryption. After finishing the lab, students should be able to gain a first-hand experience on encryption algorithms, encryption modes, paddings, and initial vector (IV). Moreover, students will be able to use tools and write programs to encrypt/decrypt messages. This lab covers the following topics:

- Secret-key encryption
- Substitution cipher and frequency analysis
- Encryption modes and paddings
- Programming using the crypto library

Lab Environment. This lab has been tested on our pre-built Ubuntu 12.04 VM and Ubuntu 16.04 VM, both of which can be downloaded from the SEED website.

2 Lab Tasks

2.1 Task 1: Frequency c Against Monoalphabetic Substitution Cipher

It is well-known that monoalphabetic substitution cipher (also known as monoalphabetic cipher) is not secure, because it can be subjected to frequency analysis. In this lab, you are given a cipher-text that is encrypted using a monoalphabetic cipher; namely, each letter in the original text is replaced by another letter, where the replacement does not vary (i.e., a letter is always replaced by the same letter during the encryption). Your job is to find out the original text using frequency analysis. It is known that the original text is an English article.

2.1.1 Task 1: solution

Shown below is the linux command

```
$ ls -l
```

in the lab1/task1 directory of my github repository. Inside that repository, the following command is used to copy the contents of the ciphertext into my X clipboard:

```
$ xclip -sel clip -i ciphertext.txt
```

```
mitch@light: ~/git/ECE471/lab1/task1 (master) $ ls -l
total 16
-rw-rw-r-- 1 mitch mitch 4759 Jan 29 18:44 ciphertext.txt
-rw-rw-r-- 1 mitch mitch 4759 Jan 29 18:49 plaintext.txt
mitch@light: ~/git/ECE471/lab1/task1 (master) $ xclip -sel clip -i ciphertext.txt
```

Figure 1: listing of lab1/task1 directory

The contents of ciphertext.txt is copied below:

```
ytn xqavhq yzhu  xu qzupvd ltmat qnncq vgxzy hmrty vbynh ytmq ixur qyhvurn
vlvhpq yhme ytn gvrrnh bnniq imsn v uxuvrnuvhmvu yxx

ytn vlvhpq hvan lvq gxxsnupnp gd ytn pncmqn xb tvhfnd lnmuqynmu vy myq xzyqny
vup ytn veevhnuy mceixqmxu xb tmq bmic axcevud vy ytn nup vup my lvq qtvenp gd
ytn ncnhrnuan xb cnyxx ymcnq ze givasrxlu eximymaq vhcavupd vaymfmqc vup
v uvymxuvi axufnhqvymxu vq ghmnb vup cvp vq v bnfnh phnvc vgxzy ltnytnh ytnhn
xzrty yx gn v ehnqmpnuy lmubhnd ytn qnvqxu pmpuy ozqy qnnc nkyhv ixur my lvq
nkyhv ixur gnavzqn ytn xqavhq lnhn cxfnp yx ytn bmhqy lnnsnup mu cvhat yx
vfxmp axubimaymur lmyt ytn aixqmur anhncxud xb ytn lmuynh xidcemaq ytvusq
ednxuratvur

xun gmr jznqymxu qzhhxzupmur ytmq dnvhq vavpncd vlvhpq mq txl xh mb ytn
anhncxud lmii vpphnqq cnyxx nqenamviid vbynh ytn rxipnu rixgnq ltmat gnavcn
v ozgmivuy axcmurxzy evhyd bxh ymcnq ze ytn cxfncnuy qenvhtnvpnp gd
```

exlnhbzi txiidlxxp lxcnu ltx tnienp hvmqn cmiimxuq xb pxiivhq yx bmrty qnkzvi tvhvqqcnuy vhxzup ytn axzuyhd

qmruvimur ytnmh qzeexhy rxipnu rixgnq vyynupnnq qlvytnp ytncqnifnq mu givas qexhynp iveni emuq vup qxzupnp xbb vgxzy qnkmqy exlnh mcgvivuanq bhxc ytn hnp avheny vup ytn qyvrn xu ytn vmh n lvq aviinp xzy vgxzy evd munjzmyd vbynh myq bxhcnh vuatxh avyy qvpinh jzmy xuan qtn invhunp ytvy qtn lvq cvsmur bvh inqq ytvu v cvin axtxqy vup pzhmur ytn anhncxud uvyvimn exhycvu yxxs v gizuy vup qvymqbdmur pmr vy ytn viicvin hxqynh xb uxcmuvynp pmhnayxhq txl axzip ytvy gn yxeenp

vq my yzhuq xzy vy invqy mu ynhcq xb ytn xqavhq my ehxgvgid lxuy gn

lxcnu mufxifnp mu ymcnq ze qvmp ytvy viytxzrt ytn rixgnq qmrumbmnp ytn mumymvymfnq ivzuat ytnd unfnh muynupnp my yx gn ozqy vu vlvhpq qnvqxu avcevmru xh xun ytvy gnavcn vqqxamvynp xuid lmyt hnpavheny vaymxuq muqynvp v qexsnqlxcvu qvmp ytn rhxze mq lxhsmur gntmup aixqnp pxxhq vup tvq qmuan vcvqqnp cmiimxu bxh myq inrvi pnbnuqn bzup ltmat vbynh ytn rixgnq lvq bixxpnp lmyt ytxzqvupq xb pxuvymxuq xb xh inqq bhxc enxein mu qxcn axzuyhmnq

ux avii yx lnvh givas rxluq lnuy xzy mu vpfvuan xb ytn xqavhq ytxzrt ytn cxfncnuy lmii vicxqy anhyvmuid gn hnbnhnuanp gnbxhn vup pzhmur ytn anhncxud nqenamviid qmuan fxavi cnyxx qzeexhynhq imsn vqtind ozpp ivzhv pnhu vup umaxin smpcvu vhn qatnpzinp ehnqnuynhq

vuxytnh bnvyzhn xb ytmq qnvqxu ux xun hnviid suxlq ltx mq rxmur yx lmu gnqy emayzhn vhrzvgid ytmq tveenuq v ixy xb ytn ymcn muvhrzvgid ytn uvmigmynh uvhhvymfn xuid qnhfnq ytn vlvhpq tden cvatmun gzy xbynu ytn enxein bxhnavqymur ytn hvan qxaviinp xqavhxixrmqyq avu cvsn xuid npzavynp rznqqnq

ytn lvd ytn vavpncd yvgzivynq ytn gmr lmuunh pxnquy tnie mu nfnhd xytnh avynrxhd ytn uxcmunn lmyt ytn cxqy fxynq lmuq gzy mu ytn gnqy emayzhn avynrxhd fxynhq vhn vqsnp yx imqy ytnmh yxe cxfmnq mu ehnbnhnuymvi xhpnh mb v cxfmn rnyq cxhn ytvu enhanuy xb ytn bmhqyeivan fxynq my lmuq ltnu ux cxfmn cvuvrnq ytvy ytn xun lmyt ytn bnlnqy bmhqyeivan fxynq mq nimcmuvynp vup myq fxynq vhn hnpmqyhmgzynp yx ytn cxfmnq ytvy rvhunhnp ytn nimcmuvynp gviixyq qnaxupeivan fxynq vup ytmq axuymuznq zuymi v lmuunh ncnhrnq

my mq vii ynhhmgid axubzqmur gzy veevhnuyid ytn axuqnuqzq bvfxhmyn axcnq xzy vtnvp mu ytn nup ytmq cnvuq ytvy nupxbqnvqxu vlvhpq atvyynh mufvhmvgid mufxifnq yxhyzhnp qenazivymxu vgxzy ltmat bmic lxzip cxqy imsnid gn fxynhq qnaxup xh ytmhp bvfxhmyn vup ytnu njzviid yxhyzhnp axuaizqmxuq vgxzy ltmat bmic cmrty ehnfvmi

```
mu  my lvq v yxqqze gnylnnu gxdtxxp vup ytn nfnuyzvi lmuunh gmhpcvu
mu  lmyt ixyq xb nkenhyq gnyymur xu ytn hnfnuvuy xh ytn gmr qtxhy ytn
ehmwn lnuy yx qexyimrty ivqy dnvh unvhid vii ytn bxhnavqynhq pnaivhnp iv
iv ivup ytn ehnqzceymfn lmuunh vup bxh ylx vup v tvib cmuzynq ytnd lnhn
axhhnay gnbxhn vu nufnixen quvbz lvq hnfnvinp vup ytn hmrtybzi lmuunh
cxxuimrty lvq ahxlunp

ytmq dnvh vlvhpq lvyatnhq vhn zunjzviid pmfmpnp gnylnnu ythnn gmiigxvhpq
xzyqmpn nggmur cmqqxzhm ytn bvfxhmyn vup ytn qtven xb lvynh ltmat mq
ytn gvrrnhq ehnpmaymxu lmyt v bnl bxhnavqymur v tvmi cvhd lmu bxh rny xzy

gzy vii xb ytxqn bmicq tvfn tmqyxhmavi xqavhfxymur evyynhuq vrvmuqy ytnc ytn
qtven xb lvynh tvq  uxcmuvymxuq cxhn ytvu vud xytnh bmic vup lvq viqx
uvcnp ytn dnvhq gnqy gd ytn ehxpzanhq vup pmhnayxhq rzmipq dny my lvq uxy
uxcmuvynp bxh v qahnnu vayxhq rzmip vlvhp bxh gnqy nuqncgin vup ux bmic tvq
lxu gnqy emayzhn lmytxzy ehnfmxzqid ivupmur vy invqy ytn vayxhq uxcmuvymxu
qmuan ghvfntnvhy mu  ytmq dnvh ytn gnqy nuqncgin qvr nupnp ze rxmur yx
ythnn gmiigxvhpq ltmat mq qmrumbmavuy gnavzqn vayxhq cvsn ze ytn vavpncdq
ivhrnqy ghvuat ytvy bmic ltmin pmfmqmfn viqx lxu ytn gnqy phvcv rxipnu rixgn
vup ytn gvbyv gzy myq bmiccvsnh cvhymu capxuvrt lvq uxy uxcmuvynp bxh gnqy
pmhnayxh vup vevhy bhxc vhrx cxfmnq ytvy ivup gnqy emayzhn lmytxzy viqx
nvhumur gnqy pmhnayxh uxcmuvymxuq vhn bnl vup bvh gnylnnu
```

The related plaintext is shown below using the final (substitution) key 'abcdefghijklmnopqrstuvwxyz' to 'CFMYPVBRLQXWIEJDSGKHNAZOTU'. The process to derive this key is outlined below with accompanying pictures.

```
$ tr 'abcdefghijklmnopqrstuvwxyz' \
     'CFMYPVBRLQXWIEJDSGKHNAZOTU' \
     < ciphertext.txt > plaintext.txt
```

