

LAB ASSIGNMENT: 01

SUBMITTED BY:

Muhammad Hassan

SP24-BSE-008

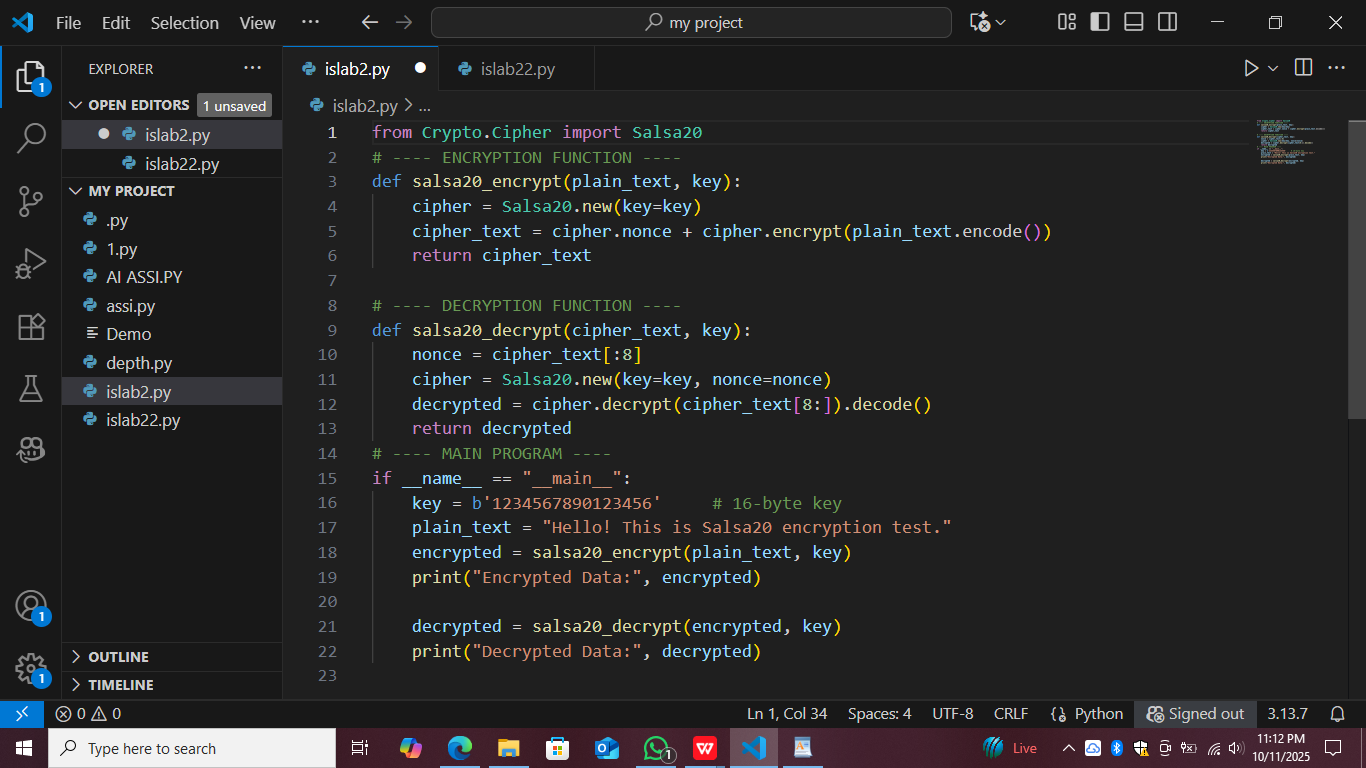
SUBMITTED TO: MA'AM AMBREEN

DATE: 12 OCT, 2025

COURSE: IS

TASK1:

1. Write python code for your designed stream cipher approach for encryption decryption, you can use approach from more than one already developed ciphers as given in lab practice exercises.



### ****Encryption****

Import Salsa20 from Crypto.Cipher.

Create cipher with Salsa20.new(key=key).

Convert plain text to bytes using .encode().

Encrypt text using cipher.encrypt().

Combine cipher.nonce + encrypted data.

Return the final cipher text.

### 🔹 ****Decryption****

Extract first 8 bytes as nonce.

Create cipher with same key and extracted nonce.

Decrypt the remaining bytes using cipher.decrypt().

Convert bytes back to text using .decode().

Return original plain text.

### 🔹 ****Main Program****

Define a 16-byte key.

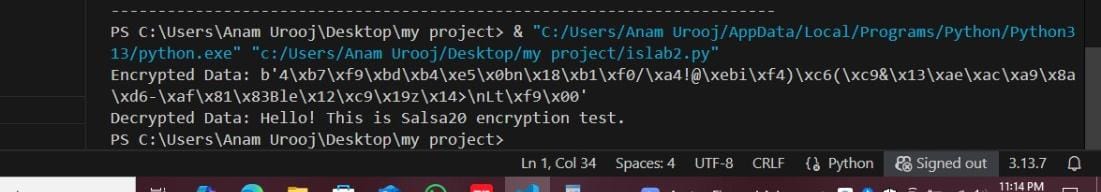
Define plain text message.

Call encryption → get unreadable cipher data.

Call decryption → get original text back.

Print both results.

