

Introduction

The Data Encryption Standard (DES) is a symmetric-key block cipher that encrypts and decrypts data in 64-bit blocks using a 56-bit key. This report explains the complete DES encryption process, including key generation, initial permutation, 16 Feistel rounds, and the production of the final ciphertext. Furthermore, it describes how the DES decryption method is used to convert ciphertext back into plaintext by using the same key and reversing the encryption process. The code follows the official DES standard, utilizing predefined permutation tables (IP, FP, E, P, PC-1, PC-2) and S-boxes, along with custom-written functions for implementing DES manually.

Overview

The Data Encryption Standard (DES) operates through 3 main phases:

1. Key scheduling: Generates a 48-bit key from the original 64-bit key
2. Data Encryption: Processes the plaintext through initial permutation, 16 Feistel rounds, Final permutation, and Outputs ciphertext and key in hexadecimal format
3. Data Decryption: Processes the ciphertext by reversing encryption using the same key and outputs decrypted text in the original format

```
1 #Start with 64-bit key
2 #Pc-1 Permutation
3 #Split into 28 bits , 28 bits
4 #16 rounds of left shifts
5 #Pc-2 Permutation
6
7
8 #Convert plaintext to binary
9 #Apply initial permutation
10 #Divide it into 2 L and R
11 #Using 16 rounds of Feistel function (F-function)
12 #Expansion: The right half (R) is expanded from 32 bits to 48 bits using an Expansion Permutation (E-table).
13 #XOR with Key: The expanded R is XORed with a subkey derived from the original key. This results in a new 48-bit value.
14 #SBox (substitution): This 48-bit value is then passed through 8 S-boxes, each of which reduces the value back to 32 bits.
15 #Permutation (P-table)==> permuted output Xor with L ==> new R
16 #Swap L and R
17 #Final Permutation
18
19 #Output Ciphertext(Hex)
20 #Output Keytext (Hex)
21
```

Figure 1- Code Overview

Step-by-Step DES Encryption Process

1. Key Generation

The encryption process in DES starts with key generation. Although a 64-bit key is initially provided, only 56 bits are used for encryption, as every eighth bit is reserved for parity checking and thus discarded. The remaining 56-bit key is permuted using the Permuted Choice 1 (PC-1) table, which rearranges the bits according to a predefined pattern. The permuted key is then split into two 28-bit halves. Throughout 16 encryption rounds, these halves undergo a series of left circular shifts. After each shift, the halves are combined and processed through the Permuted Choice 2 (PC-2) table to produce a distinct 48-bit subkey for each round.

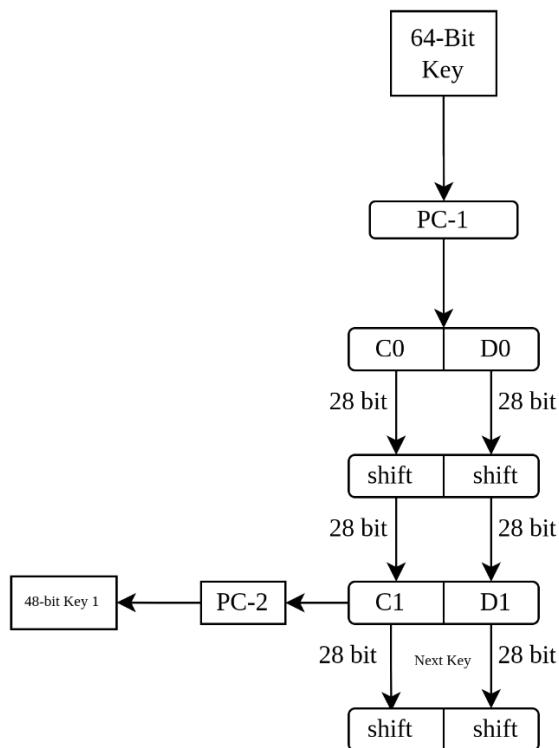


Figure 2 - Key Generation Process

1.1. Starting with a 64-bit key

First, the random library is utilized to generate a random key that serves as the input for the DES encryption process.

```
def generate_key_64bit():
    return ''.join(str(random.randint(0, 1)) for _ in range(64))
```

Figure 3 - Using Random Library for generating a key

1.2 Generating Subkeys

Sixteen subkeys are generated, each 48 bits in length, to be used in the 16 rounds of DES encryption. These subkeys are derived from the original 56-bit key through a series of bit shifts and permutations using the **PC-1** and **PC-2** tables. First, a shift table is defined to specify how many left shifts should be applied in each round (Figure 6). A left shift function is then used to perform circular left shifts on the two 28-bit halves of the key. After each shift, the halves are combined and processed using the Permuted Choice 2 (PC-2) table to produce a 48-bit subkey. A detailed explanation of how subkeys are generated using these steps is provided below.

```
# PC-1 for Key Scheduling
PC1 = [
    57, 49, 41, 33, 25, 17, 9,
    1, 58, 50, 42, 34, 26, 18,
    10, 2, 59, 51, 43, 35, 27,
    19, 11, 3, 60, 52, 44, 36,
    63, 55, 47, 39, 31, 23, 15,
    7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4
]
```

