Data Encryption Standard

SIL 765 - Assignment 1

Nichit Bodhak Goel (2017MCS2089)
Shadab Zafar (2017MCS2076)

Project Selection:

A1 = Last 4 digits of entry number of first student = 2089
A2 = Last 4 digits of entry number of second student = 2076

K = A1 + A2 mod 3 = 4165 mod 3 = 1

Project 1: Implementation of DES

Problem statement

You are required to develop a program to encrypt (and similarly decrypt) a 64-bit plaintext using DES. Instead of using an available library, I insist that you program any/every element of each of the 16 rounds of DES (and that means F-box, 32-bit exchanges, generation of sub-key required in each round). Having done that, with one or more 64-bit plaintext(s), verify that indeed the output of the Jth encryption round is identical to the output of the (16-J)th decryption round.

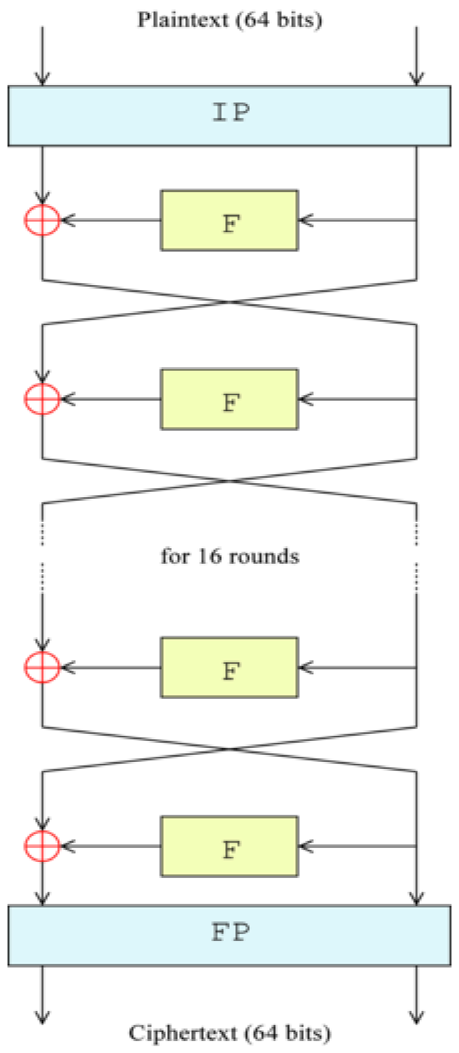
Introduction

DES is a ***symmetric key*** algorithm which is used for the encryption of electronic data. It is a ***block cipher*** which takes in fixed length plaintext and transforms it into ciphertext (of same length) with the help of a key and a series of operations.

DES specifics:-

1) Plaintext :- 64 bits
2) Key :- 56 bits (with additional 8 bits used to store parity of 7-bit blocks at 8th, 16th, … , 64th bits)
3) Ciphertext :- 64 bits

We used Python to implement and validate DES for this project.



ENCRYPTION

- We have implemented a function *encrypt(plain_text, key)* which returns the ciphertext using DES.
- Here we take plaintext to be **4c6f7665676f6f64** and key to be **4e6576696c6c6c65** in hexadecimal format.
- Plain Text: **4c6f7665676f6f64**
  Key: **4e6576696c6c6c65**
- Both plain text and the key is converted in binary format and we get a 64 bit block of plaintext and key. Out of 64 bits of key, 8 bits of key are redundant parity bits.

- encrypt() proceeds in the following manner:
    - Calculate initial permutation of plaintext and split into L and R.
    - Perform 16 rounds based on feistel cipher, with the specified DES f-box and key schedule.
    - Swap L and R, and join them.
    - Apply inverse of the initial permutation to obtain ciphertext.

Step 1:

Function *utils.permute(block, constants.IP)* permutes the input 64 bits using the permutation table IP defined in the *constants.py* file specified as:

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*

From the above table, we get the new arrangement of the bits from their initial order as such the 40th bit of the plaintext will become the first bit of result and similarly, 8th bit of the plaintext becomes the second bit of the result.

