

# SEED Labs - Secret-Key Encryption Lab

**Jatin Kesnani**  
21K-3204 Student  
FAST University

## Abstract

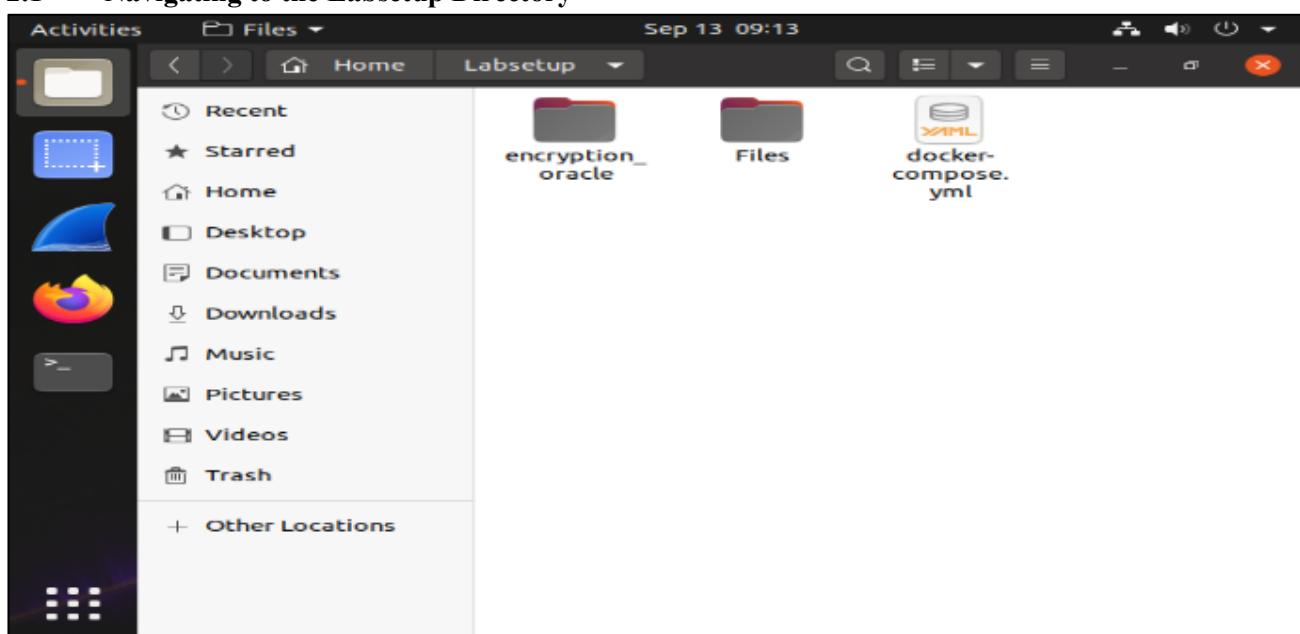
*In today's digital world, ensuring the security and privacy of information is paramount. Cryptography, through its various encryption algorithms, provides a robust means of protecting sensitive data from unauthorized access. This report explores the fundamental principles of cryptography, focusing on 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

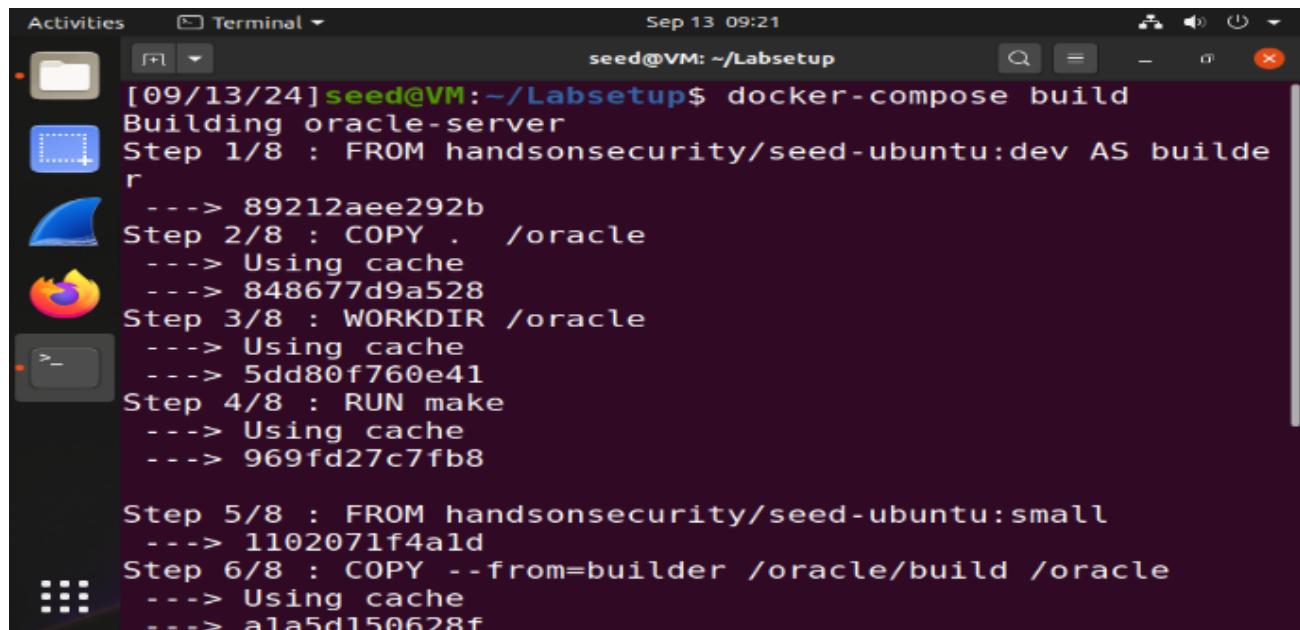
Cryptography is essential for securing digital communication and protecting sensitive data from unauthorized access. With the rise of online data exchange, encryption techniques play a key role in ensuring confidentiality and integrity. This section provides a brief look at symmetric and asymmetric encryption 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



