

Cryptography - Lab 3 - Block Cipher Operations

Javier Antxon Garrues Apecechea

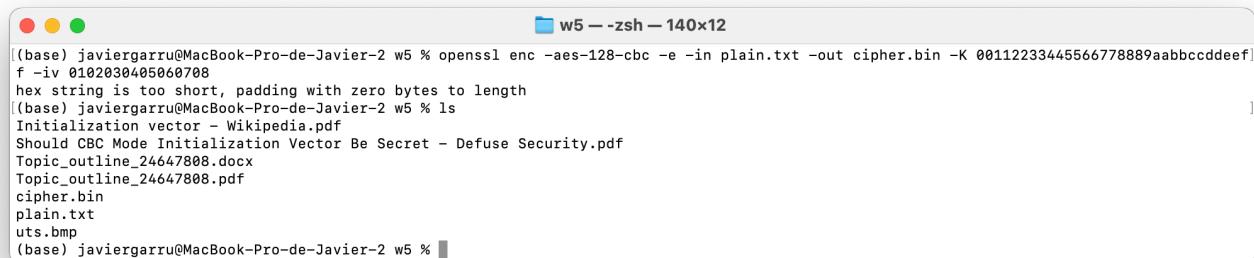
Student ID: 24647808

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

Part One: encrypting with different ciphers and modes

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

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

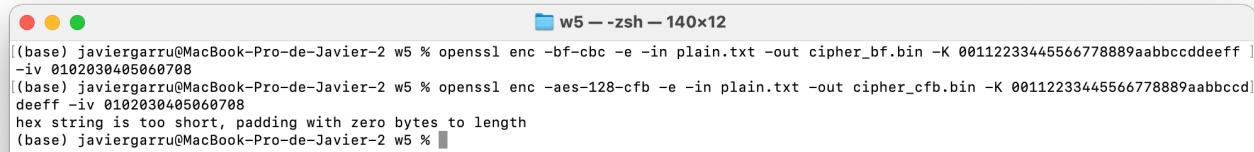


```
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -e -in plain.txt -out cipher.bin -K 00112233445566778889aabbccddeef  
f -iv 0102030405060708  
hex string is too short, padding with zero bytes to length  
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % ls  
Initialization vector - Wikipedia.pdf  
Should CBC Mode Initialization Vector Be Secret - Defuse Security.pdf  
Topic_outline_24647808.docx  
Topic_outline_24647808.pdf  
cipher.bin  
plain.txt  
uts.bmp  
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

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
```



