LAB REPORT

SWE430: Information and Network Security

Submitted to

Partha Protim Paul
Lecturer
Dept. of Software Engineering
Shahjalal University of Science and Technology

Submitted by

Md. Mehrajul Islam
2019831074

LAB 2
Attacking Classic Crypto Systems

Checkpoint 1: Breaking Caesar Cipher

Given Cipher: *odroboewscdrolocdcwkbdmyxdbkmdzvkdpybwyeddrobo*

ceaserCracker.py

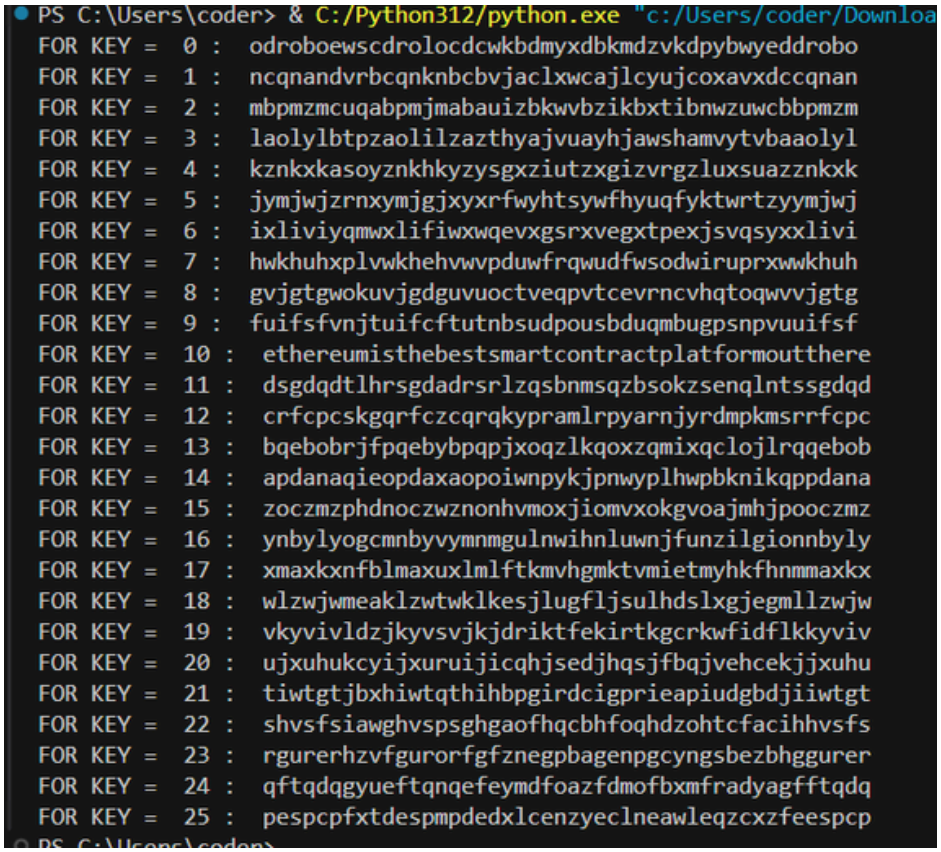
C: > Users > coder > Downloads > MEHRAJ > MEHRAJ > LAB 2 > TASK 1 > ceaserCracker.py

```python
cipherText = 'odroboewscdrolocdcwkbdmyxdbkmdzvkdpybwyeddrobo'
alphabets = 'abcdefghijklmnopqrstuvwxyz'

for key in range(len(alphabets)):

    decrypted = ''

    for ch in cipherText:

        if ch in alphabets:

            ascii = alphabets.find(ch)
            ascii = ascii - key

            if ascii < 0:
                ascii = ascii + len(alphabets)

            decrypted = decrypted + alphabets[ascii]

        else:
            decrypted = decrypted + ch

    print('FOR KEY = ', key, ': ', decrypted)
```

Algorithm:

- *There can be possible 25 keys. So first a loop with* ***key 0 to 25***
- *In each iteration for each character in ciphertext get its ASCII and deduct key value from that*
- *Add the character in the final string and print them after each iteration*
- *Lastly go through all of them and find the meaningful one*

OUTPUT



HERE FOR KEY = 10 WE GET:
ethereumisthebestsmartcontractplatformoutthere

SO DECRYPTED MESSAGE IS:
ethereum is the best smart contract platform out there

Checkpoint 2: Breaking Substitution Cipher

Cipher 1: *af p xpkcaqvnpk pfg, af ipqe qpri, gauuikifc tpw, ceiri udvk tiki afgarxifrphni cd eaowvmd popkwn, hiqpvri du ear jvaql vfgikrcpfgafm du cei xkafqaxnir du xrwqedearcdkw pfg du ear aopmafpcasi xkdhafmr afcd fit pkipr. ac tpr qdoudkcafm cd lfdt cepc au pfwceafm epxxifig cd ringdf eaorinu hiudki cei opceiopcaqr du cei uaing qdvng hi qdoxnicinw tdklig dvc pfg edt rndtnw ac xkdqiigig, pfg edt odvfcpafdvr cei dhrcpqnir--ceiki tdvng pc niprc kiopafdfi mddg oafg cepc tdvng qdfcafvi cei kiripkqe*

SUBSTITUTION CYPHER CODE

```cpp
map < char , int > cipherTextFrequency;
int numberOfReplacableCharecters = 0;

string ciphertext = "af p xpkcaqvnpk pfg, af ipqe qpri, gauuikifc tpw, ceiri udvk tiki
    afgarxifrphni cd eaowvmd popkwn, hiqpvri du ear jvaql vfgikrcpfgafm du cei xkafqaxnir du
    xrwqedearcdkw pfg du ear aopmafpcasi xkdhafmr afcd fit pkipr. ac tpr qdoudkcafm cd lfdt cepc
    au pfwceafm epxxifig cd ringdf eaorinu hiudki cei opceiopcaqr du cei uaing qdvng hi
    qdoxnicinw tdklig dvc pfg edt rndtnw ac xkdqiigig, pfg edt odvfcpafdvr cei dhrcpqnir--ceiki
    tdvng pc niprc kiopafdfi mddg oafg cepc tdvng qdfcafvi cei kiripkqe";

for(auto ch: ciphertext){
    if('a' <= ch && 'z' >= ch) cipherTextFrequency[ch]++; numberOfReplacableCharecters++;
}

vector < pair < double , char > > cipherFreqList;
string englishLanguageFreqOrder = "etaonhisrdluwmgcfybpkvjxzq";
string key = "abcdefghijklmnopqrstuvwxyz";
string ciphertextFrequencyOrder = "";

for(auto freqData: cipherTextFrequency) {
    cipherFreqList.push_back({(freqData.second * 100.00) / numberOfReplacableCharecters, freqData
        .first});
}

sort(cipherFreqList.begin(), cipherFreqList.end());

cout << "CIPHERTEXT FREQUENCY ORDER : " << endl;
for(auto freq: cipherFreqList){
    cout << freq.second << " >> " << freq.first << "% " << endl;
    ciphertextFrequencyOrder += freq.second;
}

reverse(ciphertextFrequencyOrder.begin(), ciphertextFrequencyOrder.end());

  cout << endl << "ciphertextFrequencyOrder : " << ciphertextFrequencyOrder << endl;
  cout << "englishLanguageFrequencyOrder : " << englishLanguageFreqOrder << endl << endl;

  int indx = 0;
  for(auto ch: ciphertextFrequencyOrder){
      cout << key[(int)englishLanguageFreqOrder[indx] - (int)'a'] << " will be replaced as " << ch
          << endl;
      key[(int)englishLanguageFreqOrder[indx] - (int)'a'] = ch;
      indx++;
  }

  cout << endl << " >>> KEY FOUND : " << key << endl << endl;

  string decrypted = ciphertext;

  int i = 0;
  for(auto lol: ciphertext) {
      if('a' <= ciphertext[i] && 'z' >= ciphertext[i]) decrypted[i] = key[(int)lol - (int)'a'];
      i++;
  }

  cout << " >>> AFTER DECRYPTION : " << endl;
  cout << decrypted << endl;

  return 0;
}
```

ALGORITHM FOR CRACKING SUBSTITUTION CIPHER

ALGORITHM:

- *Calculate the percentage of the frequency for the cipher text*
- *Sort them according to their appearance*
- *Compare them with the English language frequency*
- *Replace the cipher text according to the frequency of the cipher text and English language*
- *Check the result*

OUTPUT FOR CIPHER 1

CIPHERTEXT FREQUENCY ORDER :

j >> 0.203666%
s >> 0.203666%
l >> 0.610998%
h >> 1.222%
m >> 1.42566%
w >> 1.62933%
x >> 2.03666%
o >> 2.24033%
t >> 2.24033%
u >> 2.64766%
v >> 2.64766%
q >> 3.05499%
n >> 3.25866%
g >> 3.86965%
k >> 3.86965%
e >> 4.48065%
r >> 4.68432%
f >> 6.10998%
a >> 6.31365%
p >> 6.51731%
c >> 6.72098%
d >> 7.33198%
i >> 9.36864%

ciphertextFrequencyOrder :
idcpafrekgnqvutoxwmhlsj

englishLanguageFrequencyOrder :
etaonhisrdluwmgcfybpkvjxzq

KEY FOUND : cmogixtfrjlnuaphqkedqsvxwz

AFTER DECRYPTION :

cx h xhlocqsahl hxt, cx rhqi qhkr, tcqqrlrxo dhv, oirkr qgsl drlr cxtckxrxkhfar og icpvsug hphlva, frqhskr gq ick jscqn sxtrlkohxtcxu gq oir xlcxqcxark gq xkvqigickoglv hxt gq ick cphucxhocer xlgfcxuk cxog xrd hlrhk. co dhk qgpqglocxu og nxgd oiho cq hxvoicxu ihxxrxrt og kratgx icpkraq frqglr oir phoirphocqk gq oir qcrat qgsat fr qgpxarorav dglnrt gso hxt igd kagdav co xlgqrrtrt, hxt igd pgsxohcxgsk oir gfkohqark--oirlr dgsat ho arhko lrphcxgxr uggt pcxt oiho dgsat qgxocxsr oir lrkrhlqi

