Authors

fc54462 Inês Martins
fc54399 Miguel Carvalho
fc54392 Santiago Benites

Task 1: Symmetric Encryption using Different Ciphers and Modes

a)

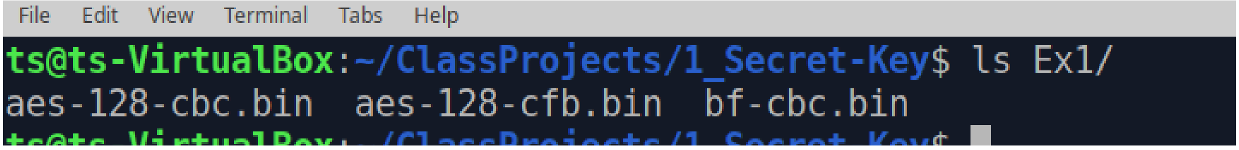
In this exercise we proceeded with the instructions and encrypted the 3 files specified with the following command

```
openssl enc -cipher -e -in plain.txt -out Ex1/newfile.bmp
```

Where -cipher would be subtituted by the used cipher, in our case we used:

- -aes-128-cbc
- -aes-128-cfb
- -aes-128-ecb

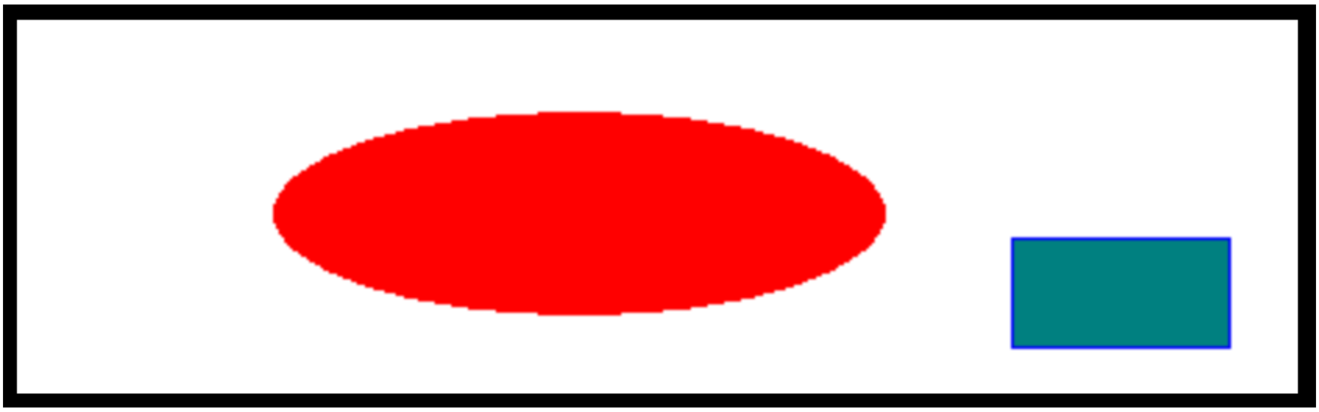
  Obtaining the following files



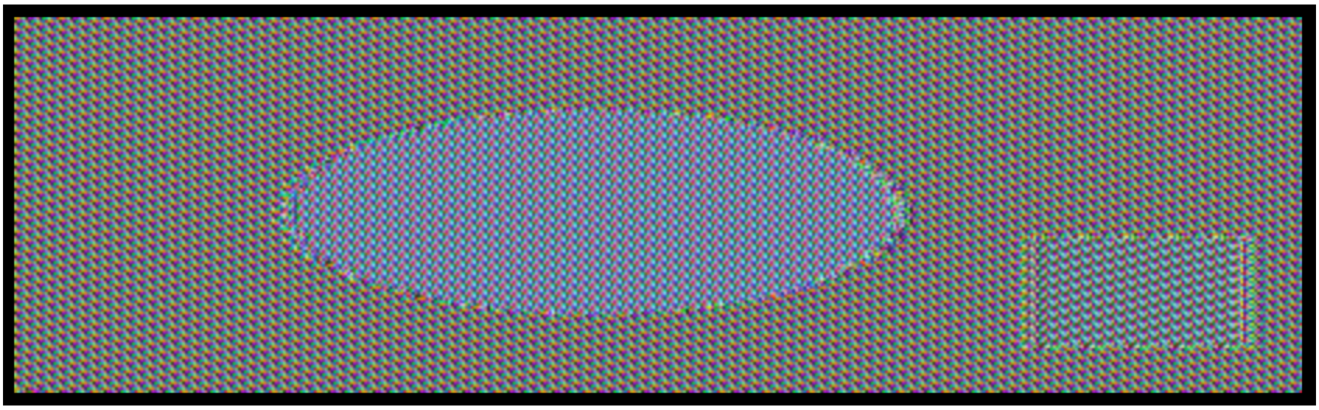
```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key$ ls Ex1/
aes-128-cbc.bin  aes-128-cfb.bin  bf-cbc.bin
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key$
```

b)

