Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчет по лабораторной работе №7

“Криптография с использованием эллиптических кривых”

|  |  |
| --- | --- |
|  | Выполнил:  студент группы 053504  Горожанкин В.О. |
|  | Проверил  ассистент кафедры информатики  Лещенко Евгений Александрович |

Минск 2023

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc151931903)

[1 Демонстрация работы программы 4](#_Toc151931904)

[2 Теоретические сведения 5](#_Toc151931905)

[Заключение 6](#_Toc151931906)

[Приложение А (обязательно) листинг программного кода 7](#_Toc151931907)

# ВВЕДЕНИЕ

Современный мир цифровых коммуникаций и информационных технологий стал свидетелем возрастающей потребности в надежных методах шифрования для защиты конфиденциальности и целостности данных. Криптография, как наука о секретных кодированиях и методах их разгадывания, играет важную роль в обеспечении безопасности информации.

Одним из важных направлений в сфере криптографии является использование эллиптических кривых. Эллиптические кривые обладают уникальными математическими свойствами, которые позволяют создавать эффективные и надежные криптографические системы. Они находят широкое применение в современных криптографических протоколах, таких как ЭЦП (Электронная Цифровая Подпись), протоколы обмена ключами и шифрования данных.

Цель данной лабораторной работы заключается в реализации схемы шифрования и дешифрования, основанной на эллиптических кривых и аналогичной алгоритму Эль-Гамаля. Алгоритм Эль-Гамаля является одним из популярных асимметричных криптографических методов, и его адаптация для работы с эллиптическими кривыми позволяет повысить уровень безопасности передачи данных.

В рамках данной лабораторной работы будут изучены основные принципы работы алгоритма Эль-Гамаля на эллиптических кривых, а также реализованы соответствующие процедуры для шифрования и дешифрования данных. Такой аналог алгоритма Эль-Гамаля на основе эллиптических кривых позволит нам оценить эффективность и надежность данной криптографической системы.

# 1 ДЕМОНСТРАЦИЯ РАБОТЫ ПРОГРАММЫ

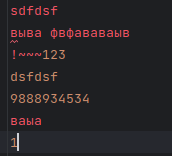


Рисунок 1 – Входные данные

Результат шифрования программы:

Point(X=114690805279596067248710427517217645281359064863700133589203811495228323104696, Y=78563092279832570919948283835983741069457531434156568448813423202480136536338, Curve=P256)

Point(X=82349876285170335880269068586725301071600765460480040487409086920174302316025, Y=1156869260313769287086562027057608067713918007065709748258782809854028701860, Curve=P256)

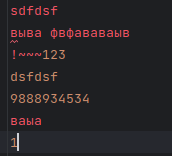
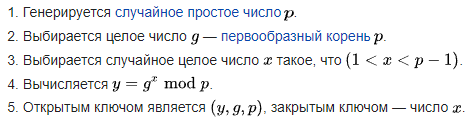
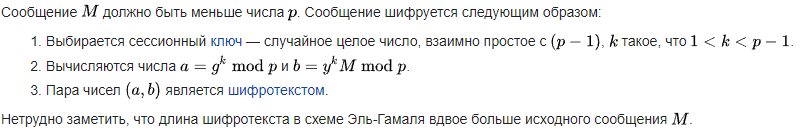


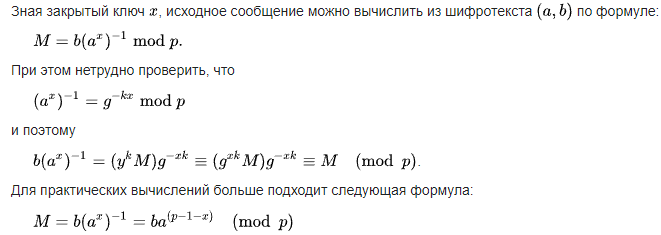
Рисунок 2 – Выходные данные

# 2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Схема Эль-Гамаля:







Алгоритм работы схемы Эль-Гамаля представлен на рисунке 2.