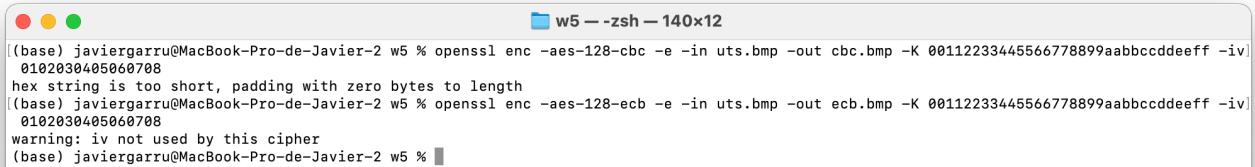
```
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -bf-cbc -e -in plain.txt -out cipher_bf.bin -K 00112233445566778889aabbccddeeff  
-iv 0102030405060708  
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cfb -e -in plain.txt -out cipher_cfb.bin -K 00112233445566778889aabbccdeff  
deeeff -iv 0102030405060708  
hex string is too short, padding with zero bytes to length  
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

Part Two: ECB vs CBC

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

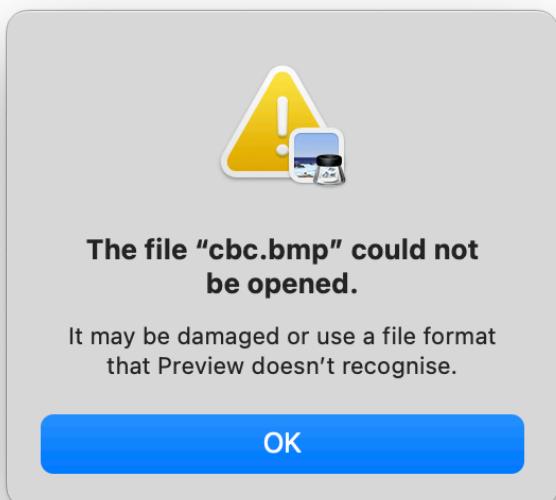
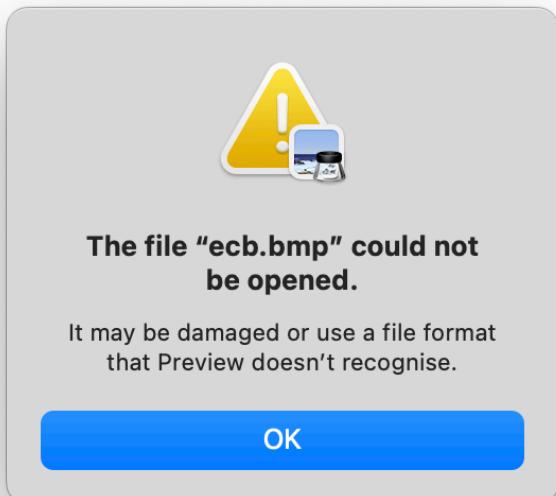
```
openssl enc -aes-128-ecb -e -in uts.bmp -out cbc.bmp -K  
00112233445566778899aabbccddeeff -iv 0102030405060708
```

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



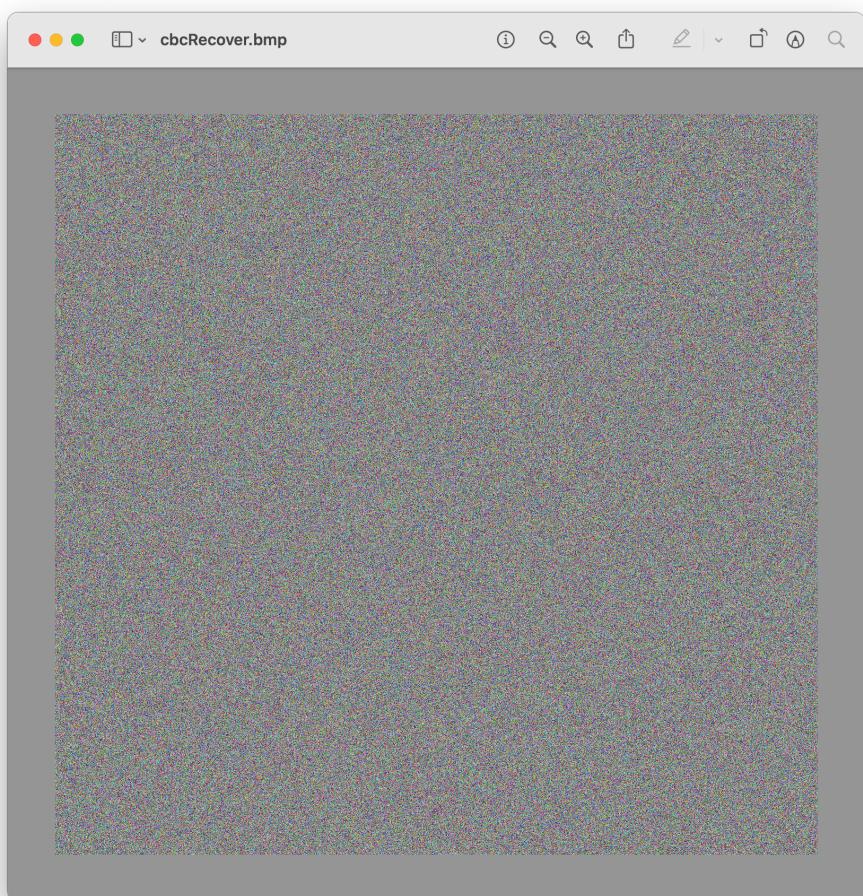
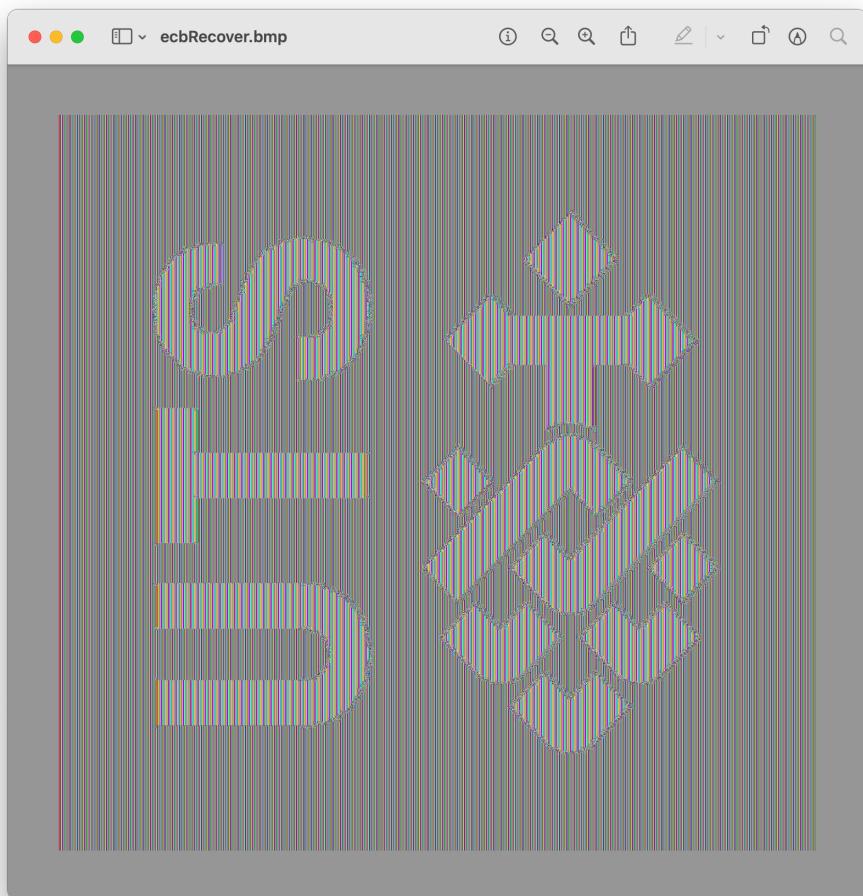
```
w5 — zsh — 140x12  
[(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -e -in uts.bmp -out cbc.bmp -K 00112233445566778899aabbccddeeff -iv  
0102030405060708  
hex string is too short, padding with zero bytes to length  
[(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -e -in uts.bmp -out ecb.bmp -K 00112233445566778899aabbccddeeff -iv  
0102030405060708  
warning: iv not used by this cipher  
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