Cipher 2: *aceah toz puvg vcdl omj puvg yudqecov, omj loj auum klu thmjuv hs klu zlcvu shv zcbkg guovz, upuv zcmdu lcz vuwovroaeu jczoyyuovomdu omj qmubyudkuj vukqvm. klu vcdluz lu loj avhqnlk aodr svhw lcz kvopuez loj mht audhwu o ehdoe eunumj, omj ck toz yhyqeoveg auecupuj, tlokupuv klu hej sher wcnlk zog, klok klu lcee ok aon umj toz sqee hs kqmmuez zkqssuj tckl kvuozqvu. omj cs klok toz mhk umhqnl shv sowu, kluvu toz oezh lcz yvhehmnuj pcnhqv kh wovpue ok. kcwu thvu hm, aqk ck zuuwuj kh lopu eckkeu ussudk hm wv. aonncmz. ok mcmukg lu toz wqdl klu zowu oz ok scskg. ok mcmukg-mcmu klug aunom kh doee lcw tuee-yvuzuvpuj; aqk qmdlomnuj thqej lopu auum muovuv klu wovr. kluvu tuvu zhwu klok zlhhr klucv luojz omj klhqnlk klcz toz khh wqdl hs o nhhj klcmn; ck zuuwuj qmsocv klokomghmu zlhqej yhzzuzz (oyyovumkeg) yuvyukqoe ghqkl oz tuee oz (vuyqkujeg) cmubloqzkcaeu tuoekl. ck tcee lopu kh au yocj shv, klug zocj. ck czm'k mokqvoe, omj kvhqaeu tcee dhwu hs ck! aqk zh sov kvhqaeu loj mhk dhwu; omj oz wv. aonncmz toz numuvhqz tckl lcz whmug, whzk yuhyeu tuvu tceecmn kh shvncpu lcw lcz hjjckcuz omj lcz nhhj shvkqmu. lu vuwocmuj hm pczckcmn kuvwz tckl lcz vueokcpuz (ubduyk, hs dhqvzu, klu zodrpceeu- aonncmzuz), omj lu loj womg juphkuj ojwcvuvz owhmn klu lhaackz hs yhhv omj qmcwyhvkomk sowcecuz. aqk lu loj mh dehzu svcumjz, qmkce zhwu hs lcz ghqmnuv dhqzcmz aunom kh nvht qy. klu uejuzk hs kluzu, omj aceah'z sophqvcku, toz ghqmn svhjh aonncmz. tlum aceah toz mcmukg-mcmu lu ojhykuj svhjh oz lcz lucv, omj avhqnlk lcw kh ecpu ok aon umj; omj klu lhyuz hs klu zodrpceeu- aonncmzuz tuvu scmoeeg jozluj. aceah omj svhjh loyyumuj kh lopu klu zowu acvkljog, zuykuwauv 22mj. ghq loj aukkuv dhwu omj ecpu luvu, svhjh wg eoj, zocj aceah hmu jog; omj klum tu dom dueuavoku hqv acvkljog-yovkcuz dhwshvkoaeg khnukluv. ok klok kcwu svhjh toz zkcee cm lcz ktuumz, oz klu lhaackz doeeuj klu cvvuzyhmzcaeu ktumkcuz auktuum dlcejlhhj omj dhwcmn hs onu ok klcvkg-klvuu*

OUTPUT FOR CIPHER 2

CIPHERTEXT FREQUENCY ORDER :

b >> 0.205339%
r >> 0.359343%
p >> 1.12936%
g >> 1.43737%
y >> 1.43737%
d >> 1.48871%
t >> 1.74538%
n >> 1.89938%
s >> 1.95072%
w >> 1.95072%
q >> 2.15606%
a >> 2.41273%
e >> 3.64476%
j >> 3.79877%
v >> 4.36345%
m >> 4.8768%
z >> 4.8768%
l >> 4.97947%
c >> 5.23614%
h >> 5.80082%
o >> 6.72485%
k >> 6.77618%
u >> 10.1643%

ciphertextFrequencyOrder :
ukohclzmvjeaqwsntdygprb

englishLanguageFrequencyOrder :
etaonhisrdluwmgcfybpkvjxzq

KEY FOUND : oynjutslzbpewchgqvmkarqxdz

AFTER DECRYPTION :

onuol khz gars rnje hwb gars dajqunhr, hwb ehb oaaw pea klwbar lm pea zenra mlr znyps sahrz, **evar** znwja enz raqhrvhoua bnzhddahrhwja hwb qwaydajpab rapqrw. pea rnjeaz ea ehb orlqcep ohjv mrlq enz prhgauz ehb wlk oajlqa h uljhu uacawb, hwb np khz dldquhrus oaunagab, kehpagar pea lub mluv qncep zhs, pehp pea enuu hp ohc awb khz mquu lm pqwwauz zpqmmab knpe prahzqra. hwb nm pehp khz wlp awlqce mlr mhqa, peara khz huzl enz drlulwcab gnclqr pl qhrgau hp. pnqa klra lw, oqp np zaaqab pl ehga unppua ammajp lw qr. ohccnwz. hp wnwaps ea khz qqje pea zhqa hz hp mnmps. hp wnwaps-wnwa peas oachw pl jhuu enq kauu-drazargab; oqp qwjehwcab klqub ehga oaaw wahrar pea qhrv. peara kara zlqa pehp zellv peanr eahbz hwb pelqcep penz khz pll qqje lm h cllb penwc; np zaaqab qwmhnr pehphwslwa zelqub **plssass** (hddhrawpus) dardapqhu slqpe hz kauu hz (radqpabus) nwayehqzpnoua kahupe. np knuu ehga pl oa dhnb mlr, peas zhnb. np nzw'p whpqrhu, hwb prlqoua knuu jlqa lm np! oqp zl mhr prlqoua ehb wlp jlqa; hwb hz qr. ohccnwz khz cawarlqz knpe enz qlwas, qlzp daldua kara knuunwc pl mlrcnga enq enz lbbnpnaz hwb enz cllb mlrpqwa. ea raqhnwab lw gnznpnwc parqz knpe enz rauhpngaz (**expcdt**, lm jlqrza, pea zhjvgnuua- ohccnwzaz), hwb ea ehb qhws baglpab hbqnrarz hqlwc pea eloonpz lm dllr hwb qwnqdlrphwp mhqnunaz. oqp ea ehb wl julza mrnawbz, qwpnu zlqa lm enz slqwcar jlqznwz oachw pl crlk qd. pea aubazp lm peaza, hwb onuol'z mhglqrnpa, khz slqwc mrlbl ohccnwz. keaw onuol khz wnwaps-wnwa ea hbldpab mrlbl hz enz eanr, hwb orlqcep enq pl unga hp ohc awb; hwb pea eldaz lm pea zhjvgnuua- ohccnwzaz kara mnwhuus bhzeab. onuol hwb mrlbl ehddawab pl ehga pea zhqa onrpebhs, zadpaqoar 22wb. slq ehb oappar jlqa hwb unga eara, mrlbl qs uhb, zhnb onuol lwa bhs; hwb peaw ka jhw jauaorhpa lqr onrpebhs-dhrpnaz jlqmlrphous plcapear. hp pehp pnqa mrlbl khz zpnuu nw enz pkaawz, hz pea eloonpz jhuuab pea nrrazdlwznoua pkawpnaz oapkaaw jenubellb hwb jlqnwc lm hca hp penrps-peraa

CONCLUSION :

Here in the both cyphertext I tried to solve it using the English language frequency. But still it was unable to crack it. Because the text was too small. As we know that if the length of the ciphertext is bigger the breaking of substitution cipher becomes very easy. As the first ciphertext is smaller that is why it cannot be decoded at all but the second one is relatively bigger which makes it a little bit better suit for decoding. So, in the second one we can see some words like “evar”(ever), “plssass”(possess) and “expcdt”(expect).

LAB 3
Symmetric encryption & hashing

TASK 3.1: AES encryption using different modes

Firstly, I openned a text file names **task1.txt**
Then I gave the following commands:

Using AES128 CBC

openssl enc -aes-128-cbc -e -in task1.txt -out task1aes128cbc.bin
-K 7794212354564627485 6abcbdcedaeab
-iv 33355500033347349994444453555222

openssl enc -aes-128-cbc -d -in task1aes128cbc.bin
-out task1aes128cbcdecrypted.txt
-K 779421235456462748 56abcbdcedaeab -iv
33355500033347349994444453555222

Using AES128 CFB

openssl enc -aes-128-cfb -e -in task1.txt -out task1aes128cfb.bin
-K 7794212354564627485 6abcbdcedaeab
-iv 33355500033347349994444453555222

openssl enc -aes-128-cfb -d -in task1aes128cfb.bin
-out task1aes128cfbdycrypted.txt
-K 7794212354564627485 6abcbdcedaeab
-iv 33355500033347349994444453555222

Using AES128 ECB

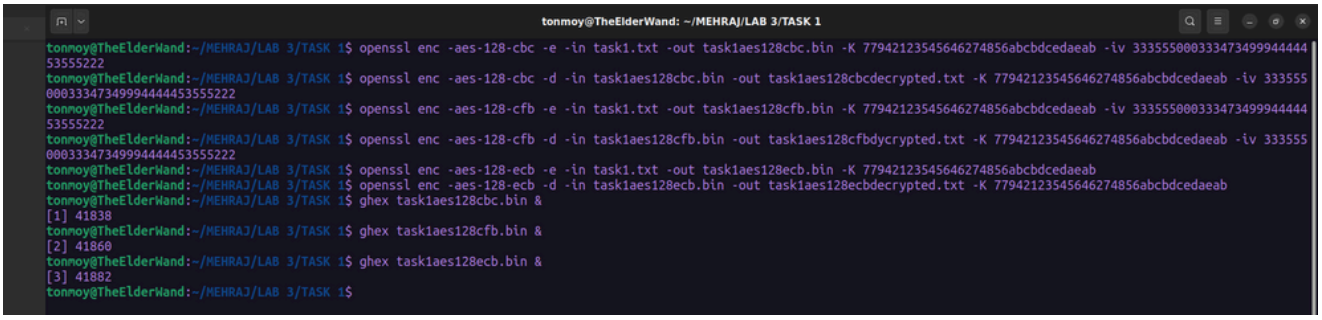
openssl enc -aes-128-ecb -e -in task1.txt -out task1aes128ecb.bin
-K 7794212354564627485 6abcbdcedaeab

openssl enc -aes-128-ecb -d -in task1aes128ecb.bin
-out task1aes128ecbdecrypted.txt
-K 7794212354564627485 6abcbdcedaeab

