# Lab 03

## Streams Ciphers and Pseudo Random Generator (PRG)

This lab will introduce students to Streams Ciphers and Pseudo Random Generator (PRG). We will first learn the working principles of stream ciphers, including how they use a key and a pseudo-random generator to produce a keystream. Implement a basic stream cipher to see how XOR-based encryption works. In this lab students will also learn how PRGs generate keystreams for stream ciphers.

## Activity Outcomes:

This lab teaches you the following topics:

- Understand the Basics of Stream Ciphers
- Implement a Simple Stream Cipher (e.g., Vernam/RC4)
- Understand Pseudo-Random Generator (PRG) and Its Role in Stream Ciphers

## Instructor Note:

| ***Sr.No*** | ***Allocated Time*** | ***Level of Complexity*** | ***CLO Mapping*** |
|---|---|---|---|
| ***1*** | ***50*** | ***Medium*** | ***CLO-6*** |
| ***2*** | ***50*** | ***Medium*** | |
| ***3*** | ***Home –Practice-Activity*** | ***High*** | |

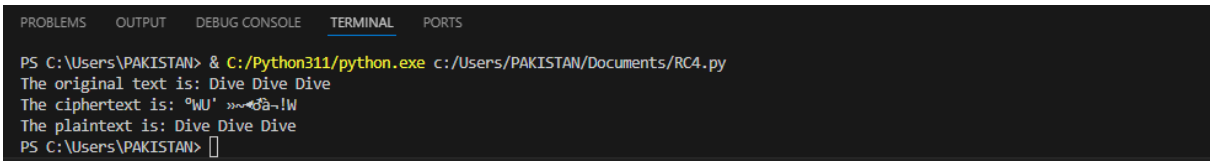
## Practice -Lab Activity 1: ARC4

The RC4 stream cipher was created by Ron Rivest in 1987. RC4 was classified as a trade secret by RSA Security but was eventually leaked to a message board in 1994. RC4 was originally trademarked by RSA Security so it is often referred to as ARCFOUR or ARC4 to avoid trademark issues. ARC4 would later become commonly used in a number of encryption protocols and standards such as SSL, TLS, WEP, and WPA. In 2015, it was prohibited for all versions of TLS by RFC 7465. ARC4 has been used in many hardware and software implementations. One of the main advantages of ARC4 is its speed and simplicity, which you will notice in the following code:

"""
Implement the ARC4 stream cipher.
"""

```
def arc4crypt(data, key):
    x = 0
    box = list(range(256))  # Ensure box is a list, not range object
    # Key-scheduling algorithm (KSA)
    for i in range(256):
        x = (x + box[i] + ord(key[i % len(key)])) % 256
        box[i], box[x] = box[x], box[i]  # Swap values

    x = 0
    y = 0
    out = []
    # Pseudo-random generation algorithm (PRGA)
    for char in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x]  # Swap values
        out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256]))  # XOR with keystream

    return ''.join(out)

# Testing the ARC4 encryption and decryption
key = 'SuperSecretKey!!'
origtext = 'Dive Dive Dive'
ciphertext = arc4crypt(origtext, key)
plaintext = arc4crypt(ciphertext, key)

print('The original text is: {}'.format(origtext))
print('The ciphertext is: {}'.format(ciphertext))
print('The plaintext is: {}'.format(plaintext))
```

OUTPUT




## Practice -Lab Activity 2: Vernam Cipher

The Vernam cipher was developed by Gilbert Vernam in 1917. It is a type of onetime pad for data streams and is considered to be unbreakable. The algorithm is symmetrical, and the plaintext is combined with a random stream of data of the same length using the Boolean XOR function; the Boolean XOR function is also known as the Boolean exclusive OR function. Claude Shannon would later mathematically prove that it is unbreakable. The characteristics of the Vernam cipher include:

- The plaintext is written as a binary sequence of 0s and 1s.
- The secret key is a completely random binary sequence and is the same length as the plaintext.
- The ciphertext is produced by adding the secret key bitwise modulo 2 to the plaintext.

- XOR Operation:
  - XOR is a bitwise operation, and it's symmetric. This means:
    - `a ^ b` will encrypt `a` with `b`.
    - Applying XOR again with `b` reverses the operation: `(a ^ b) ^ b = a`.
    - This is why using the same function for both encryption and decryption works in the Vernam cipher.

