

Information Security - Assignment #1

Course Instructor: Usama Antulay

Name: Muhammad Talha Yousif

Roll Number: 22k5146

Section: BCS-7L



Department of Computer Science

September 18, 2025

Contents

Introduction	2
1 Frequency Analysis	3
1.1 Description	3
1.2 Steps and Screenshots	3
1.3 Observations & Explanations	4
1.4 Code Snippets	4
2 Encryption using Different Ciphers and Modes	5
2.1 Description	5
2.2 Steps and Screenshots	5
2.3 Observations & Explanations	6
2.4 Code Snippets	6
3 Encryption Mode – ECB vs. CBC	7
3.1 Description	7
3.2 Steps and Screenshots	7
3.3 Observations & Explanations	8
3.4 Code Snippets	8
4 Padding	9
4.1 Description	9
4.2 Steps and Screenshots	9
4.3 Observations & Explanations	10
4.4 Code Snippets	10
5 Error Propagation – Corrupted Cipher Text	11
6 Initial Vector (IV) and Common Mistakes	13
7 Programming using the Crypto Library	15
Conclusion	17

Introduction

This report presents the detailed solutions, observations, and explanations for Assignment #1 of the Information Security course. Each task includes screenshots, explanations of observations, and relevant code snippets with commentary.

Chapter 1

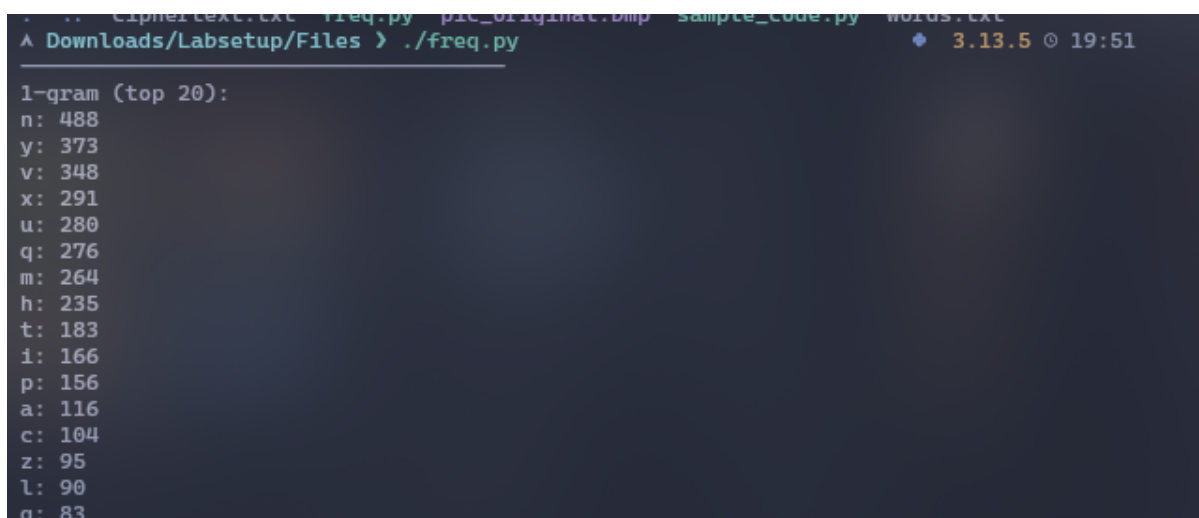
Frequency Analysis

1.1 Description

The objective of this lab is to decrypt a monoalphabetic substitution cipher using frequency analysis. The provided ciphertext (`ciphertext.txt`) was generated by randomly permuting the English alphabet and encrypting an article after removing punctuation and numbers.

