AES implementation

Homework2 - CNS Sapienza

Martina Evangelisti 1796480

30/10/2020

1 Introduction

AES, which stands for *Advanced Encryption Standard* is the most used symmetric block cipher algorithm nowadays. The standard specifies an algorithm made of four layers belonging to a round which is repeated a number of times that depends on the length of the key. AES is byte-oriented and it processes data blocks of 128 bit. In AES-128 the key length is equal to the block size, but the algorithm is designed to handle keys of 192 bits or 256 bits and it's know respectively as AES-192 and AES-256.

As Claude Shannon identified in his report *"A Mathematical Theory of Cryptography"* there are two properties that every secure cipher should have: confusion and diffusion. The meaning of confusion is that every bit of the ciphertext should depend on different parts of the key but keeping secret the relationship between them. The concept of diffusion is that the change of one bit in the ciphertext has to affect at least the fifty percent of the plaintext. In AES, confusion is guaranteed by the first of the layers composing each round: *ByteSubstitution*. The following layer, composed by *ShiftRow* and *MixColumn*, provides diffusion. Last layer is *KeyAddition* and it provides key whitening, a technique to increase a block chiper's security. The first round it's preceded by the *KeyAddition* Layer and the last round it's slightly different from the others because it skips the *MixColumn* operation.

2 Implementation of Encryption and Decryption

The input of the encryption algorithm is a block made of 128 bits that are internally stored in a matrix called State which has 4 rows and Nb columns, where Nb is the size of the block divided by 32. All the operations described by the standard are applied to the state matrix that at the end will be copied onto the output of the encryption.

In AES-128 we have 10 rounds, and the structure of each round could be seen in the figure below.

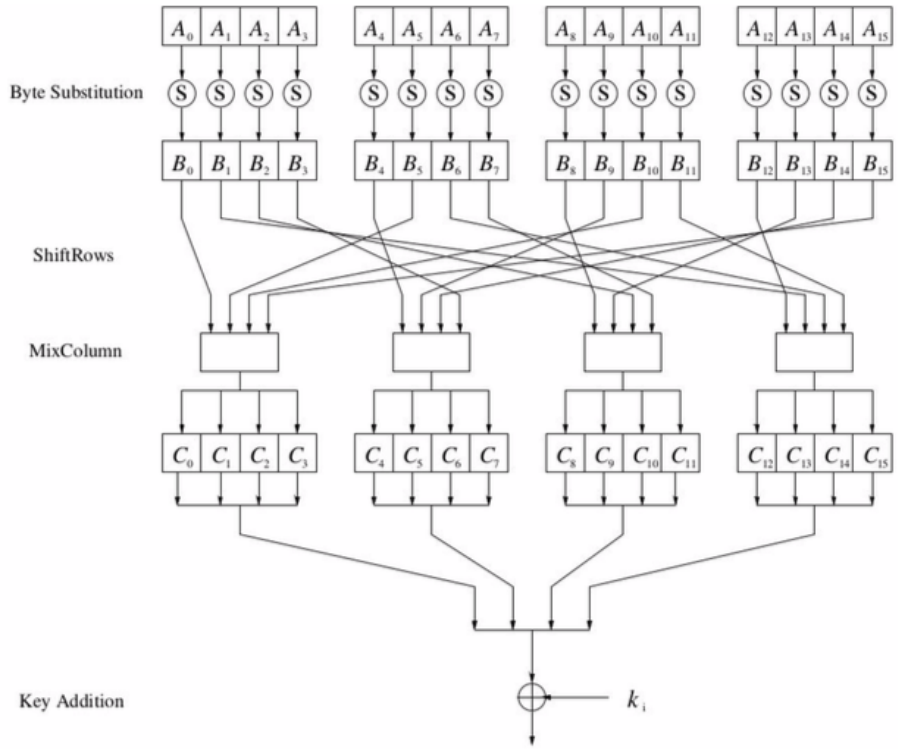


Figure 1: Work flow of a round [1]

2.1 Encryption

The encryption function that i implemented takes two parameters: input and word. The input represents a single block in AES algorithm and it's an array of int values representing the Unicode character of the characters from the plaintext. The word parameter is a matrix derived from the key-Expansion function which takes as an input the key and the value Nk.
Nk is the number of 32-bit words comprising the Cipher Key, so in AES-128 it's 4. As well as the input, the output is represented as an array of int values. The very first operation in the function body is the creation of the State from the input as a matrix of four rows and Nb columns. This is used as an intermediate Cipher result and all the steps in a round are applied to this matrix.

