Lab 3 Report: Symmetric Encryption and Hash Functions

Task 1: AES Encryption with Various Modes

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.

Method:

Step 1: Created a text file named plain.txt with multiple lines of text content, then saved the file in the working directory

Step 2: Encryption using three different Modes

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 File

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

Task 2: Encryption Mode — ECB vs CBC

Objective: To compare ECB and CBC encryption modes by encrypting a BMP image and observing the visual differences in the encrypted outputs

Method:

Step 1 — Setup

1. Converted the original image to BMP format using Paint
2. Created a working directory at: `E:\lab3\task3`
3. Changed PowerShell directory to the Task3 folder:

PS C:\Users\LENOVO> cd E:\lab3\task3

Step 2 — ECB Mode Encryption

Encryption command (run from PowerShell in `E:\lab3\task3`):

openssl enc -aes-128-ecb -e -in birdbmp.bmp -out bird_ecb.enc -K 00112233445566778899aabbccddeeff

Header Replacement Process :

1. Open the original `birdbmp.bmp` file in HxD hex editor.
2. Select **Edit** → **Select block**.
3. Copy the first 54 bytes.
4. Open the encrypted file `bird_ecb.enc` in HxD.
5. Select the first 54 bytes of the encrypted file.
6. Paste the copied header from the original `birdbmp.bmp`.
7. Save the modified file as `birdview_ecb.bmp`.

Step 3 — CBC Mode Encryption

Encryption command (run from PowerShell in `E:\lab3\task3`):

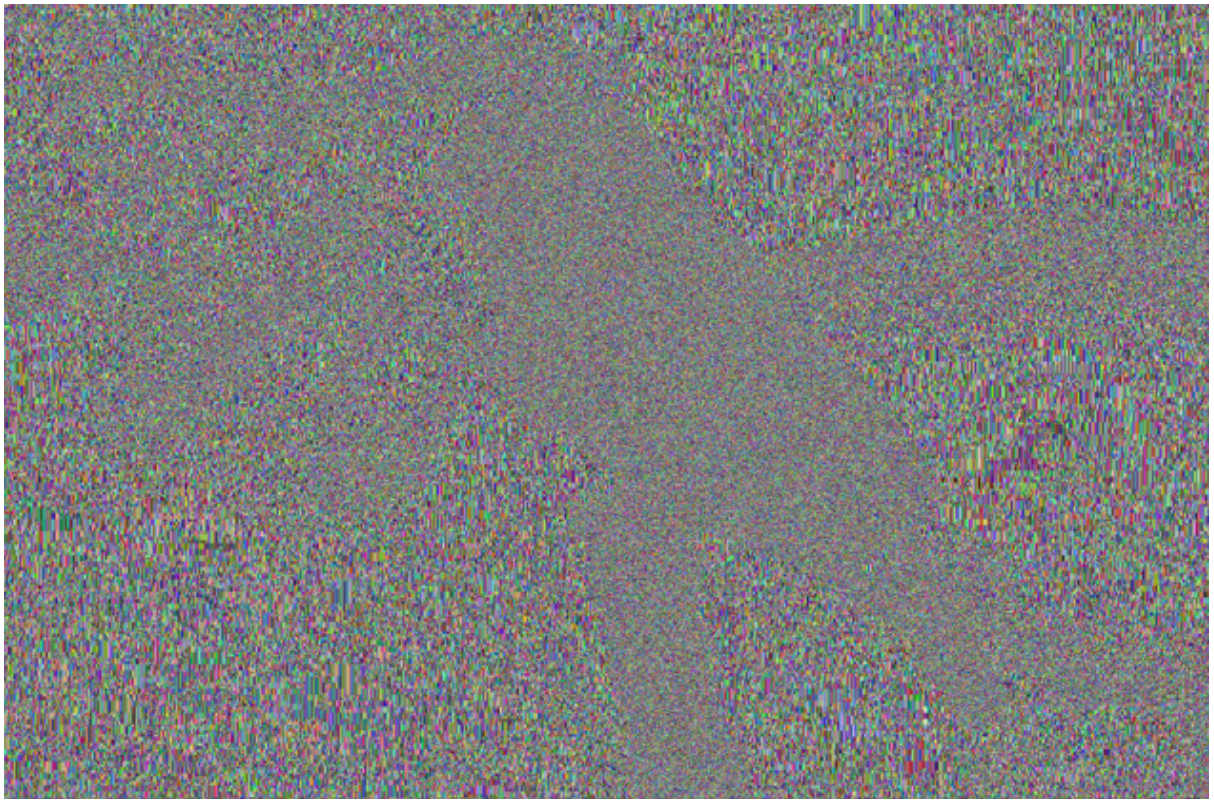
openssl enc -aes-128-cbc -e -in birdbmp.bmp -out bird_cbc.enc -K 00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10

Header Replacement Process :

1. Open the original `birdbmp.bmp` file in HxD hex editor.
2. Select and copy the first 54 bytes.
3. Open the encrypted file `bird_cbc.enc` in HxD.
4. Select the first 54 bytes of the encrypted file.
5. Paste the copied header from the original file.
6. Press **Ctrl+S** to save.
7. Save the file as `birdview_cbc.bmp`.