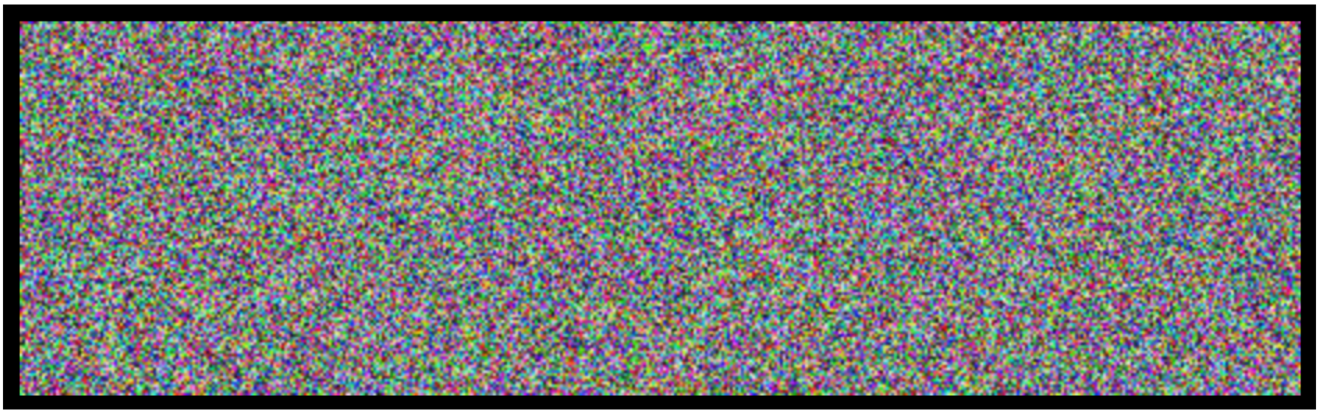
The original image will be :



When encrypting the image with AES on ECB mode we obtain the image:

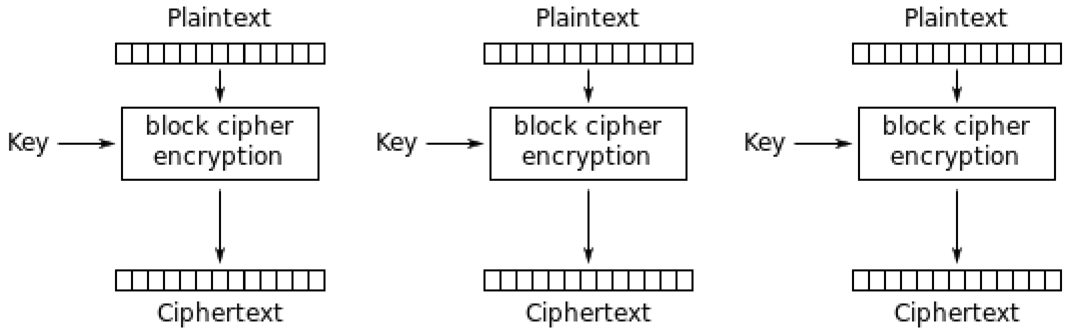


When encrypting the image with AES on CBC mode we obtain the image:

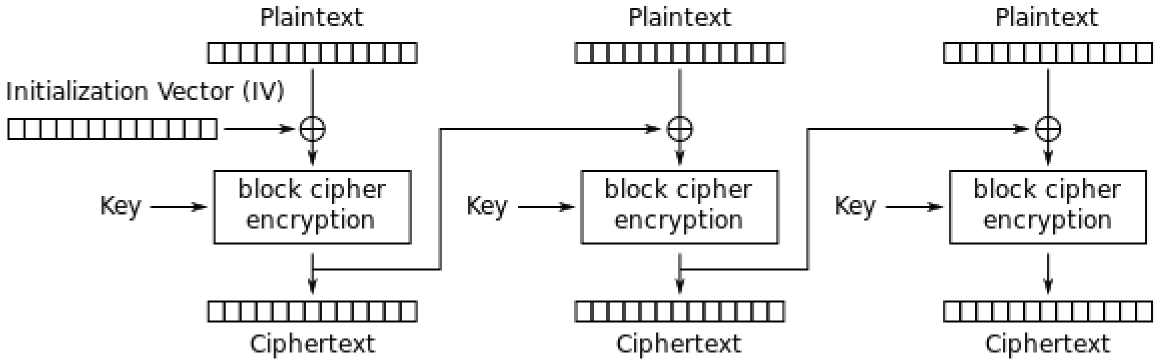


This disparity of results derive from the way ECB and CBC work.
In ECB each encryption doesn't have influence over other subsequent encryption, this lead to every block in AES suffering the same alteration, which in our case leads to every pixel changing colours, but the general pattern of the image staying the same
In CBC each subsequent encryption XOR's its plain text with the past encryption result, this leads to every subsequent encryption being affected by the previous one (The first encryption considering that it doesn't have an previous one, uses an IV). This in our case lead to

every pixel having influence over the following pixel's colour, which destroys the pattern of the original image.
The 2 modes can be seen below.



Electronic Codebook (ECB) mode encryption

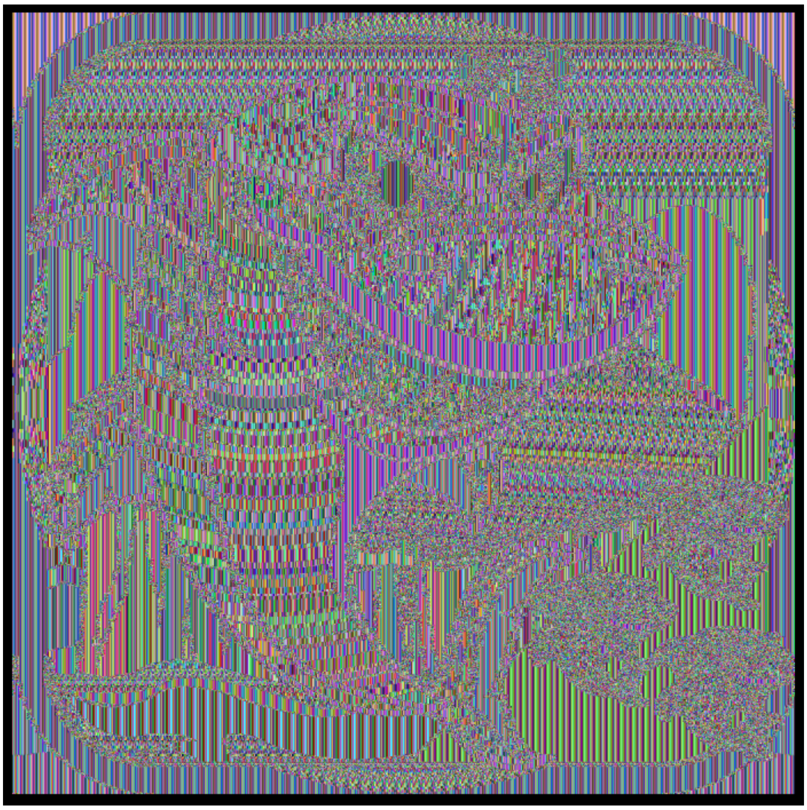


Cipher Block Chaining (CBC) mode encryption

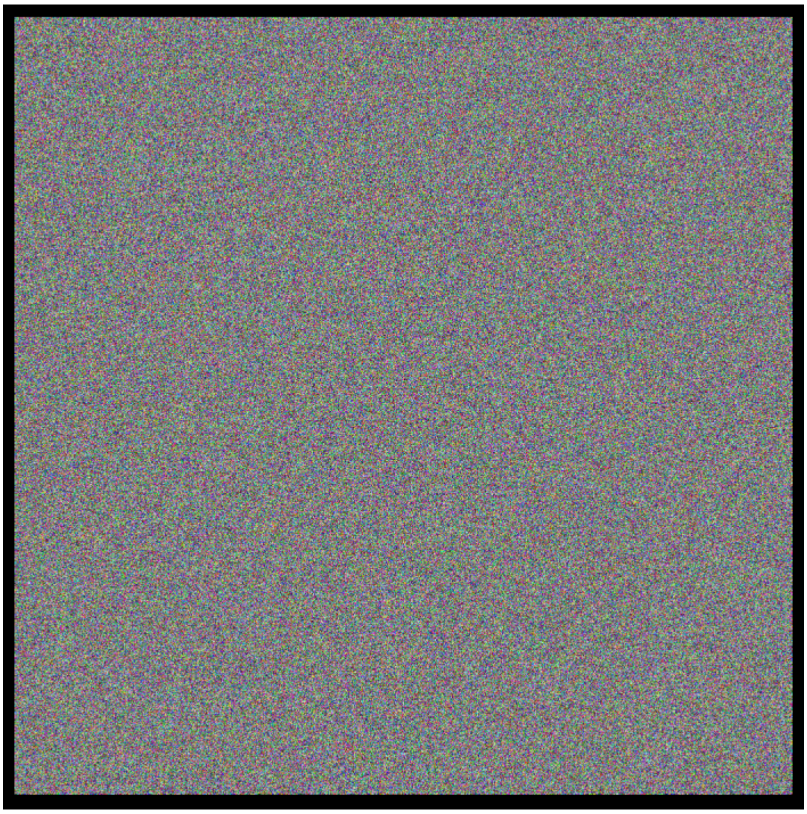
c)

