Project Report

Shamir's Secret Sharing: Key Management and Distributed Password Recovery

Information Security

Table of Contents

Introduction

SSS a cryptographic method by Shamir allows users to distribute a secret (such as passwords or keys) among several participants after it is cut into multiple shares. A sufficient number of combined shares stands as the only condition needed to reconstruct the secret. Adi Shamir introduced the method in 1979 as an approach which remains widely used for protected key storage in networks without any trusted authority holding the full secret. This project implements Shamir's Secret Sharing to reveal its deployment in both secure key maintenance systems and distributed password recovery systems. Secure key division happens when a secret becomes shares and these shares receive secure distribution before they get reconstructed with the minimum required data.

Secret Sharing Implementation

Overview

- **Polynomial Generation:** Polynomial generation is the process of creating a polynomial with the constant term serving as the secret. The polynomial has a degree of t-1, where t is the number of individuals required to reconstruct the secret.

```
# step 1: randomly generate a polynomial of degree t-1
coefficients = [secret] + [random.randint(0, p-1) for _ in range(t-1)]
```

- **Share Creation**: Each member receives a unique share that corresponds to a point on the polynomial.
- **Secret Reconstruction**: The secret is rebuilt by merging a sufficient number of shares (at least t shares) using Lagrange interpolation.

Mathematical Explanation

Polynomial: A polynomial of degree t-1 is generated where:

- The constant term is the secret.
- The remaining coefficients are randomly chosen.
- The polynomial is evaluated at different points (i.e., the participant numbers) to create the shares.

Lagrange Interpolation: Given the shares (x1,y1),(x2,y2),…,(xt,yt) we reconstruct the secret using Lagrange interpolation:

$$S = \sum_{i=1}^{t} y_i \cdot L_i(0)$$

Where Li(x) is the Lagrange basis polynomial:

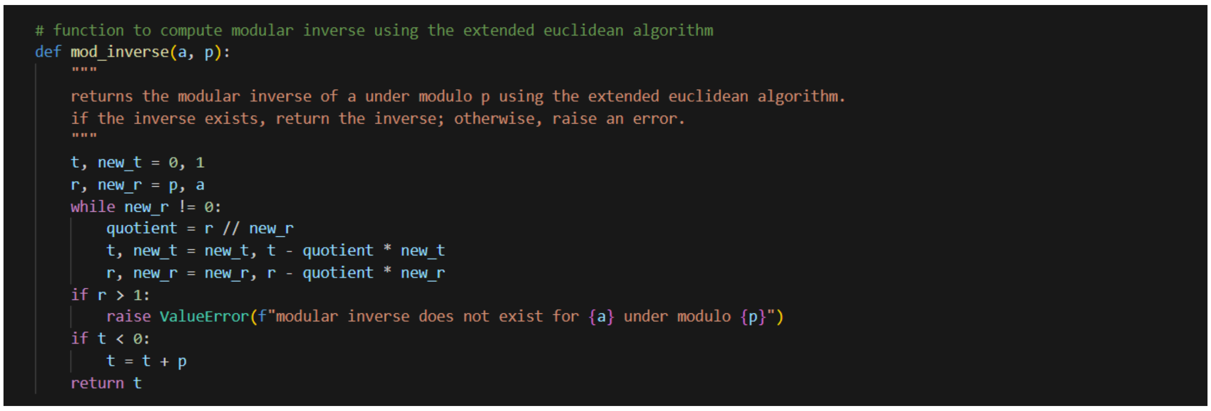
$$L_i(x) = \prod_{1 \le j \le t, j \ne i} \frac{(x - x_j)}{(x_i - x_j)}$$

This interpolation formula allows the secret to be calculated by combining the shares.

