1. Министерство науки и высшего образования Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Институт Кибербезопасности и Защиты Информации**

**ЛАБОРАТОРНАЯ РАБОТА № 6**

1. **«****Протокол электронной цифровой подписи   
   ГОСТ Р 34.10 2018»**
2. по дисциплине «Криптографические методы защиты информации»
3. Выполнил
4. студент гр. 4851003/80802 Сошнев М.Д.

<*подпись*>

1. Проверил
2. преподаватель Ярмак А.В.

<*подпись*>

1. Санкт-Петербург
2. 2022

# Цель работы

Изучение протокола электронной цифровой подписи ГОСТ Р 34.10 2018, безопасность которого основана на задаче дискретного логарифмирования в группе точек эллиптической кривой.

# Задача

Получить у преподавателя вариант задания и разработать программу **П-1**, реализующую протокол подписи согласно ГОСТ Р 34.10 2018. Программа должна поддерживать функции формирования и проверки подписи, а также допускать возможность использования различных ключей.

# Ход работы

В результате выполнения работы была разработанна программа, способная формировать электронную подпись произвольного файла и проверять сформированную подпись согласно алгоритму ГОСТ Р 34.10 2018. При этом использовались следующие параметры:

p=57896044625414088412406986721186632159605151965036429316594800028484330862739

q=28948022312707044206203493360593316079803694388568974763893400879284219004579

a=-1

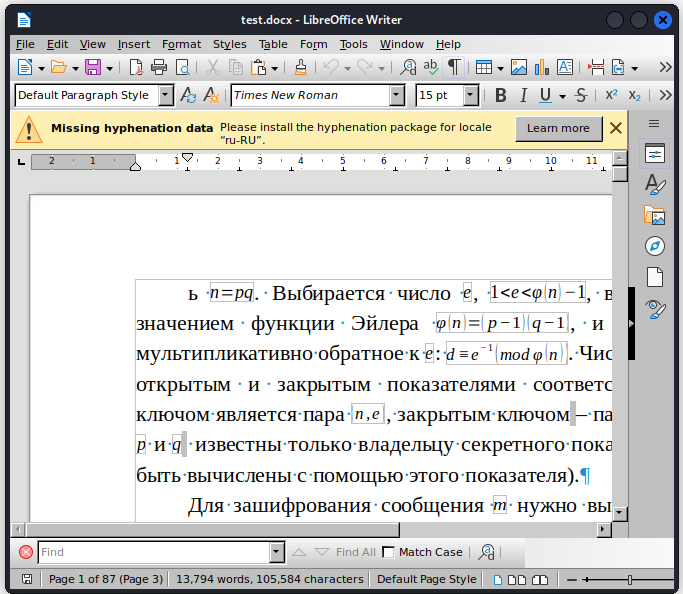
b=53520245325288251180656443226770638951803337703360722011463033447827147086694

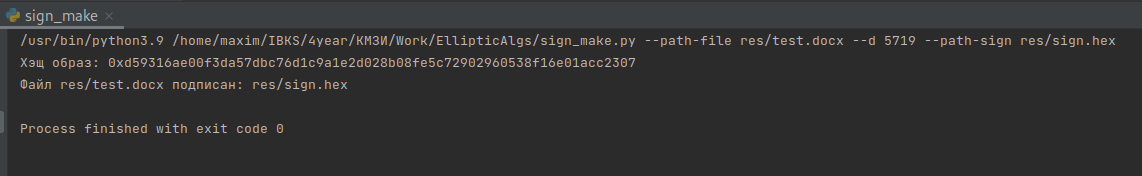
xP=36066034950041118412594006918367965339490267219250288222432003968962962331642

