

**Faculty of Engineering & Technology**

**Electrical & Computer Engineering Department**

**ENCS532**

**APPLIED CRYPTOGRAPHY**

**Crypto lab**

**Done by:**

**Amal Ziad-1192141**

**Duha Jarrar-1171742**

**Instructor:**

**Hannah Zughbi**

**Date:**

**2/11/2021**

# **Abstract:**

The aim of this experiment is to get knowledge about how cryptography works using concepts of secret-key encryption and other attacks. Also to get experience on encryption algorithms, encryption modes, padding, and initial vector. Finally, to expose common mistakes made by developers in using encryption algorithms and modes.

# **Table of Contents**

[**Abstract:** I](#_Toc87177910)

[**Table of Contents** II](#_Toc87177911)

[**Task 1: Frequency Analysis** 3](#_Toc87177912)

[**1.1.** **Finding the Key:** 3](#_Toc87177913)

[**1.2.** **Generate our Key:** 5](#_Toc87177914)

[**Task 2: Encryption using Different Ciphers and Modes** 6](#_Toc87177915)

[**Task 3: Encryption Mode – ECB vs. CBC** 7](#_Toc87177916)

[**Task 4: Padding** 10](#_Toc87177917)

[**Task 5: Error Propagation – Corrupted Cipher Text** 14](#_Toc87177918)

[**Task 6: Initial Vector (IV) and Common Mistakes** 18](#_Toc87177919)

[6.1. IV Experiment 18](#_Toc87177920)

[6.2. Common Mistake: Use the Same IV 21](#_Toc87177921)

[6.3. Common Mistake: Use a Predictable IV 24](#_Toc87177922)

[**Conclusion:** 28](#_Toc87177923)

# **Task 1: Frequency Analysis**

## **Finding the Key:**

It is well-known that monoalphabetic substitution cipher (also known as monoalphabetic cipher) is not secure, because it can be subjected to frequency analysis[[1]](#footnote-1). In this task, we have a cipher-text that is encrypted using a monoalphabetic cipher, and we have to find out the original text using frequency analysis.

So here, we replaced a letter in the alphabet with another in the plain text, depended on letters’ frequencies, because of that we can predict the original text completely or partially. After analyzing the text which we inputted it in a text file shown in figure 1.1.1

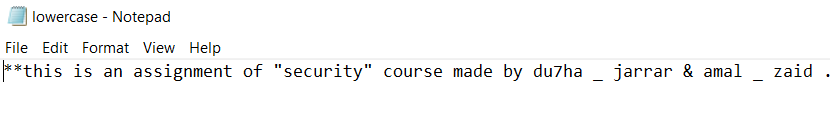


Figure 1.1.1: original text

And did some operations on it which is shown in figure 1.1.2,

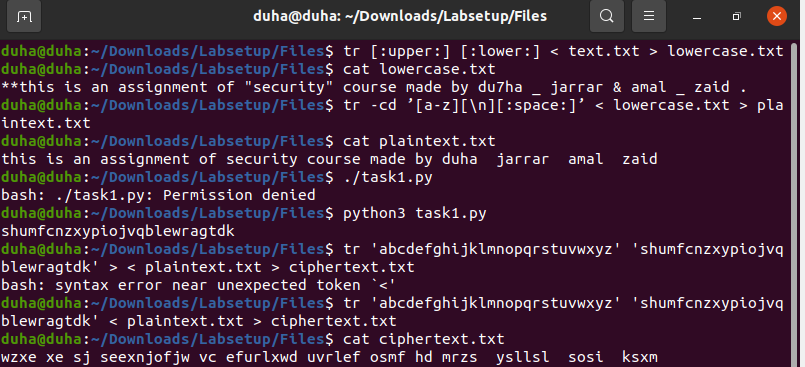


Figure 1.1.2: encrypting a text

To get the cipher text shown in figure 1.1.3

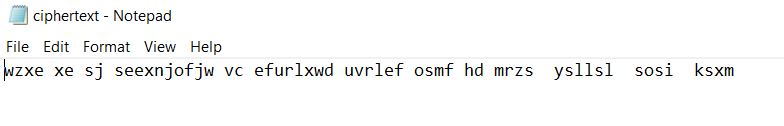


Figure 1.1.3: encrypting a text

We noticed the following:

* w z x e 🡺 t h i s
* x e 🡺 i s
* s e e x n j o f j w 🡺 a s s i g n m e n t

and so on…

After analyzing these words, some words started to become more familiar and all the other letters could be obtained until we found the key.

* **Alphabetic**: abcdefghijklmnopqrstuvwxyz
* **Cipher**: shumfcnzxypiojvqblewragtdk

By applying this command, it decrypts the cipher text to plain text as shown in figure 1.1.4:

tr ‘shumfcnzxypiojvqblewragtdk’ ‘abcdefghijklmnopqrstuvwxyz’ < ciphertext.txt > plaintext.txt

then we got the original text we encrypted before as shown in figure1.1.5.

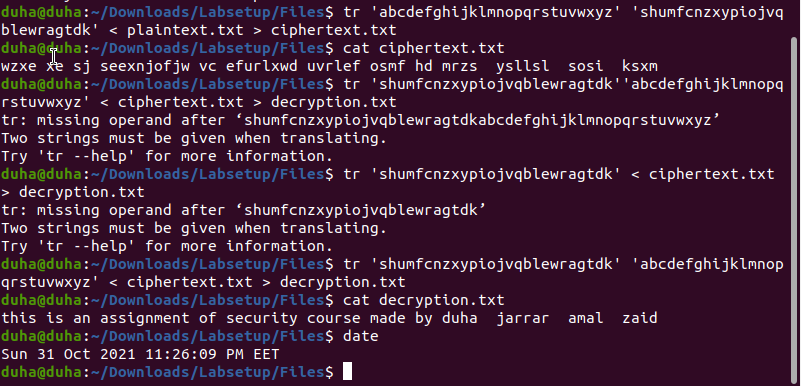


Figure 1.1.4: decrypting a text

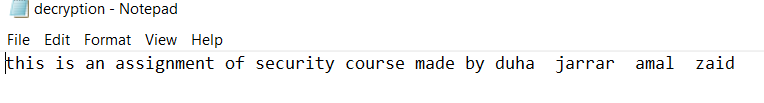


Figure 1.1.5: decrypted text

## **Generate our Key:**

Here we created a text file with our own text, and used a python code shown below, to generate a random key of length 26.

Note: the method is the same as ‘finding the key’ task.

* Python code:

#!/bin/env python3

Import random

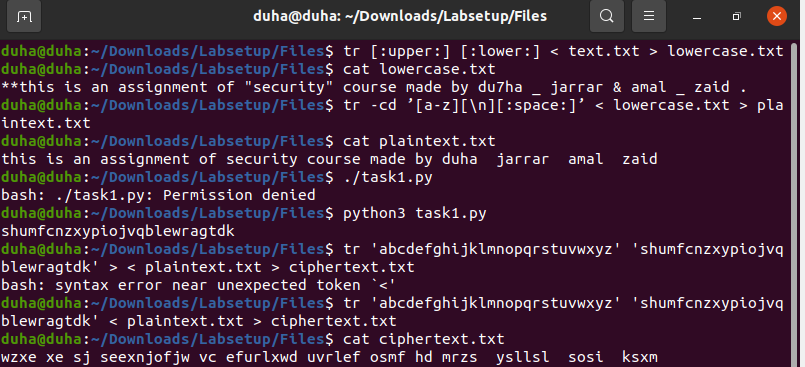
s= “abcdefghijklmnopqrstuvwxyz”

