SEED Labs - Secret-Key Encryption Lab

Jatin Kesnani
21K-3204 Student
FAST University

Abstract

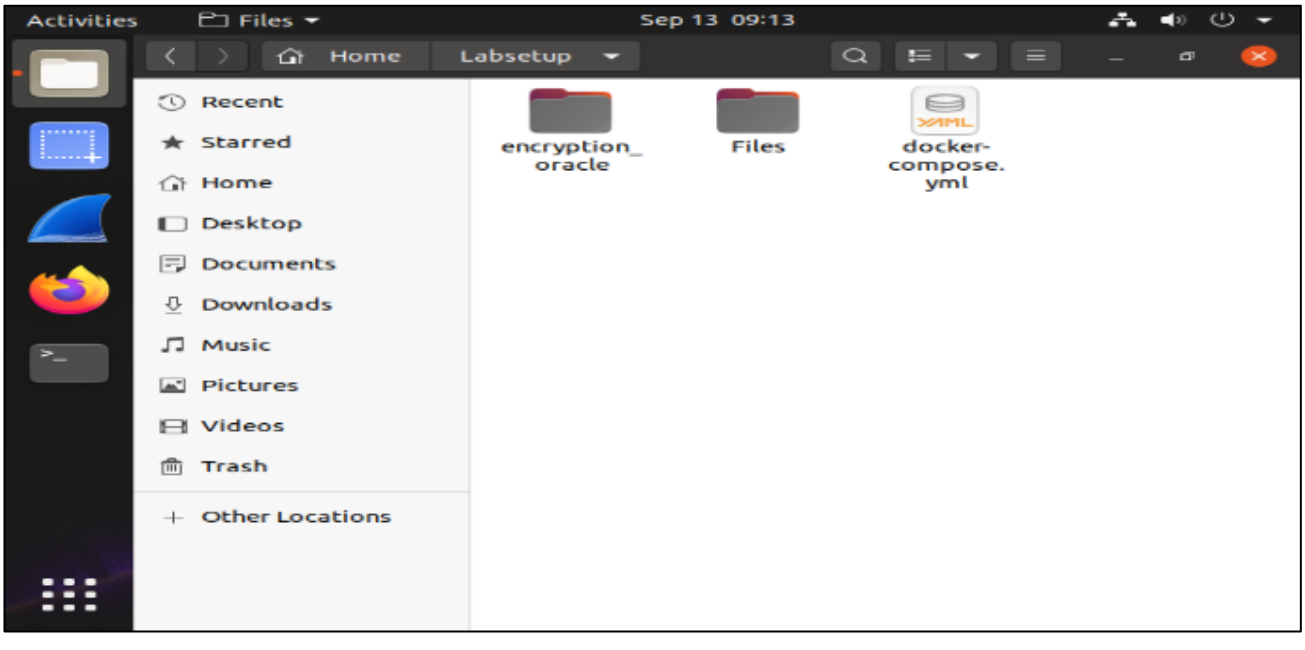
In today's digital world, ensuring the security and privacy of information is paramount. Cryptography, through its various encryption algorithms, provides a robust means of protecting sensitive data from unauthorized access. This report explores the fundamental principles of cryptography, focusing on encryption techniques that secure digital communications. We investigate various encryption methods, including symmetric and asymmetric encryption, and evaluate their effectiveness in safeguarding information in different scenarios. The objective is to gain a deeper understanding of the encryption mechanisms and their practical applications in securing data transmissions.

1. Overview

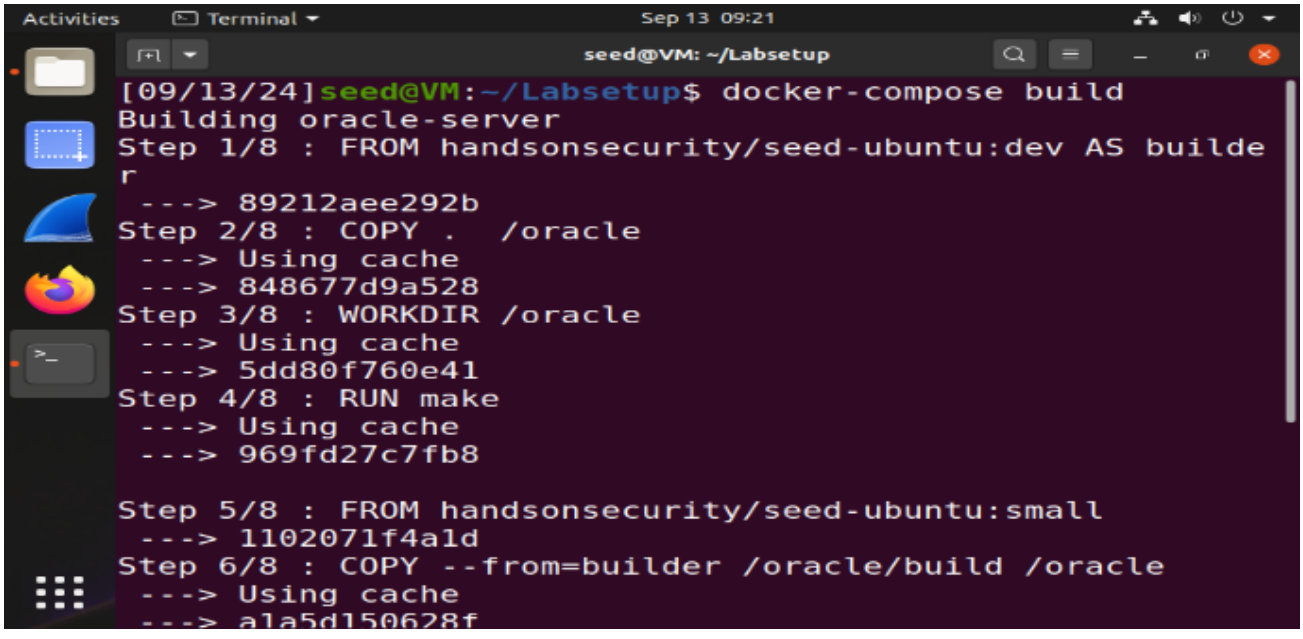
Cryptography is essential for securing digital communication and protecting sensitive data from unauthorized access. With the rise of online data exchange, encryption techniques play a key role in ensuring confidentiality and integrity. This section provides a brief look at symmetric and asymmetric encryption methods. Symmetric encryption uses a single key for both encryption and decryption, while asymmetric encryption employs a public-private key pair for secure key exchange. The report also highlights the practical applications of encryption in securing online transactions, emails, and cloud storage, emphasizing the importance of choosing the right method for different security needs.

2. Lab Environment

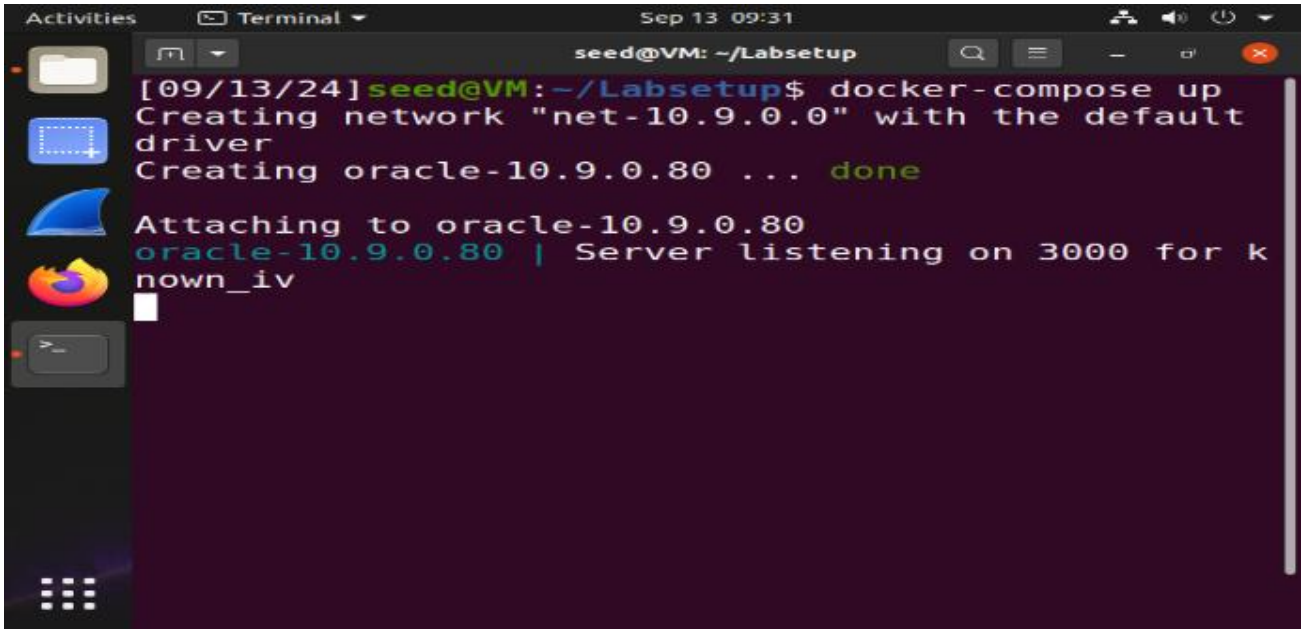
2.1 Navigating to the Labsetup Directory