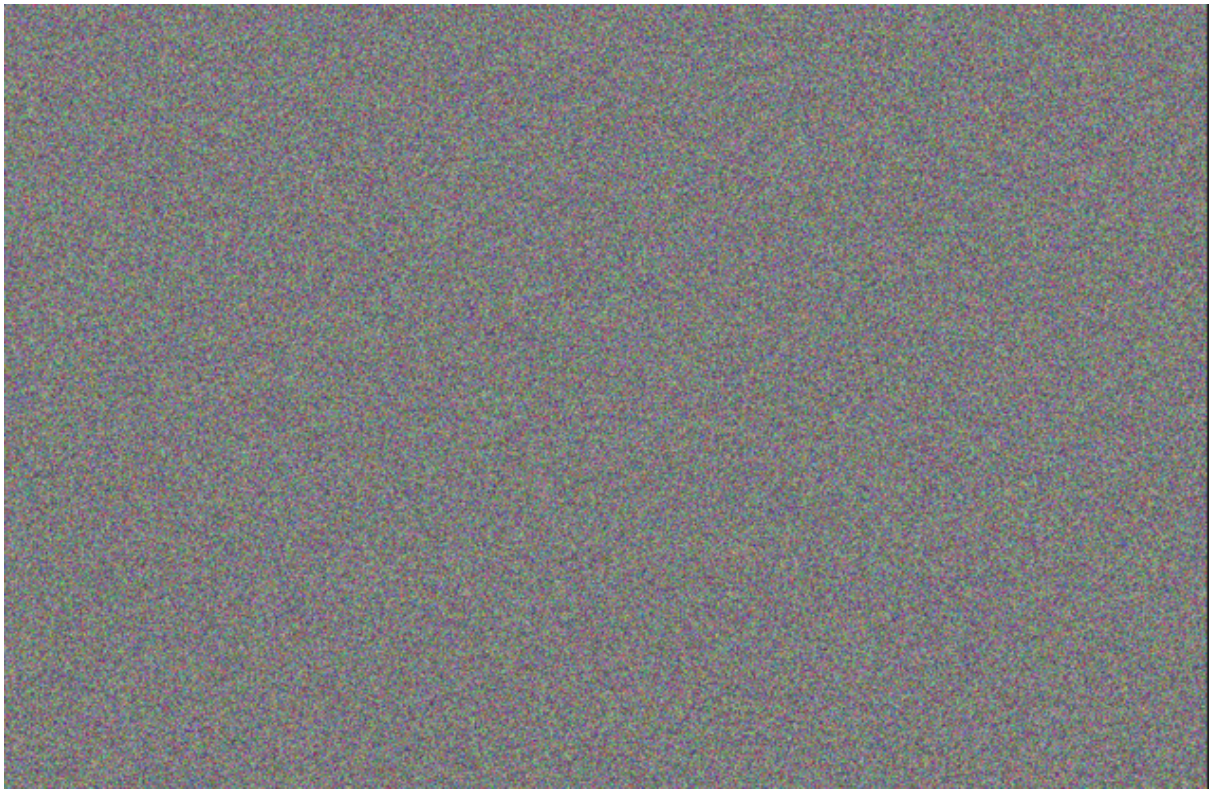
Results and Observations

ECB Mode Results (`birdview_ecb.bmp`)



1. The ECB-encrypted image reveals visible patterns from the original image.
2. Identical plaintext blocks produce identical ciphertext blocks.
3. The outline and structure of the original image remain visible (you can often see shapes, edges, or large color regions).

CBC Mode Results (`birdview_cbc.bmp`)



1. The CBC-encrypted image appears completely randomized.
2. No visible patterns or structure from the original image remain.
3. The image looks like random noise/static.
4. No useful visual information about the original image can be derived.

Analysis

Why ECB Mode is Insecure for Images

1. ECB encrypts each block independently without any chaining.
2. Identical plaintext blocks always produce identical ciphertext blocks.
3. Images often contain large areas of uniform or similar colors (repeating blocks).
4. These repetitive patterns are preserved in the encrypted output, revealing structural information.

