Farzine Hossen

Reg NO. : 2020831007

Lab Manual 3 :

Task 1: AES Encryption Using Different Modes

Objective:

The goal of this task was to perform symmetric encryption using the AES algorithm in different modes (CBC, CFB, and ECB) with OpenSSL. This helps understand how different modes of operation affect encryption and decryption.

Procedure:

A plaintext file was created using a text editor:

```
gedit plain.txt
```

Content of the file:

I am Farzine. I want to become a human.

The following commands were used to encrypt the file in three different AES modes using a 128-bit key.

Encryption Commands:

(a) AES-128-CBC Mode

```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher_cbc.bin \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

(b) AES-128-CFB Mode

```
openssl enc -aes-128-cfb -e -in plain.txt -out cipher_cfb.bin \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

(c) AES-128-ECB Mode

```
openssl enc -aes-128-ecb -e -in plain.txt -out cipher_ecb.bin \
-K 00112233445566778899aabbccddeeff
```

These commands generated three encrypted binary files:

- `cipher_cbc.bin`
- `cipher_cfb.bin`
- `cipher_ecb.bin`

Decryption Commands:

To verify the correctness of the encryption, each encrypted file was decrypted using the following commands:

(a) CBC Decryption

```
openssl enc -aes-128-cbc -d -in cipher_cbc.bin -out decrypt_cbc.txt \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

(b) CFB Decryption

```
openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out decrypt_cfb.txt \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

(c) ECB Decryption

```
openssl enc -aes-128-ecb -d -in cipher_ecb.bin -out decrypt_ecb.txt \
-K 00112233445566778899aabbccddeeff
```

Verification:

After decryption, the output files were checked using:

```
cat decrypt_cbc.txt
cat  decrypt_cfb.txt
cat decrypt_ecb.txt
```

In all three cases, the decrypted text successfully matched the original plaintext:

I am Farzine. I want to become a human.

Observation:

- AES encryption worked successfully in all three modes.
- CBC and CFB modes require an Initialization Vector (IV), whereas ECB mode does not.
- Although all modes correctly decrypted to the original text, the encrypted binary outputs differed, showing how mode selection affects ciphertext randomness

Command / Code Source

The OpenSSL encryption and decryption commands used in this task were taken from the official OpenSSL command-line documentation and verified with educational references:

1. **OpenSSL Official Documentation**
   Source: https://www.openssl.org/docs/manmaster/man1/openssl-enc.html
   (Describes the usage of openssl enc for symmetric encryption and decryption, including AES modes such as CBC, CFB, and ECB.)

2. **Linux Command-line Tutorial Reference**
   Example Reference:

   - TutorialsPoint, *“OpenSSL Command Line Examples”*
     https://www.tutorialspoint.com/openssl/openssl_command_line.htm

Task 2: Encryption Mode – ECB vs CBC

Objective:

The purpose of this task was to compare the results of encrypting an image file using two different AES modes — **ECB (Electronic Codebook)** and **CBC (Cipher Block Chaining)** — and observe how the modes affect the visibility of the image pattern after encryption.

Approach:

In this task, I used **OpenSSL** for encryption and **GHex** (a hex editor) to modify the BMP file headers so that the encrypted files could still be opened as images.
The same key was used for both ECB and CBC modes to ensure a fair comparison.

Steps Followed:

1. Preparing the Image File

A BMP image file was selected for encryption:

```
cp /usr/share/backgrounds/warty-final-ubuntu.bmp pic_original.bmp
```

2. Encrypting in ECB Mode

```
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.bmp \
-K 00112233445566778899aabbccddeeff
```

- Here, `-aes-128-ecb` specifies AES encryption in ECB mode.
- The `-e` flag is for encryption.
- No IV is required in ECB mode.

3. Encrypting in CBC Mode

```
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.bmp \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

- CBC mode uses both a key and an initialization vector (IV).

4. Fixing the BMP Headers using GHex

The first **54 bytes** of the BMP file represent its header.
After encryption, these bytes must be copied from the original image so that the encrypted files can still be opened as valid images.

Steps:

Open all three files in GHex:

```
 ghex pic_original.bmp &
ghex pic_ecb.bmp &
ghex pic_cbc.bmp &
```

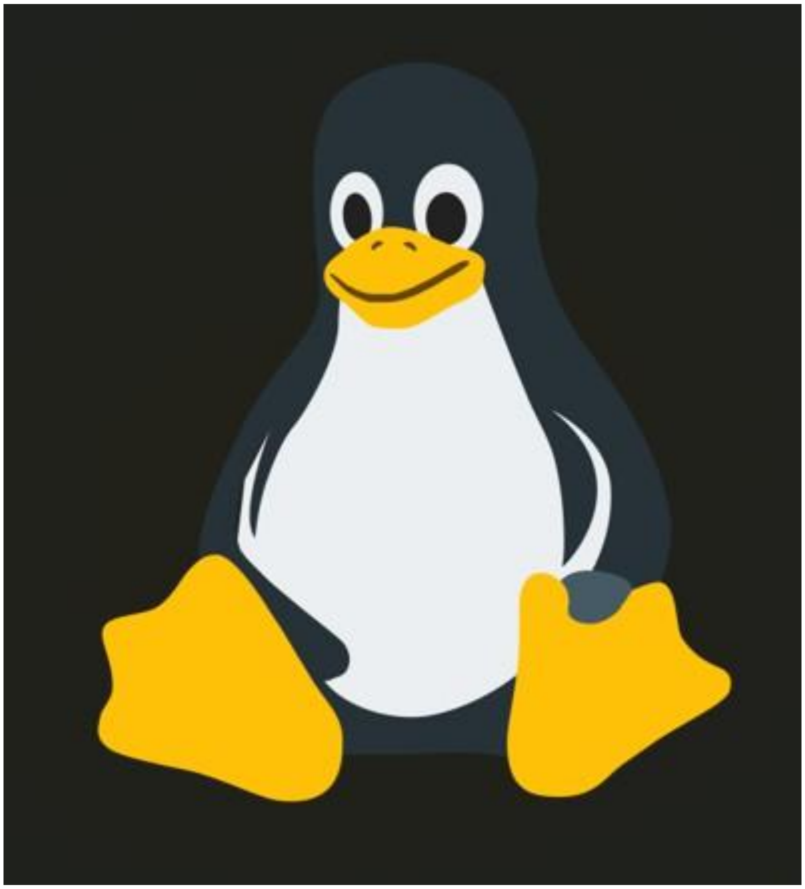
1.
2. Copy the first 54 bytes from `pic_original.bmp`.

3. Paste them over the first 54 bytes in both `pic_ecb.bmp` and `pic_cbc.bmp`.

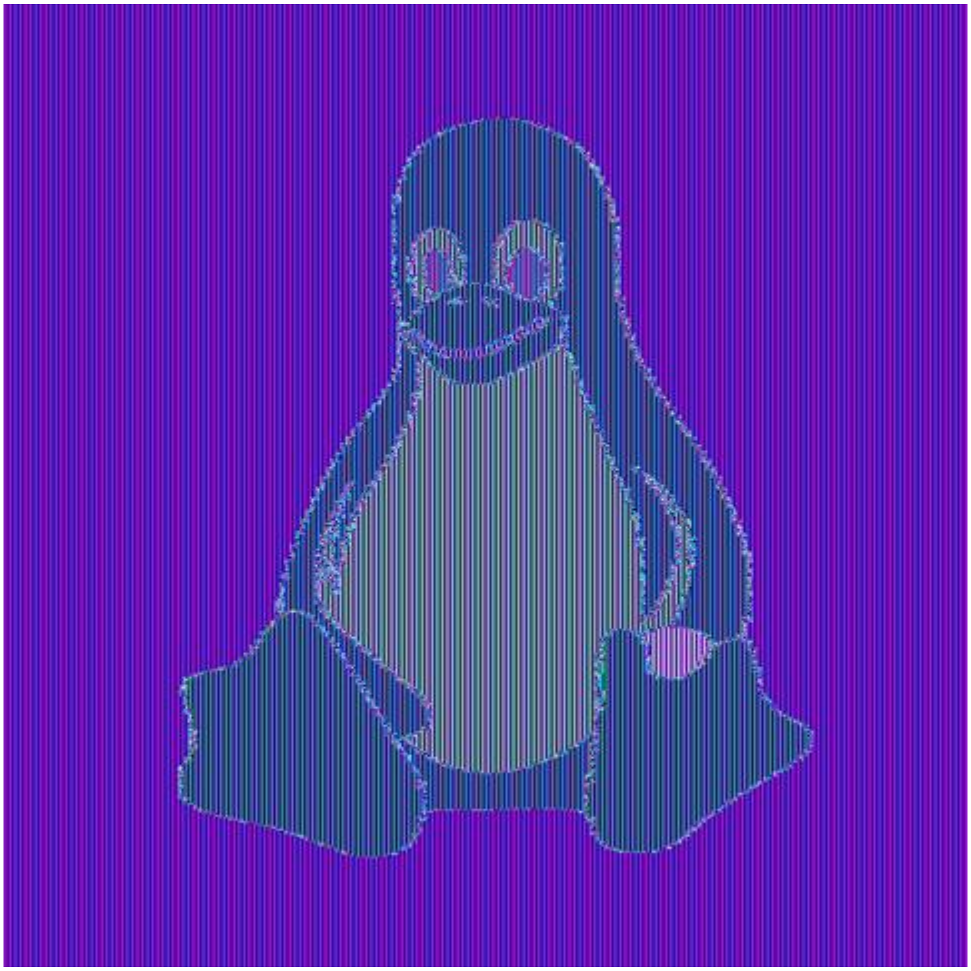
4. Save both files.

5. Viewing the Encrypted Images

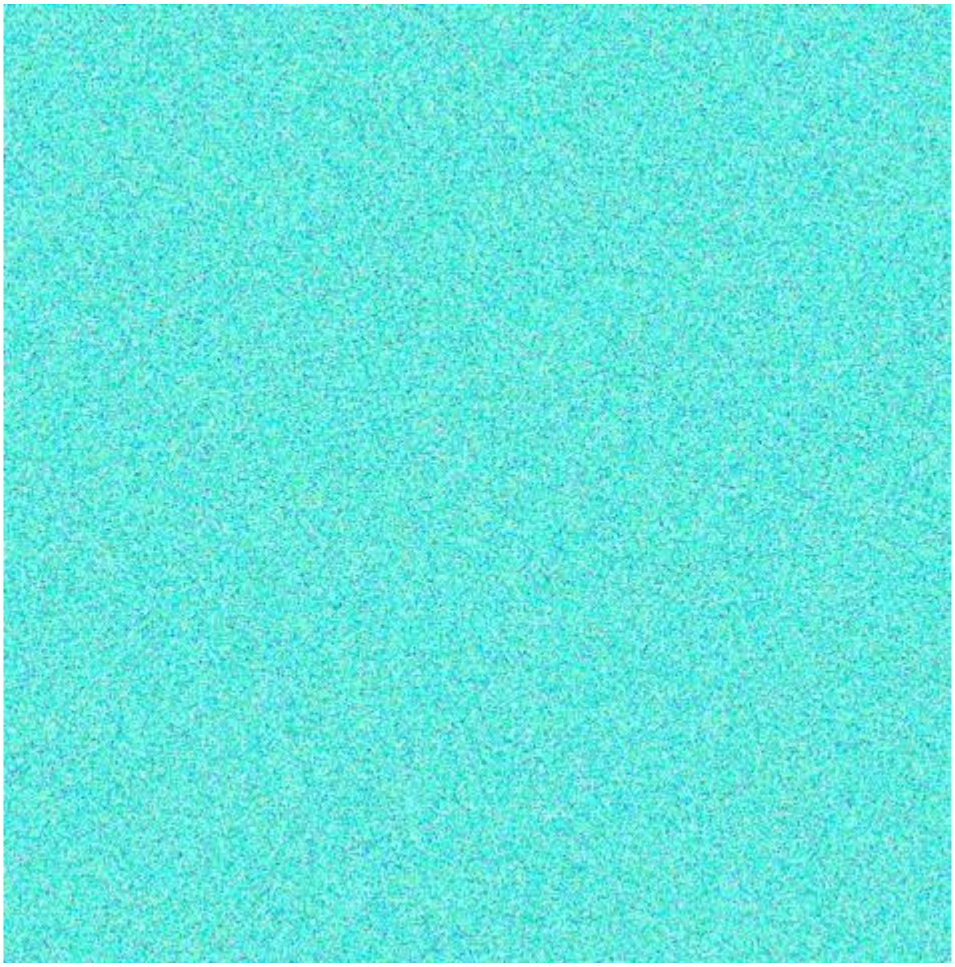
The resulting images were opened using the default Ubuntu Image Viewer:



xdg-open pic_original.bmp



