CPS 633 Section 09

Secret-Key Encryption Lab

Group 18

Roxie Reginold (501087897)
Hetu Virajkumar Patel (501215707)
Sayyada Aisha Mehvish (501106795)

TASK 1: Frequency Analysis

This task involved decrypting an encrypted file using frequency analysis.

We ran the freq.py script on the encrypted file, which generated statistics for n-grams, including single letter, bigram, and trigram frequencies.

Running freq.py script

```
[09/26/24]seed@VM:~/.../Files$ ./freq.py
----------------------------------
1-gram (top 20):
n: 488
y: 373
v: 348
x: 291
u: 280
q: 276
m: 264
h: 235
t: 183
i: 166
p: 156
a: 116
c: 104
z: 95
l: 90
g: 83
b: 83
r: 82
e: 76
d: 59
```

```
2-gram (top 20):
yt: 115
tn: 89
mu: 74
nh: 58
vh: 57
hn: 57
vu: 56
nq: 53
xu: 52
up: 46
xh: 45
yn: 44
np: 44
vy: 44
nu: 42
qy: 39
vq: 33
vi: 32
gn: 32
av: 31
```

```
3-gram (top 20):
ytn: 78
vup: 30
mur: 20
ynh: 18
xzy: 16
mxu: 14
gnq: 14
ytv: 13
nqy: 13
vii: 13
bxh: 13
lvq: 12
nuy: 12
vyn: 12
uvy: 11
lmu: 11
nvh: 11
cmu: 11
tmq: 10
vhp: 10
```

We compared the script's output with standard frequency statistics for English language n-grams, referencing:
https://en.wikipedia.org/wiki/Bigram
https://en.wikipedia.org/wiki/Trigram

By comparing our frequency analysis results with standard English language patterns, we made initial inferences:

- n likely corresponds to E (the most common letter in English)
- yt probably represents TH (a common bigram)
- ytn likely stands for THE (the most frequent trigram)

We progressively substituted encrypted characters with their probable plaintext counterparts, refining our substitutions as more text became readable.

This table shows the mapping between the encrypted letters to their respective plaintext letters:

Letter to encrypt	Plain text Letter		Letter to encrypt	Plain text Letter
y	T		t	H
n	E		v	A
d	Y		u	N
p	D		q	S
z	U		a	C

h	R		l	W
m	I		x	O
f	V		g	B
e	P		i	L
r	G		b	F
s	K		j	Q
o	J		k	X
w	Z		c	M

Throughout this process, we utilized the tr command to perform character substitutions. Our final decryption command was:

tr 'nyvtimuqxpclahzdfjgrebsokw' 'ETAHLINSODMWCRUYVQBGPFKJXZ' < ciphertext.txt > out1.txt

Output produced after running the final decryption command:

```
[09/28/24]seed@VM:~/.../Files$ cat out1.txt
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
```

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

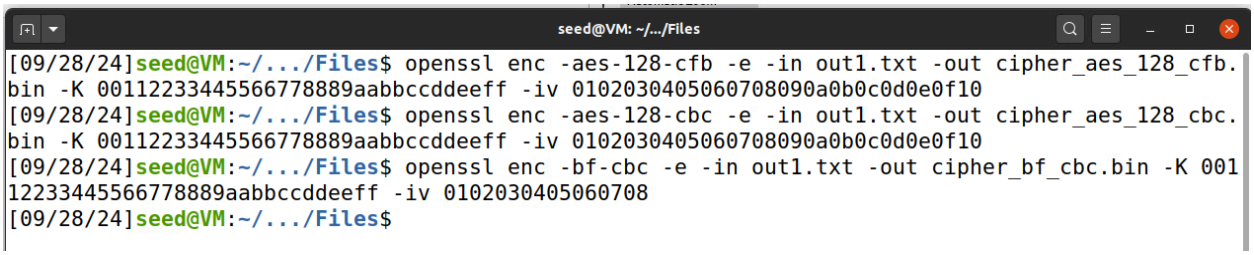
TASK 2: Encryption using Different Ciphers and Modes

OpenSSL is a toolkit for SSL and TLS protocols in which the library contains cryptography functions like operations, creating and managing private keys, public keys and parameters.

We ran the output from task1 and encrypted it using 3 different methods:

- -aes-128-cbc
- -aes-128-cfb
- -bf-cbc

Screenshot of encryption via terminal using different methods:



AES-128-CBC

AES-128-CBC (Advanced Encryption Standard with a 128-bit key in Cipher Block Chaining mode) is a symmetric encryption method that processes data in blocks of 128 bits. It uses a 128-bit key for both encryption and decryption. In CBC mode, each block of plaintext is combined with the previous ciphertext block using an XOR operation before encryption. The first block uses an Initialization Vector (IV) to start the process. This method helps hide patterns in the plaintext, making it more secure than simpler modes like ECB.

BF-CBC

BF-CBC (Blowfish in Cipher Block Chaining mode) is a symmetric encryption method that works with 64-bit blocks of data. It uses a variable-length key ranging from 32 to 448 bits. Like AES-CBC, Blowfish in CBC mode combines each plaintext block with the previous ciphertext block using XOR before encrypting it.

AES-128-CFB

AES-128-CFB (AES with a 128-bit key in Cipher Feedback mode) also uses the AES algorithm but operates differently. In CFB mode, the previous ciphertext is encrypted and then combined with the current plaintext using XOR to produce the next ciphertext block. This turns the block cipher into a stream cipher, allowing for the encryption of data in smaller units than the block size. CFB mode does not require padding.

Task 3: Encryption Mode – ECB vs. CBC

In this task, we were given a .bmp-type image. We had to encrypt it using the ECB encryption mode and CBC encryption mode.

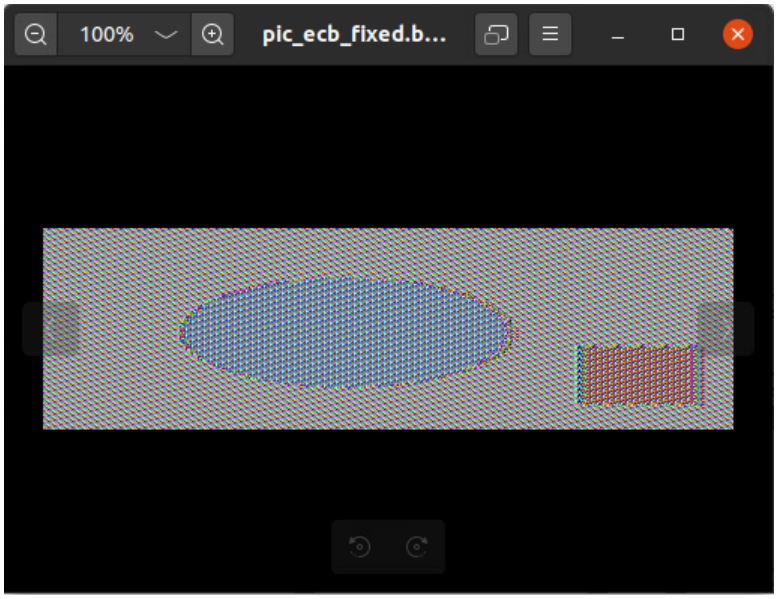
The following commands were run to encrypt this picture using the ECB and CBC mode.

```
[09/28/24]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.bmp
 -K 00112233445566778889aabbccddeeff
[09/28/24]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.bmp
 -K 00112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
[09/28/24]seed@VM:~/.../Files$ head -c 54 pic_original.bmp > header
[09/28/24]seed@VM:~/.../Files$ tail -c +55 pic_ecb.bmp > body_ecb
[09/28/24]seed@VM:~/.../Files$ tail -c +55 pic_cbc.bmp > body_cbc
[09/28/24]seed@VM:~/.../Files$ tail -c +55 pic_cbc.bmp > body_cbc_fixed.bmp
[09/28/24]seed@VM:~/.../Files$ cat header body_ecb > pic_ecb_fixed.bmp
[09/28/24]seed@VM:~/.../Files$ cat header body_cbc > pic_cbc_fixed.bmp
[09/28/24]seed@VM:~/.../Files$ eog pic_ecb_fixed.bmp
[09/28/24]seed@VM:~/.../Files$ eog pic_cbc_fixed.bmp
[09/28/24]seed@VM:~/.../Files$ eog pic_original.bmp
[09/28/24]seed@VM:~/.../Files$ eog pic_ecb_fixed.bmp
[09/28/24]seed@VM:~/.../Files$ eog pic_cbc_fixed.bmp
```

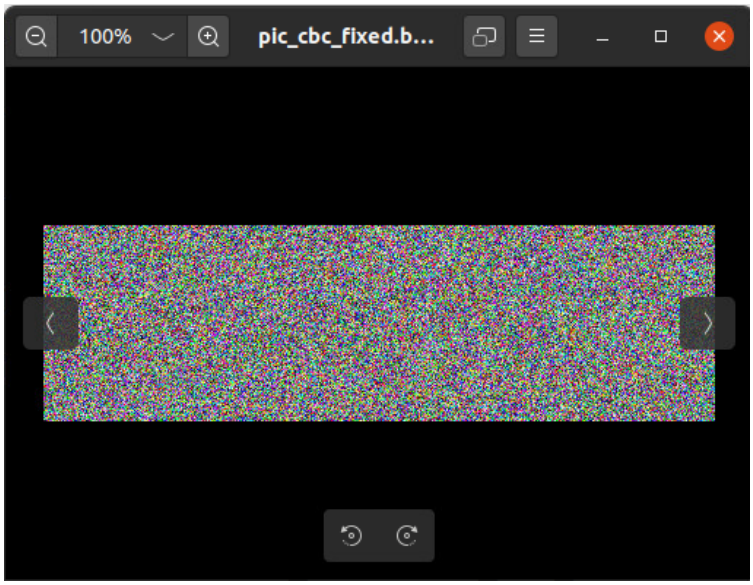
In this process, we see that all the image bytes are encrypted but we just want to cipher them from the 55th byte onwards. Therefore we grab the first 54 bytes of the original image (since that is the header for a .bmp file) and append the bytes from 55 offset to the rest of the file. This results in no errors when displaying the encrypted image.