Output:



TASK 2:

1. Design and implement an adversarial attack approach for your proposed stream cipher approach

from itertools import product

# LFSR step

def lfsr\_step(s,t,l):

    out = s & 1

    fb = 0

    for x in t: fb ^= (s>>x)&1

    s = ((s>>1) | (fb<<(l-1))) & ((1<<l)-1)

    return out,s

# keystream generator

def geffe(s1,s2,s3,t1,t2,t3,l1,l2,l3,n):

    ks=[]

    for \_ in range(n):

        b1,s1 = lfsr\_step(s1,t1,l1)

        b2,s2 = lfsr\_step(s2,t2,l2)

        b3,s3 = lfsr\_step(s3,t3,l3)

        ks.append((b1 & b2) ^ ((1^b1) & b3))

    return ks

# bits/bytes helpers (LSB-first)

def bytes\_to\_bits(b): return [ (byte>>i)&1 for byte in b for i in range(8) ]

def bits\_to\_bytes(bits):

    bits += [0]\*((-len(bits))%8)

    out=bytearray()

    for i in range(0,len(bits),8):

        v=0

        for j in range(8): v |= (bits[i+j]&1)<<j

        out.append(v)

    return bytes(out)

def xor\_bytes\_with\_ks(data,ks):

    db = bytes\_to\_bits(data)

    return bits\_to\_bytes([db[i]^ks[i] for i in range(len(db))])

# --- set params (must match encryptor) ---

l1,l2,l3 = 6,5,6

t1,t2,t3 = [0,1],[0,2],[0,1,2]

# --- Provide ciphertext hex and known prefix here ---

cipher\_hex = "PUT\_CIPHERTEXT\_HEX\_HERE"  # replace

known\_prefix = "Hello "                # replace with the known plaintext prefix

if cipher\_hex == "PUT\_CIPHERTEXT\_HEX\_HERE":

    print("Edit script: set cipher\_hex to your ciphertext (hex string) and known\_prefix.")

    raise SystemExit

ct = bytes.fromhex(cipher\_hex)

kp = known\_prefix.encode()

# observed keystream bits from known prefix

obs\_bits = bytes\_to\_bits(ct[:len(kp)])

kp\_bits = bytes\_to\_bits(kp)

obs\_ks = [ obs\_bits[i]^kp\_bits[i] for i in range(len(kp\_bits)) ]

# brute-force small seeds

found = None

for s1,s2,s3 in product(range(1<<l1), range(1<<l2), range(1<<l3)):

    if geffe(s1,s2,s3,t1,t2,t3,l1,l2,l3,len(obs\_ks)) == obs\_ks:

        found = (s1,s2,s3)

        break

if not found:

    print("No seeds found — check params / prefix length.")

else:

    print("Found seeds:", tuple(bin(x) for x in found))

    ks\_full = geffe(found[0],found[1],found[2],t1,t2,t3,l1,l2,l3,len(ct)\*8)

    recovered = xor\_bytes\_with\_ks(ct, ks\_full)

try: print("Recovered plaintext:", recovered.decode())

except: print("Recovered bytes:", recovered)

### 🔹 1. Purpose of the Code

This program performs a **known plaintext attack** on a **Geffe stream cipher**, which combines outputs from **three LFSRs (Linear Feedback Shift Registers)**.  
It tries to recover the original message and the secret LFSR seeds.

### 🔹 2. LFSR (Linear Feedback Shift Register)

The lfsr\_step function simulates one step of an LFSR.

It:

Takes the current state (s) and tap positions (t)

Calculates the **feedback bit** using XOR of specific bit positions

Shifts bits to the right and inserts the feedback bit on the left.

Returns the **output bit** and the **new state**.

### 🔹 3. Geffe Generator Function

The geffe function generates a **keystream** using three LFSRs.

It uses this combining formula:

* Z=(b1&b2)⊕((1⊕b1)&b3)Z = (b1 \& b2) \oplus ((1 \oplus b1) \& b3)Z=(b1&b2)⊕((1⊕b1)&b3)

Meaning:

If b1 is 1 → output b2

If b1 is 0 → output b3

This makes the output depend non-linearly on all three LFSRs.

### 🔹 4. Helper Functions for Bit Conversion

bytes\_to\_bits converts bytes into a list of 0s and 1s (bit sequence).

bits\_to\_bytes converts bit lists back into bytes.

xor\_bytes\_with\_ks XORs the data bits with the keystream bits (used for encryption or decryption).

### 🔹 5. Parameters

The program defines lengths (l1, l2, l3) and tap positions (t1, t2, t3) for each LFSR.

These must match whatever parameters were used during encryption.

### 🔹 6. Input Section

The user must provide:

**Ciphertext (in hex)** – the encrypted message.

**Known prefix** – a few known plaintext characters (like "Hello ").

### 🔹 7. Observed Keystream Extraction

From the **known plaintext part**, the code finds what keystream bits were used.

It does this by XORing ciphertext bits with the known plaintext bits.

### 🔹 8. Brute Force Seed Search

It tries **all possible initial seeds (starting states)** for the 3 LFSRs.

For each combination, it generates a test keystream of the same length as the known part.

If that matches the observed keystream, it has found the correct seeds.

### 🔹 9. When Seeds Are Found

Once the correct (s1, s2, s3) seeds are found:

It regenerates the **full keystream** for the whole ciphertext.

XORs that keystream with the ciphertext to get the **recovered plaintext**.

### 🔹 10. Output

Prints the **binary seeds** (the secret keys for each LFSR).

Prints the **recovered plaintext message** (or raw bytes if not decodable).

Output:

