20CYS205 – MODERN CRYPTOGRAPHY

Threshold Secret Sharing Schemes

Under the guidance of

Dr. Praveen
Assistant Professor
Amrita Vishwa Vidyapeetham
Coimbatore

Submitted by

Sree Sharvesh S S	– CB.EN.U4CYS22061
Shravan Krishnan G	– CB.EN.U4CYS22062
Vajjula Satya Siddardha	– CB.EN.U4CYS22063
Yallanuru Kishan Sai	– CB.EN.U4CYS22064

TIFAC-CORE in Cyber Security
Amrita School Of Engineering
Amrita Vishwa Vidyapeetham
Coimbatore - 641112

Acknowledgement

First of all, we would like to express our gratitude to our Mentors, **Dr. Praveen** Assistant Professors, TIFAC-CORE inCyber Security, Amrita Vishwa Vidyapeetham, Coimbatore, for his valuable suggestions and timely feedbacks during the course of this major project. It was indeed a great support from him that helped us successfully fulfill this work

I would like to thank **Dr.M.Sethumadhavan**, Professor and Head of Department, TIFAC-CORE in Cyber Security, for his constant encouragement and guidance through-out the progress of this major project.

I convey special thanks to our friends for listening to our ideas and contributing their thoughts concerning the project. All those simple doubts from their part have also made us think deeper and understand about this work.

In particular, we would like also like to extend our gratitude to all the other faculties of TIFAC-CORE in Cyber Security and all those people who have helped us in many ways for the successful completion of this project.

TABLE OF CONTENTS

Introduction:-

In the realm of cryptography, the Secret Sharing Scheme devised by Adi Shamir, George Blakley, and Louis Mignotte stands as a pioneering solution for secure information distribution. This documentation explores the core principles, mathematical foundations, and practical applications of their scheme. By distributing a secret among multiple participants, the approach ensures confidentiality, making it a crucial tool in safeguarding sensitive information across diverse contexts. Join us as we unravel the simplicity and effectiveness of Shamir, Blakley, and Mignotte's secret sharing scheme.

Shamir’s Secret Sharing Scheme:-

Shamir's secret sharing (SSS) is an efficient secret sharing algorithm for distributing private information (the "secret") among a group. The secret cannot be revealed unless a quorum of the group acts together to pool their knowledge. To achieve this, the secret is mathematically divided into parts (the "shares") from which the secret can be reassembled only when a sufficient number of shares are combined. SSS has the property of information-theoretic security, meaning that even if an attacker steals some shares, it is impossible for the attacker to reconstruct the secret unless they have stolen the quorum number of shares.

Algorithm:
Shamir's reconstruction algorithm is the heart of Shamir's Secret Sharing scheme, allowing us to piece together the original secret from a minimum number of its shares. Here's a breakdown of how it works:

1. Polynomial Construction:
    - The secret (let's call it S) is treated as a constant term in a polynomial of degree t-1 (where t is the threshold).
    - Random coefficients are chosen for the higher order terms of the polynomial.
    - This polynomial represents the "hidden curve" containing the secret point (S).
2. Share Generation:
    - The polynomial is evaluated at t distinct, public points (x1, x2, ..., xt).
    - The resulting t values (y1, y2, ..., yt) become the shares to be distributed.
3. Share Combination:

- When at least t shares are gathered, their corresponding (x, y) values become known.
- These values represent points on the hidden curve.
- Using Lagrange interpolation, a unique polynomial can be reconstructed based on these t points.
- This reconstructed polynomial will be identical to the original one created in step 1.

4. Secret Recovery:
   - The secret (S) is the constant term of the reconstructed polynomial.
   - Evaluating the reconstructed polynomial at x=0 reveals the secret S.

Blakley's Secret Sharing Scheme:-

Blakley's secret sharing scheme is a method that uses geometric principles to share a secret. The scheme involves splitting a secret into multiple parts and distributing them among selected parties. The secret can be recovered when these parties collaborate.
In three dimensions, each share is a plane, and the secret is the point where three shares intersect. Two shares are not enough to determine the secret, but they can narrow it down to the line where the two planes intersect.

