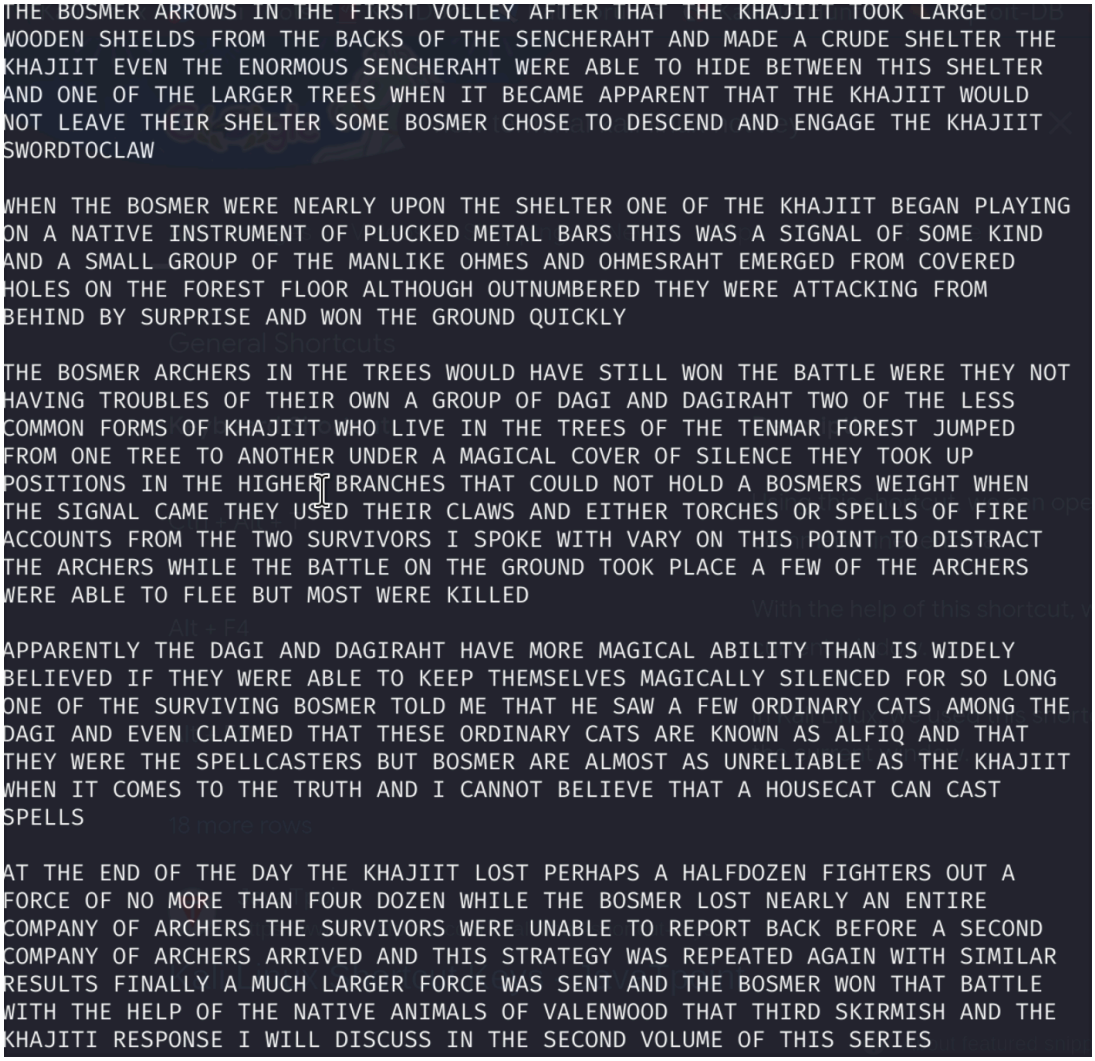
Assignment 1

Task 1: Frequency Analysis

Final Command that Produced Correct Answer”