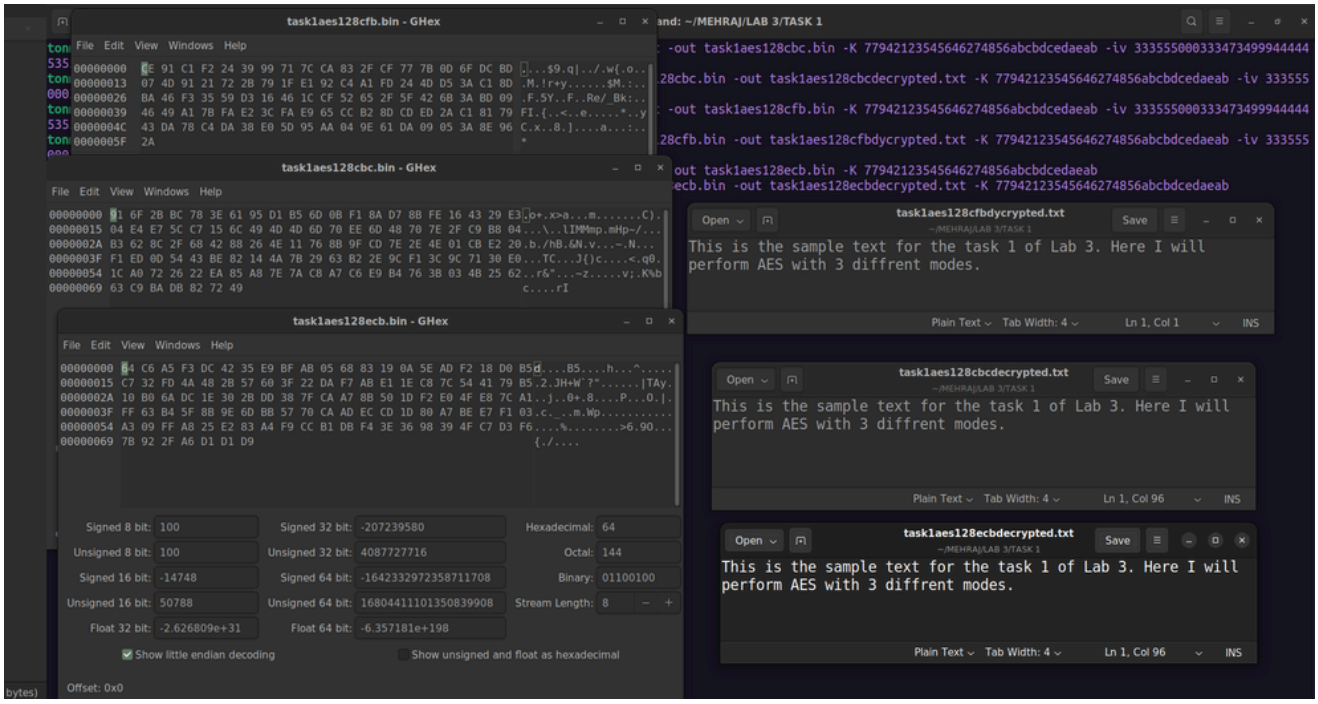
To read the encrypted binary files I used ghex:

ghex task1aes128cbc.bin &
ghex task1aes128cfb.bin &
ghex task1aes128ecb.bin &

These are the commands that I gave



This are the encrypted and decrypted files

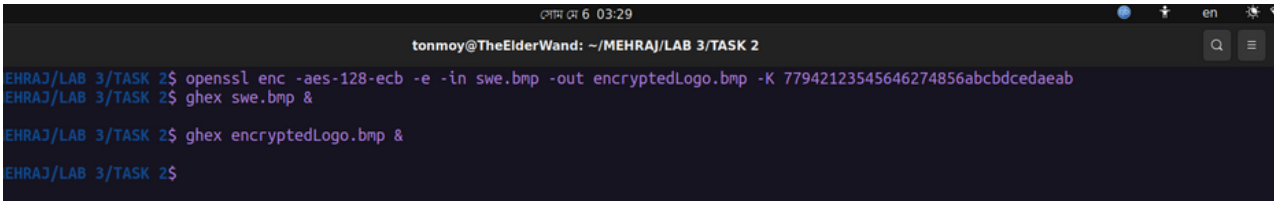


TASK 3.2: ENCRYPTION MODE - ECB VS CBC

I downloaded the logo of SWE Society and converted it to bmp file.
The name of the file is swe.bmp

ENCRYPT USING AES128-ECB

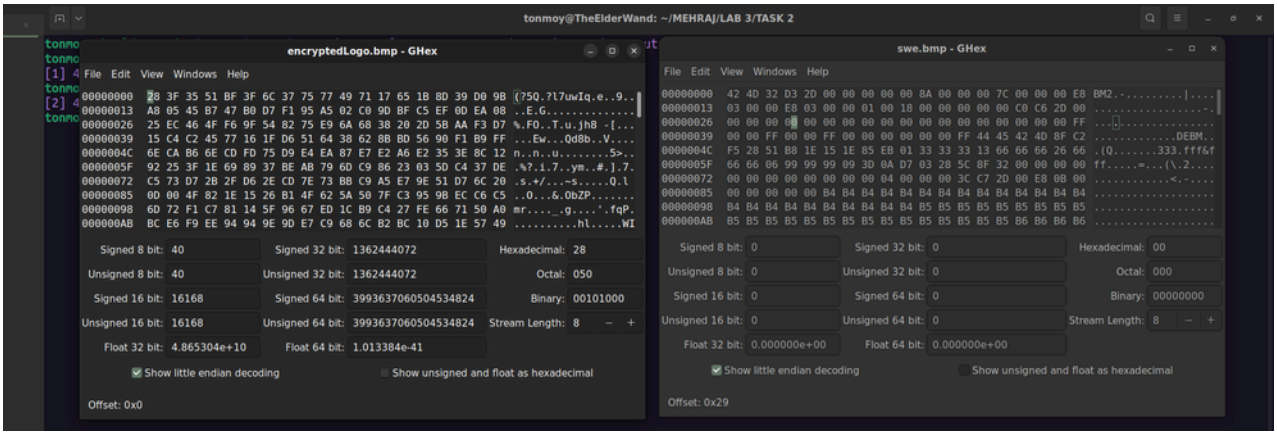
openssl enc -aes-128-ecb -e -in swe.bmp -out encryptedLogo.bmp
-K 77942123545646274856abcbdcedaeab
ghex swe.bmp &

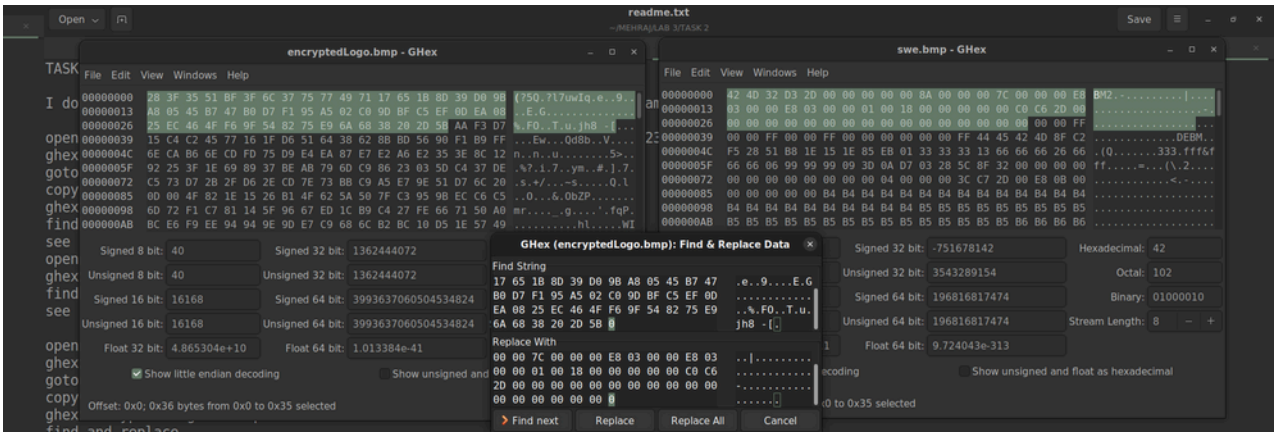


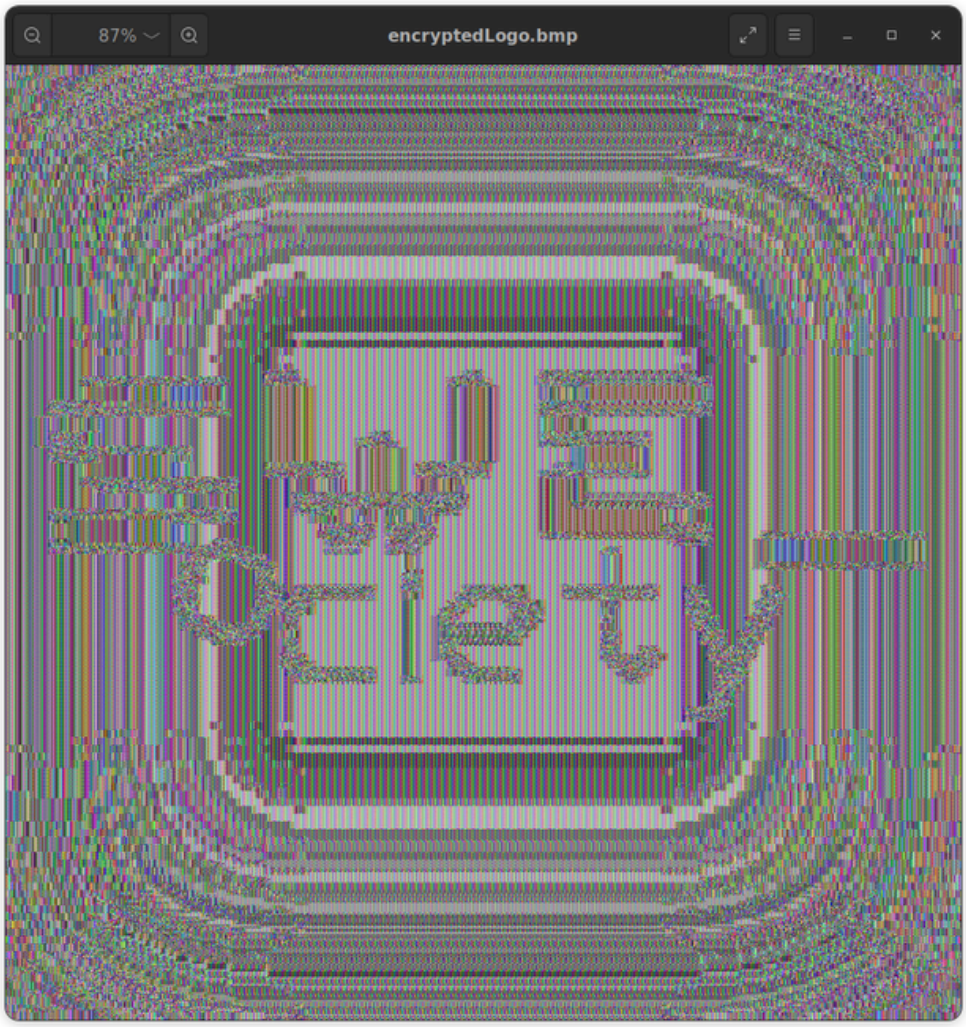
Now the encrypted image cannot be read due to the header encryption so now I will take the first 52 bits from swe.bmp and replace those bits in the encrypted image

PROCEDURE:

goto byte 54 of swe.bmp
copy 54 bits
ghex encryptedLogo.bmp &
goto byte 54
find and replace



