Cryptography - Lab 3 - Block Cipher Operations

Javier Antxon Garrues Apecechea

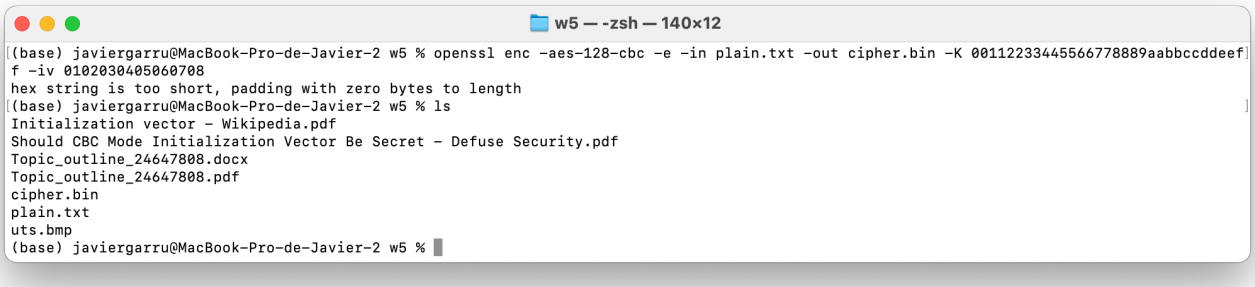
Student ID: 24647808

In this lab we will get familiar with block cipher operations.

Part One: encrypting with different ciphers and modes

First, we will encribt using aes-128 and block mode cbc:

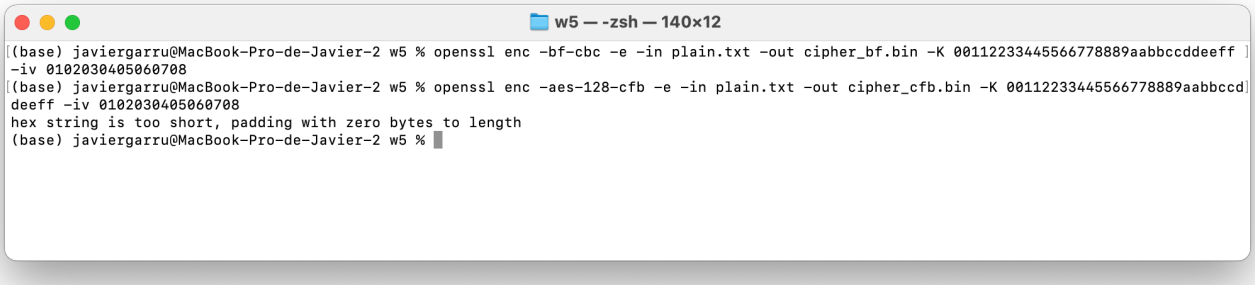
```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher.bin -K
00112233445566778899aabbccddeeff -iv 0102030405060708
```



Then, using 2 more: bf with abc mode and aes with cfb mode.

```
openssl enc -bf-cbc -e -in plain.txt -out cipher_bf.bin -K
00112233445566778899aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cfb -e -in plain.txt -out cipher_cfb.bin -K
00112233445566778899aabbccddeeff -iv 0102030405060708
```

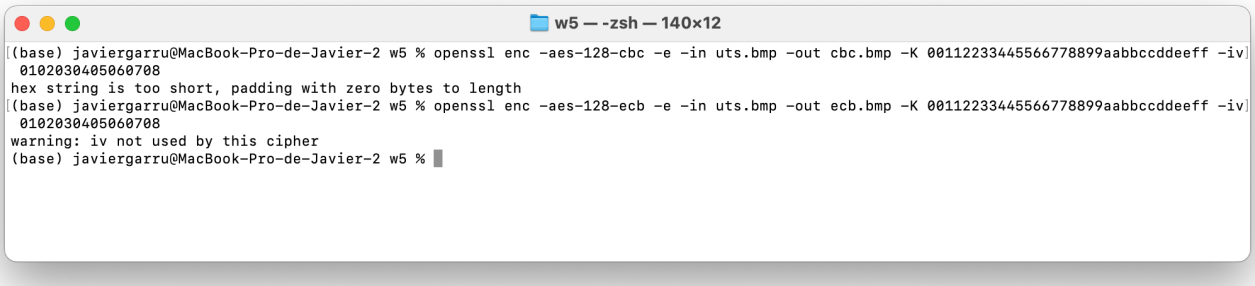


Part Two: ECB vs CBC

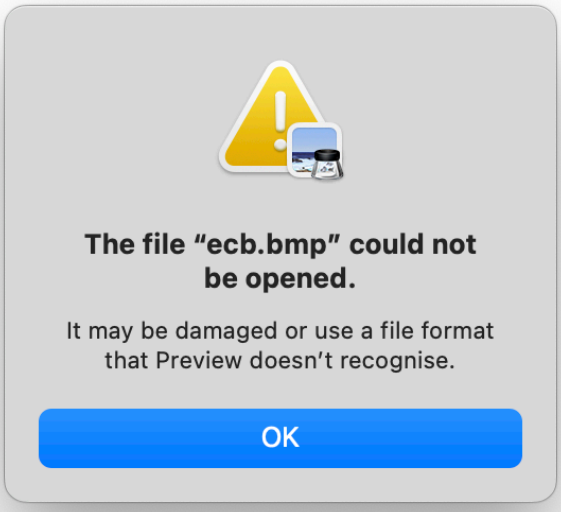
We are going to encrypt the picture **uts.bmp** with the two encryption modes and compare the results:

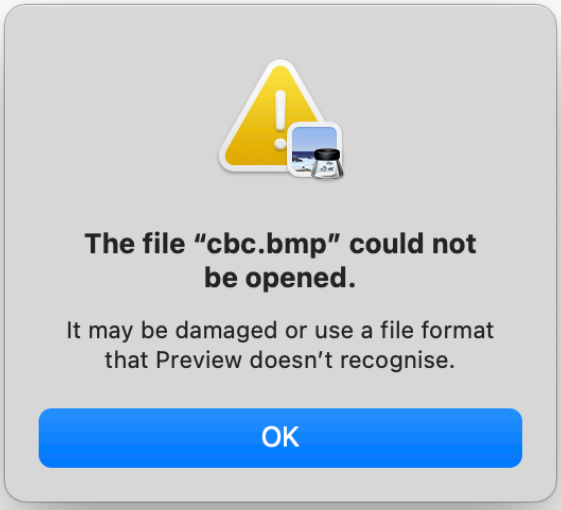
```
openssl enc -aes-128-ecb -e -in uts.bmp -out cbc.bmp -K
00112233445566778899aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cbc -e -in uts.bmp -out ecb.bmp -K
00112233445566778899aabbccddeeff -iv 0102030405060708
```

