Lab 01. Points in an elliptic curve

16.03.2023

—

Ramírez Fuentes Edgar Alejandro
Selected Topics in Cryptography - 7CM2
Díaz Santiago Sandra

Introduction

Elliptic-curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC allows smaller keys compared to non-EC cryptography (based on plain Galois fields) to provide equivalent security.

Key size comparison

RSA	ECC
1024-4096 bits	160-256 bits

For current cryptographic purposes, an elliptic curve is a plane curve over a finite field (rather than the real numbers) which consists of the points satisfying the equation:

$$y^2 = x^3 + ax + b,$$

along with a distinguished point at infinity, denoted ∞.

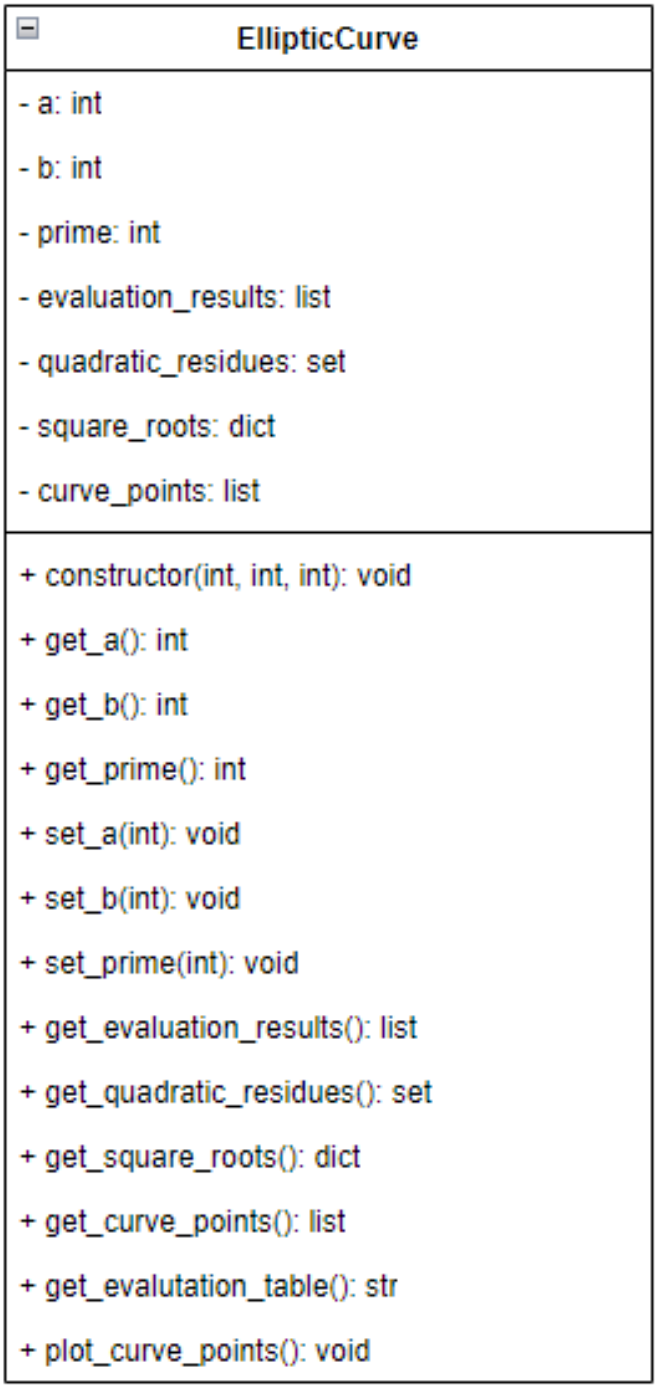
The use of elliptic curves in cryptography was suggested independently by Neal Koblitz and Victor S. Miller in 1985. Elliptic curve cryptography algorithms entered wide use in 2004 to 2005.

Early public-key systems based their security on the assumption that it is difficult to factor a large integer composed of two or more large prime factors. For later elliptic-curve-based protocols, the base assumption is that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible: this is the "elliptic curve discrete logarithm problem" (ECDLP). The security of elliptic curve cryptography depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original and product points. The size of the elliptic curve, measured by the total number of discrete integer pairs satisfying the curve equation, determines the difficulty of the problem.

The U.S. National Institute of Standards and Technology (NIST) has endorsed elliptic curve cryptography in its Suite B set of recommended algorithms, specifically elliptic-curve Diffie–Hellman (ECDH) for key exchange and Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signature.