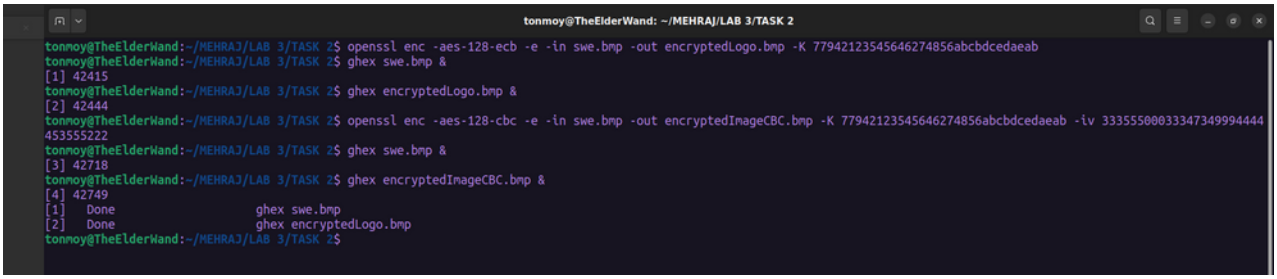


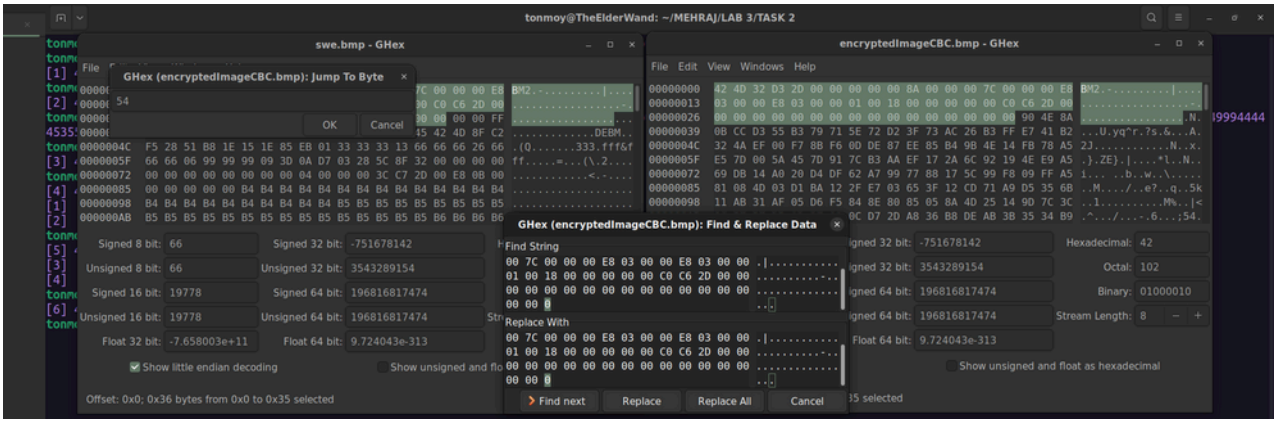


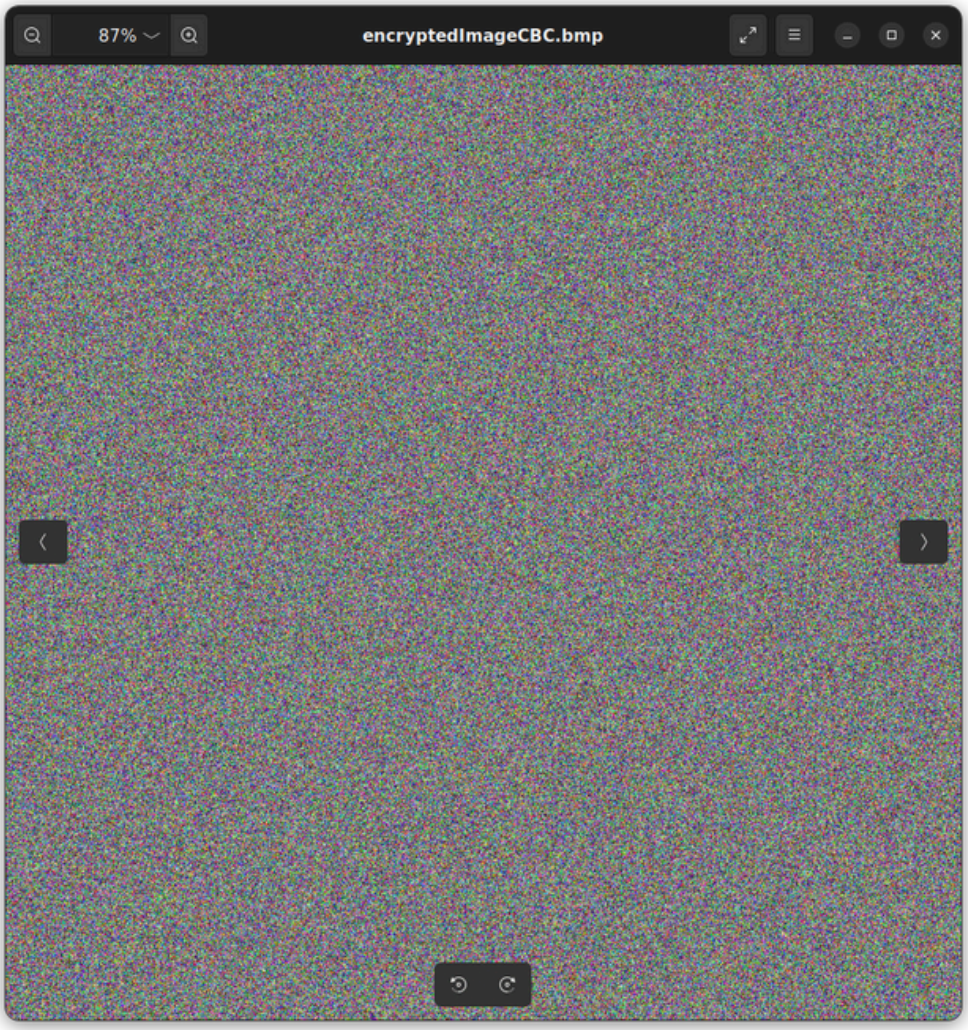
This is the output of the encryption for AES128-ECB

USING AES128-CBC

openssl enc -aes-128-cbc -e -in swe.bmp -out encryptedImageCBC.bmp
-K 7794212354564627 4856abcbdcedaeab
-iv 33355500033347349994444453555222
ghex swe.bmp &







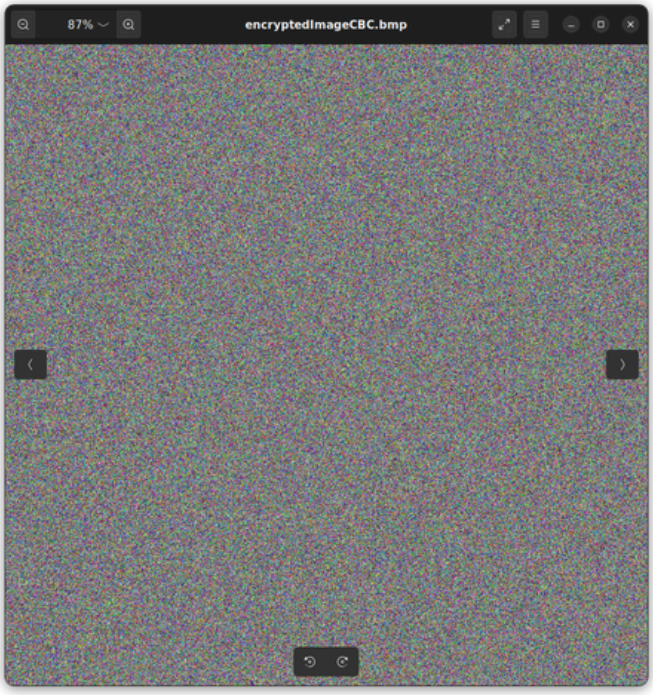
This is the output of the encryption for AES128-CBC

ECB & CBC Mode Analysis

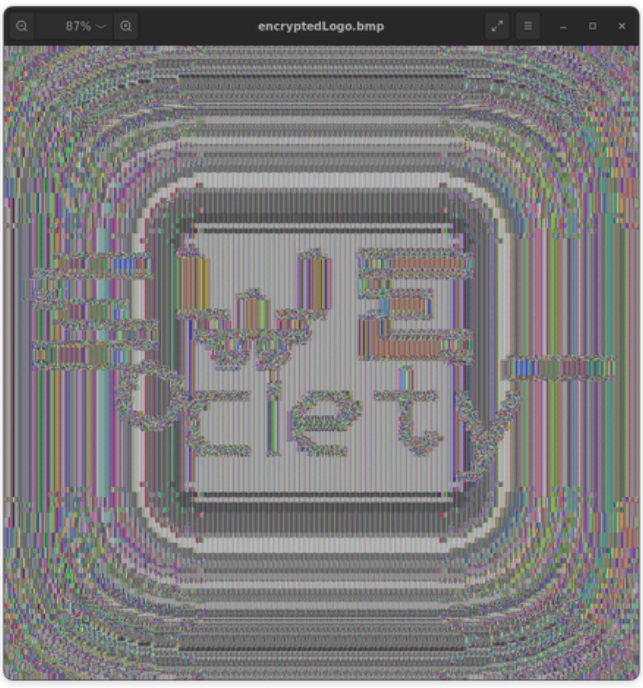
CBC (Cipher Block Chaining) and ECB (Electronic Codebook) are both block cipher encryption. where each block of plaintext is encrypted independently with the same key in ECB mode. It lacks diffusion, meaning identical blocks of plaintext will result in identical blocks of ciphertext. This can lead to security vulnerabilities, especially in image encryption, where patterns might be preserved. On the other hand, In CBC mode, each plaintext block is XORed with the previous ciphertext block before encryption. This adds diffusion, making it more resistant to patterns and repetition in the plaintext. CBC mode requires an Initialization Vector (IV) for the first block to start the chaining process.

COMPAISON:

ECB is simpler and faster, but less secure, especially for images. CBC is more secure due to its chaining mechanism, but slightly slower and requires an IV. For image encryption, CBC is generally preferred over ECB due to its better resistance to pattern preservation and higher security.



