INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Cryptography

“Permutation”

Abstract

Implementation in C language of one section of DES algorithm (IP and IP^{-1}) to mix bit by bit the word, in this case, 4 bytes instead of 8.

By:

Romero Gamarra Joel Mauricio

Professor:
MSc. NIDIA ASUNCIÓN CORTEZ DUARTE

November 2017

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

Index

Contenido

Introduction:

Data Encryption Standard is a “safety” cipher with a security of 64 bits (the security is given by the length of the key), in this case, this cipher encrypts and decrypts (the same process) 64 bits, the equivalent of 8 characters because each character (byte) is formed by 8 bits.

One of the parts of DES algorithm is called Initial Permutation (also, exists an Inverse Initial Permutation at the end of the entire process, after 16 rounds) that literally mixes bit by bit each word based on a defined table.

In this practice, is not implemented the one that DES algorithm says, but we can only change the numbers of the array to obtain the same behavior of the cipher.

Literature review:

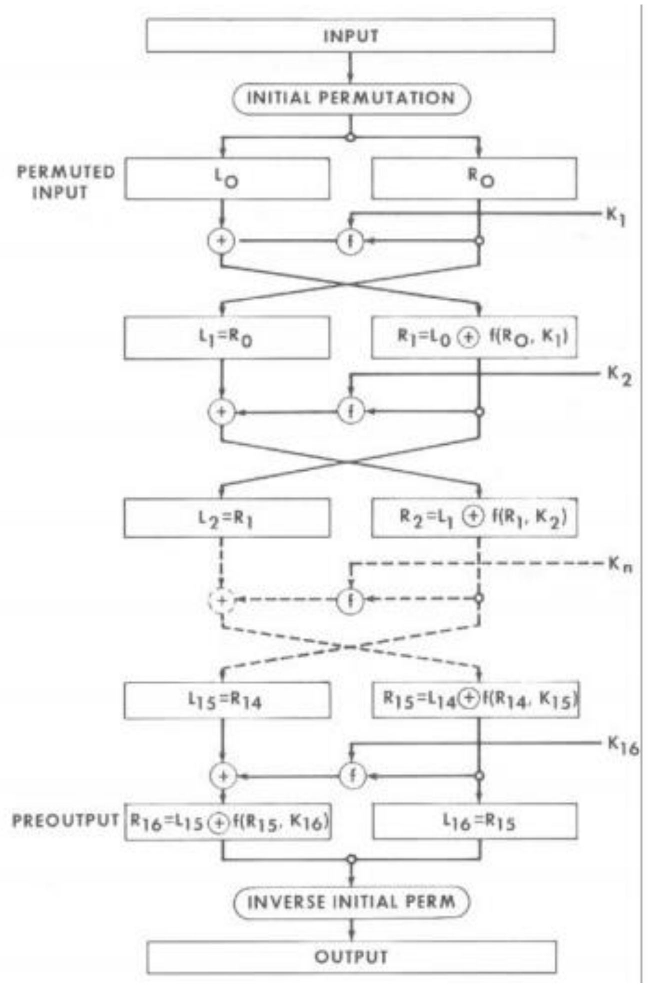
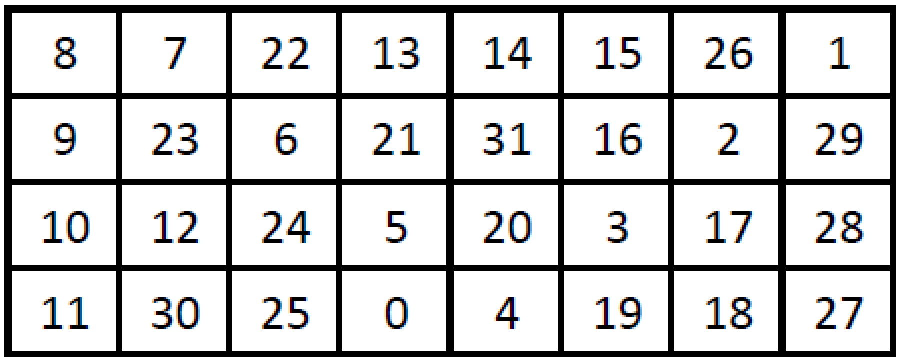


Figure 1. General block diagram of DES algorithm

On Figure 1, we can see the general block diagram of the DES Algorithm for enciphering[1].

