|  |  |  |
| --- | --- | --- |
| **ipn** | **INSTITUTO POLITÉCNICO NACIONAL**  **ESCUELA SUPERIOR DE CÓMPUTO** |  |

**Cryptography**

**Hill Cipher OM**

Abstract

There are some classical ciphers, one of these, is called Affine. The importance, is to keep your information safe and make it unreadable for unauthorized people, that’s why, this cipher could encrypt and decrypt a message ‘m’ in C language.

**By:**

**Romero Gamarra Joel Mauricio**

Professor:

MSc. NIDIA ASUNCIÓN CORTEZ DUARTE

October 2017

**Imagen que contiene texto, pizarra

Descripción generada con confianza muy alta**

To validate this report, it is necessary to include the corresponding seal

**Index**

Content

[Introduction: 1](#_Toc476313175)

[Literature review: 1](#_Toc476313176)

[Software (libraries, packages, tools): 9](#_Software_(libraries,_packages,)

[Procedure: 10](#_Toc476313178)

[Results 11](#_Toc476313179)

[Discussion: 7](#_Discussion:)

[Conclusions: 7](#_Conclusions:)

[References: 8](#_References:)

[Code 9](#_Code)

# 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 · K mod 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.

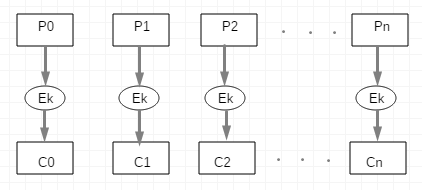


Figure 1. Electronic Codebook Diagram (Encryption)

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

C0 = Ek (P0) C1 = Ek (P1) C2 = Ek (P2) … **Cn = Ek (Pn)**

Now, to decrypt the image, we can apply “Dk” both on the left and right sides of the formula:

Cn = Ek (Pn) Dk (Cn) = Dk (Ek (Pn)) **Pn = Dk (Cn)**

Where:

* Pi = Plaintext = Pixels of the original image
* Ci = Ciphers = Pixels of the modified image
* Ek = Encryption Key (Matrix K)
* Dk = Decryption Key (Matrix K-1)

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

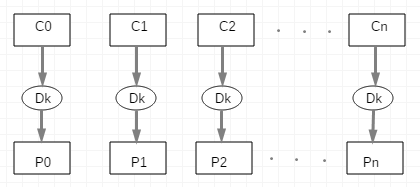


Figure 2. Electronic Codebook Diagram (Decryption)

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 (Ci) 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 C0 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:

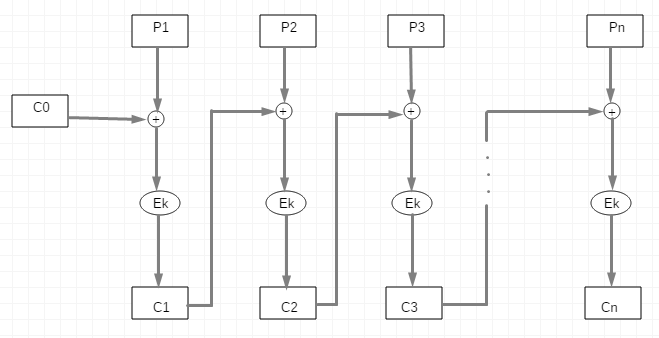


Figure 3. Cipher Block Chaining Diagram (Encryption)

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

C1 = Ek (P1 C0) C2 = Ek (P2 C1) C3 = Ek (P3 C2) … **Cn = Ek (Pn Cn-1)**

Now, for the opposite process, we need to clear **Pi:**

Cn = Ek (Pn Cn-1) Dk (Cn) = Dk (Ek (Pn Cn-1)) Dk (Cn) Cn-1 = Pn Cn-1 Cn-1

Finally:

**Pn = Dk (Cn) Cn-1**

Where:

* = XOR

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

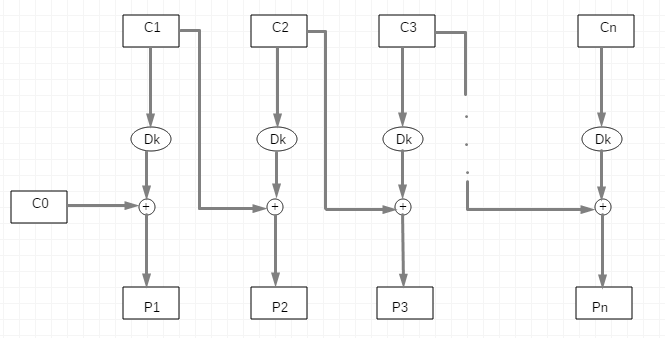


Figure 4. Cipher Block Chaining Diagram (Decryption)

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 C0 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.

Imagen que contiene texto, interior, pared, pizarra

Descripción generada con confianza muy alta

Figure 5. Output Feedback Diagram (Encryption)

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

C1 = Ek (C0) P1 C2 = Ek (Ek (C0)) P2  C3 = Ek (Ek (Ek (C0))) P3

Finally:

**Cn = Ekn (C0) Pn**

Now, for the opposite process, we need to clear **Pi:**

Cn = Ekn (C0) Pn Cn Ekn (C0) = Ekn (C0) Pn Ekn (C0)

Cn Ekn (C0) = Pn

Finally:

**Pn = Cn Ekn (C0)**

Now, we obtain the decryption process mathematically, is important to observe that for **decryption**, we don’t use **Dk**, we use **Ek** again and always applied n times (number of pixels) to C0 (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.

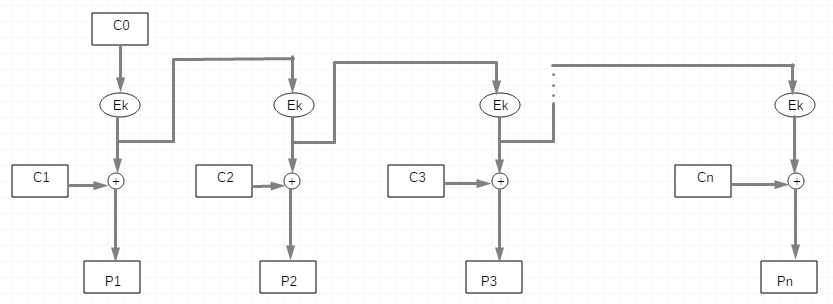


Figure 6. Output Feedback Diagram (Decryption)

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.

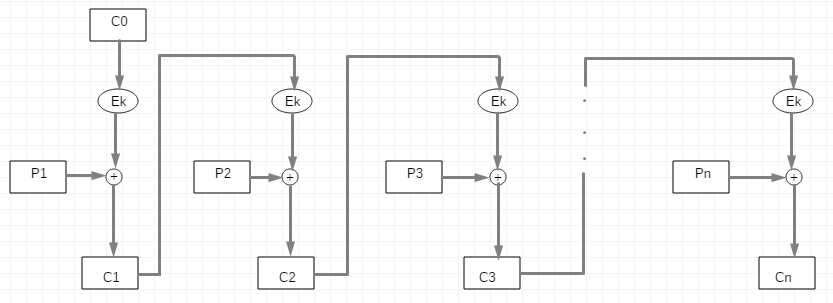


Figure 7. Cipher Feedback Diagram (Encryption)

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

C1 = Ek (C0) P1 C2 = Ek (C1) P2 C3 = Ek (C2) P3 … **Cn = Ek (Cn-1) Pn**

Now, for the opposite process, we need to clear **Pi:**

Cn = Ek (Cn-1) Pn Cn Ek (Cn-1) = Ek (Cn-1) Pn  Ek (Cn-1)Cn Ek (Cn-1) = Pn

Finally:

**Pn = Cn Ek (Cn-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.

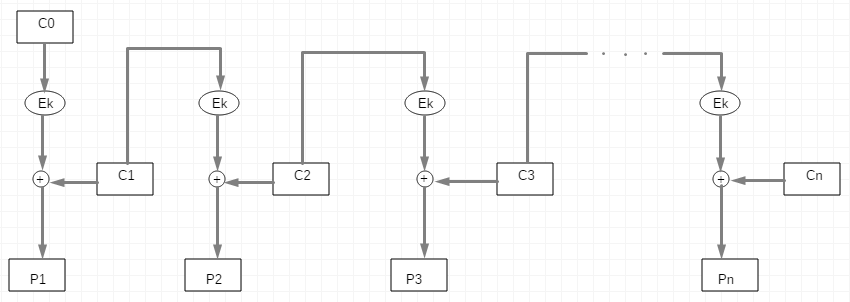


Figure 8. Cipher Feedback Diagram (Decryption)

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: **Pn = Cn Ekn (C0)** and **Cn = Ekn (C0)**  **Pn**  
    Substituting Pn on Encryption Function: **Cn = Ekn (C0) Cn Ekn (C0)**  
    Finally: **Cn = Cn**
* **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 Ek 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.

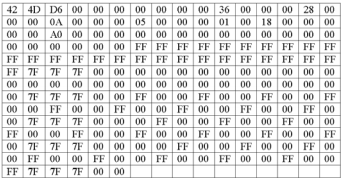


Figure 9. Structure of an example image

# Results

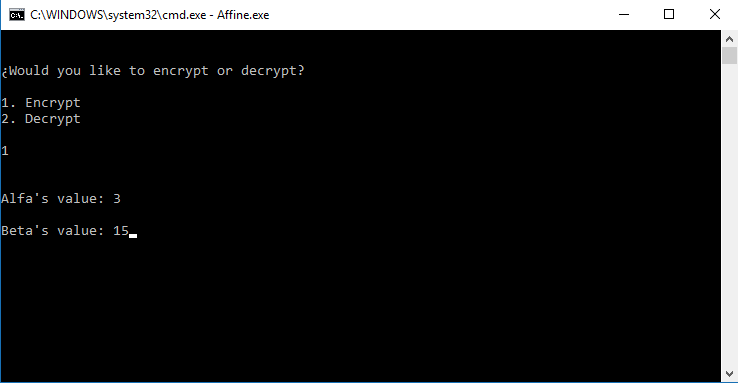


Figure 10. Main menu of the program selecting option 1 (Encrypt).

In figure 2, you could see the main menu, selecting first the option if you want to encrypt or decrypt a message from a file, then, you could select the values that alpha and beta are going to have through the execution of the program to obtain the cipher/decipher message.

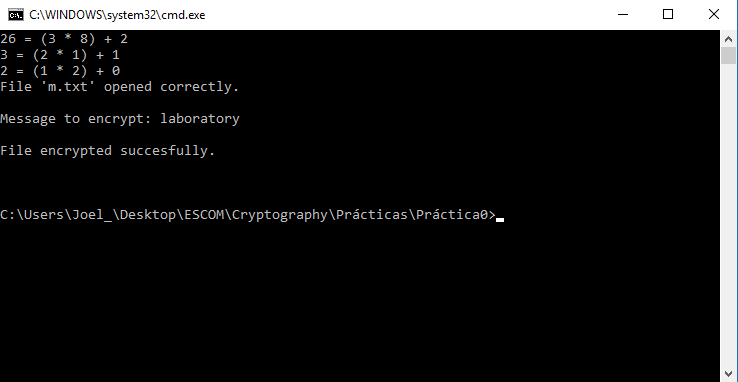


Figure 11. Execution of encryption showing the message, a text if the ciphered was successful and the steps corresponding to the Euclidean algorithm.

After we receive the message that the encryption was successful, we’ll proceed to review the text to check if it’s correctly ciphered (Figure 4) and compared to a web page4 with the same values and the same message (Figure 5).

