Information and Network Security

Course Code: SWE 430

Submitted By:

Ridwanur Rashid Siam
Registration No: 2019831060
Session: 2019-20
Department of Software Engineering, IICT, SUST.

Submitted To:

Partha Pratim Paul
Lecturer,
Institute of Information and Communication Technology, SUST.

Lab 3: Symmetric encryption & hashing

Task – 1: AES encryption using different modes

CBC:

openssl enc -aes-128-cbc -e -in test.txt -out encryptedCBC.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708091011121314151 6

openssl enc -aes-128-cbc -d -in encryptedCBC.txt -out decryptedCBC.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708091011121314151 6

CFB:

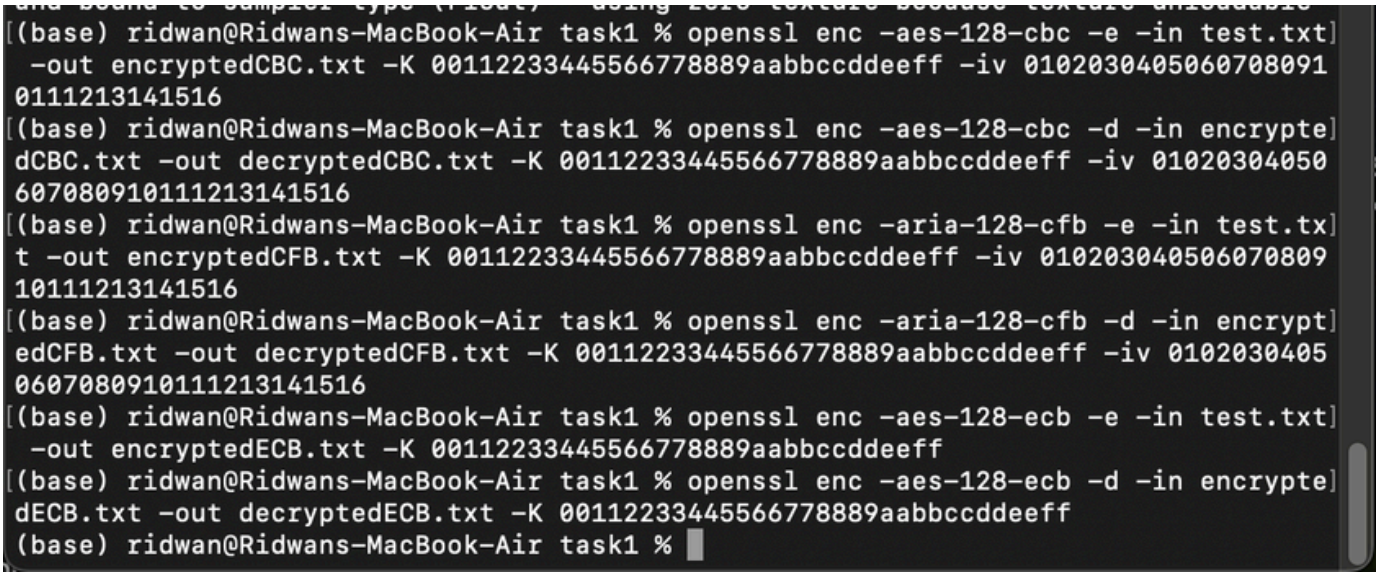
openssl enc -aria-128-cfb -e -in test.txt -out encryptedCFB.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708091011121314151 6

openssl enc -aria-128-cfb -d -in encryptedCFB.txt -out decryptedCFB.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708091011121314151 6

ECB:

openssl enc -aes-128-ecb -e -in test.txt -out encryptedECB.txt -K 00112233445566778889aabbccddeeff

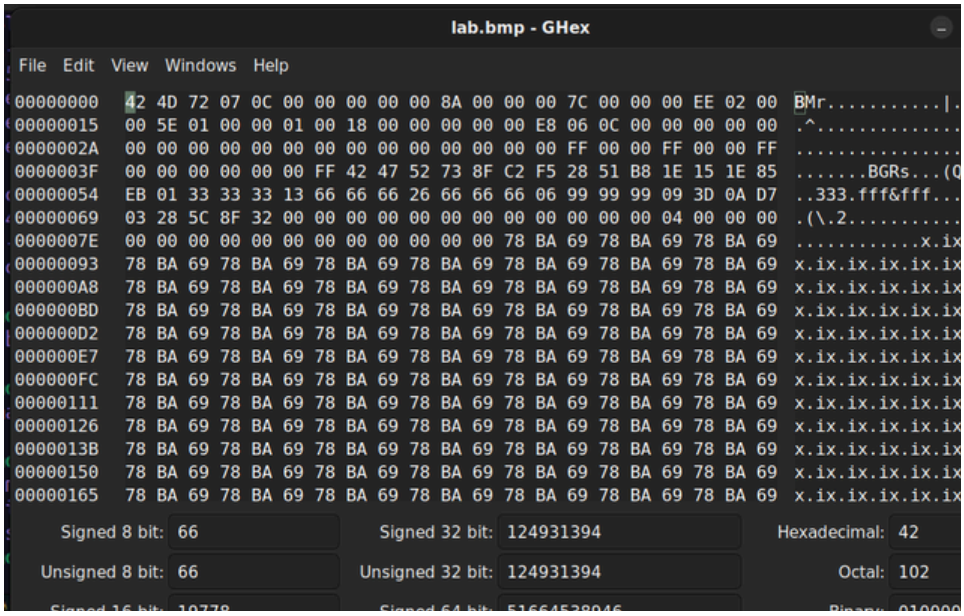
openssl enc -aes-128-ecb -d -in encryptedECB.txt -out decryptedECB.txt -K 00112233445566778889aabbccddeeff



