Program 3
CS 401 Cryptography, Spring 2024

Report

Question: Implement and instrument an Elliptic curve points implementation and explain clearly the way the system works.

Method: Using elliptic curve equation y^2 = x^3 + ax + b (mod p). Then, initialize the curve and define the base points in 5 dimensions on the curve. Then, find points on the curve generated by the base point

Usage: Run the code on google colab or any other environment using elliptic curve equation y^2 = x^3 + ax + b (mod p).

Original base points:
base_point = (21, 10, 1, 3, 7)

Analysis:

1. Encryption and Decryption:
- The encrypt_message function encrypts a message using an elliptic curve. It converts each character of the message to its ASCII code, encrypts each code, and returns the encrypted codes along with an ephemeral point used in the encryption process.
- The decrypt_message function decrypts an encrypted message. It subtracts the shared secret from each encrypted code to retrieve the original ASCII codes and then converts them back to characters to reconstruct the message.
2. Finding Points on the Curve:
- The find_points function finds points on the curve generated by repeatedly applying point doubling and addition starting from a base point. It detects cycles in the generated points and returns the points along with the maximum order found.

```python
import random

class EllipticCurve:
    def __init__(self, a, b, p):
        self.a = a  # Coefficient 'a' of the elliptic curve equation y^2 =
x^3 + ax + b (mod p)
        self.b = b  # Coefficient 'b' of the elliptic curve equation y^2 =
x^3 + ax + b (mod p)
        self.p = p  # Prime modulus defining the finite field over which
the curve operates

    def add(self, p_point, q_point):
        # Addition operation on elliptic curve points
        # Given two points 'p_point' and 'q_point', computes their sum
        x1, y1, z1, u1, v1 = p_point  # Extracting coordinates of point
'p_point'
        x2, y2, z2, u2, v2 = q_point  # Extracting coordinates of point
'q_point'

        # Handle the special cases of infinity points
        if (x1, y1) == (0, 1) and z1 == 0:
            return q_point
        if (x2, y2) == (0, 1) and z2 == 0:
            return p_point

        # Intermediate computations for the addition formula
        z1z1 = (z1 ** 2) % self.p
        z2z2 = (z2 ** 2) % self.p
        u1z1z1 = (u1 * z1z1) % self.p
        u2z2z2 = (u2 * z2z2) % self.p
        s1 = (y1 * z2 * z2z2) % self.p
        s2 = (y2 * z1 * z1z1) % self.p

        # Check if the points are equal or if they have opposite y
coordinates
        if u1 == u2 and s1 != s2:
            return (0, 1, 0, 1, 0)  # Result is the point at infinity
        if u1 == u2 and s1 == s2:
            return self.double(p_point)  # If points are equal, perform
doubling operation
```

```python
        # Intermediate computations for the addition formula
        h = (u2 - u1) % self.p
        r = (s2 - s1) % self.p
        hh = (h * h) % self.p
        hhh = (h * hh) % self.p
        u1hh = (u1 * hh) % self.p

        # Compute the x, y, z, u, v coordinates of the resulting point
        x3 = (r ** 2 - hhh - 2 * u1hh) % self.p
        y3 = (r * (u1hh - x3) - s1 * hhh) % self.p
        z3 = (h * z1 * z2) % self.p
        u3 = (x3 * y3) % self.p
        v3 = (y3 * y3) % self.p

        return (x3, y3, z3, u3, v3)

    def double(self, p_point):
        # Doubling operation on an elliptic curve point
        # Given a point 'p_point', computes its double
        x1, y1, z1, u1, v1 = p_point  # Extracting coordinates of point
'p_point'

        # Handle the special case of the point at infinity
        if y1 == 0 or z1 == 0:
            return (0, 1, 0, 1, 0)  # Result is the point at infinity

        # Intermediate computations for the doubling formula
        p = self.p
        a = self.a
        y1_squared = (y1 ** 2) % p
        four_x1_y1_squared = (4 * x1 * y1_squared) % p
        eight_y1_fourth = (8 * (y1_squared ** 2)) % p
        m = (3 * (x1 ** 2) + a * (z1 ** 4)) % p

        # Compute the x, y, z, u, v coordinates of the resulting point
        x3 = (m ** 2 - 2 * four_x1_y1_squared) % p
        y3 = (m * (four_x1_y1_squared - x3) - eight_y1_fourth) % p
        z3 = (2 * y1 * z1) % p
        u3 = (x3 * y3) % p
```

```python
        v3 = (y3 * y3) % p
        return (x3, y3, z3, u3, v3)

    def multiply(self, p_point, scalar):
        # Scalar multiplication operation on an elliptic curve point
        # Given a point 'p_point' and a scalar 'scalar', computes the scalar multiple