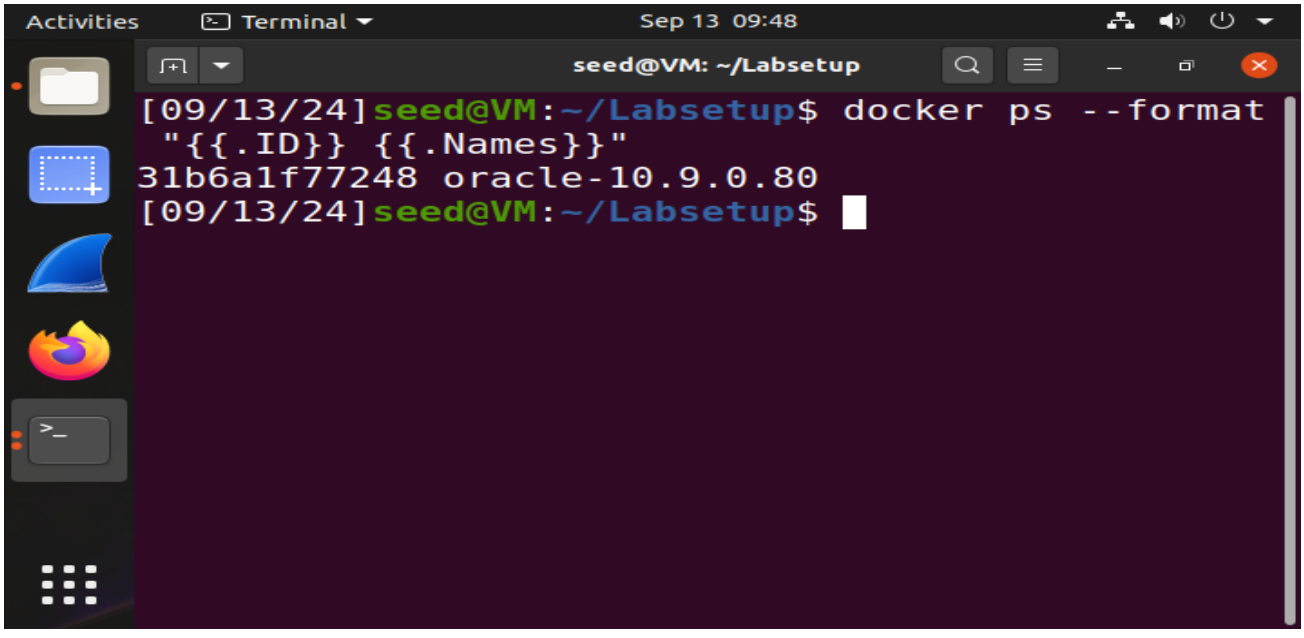
2.2 Building the Docker Container



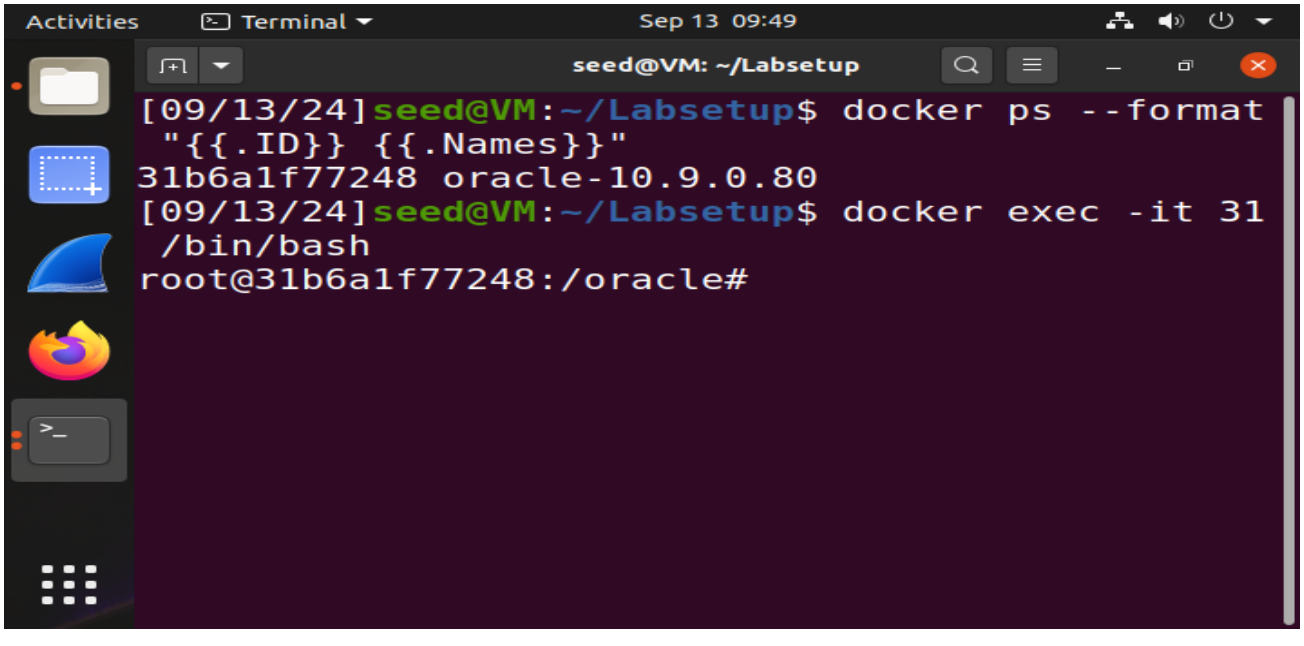
2.3 Starting the Docker Container



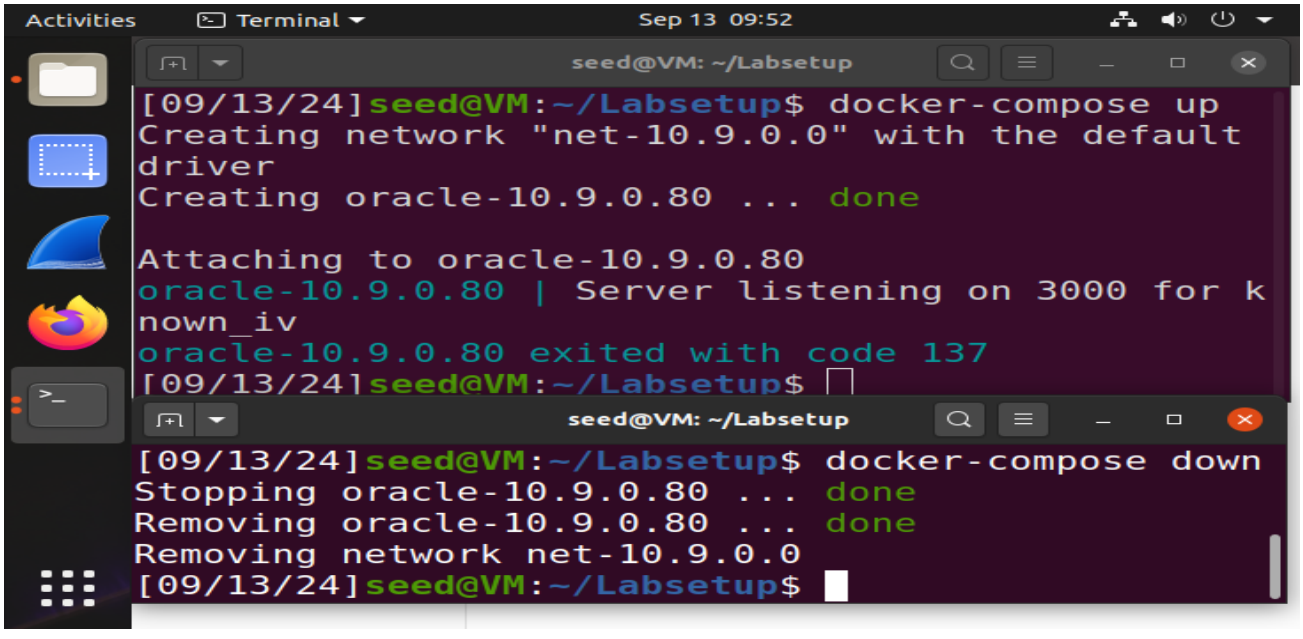
2.4 Verifying the Container is Running



2.5 Accessing the Container Shell

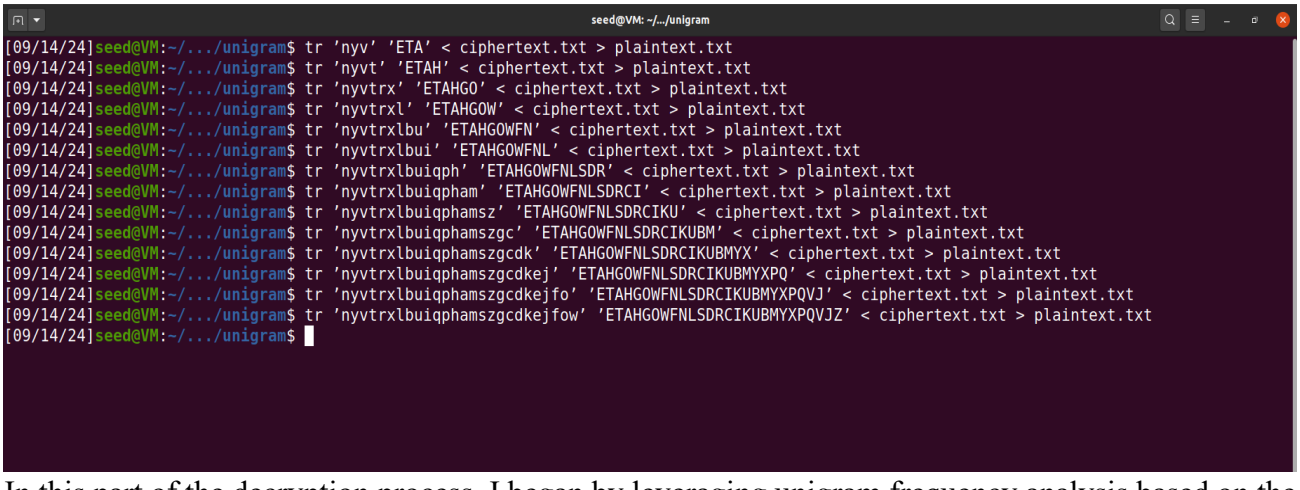


2.6 Shutting Down the Container



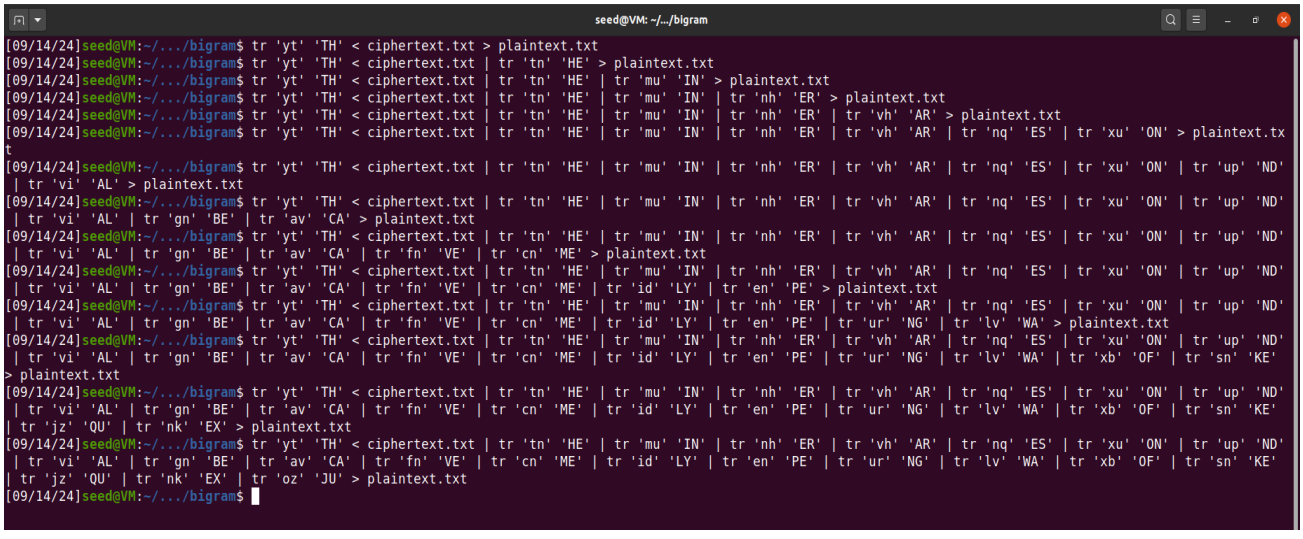
3. Task 1: Frequency Analysis

3.1 Monoalphabetic Cipher Using Unigram Frequencies



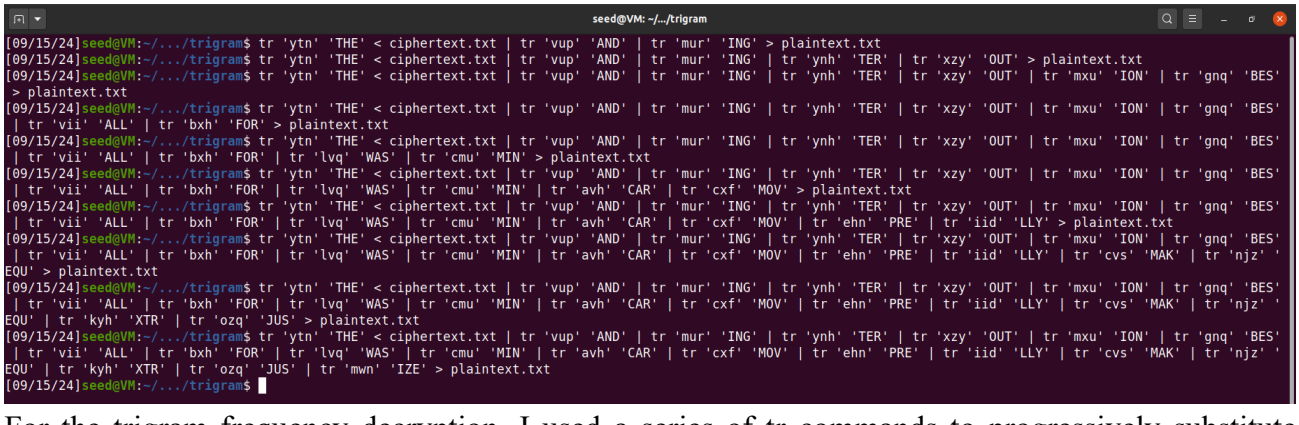
In this part of the decryption process, I began by leveraging unigram frequency analysis based on the provided Wikipedia link, which outlines the common letter frequencies in English plaintext. Initially, I applied the first three letters from the frequency list ('E', 'T', and 'A') to the ciphertext using the tr command in Linux, which allowed me to substitute these characters for their corresponding most frequent letters in the ciphertext. As I continued analyzing the output, I gradually introduced more frequent letters from the list and iterated the process, refining the decryption with each step. This systematic approach, combined with my understanding of English letter distribution, enabled me to progressively uncover the plaintext. The success of this method lies in identifying the most common characters and adapting the substitution based on patterns that emerge in the partially decrypted text. This process highlighted the efficiency of frequency analysis in cryptography, especially when dealing with simple substitution ciphers.

