**ACS 545 Cryptography and Network Security**

**Lab 8: Secret-Key Encryption Lab**

**Name:** Vijay Anand Pandian

**Task 1:**

**Step1-3:**

**Code:**

#!/bin/env python3

import random

s = "abcdefghijklmnopqrstuvwxyz"

list = random.sample(s, len(s))

key = ''.join(list)

print(key)

**Implementation and Output:**

**Graphical user interface, text

Description automatically generatedText

Description automatically generatedText

Description automatically generatedText

Description automatically generated**

**Explanation and Observation:** The above code will encrypt the given article.txt file

**Frequency Analysis:**

**Code:**

#!/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

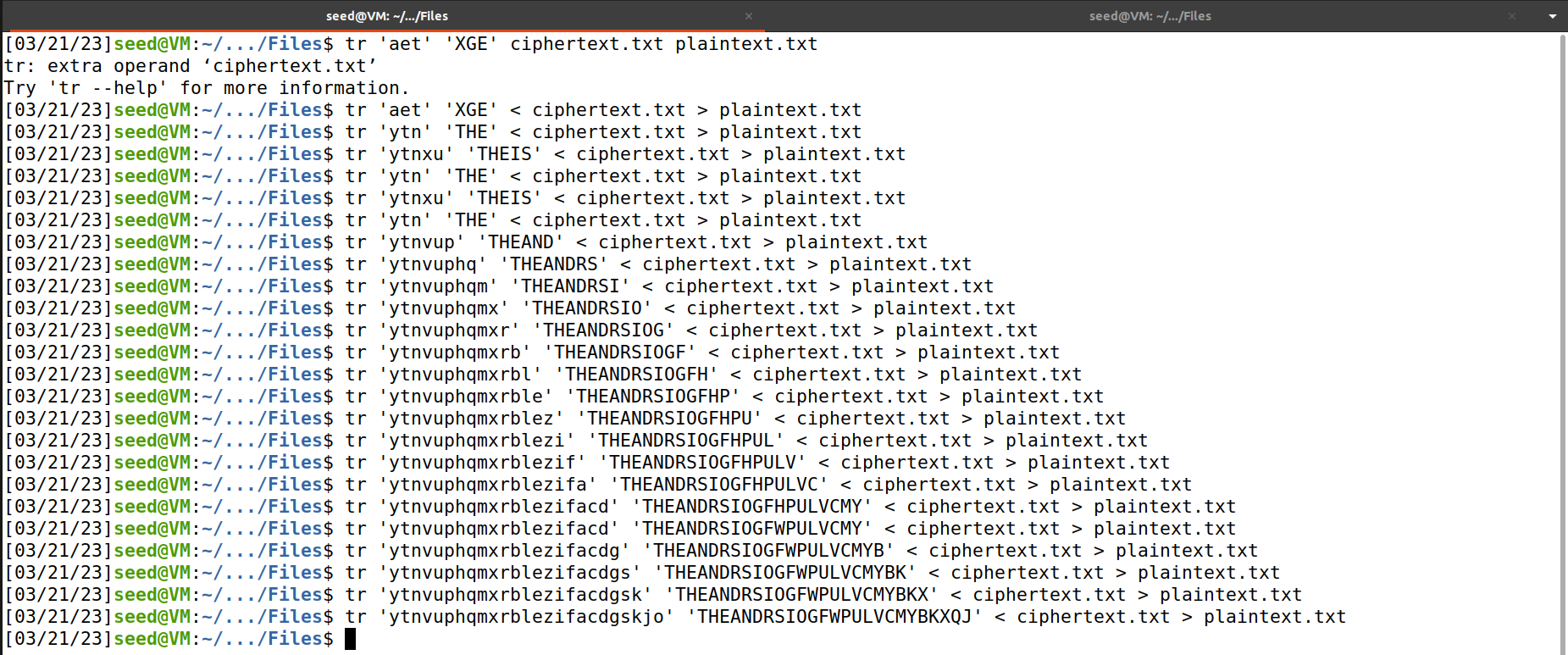
**Implementation and Output:**

**Graphical user interface, text, application

Description automatically generated**

**Text

Description automatically generated with low confidenceGraphical user interface, text