Figure 2: Command to convert ciphertext to plaintext

Figure 2 shows the Linux command that is used in order to convert the ciphertext to the following plaintext.

```
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

Now that the accompanying plaintext with the ciphertext is shown, the process to find the key is shown.

After the ciphertext is copied, a program is utilized that is supplied by cryptoclub.org. The domain is shown below.

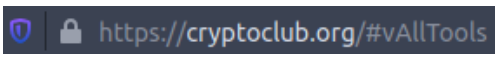

Figure 3: cryptoclub website

Below is the page you should see upon opening the URL in Figure 3

Figure 4: cryptoclub main page

The contents of the ciphertext are copied into the program as shown in Figure 5. On this page, the ciphertext shows the frequency of letters and relates the frequency of letters in the ciphertext to the frequency of letters in the English alphabet.

Figure 5: Crack Substitution Tool main page

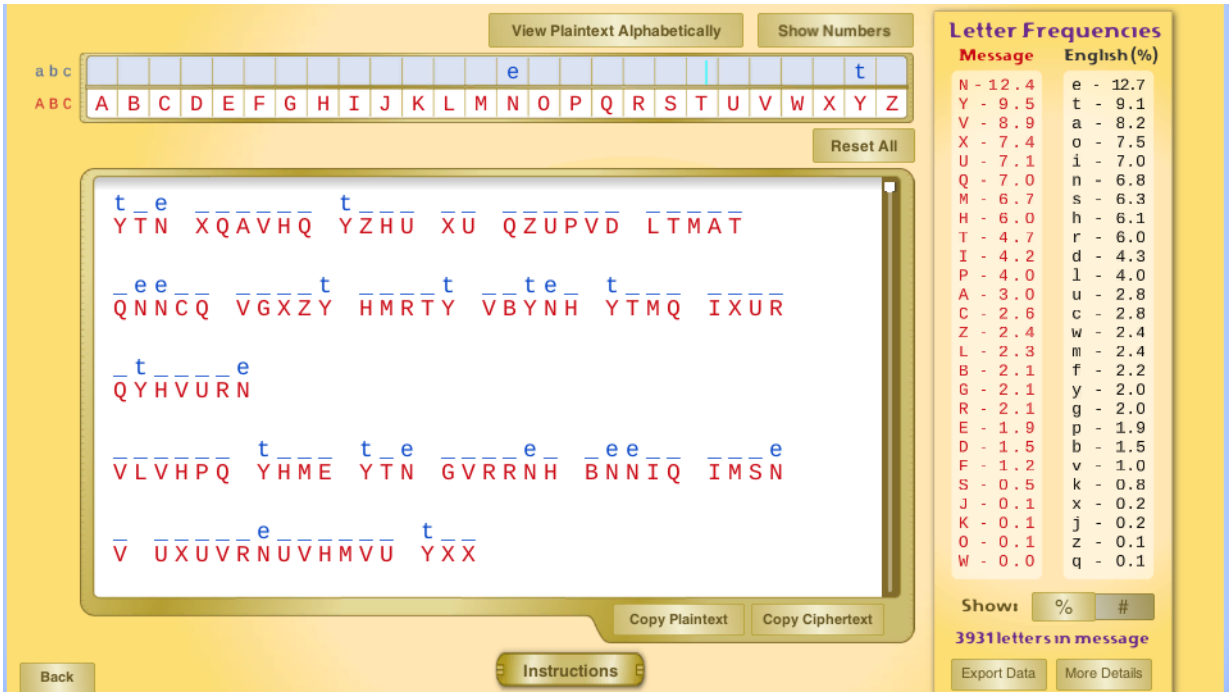

Figure 6: Mapping N to e and Y to t

In Figure 6 N is mapped to ‘e‘ and Y is mapped to ‘t‘ due to frequency analysis (You can see that N matches e and Y matches t in frequency on the right). It is very easy then to notice that T maps to ‘h‘ shown in Figure 7 to create the Trigram ‘the‘.

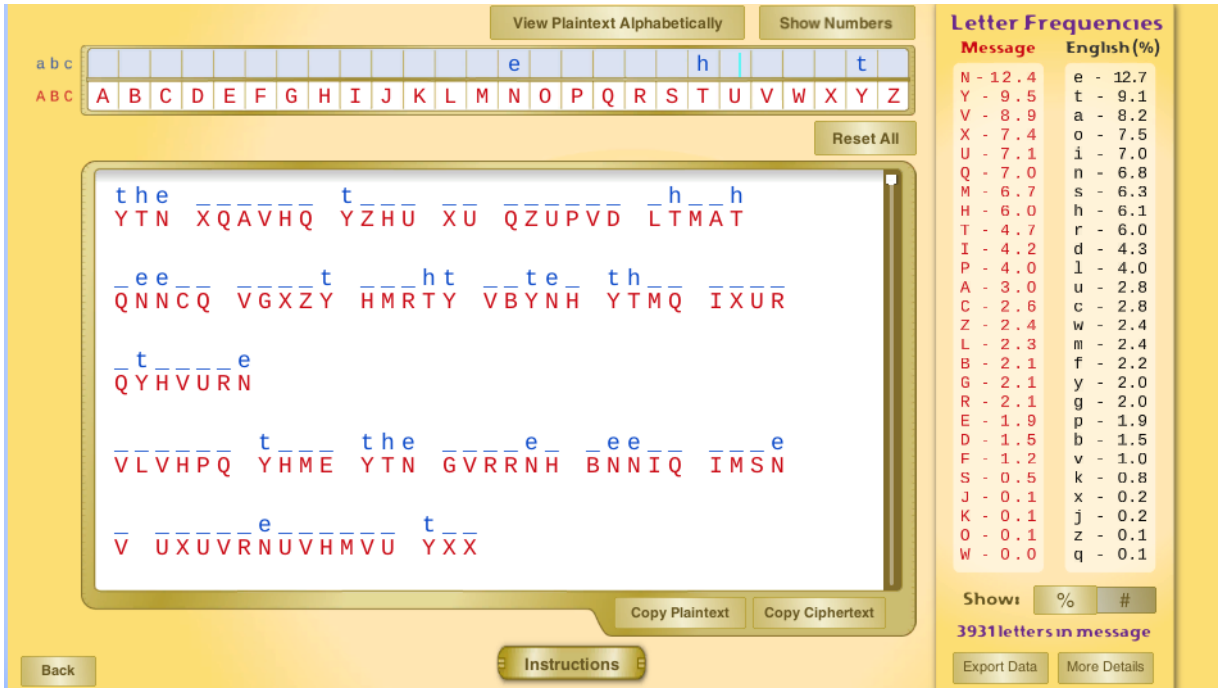

Figure 7: Mapping T to h

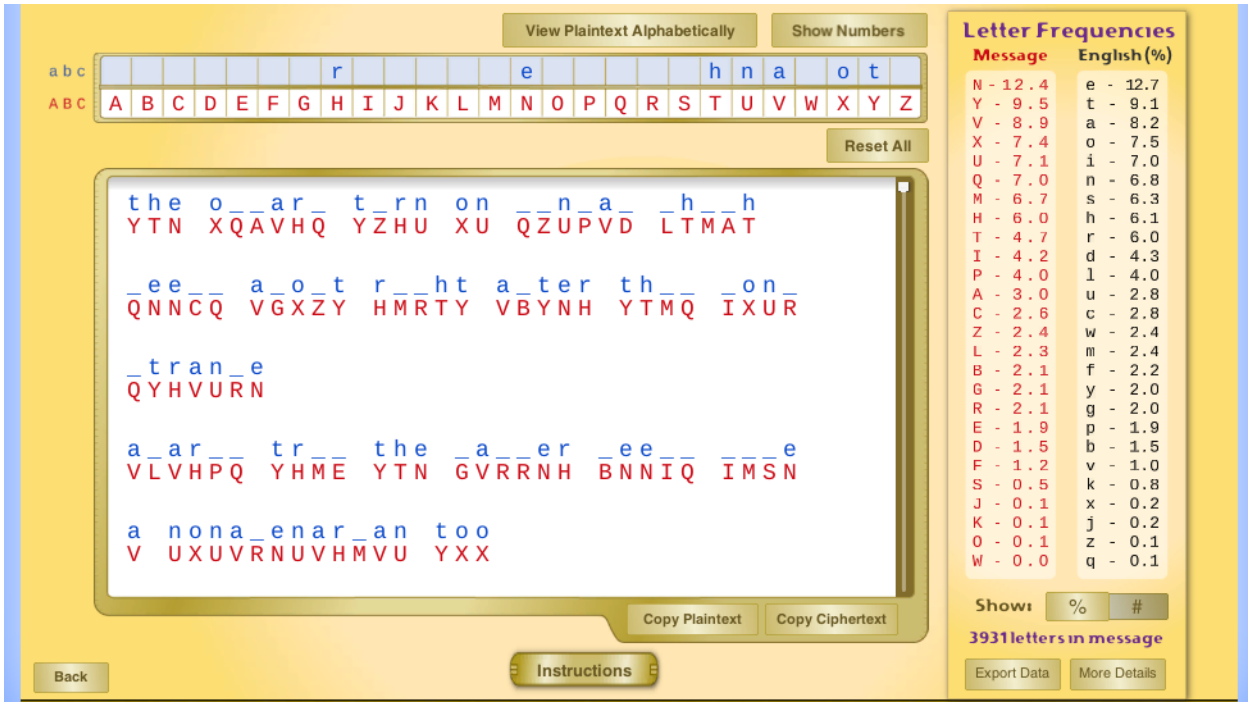

Figure 8: Mapping U to n, V to a, X to o, and H to r

Figure 8 shows how the ciphertext letters ‘UVXH‘ are mapped to ‘naor‘. This mapping was done by looking at the ciphertext frequency and mapping the frequency to the closest related

frequency in the english alphabet. Words do not particularly make sense yet, so it is just assumed for now that this is the correct mapping.

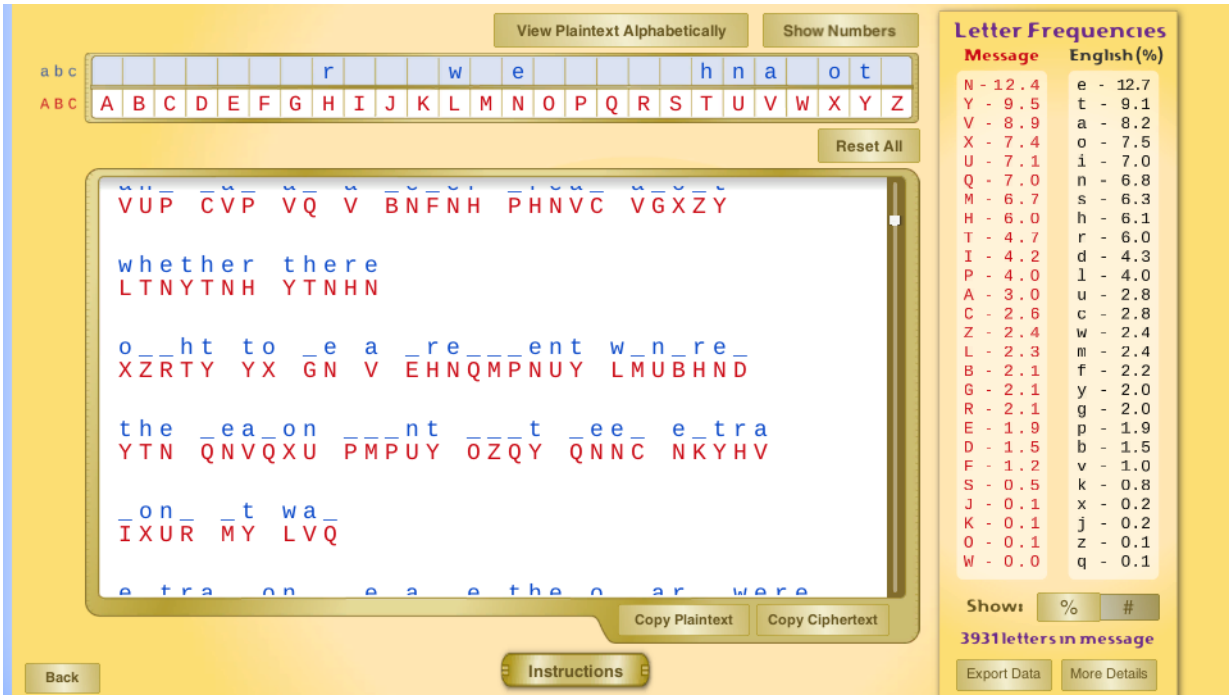

Figure 9: mapping L to w

In Figure 9 the string ‘hether‘ was shown, which clearly meant to spell ‘whether‘. Therefore, L maps to w.

