

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого

Институт Кибербезопасности и Защиты Информации

ЛАБОРАТОРНАЯ РАБОТА № 6

«Протокол электронной цифровой подписи ГОСТ Р 34.10 2018»

по дисциплине «Криптографические методы защиты информации»

Выполнил
студент гр. 4851003/80802

<подпись>

Сошнев М.Д.

Проверил
преподаватель

<подпись>

Ярмак А.В.

Санкт-Петербург

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 разработана 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. Продемонстрируем работу программы.

Формирование ЭЦП:

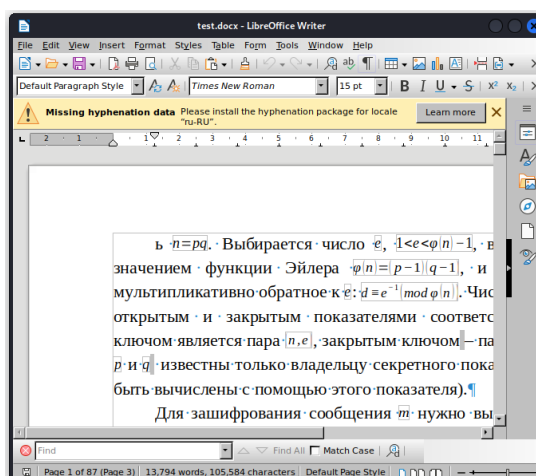


Рисунок 1 — тестовый файл

```
sign_make x
/usr/bin/python3.9 /home/maxim/IBKS/4year/KM3I/Work/EllipticAlgs/sign_make.py --path-file res/test.docx --d 5719 --path-sign res/sign.hex
Хэш образа: 0xd59316ae00f3da57dbc76d1c9a1e2d028b08fe5c72902960538f16e01acc2307
Файл res/test.docx подписан: res/sign.hex

Process finished with exit code 0
```

Рисунок 2 — формирование цифровой подписи файла

В качестве закрытого ключа использовался параметр d=5719

```
(maxim@maxim)-[~/KM3I/Work/EllipticAlgs/res]
$ dumpasn1 sign.hex
0 350: SEQUENCE {
4 346: SET {
8 342: SEQUENCE {
12 4: OCTET STRING 80 06 07 00
18 11: UTF8String 'gostSignKey'
31 68: SEQUENCE {
33 32: INTEGER
: 31 39 BA E9 20 B5 94 3E 78 72 65 67 B2 61 25 08
: 82 0A CA C3 E7 31 53 8D BC AA D1 18 CD 9C 09 C9
67 32: INTEGER
: 32 2B D6 F0 8C E0 38 EF D8 96 99 91 CB 79 FC B8
: 57 CD 9C 6A AC 17 D4 EB FA A6 00 F2 24 11 4F DB
: }
101 250: SEQUENCE {
104 35: SEQUENCE {
106 33: INTEGER
: 00 80 00 00 00 40 26 E7 55 AF 43 11 BC FB 4B 6F
: 59 D4 E5 49 2C 28 71 B4 FB A9 01 6E C4 D3 44 38
: 93
: }
141 37: SEQUENCE {
143 1: INTEGER 255
: Error: Integer is encoded as a negative value.
146 32: INTEGER
: 76 53 62 A7 77 F5 71 B1 F8 CC 78 BE 58 8B AB 98
: 98 7A 9D 92 D7 B1 54 4E 20 4D F2 01 80 58 5B 66
: }
180 68: SEQUENCE {
182 32: INTEGER
: 4F BC A7 02 AD 0E 59 DB 4D A3 C5 06 08 28 BD 53
: D8 F7 F0 34 D5 31 A3 A0 26 0E 30 44 2A FF 17 FA
216 32: INTEGER
: 79 64 40 5C 5D 8A 4E 37 18 CC B5 C2 5C 78 46 EA
: 20 74 44 E2 D2 0D 90 63 89 1F CE 93 3E 90 F9 7F
: }
250 32: INTEGER
: 40 00 00 00 20 13 73 AA D7 A1 88 DE 7D A5 B7 AC
: EB 4A 0A 43 08 5F 9C 35 91 EB BF 15 82 63 C6 A3
284 68: SEQUENCE {
286 32: INTEGER
: 2D 4A 6E 84 8D 6C 51 55 99 07 F5 2A EB 71 4B EE
: 01 A6 FD 12 58 7F 98 5C 3C AC 69 3D EE B4 60 7F
320 32: INTEGER
: 25 E4 26 38 C2 C0 F3 55 4C EF 35 A4 D6 D9 D7 3A
: 38 AB B2 FD 45 80 56 7B 49 71 71 3E 85 C3 BC 0B
: }
: }
: }
: }
```

