Report on
Lab 3: Symmetric encryption & hashing

Initial Setup

Since I have a windows PC, I will use HxD hex editor software for editing the files. I will use Windows Notebook and Photo Viewer tools to create text files and view images.

Key and IV(for all tasks)

KEY = 00112233445566778899aabbccddeeff (32 hex = 16 bytes)
IV = 010203040506070809 0a0b0c0d0e0f10 (32 hex = 16 bytes)

Task 1

CBC Encryption & Decryption:

- openssl enc -aes-128-cbc -e -in plain.txt -out cipher_cbc.bin -K 00112233445566778899aabbccddeeff -iv 010203040506070809 0a0b0c0d0e0f10
- openssl enc -aes-128-cbc -d -in cipher_cbc.bin -out dec_cbc.txt -K 00112233445566778899aabbccddeeff -iv 010203040506070809 0a0b0c0d0e0f10

ECB Encryption & Decryption:

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

CFB Encryption & Decryption:

- openssl enc -aes-128-cfb -e -in plain.txt -out cipher_cfb.bin -K 00112233445566778899aabbccddeeff -iv 010203040506070809 0a0b0c0d0e0f10
- openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out dec_cfb.txt -K 00112233445566778899aabbccddeeff -iv 010203040506070809 0a0b0c0d0e0f10

I created a simple txt file named plain.txt and encrypted it using the encryption commands. Then I decrypted the encrypted files using the decryption commands. I have added the encrypted and decrypted files in the Task1 folder.

Encrypted files name: cipher_cbc.bin, cipher_ecb.bin, cipher_cfb.bin
Decrypted files name: dec_cbc.txt, dec_ecb.txt, dec_cfb.txt

Task 2

First I created a BMP picture from a JPG format picture and named it pic_original.

Encryption:
ECB:
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.enc -K 00112233445566778899aabbccddeeff
CBC:
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.enc -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

After encrypting I used HxD to replace the header(the first 54 bytes) of the encrypted picture with that of the original picture. Then I saved the files as bmp files and then opened them in the photos app.
Observation:
I cannot see the previous images on CBC but for ECB I can see some patterns leaking. Because it doesn't mix block outputs. But CBC hides patterns because each block encryption depends on previous ciphertext + IV. That's why the ECB-encrypted image looks structured, but the CBC-encrypted image looks like random noise.

Task3

First I created a txt file of 69bytes. Then,

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

After decryption,

Question No 1 and 2: How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? And why?

Ans:

For ECB, Only the corresponding block is corrupted on decryption. Because each block is encrypted independently. Corruption doesn't propagate beyond that block.

For CBC, The current block and the next block become corrupted on decryption. Each plaintext block is XOR-ed with the previous ciphertext block. So, one corrupted ciphertext block breaks its own decryption and corrupts the next block.

For CFB, the corruption affects a few bytes but doesn't ruin the rest. CFB uses the previous ciphertext as input to the encryption step, so errors propagate for only a few bytes of plaintext, not indefinitely.

For OFB, Only one byte (or one block) is corrupted. OFB generates a keystream independent of the plaintext; errors in ciphertext affect only matching bits on decryption, not future bytes.

Question no 3: What are the implications of these differences?

Ans: Reasons are-

1. **ECB:** Good error isolation but poor security (pattern preservation)
2. **CBC:** Better security but error propagation can corrupt multiple blocks
3. **CFB:** Moderate error propagation, suitable for real-time applications
4. **OFB:** Minimal error propagation, good for applications requiring error tolerance

Task 4

Encryption:
openssl enc -aes-128-ecb -e -in plain.txt -out ecb.bin -K 00112233445566778899aabbccddeeff
openssl enc -aes-128-cbc -e -in plain.txt -out cbc.bin -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10
openssl enc -aes-128-cfb -e -in plain.txt -out cfb.bin -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10
openssl enc -aes-128-ofb -e -in plain.txt -out ofb.bin -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

After Encrypting I can see that CFB and OFB have the same file size as the original file**(69byte)**. On the other hand the ECB and CBC files have become slightly larger**(80byte)**.

Padding exists for

1. Block ciphers (like AES) work on fixed-size blocks (16 bytes for AES).
2. If the plaintext isn't an exact multiple of the block size, we must pad the final block to make it fit.
3. Padding is added before encryption and removed after decryption.

Since CFB and OFB works on **stream data and doesn't need padding**. The file size remains the same. But ECB and CBC works **block by block and needs padding**, the last **5byte**(69-64=5) needed padding and thus the file size got larger. Files attached to the Task3 folder.

Task 5

Commands and generated hash values are shown below:

openssl dgst -md5 plain.txt

MD5(plain.txt)= 3f500f75f9b98c467287eede5e2d0494

openssl dgst -sha1 plain.txt

SHA1(plain.txt)= 1764ff86fd96183a58043c6ca651904fc126c0f4

openssl dgst -sha256 plain.txt

SHA256(plain.txt)= 1b28918dd1b7183d1d250d8f0b58a0f77b4cab66ed0caeee06b078843e407630

Observations

1. **Output Length:** Each algorithm produces a fixed-length output regardless of input size.
2. **Avalanche Effect:** Small changes in input produce completely different hash values.
3. **Deterministic:** Same input always produces the same hash.
4. **One-way Function:** Hash values cannot be reversed to obtain the original input.
5. **Security:** SHA256 is currently secure, while MD5 and SHA1 are deprecated.

Task 6

Commands and generated hash values are shown below:

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

HMAC-MD5(plain.txt)= 5bc4062d0430fedfc425731924096f19

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

HMAC-SHA1(plain.txt)= 5e3b171accf1a43accd6333e2ac006b67594478e

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

HMAC-SHA256(plain.txt)= d470ba29ed5c8061121d1baa12867689067ece590465fda185c892d005e57509

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

HMAC-SHA256(plain.txt)= 733eef35cc01faac47359fdb1b4fe68b7cf8d12f832c4fe033a6aff025471a90

Question: Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?
Ans:

No. HMAC accepts keys of any length.

1. If the key > hash block size, it is first hashed (shortened).
2. If the key < hash block size, it is zero-padded to the block size.

HMAC automatically adjusts any key length, so it hashes long keys and pads short ones.

Task7

Commands and generated hash values are shown below:

openssl dgst -md5 plain.txt

MD5(plain.txt)= 3f500f75f9b98c467287eede5e2d0494

openssl dgst -sha256 plain.txt

SHA256(plain.txt)= 1b28918dd1b7183d1d250d8f0b58a0f77b4cab66ed0caeee06b078843e407630

After flipping one bit:

openssl dgst -md5 plain_flipped.txt

MD5(plain_flipped.txt)= 6c55a5cdd966347823fce46d26a99fda

openssl dgst -sha256 plain_flipped.txt

SHA256(plain_flipped.txt)= 3c3d99bf07397db2dc423a7591d48a0db721873b5d2c46cacd2f0dad0b417e5d

The test file and the program to count how many bits are the same between H1 and H2 is shown in the github code.

Observations

1. **Complete Change:** Flipping a single bit in the input file resulted in completely different hash values for both MD5 and SHA256
2. **No Similarity:** There is no visible similarity between H1 and H2, confirming the one-way property of hash functions
3. **Deterministic:** The same input always produces the same hash, but any change produces a completely different hash

Bonus

A short program to count how many bits are the same between H1 and H2 is attached to github.

Results

MD5: 66/128 bits are the same.

SHA256: 135/256 bits are the same.