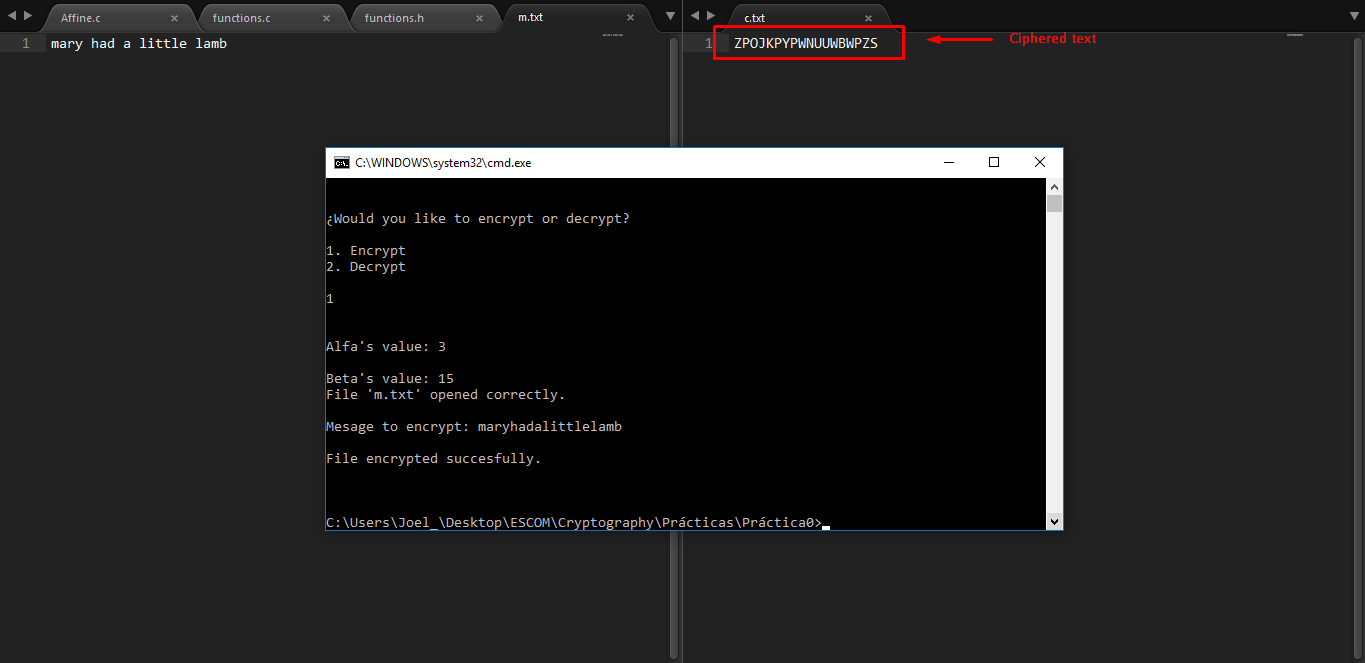


Figure 12. Demonstration of the encrypted message (right) with the original one (left).