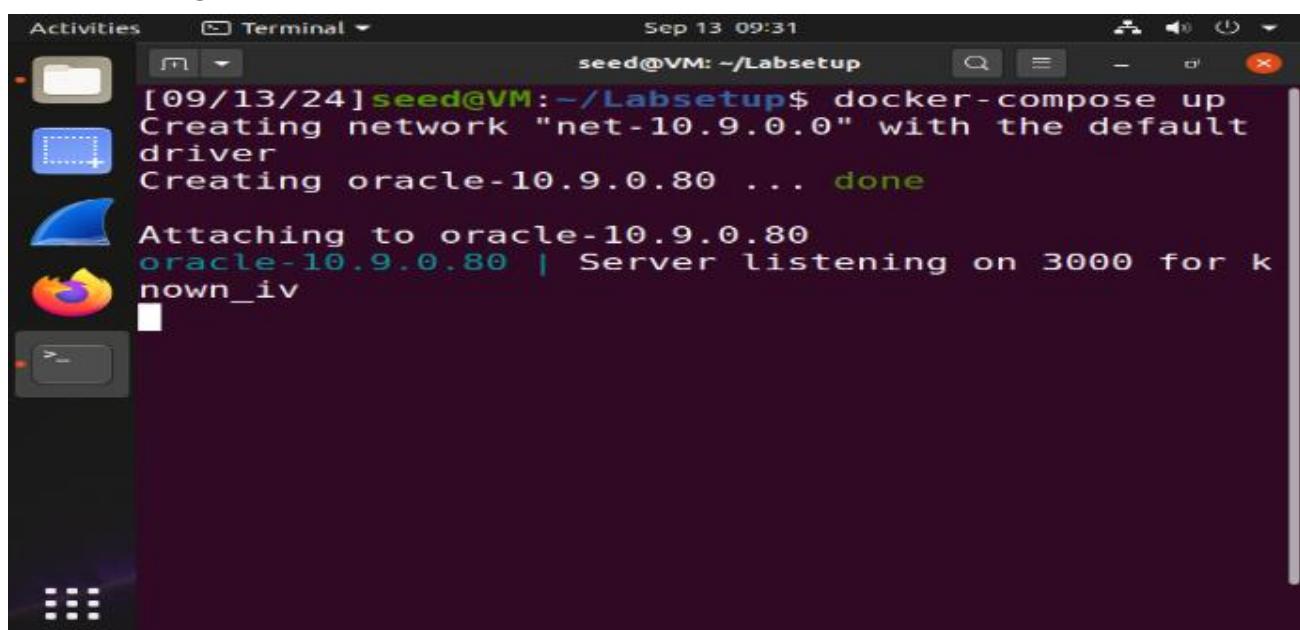
## 2.2 Building the Docker Container



```
[09/13/24] seed@VM:~/Labsetup$ docker-compose build
Building oracle-server
Step 1/8 : FROM handsonsecurity/seed-ubuntu:dev AS builder
--> 89212aee292b
Step 2/8 : COPY . /oracle
--> Using cache
--> 848677d9a528
Step 3/8 : WORKDIR /oracle
--> Using cache
--> 5dd80f760e41
Step 4/8 : RUN make
--> Using cache
--> 969fd27c7fb8

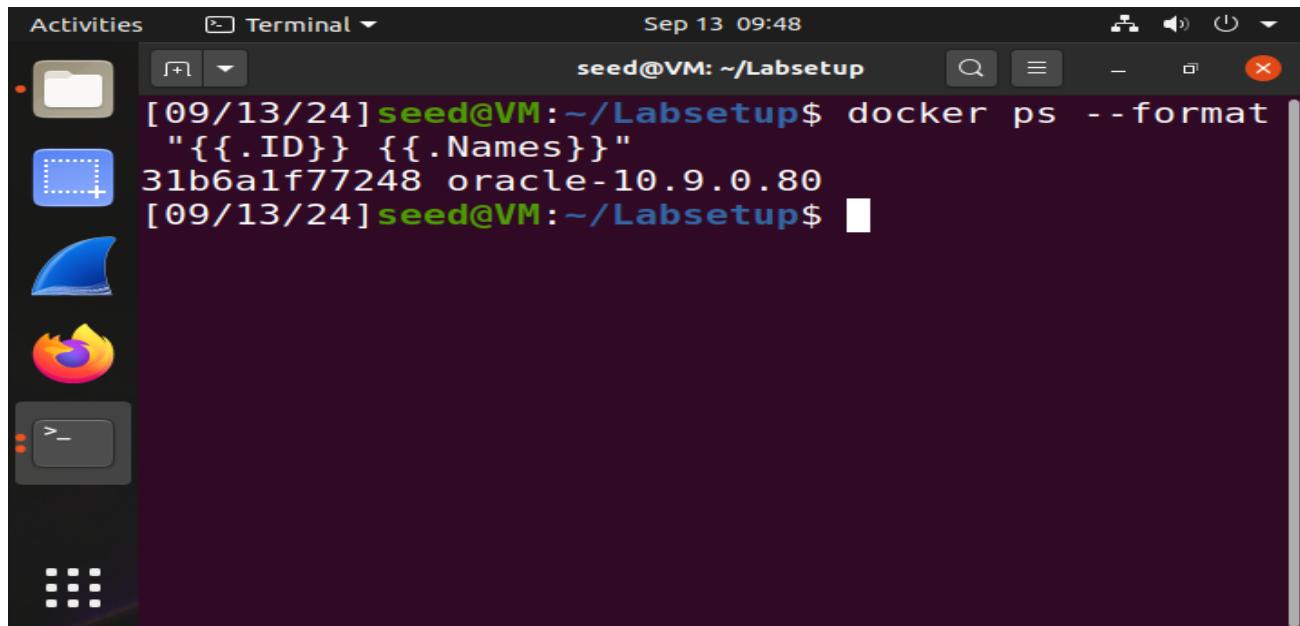
Step 5/8 : FROM handsonsecurity/seed-ubuntu:small
--> 1102071f4a1d
Step 6/8 : COPY --from=builder /oracle/build /oracle
--> Using cache
--> a1a5d150628f
```

## 2.3 Starting the Docker Container



```
[09/13/24] seed@VM:~/Labsetup$ docker-compose up
Creating network "net-10.9.0.0" with the default
driver
Creating oracle-10.9.0.80 ... done
Attaching to oracle-10.9.0.80
oracle-10.9.0.80 | Server listening on 3000 for known_iv
```

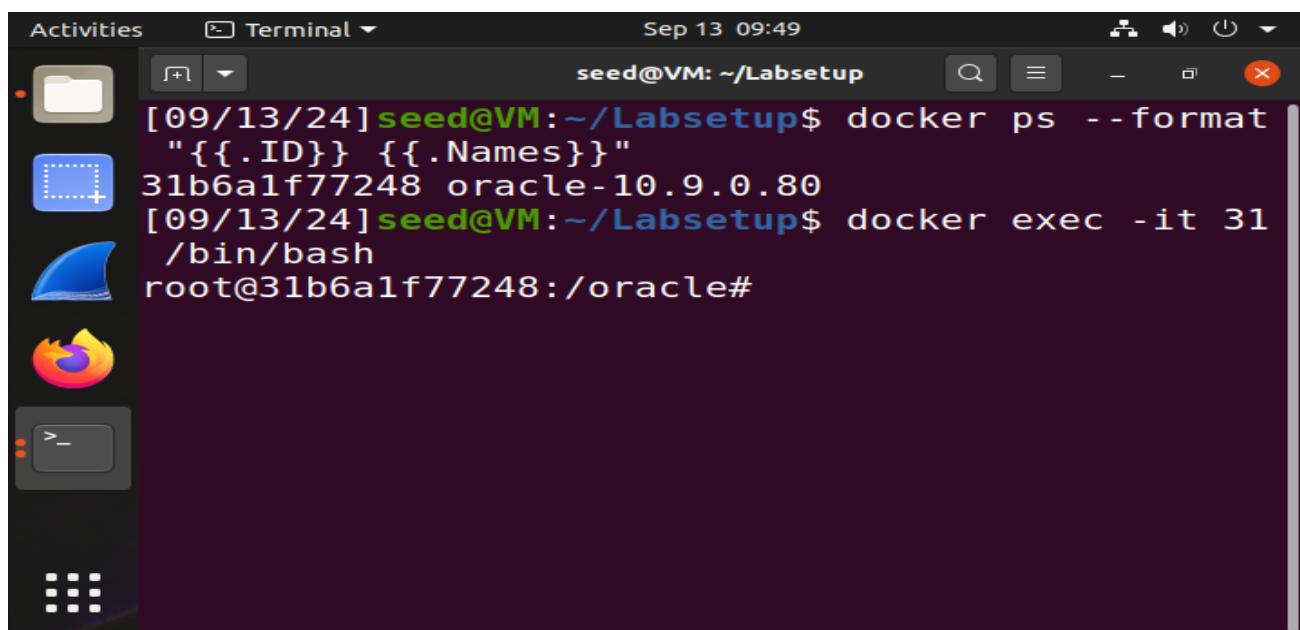
## 2.4 Verifying the Container is Running



A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window. The terminal window title is "Terminal" and the command prompt is "seed@VM: ~/Labsetup". The date and time at the top of the terminal window are "Sep 13 09:48". The terminal window displays the following command and its output:

```
[09/13/24] seed@VM:~/Labsetup$ docker ps --format
  "{{.ID}} {{.Names}}"
31b6a1f77248 oracle-10.9.0.80
[09/13/24] seed@VM:~/Labsetup$
```