Task – 2: Encryption mode - ECB vs CBC

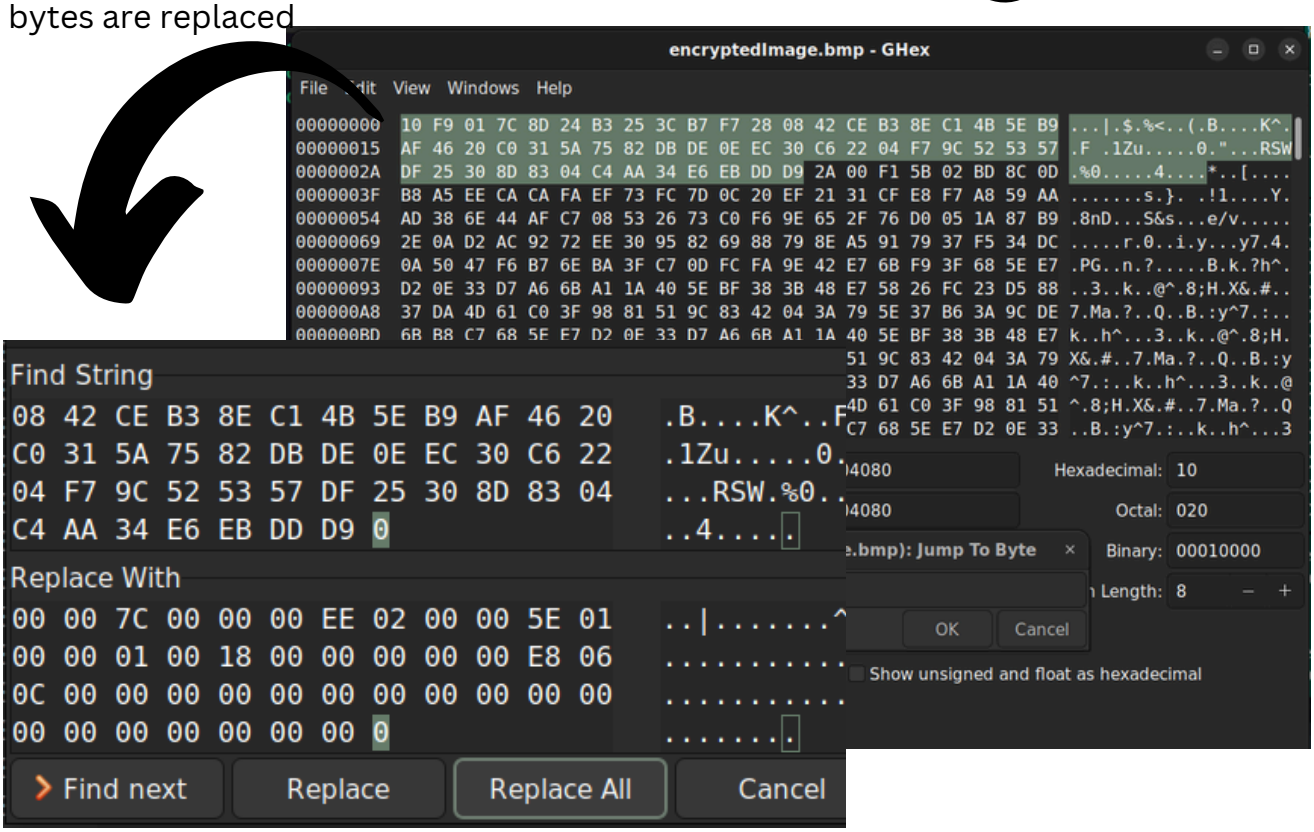
Encryption with ECB Mode

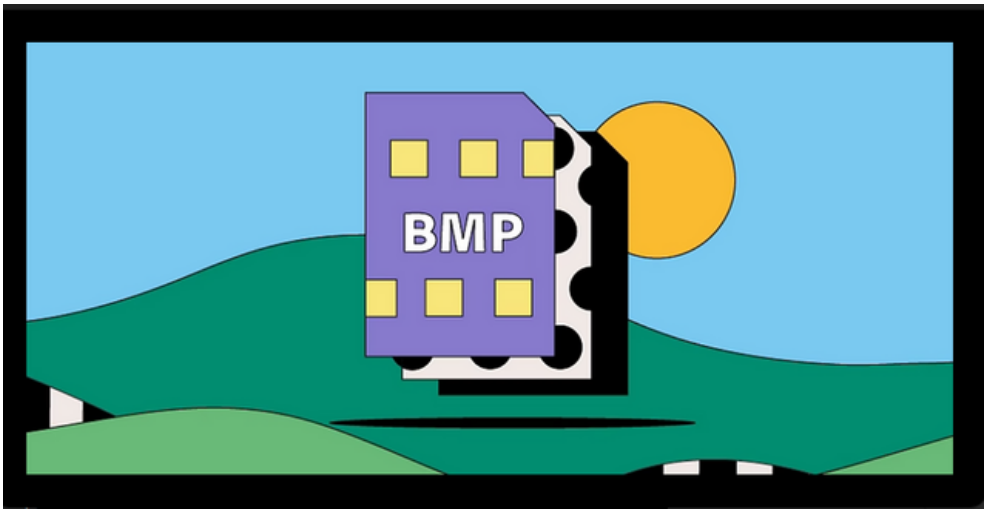
openssl enc -aes-128-ecb -e -in lab2.bmp -out encryptedPic.bmp -K 00112233445566778889aabbccddeeff



