Lab 3 Report: Symmetric Encryption & Hashing

Task 1: AES Encryption Using Different Modes (2 Marks)

Objective

Encrypt a text file using AES with three different modes (CBC, ECB, CFB) and verify the encryption by decrypting the files.

Implementation

A plain text file was created with the following content:

```
Course Title: Information & Network Security
Task: Lab 3: Symmetric Encryption & Hashing
Student Name: Nixon Deb Antu
```

Commands Used

CBC Encryption:

```
openssl enc -aes-128-cbc -e -in plain.txt -out out-cbc.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
```

ECB Encryption:

```
openssl enc -aes-128-ecb -e -in plain.txt -out out-ecb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90
```

CFB Encryption:

```
openssl enc -aes-128-cfb -e -in plain.txt -out out-cfb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
```

Verification

All encrypted files were successfully decrypted using the corresponding decryption commands, confirming that the encryption process worked correctly.

Decryption Commands:

```
# CBC Decryptionopenssl enc -aes-128-cbc -d -in out-cbc.bin -out decrypted-cbc.txt \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
# ECB Decryptionopenssl enc -aes-128-ecb -d -in out-ecb.bin -out decrypted-ecb.txt \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90
# CFB Decryptionopenssl enc -aes-128-cfb -d -in out-cfb.bin -out decrypted-cfb.txt \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
```

Output Files

- `out-cbc.bin` - CBC encrypted file
- `out-ecb.bin` - ECB encrypted file
- `out-cfb.bin` - CFB encrypted file

Task 2: Encryption Mode - ECB vs CBC (3 Marks)

Objective

Encrypt a BMP image file using ECB and CBC modes, replace the header to make it viewable, and analyze the security implications.

Implementation

Encryption Commands:

ECB Encryption:

```
openssl enc -aes-128-ecb -e -in original.bmp -out pic_ecb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90
```

CBC Encryption:

```
openssl enc -aes-128-cbc -e -in original.bmp -out pic_cbc.bin \  -K a1b2c3
d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
```