- tr 'zgtwuhxjmkslaoqecrfdvibypn' 'THEAIOCKNGSRDPUWLYFVMXBZJQ' <ciphertext.txt> out.txt
- Command was built one letter at a time built off of the deciphering of the word “THE”
  - Started with single letters, moved to diagrams and trigrams
  - Common digrams were completed based on guess and check (IN, ON, OF, etc)
  - Some of the Skyrim words I looked up after discovering the name of the piece of writing

```
MIXED UNIT TACTICS IN THE FIVE YEARS WAR

VOLUME ONE

BY CODUS CALLONUS

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

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

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

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

THE BOSMER ARCHERS TOOK UP POSITIONS IN THE REMAINING TREES WHOSE BRANCHES
WERE NOW TWENTY OR MORE FEET APART ALLOWING SOME LIGHT INTO THE FOREST FLOOR
THE BOSMER BENT THE REMAINING TREES WITH THEIR MAGICS INTO SMALL
FORTIFICATIONS FROM WHICH TO FIRE THEIR BOWS

WHEN THE TREECUTTERS ARRIVED THE NEXT MORNING A HALF DOZEN KHAJIIT FELL TO
THE BOSMER ARROWS IN THE FIRST VOLLEY AFTER THAT THE KHAJIIT TOOK LARGE
WOODEN SHIELDS FROM THE BACKS OF THE SENCHERAHT AND MADE A CRUDE SHELTER THE
KHAJIIT EVEN THE ENORMOUS SENCHERAHT WERE ABLE TO HIDE BETWEEN THIS SHELTER
AND ONE OF THE LARGER TREES WHEN IT BECAME APPARENT THAT THE KHAJIIT WOULD
NOT LEAVE THEIR SHELTER SOME BOSMER CHOSE TO DESCEND AND ENGAGE THE KHAJIIT
```



Task 2: Encryption using Different Ciphers and Modes

AES-256-CFB

```
┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ openssl enc -aes-256-cbc -e -in plain.txt -out cipher.bin
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ cat plain.txt
Hello World

┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ cat cipher.bin
Salted__2gr◆◆m◆^◆m◆[◆
◆j◆?◆◆
```

```
┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ openssl enc -aes-256-cbc -d -in cipher.bin -out output.txt
enter AES-256-CBC decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ cat output.txt
Hello World
```

openssl enc -aes256-cbc -e -in plain.txt -out cipher.bin

- Plain.txt contained Hello World
- Only requires -e/-d option and input/output param

AES-128-CFB

```
┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ openssl enc -aes-128-cfb -in plain.txt -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ cat plain.txt
Hello World

┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ cat cipher.bin
◆E◆UëB
```

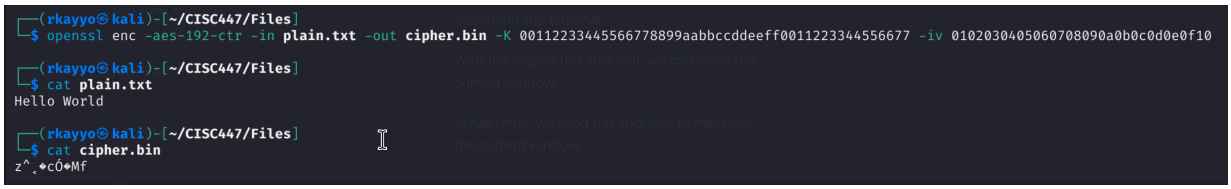
```
┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ openssl enc -aes-128-cfb -d -in cipher.bin -out output.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ cat output.txt
Hello World
```

openssl enc -aes-128-cfb -in plain.txt -out cipher.bin -K 00112233445566778899aabbccddeeff -iv