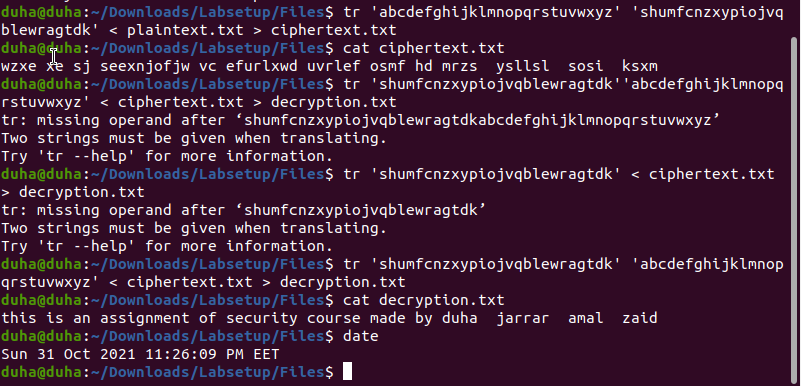
list=random.sample(s, len(s))

key=’’.join(list)

print(key)

We simplified the original text by converting all upper cases to lower cases, then removed all the punctuations and numbers (Keeping the spaces). Figure 1.2 shows all the details and the output for each command. After generating our random key, we encrypted the original text by using the key and **tr** command. Then decrypted the cipher text by using the same key and **tr** command to get the original text.

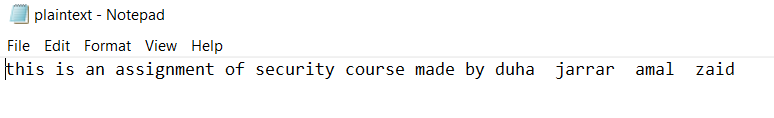




*Figure 1.2: encryption and decryption*

# **Task 2: Encryption using Different Ciphers and Modes**

In this task, we tried various encryption algorithms and modes. We used **openssl enc** command to encrypt/decrypt a text file containing the text shown in figure 2.1.

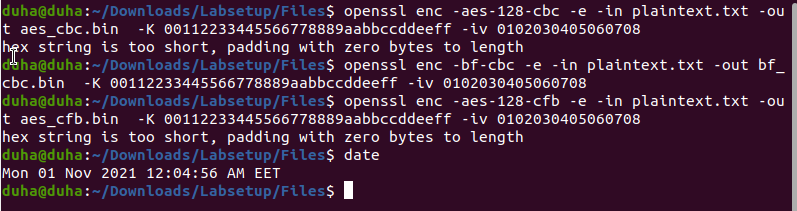


*Figure 2.1: plain text*

We applied different 3 modes of operation for AES:

* AES-128-CBC (Cipher Block Chaining)
* BF-CBC (Cipher Block Chaining for Blowfish)
* AES-182-CFB (Ciphertext FeedBack)

We used the following commands as shown in figure 2.2, creating 3 bin files as shown in figure 2.3.



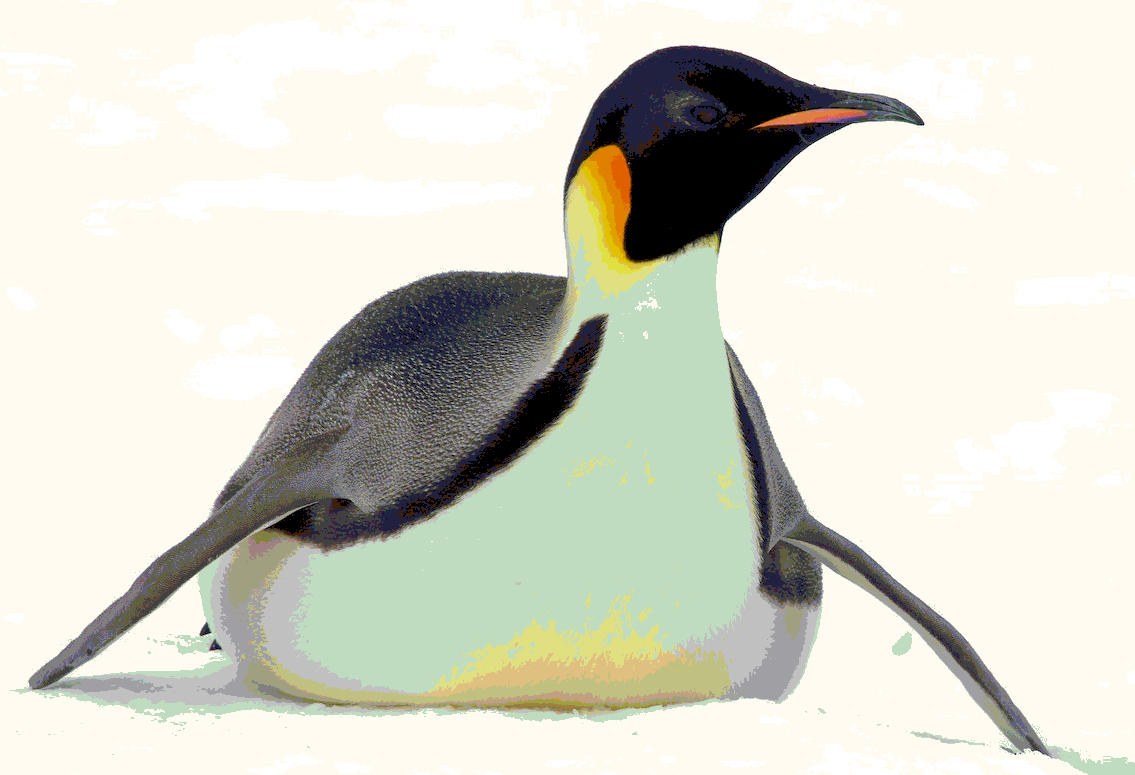
*Figure 2.2: encryption commands*



*Figure 2.3: bin files*

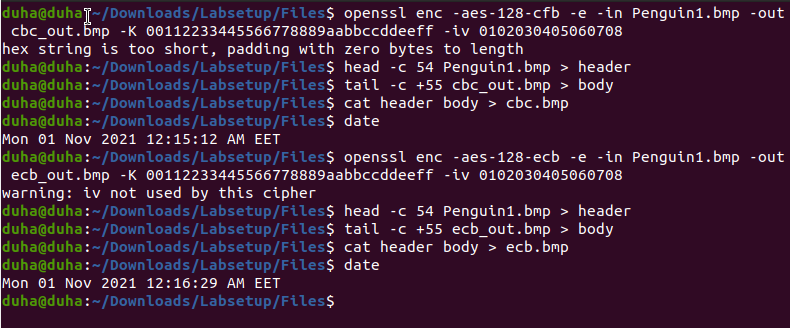
# **Task 3: Encryption Mode – ECB vs. CBC**

In this task, we encrypted a picture penguin1.bmp in figure 3.1, so people without the encryption keys cannot know what is in the picture. We used ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes to encrypt the picture.

