CSE-478 Lab 3: Symmetric Encryption & Hashing

MD Mahafuzul Hasan Shifat

reg:2020831014

November 20, 2025

Task 1: AES Encryption Using Different Modes

Commands:

Bash

```
echo "Secret message for CSE478 Lab 3." > plain_task1.txt
# Cipher Block Chaining (CBC)
openssl enc -aes-128-cbc -e -in plain_task1.txt -out cipher_task1_cbc.bin \
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
# Electronic Codebook (ECB)
openssl enc -aes-128-ecb -e -in plain_task1.txt -out cipher_task1_ecb.bin \
-K 00112233445566778889aabbccddeeff
# Cipher Feedback (CFB)
openssl enc -aes-128-cfb -e -in plain_task1.txt -out cipher_task1_cfb.bin \
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Observation:

All three modes (**CBC**, **ECB**, **CFB**) successfully encrypted the file.

Task 2: ECB vs CBC Mode

Steps:

1. Created a BMP file: `convert -size 100x100 xc:black original.bmp`
2. Encrypted `original.bmp` with **ECB** and **CBC** modes.

3. Used GHex to copy the first **54 bytes (BMP header)** from `original.bmp` to the encrypted files.
4. Renamed `pic_ecb_encrypted.bin` to `pic_ecb.bmp` and `pic_cbc_encrypted.bin` to `pic_cbc.bmp`.

Observation:

ECB mode shows visible patterns in the resulting image; **CBC mode appears as random noise**. This strongly demonstrates **ECB's vulnerability** to pattern analysis and its lack of diffusion, making it unsuitable for large data encryption where patterns exist.

Task 3: Corrupted Cipher Text

Experiment:

Bash

```
python3 -c "print('A'*64)" > plain_task3.txt
# Encrypt with different modes (ECB, CBC, CFB, OFB)
# ...
# Corrupt 30th byte using GHex
# Decrypt corrupted files
```

Results (Impact of 1-byte Corruption):

- **ECB:** Only **one block** (16 bytes) was affected.
- **CBC: One block** was completely garbled, and a **one-byte error** occurred in the next block.
- **CFB/OFB:** Only **one byte** was corrupted.

Implication:

CFB and **OFB** (stream cipher modes) are more suitable for **error-prone transmission channels** because a localized error in the ciphertext only causes a localized error upon decryption.

Task 4: Padding

Test:

Bash

```
echo "23 bytes long text." > plain_task4.txt

# ECB test (fails without padding)
openssl enc -aes-128-ecb -e -in plain_task4.txt -out cipher_ecb.bin \
```

-K 00112233445566778889aabbccddeeff -nopad

CFB test (succeeds without padding)
openssl enc -aes-128-cfb -e -in plain_task4.txt -out cipher_cfb.bin \
-K 00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad

Result:

ECB and **CBC** (block cipher modes) require padding; **CFB**, **OFB**, and **CTR** (stream cipher modes) do not require padding because they encrypt data one byte/bit at a time.

Task 5: Message Digest

Commands:
Bash
echo "Hash this data." > hash_input.txt
openssl dgst -md5 hash_input.txt
openssl dgst -sha1 hash_input.txt
openssl dgst -sha256 hash_input.txt

Observation:

All algorithms (**MD5**, **SHA1**, **SHA256**) produce fixed-length outputs with completely different hash values for the same input, demonstrating the proper **avalanche effect** and collision resistance properties (though MD5 and SHA1 are now considered insecure).

Task 6: HMAC

Commands:
Bash
echo "HMAC input data." > hmac_input.txt
openssl dgst -md5 -hmac "mykey" hmac_input.txt
openssl dgst -sha256 -hmac "mykey" hmac_input.txt

Answer:

HMAC (Hashed Message Authentication Code) does **not require a fixed key size**. Keys are automatically hashed or padded to the appropriate **block size** for the underlying hash function (**MD5** or **SHA256**) before being used in the HMAC calculation.

Task 7: Avalanche Effect

Steps:

1. `echo "Original text." > original_task7.txt`
2. `openssl dgst -sha256 original_task7.txt > H1_sha256.txt`
3. **Flipped one bit** in `original_task7.txt` to create `modified_task7.txt` using GHex.
4. `openssl dgst -sha256 modified_task7.txt > H2_sha256.txt`

Observation:

H1 and **H2** are **completely different**, with approximately **50% of the bits changed**. This demonstrates a **strong avalanche effect**, which is a crucial property for secure hash functions.