0102030405060708090a0b0c0d0e0f10

- Plain.txt contains Hello World
- Requires a 128 bit Key and IV

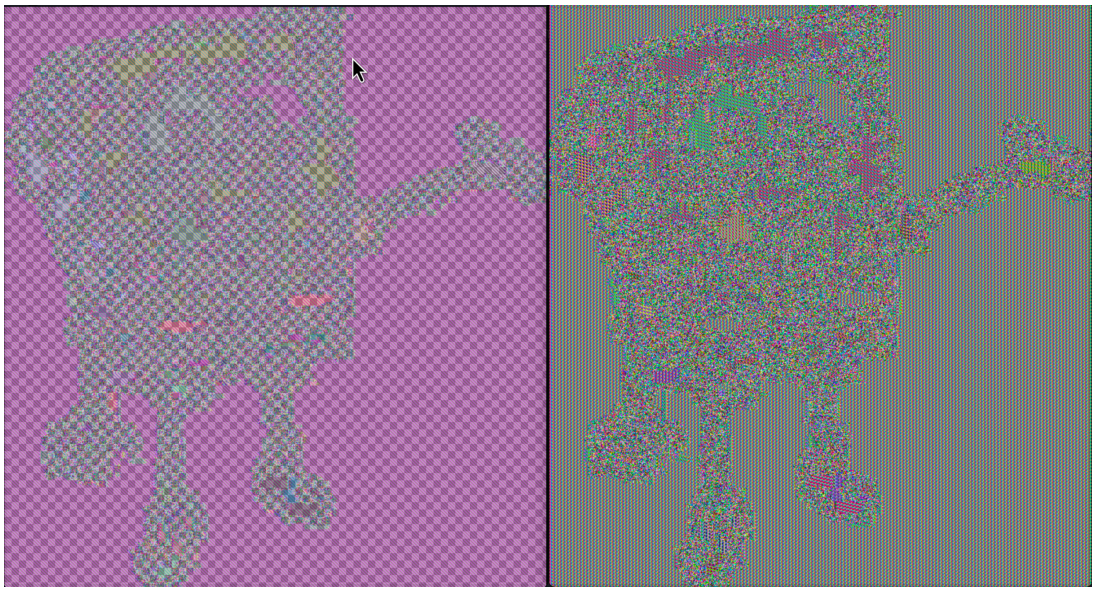
AES-192-CTR



openssl enc -aes-192-ctr -in plaintext.txt -out encrypted.dat -K 00112233445566778899aabbccddeeff0011223344556677 -iv 0102030405060708090a0b0c0d0e0f10

- Plain.txt contains Hello World
- Aes-192-ctr requires 192 bit Key and 128 bit IV

Task 3: Encryption Mode ECB vs CBC



Part 2

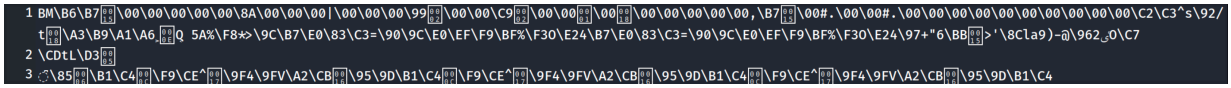
The most noticeable thing about the encrypted versions of both images is the shape of the character being portrayed. This is impart due to ecb encryption not jumbling the order of any of the ciphertext

Furthermore, assumptions can be made about the colouring in both images. Both the backgrounds are completely one colour, so we know that to be true of the original image. And, we can make out minor shapes within the image where the gaps in the character are located.

Part 3

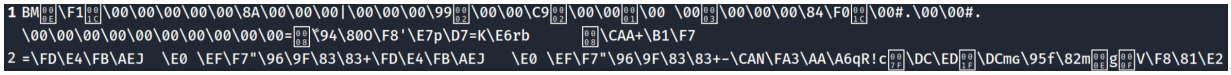
Both images share the same shape, due to ECB encryption. Because it does not mix the order of encrypted blocks, both images have the same shapes. But, due to the factors I will explain, they have different colours. I believe the difference in the 2 images comes from the differing of the bytes of the 2 headers appended at the beginning. And furthermore, the starting hex values of what I believe to be the background are also different.

F9/9F4/95 are common starting hex numbers for every line of text in new2.bmp while B1/F4Z/B7 are the common starting hex numbers of new1.bmp. The differing common hex values result in the pattern differences of the 2 encrypted images.