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:

```
w5 — zsh — 142x12
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % tail -c +55 ecb.bmp > ecbBody
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % tail -c +55 cbc.bmp > cbcBody
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % head -c 54 uts.bmp > header
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % cat header cbcBody > cbcRecover.bmp
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % cat header ecbBody > ecbRecover.bmp
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```



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:

```
w5 -- zsh -- 142x31
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -e -in f5.txt -out f5Cipher_ecb.bin -K 00112233445566778899aabbcddde
ff
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -d -out f5NP.txt -in f5Cipher_ecb.bin -K 00112233445566778899aabbcdd
eff
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -d -out f5P.txt -in f5Cipher_ecb.bin -K 00112233445566778899aabbcdd
eff -nopad
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f5NP.txt
00000000 31 32 33 34 35 36 37 38 39 30 |1234567890|
0000000a
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f5P.txt
00000000 31 32 33 34 35 36 37 38 39 30 06 06 06 06 06 06 |1234567890.....|
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -e -in f6.txt -out f6Cipher_ecb.bin -K 00112233445566778899aabbcdd
ff
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -d -out f6NP.txt -in f6Cipher_ecb.bin -K 00112233445566778899aabbcdd
eff
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -d -out f6P.txt -in f6Cipher_ecb.bin -K 00112233445566778899aabbcdd
eff -nopad
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6NP.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6P.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000010 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 |.....|
00000020
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