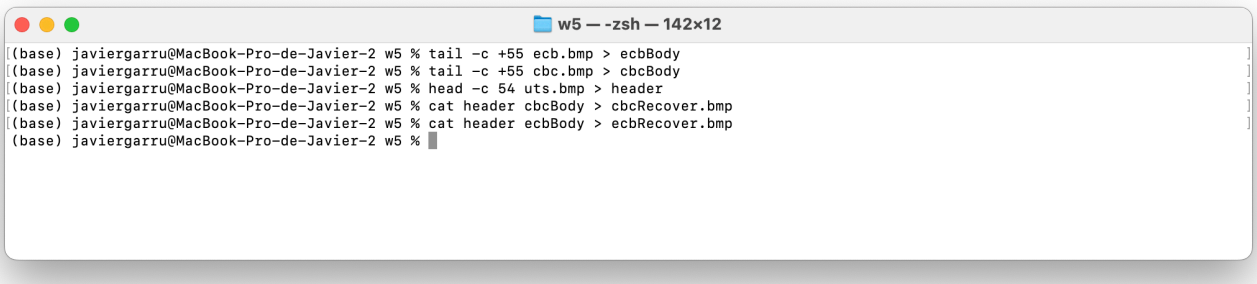


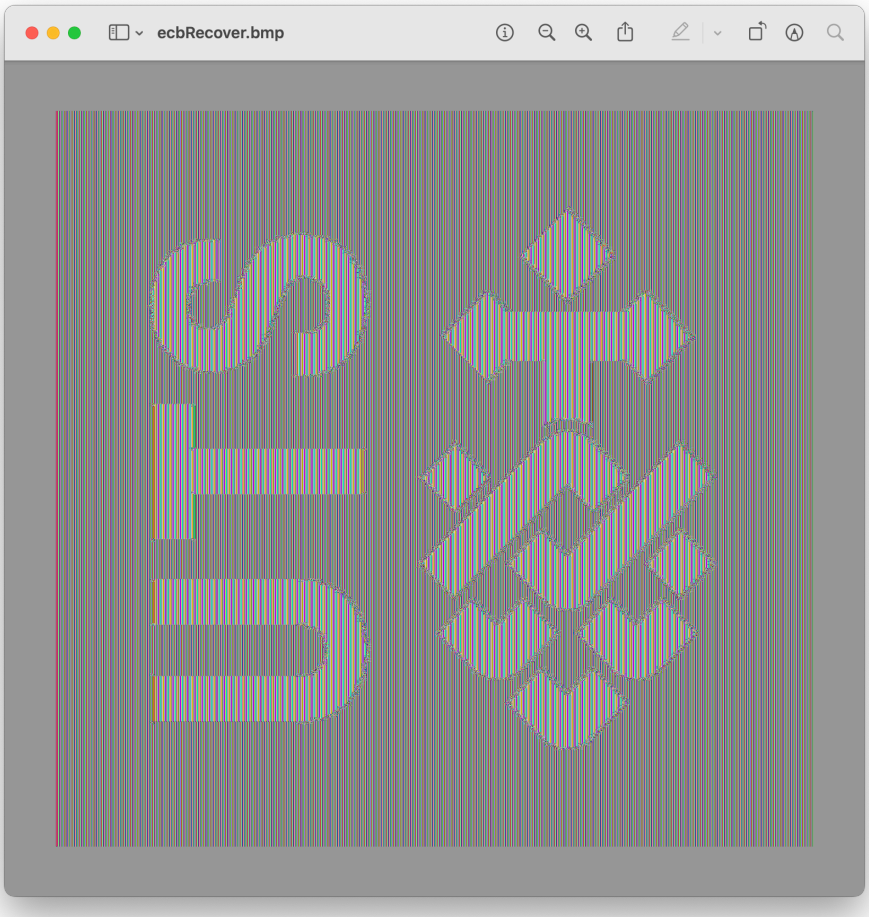
When trying to visualize the images we are shown these warnings:

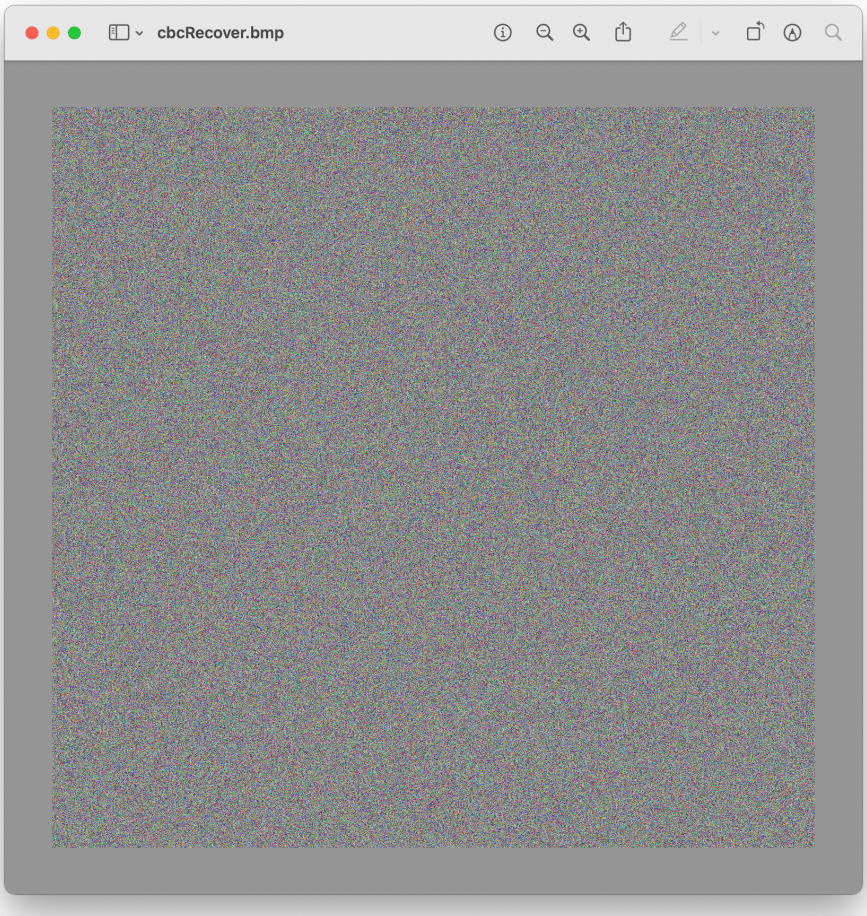




We take the header from the original file and create two new files combining this header with the bodies of each of the encrypted images:




ecbRecover.bmp


cbcRecover.bmp

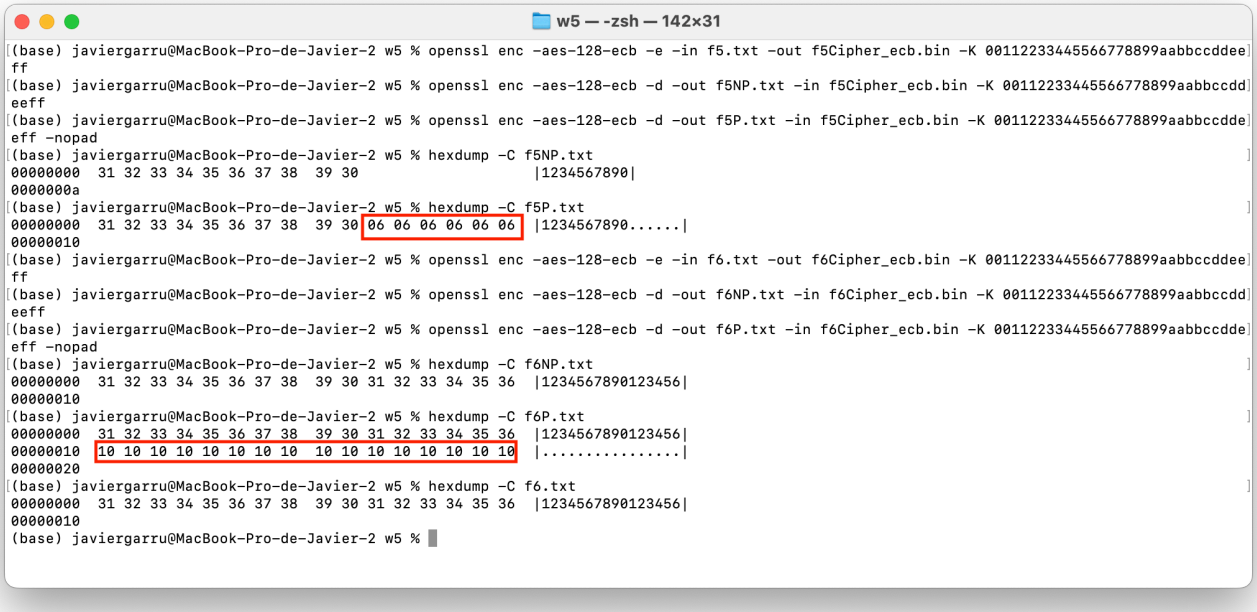
Question 1: Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

The two encrypted pictures (one using ecb and the other using cbc) are significantly different. While in the one encrypted with cbc is impossible to visually find any information of the original picture, in the one encrypted with ecb the original image general shape is clearly visible. This poses a big problem as an attacker could have access to the encrypted images and try different headers until obtaining the first of the two images from above and have a pretty good idea of what the original image was. The reason behind this phenomenon is the way ecb mode works. This mode only encrypts each block using the encryption algorithm but doesn't consider the previous blocks to encrypt the new block causing a lack of randomness. This is the reason why in the lecture was mentioned that the ecb mode wasn't recommended in most cases.

Part Three: Padding

When the lenght of the plaintext is not a multiple of the block size some block ciphers add padding to fill complete blocks. To see how the different modes do this we will encrypt and decrypt two text files: one that has **16 bytes** and another one that has **10 bytes** and will therefore need padding. When decrypting we will indicate the option **-nopad** so the padding isn't automatically eliminated after the decryption. This will allow us to see the padding of each of the modes in plaintext.

