Secret Key Encryption Project Report

Task 1: Frequency Analysis

Description: In this task, we applied frequency analysis to decipher a ciphertext that was encrypted using a monoalphabetic substitution cipher. The goal was to identify the original plaintext based on the frequency of individual letters and common letter pairs (bigrams and trigrams) in English.

Observations:

- By analyzing the frequency of letters in the ciphertext, we matched high-frequency letters to common letters in the English language.
- Bigram and trigram frequencies were analyzed, helping us identify frequent letter combinations and map them to common English word patterns.

Approach:

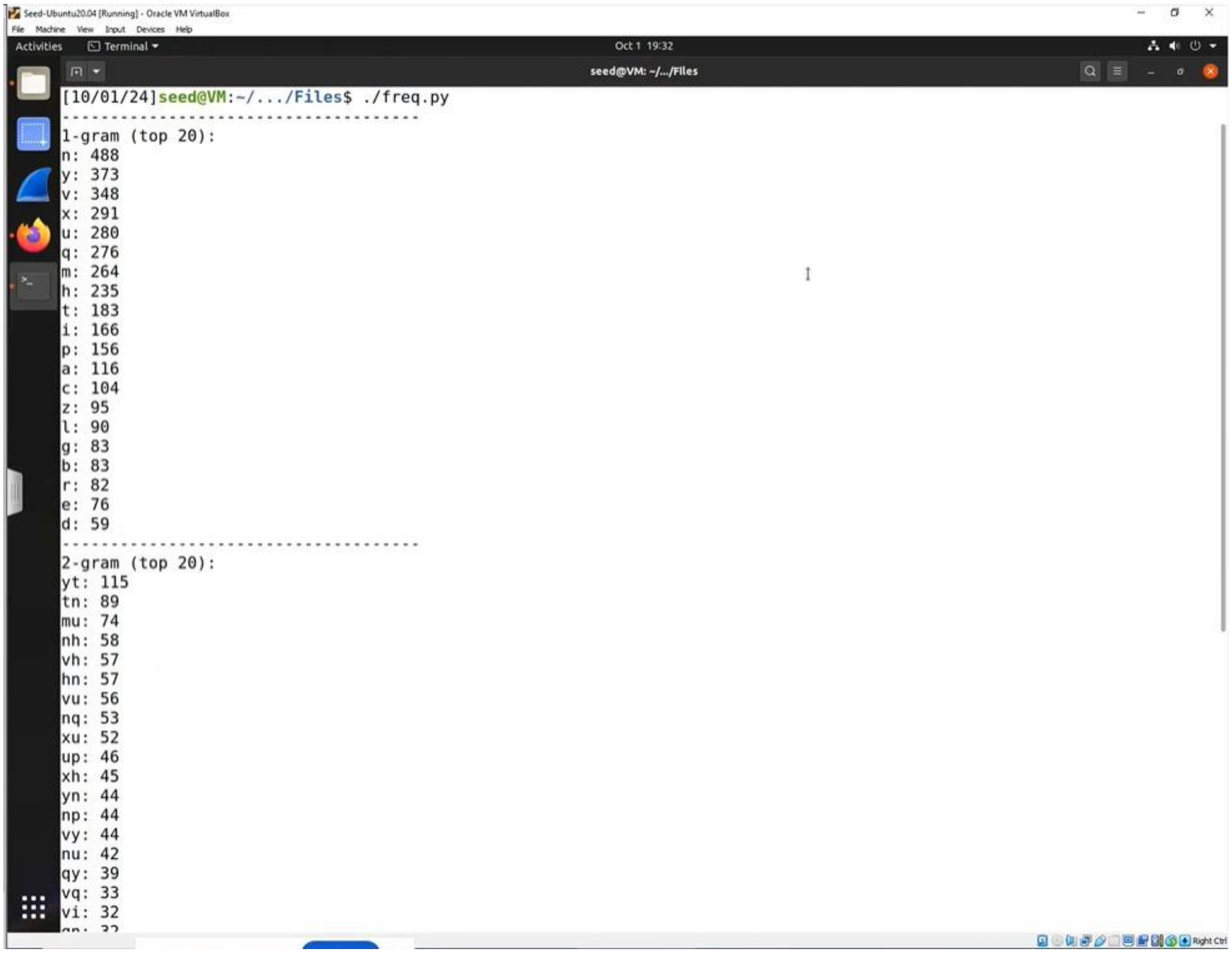
- Used the freq.py script to generate 1-gram, 2-gram, and 3-gram statistics from the ciphertext file. This script reads the file and identifies the most common patterns to help decipher the text.
- Mapped frequent ciphertext letters to common English letters such as 'E', 'T', 'A', etc
- Replaced ciphertext characters gradually to reveal the original message.

```
#!/usr/bin/env python3

from collections import Counter
import re

TOP_K  = 20
N_GRAM = 3

# Generate all the n-grams for value n
def ngrams(n, text):
    for i in range(len(text) -n + 1):
        # Ignore n-grams containing white space
        if not re.search(r'\s', text[i:i+n]):
           yield text[i:i+n]

# Read the data from the ciphertext
with open('ciphertext.txt') as f:
    text = f.read()

# Count, sort, and print out the n-grams
for N in range(N_GRAM):
   print("-------------------------------------")
   print("{}-gram (top {}):".format(N+1, TOP_K))
   counts = Counter(ngrams(N+1, text))        # Count
   sorted_counts = counts.most_common(TOP_K)  # Sort
   for ngram, count in sorted_counts:
       print("{}: {}".format(ngram, count))   # Print
```

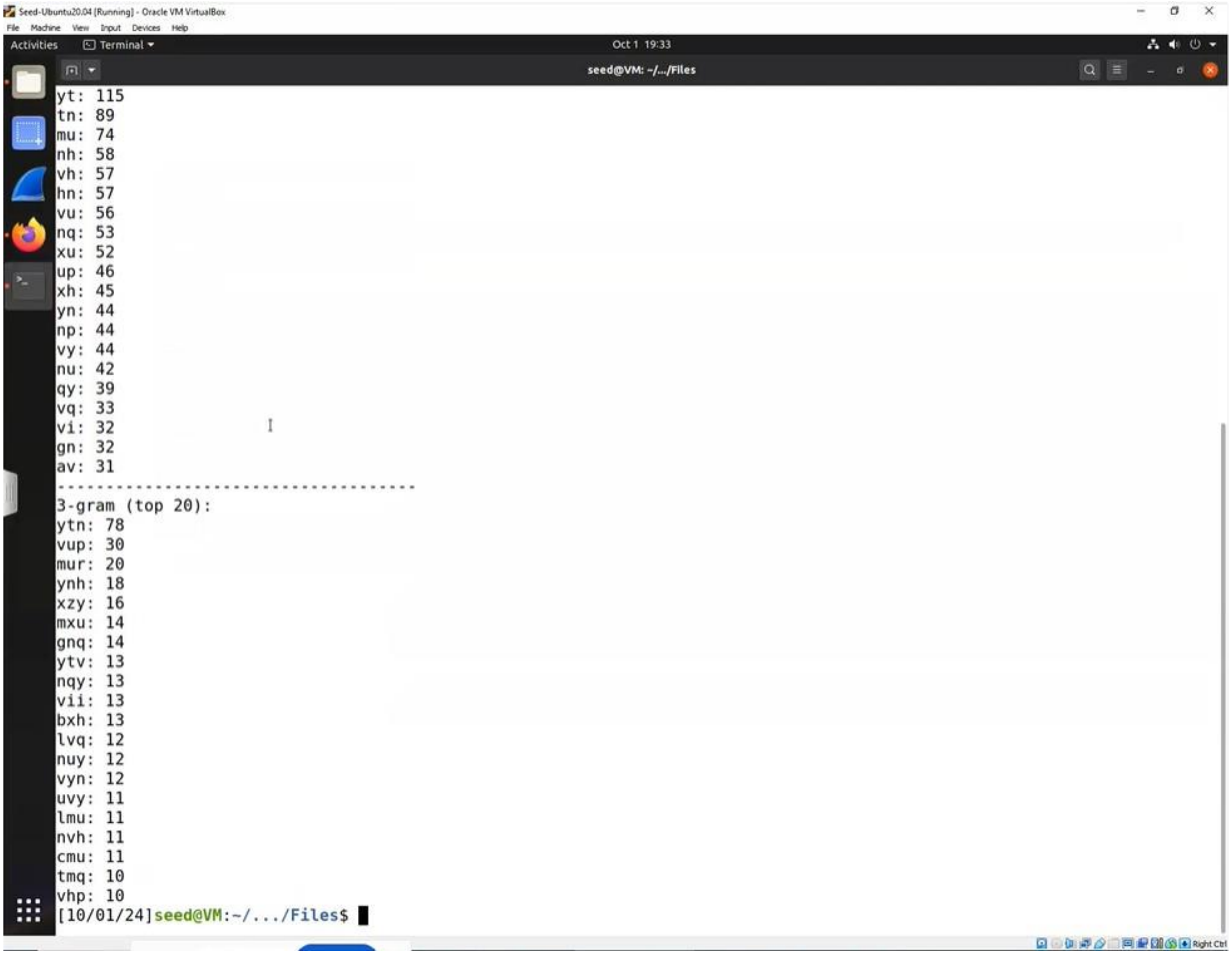
Task1 (freq.py)

```
[10/01/24]seed@VM:~/.../Files$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import random
>>> s = "abcdefghijklmnopqrstuvwxyz"
>>> list = random.sample(s, len(s))
>>> key = ''.join(list)
>>> print(key)
leyzfmcitjbxdnhswkvaqougpr
```

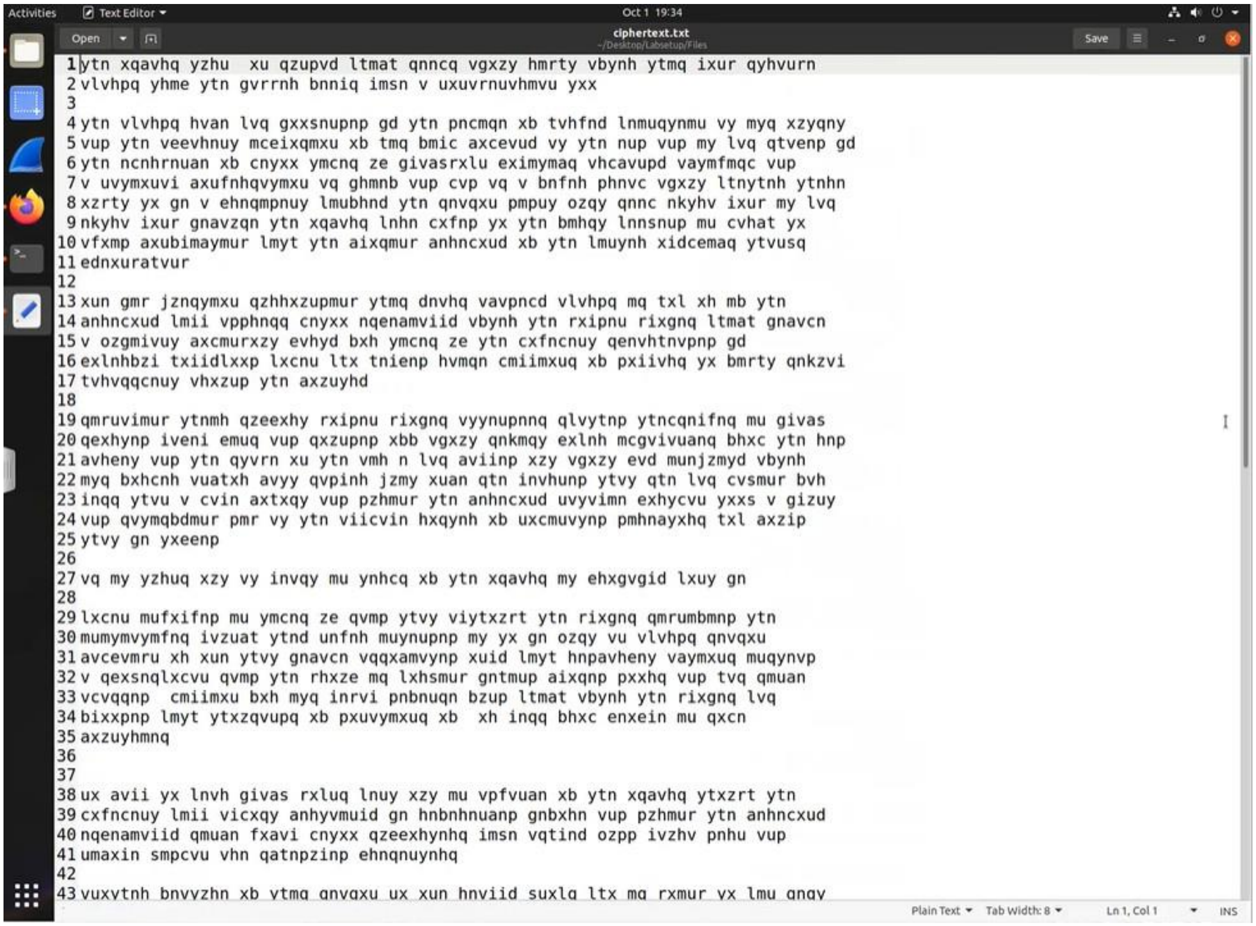
Task1 (imported random in order to generate key)



Task1 (This is the n-gram that was produced)