Original image:

Observations for ECB mode:



Observations for CBC mode:



Results:

Based on our observations, we can conclude that in CBC (Cipher Block Chaining) mode, the encrypted image is difficult to distinguish from the original image. However, in ECB (Electronic Code Book) mode, the picture can still be recognized. This difference occurs because:

- In ECB mode, each block of data is encrypted separately.
- In CBC mode, the encryption of each block depends on the previous block.

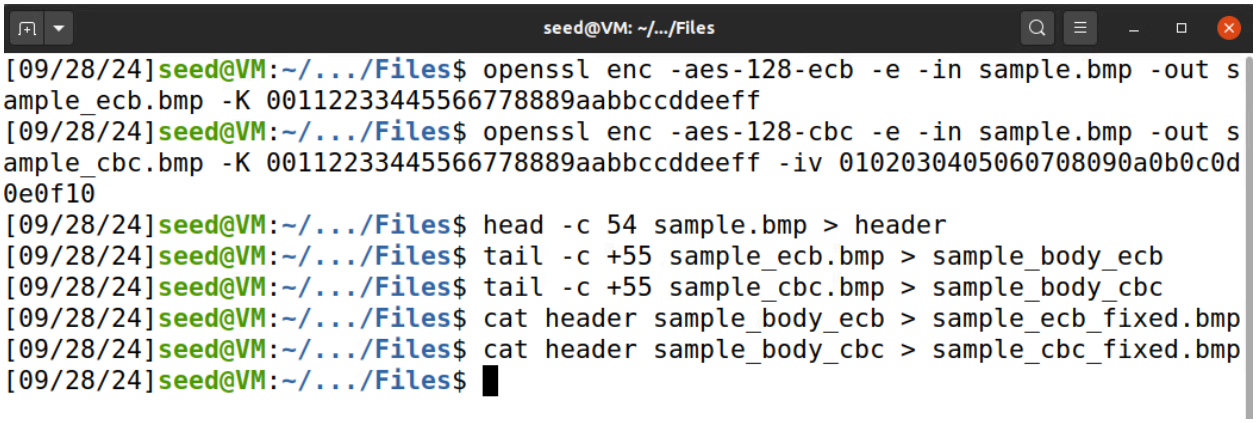
For CBC, the process works like this:

- The system takes the encrypted data from the previous block.
- It combines this with the current block's data using an XOR operation.
- The result is then encrypted.

This chaining process makes CBC more secure, as it hides patterns in the original data better than ECB mode.

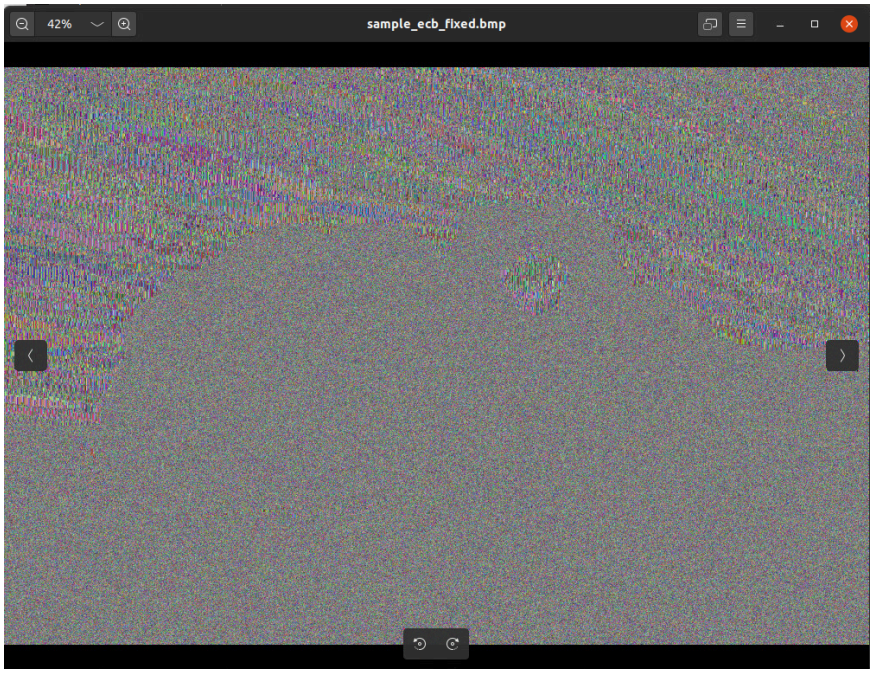
We then repeated the same procedure with another image. We got the image from https://filesampleshub.com/format/image/bmp

The following commands were run to encrypt this sample picture using the ECB and CBC mode.



Original sample image

Observations for ECB mode:

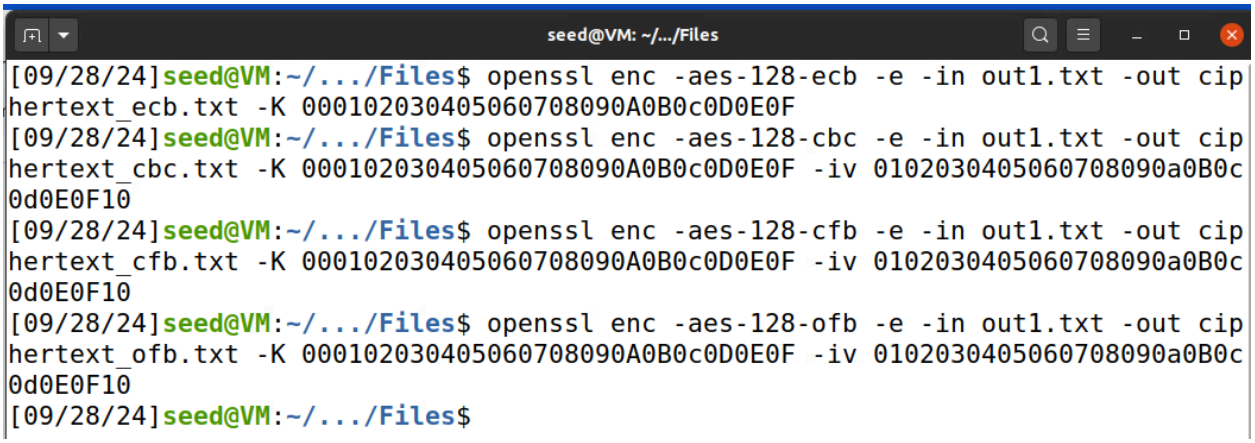


Observations for CBC mode:

Results:
We observe the same for the reasons mentioned above. CBC mode was able to encrypt the picture, such that the elements of the original image were fully hidden or hard to detect.

Task 4: Padding

1. In this task, we ran the file with different modes such as ECB, CBC, CFB and OFB.



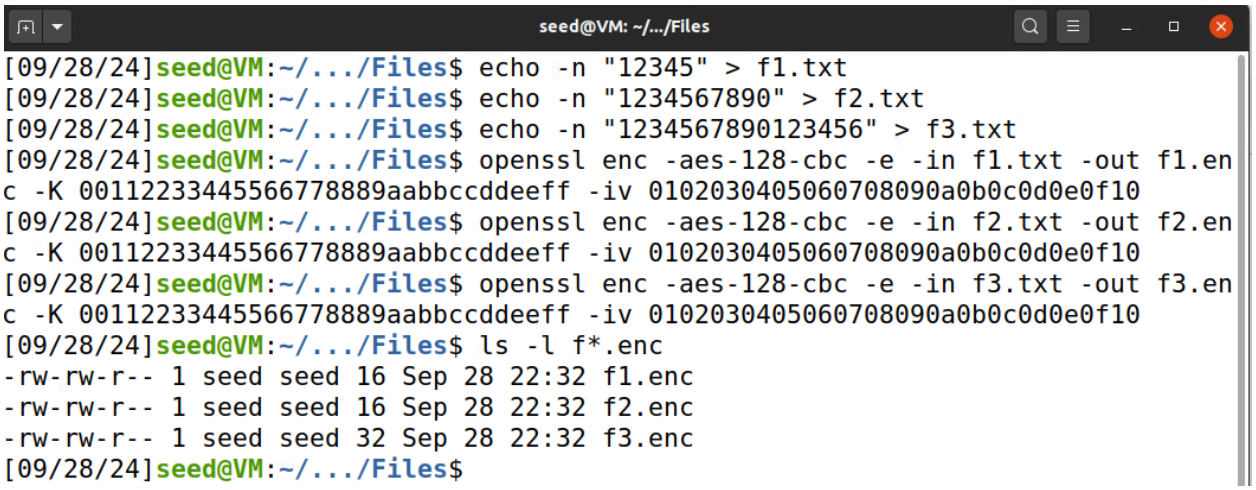
We know that block ciphers operate on fixed-size blocks of data. When the data size is not a multiple of the block size, padding is required to fill the last block.

