LAB 3:Symmetric encryption & hashing

Initial Setup:

I used my Mac for this task. For file editing, I used HexFiend (hex editor) and TextEdit (for text files). For viewing images, I used Preview.

Key and IV:

KEY = 00112233445566778899aabbccddeeff (16 bytes)

IV = 0102030405060708090a0b0c0d0e0f10 (16 bytes)

Task1: AES encryption using different modes

I created a simple text file named plain.txt and performed encryption and decryption using AES-128 in different modes.

CBC Encryption & Decryption:

1. openssl enc -aes-128-cbc -e -in plain.txt -out cipher_cbc.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
2. openssl enc -aes-128-cbc -d -in cipher_cbc.bin -out dec_cbc.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

ECB Encryption & Decryption:

3. openssl enc -aes-128-ecb -e -in plain.txt -out cipher_ecb.bin -K 00112233445566778899aabbccddeeff
4. openssl enc -aes-128-ecb -d -in cipher_ecb.bin -out dec_ecb.txt -K 00112233445566778899aabbccddeeff

CFB Encryption & Decryption:

5. openssl enc -aes-128-cfb -e -in plain.txt -out cipher_cfb.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
6. openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out dec_cfb.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

Files Created:

Encrypted Files:

cipher_cbc.bin

cipher_ecb.bin

cipher_cfb.bin

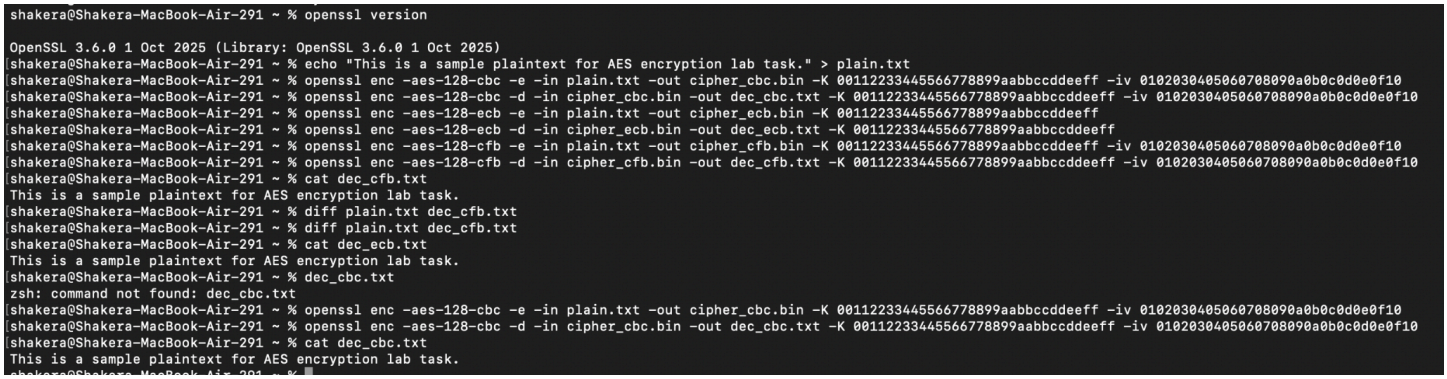
Decrypted Files:

dec_cbc.txt

dec_ecb.txt

dec_cfb.txt

I have added all the encrypted and decrypted files to the Lab3 folder.



Task2 : Encryption Mode - ECB vs CBC

The goal of this task was to compare the effects of ECB (Electronic Code Book) and CBC (Cipher Block Chaining) encryption modes on an image file. We encrypted a BMP image using both modes and observed the differences in the encrypted images.

Encrypting the Image:

The image pic_original.bmp was encrypted using AES-128 in ECB and CBC modes:

ECB Encryption:
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.enc -K 00112233445566778899aabbccddeeff

CBC Encryption:
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.enc -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

Replacing the Header:

1. The first 54 bytes of the encrypted images were replaced with the header from the original image to ensure the file was recognized as a valid BMP file.

2. This was done using a hex editor (HxD).

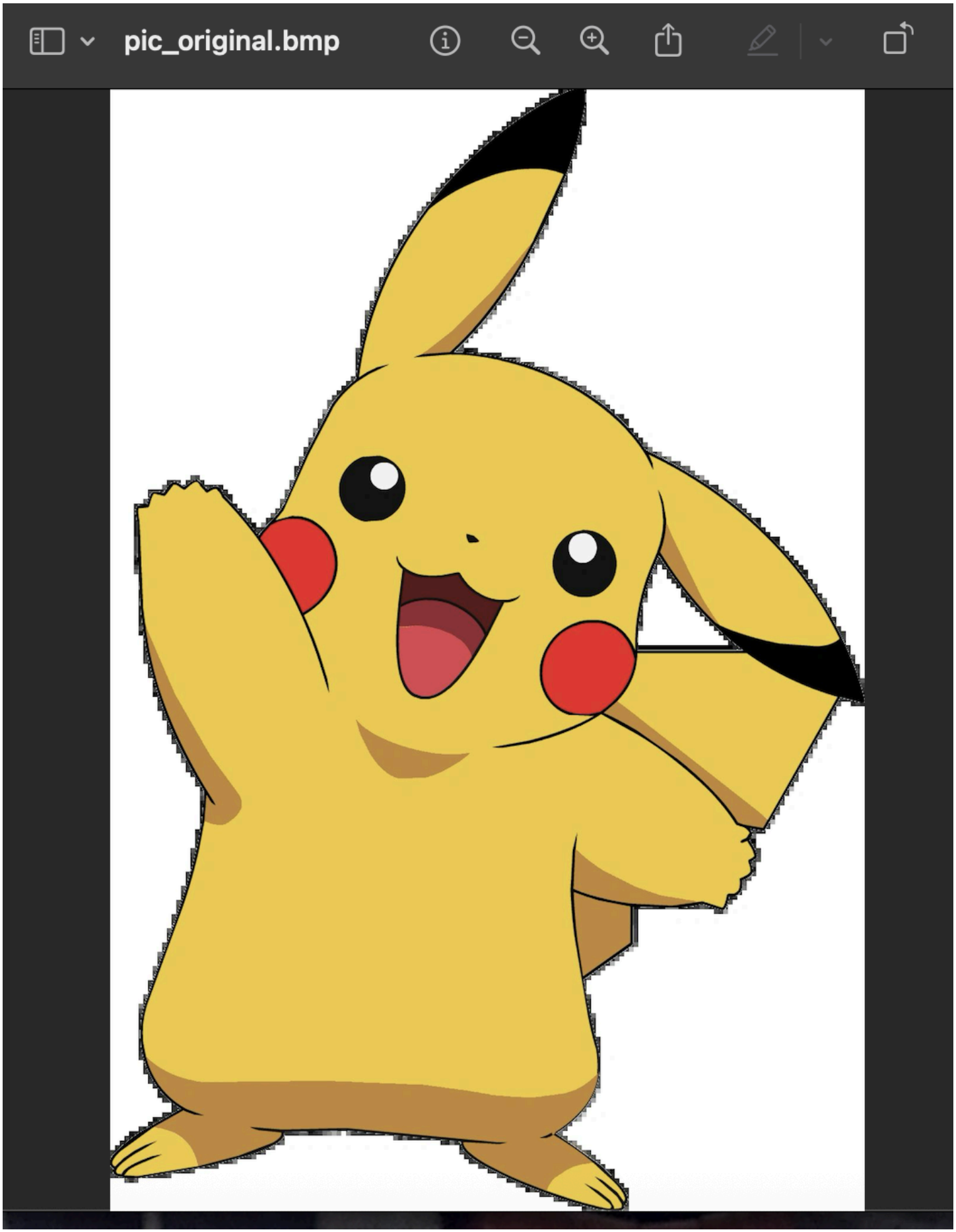
Displaying the Encrypted Images:

The encrypted images were opened in a photo viewer to observe the results.

