SECRET-KEY ENCRYPTION LAB

SEED Lab #4

Identification

- **Group nº3**
  - **José Rodrigues** : 201708806
  - **Paulo Ribeiro** : 201806505
  - **Pedro Ferreira** : 201806506

Task 1:

- First, we run the program `freq.py`, which takes the cryptogram as the input and produces the statistics for n-grams, namely the single-letter frequencies, bigram frequencies (2-letter sequence), and trigram frequencies (3-letter sequence).
- The output of the program shows that 'n' is the most common single letter, 'yt' most common bigram, and 'ytn' is the most common trigram. After some searching, we discovered that 'e' is the most common letter in the English language, 'th' is the most common bigram, and 'the' is the most common trigram. This strongly suggests that Y~t, T~h and N~e.
- The next most common letter in the cryptogram is V (N and Y are already taken). Since the first and second most frequent letters in the English language, 'e' and 't' are accounted for, we guessed that V~a since it is the third most frequent letter.
- Tentatively making these assumptions, the following partial decrypted message is obtained:

```
the XQAaHQ tZHU  XU QZUPaD LhMAh QeeCQ aGXZt HMRht aBteH thMQ IXUR QtHaURe
aLaHPQ tHME the GaRReH BeeIQ IMSe a UXUaReUaHMaU tXX

the aLaHPQ HaAe LaQ GXXSeUPeP GD the PeCMQe XB haHFeD LeMUQteMU at MtQ XZtQet
aUP the aEEaHeUt MCEIXQMXU XB hMQ BMIC AXCEaUD at the eUP aUP Mt LaQ QhaEeP GD
the eCeHReUAe XB CetXX tMCeQ ZE GIaASRXLU EXIMtMAQ aHCAaUPD aAtMFMQC aUP
a UatMXUaI (...)
```

- Using these initial guesses, we continued by looking into the message and inferring some possible correspondences by thinking of English words. Finally, we were able to fully decrypt the message.

```
the oscars turn  on sunday which seems about right after this long strange
awards trip the bagger feels like a nonagenarian too

the awards race was bookended by the demise of harvey weinstein at its outset
and the apparent implosion of his film company at the end and it was shaped by
the emergence of metoo times up blackgown politics armcandy activism and
a national (...)
```

- In the end, we've discovered that the cyphered text corresponds to this New York Times Article.

Task 2:

- For this task, we experimented with four different cypher types, namely:
  - Advanced Encryption Service as the encryption algorithm, with a bit rate of 128 and in the Cipher Block Chaining mode (aes-128-cbc)
  - Blowfish as the encryption algorithm and in the Cipher Block Chaining mode (bf-cbc)
  - Advanced Encryption Service as the encryption algorithm, with a bit rate of 128 and in the Cipher Feedback mode (aes-128-cfb)
  - Advanced Encryption Service as the encryption algorithm, with a bit rate of 128 and in the Counter mode (aes-128-ctr)

```
openssl enc -aes-128-cbc -e -in ciphertextsolved.txt -out cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708010203040506070 8

openssl enc -bf-cbc -e -in ciphertextsolved.txt -out cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cfb -e -in ciphertextsolved.txt -out cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708010203040506070 8

openssl enc -aes-128-ctr -e -in ciphertextsolved.txt -out cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708010203040506070 8
```

Task 3:

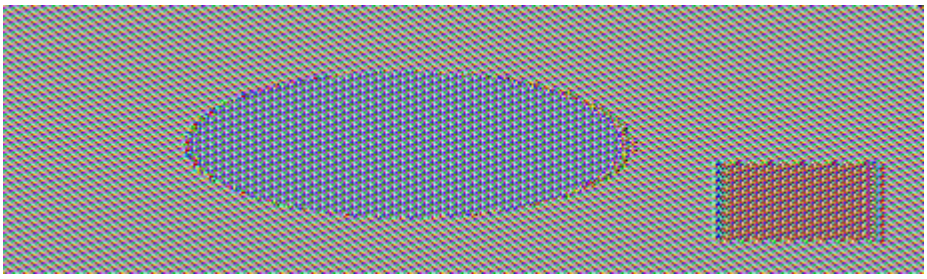
- We started by looking at the original picture, which contains a red flat circle and a blue rectangle.