Figure 4- Adding the PC-1 Table

```
# PC-2 for Key Scheduling
PC2 = [
    14, 17, 11, 24, 1, 5, 3, 28,
    15, 6, 21, 10, 23, 19, 12, 4,
    26, 8, 16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55, 30, 40,
    51, 45, 33, 48, 44, 49, 39, 56,
    34, 53, 46, 42, 50, 36, 29, 32
]
```

Figure 5- Adding the PC-2 Table

```
# Standard Shift table for left shifting
SHIFTS = [1, 1, 2, 2, 2, 2, 2, 2,
           |   |   | 1, 2, 2, 2, 2, 2, 2, 1]
```

Figure 6- Adding the Shift Table

```
def left_shift(bits, n):
    return bits[n:] + bits[:n]
```

Figure 7- Left shift function for splitting bits

```
def generate_subkeys(key64):
    key56 = permute(key64, PC1)
    C, D = key56[:28], key56[28:]
    subkeys = []
    for shift in SHIFTS:
        C, D = left_shift(C, shift), left_shift(D, shift)
        subkey = permute(C + D, PC2)
        subkeys.append(subkey)
    return subkeys
```

Figure 8- Generating Subkeys

- **PC-1 Permutation (64-bit to 56-bit)**

The original 64-bit key is processed through the **Permuted Choice 1 (PC-1)** table, which discards 8 bits (parity bits) and reduces the key to 56 bits.

- **Split into C and D**

The 56-bit key is divided into two 28-bit halves:

- **C:** The left 28 bits
- **D:** The right 28 bits

- **Left Shift Function**

A left shift function is applied to both halves (C and D). This function shifts the bits in a circular manner to prepare them for further processing.

- **16 Rounds of Left Shifts**

Over the course of 16 rounds, the two halves undergo left shifts. The shift amounts are based on a predefined **shift table**:

- **Rounds 1, 2, 9, 15:** Each shift is by **1 bit**
- **Other rounds:** Each shift is by **2 bits**

- **PC-2 Permutation (64-bit to 48-bit Subkey)**

After the left shifts, the two halves are combined and processed through the **Permuted Choice 2 (PC-2)** table. This step reduces the combined 56-bit key into a 48-bit subkey for each round.

- **Adding Generated Subkeys to the List**

The 48-bit subkeys generated in each round are added to a list, and these subkeys will be used in the 16 rounds of DES encryption.

2. DES Encryption

Now that the subkeys are generated, we proceed with the DES encryption process. This involves block processing, Feistel rounds, and ciphertext conversion.

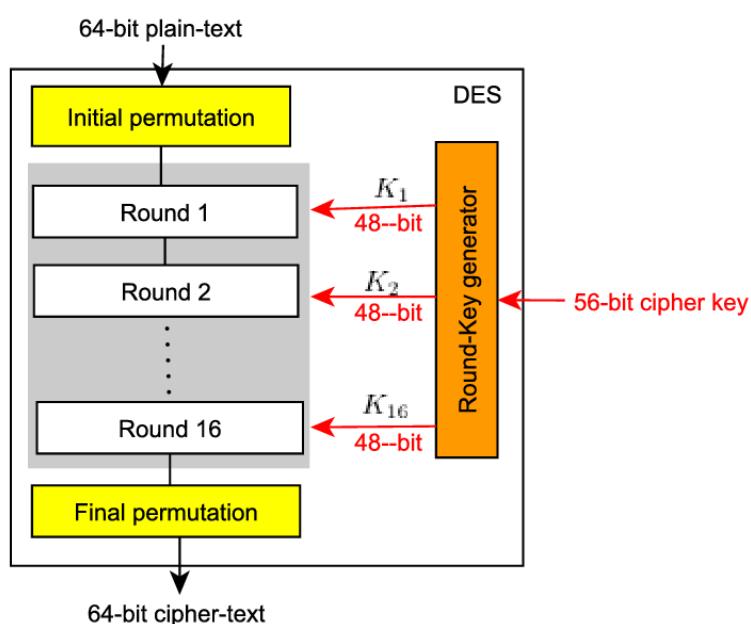


Figure 9- DES Encryption Process

2.1 Block Encryption

The core of the DES operates on 64-bit blocks of plaintext. The encryption process consists of multiple stages, starting with an initial permutation and proceeding through 16 Feistel rounds, ultimately producing a 64-bit ciphertext. The core of DES will work on 64-bit blocks. The following steps are added to the code, as can be seen in the following pictures:

```
def encryption_block(block, subkeys):
    permuted = permute(block, IP)
    L, R = split_bits(permuted)
    for key in subkeys:
        L, R = feistel_round(L, R, key)
    combined = R + L # Swap after last round
    return permute(combined, FP)
```

Figure 10- Block Encryption

2.1.1 Initial Permutation (IP)

```
# Initial Permutation Table (IP)
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
]
```

Figure 11- Using fixed permutation table

2.1.2 Splitting into L and R 32-bit halves

```
def split_bits(bits):
    #Spliting 64 bits into 2 parts (32 bits / 32bits)
    return bits[:32], bits[32:]
```

