Cryptography - Lab 0 - DES

Javier Antxon Garrues Apecechea

Student ID: 24647808

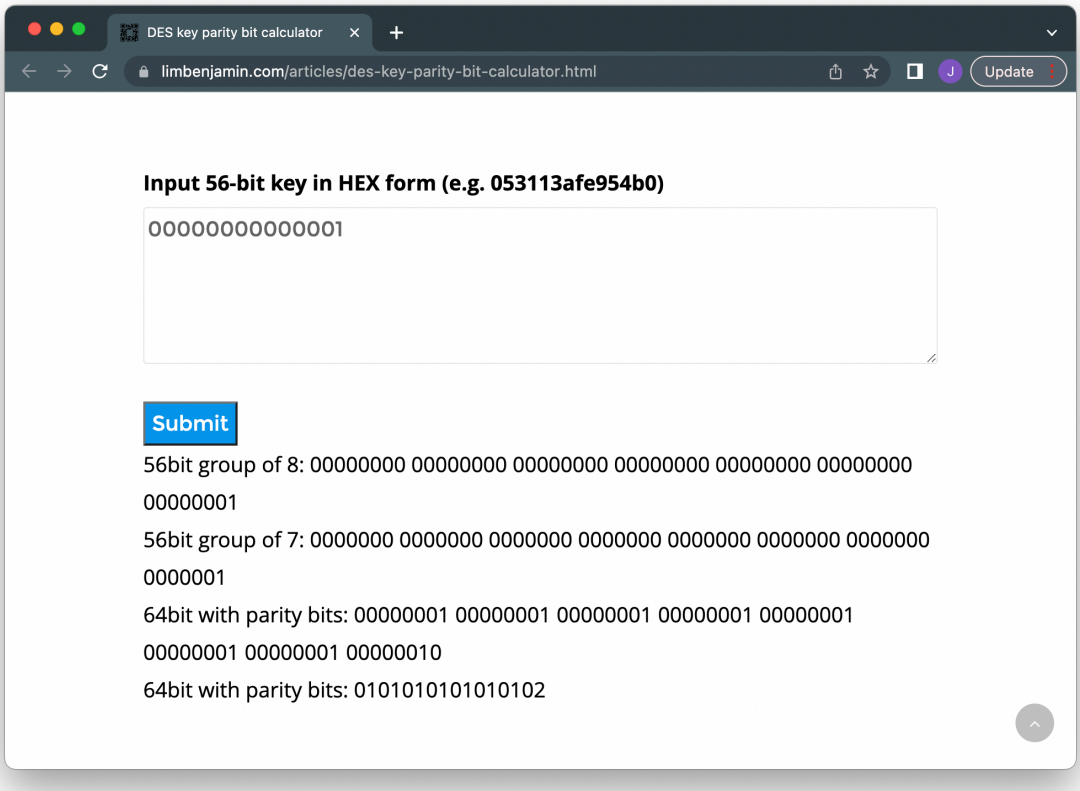
In this lab we will study how the cryptographic algorithm **DES** works.

Generating 56-bit key:

First, we need to generate a 64-bit key starting from a 56-bit key. The extra 8 bits will be added calculating the parity of each of the 8 groups of 7 bits.

In our case, we are going to start with the 56-bit key: **00000000000001**

This is an hexadecimal key and with the help of the tool provided at [this website](limbenjamin.com/articles/des-key-parity-bit-calculator.html) we are able to understand and visualize the underlying process.



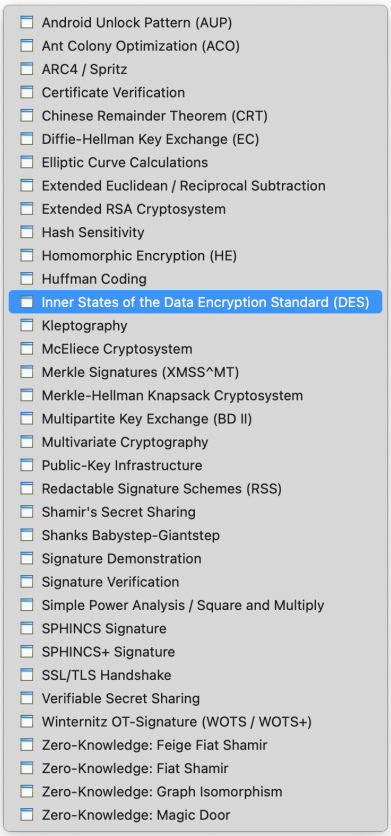
The hexadecimal 56-bit key is transformed to binary, then is divided in groups of 7 bits. Following this, the parity bit of each of the groups is calculated and added at the end of each of the 7 bit groups. At last, the 6- bit binary value is transformed to hexadecimal again.

```
Hexadecimal 56-bit KEY:00000000000001
Hexadecimal 64-bit KEY:0101010101010102
```