5. An attacker can derive significant information about the image structure (outlines, shapes, repeated patterns).

Why CBC Mode is More Secure

1. CBC uses an Initialization Vector (IV) and chains blocks together.
2. Each plaintext block is XORed with the previous ciphertext block before encryption.
3. Identical plaintext blocks produce different ciphertext blocks because of chaining.
4. The chaining mechanism ensures that visual patterns are not preserved.
5. The encrypted output appears random, preventing leakage of image structure.

Security Implications

1. **ECB mode should never be used** for encrypting images or any data with visible or predictable patterns.
2. **CBC mode (with a random IV)** provides better confidentiality by hiding visual and structural patterns.
3. The choice of encryption mode significantly impacts security — chaining mechanisms (like CBC) are essential for semantic security.
4. Always use a secure IV (for CBC) and never reuse a key-IV pair across different messages.

Task 3: Encryption Modes – Altered Ciphertext

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

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

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. [Image comparing error diffusion in ECB, CBC, CFB, and OFB]

Reasons of Differences:

1. Error tolerance and data transmission

- Modes like **OFB** and **CFB** are better suited for streaming or communication channels where small bit errors might occur.
- **ECB** and **CBC** are not good for noisy channels since bit errors can ruin entire blocks.

2. Security implications

- **ECB leaks structure**. It's deterministic and should never be used for sensitive data like images.
- **CBC, CFB, and OFB hide patterns** much better.

3. Performance and parallelism

- **ECB** can be **parallelized easily**; others (especially CBC encryption) are sequential because of block dependencies.
- **OFB** and **CFB** behave like **stream ciphers** — useful for continuous data streams.

Task 4: Padding in AES Encryption Modes

Objective

The objective of this experiment is to observe how padding is applied in different AES encryption modes when the plaintext length is not a multiple of the AES block size (16 bytes). We also examine ciphertext sizes and verify the presence or absence of padding using a hex editor.

Tools Used

- **OpenSSL for Windows**
- **PowerShell**
- **HxD Hex Editor**

Plaintext File

A plaintext file named **`plain.txt`** was created.
The size of the plaintext was checked as:

Plaintext Size = 170 bytes

AES uses a fixed block size of 16 bytes.

To determine padding:

170 mod 16 = 10

Padding needed = 16 - 10 = 6 bytes
Padding byte value = 06 (hex)

So 6 padding bytes, each equal to `0x06`, are expected to appear in padded encryption modes.

Key and IV Used

Parameter	Value
AES Key (128-bit)	`00112233445566778899aabbccddeeff`
IV (for CBC/CFB/OFB)	`0102030405060708090a0b0c0d0e0f10`

Encryption Commands Executed

openssl enc -aes-128-ecb -in plain.txt -out cipher_ecb.bin -K 00112233445566778899aabbccddeeff
openssl enc -aes-128-cbc -in plain.txt -out cipher_cbc.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
openssl enc -aes-128-cfb -in plain.txt -out cipher_cfb.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
openssl enc -aes-128-ofb -in plain.txt -out cipher_ofb.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10

Ciphertext File Sizes

File	Size (bytes)	Padding Present?	Explanation
`plain.txt`	170	—	Original plaintext
`cipher_ecb.bin`	**176**	**Yes**	ECB works on blocks → padding added
`cipher_cbc.bin`	**176**	**Yes**	CBC also works on blocks → padding added

`cipher_cfb.bin`	**170**	**No**	CFB operates like a stream → no padding
`cipher_ofb.bin`	**170**	**No**	OFB also operates like a stream → no padding

Conclusion so far:

ECB and CBC increase ciphertext size because padding is required.
CFB and OFB retain original size because no padding is required.

Padding Observation in HxD

The files **`cipher_ecb.bin`** and **`cipher_cbc.bin`** were opened in HxD.

At the end of both files, the following repeated byte pattern was observed:

06 06 06 06 06 06

This matches the expected PKCS#7 padding, since:

Padding length = 6 bytes → padding value = 0x06 repeated 6 times

The files `cipher_cfb.bin` and `cipher_ofb.bin` showed no repeating padding values. Their final bytes appeared random, confirming no padding was applied.

Decryption Verification

All ciphertexts were decrypted using OpenSSL.

Example decryption command:

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

All decrypted files were compared with the original plaintext and matched exactly, confirming correct padding and unpadding operation.

Final Conclusion

- AES-128-ECB and AES-128-CBC use PKCS#7 padding because they operate on fixed 16-byte blocks.
  When the plaintext size is not a multiple of 16, padding bytes are added.
  In our case, 170 → 176 bytes, with padding byte = `0x06`.

- AES-128-CFB and AES-128-OFB operate in stream mode, so no padding is required.
  Ciphertext size remains equal to plaintext size .

Therefore, padding is required only in ECB and CBC, but not in CFB and OFB.