Task1 (This is the n-gram that was produced)



```
ytn xqavhq yzhu  xu qzupvd ltmat qnncq vgxzy hmrty vbynh ytmq ixur qyhvurn
vlvhpq yhme ytn gvrrnh bnniq imsn v uxuvrnuvhmvu yxx

ytn vlvhpq hvan lvq gxxsnupnp gd ytn pncmqn xb tvhfnd lnmuqynmu vy myq xzyqny
vup ytn veevhnuy mceixqmxu xb tmq bmic axcevud vy ytn nup vup my lvq qtvenp gd
ytn ncnhrnuan xb cnyxx ymcnq ze givasrxlu eximymaq vhcavupd vaymfmqc vup
v uvymxuvi axufnhqvymxu vq ghmnb vup cvp vq v bnfnh phnvc vgxzy ltnytnh ytnhn
xzrty yx gn v ehnqmpnuy lmubhnd ytn qnvqxu pmpuy ozqy qnnc nkyhv ixur my lvq
nkyhv ixur gnavzqn ytn xqavhq lnhn cxfnp yx ytn bmhqy lnnsnup mu cvhat yx
vfxmp axubimaymur lmyt ytn aixqmur anhncxud xb ytn lmuynh xidcemaq ytvusq
ednxuratvur

xun gmr jznqymxu qzhhxzupmur ytmq dnvhq vavpncd vlvhpq mq txl xh mb ytn
anhncxud lmii vpphnqq cnyxx nqenamviid vbynh ytn rxipnu rixgnq ltmat gnavcn
v ozgmivuy axcmurxzy evhyd bxh ymcnq ze ytn cxfncnuy qenvhtnvpnp gd
exlnhbzi txiidlxxp lxcnu ltx tnienp hvmqn cmiimxuq xb pxiivhq yx bmrty qnkzvi
tvhvqqcnuy vhxzup ytn axzuyhd

qmruvimur ytnmh qzeexhy rxipnu rixgnq vyynupnnq qlvytnp ytncqnifnq mu givas
qexhynp iveni emuq vup qxzupnp xbb vgxzy qnkmqy exlnh mcgvivuanq bhxc ytn hnp
avheny vup ytn qyvrn xu ytn vmh n lvq aviinp xzy vgxzy evd munjzmyd vbynh
myq bxhcnh vuatxh avyy qvpinh jzmy xuan qtn invhunp ytvy qtn lvq cvsmur bvh
inqq ytvu v cvin axtxqy vup pzhmur ytn anhncxud uvyvimn exhycvu yxxs v gizuy
vup qvymqbdmur pmr vy ytn viicvin hxqynh xb uxcmuvynp pmhnayxhq txl axzip
ytvy gn yxeenp

vq my yzhuq xzy vy invqy mu ynhcq xb ytn xqavhq my ehxgvgid lxuy gn

lxcnu mufxifnp mu ymcnq ze qvmp ytvy viytxzrt ytn rixgnq qmrumbmnp ytn
mumymvymfnq ivzuat ytnd unfnh muynupnp my yx gn ozqy vu vlvhpq qnvqxu
avcevmru xh xun ytvy gnavcn vqqxamvynp xuid lmyt hnpavheny vaymxuq muqynvp
v qexsnqlxcvu qvmp ytn rhxze mq lxhsmur gntmup aixqnp pxxhq vup tvq qmuan
vcvqqnp  cmiimxu bxh myq inrvi pnbnuqn bzup ltmat vbynh ytn rixgnq lvq
bixxpnp lmyt ytxzqvupq xb pxuvymxuq xb  xh inqq bhxc enxein mu qxcn
axzuyhmnq


ux avii yx lnvh givas rxluq lnuy xzy mu vpfvuan xb ytn xqavhq ytxzrt ytn
cxfncnuy lmii vicxqy anhyvmuid gn hnbnhnuanp gnbxhn vup pzhmur ytn anhncxud
nqenamviid qmuan fxavi cnyxx qzeexhynhq imsn vqtind ozpp ivzhv pnhu vup
umaxin smpcvu vhn qatnpzinp ehnqnuynhq

vuxvtnh bnvvzhn xb vtmq qnvqxu ux xun hnviid suxlq ltx mq rxmur vx lmu qnqv
```

Task1 (ciphertext)

Based on the n-gram, we can analyze the ciphertext.txt file. The encryption key could be as follows:

ytn → THE

y → T

t → H

n → E

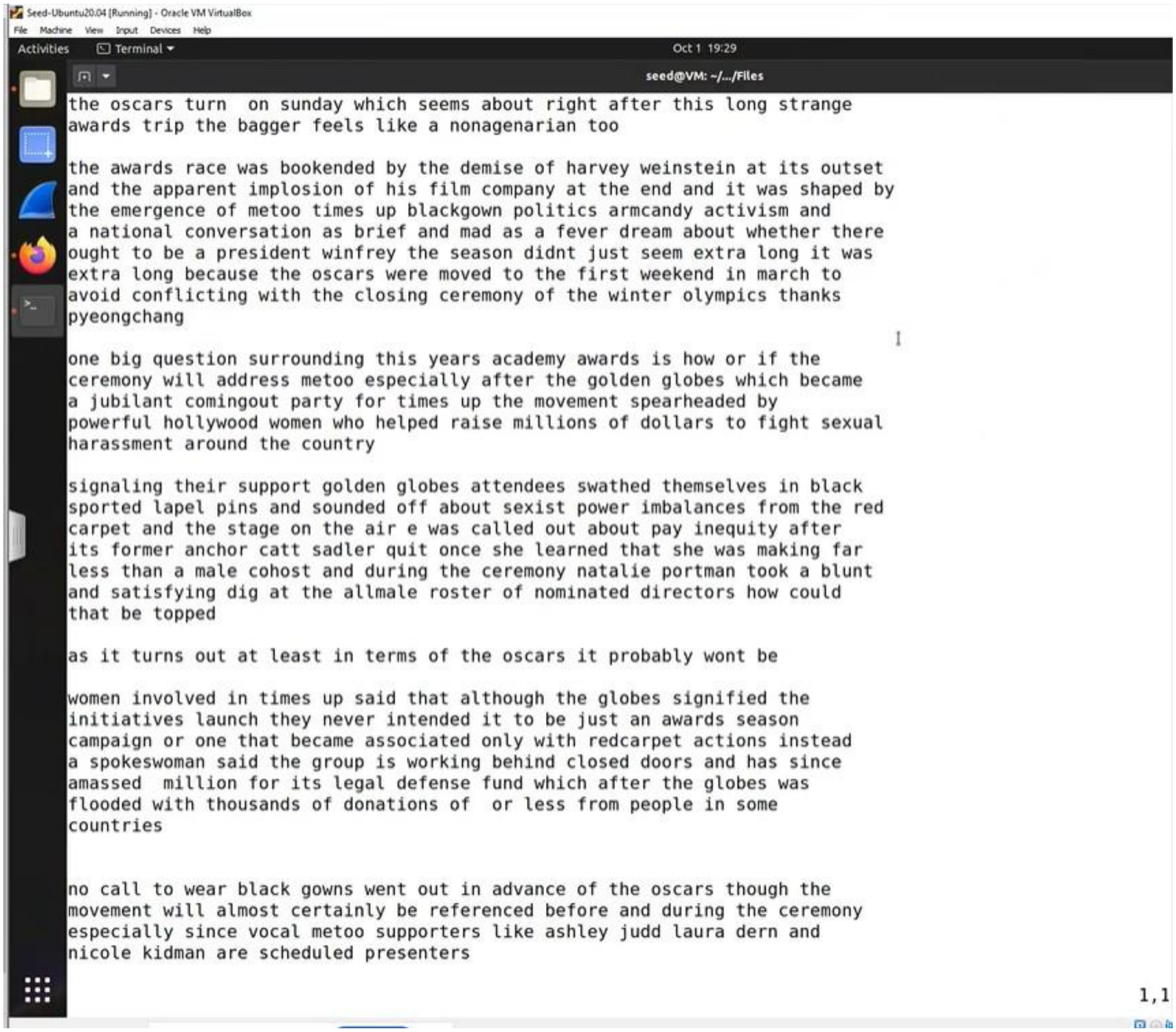
tn → HE

.

.

.

so on…



the oscars turn on sunday which seems about right after this long strange awards trip the bagger feels like a nonagenarian too

the awards race was bookended by the demise of harvey weinstein at its outset and the apparent implosion of his film company at the end and it was shaped by the emergence of metoo times up blackgown politics armcandy activism and a national conversation as brief and mad as a fever dream about whether there ought to be a president winfrey the season didnt just seem extra long it was extra long because the oscars were moved to the first weekend in march to avoid conflicting with the closing ceremony of the winter olympics thanks pyeongchang

one big question surrounding this years academy awards is how or if the ceremony will address metoo especially after the golden globes which became a jubilant comingout party for times up the movement spearheaded by powerful hollywood women who helped raise millions of dollars to fight sexual harassment around the country

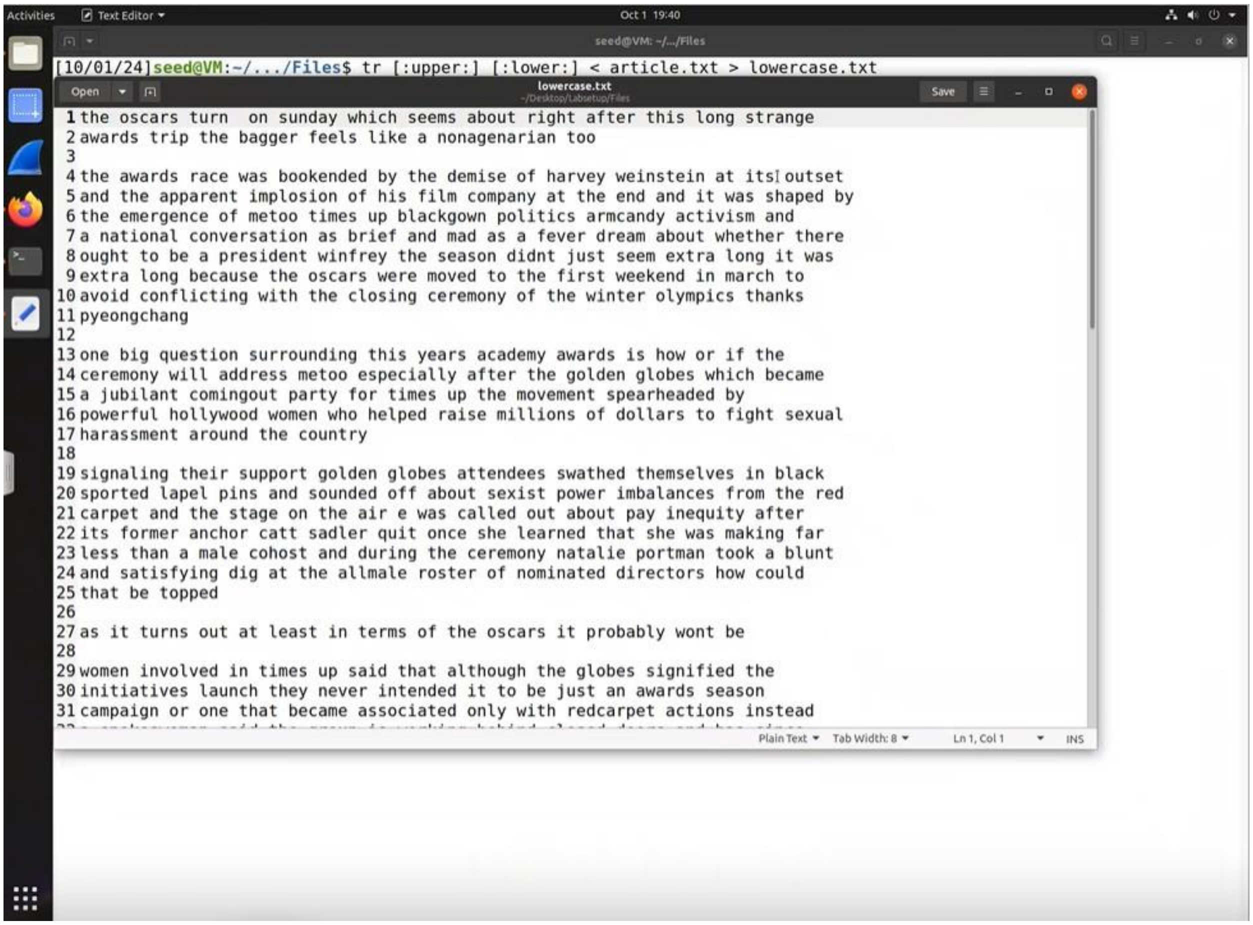
signaling their support golden globes attendees swathed themselves in black sported lapel pins and sounded off about sexist power imbalances from the red carpet and the stage on the air e was called out about pay inequity after its former anchor catt sadler quit once she learned that she was making far less than a male cohost and during the ceremony natalie portman took a blunt and satisfying dig at the allmale roster of nominated directors how could that be topped

as it turns out at least in terms of the oscars it probably wont be

women involved in times up said that although the globes signified the initiatives launch they never intended it to be just an awards season campaign or one that became associated only with redcarpet actions instead a spokeswoman said the group is working behind closed doors and has since amassed million for its legal defense fund which after the globes was flooded with thousands of donations of or less from people in some countries

no call to wear black gowns went out in advance of the oscars though the movement will almost certainly be referenced before and during the ceremony especially since vocal metoo supporters like ashley judd laura dern and nicole kidman are scheduled presenters

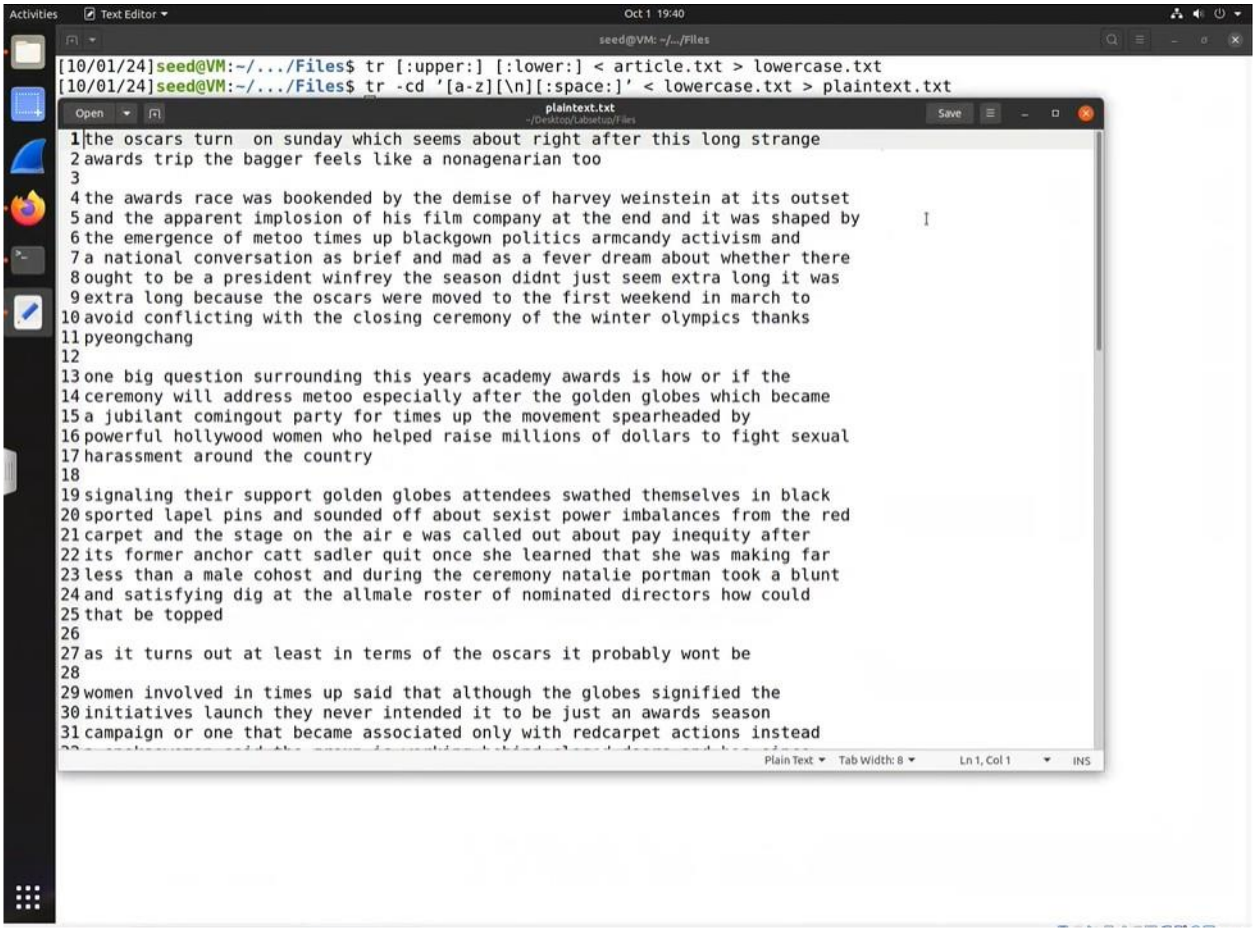
Task1 (decrypted ciphertext)