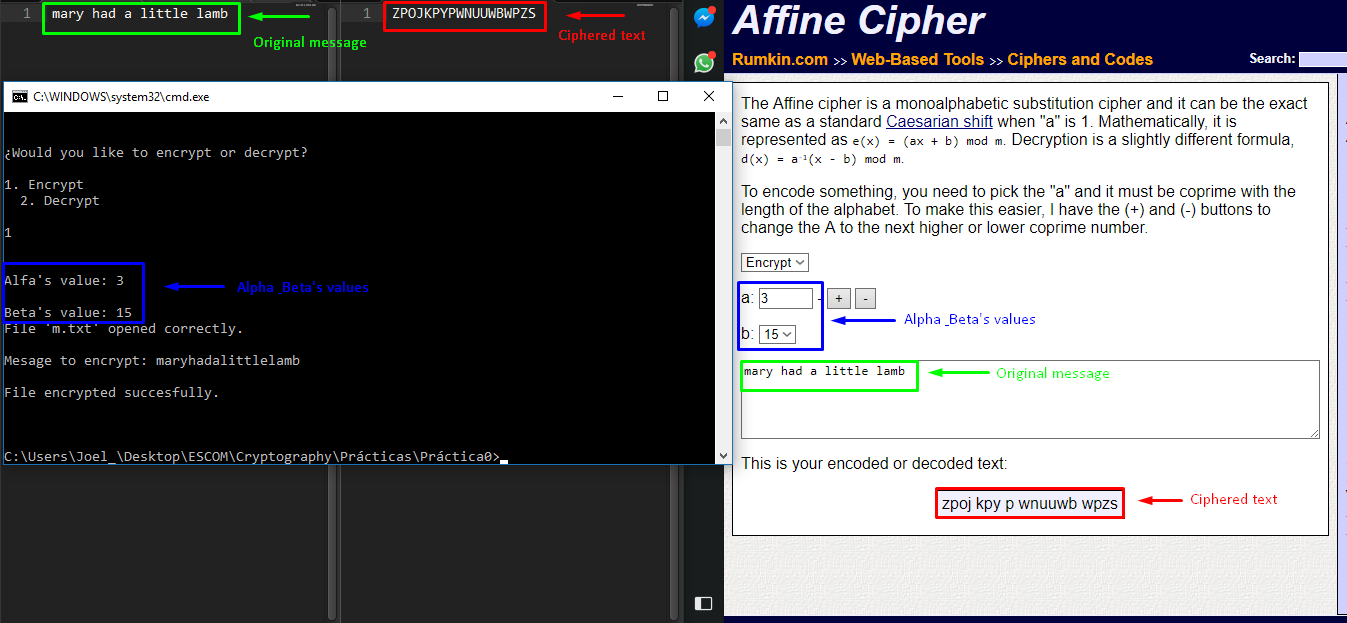


Figure 13. Comparison between a web page and my own program with the same values and messages.

Now, we proceed to verify the decryption option, first with wrong values and then with the correct ones to see if our decryption function described on Literature Review is correctly implemented.

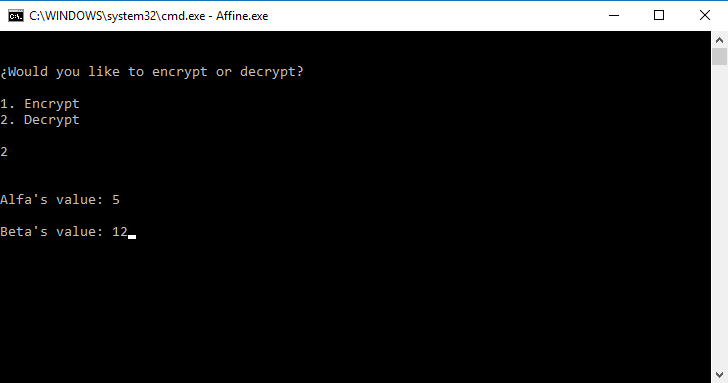


Figure 14. Main menu of the program selecting option 2 (Decrypt).

We can observe, that the value of alpha and beta are wrong (Figure 6), so, what must happen is that the message deciphered is incorrect, because the function won’t be the correct one to decrypt the message correctly.

In the following image (Figure 7), we can see de “decrypted” message, but it’s unreadable because it has no sense, and the message we were expecting is “mary had a little lamb”, like in the original message written in the text file.

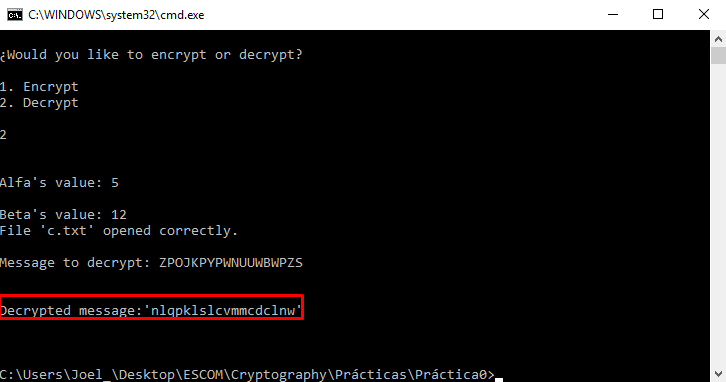


Figure 15. Wrong decrypted message due to alpha's and beta's values.

As we said before, the decrypted message is incorrect and make no sense, now, we’ll try with the correct combination of values and see if it’s correct the algorithm implemented in the “Code” Section.