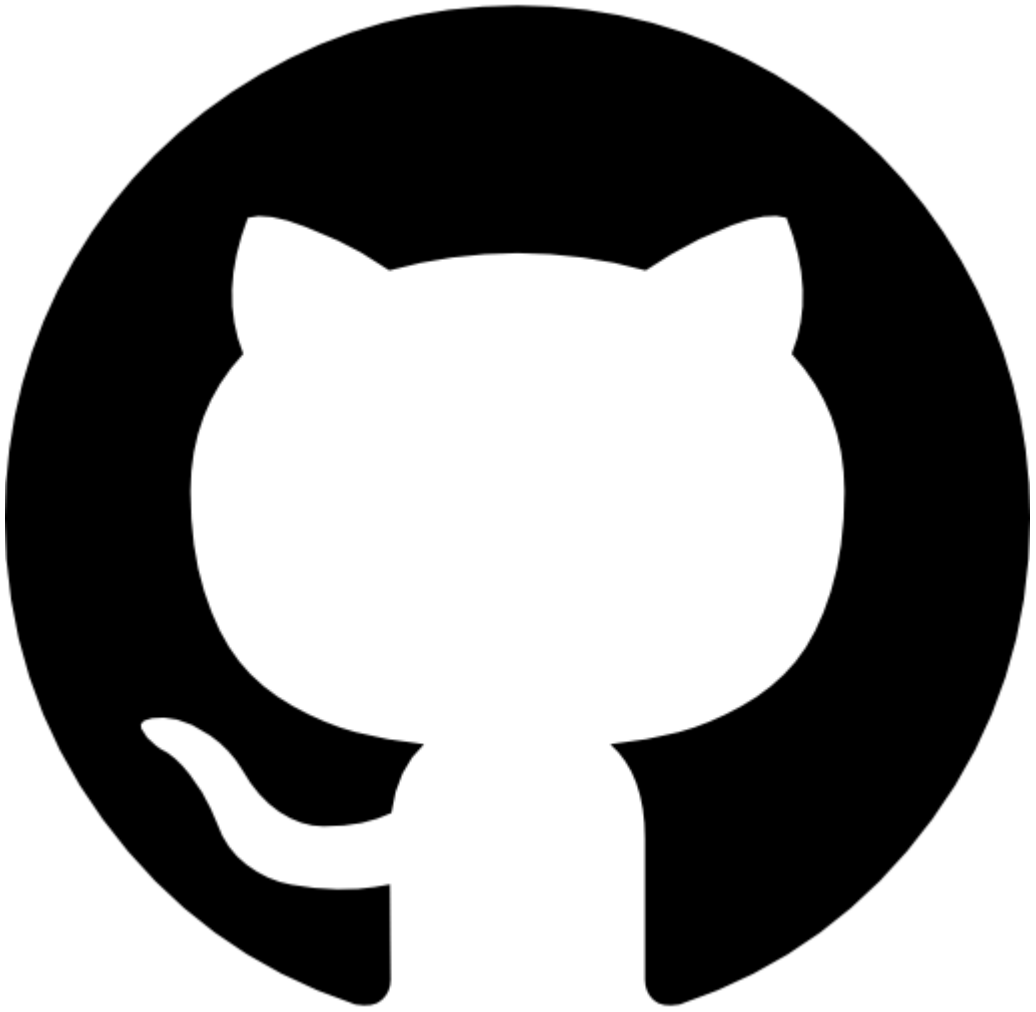
Header Replacement

Using GHex hex editor, the first 54 bytes (BMP header) from the original image were copied to the encrypted files to create viewable BMP files:

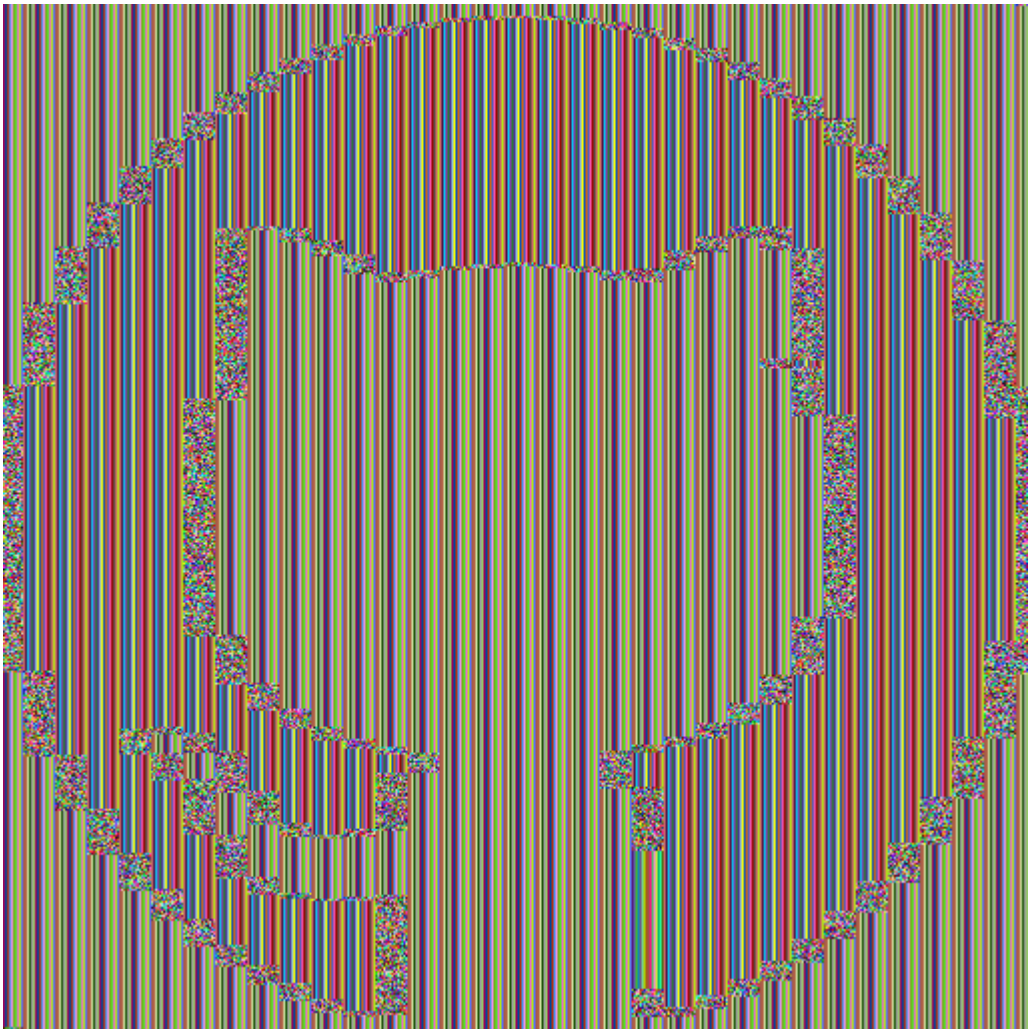
- `pic_ecb.bmp` - ECB encrypted image with original header
- `pic_cbc.bmp` - CBC encrypted image with original header

