Experiment 3

Date of Performance : 29/03/22 **Date of Submission: 02/04/22**

SAP Id: 60004190054 **Name : Jazib Dawre**

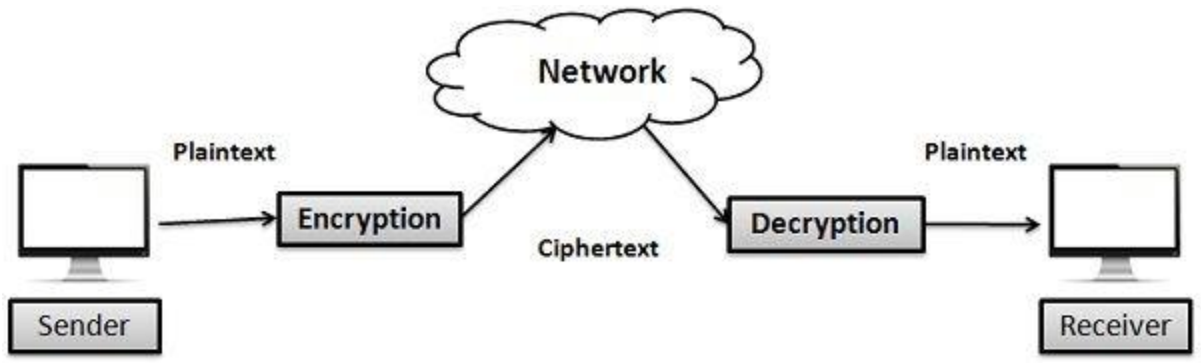
Div: A **Batch : A4**

Aim of Experiment

Implement Simplified Data Encryption Standard (S-DES).

Theory / Algorithm / Conceptual Description

Encryption and Decryption

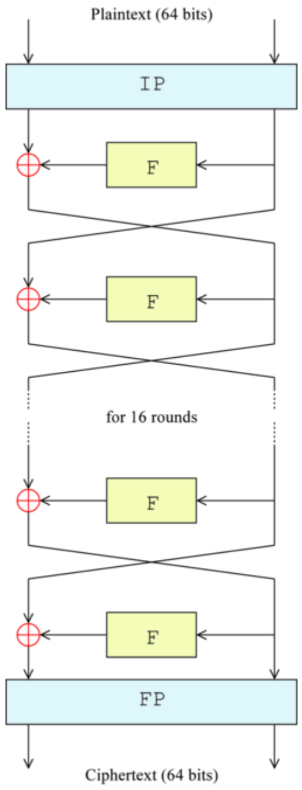


Encryption is the process by which a readable message is converted to an unreadable form to prevent unauthorized parties from reading it. Decryption is the process of converting an encrypted message back to its original (readable) format. The original message is called the plaintext message. The encrypted message is called the ciphertext message.

Digital encryption algorithms work by manipulating the digital content of a plaintext message mathematically, using an encryption algorithm and a digital key to produce a ciphertext version of the message. The sender and recipient can communicate securely if the sender and recipient are the only ones who know the key.

Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of digital data. DES is the archetypal block cipher—an algorithm that takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length. In the case of DES, the block size is 64 bits. DES also uses a key to customize the transformation, so that decryption can supposedly only be performed by those who know the particular key used to encrypt. The key ostensibly consists of 64 bits; however, only 56 of these are actually used by the algorithm. Eight bits are used solely for checking parity and are thereafter discarded. Hence the effective key length is 56 bits.



The algorithm's overall structure: there are 16 identical stages of processing, termed rounds. There is also an initial and final permutation, termed IP and FP, which are inverses (IP "undoes" the action of FP, and vice versa).

Before the main rounds, the block is divided into two 32-bit halves and processed alternately; this crisscrossing is known as the Feistel scheme. The Feistel structure ensures that decryption and encryption are very similar processes—the only difference is that the subkeys are applied in the reverse order when decrypting.

The F-function scrambles half a block together with some of the key. The output from the F-function is then combined with the other half of the block, and the halves are swapped before the next round. After the final round, the halves are swapped; this is a feature of the Feistel structure which makes encryption and decryption similar processes.

The F-function operates on half a block (32 bits) at a time and consists of four stages:

1. Expansion: the 32-bit half-block is expanded to 48 bits using the expansion permutation, denoted E in the diagram, by duplicating half of the bits. The output consists of eight 6-bit (8 × 6 = 48 bits) pieces, each containing a copy of 4 corresponding input bits, plus a copy of the immediately adjacent bit from each of the input pieces to either side.
2. Key mixing: the result is combined with a subkey using an XOR operation. Sixteen 48-bit subkeys—one for each round—are derived from the main key using the key schedule.
3. Substitution: after mixing in the subkey, the block is divided into eight 6-bit pieces before processing by the S-boxes, or substitution boxes. Each of the eight S-boxes replaces its six input bits with four output bits according to a non-linear transformation, provided in the form of a lookup table. The S-boxes provide the core of the security of DES—without them, the cipher would be linear, and trivially breakable.
4. Permutation: finally, the 32 outputs from the S-boxes are rearranged according to a fixed permutation, the P-box. This is designed so that, after permutation, the bits from the output of each S-box in this round are spread across four different S-boxes in the next round