ECB (Electronic Code Book):
Padding Required: Yes
Reason: ECB operates on fixed-size blocks, so padding is needed when data is not a perfect multiple of the block size extra bits are needed to be added to make it fit.

CBC (Cipher Block Chaining):
Padding Required: Yes
Reason: Like ECB, CBC works on blocks and requires padding if the plaintext size is not a multiple of the block size.

CFB (Cipher Feedback)/ OFB (Output Feedback):
Padding Required: No
Reason: These modes function like stream ciphers, processing data bit by bit or in smaller increments, which allows them to work without needing the data to fit into complete blocks. Since they both don't need fixed-sized blocks, any size of data can be used for an adaptable-sized block.

2. First we create 3 files which contain 5 bytes, 10 bytes, and 16 bytes, respectively. We then use "openssl enc -aes-128-cbc -e" to encrypt these three files using 128-bit AES with CBC mode:

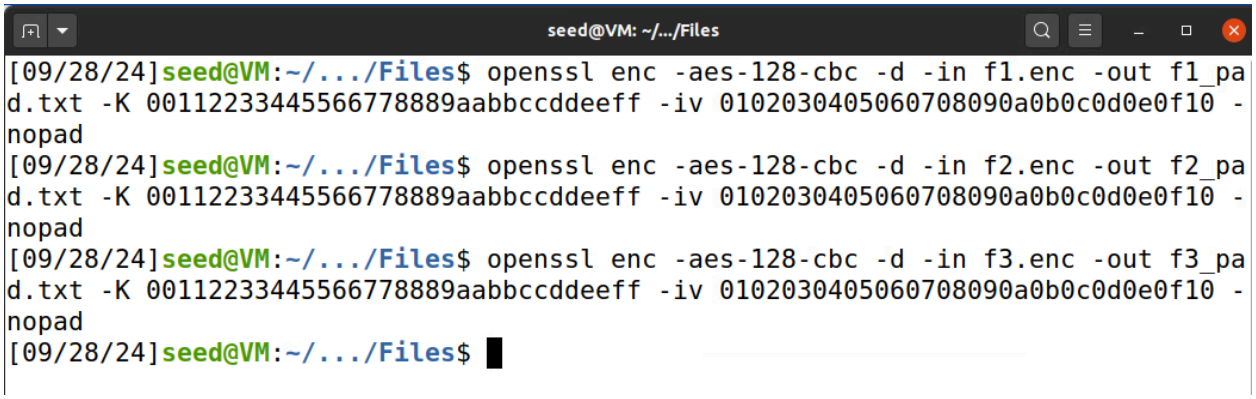


We find the size of the encrypted files to be as follows:

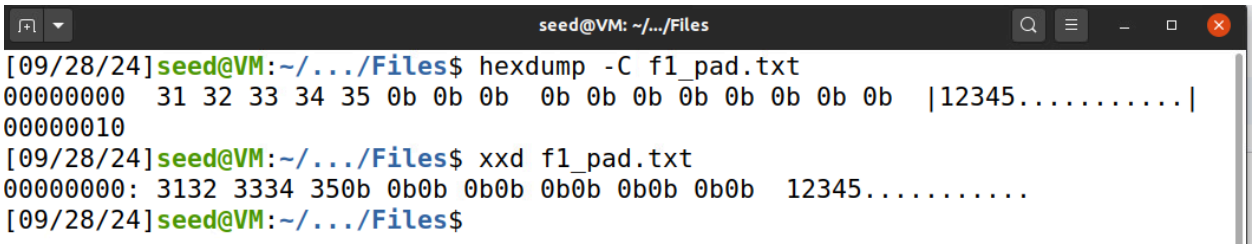
- f1.enc: 16 bytes
- f2.enc: 16 bytes
- f3.enc: 32 bytes

The reason for this is that AES has a block size of 16 bytes. Therefore the sizes are always padded to the nearest multiples of 16.

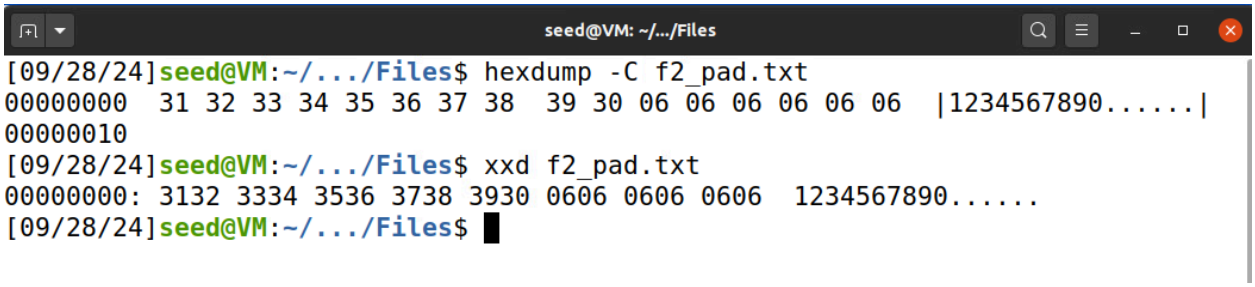
Decryption:



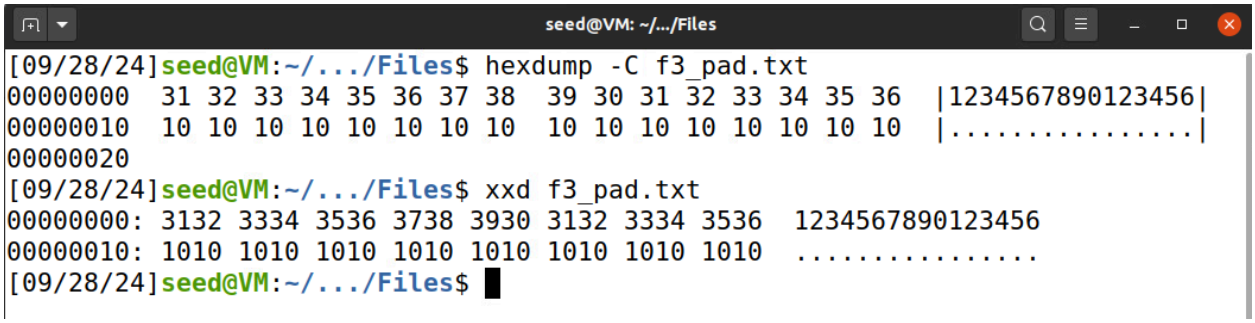
Viewing the padded content of file1



Viewing the padded content of file2



Viewing the padded content of file3



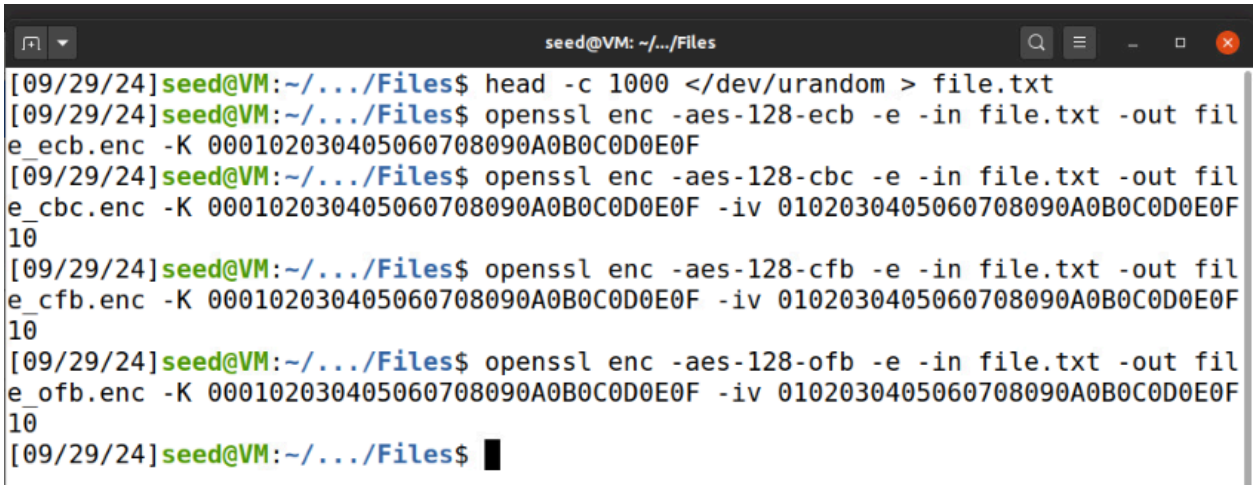
Results::

f1.txt (5 bytes):
Padding added: 11 bytes (0x0b) repeated.
f2.txt (10 bytes):
Padding added: 6 bytes (0x06) repeated.
f3.txt (16 bytes):
Full padding block added: 16 bytes (0x10) repeated.

