INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Cryptography

Hill Cipher OM

Abstract

Processing images BMP in C language for encrypting the data they have, pixel by pixel implementing 4 different modes of operations (ECB, CBC, OFB and CFB) using matrix operation and bit – level XOR, analyzing the structure of the header and the structure of an image and a pixel composed by 3 different colors (Blue, Red and Green).

By:

Romero Gamarra Joel Mauricio

Professor:
MSc. NIDIA ASUNCIÓN CORTEZ DUARTE

October 2017

Index

Content

Introduction:

Hill is a block cipher, it means it could encrypt blocks of several letters (or in this case, pixels). A change of one character in the plaintext, change potentially all the characters in the corresponding ciphered text block. Hill Cipher was invented in 1929 by Lester Hill, using for the first-time algebraic methods in cryptography.[1]

First, we need to understand that, this cipher use matrix operations. The following formula represents HILL Cipher (Encryption):

$$C = p \cdot K \bmod n$$

Where:

- C = Ciphered text
- p = Plaintext
- K = Key
- n = Size of the alphabet

Hill, is just a matrix multiplication, depending on what we want to do, that's why K is the encryption key and K^{-1} is the decryption key, finally, we apply modular division to have the result between 0 to (n – 1).

In this case, we are encrypting and decrypting images (formed by pixels), and we know that a pixel is formed by 3 elements: Blue, Green and Red. These, can be between 0 to 255, so, our alphabet is 256.

Literature review:

I will start explaining each of the Modes of Operation implemented in this practice, the first one is **Electronic Codebook (ECB)** which is represented on Figure 1.

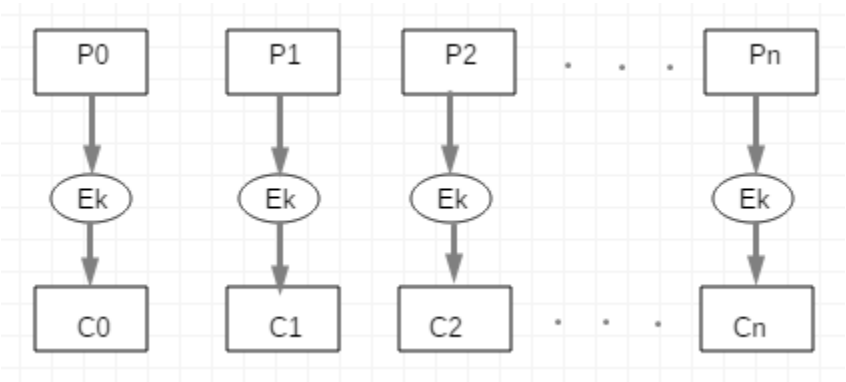


Figure 1. Electronic Codebook Diagram (Encryption)

From Figure 1, we can note the encryption process and then, we can obtain the formula:

$C_0 = E_k (P_0)$ $C_1 = E_k (P_1)$ $C_2 = E_k (P_2)$ … $\mathbf{C_n = E_k (P_n)}$

Now, to decrypt the image, we can apply "D_k" both on the left and right sides of the formula:

$C_n = E_k (P_n) \longrightarrow D_k (C_n) = D_k (E_k (P_n)) \longrightarrow \mathbf{P_n = D_k (C_n)}$

Where:

- P_i = Plaintext = Pixels of the original image
- C_i = Ciphers = Pixels of the modified image
- E_k = Encryption Key (Matrix K)
- D_k = Decryption Key (Matrix K^{-1})

Figure 2, shows the decryption process for **Electronic Codebook Mode (ECB)**.

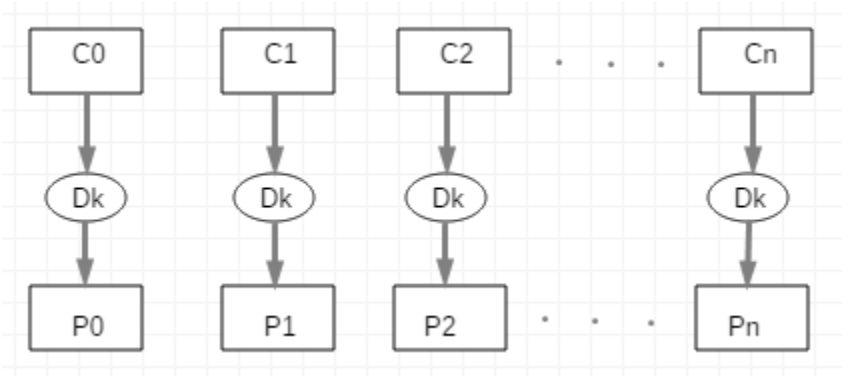


Figure 2. Electronic Codebook Diagram (Decryption)

Something interest of this mode of operation is, it is not safe, we can verify this by looking at Figure 1 and Figure 2, for example, if a Cipher (C_i) is damaged, the others are not, in an image, just a pixel won't affect too much, because you can appreciate almost all the image in good conditions. If you want to encrypt a message, just a letter will be affected, and the others not. That's why, this mode of operation is not common neither in images, nor in messages. Computationally, we can parallelize both processes, encryption and decryption because we don't need to wait for the previous calculation of the pixel to calculate the next.

Now, I continue explaining another Mode of Operation, **Cipher Block Chaining (CBC)**, this mode of operation, must has an initialization vector (IV), also called C_0 to start encrypting the information block by block. Also, we add a bit – level operation to encrypt the data, a XOR, converting the value of each BGR value (pixel) to its binary representation, and applying it.[1]

The result we obtain with XOR, enter to Hill function to multiply by the encryption key and finally obtaining the cipher, then, use the cipher to make the next XOR with the next pixel we reed from the original image like you can see at Figure 3:

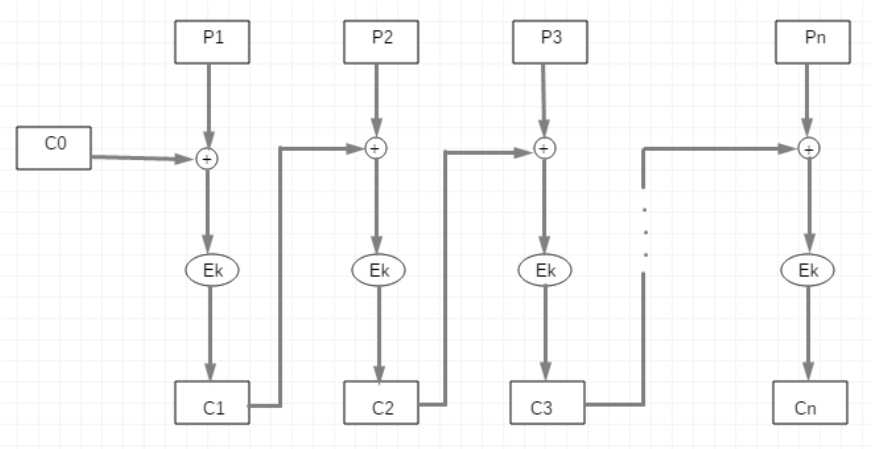


Figure 3. Cipher Block Chaining Diagram (Encryption)

From Figure 3, we can note the encryption process and then, we can obtain the formula:

$C_1 = E_k (P_1 \oplus C_0)$ $C_2 = E_k (P_2 \oplus C_1)$ $C_3 = E_k (P_3 \oplus C_2)$ … $\mathbf{C_n = E_k (P_n \oplus C_{n-1})}$

Now, for the opposite process, we need to clear $\mathbf{P_i}$**:**

$C_n = E_k (P_n \oplus C_{n-1}) \longrightarrow D_k (C_n) = D_k (E_k (P_n \oplus C_{n-1})) \longrightarrow D_k (C_n) \oplus C_{n-1} = P_n \oplus C_{n-1} \oplus C_{n-1}$

Finally:

$$\mathbf{P_n = D_k (C_n) \oplus C_{n-1}}$$

Where:

- $\oplus$ = XOR

Figure 4, shows the decryption process for **Cipher Block Chaining Mode (CBC)** according to the formula we obtained mathematically.

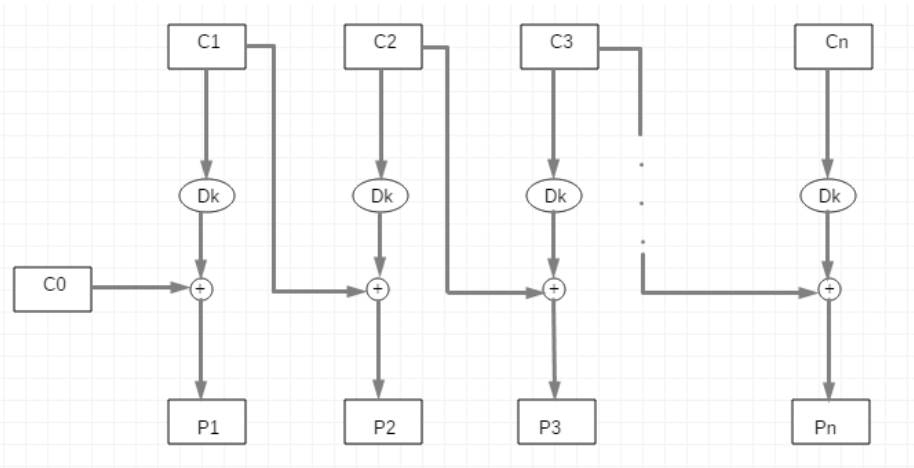


Figure 4. Cipher Block Chaining Diagram (Decryption)

In CBC Mode, we can appreciate in Figure 3 (Encryption Process), that we can't parallelize because, to obtain the next cipher, we use the previous one, that's why, we need to calculate the ciphered image pixel by pixel.