One of the disadvantages of using an OTP is that the keys must be as long as the message it is trying to conceal; therefore, for long messages, you will need a long key:

**CODE**

```
def VernamEncDec(text, key):
    result = ""
    ptr = 0
    for char in text:
        result += chr(ord(char) ^ ord(key[ptr]))  # XOR each character with the key
        ptr += 1
        if ptr == len(key):  # Reset the key pointer if it reaches the end of the key
            ptr = 0
    return result

# Key for Vernam Cipher
```

```
key = "thisismykey12345"

while True:
    input_text = input("\nEnter Text To Encrypt:\t")

    # Encrypt the input text
    ciphertext = VernamEncDec(input_text, key)
    print("\nEncrypted Vernam Cipher Text:\t" + ciphertext)

    # Decrypt the ciphertext
    plaintext = VernamEncDec(ciphertext, key)
    print("\nDecrypted Vernam Cipher Text:\t" + plaintext)
```

OUTPUT

```
================ RESTART: C:/Users/PAKISTAN/Documents/Vernam.py ================

Enter Text To Encrypt:  Cryptographic protocols

Encrypted Vernam Cipher Text:   7
XQDG

Decrypted Vernam Cipher Text:   Cryptographic protocols
```

## Home-Practice - Activity 3: Salsa20 Cipher

The Salsa20 cipher was developed in 2005 by Daniel Bernstein, and submitted to eSTREAM. The Salsa20/20 (Salsa20 with 20 rounds) is built on a pseudorandom function that is based on add-rotate-xor (ARX) operations. ARX algorithms are designed to have their round function support modular addition, fixed rotation, and XOR. These ARX operations are popular because they are relatively fast and cheap in hardware and software, and because they run in constant time, and are therefore immune to timing attacks. The rotational cryptanalysis technique attempts to attack such round functions.

The core function of Salsa20 maps a 128-bit or 256-bit key, a 64-bit nonce/IV, and a 64-bit counter to a 512-bit block of the keystream. Salsa20 provides speeds of around 4–14 cycles per byte on modern x86 processors and is considered acceptable hardware performance. The numeric indicator in the Salsa name specifies the number of encryption rounds. Salsa20 has 8, 12, and 20 variants. One of the biggest benefits of Salsa20 is that Bernstein has written several implementations that have been released to the public domain, and the cipher is not patented.

Salsa20 is composed of sixteen 32-bit words that are arranged in a 4×4 matrix. The initial state is made up of eight words of key, two words of the stream position, two words for the nonce/IV, and four fixed words or constants. The initial state would look like the following:

| | | | |
|---|---|---|---|
| Constant | Key | Key | Key |
| Key | Constant | Nonce | Nonce |
| Stream | Stream | Constant | Key |
| Key | Key | Key | Constant |

The Salsa20 core operation is the quarter-round that takes a four-word input and produces a four-word output. The quarter-round is denoted by the following function: QR(a, b, c, d). The odd-numbered rounds apply QR(a, b, c, d) to each of the four columns in the preceding 4×4 matrix; the even-numbered rounds apply the rounding to each of the four rows. Two consecutive rounds (one for a column and one for a row) operate together and are known as a double-round. To help understand how the rounds work, let us first examine a 4×4 matrix with labels from 0 to 15:

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

The first double-round starts with a quarter-round on column 1 and row 1. The first QR round examines column 1, which contains 0, 4, 8, and 12. The second QR round examines row 1, which contains 0, 1, 2, 3. The second doubleround picks up starting at the second column and second row position (5, 9, 13, 1). The even round picks up 5, 5, 7, 4. Notice that the starting cell is the same for each double-round:

| DOUBLE-ROUND 1 | DOUBLE-ROUND 3 |
|---|---|
| QR(0, 4, 8, 12) | QR(10, 14, 2, 6) |
| QR(0, 1, 2, 3) | QR(10, 11, 8, 9) |
| **DOUBLE-ROUND 2** | **DOUBLE-ROUND 4** |
| QR(5, 9, 13, 1) | QR(15, 3, 7, 11) |
| QR(5, 6, 7, 4) | QR(15, 12, 13, 14) |

A couple libraries are available that will help simplify the Salsa20 encryption scheme. You can access the salsa20 library by doing a pip install salsa20.

Once you have the library installed, you can use the XSalsa20_keystream to generate a keystream of the desired length, or you can pass any message plaintext or ciphertext) to have it XOR'd with the keystream. All values must be binary strings that include str for Python 2 or the byte for Python 3. Here, you will see a Python implementation of the salsa20 library:

```
from salsa20 import XSalsa20_xor
```

```
from os import urandom
IV = urandom(24)
KEY = b'*secret**secret**secret**secret*'
ciphertext = XSalsa20_xor(b"IT'S A YELLOW SUBMARINE", IV, KEY)
print(XSalsa20_xor(ciphertext, IV, KEY).decode())
```

```python
from nacl.secret import SecretBox
from nacl.utils import random

# The key must be 32 bytes for XSalsa20
key = b'*secret**secret**secret**secret*'

# Create a SecretBox, which uses XSalsa20 internally
box = SecretBox(key)

# The nonce must be 24 bytes for XSalsa20
nonce = random(24)

# Encrypting the message
message = b"IT'S A YELLOW SUBMARINE"
ciphertext = box.encrypt(message, nonce)

# Decrypting the message
decrypted = box.decrypt(ciphertext)

print(decrypted.decode())  # Should output: IT'S A YELLOW SUBMARINE
```

Code Output

```
IT'S A YELLOW SUBMARINE
```

One of the reasons why you should be familiar with Salsa20 is that it is consistently faster than AES. It is recommended to use Salsa20 for encryption in typical cryptographic applications.

## Home-Task - Activity 4: Salsa20 Cipher

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.