The alternation of substitution from the S-boxes, and permutation of bits from the P-box and E-expansion provides so-called "confusion and diffusion" respectively

Simplified Data Encryption Standard (S-DES)

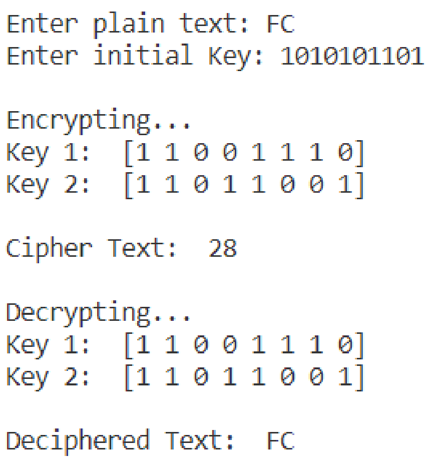
Simplified DES was designed for educational purposes only, to help students learn about modern cryptanalytic techniques. SDES has similar structure and properties to DES but has been simplified to make it much easier to perform encryption and decryption by hand with pencil and paper.

Program

A)

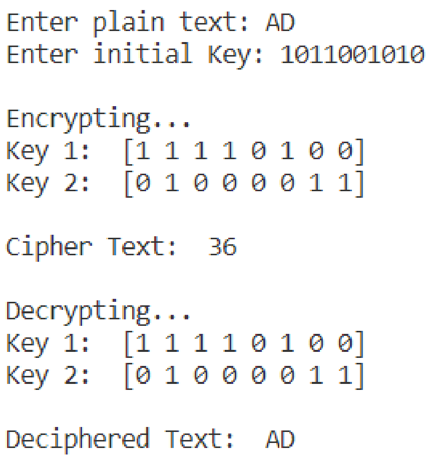
```python
import numpy as np

# Constants

P10 = np.array([3, 5, 2, 7, 4, 10, 1, 9, 8, 6]) - 1
P8 = np.array([6, 3, 7, 4, 8, 5, 10, 9]) - 1
P4 = np.array([2, 4, 3, 1]) - 1

IP = np.array([2, 6, 3, 1, 4, 8, 5, 7]) - 1
IP_inv = np.array([4, 1, 3, 5, 7, 2, 8, 6]) - 1
EP = np.array([4, 1, 2, 3, 2, 3, 4, 1]) - 1

S0 = np.array([[1, 0, 3, 2], [3, 2, 1, 0], [0, 2, 1, 3], [3, 1, 3, 2]])
S1 = np.array([[0, 1, 2, 3], [2, 0, 1, 3], [3, 0, 1, 0], [2, 1, 0, 3]])


# Internal Functions


def _process_input(input_text: str) -> np.ndarray:
    binary_string: str = format(int(input_text, 16), "08b")
    binary_array: np.ndarray = np.array(list(binary_string), "uint8")
    return binary_array


def _process_output(input_array: np.ndarray) -> str:
    output_binary_text: str = "".join(str(x) for x in input_array.tolist())
    output_hex_text: str = format(int(output_binary_text, 2), "02x").upper()
    return output_hex_text


def _process_key(input_key: str) -> np.ndarray:
    binary_array: np.ndarray = np.array(list(input_key), "uint8")
    return binary_array


def _generate_key(initial_key: np.ndarray, verbose: bool = False) -> tuple:
    key = initial_key
    key = key[P10]

    left_key = np.roll(key[:5], -1)
    right_key = np.roll(key[5:], -1)
    key1 = np.concatenate((left_key, right_key))[P8]

    left_key = np.roll(left_key, -2)
    right_key = np.roll(right_key, -2)

    key2 = np.concatenate((left_key, right_key))[P8]

    if verbose:
        print("Key 1: ", key1, "\nKey 2: ", key2)

    return key1, key2


def _substitute(input_text: np.ndarray) -> np.ndarray:
    left_text_inside: np.ndarray = input_text[:4]
    right_text_inside: np.ndarray = input_text[4:]

    left_row: np.ndarray = left_text_inside[[0, 3]]
    left_row = left_row.dot(1 << np.arange(left_row.shape[-1] - 1, -1, -1))

    left_column: np.ndarray = left_text_inside[[1, 2]]
    left_column = left_column.dot(1 << np.arange(left_column.shape[-1] - 1, -1,
-1))

    right_row: np.ndarray = right_text_inside[[0, 3]]
    right_row = right_row.dot(1 << np.arange(right_row.shape[-1] - 1, -1, -1))

    right_column: np.ndarray = right_text_inside[[1, 2]]
    right_column = right_column.dot(1 << np.arange(right_column.shape[-1] - 1,
-1, -1))

    left_text_inside = np.unpackbits(np.array([S0[left_row][left_column]],
dtype=np.uint8))[-2:]
    right_text_inside = np.unpackbits(np.array([S1[right_row][right_column]],
dtype=np.uint8))[-2:]

    inprocess_text = np.concatenate((left_text_inside, right_text_inside))

    return inprocess_text


def _stage_1(input_text: np.ndarray, key1: np.ndarray) -> np.ndarray:
    inprocess_text: np.ndarray = input_text
    inprocess_text = inprocess_text[IP]
    left_text: np.ndarray = inprocess_text[:4]
    right_text: np.ndarray = inprocess_text[4:]

    inprocess_text = right_text[EP]
    inprocess_text = np.bitwise_xor(inprocess_text, key1)
    inprocess_text = _substitute(inprocess_text)
    inprocess_text = inprocess_text[P4]