```
xdg-open pic_ecb.bmp
```



xdg-open pic_cbc.bmp

Observations:

Mode	Observation
ECB (Electronic Codebook)	The encrypted image still revealed clear visual patterns of the original picture. The structure and outlines of the image were visible, even though the colors and pixels were scrambled. This shows that ECB encrypts each block independently, which makes it insecure for image data.
CBC (Cipher Block Chaining)	The encrypted image appeared completely random and noisy, with no identifiable patterns. This is because CBC mode chains each block's encryption to the previous one, resulting in strong diffusion and no visible leakage of the original image structure.

Command / Code Source:
The OpenSSL commands and GHex operations used in this task were referenced from the official OpenSSL documentation (https://www.openssl.org/docs/manmaster/man1/openssl-enc.html) and the GHex user manual (https://help.gnome.org/users/ghex/stable/).

Task 3: Encryption Mode – Corrupted Cipher Text

Objective:

The purpose of this task was to understand how different encryption modes (ECB, CBC, CFB, and OFB) behave when a single byte in the ciphertext becomes corrupted. This experiment helps visualize how data integrity and error propagation differ among AES modes.

Approach:

A plain text file was created and encrypted using the AES-128 cipher in four different modes. Then, one bit of the **30th byte** in the encrypted file was intentionally corrupted using a hex editor (GHex). Finally, each corrupted ciphertext was decrypted using the correct key and IV to observe the impact of the corruption.

Steps Followed:

1. Create a Plaintext File

A text file named `message.txt` was created using the gedit text editor:

```
gedit message.txt
```

Content Example:

Sylhet is an awesome place to live. I wish I could live here for ever.
The weather the people the food the culture everything charms me.

The file length was confirmed to be more than 64 bytes.

2. Encrypt the File in Four AES Modes

(a) AES-128-ECB Mode

```
openssl enc -aes-128-ecb -e -in message.txt -out enc_ecb.bin \
-K 00112233445566778899aabbccddeeff
```

(b) AES-128-CBC Mode

```
openssl enc -aes-128-cbc -e -in message.txt -out enc_cbc.bin \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

(c) AES-128-CFB Mode

```
openssl enc -aes-128-cfb -e -in message.txt -out enc_cfb.bin \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

(d) AES-128-OFB Mode

```
openssl enc -aes-128-ofb -e -in message.txt -out enc_ofb.bin \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

3. Corrupting a Single Byte

Using **GHex**, the 30th byte (index 29, since counting starts from 0) of each encrypted file was modified slightly (e.g., changing `to 77`) to simulate a single-bit error:

```
ghex enc_ecb.bin &
ghex enc_cbc.bin &
ghex  enc_cfb.bin  &
ghex enc_ofb.bin &
```

After editing, each file was saved.