ECB:

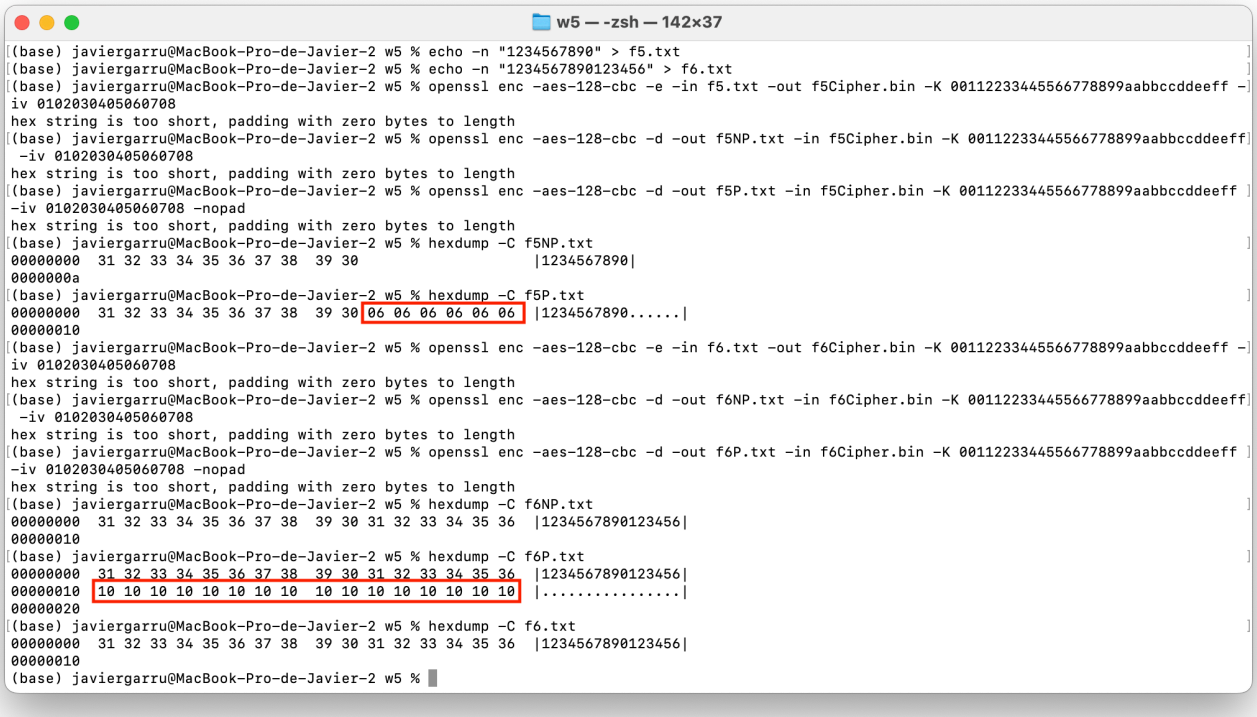


ECB does use padding as we can see in the above screenshot after encrypting the **10-byte file f5.txt** . The included padding values are **06** in hexadecimal due to being 6 bytes the total number of padding bytes needed.

When decrypting the **16-byte file f6.txt** created, we also see 16 bytes of padding (10 in hex). This is quite interesting as a 16-bytes files has a length multiple of the block size and, in theory, shouldn't need padding. The reason behind using padding even though the size of file was multiple of the block size is that it is considered good practice to always use padding. This helps ensure data integrity, prevents certain attacks,

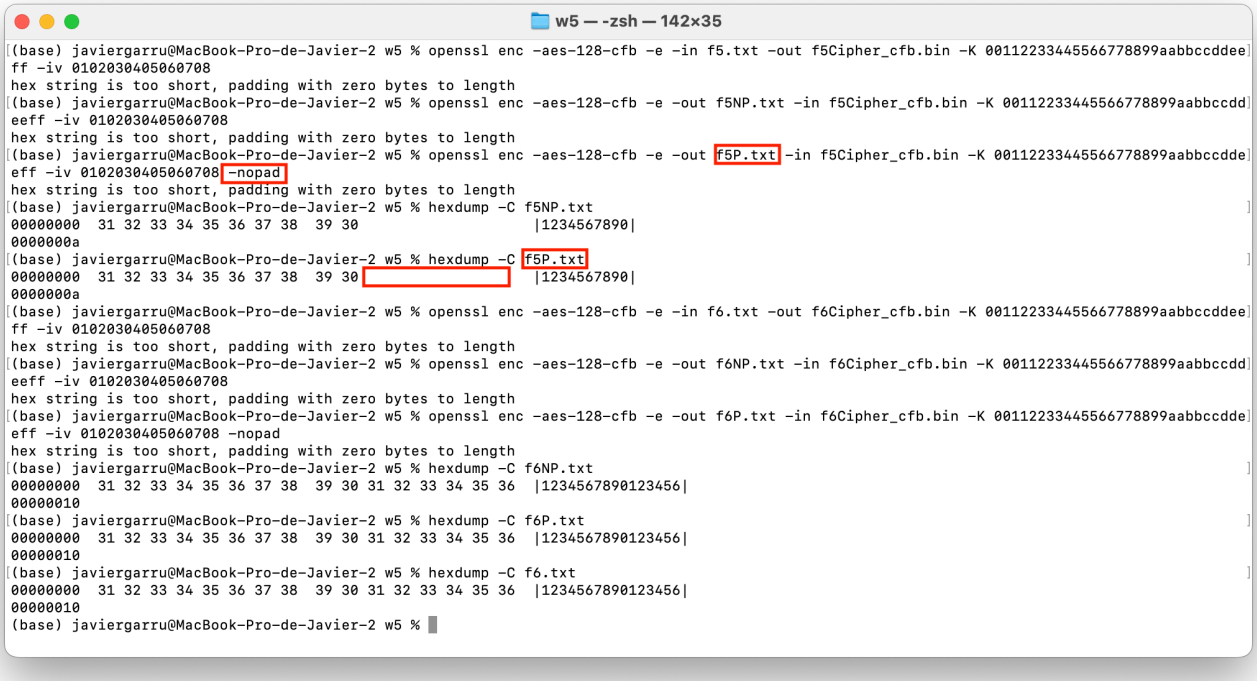
and can make it easier to work with different types of data.

CBC:



Again now padding is used to complete the encrypted file and it follows the same pattern as before using 06 as the padding value when filling 6 empty bytes.

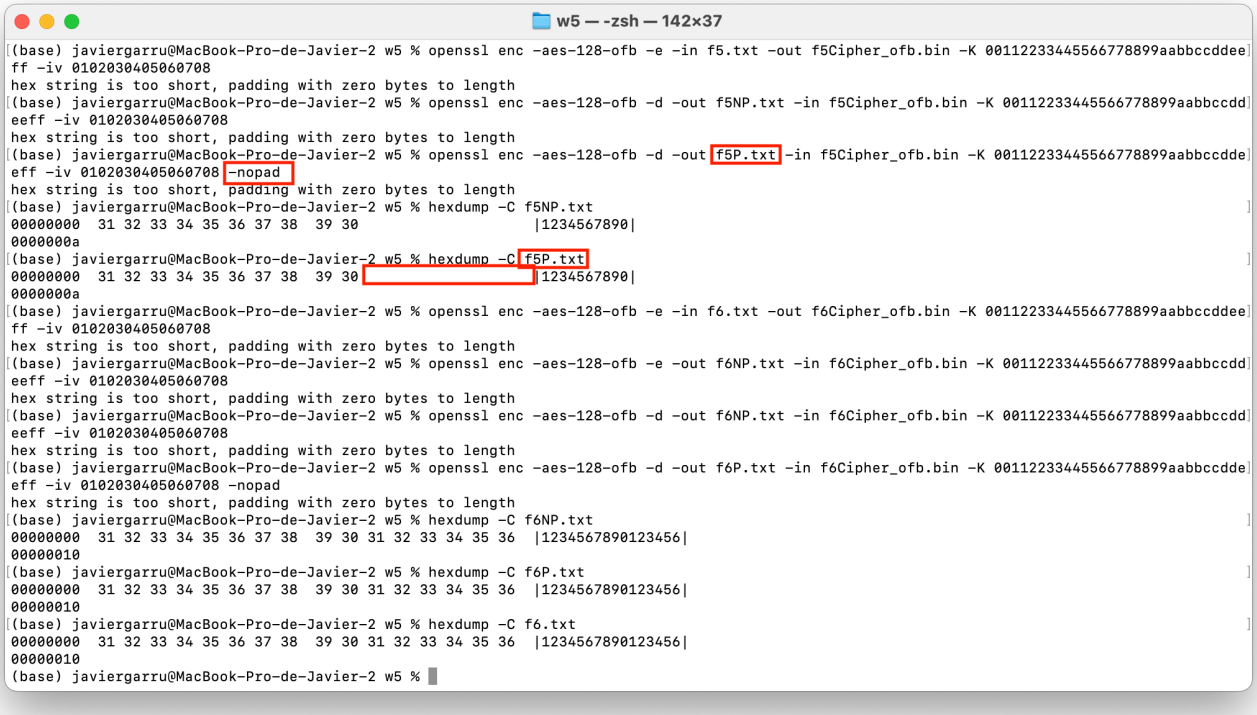
CFB:



Now, padding isn't used in this mode as we can see in the screenshot. CFB encrypts individual bytes instead of encrypting full blocks and our file has an exact number of bytes and therefore doesn't need padding. It behaves as a stream cipher.

