Lab Manual-3

Amit Hassan joy
Registration Number: 2020831026
Date: November 8, 2025

Objectives:

- To understand and implement symmetric and asymmetric encryption techniques.
- To explore AES encryption in different modes (ECB, CBC, CFB, OFB).
- To analyze the effect of mode differences on ciphertext and corruption propagation.
- To study padding mechanisms in block ciphers
- To generate and compare cryptographic hash digests using SHA and MD5.
- To implement keyed hashing and HMAC.
- To understand the properties of one-way hash functions

Tools Used:

- **OpenSSL** command-line utility
- Hex Workshop-**HxD** (for viewing/editing binary/hex files)
- Python (for comparing hash differences)
- Operating System: Linux / Windows(**WSL**)

Task 1: AES Encryption using Different Modes

Objective: To perform encryption and decryption using AES-128 in CBC, CFB, and ECB modes.
Procedure & Commands:

1. Create a plaintext file **test.txt** containing sample text.
2. Encrypt and decrypt using the following commands:

AES-128-CBC

```
openssl enc -aes-128-cbc -e -in test.txt -out encrypt-aes-128-cbc.bin -k 00112233445566778889aabbccddeeff -iv 01020304050607080102030405060708
```

```
openssl enc -aes-128-cbc -d -in encrypt-aes-128-cbc.bin -out decrypt-aes-128-cbc.txt -k 00112233445566778889aabbccddeeff -iv 01020304050607080102030405060708
```

AES-128-CFB

```
openssl enc -aes-128-cfb -e -in test.txt -out encrypt-aes-
128-cfb.bin -k 00112233445566778889aabbccddeeff -iv
01020304050607080102030405060708
```

```
openssl enc -aes-128-cfb -d -in encrypt-aes-128-cfb.bin -
out decrypt-aes-128-cfb.txt -k
00112233445566778889aabbccddeeff -iv
01020304050607080102030405060708
```

AES-128-ECB

```
openssl enc -aes-128-ecb -e -in test.txt -out encrypt-aes-
128-ecb.bin -k 00112233445566778889aabbccddeeff
```

```
openssl enc -aes-128-ecb -d -in encrypt-aes-128-ecb.bin -
out decrypt-aes-128-ecb.txt -k
00112233445566778889aabbccddeeff
```

Observation:

- Successful encryption and decryption in all modes.
- ECB does not require IV.
- CBC and CFB require an initialization vector (IV).

Task 2: Encryption Mode – ECB vs CBC

Objective: To visualize the difference between ECB and CBC modes using an image file.

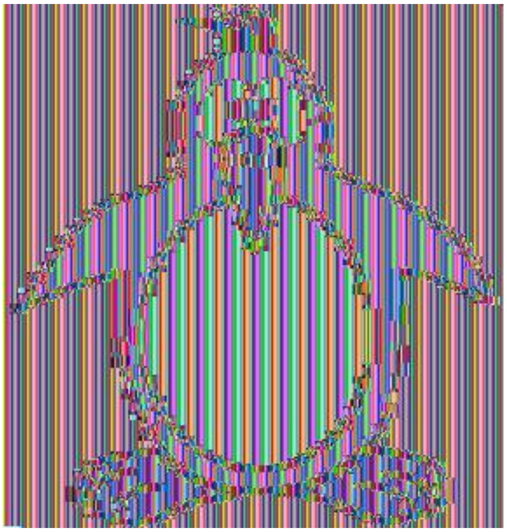
Procedure & Commands:

1. Download a .bmp image (e.g., penguin.bmp).
2. Encrypt using ECB and CBC modes:

ECB Mode

```
openssl enc -aes-128-ecb -e -in penguin.bmp -out
encryptedECB.bmp -K 00112233445566778889aabbccddeeff
```

