Lab 3: Symmetric encryption & hashing

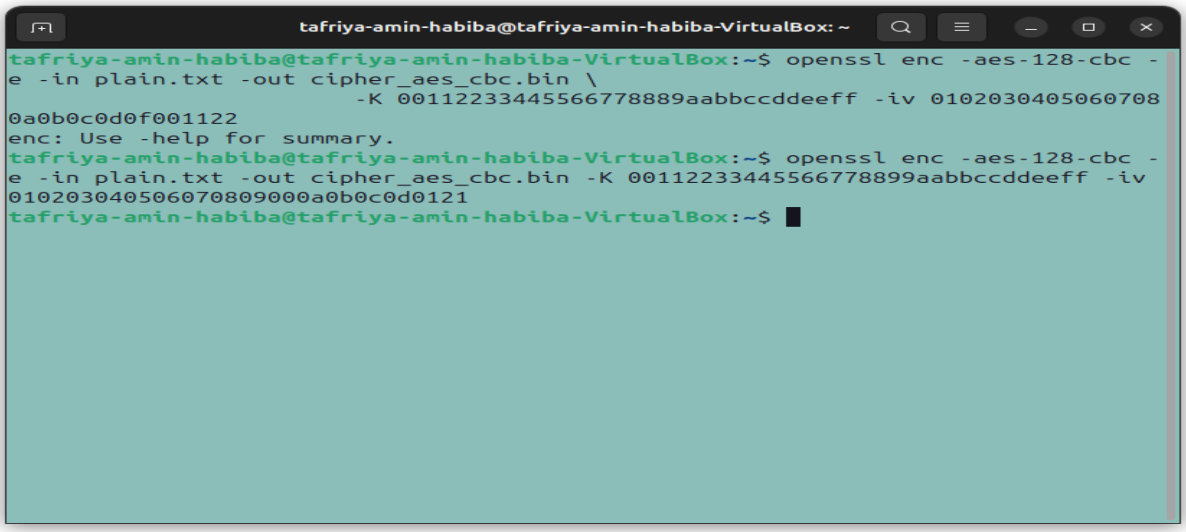
Task 1: AES Encryption Using Different Modes

1. First Created a plaintext file using the command– echo"This is a simple test msg on lab 3 on symmetric encryption" > plain.txt

2.Then Generate AES Encrypted file for each AES mode

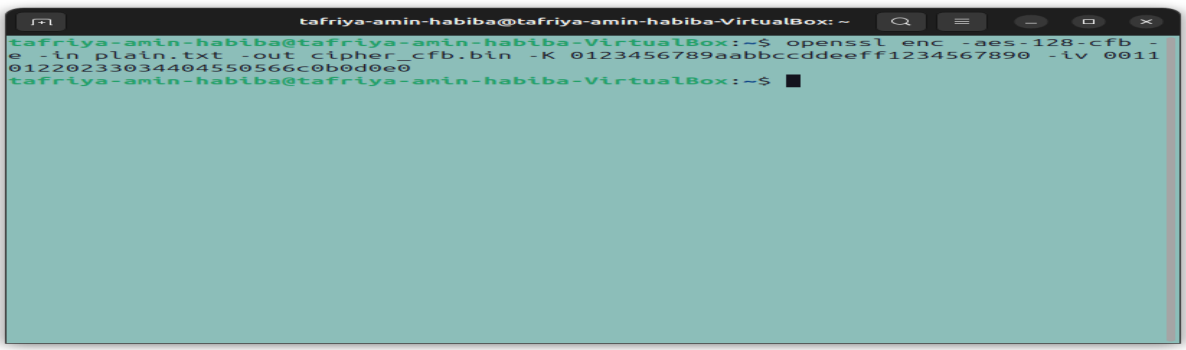
i.CBC-Command used

-openssl enc -aes-128-cbc -e -in plain.txt -out cipher_aes_cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708090a0b0c0d0121



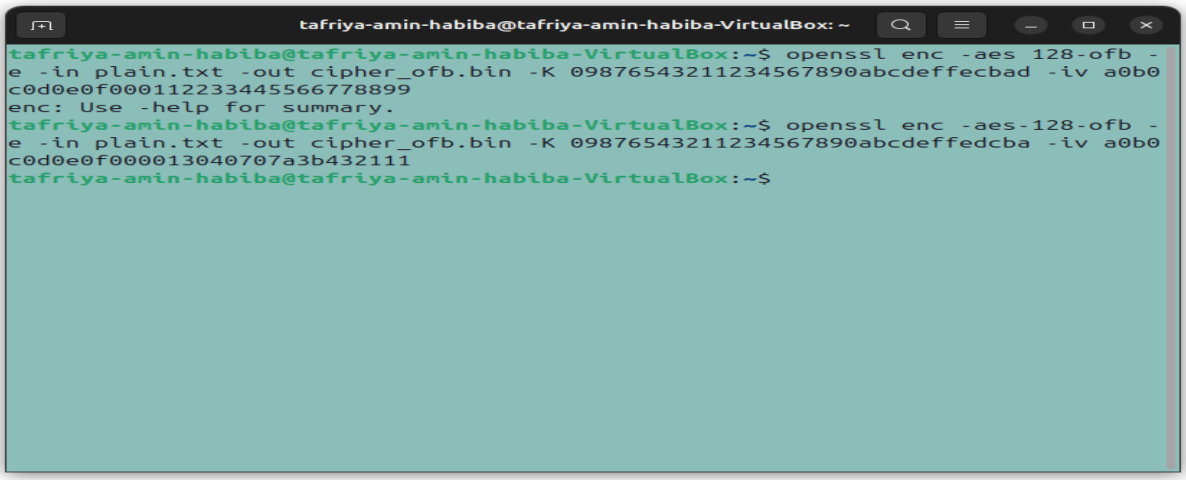
ii.CFB-Command used

-openssl enc -aes-128-cfb -e -in plain.txt -out cipher_aes_cfb.bin -K 0123456789aabbccddeeff1234567890 -iv 00110122023303440455056c0b0d0e0



iii.OFB-Command used

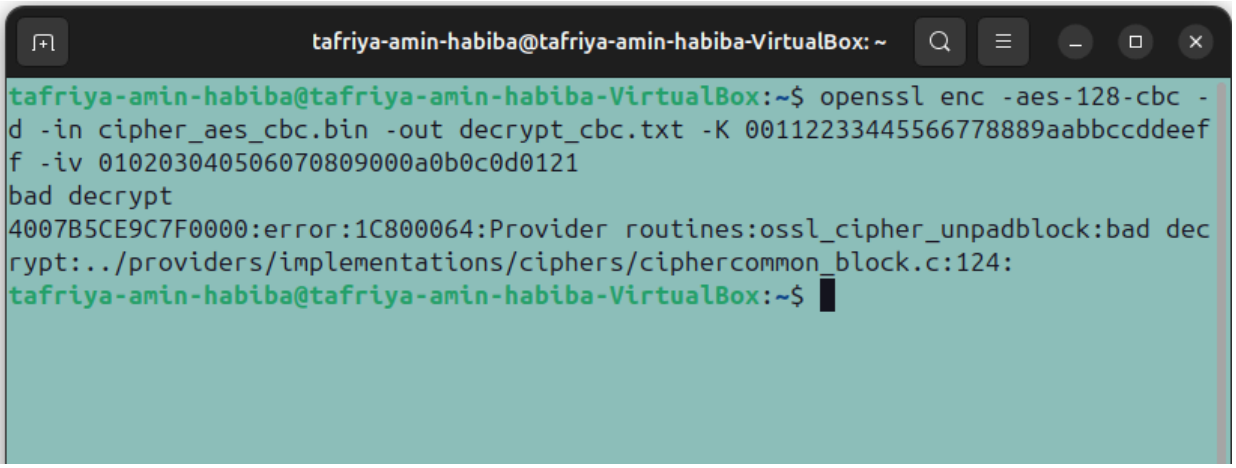
-openssl enc -aes-128-ofb -e -in plain.txt -out cipher_ofb.bin -K 0123456789aabbccddeeff1234567890 -iv 00110122023303440455056 6c0b0d0e0



3. AES Decryption to Verify

i.CBC

command-openssl enc -aes-128-cbc -d -in plain.txt -out cipher_aes_cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708090 00a0b0c0d0121