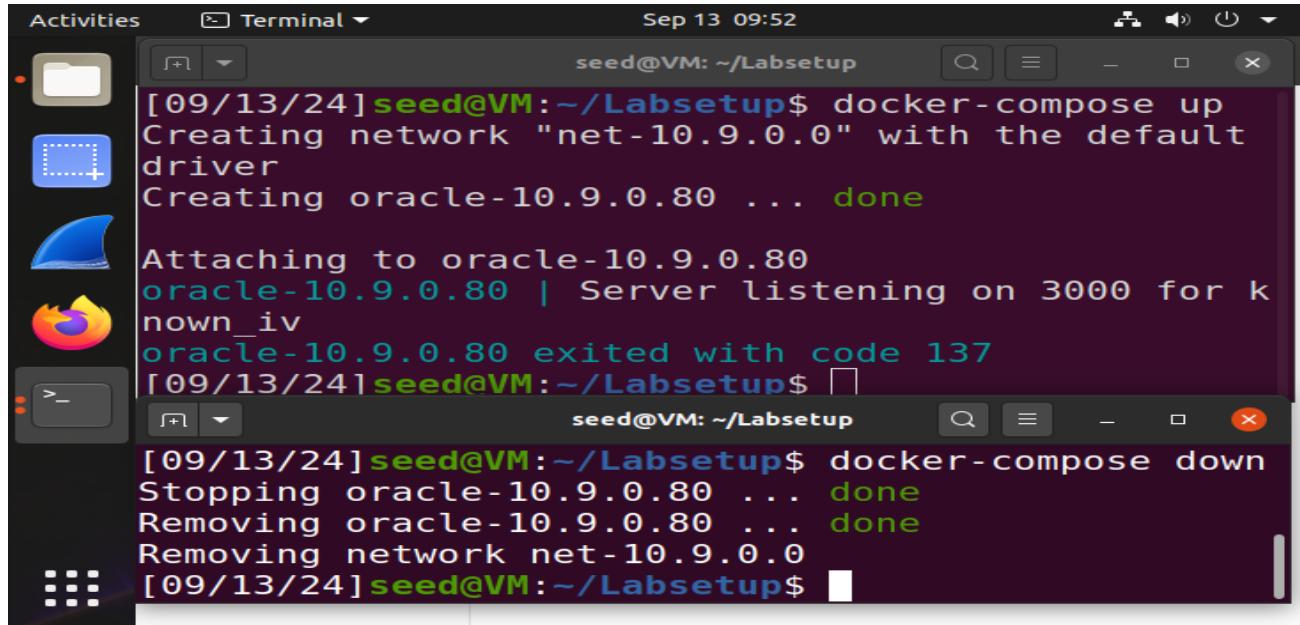
## 2.5 Accessing the Container Shell



A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window. The terminal window title is "Terminal" and the command prompt is "seed@VM: ~/Labsetup". The date and time at the top of the terminal window are "Sep 13 09:49". The terminal window displays the following commands and their outputs:

```
[09/13/24] seed@VM:~/Labsetup$ docker ps --format
  "{{.ID}} {{.Names}}"
31b6a1f77248 oracle-10.9.0.80
[09/13/24] seed@VM:~/Labsetup$ docker exec -it 31
/bin/bash
root@31b6a1f77248:/oracle#
```

## 2.6 Shutting Down the Container



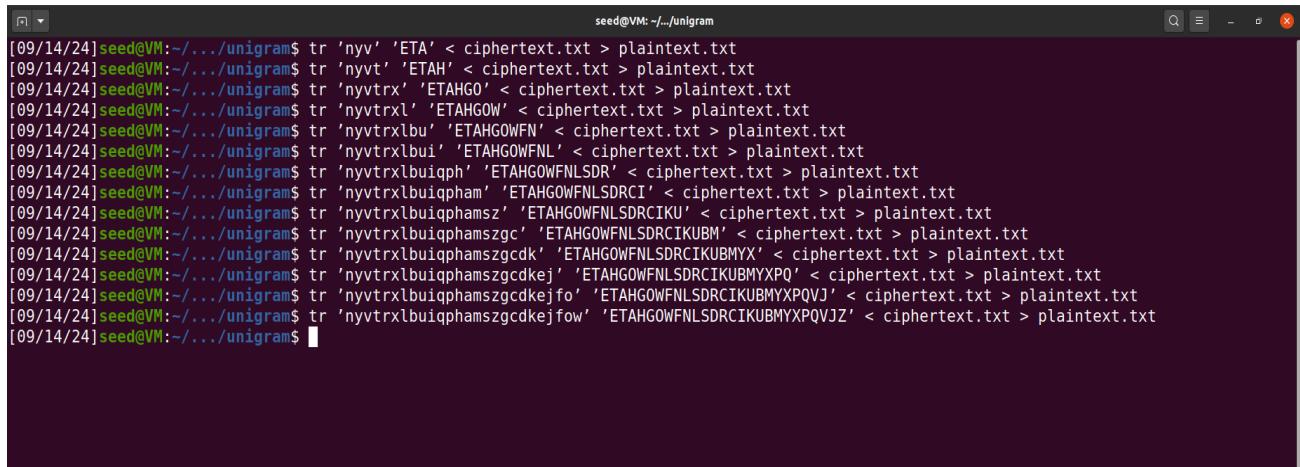
```
[09/13/24] seed@VM:~/Labsetup$ docker-compose up
Creating network "net-10.9.0.0" with the default
driver
Creating oracle-10.9.0.80 ... done

Attaching to oracle-10.9.0.80
oracle-10.9.0.80 | Server listening on 3000 for k
nown_iv
oracle-10.9.0.80 exited with code 137
[09/13/24] seed@VM:~/Labsetup$ 

[09/13/24] seed@VM:~/Labsetup$ docker-compose down
Stopping oracle-10.9.0.80 ... done
Removing oracle-10.9.0.80 ... done
Removing network net-10.9.0.0
[09/13/24] seed@VM:~/Labsetup$ 
```

## 3. Task 1: Frequency Analysis

### 3.1 Monoalphabetic Cipher Using Unigram Frequencies



```
[09/14/24] seed@VM:~/.../unigram$ tr 'nyv' 'ETA' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyt' 'ETAH' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrx' 'ETAHGO' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxl' 'ETAHGOW' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbu' 'ETAHGOWFN' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbu' 'ETAHGOWFNL' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbuqph' 'ETAHGOWFNLSDR' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbuqpham' 'ETAHGOWFNLSDRCI' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbuqphamsz' 'ETAHGOWFNLSDRCIKU' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbuqphamszgc' 'ETAHGOWFNLSDRCIKUBM' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbuqphamszgdk' 'ETAHGOWFNLSDRCIKUBMYX' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbuqphamszgdkej' 'ETAHGOWFNLSDRCIKUBMYXPQ' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbuqphamszgdkejfo' 'ETAHGOWFNLSDRCIKUBMYXPQVJ' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ tr 'nyvtrxlbuqphamszgdkejfow' 'ETAHGOWFNLSDRCIKUBMYXPQVJZ' < ciphertext.txt > plaintext.txt
[09/14/24] seed@VM:~/.../unigram$ 
```

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

```
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' | tr 'ng' 'ES' | tr 'xu' 'ON' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' | tr 'ng' 'ES' | tr 'xu' 'ON' | tr 'up' 'ND'
| tr 'vi' 'AL' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' | tr 'ng' 'ES' | tr 'xu' 'ON' | tr 'up' 'ND'
| tr 'vi' 'AL' | tr 'gn' 'BE' | tr 'av' 'CA' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' | tr 'ng' 'ES' | tr 'xu' 'ON' | tr 'up' 'ND'
| tr 'vi' 'AL' | tr 'gn' 'BE' | tr 'av' 'CA' | tr 'fn' 'VE' | tr 'cn' 'ME' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' | tr 'ng' 'ES' | tr 'xu' 'ON' | tr 'up' 'ND'
| tr 'vi' 'AL' | tr 'gn' 'BE' | tr 'av' 'CA' | tr 'fn' 'VE' | tr 'cn' 'ME' | tr 'id' 'LY' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' | tr 'ng' 'ES' | tr 'xu' 'ON' | tr 'up' 'ND'
| tr 'vi' 'AL' | tr 'gn' 'BE' | tr 'av' 'CA' | tr 'fn' 'VE' | tr 'cn' 'ME' | tr 'id' 'LY' | tr 'en' 'PE' | tr 'ur' 'NG' | tr 'lv' 'WA' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' | tr 'ng' 'ES' | tr 'xu' 'ON' | tr 'up' 'ND'
| tr 'vi' 'AL' | tr 'gn' 'BE' | tr 'av' 'CA' | tr 'fn' 'VE' | tr 'cn' 'ME' | tr 'id' 'LY' | tr 'en' 'PE' | tr 'ur' 'NG' | tr 'lv' 'WA' | tr 'xb' 'OF' | tr 'sn' 'KE' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$ tr 'yt' 'TH' < ciphertext.txt | tr 'tn' 'HE' | tr 'mu' 'IN' | tr 'nh' 'ER' | tr 'vh' 'AR' | tr 'ng' 'ES' | tr 'xu' 'ON' | tr 'up' 'ND'
| tr 'vi' 'AL' | tr 'gn' 'BE' | tr 'av' 'CA' | tr 'fn' 'VE' | tr 'cn' 'ME' | tr 'id' 'LY' | tr 'en' 'PE' | tr 'ur' 'NG' | tr 'lv' 'WA' | tr 'xb' 'OF' | tr 'sn' 'KE' | tr 'jz' 'OU' | tr 'nk' 'EX' | tr 'oz' 'JU' > plaintext.txt
[09/14/24]seed@VM:~/.../bigram$
```

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.