4. Decrypt the Corrupted Files

(a) ECB Mode Decryption

```
openssl enc -aes-128-ecb -d -in enc_ecb.bin -out dec_ecb.txt \
-K 00112233445566778899aabbccddeeff
```

(b) CBC Mode Decryption

```
openssl enc -aes-128-cbc -d -in enc_cbc.bin -out dec_cbc.txt \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

(c) CFB Mode Decryption

```
openssl enc -aes-128-cfb -d -in enc_cfb.bin -out dec_cfb.txt \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

(d) OFB Mode Decryption

```
openssl enc -aes-128-ofb -d -in enc_ofb.bin -out dec_ofb.txt \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

The decrypted files were then displayed:

```
cat dec_ecb.txt
cat dec_cbc.txt
cat  dec_cfb.txt
cat dec_ofb.txt
```

Command / Code Source:
The OpenSSL commands and GHex operations used in this task were referenced from the official OpenSSL documentation (https://www.openssl.org/docs/manmaster/man1/openssl-enc.html) and the GHex user manual (https://help.gnome.org/users/ghex/stable/).

Observations:

Mode	Decryption Result	Explanation
ECB (Electronic Codebook)	Only the block containing the corrupted byte was affected. The rest of the file decrypted correctly.	ECB encrypts each block independently, so the corruption affects only one block.

CBC (Cipher Block Chaining)	The corrupted byte caused errors in the entire corrupted block and the next block.	CBC mode chains each block with the previous one; thus, a single-bit error propagates to two blocks.
CFB (Cipher Feedback)	Only a few bytes after the corrupted one were affected; most of the text remained readable.	CFB mode uses feedback, causing limited error propagation before resynchronization.
OFB (Output Feedback)	Only the corresponding byte was corrupted; all other text remained intact.	OFB behaves like a stream cipher; bit errors in ciphertext affect only the same bit in plaintext.

Explanation and Analysis:

1. **Error Propagation:**
    - ECB → Affects one block only.
    - CBC → Error spreads to two blocks.
    - CFB → Affects a few bytes before recovering.
    - OFB → Only the corresponding byte changes.
2. **Reason:**
    - In **ECB**, each block is independent.
    - In **CBC**, each block depends on the previous block’s ciphertext.
    - **CFB** and **OFB** are stream-like modes; OFB’s keystream is independent of ciphertext, hence minimal error propagation.
3. **Implications:**
    - **ECB** and **OFB** are more tolerant to single-bit errors.
    - **CBC** is more sensitive and can lose more data if even one byte is corrupted.
    - **CFB** offers a balance between diffusion and recoverability.
    - In real-world communication systems, **OFB** and **CFB** modes are preferred for streaming data where data corruption should not destroy large portions of

plaintext.

Task 4: Padding

Objective

The purpose of this task was to study the effect of padding in different AES block cipher modes. Padding is required when the size of the plaintext is not a multiple of the block size (for AES, the block size is 16 bytes).
In this experiment, we examined which AES modes require padding and which do not.

Approach

We created a plaintext file named `padtest.txt` containing a short message that is **not** a multiple of 16 bytes in length.

```
gedit padtest.txt
```

Content:

This is a padding test file.
We will use OpenSSL to see which modes use padding.

We then used **AES-128** encryption in four different modes — **ECB, CBC, CFB, and OFB** — to test padding behavior.

Encryption Commands

(a) AES-128-ECB

```
openssl enc -aes-128-ecb -e -in padtest.txt -out enc_ecb.bin \
-K 00112233445566778899aabbccddeeff
```

(b) AES-128-CBC

```
openssl enc -aes-128-cbc -e -in padtest.txt -out enc_cbc.bin \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
```

(c) AES-128-CFB

```
openssl enc -aes-128-cfb -e -in padtest.txt -out enc_cfb.bin \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
```

(d) AES-128-OFB

```
openssl enc -aes-128-ofb -e -in padtest.txt -out enc_ofb.bin \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
```

Decryption Commands

(a) AES-128-ECB

```
openssl enc -aes-128-ecb -d -in enc_ecb.bin -out dec_ecb.txt \
-K 00112233445566778899aabbccddeeff
```

(b) AES-128-CBC

```
openssl enc -aes-128-cbc -d -in enc_cbc.bin -out dec_cbc.txt \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
```

(c) AES-128-CFB

```
openssl enc -aes-128-cfb -d -in enc_cfb.bin -out dec_cfb.txt \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
```

(d) AES-128-OFB