Description automatically generatedText

Description automatically generated  
  
Explanation and Observation:**

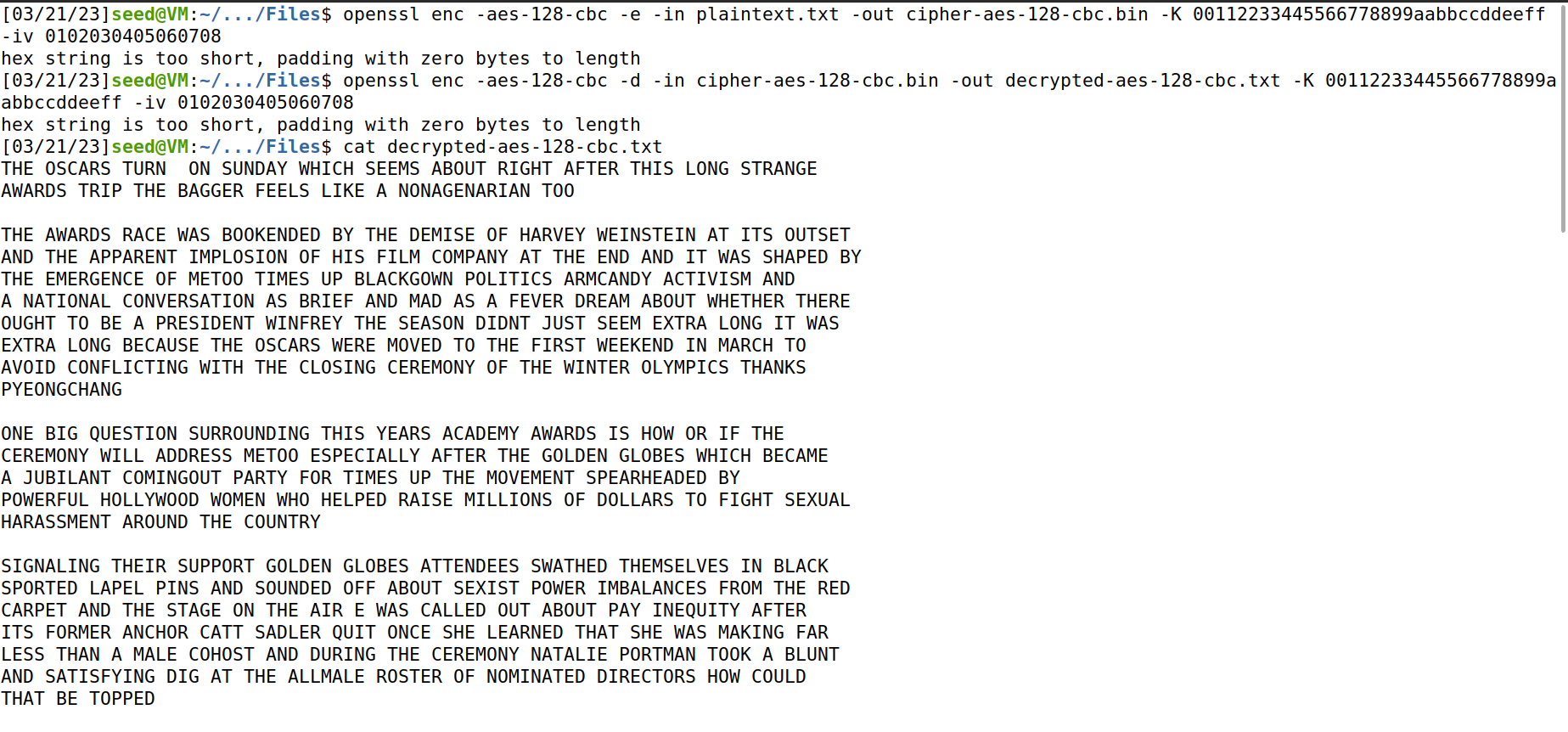
The above code will list the 1-gram, 2-gram and 3-gram frequencies in the cipher.txt file.