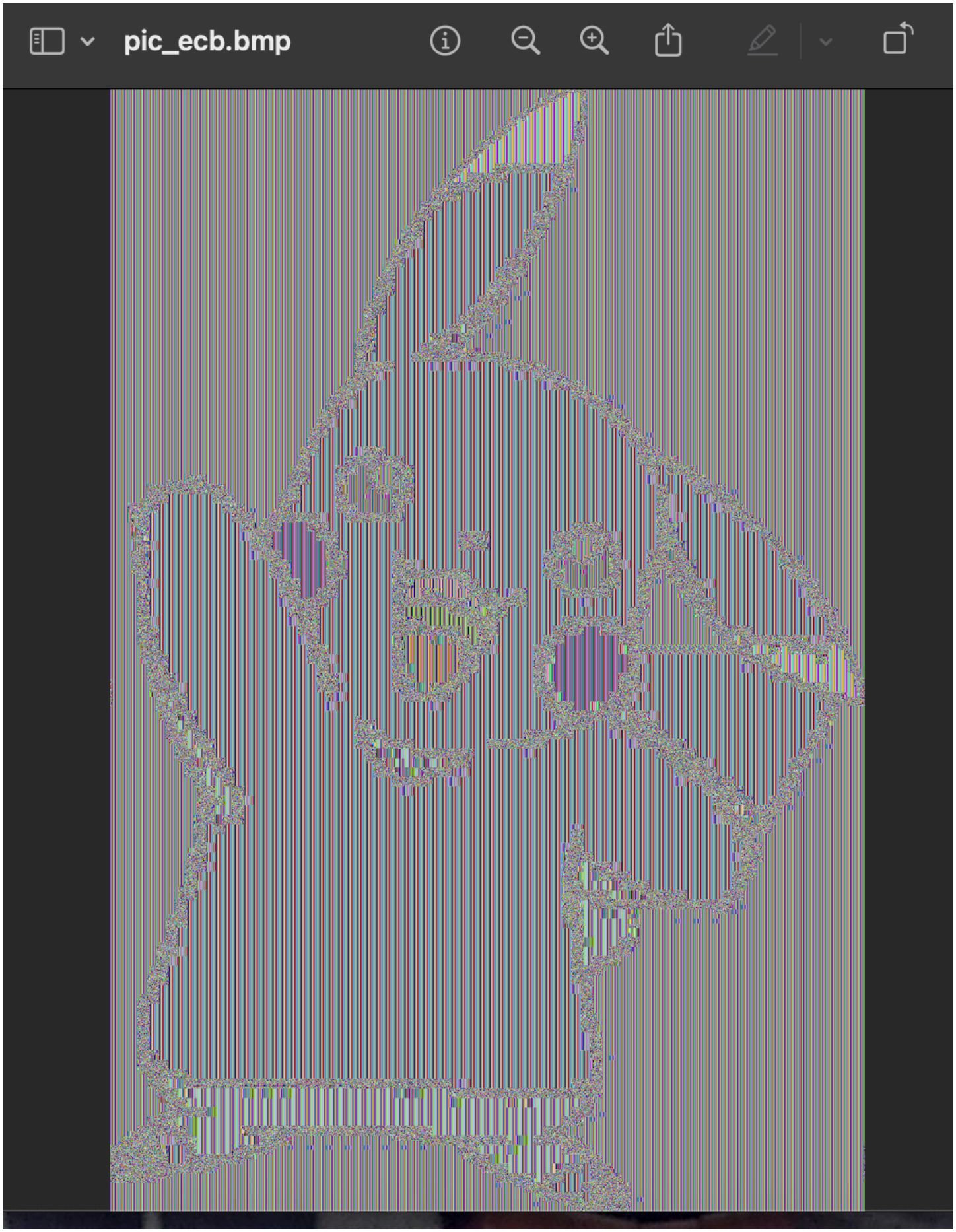
Image encryption:
https://taro.codes/posts/2022-10-27-encryption-ecb-bitmaps/

https://taro.codes/posts/2022-10-27-encryption-ecb-bitmaps/assets/input/pikachu.jpg