First 54 bytes are copied

Selected bytes are replaced



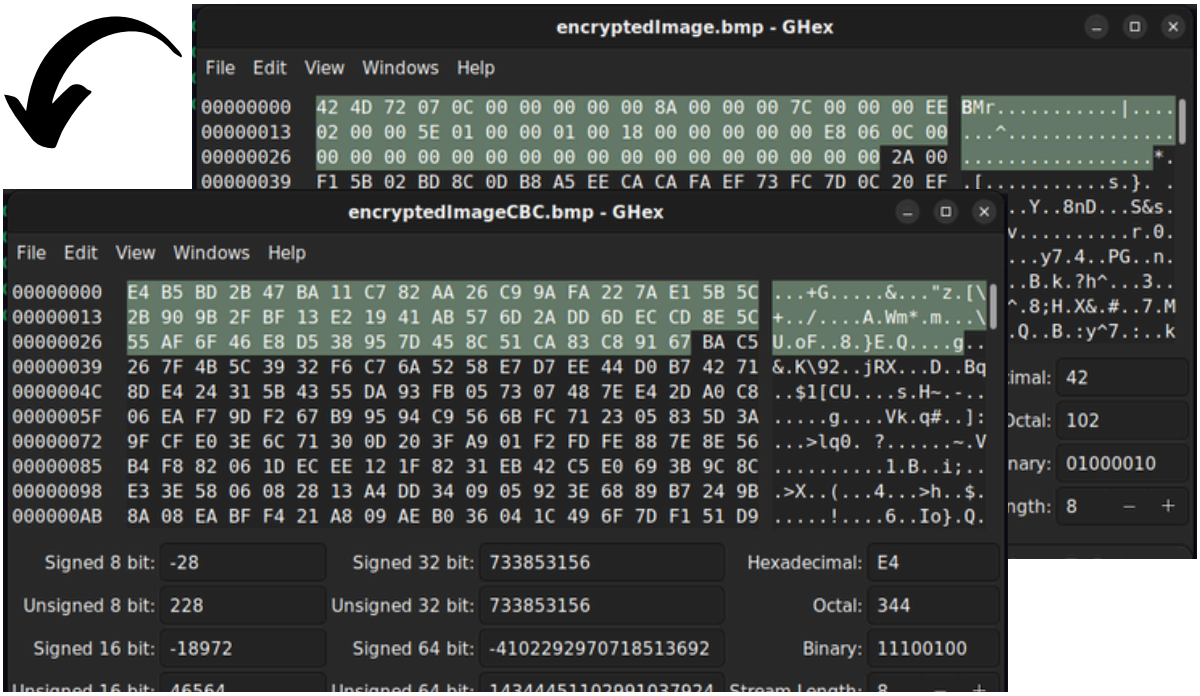


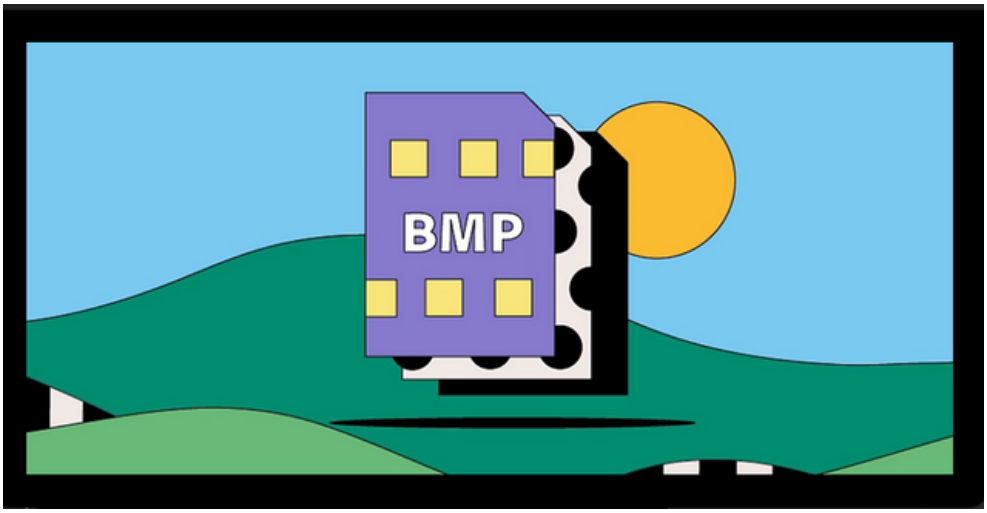
Initial Phase

After ECB encryption

Encryption with CBC Mode

openssl enc -aes-128-cbc -e -in lab.bmp -out encryptedImageCBC.bmp -K 00112233445566778889aabbccddeeff -iv 01020304050607080102030405060708



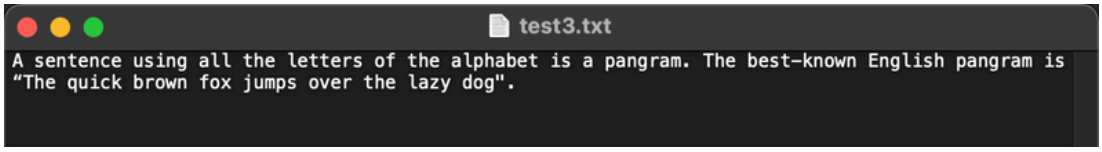


After CBC Mode Conversion

Task – 3: Encryption mode – corrupted cipher text

In the test3 .txt file, we added a text:

A sentence using all the letters of the alphabet is a pangram. The best-known English pangram is “The quick brown fox jumps over the lazy dog".