Otherwise, in decryption process if we have all the ciphered pixels, we can parallelize the process to make it faster, because, to obtain the next original pixel, we just need the cipher (and we have all of them), not the previous original pixel, that's why for this process we could use threads to increase the velocity of the image processing.

The next Mode of Operation I will talk about, is called **Output Feedback Mode (OFB)**. As the previous mode (CBC), this mode of operation uses an Initialization Vector we will call C_0 to start encrypting / decrypting the image introduced by the user. Also, it needs a bit – level operation (XOR), Figure 5 shows in detail in which vectors we need to apply XOR Operation.

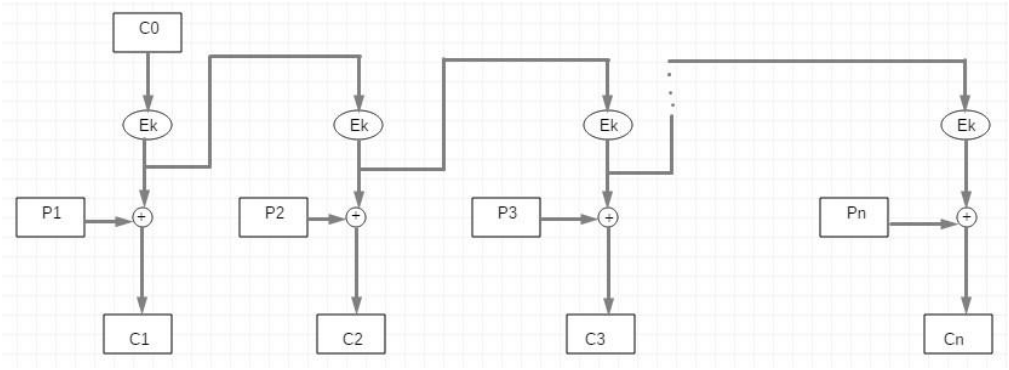


Figure 5. Output Feedback Diagram (Encryption)

From Figure 5, we can note the encryption process and then, we can obtain the formula:

$C_1 = E_k(C_0) \oplus P_1$ $\quad C_2 = E_k(E_k(C_0)) \oplus P_2$ $\quad C_3 = E_k(E_k(E_k(C_0))) \oplus P_3$

Finally:

$$\mathbf{C_n = E_k^n (C_0) \oplus P_n}$$

Now, for the opposite process, we need to clear $\mathbf{P_i}$**:**

$C_n = E_k^n(C_0) \oplus P_n \longrightarrow C_n \oplus E_k^n(C_0) = E_k^n(C_0) \oplus P_n \oplus E_k^n(C_0)$

$\longrightarrow C_n \oplus E_k^n(C_0) = P_n$

Finally:

$$\mathbf{P_n = C_n \oplus E_k^n (C_0)}$$

Now, we obtain the decryption process mathematically, is important to observe that for **decryption**, we don't use $\mathbf{D_k}$, we use $\mathbf{E_k}$ again and always applied n times (number of pixels) to C_0 (The initialization vector introduced by the user). This mode of operation is not safe and I will explain why in a little while. But first, let's see on Figure 6 the decryption process for OFB Mode.

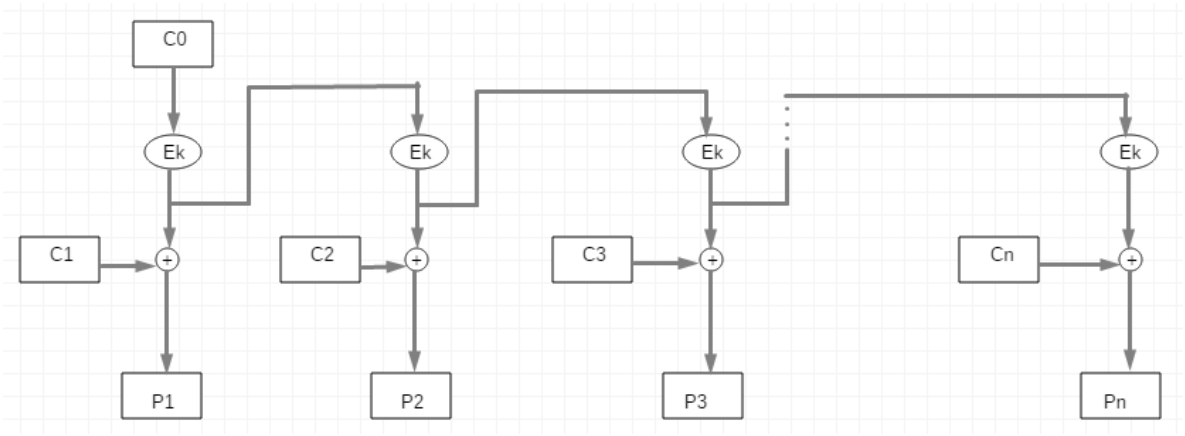


Figure 6. Output Feedback Diagram (Decryption)

The last mode of operation implemented in this practice, is called **Cipher Feedback Mode (CFB)**, this cipher needs an initialization vector to start the processes such as the others (except for Electronic Codebook), the bit – level XOR Operation is presented again to encrypt and decrypt the images.

In figure 7, the decryption process of **CFB** is showed.

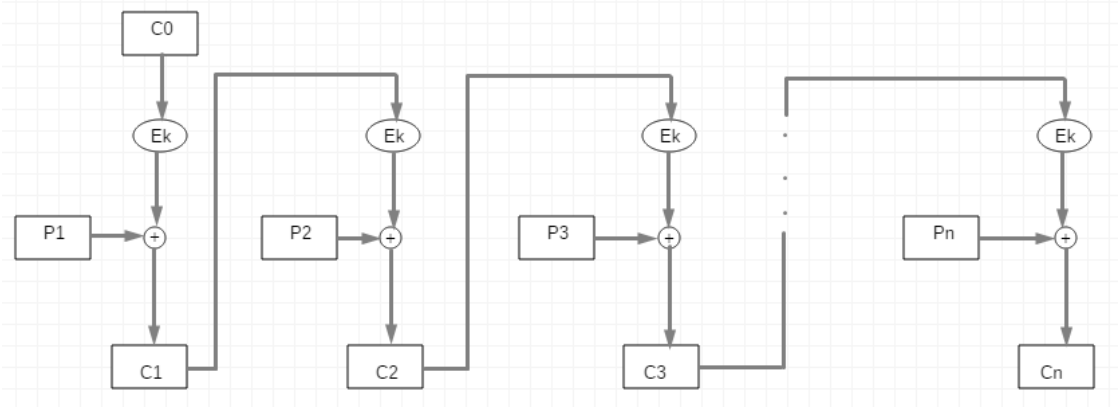


Figure 7. Cipher Feedback Diagram (Encryption)

From Figure 7, we can note the encryption process and then, we can obtain the formula:

$C_1 = E_k(C_0) \oplus P_1 \qquad C_2 = E_k(C_1) \oplus P_2 \qquad C_3 = E_k(C_2) \oplus P_3 \qquad \ldots \qquad \mathbf{C_n = E_k(C_{n-1}) \oplus P_n}$

Now, for the opposite process, we need to clear $\mathbf{P_i}$**:**

$C_n = E_k(C_{n-1}) \oplus P_n \longrightarrow C_n \oplus E_k(C_{n-1}) = E_k(C_{n-1}) \oplus P_n \oplus E_k(C_{n-1}) \longrightarrow C_n \oplus E_k(C_{n-1}) = P_n$

Finally:

$$\mathbf{P_n = C_n \oplus E_k(C_{n-1})}$$

As you can see, this time we don't use the Decryption Matrix again (just as OFB). Figure 8 shows the decryption process we obtained previously Mathematically.

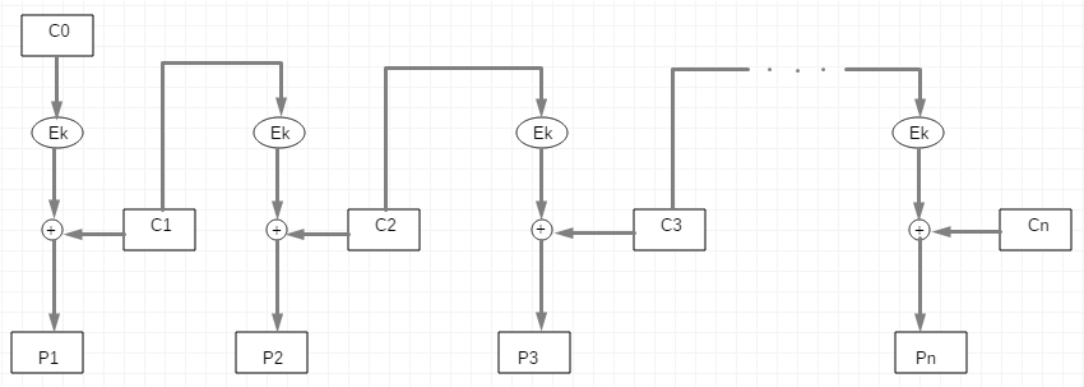


Figure 8. Cipher Feedback Diagram (Decryption)

After we know now some modes of operation for encrypting and decrypting image, we can discuss which one is better than other, or safer, etc.