Original image:

- Then, we encrypted it using the AES algorithm in the Electronic Code Book mode (**ECB**).

```
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_original_ecb.bmp -K
00112233445566778889aabbccddeeff
```

Resulting image:



- We also experimented encrypting the image using the AES algorithm but in the Cipher Block Chaining mode (**CBC**).

```
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_original_cbc.bmp -K
00112233445566778889aabbccddeeff -iv 0102030405060708010203040506070 8
```

Resulting image:

- As it can be seen, using the **ECB** mode, the two shapes are still easily recognizable in the encrypted picture. This happens because, in this mode, each ciphertext block is obtained by applying the DES encryption process to the current plaintext block directly. So the current ciphertext block has no dependency on any previous plaintext blocks. This leads to a big disadvantage of this mode since identical plaintext blocks are encrypted to identical ciphertext blocks, not hiding data patterns well. For this reason, **ECB** mode doesn't provide message confidentiality at all, and it is not recommended for cryptographic protocols.

- On the other hand, using the **CBC** mode, the encrypted picture is imperceptible, in a way that people without the encryption keys cannot know what it contains. In this mode, the first block of the plaintext is exclusive-OR'd (XOR'd) with an initialization vector (IV), which is a block of random bits of plaintext, prior to the application of the encryption key. The resultant block is the first block of the ciphertext. Each subsequent block of plaintext is then XOR'd with the previous block of ciphertext before encryption, hence the term "chaining". Due to this XOR process, the same block of plaintext will no longer result in identical ciphertext being produced.

Task 4:

Task 4.1:

- The **ECB** and **CBC** modes require the use of padding, while the CFB and OFB modes do not.
- This happens since **ECB** and **CBC** need their input to be an exact multiple of the block size (in AES it's 16), while in the CFB and OFB (streaming modes, which can encrypt and decrypt messages of any size) the ciphertext will be the same size as the plaintext so padding is not required.

Task 4.2:

- We started by creating three files of different sizes (5, 10 and 16 bytes). Then, after encrypting them using the 128-bit AES with **CBC** mode, we checked their sizes and verified they had changed to be an exact multiple of the block size (composed of 16 bytes). Notice that in the last file, despite the size of the file already being a multiple of the block size, padding still occurs, and the encrypted file ends up with the size of the next multiple of the AES block size, which in this case is 32. This happens since PKCS padding rules say that padding is always applied.

Original file size	Encrypted file size
5 bytes	16 bytes
10 bytes	16 bytes
16 bytes	32 bytes

- After decrypting the three files without removing the padding that was added (using `-no-pad`), we verify that each padding byte has a value equal to the total number of padding bytes that are added. In the first case, the value `0b` was used as the padding value since it corresponds to the eleven extra bytes added during the padding. The same happens for the other two files (`06` for 6 added bytes and `10` for 16 added bytes).

```
[04/03/22]seed@VM:~/.../Files$ hexdump -C df1.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
[04/03/22]seed@VM:~/.../Files$ hexdump -C df2.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
[04/03/22]seed@VM:~/.../Files$ hexdump -C df3.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
```

Task 5:

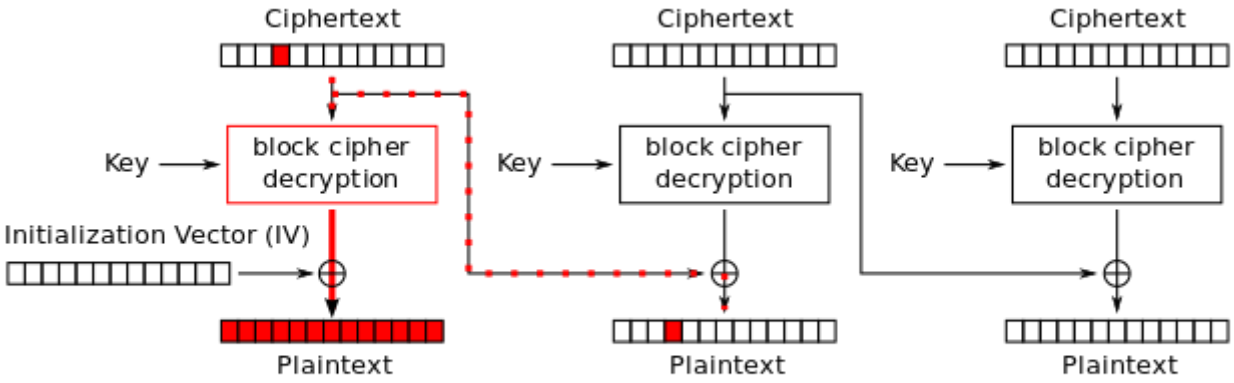
- First, we created a file with 1000 bytes and encrypted it using a AES-128 cypher in the modes **ECB**, **CBC**, **CFB**, and **OFB**.
- Then, we corrupted the 55th byte of the encrypted files using the bless hex editor.
- Finally, we decrypted the corrupted ciphertexts and checked how many bytes of the plaintext were altered during the decryption using the four different modes, using the following command:

```
cmp -lb bigfile.txt decryptedbigfile.txt | wc -l
```