Algorithm:
Blakley's algorithm, unlike Shamir's, is a geometric secret sharing scheme. It offers a different but equally effective way to divide a secret into shares and reconstruct it later. Here's how it works:

1. Secret Representation:
   - The secret (let's call it S) is represented as a single point in a two-dimensional grid.
   - The grid can be any shape, but triangles, squares, and hexagons are common choices.
2. Share Generation:
   - The grid is overlaid with t-1 geometric shapes (called "hyperplanes") in such a way that the secret point (S) lies completely within one of these shapes.
   - Each remaining portion of the grid outside the chosen hyperplane becomes a share.
3. Share Distribution:

- The t shares are distributed to different participants.

4. Secret Reconstruction:
   - When at least t participants come together, they bring their shares.
   - The secret point (S) lies at the intersection of t-1 hyperplanes (represented by the shares).
   - This intersection point can be easily calculated geometrically.

Mignotte's Secret Sharing Scheme:-

The Mignotte secret sharing scheme is a method for distributing a secret among a group of participants. It uses special sequences of integers called the (k, n)-Mignotte sequences. These sequences are made up of n integers that are pairwise coprime. The product of the smallest k integers is greater than the product of the k − 1 largest integers. Mignotte's scheme is a representative of CRT based (w,N)-threshold secret sharing schemes. A generalization of the scheme allows modules that are not necessarily pairwise coprime.

Algorithm:

1. Secret Generation: The secret (S) is an integer.
2. Share Generation:
   - A sequence of n integers, m1, m2, ..., m_n, is generated such that the following conditions are met:
     - All integers are pairwise coprime, meaning that they have no common factors other than 1.
     - The product of the first k integers is greater than S, and the product of the last n-k integers is less than S.
   - For each integer mi, a share is generated by evaluating the following polynomial at mi:

     $f(x) = x^k * S$

3.Share Distribution: The n shares are distributed to n participants.

4.Secret Reconstruction:

- If at least k shares are gathered, the secret can be reconstructed using the Chinese remainder theorem.
- The shares are used to form a system of n equations with n unknowns.
- The Chinese remainder theorem can be used to solve this system of equations to find the secret S.