Programming exercises

The programming exercises were implemented as class methods in the EllipticCurve class.



Class Code

```python
class EllipticCurve:
    """Class to represent an elliptic curve.

    Attributes:
        a (int): The a value of the equation.
        b (int): The b value of the equation.
        prime (int): The prime number.
        evaluation_results (list): Contains the evaluation results for the equation.
        quadratic_residues (set): Contains the quadratic residues for the prime.
        quadratic_roots (dict): Contains the quadratic roots for each quadratic residue.
        curve_points (list): Contains the points on the curve.
    """
```

```python
def __init__(self, a: int, b: int, prime: int) -> None:
        """Initialize the EllipticCurve class.

        Args:
            a (int): The a value of the equation.
            b (int): The b value of the equation.
            prime (int): The prime number.
        """
        self.__a = a
        self.__b = b
        self.__prime = prime
        self.__evaluation_results = []
        self.__quadratic_residues = set()
        self.__quadratic_roots = {}
        self.__curve_points = []

```

1. Design a function that receives a, b and p, i.e. the parameters given for an elliptic curve y^2 = x^3 + ax + b mod p, and stores the result of evaluating x^3 + ax + b mod p for every 0 ⩽ x ⩽ p − 1.

As mentioned above, the parameters a, b, and p are taken from the class attributes, which were set in the class constructor.

The task of this method is to return a list, which contains the result of evaluation x^3 + ax + b mod p for every 0 ≤ x ≤ p − 1. If the list has no values stored, the method will evaluate and store the results in the list, and after that It will return the list.

```python
def get_evaluation_results(self) -> list:
        """Calculate and store the evaluation results for the equation.

        Returns:
            list: Contains the evaluation results for the equation.
        """
        if not self.__evaluation_results:
            self.__evaluation_results = [(x**3 + self.__a * x + self.__b) % self.__prime for x in range(self.__prime)]

        return self.__evaluation_results
```

Create an EllipticCurve instance using the following values a = 2, b = 2, p = 17

```python
if __name__ == '__main__':
    curve = EllipticCurve(2, 2, 17)
    print(curve.get_evaluation_results())
```

The evaluation results are sorted from 0 to p-1.

```
PS C:\Users\Edgar\OneDrive\Desktop\Cryptography> python .\Lab1\EllipticCurve.py
[2, 5, 14, 1, 6, 1, 9, 2, 3, 1, 2, 12, 3, 15, 3, 7, 16]
PS C:\Users\Edgar\OneDrive\Desktop\Cryptography>
```

2. Design a function that receives as input a prime number p and find the quadratic residues modulo p. And also the square roots modulo p. For example if p = 11 and we know that 2^2 mod 11 = 4 and 9^2 mod 11 = 4 we know that 4 is a quadratic residue and its square roots are 2 and 9.

I divided this functionality into two methods:

The first method is responsible for returning a set of integers, which contains the quadratic residues modulo p (the attribute prime of the class). If the set of integers is empty, the method will calculate the quadratic residues, and it will return it.

```python
def get_quadratic_residues(self) -> set:
        """Calculate and store the quadratic residues for the prime.

        Returns:
            set: Contains the quadratic residues for the prime.
        """
        if not self.__quadratic_residues:
            self.__quadratic_residues = {(x**2) % self.__prime for x in range(1, self.__prime)}

        return self.__quadratic_residues
```

```python
if __name__ == '__main__':
    curve = EllipticCurve(1, 6, 11)
    print(curve.get_quadratic_residues())
```

```
PS C:\Users\Edgar\OneDrive\Desktop\Cryptography> python .\Lab1\EllipticCurve.py
{1, 3, 4, 5, 9}
PS C:\Users\Edgar\OneDrive\Desktop\Cryptography>
```

The second method is responsible for returning a dictionary, which contains the quadratic roots for each quadratic residue modulo p. If the dictionary is empty, the method will calculate the square roots for each quadratic residue.

```python
def get_square_roots(self) -> dict:
    """Calculate and store the quadratic roots for each quadratic residue.

    Returns:
        dict: Contains the quadratic roots for each quadratic residue.
    """
    if not self.__square_roots:
        self.__square_roots = {x : [] for x in self.get_quadratic_residues()}
        for x in range(1, self.__prime):
            self.__square_roots[(x**2) % self.__prime].append(x)

    return self.__square_roots
```

```python
if __name__ == '__main__':
    curve = EllipticCurve(1, 6, 11)
    print(curve.get_square_roots())