Step 2:
After initial permutation, we divide initially permuted block using *utils.nsplit(block, 32)* which will split the block into two 32 bits sub-blocks L and R.

Round 0 - L: ff04ff7a R: 00fe6376

Step 3:
Next we generate all the keys that will be used in the 16 rounds by the function "key_schedule(key)" and storing all the 48 bit subkeys in an array "subkeys".

Using the right 32-bit block "R" and round key "K" we execute the function "feistel(R, K)" which calculates the f box as defined in DES specifications to produce a 32 bit output block. This 32-bit output is XOR'd with "L".

T = L XOR feistel(R, K)

After this, we put L = R and R = T for the next step. R (2724bf52) will become the left sub half (L) after the first round and T will become the right sub half (R). This will continue for 16 rounds.

Output for each round:

```
 0 - L 0: ff04ff7a   R 0: 00fe6376
 1 - L 1: 00fe6376   R 1: 2764bf12
 2 - L 2: 2764bf12   R 2: 18dc334e
 3 - L 3: 18dc334e   R 3: 672ebf32
 4 - L 4: 672ebf32   R 4: 40dc730e
 5 - L 5: 40dc730e   R 5: e706876a
 6 - L 6: e706876a   R 6: 90d42b2e
 7 - L 7: 90d42b2e   R 7: ff6eef62
 8 - L 8: ff6eef62   R 8: 18b64b5e
 9 - L 9: 18b64b5e   R 9: ff66b75a
10 - L10: ff66b75a   R10: 18d4336e
11 - L11: 18d4336e   R11: 3f46c70a
12 - L12: 3f46c70a   R12: d0de2326
13 - L13: d0de2326   R13: ff6e9f02
14 - L14: ff6e9f02   R14: c0f47b4e
15 - L15: c0f47b4e   R15: 2724bf52
16 - L16: 2724bf52   R16: c8d42b5e
```

Li: Left Sub Half after ith round, Ri: Right Sub Half after ith round

Step 4:
In the final step, we swap the left sub half(L16) and right sub half(R16) and then performed an inverse permutation on R + L using "utils.permute(R+L,constants.IP_i)" where IP_i is the inverse permutation table defined in *constants.py* file specified as:

40,8,48,16,56,24,64,32,
39,7,47,15,55,23,63,31,
38,6,46,14,54,22,62,30,
37,5,45,13,53,21,61,29,
36,4,44,12,52,20,60,28,
35,3,43,11,51,19,59,27,
34,2,42,10,50,18,58,26,
33,1,41,09,49,17,57,25

After applying this, we get the ciphertext as **8c8fb94d1bac5358**.

<u>DECRYPTION</u>

Here we take ciphertext to be **8c8fb94d1bac5358** and key to be **4e6576696c6c6c65** in hexadecimal format, and decrypt it using “decrypt(cipher_text, key)”, which results in 64-bit plaintext output.

Cipher Text: **8c8fb94d1bac5358**
Key: **4e6576696c6c6c65**

Both Ciphertext and the key is converted in binary format and we get a 64 bit block of ciphertext and key. The ciphertext is same as output of encrypt() and key is identical to the one used for encryption.

<u>Step 1</u>:
Initial permutation is computed using “*utils.permute(block, constants.IP)*”, as in “encrypt()”.

<u>Step 2</u>:
Next, we divide initial permuted block into halves each of 32 bit and swap the left sub half (32 bits of the ciphertext) and the right sub half (next 32 bits of ciphertext).

<u>Step 3</u>:
Our next step is to generate all the keys that will be used in the 16 rounds by the function “key_schedule(key)” and storing all the 48 bit subkeys in an array “subkeys”.

For i^{th} round of decryption, we use the same subkey which was used in $(16 - i)^{th}$ round of encryption.

Using the round key “K” we perform the inverse of the operation performed while encryption, which operates updates “L” and “R” using “K”.
T = R XOR feistel_box(L,K)

After this, we put L = R and R = T for the next step. R (0fe6376) will become the left sub half (L) after the first round and T will become the right sub half (R). This will continue for 16 rounds.