- **Electronic Codebook**
  - If the image has a lot of colors, will be encrypted well, but 1 color
  - It is the simplest mode of operation using only matrix multiplication
  - It is the easiest mode of operation to implement
  - We can parallelize both encryption and decryption processes and make them faster
  - If a cipher came with an error, just that pixel will be affected

- **Cipher Block Chaining**
  - Encryption can't be parallelized, but Decryption
  - If a cipher came with an error, that pixel and the next will be affected, for 2 non – consecutive pixels with an error, 4 pixels will be affected, and so on
  - Even if it is an image with 1 color, or with a lot of colors, will be encrypted well
  - Bit – level operation (XOR) increase the security of the processes
  - Even if the Initialization Vector is wrong, if the other one knows the key, it will be decrypted correctly

- **Output Feedback**
  - If the initialization vector is wrong, all the pixels will be affected
  - If a cipher or a pixel came with an error, just that pixel or cipher will be affected
  - If we encrypt the image, and then encrypt the encrypted image, will give us the original one.
    Demonstration:

    We know that: $\mathbf{P_n = C_n \oplus E_k^n (C_0)}$ and $\mathbf{C_n = E_k^n (C_0) \oplus P_n}$

    Substituting P_n on Encryption Function: $\mathbf{C_n = E_k^n (C_0) \oplus C_n \oplus E_k^n (C_0)}$

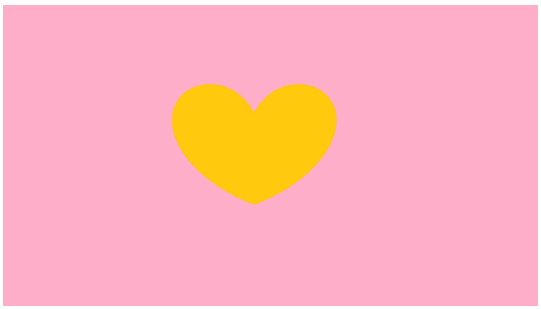
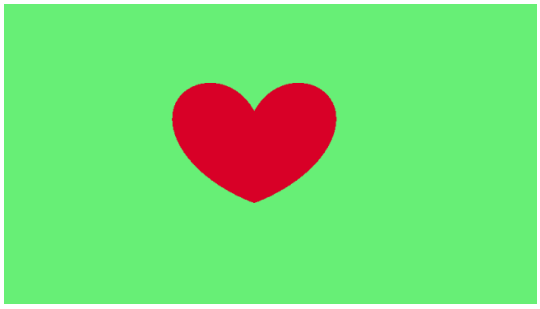
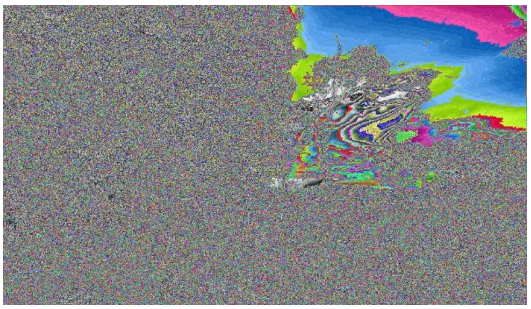
    Finally: $\mathbf{C_n = C_n}$

- **Cipher Feedback**
    - We can’t parallelize encryption process but encryption
    - If the initialization vector is wrong, just the first pixel will be afected
    - If a cipher cames with an error, 2 pixels of the image will be afected
    - It encrypt well even if the image has a lot of colors or not

Now that I explain some of the advantages and disadvantages, you can choose the cipher that suits you.

In order to give you a better idea, that makes you easier to choose one of these modes of operation, in Table 1, I present you some examples of how these ciphers work with 2 different kind of images.

The left side has as a plaintext, a very simple image with just 2 colors, and on right side, we have a complete landscape and see how they work with these images.

	Heart	Landscape
Plaintext		
Electronic Codebook		

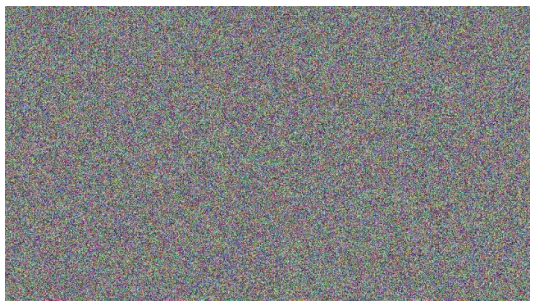
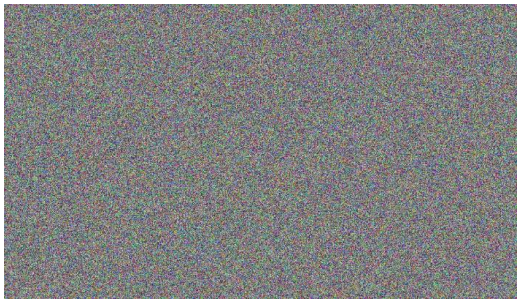
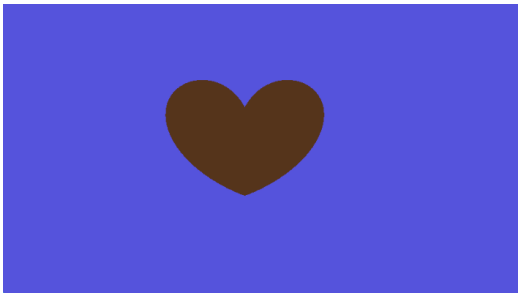
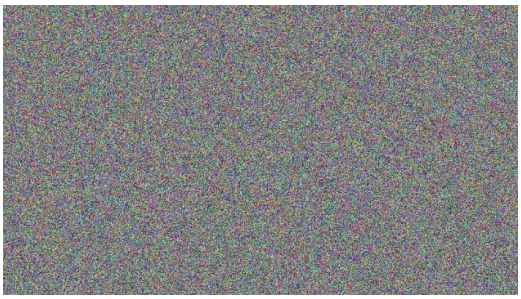
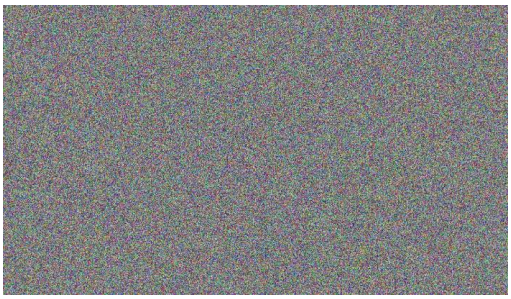
Cipher Block Chaining		
Output Feedback		
Cipher Feedback		

Table 1. Comparation of implemented Modes of Operation in 2 images

As you can see on Table 1, the safest ciphers are CBC and CFB.

- ECB, change one color, to another, so, if you have an image with just 1, 2 or few colors, the encrypted image will be very simple to know what is it.
- ECB functions well with an image with a lot of colors as we can see on the image of the landscape, because pixel by pixel the color changes drastically and just a number on a color (Blue, Green or Red), changes a lot on the encrypted image.
- CBC works very well with any kind of images, it doesn't matter if the image has a lot or a few colors, the encrypted image will be practically unreadable.
- OFB is the most insecure of all of these ciphers, we can appreciate that on Figure 5 because we are applying E_k just to the initialization vector and the original pixels won't be changed at any part.
- CFB, such as CBC, works very well with any kind of image because the image will be because we are encrypting the previous cipher.

Software (libraries, packages, tools):

Libraries:[2]

- Stdio.h: Used for the following functions:
    - int printf (const char * format, ...)
    - int scanf (const char * format, ...)
    - FILE * fopen (const char * filename, const char * mode)
    - size_t fread (void * ptr, size_t size, size_t nmemb, FILE * stream)
    - size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE * stream)
    - int fclose (FILE * stream)
- Stdlib.h: Used for the following functions:
    - void *malloc (size_t size)
    - void free (void * ptr)
    - void exit (int status)
    - int system (const char * string)
- String.h: Used for the following functions:
    - void * memset (void * str, int c, size_t n)
- Functions.h (Own): Used the following functions, making use of the above functions:
    - FILE * open_file (char * original, char * encrypted, int tipo)
    - void read_head (FILE * original, FILE * encrypted, bmp * image)
    - void hill (unsigned char * BGR, unsigned char * pixel, char option)
    - void operation_mode (FILE * original, FILE * encrypted, bmp * image, char option)
    - void print_head (bmp * image)
    - char * message (char option)
    - void ECB (FILE * original, FILE * encrypted, bmp * image, char option)
    - void CBC (FILE * original, FILE * encrypted, bmp * image, char option)
    - void CFB (FILE * original, FILE * encrypted, bmp * image, char option)
    - void OFB (FILE * original, FILE * encrypted, bmp * image, char option)

Tools:

- Star UML
- Sublime Text 3
- Paint

Procedure:

First of all, we need to analyze the structure of a BMP Image.

BMP (Windows Bitmap) is the simplest format that an image could has, it consists on a header followed by the real data (pixels), each pixel is formed by 3 colors (in this case), that are Blue, Red and Green on that order from the image.[3]

One of the most interesting advantages of this kind of image is its easy manipulation in C language and its simple composition, just a header and 3 colors (in this case), but the disadvantage is the big size of the images because they have no compression.

The following, is the structure used to store all the information of the bitmap in C language.[4]

```
char type [2];              //(2 Bytes) It contains the characters 'BM'
int file_size;              //(4 Bytes) It contains the size of the entire file
int reserved;               //(4 Bytes) It contains reserved bytes
int offset;                 //(4 Bytes) It contains the offset from the beginning

//BMP Information
int bitmap_size;            //(4 Bytes) It contains the size of the bitmap
int width;                  //(4 Bytes) Width (Horizontal pixels)
int height;                 //(4 Bytes) Height (Vertical pixels)
short no_planes;            //(2 Bytes) Number of planes of the image
short bits_per_pixel;       //(2 Bytes) Quantity of bits per pixel
int compression;            //(4 Bytes) It contains 0 if it's not compressed
int image_size;             //(4 Bytes) It contains the size of the image
int horizontal_res;         //(4 Bytes) It contains the horizontal resolution
int vertical_res;           //(4 Bytes) It contains the vertical resolution
int no_colors;              //(4 Bytes) It contains the number of colors
int important_colors;       //(4 Bytes) It contains the number of important colors
```

After we know the composition of the header of a bitmap image, we need to start reading all the other information on it, it has pixels (as we mention before, composed by 3 colors with values between 0 and 255).