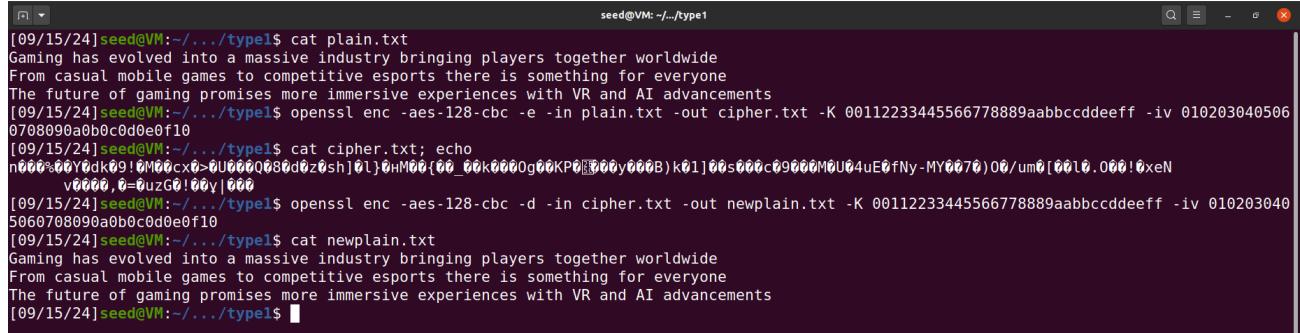
## 3.3 Monoalphabetic Cipher Using Trigram Frequencies

```
[09/15/24]seed@VM:~/.../trigram$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' > plaintext.txt
[09/15/24]seed@VM:~/.../trigram$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' > plaintext.txt
[09/15/24]seed@VM:~/.../trigram$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' | tr 'mxu' 'ION' | tr 'gnq' 'BES' > plaintext.txt
[09/15/24]seed@VM:~/.../trigrams$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' | tr 'mxu' 'ION' | tr 'gnq' 'BES' | tr 'vii' 'ALL' | tr 'bxh' 'FOR' > plaintext.txt
[09/15/24]seed@VM:~/.../trigrams$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' | tr 'mxu' 'ION' | tr 'gnq' 'BES' | tr 'vii' 'ALL' | tr 'lvq' 'WAS' | tr 'cmu' 'MIN' > plaintext.txt
[09/15/24]seed@VM:~/.../trigrams$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' | tr 'mxu' 'ION' | tr 'gnq' 'BES' | tr 'vii' 'ALL' | tr 'bxh' 'FOR' | tr 'lvq' 'WAS' | tr 'cmu' 'MIN' | tr 'avh' 'CAR' | tr 'cxf' 'MOV' > plaintext.txt
[09/15/24]seed@VM:~/.../trigrams$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' | tr 'mxu' 'ION' | tr 'gnq' 'BES' | tr 'vii' 'ALL' | tr 'bxh' 'FOR' | tr 'lvq' 'WAS' | tr 'cmu' 'MIN' | tr 'avh' 'CAR' | tr 'cxf' 'MOV' | tr 'ehn' 'PRE' | tr 'iid' 'LLY' > plaintext.txt
[09/15/24]seed@VM:~/.../trigrams$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' | tr 'mxu' 'ION' | tr 'gnq' 'BES' | tr 'vii' 'ALL' | tr 'bxh' 'FOR' | tr 'lvq' 'WAS' | tr 'cmu' 'MIN' | tr 'avh' 'CAR' | tr 'cxf' 'MOV' | tr 'ehn' 'PRE' | tr 'iid' 'LLY' | tr 'cvs' 'MAK' | tr 'njz' 'EQU' > plaintext.txt
[09/15/24]seed@VM:~/.../trigrams$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' | tr 'mxu' 'ION' | tr 'gnq' 'BES' | tr 'vii' 'ALL' | tr 'bxh' 'FOR' | tr 'lvq' 'WAS' | tr 'cmu' 'MIN' | tr 'avh' 'CAR' | tr 'cxf' 'MOV' | tr 'ehn' 'PRE' | tr 'iid' 'LLY' | tr 'cvs' 'MAK' | tr 'njz' 'EQU' | tr 'kyh' 'XTR' | tr 'ozq' 'JUS' > plaintext.txt
[09/15/24]seed@VM:~/.../trigrams$ tr 'ytn' 'THE' < ciphertext.txt | tr 'vup' 'AND' | tr 'mur' 'ING' | tr 'ynh' 'TER' | tr 'xzy' 'OUT' | tr 'mxu' 'ION' | tr 'gnq' 'BES' | tr 'vii' 'ALL' | tr 'bxh' 'FOR' | tr 'lvq' 'WAS' | tr 'cmu' 'MIN' | tr 'avh' 'CAR' | tr 'cxf' 'MOV' | tr 'ehn' 'PRE' | tr 'iid' 'LLY' | tr 'cvs' 'MAK' | tr 'njz' 'EQU' | tr 'kyh' 'XTR' | tr 'ozq' 'JUS' | tr 'mwn' 'IZE' > plaintext.txt
[09/15/24]seed@VM:~/.../trigrams$
```

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

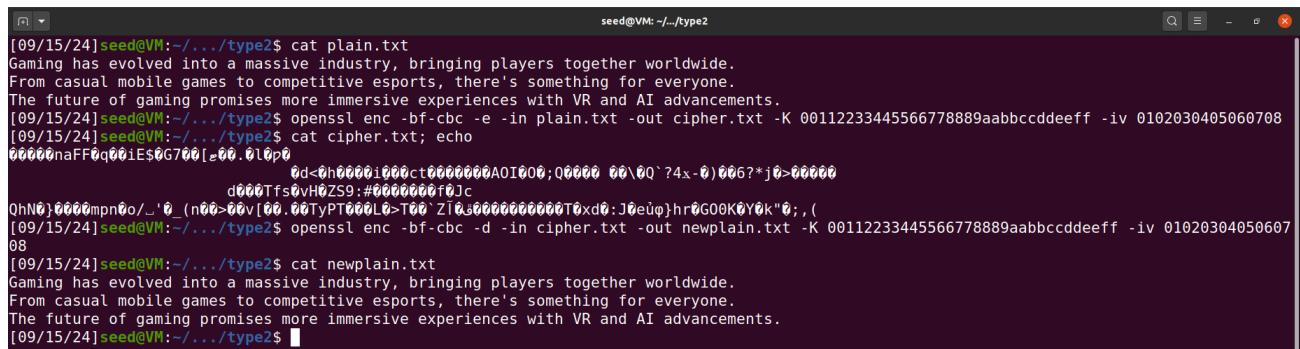
### 4.1 Cipher Type: AES-128-CBC



```
[09/15/24] seed@VM:~/.../type1$ cat plain.txt
Gaming has evolved into a massive industry bringing players together worldwide
From casual mobile games to competitive esports there is something for everyone
The future of gaming promises more immersive experiences with VR and AI advancements
[09/15/24] seed@VM:~/.../type1$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher.txt -K 00112233445566778889aabbccddeff -iv 0102030405060708
[09/15/24] seed@VM:~/.../type1$ cat cipher.txt; echo
n6N0%0y7dk9!0m0cxd0-0l000000d0zsh]0l)0H00{00_00k000g00K0[00y00B)k01]00s00c09000M0U04uE0fNy-MY0070)00/um0[00L0.000!0xeN
v0000,0=0uzG!00y|00
[09/15/24] seed@VM:~/.../type1$ openssl enc -aes-128-cbc -d -in cipher.txt -out newplain.txt -K 00112233445566778889aabbccddeff -iv 0102030405060708
[09/15/24] seed@VM:~/.../type1$ cat newplain.txt
Gaming has evolved into a massive industry bringing players together worldwide
From casual mobile games to competitive esports there is something for everyone
The future of gaming promises more immersive experiences with VR and AI advancements
[09/15/24] seed@VM:~/.../type1$
```

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



```
[09/15/24] seed@VM:~/.../type2$ cat plain.txt
Gaming has evolved into a massive industry, bringing players together worldwide.
From casual mobile games to competitive esports, there's something for everyone.
The future of gaming promises more immersive experiences with VR and AI advancements.
[09/15/24] seed@VM:~/.../type2$ openssl enc -bf-cbc -e -in plain.txt -out cipher.txt -K 00112233445566778889aabbccddeff -iv 0102030405060708
[09/15/24] seed@VM:~/.../type2$ cat cipher.txt; echo
0000naFF0q00iE$0G700[e00.0l0p0
    0d<0h00001000ct000000AOI000;00000 00\0Q`?4x-0)006?*j0>00000
    d000Tfs0vH0ZS0:#000000f0jc
0hN0)0000mpn0o/_'0_(n00>00[00.00TypT0000.0>T00_Z[&0000000000T0xd0:J0eÜ0}hr0G00K0Y0k"0;,,
[09/15/24] seed@VM:~/.../type2$ openssl enc -bf-cbc -d -in cipher.txt -out newplain.txt -K 00112233445566778889aabbccddeff -iv 0102030405060708
[09/15/24] seed@VM:~/.../type2$ cat newplain.txt
Gaming has evolved into a massive industry, bringing players together worldwide.
From casual mobile games to competitive esports, there's something for everyone.
The future of gaming promises more immersive experiences with VR and AI advancements.
[09/15/24] seed@VM:~/.../type2$
```

