Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчёт

по лабораторной работе №6

**Цифровая подпись**

|  |  |
| --- | --- |
| Выполнил:  студент группы 653502  Куликов А.Д. | Проверил:  Артемьев В.С. |

Минск 2019

# Постановка задачи

Создать программу для формирования и проверки ЭЦП на базе алгоритма ГОСТ 3410.

# Введение

## Эллиптические кривые

Эллиптическая кривая над полем K — неособая кубическая кривая на проективной плоскости над K^ (алгебраическим замыканием поля K), задаваемая уравнением 3-й степени с коэффициентами из поля K и «точкой на бесконечности». В подходящих аффинных координатах её уравнение приводится к виду:



В канонической форме если характеристика поля K не равна 2 или 3, то уравнение с помощью замены координат приводится к канонической форме (форме Вейерштрасса):



Если характеристика К = 3, то каноническим видом уравнения является вид:



Если характеристика K=2, то уравнение приводится к одному из видов:

 - суперсингулярные кривые

или

 - несуперсингулярные кривые

Особое удобство суперсингулярных эллиптических кривых в том, что для них легко вычислить порядок, в то время как вычисление порядка несуперсингулярных кривых вызывает трудности. Суперсингулярные кривые особенно удобны для создания самодельной ЕСС-криптосистемы. Для их использования можно обойтись без трудоёмкой процедуры вычисления порядка.

Точки эллиптической кривой над конечным полем представляют собой группу. И для этой группы определена операция сложения. Соответственно, можно представить умножение числа k на точку G как G+G+..+G с k слагаемыми.

Теперь можно представить, что имеется сообщение M представленное в виде целого числа. Мы можем зашифровать его используя выражение  
C=M\*G. Вопрос в том, насколько сложно восстановить M зная параметры кривой E(a,b), шифротекст С и точку G.

Данная задача называется дискретным логарифмом на эллиптической кривой и не имеет быстрого решения. Более того, считается, что задача дискретного логарифма на эллиптической кривой является более трудной для решения, чем задача дискретного логарифмирования в конечных полях.

## Алгоритм ГОСТ 34.10-2018

ГОСТ 34.10-2018 (полное название: «ГОСТ 34.10-2018. Информационная технология. Криптографическая защита информации. Процессы формирования и проверки электронной цифровой подписи», англ. «Information technology. Cryptographic data security. Signature and verification processes of electronic digital signature») — действующий межгосударственный криптографический стандарт, описывающий алгоритмы формирования и проверки электронной цифровой подписи реализуемой с использованием операций в группе точек эллиптической кривой, определенной над конечным простым полем.

Стандарт разработан на основе национального стандарта Российской Федерации ГОСТ Р 34.10-2012 и введен в действие с 1 июня 2019 года приказом Росстандарта № 1059-ст от 4 декабря 2018 года.

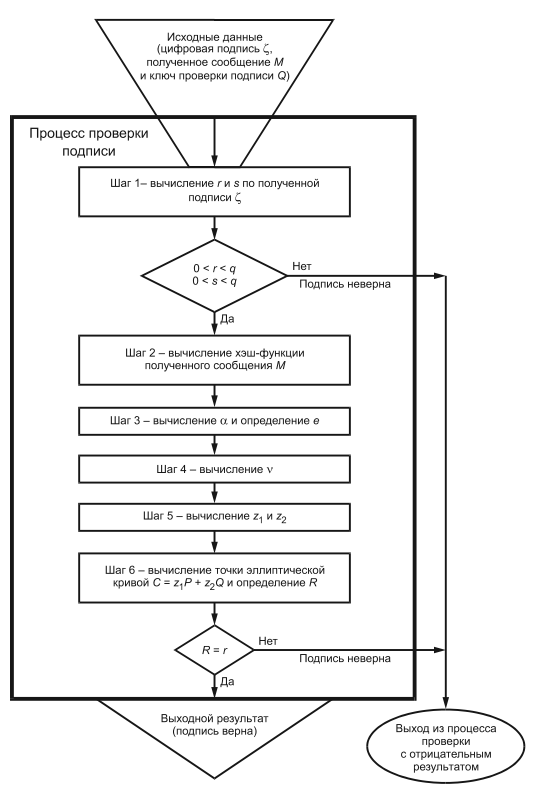
Цифровая подпись позволяет:

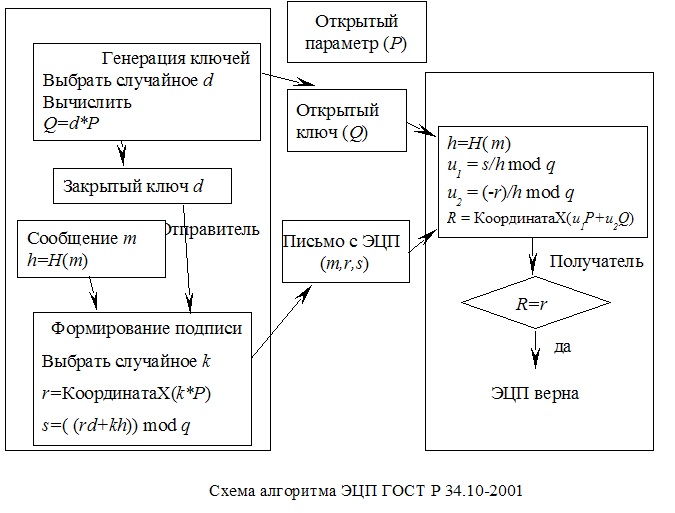
1. Аутентифицировать лицо, подписавшее сообщение;
2. Контролировать целостность сообщения;
3. Защищать сообщение от подделок;

## Безопасность алгоритма на основе эллиптических кривых

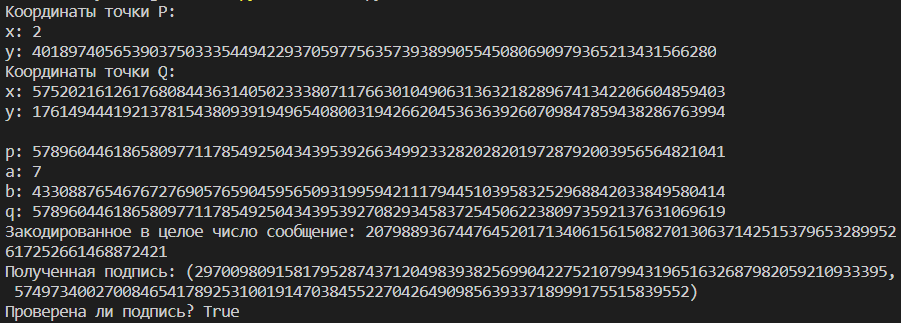
Безопасность, обеспечиваемая криптографическим подходом на основе эллиптических кривых, зависит от того, насколько трудной для решения оказывается задача определения {\displaystyle k}k по данным {\displaystyle kP}kP и {\displaystyle P}P. Эту задачу обычно называют проблемой логарифмирования на эллиптической кривой. Наиболее быстрым из известных на сегодня методов логарифмирования на эллиптической кривой является так называемый {\displaystyle p}p-метод Полларда. В таблице (см. ниже) сравнивается эффективность этого метода и метод разложения на простые множители с помощью решета в поле чисел общего вида. Из нее видно, что по сравнению с RSA в случае применения методов криптографии на эллиптических кривых примерно тот же уровень защиты достигается со значительно меньшими значениями длины ключей.

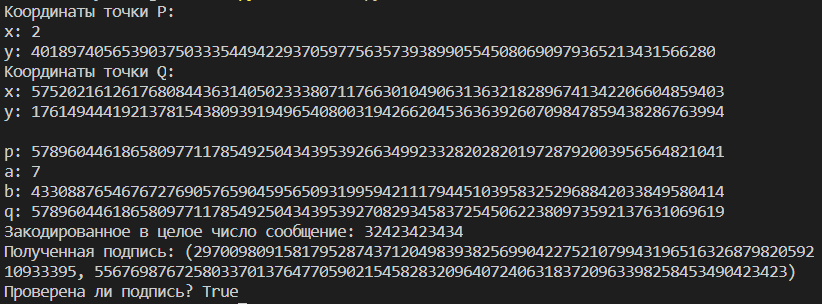
# Блок-схема алгоритма





# Результат исполнения программы





# Вывод

В ходе написания лабораторной работы были изучены алгоритмы цифровой подписи на основе эллиптических кривых, а также написаны их программные реализации. Результатом выполнения работы является программа, способная выполнять цифровую подпись, а также проверять ее подлинность.

# Приложение

**gost.py**

import random

from ec import ECPoint

class Gost:

    # p - int, EC module

    # a, b - int, EC coefficients

    # q - int, order of point P

    # p\_x, p\_y - int, point P coordinates

    def \_\_init\_\_(self, p, a, b, q, p\_x, p\_y):

        self.p\_point = ECPoint(p\_x, p\_y, a, b, p)

        self.q = q

        self.a = a

        self.b = b

        self.p = p

    # generate key pair

    def gen\_keys(self):

        d = random.randint(1, self.q - 1)

        q\_point = d \* self.p\_point

        return d, q\_point

    # sign message

    # message - int

    # private\_key - int

    def sign(self, message, private\_key, k=0):

        e = message % self.q

        if e == 0:

            e = 1

        if k == 0:

            k = random.randint(1, self.q - 1)

        r, s = 0, 0

        while r == 0 or s == 0:

            c\_point = k \* self.p\_point

            r = c\_point.x % self.q

            s = (r \* private\_key + k \* e) % self.q

        return r, s

    # verify signed message

    # message - int

    # sign - tuple

    # public\_key - ECPoint

    def verify(self, message, sign, public\_key):

        e = message % self.q

        if e == 0:

            e = 1

        nu = ECPoint.\_mod\_inverse(e, self.q)

        z1 = (sign[1] \* nu) % self.q

        z2 = (-sign[0] \* nu) % self.q

        c\_point = z1 \* self.p\_point + z2 \* public\_key

        r = c\_point.x % self.q

        if r == sign[0]:

            return True

        return False

**ec.py**

class ECPoint:

    def \_\_init\_\_(self, x=0, y=0, a=0, b=0, p=0, is\_polynomial\_basis=False):

        self.x = x

        self.y = y

        self.a = a

        self.b = b

        self.p = p

        self.pol\_basis = is\_polynomial\_basis

    # inverse int b modulo p

    @staticmethod

    def \_mod\_inverse(b, p):

        x0, x1, y0, y1, n = 1, 0, 0, 1, p

        while n != 0:

            q, b, n = b // n, n, b % n

            x0, x1 = x1, x0 - q \* x1

            y0, y1 = y1, y0 - q \* y1

        return x0 % p

    # addition of two EC points

    def \_\_add\_\_(self, other):

        p\_result = ECPoint()

        p\_result.a = self.a

        p\_result.b = self.b

        p\_result.p = self.p

        p\_result.pol\_basis = self.pol\_basis

        if self.pol\_basis:

            if self.x == other.x and self.y == other.y:

                x\_sqr = self.mult\_field(self.x, self.x, self.p)

                p\_result.x = self.sum\_field(x\_sqr,

                                            self.mult\_field(self.b, self.inv\_field(x\_sqr, self.p), self.p))

                tmp1 = self.mult\_field(self.y, self.inv\_field(self.x, self.p), self.p)

                tmp2 = self.mult\_field(self.sum\_field(self.x, tmp1), p\_result.x, self.p)

                p\_result.y = self.sum\_field(x\_sqr, tmp2, p\_result.x)

            else:

                if self.x == other.x:

                    return float('inf')

                l = self.mult\_field(self.sum\_field(self.y, other.y),

                                    self.inv\_field(self.sum\_field(self.x, other.x), self.p), self.p)

                p\_result.x = self.sum\_field(self.mult\_field(l, l, self.p), l, self.x, other.x, self.a)

                l2 = self.mult\_field(self.sum\_field(self.y, other.y),

                                     self.inv\_field(self.sum\_field(self.x, other.x), self.p), self.p)

                p\_result.y = self.sum\_field(self.mult\_field(l2, self.sum\_field(self.x, p\_result.x), self.p),

                                            p\_result.x, self.y)

        else:

            dx = (other.x - self.x) % self.p

            dy = (other.y - self.y) % self.p

            if self.x == other.x and self.y == other.y:

                l = ((3 \* self.x \*\* 2 + self.a) \* ECPoint.\_mod\_inverse(2 \* self.y, self.p)) % self.p

            else:

                if self.x == other.x:

                    return float('inf')

                dx\_inverse = ECPoint.\_mod\_inverse(dx, self.p)

                l = (dy \* dx\_inverse) % self.p

            p\_result.x = (l \* l - self.x - other.x) % self.p

            p\_result.y = (l \* (self.x - p\_result.x) - self.y) % self.p

        return p\_result

    # multiplication EC point and integer

    def \_\_rmul\_\_(self, other):

        p\_result = ECPoint(self.x, self.y, self.a, self.b, self.p, self.pol\_basis)

        temp = ECPoint(self.x, self.y, self.a, self.b, self.p, self.pol\_basis)

        x = other - 1

        while x != 0:

            if x % 2 != 0:

                p\_result += temp

                x -= 1

            x //= 2

            temp = temp + temp

        return p\_result

    # Multiplication in polynomial basis

    def mult\_field(self, x, y, n):

        mask = 1 << (n.bit\_length() - 2)

        p = 0

        while x:

            if x & 1:

                p ^= y

            if y & mask:

                y = (y << 1) ^ n

            else:

                y <<= 1

            x >>= 1

        return p

    # Addition in polynomial basis

    def sum\_field(self, \*x):

        res = 0

        for el in x:

            res ^= el

        return res

    # Inverse polynomial a modulo polynomial f

    def inv\_field(self, a, f):

        u, v = a, f

        g1, g2, = 1, 0

        while u != 1:

            j = u.bit\_length() - v.bit\_length()

            if j < 0:

                u, v = v, u

                g1, g2 = g2, g1

                j = - j

            u = self.sum\_field(u, (v << j))

            g1 = self.sum\_field(g1, (g2 << j))

        return g1