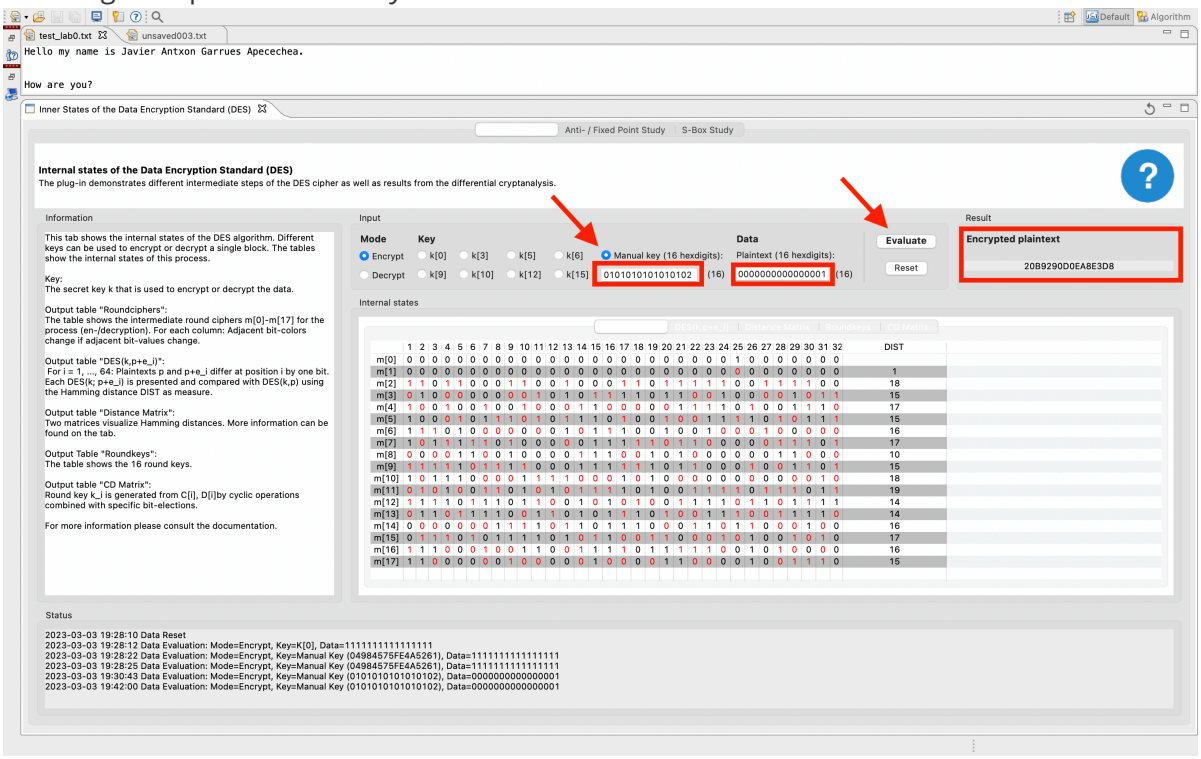
Once this is done, we proceed to make use of the tool JCrypTools to see how a basic plain test can be encrypted. For simplicity reasons, and following the same pattern as we have followed in the key selection, we will try to encrypt the text: **0000000000000001**.

Setting up JCrypTools:

Situated at the main window of the JCrypTools application we will click on *Visuals* and select *Inner States of the Data Encryption Standard (DES)*:



Now, we can proceed to type the key and plain text in the correspondent blanks after selecting the option *Manual Key*:

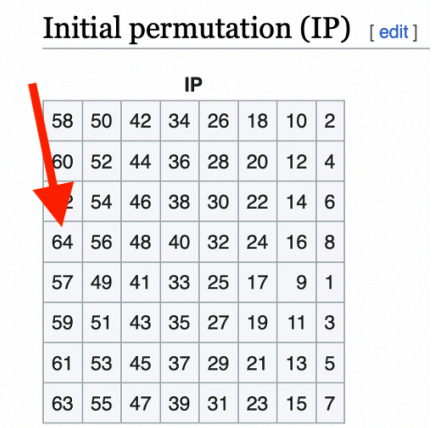


As we can see the **encrypted** plaintext is: **20B9290D0EA8E3D8**.