yP=54906983586985298119491343295734802658016371303757622466870297979342757624191

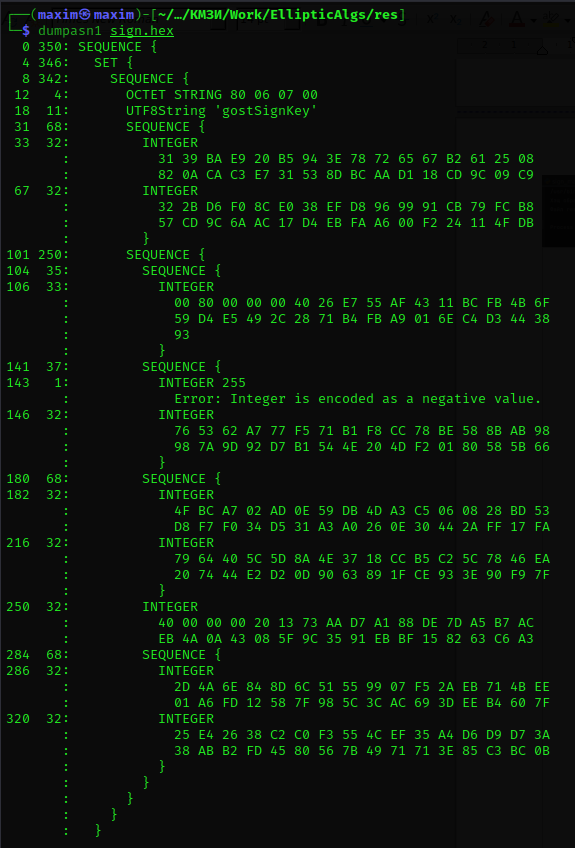
В качестве алгоритма хеш-функции использовался SHA-256. Продемонстриурем работу программы.

Формирование ЭЦП:

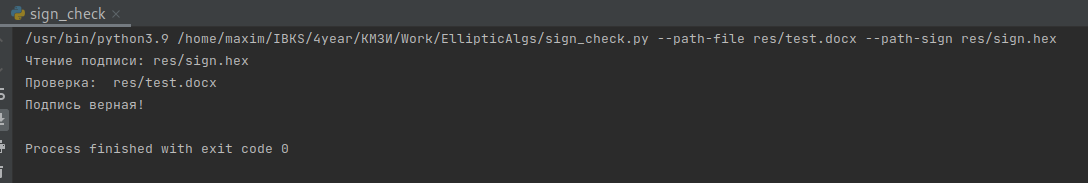
Рисунок 1 — тестовый файл

Рисунок 2 — формирование цифровой подписи файла

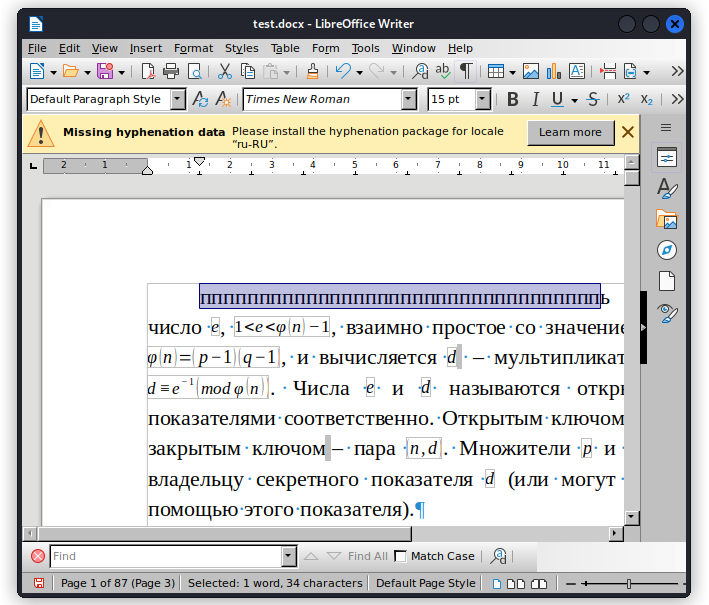
В качестве закрытого ключа использовался параметр d=5719