Figure 12- L and R 32-bit halves

2.1.3 Feistel Round Function: which will be run 16 times

The Feistel function is applied for 16 rounds, each consisting of the following steps:

```
def feistel_round(L, R, subkey):
    R_expanded = expand_right_half(R)
    xor_result = xor(R_expanded, subkey)
    sbox_result = s_box_substitution(xor_result)
    p_result = permute(sbox_result, P)
    return R, xor(L, p_result) # Swap L and R
```

Figure 13- Feistel Round Function

- a) The 32-bit right half is expanded to 48 bits using the E table.

```
def expand_right_half(R):
    return ''.join(R[i - 1] for i in E)
```

Figure 13a- Expansion

- b) Align R size (48-bit) with the 48-bit subkey for XOR operation.

```
#Using XOR
"""0 ^ 0 = 0

0 ^ 1 = 1

1 ^ 0 = 1

1 ^ 1 = 0 """
def xor(a, b):
    return ''.join(str(int(x) ^ int(y)) for x, y in zip(a, b))
```

Figure 13b- Using XOR

- c) S-BOX substitution

```

# All 8 DES S-boxes
S_BOXES = [
    [[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
     [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
     [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
     [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],
    [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
     [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
     [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
     [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],
    [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
     [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
     [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
     [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],
    [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
     [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
     [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
     [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],
    [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
     [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
     [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
     [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],
    [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
     [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
     [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
     [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],
    [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
     [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
     [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
     [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],
    [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
     [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
     [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
     [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]
]

```

Figure 13c- Defining S-BOX substitution Table

```

#Using s_box_substitution it takes a 48 bits binary string and splits it into eight 6-bits chunks
def s_box_substitution(bits):
    #Action: Splits the 48-bit input into 8 chunks of 6 bits each (since 48/6 = 8).
    chunks = [bits[i:i + 6] for i in range(0, 48, 6)]
    output = ''
    #i is the chunk index (0 to 7), selecting one of the 8 predefined S-Boxes.

    #row and col index into the S-Box to fetch a 4-bit number.
    for i, chunk in enumerate(chunks):
        row = int(chunk[0] + chunk[5], 2)
        col = int(chunk[1:5], 2)
        output += format(S_BOXES[i][row][col], '04b')
    return output

```

Figure 13c- S-BOX substitution Function

d) Permuting the Sbox_result with the P table

```
def permute(bits, table):
    return ''.join(bits[i - 1] for i in table)
```

Figure 13d- Sbox_result with the P table

```
# P Permutation Table
P = [
    16, 7, 20, 21, 29, 12, 28, 17,
    1, 15, 23, 26, 5, 18, 31, 10,
    2, 8, 24, 14, 32, 27, 3, 9,
    19, 13, 30, 6, 22, 11, 4, 25
]
```

Figure 13d- Sbox_result with the P table

- e) The left half (L) is XORed with p_result

```
p_result = permute(sbox_result, P)
return R, xor(L, p_result) # Swap L and R
```

Figure 13e- XORed with p_result

- f) The halves are swapped: (L, R)

2.1.4 Swap After the last round and Final Swap with the FP table

```
# Final Permutation Table (FP)
FP = [
    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, 9, 49, 17, 57, 25
]
```

Figure 14- Final Permutation Table (FP)

2.2 Full DES Encryption Process

The block encryption is applied to the entire plaintext after conversion to binary. The following steps are used to perform this process:

```
def encrypt_des(plaintext, subkeys):
    #Convert text to binary
    binary_data = binary_converter(plaintext)
    cipher_blocks = []
    for i in range(0, len(binary_data), 64):
        block = binary_data[i:i+64]
        cipher_blocks.append(encryption_block(block, subkeys))
    cipher_bin = ''.join(cipher_blocks)
    #Convert to Hex:
    # Converts the decimal number to a hexadecimal string
    # it means we add 2 for binary and then slice off 0x from the hex string to just correct character
    hex_cipher = hex(int(cipher_bin, 2))[2:1].upper()
    return hex_cipher
```

Figure 15- DES Encryption

2.2.1 Binary Conversion

The plain text is encoded into UTF-8 and converted to a binary string.

```
#Convert text to binary using UTF-8 (supports Unicode).
def binary_converter(text):
    #Using List
    binary = []
    for char in text:
        # Encode as UTF-8 (1-4 bytes per char)
        for byte in char.encode('utf-8'):
            #Encode your Character to 8bits
            binary.append(f"{byte:08b}")
    #Concatenated all the strings in binary
    return ''.join(binary) # Continuous binary string
```

Figure 16- Binary Conversion

2.2.2 Block Process

The binary data is split into 64-bit blocks, and the `encryption_block()` function will be run here.

2.2.3 Hexadecimal Output

The ciphertext binary string is converted to hexadecimal

2.3 Output

This phase handles the final steps of the DES encryption process, saving the generated key and ciphertext to files and displaying the results for verification.