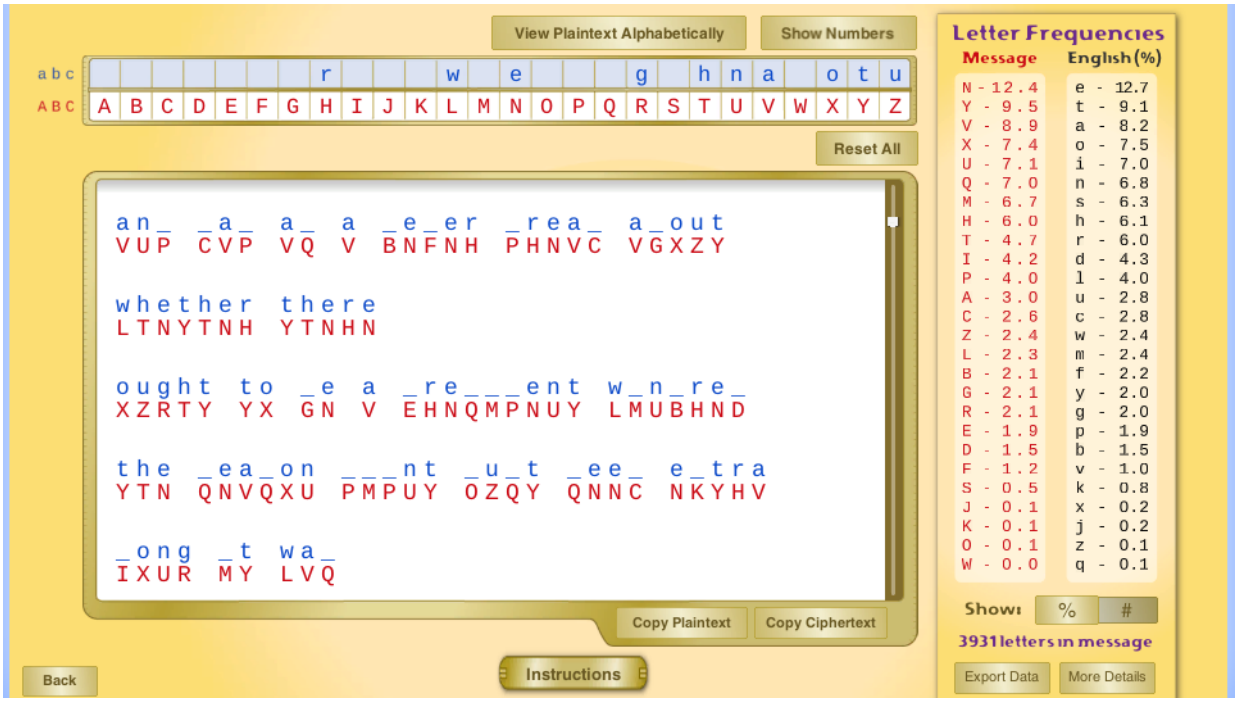

Figure 10: Mapping R to g

In Figure 10 the string 'ou-ht' is shown, which is meant to say 'ought'. Therefore, R maps to g.

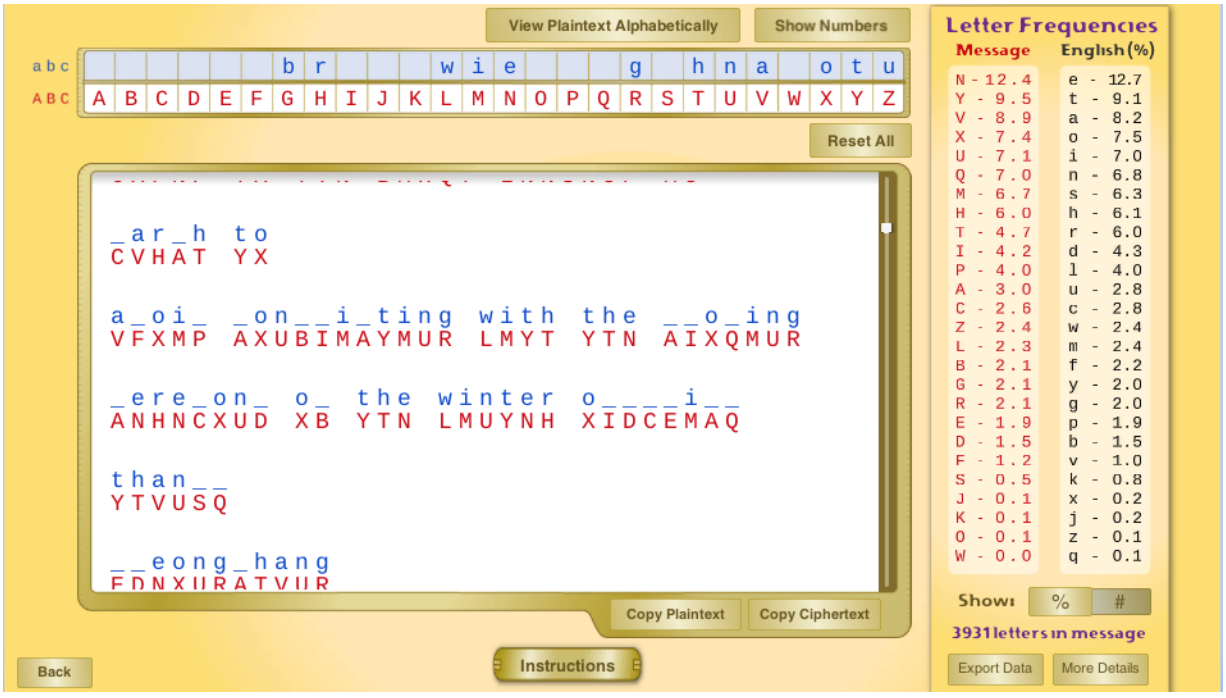

Figure 11: Mapping G to b and M to i

In Figure 11 The string 'w-nter' was shown, which should mean 'winter', Thus M maps to i. The mapping for G to b is not shown in this picture, but was done in a very similar fashion with another word.

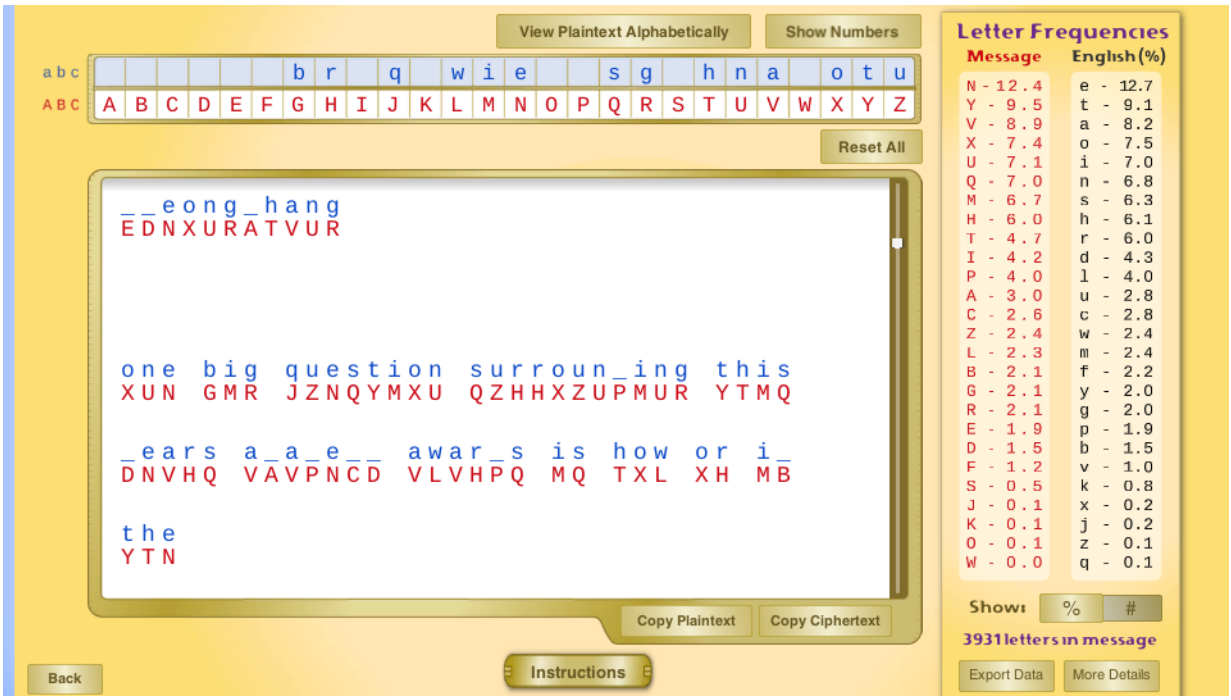

Figure 12: Mapping J to q and Q to s

In Figure 12 The string '-ue-tion' was present, which obviously meant 'question', therefore J maps to q Q maps to s.

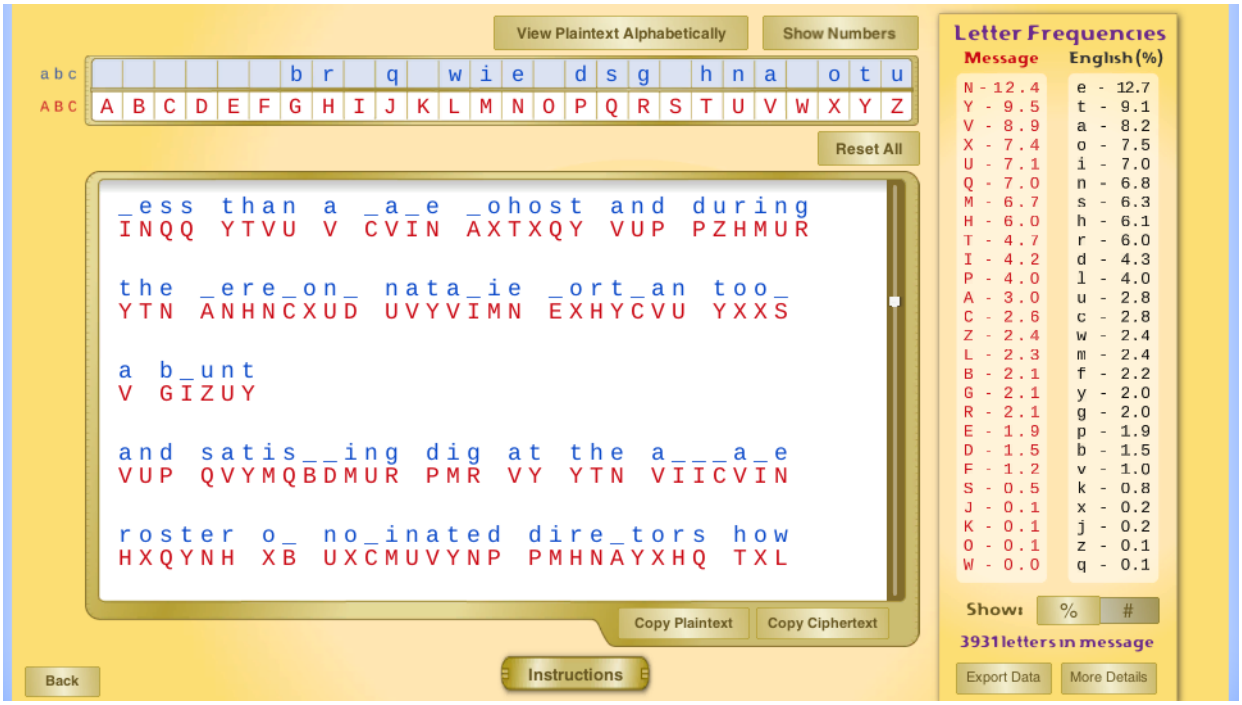

Figure 13: Mapping P to d

In Figure 13 The string 'an-' and '-ig' and '-uring' was present. Therefore, it is clear that P maps to d.

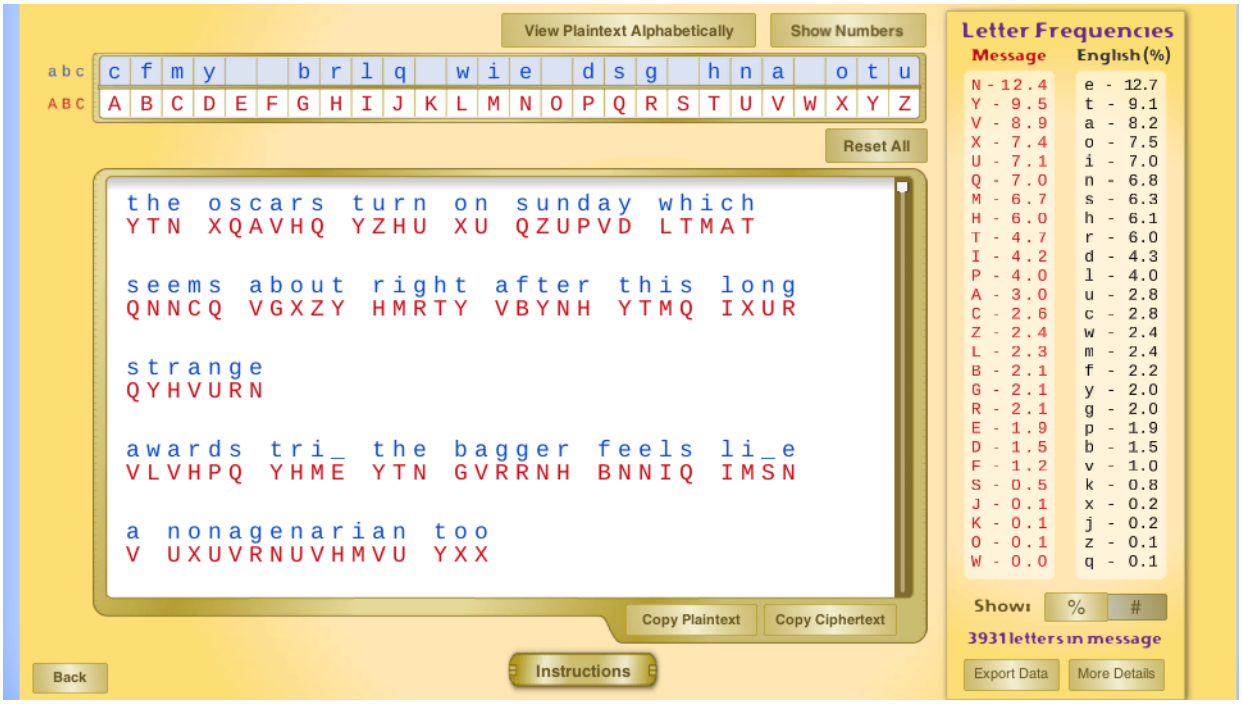

Figure 14: Mapping A to c, B to f, C to m, D to y, and I to l

In Figure 14 there are a lot of substitutions made. In particular, substitution A to c, B to f, C to m, D to y, and I to l. 'sunda-' was present which lead D to map to y, 'fee-s' which mapped I to l, 'os-ars' which mapped A to c, and the mapping from C to m is not present in Figure 14, but was done in a much similar fashion.

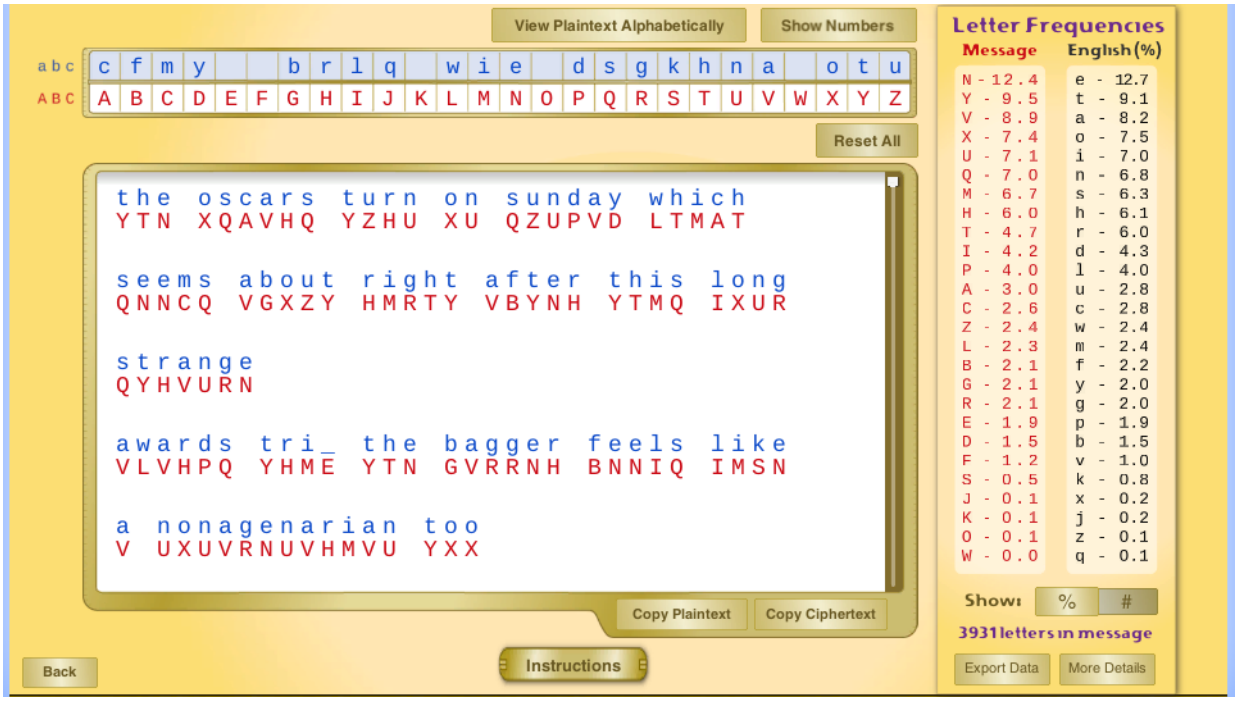

Figure 15: Mapping S to k

In figure 15 the mapping from S to k was done. This is due to the work 'li-e' being present, which is *likely* to be 'like'. Therefore, S maps to K. (get the pun)

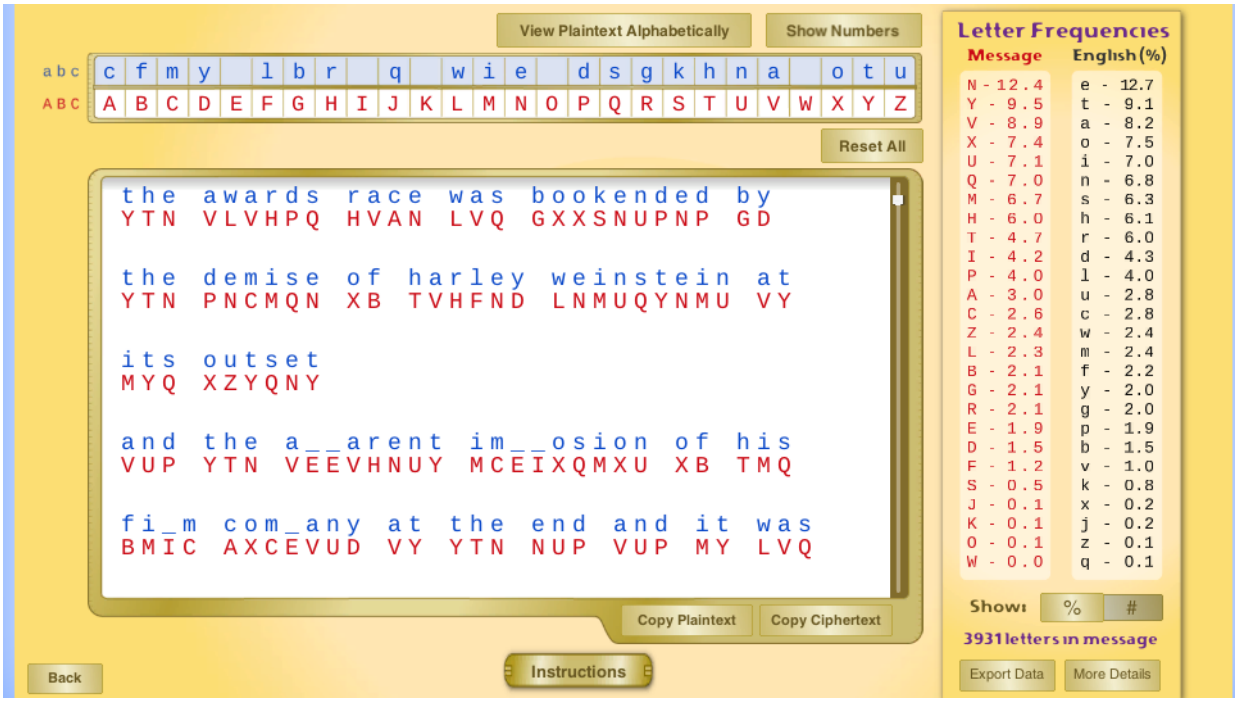

Figure 16: mapping F to l, removing I to l

In Figure 16 F is now mapped to l because the assumption was that the string 'harley' needed to be made. This removed the mapping from I to l.

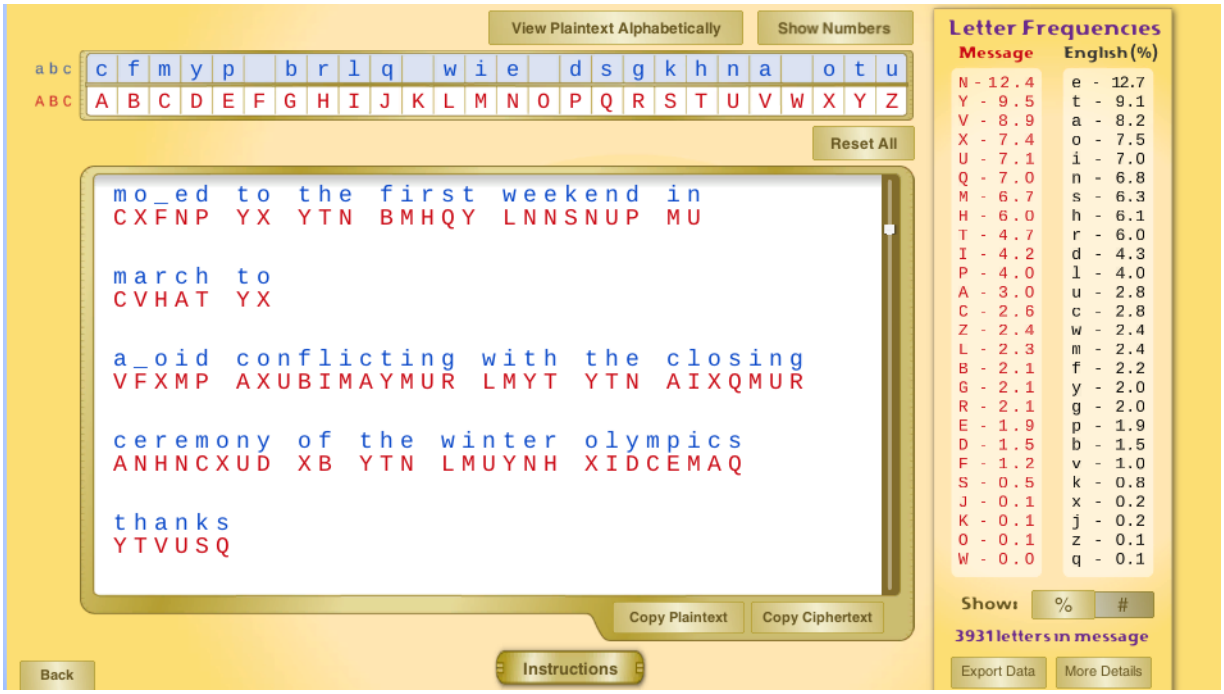

Figure 17: Mapping E to p, and re-mapping I to l