This same result can also be see in the shark picture
The original image will be :

When encrypting the image with AES on ECB mode we obtain the image:



When encrypting the image with AES on CBC mode we obtain the image:



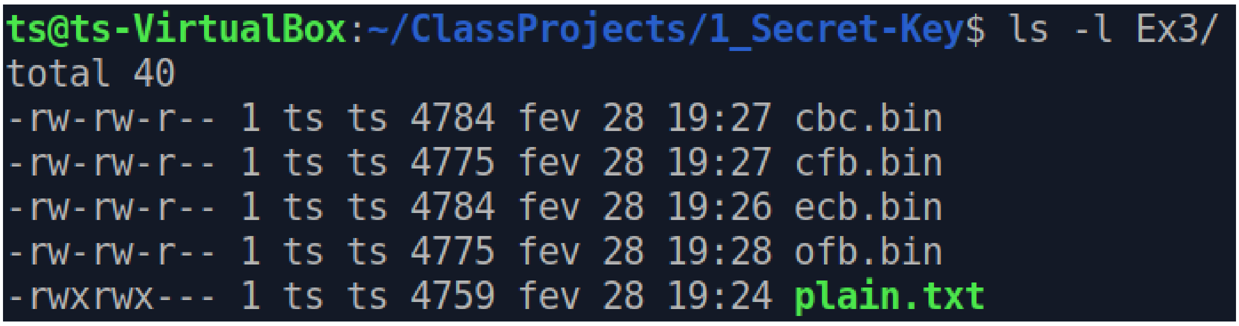
Task 2: Padding

a)

In order to detect which of the modes of encryption have padding we decided to check the size of the encryption files, assuming that the files with padding would have a bigger size than the one without
For this we just encrypted the file plain.txt with the different modes of AES

- ECB
- CBC
- CFB
- OFB

  and listed then in their directory with the size of the side



```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key$ ls -l Ex3/
total 40
-rw-rw-r-- 1 ts ts 4784 fev 28 19:27 cbc.bin
-rw-rw-r-- 1 ts ts 4775 fev 28 19:27 cfb.bin
-rw-rw-r-- 1 ts ts 4784 fev 28 19:26 ecb.bin
-rw-rw-r-- 1 ts ts 4775 fev 28 19:28 ofb.bin
-rwxrwx--- 1 ts ts 4759 fev 28 19:24 plain.txt
```

As we can see the ones with padding are the ones with a bigger file size, therefore CBC and ECB, this is confirmed by the fact that when researching we confirmed that these were the modes which did make use of padding

b)

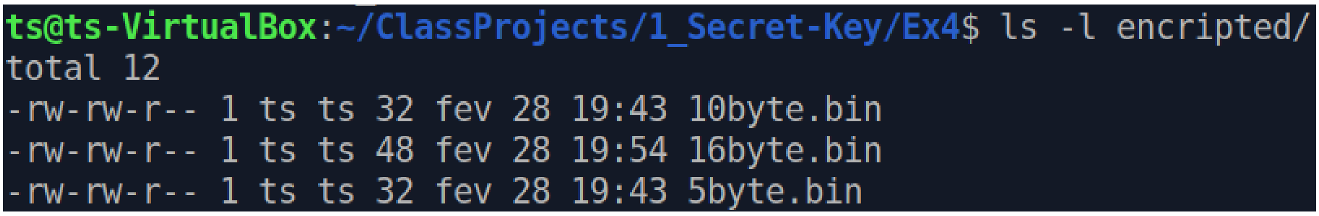
We start by creating the files as indicated in the exercise, and then we proceed to encrypting them with the command line

```
openssl enc -aes-128-cbc -e -in Ex4/input_file.txt -out Ex4/encripted.bin
```