```

```
PS C:\Users\Edgar\OneDrive\Desktop\Cryptography> python .\Lab1\EllipticCurve.py
{1: [1, 10], 3: [5, 6], 4: [2, 9], 5: [4, 7], 9: [3, 8]}
PS C:\Users\Edgar\OneDrive\Desktop\Cryptography>
```

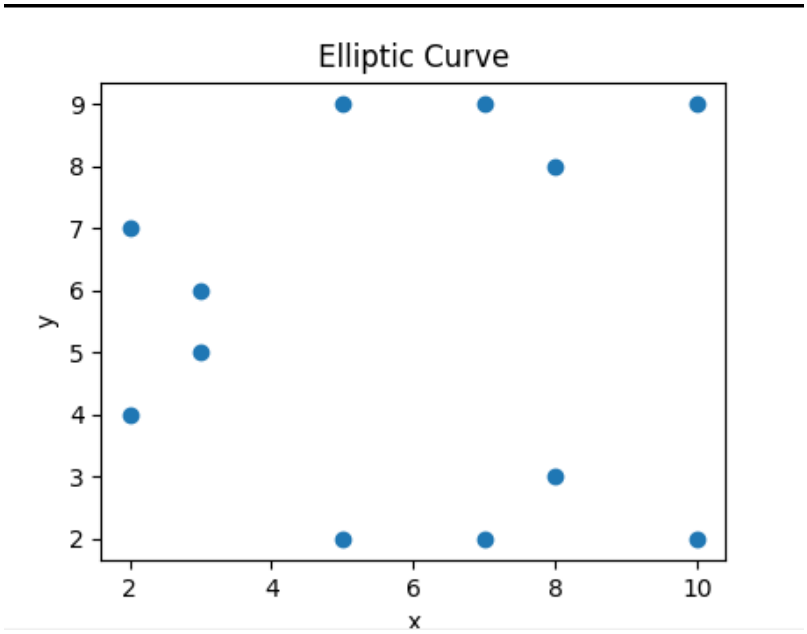
3. Using the previous functions implement a function that receives the parameters of an elliptic curve, i.e. a, b and p and as output finds the points in the curve y^2 = x^3 + ax + b mod p.

This method is responsible for returning a list, which contains sets of points in the elliptic curve. If the list is empty, it will get the points using the quadratic residues and square roots of those quadratic residues.

```python
def get_curve_points(self) -> list:
        """Calculate the points on the curve.

        Returns:
            list: Contains the points on the curve.
        """
        if not self.__evaluation_results:
            self.get_evaluation_results()

        if not self.__curve_points:
            for i in range(0, self.__prime):
                evaluated_value = self.__evaluation_results[i]

                if evaluated_value in self.get_square_roots():
                    for root in self.__square_roots[evaluated_value]:
                        self.__curve_points.append((i, root))

            # Add the point at infinity
            self.__curve_points.append((math.inf, math.inf))

        return self.__curve_points
```

```python
if __name__ == '__main__':
    curve = EllipticCurve(1, 6, 11)
    print(curve.get_curve_points())
    curve.plot_curve_points()
```

```
PS C:\Users\Edgar\OneDrive\Desktop\Cryptography> python .\Lab1\EllipticCurve.py
[(2, 4), (2, 7), (3, 5), (3, 6), (5, 2), (5, 9), (7, 2), (7, 9), (8, 3), (8, 8), (10, 2), (10, 9), (inf, inf)]
```



The class contains a method, which shows the information in a table.

```python
if __name__ == '__main__':
    curve = EllipticCurve(1, 6, 11)
    print(curve.get_evaluation_table())
```

```
PS C:\Users\Edgar\OneDrive\Desktop\Cryptography> python .\Lab1\EllipticCurve.py
a: 1, b: 6, prime: 11
|      x      |      f(x)     |      QR      |      Y      |
|      0      |      6        |      False   |      None   |
|      1      |      8        |      False   |      None   |
|      2      |      5        |      True    |      [4, 7] |
|      3      |      3        |      True    |      [5, 6] |
|      4      |      8        |      False   |      None   |
|      5      |      4        |      True    |      [2, 9] |
|      6      |      8        |      False   |      None   |
|      7      |      4        |      True    |      [2, 9] |
|      8      |      9        |      True    |      [3, 8] |
|      9      |      7        |      False   |      None   |
|      10     |      4        |      True    |      [2, 9] |

PS C:\Users\Edgar\OneDrive\Desktop\Cryptography>
```