Table of Contents

1. Objective
2. Tools and Environment
3. Tasks & Procedures
    - Task 1: AES Encryption Using Different Modes
    - Task 2: Image Encryption — ECB vs CBC
    - Task 3: Corrupted Ciphertext and Error Propagation
    - Task 4: Padding Behavior
    - Task 5: Message Digests (MD5 / SHA1 / SHA256)
    - Task 6: HMAC (Keyed Hash)
    - Task 7: Hash Avalanche & Bit-similarity Checker (Bonus)
4. Results (Outputs & Observations)
5. Analysis / Discussion
6. Conclusion
7. References
8. Appendices (Commands, Scripts, Files)

---

1. Objective

Perform experiments with symmetric encryption modes and hashing to observe:

- Differences between ECB, CBC, CFB, OFB modes
- Error propagation when ciphertext is corrupted
- Padding requirements for block vs stream modes
- Properties of cryptographic hashes and HMACs
- Avalanche property and bit-similarity measurement

---

2. Tools and Environment

- Platform: Linux / Windows (WSL recommended)
- Software: OpenSSL, GHex (or any hex editor), Python 3.10+
- Commands tested with OpenSSL 1.1.x / 3.x
- Files included in the lab folder:
    - plain.txt — sample plaintext
    - pic_original.bmp — sample test image
    - cipher_* and hash_* files — outputs produced during experiments
    - same_bits_checker.py — Python bonus script

---

3. Tasks & Procedures

Task 1 — AES Encryption Using Different Modes

Goal: Encrypt and decrypt a small plaintext using AES-128 in CBC and CFB modes and Blowfish-CBC.

Plaintext creation

echo "This is a test message for encryption and decryption" > plain.txt

Encryption commands

AES-128-CBC

openssl enc -aes-128-cbc -e -in plain.txt -out cipher_aes128cbc.bin \

-K 00112233445566778889aabbccddeeff -iv 0102030405060708

AES-128-CFB

openssl enc -aes-128-cfb -e -in plain.txt -out cipher_aes128cfb.bin \

-K 00112233445566778889aabbccddeeff -iv 0102030405060708

Blowfish-CBC

openssl enc -bf-cbc -e -in plain.txt -out cipher_bfcbc.bin \

-K 00112233445566778889aabbccddeeff -iv 0102030405060708

Decryption / Verification

openssl enc -aes-128-cbc -d -in cipher_aes128cbc.bin -out decrypted_cbc.txt \

-K 00112233445566778889aabbccddeeff -iv 0102030405060708

Compare

cat decrypted_cbc.txt

diff plain.txt decrypted_cbc.txt || echo "Files differ"

Expected result: decrypted_cbc.txt matches plain.txt.

Task 2 — Encryption Mode: ECB vs CBC (Image)

Goal: Show visual differences when encrypting an image with ECB and CBC.

Steps

1. Encrypt the BMP with AES-128-ECB and AES-128-CBC (no salt)
2. Copy the first 54 bytes (BMP header) from the original image to each encrypted file (so viewer recognizes format)
3. Open images with an image viewer

Commands

ECB

openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb_encrypted.bmp \

-K 00112233445566778889aabbccddeeff -nosalt

CBC

openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc_encrypted.bmp \

-K 00112233445566778889aabbccddeeff -iv 0102030405060708 -nosalt

Header replacement (example using dd on Linux)

Save headers

dd if=pic_original.bmp of=header.bin bs=1 count=54

Replace header in encrypted versions

dd if=header.bin of=pic_ecb_encrypted.bmp bs=1 count=54 conv=notrunc

dd if=header.bin of=pic_cbc_encrypted.bmp bs=1 count=54 conv=notrunc

Observation

- pic_ecb.bmp: original visual patterns still visible (not secure).
- pic_cbc.bmp: appears random/no recognizable pattern (more secure visually).

Insert screenshots:

- images/pic_original.png
- images/pic_ecb.png
- images/pic_cbc.png

Task 3 — Corrupted Cipher Text (Error Propagation)

Goal: Flip one bit in ciphertext and decrypt to observe error propagation depending on mode.

Procedure

1. Encrypt with AES-128-CBC:

openssl enc -aes-128-cbc -e -in plain.txt -out corrupted_cipher.bin \

-K 00112233445566778889aabbccddeeff -iv 0102030405060708

2. Open corrupted_cipher.bin with GHex or hex editor and flip a single bit or a single byte (e.g., change byte 30).
3. Decrypt the corrupted ciphertext:

openssl enc -aes-128-cbc -d -in corrupted_cipher.bin -out recovered.txt \

-K 00112233445566778889aabbccddeeff -iv 0102030405060708

Expected observations

Mode	Effect of 1-bit/1-byte corruption
ECB	Only the block containing the bit gets corrupted on decryption
CBC	The current block is corrupted and the next block is garbled
CFB	Only one byte (or a limited number) is corrupted; stream-like behavior
OFB	Only the flipped bit is corrupted (bit-flip affects only corresponding bit)

Task 4 — Padding

Goal: Demonstrate padding for ECB/CBC and no padding needed for stream modes (CFB/OFB).

Procedure

1. Create a plaintext of length not a multiple of AES block size (e.g., 37 bytes).
2. Encrypt with AES-128-ECB and AES-128-CFB.