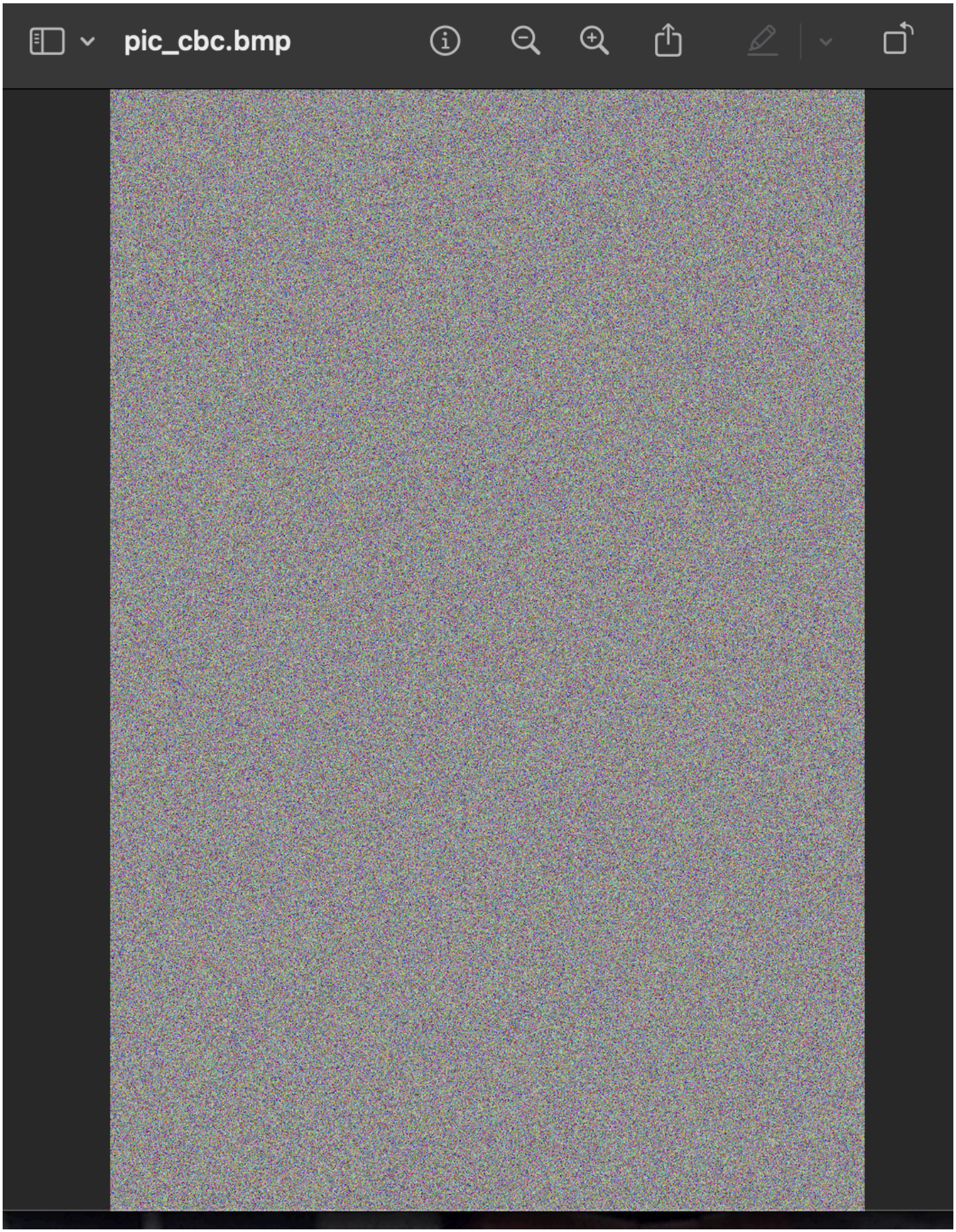
ORIGINAL



ECB



CBC



Results:

- ➔ **ECB Mode**:
  The ECB-encrypted image displayed visible patterns that resembled parts of the original image. This is because ECB encrypts each block independently, so identical blocks in the image result in identical encrypted blocks, revealing structural patterns.
- ➔ **CBC Mode**:
  The CBC-encrypted image appeared as random noise, with no visible patterns. This is because CBC mode encrypts each block based on the previous ciphertext and the initialization vector (IV), making it resistant to pattern leakage.

Conclusion:

- **ECB** mode is insecure for encrypting images as it allows patterns to remain visible, revealing information about the original image.
- **CBC** mode, however, hides the patterns by chaining block encryptions, making the encrypted image look like random noise and providing better security.

Task 3 :

First I created a txt file (named it long.txt: "This is a long sample text file with more than sixty-four bytes to test AES encryption modes. So text is longer than paintxt.") of 69bytes. Then,

Encryption:
ECB:
openssl enc -aes-128-ecb -e -in long.txt -out c_ecb.bin -K 00112233445566778899aabbccddeeff
CBC:
openssl enc -aes-128-cbc -e -in long.txt -out c_cbc.bin -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10
CFB:
openssl enc -aes-128-cfb -e -in long.txt -out c_cfb.bin -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10
OFB:
openssl enc -aes-128-ofb -e -in long.txt -out c_ofb.bin -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

Unfortunately, a single bit of the **30th byte** in the encrypted file got corrupted(Used HxD). Now,