The part of the practice, is called Initial Permutation, on Figure 2, we can see the exact permutation that we implemented on this program.

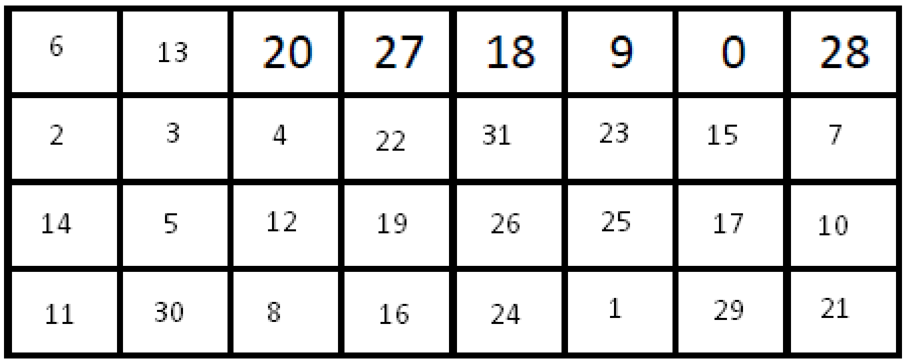


8	7	22	13	14	15	26	1
9	23	6	21	31	16	2	29
10	12	24	5	20	3	17	28
11	30	25	0	4	19	18	27

Figure 2. Initial Permutation (P)

Now, we can obtain inverse initial permutation (P^{-1}), to obtain it, what we need to do is check in which position is each number counting from right to left (from 0 to 31), for example, number 1 is positioned on number 0, number 6 is positioned on number 13 and you can obtain the others on that way.

To obtain P^{-1}, you go to the position of the number you read and put the position of that number, for example, if number 1 is positioned on number 0, you go to P^{-1} to position 1 and place the number 0 (position of number 1 on P). On Figure 3 you can see the complete inverse initial permutation P^{-1}.



6	13	20	27	18	9	0	28
2	3	4	22	31	23	15	7
14	5	12	19	26	25	17	10
11	30	8	16	24	1	29	21

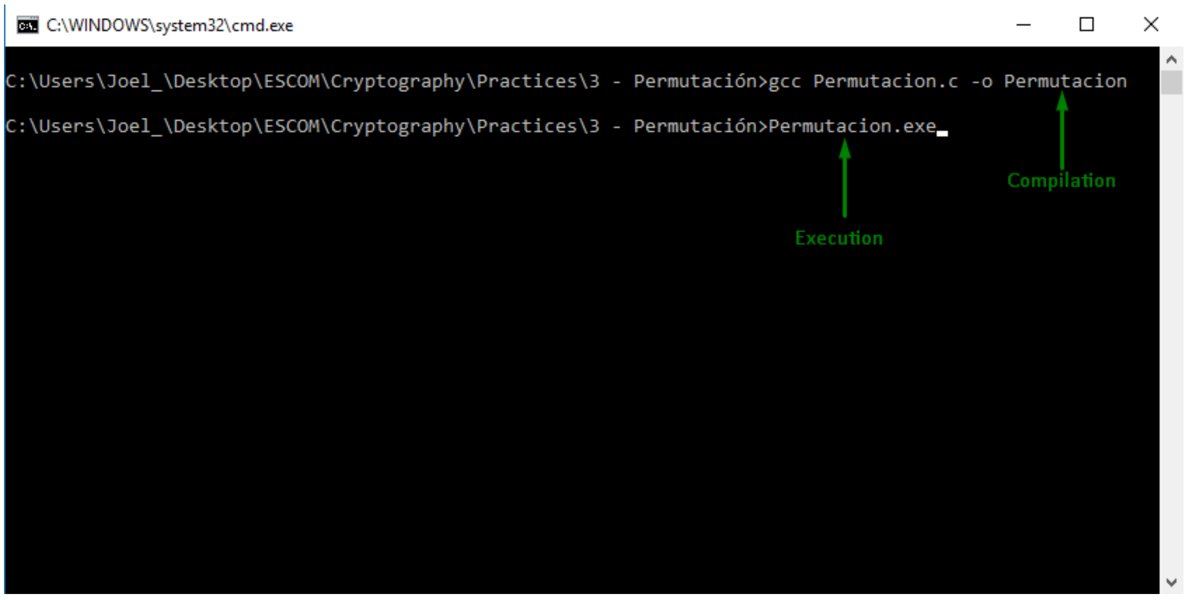
Figure 3. Inverse Initial Permutation (P^{-1})