2.1.1 Byte Substitution Layer

This layers consists in a substitution performed on a byte level, based on a substitution table called S-Box. The function *SubBytes(state)* that i implemented, applies the S-Box to each byte of the State matrix. The S-Box is used as a look-up table so that for each element in row r and column c in the State, the transformation is:

$$state[r][c] = Sbox[state[r][c]] \tag{1}$$

2.1.2 Shift Rows Sublayer

The *ShiftRows* and the *MixColumns* operations together are part of the Diffusion Layer. The *ShiftRows(state)* function cyclically shifts the last three rows of the State over a different number as it follows:

$$S'[r][c] = S[r][(c + shift(r, Nb))modNb] \tag{2}$$

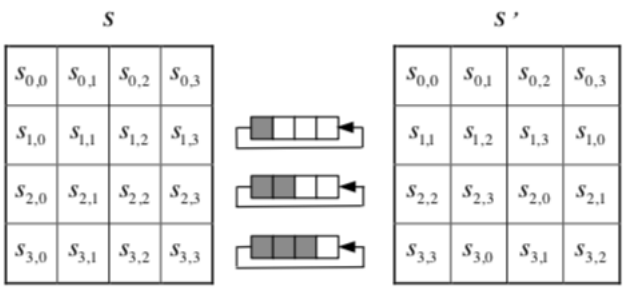


Figure 2: Shift Rows transformation applied to the State Matrix [**?**]

The function realizes this schema by some assignments, creating a new State matrix which has the values of the S' matrix shown in *Figure 2*.

2.1.3 Mix Columns Sublayer

In the implementation i made, this operation is realized by two functions: *MixColumns(state)* and *MixoneColumns(state,c)*. The first one only iterates on the columns and calls the *MixoneColumns(state,c)* function. For each columns the transformation to be applied is a multiplication where each column is considered a polynomial in GF(2^8).

$$\begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{pmatrix}$$

In *MixoneColumns(state,c)* function each cell of the column is replaced complying with the result of the previous multiplication as it follows:

$$S'_{0,c} = (\{02\} \bullet S_{0,c}) \oplus (\{03\} \bullet S_{1,c}) \oplus S_{2,c} \oplus S_{3,c} \quad (3)$$

$$S'_{1,c} = S_{0,c} \oplus (\{02\} \bullet S_{1,c}) \oplus (\{03\} \bullet S_{2,c}) \oplus S_{3,c} \quad (4)$$

$$S'_{2,c} = S_{0,c} \oplus S_{1,c} \oplus (\{02\} \bullet S_{2,c}) \oplus (\{03\} \bullet S_{3,c}) \quad (5)$$

$$S'_{3,c} = (\{03\} \bullet S_{0,c}) \oplus S_{1,c} \oplus S_{2,c} \oplus (\{02\} \bullet S_{3,c}) \quad (6)$$

I defined the function *xtime()* that implements the multiplication by x (that is the multiplication by {02}) with a byte-oriented left shift and a conditional bitwise XOR with {1b}. So i performed the previous equations using the *xtime()* function. To compute the multiplication of $S_{r,c}$ by {03} i applied the *xtime*($S_{r,c}$) and then XORed the result with $S_{r,c}$.

2.1.4 Key Addition layer

The *AddRoundKey* transformation is realized with a bitwise XOR operation between the State and the Word matrix which is the one obtained from the Key Expansion.

