Lab Report: Symmetric Encryption & Hashing

Task 1: AES Encryption Using Different Modes

Commands Used:

```
# Generate random key and IV
KEY=$(openssl rand -hex 16)
IV=$(openssl rand -hex 16)

# Encryption commands
openssl enc -aes-128-cbc -e -k $KEY -iv $IV -in plaintext.txt -out ciphertext-cbc.bin -nosalt
openssl enc -aes-128-cfb -e -k $KEY -iv $IV -in plaintext.txt -out ciphertext-cfb.bin -nosalt
openssl enc -aes-128-ofb -e -k $KEY -iv $IV -in plaintext.txt -out ciphertext-ofb.bin -nosalt

# Decryption commands (for verification)
openssl enc -aes-128-cbc -d -k $KEY -iv $IV -in ciphertext-cbc.bin -out decryptedtext-cbc.txt -nosalt
openssl enc -aes-128-cfb -d -k $KEY -iv $IV -in ciphertext-cfb.bin -out decryptedtext-cfb.txt -nosalt
openssl enc -aes-128-ofb -d -k $KEY -iv $IV -in ciphertext-ofb.bin -out decryptedtext-ofb.txt -nosalt
```

Observations:

- All three modes (CBC, CFB, OFB) successfully encrypted and decrypted the plaintext file
- The decrypted files matched the original plaintext, confirming proper encryption/decryption

Task 2: Encryption Mode - ECB vs CBC

Commands Used:

```
openssl enc -aes-128-ecb -e -k $KEY -in original.bmp -out encoded-ecb.bmp
openssl enc -aes-128-cbc -e -k $KEY -iv $IV -in original.bmp -out encoded-cbc.bmp
```

Observations:

ECB Mode:

- **Information Leaked:** Significant information visible
- **Explanation:** Identical plaintext blocks produce identical ciphertext blocks, preserving patterns from the original image
- **Visual Result:** Noisy but recognizable version of the original picture

CBC Mode:

- **Information Leaked:** No useful information visible
- **Explanation:** Chaining mechanism destroys all patterns and structures
- **Visual Result:** Appears as random, uniform noise

Conclusion:

ECB mode is vulnerable to pattern analysis and should not be used for encrypting structured data like images, while CBC provides better security by destroying patterns.

Task 3: Encryption Mode – Corrupted Cipher Text

Commands Used:

```
# Encryption
openssl enc -aes-128-ecb -e -k $KEY -iv $IV -in plaintext.txt -out encrypted-ecb.bin -nosalt
openssl enc -aes-128-cbc -e -k $KEY -iv $IV -in plaintext.txt -out encrypted-cbc.bin -nosalt
openssl enc -aes-128-cfb -e -k $KEY -iv $IV -in plaintext.txt -out encrypted-cfb.bin -nosalt
openssl enc -aes-128-ofb -e -k $KEY -iv $IV -in plaintext.txt -out encrypted-ofb.bin -nosalt

# Decryption of corrupted files
openssl enc -aes-128-ecb -d -k $KEY -iv $IV -in encrypted-ecb.bin -out decrypted-corrupted-ecb.txt -nosalt
openssl enc -aes-128-cbc -d -k $KEY -iv $IV -in encrypted-cbc.bin -out decrypted-corrupted-cbc.txt -nosalt
openssl enc -aes-128-cfb -d -k $KEY -iv $IV -in encrypted-cfb.bin -out decrypted-corrupted-cfb.txt -nosalt
openssl enc -aes-128-ofb -d -k $KEY -iv $IV -in encrypted-ofb.bin -out decrypted-corrupted-ofb.txt -nosalt
```

Results Analysis:

Mode	Error Propagation	Recovery Capability
ECB	Only the corrupted block affected	All blocks except one recovered
CBC	Corrupted block + next block affected	Two blocks lost
CFB	Limited propagation to subsequent bytes	Partial recovery with some errors
OFB	Only the corrupted byte affected	Almost full recovery

Implications:

- **ECB:** Limited error propagation but vulnerable to pattern analysis
- **CBC:** Error affects current and next block - suitable for data integrity
- **CFB/OFB:** Stream cipher modes with minimal error propagation - good for real-time data

Task 4: Padding Analysis

Experiment Setup:

- Created file: `test_23bytes.txt` (23 bytes)
- Tested with ECB, CBC, CFB, and OFB modes

Results:

Mode	Padding Required	Final Size	Reason
ECB	✅ Yes	32 bytes	Block cipher requires fixed-size blocks
CBC	✅ Yes	32 bytes	Block cipher requires fixed-size blocks
CFB	❌ No	23 bytes	Stream cipher mode encrypts bit-by-bit
OFB	❌ No	23 bytes	Stream cipher mode encrypts bit-by-bit

Explanation:

- **ECB & CBC:** Require padding to reach block boundary (16 bytes for AES)
- **CFB & OFB:** No padding needed as they operate as stream ciphers

Task 5: Generating Message Digest

Commands Used:

```
openssl dgst -md5 text.txt
# Output: MD5(text.txt)= ac6c19070852feb1485dbc3f5c1f3c84

openssl dgst -sha1 text.txt
# Output: SHA1(text.txt)= ff6c91b49f42f670560e1b16f4c44bd961805cb2

openssl dgst -sha256 text.txt
# Output: SHA2-256(text.txt)= 7f7d513c09734b18bfa55262d4d2a00d98630d2a962c44e5a8e4edd7c4789c0b
```

Observations:

- All hash algorithms produced fixed-length outputs regardless of input size
- SHA-256 produces the longest hash (256 bits), followed by SHA-1 (160 bits), then MD5 (128 bits)
- Each hash is unique to the specific input file

Task 6: Keyed Hash and HMAC

Commands and Outputs:

```
# With key "abcdefg"
openssl dgst -md5 -hmac "abcdefg" text.txt
# HMAC-MD5(text.txt)= a39a360d01a4fc77c5c7f27d43944b9e

openssl dgst -sha1 -hmac "abcdefg" text.txt
# HMAC-SHA1(text.txt)= dd9d62dd5ba3d4e9236e491e67f2554c48f607af

openssl dgst -sha256 -hmac "abcdefg" text.txt
# HMAC-SHA2-256(text.txt)= 04d27d006acea1c4f3cf2cfc593cb31e432a4d3113859b806d215a5232a8b532

# With key "abcd"
openssl dgst -md5 -hmac "abcd" text.txt
# HMAC-MD5(text.txt)= b5c985f46b65002253ea72dab88e4d77

openssl dgst -sha1 -hmac "abcd" text.txt
# HMAC-SHA1(text.txt)= 29547096d848267e204622e65c9be689e86936ce

openssl dgst -sha256 -hmac "abcd" text.txt
# HMAC-SHA2-256(text.txt)= 2652b6f2a54e227aa3ccfb37086e55f3764a5fe450d029ae83a6aaf53e253519
```

Key Size Analysis:

Question: Do we have to use a key with a fixed size in HMAC?

Answer: No, HMAC does not require a fixed-size key.

Explanation: The HMAC algorithm handles keys of any length internally:

- Short keys are padded with zeros
- Long keys are hashed to reduce length
- This normalization ensures flexibility in key choice

Task 7: Keyed Hash and HMAC - Hash Properties

Commands and Outputs:

```
# Original file hashes
```

```
openssl dgst -sha256 -hmac "abcd" text.txt
# HMAC-SHA2-256(text.txt)= 2652b6f2a54e227aa3ccfb37086e55f3764a5fe450d029ae83a6aaf53e253519

openssl dgst -md5 -hmac "abcd" text.txt
# HMAC-MD5(text.txt)= b5c985f46b65002253ea72dab88e4d77

# Modified file hashes (one bit flipped)
openssl dgst -sha256 -hmac "abcd" modified.txt
# HMAC-SHA2-256(modified.txt)= fb5a1da7e99204c8707a1a07f71246a296fbb178dbd76aecc5a559375ffdd612

openssl dgst -md5 -hmac "abcd" modified.txt
# HMAC-MD5(modified.txt)= 615d04b0e7140293852163f549f4d2ea
```

Observations:

- **Avalanche Effect:** Both MD5 and SHA-256 exhibit strong avalanche effect
- **Hash Difference:** A single bit change in input produces completely different hash outputs
- **No Similarity:** H_1 and H_2 show no visible similarity, demonstrating the one-way property of hash functions