Code Explanation

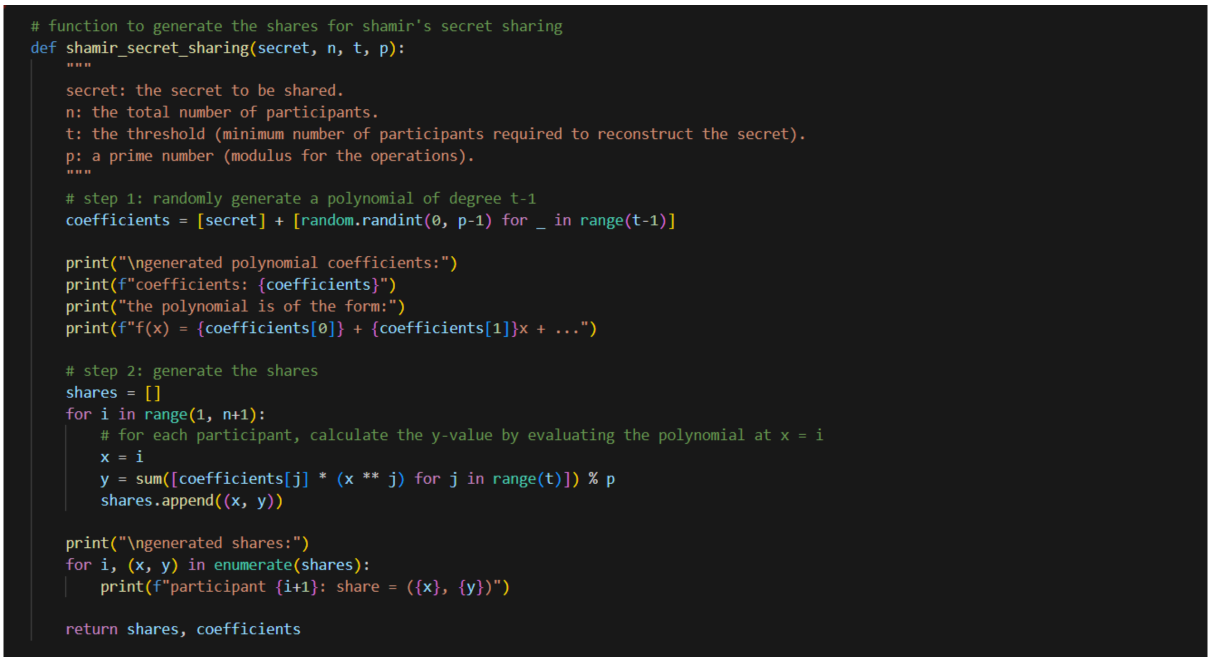
The implementation involves:

- **Secret Reconstruction**: The coefficients are randomly generated, and the polynomial is constructed.
- **Share Creation**: For each participant, the polynomial is evaluated at their assigned x value to generate their respective share.
- **Polynomial Generation**: Lagrange interpolation is used to reconstruct the secret by combining the shares.



```python
# function to compute modular inverse using the extended euclidean algorithm
def mod_inverse(a, p):
    """
    returns the modular inverse of a under modulo p using the extended euclidean algorithm.
    if the inverse exists, return the inverse; otherwise, raise an error.
    """
    t, new_t = 0, 1
    r, new_r = p, a
    while new_r != 0:
        quotient = r // new_r
        t, new_t = new_t, t - quotient * new_t
        r, new_r = new_r, r - quotient * new_r
    if r > 1:
        raise ValueError(f"modular inverse does not exist for {a} under modulo {p}")
    if t < 0:
        t = t + p
    return t
```

The **mod_inverse** function uses the **Extended Euclidean Algorithm** to calculate the modular inverse, which is essential for Lagrange interpolation (Code Reference: **mod_inverse function**).



```python
# function to generate the shares for shamir's secret sharing
def shamir_secret_sharing(secret, n, t, p):
    """
    secret: the secret to be shared.
    n: the total number of participants.
    t: the threshold (minimum number of participants required to reconstruct the secret).
    p: a prime number (modulus for the operations).
    """
    # step 1: randomly generate a polynomial of degree t-1
    coefficients = [secret] + [random.randint(0, p-1) for _ in range(t-1)]

    print("\ngenerated polynomial coefficients:")
    print(f"coefficients: {coefficients}")
    print("the polynomial is of the form:")
    print(f"f(x) = {coefficients[0]} + {coefficients[1]}x + ...")

    # step 2: generate the shares
    shares = []
    for i in range(1, n+1):
        # for each participant, calculate the y-value by evaluating the polynomial at x = i
        x = i
        y = sum([coefficients[j] * (x ** j) for j in range(t)]) % p
        shares.append((x, y))

    print("\ngenerated shares:")
    for i, (x, y) in enumerate(shares):
        print(f"participant {i+1}: share = ({x}, {y})")

    return shares, coefficients
```