2.1.5 Key Expansion

Key Expansion is needed to create the key schedule. Starting from the original input key it derives recursively subkeys. It generates Nb (Nr + 1) words of 4 bytes, where Nr is the number of rounds in the algorithm. *KeyExpansion()* function returns a matrix starting from the 4 initial words of the input key and deriving the following words by applying some transformations to the previous ones.
To implement the *KeyExpansion()* i defined the function *rotWord* which consists in a left shift of the 4 bytes in a word, and *subWord* wich applies the Sbox to the bytes of the word. Moreover i defined *Rcon* as an array that can be used as a look-up table in the same way that the Sbox is used.
The *KeyExpansion()* function implementation follows the pseudo-code below:

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Figure 3: Key expansion pseudo-code [2]

2.2 Decryption

The Decryption function takes as input an array of int values representing the ciphertext and the word derived from the key expansion. To implement the decryption i defined all the inverse function of the ones used in the cipher algorithm. This function works in the same way as the encryption one does, but the order in which the inverse operations are applied is different. In a single round the sequence of the transformation is:

- *InvShiftRows*
- *InvSubBytes*
- *AddRoundKey*
- *InvMixColumns*

As before, in the last round *InvMixColumns* is skipped. There's no need to implement an inverse function of the *AddRoundKey* because, implementing a XOR operation, it happens to be its own inverse.

2.2.1 Inverse Shift Rows Sublayer

The *InvShiftRows(state)* function it's equivalent to the *ShiftRows(state)* but the shift is applied in the opposite verse.

$$S'[r][(c + shift(r, Nb))modNb] = S'[r][c] \tag{7}$$

As its inverse it's realized by assigning to a new state matrix the right values following the previous rule.

2.2.2 Inverse Byte Substitution Layer

This layers performs a substitution at a byte level taking the new values from a table called Inv S-Box. The Inv S-Box is used as a look-up table and it is obtained by applying the inverse operation of the ones used to create the S-Box. For each element in row r and column c in the State, the transformation is:

$$state[r][c] = InvSbox[state[r][c]] \tag{8}$$

2.2.3 Inverse Mix Columns Sublayer

Likewise the encryption algorithm, this operation is realized by two functions: *InvMixColumns(state)* and *InvMixoneColumns(state,c)*. The first one only iterates on the columns and calls the *InvMixoneColumns(state,c)* function. Each column is considered, as before, a polynomial in GF(2^8).

$$\begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{pmatrix}$$

InvMixoneColumns(state,c) function assigns at each cell of the column the value obtained from the result of the previous multiplication as it follows:

$$S'_{0,c} = (\{0e\} \bullet S_{0,c}) \oplus (\{0b\} \bullet S_{1,c}) \oplus (\{0d\} \bullet S_{2,c}) \oplus (\{09\} \bullet S_{3,c}) \tag{9}$$

$$S'_{1,c} = (\{09\} \bullet S_{0,c}) \oplus (\{0e\} \bullet S_{1,c}) \oplus (\{0b\} \bullet S_{2,c}) \oplus (\{0d\} \bullet S_{3,c}) \tag{10}$$

$$S'_{2,c} = (\{0d\} \bullet S_{0,c}) \oplus (\{09\} \bullet S_{1,c}) \oplus (\{0e\} \bullet S_{2,c}) \oplus (\{0b\} \bullet S_{3,c}) \tag{11}$$

$$S'_{3,c} = (\{0b\} \bullet S_{0,c}) \oplus (\{0d\} \bullet S_{1,c}) \oplus (\{09\} \bullet S_{2,c}) \oplus (\{0e\} \bullet S_{3,c}) \tag{12}$$

I performed the multiplication using the *xtime()* function.

- $\{09\} \bullet a = xtime(xtime(xtime(a))) \oplus a$
- $\{0b\} \bullet a = xtime(xtime(xtime(a))) \oplus xtime(a) \oplus a$

- $\{0d\} \bullet a = xtime(xtime(xtime(a))) \oplus xtime(xtime(a)) \oplus a$
- $\{0e\} \bullet a = xtime(xtime(xtime(a))) \oplus xtime(xtime(a)) \oplus xtime(a)$

Because the *xtime()* function is called several times to compute the previous multiplications, in the *InvMixOneColumn* i pre-compute the values of the *xtime()* results that i need and then i applied the respective XOR operations to this values in order to optimize the function.