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 00112233445566778889aabbccddeff -iv 0102030405060708`. I then decrypted cipher.txt back to newplain.txt with `openssl enc -bf-cbc -d -in cipher.txt -out newplain.txt -K 00112233445566778889aabbccddeff -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

```
seed@VM:~/.../type3$ cat plain.txt
Gaming has evolved into a massive industry, bringing players together worldwide.
From casual mobile games to competitive esports, there's something for everyone.
The future of gaming promises more immersive experiences with VR and AI advancements.
[09/15/24]seed@VM:~/.../type3$ openssl enc -aes-128-cfb -e -in plain.txt -out cipher.txt -K 00112233445566778889aabcccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/15/24]seed@VM:~/.../type3$ cat cipher.txt; echo

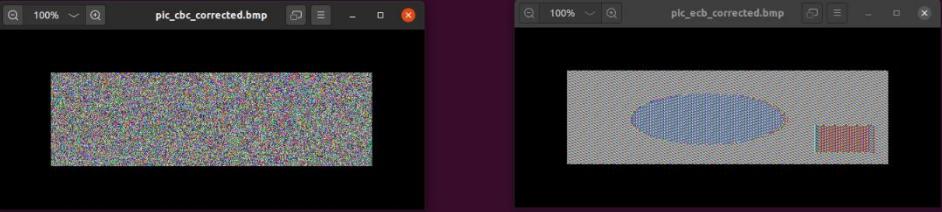
[09/15/24]seed@VM:~/.../type3$ openssl enc -aes-128-cfb -d -in cipher.txt -out newplain.txt -K 00112233445566778889aabcccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/15/24]seed@VM:~/.../type3$ cat newplain.txt
Gaming has evolved into a massive industry, bringing players together worldwide.
From casual mobile games to competitive esports, there's something for everyone.
The future of gaming promises more immersive experiences with VR and AI advancements.
[09/15/24]seed@VM:~/.../type3$
```

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 00112233445566778889aabcccddeff -iv 0102030405060708090a0b0c0d0e0f10`, 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 00112233445566778889aabcccddeff -iv 0102030405060708090a0b0c0d0e0f10`, restoring the original text to `newplain.txt`.

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

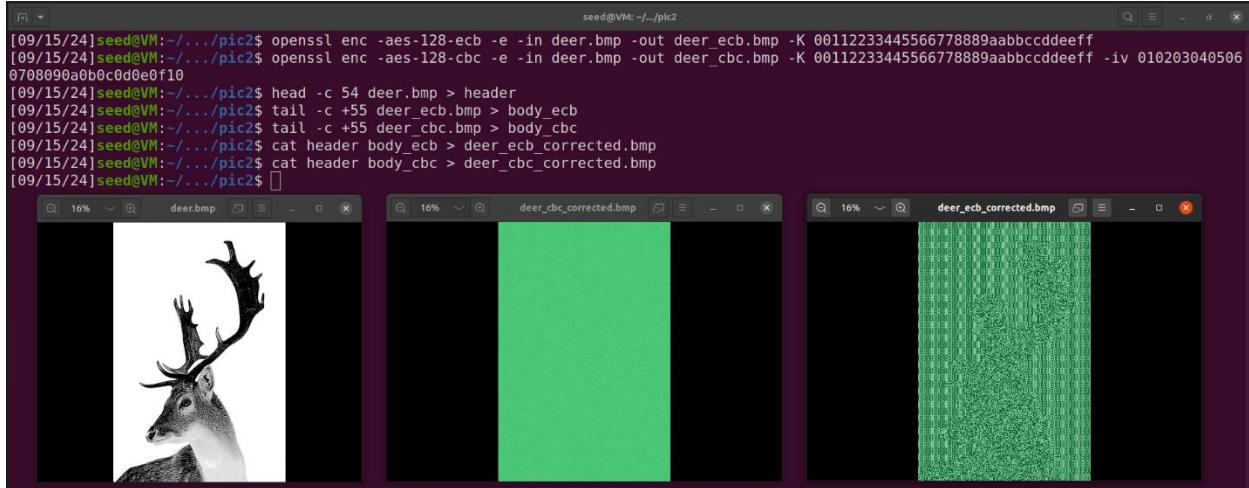
### 5.1 Encryption (pic\_original.bmp) – ECB vs. CBC

```
seed@VM:~/.../pic1$ openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.bmp -K 00112233445566778889aabcccddeff
[09/15/24]seed@VM:~/.../pic1$ openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.bmp -K 00112233445566778889aabcccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/15/24]seed@VM:~/.../pic1$ head -c 54 pic_original.bmp > header
[09/15/24]seed@VM:~/.../pic1$ tail -c +55 pic_ecb.bmp > body_ecb
[09/15/24]seed@VM:~/.../pic1$ tail -c +55 pic_cbc.bmp > body_cbc
[09/15/24]seed@VM:~/.../pic1$ cat header body_ecb > pic_ecb_corrected.bmp
[09/15/24]seed@VM:~/.../pic1$ cat header body_cbc > pic_cbc_corrected.bmp
[09/15/24]seed@VM:~/.../pic1$
```



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:

```
seed@VM:~/.../Task4$ echo -n "12345" > f1.txt
[09/15/24] seed@VM:~/.../Task4$ echo -n "1234567890" > f2.txt
[09/15/24] seed@VM:~/.../Task4$ echo -n "1234567812345678" > f3.txt
[09/15/24] seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_enc.txt -K 00112233445566778889aabccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/15/24] seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -e -in f2.txt -out f2_enc.txt -K 00112233445566778889aabccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/15/24] seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -e -in f3.txt -out f3_enc.txt -K 00112233445566778889aabccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/15/24] seed@VM:~/.../Task4$ stat --format=%s f1_enc.txt
16
[09/15/24] seed@VM:~/.../Task4$ stat --format=%s f2_enc.txt
16
[09/15/24] seed@VM:~/.../Task4$ stat --format=%s f3_enc.txt
32
[09/15/24] seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -d -in f1_enc.txt -out f1_dec_nopad.txt -K 00112233445566778889aabccddeff -iv 0102030405060708090a0b0c0d0e0f10 -nopad
[09/15/24] seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -d -in f2_enc.txt -out f2_dec_nopad.txt -K 00112233445566778889aabccddeff -iv 0102030405060708090a0b0c0d0e0f10 -nopad
[09/15/24] seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -d -in f3_enc.txt -out f3_dec_nopad.txt -K 00112233445566778889aabccddeff -iv 0102030405060708090a0b0c0d0e0f10 -nopad
[09/15/24] seed@VM:~/.../Task4$ hexdump -C f1_dec_nopad.txt
00000000 31 32 33 34 35 0b 0b 0b 0b 0b 0b 0b 0b 0b |12345.....|
00000010
[09/15/24] seed@VM:~/.../Task4$ hexdump -C f2_dec_nopad.txt
00000000 31 32 33 34 35 36 37 38 39 30 06 06 06 06 06 |1234567890.....|
00000010
[09/15/24] seed@VM:~/.../Task4$ hexdump -C f3_dec_nopad.txt
00000000 31 32 33 34 35 36 37 38 31 32 33 34 35 36 37 38 |1234567812345678|
00000010 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 |.....|
00000020
[09/15/24] seed@VM:~/.../Task4$
```

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):

The screenshot shows two terminal windows. The left window, titled 'ecb\_file.txt', contains the original text: "1 This is a sample text to create a file with more than 1000 bytes". The right window, titled 'ecb\_file\_dec.txt', shows the decrypted text. In the 55th byte position, the character 'A' has been replaced by a corrupted value, specifically the byte value 0x00.

```
[09/16/24]seed@VM:~/.../ECB$ yes "This is a sample text to create a file with more than 1000 bytes" | head -c 1000 > ecb_file.txt
[09/16/24]seed@VM:~/.../ECB$ stat --format=%s ecb_file.txt
1000
[09/16/24]seed@VM:~/.../ECB$ openssl enc -aes-128-ecb -e -in ecb_file.txt -out ecb_file_enc.txt -K 00112233445566778889aabcccddeff
[09/16/24]seed@VM:~/.../ECB$ openssl enc -aes-128-ecb -d -in ecb_file_enc.txt -out ecb_file_dec.txt -K 00112233445566778889aabcccddeff
[09/16/24]seed@VM:~/.../ECB$
```

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):