Software (libraries, packages, tools):

- Sublime Text 3
- String.h
    - void * memset (void * ptr, int value, size_t num);

Results

On Figure 4, you can appreciate the first step, compile and execute the program wrote in C language.

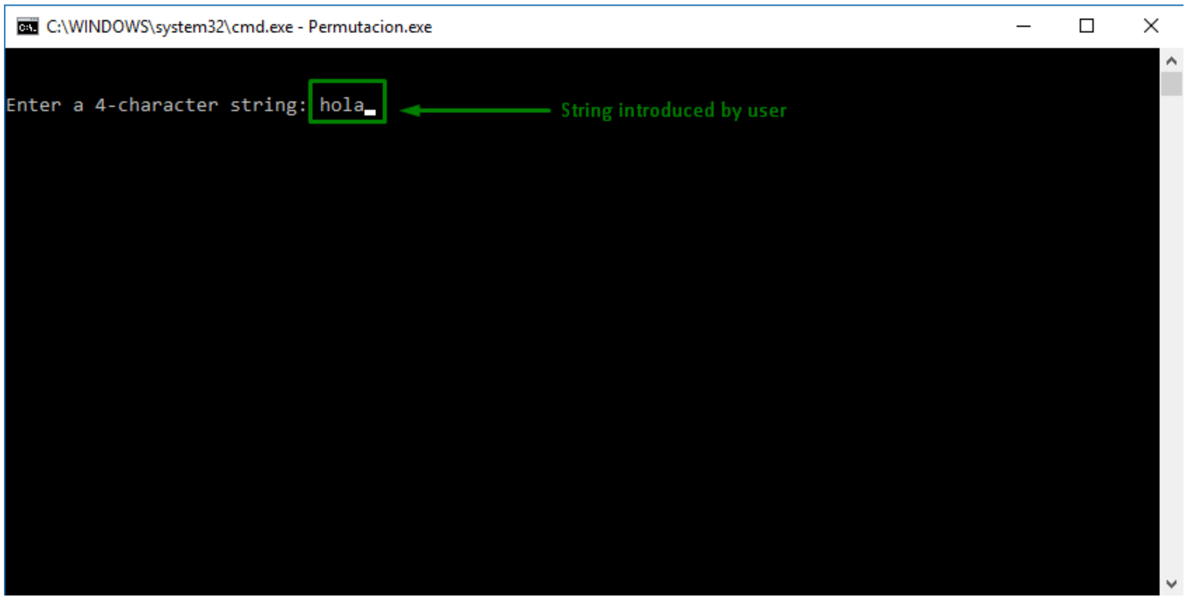


```
C:\Users\Joel_\Desktop\ESCOM\Cryptography\Practices\3 - Permutación>gcc Permutacion.c -o Permutacion

C:\Users\Joel_\Desktop\ESCOM\Cryptography\Practices\3 - Permutación>Permutacion.exe
```