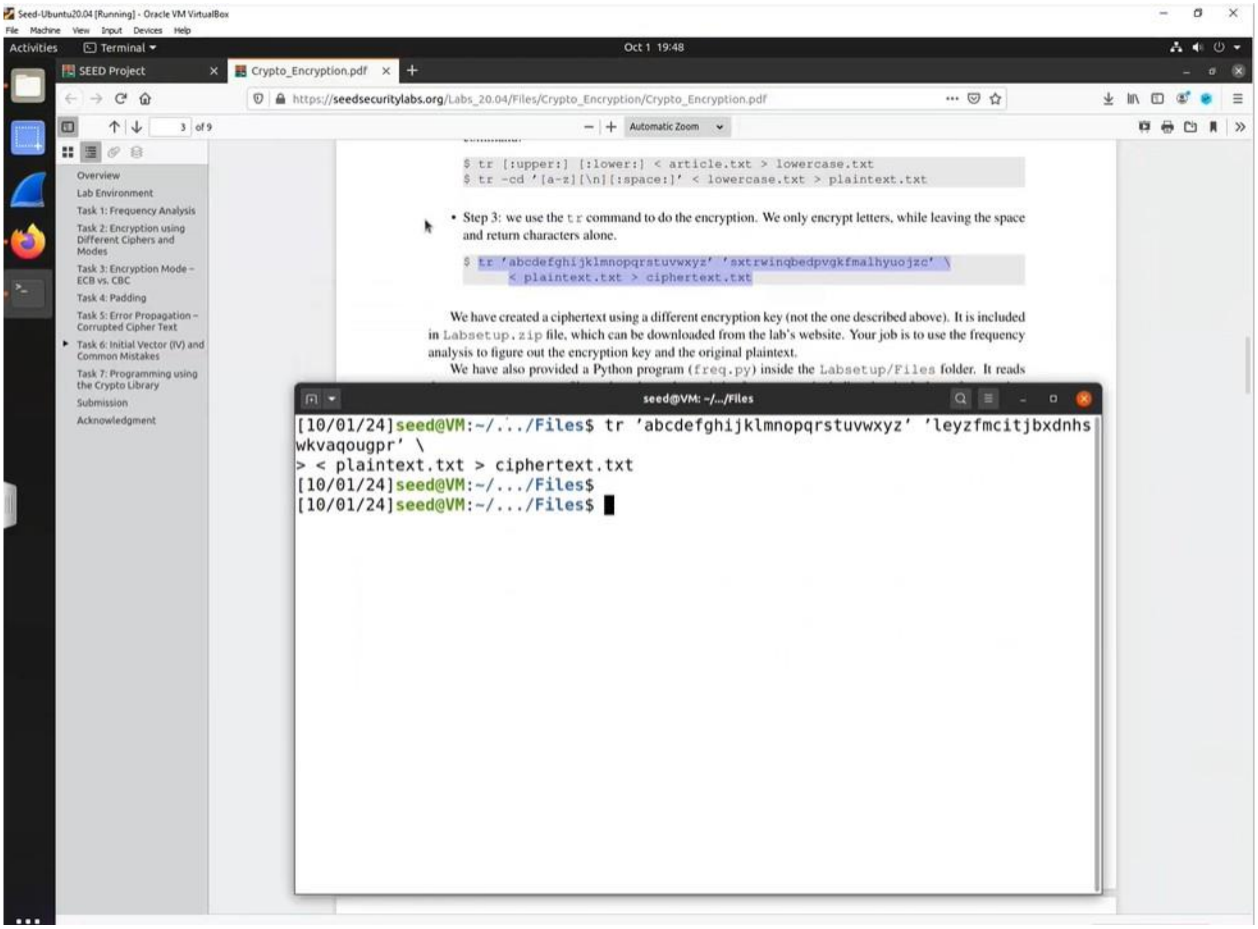
Task1 (tr [:upper:] [:lower:] < article.txt > lowercase.txt has been used)

The actual alphabet letter goes first, followed by the cipher key it corresponds to:

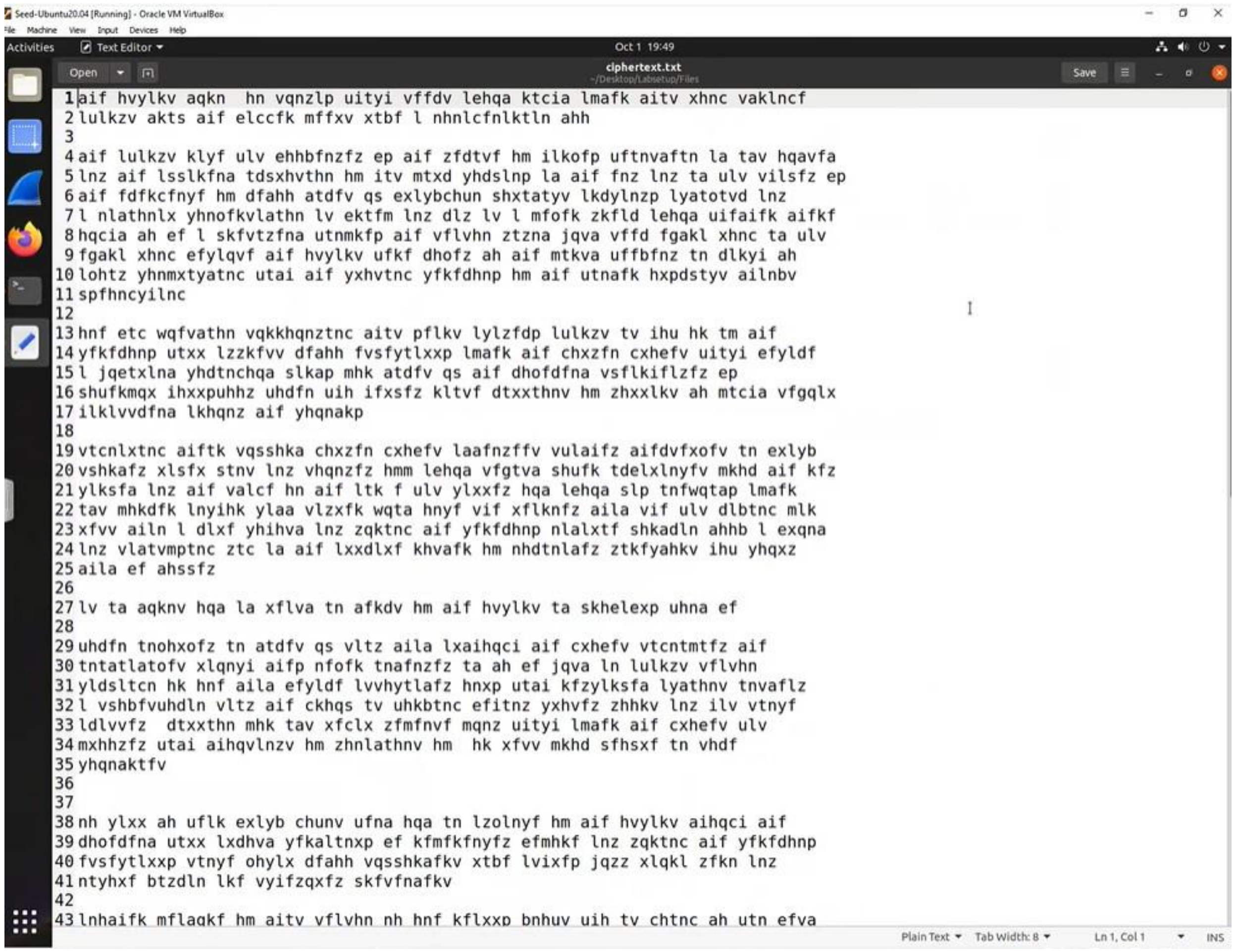
- A = V ● B = G ● C = A ● D = P ● E = N ● F = B ● G = R ● H = T ● I = M ● J = W
- K = S ● L = I ● M = C ● N = U ● O = X ● P = E ● Q = O ● R = H ● S = Q ● T = Y
- U = Z ● V = F ● W = L ● X = K ● Y = D ● Z = J



Task1 (tr -cd '[a-z][\n][:space:]' < lowercase.txt > plaintext.txt has been used)



Task1 (tr 'abcdefghijklmnopqrstuvwxyz' 'sxtrwinqbedpvgkfmalhyuojzc' \ < plaintext.txt > ciphertext.txt has been used)



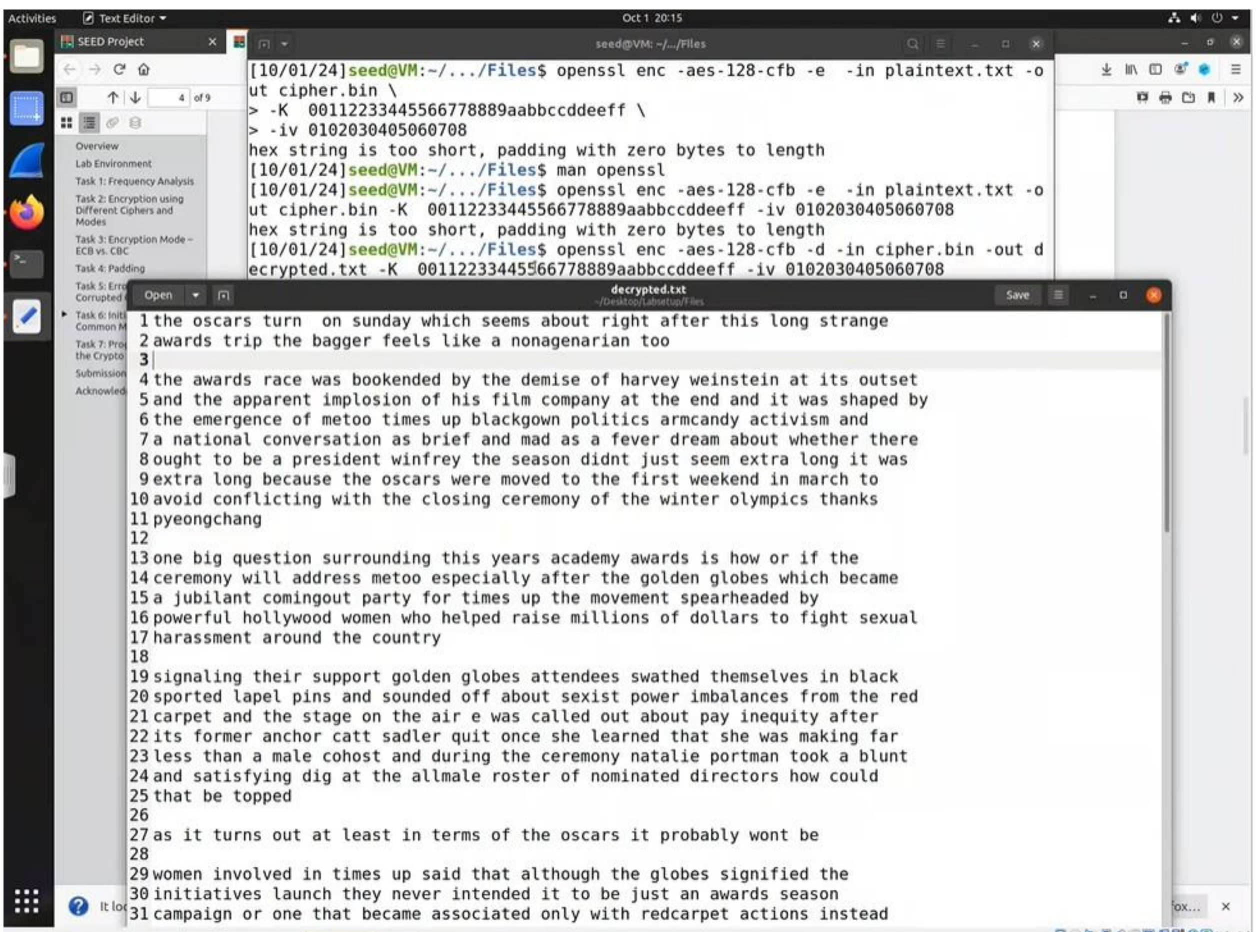
Task1 (we use the tr command to do the encryption. We only encrypt letters, while leaving the space and return characters alone)