Regarding the above image we can note a few things:

- m[0] contains the 32 bits situated at the left on the initial permutation of the plaintext. Due to the plaintext only having one value 1, it was expected to only see one of them after the permutation. We can check that this indeed true.

  The binary value 1 moves from the 63rd position to the 25th position of the 64 bit word.



Initial permutation (IP) [edit]

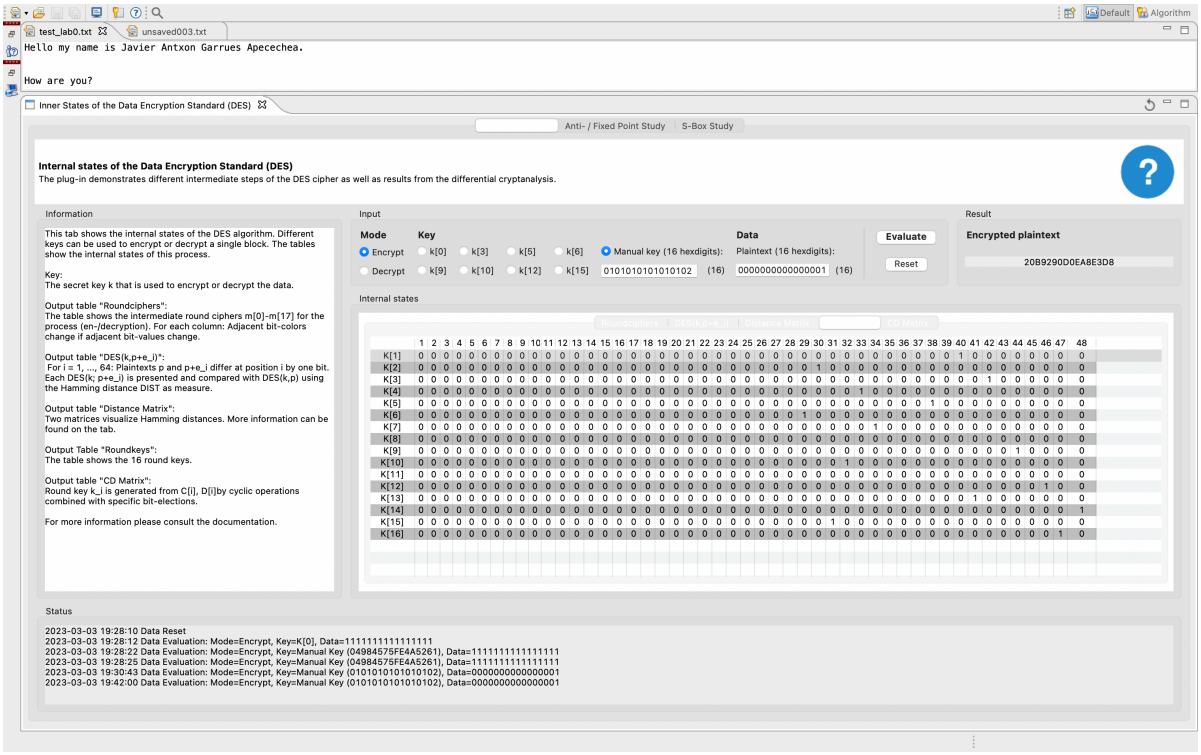
IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