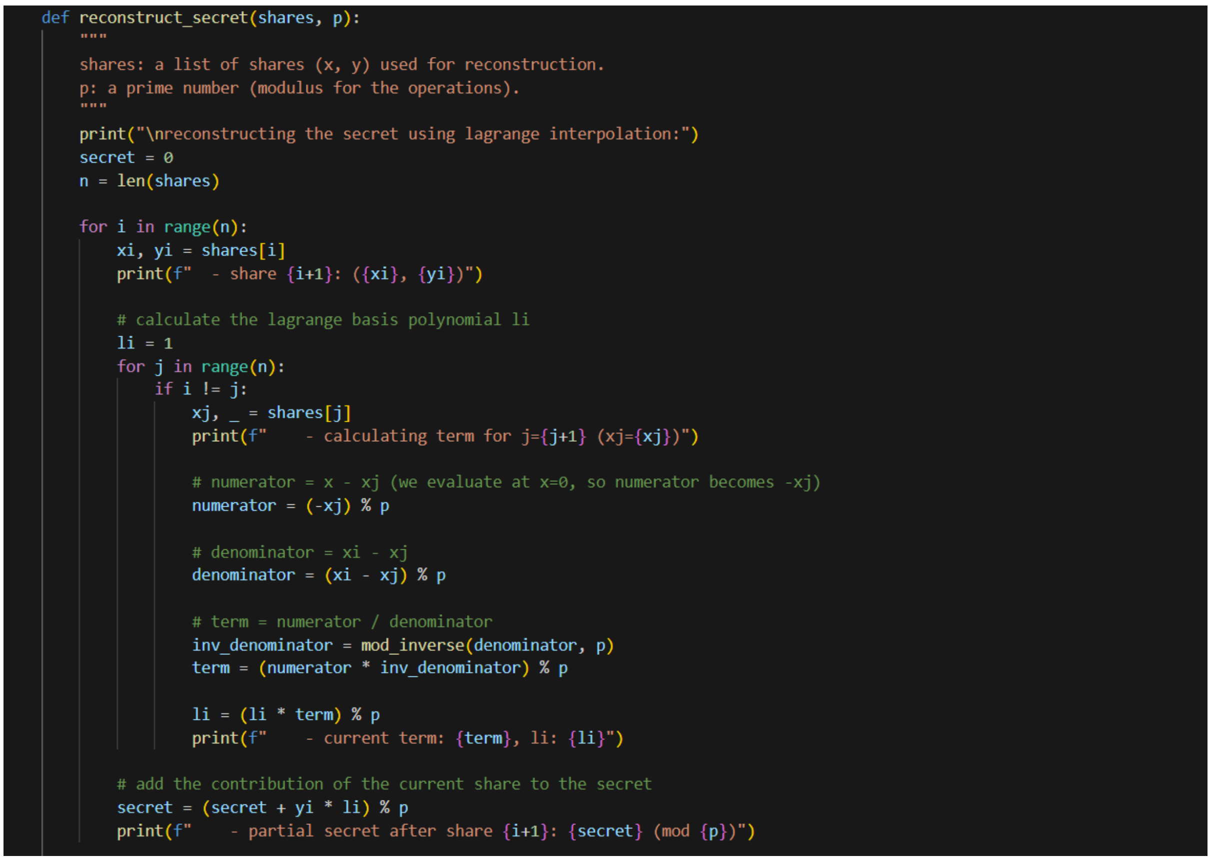
The **shamir_secret_sharing** function generates the polynomial and calculates the shares by evaluating the polynomial at different x values (Code Reference: **shamir_secret_sharing function**).

Distributed Password Recovery

Shamir's Secret Sharing is used to divide the password into multiple shares in a distributed password recovery system. Every participant gets a share, and the original password can only be recovered by combining the shares of a specific number of participants (threshold t).