and then decrypting them with the command

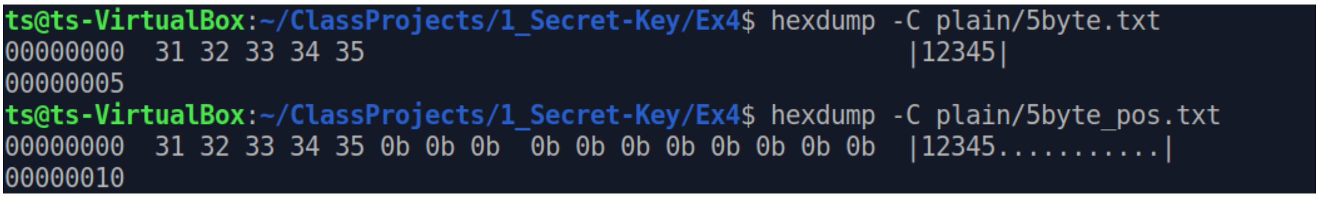
```
openssl enc -aes-128-cbc -nopad -d -in Ex4/encripted.bin -out
Ex4/output_file.txt
```

Now we can compare the size of the original file with the encrypted ones



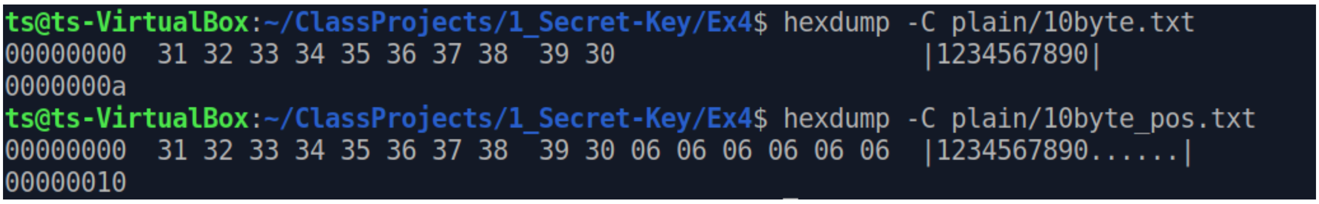
```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/Ex4$ ls -l encripted/
total 12
-rw-rw-r-- 1 ts ts 32 fev 28 19:43 10byte.bin
-rw-rw-r-- 1 ts ts 48 fev 28 19:54 16byte.bin
-rw-rw-r-- 1 ts ts 32 fev 28 19:43 5byte.bin
```

and as we can see all of them increased in size. This is expected considering we are using CBC, which adds padding.
When using now the command to decrypt with the -nopad flag, we can see the padding added to the file when it was encrypted.
First will be the original, and bellow the padded decryption
5 bytes:



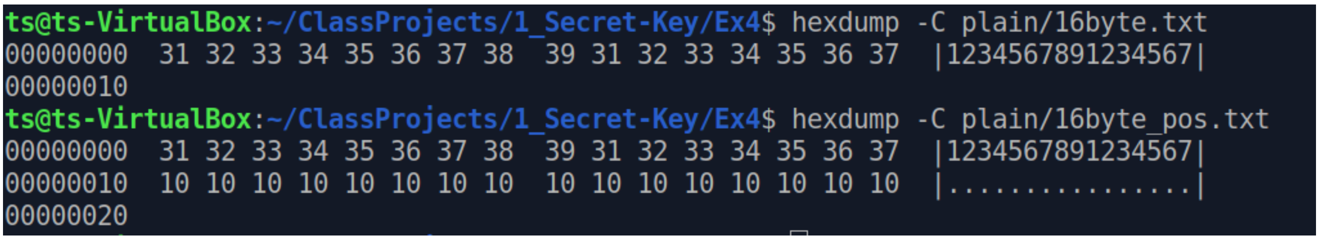
```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/Ex4$ hexdump -C plain/5byte.txt
00000000  31 32 33 34 35                                    |12345|
00000005
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/Ex4$ hexdump -C plain/5byte_pos.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
```

10 bytes:



```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/Ex4$ hexdump -C plain/10byte.txt
00000000  31 32 33 34 35 36 37 38  39 30                    |1234567890|
0000000a
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/Ex4$ hexdump -C plain/10byte_pos.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
```