**

*Figure 3.1: penguin1.bmp*

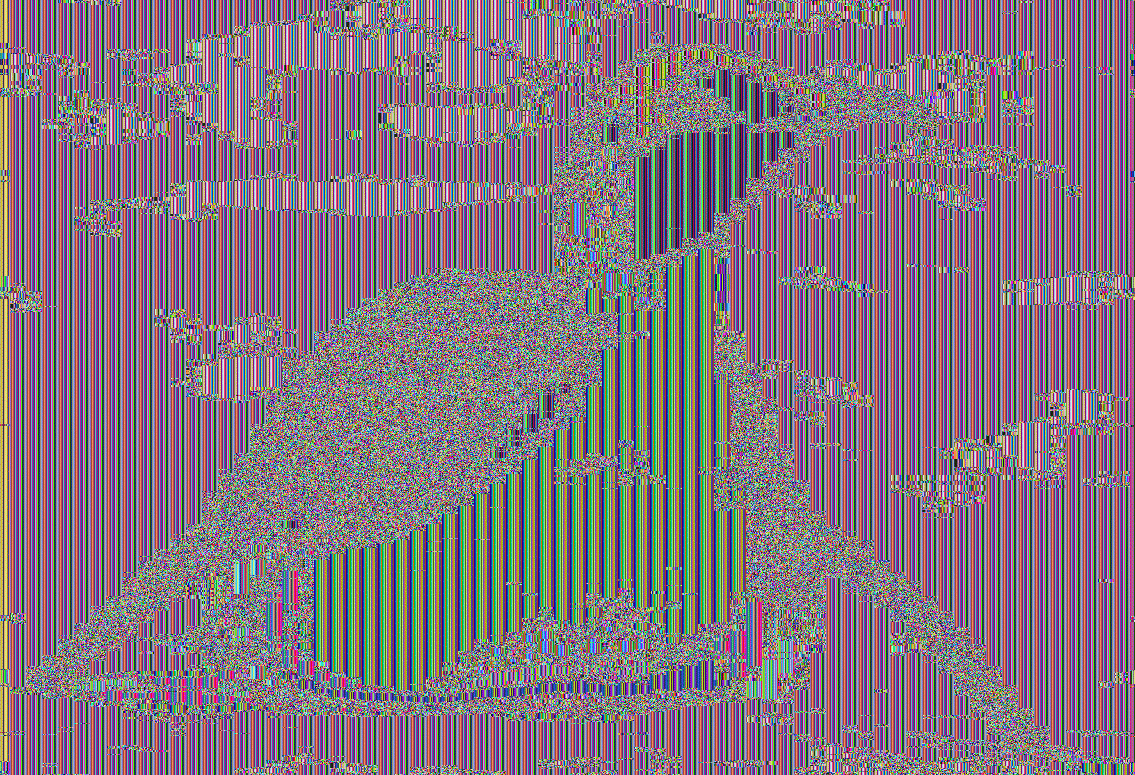
For a bmp file, the first 54 bytes contain the header information about the picture, so we had to set it correctly, so the encrypted file can be treated as a legitimate .bmp file. We replaced the header of the encrypted picture with that of the original picture as in figures bellow. As shown in figures 3.2.



*Figure 3.2: encryption and decryption using CBC*

**

*Figure 3.4: encryption using CBC*

**

*Figure 3.5: encryption using ECB*

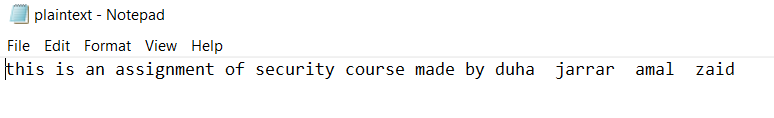
From figures 3.4 & 3.5, we can see that CBC mode completely hides picture’s information, while ECB mode doesn’t. This is usually referred to as “sematic security”.

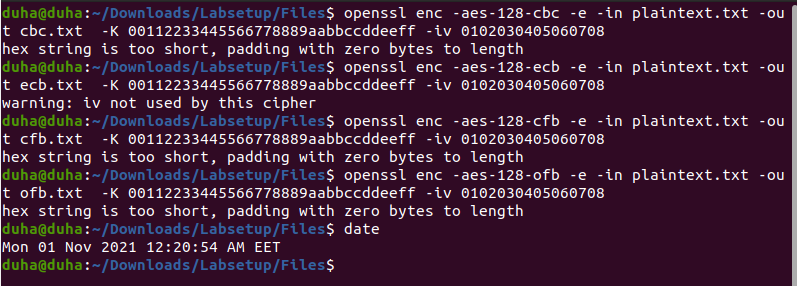
When performing encryption in ECB mode, each block of data is encrypted separately, which results in confidentiality, but this also means that the encryption process is not dynamic, which can reveal the repetition in data and fail to hide the semantics of it. However, when performing encryption in CBC mode, it is cumulative. Each plain text block of data relies on the previous block of data. So, we need an initial vector for the very first block, also, this means if we encrypt 100 similar blocks of data, it will result in 100 different cipher texts in the output, and this will completely hide any information in the data and the output will be completely different.

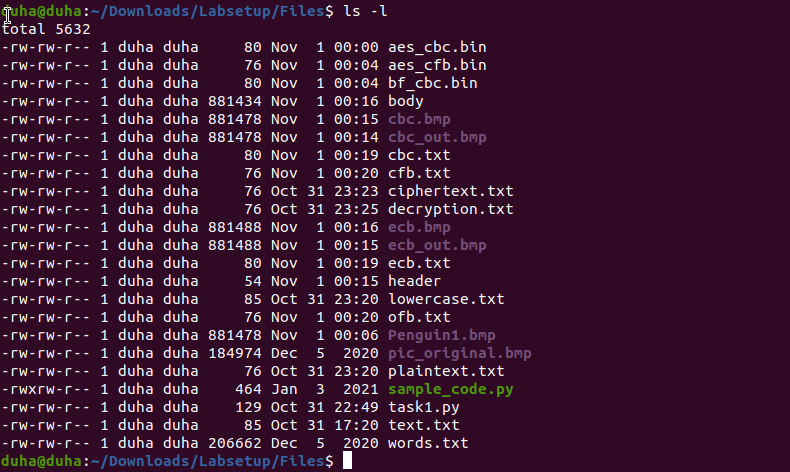
# **Task 4: Padding**

For block ciphers, when the size of a plaintext is not a multiple of the block size, padding may be required. The PKCS#5 padding scheme is widely used by many block ciphers. In this task, we will investigate how this type of padding works.

1. We used ECB, CBC, CFB, and OFB modes to encrypt the file shown in figure 4.1. We used the commands shown in figure 4.1.2 to encrypt the plaintext shown in figure 4.1.1.

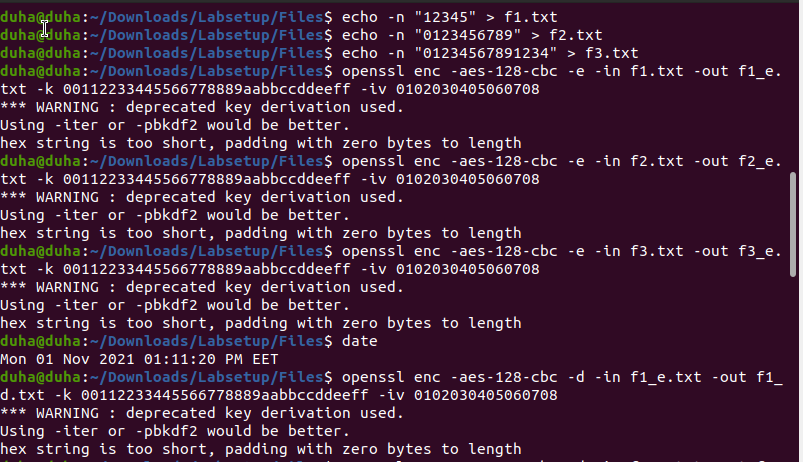
*Figure 4.1.1: plain text*

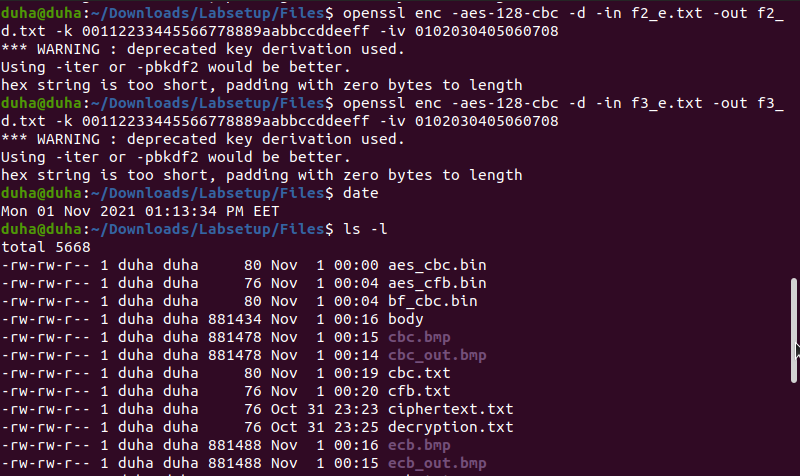
*Figure 4.1.2: commands to encrypt the plain text*