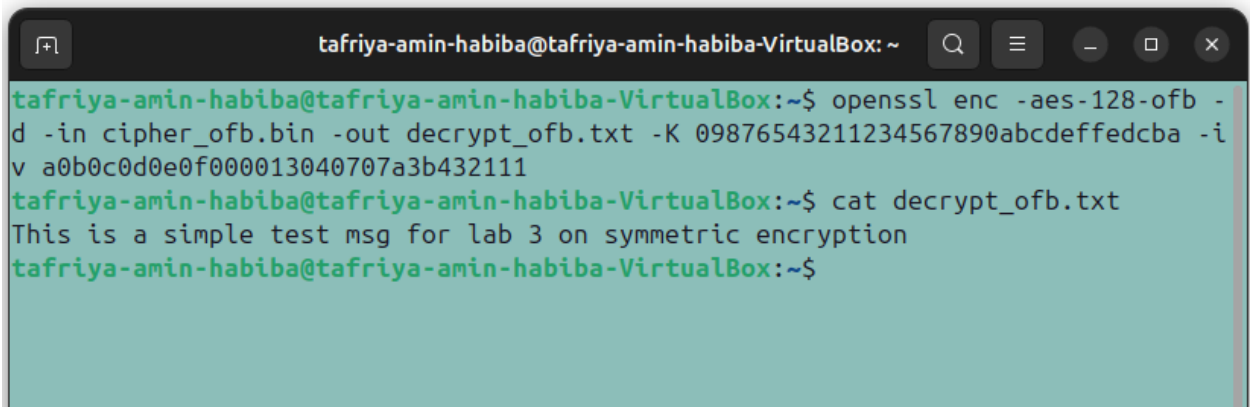


ii.CFB

Command-openssl enc -aes-128-cfb -d -in plain.txt -out cipher_aes_cfb.bin -K 0123456789aabbccddeeff1234567890 -iv 00110122023303440455056 6c0b0d0e0

iii.OFB

Command-openssl enc -aes-128-ofb -d -in plain.txt -out cipher_aes_ofb.bin -K 0123456789aabbccddeeff1234567890 -iv 00110122023303440455056 6c0b0d0e0



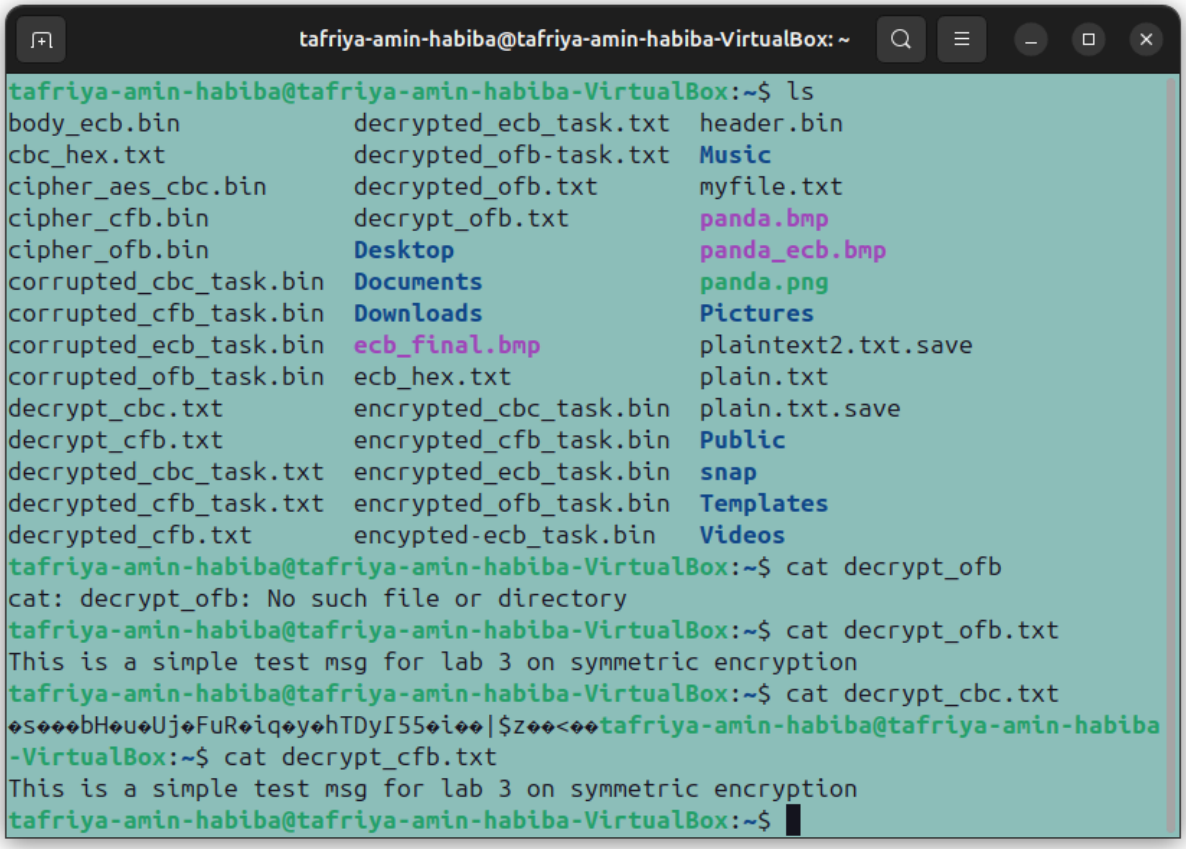
4.Verify the Results

Compare the decrypted files with the original plaintext:

cat decrypt_cbc.txt

cat decrypt_cfb.txt

cat decrypt_ofb.txt



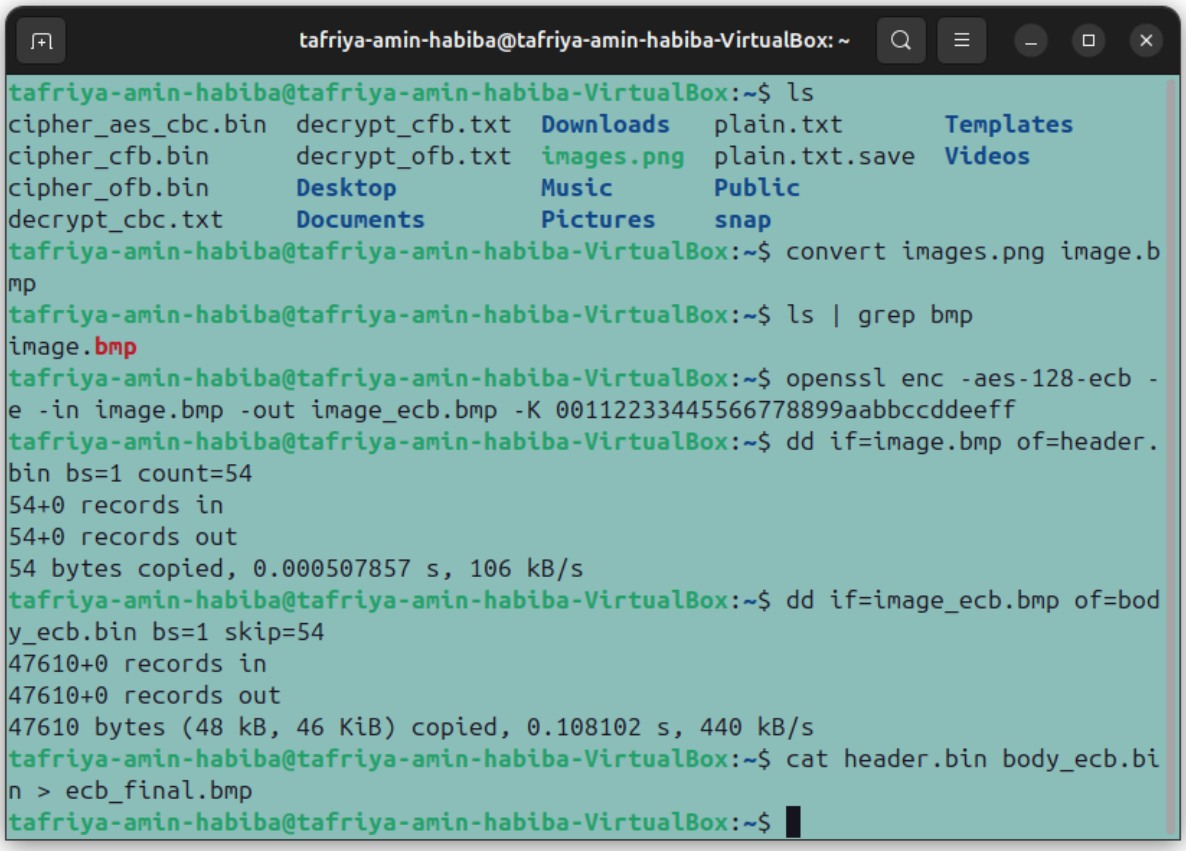
Note:Previosly we saw bad decryption in cbc thats why when we verify it we are not getting original plaintext