CFB Encrypts the IV using the encryption algorithm (aes-128 in this case), then it selects the first s-bits of the result (in this case s=8) and at last it XORs those 8 bits with the plain text's bytes. That way the encryption is done byte by byte and there is no need of padding even though the lenght of the plaintext was 10 bytes.

OFB:



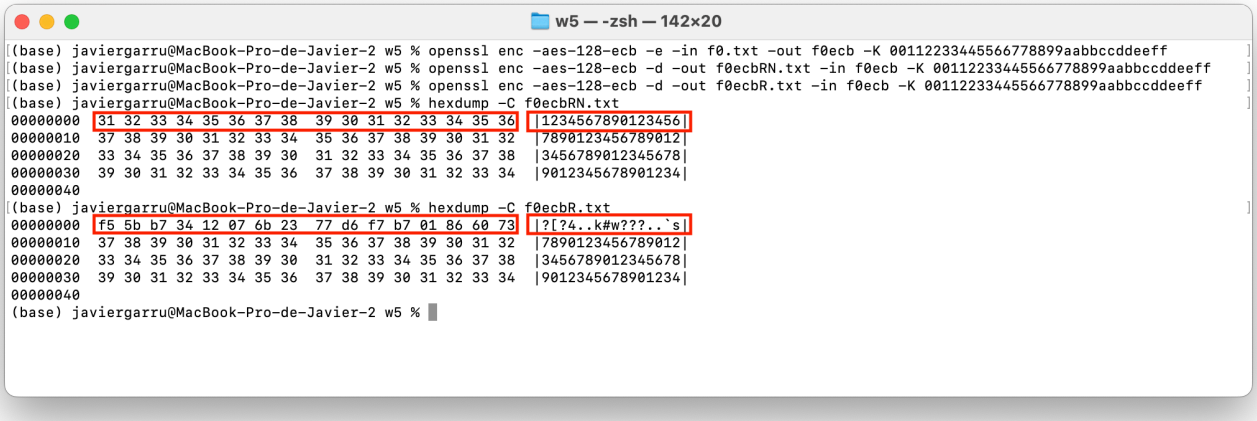
OFB doesn't use padding either as it works similarly as how CFB works. In this case, a none is encrypted and then is XORed with the plain text. Due to XOR operations not having size restrictions, the XORed plaintext can be taken of size 1-byte leading to the final encrypted file not having any padding. It behaves as a stream cipher.

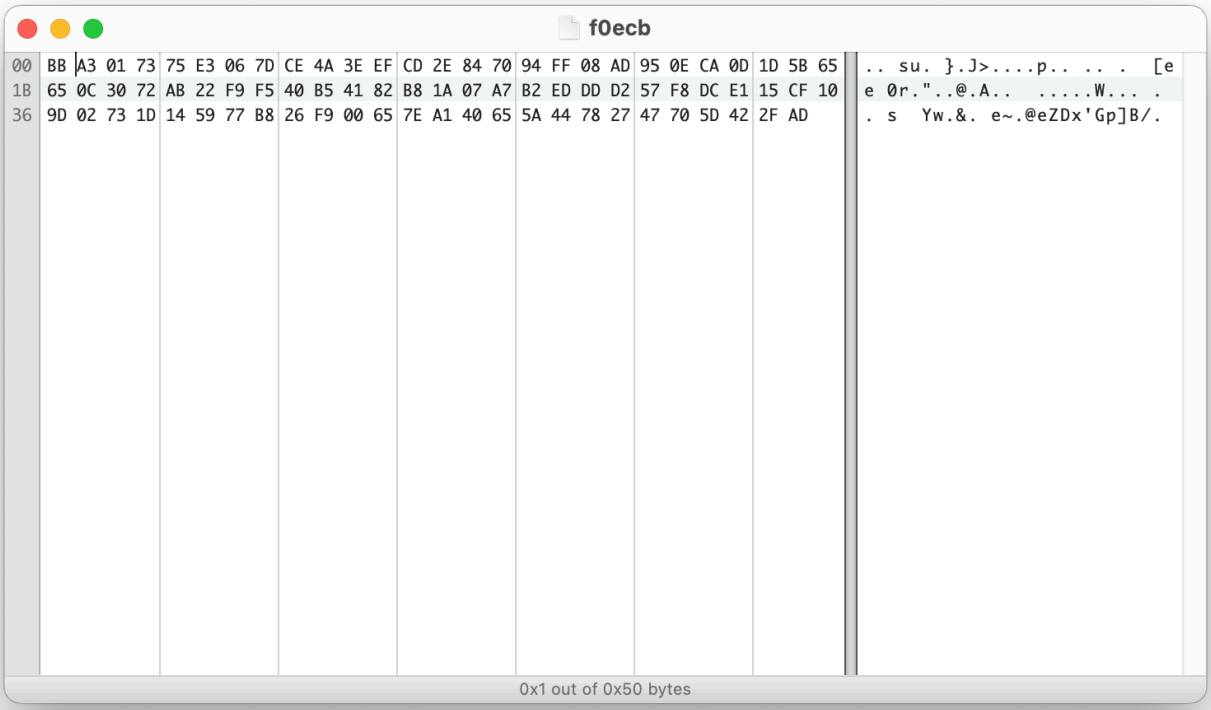
Part Four: Error Propagation - Corrupted Cipher Text

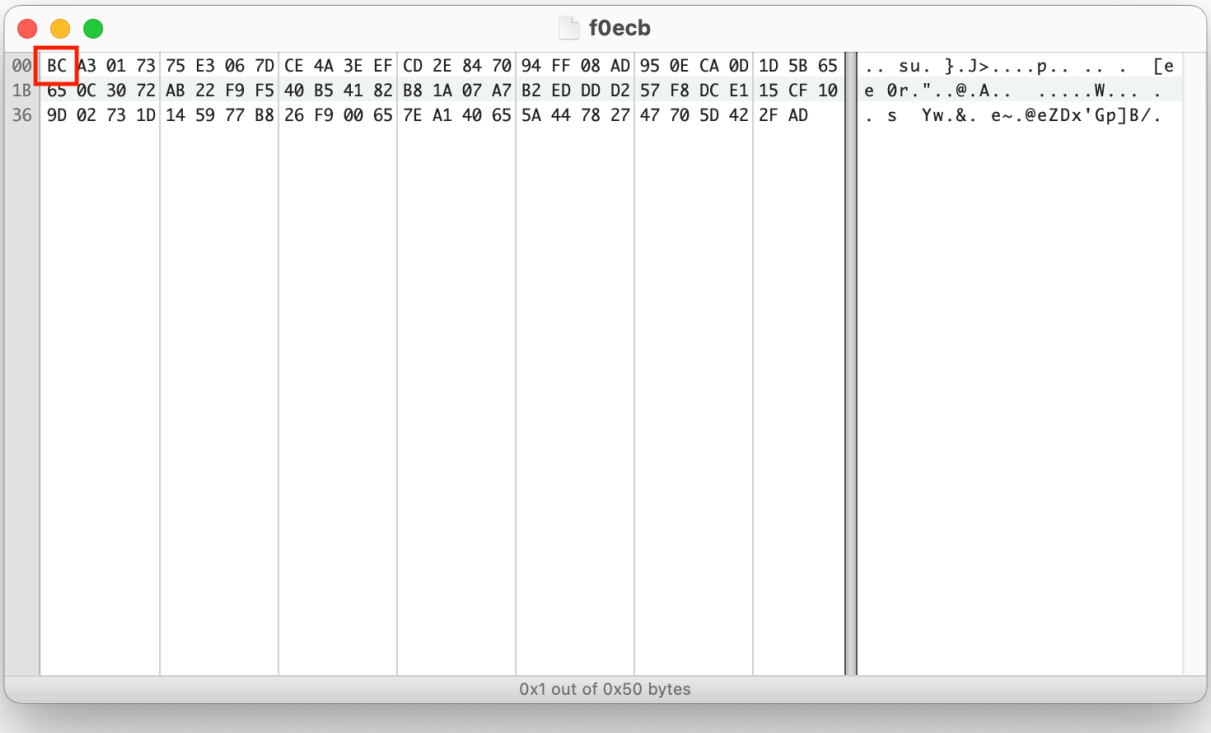
In this task, we will create a 64 byte text file, encrypted with different modes, modify a byte in the resulting encrypted file, decrypt the unmodified encrypted file and the modified one and compare the decrypted files.