Commands

```
# Create 37-byte file
printf 'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890abc' > plain37.txt
wc -c plain37.txt    # shows 37


openssl enc -aes-128-ecb -e -in plain37.txt -out pad_ecb.bin \
-K 00112233445566778889aabbccddeeff


openssl enc -aes-128-cfb -e -in plain37.txt -out pad_cfb.bin \
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Observation

- ECB/CBC will add PKCS#7 (or PKCS#5) padding to complete the block. Decryption removes padding.
- CFB/OFB behave like stream ciphers — no block padding is required.

Task 5 — Message Digest (MD5, SHA1, SHA256)

Goal: Generate digests and show avalanche effect.

Commands

openssl dgst -md5 plain.txt > hash_md5.txt

openssl dgst -sha1 plain.txt > hash_sha1.txt

openssl dgst -sha256 plain.txt > hash_sha256.txt

View

cat hash_md5.txt

cat hash_sha1.txt

cat hash_sha256.txt

Example output

MD5(plain.txt)= d41d8cd98f00b204e9800998ecf8427e

SHA1(plain.txt)= da39a3ee5e6b4b0d3255bfef95601890afd80709

SHA256(plain.txt)= e3b0c44298fc1c149afbf4c8996fb924...

Avalanche test

1. Make a one-bit modification to plain.txt → plain_mod.txt
2. Recompute hashes and compare — they should be drastically different.

Insert hash output screenshots: images/hash_outputs.png

Task 6 — Keyed Hash (HMAC)

Goal: Produce HMACs and discuss key handling.

Commands

openssl dgst -md5 -hmac "key123" plain.txt > hmac_md5.txt

openssl dgst -sha1 -hmac "key123" plain.txt > hmac_sha1.txt

openssl dgst -sha256 -hmac "key123" plain.txt > hmac_sha256.txt

Notes

- HMAC securely combines key and message using the underlying hash function.
- HMAC accepts arbitrary key lengths; internally keys shorter or longer than the block size are handled per the HMAC spec (padded or hashed).

Task 7 — Hash Avalanche & Bit Similarity Checker (Bonus)

Goal: Measure bit similarity between two hash values.

Python: same_bits_checker.py

```
def count_same_bits(h1, h2):
    b1 = bin(int(h1, 16))[2:].zfill(len(h1)*4)
    b2 = bin(int(h2, 16))[2:].zfill(len(h2)*4)
    return sum(i == j for i, j in zip(b1, b2))


if __name__ == "__main__":
    h1 = input("Enter H1: ").strip()
    h2 = input("Enter H2: ").strip()
    print("Same bits:", count_same_bits(h1, h2))
```

Procedure

1. Compute SHA-256 for original file and its one-bit modified version.
2. Measure number of equal bits using the script.
3. Expect about 50% of bits to match on average (avalanche property).

---

4. Results (Sample / Expected Outputs)

Paste your actual terminal outputs and screenshots from your run here. Example snippets:

Task 1 example output (decryption verification)

```
$ cat decrypted_cbc.txt
This is a test message for encryption and decryption
```

Task 2 image observation

- pic_ecb.bmp: pattern visible
- pic_cbc.bmp: pattern not visible

Task 3 corrupted decryption

- recovered.txt shows block-level corruption consistent with mode used.

Task 5 hash outputs

MD5(plain.txt)= 5f70b6... (example)

SHA1(plain.txt)= 2fd4e1... (example)

SHA256(plain.txt)= b94d27... (example)

Task 6 HMAC outputs

HMAC-MD5: 9e107d9d372bb6826bd81d3542a419d6

HMAC-SHA1: 2fd4e1c6...

Task 7 avalanche test

Same bits: 128 (out of 256)   # example roughly 50%

---

5. Analysis / Discussion

- **ECB vs CBC**: ECB encrypts each block independently → preserves identical plaintext blocks → visible patterns in images. CBC XORs each plaintext block with previous ciphertext → hides patterns.
- **Error propagation**: In CBC, a corrupted ciphertext block affects corresponding plaintext block and causes decryption error in next block due to XOR with wrong ciphertext; in ECB errors are localized.
- **Padding**: Block modes require padding; stream modes do not.
- **Hashing & HMAC**: Hashes provide integrity; HMAC adds key-based authenticity. HMAC is secure when used with a strong underlying hash and secret key.
- **Avalanche**: Small changes in input cause large, unpredictable changes in hash outputs.

---

6. Conclusion

This lab demonstrated practical differences between symmetric cipher modes and hashing:

- CBC offers confidentiality better than ECB in many contexts (e.g., image encryption).
- Stream modes provide lower error propagation; block modes require padding.
- Hash functions and HMACs provide integrity and authenticity primitives; HMACs handle keys of arbitrary length.
- Cryptographic hashes show expected avalanche behavior.

---

7. References

- “Advanced Encryption Standard (AES)” — NIST FIPS-197
- OpenSSL manual pages (man openssl, openssl enc)
- HMAC specification — RFC 2104
- Course lectures / laboratory manual

---

8. Appendices

Appendix A — Full commands used

(Repeat the commands in one block for reproducibility — copy the commands from each Task above.)

Appendix B — Scripts

Include the script contents as files:

- same_bits_checker.py (shown earlier)

Appendix C — Files produced

List files included with submission:

plain.txt

cipher_aes128cbc.bin

cipher_aes128cfb.bin

cipher_bfcbc.bin

pic_original.bmp

pic_ecb.bmp

pic_cbc.bmp

corrupted_cipher.bin

hash_md5.txt

hash_sha1.txt

hash_sha256.txt

hmac_md5.txt

hmac_sha1.txt

hmac_sha256.txt

same_bits_checker.py
