**COMSATS University Islamabad, Attock campus Department of Computer Science**

**Assignment #02**

**Information Security**

Brekhna Gul

Sp24-bse-051

12-October -2025

Ma’am Ambreen

**Assignment 1:**

Tasks:

1. Write Python code for your designed stream cipher approach for encryption

and decryption. You can use an approach from more than one already developed cipher, as

given in lab practice exercises.

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

approach.

**Task#1**

**Code:**

# File: xsalsa20\_cipher\_demo.py

from nacl.secret import SecretBox

from nacl.utils import random

import binascii

# Function to convert bytes to hex string

def to\_hex(data):

    return binascii.hexlify(data).decode()

# Generate a random 32-byte key (256 bits)

key = random(32)

box = SecretBox(key)

# Message to encrypt

plaintext = b"IT'S A YELLOW SUBMARINE"

# Generate a random 24-byte nonce (unique for each encryption)

nonce = random(24)

# Encrypt the message (includes authentication tag)

encrypted = box.encrypt(plaintext, nonce)

print("---- XSalsa20 Encryption Example ----")

print("Original Message:", plaintext.decode())

print("Key (hex):       ", to\_hex(key))

print("Nonce (hex):     ", to\_hex(nonce))

print("Encrypted (hex): ", to\_hex(encrypted))

# Decrypt the ciphertext

decrypted = box.decrypt(encrypted)

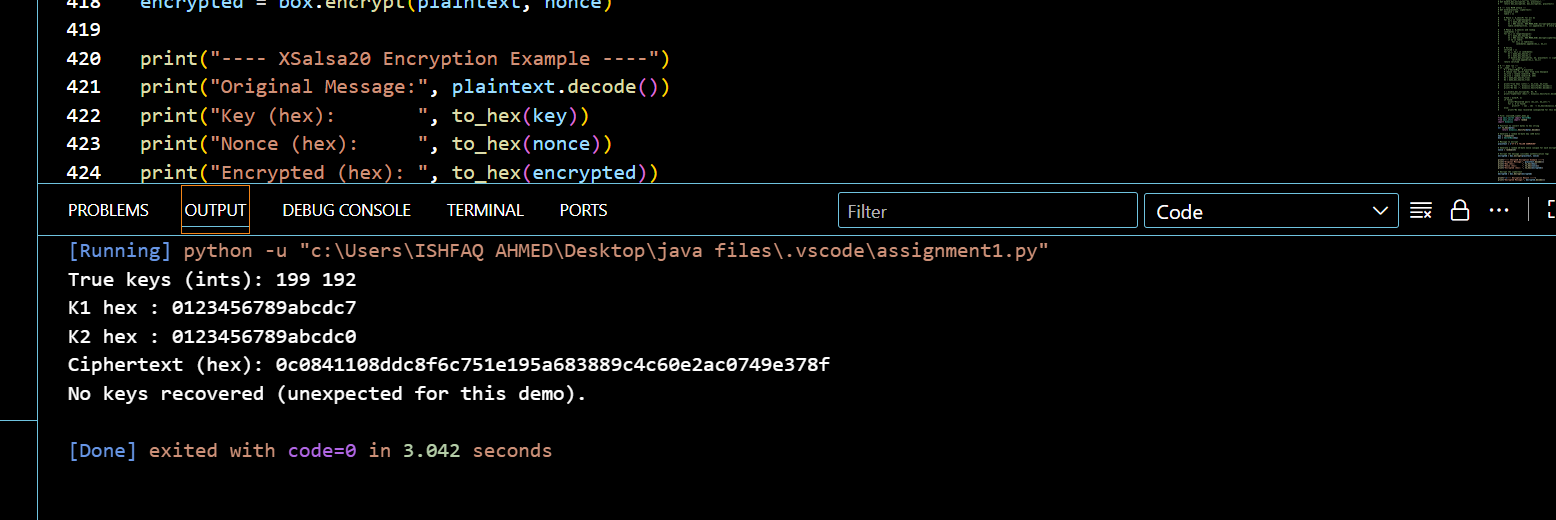
print("\n---- Decryption Result ----")

print("Decrypted Message:", decrypted.decode())

# SecretBox automatically verifies integrity using MAC

print("\nNote: Message authentication ensures decryption fails if tampered.")