Decryption:
ECB:
openssl enc -aes-128-ecb -d -in c_ecb_corrupt.bin -out d_ecb_corrupt.txt -K 00112233445566778899aabbccddeeff
CBC:
openssl enc -aes-128-cbc -d -in c_cbc_corrupt.bin -out d_cbc_corrupt.txt -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

CFB:

openssl enc -aes-128-cfb -d -in c_cfb_corrupt.bin -out d_cfb_corrupt.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

OFB:

openssl enc -aes-128-ofb -d -in c_ofb_corrupt.bin -out d_ofb_corrupt.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

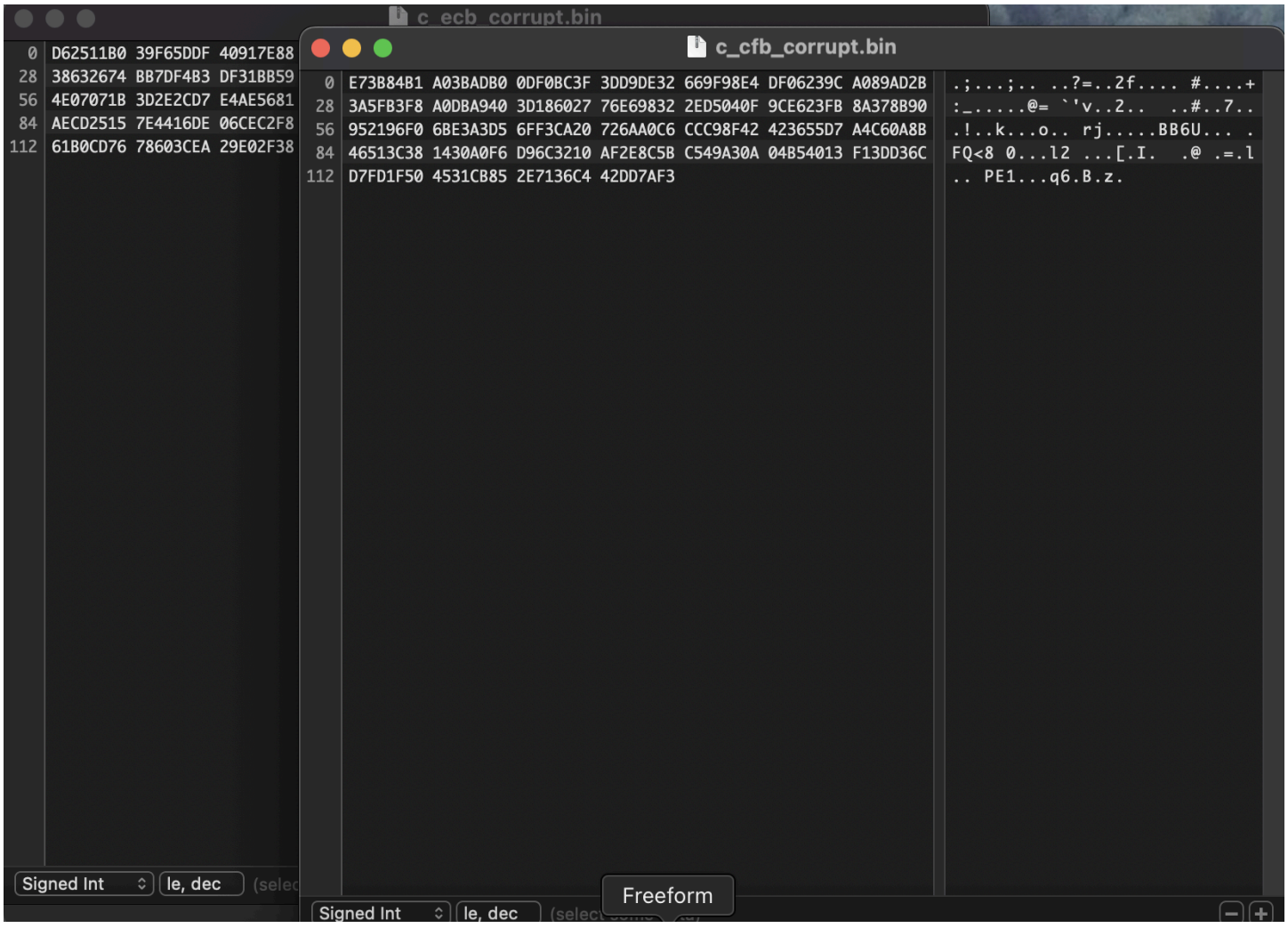
Encryption:

```
shakera@Shakera-MacBook-Air-291 ~ % echo "This is a long sample text file with more than sixty-four bytes to test AES encryption modes. So text is longer than paintxt." > long.txt
shakera@Shakera-MacBook-Air-291 ~ % openssl enc -aes-128-ecb -e -in long.txt -out c_ecb.bin -K 00112233445566778899aabbccddeeff
shakera@Shakera-MacBook-Air-291 ~ % openssl enc -aes-128-cbc -e -in long.txt -out c_cbc.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
shakera@Shakera-MacBook-Air-291 ~ % openssl enc -aes-128-cfb -e -in long.txt -out c_cfb.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
shakera@Shakera-MacBook-Air-291 ~ % openssl enc -aes-128-ofb -e -in long.txt -out c_ofb.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
```