Process

- **Password Splitting**: Shamir's Secret Sharing is used to divide the password (secret) into shares. A minimum number of participants is required to recover the password, which is why the threshold t was set.
- **Share Distribution**: The generated shares are securely distributed to the participants.
- **Password Recovery**: If the number of shares reaches the threshold t, a participant who has forgotten the password can use their share and combine it with others to retrieve the original password.



```python
def reconstruct_secret(shares, p):
    """
    shares: a list of shares (x, y) used for reconstruction.
    p: a prime number (modulus for the operations).
    """
    print("\nreconstructing the secret using lagrange interpolation:")
    secret = 0
    n = len(shares)

    for i in range(n):
        xi, yi = shares[i]
        print(f"  - share {i+1}: ({xi}, {yi})")

        # calculate the lagrange basis polynomial li
        li = 1
        for j in range(n):
            if i != j:
                xj, _ = shares[j]
                print(f"    - calculating term for j={j+1} (xj={xj})")

                # numerator = x - xj (we evaluate at x=0, so numerator becomes -xj)
                numerator = (-xj) % p

                # denominator = xi - xj
                denominator = (xi - xj) % p

                # term = numerator / denominator
                inv_denominator = mod_inverse(denominator, p)
                term = (numerator * inv_denominator) % p

                li = (li * term) % p
                print(f"    - current term: {term}, li: {li}")

        # add the contribution of the current share to the secret
        secret = (secret + yi * li) % p
        print(f"    - partial secret after share {i+1}: {secret} (mod {p})")
```

The **reconstruct_secret** function uses **Lagrange Interpolation** to reconstruct the secret based on the shares provided by participants (Code Reference: **reconstruct_secret function**).

Security Impact Analysis

Threshold Security

The Secret of Shamir As long as fewer than t shares are compromised, sharing is by nature safe. The bare minimum of shares needed to rebuild the secret is determined by the threshold t. The secret is safe even if an attacker manages to acquire less than t shares.

Resistance to Attacks

- **Share Compromise**: The secret cannot be recovered if an attacker manages to access less than t shares. However, the attacker can use Lagrange interpolation to recreate the secret if they acquire t or more shares.
- **Eavesdropping:** Even in the event that a share is intercepted, an attacker cannot recreate the secret without acquiring enough shares because the shares are distributed independently.

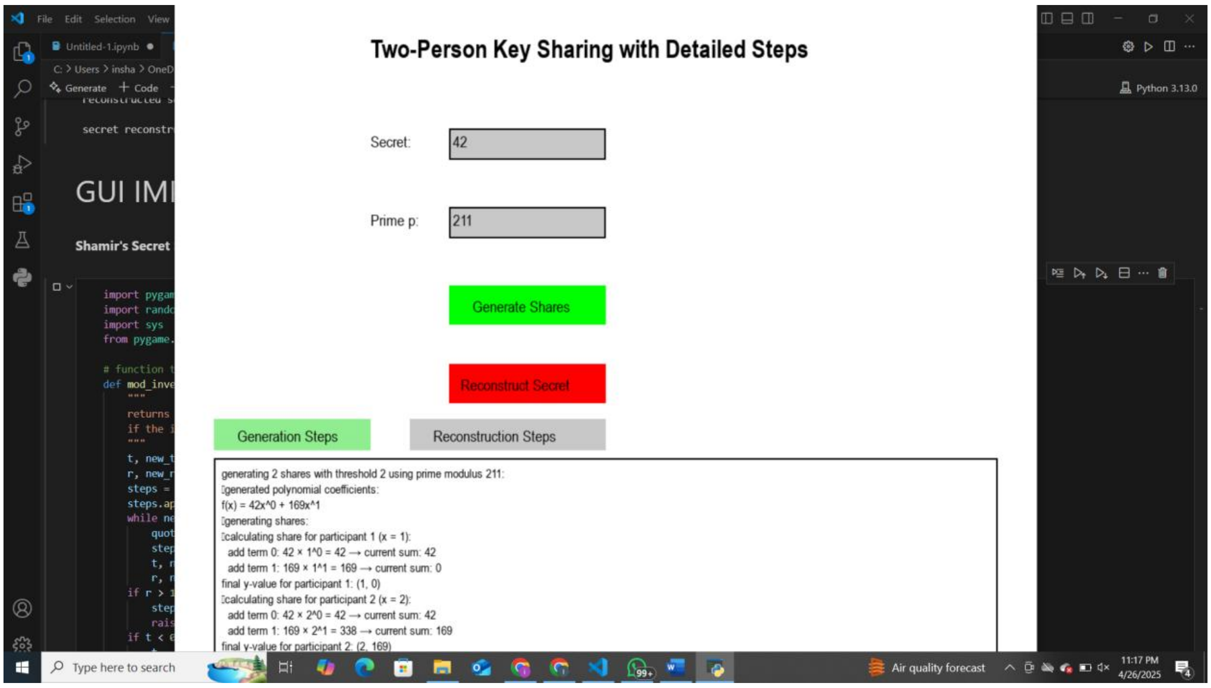
Weaknesses and Mitigations

- **Weakness**: The system is susceptible to assaults if a low threshold t (too few shares) is needed to rebuild the secret. With little effort, an adversary may acquire enough shares to reconstitute the secret.