It's important to mention that bytes are organized by the less significative to the most significative, and the image is inverted, in other words, the first line is the last of the image and vice versa with a length of a multiple of 32, the program add the missing bytes to complete until complete a multiple of 32 with 0. On Figure 9, it's showed you an example structure of an image.

42	4D	D6	00	00	00	00	00	00	00	36	00	00	00	28	00
00	00	0A	00	00	00	05	00	00	00	01	00	18	00	00	00
00	00	A0	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
FF	7F	7F	7F	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	7F	7F	7F	00	00	FF	00	00	FF	00	00	FF	00	00	FF
00	00	FF	00	00	FF	00	00	FF	00	00	FF	00	00	FF	00
00	7F	7F	7F	00	00	00	FF	00	00	FF	00	00	FF	00	00
FF	00	00	FF	00	00	FF	00	00	FF	00	00	FF	00	00	FF
00	7F	7F	7F	00	00	00	00	FF	00	00	FF	00	00	FF	00
00	FF	00	00	FF	00	00	FF	00	00	FF	00	00	FF	00	00
FF	7F	7F	7F	00	00										

Figure 9. Structure of an example image

Results

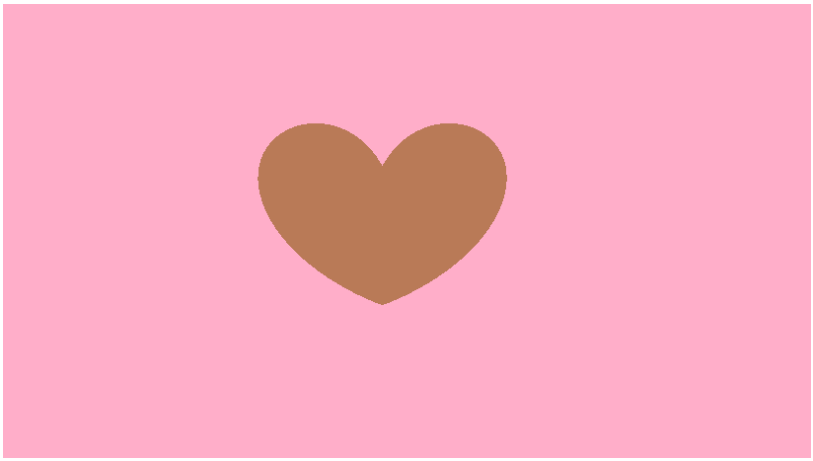


Figure 10. Original Image (Heart.bmp)

In 10, we observe the original image (similar to the one in table 1), in Figure 11, we compile and execute the program passing the name of the images as parameters in the execution.

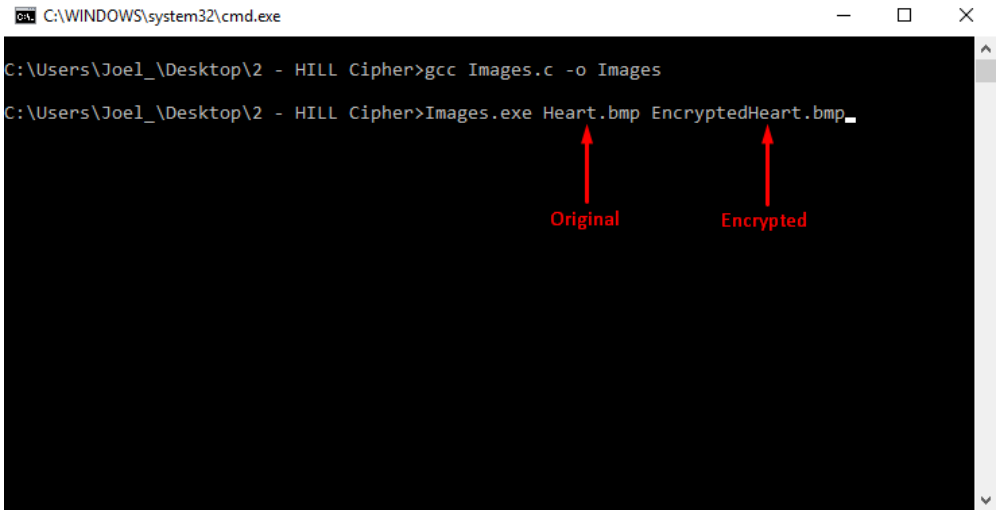


Figure 11. Example of compilation and execution of the program for encrypting process

After we execute the program, Figure 12 shows the first menu to select the option (Encrypt or Decrypt).

Then, it shows a message to the user if the first image (the original one) could open correctly and if the second image (the encrypted or decrypted) could be created correctly. After this, it shows to the user a second menu asking which one of the modes of operation we want to use (according to the option selected).

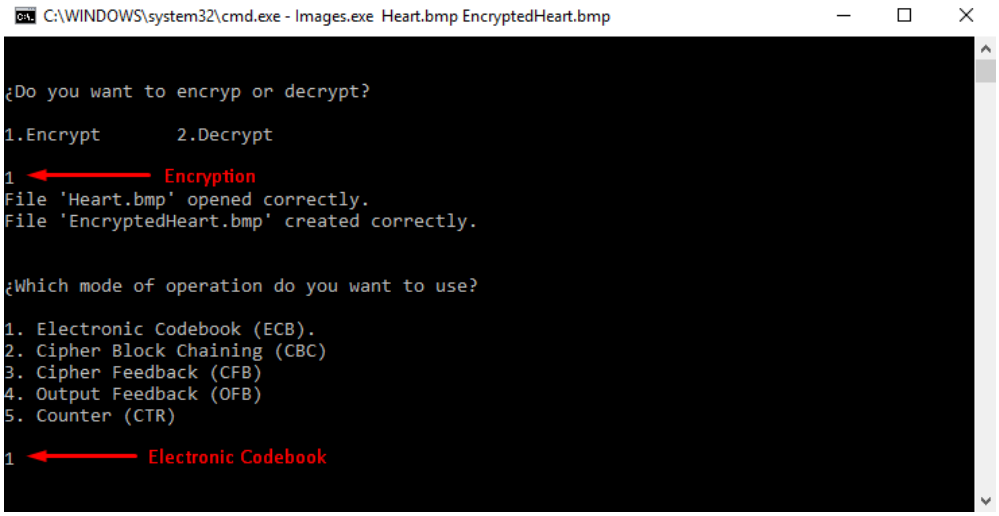


Figure 12. Example of the first look of the execution of the program

In Figure 12, we can see that we select encryption process of the ECB Mode of Operation. We can see the values of the header in Figure 13 to know that everything was correct.

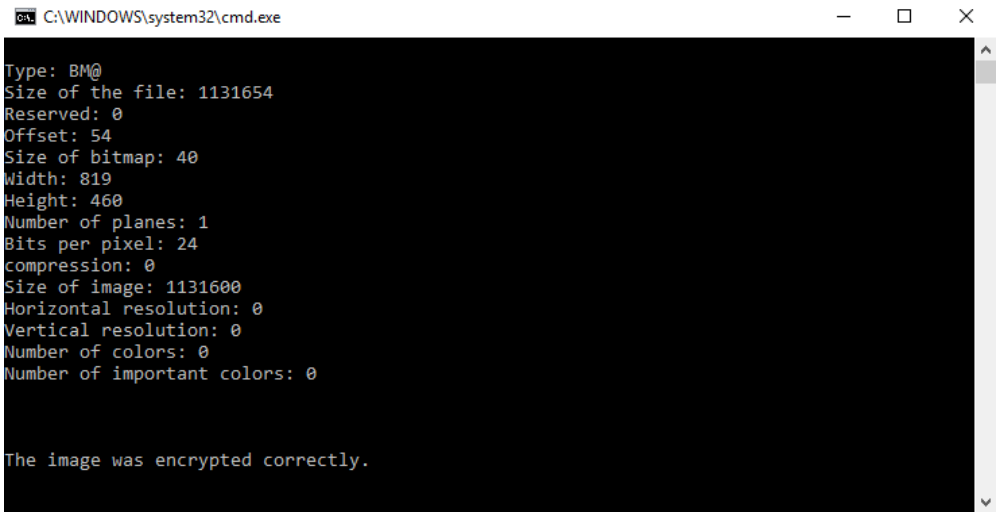


Figure 13. Header of the image (Heart.bmp)

Finally, we check the encrypted image on Figure 14 with Electronic Codebook Mode.

Figure 14. Encrypted Image with ECB Mode (EncryptedHeart.bmp)

As we can see on Figure 14, the Encrypted Image just change the colors of the image, but is too visible of what it is. Then, on Figure 15 we show the example of execution for decrypting the image, now the second parameter (original) will be the encrypted image and the third parameter (encrypted / decrypted) will be the decrypted image.

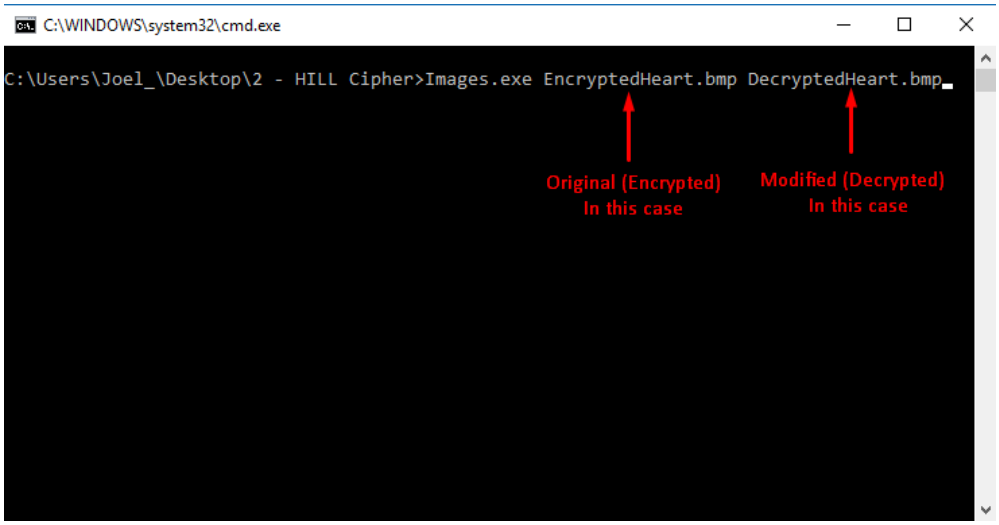


Figure 15. Example of execution of the program for decrypting process

In Figures 16 and 17, we show the first look and the header of the encrypted image to decrypt it (similar to Figures 12 and 13).