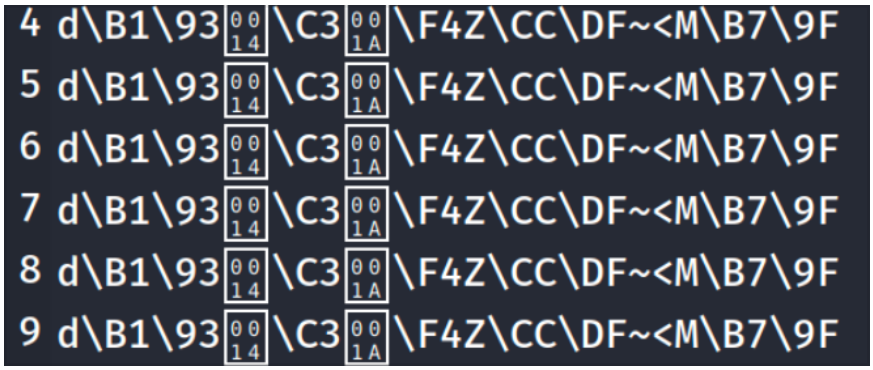


New2.bmp (right side) header

new2.bmp (right side) body



New1.bmp (left side) header



New1.bmp (left side) body

But, while the hex values control the colours, the formatting/order of each encrypted block is the same - leading to the same shape different colours we see above

Task 4
Using AES

AES-128-ECB - requires padding - due to the requirement of needing the input to be an exact multiple of the block size, aes-128-ecb will require padding if the plaintext to be encrypted is not an exact multiple
AES-128-CBC - same reasoning as above mentioned ECB

AES-128-CFB - does not require padding - because CFB acts as a stream cipher and not the normal block cipher of AES, aes-128-cfb can encrypt data bit by bit or multiple bytes a a time. Therefore it does not require padding
AES-128-OFB - same reasoning as above mentioned CFB

```
┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ hexdump -C task4F1.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
```

The 11 0b's are the padding left from when the file was encrypted

- The 5 byte file adds 11 bytes of padding

```
┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ hexdump -C task4F2.txt
00000000  31 32 33 34 35 36 37 38  39 30 0a 05 05 05 05 05  |1234567890......|
00000010
```

The 0a 05 … are the padding left from when the file was encrypted

- The 10 byte file adds 6 bytes of padding

```
┌──(rkayyo㉿kali)-[~/CISC447/Files]
└─$ hexdump -C task4F3.txt
00000000  31 32 33 34 35 36 37 38  39 30 31 32 33 34 35 36  |1234567890123456|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
```

The 16 byte file adds another 16 bytes of padding to make it 32 bytes long

- Padding is required no matter if the block is an exact multiple so that the decryption algorithm can properly determine how much padding to remove

Task 5

Presumption - ECB only specific block with error is wrong, CBC every block after erroneous block is ruined, cfb and ofb every byte after corrupted byte in keystream is ruined