Task 5: Error Propagation – Corrupted Cipher Text

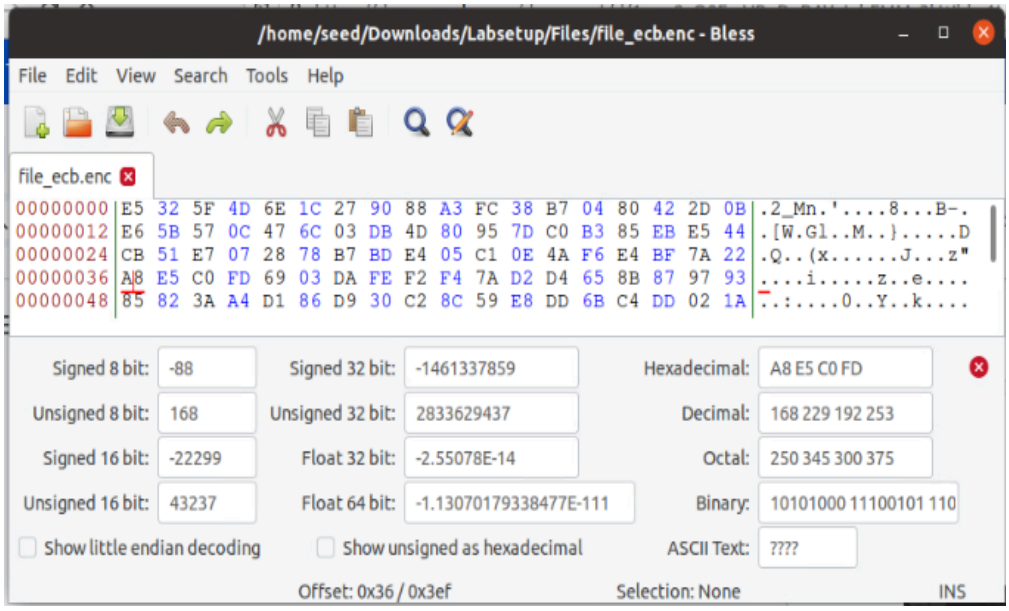
In this task we first created a text file of 1000 bytes by running this command: **head -c 1000 </dev/urandom > file.txt.**

We then encrypted the file using the AES-128 cipher.

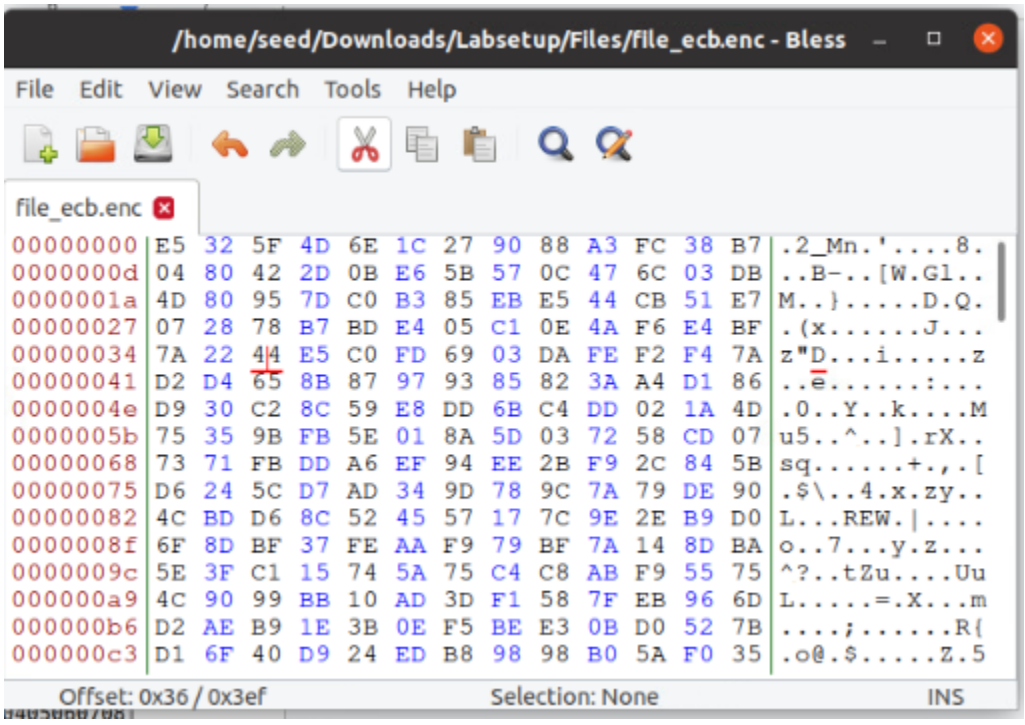


Prediction before conducting task:
When decrypting the corrupted file, the outcomes will differ based on the encryption mode employed. In modes such as ECB and OFB, we should not anticipate significant errors, as these modes do not depend on the ciphertext from prior steps. Conversely, in modes like CBC and CFB, which rely on the previous ciphertext, we can expect a higher number of errors and more pronounced changes.

ECB Mode:
Before the corruption of the 55th byte:

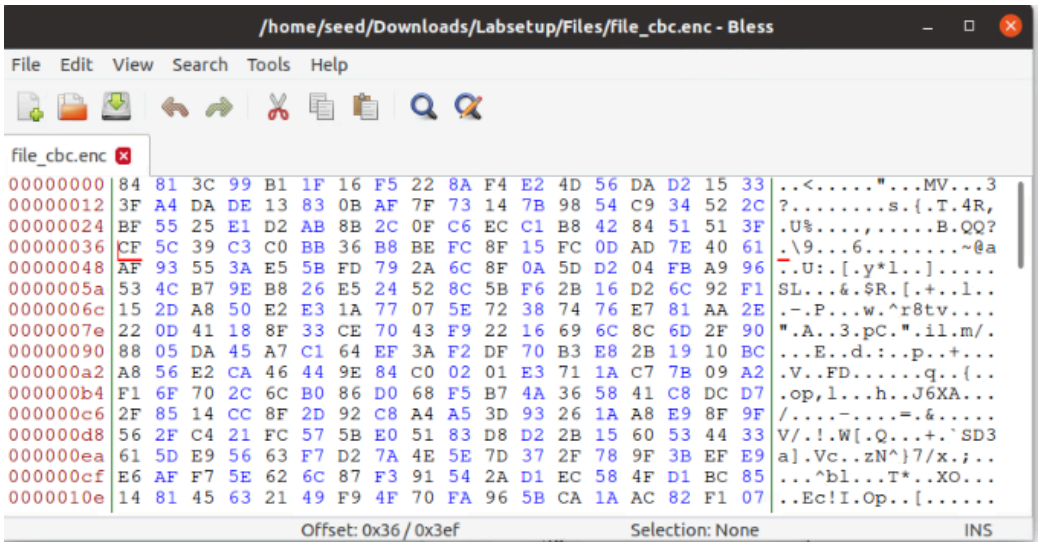


After the corruption of the 55th byte:

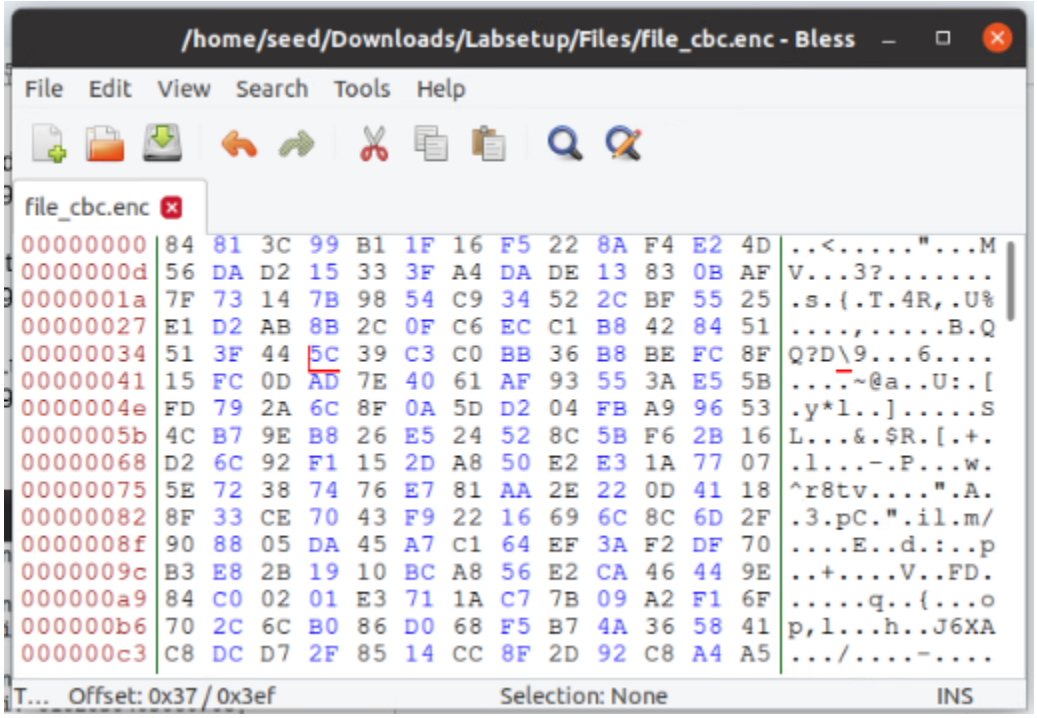


CBC Mode:

Before the corruption of the 55th byte:

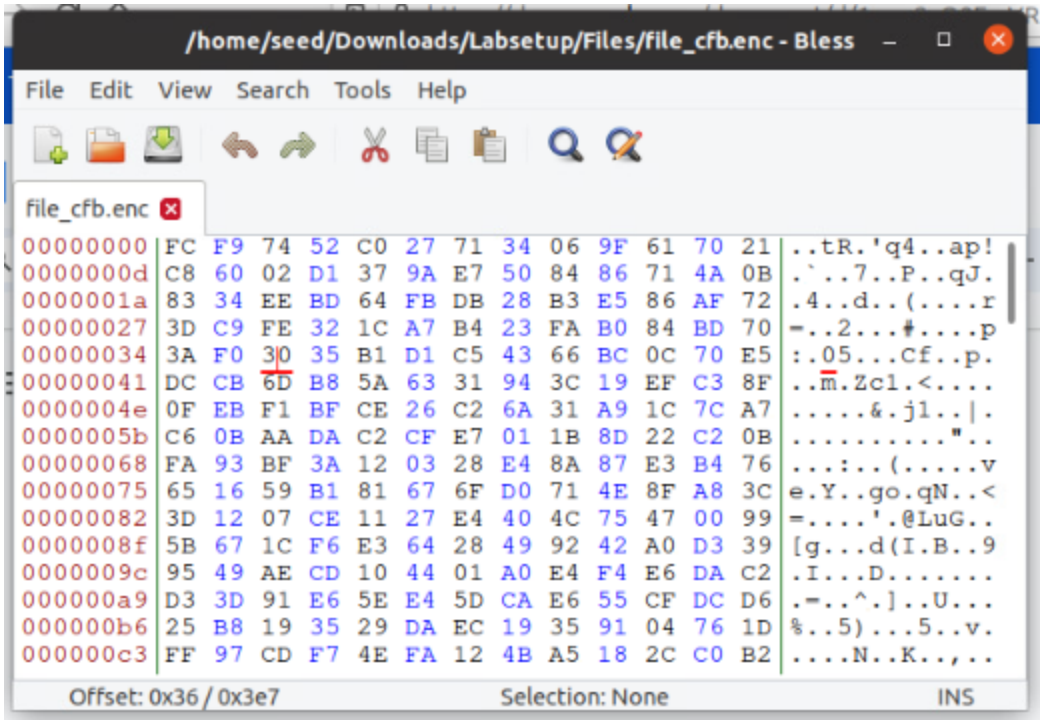


After the corruption of the 55th byte:

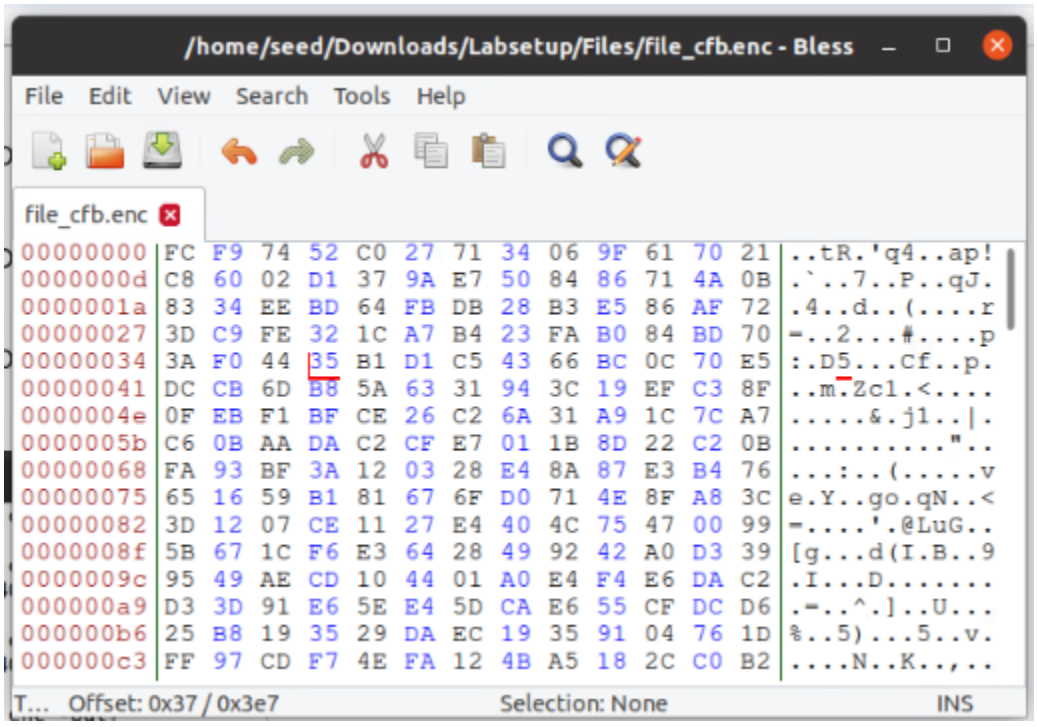


CFB Mode:

Before the corruption of the 55th byte:

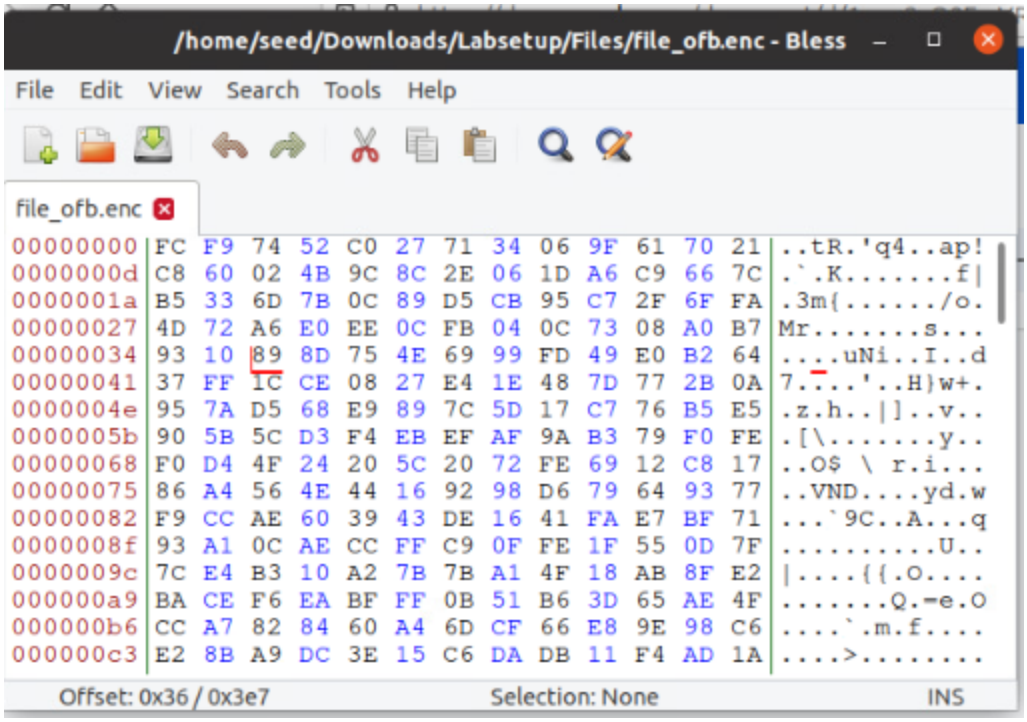


After the corruption of the 55th byte:

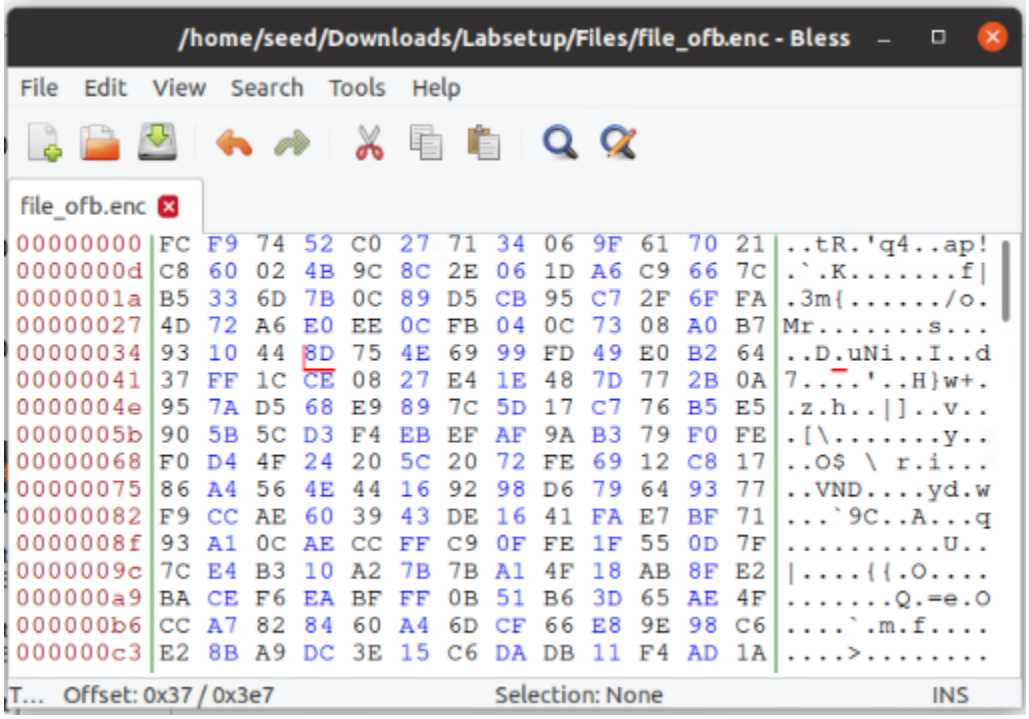


OFB Mode:

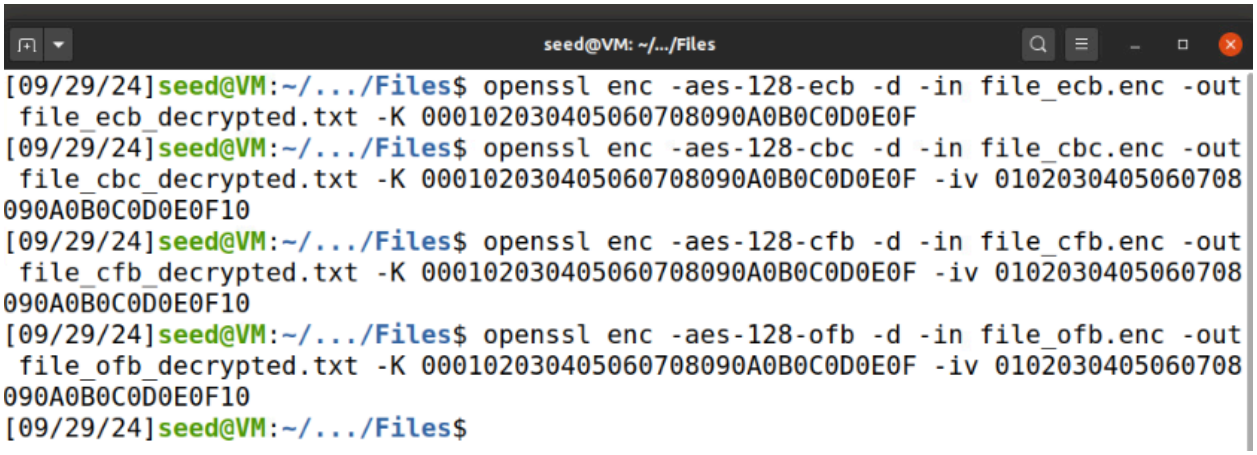
Before the corruption of the 55th byte:



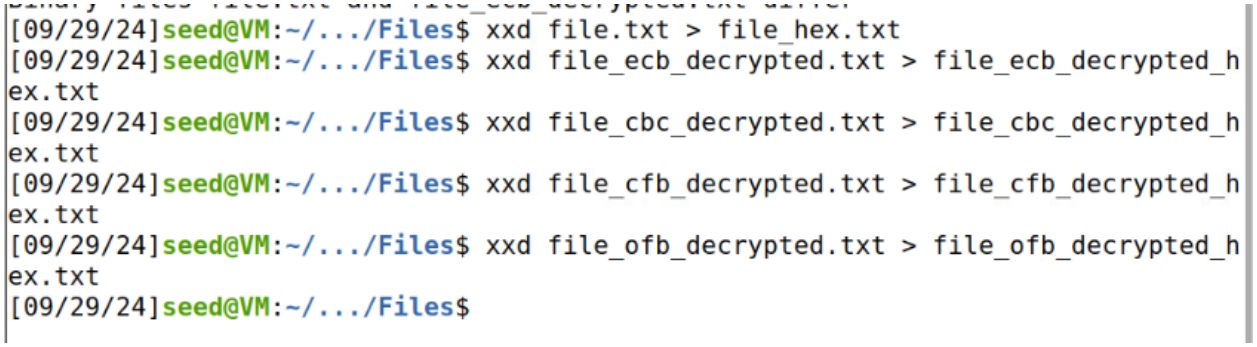
After the corruption of the 55th byte:



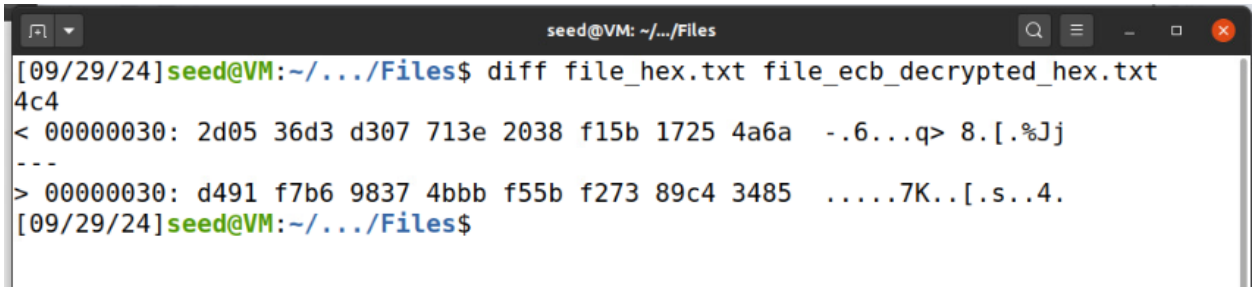
We then decrypt the files accordingly



Using the xxd command to view the hexdump of each file:

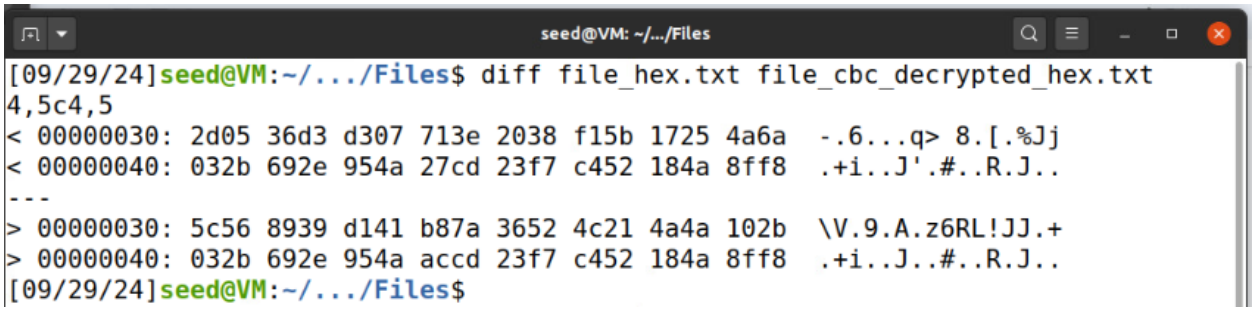


ECB mode observations:



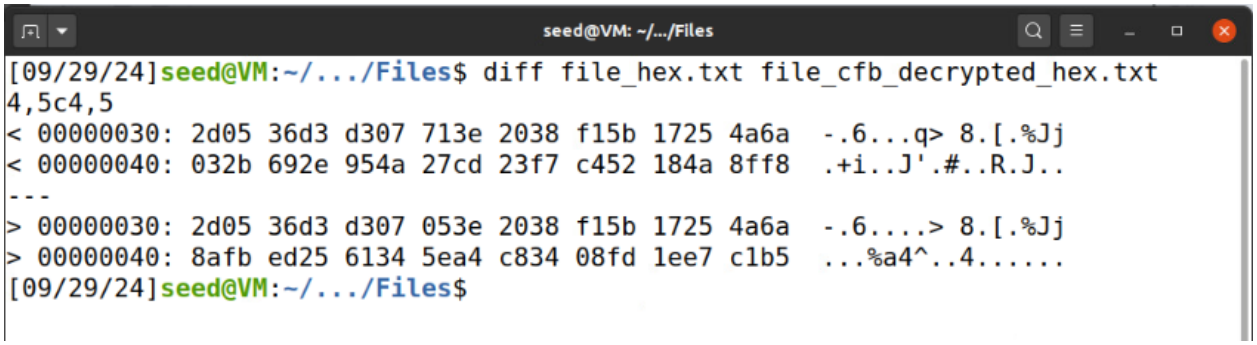
Only the block containing the corrupted byte is affected. This is because the ECB mode is not a chain, and no block depends on the previous one. Therefore, 1 block (16 bytes) is corrupted (the block containing the 55th byte).

CBC mode observations:



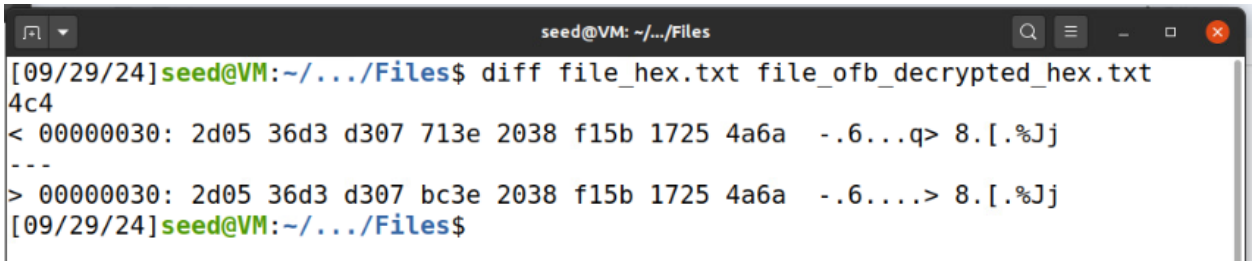
The block containing the corrupted byte and the following block are affected. Since the current block is always dependent on the previous block, the affected area is much larger. Here, 2 blocks (32 bytes) are corrupted (the block containing the 55th byte and the following block).