Encryption with CBC:

openssl enc -aes-128-cbc -e -in test3.txt -out text2Encrypt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

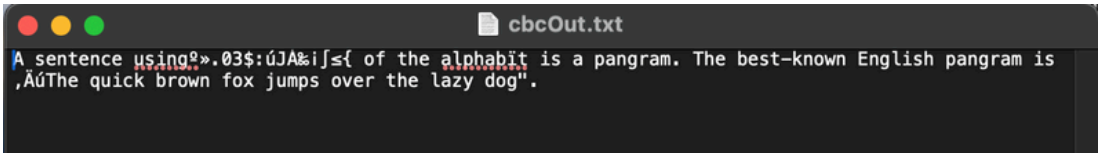
After encrypting the test3.txt file, we opened the encrypted file named text2Encrypt.txt with **ghex text2Encrypt.txt.** Then we changed the **30th byte** and got the output.

Decryption with CBC:

openssl enc -aes-128-cbc -d -in text2Encrypt.txt -out cbcOut.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Output:

A sentence usingº».03$:úJÁ ‰ ¡ʃ≤{ of the alphabït is a pangram. The best-known English pangram is ‚ÄúThe quick brown fox jumps over the lazy dog".



Encryption with ECB:

openssl enc -aes-128-ecb -e -in test3.txt -out test3ECBEncrypt.txt -K 00112233445566778889aabbccddeeff

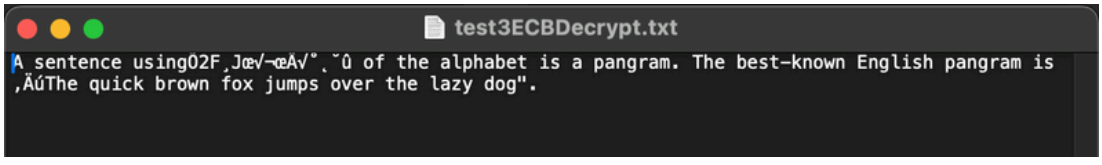
After encrypting the test3.txt file, we opened the encrypted file named test3ECBEncrypt.txt with **ghex test3ECBEncrypt.txt**. Then we changed the 30th byte from E to 7 and got the output.

Decryption with ECB:

openssl enc -aes-128-ecb -d -in test3ECBEncrypt.txt -out test3ECBDecrypt.txt -K 00112233445566778889aabbccddeeff

Output:

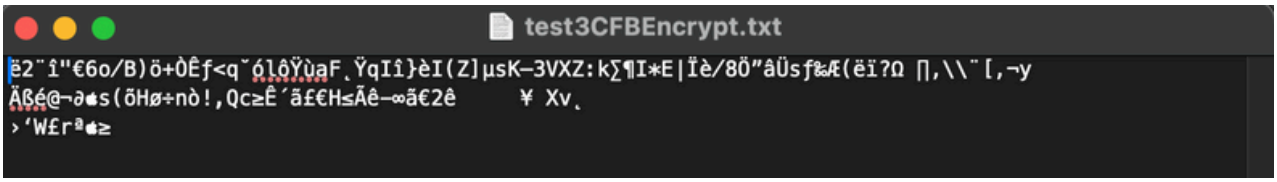
A sentence usingÕ2F¸Jœ√¬œÂ√˚¸˘û of the alphabet is a pangram. The best-known English pangram is ‚ÄúThe quick brown fox jumps over the lazy dog".



Encryption with CFB:

openssl enc -aria-128-cfb -e -in test3.txt -out test3CFBEncrypt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

After encrypting the test3.txt file, we opened the encrypted file named test3CFBEncrypt.txt with **ghex test3CFBEncrypt.txt**. Then we changed the 30th byte from D to 9 and got the output.



Decryption with CFB:

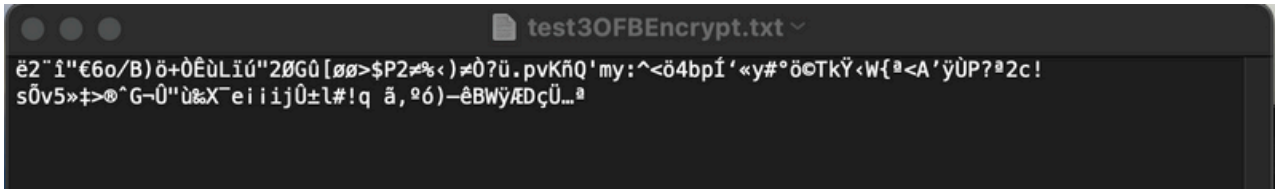
openssl enc -aria-128-cfb -d -in test3CFBEncrypt.txt -out test3CFBDecrypt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Output:
A sentence using all the lette2s ’nÕ¶ˆ\ûh6¶,–‰ is a pangram. The best-known English pangram is ‚ÄúThe quick brown fox jumps over the lazy dog".

Encryption with OFB:

openssl enc -aria-128-ofb -e -in test3.txt -out test3OFBEncrypt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

After encrypting the test3.txt file, we opened the encrypted file named test3OFBEncrypt.txt with ghex test3OFBEncrypt.txt. Then we changed the 30th byte from 2 to 9 and got the output.

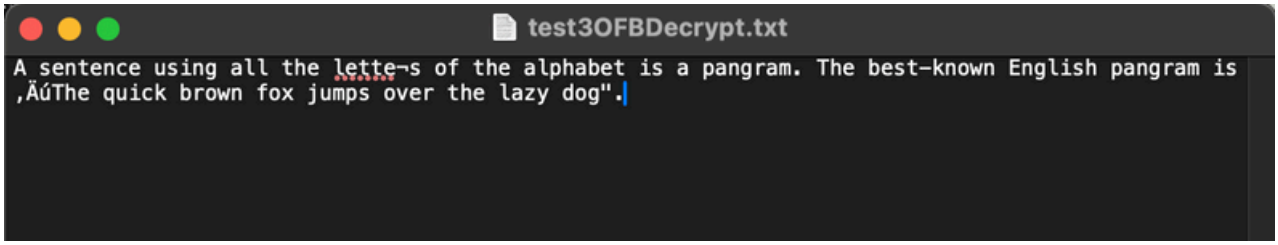


Decryption with OFB:

openssl enc -aria-128-ofb -d -in test3OFBEncrypt.txt -out test3OFBDecrypt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Output:

A sentence using all the lette¬s of the alphabet is a pangram. The best-known English pangram is ‚ÄúThe quick brown fox jumps over the lazy dog".



Question Answer:

How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Answer this question before you conduct this task, and then find out whether your answer is correct or wrong after you finish this task. Explain why? What are the differences?

I**n ECB mode**, each block of plaintext is encrypted independently. This means that if one block is corrupted, it will only affect that particular block and not the others. Therefore, in ECB mode, you could potentially recover all the information from the unaffected blocks. In my case, I found 16byte of data to be corrupted consecutively. But sometimes, if a block containing crucial information is corrupted, the data in that block will be lost, potentially leading to partial or incomplete recovery of the file, which happened with my case, where another 2 bytes of data wasn't recovered.

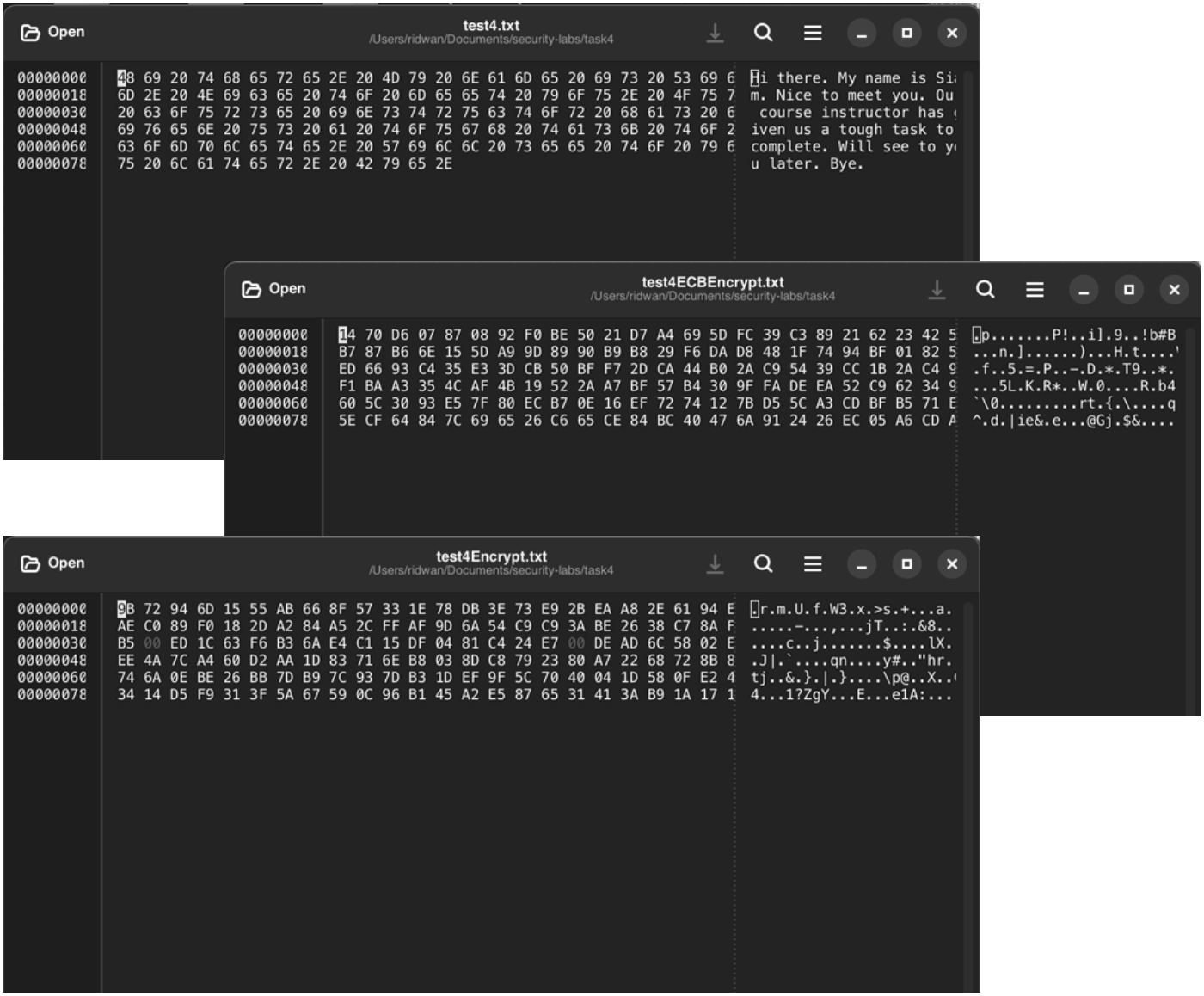
In CBC mode, each block of plaintext is XORed with the previous ciphertext block before encryption. If a block in the middle of the file is corrupted, it will affect the decryption of all subsequent blocks. However, the corruption in one block will only propagate up to the next block; after that, the decryption will become correct again. So, you could potentially recover all the information up to the point of corruption. In my case, I found 32 bytes of data to be corrupted.