Task:2- Encryption mode - ECB vs CBC

Flower.jpg

1.First Convert image from jpg to bmp

Command-convert image.png to image.bmp



2. Encryption Commands:

ECB Mode Encryption: openssl enc -aes-128-ecb -e -in pic_original.bmp -out encrypted_ecb.bmp \ -K 0123456789abcdef0123456789abcdef

CBC Mode Encryption: openssl enc -aes-128-cbc -e -in pic_original.bmp -out encrypted_cbc.bmp \ -K 0123456789abcdef0123456789abcdef \ -iv 0102030405060708090a0b0c0d0e0f10

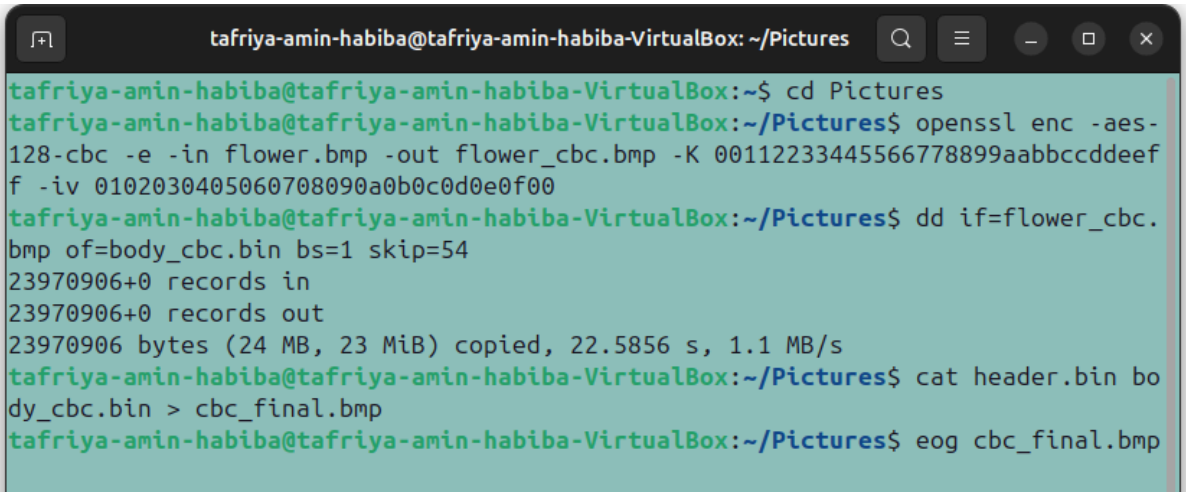
3. Header Replacement Commands:

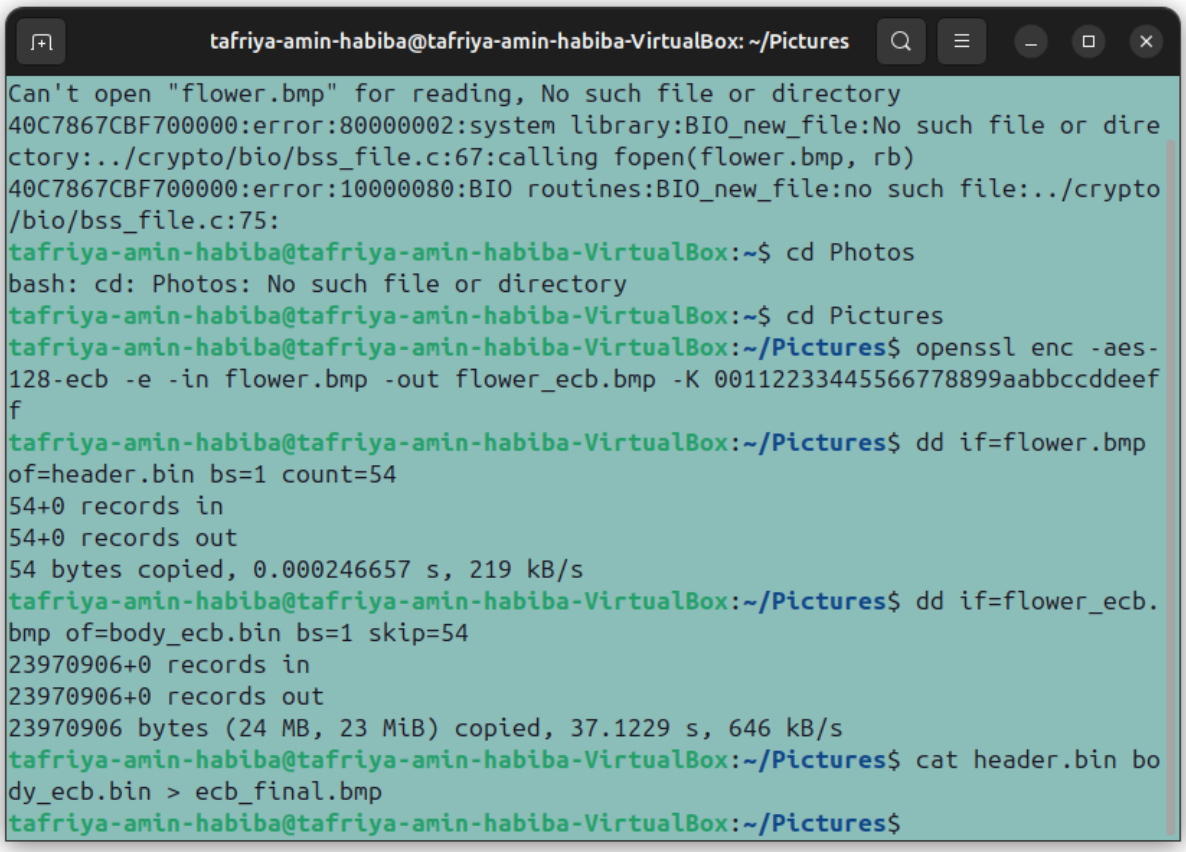
Extract original header

(first 54 bytes) dd if=pic_original.bmp of=header.bin bs=1 count=54

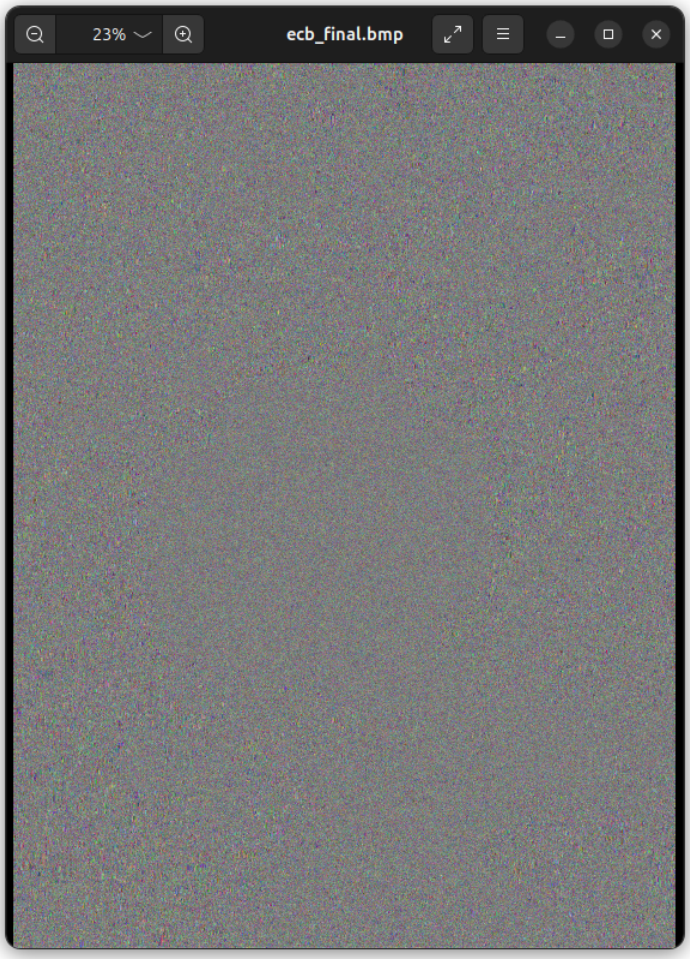
Extract ECB encrypted body (skip header)

```
 dd if=encrypted_ecb.bmp of=body_ecb.bin bs=1 skip=54
 # Extract CBC encrypted body (skip header)
dd if=encrypted_cbc.bmp of=body_cbc.bin bs=1 skip=54
# Combine original header with ECB encrypted body
cat header.bin body_ecb.bin > ecb_final.bmp
 # Combine original header with CBC encrypted body
 cat header.bin body_cbc.bin > cbc_final.bmp
```