Implementation Code:-
Python & Flask:-

App.py

```python
from flask import Flask, render_template, request, session
from shamir import *

app = Flask(__name__)
app.secret_key = 'hello'
#---------------------------------- mignotte ----------------------------------------
from random import randint

def generate_prime(bitsize,m):
    while True:
        num = randint(2**(bitsize-1), 2**bitsize - 1)
        if isprime(num) and num not in m:
            return num

def isprime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def solve_crt(rem, mod):
    M = 1
    for m in mod:
        M *= m

    result = 0
    for i in range(len(rem)):
        Mi = M // mod[i]
        if Mi == 0:
            continue
        Mi_inv = modinv(Mi, mod[i])
        result += rem[i] * Mi * Mi_inv

    return result % M

def modinv(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

#-----------------------------------mignotte----------------------------------------
```

```python

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

def get_prime_input(p):
    while True:
        if is_prime(p):
            return p
        else:
            print("Invalid input. Please enter a prime number.")

def inverse_matrix_mod_p(matrix, p):
    det_inv = mod_inv(matrix_determinant(matrix, p), p)
    adj_matrix = matrix_adjugate(matrix)
    return scalar_multiply(adj_matrix, det_inv, p)

def matrix_determinant(matrix, p):
    return (matrix[0][0] * matrix[1][1] * matrix[2][2] +
            matrix[0][1] * matrix[1][2] * matrix[2][0] +
            matrix[0][2] * matrix[1][0] * matrix[2][1] -
            matrix[0][2] * matrix[1][1] * matrix[2][0] -
            matrix[0][1] * matrix[1][0] * matrix[2][2] -
            matrix[0][0] * matrix[1][2] * matrix[2][1]) % p

def matrix_adjugate(matrix):
    return [
        [matrix[1][1] * matrix[2][2] - matrix[1][2] * matrix[2][1], matrix[0][2] * matrix[2][1] - matrix[0][1] * matrix[2][2],
         matrix[0][1] * matrix[1][2] - matrix[0][2] * matrix[1][1]],
        [matrix[1][2] * matrix[2][0] - matrix[1][0] * matrix[2][2], matrix[0][0] * matrix[2][2] - matrix[0][2] * matrix[2][0],
         matrix[0][2] * matrix[1][0] - matrix[0][0] * matrix[1][2]],
        [matrix[1][0] * matrix[2][1] - matrix[1][1] * matrix[2][0], matrix[0][1] * matrix[2][0] - matrix[0][0] * matrix[2][1],
         matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]]
    ]

def mod_inv(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

def scalar_multiply(matrix, scalar, p):
    return [[(element * scalar) % p for element in row] for row in matrix]

#------------------------------------------ blakley ------------------------------------------
```

```python
def main():
    data = request.get_json()
    button_id = data.get('id')
    if button_id == "button1":
        shamir()
    elif button_id == "button2":
        blakley()
    else:
        mignotte()

@app.route('/blakley', methods=['GET', 'POST'])
def blakley():
    if request.method == 'POST':
        p = int(request.form['p'])
        secret = int(request.form['secret'])
        p = get_prime_input(p)

        A = [[4, 19, -1], [52, 27, -1], [30, 85, -1]]

        A_inv = inverse_matrix_mod_p(A, p)

        shares = [sum(a * secret % p for a in row) for row in A]
        reconstructed_secret = sum(a * b % p for a, b in zip(A_inv[0], shares)) % p

        return render_template('result_blakley.html', secret=secret, shares=shares, reconstructed_secret=reconstructed_secret)

    return render_template('blakley.html', error="")


def shamir():
    global shares
    secret = int(request.form.get('secret'))
    minimum = int(request.form.get('minimum'))
    total_shares = int(request.form['total_shares'])

    shares = make_random_shares(secret, minimum, total_shares)
    print('Secret:', secret)
    print('Shares:')
    for share in shares:
        print('  ', share)

@app.route('/')
def welcome():
    return render_template('homepage.html')

@app.route('/shamir.html')
def shamir():
    return render_template('shamir.html')
```

```python
@app.route('/mignotte.html')
def mignotte():
    return render_template('mignotte.html')

@app.route('/enter_share', methods =['GET', 'POST'])
def compute_mignotte():
    if request.method == 'POST':
        secret = int(request.form['secret'])
        bitsize = 16  # 60-bit primes

        m = [0] * 4  # Array with five values
        share = [0] * 4  # Array with five values

        m[0] = generate_prime(bitsize,m)
        m[1] = generate_prime(bitsize,m)
        m[2] = generate_prime(bitsize,m)
        m[3] = generate_prime(bitsize,m)

        m = sorted(m)

        while m[0] * m[1] * m[2] < m[2] * m[3] and m[3] > m[2]:
            m[3] = generate_prime(bitsize,m)

        rand = randint(m[2] * m[3], m[0] * m[1] * m[2])
        secret = rand + secret

        share[0] = secret % m[0]
        share[1] = secret % m[1]
        share[2] = secret % m[2]
        share[3] = secret % m[3]

        mod = [m[0], m[1], m[2]]
        rem1 = [share[0], share[1], share[2]]
        res = solve_crt(rem1, mod)
        three_secret = res - rand

        mod = [m[0], m[1]]
        rem2 = [share[0], share[1]]
        res = solve_crt(rem2, mod)
        two_secret = res - rand

        result = {
        "made_secret": secret,
        "m0": m[0],
        "m1": m[1],
        "m2": m[2],
        "m3": m[3],
        "share0": share[0],
        "share1": share[1],
        "share2": share[2],
        "share3": share[3],
        "three_secret" : three_secret,
        "two_secret" : two_secret
        }
        return render_template('result_mignotte.html', result=result)

    return render_template('mignotte.html', error="")
```

```python
@app.route('/generate_shares', methods=['POST'])
def generate_shares():
    secret = int(request.form['secret'])
    minimum = int(request.form['minimum'])
    total_shares = int(request.form['total_shares'])

    global shares
    shares = make_random_shares(secret, minimum, total_shares)
    session['minimum'] = minimum
    return render_template('shamir_shares.html', secret=secret, shares=shares,minimum=minimum)

@app.route('/recover_secret', methods=['GET', 'POST'])
def recover_secret_view():
    minimum = session.get('minimum', 0)
    if request.method == 'POST':

        input_shares = [
        (
            int(request.form[f"real_part_{i}"]),
            int(request.form[f"imaginary_part_{i}"])
        )
        for i in range(minimum)
    ]
        recovered_secret = recover_secret(input_shares)
        return render_template('shamir_recovered_secret.html', recovered_secret=recovered_secret)

if __name__ == '__main__':
    app.run(debug=True,port=0)
```