Used a random text file generator to make a 1000 byte long file

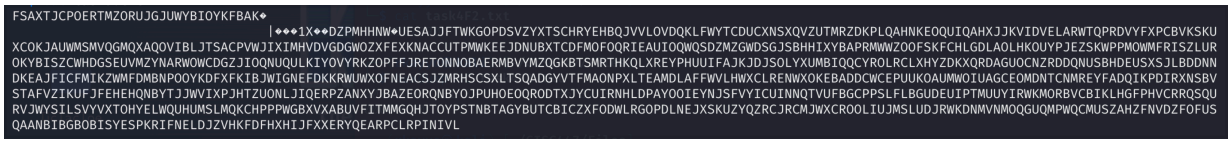
File before aes-128-ecb

- Highlighted segment is the 55th byte

```
FSAXTJCPOERTMZORUJGJUWYBIOYKFBAKOQLMFOPRNUMPDFIDZPMHHNWJUESAJJFTWKGOPDSVZYXTSCHRYEHBQJVVLOVDQKLFWYTCDUCXNSXQVZ
UTMRZDKPLQAHNKEOQUIQAHXJJKVIDVELARWTQPRDVYFXPCBVKSKUXCOKJAUWMSMVQGMQXAQOVIBLJTSACPVWJIXIMHVDVGDGWOZXFEXKNACCUT
PMWKEEJDNUBXTCDFMOFOQRIEAUIOQWQSDZMZGWDSGJSBHHIXYBAPRMWWZOOFSKFCHLGDLAOLHKOUYPJEZSKWPPMOWMFRISZLUROKYBISZCWHDG
SEUVMZYNARWOWCDGZJIOQNUQULKIYOVYRKZOPFFJRETONNOBAERMBVYMZQGKBTSMRTHKQLXREYPHUUIFAJKJDJSOLYXUMBIQQCYROLRCLXHYZD
KXQRDAGUOCNZRDDQNUSBHDEUSXSJLBDDNNDKEAJFICFMIKZWMFDMBNPOOYKDFXFKIBJWIGNEFDKKRWUWXOFNEACSJZMRHSCSXLTSQADGYVTFMA
ONPXLTEAMDLAFFWVLHWXCLRENWXOKEBADDCWCEPUUKOAUMWOIUAGCEOMDNTCNMREYFADQIKPDIRXNSBVSTAFVZIKUFJFEHEHQNBYTJJWVIXPJH
TZUONLJIQERPZANXYJBAZEORQNBYOJPUHOEOQRODTXJYCUIRNHLDPAYOOIEYNJSFVYICUINNQTVUFBGCPPSLFLBGUDEUIPTMUUYIRWKMORBVCB
IKLHGFPHVCRRQSQURVJWYSILSVYVXTOHYELWQUHUMSLMQKCHPPPWGBXVXABUVFITMMGQHJTOYPSTNBTAGYBUTCBICZXFODWLRGOPDLNEJXSKUZ
YQZRCJRCMJWXCROOLIUJMSLUDJRWKDNMVNMOQGUQMPWQCMUSZAHZFNVDZFOFUSQAANBIBGBOBISYESPKRIFNELDJZVHKFDFHXHIJFXXERYQEAR
PCLRPINIVL
```

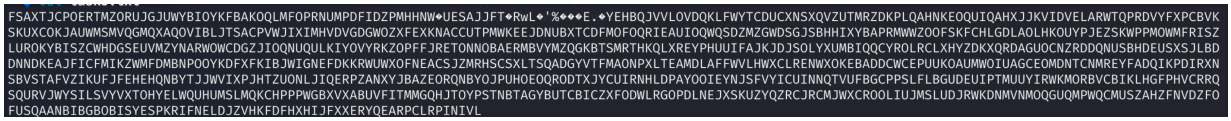
```
FSAXTJCPOERTMZORUJGJUWYBIOYKFBAKF◆K◆}!◆◆◆◆/`◆ZPMHHNWJUESAJJFTWKGOPDSVZYXTSCHRYEHBQJVVLOVDQKLFWYTCDUCXNSXQVZUTMRZDKPLQAHNKEOQUIQAHXJJKVIDVELARWTQPRDVYFXPCBVK
SKUXCOKJAUWMSMVQGMQXAQOVIBLJTSACPVWJIXIMHVDVGDGWOZXFEXKNACCUTPMWR̈EEJDNUBXTCDFMOFOQRIEAUIOQWQSDZMZGWDSGJSBHHIXYBAPRMWWZOOFSKFCHLGDLAOLHKOUYPJEZSKWPPMOWMFRISZ
LUROKYBISZCWHDGSEUVMZYNARWOWCDGZJIOQNUQULKIYOVYRKZOPFFJRETONNOBAERMBVYMZQGKBTSMRTHKQLXREYPHUUIFAJKJDJSOLYXUMBIQQCYROLRCLXHYZDKXQRDAGUOCNZRDDQNUSBHDEUSXSJLBD
DNNDKEAJFICFMIKZWMFDMBNPOOYKDFXFKIBJWIGNEFDKKRWUWXOFNEACSJZMRHSCSXLTSQADGYVTFMAONPXLTEAMDLAFFWVLHWXCLRENWXOKEBADDCWCEPUUKOAUMWOIUAGCEOMDNTCNMREYFADQIKPDIRXN
SBVSTAFVZIKUFJFEHEHQNBYTJJWVIXPJHTZUONLJIQERPZANXYJBAZEORQNBYOJPUHOEOQRODTXJYCUIRNHLDPAYOOIEYNJSFVYICUINNQTVUFBGCPPSLFLBGUDEUIPTMUUYIRWKMORBVCBIKLHGFPHVCRRQ
SQURVJWYSILSVYVXTOHYELWQUHUMSLMQKCHPPPWGBXVXABUVFITMMGQHJTOYPSTNBTAGYBUTCBICZXFODWLRGOPDLNEJXSKUZYQZRCJRCMJWXCROOLIUJMSLUDJRWKDNMVNMOQGUQMPWQCMUSZAHZFNVDZFO
FUSQAANBIBGBOBISYESPKRIFNELDJZVHKFDFHXHIJFXXERYQEARPCLRPINIVL
```