Figure 4. Compilation and execution of the program

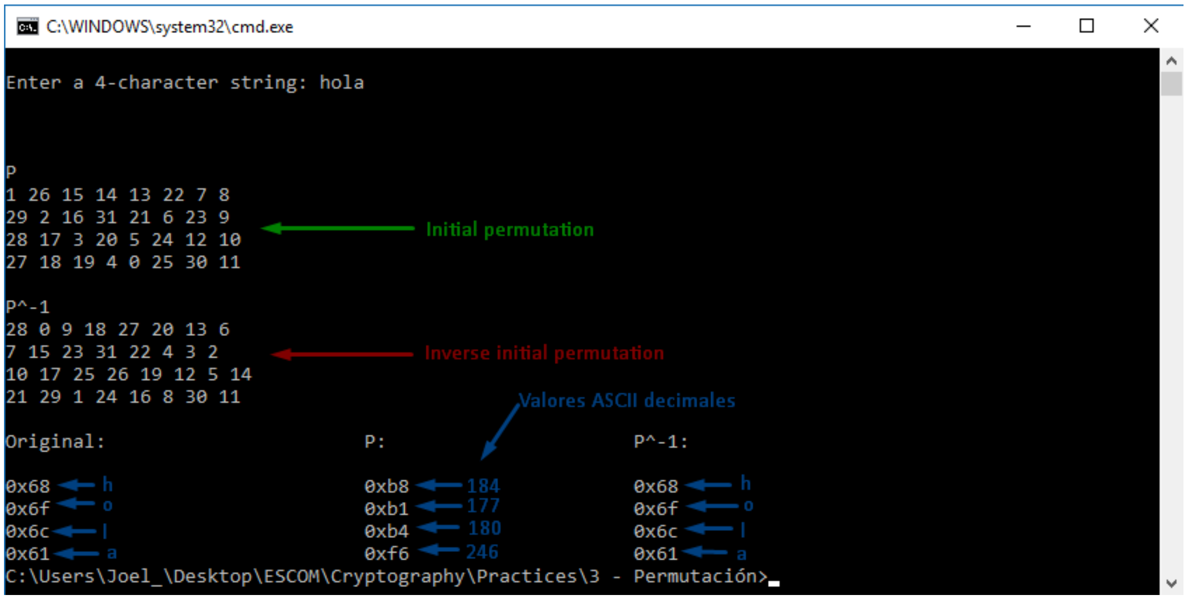
Then, you can observe on Figure 5 a message to the user to enter a string to mix their bits based on the permutation showed on Figure 2.


C:\WINDOWS\system32\cmd.exe - Permutacion.exe
String introduced by user

```
Enter a 4-character string: hola
```

Figure 5. Message to the user to introduce a string

After we have the string, we can mix its bits by following the table on Figure 2. The process is showed on Figure 6.



```
Enter a 4-character string: hola


P
1 26 15 14 13 22 7 8
29 2 16 31 21 6 23 9
28 17 3 20 5 24 12 10
27 18 19 4 0 25 30 11

P^-1
28 0 9 18 27 20 13 6
7 15 23 31 22 4 3 2
10 17 25 26 19 12 5 14
21 29 1 24 16 8 30 11

Original:                        P:                     P^-1:

0x68                             0xb8                   0x68
0x6f                             0xb1                   0x6f
0x6c                             0xb4                   0x6c
0x61                             0xf6                   0x61
C:\Users\Joel_\Desktop\ESCOM\Cryptography\Practices\3 - Permutación>
```

Figure 6. Execution of the program showing values on hexadecimal

Discussion:

In this practice, I learned how to use bit-level operations, as such as OR, AND, shift, etc.

For example, to know if a bit is 1 or 0, we can do AND operation by shifting a 1 the number of positions in which we are interested, for example, if we want to know if bit in position 5 (right to left) has a 1, we can do the next operation:

5 AND (1 shifted 4)

The previous instruction is the following on C language:

5 & (1 << 4)

If bit on position number 5 has 1, it gives us a number different than zero, that's why the AND operation was very useful. To "turn on" a bit, we can use the OR operation (obviously, with a 1) and the shifting was useful to operate with the exact bit that we want.