1.2 Steps and Screenshots

1. Run the provided Python script to generate n-gram frequencies:
`./freq.py > freq_output.txt`
2. Inspect the output to identify the most frequent letters and patterns.
3. Apply initial substitutions using `tr` or a custom Python script.
4. Iteratively refine mappings until meaningful English words appear.



```
. .. ciphertext.txt freq.py pic_original.bmp sample_code.py words.txt
^ Downloads/Labsetup/Files > ./freq.py 3.13.5 © 19:51

1-gram (top 20):
n: 488
y: 373
v: 348
x: 291
u: 280
q: 276
m: 264
h: 235
t: 183
i: 166
p: 156
a: 116
c: 104
z: 95
l: 90
g: 83
```

```

^ Downloads/Labsetup/Files > tr 'ytn' 'THE' < ciphertext.txt > step1.txt

^ Downloads/Labsetup/Files > cat step1.txt
THE xqavhq Tzhu xu qzupvd lHmaH qEEcq vgxzT hmrHT vbTEh THmq ixur qThvurE
vLvhpq Thme THE gvrrEh bEEiq lmsE v uxuvrEuvhmvu Txx

THE vLvhpq hvaE lvq gxxsEupEp gd THE pEcmqE xb HvfhEd lEmuqTEmu vT mTq xzTqET
vup THE veevhEuT mceixqmxu xb Hmq bmic axcevud vT THE Eup vup mT lvq qHveEp gd
THE EcEhrEuaE xb cETxx TmcEq ze qivasrxlu eximTmaq vhcavupd vaTmfmqc vup
v uvTmxuvi axufEhqvTmxu vq ghmEb vup cvp vq v bEfEh phEvc vgxzT LHETHEh THEhE
xZrHT Tx qE v ehEqmpEuT lmubhEd THE qEvqxu pmpuT ozqT qEEc EkThv ixur mT lvq
EkThv ixur qEavzqE THE xqavhq lEhE cxfEp Tx THE bmhqT lEEsEup mu cvhaH Tx
vfxmp axubImaTmur lmTH THE aixqmur aEhEcud xb THE lmuTEh xidcemaq THvusq
edExuraHvur

```

1.3 Observations & Explanations

- **Frequency Results:** 1-gram top letters: n (488), y (373), v (348). 2-gram: yt (115), tn (89), mu (74). 3-gram: ytn (78), vup (30), mur (20).
- **Mapping Deduction:** English statistics indicate e, t, and h dominate. The trigram ytn aligns with the, yielding:

$$y \rightarrow t, \quad t \rightarrow h, \quad n \rightarrow e$$

- Spaces in the ciphertext simplified word-boundary recognition, making further guesses (e.g., v→a/i) easier.

1.4 Code Snippets

Initial substitution with `tr`:

```
tr 'ytn' 'THE' < ciphertext.txt > step1.txt
```

Iterative mapping script (`applymap.py`):

This process was repeated until most of the plaintext was recovered, demonstrating the weakness of monoalphabetic ciphers against frequency analysis.

Chapter 2

Encryption using Different Ciphers and Modes

2.1 Description

The objective of this task is to experiment with multiple symmetric encryption algorithms and modes using the openssl enc command-line tool. A sample plaintext file (plain.txt) was encrypted and decrypted with different cipher types to observe differences in ciphertext patterns and encryption behavior.

2.2 Steps and Screenshots

1. Created a sample plaintext file:

```
echo "Confidential_message_for_encryption_lab." > plain.txt
```

2. Encrypted the file using three different ciphers:

```
openssl enc -aes-128-cbc -e -in plain.txt -out aes_cbc.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

```
openssl enc -aes-128-cfb -e -in plain.txt -out aes_cfb.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

```
openssl enc -bf-cbc -e -in plain.txt -out bf_cbc.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

3. Decrypted each file to verify correctness:

```
openssl enc -aes-128-cbc -d -in aes_cbc.bin -out decrypted_aes.txt
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

4. Confirmed that all decrypted outputs matched the original plaintext.

```

^ Downloads/Labsetup/Files > cat plain.txt
Confidential _ message _ for _ encryption _ lab . . . txt
^ Downloads/Labsetup/Files > openssl enc -aes-128-cbc -e -in plain.txt -out aes_cbc.bin
\
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708

hex string is too short, padding with zero bytes to length
^ Downloads/Labsetup/Files > cat aes_cbc.bin
***[]N*,**W**T*W.b$*J*BN*|*b*k***j*|f*uxk**UZ"G*K***J!***0***~*9*P*
^ Downloads/Labsetup/Files > |