In Figure 17 E is mapped to p because 'olym-ics' was present, which definitely meains olympics. It was also noted that I should definitly be L because the string 'with the c-osing ceremony' was present.

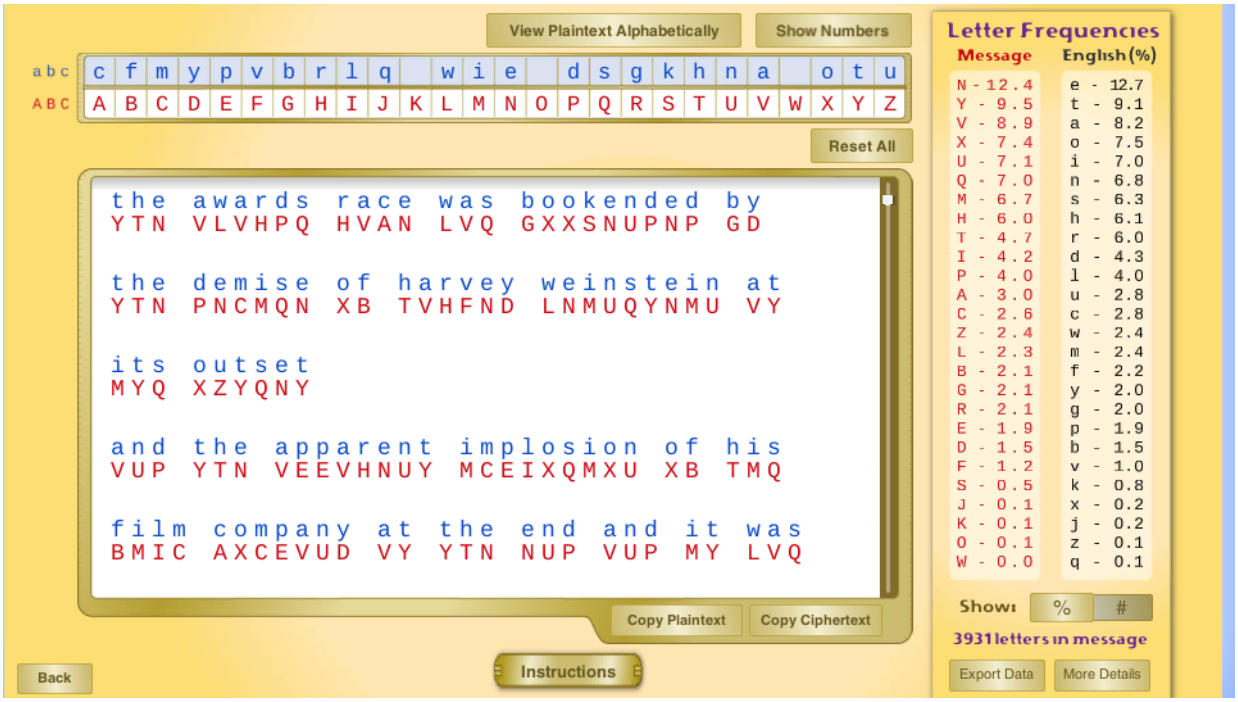

Figure 18: Mapping F to v

In Figure 18 it is clear that F should be mapped to v now, and that 'har-ey' should be 'harvey' to complete the name Harvery Weinstein.

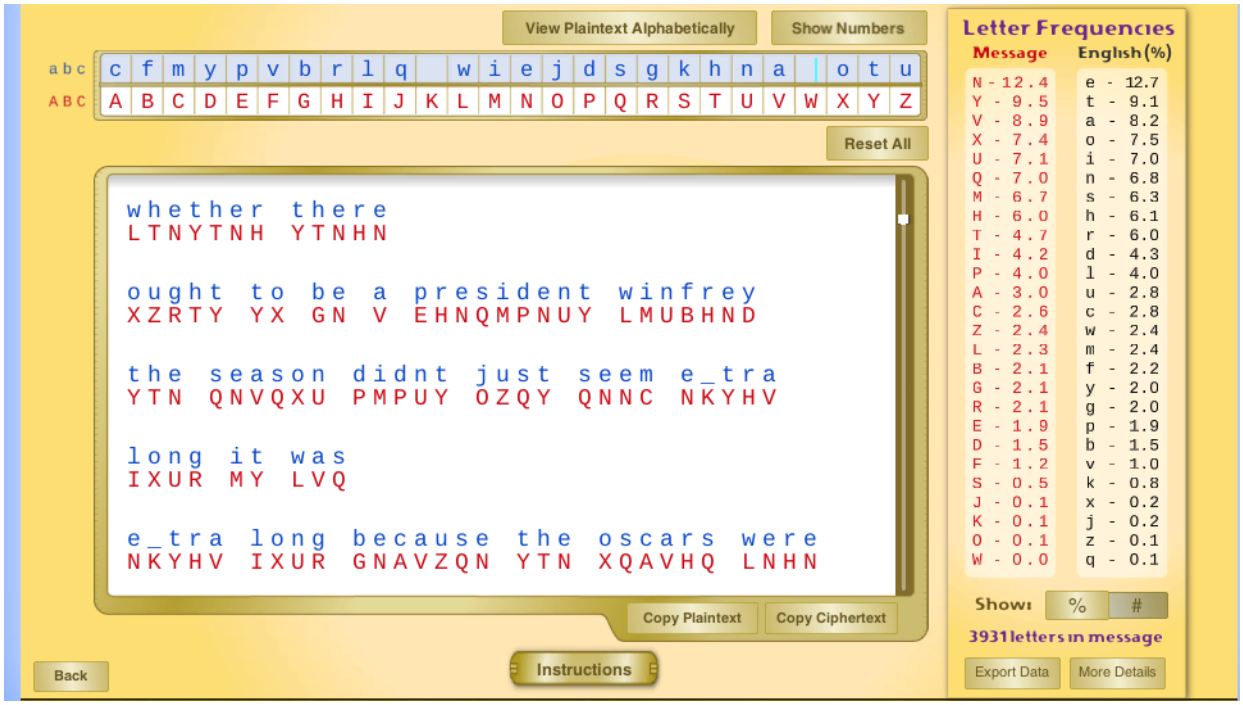

Figure 19: Mapping O to j

In Figure 19 it is clear that O should be mapped to j because the phrase 'the season didnt -ust seem' which obviously meant to say just.

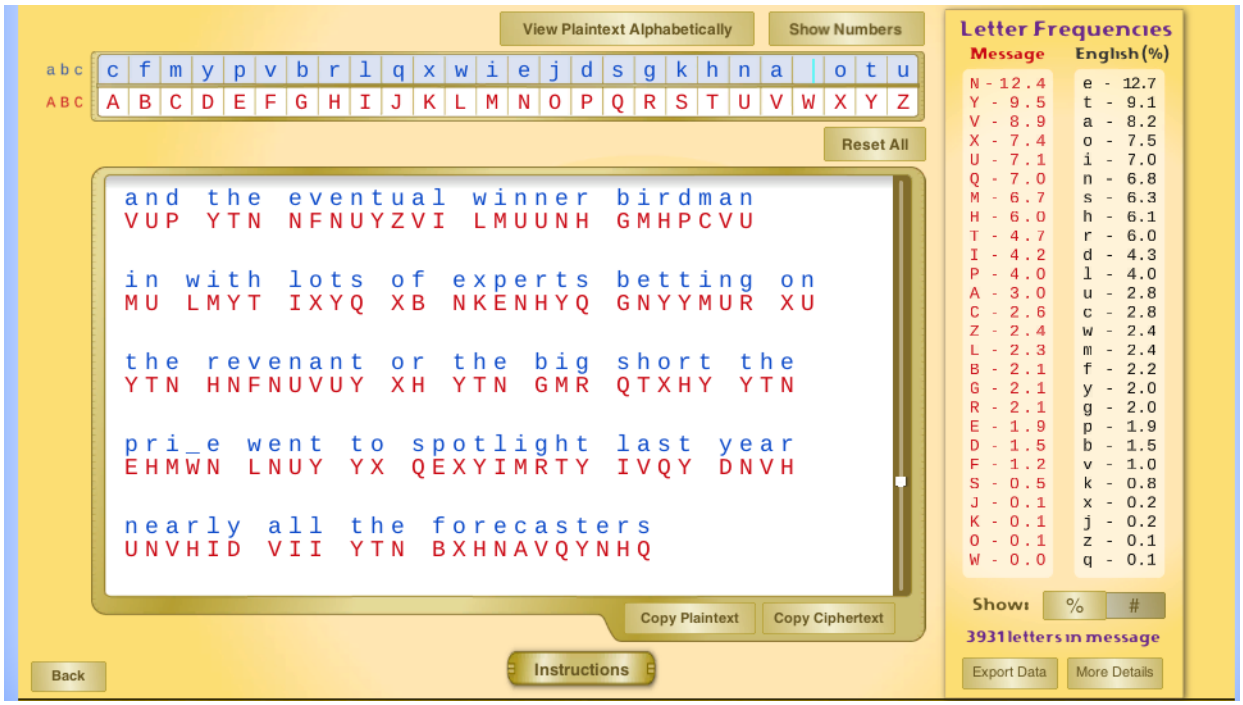

Figure 20: Mapping K to x

In Figure 20 it is clear that K should map x because the string 'e-perts' is present which should map to 'experts'.

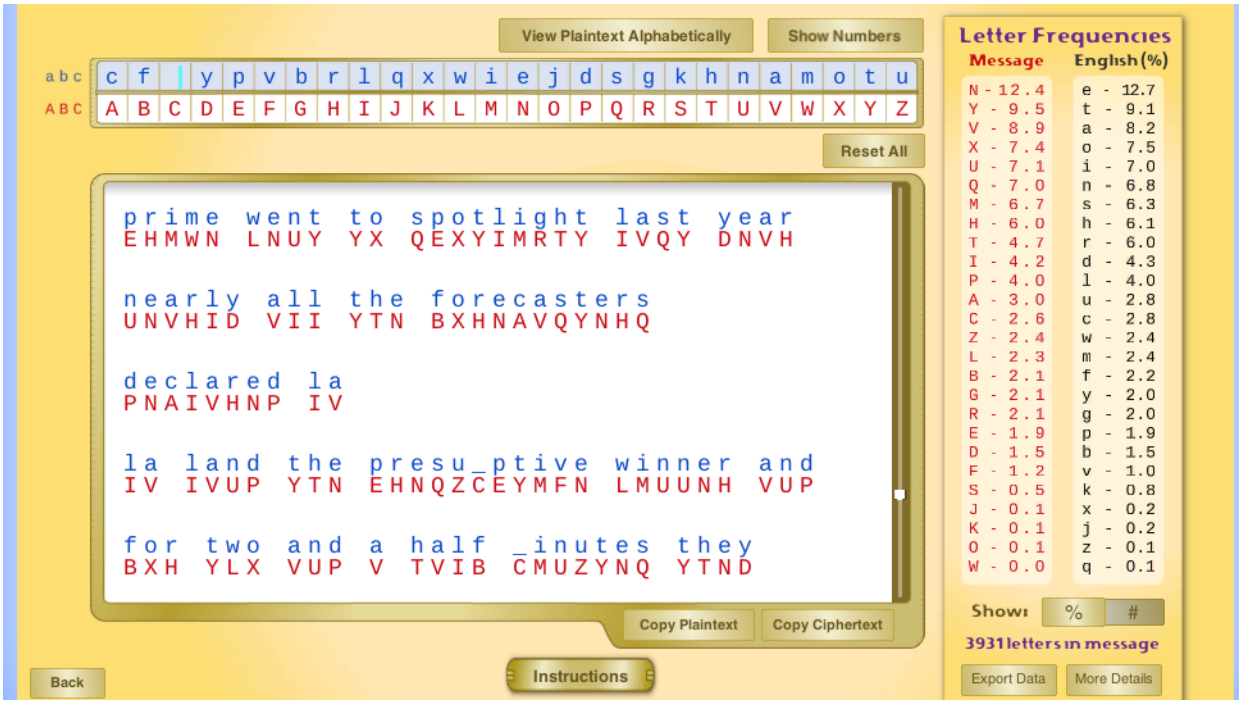

Figure 21: Mapping W to m, removing C to m

In Figure 21 it is clear that W should map to 'm' because the string 'pri-e' is present which is referring to 'prime'

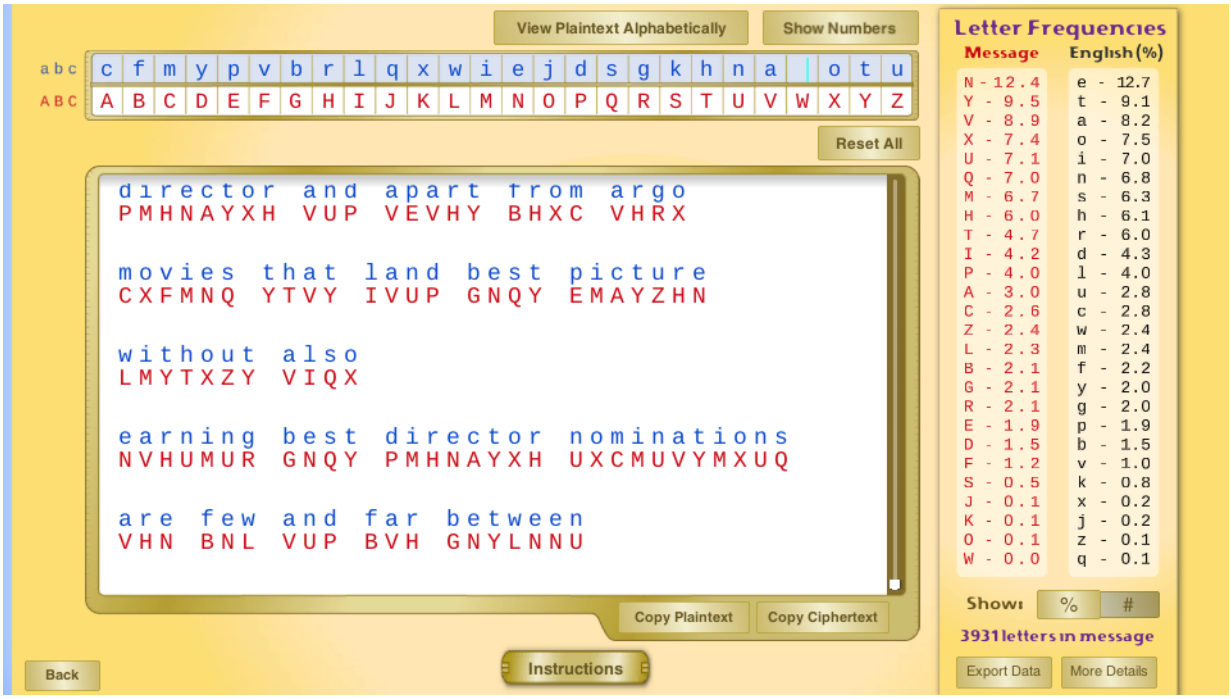

Figure 22: mapping C to m and removing W to m

In Figure 22 It is now really clear that mapping W to m was a mistake because the string 'fro- argo -ovies that' which maps to 'from argo movies that'.

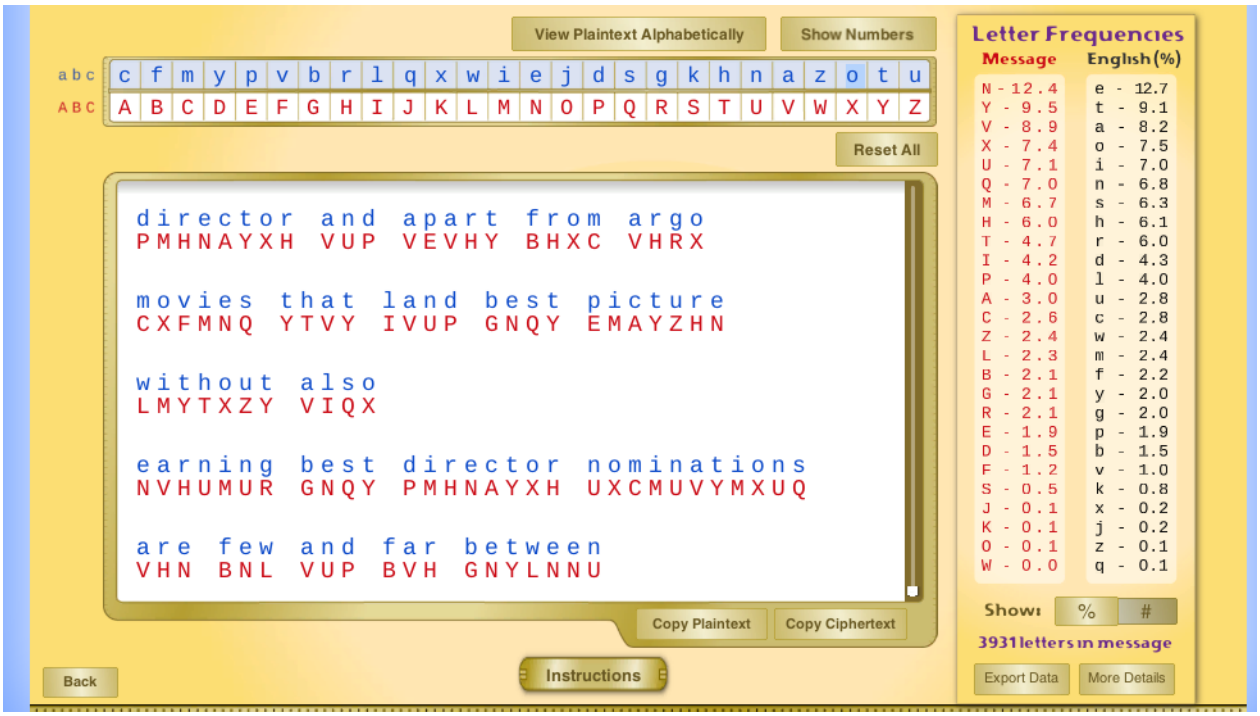

Figure 23: Mapping W to z

In Figure 23 W maps to z due to the process of elimination, because W does not show up, and neither does Z.

Finally, the key ‘abcdefghijklmnopqrstuvwxyz‘ to ‘cfmypvbrlqxwiejdsgkhnazotu‘ is obtained, where the first string ‘abcdefghijklmnopqrstuvwxyz‘ is the ciphertext letters.

2.2 Task 2: Encryption using Different Ciphers and Modes

In this task, we will play with various encryption algorithms and modes. You can use the following openssl enc command to encrypt/decrypt a file. To see the manuals, you can type man openssl and man enc.