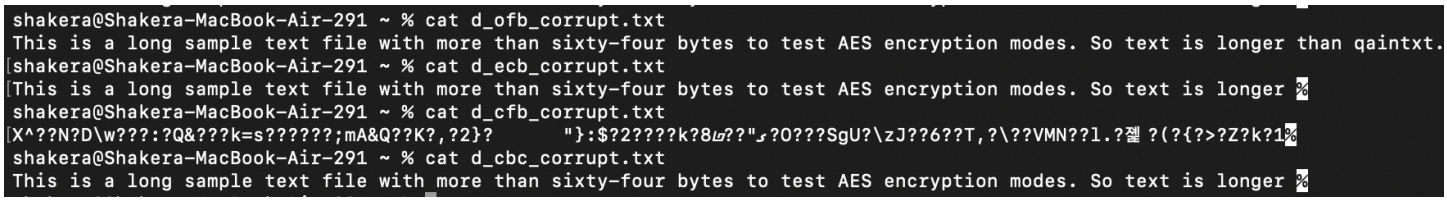
Decryption:

```
shakera@Shakera-MacBook-Air-291 ~ % openssl enc -aes-128-ecb -d -in c_ecb_corrupt.bin -out d_ecb_corrupt.txt -K 00112233445566778899aabbccddeeff
bad decrypt
40A100F101000000:error:1C800064:Provider routines:ossl_cipher_unpadblock:bad decrypt:providers/implementations/ciphers/ciphercommon_block.c:107:
shakera@Shakera-MacBook-Air-291 ~ % openssl enc -aes-128-ofb -d -in c_ofb_corrupt.bin -out d_ofb_corrupt.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

shakera@Shakera-MacBook-Air-291 ~ % openssl enc -aes-128-cbc -d -in c_cbc_corrupt.bin -out d_cbc_corrupt.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

bad decrypt
40A100F101000000:error:1C800064:Provider routines:ossl_cipher_unpadblock:bad decrypt:providers/implementations/ciphers/ciphercommon_block.c:107:
shakera@Shakera-MacBook-Air-291 ~ % openssl enc -aes-128-cfb -d -in c_cfb_corrupt.bin -out d_cfb_corrupt.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
```

Corrupting:



Matching :



After all these,

1. How much information can you recover by decrypting the corrupted file?

1. **ECB Mode:** In ECB mode, only the corrupted block is affected. All other blocks can be decrypted correctly, so most of the file can be recovered.
2. **CBC Mode:** In CBC mode, corruption in one block affects the corrupted block and the next one. Therefore, two blocks will be affected, but the rest of the file can still be decrypted correctly.
3. **CFB Mode:** CFB is a stream cipher, so a single corrupted byte only affects that byte in the decrypted file. The rest of the file is unaffected.
4. **OFB Mode:** Like CFB, OFB only affects the corrupted byte, and the rest of the file can be recovered.

2. Explain why:

1. **ECB Mode:** ECB works by encrypting each block independently. Since blocks are processed separately, only the corrupted block is affected by the corruption, and the other blocks remain intact.
2. **CBC Mode:** CBC uses chaining, where the encryption of each block depends on the previous block. If one block is corrupted, it affects both the corrupted block and the next block, but the rest of the file remains unaffected.
3. **CFB and OFB Modes:** Both CFB and OFB are stream ciphers, processing the data bit by bit or byte by byte. A single corrupted byte will only affect that byte, leaving the rest of the file intact.