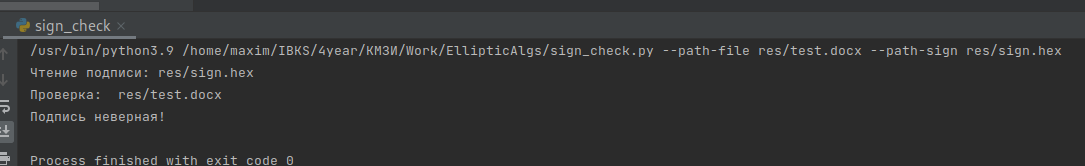
Рисунок 3 — файл цифровой подписи

Проверка цифровой подписи также проходит успешно:

Рисунок 4 — успешная проверка цифровой подписи

Теперь покажем, что при модификации файла проверка цифровой подписи не пройдет:

Рисунок 5 — модификация файла

Рисунок 6 — безуспешная проверка цифровой подписи после модификации файла

# Контрольные вопросы

1. Перечислите преимущества криптосистем на эллиптических кривых по сравнению с другими криптосистемами.

Задача дискретного логарифмирования на эллиптической кривой имеет экспоненциальную стойкость, которая практически не снижается во времени. Малое количество требований к параметрам криптосистемы, при этом которые можно легко проверить. Нет ограничений на ключи алгоритма

1. Почему в стандарте ГОСТ Р 34.102018 введено требование ?

Потому что в случае несоблюдения данного требования получится аномальная кривая (такая кривая, количество точек которой равно размеру исходного поля), которая подвержена атаке Смарта. Задача дискретного логарифмирования будет сильно упрощена.

1. Если нарушитель имеет возможность обращать хэш-функцию, как он может подделать сообщение и подпись?

Имея возможность обращения хэш-функции, нарушитель получает и возможность нахождения коллизии, так как данная задача имеет меньшую асимптотическую сложность, нежели задача обращения хэш-функции. Построив коллизию нарушитель может подделать подпись сообщения.

1. Почему случайный показатель не должен повторяться в течение времени жизни ключа?

*,* при этом, если совпадают *k*, то совпадают и *r*, так как вычисляются через *k*. Тогда, имея повторившиеся *k*, можем решить следующую систему сравнений относительно неизвестных *k* и *d,* таким образом получив закрытый ключ *d*:

# Вывод

В резальтате выполнения лабораторной работы был получен опыт в работе с эллиптическими кривыми. Был разработан алгоритм ГОСТ Р 34.10 2018 осуществляющий формирование цифровой подписи на закрытом ключе и её проверку.

# Приложение

import random  
import math  
import rsa  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
def bits(n):  
 *"""*  
 *Генерирует двоичные разряды n, начиная*  
 *с наименее значимого бита.*  
  
 *bits(151) -> 1, 1, 1, 0, 1, 0, 0, 1*  
 *"""*  
while n:  
 yield n & 1  
 n >>= 1  
  
  
class EllipticCurvePoint:  
 def \_\_init\_\_(self, x, y, curve):  
 self.x = x  
 self.y = y  
 self.\_\_curve = curve  
  
 def \_\_str\_\_(self):  
 return '({} ; {})'.format(self.x, self.y)  
  
 def clone(self):  
 return EllipticCurvePoint(self.x, self.y, self.\_\_curve)  
  
 def \_\_add\_\_(self, other):  
 if self.x == other.x:  
 m = (3 \* self.x \*\* 2 + self.\_\_curve.a) \* rsa.common.inverse(2 \* self.y, self.\_\_curve.n)  
 m %= self.\_\_curve.n  
 else:  
 dx = (self.x - other.x) % self.\_\_curve.n  
 dy = (self.y - other.y) % self.\_\_curve.n  
 m = dy \* rsa.common.inverse(dx, self.\_\_curve.n)  
 m %= self.\_\_curve.n  
  
 x = (m \*\* 2 - self.x - other.x) % self.\_\_curve.n  
 y = (self.y + m \* (x - self.x)) % self.\_\_curve.n  
 y = (-y) % self.\_\_curve.n  
  