Observations

Original Image:

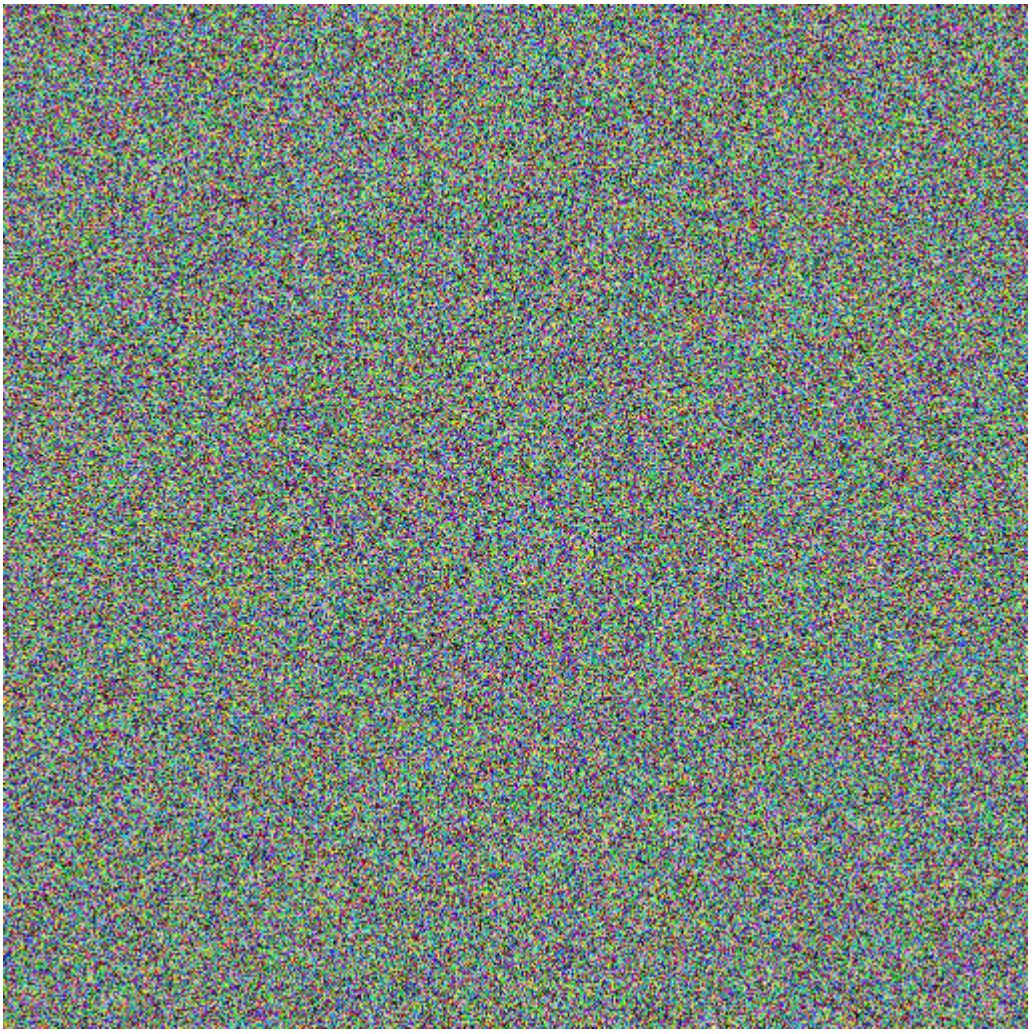


ECB Mode:



- The encrypted image reveals patterns from the original image
- Similar blocks in the plaintext produce identical ciphertext blocks
- This demonstrates ECB’s weakness: it does not hide data patterns effectively
- The encrypted image shows visible patterns that can reveal information about the original image structure

CBC Mode:



- The encrypted image appears completely random with no visible patterns
- Each block's encryption depends on the previous block through the IV and chaining
- No information about the original image can be derived from the encrypted image
- CBC provides better security by hiding patterns in the plaintext

Conclusion

ECB mode is vulnerable to pattern analysis attacks because identical plaintext blocks produce identical ciphertext blocks. CBC mode provides better security by using chaining, making it suitable for encrypting images and other data where pattern preservation is a security concern.

Output Files

- `pic_ecb.bin` - ECB encrypted binary
- `pic_ecb.bmp` - ECB encrypted image (viewable)
- `pic_cbc.bin` - CBC encrypted binary
- `pic_cbc.bmp` - CBC encrypted image (viewable)

Task 3: Encryption Mode – Corrupted Cipher Text (3 Marks)

Objective

Study the error propagation properties of different encryption modes by corrupting a single bit in the ciphertext and observing the decryption results.

Implementation

A plain text file of 64 bytes was created:

```
ABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGHABCDEFGH
```

Encryption Commands:

```
# ECBopenssl enc -aes-128-ecb -e -in plain.txt -out out-ecb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90
# CBCopenssl enc -aes-128-cbc -e -in plain.txt -out out-cbc.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
# CFBopenssl enc -aes-128-cfb -e -in plain.txt -out out-cfb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
# OFBopenssl enc -aes-128-ofb -e -in plain.txt -out out-ofb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
```

Corruption

Using GHex, a single bit in the 30th byte of each encrypted file was flipped to create corrupted versions:

- `corrupted-ecb.bin`
- `corrupted-cbc.bin`

- `corrupted-cfb.bin`
- `corrupted-ofb.bin`

Decryption Commands:

```
# ECBopenssl enc -aes-128-ecb -d -in corrupted-ecb.bin -out decrypted-ecb.txt -K a1b2c3d4e5f60718293a4b5c6d7e8f90
# CBCopenssl enc -aes-128-cbc -d -in corrupted-cbc.bin -out decrypted-cbc.txt \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
# CFBopenssl enc -aes-128-cfb -d -in corrupted-cfb.bin -out decrypted-cfb.txt \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
# OFBopenssl enc -aes-128-ofb -d -in corrupted-ofb.bin -out decrypted-ofb.txt \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
```

Analysis

1. How much information can be recovered?

ECB Mode:

- Only the block containing the corrupted byte is affected
- All other blocks decrypt correctly
- Approximately 1/4 of the data is corrupted (one block out of four 16-byte blocks)

CBC Mode:

- The corrupted block decrypts incorrectly
- The next block also decrypts incorrectly due to error propagation
- Approximately 1/2 of the data is corrupted (two blocks affected)

CFB Mode:

- The corrupted byte affects the corresponding byte in the plaintext
- The error propagates to subsequent bytes until the end of the block
- Approximately 1/4 to 1/2 of the data is corrupted (one block affected, with propagation)

OFB Mode:

- The corrupted byte affects only the corresponding byte in the plaintext
- No error propagation to subsequent blocks
- Approximately 1/64 of the data is corrupted (only one byte affected)

2. Explanation:

- **ECB**: Each block is encrypted independently, so corruption affects only one block
- **CBC**: Uses chaining where each block depends on the previous ciphertext block, causing error propagation
- **CFB**: Stream cipher mode where corruption affects the current byte and propagates within the block
- **OFB**: Stream cipher mode where corruption affects only the specific byte, no propagation