3 Modes of operations

A mode of operation is an algorithm that uses a block cipher and outlines the way in which this block cipher should be applied to the single blocks composing a larger input.

3.1 ECB: Electronic Code Book

ECB is the simplest operation mode and it encrypts each block individually, every block is independent from the previous and from the following one. This independence allows this mode of operation to be parallelized.

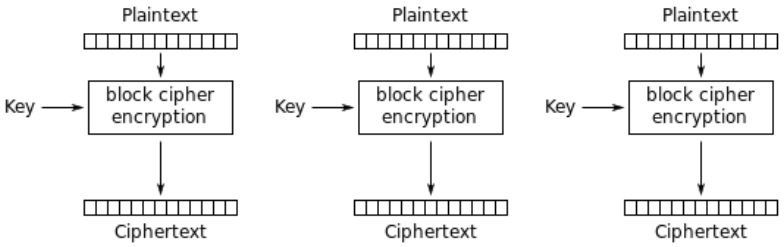


Figure 4: ECB: encryption [3]

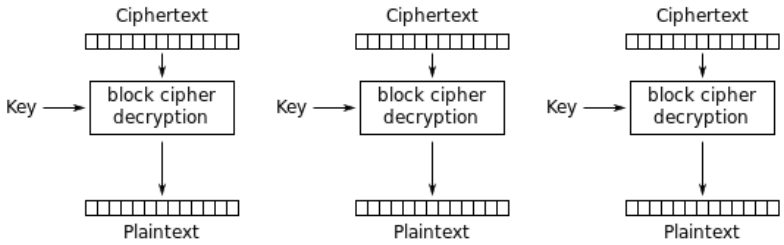


Figure 5: ECB: decryption [3]

This mode of operation requires padding, i followed **PKCS7** specification that imposes that the value of each added byte is equal to the number of

bytes that are added. In the *ecb* function padding is added to let all the blocks have the same size so that each block can be the input of the *encryption* function. Regarding the *inv_ecb* function, after applying the *decryption* function, padding is removed by searching for a sequence of characters equals to the last one from one index i and the end of the output string.

3.2 CBC: Cipher block chaining

In CBC the XOR between the first block and an initialization vector compose the first input of the *encryption* function, from the second block on the plaintext is XORed with the ciphertext of the previous block and then the *encryption* function is applied to this result. Differently from ECB the encryption is not parallelizable.

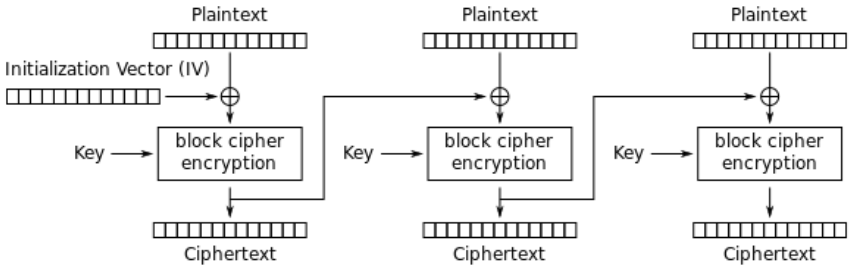


Figure 6: CBC: encryption [3]

For the inverse cipher the *decryption* function is applied to the first block and this output is XORed with the IV. For the following blocks the output of the *decryption* function is XORed with the ciphertext of the previous block. When the decryption starts the whole ciphertext is available so this process can be parallelized.