In CFB mode, ciphertext is fed back into the encryption algorithm to create the next block of ciphertext. If a block in the middle of the file is corrupted, it will affect the decryption of all subsequent blocks similar to CBC mode. However, like CBC mode, the corruption in one block will only propagate up to the next block. Therefore, you could potentially recover all the information up to the point of corruption. In my case, I found 32 byte of data to be corrupted sequentially.

In OFB mode, the encryption of the previous block is XORed with the plaintext to create the ciphertext. Corruption in one block will only affect the decryption of that block and will not propagate to subsequent blocks. So, similar to ECB mode, you could potentially recover all the information from the unaffected blocks. In my case, I found 16 bytes of data to be corrupted in a serial.

Task – 4: Padding

- **ECB, CBC**: These modes typically require padding because they process data in fixed-size blocks. If the plaintext size isn't a multiple of the block size, incomplete blocks will occur, leading to errors during decryption.



Generating a encrypted file with ECB mode

openssl enc -aes-128-ecb -e -in test4.txt -out test4ECBEncrypt.txt -K 00112233445566778889aabbccddeeff

Generating a encrypted file with CBC mode

openssl enc -aes-128-cbc -e -in test4.txt -out test4Encrypt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

In the above section, we can see that in the ECB encrypted file, the number of bytes have increased compared to the main test4.txt file. This indicates padding was included in it.

- **CFB, OFB:** These modes operate on a bit-by-bit or byte-by-byte basis using the cipher output as a pseudo-random stream. They do not inherently require padding as they can process any size of plaintext without needing complete blocks.

Generating a encrypted file with CFB mode

openssl enc -aria-128-cfb -e -in test4.txt -out test4CFBEncrypt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Generating a encrypted file with OFB mode

openssl enc -aria-128-ofb -e -in test4.txt -out test4OFBEncrypt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

In the above section, we can see that in the ECB encrypted file, the number of bytes haven’t changed compared to the main test4.txt file. This indicates padding was not included in it.

Task – 5: Generating message digest

A test5.txt file has been created for encryption.

openssl dgst -md5 test5.txt
MD5(test5.txt)= 56bd5e088857152c415fd6ead4a1781c

```
(base) ridwan@Ridwans-MacBook-Air task5 % openssl dgst -md5 test5.txt
MD5(test5.txt)= 56bd5e088857152c415fd6ead4a1781c
```

openssl dgst -sha1 test5.txt
SHA1(test5.txt)= 1f7960bd63c5c0a406e1a7b50f514f164fb55a3f

```
(base) ridwan@Ridwans-MacBook-Air task5 % openssl dgst -sha1 test5.txt
SHA1(test5.txt)= 1f7960bd63c5c0a406e1a7b50f514f164fb55a3f
```

openssl dgst -sha256 test5.txt
SHA2-256(test5.txt)=
eb15cbe2c7b133f3fb424bba67c0469db1ea21424b3dbd8328774789a78ef91f

```
(base) ridwan@Ridwans-MacBook-Air task5 % openssl dgst -sha256 test5.txt
SHA2-256(test5.txt)= eb15cbe2c7b133f3fb424bba67c0469db1ea21424b3dbd8328774789a78ef
91f
```

Task – 6: Keyed hash and HMAC

The commands and their corresponding keyed hash using HMAC-MD5, HMAC-SHA256, and HMAC-SHA1 :

openssl dgst -md5 -hmac “siam_valo_chele” test6.txt
HMAC-MD5(test6.txt)= 8a846e0ad07f2363189284023b8b76b6