3. Implications:

- **ECB**: Good error isolation but poor security (pattern preservation)
- **CBC**: Better security but error propagation can corrupt multiple blocks
- **CFB**: Moderate error propagation, suitable for real-time applications
- **OFB**: Minimal error propagation, good for applications requiring error tolerance

Output Files

- Encrypted files: `out-ecb.bin`, `out-cbc.bin`, `out-cfb.bin`, `out-ofb.bin`
- Corrupted files: `corrupted-ecb.bin`, `corrupted-cbc.bin`, `corrupted-cfb.bin`, `corrupted-ofb.bin`
- Decrypted files: `decrypted-ecb.txt`, `decrypted-cbc.txt`, `decrypted-cfb.txt`, `decrypted-ofb.txt`

Task 4: Padding (3 Marks)

Objective

Determine which encryption modes require padding and explain why.

Implementation

A plain text file was encrypted using ECB, CBC, CFB, and OFB modes:

Encryption Commands:

```
# ECBopenssl enc -aes-128-ecb -e -in plain.txt -out out-ecb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90
# CBCopenssl enc -aes-128-cbc -e -in plain.txt -out out-cbc.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
# CFBopenssl enc -aes-128-cfb -e -in plain.txt -out out-cfb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
# OFBopenssl enc -aes-128-ofb -e -in plain.txt -out out-ofb.bin \  -K a1b2c3d4e5f60718293a4b5c6d7e8f90 \  -iv 0a1b2c3d4e5f6071
```

Observations

Modes Requiring Padding:

- **ECB**: Requires padding (block cipher mode)
- **CBC**: Requires padding (block cipher mode)

Modes NOT Requiring Padding:

- **CFB**: Does not require padding (stream cipher mode)
- **OFB**: Does not require padding (stream cipher mode)

Explanation

Why ECB and CBC require padding:

- Both ECB and CBC are block cipher modes that operate on fixed-size blocks (16 bytes for AES-128)
- The plaintext must be a multiple of the block size
- If the plaintext length is not a multiple of the block size, padding is added to make it a multiple
- OpenSSL uses PKCS#7 padding by default

Why CFB and OFB do not require padding:

- CFB (Cipher Feedback) and OFB (Output Feedback) are stream cipher modes
- They convert block ciphers into stream ciphers by generating a keystream
- Stream ciphers can encrypt data of any length without padding

- They operate byte-by-byte or bit-by-bit, not on fixed blocks

Output Files

- `out-ecb.bin` - ECB encrypted (with padding)
- `out-cbc.bin` - CBC encrypted (with padding)
- `out-cfb.bin` - CFB encrypted (no padding)
- `out-ofb.bin` - OFB encrypted (no padding)

Task 5: Generating Message Digest (3 Marks)

Objective

Generate message digests using different hash algorithms and compare their outputs.

Implementation

A plain text file was created:

```
Message for digest purpose
```

Commands Used:

```
# MD5openssl dgst -md5 plain.txt > md5.txt
# SHA1openssl dgst -sha1 plain.txt > sha1.txt
# SHA256openssl dgst -sha256 plain.txt > sha256.txt
# SHA512openssl dgst -sha512 plain.txt > sha512.txt
```

Hash Values Generated

MD5:

```
MD5(plain.txt)= e777ebb98c59af9fd4142685c597d154
```

- Output length: 128 bits (32 hex characters)
- Faster but cryptographically broken (collision vulnerabilities)

SHA1:

```
SHA1(plain.txt)= 87ec3acf9ba6542b6447fd8085ac10119491816b
```

- Output length: 160 bits (40 hex characters)
- Deprecated due to security concerns

SHA256:

```
SHA2-256(plain.txt)= 939106512d73323c0164c0b3761441c10fcaa3020ffc3692b6fe68dcf67b1221
```

- Output length: 256 bits (64 hex characters)
- Currently secure and widely used

SHA512:

```
SHA2-512(plain.txt)= 76450fe383aa94624ea48dee3a2f355b247002075f49fd6c2178bb5580dbe49bdf97764099efb8ae6ee1bbc8e8d20348894b4b5b7d504555c67e72a79a8d952
```