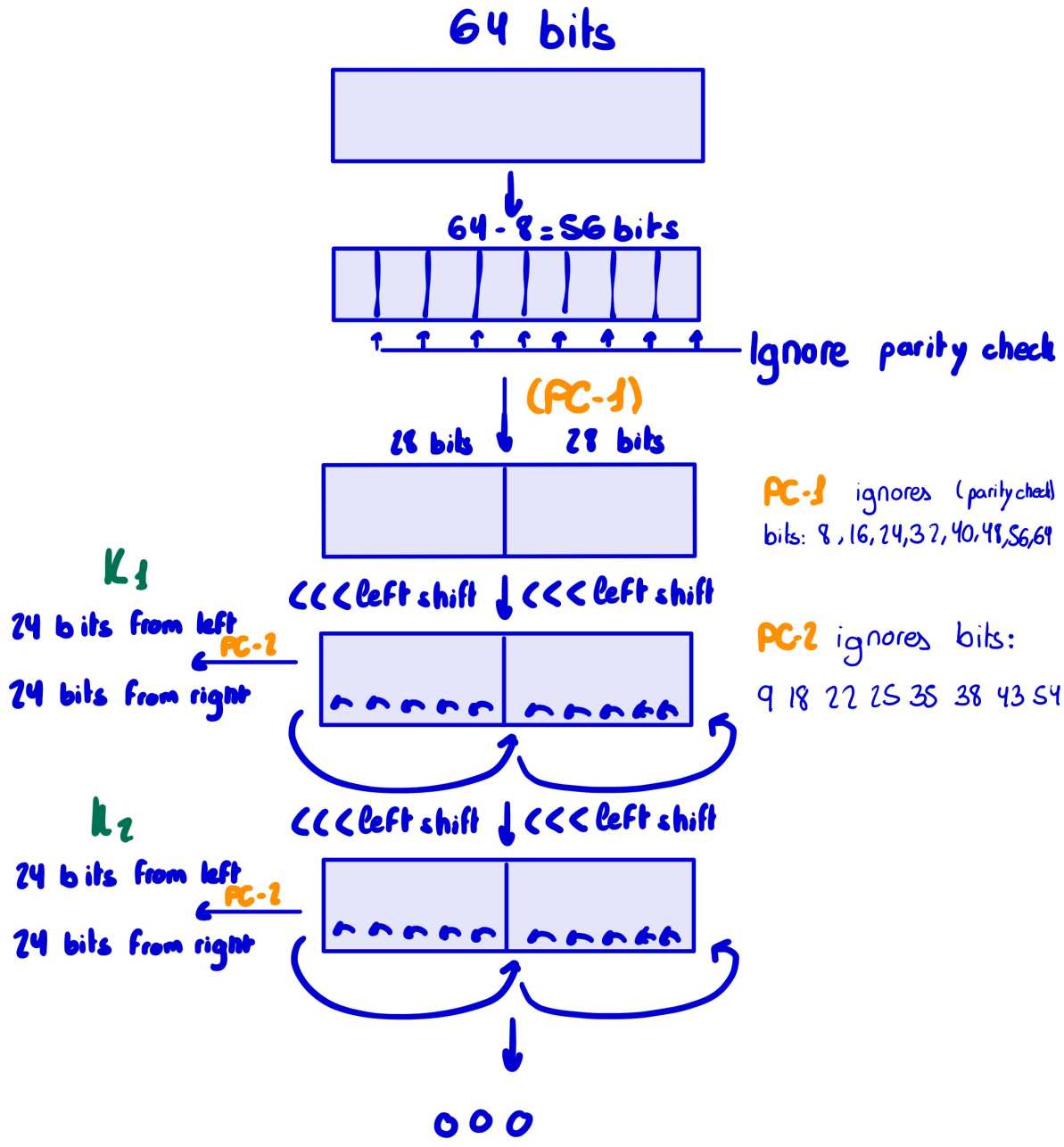
- m[i] contains the left side of the plaintext being encrypted and m[i*2+1] contains the right side.
- The column DIST counts the number of positions in which m[k] and m[k+1] differ. The specific values are highlighted in red.

Now, let's have a look at the round key window. There are **16 rounds** which means that we are going to use 16 subkeys that will be generated through circular shifts and specific bit selections of the original key.

Looking at the roundkeys:

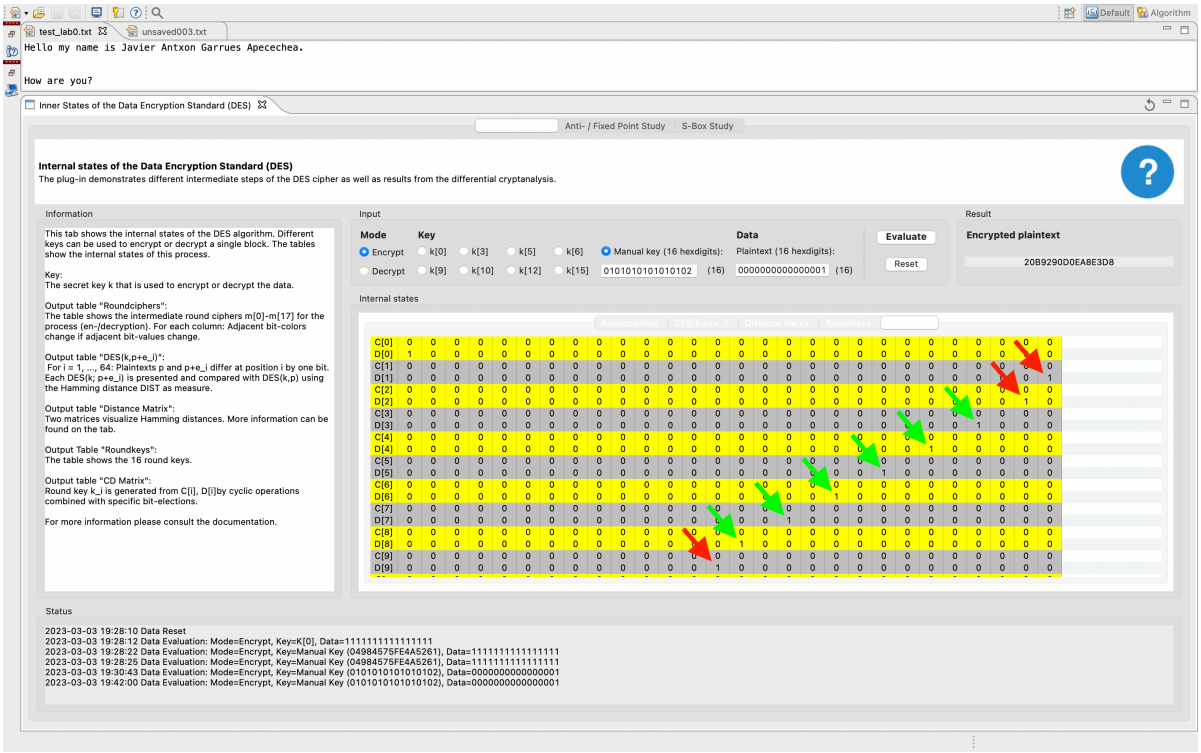


After typing in the original 64-bit, the algorithm reduces it to 56-bit getting rid of the parity bits that had been added (**PC-1**). Once that is done, the key is divided in two halves and left shift is effectuated in both independently. Then, 24 bits from each of the 28 bit halves are selected through **PC-2** and they form the new 48-bit subkey. The following diagram explains the process:


64 bits
64 - 8 = 56 bits
Ignore parity check
(PC-1)
28 bits
28 bits
PC-1 ignores (parity check) bits: 8, 16, 24, 32, 40, 48, 56, 64
K1
<<< left shift
<<< left shift
24 bits from left
PC-2
24 bits from right
PC-2 ignores bits:
9 18 22 25 35 38 43 54
K2
<<< left shift
<<< left shift
24 bits from left
PC-2
24 bits from right
000

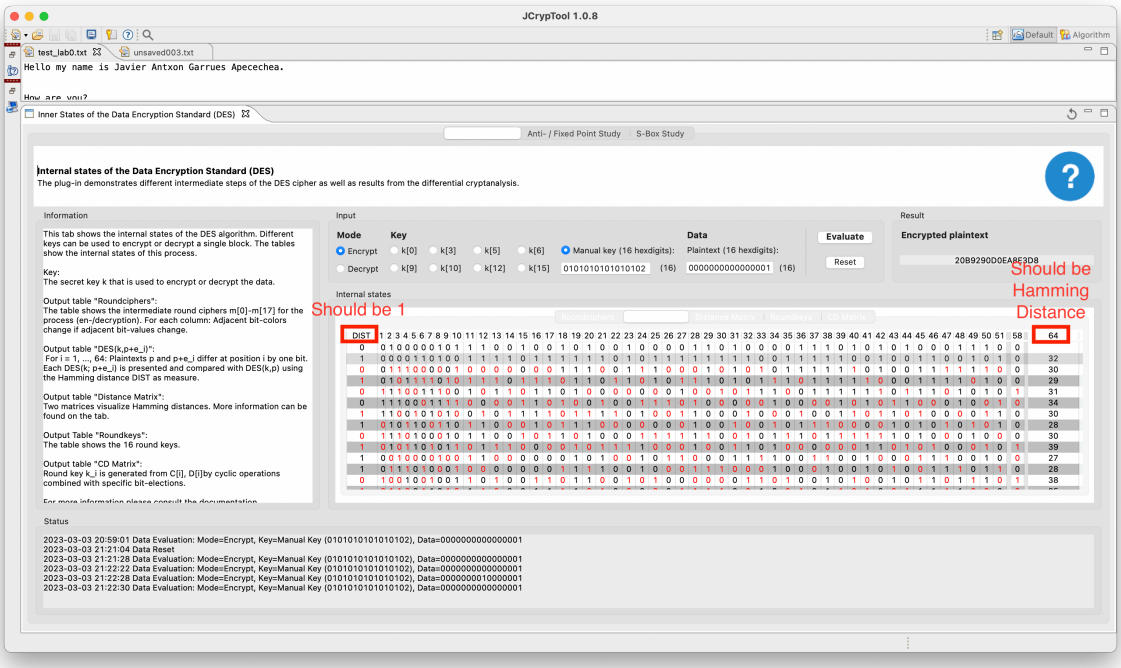
Looking at the CD Matrix:

We can also have a look at another of the tables provided by the app (**CD Matrix**) in which the changes the left and rigth sides of the sub keys experience can be seen. The state shown is after the subkey has experienced the **PC-1** (permutated choice 1) and before experiencing the **PC-2** (permutated choice 2) whose result will be shown in the Roundkeys matrix already seen.



Something that should be noted is that the left shift experience in each round can vary between a shift of 1 (red arrows) or 2 (green arrows).

Looking now at the output table DES(k,p+e_i) we need to note there is a **typo** in which **DIST** should be **index 1** and **index 64** should be **Hamming distance**:

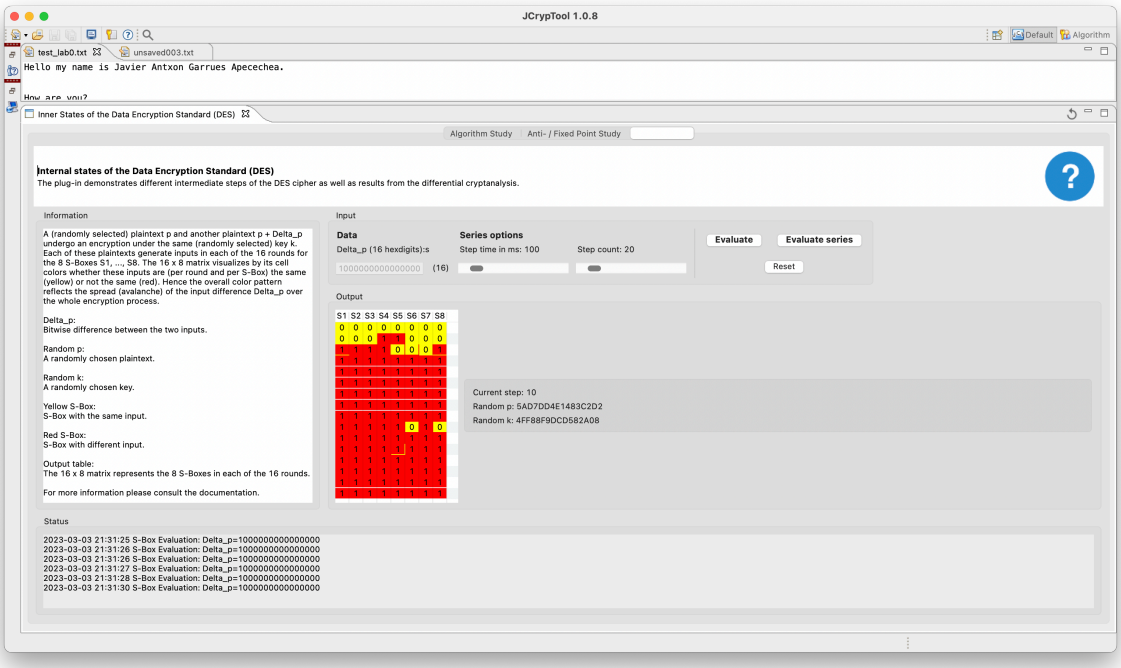


As we saw in the lecture adding all the positions where there are differences is how the Hamming distance is found.

To conclude the lab, we are going to further investigate about the avalanche effect.