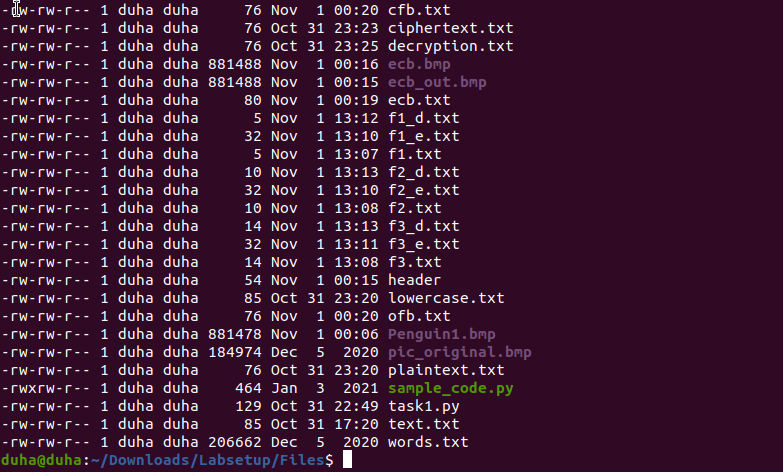
*Figure 4.1.3: sizes for the encrypted files*

We can see that the plain text file has a size of 85 bytes. It can be observed that CBC and ECB have 80 bytes size file for both, while the other two files CFB and OFB maintained the 76 bytes size for both too. The reason for this, is that CBC and ECB have a block size of 16 byte, we can see that the size of the file has increased, which means that padding has been added to the file. However, padding is not needed for CFB and OFB modes since they are stream ciphers not block ciphers, which means that they treat data as a stream .

1. In this part, we created 3 files, which contain 5 bytes, 10 bytes, and 16 bytes, respectively using the command **echo -n** as shown in figure 4.2.1.

*Figure 4.2.1: creation and encryption of the 3 files*



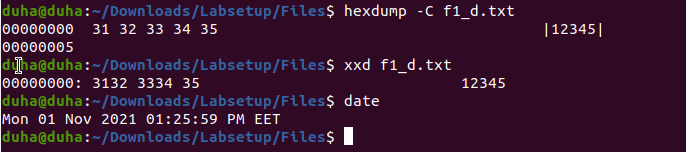


*Figure 4.2.2: sizes of the 3 encrypted and decrypted files*

From figure 4.2.2, we can see that the sizes of the 3 files have changed after encryption because of the padding. Paddings are added to complete the block.

Also we decrypted the files with the option **-nopad** which disables the padding as shown in figure 4.2.2. Because by default, the decryption process removes the padding, which makes it impossible to see. So, using this option will let us see the padded data, and it won’t remove it within the decryption process.

The padding data may not be printable, so we need to use a **hex** tool to display the content, as shown in figure 4.2.3.

* Figure 4.2.3: commands to print the padding content*

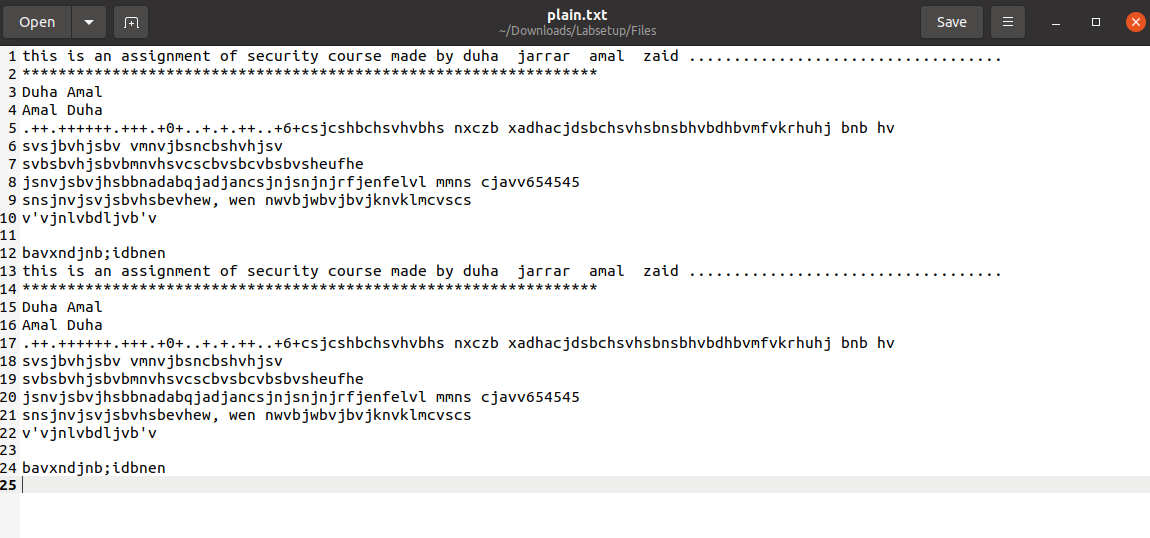
# **Task 5: Error Propagation – Corrupted Cipher Text**

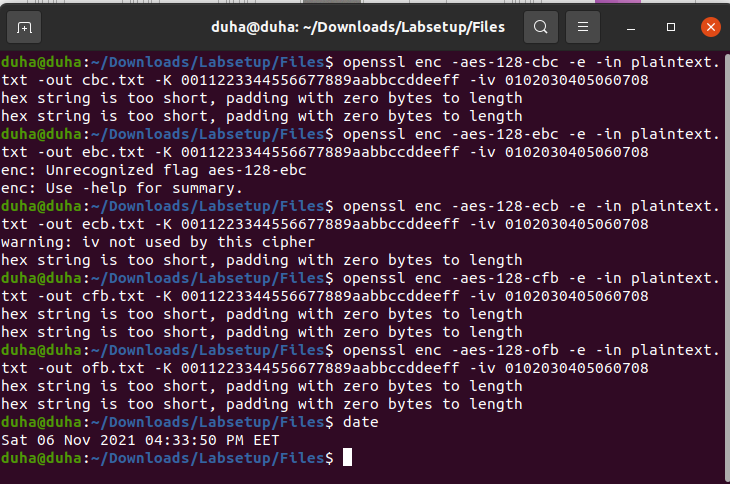
"Error propagation" properties describe how a decryption behaves during bit errors, i.e. how error in one bit cascades to different decrypted bits. Bit errors may occur randomly due to transmission errors. And also bit errors may occur intentionally in attacks. [[2]](#footnote-2)

In this task, we will demonstrate error propagation in CBC, ECB, OFB and CFB. Firstly, we created a file of at least 1000 bytes long. The file with the plain text is shown in figure 5.1 with 1300 bytes long.