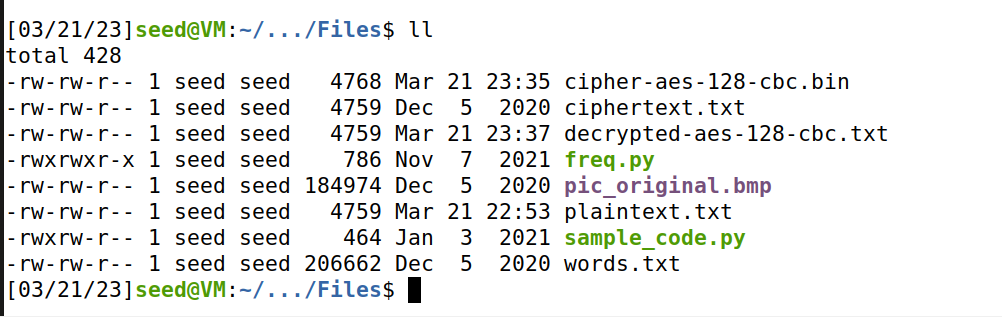
With the help of the deciphered content in the plaintext.txt file. You can see the deciphered final content of that file in the above screenshot.

**Task2 - Encryption using Different Ciphers and Modes**

**Implementation and Output:**

**AES-128-CBC Mode:**

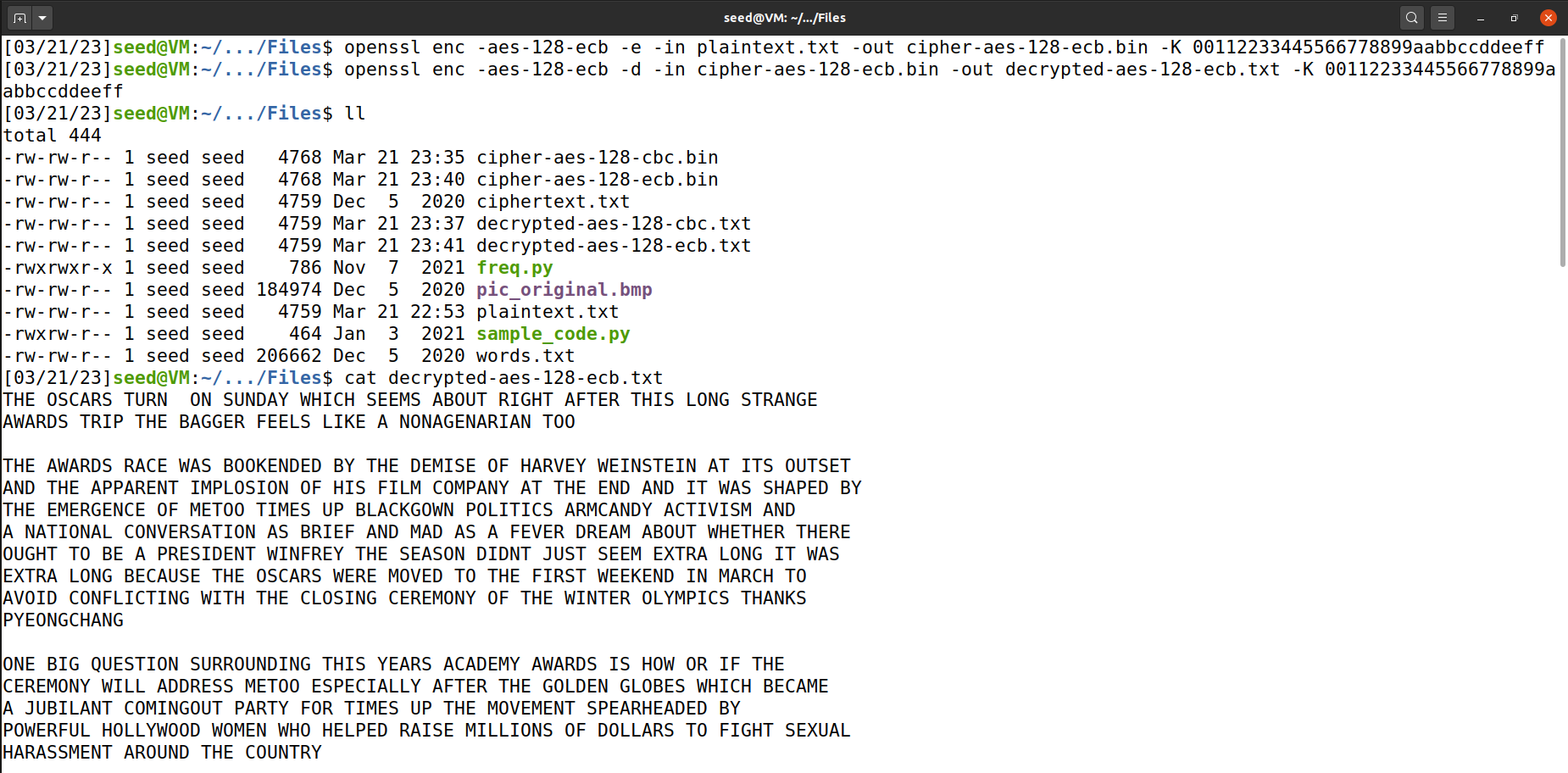
****

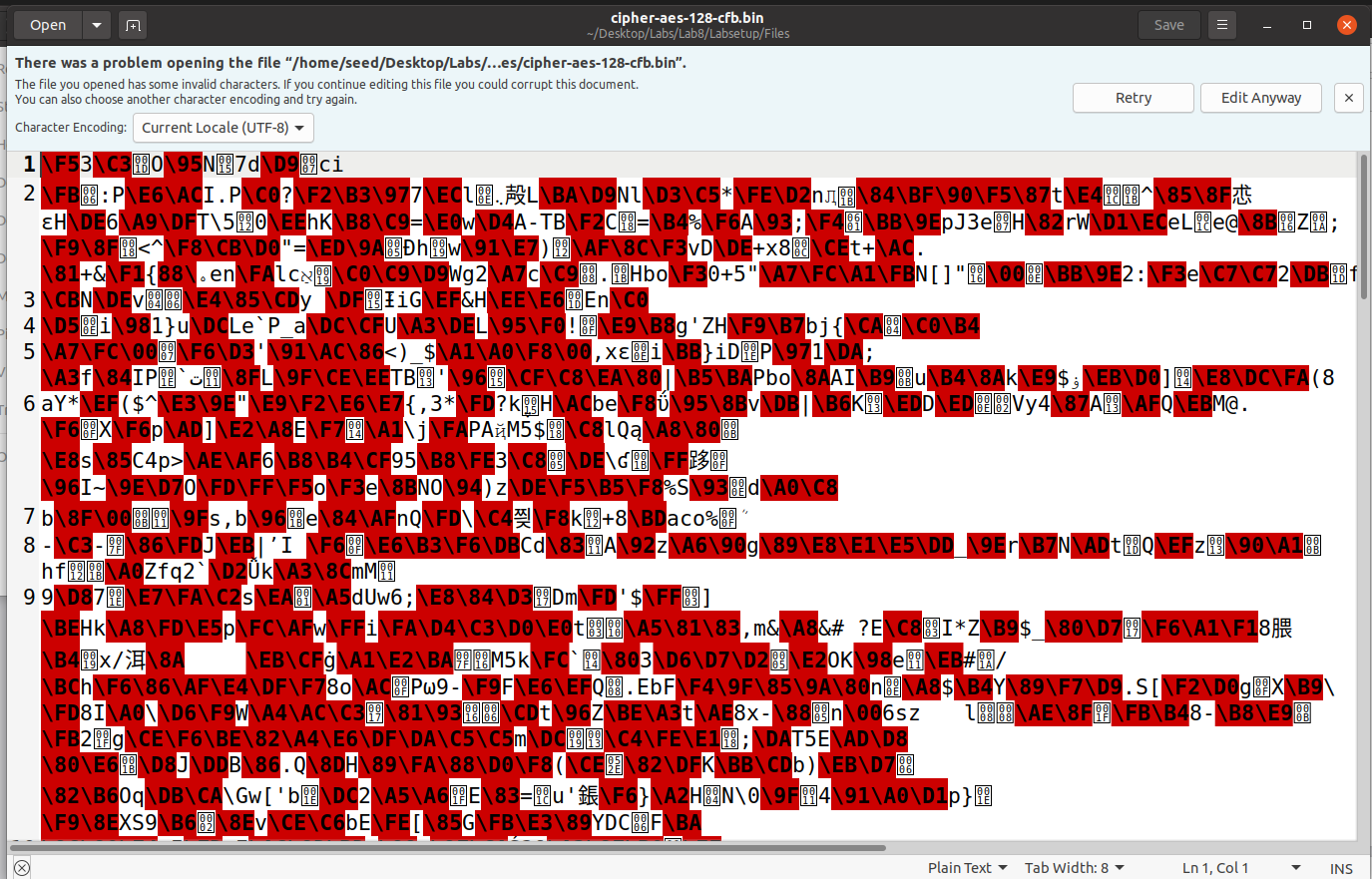
****

**A picture containing qr code

Description automatically generated**

**AES-128-ECB Mode:**

****

****

**AES-128-CFB Mode:**

**Text

Description automatically generated**

**  
  
Explanation and Observation:**

I have tried encrypting the plaintext.txt in three different encryption algorithms and modes. Above are the screenshots of the implementation.

**Task 6 – Initial Vector (IV) and Common Mistakes**

**Task6.1 – IV Experiment**

**Implementation and Output:**

**Using the same IV:**

**Text

Description automatically generated**

Encrypted files:**Qr code

Description automatically generated**

**Using the different IVs:**

**Text

Description automatically generated**

Encrypted Files:

**Qr code

Description automatically generated**

**Explanation and Output:**When encrypting with the same IV we can see that the resulting cipher text is same.

When encrypting with the different IVs we can see that the resulting cipher text is different.

We observe that IV is used to add randomness to the encryption process which will reduce the vulnerability to attacks.