3. What are the implications of these differences?

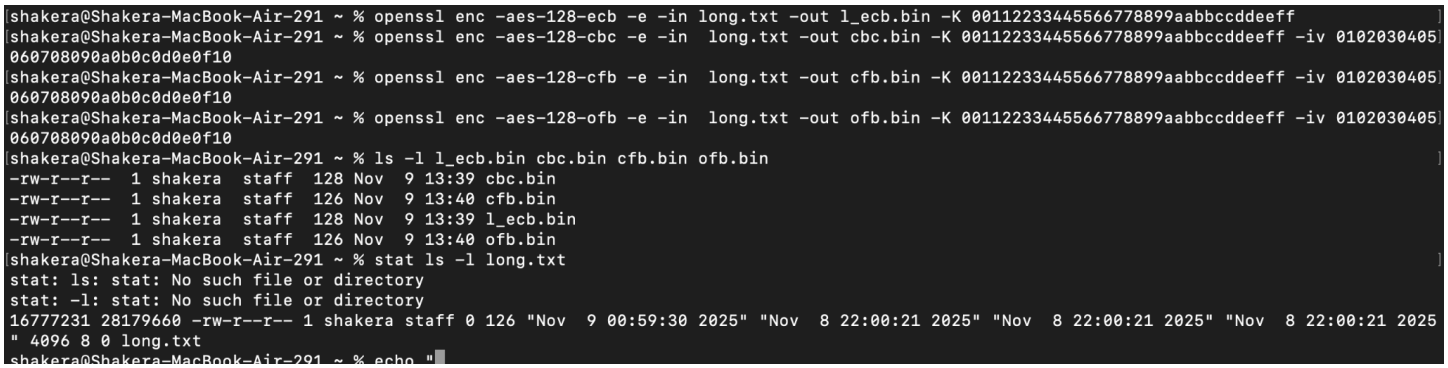
1. **ECB Mode:** While ECB is simple to implement, it is insecure because it reveals patterns in the data. It's also less resilient to data corruption because a corrupted byte only affects one block.
2. **CBC Mode:** CBC is more secure than ECB because it hides data patterns. However, it is less resilient to corruption since a single corrupted byte affects two blocks.
3. **CFB/OFB Modes:** CFB and OFB are more resilient to corruption, as they only affect one byte of data. These modes are better for environments where errors are more likely to occur, as they prevent widespread data loss.

Verdict:

- CFB and OFB are better for handling corrupted data since only one byte is affected by the corruption.

- ECB and CBC offer stronger security, but CBC's error propagation makes it less resilient in environments where data corruption is common.

Task 4:



Encryption:

openssl enc -aes-128-ecb -e -in long.txt -out l_ecb.bin -K 00112233445566778899aabbccddeeff
openssl enc -aes-128-cbc -e -in long.txt -out cbc.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
openssl enc -aes-128-cfb -e -in long.txt -out cfb.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
openssl enc -aes-128-ofb -e -in long.txt -out ofb.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

After encrypting, I observed the following:

1. CFB and OFB modes result in encrypted files that are the same size as the original file (126 bytes).
2. ECB and CBC modes result in files that are slightly larger (128 bytes).

Padding Explanation:

Block ciphers like AES operate on fixed-size blocks (16 bytes for AES). When the plaintext length is not an exact multiple of the block size, padding is required to fill the last block.

1. CFB and OFB work on stream data, and do not require padding, which is why their file sizes remain the same as the original file.
2. ECB and CBC, however, work with fixed-size blocks and need padding if the plaintext is not a multiple of the block size. In this case, the plaintext was 126 bytes, and the last 2 bytes required padding (128-126 = 2). Hence, the encrypted files in ECB and CBC modes are larger.

Conclusion:

- CFB and OFB do not require padding, so their file sizes remain the same.
- ECB and CBC require padding, and thus their file sizes are slightly larger than the original.