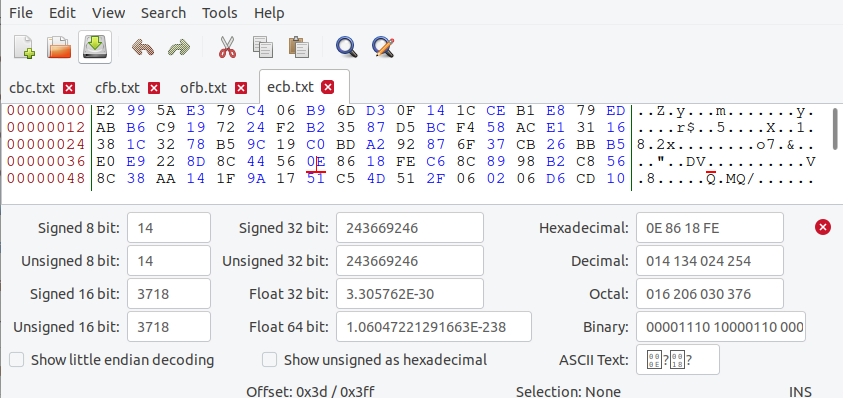
Then we encrypted the file with the 4 modes that has been mentioned, as shown in figure 5.2.

Next, we used **bless** hex editor in order to achieve a corruption in the 55th bit. Then applied decryption to investigate the difference in the avalanche effect.

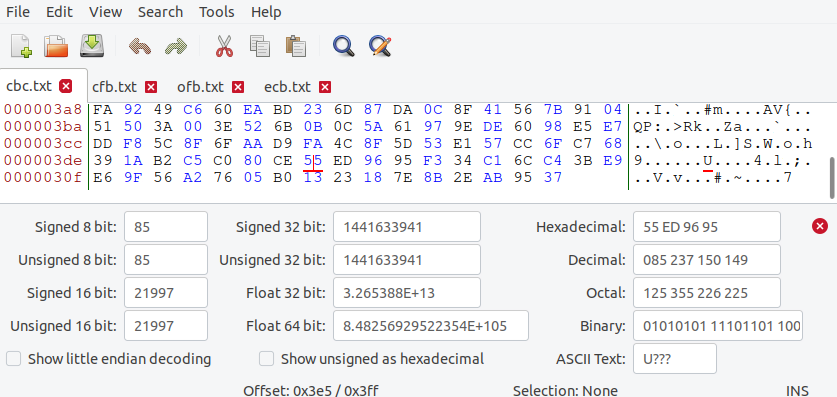
*Figure 5.1: approximately 1300 bytes long file*

*Figure 5.2: encryption commands*

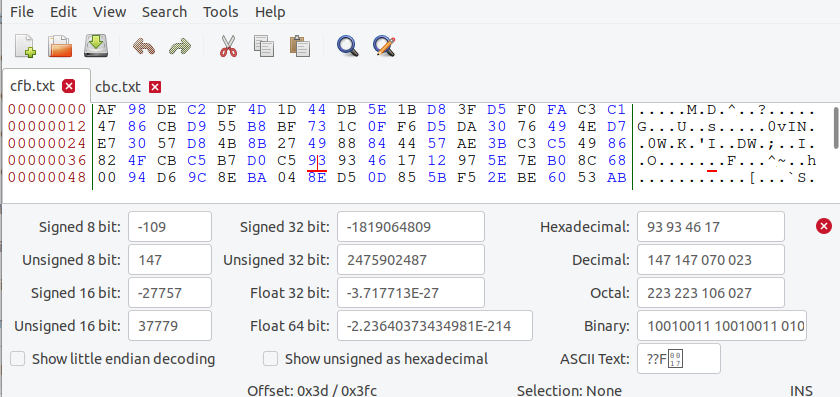
Figures 5.3, 5.4, 5.5 & 5.5 shows the changing of the 55th bit for each file.

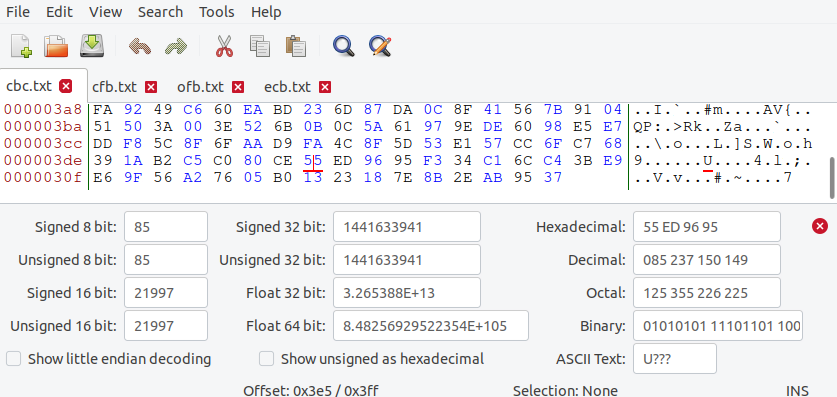


*Figure 5.3: changing the 55th bit from 43 to A9 for ECB*

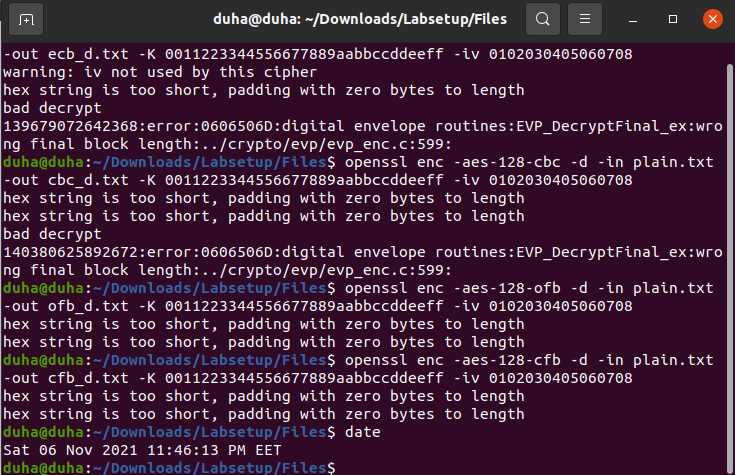


*Figure 5.4: changing the 55th bit from C7 to 12 in CBC*

*Figure 5.5: changing the 55th bit from 2A to C8 in CFB*



*Figure 5.6: changing the 55th bit from 92 to D4 in CBC*

*Figure 5.7: decryption commands*

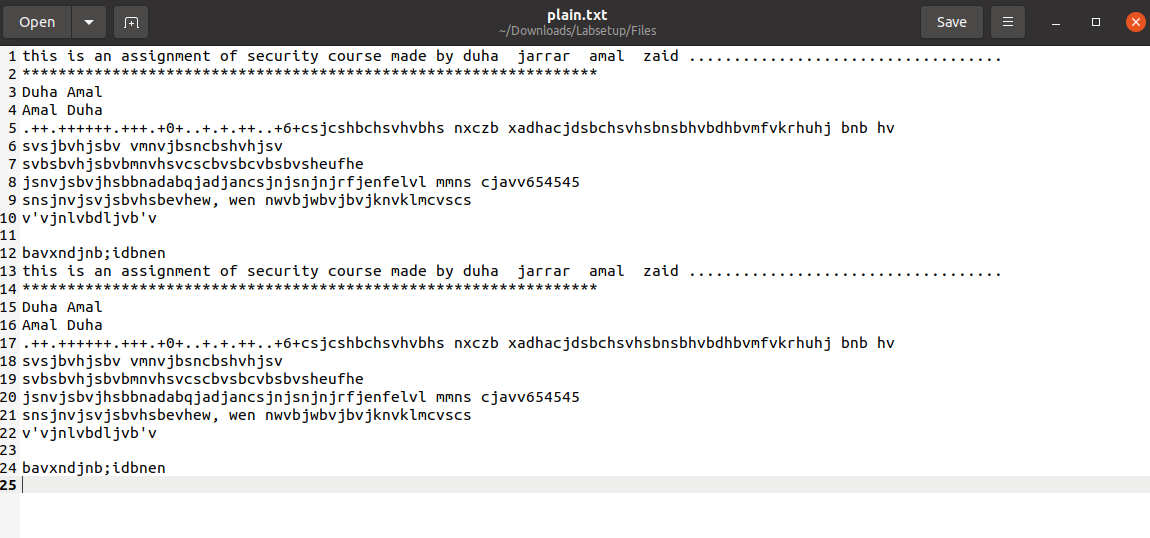
From the results above we noticed that the error propagation will occur in the 4 modes, but in different blocks effect after checking what’s inside each file after decrypting .The error in OFB will only occur in the byte corrupted because the operation of OFB mode has no block dependency[[3]](#footnote-3). In CBC, one block (16 bytes) was corrupted and a small bit of the next one. In both ECB and CFB or more bytes were corrupted, and in OFB, only the single byte affected was not retrieved. In this case, non-of the texts were completely corrupted, but this is not always the case, it didn’t happen this time maybe because of the length of text, but sometimes, due to error propagation, the whole text cannot be retrieved.