Рисунок 3 — файл цифровой подписи

Проверка цифровой подписи также проходит успешно:

```
sign_check x
/usr/bin/python3.9 /home/maxim/IBKS/4year/КМЗИ/Work/EllipticAlgs/sign_check.py --path-file res/test.docx --path-sign res/sign.hex
Чтение подписи: res/sign.hex
Проверка: res/test.docx
Подпись верная!

Process finished with exit code 0
```

Рисунок 4 — успешная проверка цифровой подписи

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

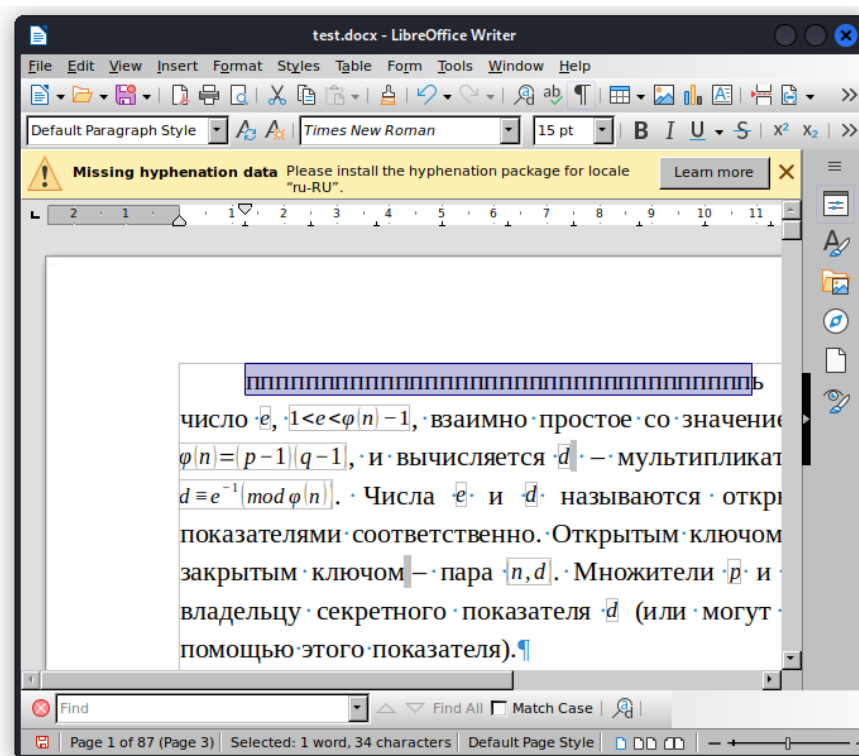


Рисунок 5 — модификация файла




```
sign_check x
/usr/bin/python3.9 /home/maxim/IBKS/4year/KM3I/Work/EllipticAlgs/sign_check.py --path-file res/test.docx --path-sign res/sign.hex
Чтение подписи: res/sign.hex
Проверка: res/test.docx
Подпись неверная!
Process finished with exit code 0
```

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

2. Почему в стандарте ГОСТ Р 34.10 /2018 введено требование $E(F_p) \neq p$?

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.

3. 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.

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

$s \equiv rd + ke$, 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 :

$$\begin{cases} s_1 \equiv rd + k e_1; \\ s_2 \equiv rd + k e_2. \end{cases}$$

Вывод

В результате 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 '{x} ; {y}'.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 & 0xffffffffffffffffffffffffffffffffffffffffffffffff
    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 =
5789604462541408841240698672118663215960515196503642931659480002848433086
2739
    q =
2894802231270704420620349336059331607980369438856897476389340087928421900
4579
    curve = EllipticCurve(-1,
5352024532528825118065644322677063895180333770336072201146303344782714708
6694, p)
    P =
EllipticCurvePoint(36066034950041118412594006918367965339490267219250288222432
003968962962331642,
5490698358698529811949134329573480265801637130375762246687029797934275762
4191, 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

```

[illegible]

```

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())

```