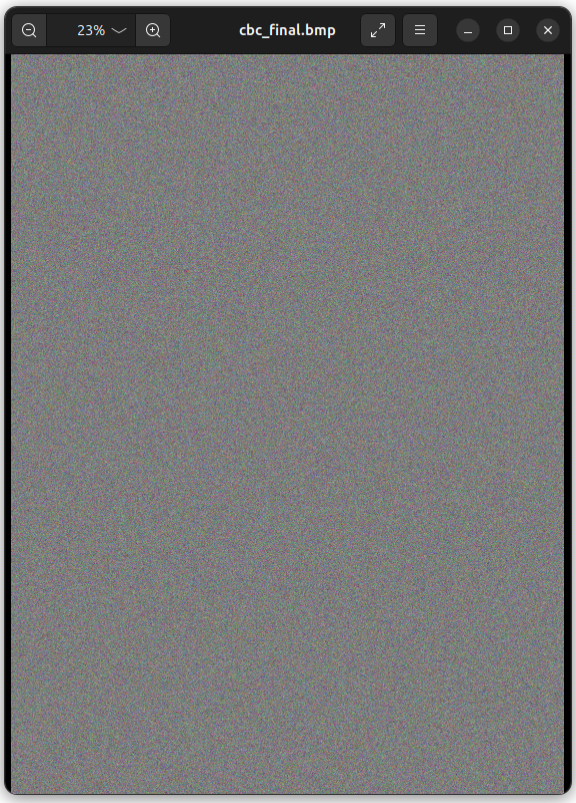

tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox: ~/Pictures
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ cd Pictures
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$ openssl enc -aes-128-cbc -e -in flower.bmp -out flower_cbc.bmp -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f00
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$ dd if=flower_cbc.bmp of=body_cbc.bin bs=1 skip=54
23970906+0 records in
23970906+0 records out
23970906 bytes (24 MB, 23 MiB) copied, 22.5856 s, 1.1 MB/s
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$ cat header.bin body_cbc.bin > cbc_final.bmp
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$ eog cbc_final.bmp


tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox: ~/Pictures
Can't open "flower.bmp" for reading, No such file or directory
40C7867CBF700000:error:80000002:system library:BIO_new_file:No such file or directory:../crypto/bio/bss_file.c:67:calling fopen(flower.bmp, rb)
40C7867CBF700000:error:10000080:BIO routines:BIO_new_file:no such file:../crypto/bio/bss_file.c:75:
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ cd Photos
bash: cd: Photos: No such file or directory
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ cd Pictures
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$ openssl enc -aes-128-ecb -e -in flower.bmp -out flower_ecb.bmp -K 00112233445566778899aabbccddeeff
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$ dd if=flower.bmp of=header.bin bs=1 count=54
54+0 records in
54+0 records out
54 bytes copied, 0.000246657 s, 219 kB/s
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$ dd if=flower_ecb.bmp of=body_ecb.bin bs=1 skip=54
23970906+0 records in
23970906+0 records out
23970906 bytes (24 MB, 23 MiB) copied, 37.1229 s, 646 kB/s
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$ cat header.bin body_ecb.bin > ecb_final.bmp
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~/Pictures$

ECB



CBC



Observation

During the experiment, I used a color image as input for AES encryption using different modes (ECB and CBC).
After performing encryption, I observed that:

- The AES-ECB encrypted image still partially preserved the structure and color patterns of the original image.
  This happened because ECB encrypts identical plaintext blocks into identical ciphertext blocks, which makes patterns visible.

- In contrast, the AES-CBC encrypted image appeared much more random and noise-like, effectively hiding all recognizable patterns.

However, since the image was colorful, the visual difference between ECB and CBC modes was less distinguishable than expected.
The colors and visual textures somewhat blurred the contrast between the two, though CBC still provides stronger security in theory

TASK-3:Encryption mode – corrupted cipher text

1. Test File Creation: -

Filename: myfile.txt - File size: 38 bytes

- Content: This is a test file for my lab task 3.

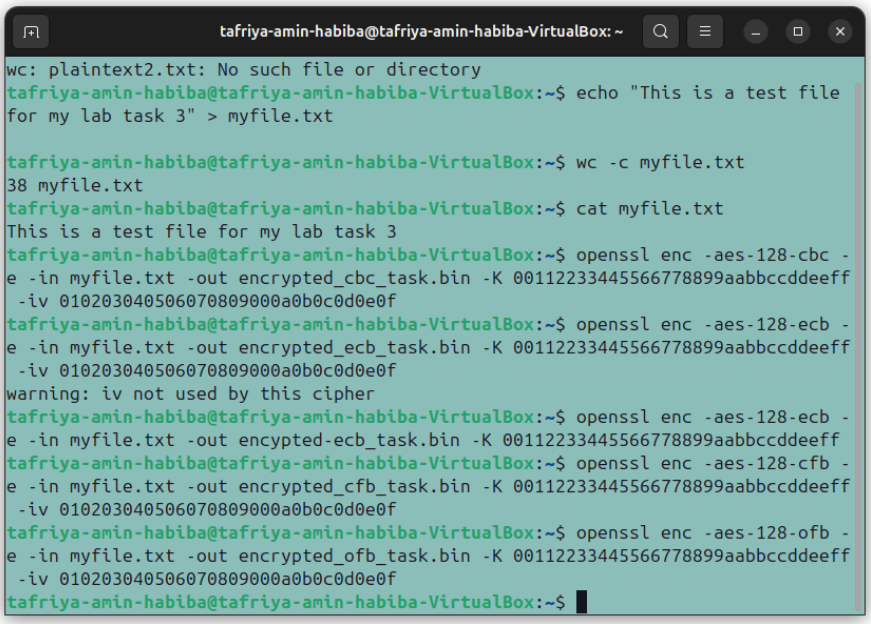
2. Encryption Commands Used:

ECB Mode: openssl enc -aes-128-ecb -e -in plaintext_task3.txt -out encrypted_ecb_task3.bin \ -K 0123456789abcdef0123456789abcdef

CBC Mode: openssl enc -aes-128-cbc -e -in plaintext_task3.txt -out encrypted_cbc_task3.bin \ -K 0123456789abcdef0123456789abcdef \ -iv 01020304050607080 90a0b0c0d0e0f10

CFB Mode: openssl enc -aes-128-cfb -e -in plaintext_task3.txt -out encrypted_cfb_task3.bin \ -K 0123456789abcdef0123456789abcdef \ -iv 01020304050607080 90a0b0c0d0e0f10

OFB Mode: openssl enc -aes-128-ofb -e -in plaintext_task3.txt -out encrypted_ofb_task3.bin \ -K 0123456789abcdef0123456789abcdef \ -iv 01020304050607080 90a0b0c0d0e0f10



3. Corruption Method: - Corrupted byte position: 30th byte (index 29

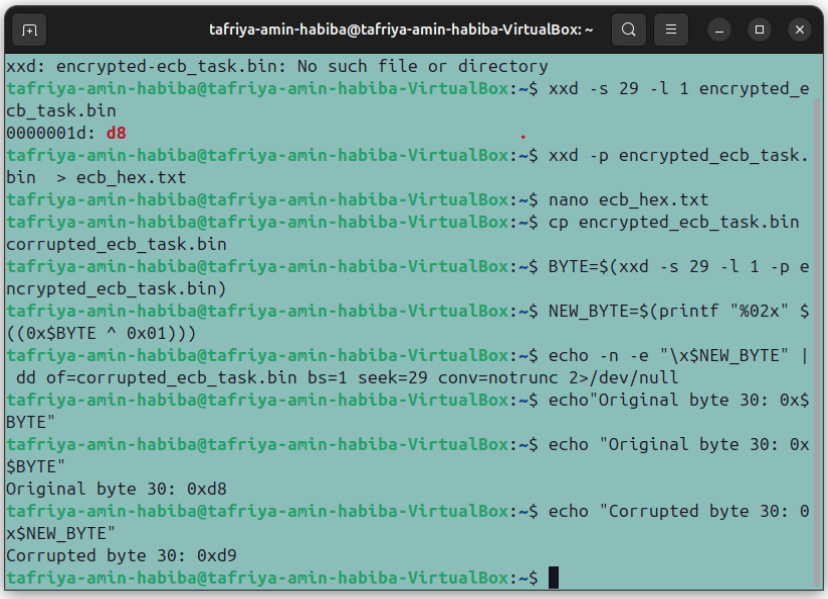
) - Method: Flipped one bit using XOR with 0x01