Shamir’s Secret Sharing Scheme:-

```python
import random

_PRIME = 2 ** 127 - 1

def _eval_at(poly, x, prime):
    """Evaluates polynomial (coefficient tuple) at x."""
    accum = 0
    for coeff in reversed(poly):
        accum *= x
        accum += coeff
        accum %= prime
    return accum

def make_random_shares(secret, minimum, shares, prime=_PRIME):
    if minimum > shares:
        raise ValueError("Pool secret would be irrecoverable.")
    poly = [secret] + [random.randint(0, prime - 1) for _ in range(minimum - 1)]
    points = [(i, _eval_at(poly, i, prime)) for i in range(1, shares + 1)]
    return points

def _extended_gcd(a, b):
    x = 0
    last_x = 1
    y = 1
    last_y = 0
    while b != 0:
        quot = a // b
        a, b = b, a % b
        x, last_x = last_x - quot * x, x
        y, last_y = last_y - quot * y, y
    return last_x, last_y

def _divmod(num, den, p):
    inv, _ = _extended_gcd(den, p)
    return num * inv

def _lagrange_interpolate(x, x_s, y_s, p):
    k = len(x_s)
    nums = [1] * k
    dens = [1] * k
    for i in range(k):
        for j in range(k):
            if i != j:
                nums[i] *= x - x_s[j]
                dens[i] *= x_s[i] - x_s[j]
    den = dens[0]
    num = sum([_divmod(nums[i] * den * y_s[i] % p, dens[i], p) for i in range(k)])
    return (_divmod(num, den, p) + p) % p
```

```python

def recover_secret(shares, prime=_PRIME):
    if len(shares) < 3:
        raise ValueError("need at least three shares")
    x_s, y_s = zip(*shares)
    return _lagrange_interpolate(0, x_s, y_s, prime)

def main():
    secret = int(input("Enter the secret: "))
    minimum = int(input("Enter the minimum number of shares required: "))
    total_shares = int(input("Enter the total number of shares to generate: "))

    shares = make_random_shares(secret, minimum, total_shares)

    print('Secret:                                                      ', secret)
    print('Shares:')
    if shares:
        for share in shares:
            print('  ', share)

    input_shares = []
    while len(input_shares) < minimum:
        share_input = input("Enter a share (format: x,y): ")
        try:
            x, y = map(int, share_input.split(','))
            input_shares.append((x, y))
        except ValueError:
            print("Invalid input. Please enter a valid share.")

    recovered_secret = recover_secret(input_shares)
    print('Secret recovered from input shares:                          ', recovered_secret)

if __name__ == '__main__':
    main()


```

Blakley’s Secret Sharing Scheme:-

```python
import random

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

def get_prime_input():
    while True:
        p = int(input("Enter a prime number (p): "))
        if is_prime(p):
            return p
        else:
            print("Invalid input. Please enter a prime number.")

p = get_prime_input()

# Generate a random secret
secret = int(input())

A = [[4, 19, -1], [52, 27, -1], [36, 65, -1]]
B = [-68, -10, -18]

print("A=", A)
print("B=", B)

def inverse_matrix_mod_p(matrix, p):
    det_inv = modinv(matrix_determinant(matrix, p), p)
    adj_matrix = matrix_adjugate(matrix)
    return scalar_multiply(adj_matrix, det_inv, p)

def matrix_determinant(matrix, p):
    return (matrix[0][0] * matrix[1][1] * matrix[2][2] +
            matrix[0][1] * matrix[1][2] * matrix[2][0] +
            matrix[0][2] * matrix[1][0] * matrix[2][1] -
            matrix[0][2] * matrix[1][1] * matrix[2][0] -
            matrix[0][1] * matrix[1][0] * matrix[2][2] -
            matrix[0][0] * matrix[1][2] * matrix[2][1]) % p

def matrix_adjugate(matrix):
    return [
        [matrix[1][1] * matrix[2][2] - matrix[1][2] * matrix[2][1], matrix[0][2] * matrix[2][1] - matrix[0][1] * matrix[2][2],
         matrix[0][1] * matrix[1][2] - matrix[0][2] * matrix[1][1]],
        [matrix[1][2] * matrix[2][0] - matrix[1][0] * matrix[2][2], matrix[0][0] * matrix[2][2] - matrix[0][2] * matrix[2][0],
         matrix[0][2] * matrix[1][0] - matrix[0][0] * matrix[1][2]],
        [matrix[1][0] * matrix[2][1] - matrix[1][1] * matrix[2][0], matrix[0][1] * matrix[2][0] - matrix[0][0] * matrix[2][1],
         matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]]
    ]
```

```python

def modinv(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

def scalar_multiply(matrix, scalar, p):
    return [[(element * scalar) % p for element in row] for row in matrix]

A_inv = inverse_matrix_mod_p(A, p)
print("Inverse A (mod p):", A_inv)

# Generate shares for the secret
shares = [sum(a * secret % p for a in row) for row in A]
print("Shares:", shares)

# Reconstruct the secret
reconstructed_secret = sum(a * b % p for a, b in zip(A_inv[0], shares)) % p
print("Reconstructed Secret:", reconstructed_secret)