Task 2: Encryption using Different Ciphers and Modes

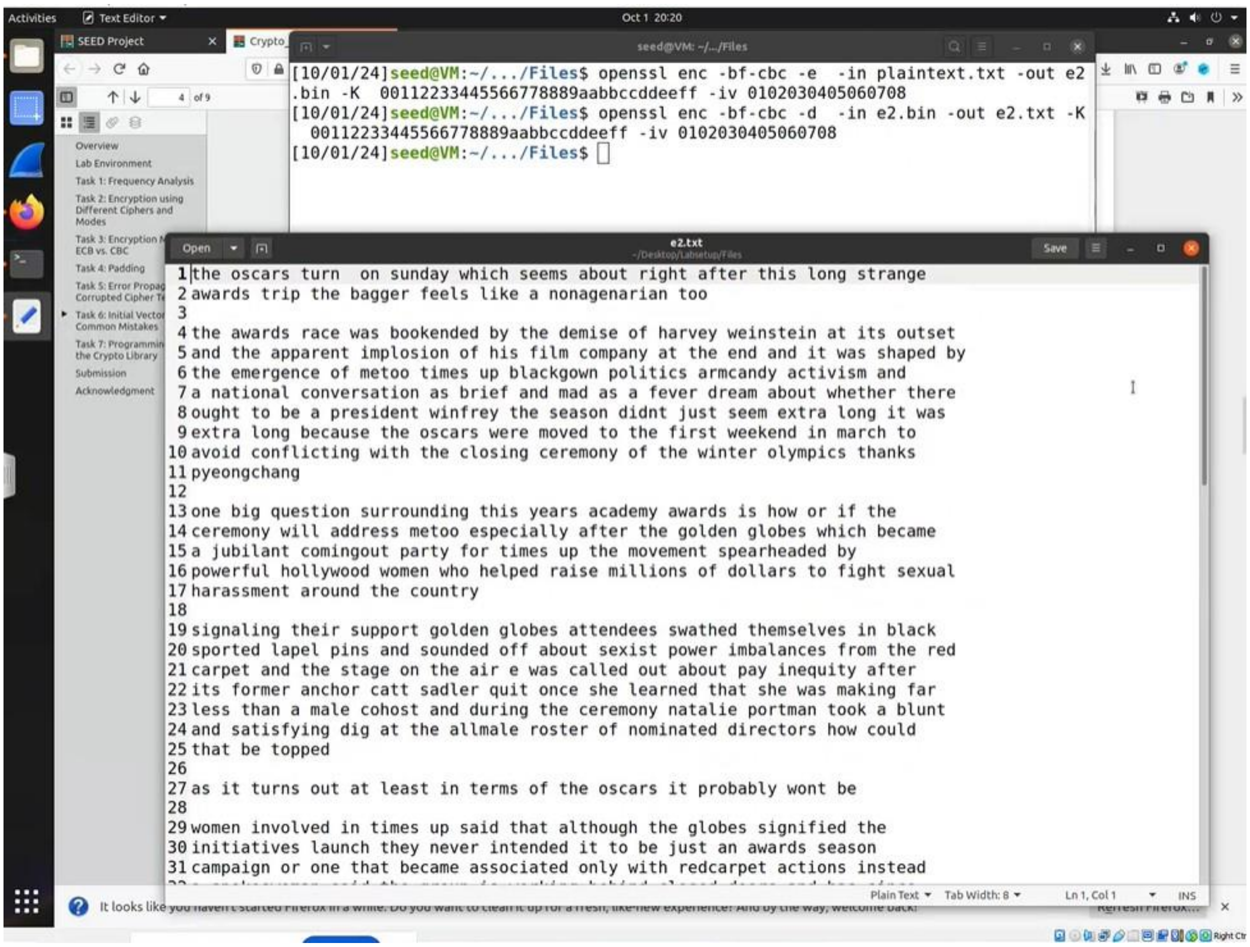
Description: This task involved experimenting with various encryption algorithms and modes using the OpenSSL tool. We encrypted plaintext using AES-128-CBC, AES-256-CBC, and AES-128-CFB to observe differences in how the encryption algorithms and modes affect the ciphertext.

Observations:

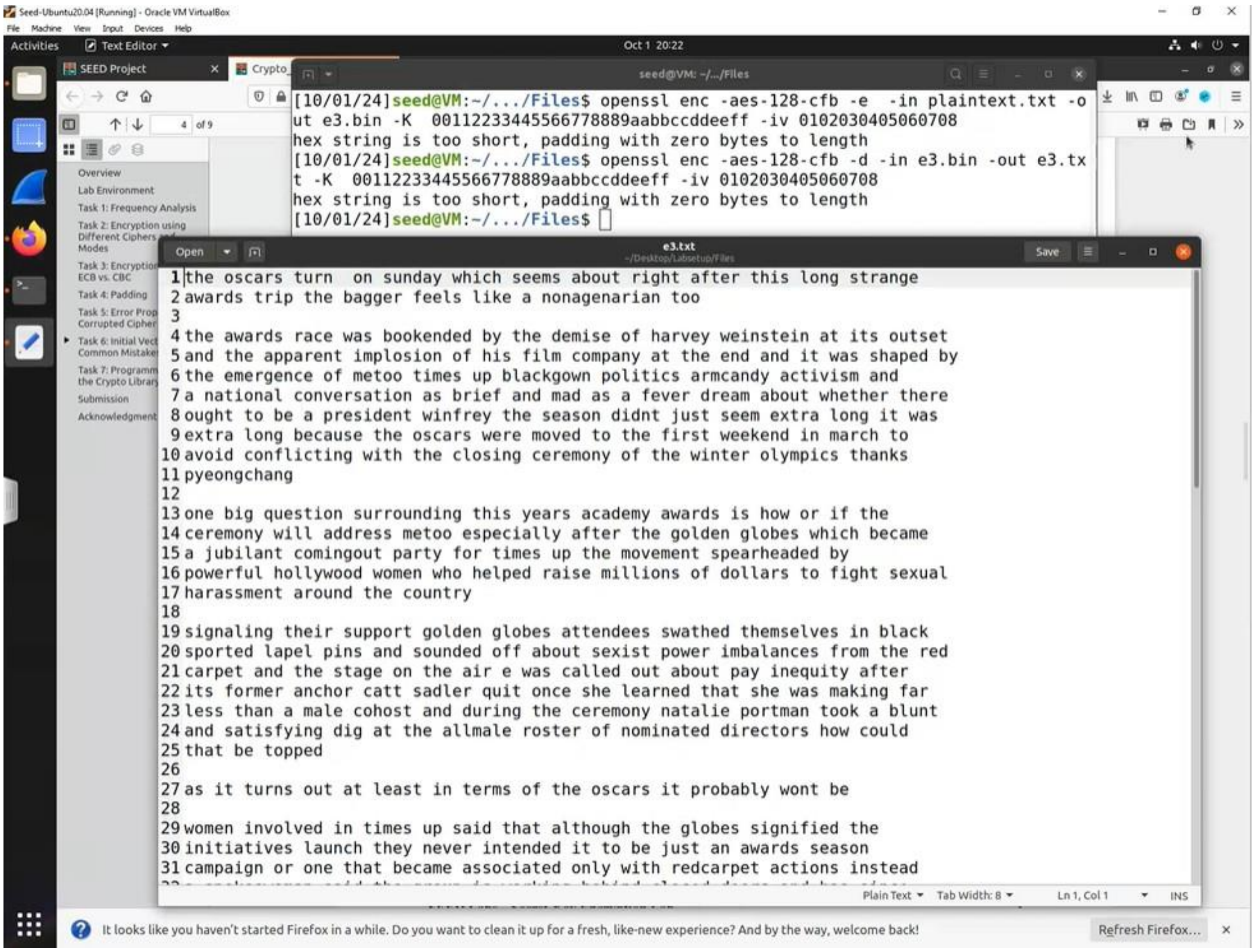
- AES-128-CBC: Standard encryption with 128-bit blocks and a 128-bit key. It required a unique initialization vector (IV) to ensure randomness.
- AES-256-CBC: Similar to AES-128-CBC but with a larger key size of 256 bits, making it more secure but computationally intensive.
- AES-128-CFB: A stream cipher mode that doesn't require padding, allowing plaintext to be encrypted bit by bit.



Task2 (openssl enc command used to encrypt/decrypt a file named decrypted.txt)



Task2 (generated a file named e2.txt)



Task2 (generated a file named e3.txt)

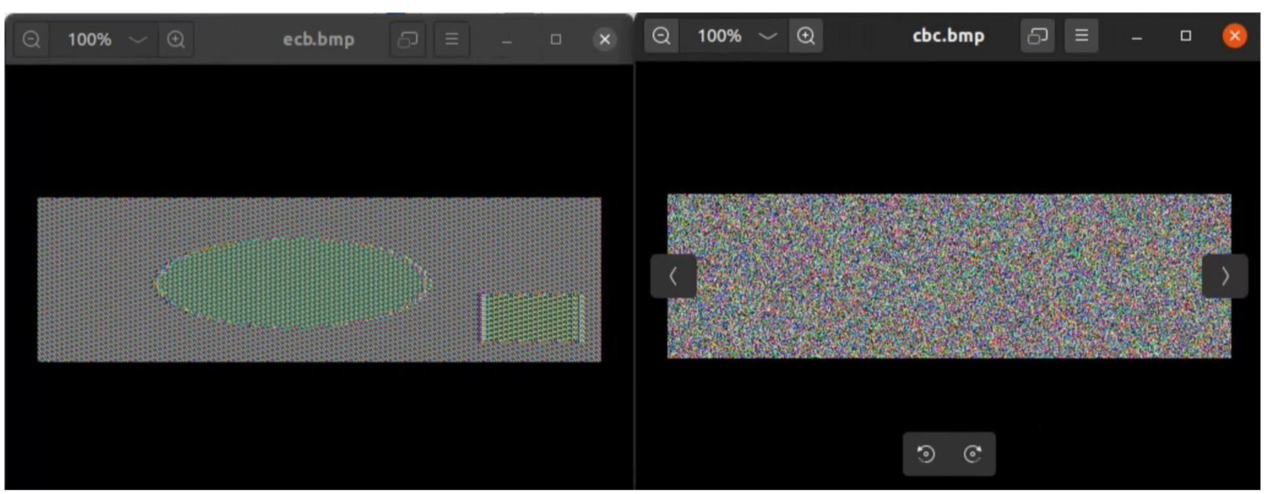
Task 3: Encryption Mode – ECB vs. CBC

Description: The objective of this task was to compare two encryption modes, Electronic Code Book (ECB) and Cipher Block Chaining (CBC), by encrypting image files and analyzing the results.

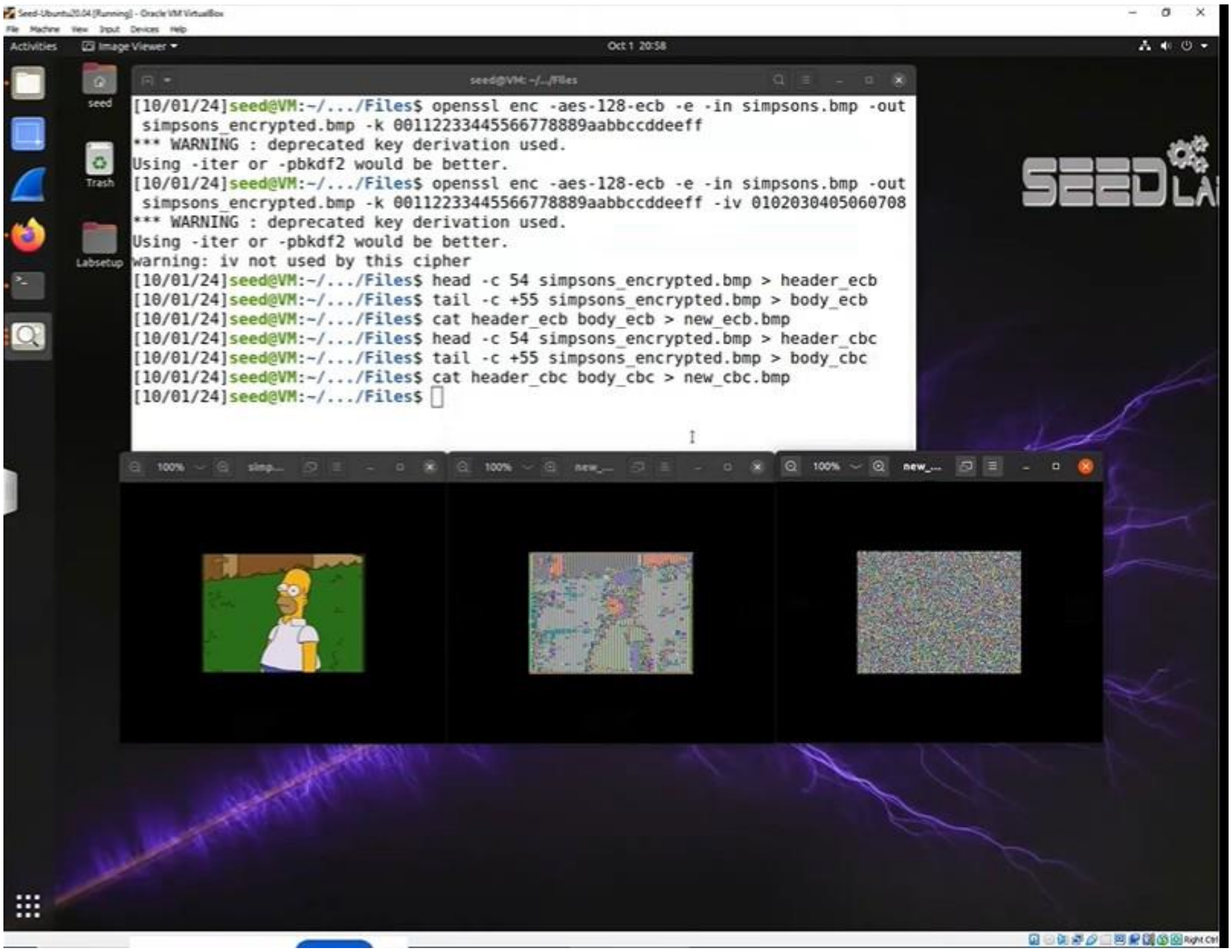
Observations:

- ECB Mode: The image encrypted with ECB showed discernible outlines, indicating that ECB does not hide patterns well, making it less secure for encrypting repetitive data.
- CBC Mode: The image encrypted with CBC was highly distorted, showing that CBC mode introduces more randomness, which results in better security as it hides the structure of the plaintext.

Task3 (we gathered the secret value and the number of hits)



Task3 (for an ecb encryption, we still observe the shapes in the picture, even though the color is heavily altered. For the cbc encryption, we can’t see anything related to the picture, and so the cbc-style encryption is stronger.)