```

2.3 Observations & Explanations

- Cipher Differences: The ciphertext sizes were identical for all algorithms because of padding, but the actual byte patterns differed significantly due to different block operations.
- Mode Impact: CBC (Cipher Block Chaining) introduces diffusion between blocks, so a single-bit change in plaintext changes multiple ciphertext blocks. CFB (Cipher Feedback) operates like a stream cipher and allows encryption of data smaller than the block size without padding.
- Key/IV Sensitivity: Changing even one hex digit in the key or IV resulted in completely different ciphertext, demonstrating the avalanche effect.

2.4 Code Snippets

Key generation and printing for verification:

```

openssl enc -aes-128-cbc -P \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708

```

Example of decrypting with the Blowfish cipher:

```

openssl enc -bf-cbc -d -in bf_cbc.bin -out decrypted_bf.txt \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708

```

These experiments show how different cipher algorithms and modes affect the structure and security properties of encrypted data.

Chapter 3

Encryption Mode – ECB vs. CBC

3.1 Description

This task demonstrates the difference between ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes by encrypting a bitmap image (pic_original.bmp) and analyzing the visual patterns of the encrypted output. The experiment highlights how encryption mode impacts the ability to infer information from encrypted data.

3.2 Steps and Screenshots

1. Encrypted the BMP image using AES in both ECB and CBC modes:

```
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.bin \
-K 00112233445566778889aabbccddeeff
```

```
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

2. Extracted the header from the original BMP and combined it with the encrypted data to create valid image files:

```
head -c 54 pic_original.bmp > header
tail -c +55 pic_ecb.bin > ecb_body
cat header ecb_body > pic_ecb_view.bmp
```

```
tail -c +55 pic_cbc.bin > cbc_body
cat header cbc_body > pic_cbc_view.bmp
```

3. Viewed the resulting files using an image viewer (eog):

```
eog pic_ecb_view.bmp
eog pic_cbc_view.bmp
```

4. Repeated the same procedure with a second image of choice to validate results on a different dataset.

3.3 Observations & Explanations

- ECB Mode: The encrypted image still revealed visible outlines and shapes of the original picture, despite encryption. This occurs because ECB encrypts identical plaintext blocks into identical ciphertext blocks, causing patterns in the image to remain visible.
- CBC Mode: The encrypted image appeared as random noise with no discernible structure. CBC introduces block chaining and randomness through the IV, preventing repeating patterns from leaking.
- Second Image Test: Similar results were observed|ECB leaked structural patterns, while CBC completely obscured the original image.

3.4 Code Snippets

Example commands for extracting headers and creating a viewable CBC-encrypted image:

```
head -c 54 my_image.bmp > header
tail -c +55 my_cbc.bin > body
cat header body > my_cbc_view.bmp
```

These experiments clearly demonstrate why ECB is insecure for data containing repeating structures, and why CBC (or other modern modes) is preferred for image or file encryption.

Chapter 4

Padding

4.1 Description

This task investigates how padding is applied in block cipher encryption when plaintext size is not a multiple of the block size. We explore different AES modes (ECB, CBC, CFB, OFB), identify which require padding, and analyze the padding bytes using PKCS#5/PKCS#7 schemes.

4.2 Steps and Screenshots

1. Encrypted a sample file (plain.txt) with AES-128 in four modes:

```
openssl enc -aes-128-ecb -e -in plain.txt -out ecb.bin -K 00112233445566778899aabbccdd
openssl enc -aes-128-cbc -e -in plain.txt -out cbc.bin -K 00112233445566778899aabbccdd
openssl enc -aes-128-cfb -e -in plain.txt -out cfb.bin -K 00112233445566778899aabbccdd
openssl enc -aes-128-ofb -e -in plain.txt -out ofb.bin -K 00112233445566778899aabbccdd
```

2. Created files of different lengths (5, 10, and 16 bytes) to observe padding:

```
echo -n "12345" > f1.txt
echo -n "1234567890" > f2.txt
echo -n "1234567890ABCDEF" > f3.txt
```

3. Encrypted the files with AES-128-CBC:

```
openssl enc -aes-128-cbc -e -in f1.txt -out f1.enc -K 00112233445566778899aabbccdd
openssl enc -aes-128-cbc -e -in f2.txt -out f2.enc -K 00112233445566778899aabbccdd
openssl enc -aes-128-cbc -e -in f3.txt -out f3.enc -K 00112233445566778899aabbccdd
```

4. Decrypted with the -nopad option to reveal raw padding:

```
openssl enc -aes-128-cbc -d -nopad -in f1.enc -out f1_dec.bin -K 00112233445566778899aabbccdd
```

5. Used hex tools to inspect padding bytes:

```
hexdump -C f1_dec.bin
xxd f1_dec.bin
```

4.3 Observations & Explanations

- **Padding Requirement:** ECB and CBC require padding when plaintext length is not a multiple of the 16-byte AES block size. CFB and OFB operate as stream ciphers and do not need padding because they process data in smaller chunks.
- **Encrypted File Sizes:**
 - 5-byte file (f1.txt) produced a 16-byte encrypted output.
 - 10-byte file (f2.txt) also produced a 16-byte encrypted output.
 - 16-byte file (f3.txt) produced a 32-byte encrypted output due to a full extra padding block.
- **Padding Pattern:** PKCS#5 padding appends N bytes of value N . Hex inspection revealed:
 - For f1.txt (5 bytes), 11 bytes of 0x0B were added.
 - For f2.txt (10 bytes), 6 bytes of 0x06 were added.
 - For f3.txt (16 bytes), a full block of 0x10 bytes was added.
- **CFB/OFB Modes:** No padding was observed since these modes encrypt data bit/byte streams directly.

4.4 Code Snippets

Example decryption command to reveal padding:

```
openssl enc -aes-128-cbc -d -nopad -in f1.enc -out f1_dec.bin \  
-K 00112233445566778889aabbccddeeff \  
-iv 0102030405060708
```

Hex dump example to examine padding bytes:

```
hexdump -C f1_dec.bin
```

Chapter 5

Error Propagation – Corrupted Cipher Text

Description

This task explores how different encryption modes handle bit corruption in ciphertext. A single-bit error was introduced to observe how it affects the decrypted plaintext across various modes.

Steps and Screenshots

1. Created a text file `long.txt` of at least 1000 bytes.
2. Encrypted the file using AES-128 with ECB, CBC, CFB, and OFB modes:

```
openssl enc -aes-128-cbc -e -in long.txt -out long_cbc.bin -K <key> -iv <iv>
```
3. Used `hex editor` to flip one bit of the 55th byte in each encrypted file.
4. Decrypted each corrupted ciphertext using the correct key and IV.
5. Screenshots captured encryption commands, hex editing, and decryption outputs.

Observations & Explanations

- ECB: Corruption affected only the 16-byte block containing the flipped bit; other blocks decrypted correctly because ECB encrypts each block independently.
- CBC: The corrupted block decrypted incorrectly, and the following block showed partial corruption due to chaining.
- CFB: Only the corrupted byte was affected; subsequent bytes remained intact because decryption feedback limits error spread.
- OFB: Only the corrupted byte was altered, similar to a stream cipher, as keystream generation is independent of ciphertext.

Code Snippets

```
# AES-128 CBC encryption
openssl enc -aes-128-cbc -e -in long.txt -out long_cbc.bin \
-K 00112233445566778889aabbccddeeff -iv 0102030405060708

# Decryption after corruption
openssl enc -aes-128-cbc -d -in long_cbc.bin -out out_cbc.txt \
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

The commands above were adapted for ECB, CFB, and OFB by changing the mode.

Chapter 6

Initial Vector (IV) and Common Mistakes

Description

This task explores the importance of IV (Initial Vector) uniqueness and unpredictability in secure encryption. Experiments were conducted to observe the effects of reusing or predicting IVs in various encryption modes.

Steps and Screenshots

1. Encrypted the same plaintext twice using AES-128-CBC with:
 - Two different IVs.
 - The same IV.
2. Compared ciphertext outputs for uniqueness testing.
3. For OFB mode, used given plaintext/ciphertext pairs to attempt a known-plaintext attack.
4. Interacted with the encryption oracle (nc 10.9.0.80 3000) to exploit predictable IVs.
5. Screenshots included encryption commands, oracle interaction, and output comparisons.

Observations & Explanations

Task 6.1 (IV Experiment):

- Different IVs produced completely different ciphertexts, even with identical plaintexts.
- Using the same IV resulted in identical ciphertexts, revealing patterns and compromising security.
- IV uniqueness prevents attackers from detecting repeated messages.

Task 6.2 (Common Mistake { Same IV}):

- In OFB mode, the keystream depends only on the key and IV. Reusing the IV allows:

$$P2 = P1 \oplus C1 \oplus C2$$

enabling full recovery of unknown plaintext P2.

- In CFB mode, only portions of P2 can be revealed because feedback introduces dependency on prior ciphertext blocks.

Task 6.3 (Predictable IV):

- A predictable IV lets an attacker craft chosen plaintexts and use the oracle's outputs to distinguish between candidate plaintexts (e.g., 'Yes' vs. 'No').
- By carefully selecting input and comparing ciphertext patterns, Bob's secret message can be determined despite AES remaining cryptographically strong.

Code Snippets

```
# Encrypt with different IVs
openssl enc -aes-128-cbc -e -in plain.txt -out cipher1.bin \
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher2.bin \
-K 00112233445566778889aabbccddeeff -iv 1122334455667788
```

```
# Known-plaintext attack in OFB
P2 = xor(xor(P1, C1), C2) # Using Python bytearray XOR
```

These commands demonstrate the impact of IV reuse and how predictable IVs can be exploited.

Chapter 7

Programming using the Crypto Library

Description

The objective of this task was to recover the AES-128-CBC encryption key used to encrypt a known plaintext by performing a dictionary-based brute-force attack. The key was known to be an English word of fewer than 16 characters, padded with pound signs (#) to make a 128-bit key. The plaintext, ciphertext, and IV were provided, and the openssl crypto library (EVP API) was used to implement the attack in C.

Steps and Screenshots

1. Stored the plaintext in a file without trailing newline:

```
echo -n "This is a top secret." > plaintext.txt
```

2. Downloaded a dictionary wordlist and iterated through each word, padding with # to 16 bytes.
3. Used the OpenSSL EVP_CipherInit and related functions to encrypt the plaintext with each candidate key and compare the result with the provided ciphertext.
4. Executed the compiled program to find the matching key.

A screenshot of the terminal output confirmed the successful discovery of the correct key.

Observations & Explanations

- AES-128 requires an exact 16-byte key, so shorter dictionary words needed padding with pound signs to meet the key length.
- The brute-force search was efficient due to the limited dictionary size and the known padding scheme.
- Upon matching the ciphertext, the correct key was quickly verified by decrypting the message back to the known plaintext.

Code Snippets

The following C code fragment illustrates the main brute-force logic:

```
for each word in dictionary {
    pad_word_with_hashes(word, key, 16);
    EVP_CipherInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv, 1);
    EVP_CipherUpdate(ctx, outbuf, &outlen, plaintext, plaintext_len);
    EVP_CipherFinal_ex(ctx, outbuf + outlen, &tmplen);
    if (compare_ciphertext(outbuf, ciphertext)) {
        printf("Key found: %s\n", word);
        break;
    }
}
```

This snippet shows the process of initializing the cipher context, encrypting the plaintext with each candidate key, and comparing the output with the given ciphertext.

Conclusion

This lab gave me practical insight into secret-key encryption and how different ciphers and modes behave. I observed how ECB leaks visible patterns while CBC and other modes provide stronger protection, learned when and why padding is needed, and saw how errors propagate differently across modes. Writing a program to brute-force an AES key reinforced the importance of strong, unpredictable keys.

Configuring OpenSSL, handling binary data, and managing hex keys were challenging but improved my debugging skills and confidence with cryptographic tools. Overall, this lab strengthened my understanding of both the theory and real-world implementation of encryption.