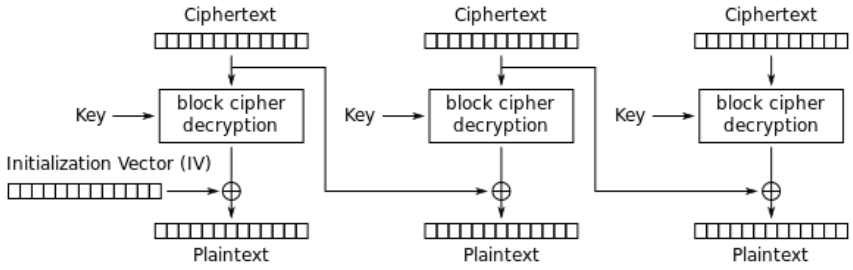


Figure 7: CBC: decryption [3]

The padding is used as in ECB mode. Having used an array of integer as input of the *encryption* function that i realized, i decided to define the initialization vector as an array of int values as well.

3.3 CFB: Cipher feedback mode

CFB mode of operation starts with the encryption of the initialization vector. The output of the *encryption* function is XORed with the plaintext of the first block. From the second block on, the input of the *encryption* is the ciphertext obtained from the previous block. Each step of the encryption needs the previous ciphertext, so this process is not parallelizable.

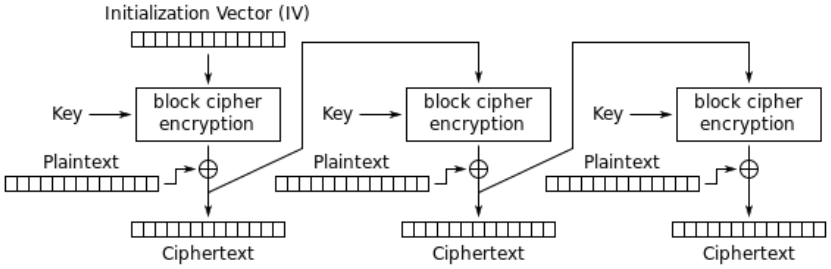


Figure 8: CFB: encryption [3]

In CFB there is not a *decryption* function, in fact the *encryption* is applied to the IV and its output is XORed with the ciphertext in order to obtain the plaintext. For the following blocks the input of the *encryption* function is the ciphertext of the previous block. As in CBC decryption is parallelizable because the ciphertext is always available.

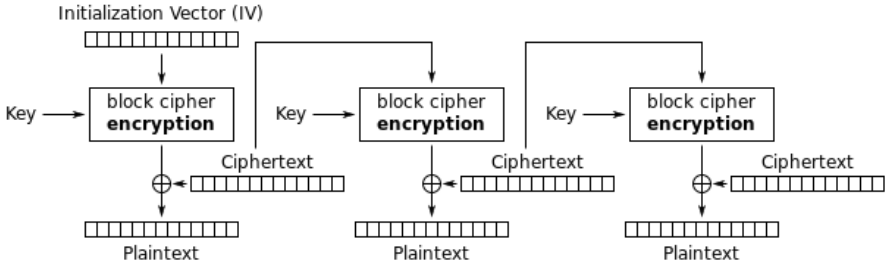


Figure 9: CFB: decryption [3]

In this mode of operation there is no need of padding because the length of the blocks that are going to be encrypted is always the same. What changes is the length of the last plaintext block, but there's not a fixed length to apply the XOR operation, so the padding is useless.

3.4 OFB: Output feedback mode

Output feedback mode lets a block cipher become a stream cipher. The first block to be encrypted is the initialization Vector, and then cyclically the output of the *encryption* function serves as the input of the following encryption block.

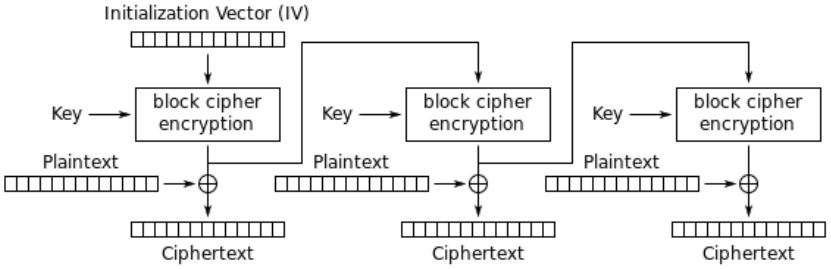


Figure 10: OFB: encryption [3]

Thanks to the XOR operator and its symmetry the encryption and the decryption behave exactly the same.