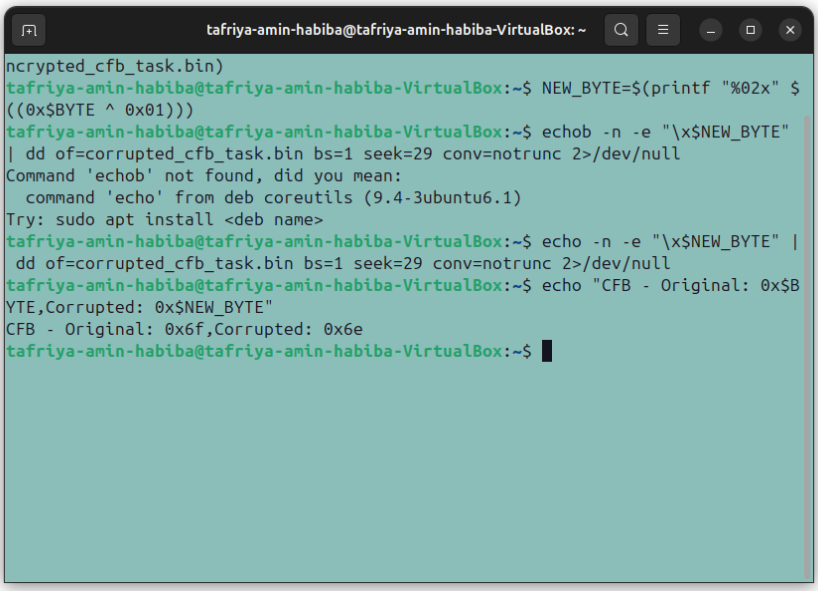
Corruption commands (example for ECB): cp encrypted_ecb_task.bin corrupted_ecb_task3.bin

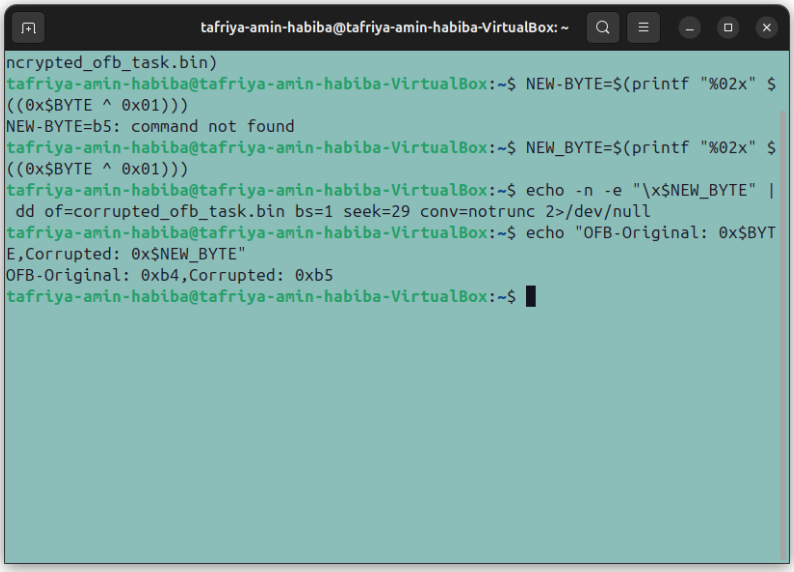
BYTE=$(xxd -s 29 -l 1 -p encrypted_ecb_task3.bin)

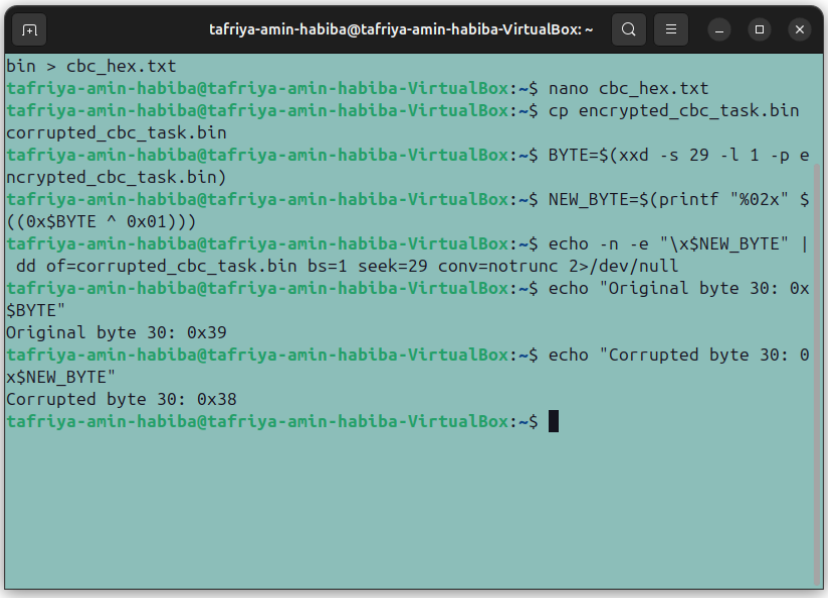
NEW_BYTE=$(printf "%02x" $((0x$BYTE ^ 0x01))) echo -n -e "\x$NEW_BYTE" | dd of=corrupted_ecb_task3.bin bs=1 seek=29 conv=notrunc

[Repeat similar commands for CBC, CFB, OFB]


tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox: ~
xxd: encrypted-ecb_task.bin: No such file or directory
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ xxd -s 29 -l 1 encrypted_ecb_task.bin
0000001d: d8
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ xxd -p encrypted_ecb_task.bin > ecb_hex.txt
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ nano ecb_hex.txt
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ cp encrypted_ecb_task.bin corrupted_ecb_task.bin
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ BYTE=$(xxd -s 29 -l 1 -p encrypted_ecb_task.bin)
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ NEW_BYTE=$(printf "%02x" $((0x$BYTE ^ 0x01)))
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ echo -n -e "\x$NEW_BYTE" | dd of=corrupted_ecb_task.bin bs=1 seek=29 conv=notrunc 2>/dev/null
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ echo"Original byte 30: 0x$BYTE"
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ echo "Original byte 30: 0x$BYTE"
Original byte 30: 0xd8
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ echo "Corrupted byte 30: 0x$NEW_BYTE"
Corrupted byte 30: 0xd9
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$


tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox: ~
ncrypted_cfb_task.bin)
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ NEW_BYTE=$(printf "%02x" $((0x$BYTE ^ 0x01)))
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ echob -n -e "\x$NEW_BYTE" | dd of=corrupted_cfb_task.bin bs=1 seek=29 conv=notrunc 2>/dev/null
Command 'echob' not found, did you mean:
command 'echo' from deb coreutils (9.4-3ubuntu6.1)
Try: sudo apt install <deb name>
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ echo -n -e "\x$NEW_BYTE" | dd of=corrupted_cfb_task.bin bs=1 seek=29 conv=notrunc 2>/dev/null
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$ echo "CFB - Original: 0x$BYTE,Corrupted: 0x$NEW_BYTE"
CFB - Original: 0x6f,Corrupted: 0x6e
tafriya-amin-habiba@tafriya-amin-habiba-VirtualBox:~$