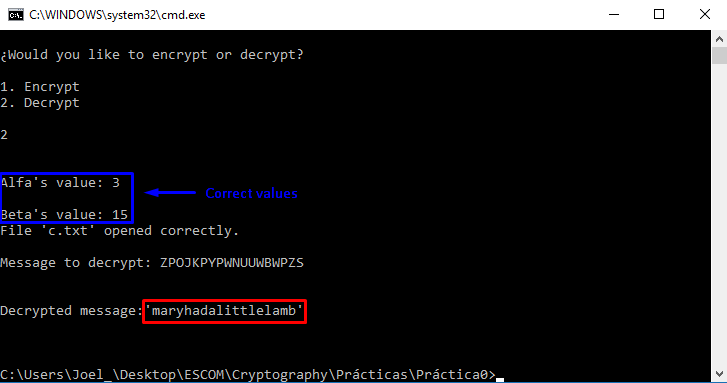


Figure 16. Correct decrypted message with the correct combination of alpha's and beta's values.

In the Figure 8 we can see that the correct combination (**key**) of the values on alpha and beta, gives us a readable and a coherent message, that was the original we tried in the text file showed before (Figure 5).

As you can see, the results are satisfactory even if the values are wrong because the message is protected by the key and the algorithm I implemented on the program, in the next section I will explain some little “errors” in the execution time and what’s the importance of this practice nowadays (as I already said, it’s a classical cipher, too weak in these days).

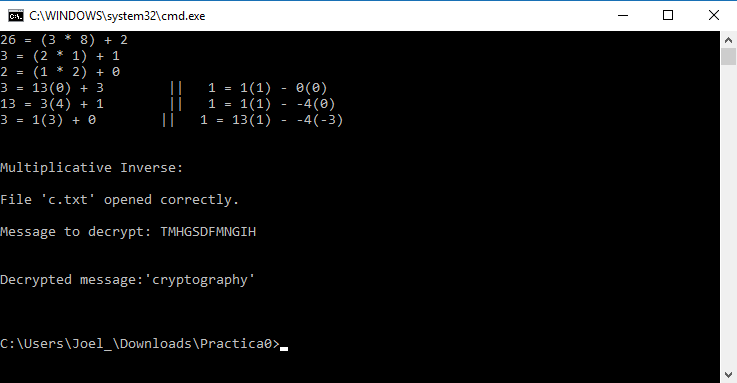
Extended Euclidean algorithm5 is implemented in Figure 9, is used to know the multiplicative inverse of alpha, only if greatest common divisor is 1, because it means that alpha and the alphabet size are prime each other.

Figure 17. Implementation of Extended Euclidean Algorithm to find the multiplicative inverse of alpha

# Discussion:

It’s clear that Affine Cipher could be a great cipher (could think that), because there are 252 possible combinations with alpha and beta, but still being a classical substitution cipher.

However, the importance of encrypting a message is hire the information from a person not allowed to read it, and this (Affine), accomplish the objective, or at least, did it some years ago.

The results obtained in the previous section was correct, because the program read a file with the message you want to encrypt and then ask you for alpha’s and beta’s values to encrypt your message and write it on another text file to decrypt it at any moment. One important application is protecting personnel information like address, salary, credit card, etc.

# Conclusions:

I learned that making a cipher is not too easy (but this practice was kind of), and is important to protect information always, even if you dedicate yourself to something else like singing, dancing, teaching, etc. And one of the needed services is provided by modern cryptography with the new algorithms that have been developed since at least, 40 years ago. For example, you could send your address and/or your phone number by one social network, but what happen if there’s a guy who’s taking information to kidnap your child.

One problem here, is that the program fails when the text to encrypt exceeds 600 characters, and we know that sensitive information has a longer length than that, so is a problem I need to solve immediately to offer my users more security in their personnel information.

Generally, my program could be applied at any alphabet, with some little fixes, but for example I implemented the alphabet size dynamic, so, we could have only a part of it and calculate correctly the ciphered/deciphered text from 2 simple values.

Of course, this program is not perfect, but I tried to optimize the most I could, for example avoiding the pair numbers in the function to obtain the multiplicative inverse because there’s time to the processor we’re wasting, in memory too.

# 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**

#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**;

}