- **Mitigation**: Setting a high enough threshold t to ensure that multiple participants must work together to reconstruct the secret is crucial to reducing this.

GUI Implemented Code

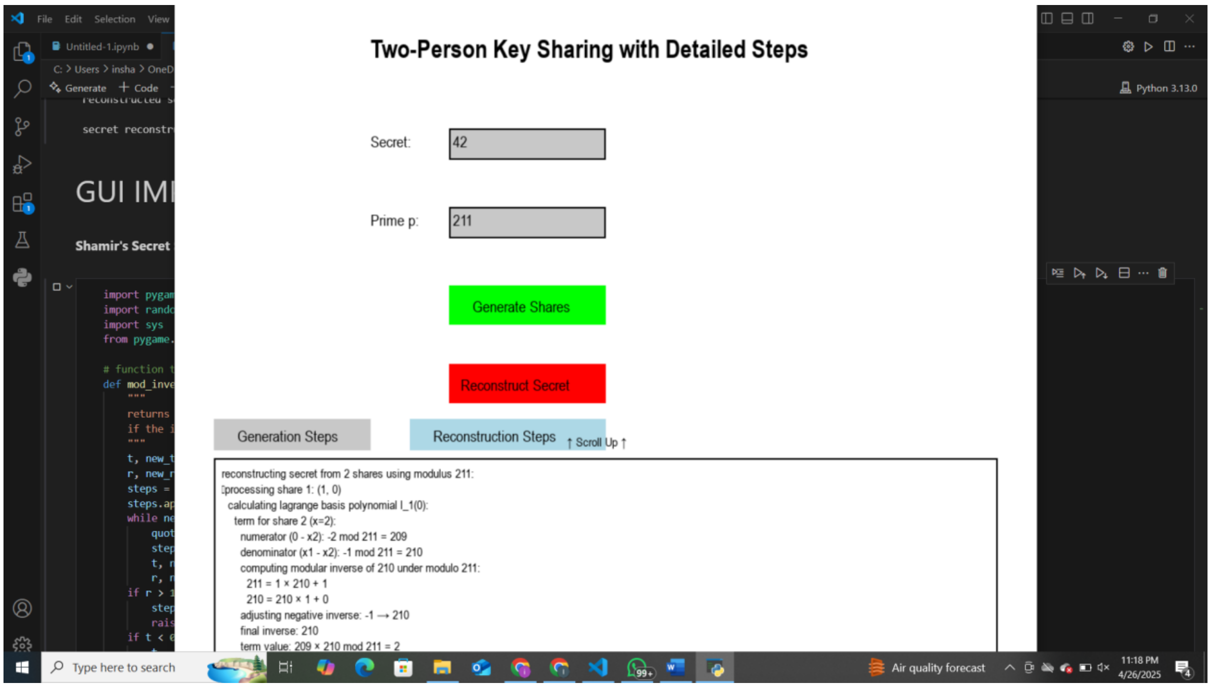
For Two-Person Key Sharing



Secret: 42

Prime p: 211

```
generating 2 shares with threshold 2 using prime modulus 211:
generated polynomial coefficients:
f(x) = 42x^0 + 169x^1
generating shares:
calculating share for participant 1 (x = 1):
  add term 0: 42 × 1^0 = 42 → current sum: 42
  add term 1: 169 × 1^1 = 169 → current sum: 0
final y-value for participant 1: (1, 0)
calculating share for participant 2 (x = 2):
  add term 0: 42 × 2^0 = 42 → current sum: 42
  add term 1: 169 × 2^1 = 338 → current sum: 169
final y-value for participant 2: (2, 169)
```