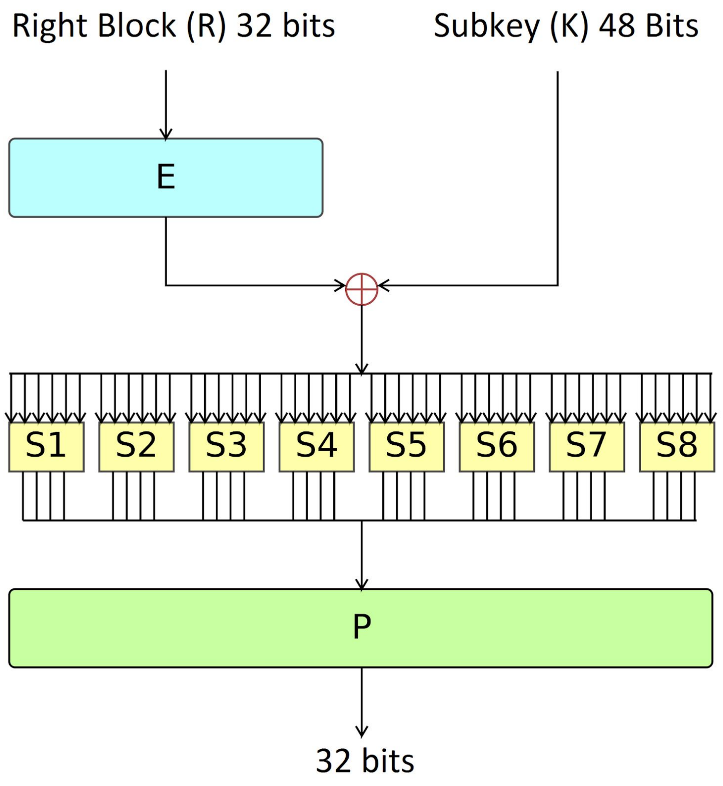
Output for each round:

```
 0 - L 0: c8d42b5e   R 0: 2724bf52
 1 - L 1: 2724bf52   R 1: c0f47b4e
 2 - L 2: c0f47b4e   R 2: ff6e9f02
 3 - L 3: ff6e9f02   R 3: d0de2326
 4 - L 4: d0de2326   R 4: 3f46c70a
 5 - L 5: 3f46c70a   R 5: 18d4336e
 6 - L 6: 18d4336e   R 6: ff66b75a
 7 - L 7: ff66b75a   R 7: 18b64b5e
 8 - L 8: 18b64b5e   R 8: ff6eef62
 9 - L 9: ff6eef62   R 9: 90d42b2e
10 - L10: 90d42b2e   R10: e706876a
11 - L11: e706876a   R11: 40dc730e
12 - L12: 40dc730e   R12: 672ebf32
13 - L13: 672ebf32   R13: 18dc334e
14 - L14: 18dc334e   R14: 2764bf12
15 - L15: 2764bf12   R15: 00fe6376
16 - L16: 00fe6376   R16: ff04ff7a
```

Li: Left Sub Half after ith round, Ri: Right Sub Half after ith round

Step 4:
In the final step, we swap the left sub half(L16) and right sub half(R16) and then performed an inverse permutation on R + L using “utils.permute(R+L,constants.IP_i)” on it to get the plaintext as **4c6f7665676f6f64**.