Task3-3 (We went through the encryption process again)

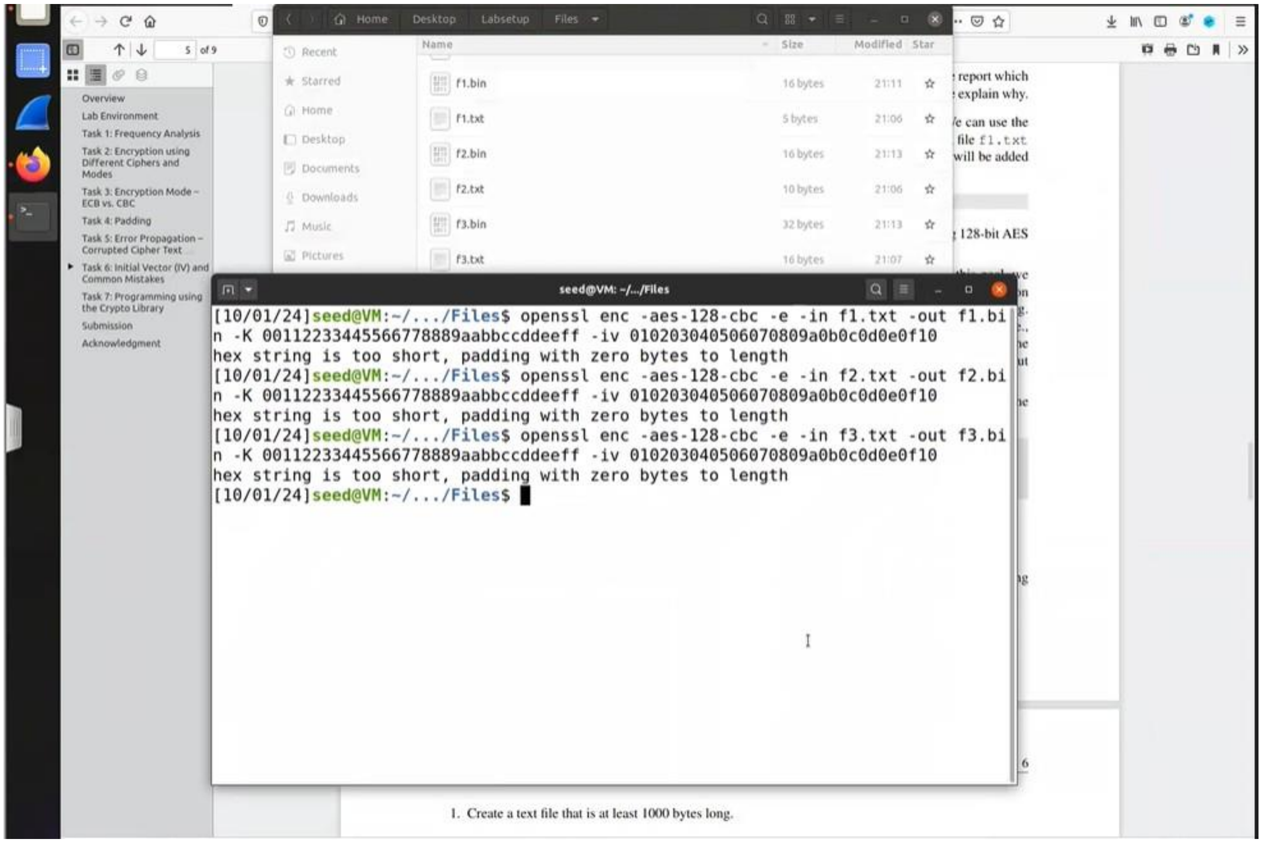
Task 4: Padding

Description: In this task, we explored how padding affects encryption when the size of the plaintext is not a multiple of the block size. We tested different encryption modes (ECB, CBC, CFB, OFB) to see which ones required padding and how padding was handled during encryption.

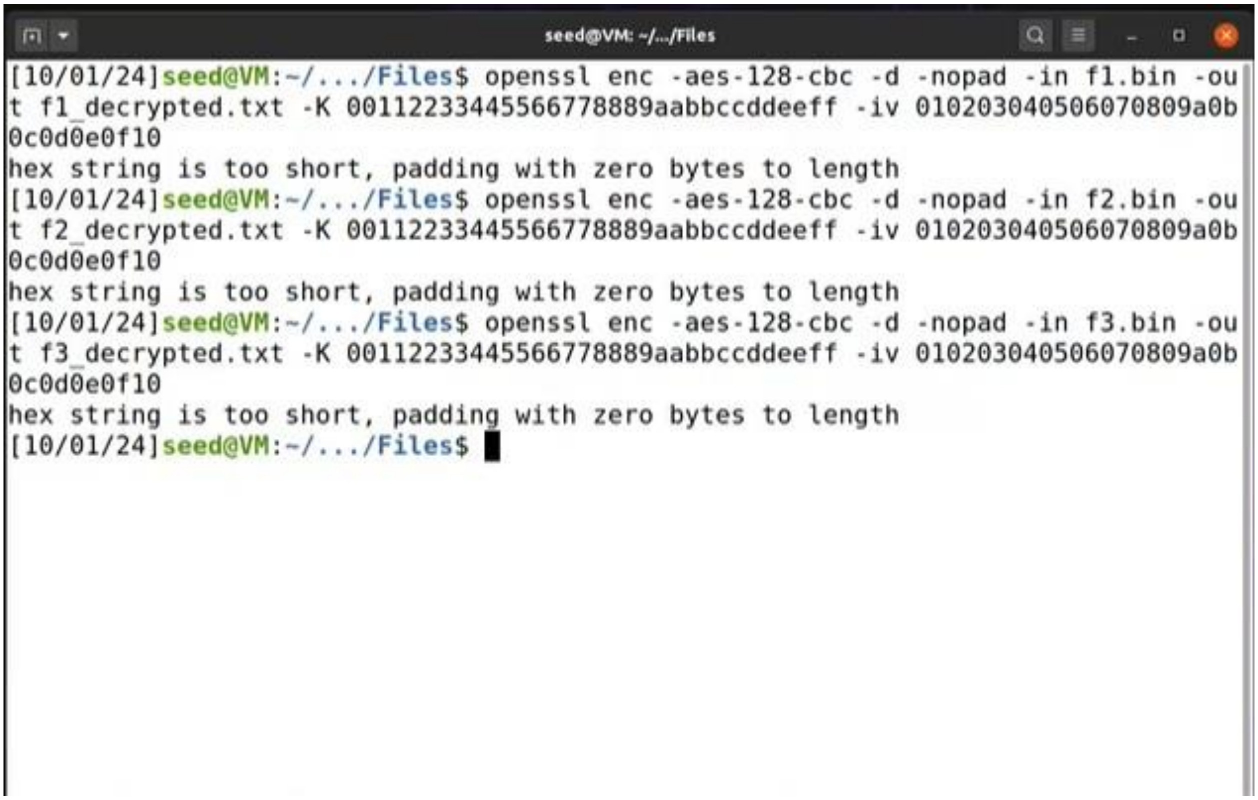
Observations:

- CBC and ECB: Both block ciphers required padding, and extra bytes were added to the plaintext to make its size a multiple of the block size.
- CFB and OFB: These stream cipher modes do not require padding as they encrypt data bit by bit or in smaller portions.

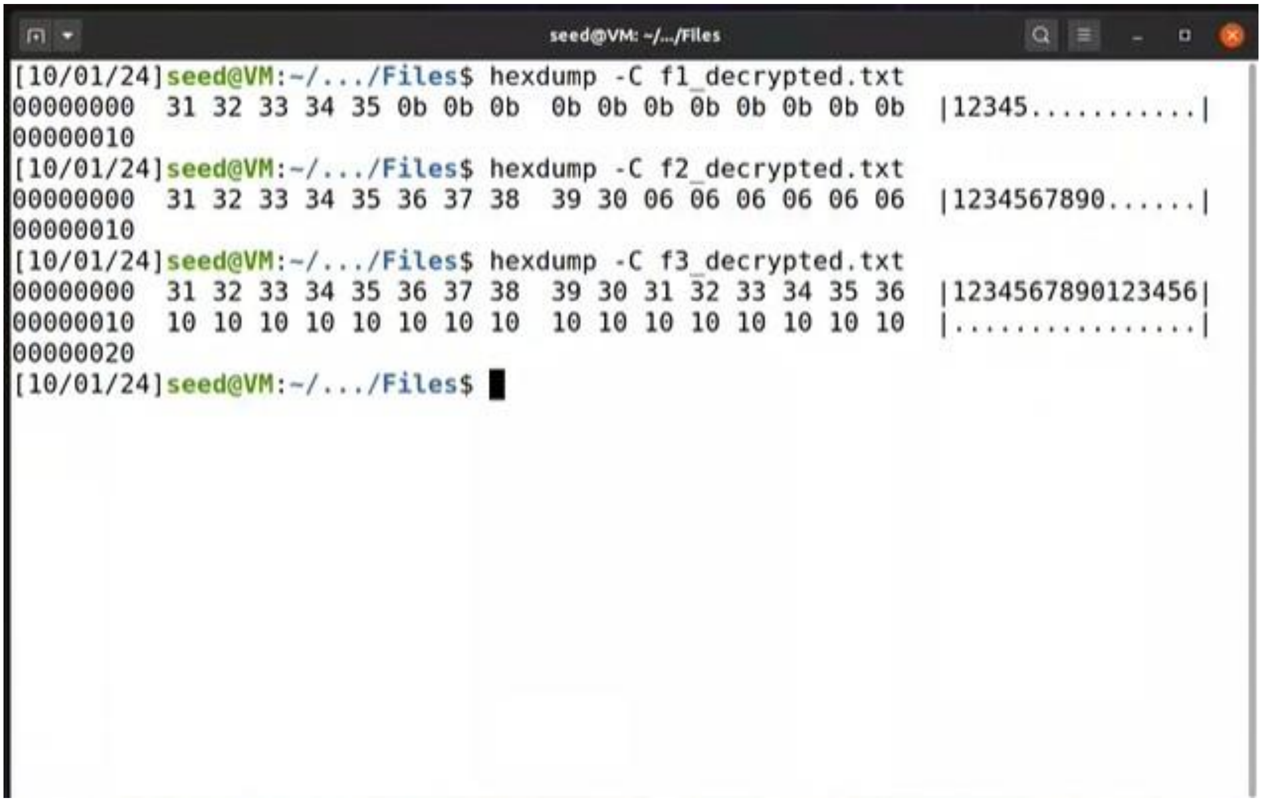
Task4 (3 files were created. File f1.txt is 5 bytes, f2.txt is 10 bytes and f3.txt is 16 bytes.)



Task4 (f1.bin, f2.bin, f3.bin files are a result of the encryption process from the code above. Those files are seen above behind the shell. The size of f1.bin is 16 bytes, f2.bin is 16 bytes and f3.bin is 32 bytes. It seems that there was some padding added to the files during the encryption process.)



Task4 (We did decrypt the file, using this code)



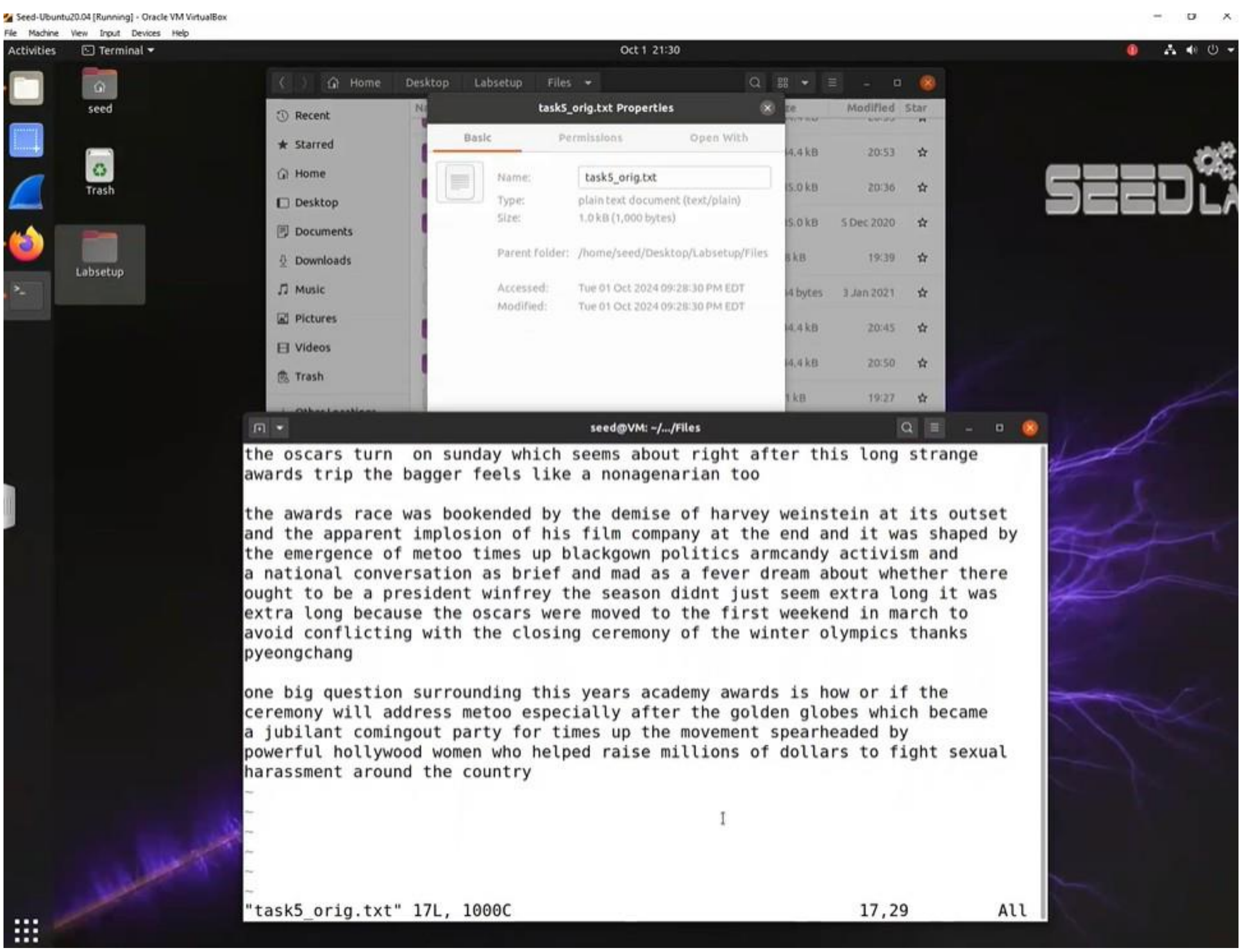
Task4 (These are the contents of the padding for the decrypted files)

Task 5: Error Propagation – Corrupted Cipher Text

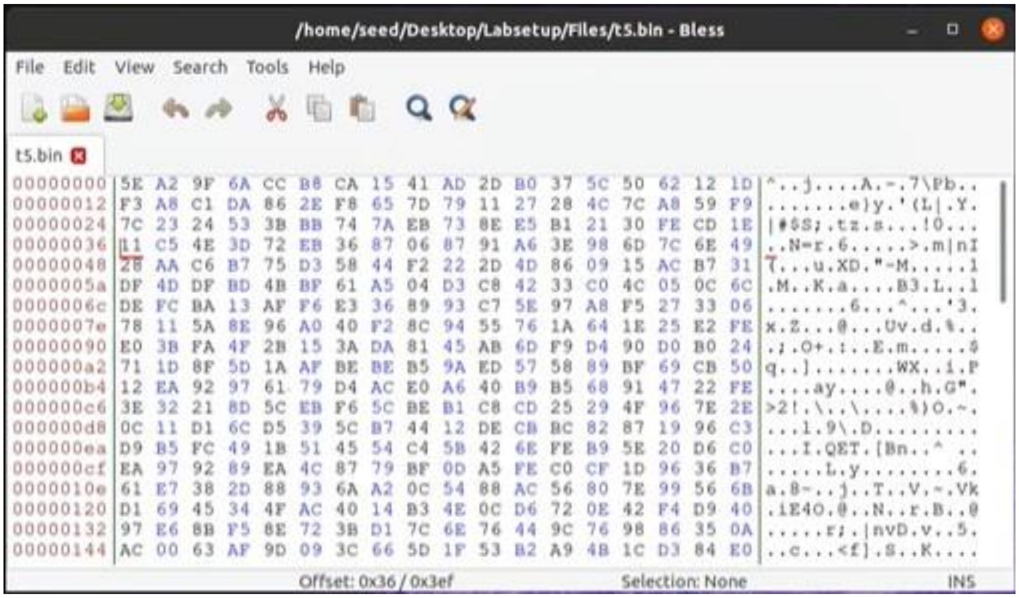
Description: This task demonstrated how errors in the ciphertext propagate during decryption, depending on the encryption mode used. We corrupted a single bit in the ciphertext and observed how much of the plaintext could still be recovered after decryption.