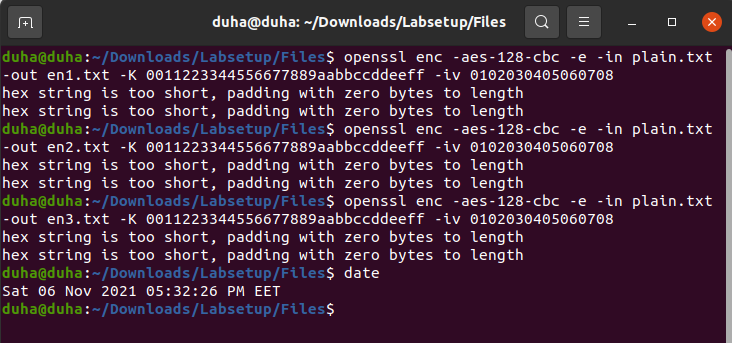
# **Task 6: Initial Vector (IV) and Common Mistakes**

## 6.1. IV Experiment

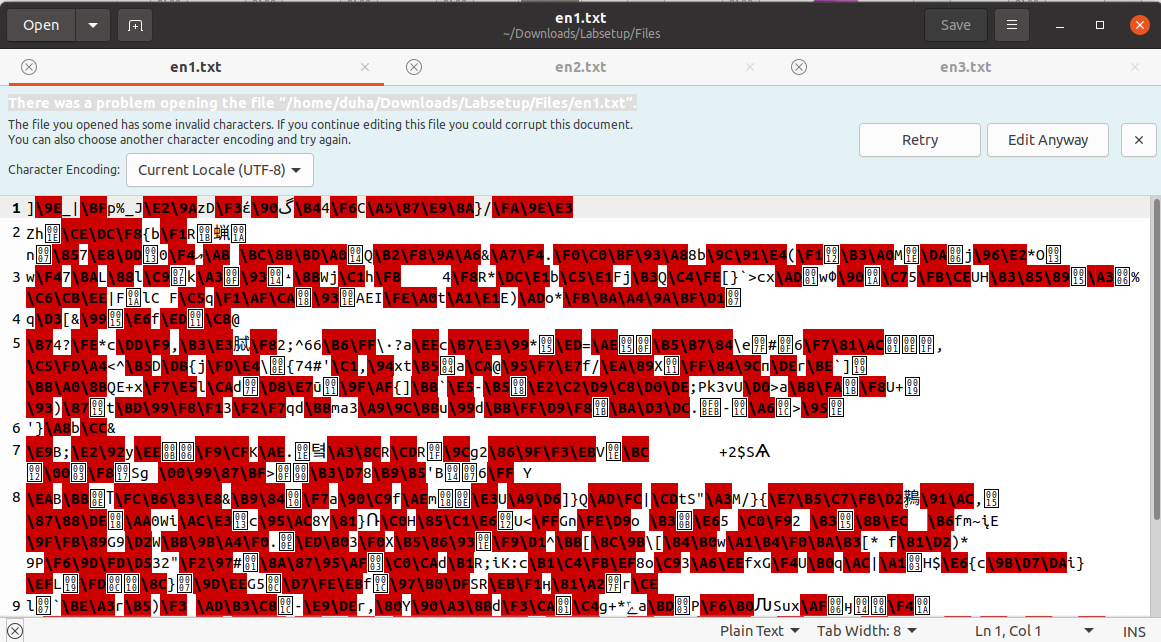
In this part we will explain why IV has to be unique, if we use the same IV in encrypting the same text, the result will be always the same, will makes the mode lack the feature of semantic security.

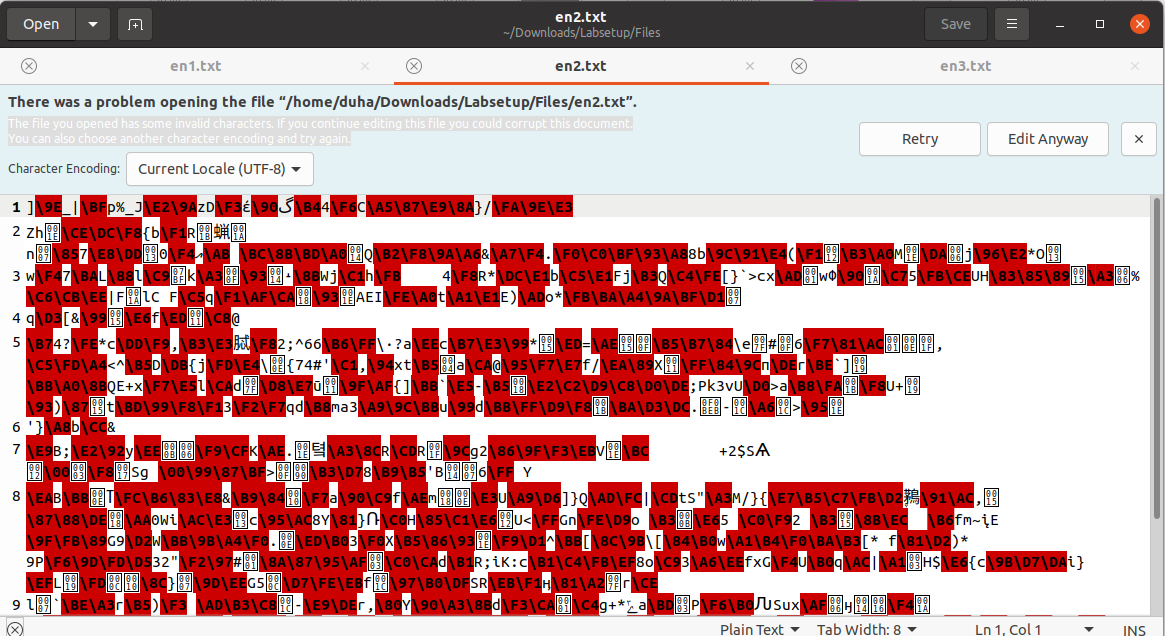
To demonstrate this, we created to file with a plain text as shown in figure 6.1.1, then we encrypted it twice using the same and IV, and once using the same key and different IV, as shown in figure 6.1.2.