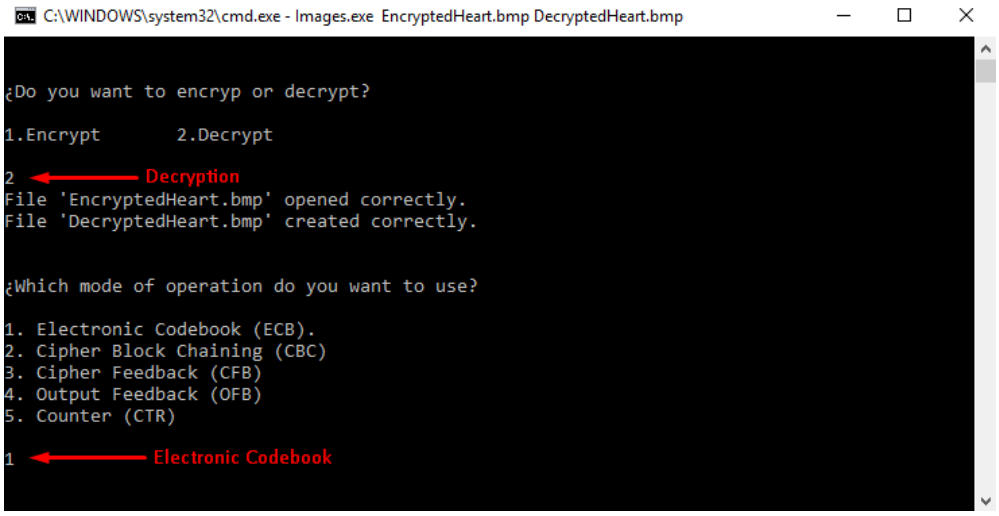


Figure 16. Example of the first look of the execution of the program

Figure 17, shows the header of the encrypted image.

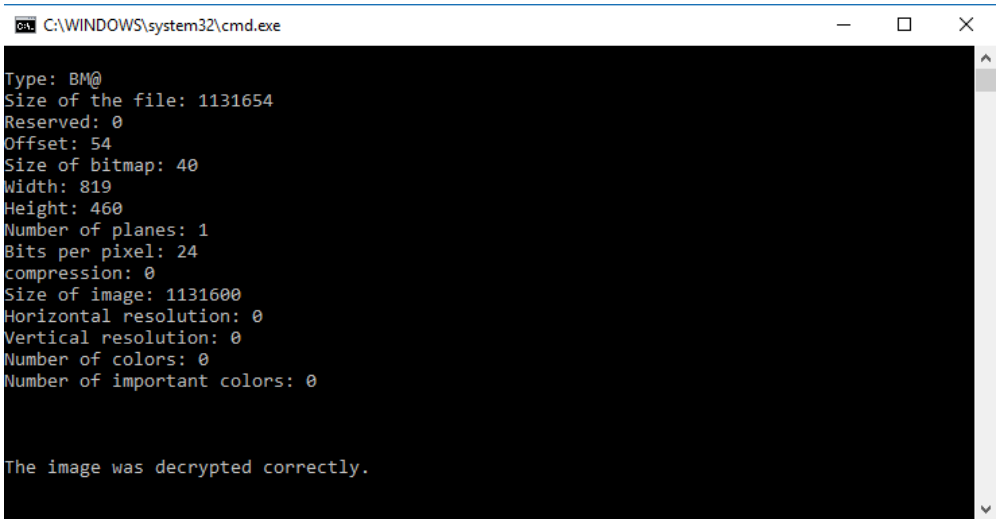


Figure 17. Header of the image (EncryptedHeart.bmp)

Finally, after we know that everything was okay, Figure 18 shows the decrypted image.

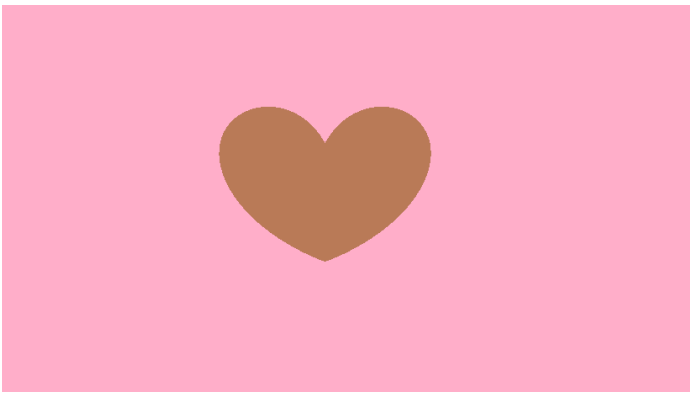


Figure 18. Decrypted Image with ECB Mode (DecryptedHeart.bmp)

As we can see, the image was decrypted correctly with the first mode of operation and the simplest image (just 2 colors). At this point, we know how the program works, that's why, in the subsequent, I only show you the image (mentioning with which mode of operation was encrypted and decrypted), all the next ciphers will be encrypted with initialization vector = 9 99 11

Figure 19. Encrypted Image with CBC Mode (EncryptedHeart.bmp)

Figure 19 shows the encryption for the original image (Heart.bmp).

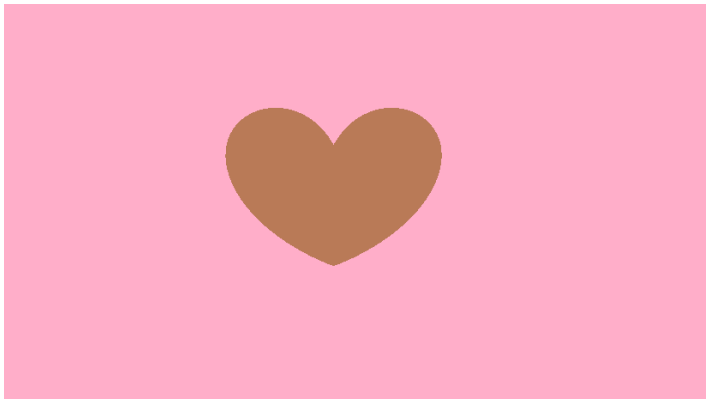


Figure 20. Decrypted Image with CBC Mode (DecryptedHeart.bmp)

Figure 20 shows the decryption for the original image (EncryptedHeart.bmp).

Figure 21. Encrypted Image with OFB Mode (EncryptedHeart.bmp)

Figure 21 shows the encryption for the original image (Heart.bmp).

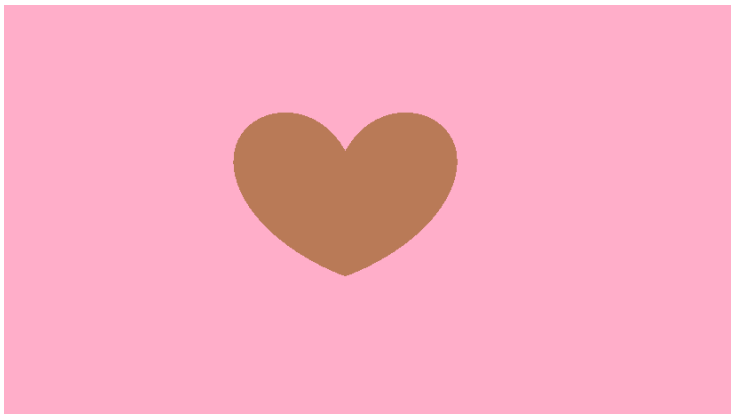


Figure 22. Decrypted Image with OFB Mode (DecryptedHeart.bmp)

Figure 22 shows the decryption for the original image (EncryptedHeart.bmp).

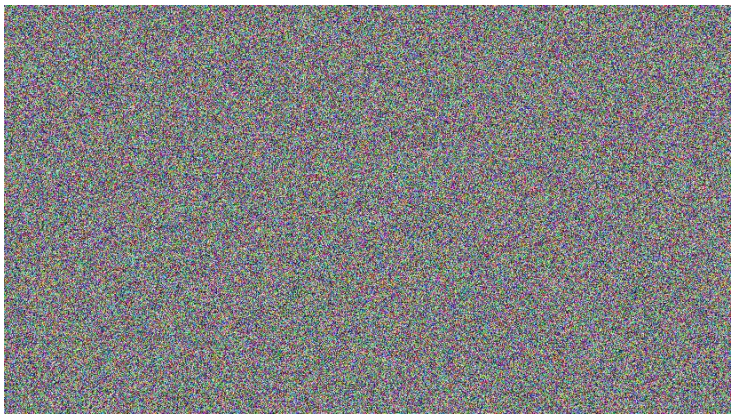


Figure 23. Encrypted Image with CFB Mode (EncryptedHeart.bmp)

Figure 23 shows the encryption for the original image (Heart.bmp).