```
openssl enc -ciphertype -e -in plain.txt -out cipher.bin \
    -K 00112233445566778889aabbccddeeff \
    -iv 0102030405060708
```

Please replace the ciphertype with a specific cipher type, such as -aes-128-cbc, -bf-cbc, -aes-128-cfb, etc. In this task, you should try at least 3 different ciphers. You can find the meaning of the command-line options and all the supported cipher types by typing "man enc". We include some common options for the openssl enc command in the following:

```
-in <file>      input file
-out <file>     output file
-e              encrypt
-d              decrypt
-K/-iv          key/iv in hex is the next argumc23ent
-[pP]           print the iv/key (then exit if -P)
```

2.2.1 Task2: solution

In order to complete this task, we must utilize the command man as shown in Figure 24 to find what all our possible encryption schemes are.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ man enc
```

Figure 24: using man on enc

```
base64              Base 64

bf-cbc              Blowfish in CBC mode
bf                  Alias for bf-cbc
blowfish            Alias for bf-cbc
bf-cfb              Blowfish in CFB mode
bf-ecb              Blowfish in ECB mode
bf-ofb              Blowfish in OFB mode

cast-cbc            CAST in CBC mode
cast                Alias for cast-cbc
cast5-cbc           CAST5 in CBC mode
cast5-cfb           CAST5 in CFB mode
cast5-ecb           CAST5 in ECB mode
cast5-ofb           CAST5 in OFB mode

chacha20            ChaCha20 algorithm

des-cbc             DES in CBC mode
des                 Alias for des-cbc
des-cfb             DES in CFB mode
des-ofb             DES in OFB mode
des-ecb             DES in ECB mode

des-ede-cbc         Two key triple DES EDE in CBC mode
des-ede             Two key triple DES EDE in ECB mode
des-ede-cfb         Two key triple DES EDE in CFB mode
des-ede-ofb         Two key triple DES EDE in OFB mode

des-ede3-cbc        Three key triple DES EDE in CBC mode
des-ede3            Three key triple DES EDE in ECB mode
des3                Alias for des-ede3-cbc
des-ede3-cfb        Three key triple DES EDE CFB mode
des-ede3-ofb        Three key triple DES EDE in OFB mode

desx                DESX algorithm.

gost89              GOST 28147-89 in CFB mode (provided by ccgost engine)
gost89-cnt         `GOST 28147-89 in CNT mode (provided by ccgost engine)

idea-cbc            IDEA algorithm in CBC mode
idea                same as idea-cbc
idea-cfb            IDEA in CFB mode
idea-ecb            IDEA in ECB mode
idea-ofb            IDEA in OFB mode
```

Figure 25: encryption schemes from enc using man part 1

```
rc2-cbc             128 bit RC2 in CBC mode
rc2                 Alias for rc2-cbc
rc2-cfb             128 bit RC2 in CFB mode
rc2-ecb             128 bit RC2 in ECB mode
rc2-ofb             128 bit RC2 in OFB mode
rc2-64-cbc          64 bit RC2 in CBC mode
rc2-40-cbc          40 bit RC2 in CBC mode

rc4                 128 bit RC4
rc4-64              64 bit RC4
rc4-40              40 bit RC4

rc5-cbc             RC5 cipher in CBC mode
rc5                 Alias for rc5-cbc
rc5-cfb             RC5 cipher in CFB mode
rc5-ecb             RC5 cipher in ECB mode
rc5-ofb             RC5 cipher in OFB mode

seed-cbc            SEED cipher in CBC mode
seed                Alias for seed-cbc
seed-cfb            SEED cipher in CFB mode
seed-ecb            SEED cipher in ECB mode
seed-ofb            SEED cipher in OFB mode

sm4-cbc             SM4 cipher in CBC mode
sm4                 Alias for sm4-cbc
sm4-cfb             SM4 cipher in CFB mode
sm4-ctr             SM4 cipher in CTR mode
sm4-ecb             SM4 cipher in ECB mode
sm4-ofb             SM4 cipher in OFB mode

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
```

Figure 26: encryption schemes from enc using man part 2

```
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
camellia-[128|192|256]-ofb  128/192/256 bit Camellia in OFB mode
```

Figure 27: encryption schemes from enc using man part 3

Figure 25, Figure 26 and Figure 27 show the results of the command shown in Figure 24 which is an impresive amount of encryption schemes that could be used, most of which have 3 levels of bits that can be used to effectively make the algorithm more resistant to certain attacks such as brute force attacks.

This task requires us to try at least 3 different ciphers on the ciphertext. Just for fun, the following ciphers are used:

1. bf-cbc
2. aria-192-ecb
3. camellia-192-ofb

First off is blowfish-cbc! There is no reason this was picked, it just sounds fun. blowfish is a symmetric-key block cipher which was designed in 1993. Blowfish was created as an alternative to the, at the time, aging DES standard. Blowfish became popular because most other ciphers were proprietary and required money and licensing to use. Blowfish was, and will be, public domain!

```
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ openssl enc -bf-cbc -e -in plain.txt -out
 cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat plain.txt
hello, I am plaintext!
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat cipher.bin && echo ""
F5o(v2jъ.BW
```

Figure 28: using bf-cbc to encrypt

Figure 28 shows the results of encrypting a simple plaintext file using bf-cbc (blowfish-cbc), and the resulting cipher.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat cipher.bin && echo ""
F5o(v2jb]BW
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ openssl enc -bf-cbc -d -in cipher.bin -ou
t out.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat out.txt
hello, I am plaintext!
```

Figure 29: using bf-cbc to decrypt

Figure 29 shows the process of decrypting blowfish-cbc using openssl. It is extremely similar to encrypting, the only difference is swapping the in/out file, and adding -d instead of -e. Just for fun, the time of encrypting and decrypting the program was done using the linux time command, which is shown below.

```
time openssl enc -bf-cbc -d -in cipher.bin -out out.txt \
    -K 00112233445566778889aabbccddeeff \
    -iv 0102030405060708
```

Encryption time: 0.004s
Decryption Time: 0.004s

In the end, blowfish-cbc is a very fun algorithm with a fun background. It's a shame it's not seen as much.

Now on to aria-192-ecb!

The process for aria-192-ecb is *extremely* similar to blowfish-cbc as shown in Figure 28. Therefore, for this section no pictures will be shown, but rather the commands will be simply printed as text.

```
$ openssl enc -aria-192-ecb -d -in plain.txt -out aria-192-ecb_cipher.bin \
      -K 00112233445566778889aabbccddeeff \
      -iv 0102030405060708
warning: iv not use by this cipher
hex string is too short, padding with zero bytes to length
```

Here, openssl outputs some interesting comments on the program, stating no IV is used, and the hex string is too short. Let's try that again

```
$ openssl enc -aria-192-ecb -d -in plain.txt -out aria-192-ecb_cipher.bin \
      -K 00112233445566778889aabbccddeeffeeffeeffeeffeeff
$ cat plain.txt
hello, I am plaintext!
$ cat aria-192-ecb_cipher.bin
6&eI-R^:'&
```

There we go, that is much more like it!

And of course, the process to decrypt is very similar to the encryption process.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat aria-192-ecb_cipher.bin && echo ""
[illegible]
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ time openssl enc -aria-192-ecb -d -in ari
a-192-ecb_cipher.bin -out out.txt -K 00112233445566778889aabbccddeeffeeffeeffeeffeeff

real    0m0.006s
user    0m0.005s
sys     0m0.001s
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat out.txt
hello, I am plaintext!
```

Figure 30: using aria-192-ecb to decrypt

Figure 30 shows the process of decrypting with aria-192-ecb. time was utilized again in order to determine how long aria-192-ecb takes.

Encryption Time: 0.010s
Decryption Time: 0.006s

Now for the final algorithm to check out, camellia-192-ofb!

```
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ openssl enc -camellia-192-ofb -e -in plai
n.txt -out camellia-256-ofb_cipher.bin -K 00112233445566778889aabbccddeeffeeffeeffeeffeeff -iv
 0102030405060708123413212343214 5
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat plain.txt
hello, I am plaintext!
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat camellia-256-ofb_cipher.bin && echo "
"
[illegible]
[illegible]
```

Figure 31: using camellia-256-ofb to encrypt

Figure 31 shows the process of encrypting with camellia-192-ofb.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ openssl enc -camellia-192-ofb -d -in came
llia-256-ofb_cipher.bin -out out.txt -K 00112233445566778889aabbccddeeffeeffeeffeeffeeff -iv 0
102030405060708123413212343214 5
mitch@light: ~/git/ECE471/lab1/task2-6/t2 (master) $ cat out.txt
hello, I am plaintext!
```

Figure 32: using camellia-256-ofb to decrypt

Figure 32 shows the process of decrypting with camellia-192-ofb. And of course, ztime was used for both. The results are shown below.

Encrypting: 0.003s
Decrypting: 0.008s

2.3 Task 3: Encryption Mode – ECB vs. CBC

The file *pic_original.bmp* can be downloaded from my github `github.com/mitchdz/ECE471`, and it contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. How- ever, For the .bmp file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate .bmp file. We will replace the header of the encrypted picture with that of the original picture. We can use the bless hex editor tool (already installed on our VM) to directly modify binary files. We can also use the following commands to get the header from p1.bmp, the data from p2.bmp (from offset 55 to the end of the file), and then combine the header and data together into a new file.

   ```
   $ head -c 54 p1.bmp > header
   $ tail -c +55 p2.bmp > body
   $ cat header body > new.bmp
   ```

2. Display the encrypted picture using a picture viewing program (we have installed an image viewer program called eog on our VM). Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

Select a picture of your choice, repeat the experiment above, and report your observations.

2.3.1 Task 3: Solution

Figure 33: Original image that is being encrypted

Figure 33 shows the original file. The file is a red oval in the center, with a teal square on the bottom right that has a blue border.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ openssl enc -aes-128-cbc -e -in pic_origi
nal.bmp -out cbc_cipher.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708123412341
341212
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ head -c 54 pic_original.bmp > header
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ tail -c +55 cbc_cipher.bmp > body
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ cat header body > cbc_cipher.bmp
```

Figure 34: using cbc to encrypt image

Figure 34 shows the commands used in order to encrypt he image using cbc.

Figure 35: results of cbc on image

Figure 35 shows the results of encrypting the image using cbc. There is not much data that is able to be extracted from this image.

Similar to above, ecb (electronic codebook is now used)

```
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ openssl enc -aes-128-ecb -e -in pic_origi
nal.bmp -out ecb_cipher.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708123412341
341212
warning: iv not use by this cipher
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ head -c 54 pic_original.bmp > header
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ tail -c +55 ecb_cipher.bmp > body
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ cat header body > ecb_cipher.bmp
```

Figure 36: encrypting image using ECB

Figure 36 shows the command used to encrypt the image using ECB. Note the comment saying that the iv is not used by this cipher. Unlike CBC, EBC does not need an initial value.

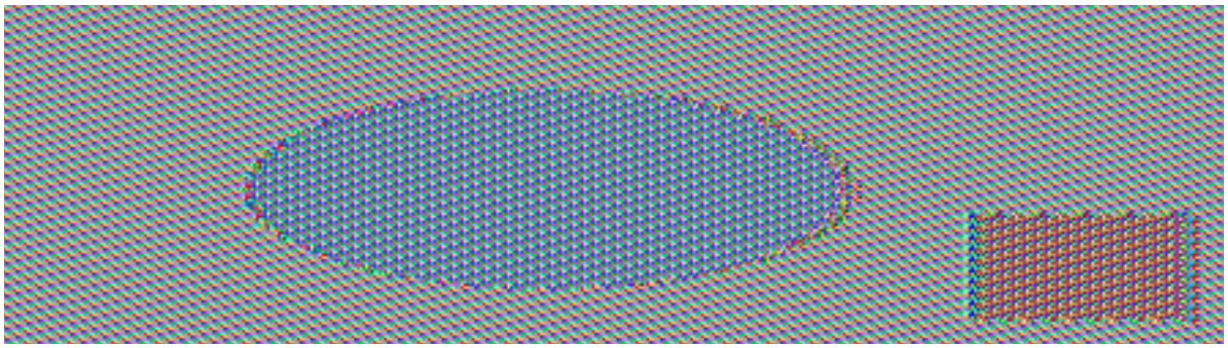

Figure 37: results of ECB on image

Figure 37 shows the resulting ciphertext after encrypting the image with ECB. It is very easy to identify what the original image is. Although some details such as the blue border are gone, and the colors of the oval and the square have changed. If this was a document of a person, the person would most likely be easily identifiable.