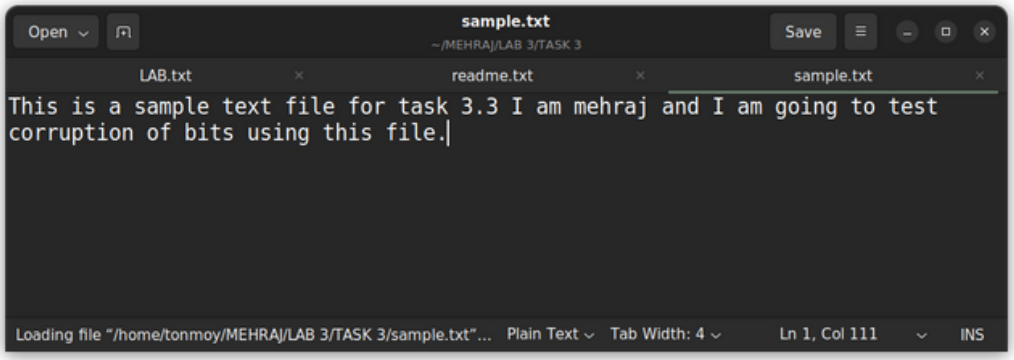
CBC



ECB

TASK 3.3: ECRIPTION MODE : CORRUPTED CYPHERTEXT

Firstly, I created a text file sample.txt

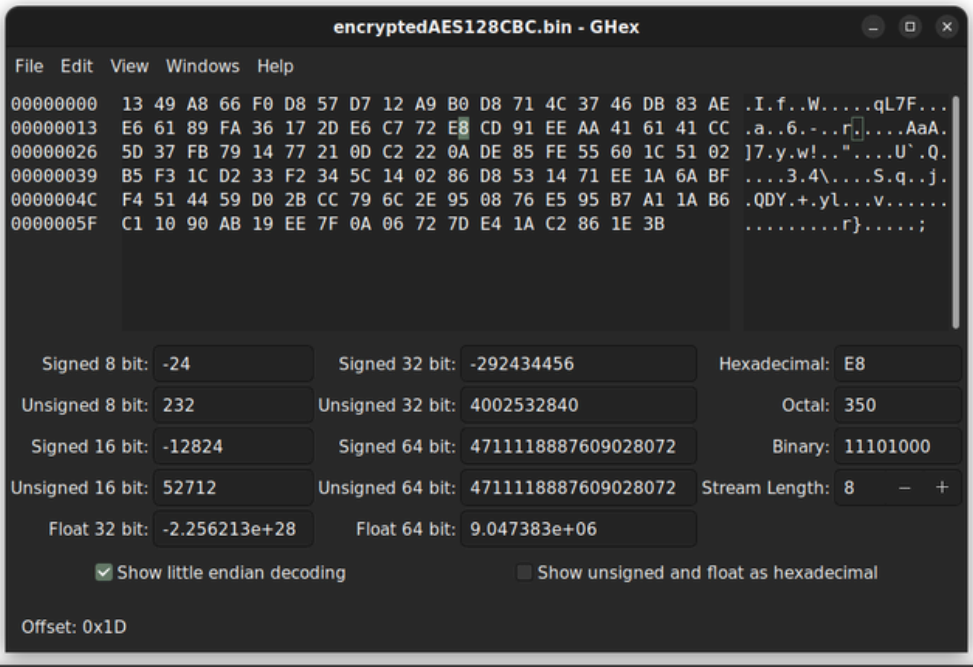


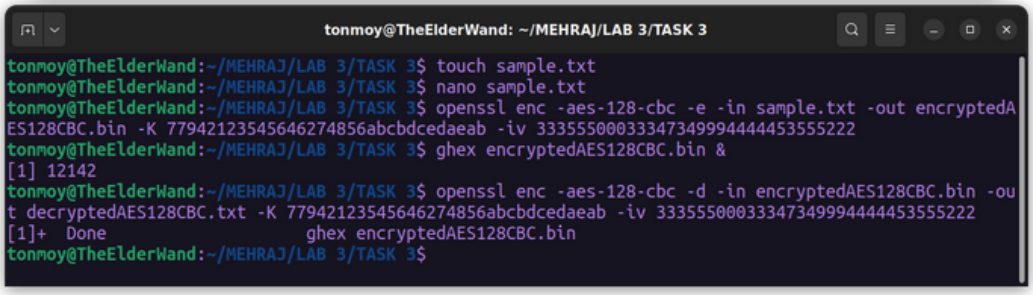
FOR CBC MODE:

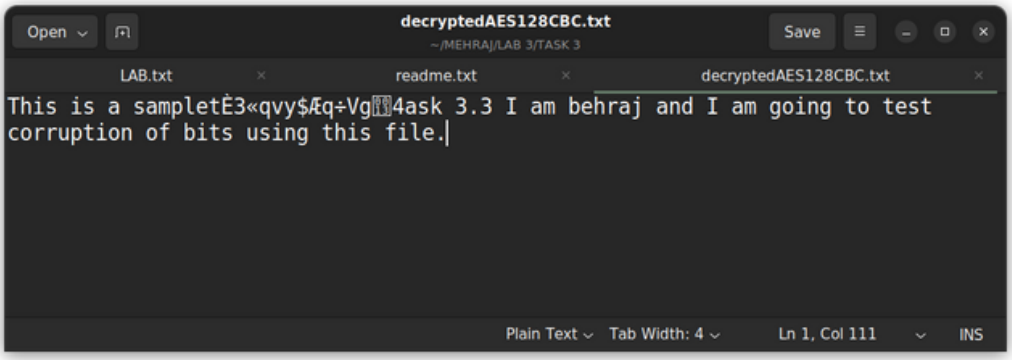
openssl enc -aes-128-cbc -e -in sample.txt -out encryptedAES128CBC.bin -K 7794212354564627485 6abcbdcedaeab -iv 33355500033347349994444453555222 ghex encryptedAES128CBC.bin &

Then I went to 30th bit and changed the HEX value

openssl enc -aes-128-cbc -d -in encryptedAES128CBC.bin -out decryptedAES128CBC.txt -K 7794212354564627485 6abcbdcedaeab -iv 33355500033347349994444453555222





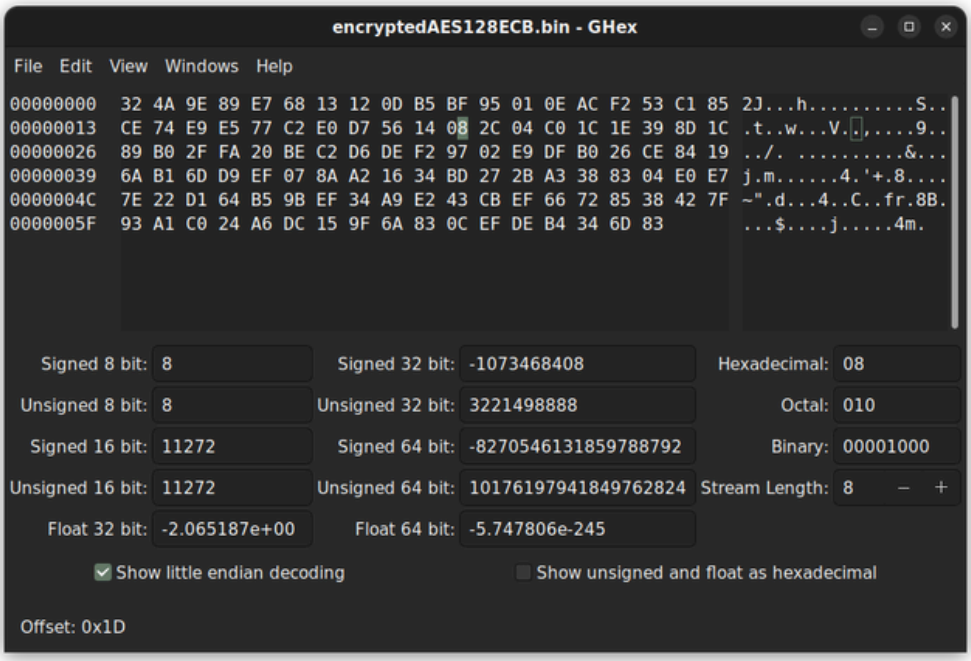


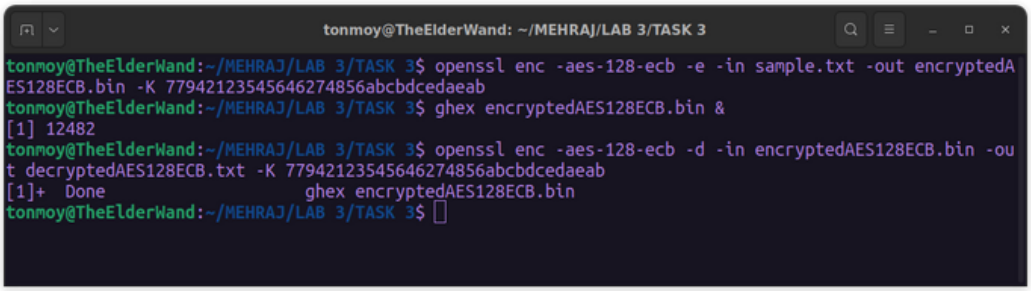
FOR ECB MODE

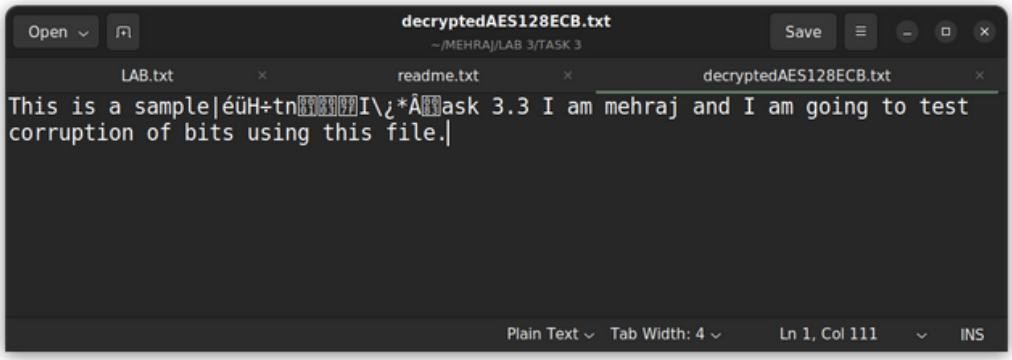
openssl enc -aes-128-ecb -e -in sample.txt -out encryptedAES128ECB.bin
-K 77942123545646274856abcbdcedaeab
ghex encryptedAES128ECB.bin &

Then I went to 30th bit and changed the HEX value

openssl enc -aes-128-ecb -d -in encryptedAES128ECB.bin -out decryptedAES128ECB.txt
-K 77942123545646274856abcbdcedaeab







FOR CFB MODE

openssl enc -aria-128-cfb -e -in sample.txt -out encryptedAES128CFB.bin
-K 77942123545646274856abcbdcedaeab -iv 33355500033347349994444453555222
ghex encryptedAES128CFB.bin &

Then I went to 30th bit and changed the HEX value

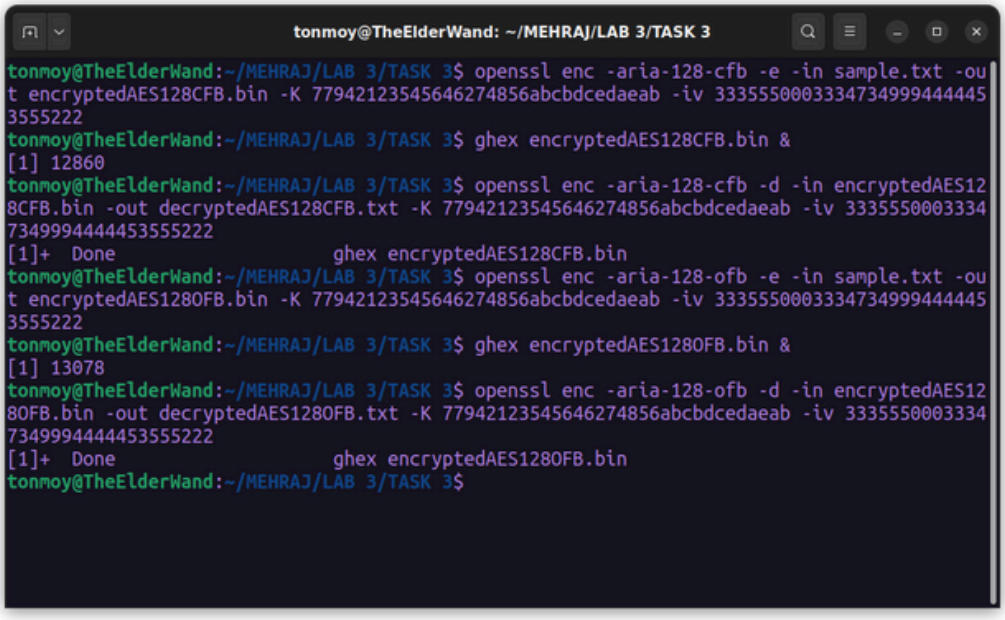
openssl enc -aria-128-cfb -d -in encryptedAES128CFB.bin -out decryptedAES128CFB.txt
-K 77942123545646274856abcbdcedaeab -iv 33355500033347349994444453555222