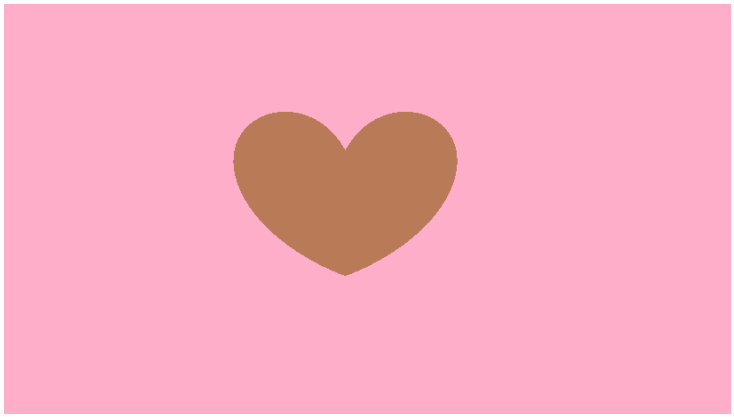


Figure 24. Decrypted Image with CFB Mode (DecryptedHeart.bmp)

Figure 24 shows the decryption for the original image (EncryptedHeart.bmp). Now, we proceed to apply all the modes of operation to another image.

Figure 25. Original Image (Landscape.bmp)

Figure 25 shows the original image (Landscape.bmp).

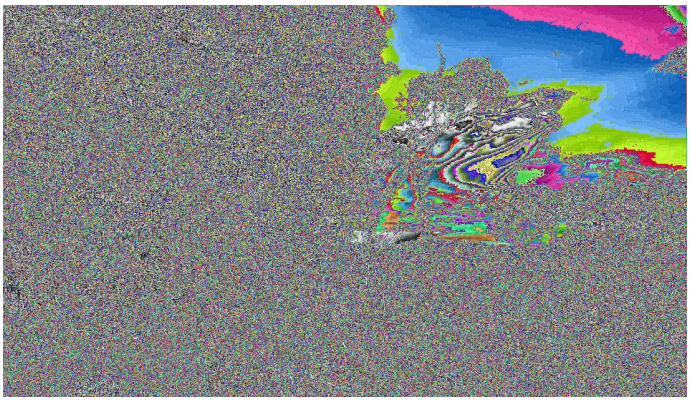


Figure 26. Encrypted Image with ECB Mode (EncryptedLandscape.bmp)

Figure 26 shows the encryption for the original image (Landscape.bmp).

Figure 27. Decrypted Image with ECB Mode (DecryptedLandscape.bmp)

Figure 27 shows the decryption for the original image (EncryptedLandscape.bmp).

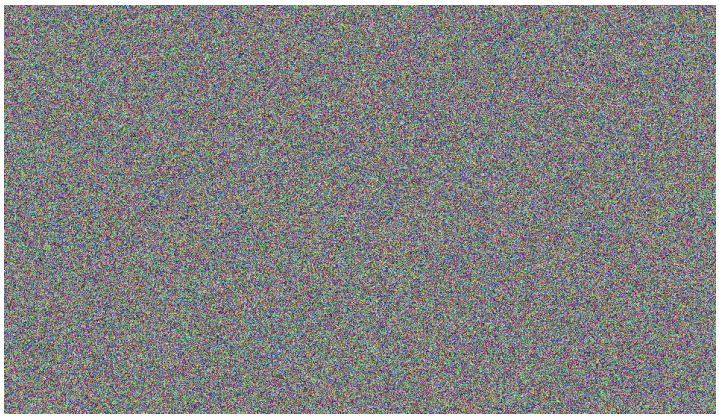


Figure 28. Encrypted Image with CBC Mode (EncryptedLandscape.bmp)

Figure 28 shows the encryption for the original image (Landscape.bmp).

Figure 29. Decrypted Image with CBC Mode (DecryptedLandscape.bmp)

Figure 29 shows the decryption for the original image (EncryptedLandscape.bmp).

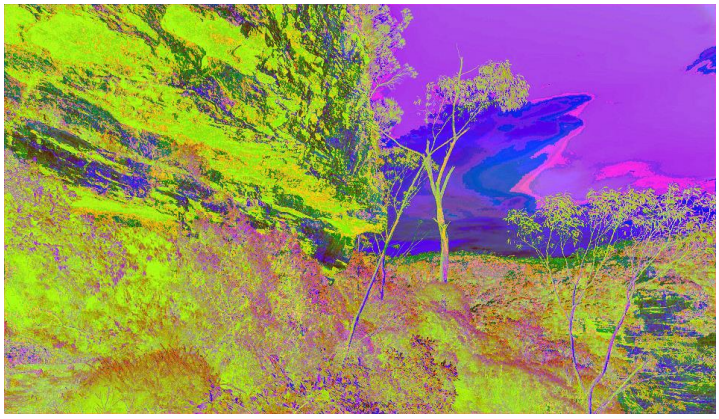


Figure 30. Encrypted Image with OFB Mode (EncryptedLandscape.bmp)

Figure 30 shows the encryption for the original image (Landscape.bmp).

Figure 31. Decrypted Image with OFB Mode (DecryptedLandscape.bmp)

Figure 31 shows the decryption for the original image (EncryptedLandscape.bmp).

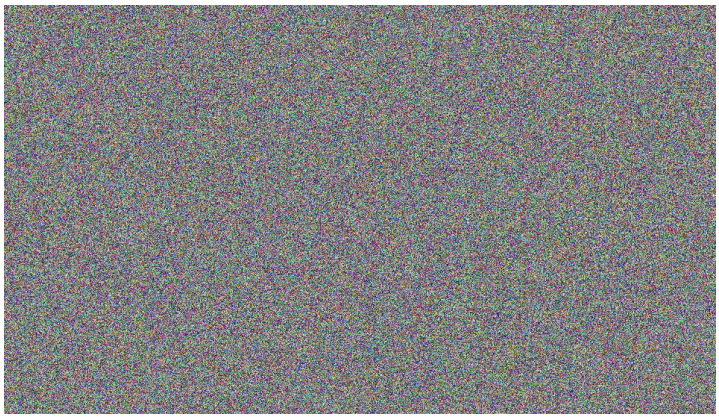


Figure 32. Encrypted Image with CFB Mode (EncryptedLandscape.bmp)

Figure 32 shows the encryption for the original image (Landscape.bmp).

Figure 33. Decrypted Image with CFB Mode (DecryptedLandscape.bmp)

Figure 33 shows the decryption for the original image (EncryptedLandscape.bmp). Now, we proceed to apply all the modes of operation to another image.

Figure 34. Original Image (Dog.bmp)

Figure 34 shows the original image (Dog.bmp).

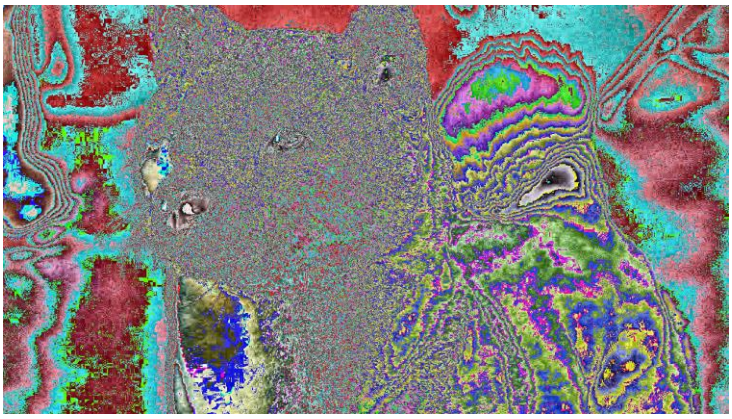


Figure 35. Encrypted Image with ECB Mode (EncryptedDog.bmp)

Figure 35 shows the encryption for the original image (Dog.bmp).

Figure 36. Decrypted Image with ECB Mode (DecryptedDog.bmp)

Figure 36 shows the decryption for the original image (EncryptedDog.bmp).

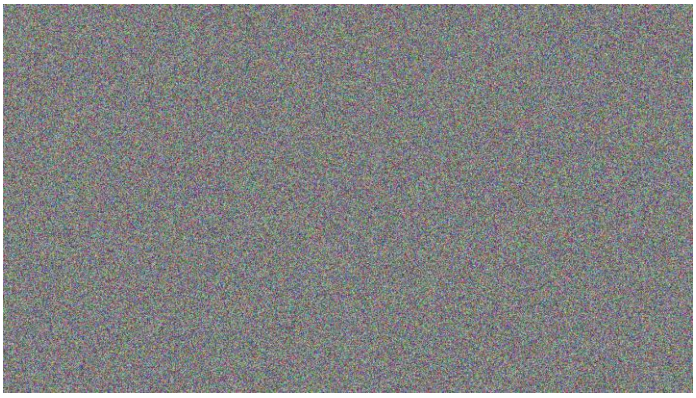


Figure 37. Encrypted Image with CBC Mode (EncryptedDog.bmp)

Figure 37 shows the encryption for the original image (Dog.bmp).

Figure 38. Decrypted Image with CBC Mode (DecryptedDog.bmp)

Figure 38 shows the decryption for the original image (EncryptedDog.bmp).

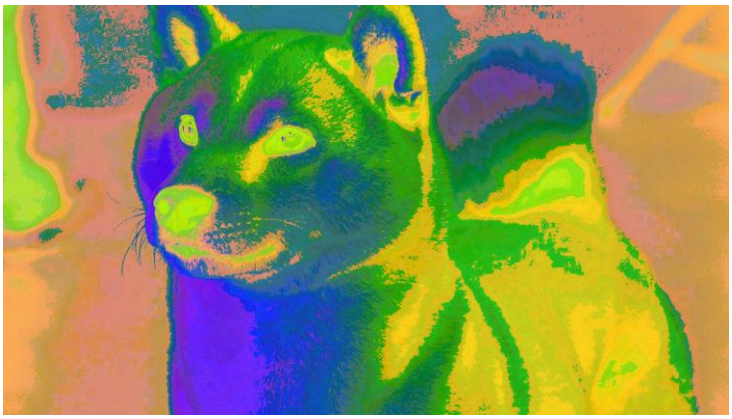


Figure 39. Encrypted Image with OFB Mode (EncryptedDog.bmp)

Figure 39 shows the encryption for the original image (Dog.bmp).