Encryption mode	Different bytes
ECB	16
CBC	17
CFB	17

Encryption mode	Different bytes
OFB	1

- In the **ECB** mode, since each block is decrypted independently, the single byte corruption only affects the block to which it belongs, leading to the alteration of 16 bytes (size of one block).
- The **CBC** mode has a self-healing property: if one block of the cypher is altered, the error propagates for at most two blocks. In this case, where only one byte of the ciphertext is altered, the whole corresponding block will be affected because the change of a single byte will cause the output block of Block Cypher Encryption to be different, consequently leading to a corrupted plaintext of that block. Besides this, since this corrupted ciphertext is used as input of the XOR that generates the plaintext of the following encryption block, one byte of that following block will also be corrupted. This leads to 16 + 1 = 17 corrupted bytes. This can be seen in the following image (red symbolizes corrupted bytes):



Cipher Block Chaining (CBC) mode decryption

- In the **CFB** mode, the plaintext of the current block is a direct result of the XOR between the output block of Block Cypher Encryption and the ciphertext, therefore, if one byte of the ciphertext is corrupted, only one byte of the current plaintext will also be corrupted. Besides this, since this corrupted ciphertext is used as input of the Block Cypher Encryption of the following block, this will result in a destroyed output block, which will then be used as input of the XOR that generates the plaintext of that block, corrupting the whole plaintext of that following block.
- In the **OFB** mode, since only a single bit of the cypher block is corrupted, when the cypher block is XOR'd with the output block of Block Cypher Encryption, only a single bit of that corresponding plaintext block will be corrupted. The error will not occur in other plaintext blocks since the corrupted cypher-block is not shared by other encryption blocks.

Task 6:

Task 6.1:

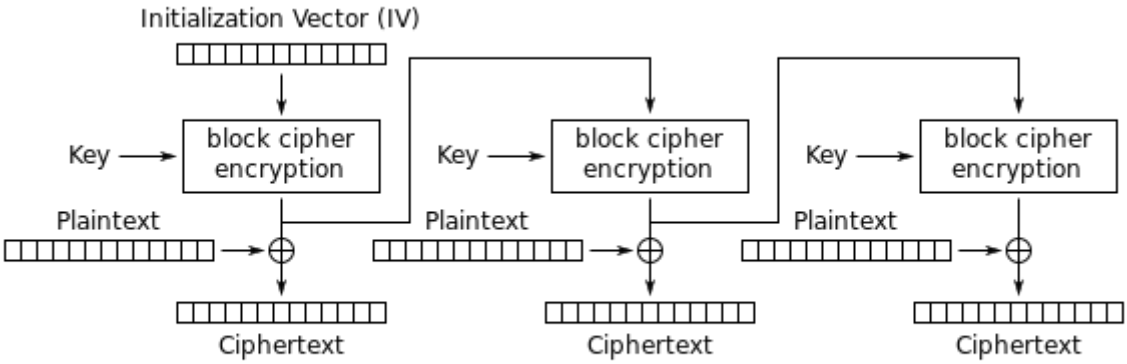
- After encrypting the plaintext of the previous task twice using the same IV and twice again but with two different IVs, we run the following commands to know how many bytes the two ciphertexts differ on:

```
[04/05/22]seed@VM:~/.../task6$ cmp -lb cypherdiff1.bin cypherdiff2.bin | wc -l
1006
[04/05/22]seed@VM:~/.../task6$ cmp -lb cypherequal1.bin cypherequal2.bin | wc -l
0
```

- As we can see, using two different IVs results in different ciphertexts, whereas using the same IV leads to the same ones. Thus, no IV may be reused under the same key in order to make cryptographic attacks harder to execute.

Task 6.2:

- In the Output Feedback mode (OFB), a *known-plaintext attack* is possible, where the attacker has access to both the plaintext and its encrypted version (ciphertext), revealing secret information by being able to decipher other ciphertexts.
- By looking at the following image that describes encryption in this mode, we conclude that reusing IV under the same key leads to the same output stream of the Block Cipher Encryption. This output stream is then XOR'd with the plaintext to generate the corresponding ciphertext. Since XOR is a symmetric operator, we may find this output stream by calculating the XOR between the known plaintext and its corresponding ciphertext. Then, as this same output stream was used to generate other ciphertexts, we are able to know any secret plaintexts just by calculating the XOR between the output stream and those ciphertexts.