Task5:

To generate and compare message digests of a file using different one-way hash algorithms (MD5, SHA-1, and SHA-256).

```
shakera@Shakera-MacBook-Air-291 ~ % openssl dgst -md5 plain.txt
openssl dgst -sha1 plain.txt
openssl dgst -sha256 plain.txt

MD5(plain.txt)= 6fe1c552afea0a0891a30ea18814eac4
SHA1(plain.txt)= 7b80b6c32a97c25db6fdf85fd5de79abd752506f
SHA2-256(plain.txt)= e1f2b59802b1c9ac6e69937738a9d3027539aeab5154dc9e0c07e15e58ece75c
shakera@Shakera-MacBook-Air-291 ~ %
```

The following openssl dgst commands were used to generate hashes for plain.txt:

openssl dgst -md5 plain.txt
openssl dgst -sha1 plain.txt
openssl dgst -sha256 plain.txt

Results:

- ➔ MD5: 6fe1c552afea0a0891a30ea18814eac4
- ➔ SHA-1: 7b80b6c32a97c25db6fdf85fd5de79abd752506f
- ➔ SHA-256: e1f2b59802b1c9ac6e69937738a9d3027539aeab5154dc9e0c07e15e58ece75c

Observations:

- ➔ MD5 produces a 128-bit hash (32 characters).
- ➔ SHA-1 produces a 160-bit hash (40 characters).
- ➔ SHA-256 produces a 256-bit hash (64 characters).

Conclusion:
Each algorithm produces a unique hash for the same file. SHA-256 is more secure than MD5 and SHA-1, making it preferable for cryptographic applications.

Task 6: Keyed Hash and HMAC

The goal of this task was to generate keyed hashes (MACs) for a file using HMAC with different hash algorithms (MD5, SHA-1, SHA-256) and varying key lengths.

We used the following openssl commands to generate the HMACs for the file plain.txt:

openssl dgst -md5 -hmac "key123" plain.txt

openssl dgst -sha1 -hmac "key123" plain.txt

openssl dgst -sha256 -hmac "key123" plain.txt

openssl dgst -sha256 -hmac "a much longer sample key" plain.txt

Results:

1. HMAC-MD5:
   HMAC-MD5(plain.txt) = c8367b6961b7230955ec7449f1e45769
2. HMAC-SHA1:
   HMAC-SHA1(plain.txt) = 046b6a8c5c5b1be1b777568f2c70486fa0a72031
3. HMAC-SHA256 (key "key123"):
   HMAC-SHA2-256(plain.txt) =
   320094d80dbf38526357d6349a215d4206c18c17cbb529124768723a0839c7c0
4. HMAC-SHA256 (key "a much longer sample key"):
   HMAC-SHA2-256(plain.txt) =
   beac952b0b03c96bf13d1760c8ad98b565a381e0753a961acc022f517f2a5efc

Observations:

- HMAC allows the use of keys with varying lengths. There is no fixed key size required, but longer keys may improve security.
- The HMAC-MD5 produces a 128-bit hash, HMAC-SHA1 produces a 160-bit hash, and HMAC-SHA256 produces a 256-bit hash, regardless of the key length.

Conclusion:
HMAC can be used with keys of different lengths, and the size of the hash remains consistent for each algorithm. Longer keys enhance security, but they do not change the hash size. This experiment showed how HMAC can be used to ensure data integrity with flexible key sizes.

Task 7

The goal of this task was to observe how a small change in a file affects the hash value generated by the SHA-256 algorithm. Specifically, I modified the file by flipping one bit and compared the original and modified hash values.

1. Created a text file and generated its SHA-256 hash:

   openssl dgst -sha256 plain.txt > hash1.txt

2. Flipped one bit of the file using Perl:

   perl -pe 's/A/B/ if $.==1' plain.txt > plain_flipped.txt

3. Generated the SHA-256 hash for the modified file:

openssl dgst -sha256 plain_flipped.txt > hash2.txt

4. Compared the original and modified hash values using `diff`:

   diff hash1.txt hash2.txt

Results:

1. Original File Hash (H1):

   SHA2-256(plain.txt) = e1f2b59802b1c9ac6e69937738a9d3027539aeab5154dc9e0c07e15e58ece75c

2. Modified File Hash (H2):

   SHA2-256(plain_flipped.txt) = 3f0ff83545174a3d8a9eda8a35fd3f80230edd066df40974efd2428db799da32

Observations:

The hash values (H1 and H2) are completely different, showing the sensitivity of SHA-256 to even small changes in the input. This is an example of the avalanche effect, where even a single bit flip drastically alters the hash.

Conclusion:
Flipping just one bit in the file results in a completely different hash, demonstrating that cryptographic hash functions are highly sensitive to changes in the input data.