 return EllipticCurvePoint(x, y, self.\_\_curve).check\_curve\_exist()  
  
 def \_\_mul\_\_(self, n: int):  
 *"""*  
 *Возвращает результат n \* self, вычисленный*  
 *алгоритмом удвоения-сложения.*  
 *"""*  
Q, P = None, self.clone()  
  
 for bit in bits(n):  
 if bit == 1:  
 if Q is None:  
 Q = P  
 else:  
 Q += P  
 P = P + P  
  
 return Q.check\_curve\_exist()  
  
 def check\_curve\_exist(self):  
 # if pow(self.y, 2, self.\_\_curve.n) != \  
 # (pow(self.x, 3, self.\_\_curve.n) + self.\_\_curve.a \* self.x + self.\_\_curve.b) % self.\_\_curve.n:  
 # print('[WARNING] {} не лежит на кривой {}'.format(self, self.\_\_curve))  
 # pass  
 return self  
  
 def show(self):  
 self.\_\_curve.show(self)  
  
  
class EllipticCurve:  
 def \_\_init\_\_(self, a, b, n):  
 self.a = a  
 self.b = b  
 self.n = n  
  
 def \_\_str\_\_(self):  
 return 'y^2 = x^3 {} {}x {} {} (mod {})'.format(  
 '+' if self.a >= 0 else '-',  
 self.a,  
 '+' if self.b >= 0 else '-',  
 self.b,  
 self.n)  
  
 def \_\_call\_\_(self, x):  
 y\_2 = (x \*\* 3 + self.a \* x + self.b) % self.n  
 Y = []  
 for y in range(self.n):  
 if pow(y, 2, self.n) == y\_2:  
 Y.append(y)  
  
 return Y  
  
 @staticmethod  
 def generate\_curve\_and\_point(n):  
 while True:  
 a = random.randint(0, n - 1)  
 x = random.randint(0, n - 1)  
 y = random.randint(0, n - 1)  
 b = (y \*\* 2 - x \*\* 3 - a \* x) % n  
  
 if (4 \* a \*\* 3 + 27 \* b \*\* 2) % n != 0:  
 break  
  
 ec = EllipticCurve(a, b, n)  
 point = EllipticCurvePoint(x, y, ec)  
  
 return ec, point  
  
 def show(self, point=None):  
 x\_space = np.linspace(0, self.n, self.n+1)  
 x\_show, y\_show = [], []  
 for x in x\_space:  
 Y = self(x)  
 for y in Y:  
 x\_show.append(x)  
 y\_show.append(y)  
  
 plt.scatter(x\_show, y\_show)  
 if point is not None:  
 plt.scatter(point.x, point.y, c='black')  
  
 plt.show()

import os.path  
  
import asn1  
from Crypto.Hash import SHA256  
import argparse  
  
from elliptic\_cryptography import \*  
  
  
class SignParams: # Заполнится из файла  
 p = None  
 q = None  
 curve = None  
 P = None  
  
  
def parse\_sign\_file(path\_sign):  
 sign\_params = SignParams()  
  
 f = open(path\_sign, 'rb')  
 data = f.read(os.path.getsize(path\_sign))  
 f.close()  
  
 decoder = asn1.Decoder()  
 decoder.start(data)  
  
 decoder.peek()  
 decoder.enter()  
 decoder.peek()  
 decoder.enter()  
 decoder.peek()  
 decoder.enter()  
  
 decoder.read(asn1.Numbers.OctetString)  
 decoder.read(asn1.Numbers.UTF8String)  
  
 decoder.peek()  
 decoder.enter()  
 \_, x\_q = decoder.read(asn1.Numbers.Integer)  
 \_, y\_q = decoder.read(asn1.Numbers.Integer)  
 # print('Qx = {}'.format(x\_q))  
 # print('Qy = {}'.format(y\_q))  
 decoder.leave()  
  
 decoder.peek()  
 decoder.enter()  
  