Question: How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, OFB, and CTR respectively? Please find it out after you finish this task and providejustification.

ECB:

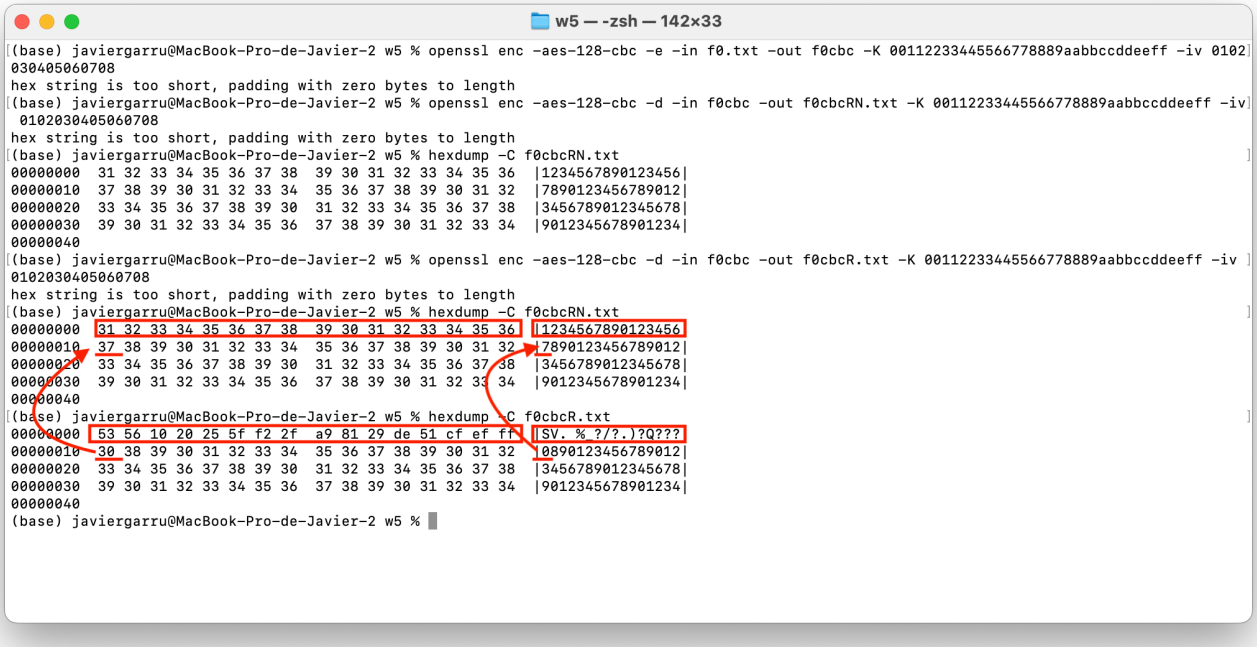






When working with ECB a modification of one bit in the initial byte of the cipher text generates random outcome when decrypted and affects the whole block. This is due to ECB working block by block therefore decrypting the whole encrypted block at once added to the avalanche effect implemented in the encryption algorithm (aes-128). As we can see in the output, the first 16 bytes of the decrypted filed are corrupted but the rest bytes are left unchanged.

CBC:

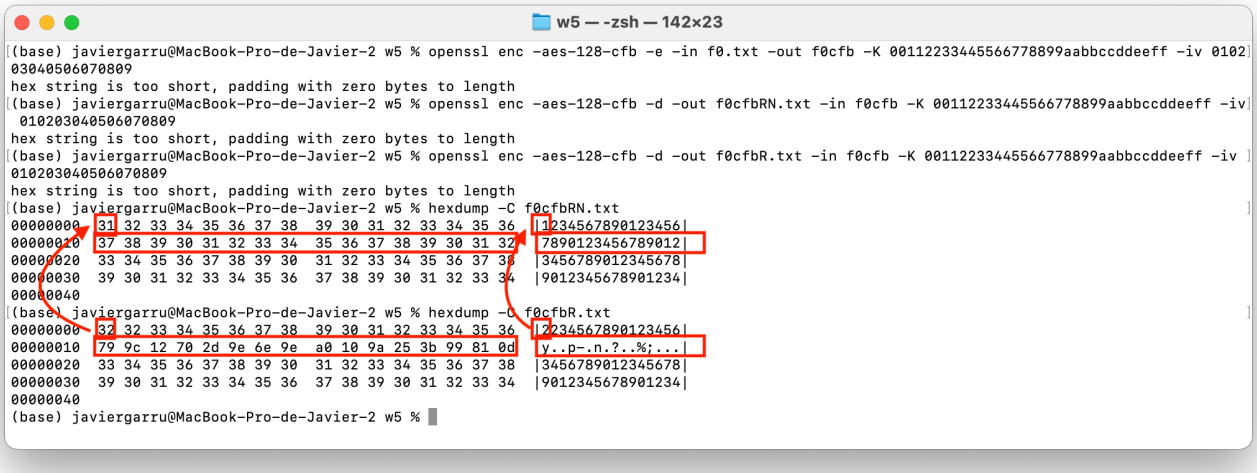


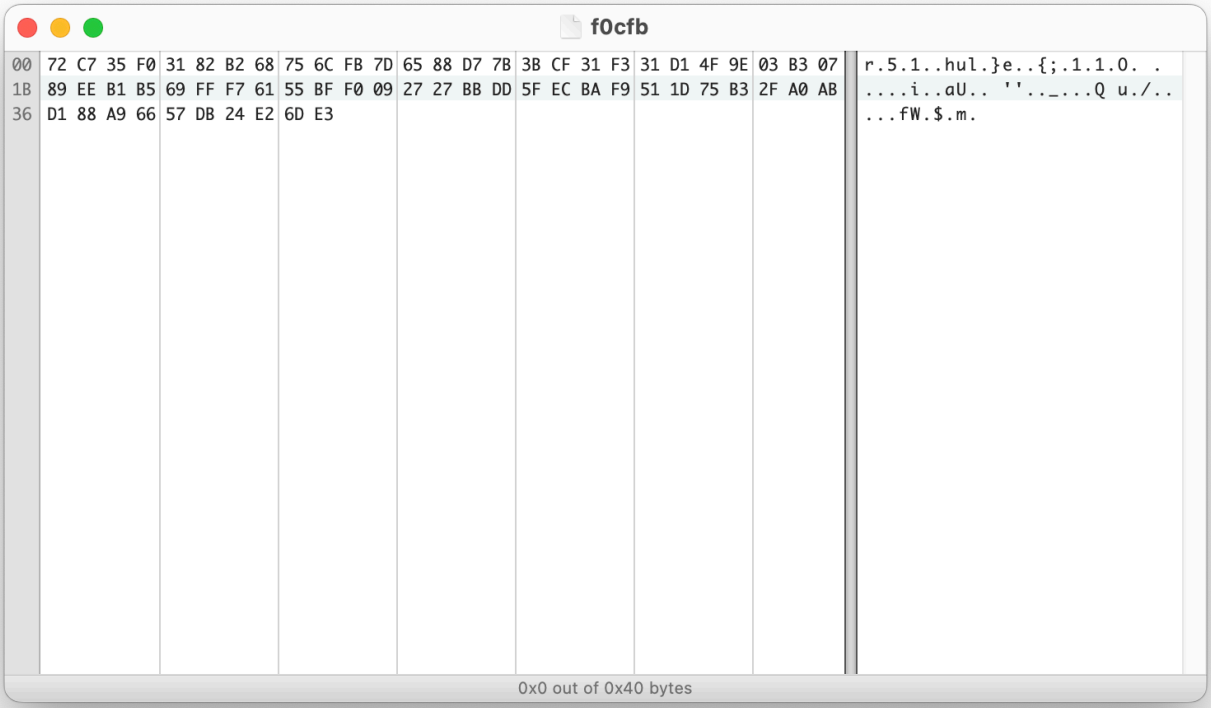
Now, the modification of 1 bit in the initial byte of the cipher text causes the deciphered text to have the first 17 bytes corrupted. This occurs because the encryption algorithm is AES-128 (16-bytes) and when decrypting a whole block is decrypted using the encryption algorithm (resulting in a fully corrupted block) and the same cipher text block is then xored with the decryption of the second block. This causes the