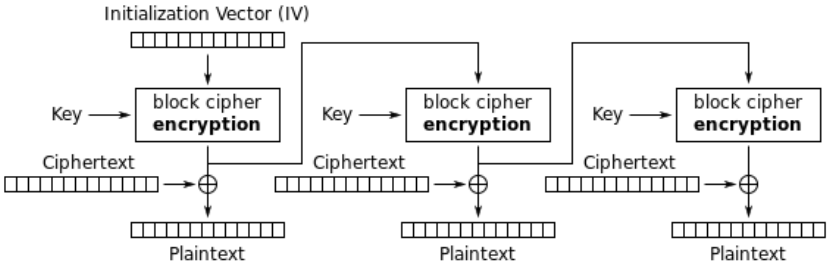


Figure 11: OFB: decryption [3]

Both encryption and decryption are not parallelizable. For the same reasons explained in CFB there's no need of padding.

3.5 CTR: Counter mode

Counter mode introduces the concepts of nonce and counter. The nonce could be seen as an initialization vector and the counter can be any function that can be incremented. Every block is encrypted separately but each one has a different value of the counter, that increases by one for every following block. This independence gives this mode of operation the possibility to be parallelized both in encryption and in decryption. The input of the *encryption* function is the concatenation of the nonce and the counter and then the output is XORed with the plaintext to obtain the ciphertext. There's no need of padding such as in CFB and in OFB modes.

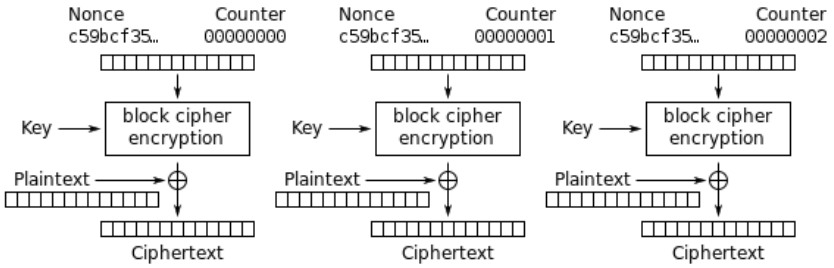


Figure 12: CTR: encryption [3]

Such as OFB mode of operation, CTR makes a block cipher into a stream cipher. As before, due to the XOR properties, Encryption and Decryption works in the same way.

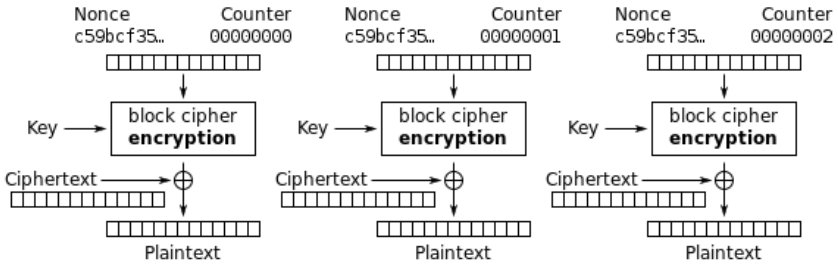


Figure 13: CTR: decryption [3]

Using a block of 128 bits i decided to use both a nonce and a counter of 64 bits. I implemented the nonce as an array of int values for consistency with the input of the *encrypt* function.
The counter is an array that starts with the value of [0,0,0,0,0,0,0,0] and i defined a function *incr_counter(counter)* that increments the counter by one every time that it's called.

4 Experimental analysis

To do an experimental analysis of the implementation of AES that i realized i used three different files with different sizes: 1KB, 100KB and 10MB.

4.1 Timing table

First of all i profiled the performances of my implementation by timing each one of the five modes of operations with the three different files.