3.2 Monoalphabetic Cipher Using Bigram Frequencies



I used bigram frequency analysis to decrypt a ciphertext by progressively replacing common bigrams in the encrypted text with corresponding common English bigrams. Using the tr command, I first substituted 'yt' with "TH", which is the most frequent bigram in English. I then continued replacing other common bigrams such as 'tn' with "HE", 'mu' with "IN", and so on, gradually making the plaintext more readable. With each additional substitution, the ciphertext transformed into clearer text, revealing the underlying message step by step. The incremental approach made the decryption process efficient and allowed for interesting observations about how quickly the message became understandable after just a few key replacements.

3.3 Monoalphabetic Cipher Using Trigram Frequencies



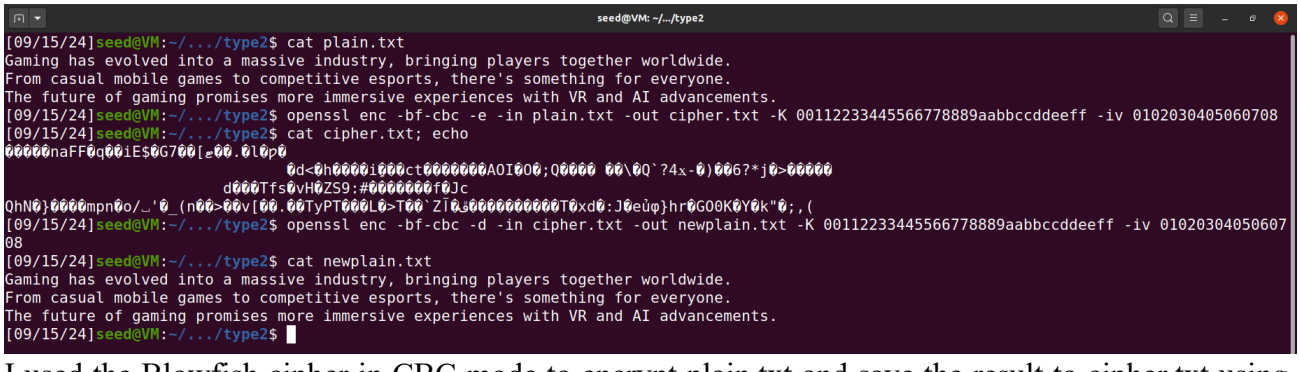
For the trigram frequency decryption, I used a series of tr commands to progressively substitute common trigrams in the ciphertext with their corresponding plaintext counterparts. By iteratively applying these transformations, I was able to decipher the text step by step. The process involved identifying frequent trigrams like "THE", "AND", "ING", and replacing them within the ciphertext. Each substitution brought the plaintext closer to readability, allowing for further analysis and refinements. This method proved effective for decoding large portions of the ciphertext, providing a clearer understanding of its content with each additional step.

4. Task 2: Encryption using Different Ciphers and Modes

4.1 Cipher Type: AES-128-CBC

In this experiment, AES-128-CBC encryption and decryption were demonstrated using OpenSSL. The plaintext file plain.txt was encrypted into cipher.txt using the command with a specified key and initialization vector (IV). The ciphertext was then decrypted back to plaintext using the same key and IV, and the result was saved in newplain.txt. The contents of cipher.txt and newplain.txt were examined to ensure the decryption process accurately restored the original text, confirming that the encryption and decryption were successfully performed.

4.2 Cipher Type: BF-CBC



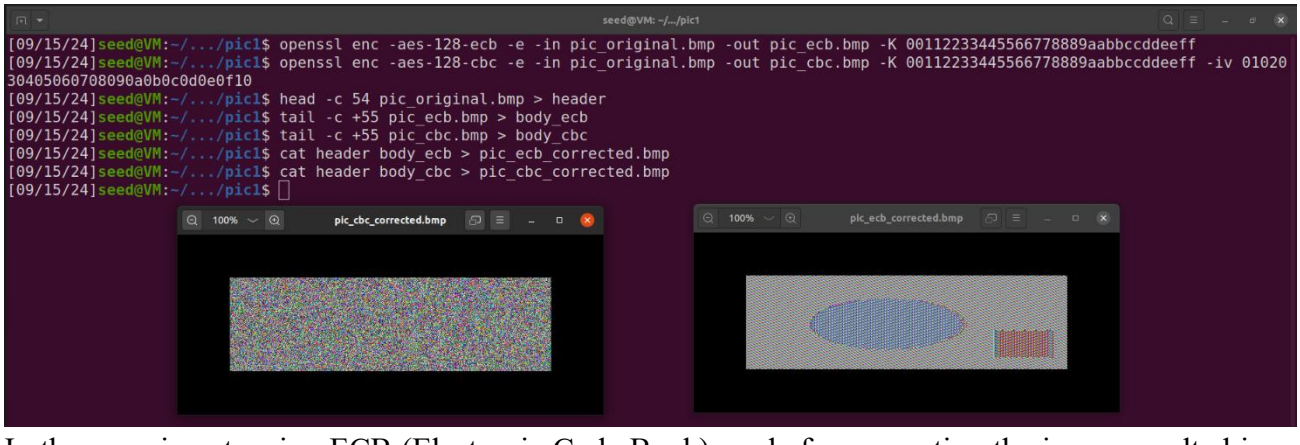
I used the Blowfish cipher in CBC mode to encrypt plain.txt and save the result to cipher.txt using the command openssl enc -bf-cbc -e -in plain.txt -out cipher.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708. I then decrypted cipher.txt back to newplain.txt with openssl enc -bf-cbc -d -in cipher.txt -out newplain.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708. The decryption successfully restored the original content, confirming the effectiveness of the Blowfish CBC encryption and decryption process.

4.3 Cipher Type: AES-128-CFB

I used AES-128-CFB for encrypting and decrypting a text file. The encryption command was openssl enc -aes-128-cfb -e -in plain.txt -out cipher.txt -K 00112233445566778889aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10, which saved the encrypted content to cipher.txt. For decryption, I used openssl enc -aes-128-cfb -d -in cipher.txt -out newplain.txt -K 00112233445566778889aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10, restoring the original text to newplain.txt.

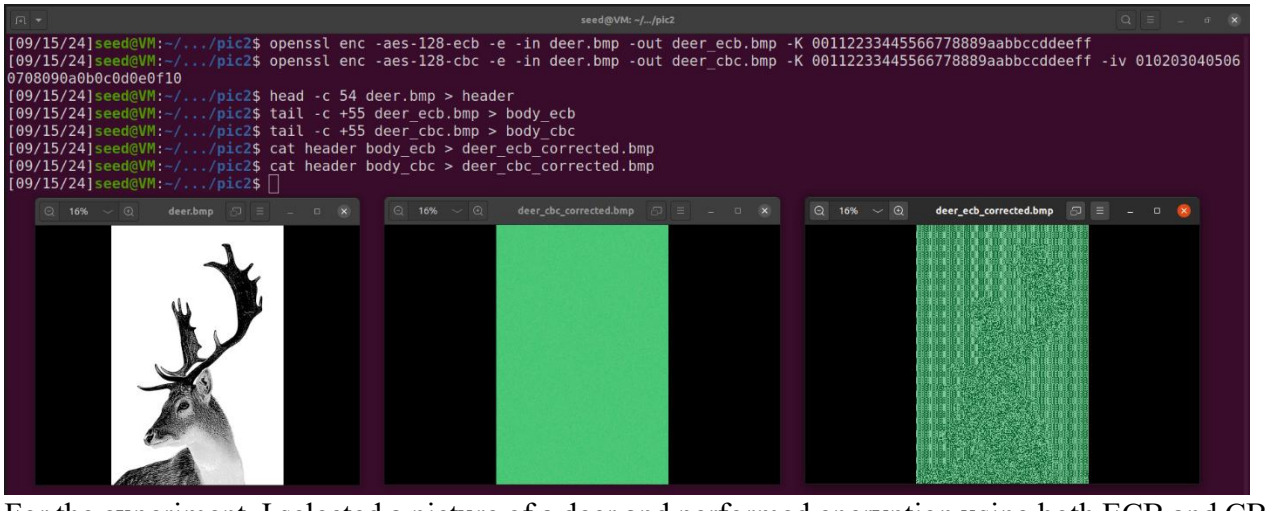
5. Task 3: Encryption Mode – ECB vs. CBC