16 bytes:



```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/Ex4$ hexdump -C plain/16byte.txt
00000000  31 32 33 34 35 36 37 38  39 31 32 33 34 35 36 37  |1234567891234567|
00000010
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/Ex4$ hexdump -C plain/16byte_pos.txt
00000000  31 32 33 34 35 36 37 38  39 31 32 33 34 35 36 37  |1234567891234567|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
```

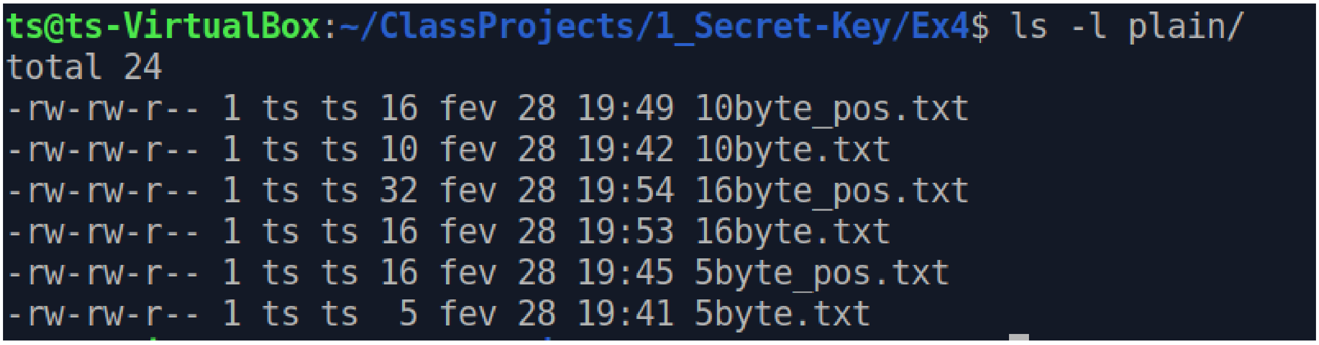
As we can see in all of the cases padding was added to the original post encryption, but more than that we can see that the padding was added to keep the total number of bytes even, the first 2 cases we will have 2 bytes and in the last one 4 bytes, this because AES needs and even number of bytes to be able to do the swaps.

In the other hand, we can also see that the total number of added padding bytes can be seen in the value of the bytes used for padding.

The first file uses 11 bytes of padding, therefore the value of the padding bytes is `0b` or 11 in hex

The second file uses 6 bytes of padding, therefore the value of the padding bytes is `06` or 6 in hex

The third file uses 16 bytes of padding, therefore the value of the padding bytes is `10` or 16 in hex

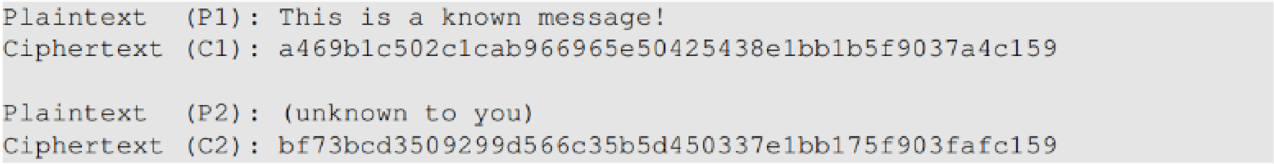


```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/Ex4$ ls -l plain/
total 24
-rw-rw-r-- 1 ts ts 16 fev 28 19:49 10byte_pos.txt
-rw-rw-r-- 1 ts ts 10 fev 28 19:42 10byte.txt
-rw-rw-r-- 1 ts ts 32 fev 28 19:54 16byte_pos.txt
-rw-rw-r-- 1 ts ts 16 fev 28 19:53 16byte.txt
-rw-rw-r-- 1 ts ts 16 fev 28 19:45 5byte_pos.txt
-rw-rw-r-- 1 ts ts  5 fev 28 19:41 5byte.txt
```

These are the sizes of the files before and after the encryption and decryption (*_pos.txt is the file after)