Only the block containing the 55th bit is corrupted, the rest is the same - ecb encrypts plaintext block by block. Each block has no impact on the next. So the corrupted block stays corrupted whilst not affecting any other blocks

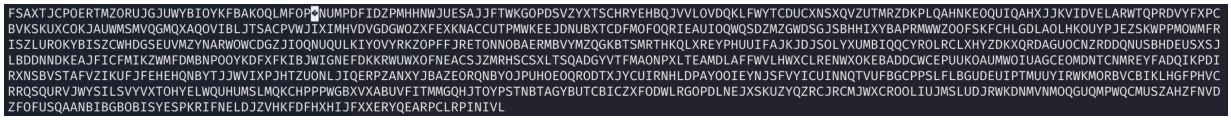


Because cbc blocks chain together their errors propagate to the next block. Therefore, when using cbc, the full 16 byte block will be corrupted and additionally, a single byte from the next block.

Using cfb I assumed each byte after the corrupted one would be left corrupted because of CFB's nature. While I am pretty sure this is the case, the resulting decrypted plain text came out like this. I am not super sure if I did it correctly but this is the output I was given.



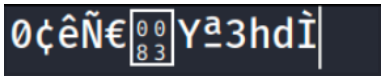
Using OFB



Only the singular bit corrupted stayed corrupted using OFB - The key stream is generated independently of the plaintext or ciphertext, and XORed with the plaintext to produce ciphertext

Task 6

6.1 (1)
Using different iv's (aes-128-cfb)

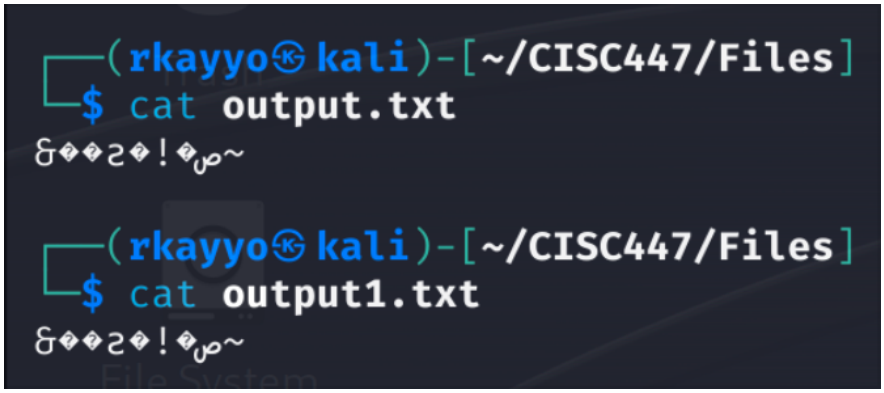


iv = 11223344556677889aabbccddeeff00

iv = 123456789123456789aaffbbeeccdd00

Each encryption has 12 characters, but the characters are seemingly random and unrelated. Both unrelated to the other ciphertext and to the characters within its own cipher text

6.2 (2)
Using the same iv's (aes-128-cfb)



Each iv creates a uniquely encrypted ciphertext, when the same iv is used, the same ciphertext is produced. This is why iv needs to be random and unique