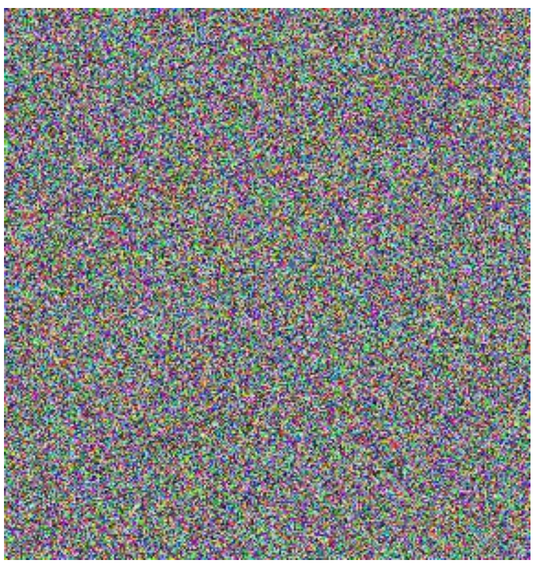
Copy first 54 bytes (header) from original penguin.bmp to encryptedECB.bmp using Hex Workshop.



CBC Mode

```
openssl enc -aes-128-cbc -e -in penguin.bmp -out
encryptedCBC.bmp -K 00112233445566778889aabbccddeeff -iv
20304050607082143234324324233333
```

Replace the first 54 bytes of encryptedCBC.bmp with the header from the original.



My Observation

ECB mode (Electronic Codebook):

- Each block of plaintext is encrypted independently with the same key.
- Identical blocks of plaintext results in identical blocks of ciphertext.
- Less secure for image encryption as patterns, shape may be recognized from encrypted file.

CBC mode (Cipher Block Chaining):

- Each block of plaintext is XORed with the previous ciphertext block before encryption.
- More resistant to patterns and repetition in the plaintext due to added diffusion.
- IV (Initialization Vector) is needed for the first block to start the chaining process. Hence, CBC is slower and more complex than ECB mode.

So, conclusion is `CBC is better than ECB for image encryption` as CBC is more resistant to pattern preservation and provides better security.

Task 3: Encryption Mode – Corrupted Cipher Text

Objective: To study how different encryption modes handle bit corruption in ciphertext.

Procedure & Commands:

1. Create plaintext.txt (≥64 bytes).
2. Encrypt using various AES-128 modes.
3. Corrupt one bit (30th byte) using Hex Workshop.
4. Decrypt using respective commands.

Example (ECB):

```
openssl enc -aes-128-ecb -e -in plaintext.txt -out
encrypted-ecb.bin -K 00112233445566778889aabbccddeeff

# Corrupt 30th byte manually

openssl enc -aes-128-ecb -d -in encrypted-ecb.bin -out
decrypted-ecb.txt -K 00112233445566778889aabbccddeeff
```

Observation Table:

Mode	Effect of Corruption	Recoverable Information
ECB	Affects only the corrupted block	First 16 bytes
CBC	Affects current & next block	First block only

CFB	Partial corruption propagation	~32 bytes intact
OFB	Similar to CFB, limited corruption	~32 bytes intact

Original File

So this is it! This is the end! Part of the journey is the end. Don't cry because it's over. Smile because it happened.