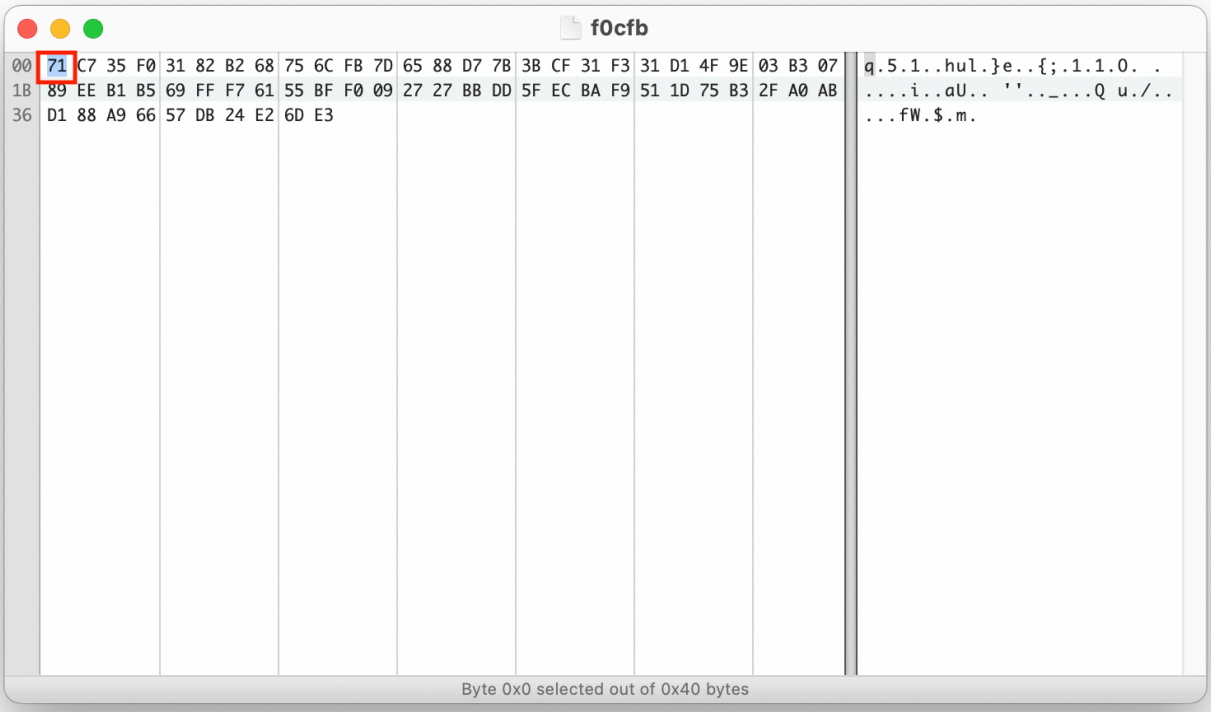
second block of deciphered text to be in error exactly in the same positions as the corrupted cipher text block was.

That is why the first value of the second row of the decrypted text visualization is 0 instead of 7 in ASCII codes or 30 and 37 in hexadecimal values. The rest of the second block is correctly deciphered as the X operation works bit by bit.

CFB:



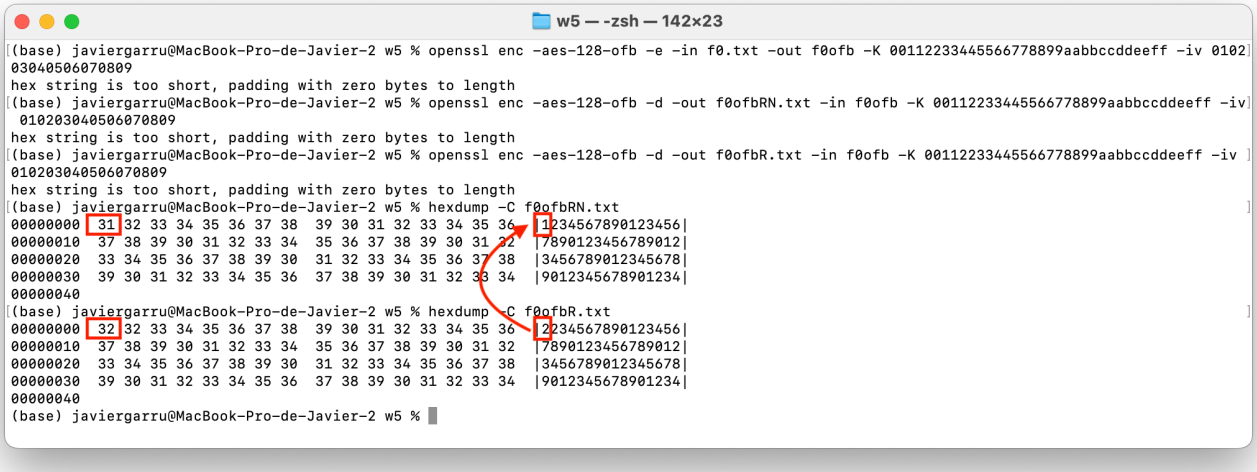


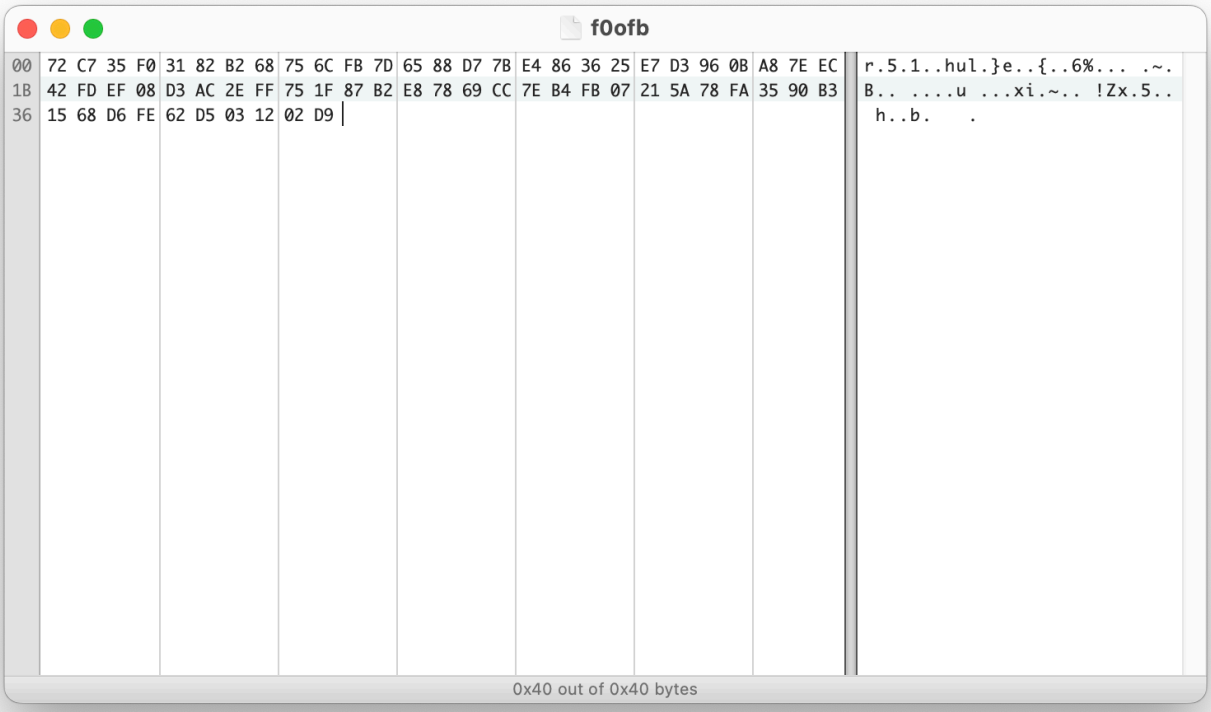


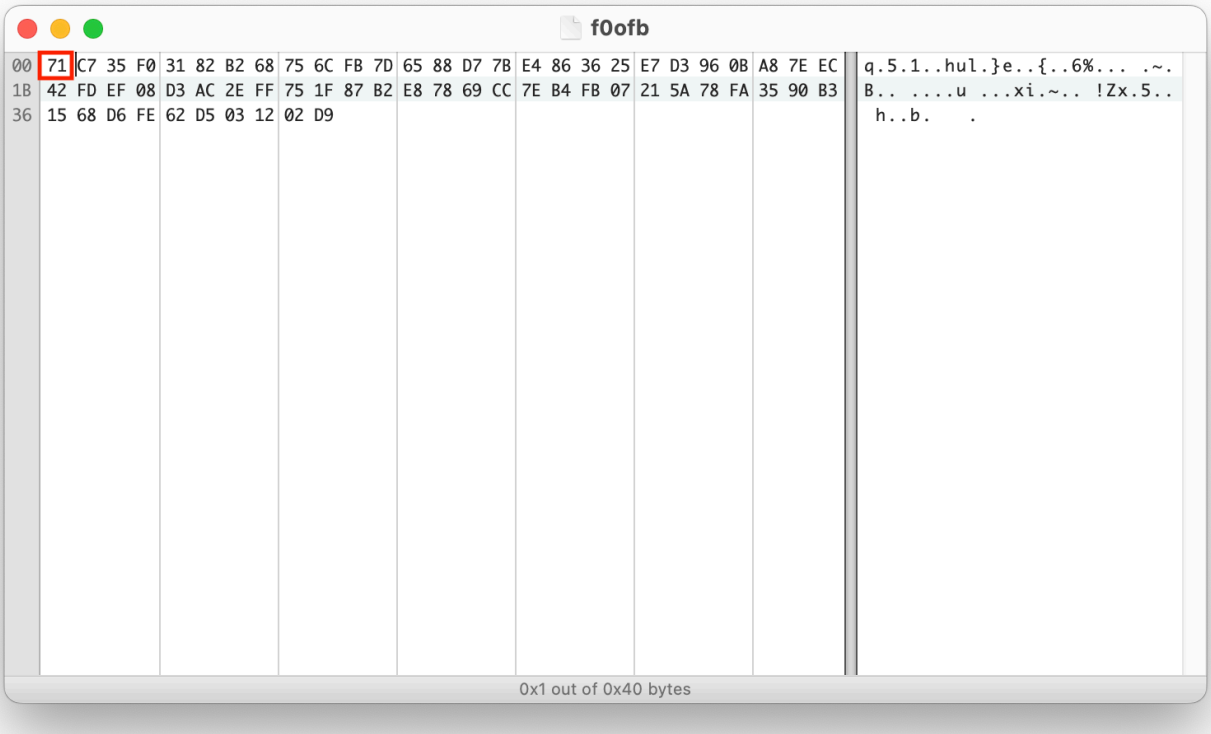
In the case of CFB mode the corrupted bit in the ciphertext makes the correspondent positions of the same block to be in error when decrypted but causes the whole following bloc to be random. This is due to the ciphertext only interacting with the selected s-bits obtained after the encryption of the IV through an XOR operation. This allows the errors on the first block to be in the same positions. On the other hand, the first ciphertext block is, after shifting, encripted and the selected s-bits are extracted from the output. The avalanche effect makes one bit change in the ciphertext to cause the output to be random; in particular the selected s-bits.

That is why changing one bit in the first byte causes the first value of the first block to go from 31 to 32 in hexadecimal but the second block to become random after decryption.

OFB:





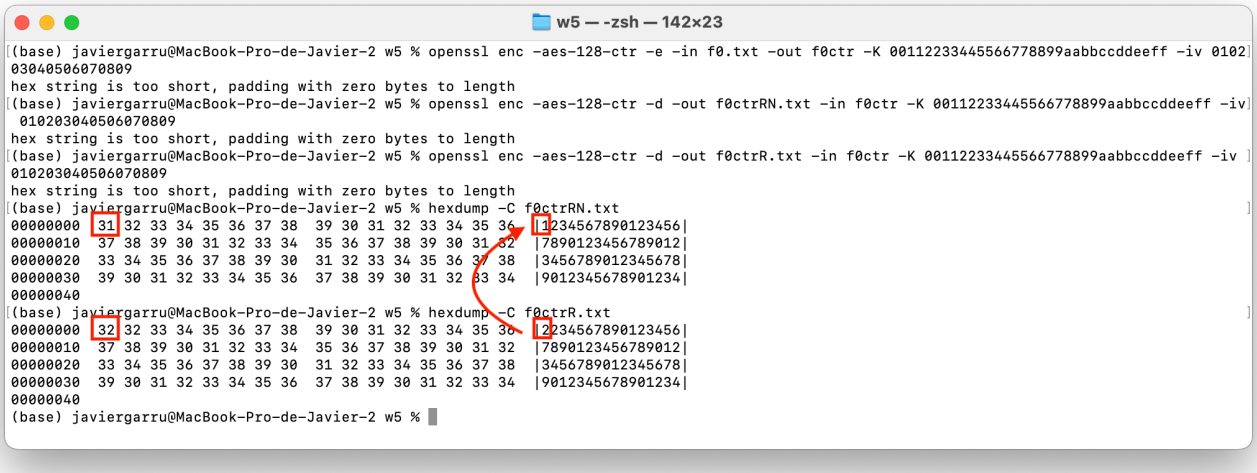


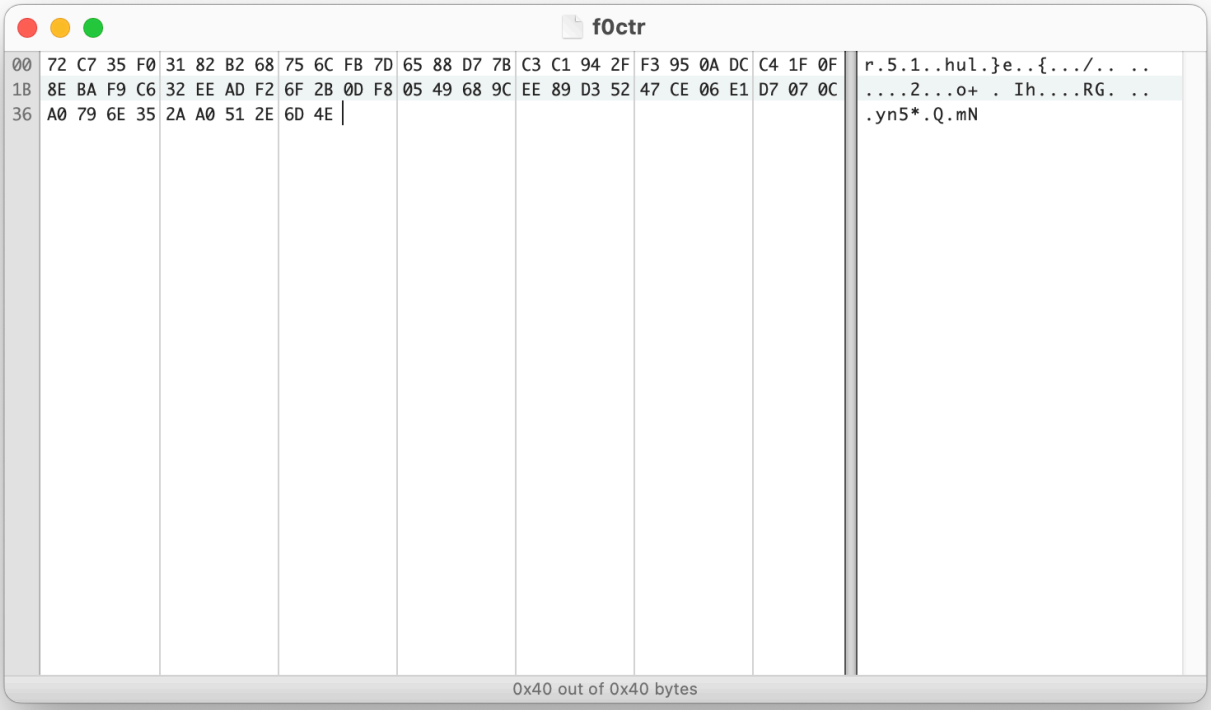
The result in here was expected as this block cipher mode behaves as a stream cipher where the IV is encrypted and then XORed with the plaintext. This means that in decryption the same process occurs but now the ciphertext is the one being XORed. When there is a corrupt bit in the ciphertext the XOR function transmits that corruption to the plaintext but, as it only works bit by bit, it doesn't allow it to expand as de