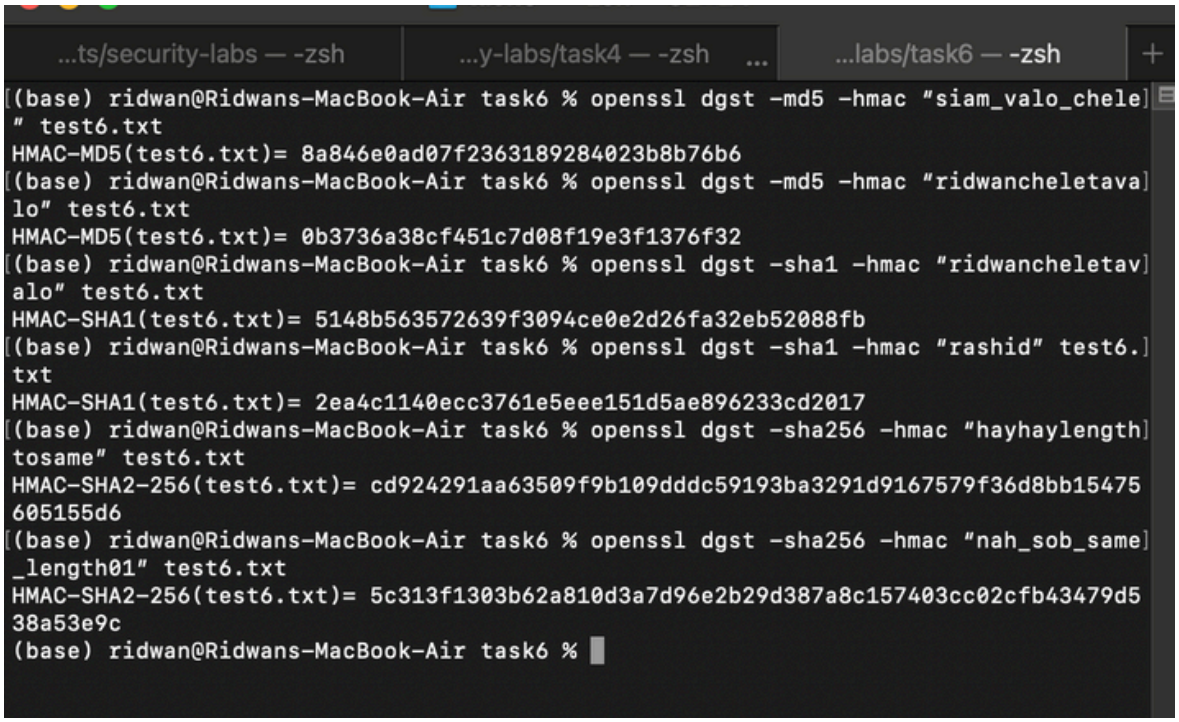
openssl dgst -md5 -hmac “ridwancheletavalo” test6.txt
HMAC-MD5(test6.txt)= 0b3736a38cf451c7d08f19e3f1376f32

openssl dgst -sha1 -hmac “ridwancheletavalo” test6.txt
HMAC-SHA1(test6.txt)= 5148b563572639f3094ce0e2d26fa32eb52088fb

openssl dgst -sha1 -hmac “rashid” test6.txt
HMAC-SHA1(test6.txt)= 2ea4c1140ecc3761e5eee151d5ae896233cd2017

openssl dgst -sha256 -hmac “hayhaylengthtosame” test6.txt
HMAC-SHA2-256(test6.txt)=
cd924291aa63509f9b109dddc59193ba3291d9167579f36d8bb15475605155d6

openssl dgst -sha256 -hmac “nah_sob_same_length01” test6.txt
HMAC-SHA2-256(test6.txt)=
5c313f1303b62a810d3a7d96e2b29d387a8c157403cc02cfb43479d538a53e9c



Question: Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?

HMAC does not require a key with a fixed size. It can accept keys of any size. However, the security of HMAC depends on the strength of the key. Using a longer key generally increases security, but it's important to choose a key length that is appropriate for the specific cryptographic algorithm being used (e.g., MD5, SHA-1, SHA-256). For example, using a key longer than the block size of the hash function does not necessarily increase security further.

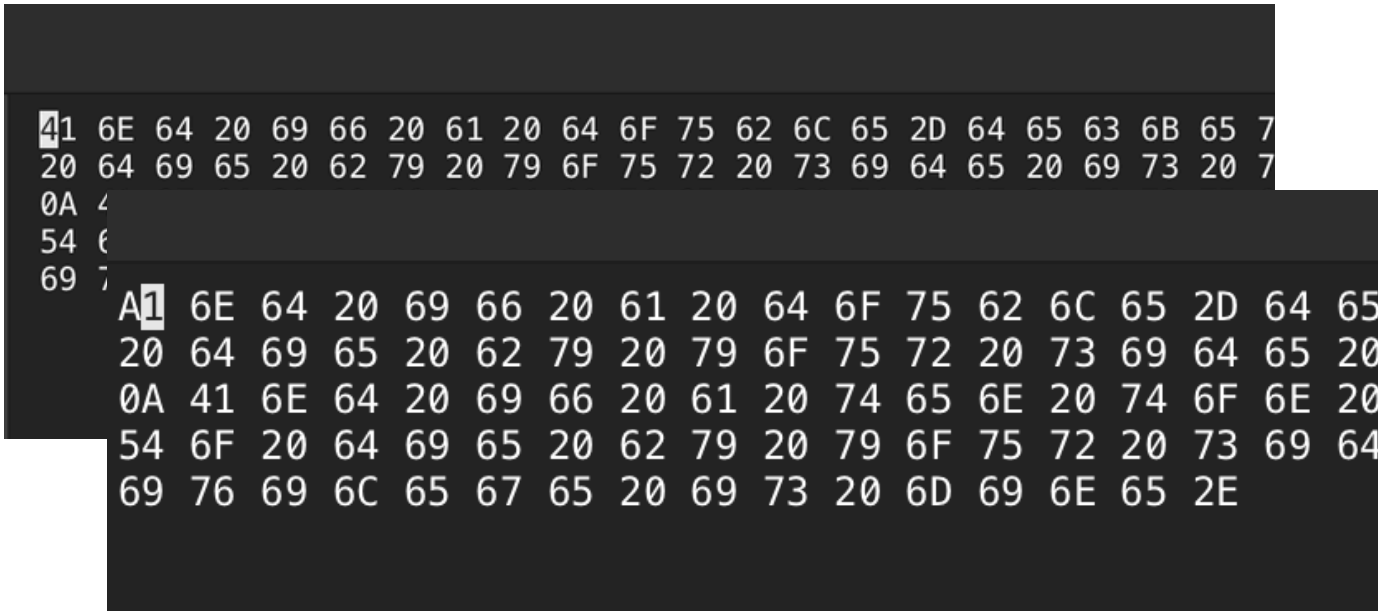
Task – 7: Keyed hash and HMAC

First of all, a new text file named test7.txt has been created. Then several hash values using two of the algorithms are generated.