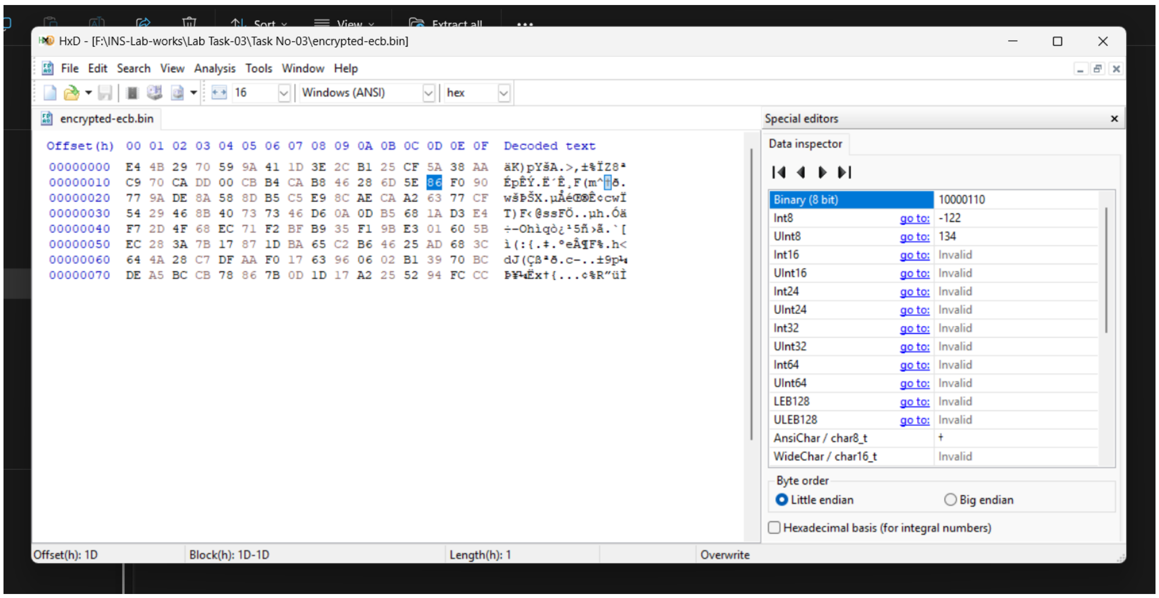


Fig: Corrupt one bit (30th byte) using Hex Workshop

Mode	Corrupted File
ECB	So this is it! T�3R3�5�/��%\�Part of the journey is the end. Don't cry because it's over. Smile because it happened.
CBC	So this is it! T�Z��U��$i|>�O`��Part of the jlurney is the end. Don't cry because it's over. Smile because it happened.
CFB	So this is it! This is the ene! ������m�%�\��3��ney is the end. Don't cry because it's over. Smile because it happened.
OFB	So this is it! This is the enc! Part of the journey is the end. Don't cry because it's over. Smile because it happened.

Summary:

- ECB mode offers the least security and diffusion, as evident from the limited recoverable information.
- CBC mode provides better security compared to ECB but still suffers from partial corruption propagation.
- CFB and OFB modes offer improved diffusion, allowing for more recoverable information compared to CBC mode, although they are still susceptible to partial corruption propagation.

Task 4: Padding

Objective: To study how padding is applied in different AES modes.
Procedure & Commands: Encrypt a non-multiple-of-16-byte file `plain.txt` using AES-128 modes:

```
openssl enc -aes-128-ecb -e -in plain.txt -out encrypted-ecb.bin -K 00112233445566778889aabbccd3322a
```

```
openssl enc -aes-128-cbc -e -in plain.txt -out encrypted-cbc.bin -K 00112233445566778889aabbccd3322a -iv 0102030405060708324123121313124f23
```

```
openssl enc -aes-128-cfb -e -in plain.txt -out encrypted-cfb.bin -K 00112233445566778889aabbccd3322a -iv 0102030405060708324123121313124f23
```

```
openssl enc -aes-128-ofb -e -in plain.txt -out encrypted-ofb.bin -K 00112233445566778889aabbccd3322a -iv 0102030405060708324123121313124f23
```

Observation

`ECB and CBC modes often need padding, while CFB and OFB modes do not.`

Here the plain text size was 23 bytes. And `CFB` and `OFB` encrypted files are also 23 bytes. Which means no padding is needed for these algorithms

But however, `ECB` and `CBC` algorithm made the size of encrypted file 32 Bytes which is a multiple of 16 (Block size of AES-128). So here padding is needed incase of these 2 algorithms.

- **ECB (Electronic Codebook):** Requires padding because it operates on fixed-size blocks of data.
- **CBC (Cipher Block Chaining):** Typically requires padding as each block depends on the previous ciphertext block.
- **CFB (Cipher Feedback):** Does not require padding as it operates in a streaming fashion.

- **OFB (Output Feedback):** Also does not require padding as it operates in a streaming fashion.

In summary, ECB and CBC modes often need padding, while CFB and OFB modes do not. Padding ensures that the plaintext length meets the requirements of the encryption mode.