Output Feedback (OFB) mode encryption

- In this case, by calculating the XOR between `P1` and `C1`, we find the output stream of the Block Cipher Encryption, and after XOR'ing it with `C2` we discover the secret plaintext `P2`. Finally, we just have to convert the hexadecimal value to ASCII characters to understand the secret message. With this in mind, we modified the given `sample_code.py` file to do the *known-plaintext* attack as we intended, revealing the unknown *plaintext*:

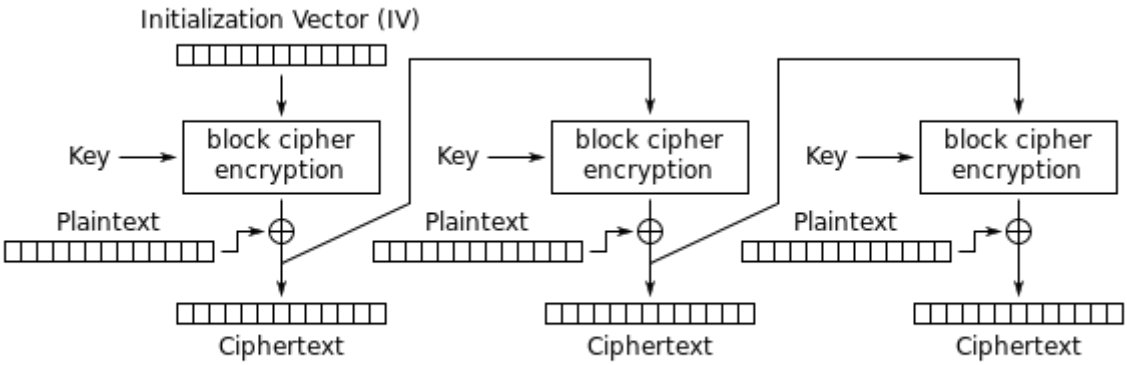
```
[04/05/22]seed@VM:~/.../task6.2$ python3 modified_sample_code.py
This is the secret key: f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a478
This is the plaintext: Order: Launch a missile!
```

modified_sample_code.py:

```
#!/usr/bin/python3

# XOR two bytearrays
def xor(first, second):
    return bytearray(x^y for x,y in zip(first, second))
```

```
MSG = "This is a known message!"
HEX_1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159"
HEX_2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc159"

# Convert ascii string to bytearray
D1 = bytes(MSG, 'utf-8')

# Convert hex string to bytearray
D2 = bytearray.fromhex(HEX_1)
D3 = bytearray.fromhex(HEX_2)

r1 = xor(D1, D2)
r2 = xor(r1, D3)
print("This is the secret key:", r1.hex())
print("This is the plaintext:", bytes.fromhex(r2.hex()).decode("ASCII",
"replace"))
```

- In the Cipher Feedback mode (CFB), this attack will only yield the plaintext of the first encrypted block since the value of the IV is only used in the Block Cipher Encryption of the very first block, and every consequent Block Cipher Encryption uses the previous ciphertext as input instead of the IV, leading to a different output stream. This can be seen in the following image:



Cipher Feedback (CFB) mode encryption

- For curiosity's sake, we tried encrypting the same two messages, `P1` and `P2`, in CFB mode using the same IV. We then used the `P1` plaintext and the resulting `C1` and `C2` ciphertexts and ran them in our `modified_sample_code.py`, getting the following result, as expected (only the first encryption block is successfully deciphered):

```
[04/05/22]seed@VM:~/.../task6.2$ python3 modified_sample_code.py
This is the secret key: 525476a0a854704b59a2e5630974255791f908c03206795c
This is the plaintext: Order: Launch a k��  
```

Task 6.3:

- After running Bob's program, we get the initial ciphertext (`C1`) which corresponds to his secret message that we know to be either "Yes" or "No" (`P1`). Besides this, we also get to know the IV used to encrypt it

(`IV1`). Then, Bob presents the next IV that will be used to encrypt the plaintext that we will give as input (`IV2`).

- We can use both these IVs to guess which of the possible options the secret message is by performing what is known as a *Chosen-Plaintext attack*.
- In this attack, we calculate the XOR between `IV1`, `IV2`, and our guessed plaintext (`G`), to obtain the plaintext which we will send as the input to Bob (`P2 = IV1 XOR IV2 XOR G`). Bob will then encrypt that input.
- The resulting ciphertext (`C2`), if we guessed the secret message correctly, will be an exact match to the initial ciphertext, indicating to us that we have indeed found the secret message, which in this case, was "Yes".
- Note that for our guessed plaintext input, we had to account for padding, by adding the amount of bytes needed to achieve the block size of 16 bytes, repeating the byte representing that amount. The output ciphertext contains 32 bytes, but we are only interested in the first 16 since the others result from the necessary padding.