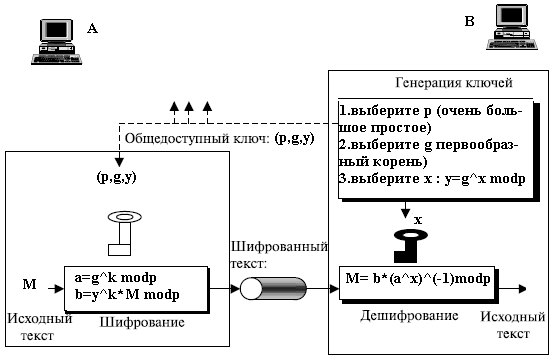


Рисунок 2 – алгоритм работы схемы Эль-Гамаля

# ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была реализована схема шифрования и дешифрования на основе эллиптических кривых, аналогичная алгоритму Эль-Гамаля. Работа с эллиптическими кривыми позволяет повысить уровень безопасности криптографических операций, а также улучшить эффективность передачи и защиту данных.

В заключение, лабораторная работа по реализации схемы шифрования на основе эллиптических кривых, подобной алгоритму Эль-Гамаля, позволила понять принципы работы этой криптографической системы и оценить ее эффективность. Эллиптические кривые продолжают оставаться актуальным и перспективным инструментом в области информационной безопасности, и их применение может быть ключевым для обеспечения конфиденциальности данных в современном цифровом мире.

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Листинг программного кода**

from os import urandom

from abc import ABC, abstractmethod

from dataclasses import dataclass

from typing import Optional

from utils import int\_length\_in\_byte, modsqrt, modinv

with open("input.txt", "r", encoding="utf-8") as file:

text = file.read()

@dataclass

class Point:

x: Optional[int]

y: Optional[int]

curve: "Curve"

def is\_at\_infinity(self) -> bool:

return self.x is None and self.y is None

def \_\_post\_init\_\_(self):

if not self.is\_at\_infinity() and not self.curve.is\_on\_curve(self):

raise ValueError("The point is not on the curve.")

def \_\_str\_\_(self):

if self.is\_at\_infinity():

return f"Point(At infinity, Curve={str(self.curve)})"

else:

return f"Point(X={self.x}, Y={self.y}, Curve={str(self.curve)})"

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other):

return self.curve == other.curve and self.x == other.x and self.y == other.y

def \_\_neg\_\_(self):

return self.curve.neg\_point(self)

def \_\_add\_\_(self, other):

return self.curve.add\_point(self, other)

def \_\_radd\_\_(self, other):

return self.\_\_add\_\_(other)

def \_\_sub\_\_(self, other):

negative = - other

return self.\_\_add\_\_(negative)

def \_\_mul\_\_(self, scalar: int):

return self.curve.mul\_point(scalar, self)

def \_\_rmul\_\_(self, scalar: int):

return self.\_\_mul\_\_(scalar)

@dataclass

class Curve(ABC):

name: str

a: int

b: int

p: int

n: int

G\_x: int

G\_y: int

def \_\_str\_\_(self):

return self.name

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def \_\_eq\_\_(self, other):

return (

self.a == other.a and self.b == other.b and self.p == other.p and

self.n == other.n and self.G\_x == other.G\_x and self.G\_y == other.G\_y

)

@property

def G(self) -> Point:

return Point(self.G\_x, self.G\_y, self)

@property

def INF(self) -> Point:

return Point(None, None, self)

def is\_on\_curve(self, P: Point) -> bool:

if P.curve != self:

return False

return P.is\_at\_infinity() or self.\_is\_on\_curve(P)

@abstractmethod

def \_is\_on\_curve(self, P: Point) -> bool:

pass

def add\_point(self, P: Point, Q: Point) -> Point:

if (not self.is\_on\_curve(P)) or (not self.is\_on\_curve(Q)):

raise ValueError("The points are not on the curve.")

if P.is\_at\_infinity():

return Q

elif Q.is\_at\_infinity():

return P

if P == -Q:

return self.INF

if P == Q:

return self.\_double\_point(P)

return self.\_add\_point(P, Q)

@abstractmethod

def \_add\_point(self, P: Point, Q: Point) -> Point:

pass

@abstractmethod

def \_double\_point(self, P: Point) -> Point:

pass

def mul\_point(self, d: int, P: Point) -> Point:

if not self.is\_on\_curve(P):

raise ValueError("The point is not on the curve.")