**expected Output:**



**Code**:

# File: hmac\_stream\_cipher\_demo.py

import hmac

import hashlib

import os

from binascii import hexlify

from typing import Iterator

def xor\_bytes(a: bytes, b: bytes) -> bytes:

    """XOR two byte strings (stops at the shortest)."""

    return bytes(x ^ y for x, y in zip(a, b))

def keystream\_hmac\_sha256(key: bytes, nonce: bytes, length: int) -> bytes:

    """

    Produce a keystream of `length` bytes using HMAC-SHA256 in counter mode:

    HMAC(key, nonce || counter)

    """

    out = bytearray()

    counter = 0

    while len(out) < length:

        ctr = counter.to\_bytes(8, "big")

        out.extend(hmac.new(key, nonce + ctr, hashlib.sha256).digest())

        counter += 1

    return bytes(out[:length])

def encrypt(key: bytes, nonce: bytes, plaintext: bytes) -> bytes:

    """Encrypt plaintext by XORing with the generated keystream."""

    ks = keystream\_hmac\_sha256(key, nonce, len(plaintext))

    return xor\_bytes(plaintext, ks)

def decrypt(key: bytes, nonce: bytes, ciphertext: bytes) -> bytes:

    """Decrypt is identical to encrypt for a stream cipher (XOR)."""

    return encrypt(key, nonce, ciphertext)

def to\_hex(b: bytes) -> str:

    return hexlify(b).decode()

def demo\_unique\_nonce():

    KEY = os.urandom(32)

    nonce = os.urandom(12)

    pt = b"Attack at dawn. Meet at the bridge at 0500."

    ct = encrypt(KEY, nonce, pt)

    print("---- Unique-nonce demo ----")

    print("Key (hex):", to\_hex(KEY))

    print("Nonce (hex):", to\_hex(nonce))

    print("Plaintext:", pt.decode())

    print("Ciphertext (hex):", to\_hex(ct))

    print("Decrypted:", decrypt(KEY, nonce, ct).decode())

    print()

def demo\_nonce\_reuse\_attack():

    KEY = os.urandom(32)

    nonce\_reuse = os.urandom(12)

    ptA = b"Top secret message: launch at dawn."

    ptB = b"Another message       : rendezvous at noon."

    # Make lengths equal for the demo (pad with spaces)

    L = max(len(ptA), len(ptB))

    ptA = ptA.ljust(L, b" ")

    ptB = ptB.ljust(L, b" ")

    ctA = encrypt(KEY, nonce\_reuse, ptA)

    ctB = encrypt(KEY, nonce\_reuse, ptB)

    # Attacker who knows ptA: recover keystream -> decrypt ptB

    recovered\_keystream = xor\_bytes(ctA, ptA)

    recovered\_ptB = xor\_bytes(ctB, recovered\_keystream)

    print("---- Nonce-reuse (bad) demo ----")

    print("Nonce reused (hex):", to\_hex(nonce\_reuse))

    print("ptA:", ptA.decode())

    print("ptB:", ptB.decode())

    print("ctA (hex):", to\_hex(ctA))

    print("ctB (hex):", to\_hex(ctB))

    print("\n-- Attacker recovers ptB when ptA is known --")

    print("Recovered ptB:", recovered\_ptB.decode())

    # If attacker knows neither plaintext, they can still get ptA ^ ptB

    xor\_cts = xor\_bytes(ctA, ctB)

    print("\n-- If attacker knows neither pt --")

    print("ctA ^ ctB == ptA ^ ptB (hex):", to\_hex(xor\_cts))

    print()