CFB mode observations:



The block containing the corrupted byte and the following block are affected.
CFB operates as a stream cipher. The corrupted bit will cause all following bits to be XORed with incorrect values, propagating the error throughout.

OFB mode observations:

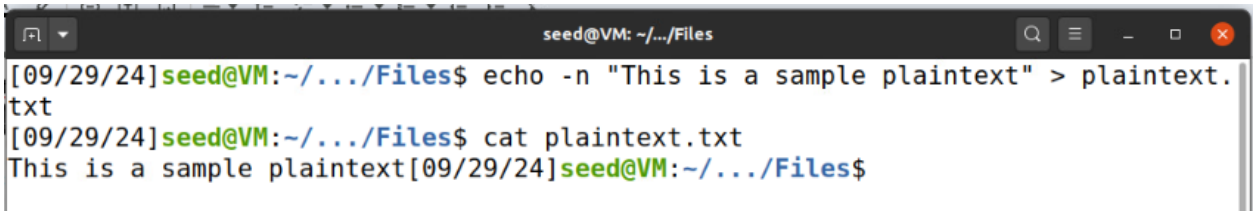


Only the single corrupted byte is affected. OFB generates a keystream independent of the ciphertext. A corrupted bit in the ciphertext only affects the corresponding plaintext bit only.

Task 6: Initial Vector (IV) and Common Mistakes

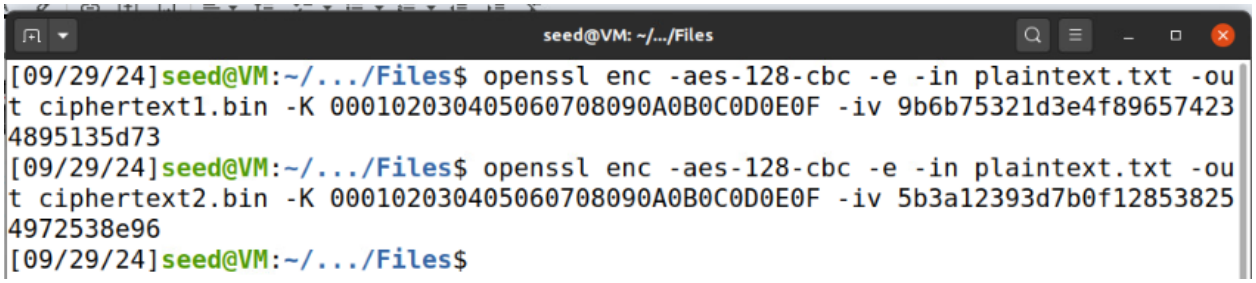
Task 6.1. IV Experiment

In this task, we first created a sample plaintext.

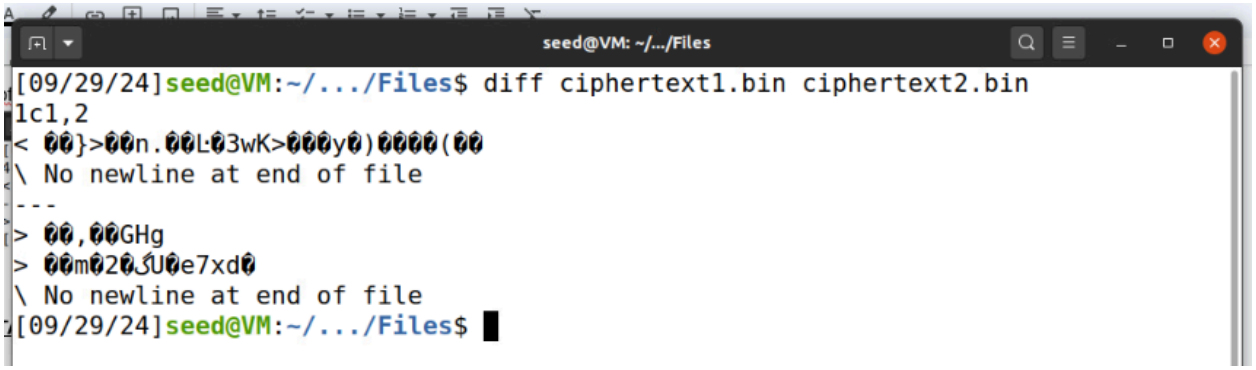


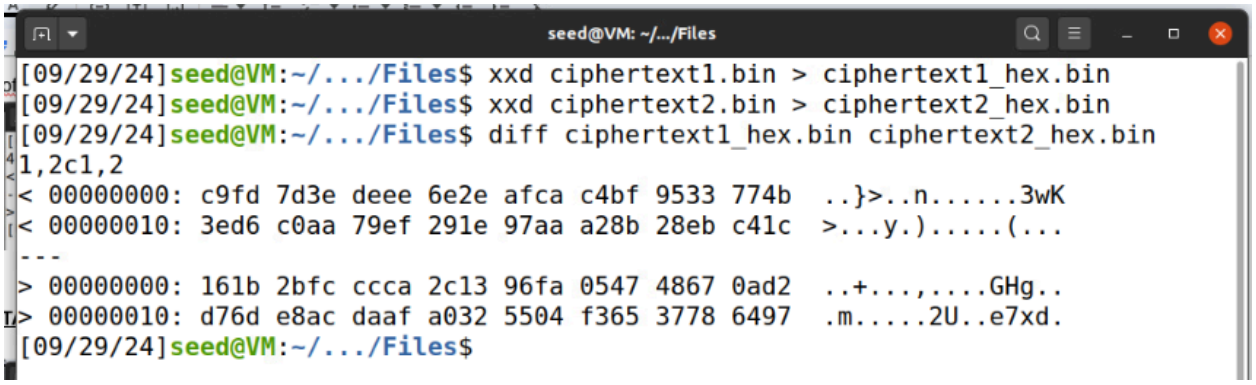
We then encrypted the file with different Initialization vectors and the same key:

- IV1 = 9b6b75321d3e4f896574234895135d73
- IV2 = 5b3a12393d7b0f128538254972538e96



We then check if there is any difference between the two encrypted files:

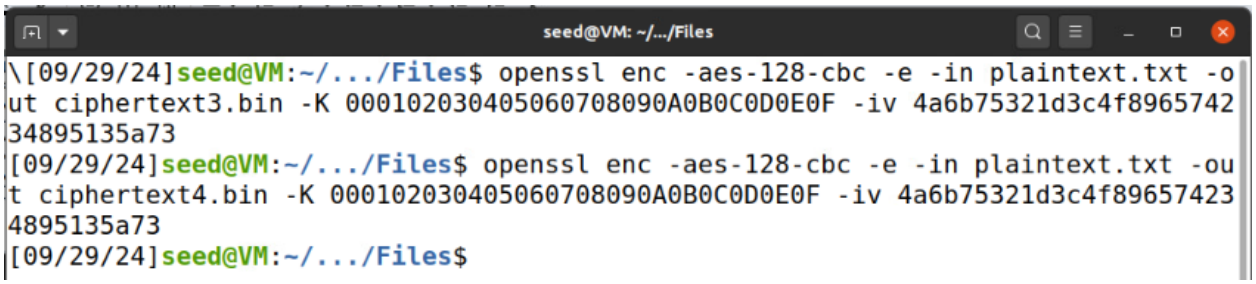




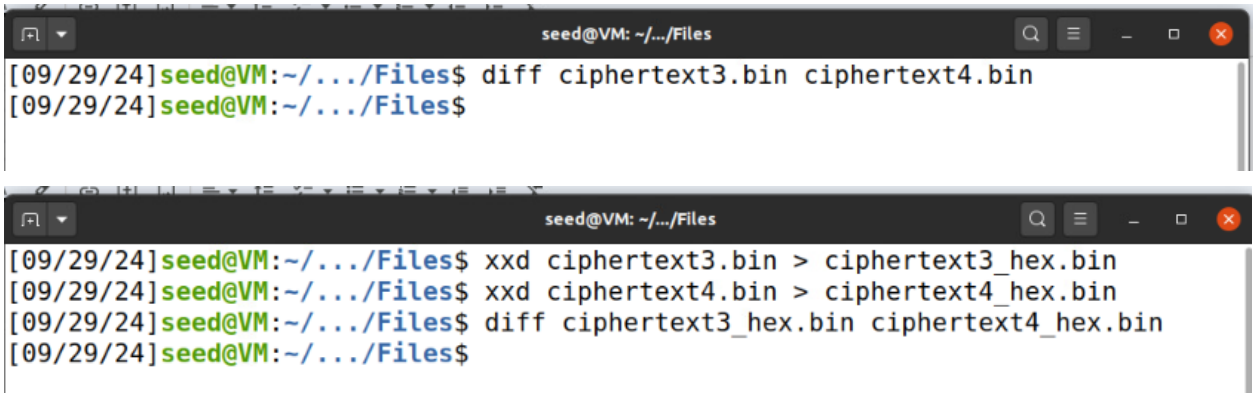
Observation:
Different IVs produce different ciphertexts for the same plaintext.