- Output length: 512 bits (128 hex characters)
- Strongest security, suitable for high-security applications

Observations

1. **Output Length**: Each algorithm produces a fixed-length output regardless of input size
2. **Avalanche Effect**: Small changes in input produce completely different hash values
3. **Deterministic**: Same input always produces the same hash
4. **One-way Function**: Hash values cannot be reversed to obtain the original input
5. **Security**: SHA256 and SHA512 are currently secure, while MD5 and SHA1 are deprecated

Output Files

- `md5.txt` - MD5 hash

- `sha1.txt` - SHA1 hash
- `sha256.txt` - SHA256 hash
- `sha512.txt` - SHA512 hash

Task 6: Keyed Hash and HMAC (3 Marks)

Objective

Generate HMAC values using different hash algorithms and keys of varying lengths to understand key size requirements.

Implementation

A plain text file was created:

```
Message for digest purpose
```

Commands Used:

The following script was used to test HMAC with different key lengths:

```
#!/usr/bin/env bash
# hmac.sh — Generate HMAC digests with various key lengths

KEYS=(
    "a"                 # 1 byte key
    "abcdefg"           # 7 byte key
    "0123456789abcdef"     # 16 byte key
    "This is a longer key for HMAC testing with more than 64 bytes to test ke
y length handling"  # > 64 bytes
)

# HMAC-MD5
echo "=== HMAC-MD5 ==="
for i in "${!KEYS[@]}"; do
    KEY="${KEYS[$i]}"
    KEY_LEN=${#KEY}
    echo "Key length: $KEY_LEN bytes"
    echo "Key: $KEY"
```

```
    openssl dgst -md5 -hmac "$KEY" plain.txt
    echo ""
done

# HMAC-SHA1
echo "=== HMAC-SHA1 ==="
for i in "${!KEYS[@]}"; do
    KEY="${KEYS[$i]}"
    KEY_LEN=${#KEY}
    echo "Key length: $KEY_LEN bytes"
    echo "Key: $KEY"
    openssl dgst -sha1 -hmac "$KEY" plain.txt
    echo ""
done

# HMAC-SHA256
echo "=== HMAC-SHA256 ==="
for i in "${!KEYS[@]}"; do
    KEY="${KEYS[$i]}"
    KEY_LEN=${#KEY}
    echo "Key length: $KEY_LEN bytes"
    echo "Key: $KEY"
    openssl dgst -sha256 -hmac "$KEY" plain.txt
    echo ""
done
```

HMAC Results

HMAC-MD5:

- Key: “a” (1 byte) → `fdef49ee857fd2a6408b394faaf9d761`
- Key: “abcdefg” (7 bytes) → `16f0cfb70d107a98165878d892f30580`
- Key: “0123456789abcdef” (16 bytes) → `a71d32856507edffe52c09b842ee912d`
- Key: “This is a longer key...” (89 bytes) → `b9f2eed319b6a10701fd9db515d14a12`

HMAC-SHA1:

- Key: “a” (1 byte) → `fba55873d89bdaedfef135f0db2ba6192229fbe3`
- Key: “abcdefg” (7 bytes) → `398e52807937514239874edce6b1315df732a7fe`

- Key: “0123456789abcdef” (16 bytes) → `1af5cf1babe3a3f0ee150db539f56622154fddca`
- Key: “This is a longer key...” (89 bytes) → `dadd43f4492b8bfb81c2dcdb95868a9f9e4d50ee`

HMAC-SHA256:

- Key: “a” (1 byte) → `9450cfe629e371c30f171d1b08e1df4176ce770da8eccf9eef23225f51edd73a`
- Key: “abcdefg” (7 bytes) → `98eb0d1d223e721458eb908216e75e3b07ae721ab34c426e4bda322f628d3211`
- Key: “0123456789abcdef” (16 bytes) → `c0f75f0d9e7b512abfd0c4119026bd99f0ed2e11c98d63176bdacc25ff28c070`
- Key: “This is a longer key...” (89 bytes) → `08f1cfebb9b987f2c35f68f5ab6db481f31c9c40176fcb25ce1b765606f8b4e4`

Key Size Analysis

Question: Do we have to use a key with a fixed size in HMAC?