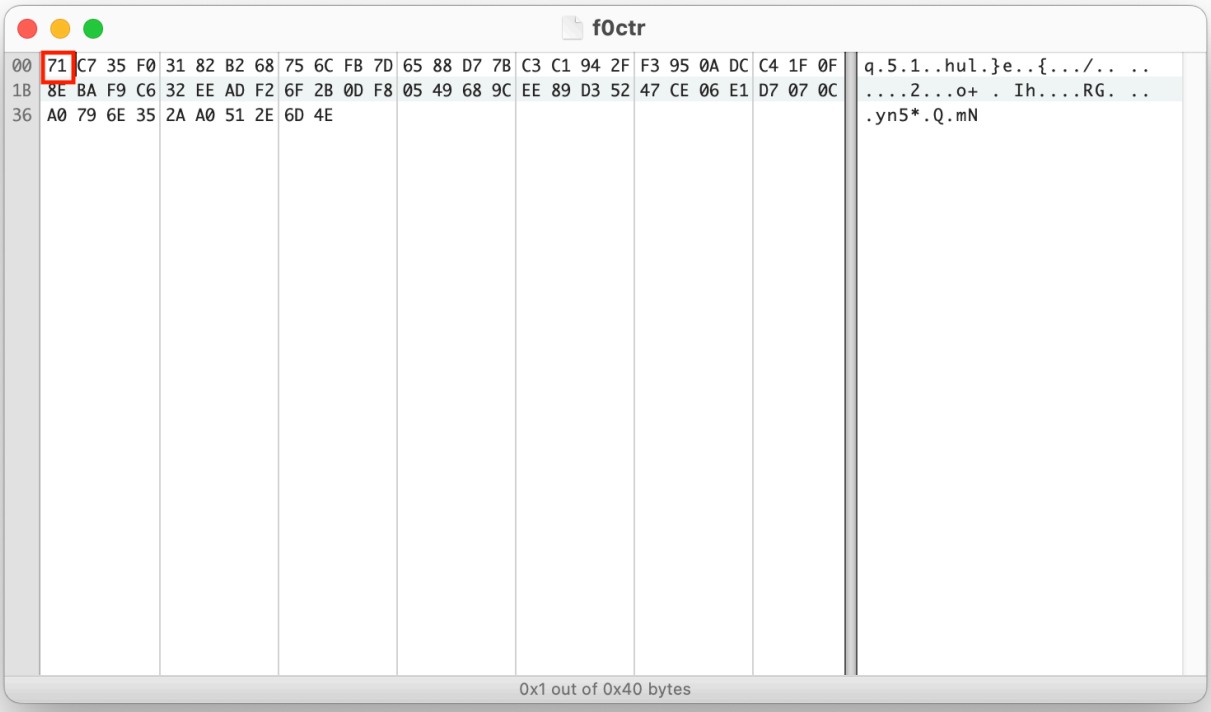
encryption function would do due to the avalanche effect.

That is why only the first byte of the decrypted corrupted ciphertext varies with respect to the original plaintext from 1 to 2 (31 to 32 in hexadecimal values).

CTR:





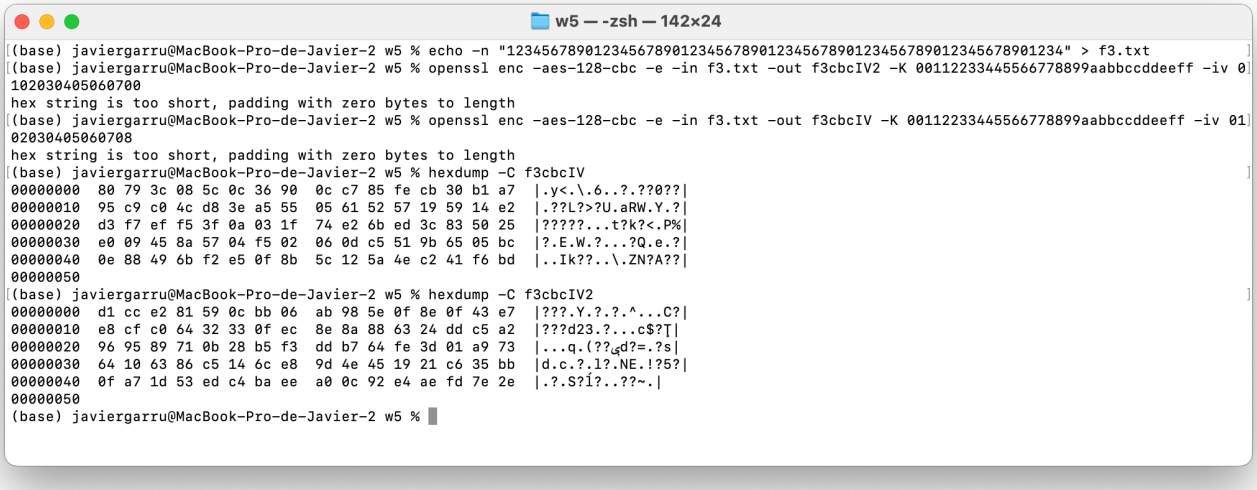


The same that occurred with the last block cipher mode occurs here. XORing the corrupted ciphertext with the encrypted counter makes the corrupted bit to corrupt only an individual bit of the output plaintext making the byte of ASCII code associated 1 to become 2 (31 becomes 32 in hex). No decryption algorithm is applied on the ciphertext and therefore the avalanche effect is avoided.

Part Five: Initial Vector (IV)

To conclude the lab we will encrypt the same file with the same encryption mode and key **but** different IV:

```
#IV 2: 0102030405060700        -> f3cbcIV2
#IV 1: 0102030405060708        -> f3cbcIV
```



As it is clear from the hexdump execution the change of a single byte in the IV completely changes the encrypted text obtained. This is great as it introduces an extra level of security when encrypting. The **reason why** it should be **unique** and not to be repeated is the fact that the absence of uniqueness would allow attackers to find pairs of blocks plaintext-ciphertext from the common parts of the messages. For instance, some parts of the header of a specific protocol packet will always be the same and if loads of them are sent without encrypting with a random unique component (IV), the attacker would be able to infer information from the key and end up finding it.

Question: What properties the IV should have? Why?

The first and most important property an IV should have is **uniqueness**. Uniqueness ensures that the same plaintext is never encrypted in the same way (any two encryptions of a ciphertext are different). If this wouldn't happen the following attack could occur when encrypting with a **stream cipher**:

C1 = (P1 xor K)

C2 = (P2 xor K)

And the attacker knows C1, C2 and either of P1 or P2. Then that would be enough to know the value of the other one:

C1 xor C2 = (P1 xor K) xor (P2 xor K) = P1 xor P2

If K is the key stream obtained by encrypting IV with the encryption algorithm (K = E(IV,key)), and considering that key stays constant, the repetition of the IV would yield the same K and therefore the same cipher text. Then if the attacker would learn the encryption of a fragment that is encrypted lots of times during a normal communication, such as a handshake header, he/she would be in possesion of P1 and C1. Sniffing other communications and extracting C2 would easily allow him/her to find P2.

Another property thay any IV should have is **unpredictability** and **randomness**. Both properties come together but the birthday problem must be considered in order to avoid collisions and lose the first of the properties mentioned. If an attacker was able to predict any IV of the future (even if it was the first time it was being used) the following chosen-plaintext attack could occur:

Alice is using **CBC mode** to encrypt messages

Eve is able to forward plain-text messages to Alice for encryption

Alice sends: C_a = P_a xor K where K has been obtained E(IV_0,key) = K

Now Eve wants to check if her guess of P_a (called P_e) is correct and she has acces to IV_0 and can guess IV_1 (following IV), then she can forward Alice:

IV_1 xor P_e xor IV_0 and ask her to encrypt it

Alice would encrypt this message: IV_1 xor P_e xor IV_0

C_a = E(P_e xor IV_0) = E(IV_1 xor P_e xor IV_0 xor IV_1)

And if Eve receives C_a she will know that her guess was right.

If the IV for a specific cryptographic scheme doesn't need to be random but **only unique**, the **scheme** is called **stateful** (in the other case would be randomized). In stateful schemes sender and receiver share a common IV state and, even though the randomness property is not necessary, the **unpredictability** is **crucial** to make these schemes secure.

Another aspect related to IVs that isn't specifically a property is related to their secrecy after encrypting a message. IVs value associated to a specific message **can** be made **public after** the **message** has been **encrypted** as it has already served its purpose. On the other hand, if this value can be modified/altered before decryption occurs, the first block of the cipher text could be corrupted:

Attacker knows P (first block of plain text) and OIV (original IV), wants to decrypt to Z and can modified IV. Then he or she makes IV = P xor OIV xor Z and when decrypting the first block wll be Z as:

Z = (P xor OIV xor Z) xor (P xor OIV)

This shows how IVs **shouldn't** be able to **be altered before decryption**.

If the encryption occurs in an environment where the IV can be altered, some authentication options such as MAC or HMAC should be used.

Lab summary:

In this lab we have learnt how to use openssl to encrypt and decrypt files. We have also understood the difference between the block cipher modes respecting padding and error propagation. We have also see the effects IVs have on the ciphertext and the importance of maintaining their uniqueness, randomness and unpredictability. Possible attacks if these properties aren't ensured have been analysed and the vulnerabilities of using ECB have been clearly shown. In the provided readings the IVs properties were further explained and it was highlighted that encryption only provides secrecy and not integrity or authentication.