Result:

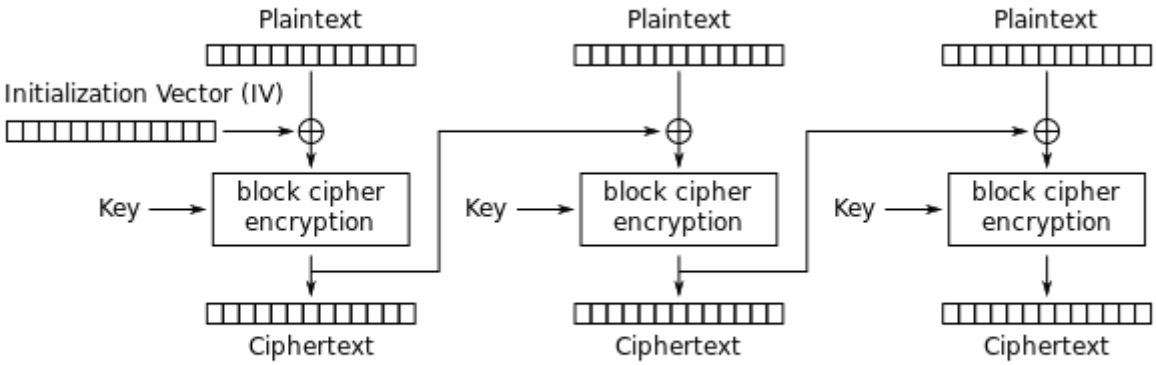
```
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: df8d1f1f97a523b96ae06eb3bbc86795
The IV used    : 248b724e5f1206067adaf98cd2b3462f

Next IV        : ee6cffa65f1206067adaf98cd2b3462f
Your plaintext : 9382fee50d0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: df8d1f1f97a523b96ae06eb3bbc867954e085de517e32e972ee825c229b58a64
```

Code we made to achieve this:

```
# XOR two bytearrays
def xor(first, second):
    return bytearray(x^y for x,y in zip(first, second))

MSG = "Yes"
knownIV = "248b724e5f1206067adaf98cd2b3462f"
nextIV = "945bb7b25f1206067adaf98cd2b3462f"

# Convert message string to bytearray
bMSG = bytes(MSG, 'utf-8')
psize = 16 - len(MSG)
bMSG += bytearray((psize,)) * psize

# Convert hex string to bytearray
bknownIV = bytearray.fromhex(knownIV)
bnextIV = bytearray.fromhex(nextIV)

sK = xor(bknownIV, bnextIV)
cMSG = xor(sK, bMSG)

print("Secret Key:", sK.hex())
print("Encrypt this:", cMSG.hex())
```

Task 7:

- By calculating the XOR between the plaintext and the IV we obtain the stream that will be used as input to the Block Cipher Encryption. This stream will then be used together with the secret key to generate the ciphertext, as we can see in the following figure.



Cipher Block Chaining (CBC) mode encryption

- Therefore, we could try to brute force this problem by experimenting to encrypt the plaintext using each key of the english word list provided (we know that the key is an english word with less than 16 characters, therefore it will only consist of a block) along with the stream. If the generated ciphertext is the same as the one provided, we've found the key that was used for the encryption.

Result:

```
Line: Syracuse

764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2
764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2


The secret key is: Syracuse
```

Code we wrote to achieve this result:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

plaintext = "This is a top secret."
ciphertext = "764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2"
IV = "aabbccddeeff00998877665544332211"

# Pad to AES Block Size
msg = pad(plaintext.encode(), AES.block_size)

# Convert hex string to bytearray
bIV = bytearray.fromhex(IV)

words = open('words.txt', 'r')
Lines = words.readlines()

for line in Lines:
```

```
    key = line.rstrip('\n')
    psize = 17 - len(line)
    key += '#' * psize
    key = bytes(key, 'utf-8')
    if(len(key) > 16):
        continue
    cipher = AES.new(key, AES.MODE_CBC, bIV)
    r_ciphertext = cipher.encrypt(msg)
    print("Line:", line)
    print(r_ciphertext.hex())
    print(ciphertext)
    print("\n")
    if(r_ciphertext.hex() == ciphertext):
        print("The secret key is:", line)
        quit()

print("The secret key was not found.. :(")
```