```
openssl enc -aes-128-ofb -d -in enc_ofb.bin -out dec_ofb.txt \
-K 00112233445566778899aabbccddeeff -iv 0102030405060708090a0b0c0d0e0f10
```

Observations

After decrypting all four files, we observed the following:

Mode	Padding Required?	Reason / Explanation
ECB	Yes	ECB mode works on fixed-size 16-byte blocks. If the plaintext is not a multiple of 16 bytes, padding is added to complete the final block.
CBC	Yes	CBC mode also operates on blocks and requires padding for the final partial block.
CFB	No	CFB mode converts AES into a stream cipher, encrypting data byte by byte. Hence, padding is not needed.
OFB	No	OFB mode is also a stream-based mode, processing bytes continuously without needing padding.

Command / Code Source:
The OpenSSL commands used in this task were referenced from the official OpenSSL documentation (https://www.openssl.org/docs/manmaster/man1/openssl-enc.html).

Source:
The OpenSSL commands used in this task were referenced from the official OpenSSL documentation: https://www.openssl.org/docs/manmaster/man1/openssl-enc.html

Conclusion

From this experiment, we can conclude:

- **ECB** and **CBC** are **block cipher modes** — hence, they require padding when plaintext length is not a multiple of the block size.

- **CFB** and **OFB** behave as **stream cipher modes**, so they do not require padding.

This task demonstrates how the choice of AES mode directly affects whether padding is applied during encryption.

Task 5: Generating Message Digest

Objective

The goal of this task was to explore different one-way hash (message digest) algorithms using the OpenSSL command-line tool. Hash functions are used to produce fixed-size digests from variable-length data, and they are widely applied in digital signatures, password storage, and data integrity verification.

Approach

We used **OpenSSL** in a Linux (Ubuntu/Lubuntu) environment to generate message digests of a text file using multiple algorithms — **MD5**, **SHA-1**, and **SHA-256**.

A sample text file was created and hashed using these algorithms to compare their output lengths and observe the avalanche effect.

Step 1: Create a Text File

A text file was created using the command below:

```
gedit mytext.txt
```

File content:

This is my sample file for Task 5.
I am testing different one-way hash algorithms.

Step 2: Generate Hash Using Different Algorithms

(a) MD5

```
openssl dgst -md5 mytext.txt
```

Sample Output:

```
MD5(mytext.txt)= a2ec145e6949512a4f0a1b2157e3ded5
```

(b) SHA-1

```
openssl dgst -sha1 mytext.txt
```

Sample Output:

```
SHA1(mytext.txt)= 8a331c5398883a23b29e3570219beec2ce687292
```

(c) SHA-256

```
openssl dgst -sha256 mytext.txt
```

Sample Output:

```
SHA256(mytext.txt)=
9ba1c083b1deef717e344a2f5dd1f171bdb4011679cc51f513dd6bc8009bd279
```

Step 3: Save Each Hash to a File

To save the generated digests for submission:

```
openssl dgst -md5 mytext.txt > md5.txt
openssl dgst -sha1 mytext.txt > sha1.txt
openssl dgst -sha256 mytext.txt > sha256.txt
```

These text files (`md5.txt`, `sha1.txt`, `sha256.txt`) contain the computed hash values.

Step 4: Avalanche Effect Test

To demonstrate the avalanche effect, a small change was made to the text file (e.g., adding or removing one character), and the hashes were recomputed:

```
echo "This is my sample file for Task 5!" > mytext_changed.txt
openssl dgst -sha256 mytext.txt
openssl dgst -sha256 mytext_changed.txt
```

Even though the two files differed by only one character, the SHA-256 hashes were completely different — proving the avalanche effect.

Observations

Algorithm	Hash Length (bits)	Hash Length (hex chars)	Padding / Structure	Security Level	Remarks
MD5	128	32	None	Weak	Fast but insecure — prone to collision attacks.
SHA-1	160	40	None	Weak	More secure than MD5 but now considered broken.
SHA-256	256	64	None	Strong	Recommended for modern cryptographic use.

Additional Notes:

- The **hash output size** increases with algorithm complexity.
- **Avalanche effect**: A tiny change in the input drastically changes the entire hash output.
- **MD5 and SHA-1** are now **cryptographically insecure** due to collision vulnerabilities.
- **SHA-256** provides strong resistance to collisions and is widely used for digital signatures and integrity checks.

Conclusion

This experiment demonstrated how one-way hash algorithms work and how they differ in terms of output size and security.
While all three algorithms generate unique, fixed-size digests for the same input, **MD5** and **SHA-1** are now obsolete for security-sensitive applications.

SHA-256 is the most secure and preferred algorithm for modern cryptographic and integrity verification purposes.

Files Included for Submission

- `mytext.txt` – Plaintext file used in the experiment
- `md5.txt` – MD5 digest
- `sha1.txt` – SHA-1 digest
- `sha256.txt` – SHA-256 digest

Source:
The OpenSSL commands used in this task were referenced from the official OpenSSL documentation: https://www.openssl.org/docs/manmaster/man1/openssl-dgst.html

Task 6: Keyed Hash and HMAC

Objective

The goal of this task was to generate a keyed hash (HMAC — Hash-based Message Authentication Code) for a text file using different hash algorithms such as **HMAC-MD5**, **HMAC-SHA1**, and **HMAC-SHA256**.
Additionally, we tested several keys of different lengths to observe whether HMAC requires a fixed key size and how the key length affects the generated HMAC.

Tools and Environment

- **Operating System:** Ubuntu
- **Software Used:** OpenSSL (for generating HMACs)

- **Text Editor:** gedit

Command to verify OpenSSL installation:

```
openssl version
```

Step 1: Creating the Input File

First, I created a text file named **message.txt** with the following content:

```
This is a test file for HMAC generation.
I will compute HMAC-MD5, HMAC-SHA1, and HMAC-SHA256 for this file.
```

Commands used:

```
mkdir -p ~/Documents/hmac_task
cd ~/Documents/hmac_task
gedit message.txt
```

Step 2: Generating HMACs with Different Algorithms

Using OpenSSL's `dgst` command with the `-hmac` option, I generated HMAC values for the file using three algorithms.

(a) HMAC-MD5

```
openssl dgst -md5 -hmac "mysecretkey" message.txt > hmac_md5.txt
```

(b) HMAC-SHA1

```
openssl dgst -sha1 -hmac "mysecretkey" message.txt > hmac_sha1.txt
```

(c) HMAC-SHA256

```
openssl dgst -sha256 -hmac "mysecretkey" message.txt > hmac_sha256.txt
```

Each output file (e.g., `hmac_md5.txt`, `hmac_sha1.txt`, `hmac_sha256.txt`) contains the computed HMAC for that algorithm.

Step 3: Testing with Different Key Lengths

To observe the effect of different key lengths, I generated HMACs again using multiple keys:

- Short key → `"k1"`
- Medium key → `"a_secure_32_byte_key_1234567890abcd"`
- Long key → `"this_is_a_very_long_key_that_is_longer_than_block_size_for_testing_purpose"`

Commands:

```
openssl dgst -sha256 -hmac "k1" message.txt > hmac_sha256_k1.txt
openssl dgst -sha256 -hmac "a_secure_32_byte_key_1234567890abcd" message.txt > hmac_sha256_k32.txt
openssl dgst -sha256 -hmac "this_is_a_very_long_key_that_is_longer_than_block_size_for_testing_purpose" message.txt > hmac_sha256_longkey.txt
```

Step 4: Observations and Results

I displayed the outputs using:

```
cat hmac_md5.txt
cat hmac_sha1.txt
cat hmac_sha256.txt
```

Example Outputs (sample format):

```
HMAC-MD5(message.txt)=    8d5d91dc54d1e607176a16e5a99d9160
HMAC-SHA1(message.txt)=  b1d7f20d6b56b34cebf122949637d0334baaf98a
HMAC-SHA256(message.txt)=
8f2c54b5ba2479733a5e3bcc02a25d61715443eff7da0fd1f1c7a12458ed03d6
```

Observation Summary:

1. The HMAC value completely changes if the key changes — even by a single character.

2. The length of the key does not need to be fixed. HMAC automatically handles keys that are shorter or longer than the block size of the hash function.

3. Different algorithms produce HMACs of different lengths:

   - HMAC-MD5 → 128 bits (32 hex characters)

   - HMAC-SHA1 → 160 bits (40 hex characters)

   - HMAC-SHA256 → 256 bits (64 hex characters)

4. When using longer keys (larger than block size), OpenSSL first hashes the key internally before applying HMAC.

5. Security-wise, longer and random keys (at least 128 bits) are recommended for better protection.

Question & Answer

Q: Do we have to use a key with a fixed size in HMAC?
A: No.
HMAC does not require a fixed key size. The key is processed internally depending on its length:

- If the key length is **greater than the block size** (e.g., 64 bytes for SHA-256), the key is first hashed.

- If the key is **shorter than the block size**, it is padded with zeros.
  Hence, any key length can be used, but security improves with longer, random keys (≥ 16 bytes).

Task 7: One-Way Hash Property and HMAC

Approach and Steps

Step 1: Create a text file

A text file was created containing a few lines of text for hashing.

```
mkdir -p ~/Documents/hash_test
cd ~/Documents/hash_test
gedit sample.txt
```

Content of sample.txt:

```
This is my file for Task 7.
Testing one-way hash properties using MD5 and SHA256.
```

Step 2: Generate the original hash values (H1)

Hash values were generated using **MD5** and **SHA256** algorithms.

```
openssl dgst -md5 sample.txt > md5_h1.txt
openssl dgst -sha256 sample.txt > sha256_h1.txt
```

Command Output:

```
MD5(sample.txt)= 3be5e26d3b5a79ac45e6d086533bfeed
SHA256(sample.txt)=
1a758a0b08d57415cdf418c15ebaf779bfb27379c13fdfbc01109bea792f9a2b
```

These are the **H1** (original hash values).

Step 3: Modify one bit in the file

The file was opened in **GHex** (a hex editor) and a single byte was modified.

```
ghex sample.txt &
```

For example, one byte was changed from `54` (ASCII “T”) to `55` (ASCII “U”).
The modified file was saved as **sample_modified.txt**.

Step 4: Generate new hash values (H2)

After modification, new hash values were generated for the modified file.