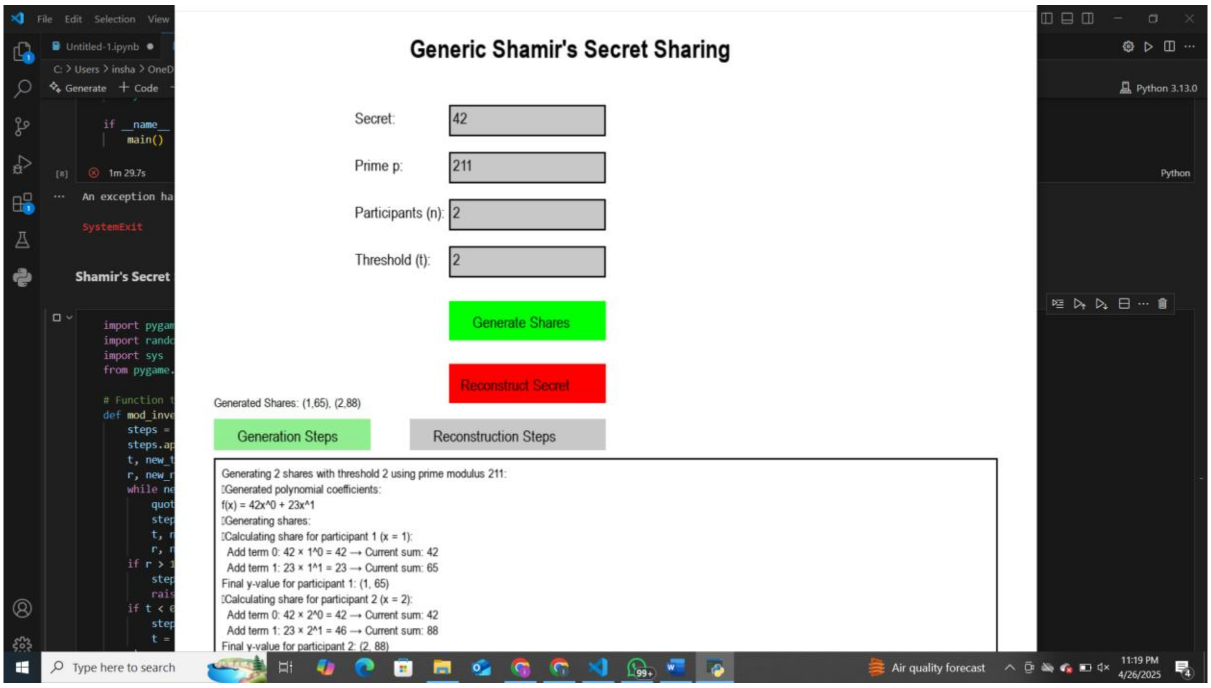


Secret: 42

Prime p: 211

```
reconstructing secret from 2 shares using modulus 211:
processing share 1: (1, 0)
  calculating lagrange basis polynomial l_1(0):
    term for share 2 (x=2):
      numerator (0 - x2): -2 mod 211 = 209
      denominator (x1 - x2): -1 mod 211 = 210
      computing modular inverse of 210 under modulo 211:
        211 = 1 × 210 + 1
        210 = 210 × 1 + 0
      adjusting negative inverse: -1 → 210
      final inverse: 210
      term value: 209 × 210 mod 211 = 2
```