```

Mignotte’s Secret Sharing Scheme:-

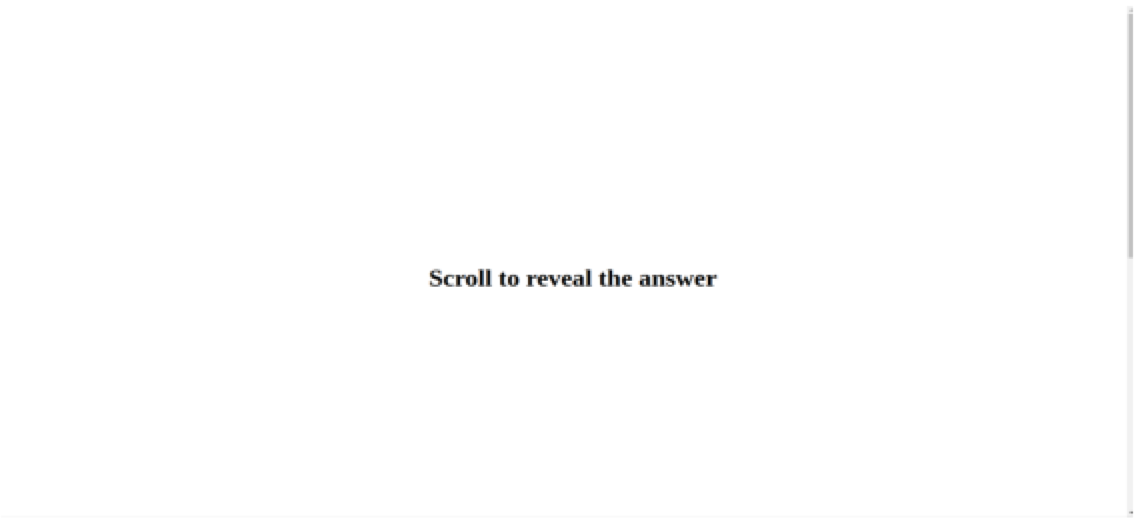
```python
from random import randint

def generate_prime(bitsize,m):
    while True:
        num = randint(2**(bitsize-1), 2**bitsize - 1)
        if isprime(num) and num not in m:
            return num

def isprime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def solve_crt(rem, mod):
    M = 1
    for m in mod:
        M *= m

    result = 0
    for i in range(len(rem)):
        Mi = M // mod[i]
        if Mi == 0:
            continue
        Mi_inv = modinv(Mi, mod[i])
        result += rem[i] * Mi * Mi_inv

    return result % M

def modinv(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

bitsize = 16  # 60-bit primes

m = [0] * 4  # Array with five values
share = [0] * 4  # Array with five values

```

```python
m = [0] * 4  # Array with five values
share = [0] * 4  # Array with five values

m[0] = generate_prime(bitsize,m)
m[1] = generate_prime(bitsize,m)
m[2] = generate_prime(bitsize,m)
m[3] = generate_prime(bitsize,m)

m = sorted(m)

while m[0] * m[1] * m[2] < m[2] * m[3] and m[3] == m[2]:
    m[3] = generate_prime(bitsize,m)

secret = int(input())

rand = randint(m[2] * m[3], m[0] * m[1] * m[2])
secret = rand + secret

share[0] = secret % m[0]
share[1] = secret % m[1]
share[2] = secret % m[2]
share[3] = secret % m[3]
'''
print("Secret: ", secret)
print("\nPrime0: ", m[0])
print("Prime1: ", m[1])
print("Prime2: ", m[2])
print("Prime3: ", m[3])

print("\nShare 1 (s1,m1): ", share[0], m[0])
print("Share 2 (s2,m2): ", share[1], m[1])
print("Share 3 (s3,m3): ", share[2], m[2])
print("Share 4 (s4,m4): ", share[3], m[3])

print("\nNow using the first three shares and solve CRT")
'''
mod = [m[0], m[1], m[2]]
rem = [share[0], share[1], share[2]]
res = solve_crt(rem, mod)
three_secret = res - rand

#print("Secret: ", res - rand)

#print("\nNow using the first two shares and solve CRT")
mod = [m[0], m[1]]
rem = [share[0], share[1]]
res = solve_crt(rem, mod)
two_secret = res - rand
#print("Secret: ", res - rand)
```