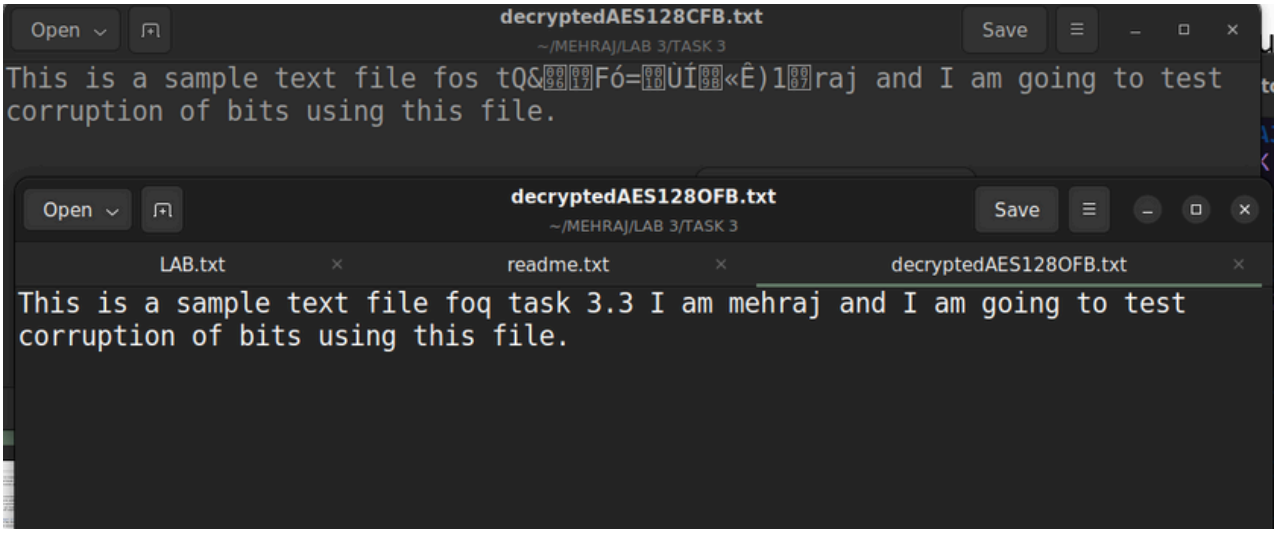
FOR OFB MODE

openssl enc -aria-128-ofb -e -in sample.txt -out encryptedAES128OFB.bin
-K 77942123545646274856abcbdcedaeab -iv 33355500033347349994444453555222
ghex encryptedAES128OFB.bin &

Then I went to 30th bit and changed the HEX value

openssl enc -aria-128-ofb -d -in encryptedAES128OFB.bin -out decryptedAES128OFB.txt
-K 77942123545646274856abcbdcedaeab -iv 33355500033347349994444453555222





Here these are the results of CFB & OFB

ANALYSIS

ECB Mode

Corrupted Information: "|éüH÷tn□□□l\¿*Âƒ" (16 bytes)

Explanation: In ECB mode, each block is encrypted independently. Therefore, the corruption only affects the specific block where it occurred. The rest of the plaintext remains intact.

CBC Mode:

Corrupted Information: "tÈ3«qvy$Æq÷Vg□4" (16 bytes)

Explanation: While CBC mode encrypts blocks in a chained manner, the corruption in one block affects the decryption of subsequent blocks. However, the first block can still be recovered as it only depends on the IV and the first block of ciphertext.

CFB Mode:

Corrupted Information: "Q&-□Fó=□Ùí› «Ê)1□" (16 bytes)

Explanation: In CFB mode, the corruption in one ciphertext block affects the decryption of subsequent blocks. However, due to the feedback mechanism, only a part of the subsequent blocks is corrupted. As a result, more information can be recovered compared to CBC mode.

OFB Mode:

Corrupted Information: "q" (1 bytes)

Explanation: Similar to CFB mode, the corruption in one ciphertext block affects the decryption of subsequent blocks. However, in OFB mode, the feedback mechanism is independent of the plaintext, resulting in a similar recovery as in CFB mode.

IMPLICATIONS:

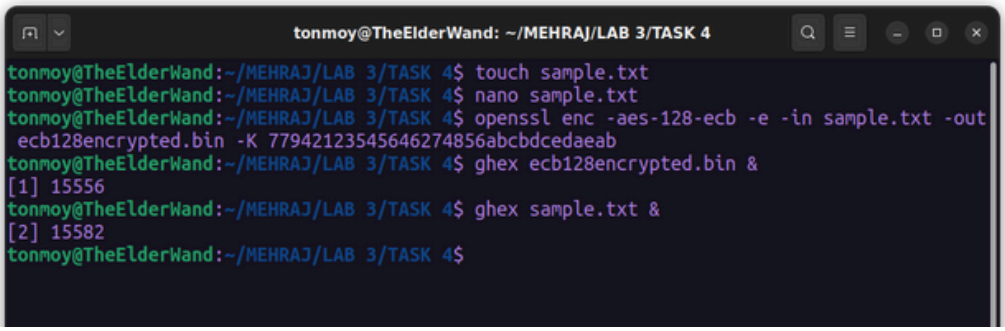
- ECB mode offers the least security and diffusion, as evident from the limited recoverable information.
- CBC mode provides better security compared to ECB but still suffers from partial corruption propagation.
- CFB offers improved diffusion, allowing for more recoverable information compared to CBC mode, although still susceptible to partial corruption propagation.
- OFB gives the best result among them, as it has only 1 corrupted bit and all other bits have been recovered being significant improvements.

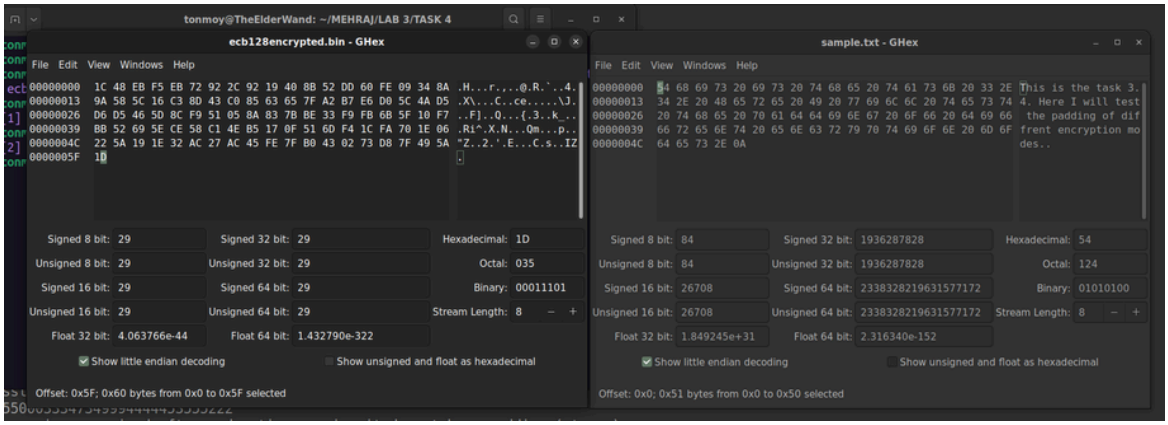
TASK 3.4: PADDING

Firstly I created a text file smaple.txt

FOR ECB

openssl enc -aes-128-ecb -e -in sample.txt -out ecb128encrypted.bin
-K 77942123545646274856abcbdcedaeab
ghex ecb128encrypted.bin &
ghex sample.txt &

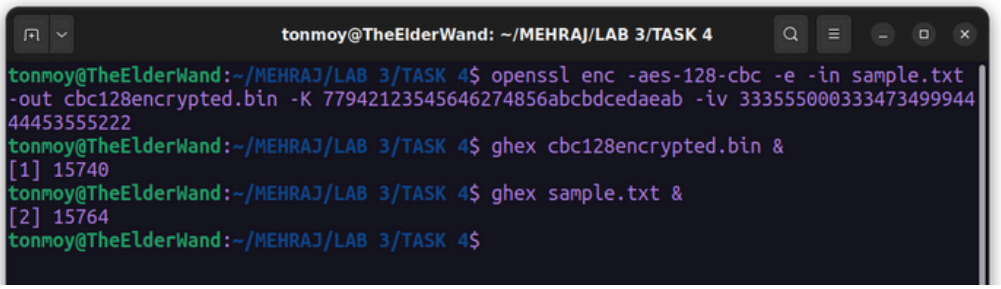


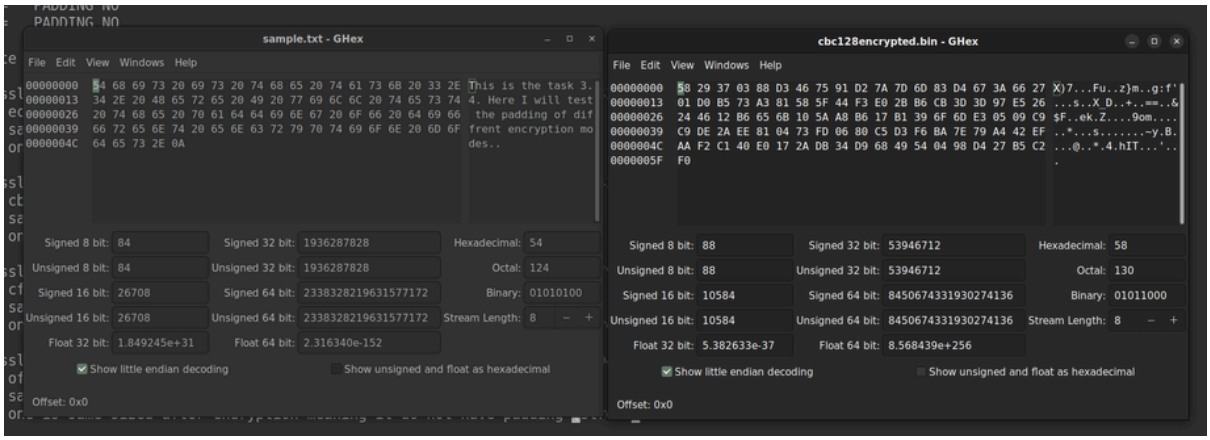


ECB generates larger output after encryption meaning it has padding

FOR CBC

openssl enc -aes-128-cbc -e -in sample.txt -out cbc128encrypted.bin
-K 7794212354564627485 6abcbdcedaeab -iv 33355500033347349994444453555222
ghex cbc128encrypted.bin &
ghex sample.txt &

