Lab Report — Symmetric Encryption & Hashing

Tools Used:
Ubuntu Bash Terminal, OpenSSL, GHex (Hex Editor)

Objective

To perform experiments with symmetric encryption (AES in multiple modes), hashing algorithms, and keyed hashing (HMAC). The aims were to explore encryption/decryption behavior, visualize the effect of block cipher modes on images, observe the impact of ciphertext corruption, understand padding requirements, and compare digest outputs across algorithms.

Environment & Setup

- Ubuntu environment running under Windows subsystem
- OpenSSL pre-installed (`openssl` command available)
- GHex installed for manual byte-level editing
- Workspace contained: plaintext files, encrypted binary files, BMP images, hash outputs, and HMAC outputs
- All commands were executed directly from the terminal

Task 1 — AES Encryption With Different Modes

Goal

Encrypt and decrypt a text file using at least three AES modes (CBC, ECB, CFB) and verify correctness.

Files Used

`plain.txt` — a simple text file.

Commands Executed (Examples)

AES-128-CBC

```
openssl enc -aes-128-cbc -e -in plain.txt -out aes128_cbc.bin \
  -K 00112233445566778899aabbccddeeff \
  -iv 112233445566778899aabbccddeeff00

openssl enc -aes-128-cbc -d -in aes128_cbc.bin -out decrypted_cbc.txt \
  -K 00112233445566778899aabbccddeeff \
  -iv 112233445566778899aabbccddeeff00
```

AES-128-ECB

```
openssl enc -aes-128-ecb -e -in plain.txt -out aes128_ecb.bin \
  -K 00112233445566778899aabbccddeeff
```

AES-128-CFB

```
openssl enc -aes-128-cfb -e -in plain.txt -out aes128_cfb.bin \
  -K 00112233445566778899aabbccddeeff \
  -iv 112233445566778899aabbccddeeff00
```

Observations

- Decryption succeeded for all modes; output matched the original plaintext.

- CBC and CFB required IVs; ECB ignored IV.
- Stream-like modes behave differently in terms of padding and internal processing.

Task 2 — ECB vs CBC Encryption on an Image

Goal

Encrypt a BMP image in ECB and CBC mode, patch the header, and observe visual differences.

Procedure

1. Encrypt the file using AES-128-ECB and AES-128-CBC.
2. Open the encrypted binaries using GHex.
3. Copy the first 54 bytes (BMP header) from the original file into each encrypted output.
4. Save modified files as viewable BMP images.

Observations

- **ECB Image:** Structural patterns from the original were still visible because ECB encrypts each block independently. Repeated blocks in the plaintext produce repeated blocks in ciphertext.
- **CBC Image:** Appeared as random noise. No recognizable shapes. The chaining mechanism hides repeated patterns.

Conclusion

CBC provides better protection of visual structure than ECB.

Task 3 — Corrupted Ciphertext Behavior

Goal

Modify a single byte in ciphertext and observe how corruption spreads across different AES modes.

Steps

1. Create a long text file (`long_plain.txt`) with >64 bytes.
2. Encrypt using modes ECB, CBC, CFB, OFB.
3. Open ciphertext in GHex.
4. Flip one bit at byte index 29 and save as a corrupted file.
5. Decrypt using same key/IV.

Results

- **ECB:** Only the block containing the corrupted byte was affected. Others decrypted normally.
- **CBC:** The corrupted block was completely garbled, and the corresponding byte in the next block flipped. Remaining blocks unaffected.
- **CFB:** Only a limited region was corrupted. Resembles stream cipher behavior.
- **OFB:** Only the specific byte position was corrupted; no propagation occurred.

Task 4 — Padding Requirements

Goal

Determine which AES modes require padding.

Experiment

Encrypted a short text file (<16 bytes) with CBC, ECB, CFB, and OFB.

Findings

- **Modes requiring padding:**
  - **ECB**, **CBC** (because they operate in fixed 16-byte blocks)
- **Modes not requiring padding:**
  - **CFB**, **OFB** (work byte-by-byte)

OpenSSL automatically added padding for CBC and ECB.

Task 5 — Generating Message Digests

Goal

Generate digests of a file using MD5, SHA-1, and SHA-256.

Commands

```
openssl dgst -md5 file.txt
openssl dgst -sha1 file.txt
openssl dgst -sha256 file.txt
```

Observations

- Each algorithm produced a different fixed-length hash.
- SHA-256's 256-bit digest is longest and provides the strongest security.
- Even 1-bit change in the input caused drastically different hash outputs.

Task 6 — Keyed Hashing (HMAC)

Goal

Generate HMAC values for a file using different keys and algorithms.

Commands

openssl dgst -sha256 -hmac "mykey123" file.txt
openssl dgst -sha1 -hmac "mykey123" file.txt
openssl dgst -md5   -hmac "mykey123" file.txt

Observations

- HMAC outputs changed whenever the key changed.
- HMAC accepts keys of any length; long keys are internally hashed or padded.
- SHA-256 based HMAC produced the longest and strongest output

Task 7 — Avalanche Effect Test

Goal

The objective of this task is to demonstrate the **Avalanche Effect**, a fundamental property of cryptographic hash functions. The Avalanche Effect states that even if a single bit in the input changes, more than 50% of the output hash bits will change, making the new hash appear completely unrelated to the original.

Step 1 — Creating the Input File and Generating H_1

A. Creating the Input File

A text file named `input_avalanche.txt` was created using the following command:

```
gedit input_avalanche.txt
```

A few lines of text (3–4 lines) were manually added and the file was saved.

B. Generating the First Hash Value (H_1)

The initial hash values were generated using MD5 and SHA256:

```
# MD5 H1

openssl dgst -md5 input_avalanche.txt


# SHA256 H1

openssl dgst -sha256 input_avalanche.txt
```

These two outputs were recorded as **H_1(MD5)** and **H_1(SHA256)** in the report.

Step 2 — Modifying One Bit and Generating H_2

A. Flipping One Bit Using GHex

The file was opened in the hex editor:

```
ghex input_avalanche.txt &
```

Inside GHex, one byte was modified to change exactly one bit.
For example:
A byte with value **3A** was changed to **3B**, which differs by only 1 bit.

The file was then saved.

B. Generating the Second Hash Value (H_2)

```
# MD5 H2

openssl dgst -md5 input_avalanche.txt
```

```
# SHA256 H2

openssl dgst -sha256 input_avalanche.txt
```

These outputs were recorded as **H_2(MD5)** and **H_2(SHA256)**.

Step 3 — Observation

Difference Between H_1 and H_2

A clear and significant difference was observed:

- The H_1 and H_2 hash values looked completely different.
- Even though only a single bit in the input was changed, both MD5 and SHA256 produced totally different output hashes.
- It appears as if the hashes came from two entirely different files.

Reason — Avalanche Effect

This behavior confirms the **Avalanche Effect**:

- A minor change (even a 1-bit modification) in the input results in a drastically different hash value.
- For secure hash functions, **more than 50% of the output bits typically change**.
- This property ensures that hash functions are highly sensitive to input changes and prevents attackers from predicting output patterns.