openssl dgst -md5 test7.txt
MD5(test7.txt)= 54f126280be51ff993c11424930c8b52

```
(base) ridwan@Ridwans-MacBook-Air task7 % openssl dgst -md5 test7.txt
MD5(test7.txt)= 54f126280be51ff993c11424930c8b52
(base) ridwan@Ridwans-MacBook-Air task7 % ghex test7.txt
```

When we flipped a bit of the input file, the hash value has been changed.



openssl dgst -md5 test7.txt
MD5(test7.txt)= 681f9aede2f03be7940fc5f9028c0ac6

```
(base) ridwan@Ridwans-MacBook-Air task7 % openssl dgst -md5 test7.txt
MD5(test7.txt)= 681f9aede2f03be7940fc5f9028c0ac6
(base) ridwan@Ridwans-MacBook-Air task7 % ghex test7.txt
```

Likewisely, it gets changed even if we use different algorithms for hash value.

Before flipping bit:

openssl dgst -sha256 test7.txt
SHA2-256(test7.txt)= aeae96038f7b892bfca07225e624bf2b6e9772675477c0aa4108b1032436ee90

After flipping bit:

openssl dgst -sha256 test7.txt
SHA2-256(test7.txt)= a64f1d01ea7182224dfaf31a624ab38281711e8c10442d663718904d36c1fdb4

```
(base) ridwan@Ridwans-MacBook-Air task7 % openssl dgst -sha256 test7.txt
SHA2-256(test7.txt)= aeae96038f7b892bfca07225e624bf2b6e9772675477c0aa4108b1
e90
```

```
(base) ridwan@Ridwans-MacBook-Air task7 % openssl dgst -sha256 test7.txt
SHA2-256(test7.txt)= a64f1d01ea7182224dfaf31a624ab38281711e8c10442d663718904d36c1f
db4
```

One-way hash functions are designed to be highly sensitive to changes in the input. Modifying even a single bit in the message significantly alters the internal calculations, resulting in a completely different hash value.

Code for comparing bits difference between two hexadecimal hashed strings are given below:

```
#include <bits/stdc++.h>
int count_matching_bits(const string& H1, const string& H2) {
  string bin_H1 = "", bin_H2 = "";
  for(char c : H1) {
    switch(c) {
      case '0': bin_H1 += "0000"; break;
      case '1': bin_H1 += "0001"; break;
      case '2': bin_H1 += "0010"; break;
      case '3': bin_H1 += "0011"; break;
      case '4': bin_H1 += "0100"; break;
      case '5': bin_H1 += "0101"; break;
      case '6': bin_H1 += "0110"; break;
```

```
    case '7': bin_H1 += "0111"; break;
    case '8': bin_H1 += "1000"; break;
    case '9': bin_H1 += "1001"; break;
    case 'a': case 'A': bin_H1 += "1010"; break;
    case 'b': case 'B': bin_H1 += "1011"; break;
    case 'c': case 'C': bin_H1 += "1100"; break;
    case 'd': case 'D': bin_H1 += "1101"; break;
    case 'e': case 'E': bin_H1 += "1110"; break;
    case 'f': case 'F': bin_H1 += "1111"; break;
    default: cout << "Invalid hexadecimal character: " << c << endl;
   }
  }
  for(char c : H2) {
   switch(c) {
    case '0': bin_H2 += "0000"; break;
    case '1': bin_H2 += "0001"; break;
    case '2': bin_H2 += "0010"; break;
    case '3': bin_H2 += "0011"; break;
    case '4': bin_H2 += "0100"; break;
    case '5': bin_H2 += "0101"; break;
    case '6': bin_H2 += "0110"; break;
    case '7': bin_H2 += "0111"; break;
    case '8': bin_H2 += "1000"; break;
    case '9': bin_H2 += "1001"; break;
    case 'a': case 'A': bin_H2 += "1010"; break;
    case 'b': case 'B': bin_H2 += "1011"; break;
    case 'c': case 'C': bin_H2 += "1100"; break;
    case 'd': case 'D': bin_H2 += "1101"; break;
    case 'e': case 'E': bin_H2 += "1110"; break;
    case 'f': case 'F': bin_H2 += "1111"; break;
    default: cout << "Invalid hexadecimal character: " << c << endl;
   }
  }

  for(size_t i = 0; i < bin_H1.size() && i < bin_H2.size(); ++i) {
   if(bin_H1[i] == bin_H2[i]) {
    matching_bits++;
   }
  }

  return matching_bits;
}

int main() {
  string h1, h2;
  cin >> h1 >> h2;

  int matching_bits = count_matching_bits(h1, h2);
  cout << "Number of matching bits: " << matching_bits << endl;

  return 0;
}
```