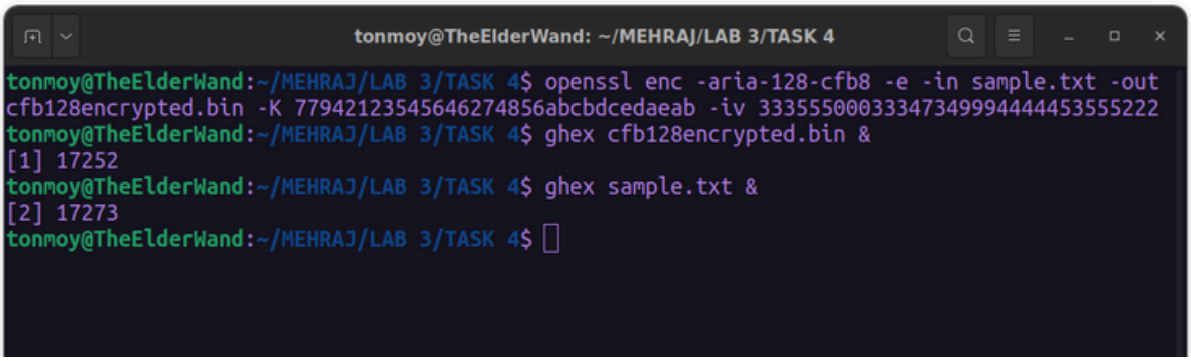


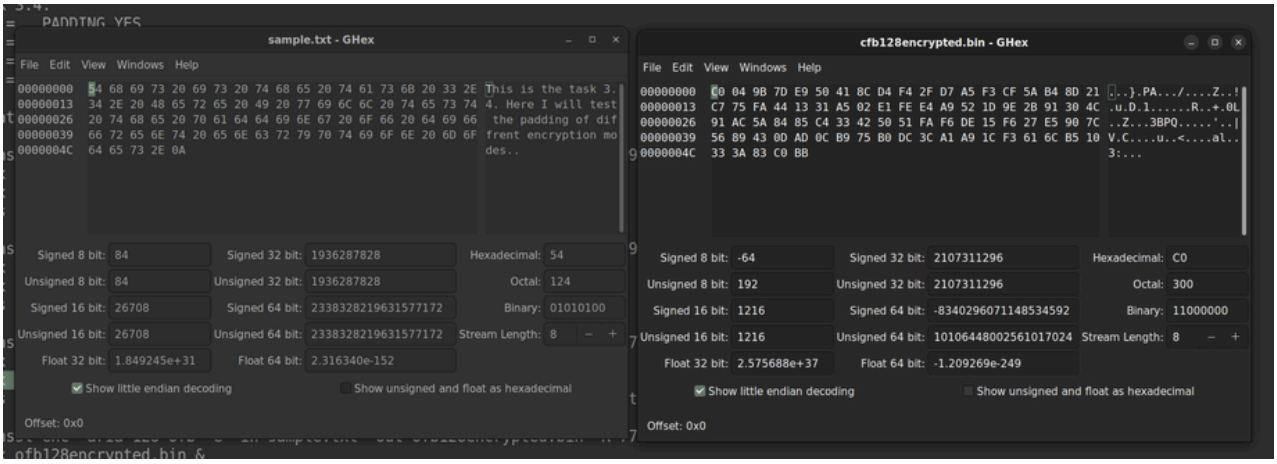


CBC generates larger output after encryption meaning it has padding

FOR CFB

openssl enc -aria-128-cfb8 -e -in sample.txt -out cfb128encrypted.bin
-K 7794212354564627485 6abcbdcedaeab -iv 33355500033347349994444453555222
ghex cfb128encrypted.bin &
ghex sample.txt &





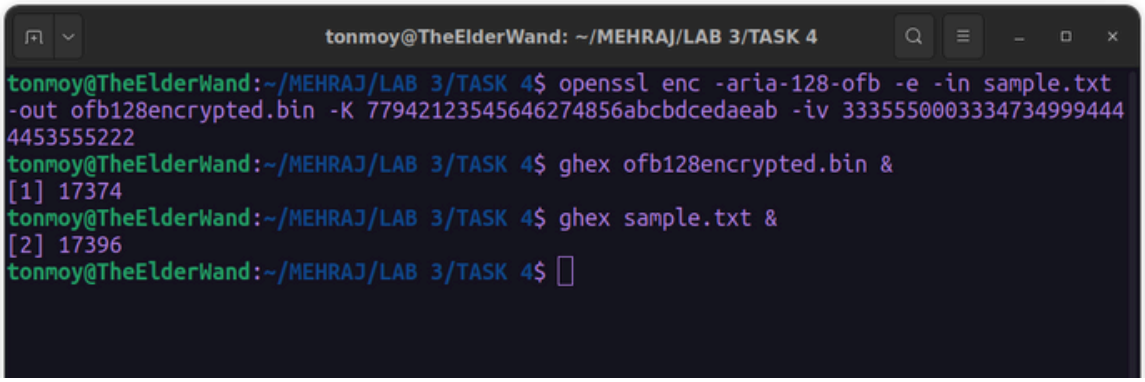
CFB do not generate larger output after encryption meaning it do not have padding

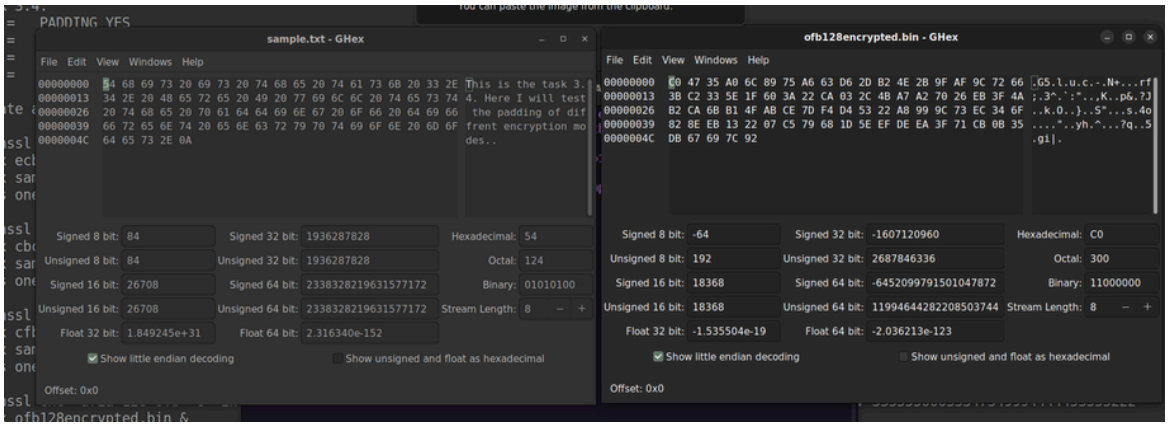
FOR OFB

openssl enc -aria-128-ofb -e -in sample.txt -out ofb128encrypted.bin -K 77942123545646274856abcbdcedaeab -iv 33355500033347349994444453555222

ghex ofb128encrypted.bin &

ghex sample.txt &





OFB do not generate larger output after encryption meaning it do not have padding

SO MY FINDINGS ARE:

ECB HAS PADDING
CBC HAS PADDING

CFB DON'T HAVE PADDING
OFB DON'T HAVE PADDING

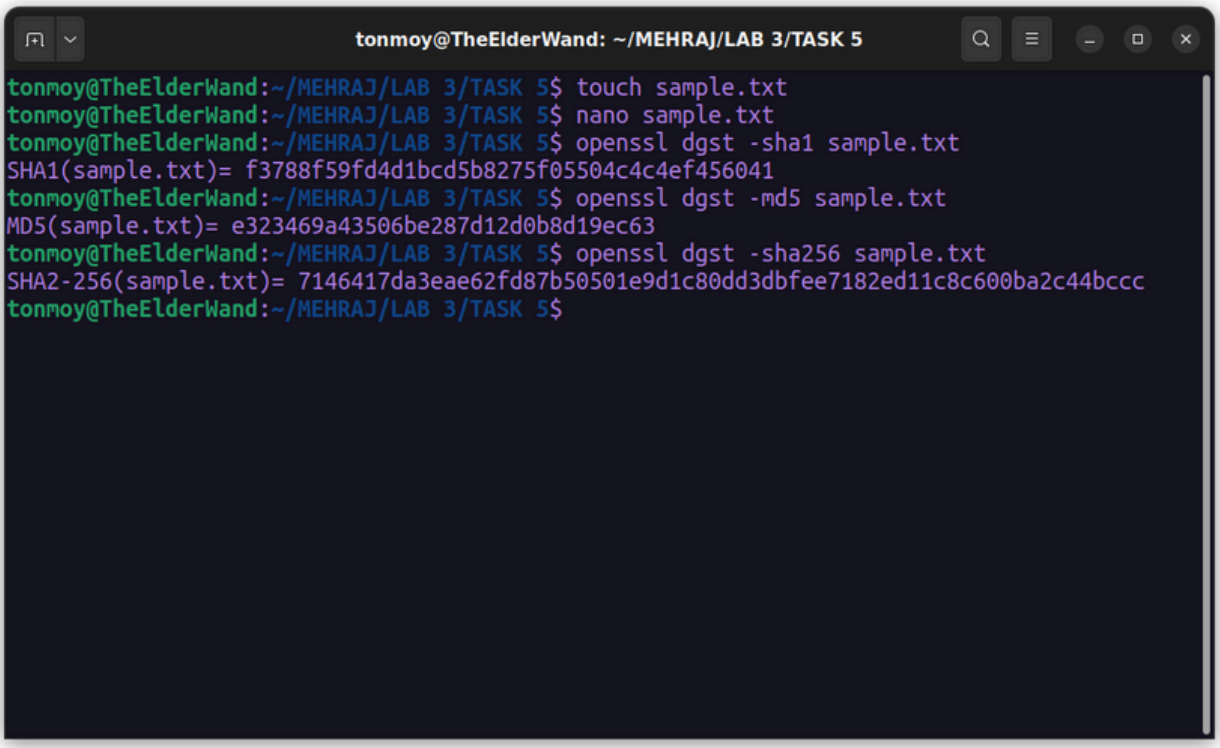
TASK 3.5: GENERATING MESSAGE DIGEST

I created a text file nammed sample.txt

openssl dgst -sha1 sample.txt
output: f3788f59fd4d1bcd5b8275f05504c4c4ef456041

openssl dgst -md5 sample.txt
output: e323469a43506be287d12d0b8d19ec63

openssl dgst -sha256 sample.txt
output: 7146417da3eae62fd87b50501e9d1c80dd3dbfee7182ed11c8c600ba2c44bccc