In this case, we start with a 4-character string filled with 0 to only turn on the bit if the bit on the position that the table indicates.

Conclusions:

In this practice, the management of bit level operations was very important, but is kind of difficult (not too much), but this way to program is programing in low level, something hard.

The most useful operation for me, was shifting bits, because we can create a mask to obtain exactly the bit that we want to know if it is or not turn on, or to turn on that bit, or to change it.

In class, our condition was raise 2 to residue (number % 8) with pow function defined on library math.h, but we know that work at bit level is faster, that's why I implemented the same condition but using shifting operator, that's why for me was the most useful operation at bit level.

On section of Code, you can appreciate the operations that I implement, declaring 3 different arrays:

1. To save what the user writes on console
2. To obtain the mixed message and not saving it on 32 characters
3. To re-obtain the original message

Also, I declare another array to save the position of the permutation table (Figure 2), that was to be static because the table is not defined on the standard, but the obtaining of the p^{-1} is showed on the first for loop with a simple assignation.

We can think that DES algorithm provides us security, and it could be, but certainly, 64 bit security nowadays is not too much, this standard has a high level of complexity of implementation, at least, in C language because it uses a lot of shifting, tables, XOR, etc, in conclusion, a lot of bit level operations that we know that are not too easy to implement it, in addition, it has 16 rounds and also 16 keys that we obtain with another number of shifts and then other operations, finally, there are another permutation called Inverse Initial permutation to mix the last bits of the 8 character word.

References:

[1] William M. Dalley, 'DATA ENCRYPTION STANDARD (DES)', 1999, pp. 1 – 12

Code

Permutacion.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char const * argv [])
{
         unsigned char mensaje [4], revuelto [4], nuevo [4], i;
         char resultado, residuo;
         unsigned char p [32] = {1, 26, 15, 14, 13, 22, 7, 8,
                                                29, 2, 16, 31, 21, 6, 23, 9,
                                                28, 17, 3, 20, 5, 24, 12, 10,
                                                27, 18, 19, 4, 0, 25, 30, 11};
         unsigned char p_inversa [32];
         memset (revuelto, '0', 4);
         memset (nuevo, '0', 4);
         memset (p_inversa, '0', 32);
         system ("cls");
         printf ("\n\nEnter a 4-character string: ");
         scanf ("%s", mensaje);
         printf ("\n");
         for (i = 0; i < 32; i ++)
         {
                  resultado = (p [i] / 8);
                  residuo = (p [i] % 8);
                  p_inversa [(p [i])] = i;
                  if ((mensaje [resultado]) & (1 << residuo))
                  {
                           resultado = (i / 8);
                           residuo = (i % 8);
                           revuelto [resultado] = ((revuelto [resultado]) | (1 <<
residuo));
                  }
         }
         printf ("\n\nP");
         for (i = 0; i < 32; i ++)
         {
                  if (i % 8 == 0)
                           printf ("\n");
                  printf ("%d ", p [i]);
         }
         printf ("\n\nP^-1");
         for (i = 0; i < 32; i ++)
         {
                  if (i % 8 == 0)
                           printf ("\n");
                  printf ("%d ", p_inversa [i]);
         }
         for (i = 0; i < 32; i ++)
         {
                  resultado = (p_inversa [i] / 8);
                  residuo = (p_inversa [i] % 8);
                  if ((revuelto [resultado]) & (1 << residuo))
                  {
                           resultado = (i / 8);
                           residuo = (i % 8);
                           nuevo [resultado] = ((nuevo [resultado]) | (1 << residuo));
                  }
         }
         printf ("\n\nOriginal:\t\t\tP:\t\t\tP^-1:\n");
         for (i = 0; i < 4; i ++)
                  printf ("\n0x%x\t\t\t\t0x%x\t\t\t0x%x", mensaje [i], revuelto [i], nuevo
[i]);
```

```
          return 0;
}
```