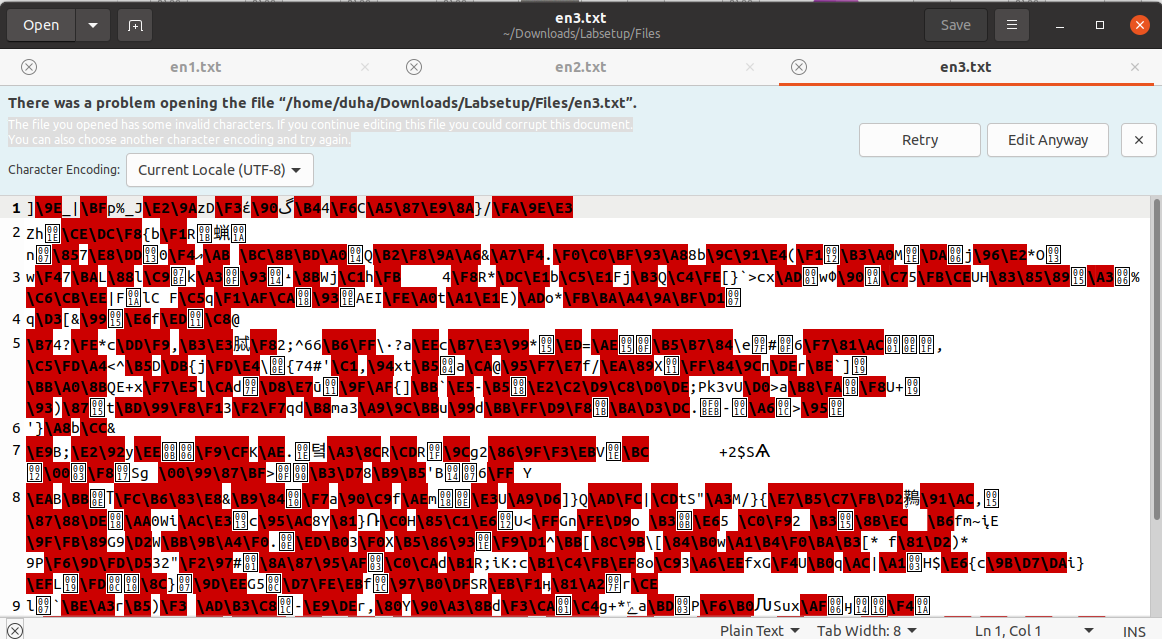
* Figure 6.1.1: Plain text*



*Figure 6.1.2: encryption commands with same IV twice and once with different IV*

*Figure 6.1.3: encrypted file with the same IV*

* Figure 6.1.4: encrypted file with the same IV*

* Figure 6.1.5: encrypted file with different IV*

From figures 6.1.3, 6.1.4 & 6.1.5, we notice that when the IVs are the same, the 2 files (en1 & en2) have exactly the same cipher text, which is not secure. But when we used a different IV for en3, the cipher text is completely different from the other files, so this is more secure.

## 6.2. Common Mistake: Use the Same IV

This part shows that the repeated Initial vector, we tried to figure out the cipher text knowing that the IV is always the same. We have the following information:

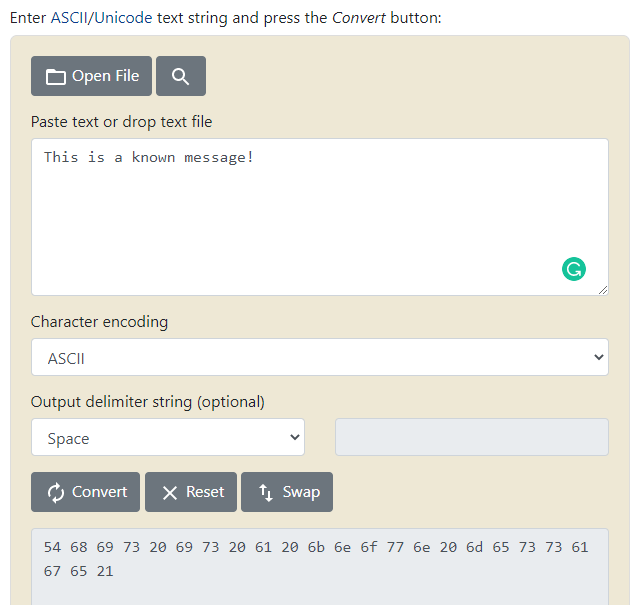
Plaintext (P1): This is a known message!

Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

Plaintext (P2): (unknown to you)

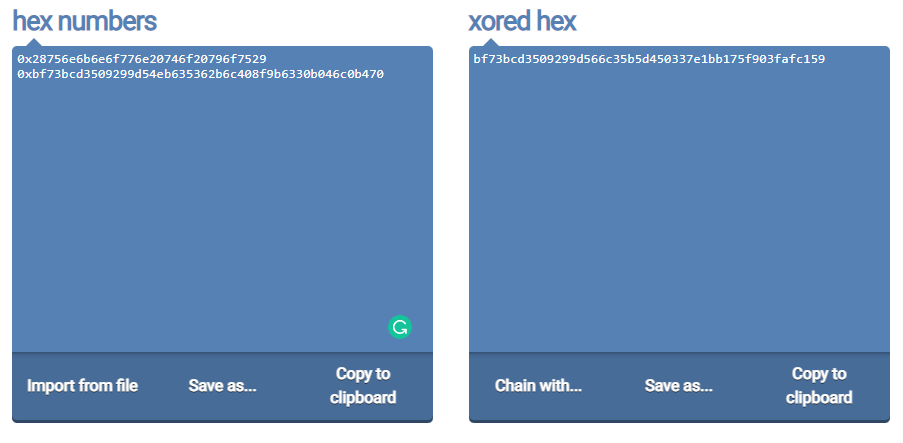
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159

By following the steps bellow:

*Figure 6.2.1: convert P1 from text to hex*

Next, we did **XOR** the Hex value of the plain text (P1) with the Cipher text (C1), as shown in figure 6.2.2. We will get the first key:

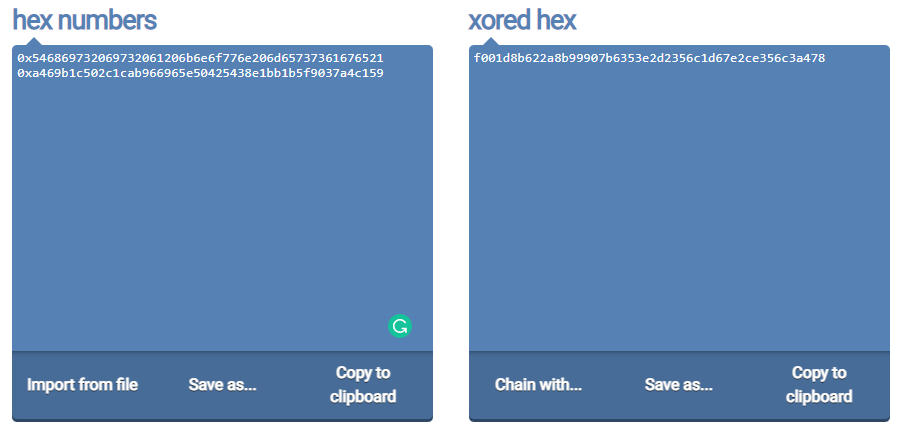
Key from 1st XOR: bf73bcd3509299d566c35b5d450337e1bb175f903fafc159



*Figure 6.2.2: Hex text XOR C1*

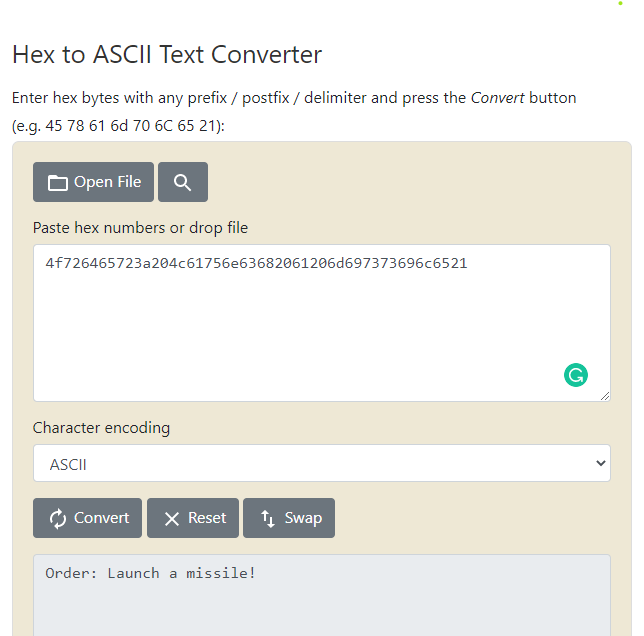
Then, we applied **XOR** the result from the first XOR as shown in figure 6.2.3, we get the following result:

Result of 2nd XOR: f001d8b622a8b9907b6353e2d2356c1d67e2cce3a478

**

*Figure 6.2.3: The first key XOR C2*

After that we only converted the result from the second XOR from hex to text, as shown in figure 6.2.4. And we will get the second plain text P2.