The detailed steps are explained below:

```
def main():
    #Encryption
    with open("/opt/DES/Task3/plain.txt", "r", encoding ='utf-8') as f:
        |     plaintext = f.read().strip()

    key64 = generate_key_64bit()
    subkeys = generate_subkeys(key64)
    cipher_hex = encrypt_des(plaintext, subkeys)
    key_hex = hex(int(key64, 2))[2:].upper()
    #Save Key text here:
    with open("/opt/DES/Task3/key.txt", "w") as f:
        |     f.write(key_hex)
    #Save encrypted text here
    with open("/opt/DES/Task3/cipher.txt", "w") as f:
        |     f.write(cipher_hex)
    print("Original Text:", plaintext)
    print("64-bit Key:", key64)
    print("Encrypted Cipher (HEX):", cipher_hex)

if __name__ == "__main__":
    print("This is Task3-enc.py")
    print("Kimia Sadat Karbasi - Student ID Number ='60393958''")
    main()
```

Figure 17- Encryption Final Steps

- Reading the Plaintext ()
- Key Generation: A 64-bit key is randomly generated
- Subkey Generation: The 64-bit key is expanded into 16 subkeys
- Encryption: The plaintext is encrypted using encrypt_des()
- Saving Outputs: The 64-bit key and encrypted result are saved into the defined path
- Console Output: Display the original text, 64-bit key, and ciphertext

2.4 Execution and Results

Here are all the encryption code screenshots which were executed on a Linux machine. Each Picture has caption that indicates the corresponding step in DES encryption process.

```

root@Docker-PAM:/opt/DES/Task3# python3 Task3-enc.py
This is Task3-enc.py
Kimia Sadat Karbasi - Student ID Number ='60393958'
Original Text: "The Clockmaker of Rothenburg"
In the heart of Bavaria, nestled between rolling green hills and
cobblestone streets, sat the town of Rothenburg ob der Tauber.
Known for its perfectly preserved medieval walls and timber-framed
houses, the town looked like a postcard from the past. But what truly
gave Rothenburg its charm was the little clock shop on
Schmiedgasse Street, run by the old clockmaker, Herr Baumann.
Herr Baumann had lived in Rothenburg all his life. His small shop,
Zeit & Seele-Time and Soul-was filled with all kinds of clocks:
cuckoo clocks with dancing figurines, grandfathers with deep chimes,
and delicate pocket watches that glimmered like stardust. People
came from all over Germany-and even farther-to see his creations.
But what they didn't know was that Herr Baumann's clocks held more
than gears and springs. They held memories.
It was said that when Herr Baumann built a clock, he carved into it a
tiny piece of someone's story. A couple in love, a soldier coming
home, a child's laughter-he somehow captured these moments, and
his clocks ticked with life, echoing not just time but feeling.
One chilly April morning, a young boy named Lukas wandered into
the shop. He was ten, wearing a red scarf too big for his neck, and
had curious eyes that never stopped moving.
"Guten Tag," said Herr Baumann with a smile. "Looking for
something?"
Lukas shook his head. "Just looking. Papa says you make magic
clocks."
Herr Baumann chuckled, kneeling down so his eyes met Lukas's.
"Magic? Well, maybe a little."
The boy's smile faded. "Do you make clocks that can go backward?"
The clockmaker blinked. "Backward?"
```