Answer: No, HMAC does not require a fixed key size. The key can be of any length.

Explanation:

- HMAC accepts keys of any length
- If the key is shorter than the hash function’s block size (64 bytes for MD5/SHA1/SHA256), it is padded with zeros
- If the key is longer than the block size, it is first hashed to reduce it to the hash output length
- This ensures the key is always processed consistently regardless of its original length
- The HMAC algorithm handles key normalization internally, making it flexible for keys of any size

Security Recommendation:

- While any key length works, longer keys generally provide better security
- Keys should be at least as long as the hash output length for optimal security
- For SHA256, a key of at least 32 bytes (256 bits) is recommended

---

Task 7: Hash Function Properties (3 Marks)

Objective

Demonstrate the avalanche effect of hash functions by flipping a single bit in the input and comparing the hash outputs.

Implementation

Original File (`plain.txt`):

```
Message for hash testing
Author: Nixon Deb Antu
```

Commands Used:

```
# Generate hash for original fileopenssl dgst -md5 plain.txt > plain_md5.txt
openssl dgst -sha256 plain.txt > plain_sha256.txt
# Generate hash for modified file (one bit flipped)openssl dgst -md5 modifi
ed.txt > modified_md5.txt
openssl dgst -sha256 modified.txt > modified_sha256.txt
```

Hash Values

MD5 Hashes:

Original file:

```
MD5(plain.txt)= e635e11d8d684e7f1a3db2e349a873a4
```

Modified file (one bit flipped):

```
MD5(modified.txt)= 54ed6cd13b13b327784b183bc42440a0
```

SHA256 Hashes:

Original file:

```
SHA2-256(plain.txt)= 47c7c246016a6a12c4218ac150bb71c4eb177bf303e6
c30563b61a54bb56dac2
```

Modified file (one bit flipped):

```
SHA2-256(modified.txt)= ec0fc6cf8ead84f0e38df808ac17dd47638e1e182
6ddf52d8ecc6aad3ee8478c
```

Observations

1. **Complete Change**: Flipping a single bit in the input file resulted in completely different hash values for both MD5 and SHA256
2. **Avalanche Effect**: The hash functions demonstrate the avalanche effect - a small change in input causes approximately 50% of the output bits to change
3. **No Similarity**: There is no visible similarity between H1 and H2, confirming the one-way property of hash functions
4. **Deterministic**: The same input always produces the same hash, but any change produces a completely different hash

Bonus: Bit Comparison Program (2 Marks)

A Python program was written to count the number of matching bits between two hash values:

```
def hex_to_binary(hex_string):
    return bin(int(hex_string, 16))[2:].zfill(len(hex_string) * 4)
def count_same_bits(hex1, hex2):
    if len(hex1) != len(hex2):
        raise ValueError("Both hash values must have the same length.")
    bin1 = hex_to_binary(hex1)
    bin2 = hex_to_binary(hex2)
    same_bits = sum(1 for a, b in zip(bin1, bin2) if a == b)
    total_bits = len(bin1)
    return same_bits, total_bits
```

Results:

MD5:

- Same bits: 63 out of 128
- Percentage: 49.22%

SHA256:

- Same bits: 123 out of 256
- Percentage: 48.05%

Analysis

The bit comparison confirms the avalanche effect:

- Approximately 50% of the bits differ between H1 and H2
- This is expected behavior for cryptographic hash functions
- The slight variation from exactly 50% is normal due to the deterministic nature of the hash algorithm
- This property ensures that even minor changes to input data are easily detectable

Output Files

- `plain_md5.txt` - MD5 hash of original file
- `plain_sha256.txt` - SHA256 hash of original file
- `modified_md5.txt` - MD5 hash of modified file
- `modified_sha256.txt` - SHA256 hash of modified file
- `compare.py` - Python program for bit comparison