Avalanche effect on plaintext:

To do this, we are going to click on the *S-Box Study* window and visualize what happens if we apply the same encryption (same key, same number of rounds) to plaintext p and plaintext p + a modification defined by us. The S-Box Matrix will indicate which S-Boxes will be equal in both cases at the different rounds of encryption. Each row represents a round and each column an S-Box. As it is clear by the following image, a **small modification** such as adding 1000000000000000 **leads** to a **great difference** in the final result.



This process is **precisely explained** in the information box included next to the matrix:

Information

A (randomly selected) plaintext p and another plaintext p + Delta_p undergo an encryption under the same (randomly selected) key k. Each of these plaintexts generate inputs in each of the 16 rounds for the 8 S-Boxes S1, ..., S8. The 16 x 8 matrix visualizes by its cell colors whether these inputs are (per round and per S-Box) the same (yellow) or not the same (red). Hence the overall color pattern reflects the spread (avalanche) of the input difference Delta_p over the whole encryption process.

Delta_p:
Bitwise difference between the two inputs.

Random p:
A randomly chosen plaintext.

Random k:
A randomly chosen key.

Yellow S-Box:
S-Box with the same input.

Red S-Box:
S-Box with different input.

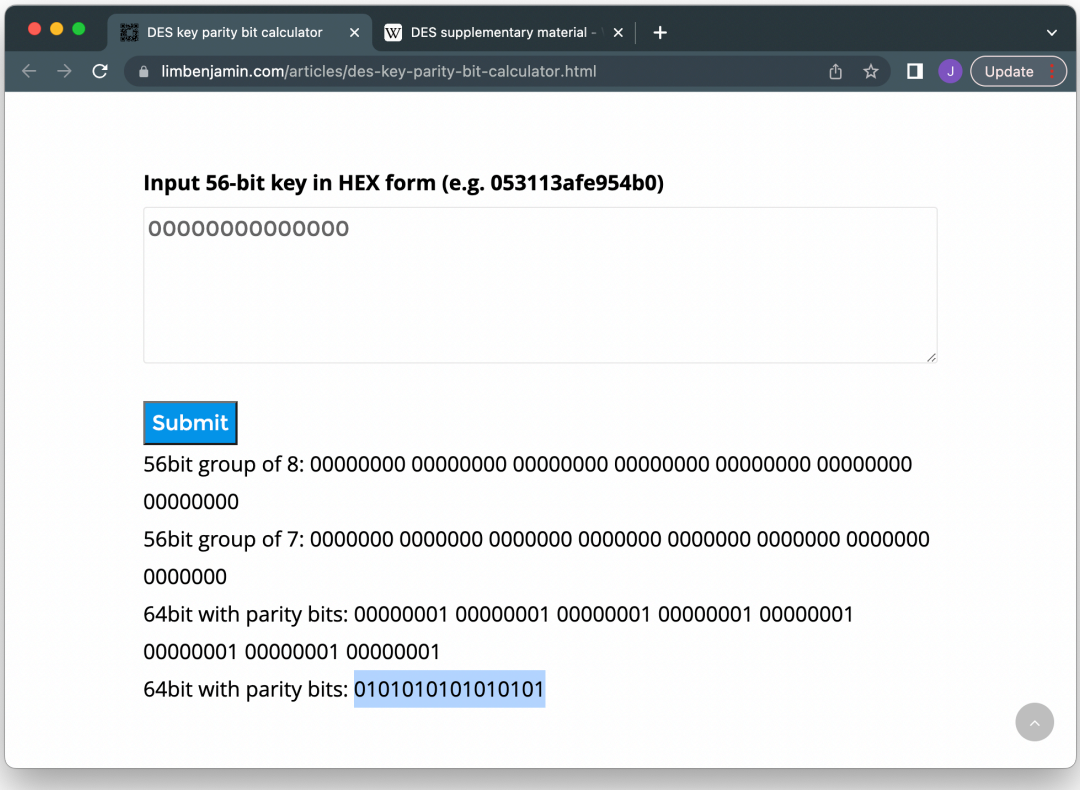
Output table:
The 16 x 8 matrix represents the 8 S-Boxes in each of the 16 rounds.

For more information please consult the documentation.