        q_point = (0, 1, 0, 1, 0)  # Initialize the result as the point at infinity
        while scalar > 0:
            if scalar % 2 == 1:
                q_point = self.add(q_point, p_point)  # Add 'p_point' to the result if the scalar bit is 1
            p_point = self.double(p_point)  # Double 'p_point' for the next iteration
            scalar //= 2  # Shift the scalar to the right
        return q_point

    def normalize(self, p_point):
        # Normalize the coordinates of an elliptic curve point
        # Given a point 'p_point', computes its normalized representation
        x, y, z, u, v = p_point
        if z == 0:
            return (0, 1, 0, 1, 0)  # If z-coordinate is zero, return the point at infinity
        p = self.p
        z_inv = pow(z, p - 2, p)  # Compute the modular inverse of z
        # Normalize the coordinates using the inverse of z
        x = (x * z_inv**2) % p
        y = (y * z_inv**3) % p
        u = (u * z_inv**4) % p
        v = (v * z_inv**6) % p
        return (x, y, 1, u, v)

    def is_quadratic_residue(self, a):
        # Check if a is a quadratic residue modulo p
        return pow(a, (self.p - 1) // 2, self.p) == 1

    def count_points(self):
        # Count the number of points on the elliptic curve
```

```
        count = 1
        max_order = 1
        for x in range(self.p):
            rhs = (x**3 + self.a*x + self.b) % self.p
            if self.is_quadratic_residue(rhs):
                count += 2 if rhs != 0 else 1
                max_order = max(max_order, count)
        return count, max_order

def encrypt_message(message, public_key, curve):
    # Encrypt a message using elliptic curve
    ascii_codes = [ord(char) for char in message]  # Convert characters to
ASCII codes
    encrypted_codes = []
    # Generate an ephemeral key pair and compute the shared secret
    ephemeral_point = curve.multiply(curve.base_point, random.randint(1,
curve.p - 1))
    shared_secret = curve.multiply(public_key, ephemeral_point[4])
    # Encrypt each ASCII code using the shared secret
    for code in ascii_codes:
        encrypted_code = (code + shared_secret[0]) % curve.p
        encrypted_codes.append(encrypted_code)
    return encrypted_codes, ephemeral_point

def decrypt_message(encrypted_codes, private_key, ephemeral_point, curve):
    # Decrypt a message using elliptic curve
    shared_secret = curve.multiply(ephemeral_point, private_key)  #
Compute the shared secret
    decrypted_codes = []
    # Decrypt each encrypted code using the shared secret
    for code in encrypted_codes:
        decrypted_code = (code - shared_secret[0]) % curve.p
        decrypted_codes.append(decrypted_code)
    # Convert decrypted ASCII codes back to characters
    decrypted_message = ''.join(chr(code) for code in decrypted_codes)
    return decrypted_message

def find_points(curve, base_point):
    # Find points on the elliptic curve generated by a base point
    p_point = base_point
```

```python
    points = [p_point]
    seen_points = set()
    seen_points.add(str(curve.normalize(p_point)))
    i = 2
    max_order = 1
    # Double the base point iteratively until reaching the point at infinity
    while True:
        p_point = curve.double(p_point)
        normalized_point = curve.normalize(p_point)
        point_str = str(normalized_point)
        if point_str in seen_points:
            print(f"Cycle detected at iteration {i}.")
            break
        points.append(p_point)
        seen_points.add(point_str)
        max_order = max(max_order, i)
        if p_point == (0, 1, 0, 1, 0):
            break
        i += 1
    return points, max_order

def main():
    p = 31
    a = 3
    b = 4
    curve = EllipticCurve(a, b, p)  # Initialize an elliptic curve
    base_point = (21, 10, 1, 3, 7)  # Define a base point on the curve
    points, max_order = find_points(curve, base_point)  # Find points on the curve generated by the base point
    print("Points on the curve:")
    # Print the normalized coordinates of each point on the curve
    for i, point in enumerate(points, start=0):
        normalized_point = curve.normalize(point)
        print(f"Point {i}: {normalized_point}")
    print(f"Maximum order found: {max_order}")

# Check if the script is executed directly
if __name__ == "__main__":
    main()
```

Result:

Original base points:

base_point = (21, 10, 1, 3, 7)

```
Cycle detected at iteration 7.
Points on the curve:
Point 0: (21, 10, 1, 3, 7)
Point 1: (14, 17, 1, 17, 10)
Point 2: (0, 13, 1, 0, 14)
Point 3: (14, 14, 1, 3, 10)
Point 4: (0, 18, 1, 0, 14)
Point 5: (14, 17, 1, 6, 10)
Maximum order found: 6
```