Figure 18 - Plaintext

```
64-bit Key: 7B7932A56E4DA3CC
```

Figure 19 - bit Key (Hex Format)

```

Encrypted Cipher (HEX): 1092AC7CEFC159626C5784CFB6171247964C894C577B0470B7B3D9B81505936589F5D
4FC0DDA84343A0367D2F9058FEC7702420BDC38B46C8F899A6ABC5D4F5BF67F1AB7566184FB97155F1E1432B7BC1
9011210EC8873AD8603012BA4A6958EF7968CBF1D00714588722BA08C995E07F2A13C9216F5B9A917A90285D61E2
3B40C36688A8171A75C929A9578A121F1D6B447D79F2F2940AC9973BF4138008569B5E22E60E146455684DDC53F8
77988F47DD208725294C09E9A85777C53305ACC5D8982CD95610F8456B671845FABCFC310912ADE94E34C70367E
25AE9C838C5303AD3142780CA267783C45991176DD31940C524618BAS945378200459E6406AF3BE165D2ADA13B
C0FDDB8CE5D87AA86827451B88686A04701BDE9E9BAE90D2209F284982A16A823D3A1AD80A2C5C929A9578A121F1878
3AC2A57182D94BF08964CC1EC91F815B9D06392017636A4BD02FAA1EDA0E955C5A0ACF69BAD33387941B84F08B69C9
419FBFD74D48B15CA4471D361D396980480165A47939AE3E33CF6D39434F8DCB6D40B4F66529F50FFB43178D0A6
F114EBCFA055E1B2F1FA39E4E206F88B6652301382B771E5A53F0A6669FAD27C81D5608A2A5606BCB
6A2EC41B3D2CA9426B5C83C238AE118BE92FDA878042EA50E2E1EC0464120AFEB7C6B7434232028FC22EBC0C34C
8FE4CF12FEF273A587E775C8A2FEA6D8B04C854DE826E5297B301C7E50C289343DB903D413F789A1AD66D105AE7
F87757BE62FE1F7AEEB509F4FB1BC655F2552F5AD79B554C247E33B9732B3507753FA28AE6A866117291C8EAS
E52822344B2AA75CA1AD940AE7C23E3190DB75F32B94868E2A93A29649D2FEDD3014C9E4319272CB0422D9F247
F0D90B07E9E78F80D61D18340DEB483976169B51F72BEA2C0E35E611C80A54C1554C9B079E3F88EBD364F930E753
C173DB515062F5C7621DE36CA905676B632A141EA2C04C698FC23E422A0E97B1E2E739073B6A0676F8A497A27
01F9B0603151BAB2F41FAB8B32EACF69394511F0C84E8F510F4A6E2D0B1E97B1E2E739073B6A0676F8A497A27
8BE435001C0688906504C669C92E1FA39E4E206F88174CF744FCF0A705D4F717E128844F0430798527E6A1B37E2C36
5F52B8580DEF6CB6807D6DAFB29840B022D788B680594EDB22583EAF337895EFD8540C9796563680540F3469991FF
CC2178D192973CA233F88142F3635134A5E075D02B289E0528E1557123A1C8F45009228E15C59B5185F6C506C0
721EFD4E280D6585A49F7B7148AFCFA42E0C3D088E3562332F17F5029E88FB435D054B394E37EC9042062B2BC
5A768C38BE228944CF49044A0F34620D26562D01B2F6E875E225E1B74851EDC8149862E5447613183FBBF2217A825
A8C8EEA7B77FDB493086E43AEDFDE86008E7FE0248CC833795B35152E50B8393027993854BCF5B2010EA901B64
C3D0AFE78C1B2B6A8258C9440D48C5A58A769D9F4F80BCFED46E6A2666248CED243C8DB3D6F22E
606991E9FA440CE7975C9F0DB34EC6B5C72D31BFB6D358194790CDAE0FCBAFZ4A9B86C5EB7248E619B81176D3378A
5FD2791F1FD40956C58CC828993361A3B81D40B9707C14383598B8A74805C4D08910710741DE1E0C1EE5754745
F6EC20A2D607E1BCF3FE8BF3052C69C073F92B61B46759D443B81EA580E7D57433076909A655233E264D0716035
9796E6A962AB914454419C6180EFCFF69A376D891A913071671DA3544320DCE2F86469D12DF74322AF271B629B89E
9705EA26FEB05A810E61ACE04523BE60EFCB9E113BE559B6E306948265E7364ADB784F5CC116F923C0BDE4F
F843178D0D60A6F11AD72A0E56E42CB6D142170D51555A1A14DF41F906B8852E0FC2CC7AFC7D13FCC1E6ABDF70C
B1B2C469A78BB93F1DB6E0B8926EAB027D56F6C62E5969620F1909841899B8202BFCB8E4EF4C14CF7A127987D4C2
```

Figure 20 - Encryption Output (Ciphertext)

```

root@Docker-PAM:/opt/DES/Task3# ls
cipher.txt  key.txt  plain.txt  Task3-enc.py
```

Figure 21 - File Output (key.txt)

```

root@Docker-PAM:/opt/DES/Task3# cat key.txt
7B7932A56E4DA3CCroot@Docker-PAM:/opt/DES/Task3#
```