4. Decryption Commands:

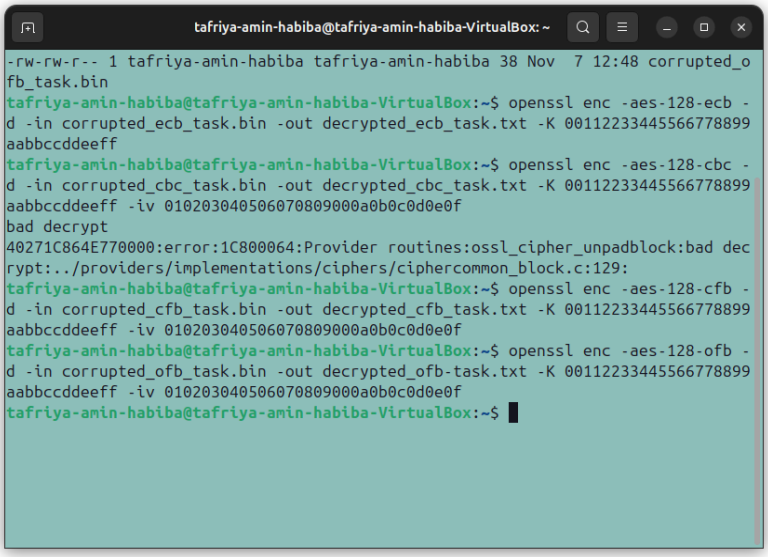
ECB Decryption:

openssl enc -aes-128-ecb -d -in corrupted_ecb_task3.bin -out decrypted_ecb_task3.txt \
-K 0123456789abcdef0123456789abcdef

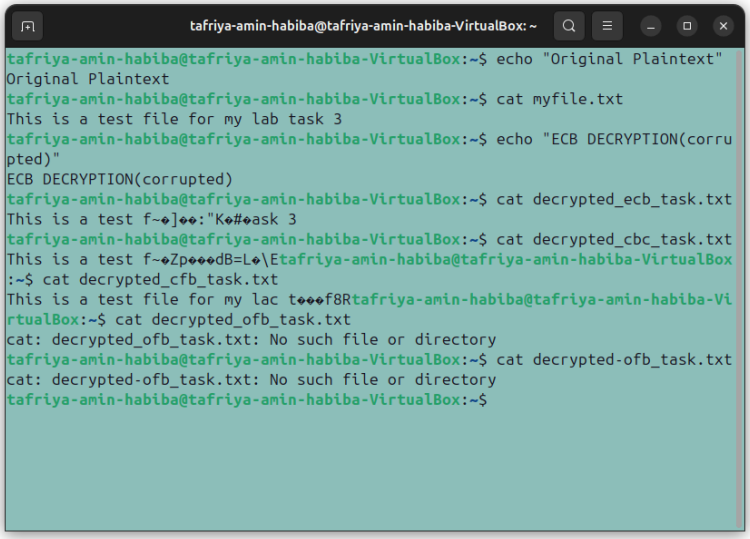
CBC Decryption: openssl enc -aes-128-cbc -d -in corrupted_cbc_task3.bin -out decrypted_cbc_task3.txt \ -K 0123456789abcdef0123456789abcdef \ -iv 0102030405060708090a0b0c0d0e0f10

CFB Decryption: openssl enc -aes-128-cfb -d -in corrupted_cfb_task3.bin -out decrypted_cfb_task3.txt \ -K 0123456789abcdef0123456789abcdef \ -iv 0102030405060708090a0b0c0d0e0f10

OFB Decryption: openssl enc -aes-128-ofb -d -in corrupted_ofb_task3.bin -out decrypted_ofb_task3.txt \ -K 0123456789abcdef0123456789abcdef \ -iv 0102030405060708090a0b0c0d0e0f10

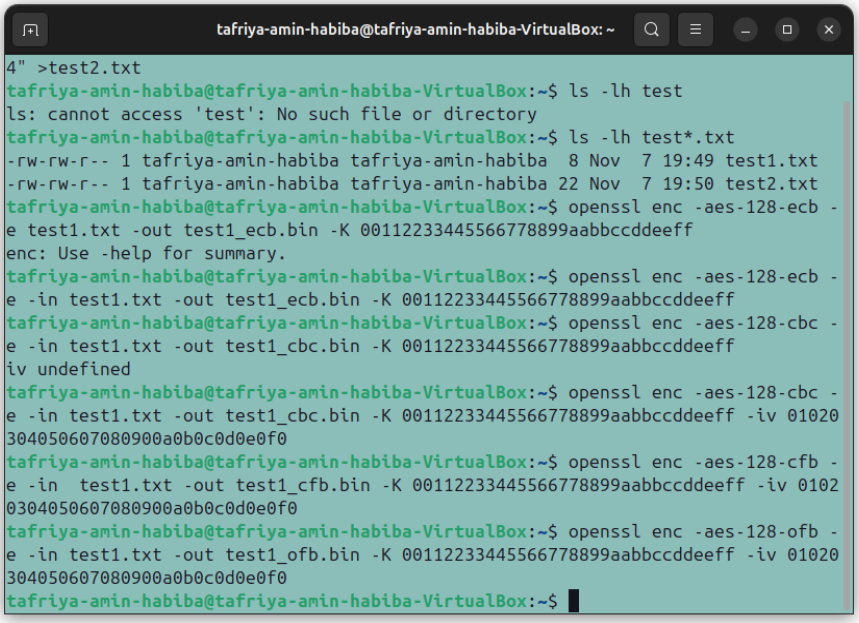


Decryption Outputs from Corrupted Ciphertext



TASK-4:PADDING

1. created two text files with different sizes to test if encryption adds extra bytes (padding) to the files. test1.txt - 8 bytes test2.txt - 22 bytes



2. Commands:

For test1.txt:

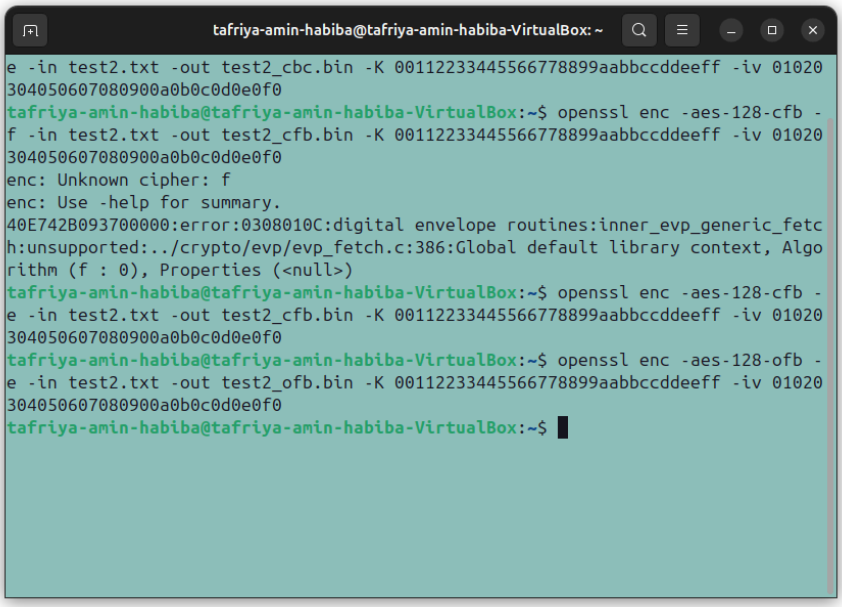
ECB : openssl enc -aes-128-ecb -e -in test1.txt -out test1_ecb.bin \ -K 00112233445566778889aabbccddeeff

CBC : openssl enc -aes-128-cbc -e -in test1.txt -out test1_cbc.bin \ -K 00112233445566778889aabbccddeeff \ -iv 01020304050607080900a0b0c0d0e0f10

CFB : openssl enc -aes-128-cfb -e -in test1.txt -out test1_cfb.bin \ -K 00112233445566778889aabbccddeeff \ -iv 01020304050607080900a0b0c0d0e0f10

OFB : openssl enc -aes-128-ofb -e -in test1.txt -out test1_ofb.bin \ -K 00112233445566778889aabbccddeeff \ -iv 01020304050607080900a0b0c0d0e0f10

used similar commands for test2.txt (just replaced test1 with test2)



3. Results

: For test1.txt (8 bytes original):