Now it is time to try this experiment on my own image!

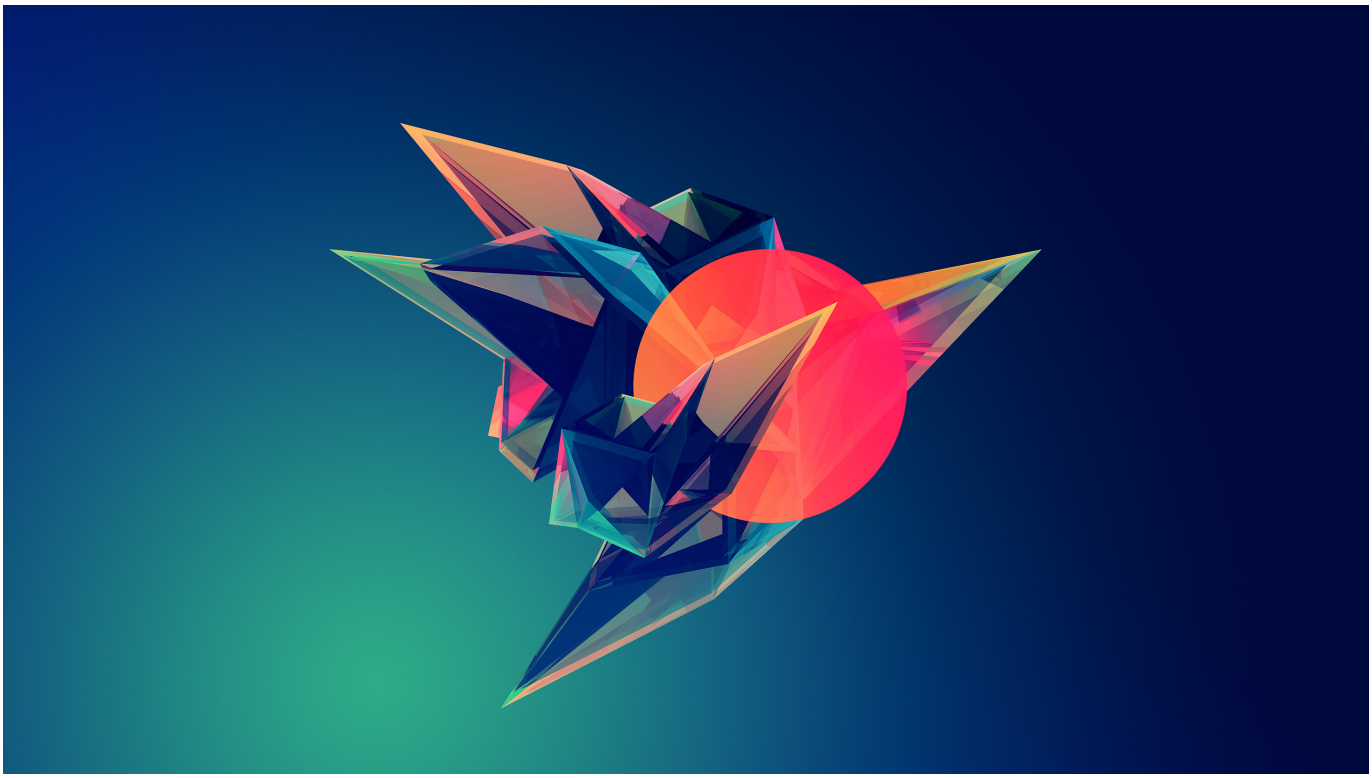

Figure 38: Personal image to be encrypted

Figure 38 shows the personal image used to do this experiment on. This image is the background that I use for my laptop.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ openssl enc -aes-128-cbc -e -in t3_person
al_image.bmp -out personal_cbc_cipher.bmp -K 00112233445566778889aabbccddeeff
iv undefined
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ head -c 54 t3_personal_image.bmp > header
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ tail -c +55 personal_cbc_cipher.bmp > bod
y
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ cat header body > t3_personal_cbc.bmp
```

Figure 39: Encrypting personal image using CBC

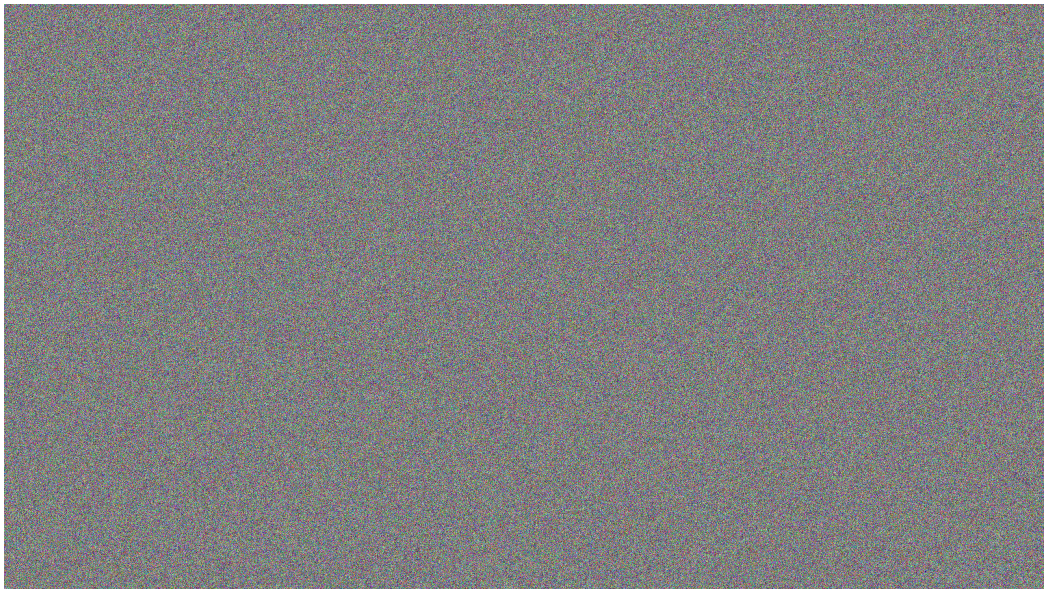

Figure 40: results of encrypting Figure 38 with CBC

Figure 40 shows the results of encrypting Figure 38 with CBC. The result is, as expected, seemingly random. There is nothing to be able to determine about the picture from viewing this image.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ openssl enc -aes-128-ecb -e -in t3_person
al_image.bmp -out personal_ecb_cipher.bmp -K 00112233445566778889aabbccddeeff
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ head -c 54 t3_personal_image.bmp > header
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ tail -c +55 personal_ecb_cipher.bmp > bod
y
mitch@light: ~/git/ECE471/lab1/task2-6/t3 (master) $ cat header body > new.bmp
```

Figure 41: Encrypting personal image using ECB

Figure 41 shows the process of encrypting the personal image using ECB. This process is very similar to Figure 36 and Figure 34.

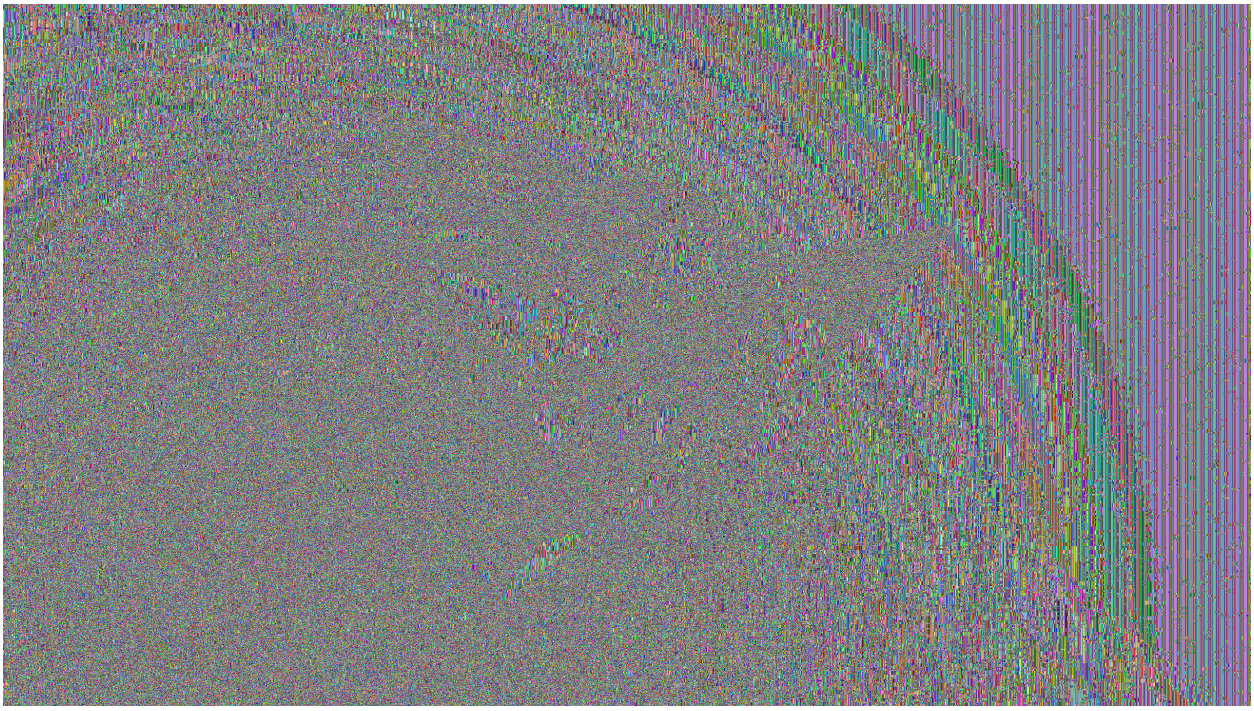

Figure 42: Result of encrypting with ECB

Figure 42 shows the results of encrypting Figure 38 with ECB, which honestly is not too easy to identify what the image should be. This leads me to believe that ECB only reveals the contents via inspection if there is a large enough contrast between objects and colors.

2.4 Task 4: Padding

For block ciphers, when the size of a plaintext is not a multiple of the block size, padding may be required. All the block ciphers normally use PKCS5 padding, which is known as standard block padding. We will conduct the following experiments to understand how this type of padding works:

1. Use ECB, CBC, CFB, and OFB modes to encrypt a file (you can pick any cipher). Please report which modes have paddings and which ones do not. For those that do not need paddings, please explain why.

2. Let us create three files, which contain 5 bytes, 10 bytes, and 16 bytes, respectively. We can use the following "echo -n" command to create such files. The following example creates a file f1.txt with length 5 (without the -n option, the length will be 6, because a newline character will be added by echo):

   ```
   $ echo -n "12345" > f1.txt
   ```

   We then use "openssl enc -aes-128-cbc -e" to encrypt these three files using 128-bit AES with CBC mode. Please describe the size of the encrypted files.

   We would like to see what is added to the padding during the encryption. To achieve this goal, we will decrypt these files using "openssl enc -aes-128-cbc -d". Unfortunately, decryption by default will automatically remove the padding, making it impossible for us to see the padding. However, the command does have an option called "-nopad", which disables the padding, i.e., during the decryption, the command will not remove the padded data. Therefore, by looking at the decrypted data, we can see what data are used in the padding. Please use this technique to figure out what paddings are added to the three files.

   It should be noted that padding data may not be printable, so you need to use a hex tool to display the content. The following example shows how to display a file in the hex format:

   ```
   $ hexdump -C p1.txt
   00000000 31 32 33 34 35 36 37 38 39 49 4a 4b 4c 0a  |123456789IJKL.|
   $ xxd p1.txt
   00000000: 3132 3334 3536 3738 3949 4a4b 4c0a         123456789IJKL.
   ```

2.4.1 Task 4: Solution

```
mitch@light: ~/git/ECE471/lab1/task2-6/t4 (master) $ cat f1.txt && echo ""
12345
mitch@light: ~/git/ECE471/lab1/task2-6/t4 (master) $ cat f2.txt && echo ""
1234512345
mitch@light: ~/git/ECE471/lab1/task2-6/t4 (master) $ cat f3.txt && echo ""
1234512345123451
mitch@light: ~/git/ECE471/lab1/task2-6/t4 (master) $ ls -l
total 12
-rw-rw-r-- 1 mitch mitch  5 Feb  3 20:18 f1.txt
-rw-rw-r-- 1 mitch mitch 10 Feb  3 20:19 f2.txt
-rw-rw-r-- 1 mitch mitch 16 Feb  3 20:20 f3.txt
```

Figure 43: task 4 test files

Figure 43 shows the test files being used in this section. The files are very simple, and the linux command used shows the size of the files. f1.txt is 5 bytes, f2.txt is 10 bytes, and f3.txt is 16 bytes. Below is what the name of each of the encryption protocols we are using expands to.