Observations:

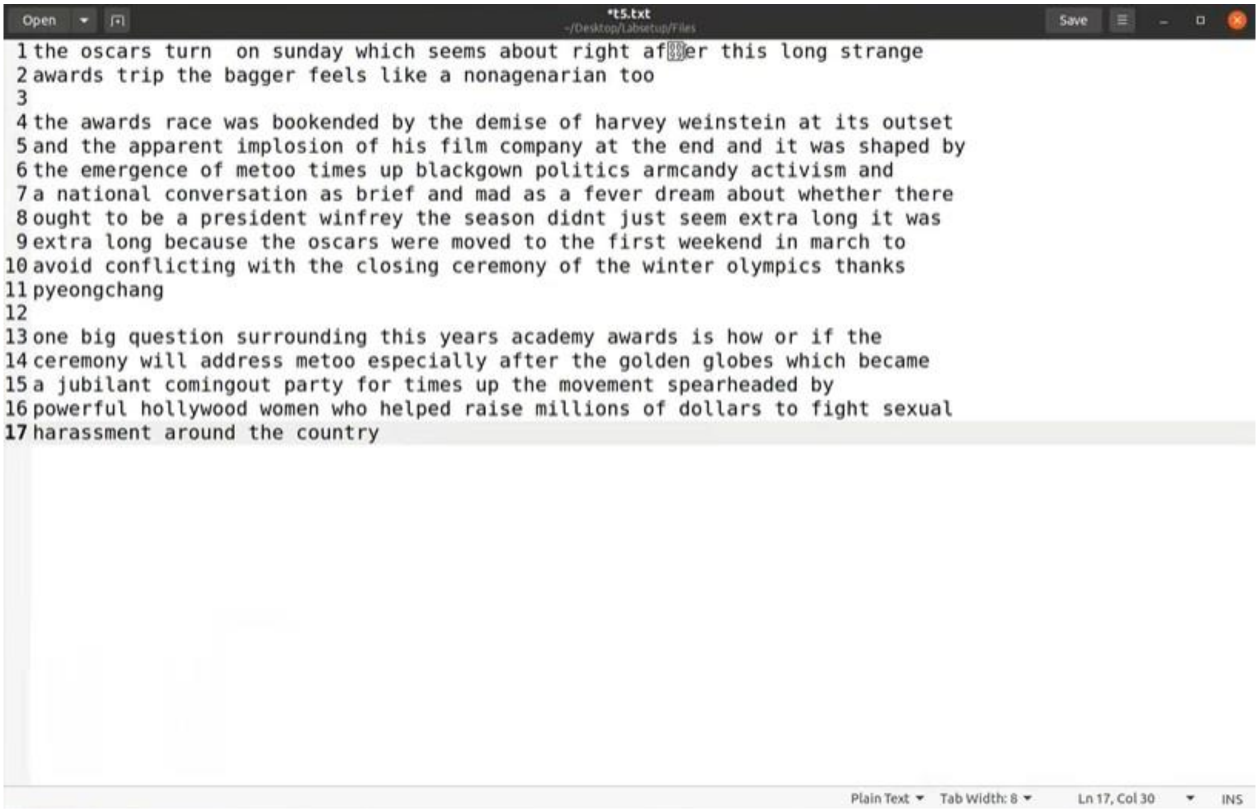
- ECB Mode: Only the corrupted block was affected, allowing us to recover most of the plaintext except for the corrupted block.
- CBC Mode: Error propagation affected the current and subsequent blocks, making more of the decrypted text unreadable.
- CFB and OFB Modes: These modes behaved similarly, with errors affecting only the current and some subsequent bits, but not entire blocks.



Task5 (One file was created with 1000 bytes and these are the contents of the file)

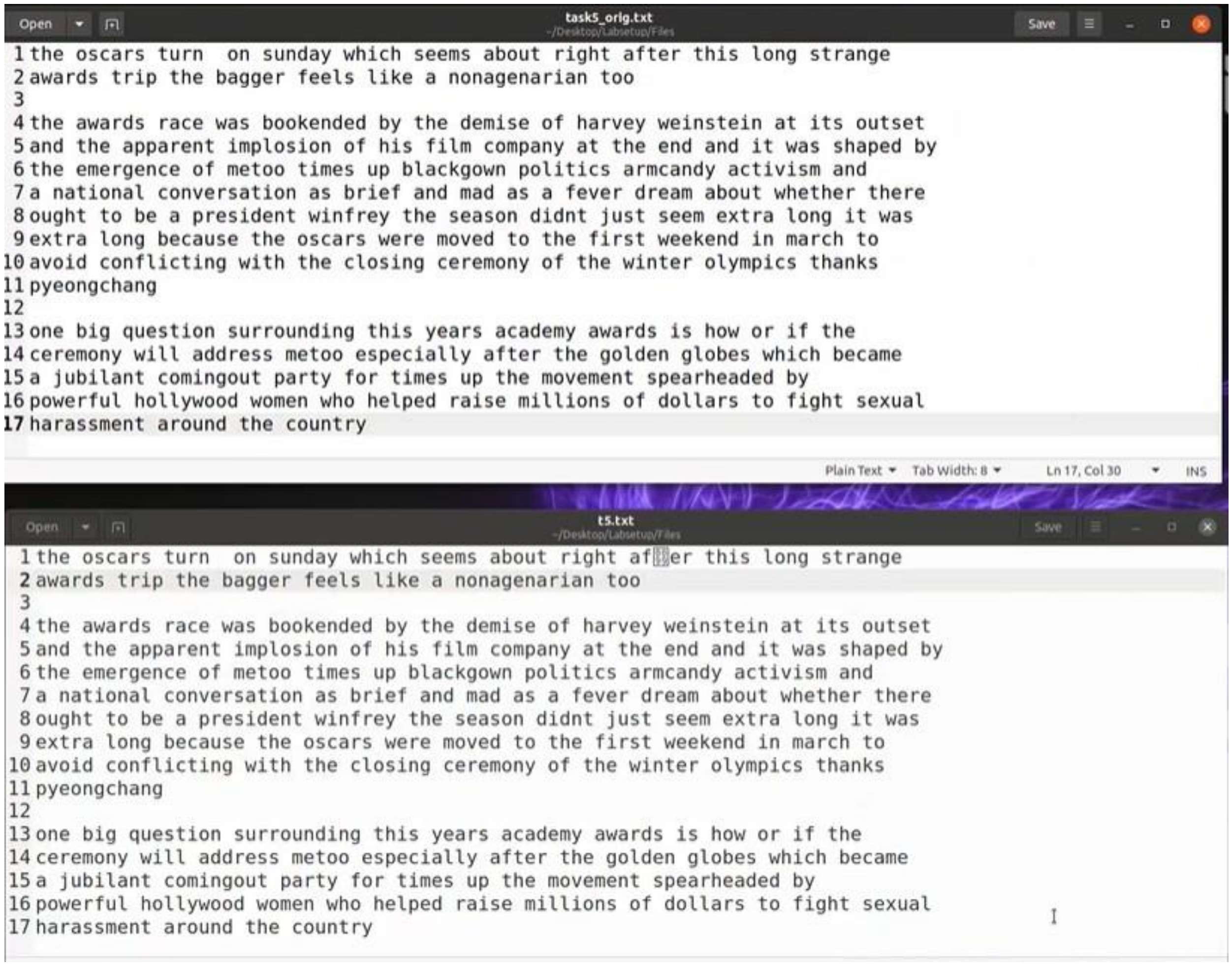


Task5 (The file was encrypted with ECB and then the 55th bit was corrupted)




```
the oscars turn  on sunday which seems about right af�er this long strange
awards trip the bagger feels like a nonagenarian too

the awards race was bookended by the demise of harvey weinstein at its outset
and the apparent implosion of his film company at the end and it was shaped by
the emergence of metoo times up blackgown politics armcandy activism and
a national conversation as brief and mad as a fever dream about whether there
ought to be a president winfrey the season didnt just seem extra long it was
extra long because the oscars were moved to the first weekend in march to
avoid conflicting with the closing ceremony of the winter olympics thanks
pyeongchang

one big question surrounding this years academy awards is how or if the
ceremony will address metoo especially after the golden globes which became
a jubilant comingout party for times up the movement spearheaded by
powerful hollywood women who helped raise millions of dollars to fight sexual
harassment around the country
```

Task5 (corrupted file)



Task5 (encrypted decrypted file is on the bottom and the original file is in the top)

Task 6: Initial Vector (IV) and Common Mistakes

Description: This task focused on the role of the initialization vector (IV) in encryption. We experimented with the consequences of reusing the same IV or using predictable IVs in encryption.

Observations:

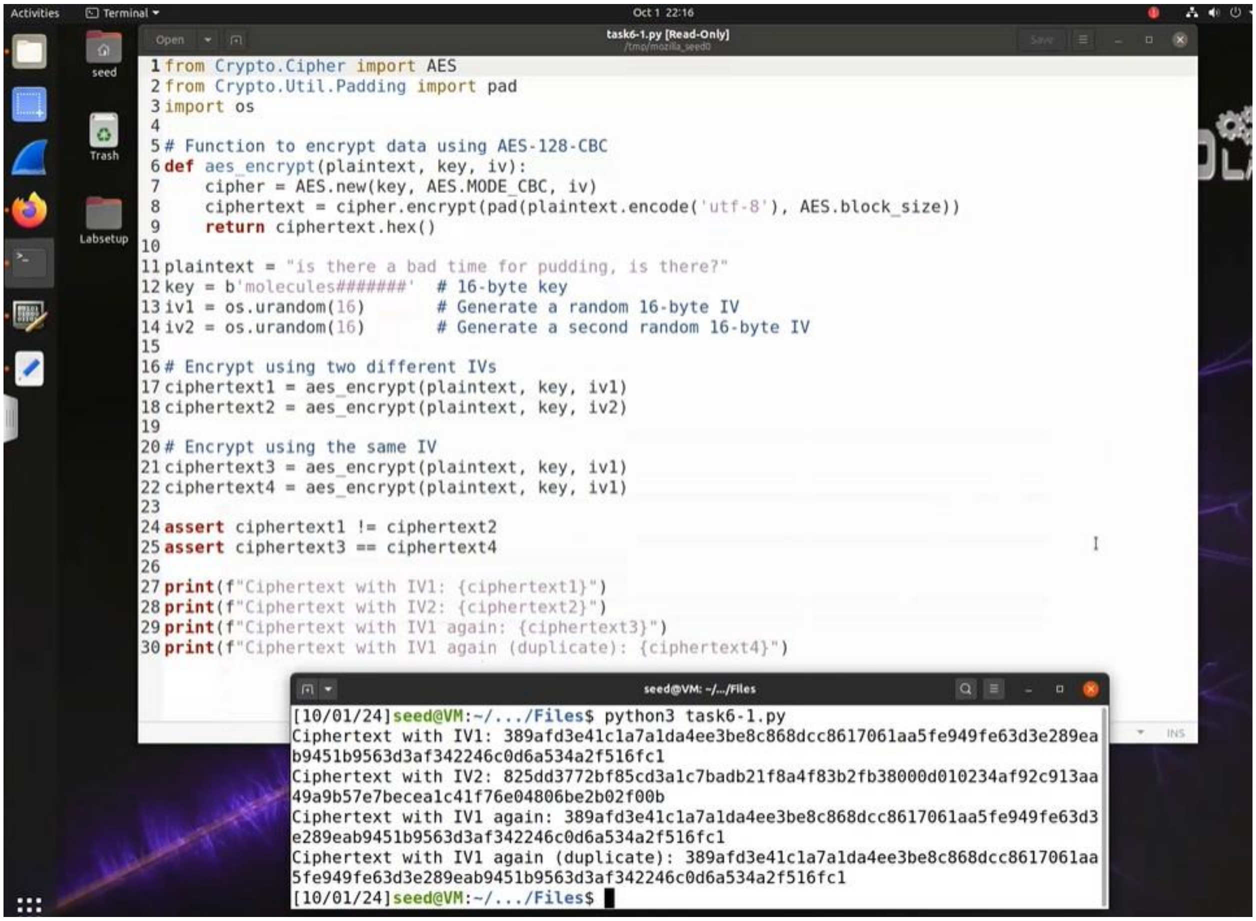
- IV Uniqueness: When different IVs were used for the same plaintext, the resulting ciphertexts were distinct, ensuring confidentiality.
- IV Reuse: Reusing the same IV resulted in identical ciphertexts for the same plaintext, creating a security vulnerability where attackers could infer patterns.
- Predictable IVs: If IVs are predictable, attackers can carry out attacks like the chosen-plaintext attack, leading to potential exposure of sensitive information.

Task 6.1. IV Experiment

Description: In this task, we experimented with the importance of using a unique initialization vector (IV) for each encryption session. We encrypted the same plaintext using two different IVs and then used the same IV to encrypt the same plaintext again to observe the differences.

Observations:

- When different IVs were used, the resulting ciphertexts were entirely different, even though the plaintext and key were identical. This demonstrates that the IV introduces randomness, making the ciphertext unpredictable and secure.
- When the same IV was reused, the ciphertexts were identical, revealing that reusing IVs can lead to vulnerabilities, as attackers may recognize patterns and infer information about the plaintext.