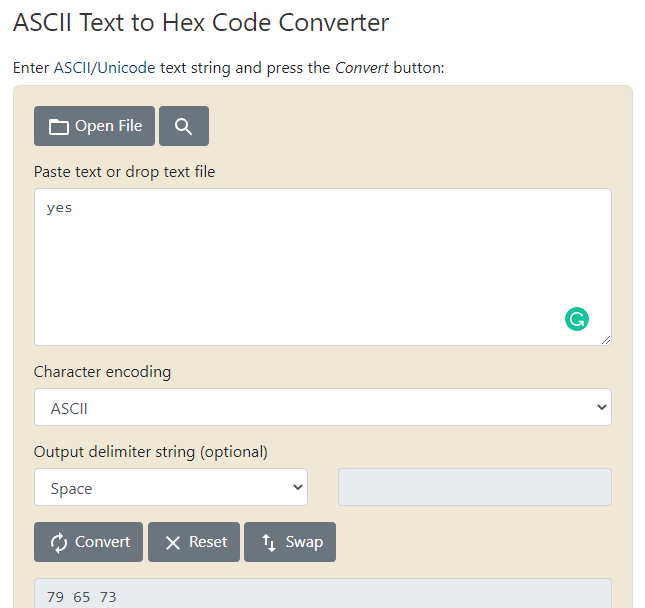
* Figure 6.2.4: The second plain text (P2)*

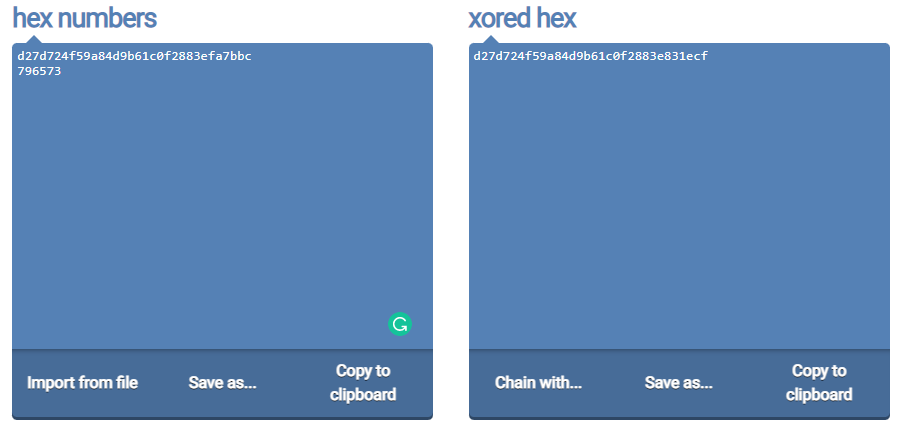
## 6.3. Common Mistake: Use a Predictable IV

This part introduces a new important feature of initial vectors, which is that the initial vector should never be predictable, meaning it should never be only an increment by 1 or something similar to the previous one. Because the same attack that was used in the previous part can be done again with a small difference.

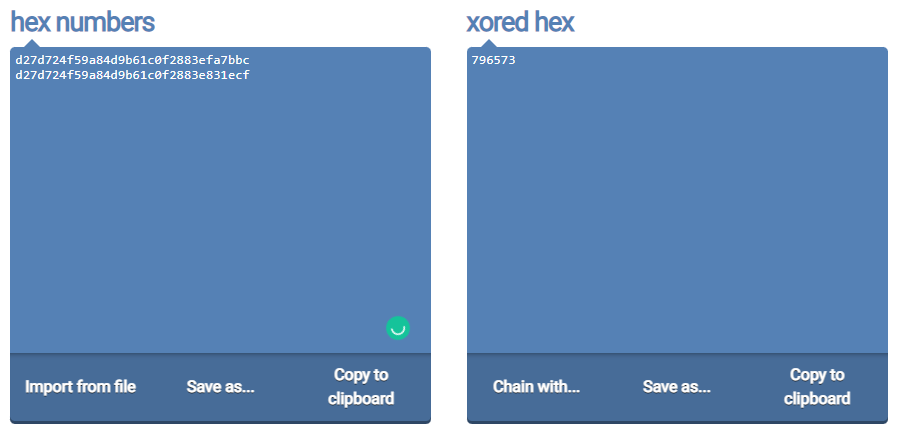
A good cipher should not only tolerate the known-plaintext attack described previously, it should also tolerate the chosen-plaintext attack, which is an attack model for cryptanalysis where the attacker can obtain the ciphertext for an arbitrary plaintext

Depending on the fact that the message is either yes or no, we can send Bob message “yes”, and then he will give us it’s cipher text, then we will XOR it with IV1, and then XOR the result with IV2, if the answer is similar to the first answer with a few differences, and the ASCII is yes, then the original message was yes. If not, it is no. The steps are as shown in the figures below.

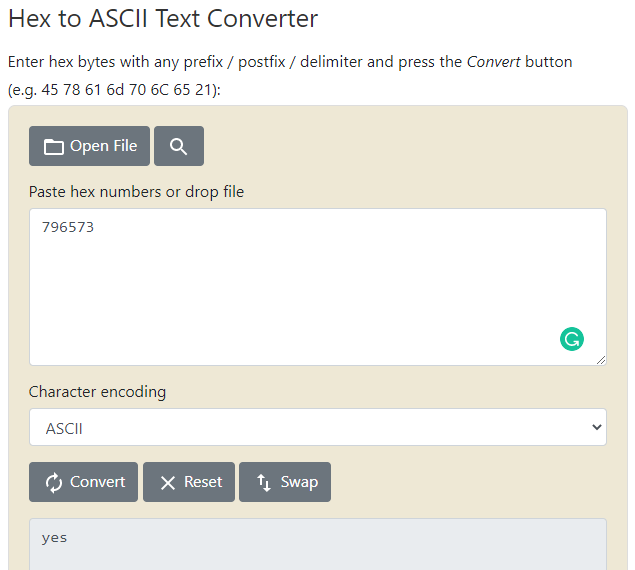
*Figure 6.3.1:* *convert “yes” from text to hex*

**

*Figure 6.3.2:* *XOR “yes” with the first IV*

**

*Figure 6.3.3:* *XOR the result with the second IV*

* Figure 6.3.4:* *convert the result from hex to text*

# **Conclusion:**

Finally we reached the knowledge of how to apply the encryption and decryption algorithms, and how different modes work by applying on different types of messages such that texts and images. Also we learnt how to avoid some common mistakes and behaviors while encrypting which may cause some problems regarding the security of the data and how to less error propagation in our works.

1. https://crypto.interactive-maths.com/monoalphabetic-substitution-ciphers.html [↑](#footnote-ref-1)
2. https://en.wikipedia.org/wiki/Block\_cipher\_mode\_of\_operation#Error\_propagation [↑](#footnote-ref-2)
3. https://en.wikipedia.org/wiki/Block\_cipher\_mode\_of\_operation [↑](#footnote-ref-3)