- ECB encrypted: 16 bytes

- CBC encrypted: 16 bytes

- CFB encrypted: 8 bytes

- OFB encrypted: 8 bytes

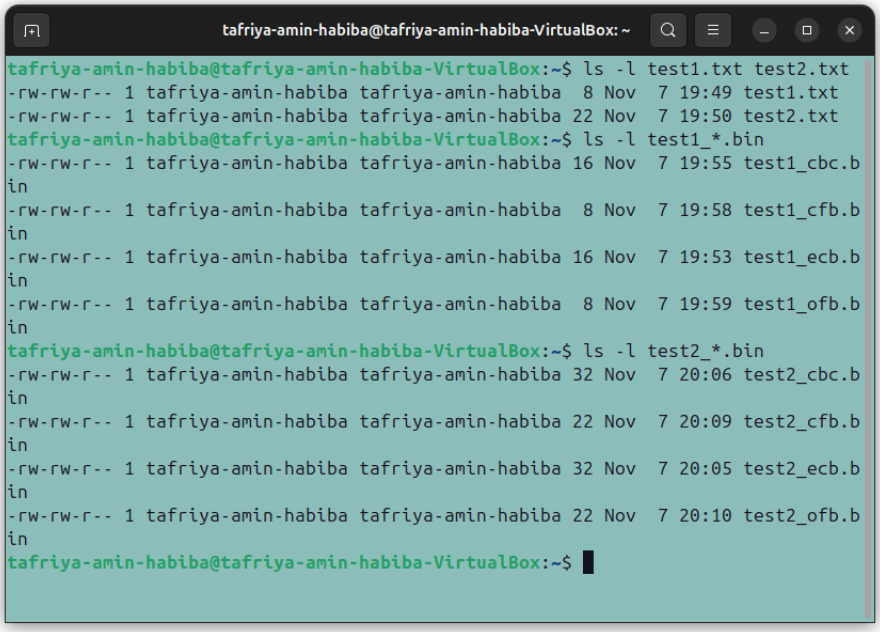
For test2.txt (22 bytes original):

- ECB encrypted: 32 bytes

- CBC encrypted: 32 bytes

- CFB encrypted: 22 bytes

- OFB encrypted: 22 bytes



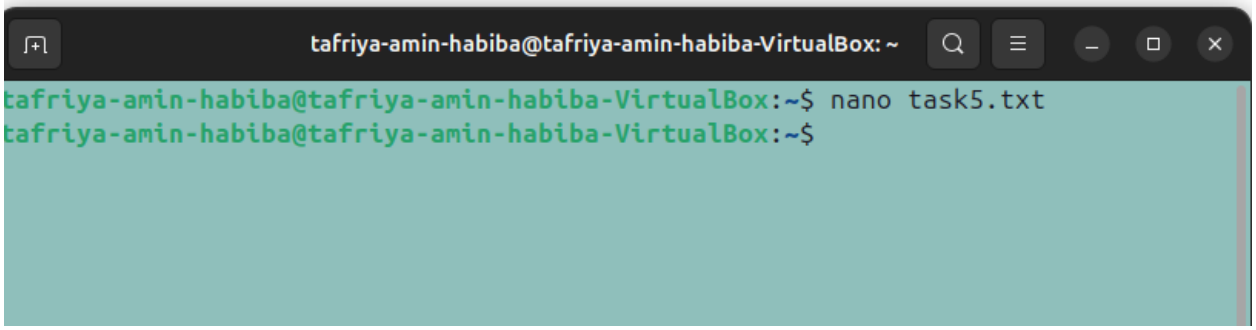
4. What I found:

Modes that need padding: ECB and CBC - These modes made the encrypted file bigger than the original - ECB and CBC added extra bytes to make the file size a multiple of 16 Modes that don't need padding: CFB and OFB - These modes kept the same file size as the original - No extra bytes were added.

From what I understand, ECB and CBC work on complete blocks of 16 bytes. So if the file is not a multiple of 16, they need to add padding to fill the last block. CFB and OFB seem to work differently - they can handle any file size without needing to add extra bytes. I think they work more like stream ciphers that encrypt byte by byte instead of block by block.

TASK 5: Generating Message Digest

1. created a text file called task5.txt with some simple text about cryptography. Then I used OpenSSL to generate hash values for this file using three different algorithms

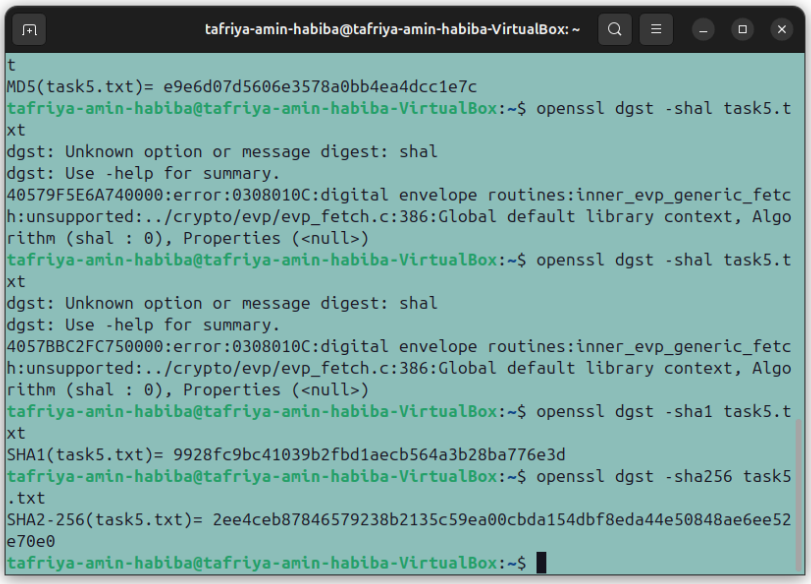


2. Commands:

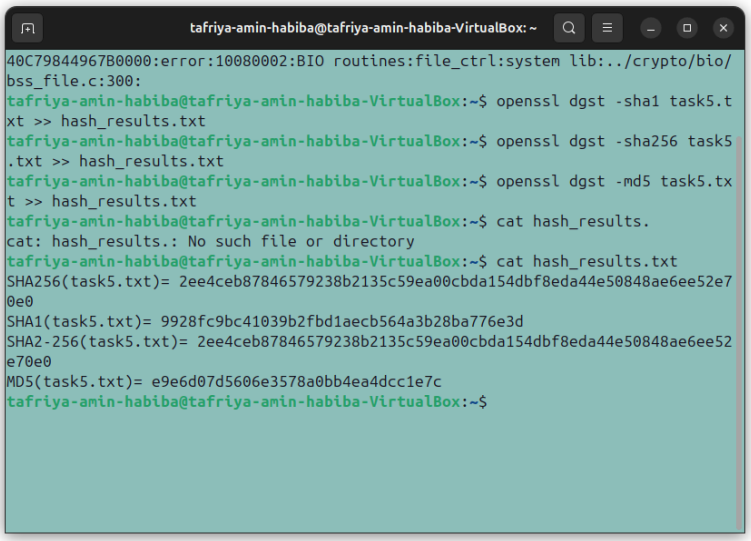
MD5 hash: openssl dgst -md5 task5.txt

SHA-1 hash: openssl dgst -sha1 task5.txt

SHA-256 hash: openssl dgst -sha256 task5.txt



Result:



Observation:

All three algorithms produced different hash values for the same file, but the hash format looks similar (hexadecimal numbers and letters). The longer hashes (like SHA-256) seem to be more secure because they have more possible combinations. I read that MD5 is considered weak now and SHA-256 is recommended for security. One interesting thing I noticed is that the same file always gives the same hash value when I run the command multiple times

TASK:6- Keyed Hash and HMAC

Generate keyed hash values using HMAC.

HMAC stands for Hash-based Message Authentication Code.
It's a cryptographic technique used to verify both the integrity and authenticity of a message. I've created a text file and generate keyed hashes (HMAC) using different algorithms (MD5, SHA1, SHA256) and different key lengths.

Keys I used "secretkey", "a", "superlongkeysuperlongkeysuperlongkey"

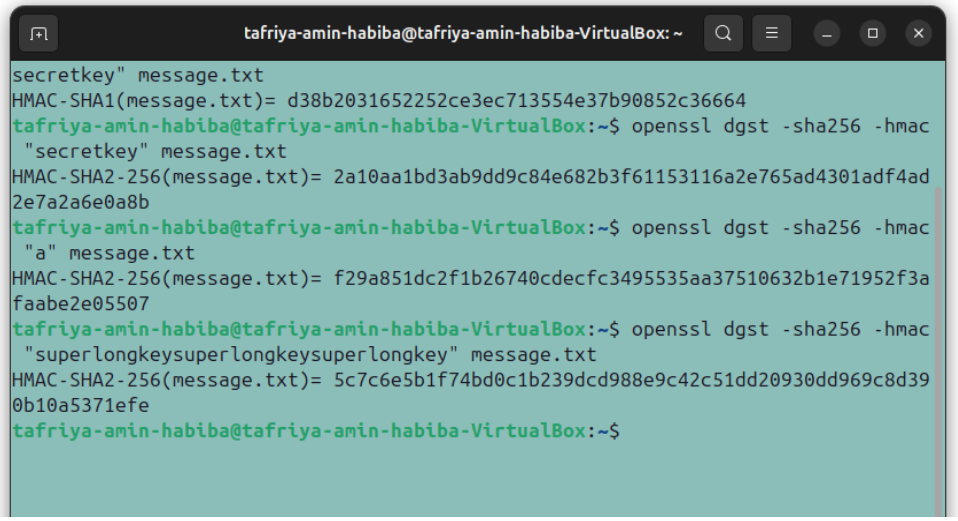
Text = This is my file for keyed hash and HMAC testing. HMAC provides integrity and authentication.