 decoder.peek()  
 decoder.enter()  
 \_, sign\_params.p = decoder.read(asn1.Numbers.Integer)  
 # print('p = {}'.format(sign\_params.p))  
 decoder.leave()  
  
 decoder.peek()  
 decoder.enter()  
 \_, A = decoder.read(asn1.Numbers.Integer)  
 \_, B = decoder.read(asn1.Numbers.Integer)  
 # print('A = {}'.format(A))  
 # print('B = {}'.format(B))  
 decoder.leave()  
  
 decoder.peek()  
 decoder.enter()  
 \_, x\_p = decoder.read(asn1.Numbers.Integer)  
 \_, y\_p = decoder.read(asn1.Numbers.Integer)  
 # print('Px = {}'.format(x\_p))  
 # print('Py = {}'.format(y\_p))  
 decoder.leave()  
  
 \_, sign\_params.q = decoder.read(asn1.Numbers.Integer)  
 # print('q = {}'.format(sign\_params.q))  
  
 decoder.peek()  
 decoder.enter()  
 \_, r = decoder.read(asn1.Numbers.Integer)  
 \_, s = decoder.read(asn1.Numbers.Integer)  
  
 sign = (r << 256) | s  
 # print('r = {}'.format(r))  
 # print('s = {}'.format(s))  
  
 decoder.leave()  
 decoder.leave()  
 decoder.leave()  
 decoder.leave()  
 decoder.leave()  
  
 sign\_params.curve = EllipticCurve(A, B, sign\_params.p)  
 sign\_params.P = EllipticCurvePoint(x\_p, y\_p, sign\_params.curve)  
 Q = EllipticCurvePoint(x\_q, y\_q, sign\_params.curve)  
  
 return sign\_params, Q, sign  
  
  
def sign\_check(path\_file, sign, Q, sign\_params):  
 print('Проверка: {}'.format(path\_file))  
  
 r = sign & 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  
 s = sign >> 256  
  
 # print('s = {}'.format(hex(s)))  
 # print('r = {}'.format(hex(r)))  
  
 check = (0 < r < sign\_params.q)  
 check &= (0 < s < sign\_params.q)  
 if not check:  
 return False  
  
 f = open(path\_file, 'rb')  
 data = f.read()  
 f.close()  
  
 hash\_instance = SHA256.new(data)  
 h = hash\_instance.digest()  
 h = int.from\_bytes(h, byteorder='big')  
 # print('h = {}'.format(hex(h)))  
  
 e = h % sign\_params.q  
 if e == 0:  
 e = 1  
 # print('e = {}'.format(hex(e)))  
  
 v = rsa.common.inverse(e, sign\_params.q)  
 # print('v = {}'.format(hex(v)))  
  
 z1 = (s \* v) % sign\_params.q  
 z2 = (-r \* v) % sign\_params.q  
  
 # print('z1 = {}'.format(z1))  
 # print('z2 = {}'.format(z2))  
  
 C = sign\_params.P \* z1 + Q \* z2  
 # print('C = {}'.format(C))  
  
 R = C.x % sign\_params.q  
 # print('R = {}'.format(hex(R)))  
  
 return r == R  
  
  
def main(context):  
 print('Чтение подписи: {}'.format(context.path\_sign))  
 sign\_params, Q, sign = parse\_sign\_file(context.path\_sign)  
  
 check = sign\_check(context.path\_file, sign, Q, sign\_params)  
 if check:  
 print('Подпись верная!')  
 else:  
 print('Подпись неверная!')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--path-file', required=True)  
 parser.add\_argument('--path-sign', required=True)  
  
 main(parser.parse\_args())

import asn1  
  
from sign\_check import \*  
  
  
class SignParams:  
 p = 57896044625414088412406986721186632159605151965036429316594800028484330862739  
 q = 28948022312707044206203493360593316079803694388568974763893400879284219004579  
 curve = EllipticCurve(-1, 53520245325288251180656443226770638951803337703360722011463033447827147086694, p)  
 P = EllipticCurvePoint(36066034950041118412594006918367965339490267219250288222432003968962962331642,  
 54906983586985298119491343295734802658016371303757622466870297979342757624191, curve)  
  
  
def sign\_file(path, d, sign\_params):  
 # print('p = {}'.format(sign\_params.p))  
  
 f = open(path, 'rb')  
 data = f.read()  
 f.close()  
  