Due to OFB's properties, when using the same key and iv, the result of c1 xor c2 = p1 xor p2
From this, because we know p1, p1 xor (p1 xor p2) = p2
Using xxd -r -p (-r reverses effect and transforms hex to binary/original form) (-p specifies plain hex formatting)

Task 6.3
Ive noticed that after a certain number of iterations ~3/4, the 10th bit increments by 1
ive discovered that the iv xor with yes or no gives an almost identical hexadecimal code. in regards to xor'ing with yes, the last 6 digits change and in regards to no, the last 5 digits change. Although, when those 5/6 digits are xor'd with the last 5/6 digits of the original iv they output Yes or No

Attempts to Solve:

1. I tried xor'ing the previous and next IV, then xor'ing the trailing digits with both Yes and No to try and find a pattern within the given ciphertext that related to the xor'd digits.
2. I also attempted xor'ing the two cipher texts together then xor'ing the given value with the hex of both Yes and No to try and recover the plaintext.
3. Because we know the last 24 bits of the IVs stay the same for a certain number of inputs, I thought that the last 24 bits of the ciphertext might

be very similar, or similar if I xor'd them

4. After xor'ing Bob's ciphertext with my own created from inputting either 5965730A0A0A0A0A (Yes with PKCS#7 padding) or 4E6F0C0C0C0C0C0C (No with PKCS#7 padding), I then xor'd the resulting hex string with the same strings correlating to Yes and No. After trying this with a plaintext input of both (Yes or No) the final xor product, with ciphertext generated from Yes or No were the same. 170a7f0606060606

Answer: Original Plaintext is Yes. From line 4 - the string of 06's acts as padding, and 17 0a 7f is 3 bytes. Yes is 3 bytes. I believe this is an encrypted version of the word Yes.

```
┌──(root㉿rkayyo)-[~]
└─# nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 4184dca3a8e17fd2a62c6f6c5d588fde
The IV used    : 4981608a5aee160533ed4ab2c5710719

Next IV        : 50d882a95aee160533ed4ab2c5710719
Your plaintext : 596573
Your ciphertext: 0d2c3c5f4ce11b06531dbe44f2d6631c

Next IV        : 11f73eaf5aee160533ed4ab2c5710719
Your plaintext : 4E6F
Your ciphertext: bf2da40302b5f7badc777a03ae28f545

Next IV        : 68b4fddc5aee160533ed4ab2c5710719
Your plaintext : 5965730A0A0A0A0A
Your ciphertext: 43525c4e7efab56f62273cf52b997d76

Next IV        : 220f1aea5aee160533ed4ab2c5710719
Your plaintext : ^C

┌──(root㉿rkayyo)-[~]
└─# nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 4fec6eda38f7ae65d7c68cf2d67bf39b
The IV used    : 6609864c116745f7676e2cbebeddf6f2

Next IV        : 95ea3850116745f7676e2cbebeddf6f2
Your plaintext : 5965730A0A0A0A0A
Your ciphertext: 8912e1cf6e124da1a77a8006e44c295f

Next IV        : 49003a9c116745f7676e2cbebeddf6f2
Your plaintext : ^C

┌──(root㉿rkayyo)-[~]
└─# nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: c34049fb53534c9938315e237d4b1377
The IV used    : 52f7b64b79d9ca101ea76c2b5b409c0c

Next IV        : 922fdd7a79d9ca101ea76c2b5b409c0c
Your plaintext : 5965730A0A0A0A0A
Your ciphertext: d0de5c1d7b258f92c456ac88976bda64

Next IV        : cc5ea2ec79d9ca101ea76c2b5b409c0c
Your plaintext : 4E6F0C0C0C0C0C0C
Your ciphertext: 4f5c1fed8f99d8ac95c4985fd252e40a

Next IV        : 4fae803f7ad9ca101ea76c2b5b409c0c
Your plaintext : ^C
```

I tried many different attempts at xor'ing different values together to try and interpret a pattern. I don't think I understand CBC well enough to come to a complete answer.

Task 7

I did not leave myself enough time to solve this due to the difficulty of task 6.3 I will manage my time on the next assignment better.