5.1 Encryption (pic_original.bmp) – ECB vs. CBC



In the experiment, using ECB (Electronic Code Book) mode for encrypting the image resulted in a partially visible picture with noticeable encrypted patterns. This occurs because ECB encrypts each block of plaintext independently, preserving patterns from the original image. Consequently, repeated sections in the image remain somewhat recognizable in the encrypted output. In contrast, CBC (Cipher Block Chaining) mode, which incorporates an initialization vector (IV) and chains blocks together, completely obscured the picture. CBC mode ensures that identical plaintext blocks produce different ciphertext blocks due to the chaining effect, making the encrypted image appear as random noise. This demonstrates that CBC mode provides better security by preventing pattern recognition, while ECB mode's pattern preservation can compromise confidentiality.

5.2 Encryption (deer.bmp) – ECB vs. CBC



For the experiment, I selected a picture of a deer and performed encryption using both ECB and CBC modes. The results of the encryption were as follows:

In CBC mode, the image turned light green, and the deer was not visible, demonstrating that the encryption method effectively concealed the contents of the picture. In ECB mode, the encrypted image showed visible pixel patterns, particularly where the deer was in the original picture. Although the exact shape of the deer couldn't be discerned, the difference in pixel values indicated where the deer was present against the white background. This shows that ECB mode is less secure for encrypting images, as it preserves visible patterns, whereas CBC mode fully obscures the image content.

6. Task 4: Padding

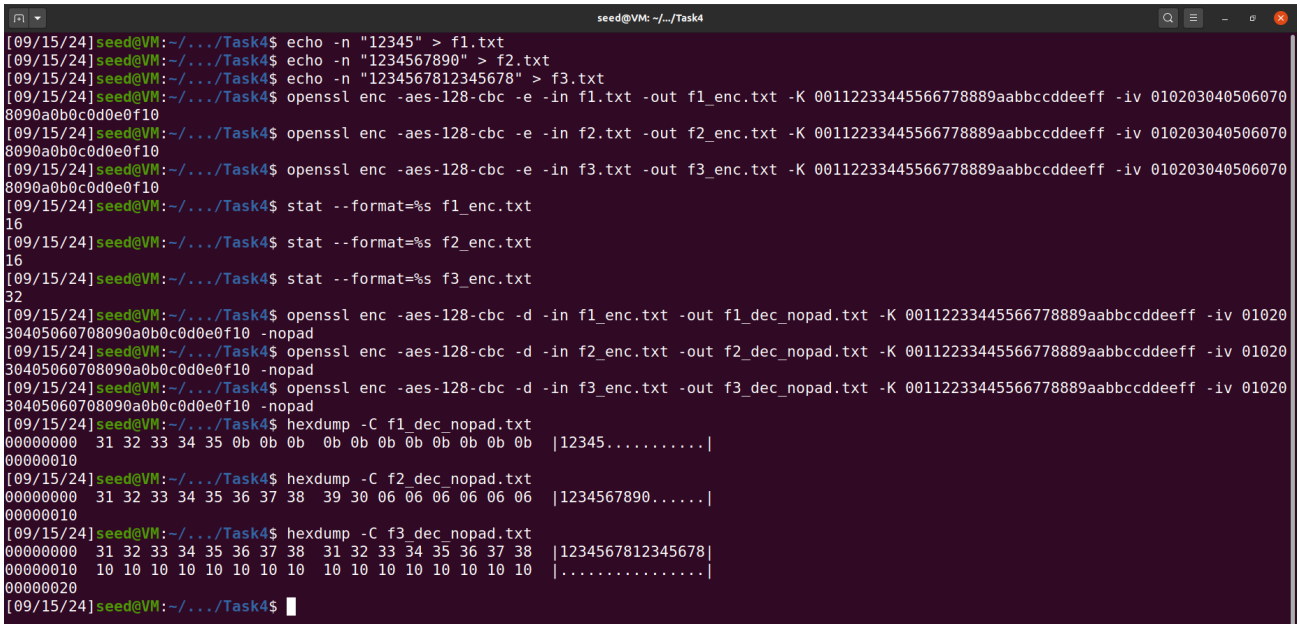
6.1.1 Modes that Require Padding:

- **ECB (Electronic Codebook Mode):** ECB requires padding because it processes the plaintext in fixed block sizes (in this case, 128 bits or 16 bytes). If the length of the plaintext is not a multiple of the block size, padding is needed to ensure that each block is of uniform length.
- **CBC (Cipher Block Chaining Mode):** CBC mode also requires padding. Like ECB, it operates on blocks of fixed size, and padding is added if the final block of plaintext is shorter than the block size. The padding ensures the final block has the required size for encryption.

6.1.2 Modes that Do Not Require Padding:

- **CFB (Cipher Feedback Mode):** CFB does not require padding because it operates in a stream-like manner. It encrypts data in smaller segments (less than the block size), so the plaintext can be any length, and padding is unnecessary.
- **OFB (Output Feedback Mode):** OFB also works as a stream cipher, encrypting smaller segments of the plaintext at a time. It does not require padding because the mode does not rely on processing data in fixed block sizes.

6.2 Data Used in Padding:



In this task, three files (5, 10, and 16 bytes) were encrypted using AES-128 in CBC mode. The encrypted file sizes were 16 bytes for the 5 and 10-byte files, and 32 bytes for the 16-byte file. Padding was added to make each file a multiple of the 16-byte block size. The 5-byte file was padded with 11 bytes of "0x0b", the 10-byte file with 6 bytes of "0x0b", and the 16-byte file with a full block of 16 bytes of "0x0b" padding. This ensures proper encryption alignment for block ciphers.

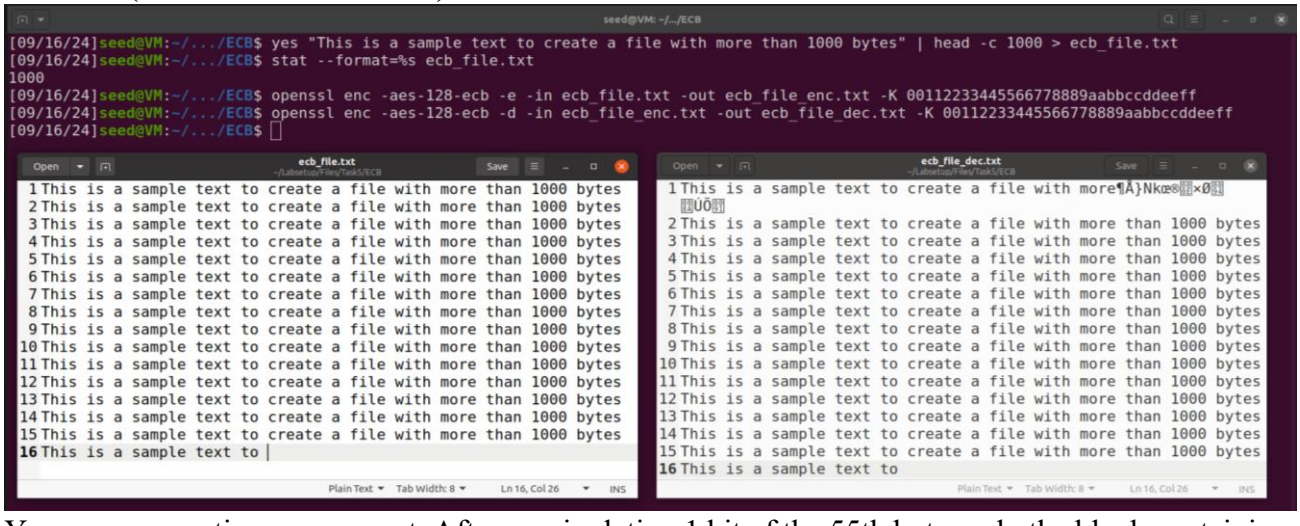
7. Task 5: Error Propagation – Corrupted Cipher Text