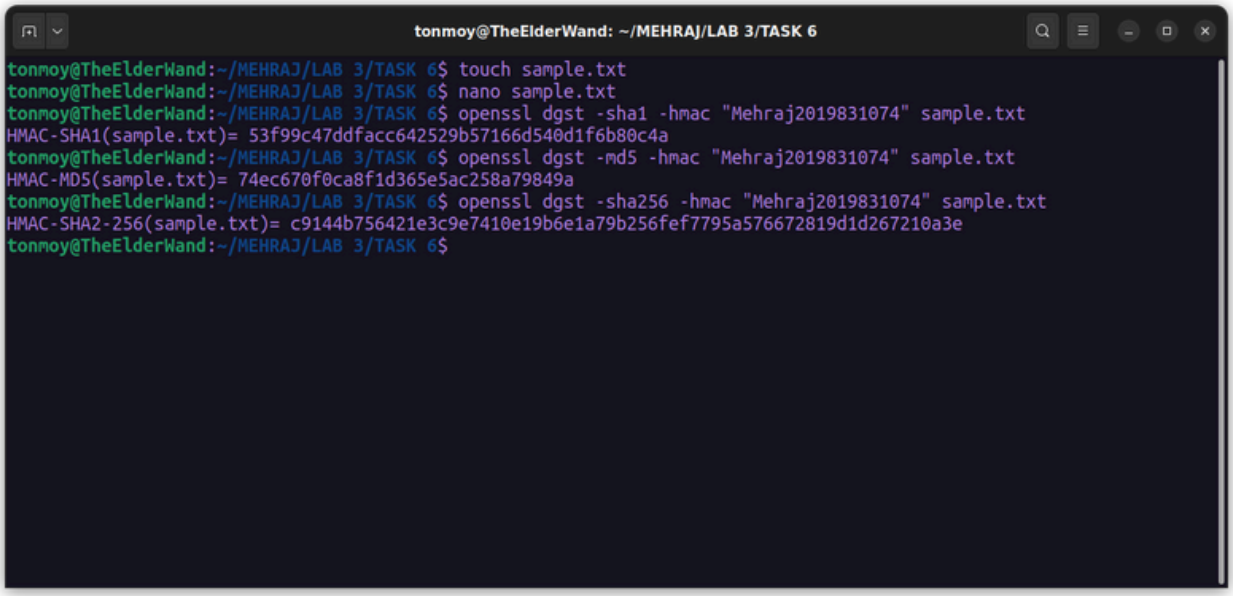
TASK 3.6: KEYED HASH AND HMAC

I created a text file nammed sample.txt

openssl dgst -sha1 -hmac "Mehraj2019831074" sample.txt
output: 53f99c47ddfacc642529b57166d540d1f6b80c4a

openssl dgst -md5 -hmac "Mehraj2019831074" sample.txt
output: 74ec670f0ca8f1d365e5ac258a79849a

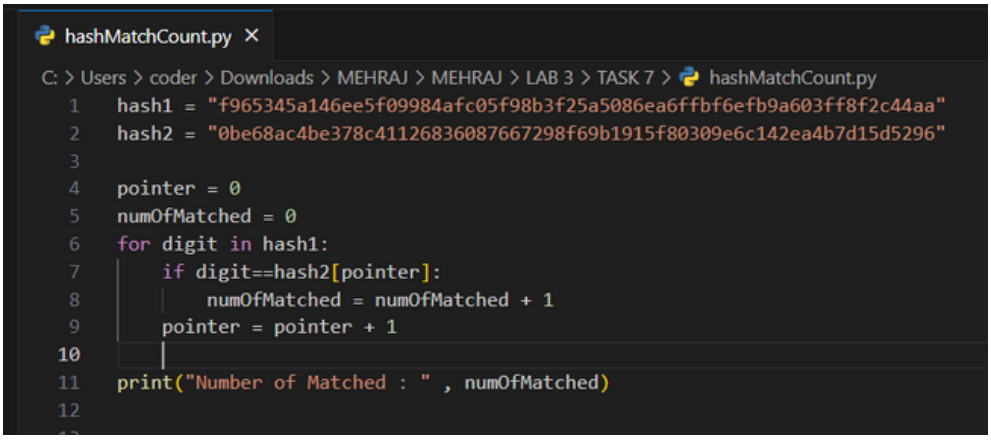
openssl dgst -sha256 -hmac "Mehraj2019831074" sample.txt
output: c9144b756421e3c9e7410e19b6e1a79b256fef7795a576672819d1d267210a3e



TASK 3.7: KEYED HASH AND HMAC

I created a text file nammed sample.txt

Then I created a program for counting hash match named hashMatchCount.py



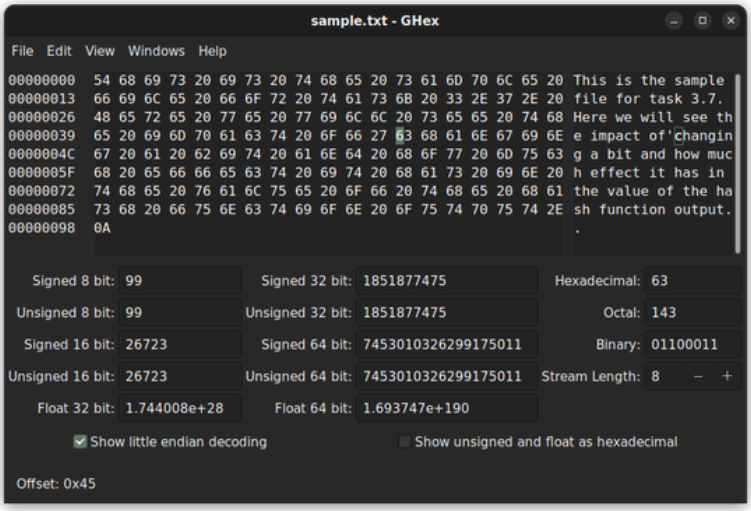
FOR MD5

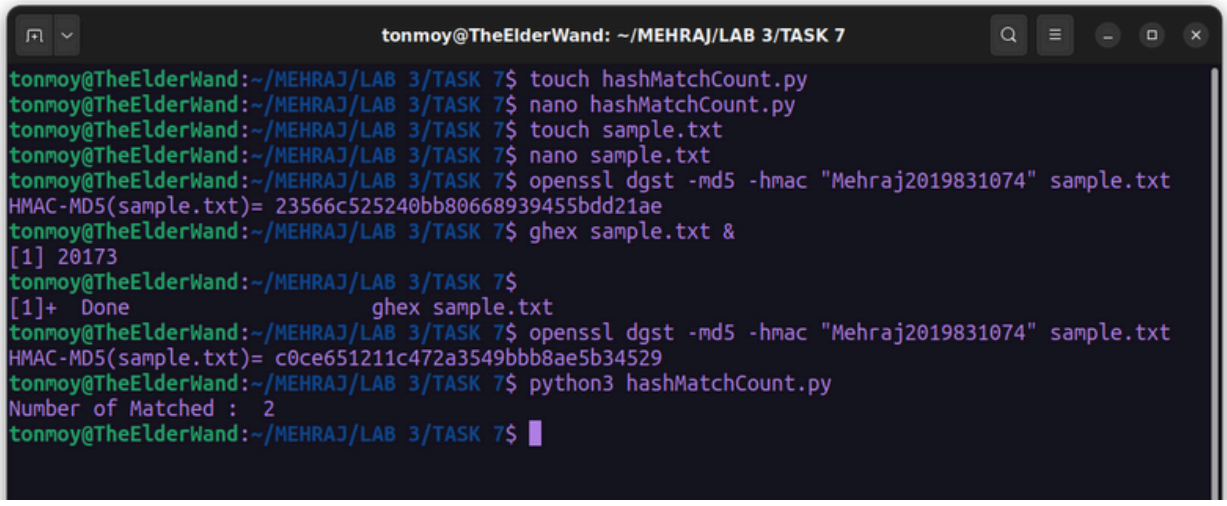
openssl dgst -md5 -hmac "Mehraj2019831074" sample.txt
output: **23566c525240bb80668939455bdd21ae**
ghex sample.txt &

Then I changed a random bit

openssl dgst -md5 -hmac "Mehraj2019831074" sample.txt
output: **c0ce651211c472a3549bbb8ae5b34529**

Then updated the python code (replaced hash values)





Here number of matched characters is: 2

FOR SHA256

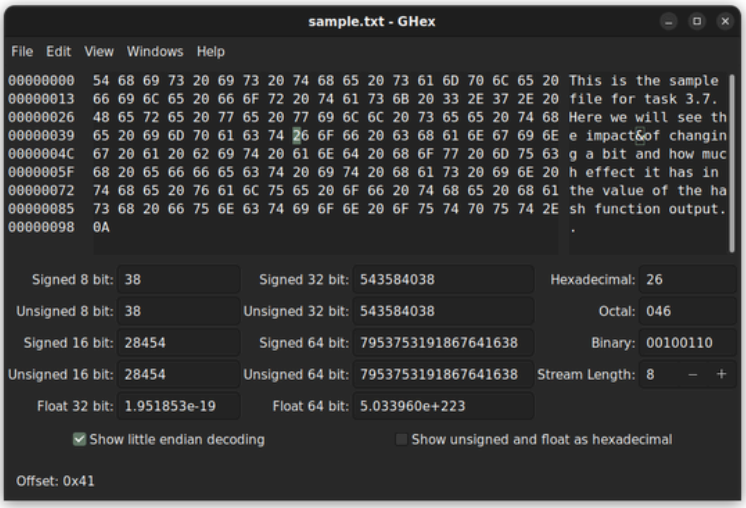
update the text file text.txt to the previous main content

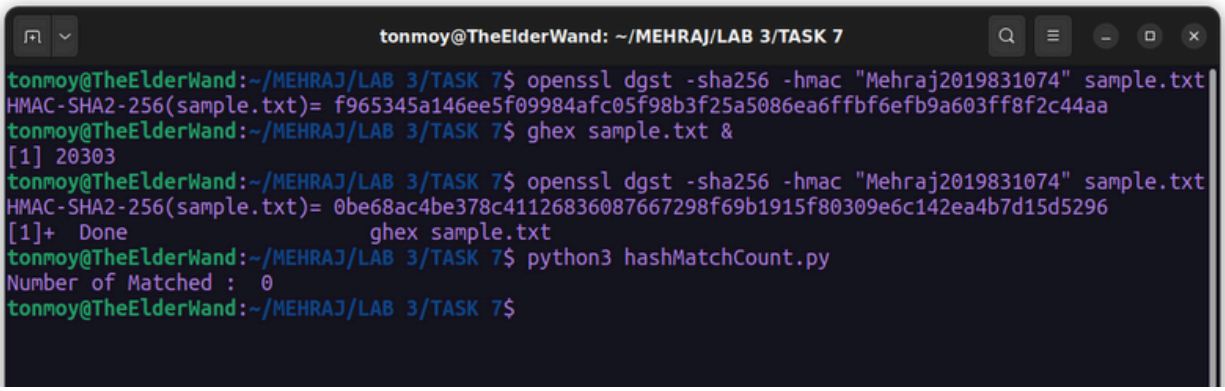
openssl dgst -sha256 -hmac "Mehraj2019831074" text.txt
output:
923b829a244a667348611ffe93016ddf798224b71a0ad6c650e68f9e10cd1c6e
ghex text.txt &

Then I changed a random bit

openssl dgst -sha256 -hmac "Mehraj2019831074" text.txt
output:
5f4a04044aa5e14da6887551c00326cac4abba24355c076c38fb9f8dd6151e8f

Then updated the python code (replaced hash values)





Here number of matched characters is: 0

So, changing even a bit changes the hash value completely