Figure 21 - File Output (key.txt)

```

root@Docker-PAM:/opt/DES/Task3# cat cipher.txt
1092AC7EFC159626C5784CFB6171247964C894C577B0470B7B3D9B81505936589F5D4FC0DDA84343A0367D2F9058
FEC7702420BDC38B46C8F899A6ABC504F5BFF67F1AB7566184FB97155F1E1432B7BC1901121DEC8873A0D86D3012BA
4A6958EF7968CBFDD07145887228A08C995E072A13C9216F5B9A917A90285D6E1E23B40C36688A817A75C929A9
578A121F106BF447D79F2F2940AC9973BF413800856985E22E60E146455684DDC53F877988F47D208725294C09E9
A85777C543305ACC5D0892CD95610FE8456B671B45FABC310912ADE94E34C70367E25AE9C83C3503DEAD314278
0CA267783C45991176D31940C5246188A5F945378200459E6406A3BE165D2AD13BC0FDBD8CE5D87AA86827451B
8686A04701BDE9EBE9DD2209F2849B2A16A823D3A1ADB0A2C5C929A9578A121F18783AC2A57182D94BF08964CC1E
C91F815BF063920176364A8D02FAA1EDA4E955C5A0ACF69B8D33387941B84F08B69C9419FBFD74D48B15CA4471D36
1D396980480165A47939A6E333C6D39434F80CB6D40B4F66529F50F843178D06A6F114EB8CFA055E1B2F42E1FA
39E4E206F88B652BA35253013828B771EA53C1FAF669GD27C81D5608A2A56D6B86A2E4C183D2A9426B5C3C2
38EAE118BE92F0A878042E50E21EFC0464120AFEB8C7C6B7434232028FC0C34C8FE4CF12FF273A587EA775C
8A2FEA6D804C854DE8265297B301C7E50C289343D903D413F789A1AD66D105AE7F87757BE62F1F7AE8509F4
FB1BC655FE2552F5A0D798554C247E3389732B35077573FA28AECEA866117291C8E5E52822344B2A75CA1AD8949
AE7C3E319DB75F32B94868E2A93A29649D2FEDD3014C9E4319272CB422D9FD247F0090B07E8E78F80D61D1834
0EDEB483976169B51F72B8E2C0E35E611CB0A54C1554C9B079E3F88EBD364F930E753C173DB515062F5C7621DE36C
A905676B632A141EA2C04C698F23E422A0EE485D887F9A0B6C503B9028948CDC5E2B01F9B06030151BA82F41FAB8
B32EACF69394511F0C84EBF510F446E2D81091E97B1E2F739073B8A60676F8A497A278BE435D002068B906504C66
9C92E1FA39E4E206F88174CF744FCFA0705D4E717E12B8844F043079852E1A1B37E2C365F52BB58DEF6C86807D6DAFB
29840B02D0788B680594ED8225B3EAF337895EFD854DC978656368054DF3469D91FFC2178D192973CA233F86142
F3635134A5E8075D02B2898ED528E1557123AA1C8F45009228E15C59B5185F6C5D6C0721EFD4E28D9D65854A9F7
B7148AFCFA42E0C3D08322F17F5029E88FBAD5E054B394E37EC9042062BCC5A768C38BE22894CF49044A
0F34620D26562D012B2F6E875E225E1B7485E25E13183FBF217A825A8CBEEA777FD8493086E3A
EDFDE86D08E7F0E248C8337953B35152E5088393027993854BCCF5B2010EEA9D1B64C3DAFED78C12B86C38289AC
4C6A8258C944020A8CDA58A769D9F4F48DBCFED46E6A2666248CED243C8D83D6E2F2E606991E9FA440CE7975C9FDB
034EC6B5C72D31BF86D358194790COAEFOCBF24A9B86C5E87248E19B117603378A5FD2791F1FD40956C59F8CCC
82899361A3B814383598BEA8748056CA0810710741DE1E0C1E5754745F6EC20A2D607E1BCF3F88F3
052C69CD732F9B261B46759D443B81E5A80EE7D57433076990A65523E264D7A16035979E6A962A891445419C618
0EFCF69A376D891A913071671A354432DCE02F86469012DF74322A7B162989E9705E26FEB05AC8AB10E61A
CCE04523B8E60EBCF9E113BE559BD6E306948265E73644DB784F5CC116F923C0BDE4FF843178D0A6F11AD72A0E5
6E42C8D6142170DD51555A1A14DF1F096BE8852E0FC2CC7AFC7D13FC1E6ABDF70CB1BC2469A78BB93F1DB6E0B8

```

Figure 22 - File Output (cipher.txt)

3. DES Decryption Process

After generating key.txt and cipher.txt, we proceed with the DES decryption process. This involves block processing, Feistel rounds, and applying inverse operations to recover the original plaintext.

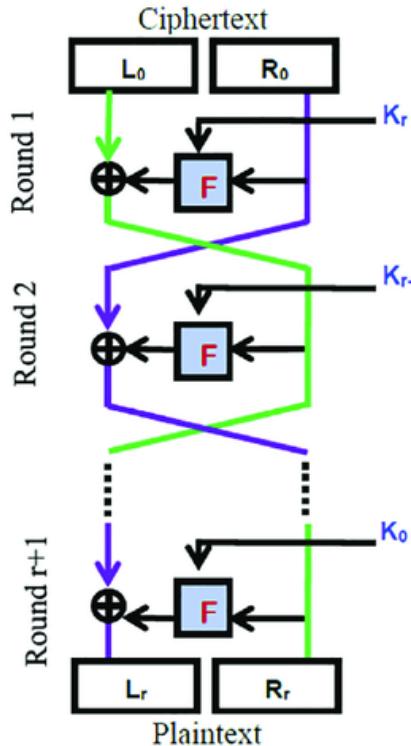


Figure 23 – DES Decryption

3.1 Block Decryption

The DES decryption process operates on 64-bit blocks of ciphertext with the same key we generated, reversing the encryption steps through:

```
162     def decryption_block(block, subkeys):
163         permuted = permute(block, IP)
164         L, R = split_bits(permuted)
165         for key in reversed(subkeys):
166             L, R = feistel_round(L, R, key)
167         combined = R + L # Swap after last round
168         return permute(combined, FP)
```

Figure 24 – DES Decryption

3.1.1 Initial Permutation (IP): Using fixed permutation table

```
# Initial Permutation Table (IP)
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
]
```

Figure 25 – Initial Permutation (IP)

3.1.2 Splitting into L and R 32-bit halves

```
def split_bits(bits):
    #Spliting 64 bits into 2 parts (32 bits / 32bits)
    return bits[:32], bits[32:]
```

Figure 26 – Splitting into L and R 32-bit

3.1.3 Feistel Round Function: which will be run 16 times in reverse order (k16 --> k1)

```
def feistel_round(L, R, subkey):
    R_expanded = expand_right_half(R)
    xor_result = xor(R_expanded, subkey)
    sbox_result = s_box_substitution(xor_result)
    p_result = permute(sbox_result, P)
    return R, xor(L, p_result) # Swap L and R
```