7.1 Before Conducting the Task:

- **ECB (Electronic Code Book)**: In ECB mode, each block is encrypted independently. If a single bit in the 55th byte gets corrupted, only that specific block will be affected. The rest of the blocks will decrypt correctly. Therefore, most of the information can still be recovered, except for the corrupted block.
- **CBC (Cipher Block Chaining)**: In CBC mode, each block is dependent on the previous block. If a bit in the 55th byte (within a block) is corrupted, that entire block will be corrupted, and due to the chaining, the next block will also be affected. Thus, the error will propagate, causing two blocks to be corrupted, but the rest will decrypt correctly.
- **CFB (Cipher Feedback)**: In CFB mode, errors will propagate only for the duration of one block size (128 bits). A single corrupted bit will affect the current block, but future blocks will decrypt correctly. Thus, one block of data will be corrupted, but the rest of the file will be recoverable.
- **OFB (Output Feedback)**: In OFB mode, the corruption of one bit in the ciphertext will only affect the corresponding bit in the decrypted output, as OFB does not propagate errors to subsequent blocks. Therefore, almost the entire file will be recoverable except for one corrupted bit.

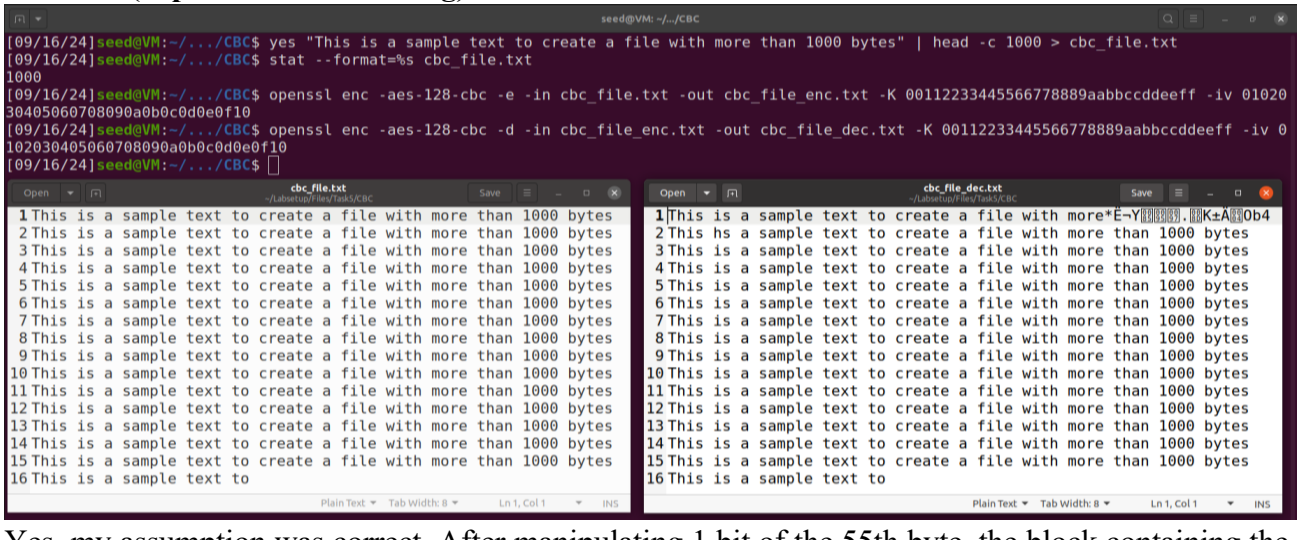
7.2 After Conducting the Task:

- **ECB (Electronic Code Book)**:



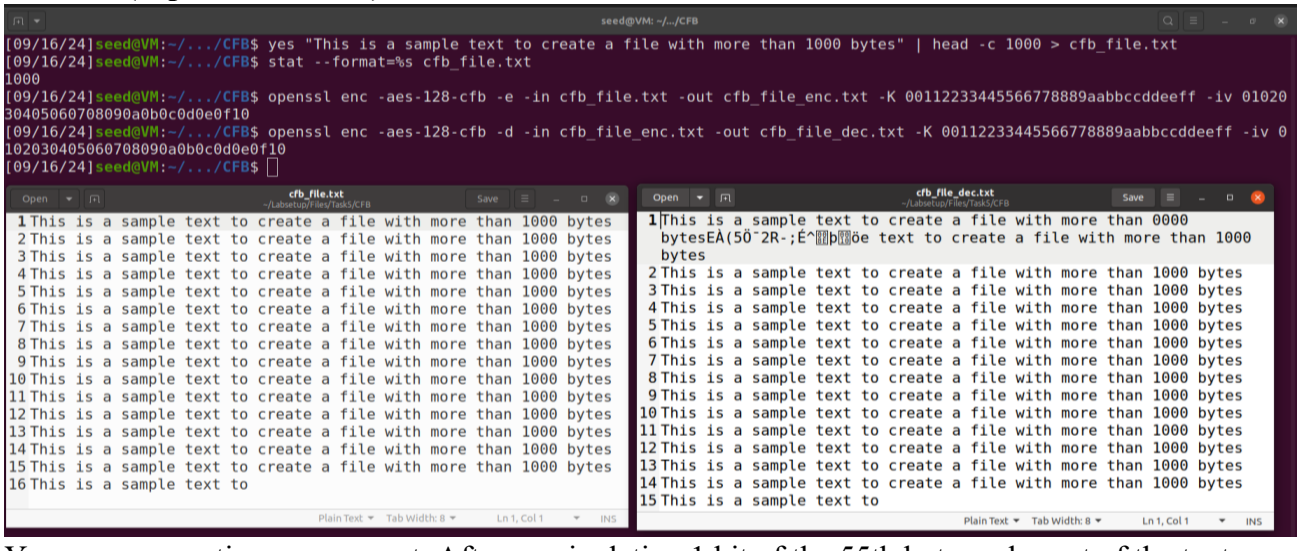
Yes, my assumption was correct. After manipulating 1 bit of the 55th byte, only the block containing that byte became corrupted, while the rest of the file decrypted correctly. This is because ECB mode encrypts each block independently, so the error doesn't propagate to other blocks. The corrupted part in the decrypted text confirms that only one block was affected, validating my assumption.

- **CBC (Cipher Block Chaining)**:



Yes, my assumption was correct. After manipulating 1 bit of the 55th byte, the block containing the corruption and the subsequent block were affected due to the chaining nature of CBC mode. The rest of the text decrypted correctly. This confirms that CBC encryption propagates errors to the next block, as predicted. The results show two corrupted blocks, validating my assumption.