The screenshot shows two terminal windows. The left window, titled 'cbc\_file.txt', contains the original text: "1 This is a sample text to create a file with more than 1000 bytes". The right window, titled 'cbc\_file\_dec.txt', shows the decrypted text. Both the 55th and 56th bytes are corrupted, appearing as 'E-Y' and 'K=AhOb4' respectively, indicating that the corruption in one block has propagated to the next due to the chaining nature of CBC mode.

```
[09/16/24]seed@VM:~/.../CBC$ yes "This is a sample text to create a file with more than 1000 bytes" | head -c 1000 > cbc_file.txt
[09/16/24]seed@VM:~/.../CBC$ stat --format=%s cbc_file.txt
1000
[09/16/24]seed@VM:~/.../CBC$ openssl enc -aes-128-cbc -e -in cbc_file.txt -out cbc_file_enc.txt -K 00112233445566778889aabcccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/16/24]seed@VM:~/.../CBC$ openssl enc -aes-128-cbc -d -in cbc_file_enc.txt -out cbc_file_dec.txt -K 00112233445566778889aabcccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/16/24]seed@VM:~/.../CBC$
```

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):**

```
[09/16/24] seed@VM:~/.../CFB$ yes "This is a sample text to create a file with more than 1000 bytes" | head -c 1000 > cfb_file.txt
[09/16/24] seed@VM:~/.../CFB$ stat --format=%s cfb_file.txt
1000
[09/16/24] seed@VM:~/.../CFB$ openssl enc -aes-128-cfb -e -in cfb_file.txt -out cfb_file_enc.txt -K 00112233445566778889abbccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/16/24] seed@VM:~/.../CFB$ openssl enc -aes-128-cfb -d -in cfb_file_enc.txt -out cfb_file_dec.txt -K 00112233445566778889abbccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/16/24] seed@VM:~/.../CFB$ 
```

**cfb\_file.txt:**

```
1 This is a sample text to create a file with more than 1000 bytes
2 This is a sample text to create a file with more than 1000 bytes
3 This is a sample text to create a file with more than 1000 bytes
4 This is a sample text to create a file with more than 1000 bytes
5 This is a sample text to create a file with more than 1000 bytes
6 This is a sample text to create a file with more than 1000 bytes
7 This is a sample text to create a file with more than 1000 bytes
8 This is a sample text to create a file with more than 1000 bytes
9 This is a sample text to create a file with more than 1000 bytes
10 This is a sample text to create a file with more than 1000 bytes
11 This is a sample text to create a file with more than 1000 bytes
12 This is a sample text to create a file with more than 1000 bytes
13 This is a sample text to create a file with more than 1000 bytes
14 This is a sample text to create a file with more than 1000 bytes
15 This is a sample text to create a file with more than 1000 bytes
16 This is a sample text to 
```

**cfb\_file\_dec.txt:**

```
1 This is a sample text to create a file with more than 0000 bytes
2 This is a sample text to create a file with more than 1000 bytes
3 This is a sample text to create a file with more than 1000 bytes
4 This is a sample text to create a file with more than 1000 bytes
5 This is a sample text to create a file with more than 1000 bytes
6 This is a sample text to create a file with more than 1000 bytes
7 This is a sample text to create a file with more than 1000 bytes
8 This is a sample text to create a file with more than 1000 bytes
9 This is a sample text to create a file with more than 1000 bytes
10 This is a sample text to create a file with more than 1000 bytes
11 This is a sample text to create a file with more than 1000 bytes
12 This is a sample text to create a file with more than 1000 bytes
13 This is a sample text to create a file with more than 1000 bytes
14 This is a sample text to create a file with more than 1000 bytes
15 This is a sample text to create a file with more than 1000 bytes
16 This is a sample text to 
```

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):**

```
[09/16/24] seed@VM:~/.../OFB$ yes "This is a sample text to create a file with more than 1000 bytes" | head -c 1000 > ofb_file.txt
[09/16/24] seed@VM:~/.../OFB$ stat --format=%s ofb_file.txt
1000
[09/16/24] seed@VM:~/.../OFB$ openssl enc -aes-128-ofb -e -in ofb_file.txt -out ofb_file_enc.txt -K 00112233445566778889abbccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/16/24] seed@VM:~/.../OFB$ openssl enc -aes-128-ofb -d -in ofb_file_enc.txt -out ofb_file_dec.txt -K 00112233445566778889abbccddeff -iv 0102030405060708090a0b0c0d0e0f10
[09/16/24] seed@VM:~/.../OFB$ 
```

**ofb\_file.txt:**

```
1 This is a sample text to create a file with more than 1000 bytes
2 This is a sample text to create a file with more than 1000 bytes
3 This is a sample text to create a file with more than 1000 bytes
4 This is a sample text to create a file with more than 1000 bytes
5 This is a sample text to create a file with more than 1000 bytes
6 This is a sample text to create a file with more than 1000 bytes
7 This is a sample text to create a file with more than 1000 bytes
8 This is a sample text to create a file with more than 1000 bytes
9 This is a sample text to create a file with more than 1000 bytes
10 This is a sample text to create a file with more than 1000 bytes
11 This is a sample text to create a file with more than 1000 bytes
12 This is a sample text to create a file with more than 1000 bytes
13 This is a sample text to create a file with more than 1000 bytes
14 This is a sample text to create a file with more than 1000 bytes
15 This is a sample text to create a file with more than 1000 bytes
16 This is a sample text to 
```

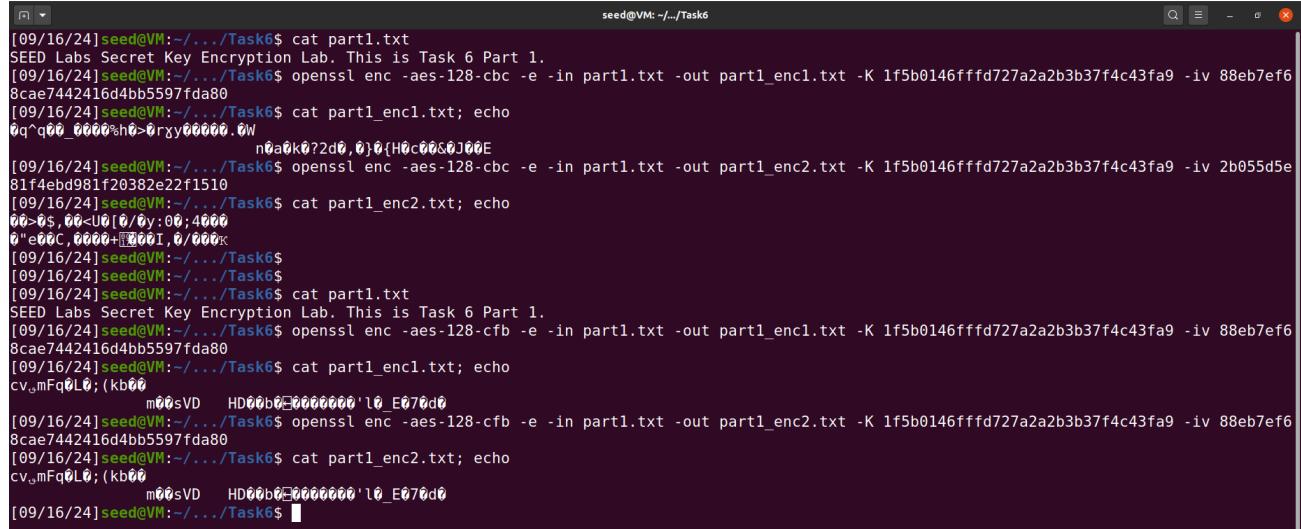
**ofb\_file\_dec.txt:**

```
1 This is a sample text to create a file with more than 0000 bytes
2 This is a sample text to create a file with more than 1000 bytes
3 This is a sample text to create a file with more than 1000 bytes
4 This is a sample text to create a file with more than 1000 bytes
5 This is a sample text to create a file with more than 1000 bytes
6 This is a sample text to create a file with more than 1000 bytes
7 This is a sample text to create a file with more than 1000 bytes
8 This is a sample text to create a file with more than 1000 bytes
9 This is a sample text to create a file with more than 1000 bytes
10 This is a sample text to create a file with more than 1000 bytes
11 This is a sample text to create a file with more than 1000 bytes
12 This is a sample text to create a file with more than 1000 bytes
13 This is a sample text to create a file with more than 1000 bytes
14 This is a sample text to create a file with more than 1000 bytes
15 This is a sample text to create a file with more than 1000 bytes
16 This is a sample text to 
```

