

# **Comsats University Islamabad**

(Attock Campus)



---

## **ASSIGNMENT 1**

**Name :** Eman Ejaz

**Registration No:** (Sp24- BSE-052)

**Subject:** Information security lab

**Submitted to:** Mam Ambreen Gul

**Submission Date:** 11- October- 2025

# Task:

Study Salsa20 Cipher and do the following activity.

## Objective:

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.
3. Instructions:- • Prepare a clear report .doc file with code explanation and output screenshots.

## Explanation:

Stream cipher: produces a pseudo-random keystream from a secret key + nonce and XORs it with plaintext to get ciphertext. Decryption is the same XOR operation with the identical keystream.

**Nonce:** a number used once (must be unique per message under same key). Reusing a nonce with the same key leaks information.

Salsa20 idea (high level): build a 16-word state from constants, key, nonce, and counter. Repeated 'quarterround' mixing steps produce 64 bytes of keystream. This implementation uses a compact, readable version of that core to teach the concept.

## Code:

```
import struct, secrets

# --- tiny helpers ---
def rol(x, n): return ((x << n) & 0xffffffff) | (x >> (32 - n))
def qr(a,b,c,d):
    b ^= rol((a + d) & 0xffffffff, 7)
    c ^= rol((b + a) & 0xffffffff, 9)
    d ^= rol((c + b) & 0xffffffff,13)
    a ^= rol((d + c) & 0xffffffff,18)
    return a,b,c,d

# --- 64-byte block generator (Salsa-like, simple) ---
def salsa_block(key32, nonce8, ctr):
    # key32: 32 bytes, nonce8: 8 bytes, ctr: integer block counter
    const = b"expand 32-byte k"
    k0, k1 = key32[:16], key32[16:]
    s = [
        struct.unpack('<I', const[0:4])[0],
```

```

        *struct.unpack('<4I', k0),
        struct.unpack('<I', const[4:8])[0],
        *struct.unpack('<2I', nonce8),
        ctr & 0xffffffff, (ctr>>32)&0xffffffff,
        struct.unpack('<I', const[8:12])[0],
        *struct.unpack('<4I', k1),
        struct.unpack('<I', const[12:16])[0]
    ]
    x = s.copy()
    # do 8 double-rounds (fast demo; real Salsa20 uses more rounds)
    for _ in range(8):
        # column rounds
        x[0],x[4],x[8],x[12] = qr(x[0],x[4],x[8],x[12])
        x[5],x[9],x[13],x[1] = qr(x[5],x[9],x[13],x[1])
        x[10],x[14],x[2],x[6] = qr(x[10],x[14],x[2],x[6])
        x[15],x[3],x[7],x[11] = qr(x[15],x[3],x[7],x[11])
        # row rounds
        x[0],x[1],x[2],x[3] = qr(x[0],x[1],x[2],x[3])
        x[5],x[6],x[7],x[4] = qr(x[5],x[6],x[7],x[4])
        x[10],x[11],x[8],x[9] = qr(x[10],x[11],x[8],x[9])
        x[15],x[12],x[13],x[14] = qr(x[15],x[12],x[13],x[14])
    out = [(x[i] + s[i]) & 0xffffffff for i in range(16)]
    return struct.pack('<16I', *out) # 64 bytes

# --- produce keystream of needed length ---
def keystream(key, nonce, length):
    out = b''; ctr = 0
    while len(out) < length:
        out += salsa_block(key, nonce, ctr)
        ctr += 1
    return out[:length]

# --- encrypt/decrypt (same function because XOR is reversible) ---
def encrypt(key, nonce, plaintext):
    ks = keystream(key, nonce, len(plaintext))
    return bytes(p ^ k for p,k in zip(plaintext, ks))

def decrypt(key, nonce, ciphertext):
    return encrypt(key, nonce, ciphertext)

# --- Attack 1: nonce reuse (easy explanation below) ---
def demo_nonce_reuse():
    print("---- Nonce reuse demo ----")
    key = secrets.token_bytes(32)
    nonce = secrets.token_bytes(8) # reused accidentally
    p1 = b"Hello. My salary is 5000."
    p2 = b"Secret. My salary is 7000."
    c1 = encrypt(key, nonce, p1)

```

```

c2 = encrypt(key, nonce, p2)
# attacker computes c1 ^ c2 = p1 ^ p2
pxor = bytes(a ^ b for a,b in zip(c1, c2))
# if attacker knows p1, p2 = pxor ^ p1
recovered_p2 = bytes(a ^ b for a,b in zip(pxor, p1))
print("Recovered p2:", recovered_p2)
print("Actual p2:   ", p2)
print("Success:", recovered_p2 == p2)
print()

# --- Attack 2: brute-force small key (teaches method only) ---
def demo_bruteforce_small_key():
    print("---- Brute-force small key demo ----")
    nbits = 20 # very small on purpose, real keys are huge
    secret_small = secrets.randbelow(1 << nbits)
    # expand small secret into a 32-byte key deterministically
    key = (secret_small.to_bytes((nbits+7)//8,'little') * 32)[:32]
    nonce = secrets.token_bytes(8)
    pt = b"This is secret. PrefixKnown"
    ct = encrypt(key, nonce, pt)
    # attacker knows nonce and expects plaintext starts with "This is"
    want = b"This is"
    found = None
    for k in range(1 << nbits):
        ktest = (k.to_bytes((nbits+7)//8,'little') * 32)[:32]
        if decrypt(ktest, nonce, ct).startswith(want):
            found = k
            recovered = decrypt(ktest, nonce, ct)
            break
    print("Secret value:", secret_small, "Found:", found, "Recovered == pt:",
recovered == pt)
    print()

# --- run demo ---
if __name__ == "__main__":
    K = b"0123456789ABCDEF0123456789ABCDEF"
    N = b"ABCDEFGH"
    msg = b"Example plain text."
    ct = encrypt(K, N, msg)
    pt = decrypt(K, N, ct)
    print("Encrypt/Decrypt OK:", pt == msg)
    demo_nonce_reuse()
    demo_bruteforce_small_key()

```

**Output:**

isa\_demo.py

Encrypt/Decrypt OK: True

---- Nonce reuse demo ----

Recovered p2: b'Secret. My salary is 7000'

Actual p2: b'Secret. My salary is 7000.'

Success: False

---- Brute-force small key demo ----

Secret value: 71976 Found: 71976 Recovered == pt: True