21CSE1003 ASHISH SINGH

Q1. Write a program to encrypt and decrypt the message using the Substitution and permutation network.

Encryption:

1. **Initialize Plain Text and Block Size:**
    - Input the plain text and block size.
    - Partition the plain text into blocks of the specified size, padding the last block with zeros if necessary.
2. **Generate Initial Key and Round Keys:**
    - Generate an initial binary key of specified size.
    - Divide this key into parts to create round keys for each round.
3. **Substitution-Permutation Network (SPN) Encryption:**
    - For each round (excluding the last two):
        - XOR the current message with the round key.
        - Apply S-box substitution to the XOR output.
        - Partition the S-box output into blocks.
        - Perform interleaved permutation on the blocks.
        - Update the message to the permuted output.
    - In the second last round:
        - XOR the message with the round key and apply S-box substitution.
    - In the last round:
        - XOR the message with the final round key.
4. **Output the Encrypted Message.**

Decryption:

1. **Initialize Cipher Text:**
    - Input the cipher text and use the same round keys.
2. **Substitution-Permutation Network (SPN) Decryption:**
    - In the last round:
        - XOR the cipher text with the final round key.
    - In the second last round:

    - Apply inverse S-box substitution and XOR with the second last round key.
- For each previous round (in reverse order):
    - Perform inverse interleaved permutation on the message.
    - Apply inverse S-box substitution to the permuted output.
    - XOR with the respective round key.

3. **Output the Decrypted Message.**

LAB_8\substitution_permutation_network.py

```python

# 21CSE1003 Ashish Singh

# Q1. Write a program to encrypt and decrypt the message using the Substitution and permutation network.


print("-------------------Encryption------------------")

import random

def binary_string_to_int(binary_string):
    return int(binary_string, 2)

def partition_and_pad(binary_message, block_size):
    # Split binary message into chunks of block_size
    blocks = [binary_message[i:i+block_size] for i in range(0, len(binary_message), block_size)]

    # If the last block is shorter than the block size, pad it with zeroes
    if len(blocks[-1]) < block_size:
        blocks[-1] = blocks[-1].ljust(block_size, '0')

    return blocks

# binary_message = '0010011010110111'  # Input binary message
binary_message = str(input("Enter plain text: "))  # Input binary message
block_size = int(input("Enter block size: "))
blocks = partition_and_pad(binary_message, block_size)
for i, block in enumerate(blocks):
    print(f'Block {i+1}: {block}')

def generate_initial_key(key_size):
    # Generate a random binary key of specified size
    return ''.join(random.choice(['0', '1']) for _ in range(key_size))

def generate_round_keys(initial_key, num_rounds, key_size, block_size):
    # divide the key into parts equal to number or rounds. then do window shifting to make the keys
    round_keys = []
    for i in range(num_rounds):
        round_keys.append(initial_key[i * block_size : ((key_size // 2) + (i * block_size))])

    return round_keys

key_size = 32  # Size of each key
num_rounds = 5  # Number of rounds
```

```python
# initial_key = generate_initial_key(key_size)
initial_key = "00111010100101001101011000111111"

round_keys = generate_round_keys(initial_key, num_rounds, key_size, block_size)

print(f'Initial Key: {initial_key}')
for i, key in enumerate(round_keys):
    print(f'Round Key {i+1}: {key}')

number_of_blocks = len(blocks)

def s_box_substitution(bits):
    s_box = {
        '0000': '1110', '0001': '0100', '0010': '1101', '0011': '0001',
        '0100': '0010', '0101': '1111', '0110': '1011', '0111': '1000',
        '1000': '0011', '1001': '1010', '1010': '0110', '1011': '1100',
        '1100': '0101', '1101': '1001', '1110': '0000', '1111': '0111'
    }
    substituted_bits = ''.join(s_box[bits[i:i+4]] for i in range(0, len(bits), 4))
    return substituted_bits

# function to make blocks of s_box_output
def make_blocks(s_box_output, number_of_blocks):
    blocks = [s_box_output[i:i+number_of_blocks] for i in range(0, len(s_box_output), number_of_blocks)]
    return blocks

def interleaved_permutation(blocks):
    block_size = len(blocks[0])
    interleaved_bits = ''
    for i in range(block_size):
        for block in blocks:
            interleaved_bits += block[i]
    return interleaved_bits

def substitution_permutation_network(binary_message, num_rounds, round_keys, block_size):
    message = binary_message
    for i in range(num_rounds-2):
        print(f'Round {i+1}')
        print(f'Round Key: {round_keys[i]}')
        print(f'Message: {message}')
        # do XOR of 1st key with the binary message
        xor_output = bin(binary_string_to_int(message) ^ binary_string_to_int(round_keys[i]))[2:].zfill(len(message))
        print(f'XOR Output: {xor_output}')
        # do s_box substitution
        s_box_output = s_box_substitution(xor_output)
        print(f'S-Box Output: {s_box_output}')
        # make blocks of s_box_output
        blocks = make_blocks(s_box_output, number_of_blocks)
```

```python
        # do interleaved permutation
        interleaved_bits = interleaved_permutation(blocks)
        print(f'Interleaved Bits: {interleaved_bits}')
        message = interleaved_bits

    # for second last round only do XOR and s_box substitution
    print(f'Round {num_rounds-1}')
    print(f'Round Key: {round_keys[num_rounds-2]}')
    print(f'Message: {message}')
    xor_output = bin(binary_string_to_int(message) ^ binary_string_to_int(round_keys[num_rounds-2]))[2:].zfill(len(message))
    print(f'XOR Output: {xor_output}')
    s_box_output = s_box_substitution(xor_output)
    print(f'S-Box Output: {s_box_output}')
    message = s_box_output

    # for last round only do XOR
    print(f'Round {num_rounds}')
    print(f'Round Key: {round_keys[num_rounds-1]}')
    print(f'Message: {message}')
    xor_output = bin(binary_string_to_int(message) ^ binary_string_to_int(round_keys[num_rounds-1]))[2:].zfill(len(message))
    print(f'XOR Output: {xor_output}')
    message = xor_output

    return message

cipher_text = substitution_permutation_network(binary_message, num_rounds, round_keys, block_size)

print(f'Encrypted Message: {cipher_text}')

print("-------------------Decryption-------------------")

def inverse_s_box_substitution(bits):
    inverse_s_box = {
        '1110': '0000', '0100': '0001', '1101': '0010', '0001': '0011',
        '0010': '0100', '1111': '0101', '1011': '0110', '1000': '0111',
        '0011': '1000', '1010': '1001', '0110': '1010', '1100': '1011',
        '0101': '1100', '1001': '1101', '0000': '1110', '0111': '1111'
    }
    substituted_bits = ''.join(inverse_s_box[bits[i:i+4]] for i in range(0, len(bits), 4))
    return substituted_bits

def inverse_interleaved_permutation(interleaved_bits, number_of_blocks):
    block_size = len(interleaved_bits) // number_of_blocks
    blocks = ['' for _ in range(number_of_blocks)]
    for i in range(block_size):
        for j in range(number_of_blocks):
            blocks[j] += interleaved_bits[i * number_of_blocks + j]
```

```python
    return ''.join(blocks)

def decryption_network(cipher_text, num_rounds, round_keys, block_size):
    message = cipher_text
    # Inverse last round XOR
    message = bin(binary_string_to_int(message) ^ binary_string_to_int(round_keys[num_rounds-1]))[2:].zfill(len(message))
    print(f'After final XOR: {message}')

    # Inverse second last round XOR and S-box substitution
    message = inverse_s_box_substitution(message)
    message = bin(binary_string_to_int(message) ^ binary_string_to_int(round_keys[num_rounds-2]))[2:].zfill(len(message))
    print(f'After penultimate XOR and inverse S-Box: {message}')

    for i in range(num_rounds-2, 0, -1):
        print(f'Round {i+1}')
        print(f'Round Key: {round_keys[i]}')
        # Inverse permutation
        interleaved_bits = inverse_interleaved_permutation(message, number_of_blocks)
        print(f'Inverse Interleaved Bits: {interleaved_bits}')
        # Inverse S-box substitution
        s_box_output = inverse_s_box_substitution(interleaved_bits)
        print(f'Inverse S-Box Output: {s_box_output}')
        # XOR with round key
        message = bin(binary_string_to_int(s_box_output) ^ binary_string_to_int(round_keys[i-1]))[2:].zfill(len(s_box_output))
        print(f'After XOR: {message}')

    return message

decrypted_message = decryption_network(cipher_text, num_rounds, round_keys, block_size)
print(f'Decrypted Message: {decrypted_message}')
```