1KB FILE

MODE	encryption running time	decryption running time
ECB	0.0291769504547 sec	0.0302331447601 sec
CBC	0.0340280532837 sec	0.0299999713898 sec
CFB	0.0292088985443 sec	0.0239210128784 sec
OFB	0.0360479354858 sec	0.0218260288239 sec
CTR	0.0296838283539 sec	0.0237491130829 sec

100KB FILE

MODE	encryption running time	decryption running time
ECB	2.28378415108 sec	2.99603414536 sec
CBC	2.23400402069 sec	2.98720002174 sec
CFB	2.23374199867 sec	2.38853096962 sec
OFB	2.24022102356 sec	2.33336806297 sec
CTR	2.28250312805 sec	2.3409268856 sec

10MB FILE

MODE	encryption running time	decryption running time
ECB	247.580243111 sec	343.740667105 sec
CBC	260.572172165 sec	297.253606081 sec
CFB	224.638135195 sec	225.076455116 sec
OFB	223.425223827 sec	221.951869011 sec
CTR	240.475089073 sec	238.360127211 sec

It's possible to observe that the *decryption* function is slower than the encryption one, in fact CFB, OFB and CTR modes have a 'faster' decryption running time because they use the *encryption* function even in the decryption phase.
We could make the algorithm faster introducing a parallel implementation where it is possible. ECB and CTR could be more efficient because they're parallelizable both in encryption and in decryption. Moreover we could parallelize the decryption phase in CBC and in CFB.

4.2 Comparison with the performances of a real-world implementation

I tested the performances of the real-world AES implementation made available by *PyCryptodome* Python package.
The ciphertexts generated by this implementation are the same obtained by the implementation that i realized.

4.2.1 Timing table of the real-world implementation

Performances of *PyCryptodome* AES implementation on the same 3 files.

1KB FILE

MODE	encryption running time	decryption running time
ECB	0.000196933746338 sec	3.38554382324e-05 sec
CBC	0.000375986099243 sec	3.09944152832e-05 sec
CFB	0.000371932983398 sec	4.50611114502e-05 sec
OFB	0.000159025192261 sec	2.19345092773e-05 sec
CTR	0.000197172164917 sec	5.50746917725e-05 sec

100KB FILE

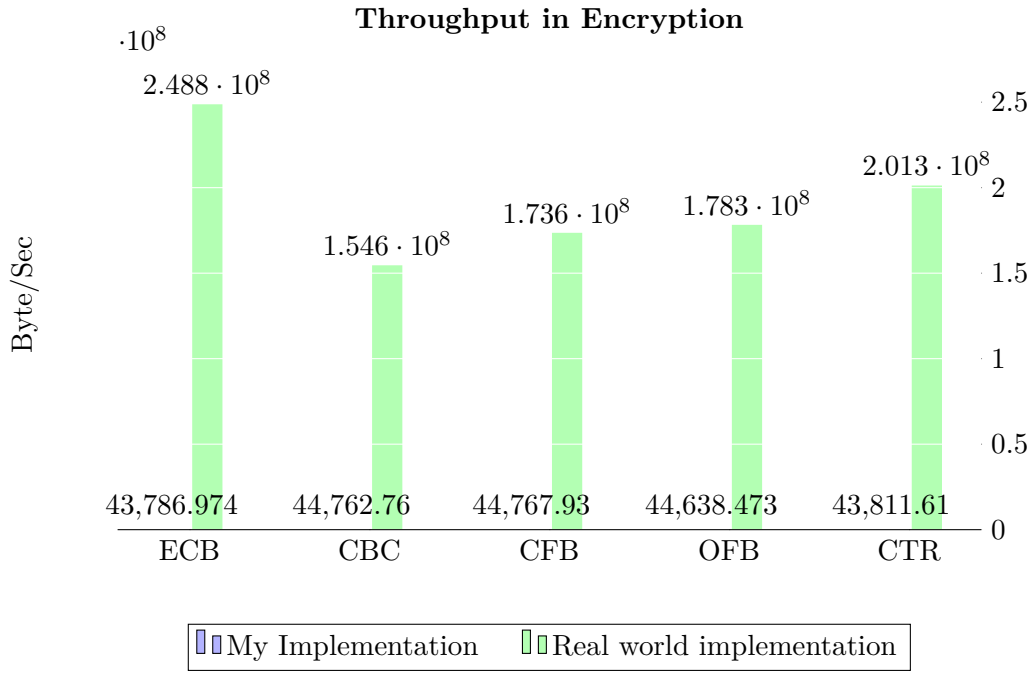
MODE	encryption running time	decryption running time
ECB	0.000401973724365 sec	0.000361919403076 sec
CBC	0.000646829605103 sec	0.00047492980957 sec
CFB	0.000576019287109 sec	0.000514030456543 sec
OFB	0.000560998916626 sec	0.000377893447876 sec
CTR	0.000496864318848 sec	0.000270843505859 sec