Task 3: Initial Vector (IV) and Common Mistakes

When trying to solve this exercise we decided to alter the given python script to solve the problem.



```
Plaintext  (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

Plaintext  (P2): (unknown to you)
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```

The process of getting `C1` from `P1` goes as follows

- Get IV (Initialisation Vector)
- Pass IV through block cipher
- XOR block cipher result and plain text
- Obtain cipher text
  Considering that we know that for both $C1$ and $C2$ the same IV was used, and that

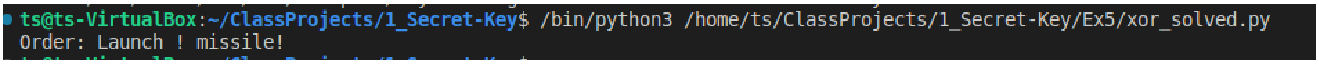
  $$(x\ XOR\ b)\ XOR\ b = x$$

  We can conclude that the best way of discovering $P2$ will be to discover the IV , from $XOR'ing\ P1$ and $C1$, and to use it to XOR $C2$ giving $P2$
  The script we can up with was

```
#!/usr/bin/python3
def xor(first, second):

    return bytearray(x^y for x,y in zip(first, second))


P1 = bytes("This is a known message!", 'utf-8')
C1 = bytearray.fromhex("a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159")
C2 = bytearray.fromhex("bf73bcd3509299d566c35b5d450377e1bb175f903fafc159")
```

```
iv = xor(C1,P1)


print(xor(iv,C2).decode())
```

When running it we got the `C2 = Order: Launch ! missile!`



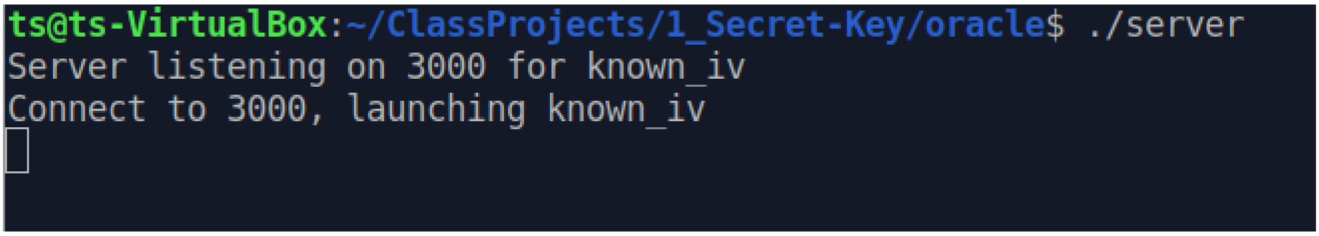
```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key$ /bin/python3 /home/ts/ClassProjects/1_Secret-Key/Ex5/xor_solved.py
Order: Launch ! missile!
```

b)

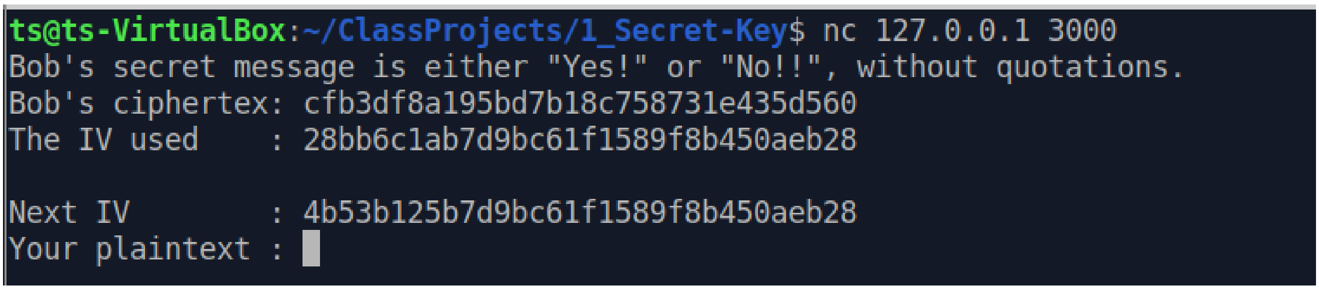
We can get some of the information in the begging of $P2$, but eventually because of the ciphers using the previous ciphers result as input for their own block cipher, we will stop getting coherent information.

Chosen-Plain-text Attack: use a predictable IV

As indicated in the exercise we started by initialling both the server and the client



```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key/oracle$ ./server
Server listening on 3000 for known_iv
Connect to 3000, launching known_iv
```



```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key$ nc 127.0.0.1 3000
Bob's secret message is either "Yes!" or "No!!", without quotations.
Bob's ciphertex: cfb3df8a195bd7b18c758731e435d560
The IV used     : 28bb6c1ab7d9bc61f1589f8b450aeb28