**Task6.2 - Common Mistake: Use the Same IV**

**Code:**

#!/usr/bin/python3

from sys import argv

script, first, second = argv

aa = bytearray.fromhex(first)

bb = bytearray.fromhex(second)

xord = bytearray(x^y for x,y in zip(aa,bb))

print(xord.hex())

**Implementation and Output:**

**Graphical user interface, text, application

Description automatically generated**

**Explanation and Observation:**

The above code will perform the XOR operation.When reusing OFB and the same IV for encryption, even if our two plaintext messages to be encrypted are not identical there patterns will start to emerge and the algorithm will become deterministic. IN the case of P1> C1, P2>C2 there are some repeats in the ciphertext between C1 and C2, this can be used to crack the encryption.

When using CBC mode the attacker can only know the first block of the message.

**Task6.3 – Common Mistake: Use a Predictable IV**

**Code:**

#!/usr/bin/python3

from sys import argv

script, first, second = argv

aa = bytearray.fromhex(first)

bb = bytearray.fromhex(second)

xord = bytearray(x^y for x,y in zip(aa,bb))

print(xord.hex())

**Implementation and Output:**

**Graphical user interface, text, application

Description automatically generatedText

Description automatically generated**

**Explanation and Observation:**

We can see that the attack works because the CBC mode of AES encryption uses the previous block's ciphertext as the IV for the next block. If the IV is predictable, then an attacker can construct messages that result in the same ciphertext as other messages. Therefore, predictable IVs can compromise the security of AES encryption, and it is essential to use unpredictable IVs to ensure security.

**Task7 – Programming using the Crypto Library**

**Code:**

#!/usr/bin/python3

from Crypto.Cipher import AES

from Crypto.Util import Padding

data = b'This is a top secret.'

expected\_ciphertext = '3879c71b232cd0d2fc6f5ffcc1d76f074c0fcbe007d9cc53939fdeebf1d6ffd2'

iv\_hex\_string = 'aabbccddeeff00998877665544332211'

iv = bytes.fromhex(iv\_hex\_string)

f = open('words.txt','r')

for word in f:

if(len(word) <= 16):

k = word.strip()

key\_string = k + ('#' \*(16 - len(k)))

key = bytes.fromhex(key\_string.encode('utf-8').hex())

cipher = AES.new(key, AES.MODE\_CBC, iv)

ciphertext = cipher.encrypt(Padding.pad(data, 16))

if(expected\_ciphertext == ciphertext.hex()):

print ("Key is:",key\_string);

**Implementation and Output:**

**Graphical user interface, text

Description automatically generated**

**Explanation and Observation:**

The above code will decrypt the given cipher text using the key obtained by brute force search from words.txt file which is like an English dictionary. It will also append ‘#’ to the end of the key to ensure the key is 128-bit.