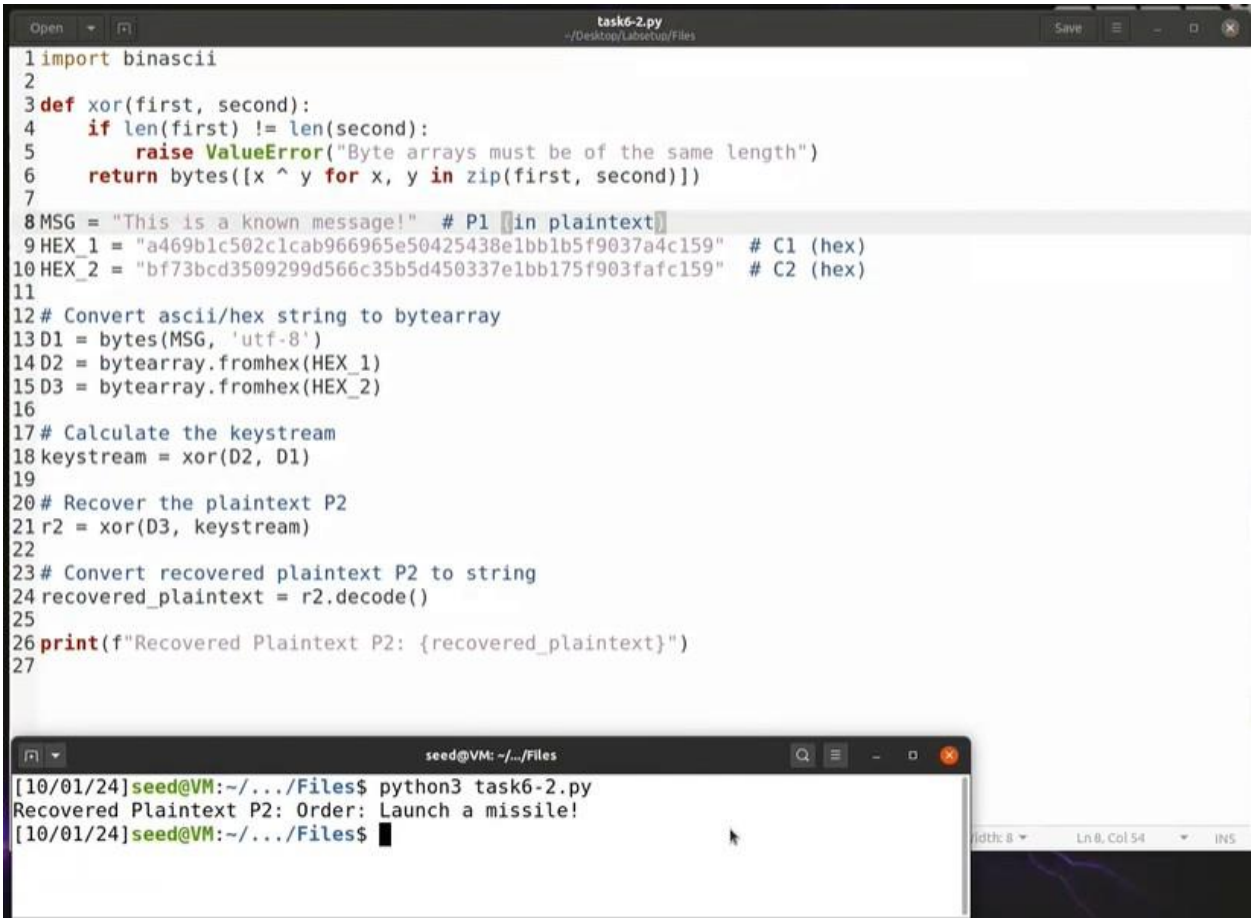
Task6-1 (encrypted the same plaintext using two different IVs)

Task 6.2. Common Mistake: Use the Same IV

Description: This task explored the security risks associated with reusing the same IV for different plaintexts. Using the Output Feedback (OFB) mode, we examined how the reuse of an IV could allow attackers to decrypt subsequent messages when they have access to the previous plaintext and ciphertext.

Observations:

- Reusing the same IV under OFB mode allowed us to infer the keystream, which, combined with the known plaintext and ciphertext, enabled us to partially decrypt another message encrypted with the same IV.
- This experiment illustrated how reusing IVs can significantly weaken encryption, as attackers can perform a known-plaintext attack, deducing the contents of new messages based on old ones.



```
import binascii

def xor(first, second):
    if len(first) != len(second):
        raise ValueError("Byte arrays must be of the same length")
    return bytes([x ^ y for x, y in zip(first, second)])

MSG = "This is a known message!"  # P1 (in plaintext)
HEX_1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159"  # C1 (hex)
HEX_2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc159"  # C2 (hex)

# Convert ascii/hex string to bytearray
D1 = bytes(MSG, 'utf-8')
D2 = bytearray.fromhex(HEX_1)
D3 = bytearray.fromhex(HEX_2)

# Calculate the keystream
keystream = xor(D2, D1)

# Recover the plaintext P2
r2 = xor(D3, keystream)

# Convert recovered plaintext P2 to string
recovered_plaintext = r2.decode()

print(f"Recovered Plaintext P2: {recovered_plaintext}")
```

```
[10/01/24]seed@VM:~/.../Files$ python3 task6-2.py
Recovered Plaintext P2: Order: Launch a missile!
[10/01/24]seed@VM:~/.../Files$
```

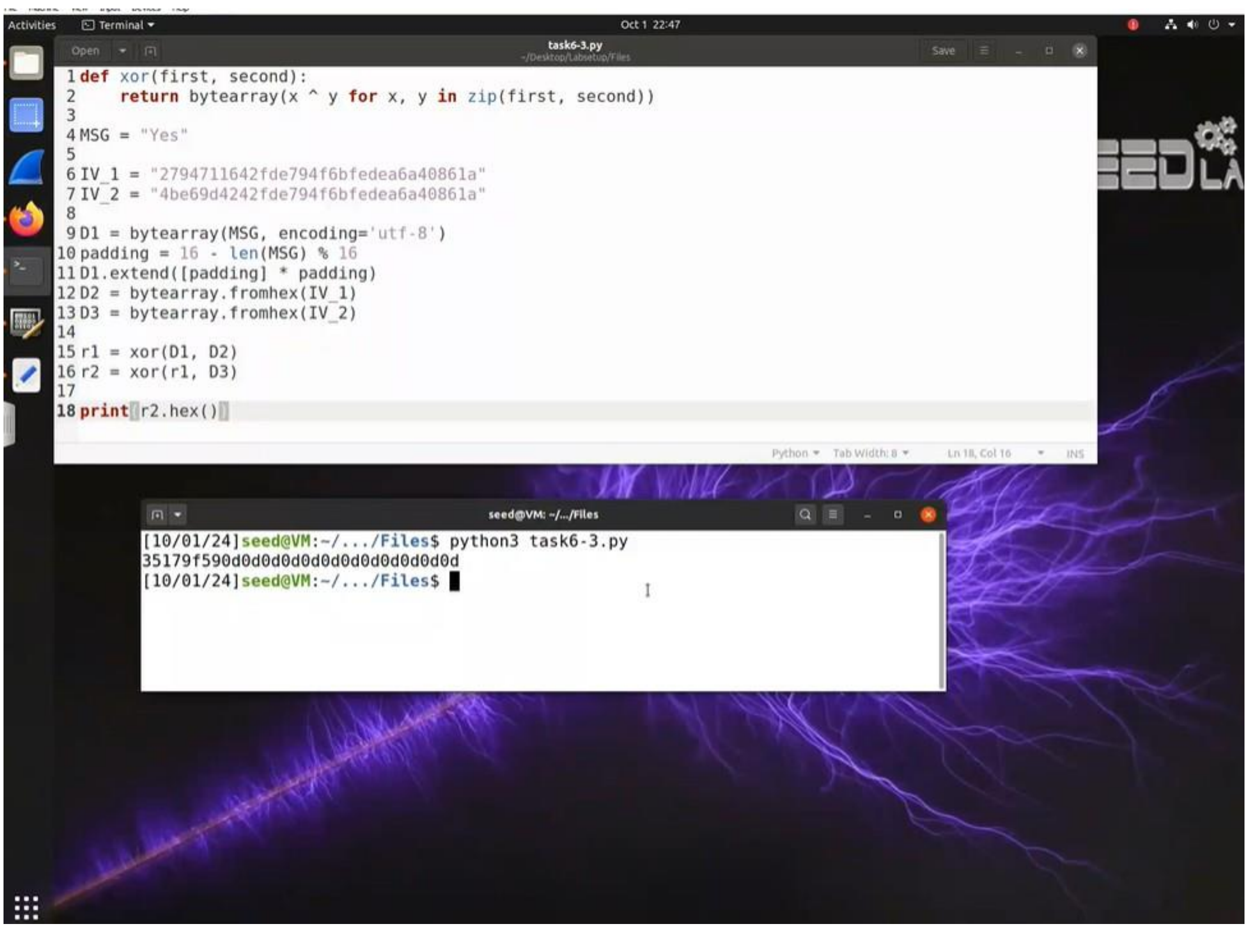
Task6-2 (In this case, if CFB mode is used instead of OFB, the level of exposure of the plaintext P2 would depend significantly on the keystream derived from the known plaintext P1 and its corresponding ciphertext C1. In CFB mode, each block of ciphertext is produced by encrypting the previous block of ciphertext and XORing the result with the current block of plaintext. Therefore, the first block of ciphertext is influenced by the IV, while subsequent blocks depend on the preceding ciphertext block. Since the IV is reused and P1 and C1 are known, the keystream generated by encrypting the IV can be recovered. This derived keystream enables the decryption of the first block of any other message encrypted with the same IV, revealing not only the first block of P2 but potentially more, especially if patterns in the plaintext continue across blocks.)

Task 6.3. Common Mistake: Use a Predictable IV

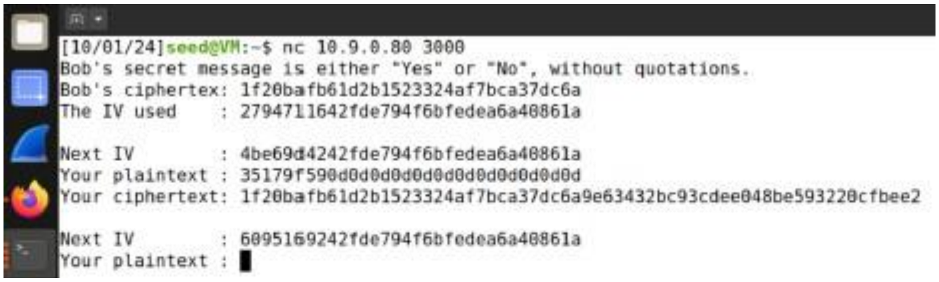
Description: This task examined the risks of using predictable IVs in encryption. We simulated an attack where Eve, knowing that Bob's IVs were predictable, attempted to determine whether Bob's encrypted message was "Yes" or "No" by carefully choosing the plaintext to submit for encryption and analyzing the resulting ciphertexts.

Observations:

- The predictability of the IV allowed us to craft a plaintext that, when encrypted with Bob's IV, resulted in the same ciphertext as Bob's secret message. This confirmed that the secret message was either "Yes" or "No."
- This task demonstrated that predictable IVs enable attackers to launch chosen-plaintext attacks, where they can manipulate inputs to reveal sensitive information from encrypted messages.



Task6-3(task6-3.py code has been used and the resulting plaintext hex code is below the code)



Task6-3 (Bob's ciphertext (1f20bafb61d2b1523324af7bca37dc6a) was given. I then submitted the plaintext 351f795d00d0d0d0d0d0d0d0d0d0d0, and the resulting ciphertext matched Bob's, confirming that my submitted plaintext was the same as Bob's secret message. This matching ciphertext suggests the message is either "Yes" or "No" and likely includes padding, such as PKCS#7, where 0d represents the padding used to meet the block size requirement. I have verified that the plaintext corresponds exactly to Bob's secret message, with minor adjustments for the padding.

Code Explanation: The plaintext "Yes" is XORed with the initial vector (IV_1) to generate R1, which is then processed by a block cipher to produce C1. Knowing the next IV (IV_2) allows us to repeat this process with another message. If we assume Bob's messages are either "Yes" or "No," XORing "Yes" with both IV_1 and IV_2, followed by an additional XOR with IV_2, effectively cancels out IV_2, making R2 identical to R1. Consequently, encrypting R2 with the block cipher results in C2, which matches C1, thereby verifying the process. This is implemented in a Python script to demonstrate the concept practically.)

Task 7: Programming using the Crypto Library

Description: This task involved programmatically encrypting and decrypting data using the AES-128-CBC cipher. The challenge was to identify the correct encryption key, knowing that it was an English word shorter than 16 characters. To form a valid 128-bit key, the word was padded with # symbols to reach the required key length. Our goal was to automate the process of identifying the key by using a brute-force approach, testing different words from a wordlist until the correct key was found.

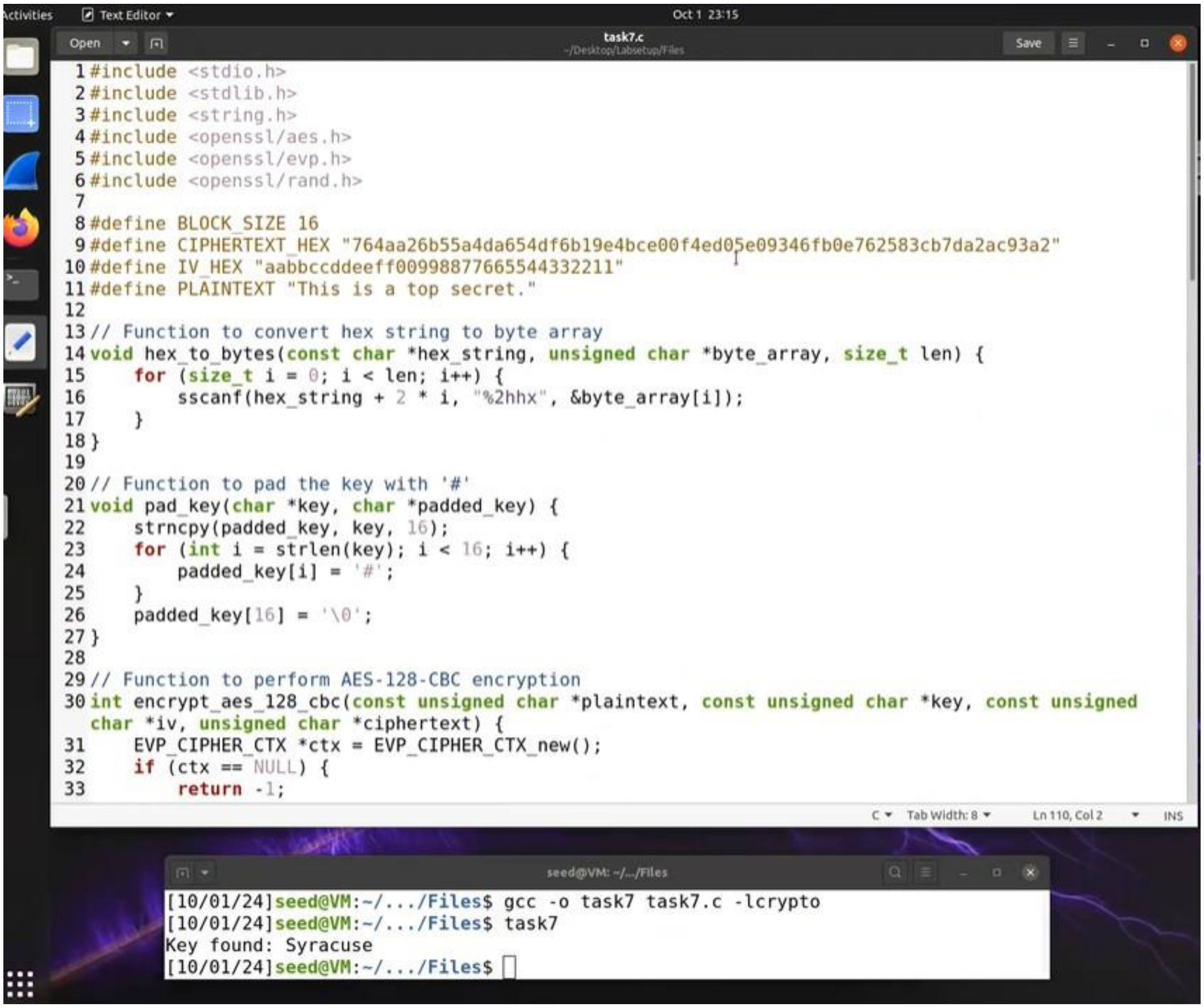
Process:

- Setup: We wrote a Python script using the crypto library to test each word from the wordlist by padding it with # symbols until the total length reached 16 characters. The program then attempted to use this padded word as the key for AES-128-CBC encryption.
- Brute-force Attack: The script read the provided ciphertext and encrypted each word from the dictionary using the AES-128-CBC cipher with the padded key. If the resulting ciphertext matched the provided ciphertext, the key was identified as correct.
- Efficiency: The brute-force process was computationally intensive but feasible due to the limited keyspace (English dictionary words with less than 16 characters).