FEISTEL BOX

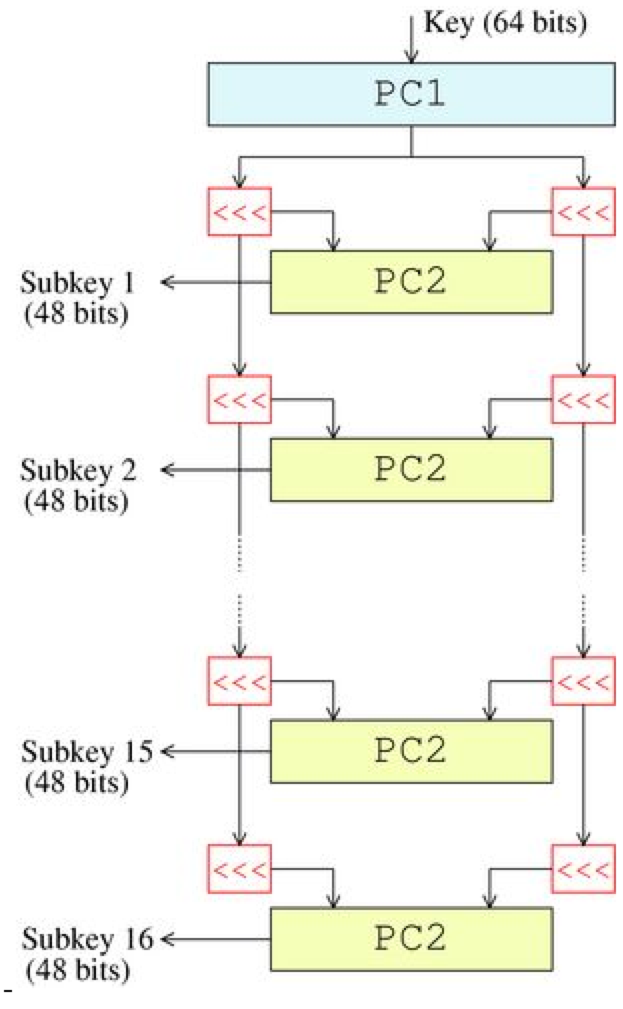


-

- Feistel box is implemented by the function “feistel(R,K)” which returns 32-bit output block from 32 bit right block “R” and 48 bit round key “K”.
- We first expand R to 48-bits using “utils.permute(R, C.EXPAND)” and obtain T.
- Then the expanded block “T” and round key “K”are XOR’d using “utils.xor(T,K)”

- Then S-box is applied to 48 bit input and returns a 32 bit output.
- S-box is implemented by the function “substitute(input)”.
- Finally the function “utils.permute(input, constants.P)” permutes the input using permutation table P defined in constants.py file and calculates the final 32-bit output of feistel function.

KEY SCHEDULE



- All the keys are generated by function “key_schedule(key)” for both encryption and decryption. Here the input key is 64 bit key in string format.

- Firstly, the key will be converted into binary format. Then, the 64 bit key will be converted into 56 bits using PC_1 table defined in constants.py file .
- Then, we will split 56 bit key into two halves “c” and “d”.
- Using half-keys “c” and “d” repetitively, we are going to generate all the 16 subkeys.
- For ith key generation, “c” and “d” will be updated by left-circular shifts following a shifting schedule as defined in the SHIFT array defined in constants.py file.
- After that, we will combine “c” and “d” and permute them according to constants.PC_2 and a new subkey will be generated which we will store at keys[i].
- Finally this “keys" array will be returned.

VALIDATION

To validate the DES implementation, output of the J^{th} encryption round should be identical to the output of the $(16\text{-}J)^{th}$ decryption round.

In function “main()”, we verify this by storing intermediate outputs (L,R) pairs from encryption and decryption and asserting *encry_rounds[i] == decry_rounds[16-i]*, for each i.

For instance, let J = 15: It means output of 15th encryption round should be identical to the output of the 1st round of decryption.

Output of 15th encryption round:

```
15 - LE: c0f47b4e RE: 2724bf52
```

Output of 1st decryption round:

01 - LD: 2724bf52 RD: c0f47b4e

Here, **LE = RD** and **RE = LD.**

TRIPLE-DES (More Secure)

Further, we also implemented Triple-DES where we used three different keys (K1, K2, K3) to encrypt and decrypt the same plain text.

Encryption :- Cipher Text = $E_{K3}(D_{K2}\ (E_{K1}(\text{Plain Text})\ )\ )$
Decryption :- Plain Text = $D_{K1}(E_{K2}\ (D_{K3}(\text{Cipher Text})\ )\ )$

Plain text = 4c6f7665676f6f64
K1 = 626564617a7a6c65
K2 = 4d697261636c6573
K3 = 4c6f67696369616e

Cipher Text = 8cbfb9479ba01b18

De-Ciphered Plain Text = 4c6f7665676f6f64

RESULTS

All assertions in the program are passed and thus, the implementation of DES and Triple-DES is validated.

References

1. FIPS 46-3 - The official NIST document describing DES.
2. DES - Wikipedia