Task 5: Generating Message Digest

Objective: To generate and compare message digests using SHA-256, SHA-1, and MD5.

Procedure & Commands:

```
openssl dgst -sha256 text.txt

openssl dgst -sha1 text.txt

openssl dgst -md5 text.txt
```

Generated Hashes:

SHA-256 → 2862d2fda986953340b9ad696afb168a6bd02eaa04efaea80452278f5852d416

SHA-1 → 8f0e7d6587f3d754343ead29c2115174891a6c1e

MD5 → 4ef7690f6ba6af63db4de8e29da21bd9

Tabular Comparison:

Algorithm	Hash Size	Security	Remarks
SHA-256	256-bit	Strong	Used in modern cryptography
SHA-1	160-bit	Weak	Collision attacks possible
MD5	128-bit	Very weak	Used for integrity checks only

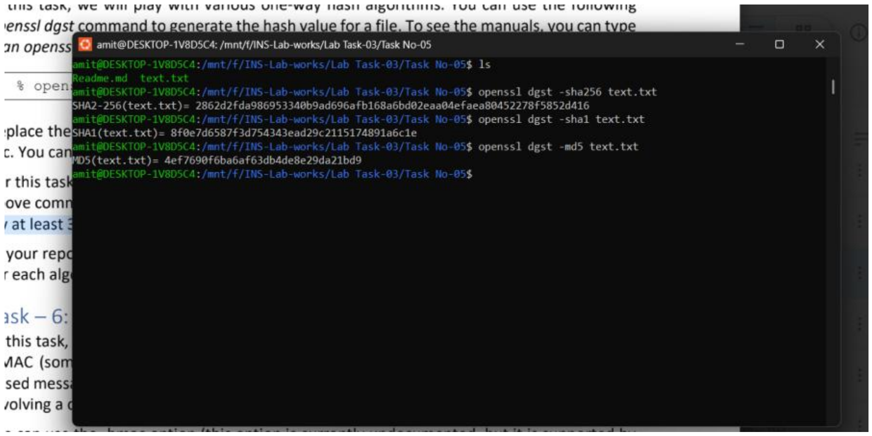


Fig: Command Output Showing Hash Values

Observations

SHA-256

- Produces longer hash value (256bit, 32-byte) compared to MD5 and SHA-1.
- Provides better security against collisions and widely used in modern cryptographic application including digital signatures, certificate authorities, password hashing, and blockchain technology.

SHA-1

- Produces 160 bit (20 byte) hash value.
- Considered weak and vulnerable to collision attacks.

MD5

- Produces 128 bit (16 byte) hash value.
- Fast and commonly used for checksums and data integrity verification.
- Vulnerable to collision attacks.

So, `SHA-256 is the most secure one.` MD5, thought fast, considered insecure. SHA-1 is stronger than MD5 but also vulnerable to collision attacks and less secure than SHA-256.

Task 6: Keyed Hash and HMAC

Objective: To generate HMAC using MD5, SHA-1, and SHA-256 algorithms.
Procedure & Commands:

```
openssl dgst -md5 -hmac "key for hash based mac" text.txt
```

```
openssl dgst -sha1 -hmac "key for hash based mac" text.txt

openssl dgst -sha256 -hmac "key for hash based mac"
text.txt
```

Generated Hashes:

HMAC-MD5: 0eecf7180df087de9b3c42cbc0961243

HMAC-SHA1: 9935d4ad0a05a67b6bf1e2f57e936a6a00703a72

HMAC-SHA256:
85805be3217d735e2c998bca83457f208aa19eb7866b17f4da4bd97eab29ab38

Observation:

- HMAC key length can vary; recommended to match block size of the hash function.
- Provides message integrity and authentication using a shared secret key.