Figure 40. Decrypted Image with OFB Mode (DecryptedDog.bmp)

Figure 40 shows the decryption for the original image (EncryptedDog.bmp).

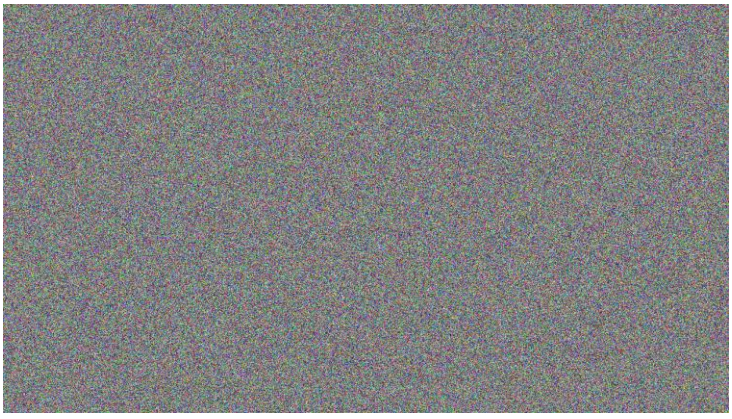


Figure 41. Encrypted Image with CFB Mode (EncryptedDog.bmp)

Figure 41 shows the encryption for the original image (Dog.bmp).

Figure 42. Decrypted Image with CFB Mode (DecryptedDog.bmp)

Figure 42 shows the decryption for the original image (EncryptedDog.bmp).

Discussion:

We can appreciate in the previous section that all the modes of operations work correctly, encrypting and decrypting 3 different images. Two of them have a lot of colors, the other one (the same image that appears on Table 1) has just 2 colors. If we do a posteriori analysis of the modes of operation, we can note that as we mention on the Literature Review of this report, the Mode of Operation (OFB) is the most insecure, because even if the image is encrypted, you can know what the image is, also, if we encrypt twice, the result will be the original image as what we obtain with the decryption process. The results obtained in the previous part were obtained with a very simple algorithm (talking about all of the modes of operation), but, if we think a little, the algorithm is too simple but very expensive (on execution, too much operations)[5], because the use of memory is just 3 matrixes (in the worst case) of 1x3, just 3 bytes (unsigned char).

But, the part of the code in which we read, write, etc. all the cipher or original pixels, is very expensive, the reason is:

- If the image has a size of 1,131,600, we do that number of iterations (reading and writing), but also, we do a multiplication by the corresponding key (another 3 operations inside the big loop), and if we use the bit – level operation, another 3 operations inside the big one.

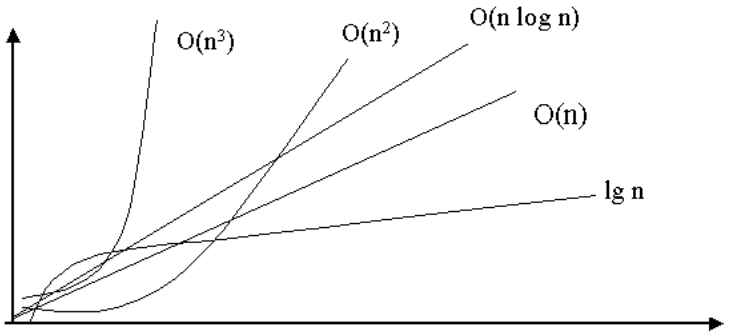


Figure 43. Complexity orders of an algorithm

Conclusions:

Encrypting images was an interesting practice, because we are involved with them every day, simple, in our Facebook, WhatsApp, etc.

That's why, is sensible information, and as users, we don't want to be our photos to the reach of everyone, encrypting using one of the modes of operation used in this practice could be a great idea to not to store the images for example in our database as a plaintext, we can store them encrypted and give us (the users) a kind of security with some of our personal information.

The use of BMP images was too simple, because it is only composed by a header and then, the pixels in inverse order (first the last part of the image), composed by 3 colors that we can change applying (for the first time) matrix operations and bit – level operations (make the ciphers faster and safer).

References:

[1] "Block Ciphers - HILL Cipher", class notes for Cryptography, Department of Engineering in Computer Systems, Escuela Superior de Cómputo, 2017.

[2] Tutorialspoint, 'The C Standard Library', 2012, [Online], Available: https://www.tutorialspoint.com/c_standard_library/**.** [Accessed: 27 – September – 2017].

[3] Agustín Cruz Contreras, Juan Carlos González Robles, Juan Carlos Herrera Lozada, 'Procesamiento de Imágenes: Estructura de Archivos BMP', 2004, [Online], Available: https://www.polibits.gelbukh.com/2004_30/Procesamiento%20de%20Imagenes_%20Estructura%20de%20Archivos%20BMP.pdf. [Accessed: 27 – September – 2017].

[4] Edgardo Adrián Franco Martínez, 'Lectura de Imágenes BMP', 2010, [Online]. Available: http://www.eafranco.com/docencia/sistemasoperativosii/files/programas/BMP.c. [Accessed: 27 – September – 2017].

Code

Images.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Functions.c"

int main (int argc, char* argv[])
{
          FILE * original, * encrypted;
          bmp image;
          int i, option;
          char * encryptedImage = (char *) malloc (sizeof (char));
          char * originalImage = (char *) malloc (sizeof (char));
          system ("cls");
          if (argc < 3)
          {
                    printf("Error, missing arguments.\nExample: %s Image.bmp
EncryptedImage.bmp\n\n", argv [0]);
                    exit (0);
          }else
          {
                    originalImage = (char *) argv [1];
                    encryptedImage = (char *) argv [2];
          }
          printf("\n\n%cDo you want to encryp or decrypt?\n\n1.Encrypt\t2.Decrypt\n\n",
168);
          scanf ("%d", &option);

          //We open each file in binary mode
          original = open_file (originalImage, encryptedImage, 1);
          encrypted = open_file (originalImage, encryptedImage, 2);

          //We read and write the head of the file
          read_head (original, encrypted, &image);
          if (option == 1)
                    operation_mode (original, encrypted, &image, 'e');
          else if (option == 2)
                    operation_mode (original, encrypted, &image, 'd');
          exit (0);
}
```

Functions.h

```
//Estructura para almacenar la cabecera de la imagen BMP y un apuntador a la matriz de
pixeles
typedef struct BMP
{
          char type [2];               //(2 Bytes) It contains the characters 'BM'
          int file_size;               //(4 Bytes) It contains the size of the entire file
          int reserved;                //(4 Bytes) It contains reserved bytes
          int offset;                  //(4 Bytes) It contains the offset from the beginning

          //BMP Information
          int bitmap_size;             //(4 Bytes) It contains the size of the bitmap
          int width;                   //(4 Bytes) Width (Horizontal pixels)
          int height;                  //(4 Bytes) Height (Vertical pixels)
          short no_planes;             //(2 Bytes) Number of planes of the image
          short bits_per_pixel;        //(2 Bytes) Quantity of bits per pixel
          int compression;             //(4 Bytes) It contains 0 if it's not compressed
          int image_size;              //(4 Bytes) It contains the size of the image
          int horizontal_res;          //(4 Bytes) It contains the horizontal resolution
          int vertical_res;            //(4 Bytes) It contains the vertical resolution
          int no_colors;               //(4 Bytes) It contains the number of colors
          int important_colors;        //(4 Bytes) It contains the number of important
colors
}bmp;

typedef struct llave
{
          unsigned char Ek [3][3];
          unsigned char Dk [3][3];
}llave;

llave key =
                                       //We initialize the two matrixes of the struct
          {
                    {
                              {1, 2, 3},
                              {4, 5, 6},
                              {11, 9, 8}
                    },
                    {
                              {90, 167, 1},
                              {74, 179, 254},
                              {177, 81, 1}
                    }
          };