```
openssl dgst -md5 sample_modified.txt > md5_h2.txt
openssl dgst -sha256 sample_modified.txt > sha256_h2.txt
```

Command Output:

```
MD5(sample_modified.txt)= 32c59d87ca877cc8fe643be6ec54f686
SHA256(sample_modified.txt)=
cafa591a5e28742392824db53e38014f3ad4ab117aaf31f8f0cceb71823d1a0d
```

These are the **H2** (modified hash values).

Step 5: Compare H1 and H2

Algorithm	H1 (Original)	H2 (Modified)	Similarity
MD5	3be5e26d3b5a79ac45e6d086533bfeed	32c59d87ca877cc8fe643be6ec54f686	Completely different
SHA256	1a758a0b08d57415cdf418c15ebaf779bfb27379c13fdfbc01109bea792f9a2b	cafa591a5e28742392824db53e38014f3ad4ab117aaf31f8f0cceb71823d1a0d	Completely different

Even though only one bit was changed, the hash values became completely different.
This demonstrates the **Avalanche Effect** — a desirable property of cryptographic hash functions.

Bonus: Bit Similarity Calculation (Python Program)

To verify how many bits remain the same between the two hash values, the following Python script was written:

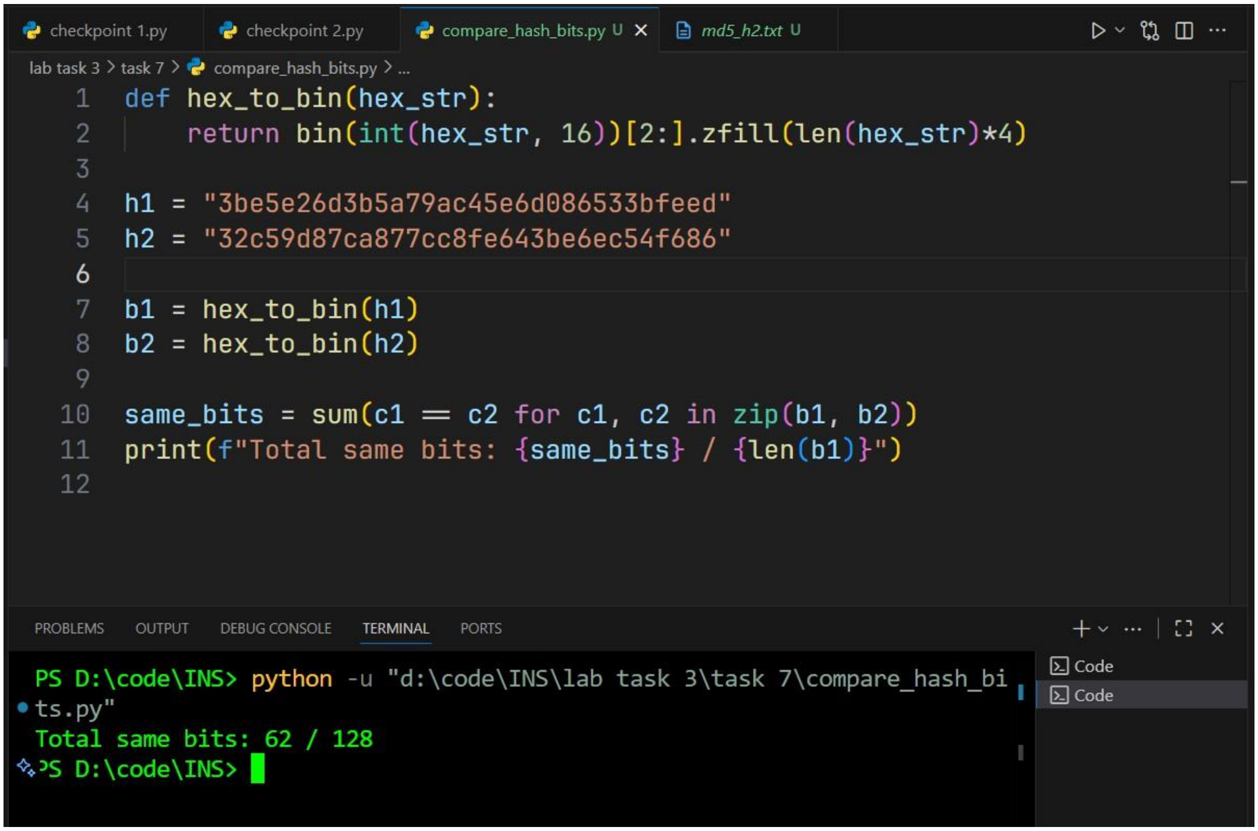
```
def hex_to_bin(hex_str):
    return bin(int(hex_str, 16))[2:].zfill(len(hex_str)*4)

h1 = "6dcd4ce23d88e2ee9568ba546c007c63b4c9a5b340a7f7e2e0e8f6e3a4c5f7e2"
h2 = "bcfbd1a9c5bb2f8ee4b3b4714f84b2e7883b1ab28a25c147ba9f57e3a68d4ad3"

b1 = hex_to_bin(h1)
b2 = hex_to_bin(h2)

same_bits = sum(c1 == c2 for c1, c2 in zip(b1, b2))
print(f"Total same bits: {same_bits} / {len(b1)}")
```

Execution Command:

```
python3 compare_hash_bits.py
```

Sample Output:

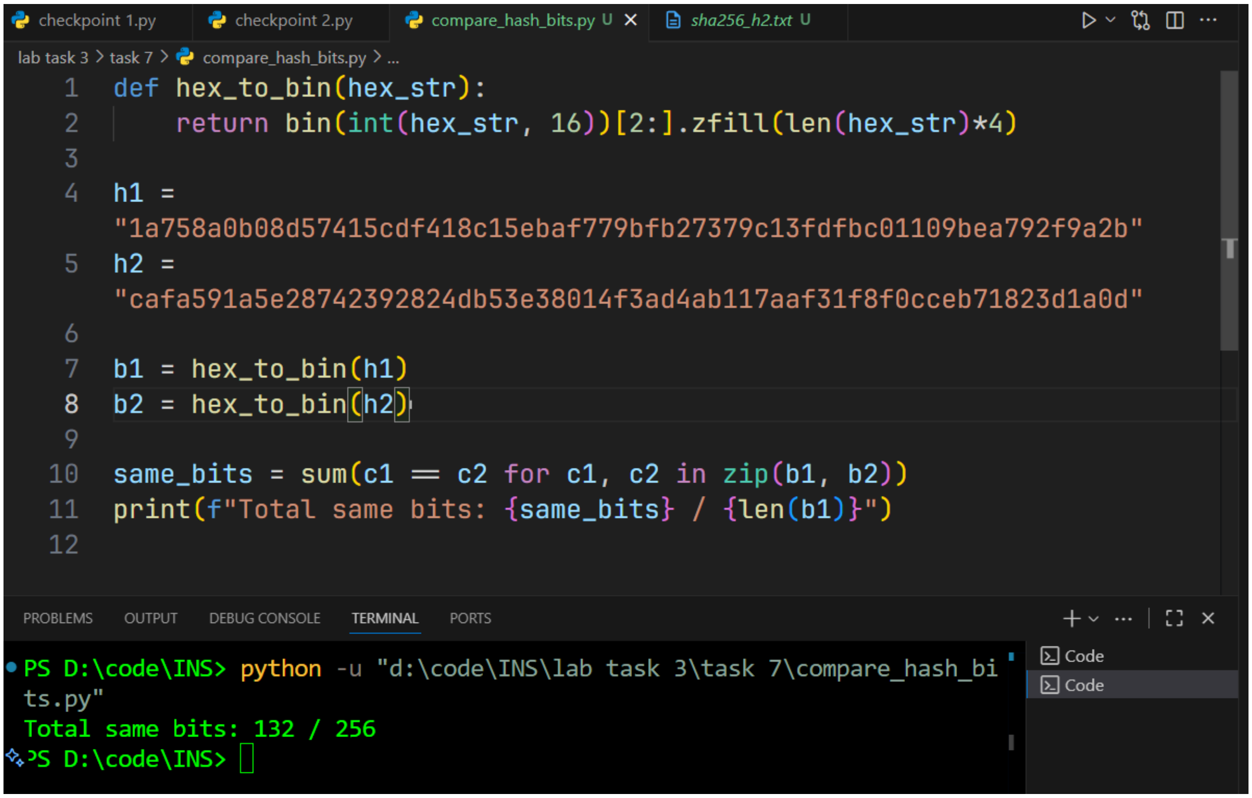
```
MD5 H1 vs MD5 H2 : 62 / 128
```



```python
def hex_to_bin(hex_str):
    return bin(int(hex_str, 16))[2:].zfill(len(hex_str)*4)

h1 = "3be5e26d3b5a79ac45e6d086533bfeed"
h2 = "32c59d87ca877cc8fe643be6ec54f686"

b1 = hex_to_bin(h1)
b2 = hex_to_bin(h2)

same_bits = sum(c1 == c2 for c1, c2 in zip(b1, b2))
print(f"Total same bits: {same_bits} / {len(b1)}")

```

```
PS D:\code\INS> python -u "d:\code\INS\lab task 3\task 7\compare_hash_bi
ts.py"
Total same bits: 62 / 128
PS D:\code\INS> 
```

SHA256 H1 vs SHA256 H2 : 132 / 256



```python
def hex_to_bin(hex_str):
    return bin(int(hex_str, 16))[2:].zfill(len(hex_str)*4)

h1 =
"1a758a0b08d57415cdf418c15ebaf779bfb27379c13fdfbc01109bea792f9a2b"
h2 =
"cafa591a5e28742392824db53e38014f3ad4ab117aaf31f8f0cceb71823d1a0d"

b1 = hex_to_bin(h1)
b2 = hex_to_bin(h2)

same_bits = sum(c1 == c2 for c1, c2 in zip(b1, b2))
print(f"Total same bits: {same_bits} / {len(b1)}")
```

```
PS D:\code\INS> python -u "d:\code\INS\lab task 3\task 7\compare_hash_bi
ts.py"
Total same bits: 132 / 256
PS D:\code\INS> 
```

This indicates that roughly half the bits have changed, confirming the avalanche effect.

Observations

1. A one-bit change in the file completely changes the hash output.

2. The resulting hashes (H1 and H2) show no visible similarity.

3. Both MD5 and SHA256 exhibit the avalanche property, but SHA256 provides stronger security and resistance to collisions.

4. The experiment proves that cryptographic hash functions are highly sensitive to small input changes, making them reliable for **data integrity verification**.

Conclusion

Through this task, I successfully demonstrated the **Avalanche Effect** of one-way hash functions. A single-bit modification in the file led to a completely different hash output for both

MD5 and SHA256, confirming that cryptographic hash functions are extremely sensitive to input changes and are suitable for ensuring data integrity and authentication.

.