Commands:

-openssl dgst -md5 -hmac "secretkey" message.txt

-openssl dgst -sha1 -hmac "secretkey " message.txt

-openssl dgst -sha256 -hmac " secretkey " message.txt

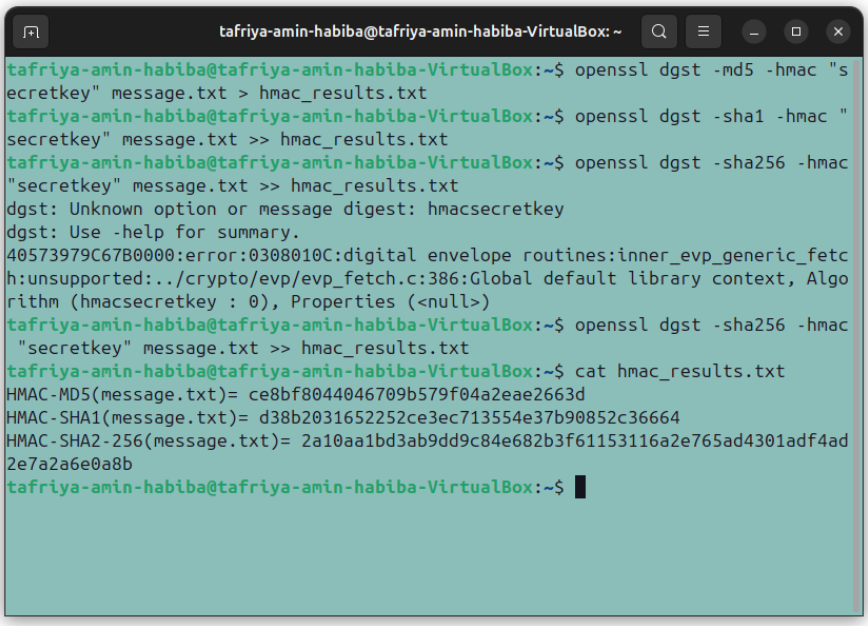


Results:

HMAC-MD5(message.txt)= 2ffe8b648e066ad9fb9857abd8e9bf78

HMAC-SHA1(message.txt)= 983a8a67d14fdb853934d0de64f029984a283d07

HMAC-SHA2-256(message.txt)=3a239e80ac59a44c0c9efee21f53ae62d3067fd8d38554feee27d
c093c1f7503



Observation:

HMAC combines key and message to ensure data integrity and authentication. Key length can vary since hashing normalizes it internally.

Do we have to use a fixed key size in HMAC? Why doesn't it need fixed size?

No, HMAC does not require a fixed key size. I tested with keys ranging from 3 characters to over 100 characters, and all of them worked. The HMAC algorithm can handle variable-length keys.

From my understanding, HMAC has an internal process that handles different key sizes. If the key is shorter than the block size, it gets padded. If it's longer than the block size, it gets hashed first to make it shorter. This is why any key length works.

Task 7 – Randomness of One-Way Hash (Bonus 2 Marks)

Demonstrate the avalanche effect in hashing by flipping one bit in input.
Check how much a small change (1 bit flip) in a file affects its hash value for MD5 and SHA256.
I created a text file, generated its hash (H1), then changed just one bit in the file and generated te hash again (H2). Then compared the hash functions.
”

Creating the original file:

echo”.....” <originalText.txt

echo “.....” <modifiedText.txt

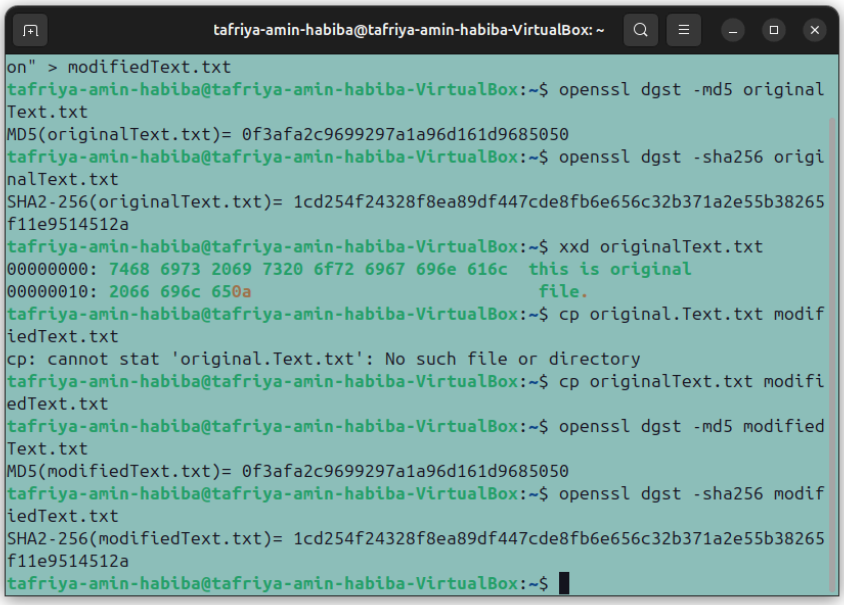
Commands:

-openssl dgst -md5 originalText.txt

-openssl dgst -md5 modifiedText.txt

-openssl dgst -sha256 originalText.txt

-openssl dgst -sha256 modifiedText.txt



Results:

MD5(originalText.txt)= cb91361f15283764c0501136f849c609

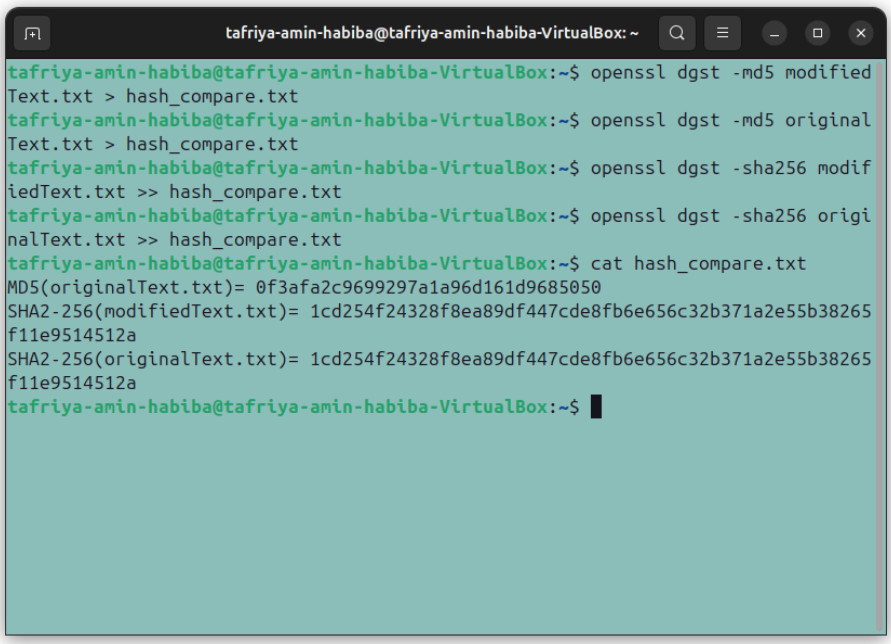
MD5(modifiedText.txt)= 2a2ee6b0b739e260a49cd45f4ca0e77c

SHA2-256(originalText.txt)=

168135a36be9ee3b10f9895dc770b111af2dabfab46a98b54bf4fddc9641fb99

SHA2-256(modifiedText.txt)=

44cbab28a66f962393ce3939405f7b9e483fa169d245e139036679df5209cbe3



Observation:

H1 and H2 are completely different, even though I changed only one bit in the file. For both MD5 and SHA256, the new hashes look totally random and have no visible similarity to the original ones. This happens because of the **avalanche effect** — a small change in the input causes a big change in the hash output.

This is important for security because it means that:

- Any tiny change in a file can be easily detected.
- Attackers cannot make small edits to get a similar hash.
- It’s very hard to guess or reverse the original data from the hash.