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

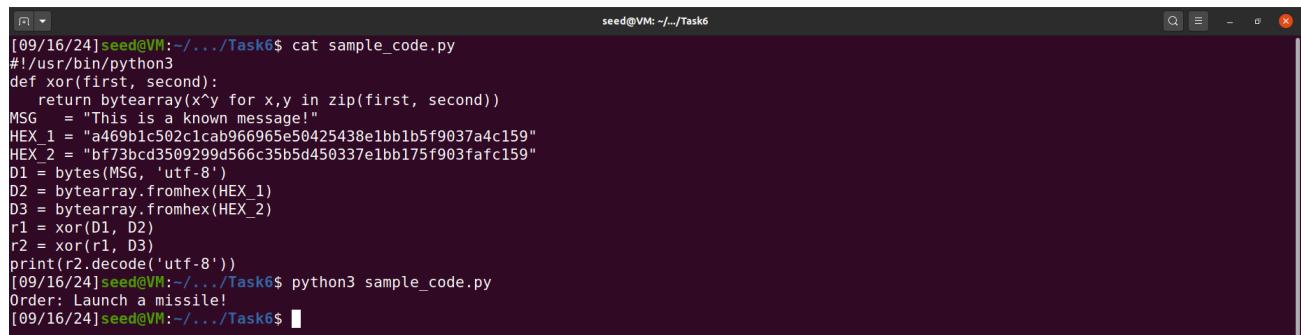


```
[09/16/24]seed@VM:~/.../Task6$ cat part1.txt
SEED Labs Secret Key Encryption Lab. This is Task 6 Part 1.
[09/16/24]seed@VM:~/.../Task6$ openssl enc -aes-128-cbc -e -in part1.txt -out part1_enc1.txt -K 1f5b0146ffffd727a2a2b3b37f4c43fa9 -iv 88eb7ef6
8cae7442416d4bb597fd80
[09/16/24]seed@VM:~/.../Task6$ cat part1_enc1.txt; echo
0q^q00_0000%0>0xy0000.0W
n0ak0?2d0,0}0{H0c0000J00E
[09/16/24]seed@VM:~/.../Task6$ openssl enc -aes-128-cbc -e -in part1.txt -out part1_enc2.txt -K 1f5b0146ffffd727a2a2b3b37f4c43fa9 -iv 2b055d5e
81f4ebd981f20382e22f1510
[09/16/24]seed@VM:~/.../Task6$ cat part1_enc2.txt; echo
00>0$,00<0|0/0y:0;0000
0"e0C_0000+000I_0/000K
[09/16/24]seed@VM:~/.../Task6$ 
[09/16/24]seed@VM:~/.../Task6$ cat part1.txt
SEED Labs Secret Key Encryption Lab. This is Task 6 Part 1.
[09/16/24]seed@VM:~/.../Task6$ openssl enc -aes-128-cfb -e -in part1.txt -out part1_enc1.txt -K 1f5b0146ffffd727a2a2b3b37f4c43fa9 -iv 88eb7ef6
8cae7442416d4bb597fd80
[09/16/24]seed@VM:~/.../Task6$ cat part1_enc1.txt; echo
cv,mFq0L0;(kb00
m0sVD HD00b0E00000000'10_E070d0
[09/16/24]seed@VM:~/.../Task6$ openssl enc -aes-128-cfb -e -in part1.txt -out part1_enc2.txt -K 1f5b0146ffffd727a2a2b3b37f4c43fa9 -iv 88eb7ef6
8cae7442416d4bb597fd80
[09/16/24]seed@VM:~/.../Task6$ cat part1_enc2.txt; echo
cv,mFq0L0;(kb00
m0sVD HD00b0E00000000'10_E070d0
[09/16/24]seed@VM:~/.../Task6$
```

- 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



```
[09/16/24]seed@VM:~/.../Task6$ cat sample_code.py
#!/usr/bin/python3
def xor(first, second):
    return bytearray(x'y for x,y in zip(first, second))
MSG = "This is a known message!"
HEX_1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159"
HEX_2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc159"
D1 = bytes(MSG, 'utf-8')
D2 = bytearray.fromhex(HEX_1)
D3 = bytearray.fromhex(HEX_2)
r1 = xor(D1, D2)
r2 = xor(r1, D3)
print(r2.decode('utf-8'))
[09/16/24]seed@VM:~/.../Task6$ python3 sample_code.py
Order: Launch a missile!
[09/16/24]seed@VM:~/.../Task6$
```

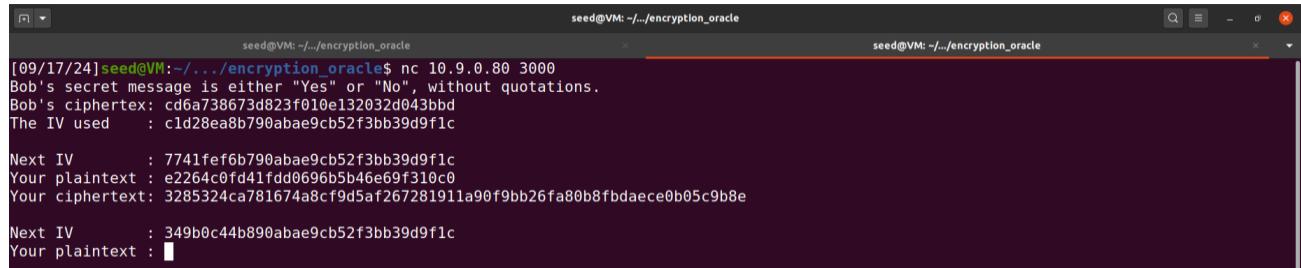
In OFB mode, if the same IV is reused and the attacker knows the plaintext (P1) and corresponding ciphertext (C1), they can decrypt other ciphertexts (like C2) by leveraging the XOR operation. Specifically, since the keystream generated from the IV remains the same, the attacker can recover the keystream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



The screenshot shows two terminal windows side-by-side. Both windows have the title "seed@VM: ~/.../encryption\_oracle". The left window contains the command "nc 10.9.0.80 3000" and the response: "Bob's secret message is either "Yes" or "No", without quotations. Bob's ciphertext: cd6a738673d823f010e132032d043bbd The IV used : c1d28ea8b790abae9cb52f3bb39d9f1c". The right window shows the same command and response, with the addition of "Your plaintext : e2264c0fd41fdd0696b5b46e69f310c0" and "Your ciphertext: 3285324ca781674a8cf9d5af267281911a90f9bb26fa80b8fbdaece0b05c9b8e". Below these, it shows "Next IV : 7741fef6b790abae9cb52f3bb39d9f1c" and "Your plaintext : █".

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.