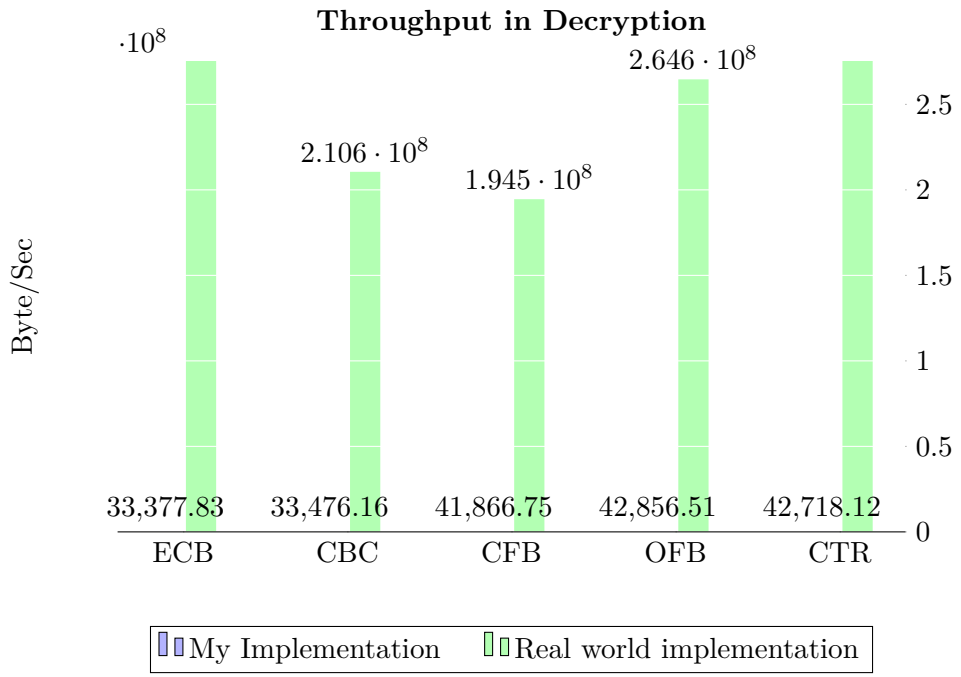
10MB FILE

MODE	encryption running time	decryption running time
ECB	0.0277900695801 sec	0.0144109725952 sec
CBC	0.041002035141 sec	0.0384941101074 sec
CFB	0.04696393013 sec	0.0429830551147 sec
OFB	0.0428531169891 sec	0.0438158512115 sec
CTR	0.0210070610046 sec	0.0239140987396 sec

4.2.2 Throughput comparison

The two implementation are compared using the throughput as the metric of the comparison.
I used the 100KB file to compute the throughput.





There should have been a blue bar representing my implementation,but it's not possible to see it because the performances of the AES i realized are very far from the real one. So the difference from the two values it's too big to represent the two plots in the same scale and make the blue one visible. The differences in the performaces of the two implementations could be seen even from the tables of the times shown in the previous paragraph.

References

[1] Understanding Cryptography by Christof Paar and Jan Pelzi Chapter 4

[2] Federal Information Processing Standards Publication 197. Announcing the Advanced Encryption Standard. November 26, 2001

[3] Wikipedia: Block cipher mode of operation