ECB - Electronic CodeBook
CBC - Cipher-Block-Chaining
CFB - Cipher-Feedback
OFB - Output-FeedBack

To these three files, we need to encrypt and decrypt the file using ECB, CBC, CFB, and OFB. Because this is a lot of similar commands being utilized, a script was made to run this session. The script is copied below, and can be found in the most current state at `https://github.com/mitchdz/ECE471/blob/master/lab1/task2-6/t4/t4_commands.sh`.

```
#!/usr/bin/bash
KEY="00112233445566778889aabbccddeeff"
IV="01020304050607081234123412341212"
ENC_PREF="openssl enc -aes-128-"

CIPHER_PREF="cipher"
PLAIN_PREF="plain"
mkdir -p ${CIPHER_PREF}/
mkdir -p ${PLAIN_PREF}/

#encrypting
${ENC_PREF}ecb -e -in f1.txt -out ${CIPHER_PREF}/f1_ecb_cipher.txt -K ${KEY}
${ENC_PREF}ecb -e -in f2.txt -out ${CIPHER_PREF}/f2_ecb_cipher.txt -K ${KEY}
${ENC_PREF}ecb -e -in f3.txt -out ${CIPHER_PREF}/f3_ecb_cipher.txt -K ${KEY}
${ENC_PREF}cbc -e -in f1.txt -out ${CIPHER_PREF}/f1_cbc_cipher.txt -K ${KEY} \
    -iv ${IV}
```

```
${ENC_PREF}cbc -e -in f2.txt -out ${CIPHER_PREF}/f2_cbc_cipher.txt -K ${KEY} \
    -iv ${IV}
${ENC_PREF}cbc -e -in f3.txt -out ${CIPHER_PREF}/f3_cbc_cipher.txt -K ${KEY} \
    -iv ${IV}
${ENC_PREF}cfb -e -in f1.txt -out ${CIPHER_PREF}/f1_cfb_cipher.txt -K ${KEY} \
    -iv ${IV}
${ENC_PREF}cfb -e -in f2.txt -out ${CIPHER_PREF}/f2_cfb_cipher.txt -K ${KEY} \
    -iv ${IV}
${ENC_PREF}cfb -e -in f3.txt -out ${CIPHER_PREF}/f3_cfb_cipher.txt -K ${KEY} \
    -iv ${IV}
${ENC_PREF}ofb -e -in f1.txt -out ${CIPHER_PREF}/f1_ofb_cipher.txt -K ${KEY} \
    -iv ${IV}
${ENC_PREF}ofb -e -in f2.txt -out ${CIPHER_PREF}/f2_ofb_cipher.txt -K ${KEY} \
    -iv ${IV}
${ENC_PREF}ofb -e -in f3.txt -out ${CIPHER_PREF}/f3_ofb_cipher.txt -K ${KEY} \
    -iv ${IV}
#decrypting
${ENC_PREF}ecb -d -nopad -in ${CIPHER_PREF}/f1_ecb_cipher.txt \
    -out ${PLAIN_PREF}/f1_ecb_plain.txt -K ${KEY}
${ENC_PREF}ecb -d -nopad -in ${CIPHER_PREF}/f1_cbc_cipher.txt \
    -out ${PLAIN_PREF}/f1_cbc_plain.txt -K ${KEY}
${ENC_PREF}cfb -d -nopad -in ${CIPHER_PREF}/f1_cfb_cipher.txt \
    -out ${PLAIN_PREF}/f1_cfb_plain.txt -K ${KEY} -iv ${IV}
${ENC_PREF}ofb -d -nopad -in ${CIPHER_PREF}/f1_ofb_cipher.txt \
    -out ${PLAIN_PREF}/f1_ofb_plain.txt -K ${KEY} -iv ${IV}

${ENC_PREF}ecb -d -nopad -in ${CIPHER_PREF}/f2_ecb_cipher.txt \
    -out ${PLAIN_PREF}/f2_ecb_plain.txt -K ${KEY}
${ENC_PREF}cbc -d -nopad -in ${CIPHER_PREF}/f2_cbc_cipher.txt \
    -out ${PLAIN_PREF}/f2_cbc_plain.txt -K ${KEY} -iv ${IV}
${ENC_PREF}cfb -d -nopad -in ${CIPHER_PREF}/f2_cfb_cipher.txt \
    -out ${PLAIN_PREF}/f2_cfb_plain.txt -K ${KEY} -iv ${IV}
${ENC_PREF}ofb -d -nopad -in ${CIPHER_PREF}/f2_ofb_cipher.txt \
    -out ${PLAIN_PREF}/f2_ofb_plain.txt -K ${KEY} -iv ${IV}

${ENC_PREF}ecb -d -nopad -in ${CIPHER_PREF}/f3_ecb_cipher.txt \
    -out ${PLAIN_PREF}/f3_ecb_plain.txt -K ${KEY}
${ENC_PREF}cbc -d -nopad -in ${CIPHER_PREF}/f3_cbc_cipher.txt \
    -out ${PLAIN_PREF}/f3_cbc_plain.txt -K ${KEY} -iv ${IV}
${ENC_PREF}cfb -d -nopad -in ${CIPHER_PREF}/f3_cfb_cipher.txt \
    -out ${PLAIN_PREF}/f3_cfb_plain.txt -K ${KEY} -iv ${IV}
${ENC_PREF}ofb -d -nopad -in ${CIPHER_PREF}/f3_ofb_cipher.txt \
    -out ${PLAIN_PREF}/f3_ofb_plain.txt -K ${KEY} -iv ${IV}

#printing out results
```

```
echo "f1.txt"
printf "plain: "; xxd f1.txt
printf "ecb  : "; xxd plain/f1_ecb_plain.txt
printf "cbc  : "; xxd plain/f1_cbc_plain.txt
printf "cfb  : "; xxd plain/f1_cfb_plain.txt
printf "ofb  : "; xxd plain/f1_ofb_plain.txt

echo ""
echo "f2.txt"
printf "plain: "; xxd f2.txt
printf "ecb  : "; xxd plain/f2_ecb_plain.txt
printf "cbc  : "; xxd plain/f2_cbc_plain.txt
printf "cfb  : "; xxd plain/f2_cfb_plain.txt
printf "ofb  : "; xxd plain/f2_ofb_plain.txt

echo ""
echo "f3.txt"
printf "plain: "; xxd f3.txt
printf "ecb  : "; xxd plain/f3_ecb_plain.txt
printf "cbc  : "; xxd plain/f3_cbc_plain.txt
printf "cfb  : "; xxd plain/f3_cfb_plain.txt
printf "ofb  : "; xxd plain/f3_ofb_plain.txt
```

```
mitch@light: ~/git/ECE471/lab1/task2-6/t4 (master) $ ./t4_commands.sh
mitch@light: ~/git/ECE471/lab1/task2-6/t4 (master) $ ls -l
total 116
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f1_cbc_cipher.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f1_cbc_plain.txt
-rw-rw-r-- 1 mitch mitch    5 Feb  4 20:47 f1_cfb_cipher.txt
-rw-rw-r-- 1 mitch mitch    5 Feb  4 20:47 f1_cfb_plain.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f1_ecb_cipher.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f1_ecb_plain.txt
-rw-rw-r-- 1 mitch mitch    5 Feb  4 20:47 f1_ofb_cipher.txt
-rw-rw-r-- 1 mitch mitch    5 Feb  4 20:47 f1_ofb_plain.txt
-rw-rw-r-- 1 mitch mitch    5 Feb  3 20:18 f1.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f2_cbc_cipher.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f2_cbc_plain.txt
-rw-rw-r-- 1 mitch mitch   10 Feb  4 20:47 f2_cfb_cipher.txt
-rw-rw-r-- 1 mitch mitch   10 Feb  4 20:47 f2_cfb_plain.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f2_ecb_cipher.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f2_ecb_plain.txt
-rw-rw-r-- 1 mitch mitch   10 Feb  4 20:47 f2_ofb_cipher.txt
-rw-rw-r-- 1 mitch mitch   10 Feb  4 20:47 f2_ofb_plain.txt
-rw-rw-r-- 1 mitch mitch   10 Feb  3 20:19 f2.txt
-rw-rw-r-- 1 mitch mitch   32 Feb  4 20:47 f3_cbc_cipher.txt
-rw-rw-r-- 1 mitch mitch   32 Feb  4 20:47 f3_cbc_plain.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f3_cfb_cipher.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f3_cfb_plain.txt
-rw-rw-r-- 1 mitch mitch   32 Feb  4 20:47 f3_ecb_cipher.txt
-rw-rw-r-- 1 mitch mitch   32 Feb  4 20:47 f3_ecb_plain.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f3_ofb_cipher.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  4 20:47 f3_ofb_plain.txt
-rw-rw-r-- 1 mitch mitch   16 Feb  3 20:20 f3.txt
drwxrwxr-x 2 mitch mitch 4096 Feb  4 20:46 pictures
-rwxrwxr-x 1 mitch mitch 2057 Feb  4 20:39 t4_commands.sh
```

Figure 44: task 4 script

Figure 44 shows the resulting file sizes after executing the script made to encrypt and decrypt each file without removing the padding. Notice that cfb and ofb kept the file size the same for all three files. CBC and ECB however padded each file to either 16 or 32 bits.

Now to examine how the encryption schemes padded the file. Below is what the script outputs when ran from the directory.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t4 (master) $ ./t4_commands.sh
f1.txt
plain: 00000000: 3132 3334 35                                     12345
ecb  : 00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
cbc  : 00000000: 3030 3030 300d 0c03 193f 193f 193f 1919  00000....?.?.?..
cfb  : 00000000: 3132 3334 35                                     12345
ofb  : 00000000: 3132 3334 35                                     12345

f2.txt
plain: 00000000: 3132 3334 3531 3233 3435                         1234512345
```

```
ecb  : 00000000: 3132 3334 3531 3233 3435 0606 0606 0606  1234512345......
cbc  : 00000000: 3132 3334 3531 3233 3435 0606 0606 0606  1234512345......
cfb  : 00000000: 3132 3334 3531 3233 3435                 1234512345
ofb  : 00000000: 3132 3334 3531 3233 3435                 1234512345

f3.txt
plain: 00000000: 3132 3334 3531 3233 3435 3132 3334 3531  1234512345123451
ecb  : 00000000: 3132 3334 3531 3233 3435 3132 3334 3531  1234512345123451
       00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
cbc  : 00000000: 3132 3334 3531 3233 3435 3132 3334 3531  1234512345123451
       00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
cfb  : 00000000: 3132 3334 3531 3233 3435 3132 3334 3531  1234512345123451
ofb  : 00000000: 3132 3334 3531 3233 3435 3132 3334 3531  1234512345123451
```

As shown by the size of each file in the results of the bash script, OFB and CFB did not increase the size of the resulting plaintext, so the results shown in Figure 44 are as expected. However, ecb and cbc both padded the values to 16 bits, padding with the value ‘0b‘ over and over. The decimal equivalent of hex 0b is 11, which happens to be exactly the number of bytes that were used to pad the message! PKCS5 pads the message repeatedly with the number of bytes required to pad the message.

f2.txt, the 10 byte file, has very similar results as f1.txt, however the major difference is the extra data is ‘0606‘ repeating. ‘06‘ in decimal is 6 which is the number of bytes added to the padding!

f3.txt, the 16 byte file, has very similar results as f1.txt and f2.txt, however the major difference is the extra data is an entire 16 bytes of ‘1010‘ repeating. It may seem fair to assume that no paddin gshould be done for ecb and cbc for this message because the message is 16 bytes. However, if the message itself ends with cewrtain bits that may look like it is being padded, part of the message may be lost. Therefore, the message is padded to the next multiple of 2 bytes, 32 bytes.

2.5 Task 5: Error Propagation – Corrupted Cipher Text

To understand the error propagation property of various encryption modes, we would like to do the following exercise:

1. Create a text file that is at least 1000 bytes long.
2. Encrypt the file using the AES-128 cipher.
3. Unfortunately, a single bit of the 55th byte in the encrypted file got corrupted. You can achieve this corruption using the bless hex editor.
4. Decrypt the corrupted ciphertext file using the correct key and IV.

Please answer the following question: How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Please answer this question before you conduct this task, and then find out whether your answer is correct or wrong after you finish this task. Please provide justification.

2.5.1 Task 5: Solution

In order to create a file of at least 1000 bytes long, the only obvious choice was to use the bee movie. The file is found at this location - `https://github.com/mitchdz/ECE471/blob/master/lab1/task2-6/t5/bee_movie.txt`. The bee movie turns out to be 55394 bytes which is a lot more than 1000. Now the script needs to be encrypted using 128 bit AES. For this task, cbc is used.

```
mitch@light: ~/git/ECE471/lab1/task2-6/t5 (master) $ head bee_movie.txt
HOME
The Entire Bee Movie Script

Bee Movie Script - Dialogue Transcript


According to all known laws
of aviation,
```

The first 9 lines of the bee movie are shown using the Linux ‘head‘ command.

Now the bee movie needs to be encrypted and decrypted using ECB, CBC, CFB, and OFB, then to see how much corrupting one bit changes the resulting plaintext when decrypted.

Predictions:

ECB - not much change, mainly because the lack of difussion as shown in the previous tasks.

CBC - Not much change, simply because the algorithm is similar to ECB and works byte by byte, so the area where the bit is changed should look a lot different, but not the entire ciphertext. CFB - A lot of change. CFB stands for Cipher Feedback, so the preceding

blocks effect the future blocks. OFB - A lot of change, similar to CFB. Same reasoning for CFB.

The procedure will be shown for CBC only, but know tha the procedure is the exact same for ECB, CBC, CFB, and OFB.

```
$ openssl enc -aes-128-cbc -e -in bee_movie.txt -out cipher.bin \
    -K 00112233445566778889aabbccddeeff \
    -iv 01020304050607081234123412341212
```

The preceding command creates a file named cipher.bin. which is the encrypted file of the bee movie. Let's mess up a bit in the 55th byte of the bee movie now.

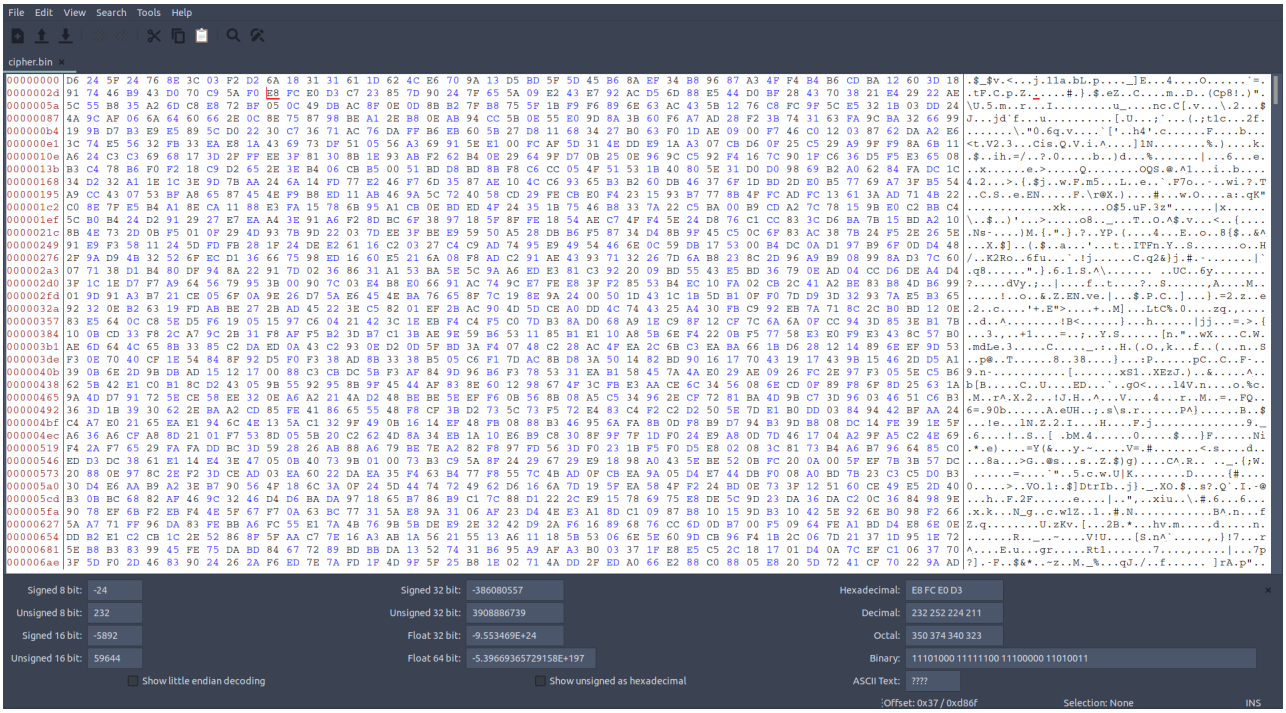

Figure 45: bless editor showing bee movie encrypted with cbc

Figure 45 shows the bless hex editor viewing the bee movie encrypted using aes-128-cbc.

Figure 46: bless editor showing the offset

Figure 46 shows Figure 45 zoomed in on the bottom right, where the bless hex editor shows the offset that your cursor is at. The offest is 0x37 because 0x37 in hex refers to 55 in decimal.

0xE8 is at position 0x37 in bless, and 0xE8 is 232 in decimal. Let's change that value to 231, so that value is changed to E7 as shown in the following picture.

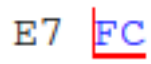

Figure 47: Bless editor showing modified value at byte 0x37

Figure 47 clearly shows the value is now E7. To save the file, press ctrl + c, and the file is saved! Now we need to decrypt the file. The following command is used in order to decrypt the file.