Figure 27 – Feistel Round Function

- a) The 32-bit right half is expanded to 48 bits using the E table.

```
def expand_right_half(R):
    return ''.join(R[i - 1] for i in E)
```

Figure 27a – Feistel Round Function

- b) Align R size (48-bit) with the 48-bit subkey for XOR operation.

```
#Using XOR
"""0 ^ 0 = 0

0 ^ 1 = 1

1 ^ 0 = 1

1 ^ 1 = 0 """
def xor(a, b):
    return ''.join(str(int(x) ^ int(y)) for x, y in zip(a, b))
```

Figure 27b – XOR operation

- c) S-BOX substitution

```
# All 8 DES S-boxes
S_BOXES = [
    [[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
     [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
     [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
     [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],
    [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
     [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
     [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
     [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],
    [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
     [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
     [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
     [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],
    [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
     [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
     [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
     [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],
    [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
     [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
     [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
     [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],
    [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
     [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
     [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
     [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],
    [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
     [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
     [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
     [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],
    [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
     [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
     [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
     [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]
```

Figure 27c – S-BOX substitution

```
#Using s_box_substitution it takes a 48 bits binary string and splits it into eight 6-bits chunks
def s_box_substitution(bits):
    #Action: Splits the 48-bit input into 8 chunks of 6 bits each (since 48/6 = 8).
    chunks = [bits[i:i + 6] for i in range(0, 48, 6)]
    output = ''
    #i is the chunk index (0 to 7), selecting one of the 8 predefined S-Boxes.

    #row and col index into the S-Box to fetch a 4-bit number.
    for i, chunk in enumerate(chunks):
        row = int(chunk[0] + chunk[5], 2)
        col = int(chunk[1:5], 2)
        output += format(S_BOXES[i][row][col], '04b')
    return output
```

Figure 27c – S-BOX substitution

- d) Permuting the Sbox_result with the P table

```
def permute(bits, table):
    return ''.join(bits[i - 1] for i in table)
```

Figure 27d – Sbox_result with the P table

```
# P Permutation Table
P = [
    16, 7, 20, 21, 29, 12, 28, 17,
    1, 15, 23, 26, 5, 18, 31, 10,
    2, 8, 24, 14, 32, 27, 3, 9,
    19, 13, 30, 6, 22, 11, 4, 25
]
```

Figure 27d – P Permutation Table

- e) The left half (L) is XORed with p_result

```
p_result = permute(sbox_result, P)
return R, xor(L, p_result) # Swap L and R
```

Figure 27e – XORed with p_result

- f) The halves are swapped: (L, R)

3.1.4 Swap After the last round and Final Swap with the FP table

```
# Final Permutation Table (FP)
FP = [
    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, 9, 49, 17, 57, 25
]
```

Figure 28 – Final Permutation FP table

3.2 Full DES Encryption Process

The DES decryption process is applied to the entire ciphertext after it has been converted into binary format. It mirrors the encryption steps in reverse order, using the same subkeys applied in reverse to recover the original plaintext. Decryption steps are explained below:

```
def decrypt_des(cipher_hex, subkeys):
    cipher_bin = bin(int(cipher_hex, 16))[2:].zfill(len(cipher_hex)*4)

    decrypted_blocks = []
    for i in range(0, len(cipher_bin), 64):
        block = cipher_bin[i:i+64]
        decrypted_blocks.append(decryption_block(block, subkeys))

    decrypted_bin = ''.join(decrypted_blocks)
    convert_text = binary_to_text(decrypted_bin)
    return convert_text
```

Figure 29 – Full DES Encryption Process

3.2.1 Hexadecimal to Binary Conversion

Convert human-readable hex to binary for bitwise operations.

3.2.2 Block Processing

The binary data is split into 64-bit blocks, and the decryption_block() function will be run here.

3.2.3 Binary to Text Conversion

The ciphertext binary string is converted to the original text using UTF-8 decoding.

```
def binary_to_text(binary_str):
    """Convert binary back to text using UTF-8."""
    bytes_list = []
    for i in range(0, len(binary_str), 8):
        byte = binary_str[i:i+8]
        if len(byte) == 8:
            bytes_list.append(int(byte, 2))
    # Reconstruct UTF-8 bytes and decode
    # We add replacement which means ? to prevent from crashing
    return bytes(bytes_list).decode('utf-8', errors='replace')
```

Figure 30 – Convert Binary to Text

3.3 Output

The main() function orchestrates the complete decryption workflow, handling data I/O, key processing, and result verification. This section documents the critical output operations.

```
def main():
    # Load the cipher text and key
    with open("/opt/DES/Task3/cipher.txt", "r") as f:
        cipher_hex = f.read().strip()

    with open("/opt/DES/Task3/key.txt", "r") as f:
        key_hex = f.read().strip()

    # Convert key_hex to key64 bits
    key64 = bin(int(key_hex, 16))[2:].zfill(64)

    subkeys = generate_subkeys(key64)

    decrypted_text = decrypt_des(cipher_hex, subkeys)
    with open("/opt/DES/Task3/decrypted.txt", "w", encoding='utf-8') as f:
        f.write(decrypted_text)

    print("Decryption complete. Result saved to decrypted.txt")
    print("Decrypted Text:", decrypted_text)

if __name__ == "__main__":
    print("This is Task3-dec.py")
    print("Kimia Sadat Karbasi - Student ID Number ='60393958'''")
    main()
```