FILE * open_file (char * original, char * encrypted, int tipo);
void read_head (FILE * original, FILE * encrypted, bmp * image);
void hill (unsigned char * BGR, unsigned char * pixel, char option);
void operation_mode (FILE * original, FILE * encrypted, bmp * image, char option);
void print_head (bmp * image);
char * message (char option);
void ECB (FILE * original, FILE * encrypted, bmp * image, char option);
void CBC (FILE * original, FILE * encrypted, bmp * image, char option);
void CFB (FILE * original, FILE * encrypted, bmp * image, char option);
void OFB (FILE * original, FILE * encrypted, bmp * image, char option);
void CTR (FILE * original, FILE * encrypted, bmp * image, char option);
```

Functions.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Functions.h"

int i, j;
                                                //Global variables for loops
unsigned char BGR [3], pixel [3], aux [3];
          //Arrays for reading and writing bmp images

FILE * open_file (char * original, char * encrypted, int tipo)
{
          FILE * pt1, * pt2;
          //We open the file in binary mode to read
          pt1 = fopen (original, "rb");
          if (pt1 == NULL)
          {
                    printf("Error while opening file: '%s'.\n", original);
                    exit(0);
          }

          //We open the file in binary mode to write
          pt2 = fopen (encrypted, "wb");
          if (pt2 == NULL)
          {
                    printf("Error while creating file: '%s'.\n", encrypted);
                    exit(1);
          }
          if (tipo == 1)
          {
                    printf("File '%s' opened correctly.\n", original);
                    return pt1;
          }
          else
          {
                    printf("File '%s' created correctly.\n", encrypted);
                    return pt2;
          }
}

void read_head (FILE * original, FILE * encrypted, bmp * image)
{
          //Type (must be 'BM')
          fread (&image -> type, sizeof (char), 2, original);
          fwrite (&image -> type, sizeof (char), 2, encrypted);

          //Size of the file
          fread (&image -> file_size, sizeof (int), 1, original);
          fwrite (&image -> file_size, sizeof (int), 1, encrypted);

          //Reserved bytes
          fread (&image -> reserved, sizeof (int), 1, original);
          fwrite (&image -> reserved, sizeof (int), 1, encrypted);

          //Offset
          fread (&image -> offset, sizeof (int), 1, original);
          fwrite (&image -> offset, sizeof (int), 1, encrypted);

          //Size of the bitmap
          fread (&image -> bitmap_size, sizeof (int), 1, original);
          fwrite (&image -> bitmap_size, sizeof (int), 1, encrypted);

          //Width
          fread (&image -> width, sizeof (int), 1, original);
          fwrite (&image -> width, sizeof (int), 1, encrypted);

          //Height
          fread (&image -> height, sizeof (int), 1, original);
```

```
        fwrite (&image -> height, sizeof (int), 1, encrypted);

        //Number of planes
        fread (&image -> no_planes, sizeof (short),1, original);
        fwrite (&image -> no_planes, sizeof (short),1, encrypted);

        //Bits per pixel
        fread (&image -> bits_per_pixel, sizeof (short),1, original);
        fwrite (&image -> bits_per_pixel, sizeof (short),1, encrypted);

        //Type of compression (must be 0)
        fread (&image -> compression, sizeof (int), 1, original);
        fwrite (&image -> compression, sizeof (int), 1, encrypted);

        //Size of the image
        fread (&image -> image_size, sizeof (int), 1, original);
        fwrite (&image -> image_size, sizeof (int), 1, encrypted);

        //Horizontal resolution
        fread (&image -> horizontal_res, sizeof (int), 1, original);
        fwrite (&image -> horizontal_res, sizeof (int), 1, encrypted);

        //Vertical resolution
        fread (&image -> vertical_res, sizeof (int), 1, original);
        fwrite (&image -> vertical_res, sizeof (int), 1, encrypted);

        //Number of colors
        fread (&image -> no_colors, sizeof (int), 1, original);
        fwrite (&image -> no_colors, sizeof (int), 1, encrypted);

        //Number of important colors
        fread (&image -> important_colors, sizeof (int), 1, original);
        fwrite (&image -> important_colors, sizeof (int), 1, encrypted);

        //We check if the selected file is a bitmap
        if (image -> type [0] != 'B' || image -> type [1] != 'M')
        {
                printf ("The image must be a bitmap.\n");
                exit (1);
        }
        if (image -> bits_per_pixel != 24)
        {
                printf ("The image must be 24-bits.\n");
                exit (1);
        }
}

void operation_mode (FILE * original, FILE * encrypted, bmp * image, char option)
{
        int selected_mode = 3;
        printf("\n\n%cWhich mode of operation do you want to use?\n\n", 168);
        printf("1. Electronic Codebook (ECB).\n");
        printf("2. Cipher Block Chaining (CBC)\n");
        printf("3. Cipher Feedback (CFB)\n");
        printf("4. Output Feedback (OFB)\n");
        printf("5. Counter (CTR)\n\n");
        scanf ("%d", &selected_mode);
        system ("cls");
        print_head (image);
        if (selected_mode == 1)
                ECB (original, encrypted, image, option);
        else if (selected_mode == 2)
                CBC (original, encrypted, image, option);
        else if (selected_mode == 3)
                CFB (original, encrypted, image, option);
        else if (selected_mode == 4)
                OFB (original, encrypted, image, option);
        else
                CTR (original, encrypted, image, option);
}
```

```
void print_head (bmp * image)
{
          printf ("\n\nType: %s\n", image -> type);
          printf ("Size of the file: %d\n", image -> file_size);
          printf ("Reserved: %d\n", image -> reserved);
          printf ("Offset: %d\n", image -> offset);
          printf ("Size of bitmap: %d\n", image -> bitmap_size);
          printf ("Width: %d\n", image -> width);
          printf ("Height: %d\n", image -> height);
          printf ("Number of planes: %d\n", image -> no_planes);
          printf ("Bits per pixel: %d\n", image -> bits_per_pixel);
          printf ("compression: %d\n", image -> compression);
          printf ("Size of image: %d\n", image -> image_size);
          printf ("Horizontal resolution: %d\n", image -> horizontal_res);
          printf ("Vertical resolution: %d\n", image -> vertical_res);
          printf ("Number of colors: %d\n", image -> no_colors);
          printf ("Number of important colors: %d\n", image -> important_colors);
}

char * message (char option)
{
          if (option == 'e')
                    return "encrypted";
          else
                    return "decrypted";
}

void ECB (FILE * original, FILE * encrypted, bmp * image, char option)
{
          for (i = 0; i < (image -> image_size); i ++)
          {
                    fread (&BGR, sizeof (char), 3, original);
                    hill ((unsigned char * ) BGR, (unsigned char * ) pixel, option);
                    fwrite (&pixel, sizeof (char), 3, encrypted);
                    memset (pixel, 0, 3);
          }
          printf ("\n\n\nThe image was %s correctly.\n\n", message (option));
}

void CBC (FILE * original, FILE * encrypted, bmp * image, char option)
{
          printf ("\n\nIntroduce the initialization vector separated by spaces:\t");
          scanf ("%u %u %u", &pixel [0], &pixel [1], &pixel [2]);
          if (option == 'e')
          {
                    for (i = 0; i < (image -> image_size); i ++)
                    {
                              fread (&BGR, sizeof (char), 3, original);
                              for (j = 0; j < 3; j ++)
                                        BGR [j] = (pixel [j] ^ BGR [j]);
                                                  //We realize XOR between pixel and
BGR from Image
                              hill ((unsigned char * ) BGR, (unsigned char * ) pixel,
option);
                              fwrite (&pixel, sizeof (char), 3, encrypted);
                    }
          }else
          {
                    for (i = 0; i < (image -> image_size); i ++)
                    {
                              fread (&BGR, sizeof (char), 3, original);
                              hill ((unsigned char * ) BGR, (unsigned char * ) aux, option);
                              for (j = 0; j < 3; j ++)
                                        pixel [j] = (pixel [j] ^ aux [j]);
                                                  //We realize XOR between pixel and
BGR from Image
                              fwrite (&pixel, sizeof (char), 3, encrypted);
                              for (j = 0; j < 3; j ++)
                                        pixel [j] = BGR [j];
                    }
          }
```

```
        printf ("\n\n\nThe image was %s correctly.\n\n", message (option));
}

void CFB (FILE * original, FILE * encrypted, bmp * image, char option)
{
        if (option == 'e')
        {
                printf ("\n\nIntroduce the initialization vector separated by
spaces:\t");
                scanf ("%u %u %u", &pixel [0], &pixel [1], &pixel [2]);
                for (i = 0; i < (image -> image_size); i ++)
                {
                        fread (&BGR, sizeof (char), 3, original);
                        hill ((unsigned char * ) pixel, (unsigned char * ) aux,
option);
                        for (j = 0; j < 3; j ++)
                                pixel [j] = (aux [j] ^ BGR [j]);
                                        //We realize XOR between pixel and
BGR from Image
                        fwrite (&pixel, sizeof (char), 3, encrypted);
                }
        }else
        {
                printf ("\n\nIntroduce the initialization vector separated by
spaces:\t");
                scanf ("%u %u %u", &BGR [0], &BGR [1], &BGR [2]);
                for (i = 0; i < (image -> image_size); i ++)
                {
                        hill ((unsigned char * ) BGR, (unsigned char * ) aux, 'e');
                        fread (&BGR, sizeof (char), 3, original);
                        for (j = 0; j < 3; j ++)
                                pixel [j] = (aux [j] ^ BGR [j]);
                                        //We realize XOR between pixel and
BGR from Image
                        fwrite (&pixel, sizeof (char), 3, encrypted);
                }
        }
        printf ("\n\n\nThe image was %s correctly.\n\n", message (option));
}

void OFB (FILE * original, FILE * encrypted, bmp * image, char option)
{
        unsigned char aux2 [3];
        printf ("\n\nIntroduce the initialization vector separated by spaces:\t");
        scanf ("%u %u %u", &pixel [0], &pixel [1], &pixel [2]);
        hill ((unsigned char * ) pixel, (unsigned char * ) aux, 'e');
        for (i = 0; i < 3; i ++)
                aux2 [i] = aux [i];
        for (i = 0; i < (image -> image_size); i ++)
        {
                fread (&BGR, sizeof (char), 3, original);
                for (j = 0; j < 3; j ++)
                        pixel [j] = (aux2 [j] ^ BGR [j]);
                                //We realize XOR between pixel and BGR from
Image
                fwrite (&pixel, sizeof (char), 3, encrypted);
                hill ((unsigned char * ) aux, (unsigned char * ) aux2, 'e');
        }
        printf ("\n\n\nThe image was %s correctly.\n\n", message (option));
}

void CTR (FILE * original, FILE * encrypted, bmp * image, char option)
{
        //
}

void hill (unsigned char * BGR, unsigned char * pixel, char option)
{
        int i;
        for (i = 0; i < 3; i ++)
        {
```

```
                    if (option == 'd')                                  //D from decryption
                                pixel [i] = ((BGR [0] * key.Dk [0][i]) + (BGR [1] * key.Dk
[1][i]) + (BGR [2] * key.Dk [2][i])) % 256;
                    else
                                pixel [i] = ((BGR [0] * key.Ek [0][i]) + (BGR [1] * key.Ek
[1][i]) + (BGR [2] * key.Ek [2][i])) % 256;
          }
          return;
}
```