 # print('q = {}'.format(hex(sign\_params.q)))  
  
 hash\_instance = SHA256.new(data)  
 h = hash\_instance.digest()  
 h = int.from\_bytes(h, byteorder='big')  
 # print('h = {}'.format(hex(h)))  
  
 e = h % sign\_params.q  
 if e == 0:  
 e = 1  
 # print('e = {}'.format(hex(e)))  
  
 while True:  
 k = random.randint(1, sign\_params.q)  
 # print('k = {}'.format(hex(k)))  
  
 C = sign\_params.P \* k  
 r = C.x % sign\_params.q  
 if r == 0:  
 continue  
  
 s = (r \* d + k \* e) % sign\_params.q  
 if s == 0:  
 continue  
  
 break  
  
 # print('r = {}'.format(hex(r)))  
 # print('s = {}'.format(hex(s)))  
  
 return r << 256 | s  
  
  
def write\_sign\_file(path\_sign, sign, Q, sign\_params):  
 encoder = asn1.Encoder()  
 encoder.start()  
  
 encoder.enter(asn1.Numbers.Sequence)  
 encoder.enter(asn1.Numbers.Set)  
 encoder.enter(asn1.Numbers.Sequence)  
 encoder.write(b'\x80\x06\x07\x00', asn1.Numbers.OctetString) # протокол подписи гост  
 encoder.write(b'gostSignKey', asn1.Numbers.UTF8String)  
  
 encoder.enter(asn1.Numbers.Sequence)  
 encoder.write(Q.x, asn1.Numbers.Integer)  
 encoder.write(Q.y, asn1.Numbers.Integer)  
 encoder.leave()  
  
 encoder.enter(asn1.Numbers.Sequence)  
  
 encoder.enter(asn1.Numbers.Sequence)  
 encoder.write(sign\_params.p, asn1.Numbers.Integer)  
 encoder.leave()  
  
 encoder.enter(asn1.Numbers.Sequence)  
 encoder.write(sign\_params.curve.a, asn1.Numbers.Integer)  
 encoder.write(sign\_params.curve.b, asn1.Numbers.Integer)  
 encoder.leave()  
  
 encoder.enter(asn1.Numbers.Sequence)  
 encoder.write(sign\_params.P.x, asn1.Numbers.Integer)  
 encoder.write(sign\_params.P.y, asn1.Numbers.Integer)  
 encoder.leave()  
  
 encoder.write(sign\_params.q, asn1.Numbers.Integer)  
  
 r = sign & 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  
 s = sign >> 256  
  
 encoder.enter(asn1.Numbers.Sequence)  
 encoder.write(r, asn1.Numbers.Integer)  
 encoder.write(s, asn1.Numbers.Integer)  
 encoder.leave()  
  
 encoder.leave()  
 encoder.leave()  
 encoder.leave()  
 encoder.leave()  
  
 f = open(path\_sign, 'wb')  
 f.write(encoder.output())  
 f.close()  
  
  
def main(context):  
 d = int(context.d)  
 sign\_params = SignParams()  
  
 sign = sign\_file(context.path\_file, d, sign\_params)  
  
 Q = sign\_params.P \* d  
  
 write\_sign\_file(context.path\_sign, sign, Q, sign\_params)  
  
 print('Файл {} подписан: {}'.format(context.path\_file, context.path\_sign))  
 # assert sign\_check(context.path\_file, sign, Q, sign\_params)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--path-file', required=True)  
 parser.add\_argument('--path-sign', required=True)  
 parser.add\_argument('--d', required=True)  
  
 main(parser.parse\_args())