**LAB 9**

1. **Write a python script to get the binary values from the user and perform XOR operation.**

def xor\_binary(bin1, bin2):

# Ensure both binary strings are of the same length by padding the shorter one

max\_len = max(len(bin1), len(bin2))

bin1 = bin1.zfill(max\_len)

bin2 = bin2.zfill(max\_len)

# Perform XOR operation

xor\_result = ''.join('1' if bit1 != bit2 else '0' for bit1, bit2 in zip(bin1, bin2))

return xor\_result

# Get binary inputs from the user

binary1 = input("Enter the first binary number: ")

binary2 = input("Enter the second binary number: ")

# Validate input

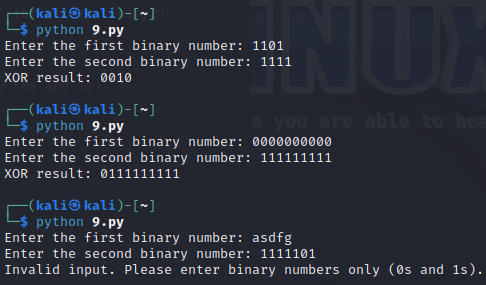
if all(bit in '01' for bit in binary1) and all(bit in '01' for bit in binary2):

result = xor\_binary(binary1, binary2)

print(f"XOR result: {result}")

else:

print("Invalid input. Please enter binary numbers only (0s and 1s).")



1. **Write a Python script that implements a simple 4-bit LFSR. The initial state of the register and the tap positions should be user inputs.**

**Simulate 10 steps of the LFSR, displaying the state of the register at each step.**

def lfsr(initial\_state, taps, steps):

# Convert initial state to a list of integers (0 or 1)

state = [int(bit) for bit in initial\_state]

# List to store states for display

states = [initial\_state]

for \_ in range(steps):

# Calculate the feedback bit based on the tap positions

feedback = 0

for tap in taps:

feedback ^= state[tap] # XOR the tapped bits

# Shift the register and insert the feedback bit at the front

state = [feedback] + state[:-1]

# Convert state back to string and store it

states.append(''.join(map(str, state)))

return states

# Get user inputs

initial\_state = input("Enter the initial 4-bit state: ")

taps\_input = input("Enter the tap positions: ")

# Convert tap positions to a list of integers

taps = list(map(int, taps\_input.split()))

# Validate inputs

if (len(initial\_state) == 4 and all(bit in '01' for bit in initial\_state) and

all(0 <= tap < 4 for tap in taps)):

# Simulate the LFSR

steps = 10

states = lfsr(initial\_state, taps, steps)

# Display the states

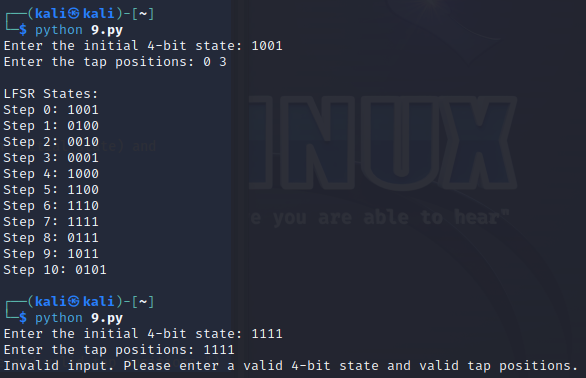
print("\nLFSR States:")

for i, state in enumerate(states):

print(f"Step {i}: {state}")

else:

print("Invalid input. Please enter a valid 4-bit state and valid tap positions.")



1. **Write a report on attacks on LFSR. Explain any one attack in detail.**

Common Attacks on LFSRs

1. **Brute Force Attack**
2. **Algebraic Attacks**
3. Correlation Attack
4. Differential Attack

**DIFFERENTIAL ATTACK**

How LFSRs Work

LFSRs generate sequences of bits based on a linear function of their previous states. The output is determined by the initial state and the feedback polynomial, which dictates which bits are tapped for feedback.

Steps in Differential Cryptanalysis on LFSRs

1. Selecting Input Pairs: The attacker selects pairs of input states (or known output sequences) that differ in a specific bit or bits. For example, if the initial state is SSS, the attacker may consider SSS and S′S'S′ where S′S'S′ differs from SSS in one or more bits.
2. Observing Output Differences: The attacker examines the output produced by these input pairs. For an LFSR defined by a polynomial, the relationship between input and output is linear, making the output differences predictable based on the input differences.
3. Building Differential Tables: By observing many input-output pairs, the attacker constructs a differential table that shows the frequencies of output differences corresponding to each input difference. This table helps identify patterns and can reveal potential weaknesses in the LFSR's structure.
4. Identifying Key Bits: If certain input differences consistently lead to specific output differences, the attacker can use this information to deduce the values of some internal bits or the feedback polynomial itself. The attacker can exploit this information iteratively to gain more insight into the LFSR's state.
5. Exploiting Linearity: Because LFSRs are linear, the relationships between input and output can often be simplified using linear algebra techniques. The attacker can set up equations based on the observed differences and solve them to recover the internal state or feedback polynomial.

Example of Differential Cryptanalysis on LFSRs

1. Assuming an LFSR: Consider a 4-bit LFSR defined by the polynomial x4+x+1x^4 + x + 1x4+x+1. The taps correspond to bits 4 and 1.
2. Input Differences: The attacker selects two initial states, for example, 1001 and 1101, which differ only in the second bit.
3. Output Generation: The attacker records the outputs for both initial states over several steps. The output for 1001 might be 0110, while for 1101, it might be 0010.
4. Analysing Output Differences: The attacker calculates the differences between the outputs. In this case:
   * 0110 (for 1001)
   * 0010 (for 1101)
   * Output difference: 0100
5. Building the Differential Table: By repeating this process for various pairs of inputs, the attacker fills in a differential table to identify which input differences lead to which output differences.
6. Extracting Information: If certain output differences are consistently observed for specific input differences, the attacker can infer information about the internal bits or potentially the polynomial.

**BONUS POINT:**

**4. write a python script to break hill cipher (2X2) using known plain text attack.**

**Known Plaintext: "MEET"**

**Corresponding Ciphertext: "URRG"**

import numpy as np

# Mapping for characters

def char\_to\_num(c):

return ord(c) - ord('A')

def num\_to\_char(n):

return chr((n % 26) + ord('A'))

# Known plaintext and ciphertext (both need to be converted to numerical matrices)

plaintext = "MEET"

ciphertext = "URRG"

# Convert plaintext and ciphertext to numerical matrices

P = np.array([[char\_to\_num(plaintext[0]), char\_to\_num(plaintext[1])],

[char\_to\_num(plaintext[2]), char\_to\_num(plaintext[3])]])

C = np.array([[char\_to\_num(ciphertext[0]), char\_to\_num(ciphertext[1])],

[char\_to\_num(ciphertext[2]), char\_to\_num(ciphertext[3])]])

# Calculate the inverse of P modulo 26

P\_inv = np.linalg.inv(P) \* np.linalg.det(P)

P\_inv = np.round(P\_inv).astype(int) % 26 # Ensure elements are integers

# Compute key matrix as K = C \* P\_inv mod 26

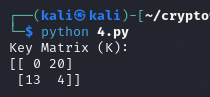
K = (C @ P\_inv) % 26

K = np.round(K).astype(int)

# Display the key matrix

print("Key Matrix (K):")

print(K)



**=========================================================================================================================**

**“Dream, Dream, Dream. Dreams transform into thoughts and thoughts result in action.” - Dr. APJ**

**=========================================================================================================================**