For N-Persons Key Sharing



Generic Shamir's Secret Sharing

Secret: 42

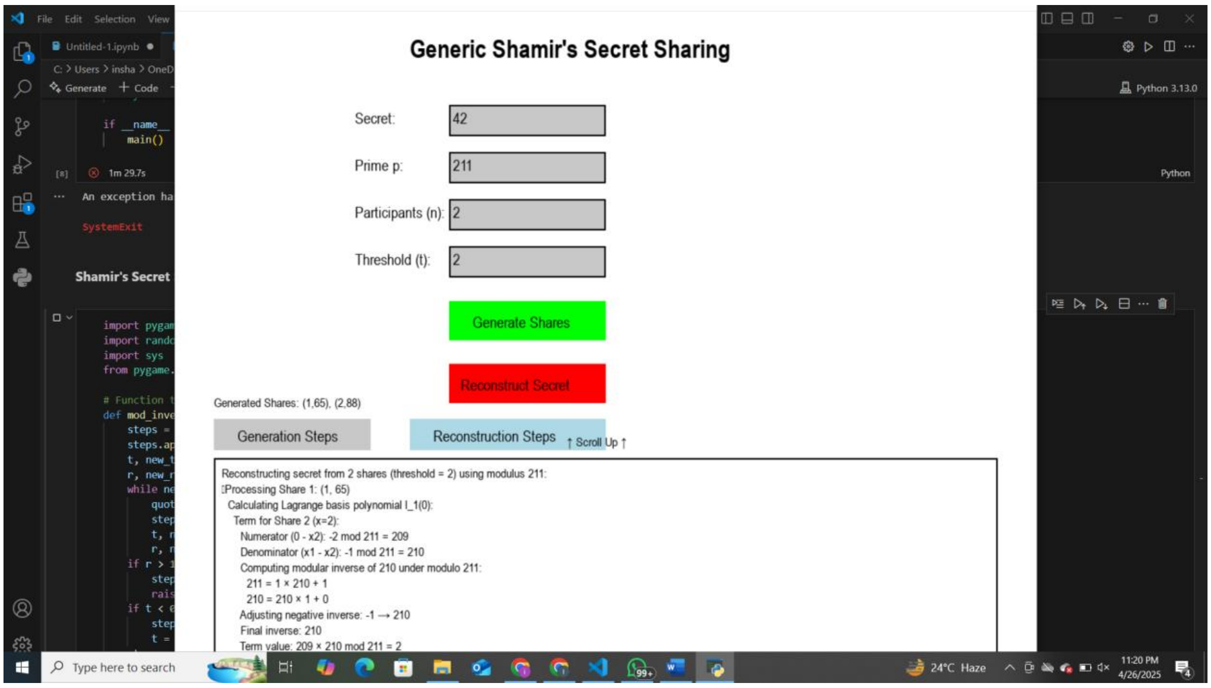
Prime p: 211

Participants (n): 2

Threshold (t): 2

Generated Shares: (1,65), (2,88)

```
Generating 2 shares with threshold 2 using prime modulus 211:
Generated polynomial coefficients:
f(x) = 42x^0 + 23x^1
Generating shares:
Calculating share for participant 1 (x = 1):
 Add term 0: 42 × 1^0 = 42 → Current sum: 42
 Add term 1: 23 × 1^1 = 23 → Current sum: 65
Final y-value for participant 1: (1, 65)
Calculating share for participant 2 (x = 2):
 Add term 0: 42 × 2^0 = 42 → Current sum: 42
 Add term 1: 23 × 2^1 = 46 → Current sum: 88
Final y-value for participant 2: (2, 88)
```



Generic Shamir's Secret Sharing

Secret: 42

Prime p: 211

Participants (n): 2

Threshold (t): 2

Generated Shares: (1,65), (2,88)

```
Reconstructing secret from 2 shares (threshold = 2) using modulus 211:
Processing Share 1: (1, 65)
  Calculating Lagrange basis polynomial l_1(0):
   Term for Share 2 (x=2):
    Numerator (0 - x2): -2 mod 211 = 209
    Denominator (x1 - x2): -1 mod 211 = 210
    Computing modular inverse of 210 under modulo 211:
     211 = 1 × 210 + 1
     210 = 210 × 1 + 0
    Adjusting negative inverse: -1 → 210
    Final inverse: 210
    Term value: 209 × 210 mod 211 = 2
```

Summary

This project effectively used Shamir's Secret Sharing to offer secure key management and distributed password recovery. The concept was demonstrated using a simple example, and the Lagrange Interpolation technique was used to safely recover the secret.
The security assessment indicates that the system's strongest point is the threshold parameter. As long as fewer than t shares are compromised, the data is secure. However, the system's security is jeopardized if the threshold is set too low.
This approach is effective for distributed systems where no single individual should have access to

the entire secret, such as multi-signature schemes, password recovery systems, and key management for cryptographic systems.

References

1. Shamir, A. (1979). "How to share a secret." *Communications of the ACM*, 22(11), 612-613.
2. Stinson, D. R. (2005). "Cryptography: Theory and Practice". CRC Press.
3. Menezes, A., van Oorschot, P., & Vanstone, S. (1997). "Handbook of Applied Cryptography". CRC Press.