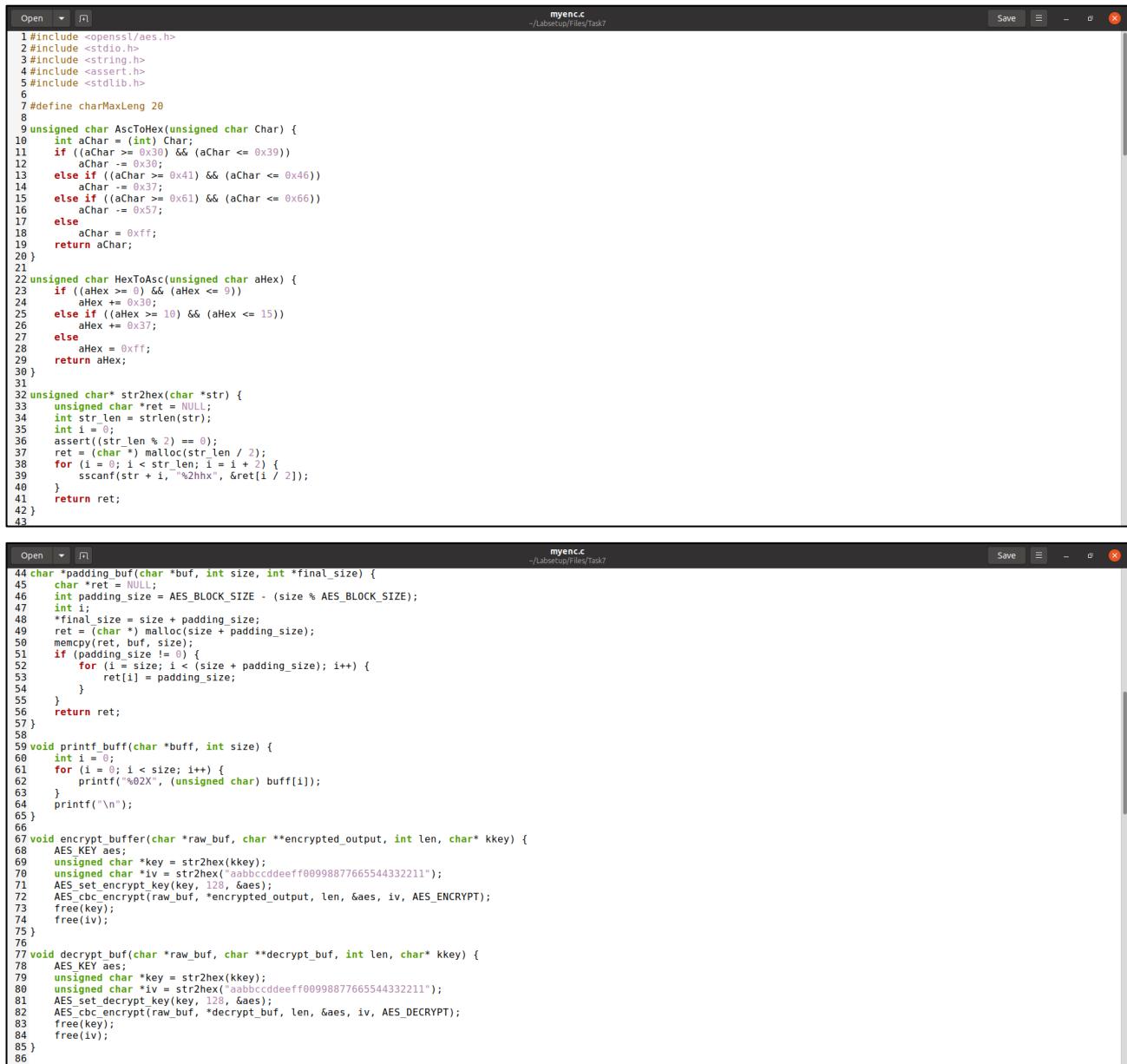
### 8.4 Additional Readings

**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



The image shows two terminal windows side-by-side, both titled "myenc.c". The left window contains code for hex-to-ASCII conversion, and the right window contains code for AES encryption and decryption.

```
myenc.c
~/LabSetup/Files/Task7

1 #include <openssl/aes.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <assert.h>
5 #include <stdlib.h>
6
7 #define charMaxLen 20
8
9 unsigned char AscToHex(unsigned char Char) {
10     int aChar = (int) Char;
11     if ((aChar >= 0x30) && (aChar <= 0x39))
12         aChar -= 0x30;
13     else if ((aChar >= 0x41) && (aChar <= 0x46))
14         aChar -= 0x37;
15     else if ((aChar >= 0x61) && (aChar <= 0x66))
16         aChar -= 0x57;
17     else
18         aChar = 0xff;
19     return aChar;
20 }
21
22 unsigned char HexToAsc(unsigned char aHex) {
23     if ((aHex >= 0) && (aHex <= 9))
24         aHex += 0x30;
25     else if ((aHex >= 10) && (aHex <= 15))
26         aHex += 0x37;
27     else
28         aHex = 0xff;
29     return aHex;
30 }
31
32 unsigned char* str2hex(char *str) {
33     unsigned char *ret = NULL;
34     int str_len = strlen(str);
35     int i = 0;
36     assert((str_len % 2) == 0);
37     ret = (char *) malloc(str_len / 2);
38     for (i = 0; i < str_len; i = i + 2) {
39         sscanf(str + i, "%2hx", &ret[i / 2]);
40     }
41     return ret;
42 }
43
```

```
myenc.c
~/LabSetup/Files/Task7

44 char *padding_buf(char *buf, int size, int *final_size) {
45     char *ret = NULL;
46     int padding_size = AES_BLOCK_SIZE - (size % AES_BLOCK_SIZE);
47     int i;
48     *final_size = size + padding_size;
49     ret = (char *) malloc(size + padding_size);
50     memcpy(ret, buf, size);
51     if (padding_size != 0) {
52         for (i = size; i < (size + padding_size); i++) {
53             ret[i] = padding_size;
54         }
55     }
56     return ret;
57 }
58
59 void printf_buff(char *buff, int size) {
60     int i = 0;
61     for (i = 0; i < size; i++) {
62         printf("%02X", (unsigned char) buff[i]);
63     }
64     printf("\n");
65 }
66
67 void encrypt_buffer(char *raw_buf, char **encrypted_output, int len, char* kkey) {
68     AES_KEY aes;
69     unsigned char *key = str2hex(kkey);
70     unsigned char *iv = str2hex("aahhcdddeeff00998877665544332211");
71     AES_set_encrypt_key(key, 128, &aes);
72     AES_cbc_encrypt(raw_buf, *encrypted_output, len, &aes, iv, AES_ENCRYPT);
73     free(key);
74     free(iv);
75 }
76
77 void decrypt_buf(char *raw_buf, char **decrypt_buf, int len, char* kkey) {
78     AES_KEY aes;
79     unsigned char *key = str2hex(kkey);
80     unsigned char *iv = str2hex("aabcccddeeff00998877665544332211");
81     AES_set_decrypt_key(key, 128, &aes);
82     AES_cbc_encrypt(raw_buf, *decrypt_buf, len, &aes, iv, AES_DECRYPT);
83     free(key);
84     free(iv);
85 }
```

```
myenc.c
~/LabSetup/Files/Task7
87 int main(int argc, char* argv[]) {
88     char* target = "764AA26B55A4DA654DF6B19E4BCE00F4ED05E09346FB0E762583CB7DA2AC93A2";
89     FILE* p = NULL;
90
91     if ((p = fopen("words.txt", "r")) == NULL) {
92         printf("ERROR\n");
93         return 1;
94     }
95
96     char buffer[charMaxLeng];
97     char buf2[charMaxLeng];
98     int flag = 0;
99
100    while (!feof(p)) {
101        int i = 0;
102        memset(buffer, '\0', charMaxLeng * sizeof(char));
103        memset(buf2, '\0', charMaxLeng * sizeof(char));
104        fgets(buffer, charMaxLeng, p);
105
106        while (i < charMaxLeng) {
107            buf2[i] = buffer[i];
108            i += 1;
109        }
110
111        size_t len = strlen(buffer);
112        if (len == 1) continue;
113
114        char *raw_buf = NULL;
115        char *after_padding_buf = NULL;
116        int padding_size = 0;
117        char *encrypted_output = NULL;
118        char *decrypt_buf = NULL;
119
120        i = 0;
121        unsigned char* key = NULL;
122        key = (unsigned char*) malloc(33);
123
124        while (i < strlen(buffer)) {
125            unsigned char letter = buffer[i];
126            key[2 * i] = HexToAsc(letter / 0x10);
127            key[2 * i + 1] = HexToAsc(letter % 0x10);
128            ++i;
129            if (i == 0x0f || buffer[i] < 0x20) break;
130        }
131    }
```

```
myenc.c
~/LabSetup/Files/Task7
131
132    while (i < 0x10) {
133        key[2 * i] = '2';
134        key[2 * i + 1] = '3';
135        ++i;
136    }
137
138    key[0x20] = '\0';
139    raw_buf = (char*) malloc(21);
140    memcpy(raw_buf, "This is a top secret.", 21);
141
142    after_padding_buf = padding_buf(raw_buf, 21, &padding_size);
143    encrypted_output = (char*) malloc(padding_size);
144
145    encrypt_buffer(after_padding_buf, &encrypted_output, padding_size, key);
146
147    i = 0;
148    char temp = '\0';
149    flag = 1;
150
151    while (i < padding_size) {
152        temp = HexToAsc((unsigned char) encrypted_output[i] / 0x10);
153        if (temp != target[2 * i]) {
154            flag = 0;
155            break;
156        }
157        temp = HexToAsc((unsigned char) encrypted_output[i] % 0x10);
158        if (temp != target[2 * i + 1]) {
159            flag = 0;
160            break;
161        }
162        i += 1;
163    }
164
165    if (flag == 0) {
166        continue;
167    }
168
169    printf("%s", buf2);
170    //print_buff(encrypted_output, padding_size);
171    printf("\n", target);
172
173    free(raw_buf);
174    free(after_padding_buf);
175 }
```

```
myenc.c
~/LabSetup/Files/Task7
175     free(encrypted_output);
176     free(decrypt_buf);
177     break;
178 }
179
180 fclose(p);
181 return 0;
182 }
```

```
seed@VM: ~/.../Task7$ gcc -o myenc myenc.c -lcrypto
[09/17/24] seed@VM:~/.../Task7$ ./myenc
Syracuse
764AA26B55A4DA654DF6B19E4BCE00F4ED05E09346FB0E762583CB7DA2AC93A2
[09/17/24] seed@VM:~/.../Task7$
```

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.