    inprocess_text = np.bitwise_xor(inprocess_text, left_text)
    inprocess_text = np.concatenate((right_text, inprocess_text))

    return inprocess_text


def _stage_2(input_text: np.ndarray, key2: np.ndarray) -> np.ndarray:
    inprocess_text: np.ndarray = input_text
    left_text: np.ndarray = inprocess_text[:4]
    right_text: np.ndarray = inprocess_text[4:]

    inprocess_text = right_text[EP]
    inprocess_text = np.bitwise_xor(inprocess_text, key2)
    inprocess_text = _substitute(inprocess_text)
    inprocess_text = inprocess_text[P4]
    inprocess_text = np.bitwise_xor(inprocess_text, left_text)
    inprocess_text = np.concatenate((inprocess_text, right_text))
    inprocess_text = inprocess_text[IP_inv]

    return inprocess_text


def _process(input_text: str, initial_key: str, method: str, verbose: bool =
False) -> str:
    initial_key_array = _process_key(input_key=initial_key)
    input_text_array = _process_input(input_text=input_text)

    if method == "encrypt":
        key1, key2 = _generate_key(initial_key=initial_key_array,
verbose=verbose)
    elif method == "decrypt":
        key2, key1 = _generate_key(initial_key=initial_key_array,
verbose=verbose)
    else:
        raise ValueError("Invalid Method")

    stage_1_cipher_text = _stage_1(input_text=input_text_array, key1=key1)
    stage_2_cipher_text = _stage_2(input_text=stage_1_cipher_text, key2=key2)
    output_text = _process_output(input_array=stage_2_cipher_text)

    return output_text


# External Functions


def encrypt(plain_text: str, initial_key: str, verbose: bool = False) -> str:
    return _process(
        input_text=plain_text,
        initial_key=initial_key,
        method="encrypt",
        verbose=verbose,
    )


def decrypt(cipher_text: str, initial_key: str, verbose: bool = False) -> str:
    return _process(
        input_text=cipher_text,
        initial_key=initial_key,
        method="decrypt",
        verbose=verbose,
    )


# Driver Function

if __name__ == "__main__":
    plain_text = input("\nEnter plain text: ")
    initial_key = input("Enter initial Key: ")

    print("\nEncrypting...")
    cipher_text = encrypt(plain_text=plain_text, initial_key=initial_key,
verbose=True)
    print("\nCipher Text: ", cipher_text)

    print("\nDecrypting...")
    deciphered_text = decrypt(cipher_text=cipher_text, initial_key=initial_key,
verbose=True)
    print("\nDeciphered Text: ", deciphered_text)
```

Output



```
Enter plain text: FC
Enter initial Key: 1010101101

Encrypting...
Key 1:  [1 1 0 0 1 1 1 0]
Key 2:  [1 1 0 1 1 0 0 1]

Cipher Text:  28

Decrypting...
Key 1:  [1 1 0 0 1 1 1 0]
Key 2:  [1 1 0 1 1 0 0 1]

Deciphered Text:  FC
```

Example 1



```
Enter plain text: AD
Enter initial Key: 1011001010

Encrypting...
Key 1:  [1 1 1 1 0 1 0 0]
Key 2:  [0 1 0 0 0 0 1 1]

Cipher Text:  36

Decrypting...
Key 1:  [1 1 1 1 0 1 0 0]
Key 2:  [0 1 0 0 0 0 1 1]

Deciphered Text:  AD
```

Example 2

Conclusion

Encryption methods allow for obfuscation of data when it is vulnerable while being transported over various channels. Key based encryption algorithms are quite prominent which rely on a pre-shared key between the sender and the recipient. The sender encrypts the data using the key and the receiver decrypts it with the key. Attackers without the key should not be able to decrypt and understand the data. DES is a block cipher and at the time of its introduction was a strong cipher. It works on 64-bit data by using a series of 16 common rounds that use the Feistel scheme which allows for symmetric encryption and decryption using the same keys in reverse order.