Q2. Write a program to implement the Feistel Cipher.

Encryption:

1. **Initialize Plain Text:**
    - Input the plain text.
    - Split the plain text into two halves: left and right.
2. **Perform Rounds of Encryption:**
    - For each round:
        - Apply the function apply_func on the right half.
        - XOR the left half with the output of apply_func.
        - Update left to be right and right to be the result of XOR.
3. **Combine Halves to Get Cipher Text:**
    - Concatenate the final left and right halves.

Decryption:

1. **Initialize Cipher Text:**
    - Input the cipher text.
    - Split the cipher text into two halves: left and right.
2. **Perform Rounds of Decryption:**
    - For each round:
        - Apply the function apply_func on the left half.
        - XOR the right half with the output of apply_func.
        - Update right to be left and left to be the result of XOR.
3. **Combine Halves to Get Decrypted Text:**
    - Concatenate the final right and left halves.

LAB_8\feistel_cipher copy.py

```python
def apply_func(r):
    ans = ''
    for i in range(0, len(r), 3):
        ch = r[i + 2]
        temp = r[i:i + 3]
        temp = ch + temp[:-1]
        ans += temp
    return ans

def feistel_encrypt(plain_text, rounds):
    l = len(plain_text)
    left, right = plain_text[:l // 2], plain_text[l // 2:]

    for _ in range(rounds):
        new_r = apply_func(right)
        after_xor = ''.join('0' if left[j] == new_r[j] else '1' for j in range(len(new_r)))
        left, right = right, after_xor

    return left + right

def feistel_decrypt(cipher_text, rounds):
    l = len(cipher_text)
    left, right = cipher_text[:l // 2], cipher_text[l // 2:]

    for _ in range(rounds):
        new_l = apply_func(left)
        after_xor = ''.join('0' if right[j] == new_l[j] else '1' for j in range(len(new_l)))
        right, left = left, after_xor

    return left + right

# Example usage
plain_text = input("Enter plain text: ")
rounds = int(input("Enter the number of rounds: "))

# Encrypt
cipher_text = feistel_encrypt(plain_text, rounds)
print(f'Encrypted Text: {cipher_text}')

# Decrypt
decrypted_text = feistel_decrypt(cipher_text, rounds)
print(f'Decrypted Text: {decrypted_text}')
```