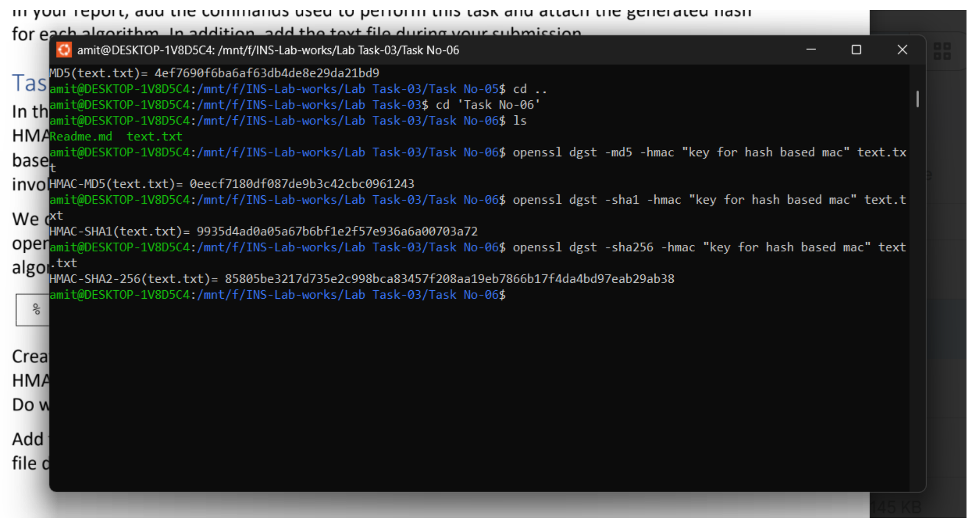


fig: HMAC Command Outputs

Key size in HMAC

- HMAC does not require a key with a fixed size. It can accept keys of any length.
- The key size should be chosen based on the security requirements of the application and the cryptographic algorithm being used.

- However, for HMAC, it's `recommended to use keys that are at least as long as the block size` of the underlying hash function. Such as 16 bytes for HMAC-MD5, 20 bytes for HMAC-SHA1, 32 bytes for HMAC-SHA256.
- If the provided key is shorter than the block size of the hash function, it is usually padded to match the block size using appropriate padding schemes
- Using longer keys can provide better security against brute-force attacks, but excessively long keys may not necessarily enhance security significantly and can incur additional overhead in terms of processing and storage.

Task 7: Avalanche Effect in HMAC

Objective: To verify the avalanche property of cryptographic hash functions using HMAC.
Procedure & Commands:

1. Generate hash H1:

```
openssl dgst -md5 -hmac "this is task 7" for-md5.txt
```

2. Modify one bit of for-md5.txt, then generate hash H2.
3. Generated Hash, H2:

```
823a13f73f84674bd64ec678e074ef16
```

4. Compare both hashes using Python:

```
def checkSame(h1, h2):

  i = 0

  cnt = 0

  for ch in h1:

    if ch==h2[i]:

      cnt = cnt + 1

    i = i + 1

  print("Number of same bit: ", cnt)

h1 = "482c48d5234eff1ca0a1cb3f6d47f8fc"

h2 = "823a13f73f84674bd64ec678e074ef16"

checkSame(h1, h2)
```

Output:

```
Number of same bit : 1
```

Now let's repeat the instructions for `SHA-256` algorithm.

- Copy the text of text.txt and paste into a new file named for-sha256.txt
- Generate the hash value, H1 for this file using `SHA-256` algorithm using the following command:

```
openssl dgst -sha256 -hmac "this is task 7" for-sha256.txt
```

- Generated Hash, H1:

```
3f99d45079d1ad2dd1310ffc66f6bc92e6c119cde8cf093b5dadbcb62b1a1945
```

- After changing a bit in the file, Generated Hash, H2:

```
45837b84493aefc8b6ae765ecc39112563f221de67a2d372efed28a5b6c87e50
```

- Run the ***`count.py`*** with the hash value H1 and H2.

```
Number of same bit : 2
```

So, any single modification in the source file will change hash value to a great extent.

This property of hash functions is widely used in various applications for data integrity verification, digital signatures, and detecting unauthorized modifications.

Observation:

- For MD5, only 1 bit matched between H1 and H2.
- For SHA-256, only 2 bits matched.
- Even a single-bit change causes drastic hash differences, confirming strong avalanche effects.