- **CFB (Cipher Feedback)**:



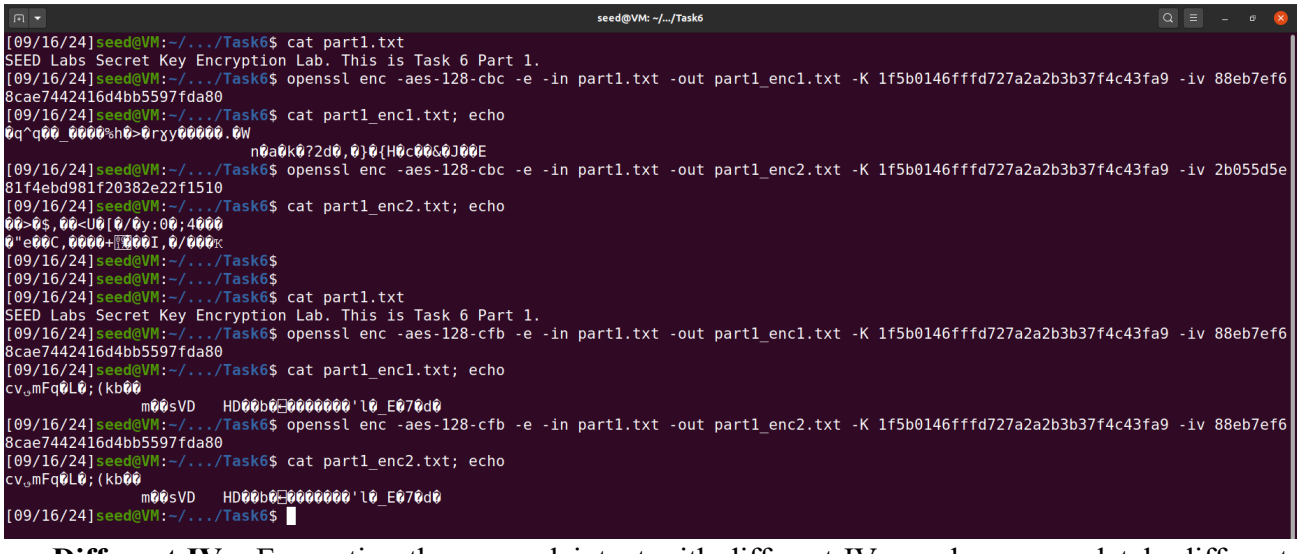
Yes, my assumption was correct. After manipulating 1 bit of the 55th byte, only part of the text was corrupted for one block (128 bits), as expected in CFB mode. The rest of the text decrypted correctly, confirming that the error only affected the current block and did not propagate beyond that. This validates my understanding of how CFB encryption handles errors.

- **OFB (Output Feedback)**:

Yes, my assumption was correct. After manipulating 1 bit of the 55th byte, only that specific bit was corrupted in the decrypted output, while the rest of the file remained intact. This confirms that in OFB mode, errors do not propagate to other blocks, and only the corresponding bit is affected, as expected.

8. Task 6: Initial Vector (IV) and Common Mistakes

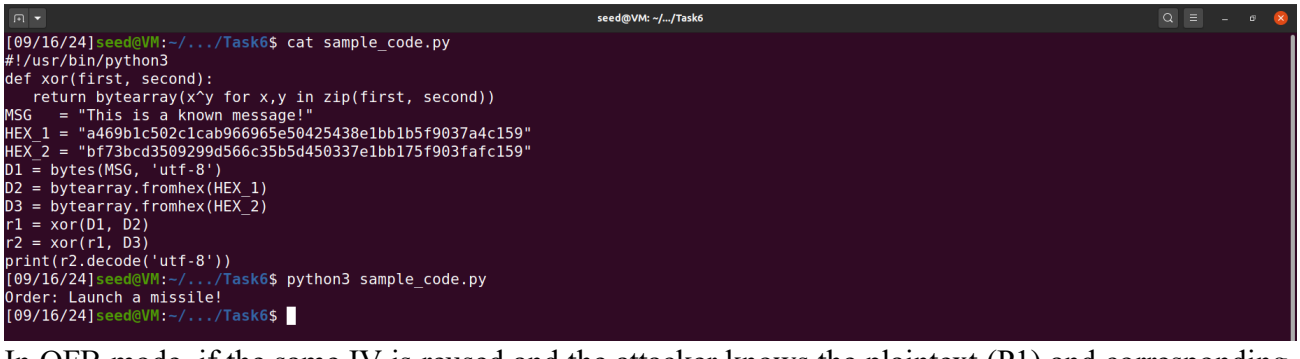
8.1 Task 6.1. IV Experiment



- **Different IVs:** Encrypting the same plaintext with different IVs produces completely different ciphertexts, ensuring uniqueness in the output.
- **Same IV:** Reusing the same IV with the same key results in identical ciphertexts, revealing that the same plaintext is being transmitted, which can leak information.

Explanation: The IV must be unique to ensure different ciphertexts for the same plaintext. Reusing an IV allows patterns to emerge, compromising the security of the encryption and potentially exposing the plaintext to attackers.

8.2 Task 6.2. Common Mistake: Use the Same IV



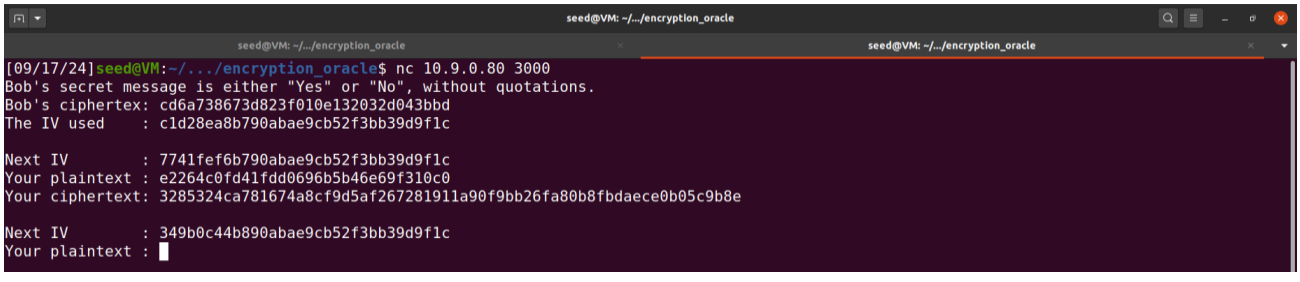
In OFB mode, if the same IV is reused and the attacker knows the plaintext (P1) and corresponding ciphertext (C1), they can decrypt other ciphertexts (like C2) by leveraging the XOR operation. Specifically, since the keystream generated from the IV remains the same, the attacker can recover the keystream used to encrypt P2 by performing:

1. XOR between P1 and C1 to reveal the keystream.
2. XOR the resulting keystream with C2 to reveal P2.

This means that, in OFB mode, if the IV is reused, the entire content of P2 can be revealed using known-plaintext attacks.

For CFB mode, however, only the first block (in this case, the first 128 bits or 16 bytes) of P2 can be revealed because CFB mode encrypts each block using feedback from the previous ciphertext. Thus, the attack will only partially reveal P2, but not beyond the first block.

8.3 Task 6.3. Common Mistake: Use a Predictable IV



In this task, we aimed to determine whether Bob's encrypted message was "Yes" or "No" by exploiting the predictability of the initialization vector (IV) in the AES-CBC encryption scheme. Bob's message is either "Yes" or "No", and both the ciphertext and IV used in encryption were provided. Since the IVs were predictable, we used this information to reverse the encryption process. By XORing Bob's ciphertext (cd6a738673d823f010e132032d043bbd) with the IV used (c1d28ea8b790abae9cb52f3bb39d9f1c), we obtained the intermediate result (0cb8fd2ec448885e8c541d389e99a4a1). To determine the original plaintext, we XORed this result with the ASCII representations of "Yes" and "No". The XOR result with "Yes" (55dd8e2ec448885e8c541d389e99a4a1) yielded a valid structure, confirming that Bob's secret message was **"Yes"**. This exercise highlights the vulnerability introduced by using predictable IVs, allowing an attacker to deduce the plaintext even when strong encryption algorithms like AES are used.

8.4 Additional Readings

Article: In CBC mode encryption, the Initialization Vector (IV) must be random and unpredictable to ensure security. The IV prevents attackers from identifying patterns by ensuring that the same plaintext encrypts to different ciphertexts each time. A predictable IV can lead to vulnerabilities, allowing attackers to exploit correlations between the IV and plaintext, especially in chosen-plaintext attacks where they manipulate the system to learn which ciphertext matches a known message.
After encryption, the IV can be made public without compromising security, but it must be protected during encryption. If an attacker can alter the IV before decryption, they can tamper with the first block of plaintext. To prevent this, the IV and ciphertext should be authenticated using a Message Authentication Code (MAC) to ensure integrity and prevent tampering. Random, unpredictable IVs combined with authentication provide robust security in CBC encryption.
Random Number Generation Lab: To generate cryptographically strong pseudo-random numbers, it's crucial to use sources that ensure randomness and unpredictability. In Linux, this can be achieved through the /dev/random and /dev/urandom devices, which collect random data from physical sources like mouse movements and keystrokes. /dev/random is more secure but can block when entropy is low, making it susceptible to Denial-of-Service (DoS) attacks. /dev/urandom, on the other hand, continuously generates random numbers without blocking, re-seeding when new entropy is available. For most applications, /dev/urandom is recommended as it provides a balance of security and performance without risking DoS. Cryptographic programs should use this device to generate keys, as shown in the code snippet provided, which reads random data to generate a 128-bit key. By

modifying this code, a 256-bit key can also be created, ensuring strong security for encryption purposes.