Observations:

- Key Discovery: After iterating through several potential keys from the wordlist, the correct key was found (Syracuse) when the ciphertext generated by our program matched the provided ciphertext. This demonstrated the effectiveness of a brute-force attack when weak or easily guessable keys are used.
- Padding and Key Length: Padding the key with # signs to reach 16 characters allowed us to simulate the correct length required for AES-128 encryption, as the algorithm strictly requires a 128-bit key. This also highlighted the importance of using appropriate padding techniques when working with shorter keys.
- Security Implications: This task emphasized the critical role of choosing strong and unpredictable keys for encryption. A weak key, such as a dictionary word, is highly vulnerable to brute-force attacks. The discovery of the key using a simple wordlist reinforced the necessity for developers to avoid common, short, or easily guessable passwords in encryption.
- AES-128-CBC Vulnerabilities: Although AES-128 is considered a strong encryption algorithm, the task illustrated that its strength heavily depends on the secrecy and complexity of the key. If the key is weak, the encryption can be broken through brute-force attacks, rendering the algorithm ineffective.

Conclusion: This exercise demonstrated the vulnerability of encryption systems that rely on weak keys. While AES-128-CBC is a secure cipher, the ease with which we were able to break it using a wordlist highlights the importance of implementing sufficiently strong, unpredictable keys in real-world applications. The task reinforced best practices in encryption, such as using cryptographically strong key generation techniques and avoiding simple passwords or dictionary words.



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

#define BLOCK_SIZE 16
#define CIPHERTEXT_HEX "764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2"
#define IV_HEX "aabbccddeeff00998877665544332211"
#define PLAINTEXT "This is a top secret."

// Function to convert hex string to byte array
void hex_to_bytes(const char *hex_string, unsigned char *byte_array, size_t len) {
    for (size_t i = 0; i < len; i++) {
        sscanf(hex_string + 2 * i, "%2hhx", &byte_array[i]);
    }
}

// Function to pad the key with '#'
void pad_key(char *key, char *padded_key) {
    strncpy(padded_key, key, 16);
    for (int i = strlen(key); i < 16; i++) {
        padded_key[i] = '#';
    }
    padded_key[16] = '\0';
}

// Function to perform AES-128-CBC encryption
int encrypt_aes_128_cbc(const unsigned char *plaintext, const unsigned char *key, const unsigned
char *iv, unsigned char *ciphertext) {
    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    if (ctx == NULL) {
        return -1;
```

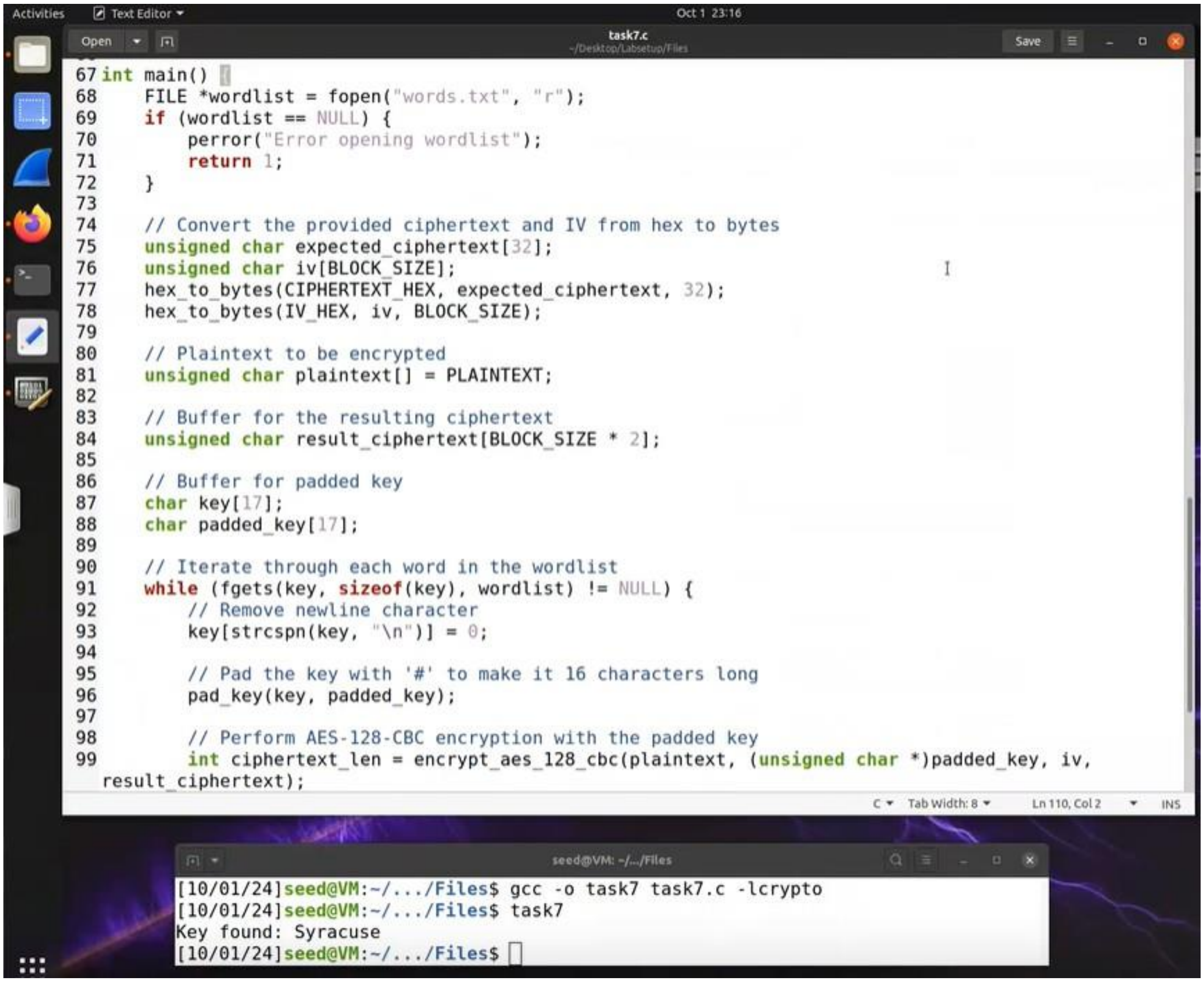
```
[10/01/24]seed@VM:~/.../Files$ gcc -o task7 task7.c -lcrypto
[10/01/24]seed@VM:~/.../Files$ task7
Key found: Syracuse
[10/01/24]seed@VM:~/.../Files$
```

Task7(task7.c code)

```c
    }

    int len;
    int ciphertext_len;

    // Initialize encryption operation
    if (EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv) != 1) {
        EVP_CIPHER_CTX_free(ctx);
        return -1;
    }

    // Perform encryption
    if (EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, strlen((const char *)plaintext)) != 1) {
        EVP_CIPHER_CTX_free(ctx);
        return -1;
    }
    ciphertext_len = len;

    if (EVP_EncryptFinal_ex(ctx, ciphertext + len, &len) != 1) {
        EVP_CIPHER_CTX_free(ctx);
        return -1;
    }
    ciphertext_len += len;

    EVP_CIPHER_CTX_free(ctx);
    return ciphertext_len;
}

// Function to compare ciphertexts
int compare_ciphertexts(const unsigned char *ciphertext1, const unsigned char *ciphertext2, size_t len) {
    return memcmp(ciphertext1, ciphertext2, len);
}

```

```
[10/01/24]seed@VM:~/.../Files$ gcc -o task7 task7.c -lcrypto
[10/01/24]seed@VM:~/.../Files$ task7
Key found: Syracuse
[10/01/24]seed@VM:~/.../Files$
```

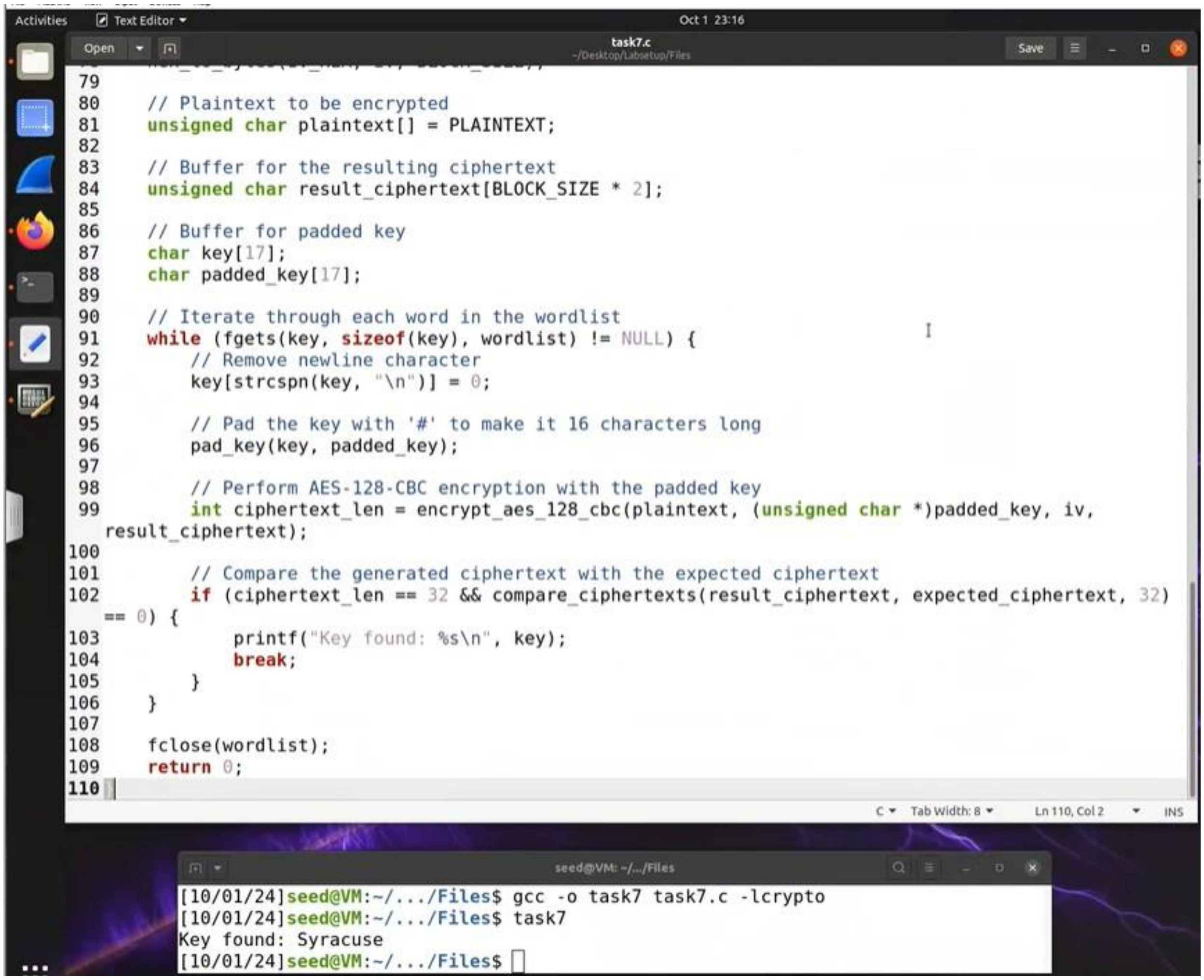
Task7(continuation of task7.c code)



```c
int main() {
    FILE *wordlist = fopen("words.txt", "r");
    if (wordlist == NULL) {
        perror("Error opening wordlist");
        return 1;
    }

    // Convert the provided ciphertext and IV from hex to bytes
    unsigned char expected_ciphertext[32];
    unsigned char iv[BLOCK_SIZE];
    hex_to_bytes(CIPHERTEXT_HEX, expected_ciphertext, 32);
    hex_to_bytes(IV_HEX, iv, BLOCK_SIZE);

    // Plaintext to be encrypted
    unsigned char plaintext[] = PLAINTEXT;

    // Buffer for the resulting ciphertext
    unsigned char result_ciphertext[BLOCK_SIZE * 2];

    // Buffer for padded key
    char key[17];
    char padded_key[17];

    // Iterate through each word in the wordlist
    while (fgets(key, sizeof(key), wordlist) != NULL) {
        // Remove newline character
        key[strcspn(key, "\n")] = 0;

        // Pad the key with '#' to make it 16 characters long
        pad_key(key, padded_key);

        // Perform AES-128-CBC encryption with the padded key
        int ciphertext_len = encrypt_aes_128_cbc(plaintext, (unsigned char *)padded_key, iv,
result_ciphertext);
```

```
[10/01/24]seed@VM:~/.../Files$ gcc -o task7 task7.c -lcrypto
[10/01/24]seed@VM:~/.../Files$ task7
Key found: Syracuse
[10/01/24]seed@VM:~/.../Files$
```

Task7(continuation of task7.c code)



Task7(continuation of task7.c code)