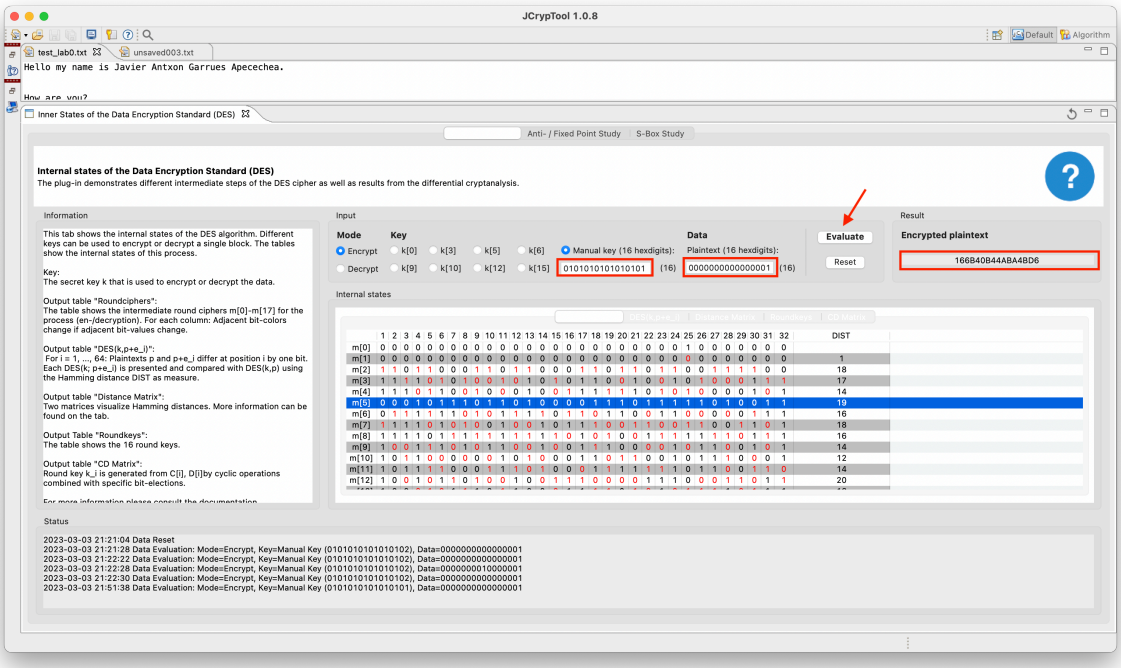
Avalanche effect on key:

In this case, we are going to change 1 bit in the key to see how it affects the result:

```
Original hexadecimal 56-bit KEY:00000000000001
Modified hexadecimal 56-bit KEY:00000000000000
Original hexadecimal 64-bit KEY:0101010101010102
Modified hexadecimal 64-bit KEY:0101010101010101
```



Now, we just have to redo the same process as before typing in the tool the new key maintaining the plaintext constant:



It is clear that the **encrypted plaintext is significantly different** from the original one (20B9290D0EA8E3D8 vs 166B40B44ABA4BD6). Furthermore, we can see how the roundcipher table rows differ from the original output table.

For instance, if we pay attention at the DIST column, we can see how in the newly generated table the values are: 1, 18, 17, 14, 19, 16 ... On the other hand, the values when using the original key where: 1, 18, 15, 17, 15, 16 ... This allow us to see, without looking at the specific values that they are different.

If we want to see if they are significantly different we can extract the same m[i] from both executions and compare them visually:

m[15]	0	1	1	1	0	1	0	1	1	1	1	0	1	0	1	1	0	0	1	1	0	0	0	1	0	1	0	0	1	0	1	0	17
m[15]	1	1	0	1	0	1	0	0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	0	1	0	0	0	0	0		19

At the top, the original value at round 15 (when using key 0101010101010102) and below the one obtained when using key 0101010101010101.

It is clear that there is a big difference. We can find the relative Hamming Distance between both by comparing the specific values for each of the positions and the obtained value is 17.

Final summary:

In this lab we have understood the process the plaintext and key go through during DES encryption. The **plaintext** is divided in left and right m[2 * i] and m[2 * i + 1], expanded, XORed with the key, and further reduced through the S-boxes to 32 bits during the application of the F function. On the other hand, the **key** (after using an online tool to add

the parity bits) is applied the PC-1 (going from 64 to 56 bits), then is divided in two parts and independently shifted left (1 or 2 bits) and PC-2 is applied sequently to obtain the round keys.

We have also learned how to visualize the process through the different output tables provided at the **JCrypTool**. In addition, the **Avalanche effect** has been practically shown in plaintext and key, showing that a small change in one of them leads to great differences in the round ciphers and final encrypted text.