Now we perform the same procedure using the same IVs.
IV = 4a6b75321d3c4f896574234895135a73

Encrypting the file using the same IV:



We then check if there is any difference between the two encrypted files:



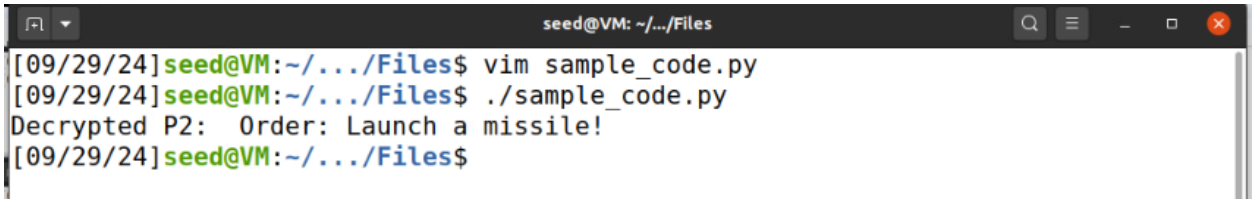
Observation:
When using the same IV, we notice that it produces identical ciphertexts. This demonstrates why IVs need to be unique - reusing an IV can reveal patterns in the encrypted data. Therefore having different IVs is important to ensure that the encrypted files differ. IVs need to be unique to prevent identical ciphertexts for confidentiality and security purposes.

Task 6.2. Common Mistake: Use the Same IV

In this task, we have to find the value of “P2”, based on the values of C2, P1, and C1.

We changed the sample code provided accordingly so that it gave out the output for P2.

Running the script we get:



As observed, the plain text message was `Order: Launch a missile!`.

Question - 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.

Answer:
OFB mode requires a new Initialization Vector IV for each message. In OFB, the IV is used to create a special sequence of random-looking data called a keystream. This keystream is then combined with the message to encrypt it. If you use the same IV more than once, you'll get the same keystream. This is dangerous because if an attacker gets two messages encrypted with the same keystream, they might be able to figure out parts of both messages. By using a different IV

each time, you ensure that each message has its unique keystream, making it much harder for attackers to break the encryption.

CFB mode works a bit differently. Here, even if an attacker knows the IV, they can only potentially figure out the first part of the message. This is because CFB uses each piece of encrypted data to help encrypt the next piece. So, after the first part, the IV isn't used anymore. Instead, the previous encrypted piece acts like a new IV for the next piece. This means that even if an attacker knows the IV, they can't easily decrypt the whole message. They might get a small part at the beginning, but the rest stays secret unless they know the encryption key. This makes CFB more resistant to attacks, especially if the IV is somehow compromised or reused.

Task 6.3. Common Mistake: Use a Predictable IV

In this task, we have to find out Bob’s message. As the question suggests we know the value of his message can be “Yes” or “No”.

Since we are using CBC mode, it is known that using predictable IV can make the algorithm vulnerable, making it possible to discover Bob’s message.

We modified the script provided to grab the value of plaintext.

```
seed@VM: ~/.../Files
#!/usr/bin/python3

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

IV_used = "baf4246d46dcef31428b3cee1ccfa86d"
Next_IV = "48b9237d46dcef31428b3cee1ccfa86d"
MSG = "Yes"

pad_len = 16 - len(MSG) % 16
D1 = bytearray(MSG, encoding='utf-8')
D1.extend([pad_len] * pad_len)

# Convert hex string to bytearray
D2 = bytearray.fromhex(IV_used)
D3 = bytearray.fromhex(Next_IV)

r1 = xor(D1, D2)

plaintext = xor(r1, D3)

print("Plaintext in hex: ", plaintext.hex())
```

We then run this script to get the value of plaintext in hexadecimal

```
[09/29/24]seed@VM:~/.../Files$ vim iv_attack.py
[09/29/24]seed@VM:~/.../Files$ ./iv_attack.py
Plaintext in hex:  ab28741d0d0d0d0d0d0d0d0d0d0d0d0d
```

After receiving this value, we feed it to the oracle and get the ciphertext. We then compare if both of them are equal.

```
seed@VM: ~/.../Labsetup
Next IV        : 59869881021f379f7db5113d19f3af47
Your plaintext : ^C
[09/29/24]seed@VM:~/.../Labsetup$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 4520218115fdc4694e2c2d54799636c7
The IV used    : baf4246d46dcef31428b3cee1ccfa86d

Next IV        : 48b9237d46dcef31428b3cee1ccfa86d
Your plaintext : ab28741d0d0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: 4520218115fdc4694e2c2d54799636c7e701fee645544eeda9cb9acec7e37cd
b

Next IV        : e0cf38ab46dcef31428b3cee1ccfa86d
Your plaintext :
```

The answer 'Yes' is right. We can see that only the first 16 bytes (128 bits) of the ciphertext are the same as Bob's ciphertext. This happens because the CBC mode needs the input to be exactly 16 bytes long. The word 'Yes' is shorter than 16 bytes, so extra characters (padding) were added to make it 16 bytes. The oracle only accepts input in hex format, so this padded version of 'Yes' was used. This created two blocks of ciphertext: the first block has the encrypted 'Yes' with some padding, and the second block is just the encrypted padding. That's why only the first block matches Bob's ciphertext.

Task 7: Programming using the Crypto Library

gcc -o decrypt decrypt.c -lcrypto

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

int main() {
    // Array to store the plaintext "This is a top secret."
    unsigned char plaintext[21];

    // Given ciphertext in hexadecimal format (adding 0x)
764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2
    unsigned char ciphertext[] = {
        0x76, 0x4a, 0xa2, 0x6b, 0x55, 0xa4, 0xda, 0x65,
        0x4d, 0xf6, 0xb1, 0x9e, 0x4b, 0xce, 0x00, 0xf4,
        0xed, 0x05, 0xe0, 0x93, 0x46, 0xfb, 0x0e, 0x76,
        0x25, 0x83, 0xcb, 0x7d, 0xa2, 0xac, 0x93, 0xa2
    };

    // Initialization vector in hexadecimal format
aabbccddeeff00998877665544332211
    unsigned char iv[] = {
        0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x00, 0x99,
        0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11
    };

    // Array to store the encryption key
    unsigned char key[16];
    // Array to store each word from the word list
    char word[16];
    // Array to store the encrypted output
    unsigned char encryptedop[32];

    // Open the plaintext file for reading
    FILE *plaintext_file = fopen("plaintext.txt", "rb");
    if (!plaintext_file) {
        fprintf(stderr, "Error opening plaintext file\n");
        return 1;
    }

    // Read the plaintext from the file
    fread(plaintext, 1, 21, plaintext_file);
```

```
    fclose(plaintext_file);

    // Open the word list file for reading
    FILE *words = fopen("words.txt", "r");
    if (!words) {
        fprintf(stderr, "Error opening word list file\n");
        return 1;
    }

    // Read each word from the word list
    while (fgets(word, sizeof(word), words)) {
        // Remove newline character from the word
        word[strcspn(word, "\n")] = 0;


        // Copy the word into the key array and pad with '#'
        memcpy(key, word, strlen(word));
        memset(key + strlen(word), '#', 16 - strlen(word));


        int enc_l, final_l;
        // Create encryption context
        EVP_CIPHER_CTX *ctx;
        ctx = EVP_CIPHER_CTX_new();

        EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv);  //
Initialize encryption

        // Encrypt the plaintext
        EVP_EncryptUpdate(ctx, encryptedop, &enc_l, plaintext, strlen((char
*)plaintext));
        EVP_EncryptFinal_ex(ctx, encryptedop + enc_l, &final_l);  // Finalize
the encryption
        enc_l += final_l;

        EVP_CIPHER_CTX_free(ctx);  // Free the encryption context

        // Compare the encrypted output with the given ciphertext
        if (memcmp(encryptedop, ciphertext, 32) == 0) {
            printf("Found key: %s\n", word);  // Print the found key
            fclose(words);  // Close the word list file
            return 0;  // Exit the program
        }
    }
```

```
    printf("Key not found\n");
    fclose(words);  // Close the word list file
    return 1;
}
```

```
[10/01/24]seed@VM:~/.../Files$ gcc -o decrypt decrypt.c -lcrypto
[10/01/24]seed@VM:~/.../Files$ ./decrypt
Found key: Syracuse
```

We stored the plaintext in a file using ***echo -n "This is a top secret." > plaintext.txt***. Then, we retrieved it, ensured it was 21 bytes long, and stored it in a char array. The given ciphertext and IV are stored in char arrays, where each element in C represents 1 byte. Both are in hexadecimal format.

Next, we allocated memory for the key, encrypted output, and word. The word has a size of 16 bytes since that's the size of one AES-128 block.

We open words.txt and read each word, padding it with # if it's less than 16 bytes to fit the AES-128 block size. The memcpy() function copies the word into the key, and memset() fills the remaining space with padding.

The key is set using the padded word, and we create an encryption context with EVP_CIPHER_CTX_new(), which initializes the context needed for encryption. The EVP_EncryptInit_ex() function sets up the encryption using AES-128 in CBC mode, with the provided key and IV. This prepares the encryption operation.

Next, EVP_EncryptUpdate() encrypts the plaintext in blocks, writing the result to the encrypted array and storing the length of the encrypted data. EVP_EncryptFinal_ex() completes the encryption process by handling any remaining data and final padding.

The memcmp() function compares the encrypted output (encryptedop) with the given ciphertext. If they match, the correct key is found, and the program prints it.

If no match is found after going through all the words, the program prints "Key not found." The encryption context is cleaned up with EVP_CIPHER_CTX_free() to prevent memory leaks, and fclose() is used to properly close the file.

In this case, the key was found, meaning the correct word “Syracuse” from words.txt successfully decrypted the ciphertext, verifying it as the encryption key.