Next IV         : 4b53b125b7d9bc61f1589f8b450aeb28
Your plaintext : 
```

In this problem, while the IV isn't always the same we can predict the next iv to be used by the server.
We also know that Bob's secret will be either Yes! or No!!.
For this exact reason, our plan of attack will be to see of the cipher text given by the server when inputted either Yes! or No!!, is equal to Bob's cipher-text.
For this we will again use the properties of the XOR function, and AES in CBC mode.

$$(a\ XOR\ IV_1)\ XOR\ IV_1 = a$$

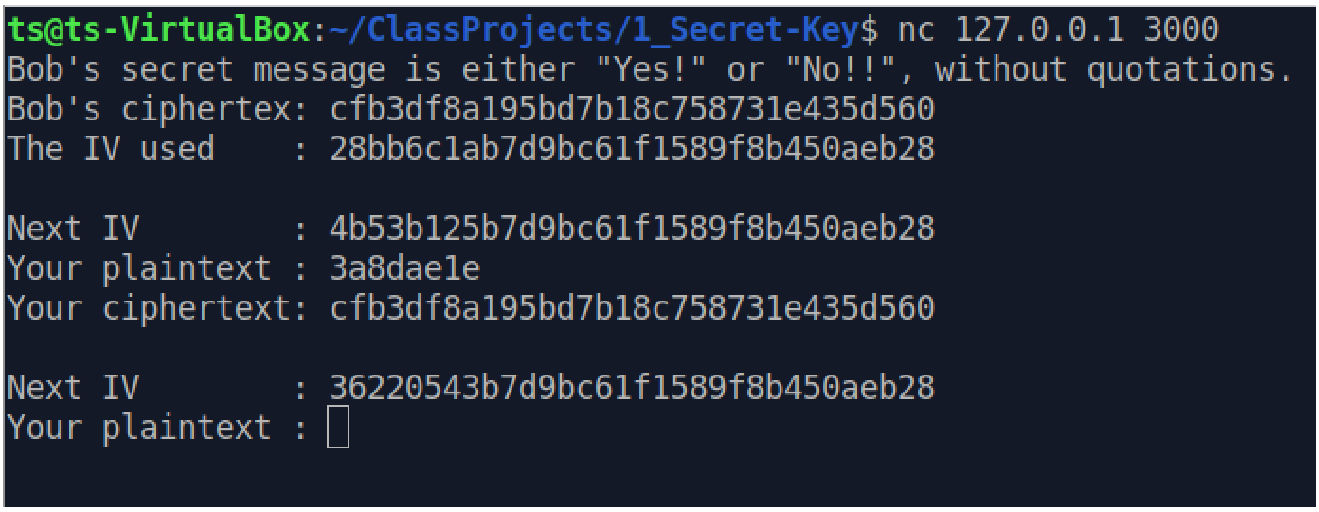
We will therefore, XOR or message ("Yes!" or "No!!"), with the next IV so that when the server XOR's it again with next IV it undo's our action.
We will then XOR it with the previous IV so that we can check the output cipher-text against Bob's original cipher-text
For this purpose we decided to alter the python script again so that it gave us the value of:

$$(\text{"Yes!" or "No!!"})\ XOR\ IV_{Bob}\ XOR\ IV_{New}$$

The script is bellow

```
#!/usr/bin/python3
# XOR two bytearrays
def xor(first, second):

    return bytearray(x^y for x,y in zip(first, second))

BMessage = bytes("Yes!", 'utf-8')
BIV = bytearray.fromhex("28bb6c1ab7d9bc61f1589f8b450aeb28")
NIV = bytearray.fromhex("4b53b125b7d9bc61f1589f8b450aeb28")

r1 = xor(NIV,BIV)
r2 = xor(r1,BMessage)
print(r2.hex())
```

With this we conclude that Bob's secret message was in fact "Yes!"



```
ts@ts-VirtualBox:~/ClassProjects/1_Secret-Key$ nc 127.0.0.1 3000
Bob's secret message is either "Yes!" or "No!!", without quotations.
Bob's ciphertex: cfb3df8a195bd7b18c758731e435d560
The IV used    : 28bb6c1ab7d9bc61f1589f8b450aeb28

Next IV        : 4b53b125b7d9bc61f1589f8b450aeb28
Your plaintext : 3a8dae1e
Your ciphertext: cfb3df8a195bd7b18c758731e435d560

Next IV        : 36220543b7d9bc61f1589f8b450aeb28
Your plaintext : 
```