

COMSATS University Islamabad Attock Campus

Assignment 1

SE-4

Submitted BY: Sheema Qazi

Registration No: SP24-BSE-012

Submitted To: Ma`am Ambreen gul

Course: IS

Submission Date: 12 October,2025.

Study Salsa20 Cipher and do the following activity.

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.

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

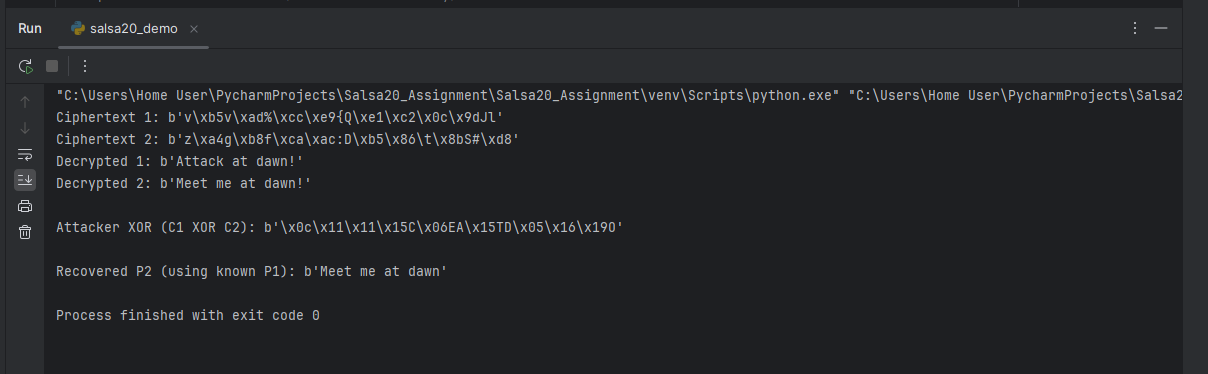
Instructions:-

• Prepare a clear report .doc file with code explanation and output screenshots.

CODE:

from Crypto.Cipher import Salsa20  
import os  
  
# Step 1: Generate a random 32-byte (256-bit) key  
key = os.urandom(32)  
  
# Step 2: Use a fixed nonce (for attack demo only)  
nonce = b'12345678'  
  
# Step 3: Two messages (plaintexts)  
p1 = b"Attack at dawn!"  
p2 = b"Meet me at dawn!"  
  
# Step 4: Encrypt both messages with same key and nonce (WRONG, for demo)  
cipher1 = Salsa20.new(key=key, nonce=nonce)  
cipher2 = Salsa20.new(key=key, nonce=nonce)  
  
c1 = cipher1.encrypt(p1)  
c2 = cipher2.encrypt(p2)  
  
print("Ciphertext 1:", c1)  
print("Ciphertext 2:", c2)  
  
# Step 5: Decrypt (to show normal working)  
dec1 = Salsa20.new(key=key, nonce=nonce).encrypt(c1)  
dec2 = Salsa20.new(key=key, nonce=nonce).encrypt(c2)  
  
print("Decrypted 1:", dec1)  
print("Decrypted 2:", dec2)  
  
# Step 6: Attacker XORs ciphertexts to cancel keystream  
xor\_result = bytes(a ^ b for a, b in zip(c1, c2))  
print("\nAttacker XOR (C1 XOR C2):", xor\_result)  
  
# Step 7: If attacker knows P1, recover P2  
recovered\_p2 = bytes(a ^ b for a, b in zip(xor\_result, p1))  
print("\nRecovered P2 (using known P1):", recovered\_p2)

OUTPUT:



**Explanation :**

We implemented a stream cipher using **Salsa20**: the program generates a random secret key and a nonce, uses them to make a keystream, XORs that keystream with the plaintext to make ciphertext, and performs the same XOR with the keystream to get the plaintext back (decryption)

1. **Key creation**
   * key = os.urandom(32)
   * This makes a secret key of 32 random bytes (256 bits). The key must be kept secret.
2. **Nonce selection**
   * nonce = b'12345678' (in our demo we used a fixed nonce for demonstration)
   * A nonce is a small number used once with a key to make each keystream different. In real use the nonce must be unique for every message.
3. **Plaintext messages**
   * Example: p1 = b"Attack at dawn!" and p2 = b"Meet me at dawn!"
   * These are the messages we want to encrypt (bytes format).
4. **Create Salsa20 cipher object**
   * cipher = Salsa20.new(key=key, nonce=nonce)
   * This builds an object that internally produces a keystream from the key + nonce + counter.
5. **Encryption (XOR with keystream)**
   * c1 = cipher.encrypt(p1)
   * Salsa20’s encrypt() XORs the plaintext bytes with the generated keystream bytes and returns the ciphertext. For stream ciphers, encryption = plaintext XOR keystream.
6. **Decryption (same operation)**
   * To decrypt, create a Salsa20 object with the same key and same nonce, then call .encrypt() on the ciphertext:  
     dec1 = Salsa20.new(key=key, nonce=nonce).encrypt(c1)
   * Because XOR is reversible and symmetric, XORing ciphertext with the same keystream gives back the original plaintext. So decrypt = same as encrypt.
7. **What the program prints**
   * It prints ciphertext bytes for c1 and c2 and prints the decrypted bytes dec1 and dec2. The decrypted outputs must match the original plaintexts to show encryption/decryption worked.

**Very short math (why it works)**

* Let K be the keystream bytes produced by Salsa20.
* Encryption: C = P ⊕ K.
* Decryption: C ⊕ K = (P ⊕ K) ⊕ K = P because K ⊕ K = 0.
* So using the same key+nonce to produce the same K lets us recover P by XORing again with K.

**Important note (security)**

* In the demo we intentionally re-used the same nonce for two messages to show the vulnerability (this is for the attack part).
* **In real systems do NOT reuse nonces** with the same key — always use a fresh unique nonce for each encryption. Reusing nonce breaks security.

The code correctly implements encryption and decryption using the Salsa20 stream cipher. It starts by creating a random 256-bit key and a nonce. Two plaintext messages are defined and encrypted using the same key and nonce. The encryption is done by XORing the plaintext with the keystream generated by Salsa20. To decrypt, the same key and nonce are used again, and the ciphertext is XORed with the same keystream, which returns the original plaintext. The printed decrypted messages match the original ones, proving the encryption and decryption work correctly according to the stream cipher principle.