Lab 3 Report: Symmetric Encryption and Hash Functions

Task 1: AES Encryption with Various Modes (2 Marks)

Objective:
Encrypt a plaintext file using AES in three distinct modes (CBC, ECB, CFB) and confirm the encryption integrity by performing decryption on the resulting files.

Commands Used:

CBC Encryption:
`openssl enc -aes-128-cbc -e -in plain.txt -out cipher_cbc.bin -K 1a2b3c4d5e6f708192a3b4c5d6e7f80 -iv 1b2c3d4e5f607182`

ECB Encryption:
`openssl enc -aes-128-ecb -e -in input.txt -out encrypted-ecb.dat -K 1a2b3c4d5e6f708192a3b4c5d6e7f80`

CFB Encryption:
`openssl enc -aes-128-cfb -e -in input.txt -out encrypted-cfb.dat -K 1a2b3c4d5e6f708192a3b4c5d6e7f80 -iv 1b2c3d4e5f607182`

Verification:
All output files were decrypted successfully with the matching decryption commands, validating the accuracy of the encryption procedure.

Decryption Commands:

CBC Decryption
`openssl enc -aes-128-cbc -d -in encrypted-cbc.dat -out recovered-cbc.txt -K 1a2b3c4d5e6f708192a3b4c5d6e7f80 -iv 1b2c3d4e5f607182`

ECB Decryption
`openssl enc -aes-128-ecb -d -in encrypted-ecb.dat -out recovered-ecb.txt -K 1a2b3c4d5e6f708192a3b4c5d6e7f80`

CFB Decryption
`openssl enc -aes-128-cfb -d -in encrypted-cfb.dat -out recovered-cfb.txt -K 1a2b3c4d5e6f708192a3b4c5d6e7f80 -iv 1b2c3d4e5f607182`

Output Files
- `encrypted-cbc.dat` - CBC encrypted output
- `encrypted-ecb.dat` - ECB encrypted output
- `encrypted-cfb.dat` - CFB encrypted output

Task 2: Encryption Modes - ECB vs CBC (3 Marks)

Objective:
Apply ECB and CBC modes to encrypt a BMP image, restore the file header for visibility, and evaluate the associated security risks.

Implementation:

Encryption:
ECB:
openssl enc -aes-128-ecb -e -in panda.bmp -out panda_ecb.encrypt -K 00112233445566778899aabbccddeeff
CBC:
openssl enc -aes-128-cbc -e -in panda.bmp -out panda_cbc.enc -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

Header Replacement:
the first 54 bytes (BMP header) from the original image
were copied to the encrypted files to create viewable BMP files:
Ecb_m - ECB encrypted image with original header
Cbc_m - CBC encrypted image with original header

Observations:

Panda.bmp

ECB Mode:

- The output displays discernible patterns from the source material.
- Matching plaintext segments yield matching ciphertext segments.
- This highlights ECB's limitation: inadequate concealment of data structures.
- Visible artifacts in the encrypted view expose details of the original layout.

CBC Mode:

- The result looks entirely noisy, lacking any identifiable patterns.
- Block processing relies on prior ciphertext via IV and linking mechanism.
- No structural clues from the source are extractable.
- CBC excels in obscuring plaintext characteristics.

Conclusion:

ECB is prone to structural leakage attacks as duplicate plaintext yields duplicate ciphertext. In contrast, CBC enhances protection through inter-block dependencies, ideal for visual data like images where pattern exposure poses risks.

Output Files

- `img-ecb.dat` - ECB encrypted raw
- `img-ecb.bmp` - ECB viewable image
- `img-cbc.dat` - CBC encrypted raw
- `img-cbc.bmp` - CBC viewable image

Task 3: Encryption Modes – Altered Ciphertext (3 Marks)

Objective

Examine error diffusion in encryption modes by altering one bit in the ciphertext and reviewing decryption outcomes.

Implementation

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

Corruption:

a single bit in the 30th byte of each encrypted file was flipped.

Decryption:

ECB:

openssl enc -aes-128-ecb -d -in corrupt_ecb.bin -out d_corrupt_ecb.txt -K 00112233445566778899aabbccddeeff

CBC:

openssl enc -aes-128-cbc -d -in corrupt_cbc.bin -out d_corrupt_cbc.txt -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

CFB:

openssl enc -aes-128-cfb -d -in corrupt_cfb.bin -out d_corrupt_cfb.txt -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

OFB:

openssl enc -aes-128-ofb -d -in corrupt_ofb.bin -out d_corrupt_ofb.txt -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

Analysis

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.

Reasons of differences:

1. **Error tolerance and data transmission**
    a. Modes like OFB and CFB are better suited for streaming or communication channels where small bit errors might occur.

    b. ECB and CBC are not good for noisy channels since bit errors can ruin entire blocks.
2. **Security implications**
    a. ECB leaks structure. It's deterministic and should never be used for sensitive data like images.
    b. CBC, CFB, and OFB hide patterns much better.
3. **Performance and parallelism**
    a. ECB can be parallelized easily; others (especially CBC encryption) are sequential because of block dependencies.
    b. OFB and CFB behave like stream ciphers — useful for continuous data streams.

Task 4: Padding Requirements (3 Marks)

Objective:
Identify padding needs across encryption modes and justify the differences.

Implementation:
The plaintext file was processed in ECB, CBC, CFB, and OFB modes:

Encryption Commands:
ECB
`openssl enc -aes-128-ecb -e -in plain.txt -out out-ecb.bin \ -K a1b2c 3d4e5f60718293a4b5c6d7e8f90`

CBC
`openssl enc -aes-128-cbc -e -in plain.txt -out out-cbc.bin \ -K a1b2 c3d4e5f60718293a4b5c6d7e8f90 \ -iv 0a1b2c3d4e5f6071`

CFB
`openssl enc -aes-128-cfb -e -in plain.txt -out out-cfb.bin \ -K a1b2c 3d4e5f60718293a4b5c6d7e8f90 \ -iv 0a1b2c3d4e5f6071

OFB
openssl enc -aes-128-ofb -e -in plain.txt -out out-ofb.bin \ -K a1b2c 3d4e5f60718293a4b5c6d7e8f90 \ -iv 0a1b2c3d4e5f6071`

Observations:

Modes Needing Padding:
ECB: Mandatory (block mode).
CBC: Mandatory (block mode).

Modes Without Padding:
CFB: Optional (stream mode).
OFB: Optional (stream mode).

Explanation:

ECB and CBC Padding Necessity:
- Operate on uniform blocks (16 bytes in AES-128).
- Input requires block-size multiples.
- Non-multiples trigger padding addition (PKCS#7 default in OpenSSL).

CFB and OFB No-Padding Rationale:
- Transform blocks into streams via keystream production.
- Accommodate arbitrary lengths seamlessly.
- Process incrementally, bypassing block constraints.

Output Files:
- `enc-ecb.dat` - ECB with padding
- `enc-cbc.dat` - CBC with padding
- `enc-cfb.dat` - CFB unpadded
- `enc-ofb.dat` - OFB unpadded

Task 5: Message Digest Generation (3 Marks)

Objective:
Produce digests via multiple hash methods and assess their results.

Implementation:
Plaintext file content:
Message for digest purpose

Commands Used:

```
# MD5openssl dgst -md5 plain.txt > md5.txt
# SHA1openssl dgst -sha1 plain.txt > sha1.txt
# SHA256openssl dgst -sha256 plain.txt > sha256.txt
# SHA512openssl dgst -sha512 plain.txt > sha512.txt
```

Generated Digests:

MD5:
`MD5(input.txt)= e777ebb98c59af9fd4142685c597d154`
- Length: 128 bits (32 hex).
- Quick but insecure (collision risks).

SHA1:
`SHA1(input.txt)= 87ec3acf9ba6542b6447fd8085ac10119491816b`
- Length: 160 bits (40 hex).

- Phased out for vulnerabilities.

SHA256:
`SHA2-256(input.txt)= 939106512d73323c0164c0b3761441c10fcaa3020ffc3
692b6fe68dcf67b1221
- Length: 256 bits (64 hex).
- Secure and standard.

SHA512:
`SHA2-512(input.txt)= 76450fe383aa94624ea48dee3a2f355b247002075f4
9fd6c2178bb5580dbe49bdf97764099efb8ae6ee1bbc8e8d20348894b4b5
b7d504555c67e72a79a8d952
`
- Length: 512 bits (128 hex).
- Maximal strength for critical use.

Observations:
1. Fixed Output: Uniform size irrespective of input.
2. Cascade Impact: Minor input tweaks yield major hash shifts.
3. Consistency: Identical inputs match hashes.
4. Irreversibility: Outputs defy input reconstruction.
5. Viability: SHA256/SHA512 robust; MD5/SHA1 obsolete.

Output Files
- `md5.txt` - MD5 result
- `sha1.txt` - SHA1 result
- `sha256.txt` - SHA256 result
- `sha512.txt` - SHA512 result

Task 6: Keyed Hashing and HMAC (3 Marks)

Objective:
Compute HMAC instances with varied hash types and key sizes to grasp sizing flexibility.

Implementation:
Plaintext:
`Message for digest purpose`

Commands Used:
A bash script tested HMAC across key variants:

```
#!/usr/bin/env bash
# hmac.sh — Generate HMAC digests with various key lengths

KEYS=(
"a" # 1 byte key
"abcdefg" # 7 byte key
```

```
"0123456789abcdef" # 16 byte key
"This is a longer key for HMAC testing with more than 64 bytes to test ke
y length handling" # > 64 bytes
)

# HMAC-MD5
echo "=== HMAC-MD5 ==="
for i in "${!KEYS[@]}"; do
KEY="${KEYS[$i]}"
KEY_LEN=${#KEY}
echo "Key length: $KEY_LEN bytes"
echo "Key: $KEY"
openssl dgst -md5 -hmac "$KEY" plain.txt
echo ""
done

# HMAC-SHA1
echo "=== HMAC-SHA1 ==="
for i in "${!KEYS[@]}"; do
KEY="${KEYS[$i]}"
KEY_LEN=${#KEY}
echo "Key length: $KEY_LEN bytes"
echo "Key: $KEY"
openssl dgst -sha1 -hmac "$KEY" plain.txt
echo ""
done
# HMAC-SHA256
echo "=== HMAC-SHA256 ==="
for i in "${!KEYS[@]}"; do
KEY="${KEYS[$i]}"
KEY_LEN=${#KEY}
echo "Key length: $KEY_LEN bytes"
echo "Key: $KEY"
openssl dgst -sha256 -hmac "$KEY" plain.txt
echo ""
done
```

HMAC Outputs

HMAC-MD5:

Key: “a” (1 byte) → **fdef49ee857fd2a6408b394faaf9d761**
Key: “abcdefg” (7 bytes) →**16f0cfb70d107a98165878d892f30580**
Key: “0123456789abcdef” (16 bytes) →**a71d32856507edffe52c09b842ee912d**
Key: “This is a longer key…” (89 bytes) →**b9f2eed319b6a10701fd9db515d14a12**

HMAC-SHA1:
Key: “a” (1 byte) → **fba55873d89bdaedfef135f0db2ba6192229fbe3**
Key: “abcdefg” (7 bytes) → **398e52807937514239874edce6b1315df732a7fe**
Key: “0123456789abcdef” (16 bytes) → **1af5cf1babe3a3f0ee150db539f56622154fddca**
Key: “This is a longer key…” (89 bytes) → **dadd43f4492b8bfb81c2dcdb95868a9f9e4d50ee**
`

HMAC-SHA256:
Key: “a” (1 byte) → **9450cfe629e371c30f171d1b08e1df4176ce770da8eccf9eef23225f51edd73a**
Key: “abcdefg” (7 bytes) → **98eb0d1d223e721458eb908216e75e3b07ae721ab34c426e4bda322f628d3211**
Key: “0123456789abcdef” (16 bytes) →
c0f75f0d9e7b512abfd0c4119026bd99f0ed2e11c98d63176bdacc25ff28c070
Key: “This is a longer key…” (89 bytes) →
08f1cfebb9b987f2c35f68f5ab6db481f31c9c40176fcb25ce1b765606f8b4e4
`

Key Sizing Evaluation:
Fixed Key Size Mandatory?-> No. HMAC supports arbitrary lengths.

Details:
- Short keys (< block size, e.g., 64 bytes for MD5/SHA1/SHA256) get zero-padded.
- Long keys (> block) are hashed down to output size.
- Internal normalization ensures uniformity.

Advice: Favor extended keys for strength; match hash output (e.g., 32+ bytes for SHA256).

Task 7: Hash Function Characteristics (3 Marks)

Objective:
Illustrate the diffusion property by altering one input bit and contrasting hashes.

Implementation:

Original (`input.txt`):
Message for hash testing
Author: Yasin Bashar
`

Commands:

Generate hash for original fileopenssl dgst -md5 plain.txt > plain_md5.txt
openssl dgst -sha256 plain.txt > plain_sha256.txt
Generate hash for modified file (one bit flipped)openssl dgst -md5 modifi
ed.txt > modified_md5.txt

openssl dgst -sha256 modified.txt > modified_sha256.txt

Hash Outputs:

MD5:
Original: MD5(plain.txt)= e635e11d8d684e7f1a3db2e349a873a4
Altered: MD5(modified.txt)= 54ed6cd13b13b327784b183bc42440a0

SHA256:
Original: SHA2-256(plain.txt)= 47c7c246016a6a12c4218ac150bb71c4eb177bf303e6
c30563b61a54bb56dac2
`
Altered:SHA2-256(modified.txt)= ec0fc6cf8ead84f0e38df808ac17dd47638e1e182
6ddf52d8ecc6aad3ee8478c
`

Observations:

1. Total Overhaul: Single-bit change yields unrelated hashes.
2. Diffusion: ~50% bit variance expected.
3. Unrelatedness: Outputs show no overlap, affirming irreversibility.
4. Predictability: Unchanged inputs yield identical results.

Bonus: Bit Diff Analysis Script (2 Marks)

Python tool for bit matching:

```
def hex_to_binary(hex_string):
return bin(int(hex_string, 16))[2:].zfill(len(hex_string) * 4)
def count_same_bits(hex1, hex2):
if len(hex1) != len(hex2):
raise ValueError("Both hash values must have the same length.")
bin1 = hex_to_binary(hex1)
bin2 = hex_to_binary(hex2)
same_bits = sum(1 for a, b in zip(bin1, bin2) if a == b)
total_bits = len(bin1)
return same_bits, total_bits
```

Results:
MD5: 62 / 128 matches (~48.44%)
SHA256: 123 / 256 matches (~48.05%)

Evaluation:

~50% divergence validates diffusion; minor deviations are algorithmic norms. Ensures tampering detection.

Output Files:

plain_md5.txt - MD5 hash of original file

Plain_sha256.txt - SHA256 hash of original file

Modified_md5.txt - MD5 hash of modified file

Modified_sha256.txt - SHA256 hash of modified file

compare.py - Python program for bit comparison