Output:-

Homepage

Shamir's Secret Sharing Scheme

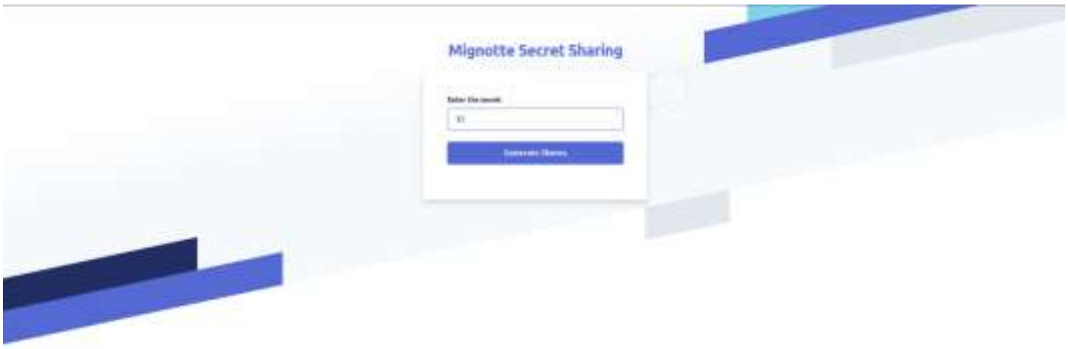


Recovered Secret: 32

Blakley's Secret Sharing Scheme

Blakley's Secret Sharing Result
Scroll to reveal the answer

Mignotte’s Secret Sharing Scheme



Errors:-

- In Blakley's and Mignotte's secret sharing scheme, no separate pages were given for reconstructing the secret. The secret is reconstructed automatically. This is because we did not have time to make separate pages for the UI but the code reconstructs the secret correctly!
- The code for Mignotte's secret sharing scheme could generate primes till 60 bits but it will take nearly 2.5 minutes for the primes to generate. So, we limited this to only 16 bits.
- We have assumed a predefined 3x3 matrix for Mignotte's secret sharing scheme which instead should be given as the input by the user. This is because the matrix should be an invertible one. Each time the user would not be able to input a 3x3 matrix which would be invertible.

User Manual:-

1.The Homepage contains the option to select which secret sharing method you want to use. Users can select can select any of the 3 methods:

- Shamir
- Blakley
- Mignotte

Clicking any one of these redirects into the corresponding methods home page.

2.Shamir

i. If you click Shamir, it redirects to the Shamir's Secret Sharing Scheme's homepage. User has to enter the secret value, threshold no.of shares to be generated and the maximum no.of shares to be generated. Click "Generate Shares" to generate the shares.

ii. In the generated shares page, users can also give the threshold no.of shares as the input to find the secret value again.

3.Blakley

In Blakley, Users have to enter the their prime number and their secret. It will automatically generate the shares.

4.Mignotte

In Mignotte, Users have to enter their secret. It will automatically generate these things:

- Refurbished secret
- 4 Primes
- 4 Shares
- Secret reconstructed if only 2 shares are used using CRT (Wrong)
- Secret reconstructed if 3 shares are used using CRT

Conclusion:-

In conclusion, Shamir, Blakley, and Mignotte's secret sharing scheme has proven to be a robust and efficient method for safeguarding sensitive information. By dividing a secret into multiple shares distributed among participants, this scheme ensures that no single entity can reconstruct the original secret without the collaboration of a predefined threshold of participants.

The mathematical foundations laid by Shamir, Blakley, and Mignotte provide a solid framework for implementing secure and flexible secret sharing protocols in various applications, ranging from cryptographic protocols to secure data storage and transmission. As technology continues to advance, the principles underlying this secret sharing scheme remain relevant, offering a versatile and reliable approach to protecting confidential information in a distributed and collaborative environment.