```
$ openssl enc -aes-128-cbc -d -in cipher.bin -out cbc_scrambled.txt \
    -K 00112233445566778889aabbccddeeff \
    -iv 01020304050607081234123412341212
```

Now the head of each file is shown below.

ECB:

```
HOME
The Entire Bee Movie Script

Bee Movie Scri#>$'BA96anscript


According to all known laws
of aviation,
```

CBC:

```
HOME
The Entire Bee Movie Script

Bee Movie Scri)WHw/,1anscrip{


According to all known laws
of aviation,
```

CFB (shown as picture):

Figure 48: Output of the plaintext after corrupting a bit using CFB

OFB:

```
HOME
The Entire Bee Movie Script

Bee Movie Script - Diflogue Transcript


According to all known laws
of aviation,
```

Therefore, 3/4 of my predictions were correct! Figure 48 is shown as a picture because copying those characters broke LaTeX. Only CFB was majorly messed up. I was very surprised that OFB was not messed up because it is a stream cipher. This is because even though OFB is a cipher, the algorithm works like CBC/ECB because OFB streams the text BEFORE it is ciphertext. CFB encrypts the block and then pushes it through the stream. More information is found here on wikipedia `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_Feedback_(CFB)`.

2.6 Task 6: Initial Vector (IV)

Most of the encryption modes require an initial vector (IV). Properties of an IV depend on the cryptographic scheme used. If we are not careful in selecting IVs, the data encrypted by us may not be secure at all, even though we are using a secure encryption algorithm and mode. The objective of this task is to help students understand the problems if an IV is not selected properly. Please do the following experiments:

- **Task 6.1.** A basic requirement for IV is uniqueness, which means that no IV may be reused under the same key. To understand why, please encrypt the same plaintext using (1) two different IVs, and (2) the same IV. Please describe your observation, based on which, explain why IV needs to be unique.

- **Task 6.2.** One may argue that if the plaintext does not repeat, using the same IV is safe. Let us look at the Output Feedback (OFB) mode. Assume that the attacker gets hold of a plaintext (P1) and a ciphertext (C1) , can he/she decrypt other encrypted messages if the IV is always the same? You are given the following information, please try to figure out the actual content of P2 based on C2, P1, and C1.

  ```
  Plaintext (P1): This is a known message!
  Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159
  Plaintext (P2): (unknown to you)
  Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
  ```

  If we replace OFB in this experiment with CFB (Cipher Feedback), how much of P2 can be revealed? You only need to answer the question; there is no need to demonstrate that.

  The attack used in this experiment is called the known-plaintext attack, which is an attack model for cryptanalysis where the attacker has access to both the plaintext and its encrypted version (ciphertext). If this can lead to the revealing of further secret information, the encryption scheme is not considered as secure.

- **Task 6.3.** From the previous tasks, we now know that IVs cannot repeat. Another important requirement on IV is that IVs need to be unpredictable for many schemes, i.e., IVs need to be randomly generated. In this task, we will see what is going to happen if IVs are predictable.

  Assume that Bob just sent out an encrypted message, and Eve knows that its content is either Yes or No; Eve can see the ciphertext and the IV used to encrypt the message, but since the encryption algorithm AES is quite strong, Eve has no idea what the actual content is. However, since Bob uses predictable IVs, Eve knows exactly what IV Bob is going to use next. The following summarizes what Bob and Eve know:

  ```
  Encryption method: 128-bit AES with CBC mode.
  Key (in hex): 00112233445566778899aabbccddeeff (known only to Bob)
  Ciphertext (C1): bef65565572ccee2a9f9553154ed9498 (known to both)
  IV used on P1 (known to both)
  ```

```
    (in ascii): 1234567890123456
    (in hex)  : 31323334353637383930313233343536
Next IV (known to both)
    (in ascii): 1234567890123457
    (in hex)  : 31323334353637383930313233343537
```

A good cipher should not only tolerate the known-plaintext attack described previously, it should also tolerate the chosen-plaintext attack, which is an attack model for cryptanalysis where the attacker can obtain the ciphertext for an arbitrary plaintext. Since AES is a strong cipher that can tolerate the chosen-plaintext attack, Bob does not mind encrypting any plaintext given by Eve; he does use a different IV for each plaintext, but unfortunately, the IVs he generates are not random, and they can always be predictable.

Your job is to construct a message P2 and ask Bob to encrypt it and give you the ciphertext. Your objective is to use this opportunity to figure out whether the actual content of P1 is Yes or No. You can read this Wikipedia page for ideas: `https://en.wikipedia.org/wiki/Initialization_vector`.

There are more advanced cryptanalysis on IV that is beyond the scope of this lab. Students can read the article posted in this URL: `https://defuse.ca/cbcmodeiv.htm`. Because the requirements on IV really depend on cryptographic schemes, it is hard to remember what properties should be maintained when we select an IV. However, we will be safe if we always use a new IV for each encryption, and the new IV needs to be generated using a good pseudo random number generator, so it is unpredictable by adversaries. See another SEED labs (Random Number Generation Lab) for details on how to generate cryptographically strong pseudo random numbers.

2.6.1 Task 6.1: Solution

The plaintext is created using the Linux command echo, and then redirecting the output to a file.

```
$ echo "hello, I am a plaintext!" > plain.txt
```

file 1.bin, 2.bin, and 3.bin are created in the following fashion:

```
$ openssl enc -aes-128-cbc -e -in plain.txt -out 1.bin \
    -K 00112233445566778889aabbccddeeff \
    -iv 01020304050607081234123412341212

$ openssl enc -aes-128-cbc -e -in plain.txt -out 2.bin \
    -K 00112233445566778889aabbccddeeff \
    -iv 01020304050607081234123412341212

$ openssl enc -aes-128-cbc -e -in plain.txt -out 3.bin \
    -K 00112233445566778889aabbccddeeff \
    -iv 00000000000000000000000000000000
```

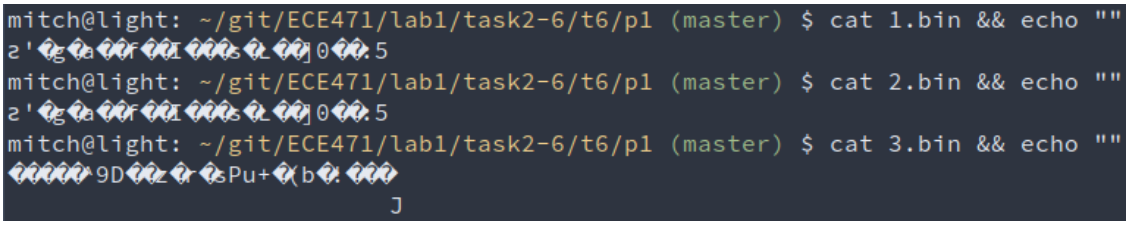

Figure 49: Output of the three binary files from Task 2.6.1

Figure 49 shows the output of 1.bin, 2.bin, and 3.bin. 1.bin 2.bin both used the same IV, whereas 3.bin used a different IV. It is apparent that 1.bin and 2.bin are the exact same file, which shows that using the same IV will result in the same ciphertext. This means that the ciphertext is repeatable, and if the same IV is used multiple times, it may be possible to determine what the IV is.

2.6.2 Task 6.2: Solution

In order to accomplish this task, the website `http://xor.pw/#` was used.

Output Feedback (OFB) is a stream cipher which relies heavily on XORing plaintext blocks with the key and an IV. If an IV and the key are the same for different plaintexts, it is trivial to decrypt P2 while utilizing a known-plaintext attack. The theory is outlined below:

$$\mathrm{K} = \mathrm{P1} \bigoplus C1$$
$$\mathrm{P2} = \mathrm{K} \bigoplus C2$$

Let's see how well this theory holds up!

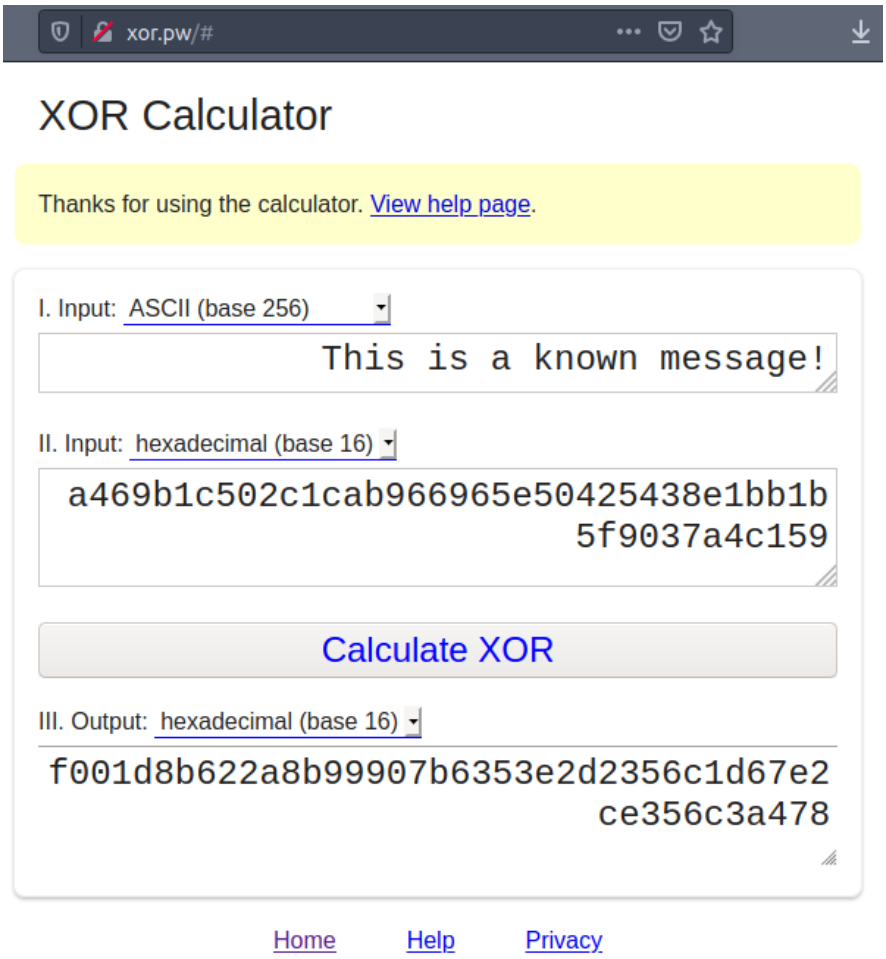

Figure 50: Task 6.2: Finding K

Figure 50 shows using `http://xor.pw/#` in order to calculate what K is. This website is great because we can directly XOR between ascii and hex, so there's no need to convert P1 to hex with another program first. The same operation is shown below in a format that is copy-able.

```
This is a known message!                    (ascii)
```

```
XOR a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159 (hex)
    ----------------------------------------------
    f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a478 (hex)
```

Therefore, the key, K, is f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a478. Finding the plaintext associated with C2 is then trivial using the following operation.

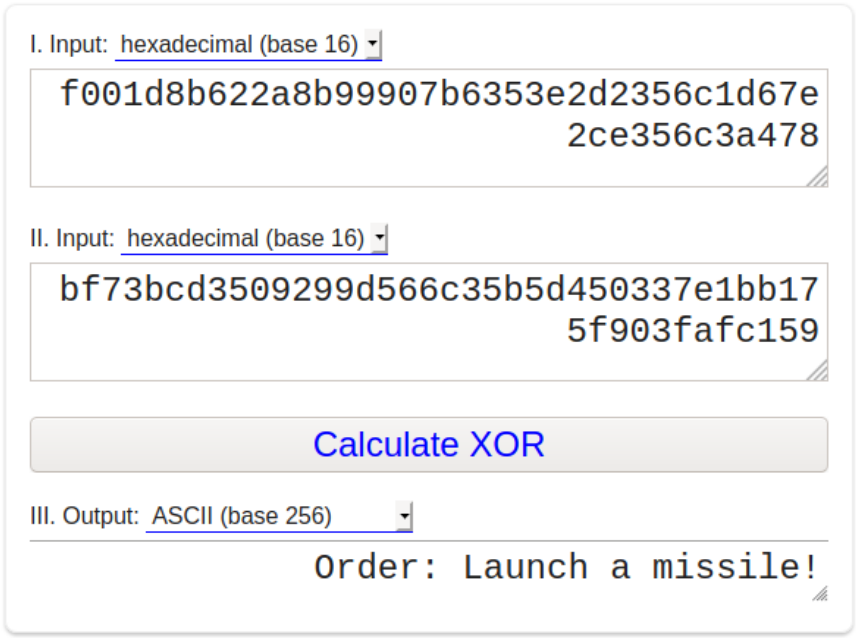

Figure 51: Task 6.2: Decrypting C2

Figure 51 shows deciphering C2 using the key found from Figure 50! The resulting string is "Order: Launch a missile!". The XOR operation is shown below again.

```
    f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a478 (hex)
XOR bf73bcd3509299d566c35b5d450337e1bb175f903fafc159 (hex)
    ----------------------------------------------
    Order: Launch a missile!                     (ascii)
```

P2 is therefore **Order: Launch a missile!**

For CFB, reusing an IV will leak some information about the first block of the plaintext. Any common prefix will also be leaked. CFB is a little bit more secure and will not completely break like OFB did.

2.6.3 Task 6.3: Solution

I choose my P2 to be the following:

$$P2 = IV_1 \bigoplus IV_2 \bigoplus "No"$$

Bob will then encrypt the given P2 as such:

$$C_2 = E_k(IV_2 \bigoplus P_2) = E_k(IV_2 \bigoplus (IV_2 \bigoplus IV_1 \bigoplus "No"))$$

$IV_2 \bigoplus IV_2$ cancel out, meaning that the operation that Bob ends up producing is the following:

$$C_2 = E_k(IV_1 \bigoplus "No")$$

Now we should be able to compare C_1 and C_2 to see if they match! If they differ, then P1 says "Yes".

Therefore, carefully picking P2 will allow us to encrypt data through Bob using IV_1. $IV_1 \bigoplus IV_2 =$ 0x1 because the difference between them is just one number. Therefore the P2 when we use "Yes" as our reference is calculated as the following:

```
    5965730d0d0d0d0d0d0d0d0d0d0d0d0d (ascii)
XOR 00000000000000000000000000000001  (hex)
    -----------
    5965730d0d0d0d0d0d0d0d0d0d0d0d0c (hex)
```

```
mitch@light: ~/git/ECE471/lab1/task2-6/t6/p3/2nd_attempt (master) $ openssl enc -aes-128-cbc -
e -in P2 -K 00112233445566778899aabbccddeeff -iv 31323334353637383930313233343537 -nosalt -nop
ad | xxd -p
bef65565572ccee2a9f9553154ed9498
```

Figure 52: executing the chosen plaintext attack

Figure 52 shows the plaintext attack being executed. For easier reading, the command is placed below:

```
$ openssl enc -aes-128-cbc -e -in P2 \
    -K 00112233445566778899aabbccddeeff \
    -iv 31323334353637383930313233343537 \
    -nosalt -nopad | xxd -p
```

the end result gave us "bef65565572ccee2a9f9553154ed9498" which is infact the C1 that we are trying to match! Therefore, we can conclude that P1 is "Yes" and not "No".

To recap, we use the plaintext P2 "5965730d0d0d0d0d0d0d0d0d0d0d0d0d0c", and that confirms that P1 is "Yes".