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:

```
w5 --zsh - 142x37
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % echo -n "1234567890" > f5.txt
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % echo -n "1234567890123456" > f6.txt
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -e -in f5.txt -out f5Cipher.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -d -out f5NP.txt -in f5Cipher.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -d -out f5P.txt -in f5Cipher.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708 -nopad
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f5NP.txt
00000000 31 32 33 34 35 36 37 38 39 30 |1234567890|0000000a
0000000a
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f5P.txt
00000000 31 32 33 34 35 36 37 38 39 30 |06 06 06 06 06 06|00000010
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -e -in f6.txt -out f6Cipher.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -d -out f6NP.txt -in f6Cipher.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -d -out f6P.txt -in f6Cipher.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708 -nopad
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6NP.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|00000010
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6P.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|00000010
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|00000010
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

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.

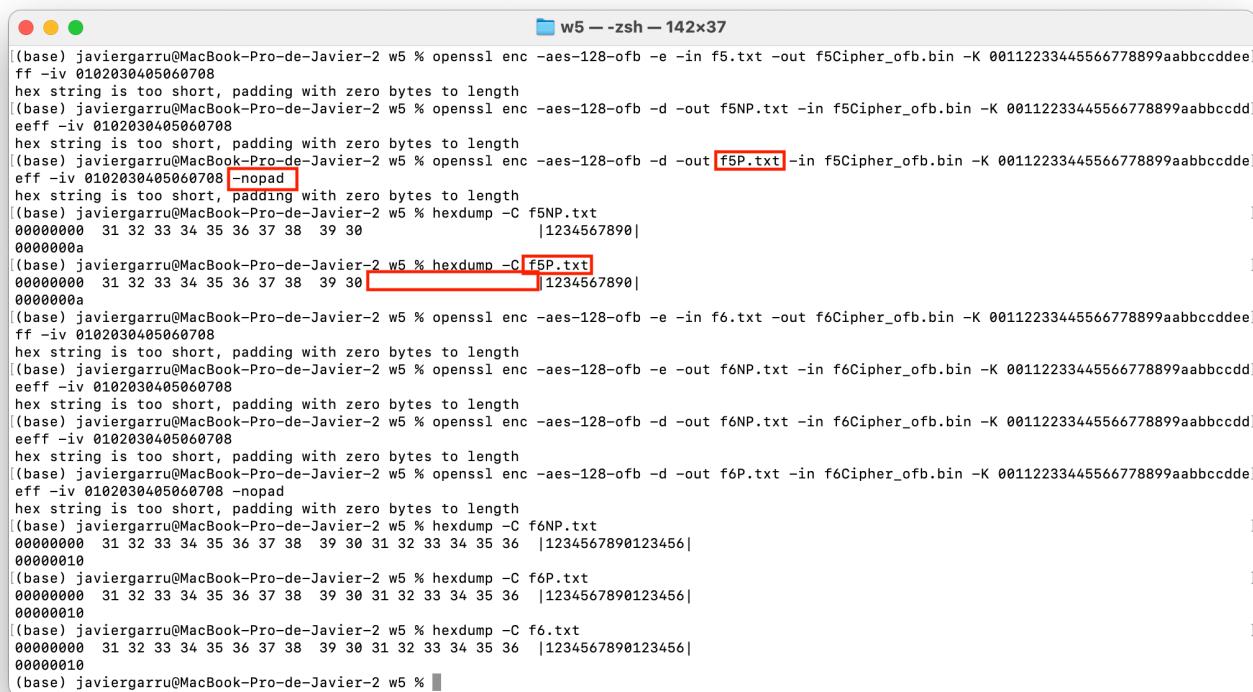
CFB:

```
w5 --zsh - 142x35
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cfb -e -in f5.txt -out f5Cipher_cfb.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cfb -e -out f5NP.txt -in f5Cipher_cfb.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708 -nopad
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f5NP.txt
00000000 31 32 33 34 35 36 37 38 39 30 |1234567890|0000000a
0000000a
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f5P.txt
00000000 31 32 33 34 35 36 37 38 39 30 |1234567890|00000010
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cfb -e -in f6.txt -out f6Cipher_cfb.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cfb -e -out f6NP.txt -in f6Cipher_cfb.bin -K 00112233445566778899aabccddeeff -iv 0102030405060708 -nopad
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6NP.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|00000010
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6P.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|00000010
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f6.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|00000010
00000010
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

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:



The screenshot shows a terminal window titled "w5 -- zsh - 142x37". The terminal displays a series of OpenSSL commands for OFB mode. The user runs "openssl enc -aes-128-ofb -e -in f5.txt -out f5Cipher_ofb.bin -K 00112233445566778899aabccdd" followed by "ff -iv 0102030405060708". A message indicates "hex string is too short, padding with zero bytes to length". Then, "openssl enc -aes-128-ofb -d -out f5NP.txt -in f5Cipher_ofb.bin -K 00112233445566778899aabccdd" is run with the same IV, resulting in another "padding with zero bytes to length" message. The user then runs "openssl enc -aes-128-ofb -d -out f5P.txt -in f5Cipher_ofb.bin -K 00112233445566778899aabccdd" with the IV "ff -iv 0102030405060708 -nopad", which succeeds without padding. Hex dumps of f5NP.txt and f5P.txt are shown, both containing the same data: "00000000 31 32 33 34 35 36 37 38 39 30 |1234567890|". The user then runs "hexdump -C f5P.txt" and "hexdump -C f6P.txt" (with the IV "ff -iv 0102030405060708") to compare them, showing identical results: "00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|". Finally, "hexdump -C f6P.txt" is run again with the IV "ff -iv 0102030405060708", showing the same output.

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 provide justification.

ECB:

```
w5 --zsh - 142x20
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -e -in f0.txt -out f0ecb -K 00112233445566778899aabbccddeeff
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -d -out f0ecbRN.txt -in f0ecb -K 00112233445566778899aabbccddeeff
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ecb -d -out f0ecbR.txt -in f0ecb -K 00112233445566778899aabbccddeeff
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0ecbRN.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 [1234567890123456]
00000010 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 [7890123456789012]
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 [3456789012345678]
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 [9012345678901234]
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0ecbR.txt
00000000 f5 5b b7 34 12 07 6b 23 77 d6 f7 b7 01 86 60 73 [?/?..k#?w???.`$]
00000010 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 [7890123456789012]
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 [3456789012345678]
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 [9012345678901234]
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

f0ecb																
00	BB	A3	01	73	75	E3	06	7D	CE	4A	3E	EF	CD	2E	84	70
1B	65	0C	30	72	AB	22	F9	F5	40	B5	41	82	B8	1A	07	A7
36	9D	02	73	1D	14	59	77	B8	26	F9	00	65	7E	A1	40	65

```

f0ecb
00 BC A3 01 73 75 E3 06 7D CE 4A 3E EF CD 2E 84 70 94 FF 08 AD 95 0E CA 0D 1D 5B 65 .. su. ].J>....p... . . [e
1B 65 0C 30 72 AB 22 F9 F5 40 B5 41 82 B8 1A 07 A7 B2 ED DD D2 57 F8 DC E1 15 CF 10 e 0r."..@.A.. ....W.... .
36 9D 02 73 1D 14 59 77 B8 26 F9 00 65 7E A1 40 65 5A 44 78 27 47 70 5D 42 2F AD . s Yw.&. e~.@eZDx'Gp]B/.

0x1 out of 0x50 bytes

```

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 file are corrupted but the rest bytes are left unchanged.

CBC:

```

w5 --zsh -- 142x33
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -e -in f0.txt -out f0cbc -K 00112233445566778889aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -d -in f0cbc -out f0cbcRN.txt -K 00112233445566778889aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0cbcRN.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000010 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 |7890123456789012|
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 |3456789012345678|
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 |9012345678901234|
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -d -in f0cbc -out f0cbcR.txt -K 00112233445566778889aabccddeeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0cbcRN.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000010 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 |7890123456789012|
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 |3456789012345678|
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 |9012345678901234|
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0cbcR.txt
00000000 53 56 10 20 25 5f f2 2f a9 81 29 de 51 cf ef ff |SV. % ?/?.)?Q???
00000010 30 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 |0890123456789012|
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 |3456789012345678|
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 |9012345678901234|
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %

```

f0cbc

00	EB	FB	05	81	98	77	71	C9	46	02	98	BA	32	85	A2	C2	B4	B0	CA	87	E6	EB	31	51	2E	EE	16wq.F ..2.....1Q..
1B	S2	06	8D	AD	38	A2	82	36	9C	EB	4E	D6	B5	1C	0B	A8	30	7E	FB	1B	89	8B	0F	F8	19	7A	32	R ..8..6..N.. .0~. . . z2
36	8C	98	F6	71	7E	14	7E	02	AF	00	2B	C6	2F	69	17	85	87	7A	15	22	F9	55	DE	CC	19	74	...q~ ~ . +./i ..z ".U.. t	

Byte 0x0 selected out of 0x50 bytes

f0cbc

00	EC	FB	05	81	98	77	71	C9	46	02	98	BA	32	85	A2	C2	B4	B0	CA	87	E6	EB	31	51	2E	EE	16wq.F ..2.....1Q..
1B	S2	06	8D	AD	38	A2	82	36	9C	EB	4E	D6	B5	1C	0B	A8	30	7E	FB	1B	89	8B	0F	F8	19	7A	32	R ..8..6..N.. .0~. . . z2
36	8C	98	F6	71	7E	14	7E	02	AF	00	2B	C6	2F	69	17	85	87	7A	15	22	F9	55	DE	CC	19	74	...q~ ~ . +./i ..z ".U.. t	

0x0 out of 0x50 bytes

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:

```
w5 -- zsh -- 142x23
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cfb -e -in f0.txt -out f0cfb -K 00112233445566778899aabccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cfb -d -out f0cfbRN.txt -in f0cfb -K 00112233445566778899aabccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cfb -d -out f0cfbR.txt -in f0cfb -K 00112233445566778899aabccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0cfbRN.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000001 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 |7890123456789012|
00000002 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 |3456789012345678|
00000003 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 |9012345678901234|
00000004
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0cfbR.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000001 79 9c 12 70 2d 9e 6e 9e a0 10 9a 25 3b 99 81 0d |y.p.n.?..%...|
00000002 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 |3456789012345678|
00000003 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 |9012345678901234|
00000004
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

The screenshot shows a hex editor window titled "f0cfb". The left pane displays a 16x16 grid of hex bytes. The first byte at address 00 is highlighted with a red box and contains the value 71. The right pane shows the corresponding ASCII representation: "q.5.1..hul.{e..{.1.1.0.. . . . i...aU.. '....Q u./.. . . fW.\$..m.". A status bar at the bottom indicates "Byte 0x0 selected out of 0x40 bytes".

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, encrypted 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 screenshot shows a terminal window titled "w5 -- zsh -- 142x23". The user runs several commands to demonstrate OFB mode. They generate a file "f0.txt" with an IV, encrypt it with "openssl enc -aes-128-ofb", and then decrypt it with "openssl enc -aes-128-ofb -d". They also use "hexdump -C" to show the raw bytes of the files. Red boxes highlight specific bytes in the hex dump of the decrypted file, specifically the first byte of the first block (00000000) which is 31 and 32 respectively, and the first byte of the second block (00000010) which is 37 and 38. A red arrow points from the first byte of the first block in the hex dump to its corresponding byte in the first block of the second file's hex dump.

```
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ofb -e -in f0.txt -out f0fb -K 00112233445566778899aabcccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ofb -d -out f0fbRN.txt -in f0fb -K 00112233445566778899aabcccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ofb -d -out f0fbR.txt -in f0fb -K 00112233445566778899aabcccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0fbRN.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36  [34567890123456]
00000010 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32  [7890123456789012]
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38  [3456789012345678]
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34  [9012345678901234]
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0fbR.txt
00000000 32 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36  [234567890123456]
00000010 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32  [7890123456789012]
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38  [3456789012345678]
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34  [9012345678901234]
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

00	72	C7	35	F0	31	82	B2	68	75	6C	FB	7D	65	88	D7	7B	E4	86	36	25	E7	D3	96	0B	A8	7E	EC	r.5.1..hul.{e..{..6%... .~.
1B	42	FD	EF	08	D3	AC	2E	FF	75	1F	87	B2	E8	78	69	CC	7E	B4	FB	07	21	5A	78	FA	35	90	B3	B..u ...xi.~... !Zx.5..
36	15	68	D6	FE	62	D5	03	12	02	D9																h..b. .		

0x40 out of 0x40 bytes

00	71	C7	35	F0	31	82	B2	68	75	6C	FB	7D	65	88	D7	7B	E4	86	36	25	E7	D3	96	0B	A8	7E	EC	q.5.1..hul.{e..{..6%... .~.
1B	42	FD	EF	08	D3	AC	2E	FF	75	1F	87	B2	E8	78	69	CC	7E	B4	FB	07	21	5A	78	FA	35	90	B3	B..u ...xi.~... !Zx.5..
36	15	68	D6	FE	62	D5	03	12	02	D9																h..b. .		

0x1 out of 0x40 bytes

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:

```
w5 -- zsh - 142x23

(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ctr -e -in f0.txt -out f0ctr -K 00112233445566778899aabccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ctr -d -out f0ctrRN.txt -in f0ctr -K 00112233445566778899aabccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-ctr -d -out f0ctrR.txt -in f0ctr -K 00112233445566778899aabccddeeff -iv 010203040506070809
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0ctrRN.txt
00000000 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000010 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 |7890123456789012|
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 |3456789012345678|
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 |9012345678901234|
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f0ctrR.txt
00000000 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 |1234567890123456|
00000010 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 |7890123456789012|
00000020 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 |3456789012345678|
00000030 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 |9012345678901234|
00000040
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %
```

	f0ctr															
00	71 C7 35 F0 31 82 B2 68 75 6C FB 7D 65 88 D7 7B C3 C1 94 2F F3 95 0A DC C4 1F 0F															
1B	8E BA F9 C6 32 EE AD F2 6F 2B 0D F8 05 49 68 9C EE 89 D3 52 47 CE 06 E1 D7 07 0C															
36	A0 79 6E 35 2A A0 51 2E 6D 4E															

0x1 out of 0x40 bytes

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
```

```

(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % echo -n "1234567890123456789012345678901234567890123456789012345678901234" > f3.txt
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -e -in f3.txt -out f3cbcIV2 -K 00112233445566778899aabcccddeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % openssl enc -aes-128-cbc -e -in f3.txt -out f3cbcIV -K 00112233445566778899aabcccddeff -iv 0102030405060708
hex string is too short, padding with zero bytes to length
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f3cbcIV
00000000  80 79 3c 08 5c 0c 36 90  0c c7 85 fe cb 30 b1 a7  |..y<.\.6..??.?0??|
00000010  95 c9 c0 4c d8 3e a5 55  05 61 52 57 19 59 14 e2  |.??L??.?U.aRW.Y.?|
00000020  d3 f7 ef f5 3f 0a 03 1f  74 e2 6b ed 3c 83 50 25  |?????..t?K?<.P%|
00000030  e0 09 45 8a 57 04 f5 02  06 0d c5 51 9b 65 05 bc  |?.E.W?...?Q.e.?|
00000040  0e 88 49 6b f2 e5 0f 8b  5c 12 5a 4e c2 41 f6 bd  |..Ik??.\ZN?A??|
00000050
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 % hexdump -C f3cbcIV2
00000000  d1 cc e2 81 59 0c bb 06  ab 98 5e 0f 8e 0f 43 e7  |???.Y.?..?^.C?|
00000010  e8 cf c0 64 32 33 0f ec  8e 8a 88 63 24 dd c5 a2  |???.d23.?.?..c$?T|
00000020  96 95 89 71 0b 28 b5 f3  dd b7 64 fe 3d 01 a9 73  |...q.(?.?d?=.?S|
00000030  64 10 63 86 c5 14 6c e8  9d 4e 45 19 21 c6 35 bb  |d.c.?..1?.NE.!?5?|
00000040  0f a7 1d 53 ed c4 ba ee  a0 0c 92 e4 ae fd 7e 2e  |.?.S?1?..??.|
00000050
(base) javiergarru@MacBook-Pro-de-Javier-2 w5 %

```

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**:

$$C1 = (P1 \text{ xor } K)$$

$$C2 = (P2 \text{ xor } K)$$

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:

$$C1 \text{ xor } C2 = (P1 \text{ xor } K) \text{ xor } (P2 \text{ xor } K) = P1 \text{ xor } P2$$

If K is the key stream obtained by encrypting IV with the encryption algorithm ($K = E(IV, \text{key})$), 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 possession of P1 and C1. Sniffing other communications and extracting C2 would easily allow him/her to find P2.

Another property that 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: $C_a = P_a \text{ xor } K$ where K has been obtained $E(IV_0, \text{key}) = K$

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

$IV_1 \text{ xor } P_e \text{ xor } IV_0$ and ask her to encrypt it

Alice would encrypt this message: $IV_1 \text{ xor } P_e \text{ xor } IV_0$

$C_a = E(P_e \text{ xor } IV_0) = E(IV_1 \text{ xor } P_e \text{ xor } IV_0 \text{ xor } IV_1)$

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 modify IV. Then he or she makes $IV = P \text{ xor } OIV \text{ xor } Z$ and when decrypting the first block will be Z as:

$Z = (P \text{ xor } OIV \text{ xor } Z) \text{ xor } (P \text{ xor } OIV)$

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 seen 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.