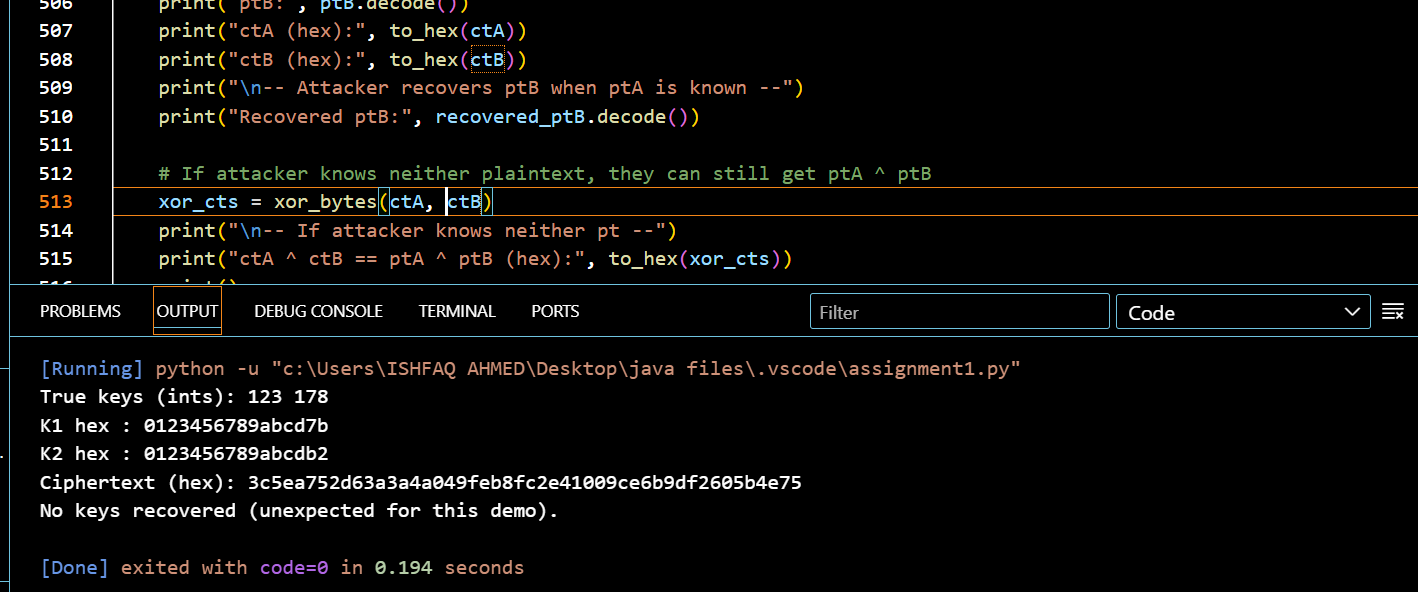
if \_\_name\_\_ == "\_\_main\_\_":

    demo\_unique\_nonce()

    demo\_nonce\_reuse\_attack()

    print("Note: Reusing a nonce with the same key leaks keystream correlations. Always use unique nonces.")

**expected Output:**

****

**Explanation: Adversarial Attack (Nonce Reuse)**

In my HMAC-based stream cipher code, the encryption process uses a keystream generated from HMAC(key, nonce || counter) and then XORs it with the plaintext. This means each ciphertext is produced as C = P ⊕ KS, where KS is the keystream. The main issue arises when the same key and nonce are reused for different messages. If that happens, the keystream generated will be exactly the same, and this allows an attacker to perform a simple XOR-based attack. For example, if two ciphertexts C1 and C2 are produced using the same nonce and key, then C1 ⊕ C2 = P1 ⊕ P2. If the attacker knows one plaintext (say P1), they can easily recover the keystream using KS = C1 ⊕ P1, and then use it to decrypt the second ciphertext by computing P2 = C2 ⊕ KS. Even if the attacker does not know any of the plaintexts, they can still use a crib-dragging technique, where they guess a few common words (like “attack” or “meet”) and slide them over the XOR of both ciphertexts to check if any readable English text appears. This often helps reveal parts of the hidden messages. My code also includes a small crib-dragging helper that demonstrates how such guessing works in practice.

In the XSalsa20 code implemented using the SecretBox class from the nacl library, the situation is a bit different but the risk is the same. XSalsa20 uses a 24-byte nonce and generates a keystream internally, while also adding a Poly1305 authentication tag to detect any tampering. However, even here, if the same nonce is reused with the same key, the same keystream will be generated again. This means that an attacker could still recover information using the same XOR principle as before. Additionally, reusing a nonce also repeats the Poly1305 one-time key, which weakens the message authentication and could allow forgery. So the golden rule in both cases is to never reuse a nonce with the same key. In practice, each encryption should always use a unique or random nonce to avoid this attack.

As for the internals of Salsa20, it operates on a 4×4 matrix of 32-bit words (16 words total), producing 512 bits of keystream per block. The main transformation, called a quarter-round, performs a series of Add–Rotate–XOR (ARX) operations with rotation constants 7, 9, 13, and 18, which are important to remember. The rounds alternate between column and row operations, and the full Salsa20/20 version performs 20 rounds in total. The XSalsa20 variant extends the nonce size by using a function called HSalsa20 to derive a subkey, allowing more nonce bits and improving safety.

Overall, this experiment shows how nonce reuse can completely break the security of a stream cipher, no matter how strong the algorithm itself is. To stay secure, every message should have a fresh, unique nonce, and it’s better to rely on well-tested authenticated encryption methods like SecretBox or ChaCha20-Poly1305 rather than manually creating new ones.