9. Task 7: Programming using the Crypto Library

myenc.c
~/Labsetup/Files/Task7

```c
#include <openssl/aes.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>

#define charMaxLeng 20

unsigned char AscToHex(unsigned char Char) {
    int aChar = (int) Char;
    if ((aChar >= 0x30) && (aChar <= 0x39))
        aChar -= 0x30;
    else if ((aChar >= 0x41) && (aChar <= 0x46))
        aChar -= 0x37;
    else if ((aChar >= 0x61) && (aChar <= 0x66))
        aChar -= 0x57;
    else
        aChar = 0xff;
    return aChar;
}

unsigned char HexToAsc(unsigned char aHex) {
    if ((aHex >= 0) && (aHex <= 9))
        aHex += 0x30;
    else if ((aHex >= 10) && (aHex <= 15))
        aHex += 0x37;
    else
        aHex = 0xff;
    return aHex;
}

unsigned char* str2hex(char *str) {
    unsigned char *ret = NULL;
    int str_len = strlen(str);
    int i = 0;
    assert((str_len % 2) == 0);
    ret = (char *) malloc(str_len / 2);
    for (i = 0; i < str_len; i = i + 2) {
        sscanf(str + i, "%2hhx", &ret[i / 2]);
    }
    return ret;
}

```

myenc.c
~/Labsetup/Files/Task7

```c
char *padding_buf(char *buf, int size, int *final_size) {
    char *ret = NULL;
    int padding_size = AES_BLOCK_SIZE - (size % AES_BLOCK_SIZE);
    int i;
    *final_size = size + padding_size;
    ret = (char *) malloc(size + padding_size);
    memcpy(ret, buf, size);
    if (padding_size != 0) {
        for (i = size; i < (size + padding_size); i++) {
            ret[i] = padding_size;
        }
    }
    return ret;
}

void printf_buff(char *buff, int size) {
    int i = 0;
    for (i = 0; i < size; i++) {
        printf("%02X", (unsigned char) buff[i]);
    }
    printf("\n");
}

void encrypt_buffer(char *raw_buf, char **encrypted_output, int len, char* kkey) {
    AES_KEY aes;
    unsigned char *key = str2hex(kkey);
    unsigned char *iv = str2hex("aabbccddeeff00998877665544332211");
    AES_set_encrypt_key(key, 128, &aes);
    AES_cbc_encrypt(raw_buf, *encrypted_output, len, &aes, iv, AES_ENCRYPT);
    free(key);
    free(iv);
}

void decrypt_buf(char *raw_buf, char **decrypt_buf, int len, char* kkey) {
    AES_KEY aes;
    unsigned char *key = str2hex(kkey);
    unsigned char *iv = str2hex("aabbccddeeff00998877665544332211");
    AES_set_decrypt_key(key, 128, &aes);
    AES_cbc_encrypt(raw_buf, *decrypt_buf, len, &aes, iv, AES_DECRYPT);
    free(key);
    free(iv);
}

```

myenc.c
~/Labsetup/Files/Task7

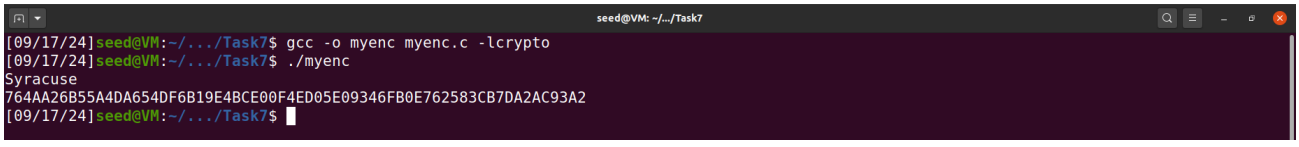
```c
int main(int argc, char* argv[]) {
    char* target = "764AA26B55A4DA654DF6B19E4BCE00F4ED05E09346FB0E762583CB7DA2AC93A2";
    FILE* p = NULL;

    if ((p = fopen("words.txt", "r")) == NULL) {
        printf("ERROR\n");
        return 1;
    }

    char buffer[charMaxLeng];
    char buf2[charMaxLeng];
    int flag = 0;

    while (!feof(p)) {
        int i = 0;
        memset(buffer, '\0', charMaxLeng * sizeof(char));
        memset(buf2, '\0', charMaxLeng * sizeof(char));
        fgets(buffer, charMaxLeng, p);

        while (i < charMaxLeng) {
            buf2[i] = buffer[i];
            i += 1;
        }

        size_t len = strlen(buffer);
        if (len == 1) continue;

        char *raw_buf = NULL;
        char *after_padding_buf = NULL;
        int padding_size = 0;
        char *encrypted_output = NULL;
        char *decrypt_buf = NULL;

        i = 0;
        unsigned char* key = NULL;
        key = (unsigned char*) malloc(33);

        while (i < strlen(buffer)) {
            unsigned char letter = buffer[i];
            key[2 * i] = HexToAsc(letter / 0x10);
            key[2 * i + 1] = HexToAsc(letter % 0x10);
            ++i;
            if (i == 0x0f || buffer[i] < 0x20) break;
        }
```

myenc.c
~/Labsetup/Files/Task7

```c

        while (i < 0x10) {
            key[2 * i] = '2';
            key[2 * i + 1] = '3';
            ++i;
        }

        key[0x20] = '\0';
        raw_buf = (char *) malloc(21);
        memcpy(raw_buf, "This is a top secret.", 21);

        after_padding_buf = padding_buf(raw_buf, 21, &padding_size);
        encrypted_output = (char *) malloc(padding_size);

        encrypt_buffer(after_padding_buf, &encrypted_output, padding_size, key);

        i = 0;
        char temp = '\0';
        flag = 1;

        while (i < padding_size) {
            temp = HexToAsc((unsigned char) encrypted_output[i] / 0x10);
            if (temp != target[2 * i]) {
                flag = 0;
                break;
            }
            temp = HexToAsc((unsigned char) encrypted_output[i] % 0x10);
            if (temp != target[2 * i + 1]) {
                flag = 0;
                break;
            }
            i += 1;
        }

        if (flag == 0) {
            continue;
        }

        printf("%s", buf2);
        //printf_buff(encrypted_output, padding_size);
        printf("%s\n", target);

        free(raw_buf);
        free(after_padding_buf);
```

myenc.c
~/Labsetup/Files/Task7

```c
        free(encrypted_output);
        free(decrypt_buf);
        break;
    }

    fclose(p);
    return 0;
}
```



In this task, the goal was to identify the encryption key used in AES-128-CBC encryption given a plaintext, ciphertext, and initialization vector (IV).

The provided plaintext was:

- "This is a top secret."

The ciphertext was:

- 764AA26B55A4DA654DF6B19E4BCE00F4ED05E09346FB0E762583CB7DA2AC93A2

The IV was:

- aabbccddeeff00998877665544332211.

To solve this, a C program was written utilizing the OpenSSL library. The program read a list of candidate keys from a file and tested each one to see if it could generate the given ciphertext when encrypting the plaintext. It performed the following steps: padding the plaintext to match the AES block size, encrypting it using the candidate key, and comparing the resulting ciphertext to the given ciphertext.

Upon executing the program, it successfully identified the correct key as **"Syracuse"** by matching the generated ciphertext with the provided one. The key was found to be the valid one when it produced an output that matched the expected ciphertext. The final result was that the program correctly pinpointed the key used in the encryption process, demonstrating the effectiveness of the implemented solution in cracking AES-128-CBC encryption based on known plaintext and ciphertext.