Figure 31 – Complete Decryption Workflow with main() function

- a) File handling implementation (Reading cipher and key from a specific path)
- b) Key processing (converting hex to binary and generating subkeys)
- c) Result output (saving decrypted text in the correct path)
- d) Console output

3.4 Execution and Results

Here are all the decryption code screenshots executed on a Linux machine. Each Picture has caption that indicates the corresponding step in DES decryption process.

```
root@Docker-PAM:/opt/DES/Task3# ls
cipher.txt  key.txt  plain.txt  Task3-dec.py  Task3-enc.py
root@Docker-PAM:/opt/DES/Task3# python3 Task3-dec.py
This is Task3-dec.py
Kimia Sadat Karbasi - Student ID Number ='60393958'
Decryption complete. Result saved to decrypted.txt
Decrypted Text: "The Clockmaker of Rothenburg"
In the heart of Bavaria, nestled between rolling green hills and
cobblestone streets, sat the town of Rothenburg ob der Tauber.
Known for its perfectly preserved medieval walls and timber-framed
houses, the town looked like a postcard from the past. But what truly
gave Rothenburg its charm was the little clock shop on
Schmiedgasse Street, run by the old clockmaker, Herr Baumann.
Herr Baumann had lived in Rothenburg all his life. His small shop,
Zeit & Seele—Time and Soul—was filled with all kinds of clocks:
cuckoo clocks with dancing figurines, grandfathers with deep chimes,
and delicate pocket watches that glimmered like stardust. People
came from all over Germany—and even farther—to see his creations.
But what they didn't know was that Herr Baumann's clocks held more
than gears and springs. They held memories.
It was said that when Herr Baumann built a clock, he carved into it a
tiny piece of someone's story. A couple in love, a soldier coming
home, a child's laughter—he somehow captured these moments, and
his clocks ticked with life, echoing not just time but feeling.
One chilly April morning, a young boy named Lukas wandered into
the shop. He was ten, wearing a red scarf too big for his neck, and
had curious eyes that never stopped moving.
"Guten Tag," said Herr Baumann with a smile. "Looking for
something?"
Lukas shook his head. "Just looking. Papa says you make magic"
```

Figure 32 – Decryption Output (Decrypted Text)

```

root@Docker-PAM:/opt/DES/Task3# ls
cipher.txt decrypted.txt key.txt plain.txt Task3-dec.py Task3-enc.py
root@Docker-PAM:/opt/DES/Task3# cat decrypted.txt
"The Clockmaker of Rothenburg"
In the heart of Bavaria, nestled between rolling green hills and
cobblestone streets, sat the town of Rothenburg ob der Tauber.
Known for its perfectly preserved medieval walls and timber-framed
houses, the town looked like a postcard from the past. But what truly
gave Rothenburg its charm was the little clock shop on
Schmiedgasse Street, run by the old clockmaker, Herr Baumann.
Herr Baumann had lived in Rothenburg all his life. His small shop,
Zeit & Seele—Time and Soul—was filled with all kinds of clocks:
cuckoo clocks with dancing figurines, grandfathers with deep chimes,
and delicate pocket watches that glimmered like stardust. People
came from all over Germany—and even farther—to see his creations.
But what they didn't know was that Herr Baumann's clocks held more
than gears and springs. They held memories. 
It was said that when Herr Baumann built a clock, he carved into it a
tiny piece of someone's story. A couple in love, a soldier coming
home, a child's laughter—he somehow captured these moments, and
his clocks ticked with life, echoing not just time but feeling.
One chilly April morning, a young boy named Lukas wandered into
the shop. He was ten, wearing a red scarf too big for his neck, and
had curious eyes that never stopped moving.
"Guten Tag," said Herr Baumann with a smile. "Looking for
something?"
Lukas shook his head. "Just looking. Papa says you make magic
clocks."
Herr Baumann chuckled, kneeling down so his eyes met Lukas's.
"Magic? Well, maybe a little."
The boy's smile faded. "Do you make clocks that can go backward?"
The clockmaker blinked. "Backward?"
"My mama used to bring me here," Lukas whispered. "Before she got
sick. I want to go back... to a time when she was okay."
Silence filled the shop like snow falling quietly. Herr Baumann stood
and gently placed a hand on Lukas's shoulder. "Come with me."
He led the boy to the back room, where the oldest clocks lived. Dust

```

Figure 33 – File Output (Decrypted.txt)

4. Conclusion

This report provides a comprehensive implementation of the Data Encryption Standard (DES) algorithm, addressing key generation, Feistel encryption/decryption, and secure output handling. The project successfully demonstrated the following:

- **Precise Key Scheduling:** Accurate generation and application of round keys.
- **Hands-on Encryption/Decryption Implementation:** Practical execution of both the encryption and decryption processes.
- **Secure Data Handling:** Effective management of data throughout the encryption and decryption stages.