if P.is\_at\_infinity():

return self.INF

if d == 0:

return self.INF

res = self.INF

is\_negative\_scalar = d < 0

d = -d if is\_negative\_scalar else d

tmp = P

while d:

if d & 0x1 == 1:

res = self.add\_point(res, tmp)

tmp = self.add\_point(tmp, tmp)

d >>= 1

if is\_negative\_scalar:

return -res

else:

return res

def neg\_point(self, P: Point) -> Point:

if not self.is\_on\_curve(P):

raise ValueError("The point is not on the curve.")

if P.is\_at\_infinity():

return self.INF

return self.\_neg\_point(P)

@abstractmethod

def \_neg\_point(self, P: Point) -> Point:

pass

@abstractmethod

def compute\_y(self, x: int) -> int:

pass

def encode\_point(self, plaintext: bytes) -> Point:

plaintext = len(plaintext).to\_bytes(1, byteorder="big") + plaintext

while True:

x = int.from\_bytes(plaintext, "big")

y = self.compute\_y(x)

if y:

return Point(x, y, self)

plaintext += urandom(1)

def decode\_point(self, M: Point) -> bytes:

byte\_len = int\_length\_in\_byte(M.x)

byte\_len = len(text.encode('utf-8'))

plaintext\_len = (M.x >> ((byte\_len - 1) \* 8)) & 0xff

plaintext = ((M.x >> ((byte\_len - plaintext\_len - 1) \* 8))

& (int.from\_bytes(b"\xff" \* plaintext\_len, "big")))

return plaintext.to\_bytes(plaintext\_len, byteorder="big")

class ShortWeierstrassCurve(Curve):

"""

y^2 = x^3 + a\*x + b

"""

def \_is\_on\_curve(self, P: Point) -> bool:

left = P.y \* P.y

right = (P.x \* P.x \* P.x) + (self.a \* P.x) + self.b

return (left - right) % self.p == 0

def \_add\_point(self, P: Point, Q: Point) -> Point:

# s = (yP - yQ) / (xP - xQ)

# xR = s^2 - xP - xQ

# yR = yP + s \* (xR - xP)

delta\_x = P.x - Q.x

delta\_y = P.y - Q.y

s = delta\_y \* modinv(delta\_x, self.p)

res\_x = (s \* s - P.x - Q.x) % self.p

res\_y = (P.y + s \* (res\_x - P.x)) % self.p

return - Point(res\_x, res\_y, self)

def \_double\_point(self, P: Point) -> Point:

# s = (3 \* xP^2 + a) / (2 \* yP)

# xR = s^2 - 2 \* xP

# yR = yP + s \* (xR - xP)

s = (3 \* P.x \* P.x + self.a) \* modinv(2 \* P.y, self.p)

res\_x = (s \* s - 2 \* P.x) % self.p

res\_y = (P.y + s \* (res\_x - P.x)) % self.p

return - Point(res\_x, res\_y, self)

def \_neg\_point(self, P: Point) -> Point:

return Point(P.x, -P.y % self.p, self)

def compute\_y(self, x) -> int:

right = (x \* x \* x + self.a \* x + self.b) % self.p

y = modsqrt(right, self.p)

return y

P256 = ShortWeierstrassCurve(

name="P256",

a=-3,

b=41058363725152142129326129780047268409114441015993725554835256314039467401291,

p=0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff,

n=0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551,

G\_x=0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296,

G\_y=0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5

)

secp256k1 = ShortWeierstrassCurve(

name="secp256k1",

a=0,

b=7,

p=0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f,

n=0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141,

G\_x=0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,

G\_y=0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8

)

import random

from os import urandom

from typing import Callable, Tuple

from dataclasses import dataclass

from curve import Curve, Point

@dataclass

class ElGamal:

curve: Curve

def encrypt(self, plaintext: bytes, public\_key: Point,

randfunc: Callable = None) -> Tuple[Point, Point]:

return self.encrypt\_bytes(plaintext, public\_key, randfunc)

def decrypt(self, private\_key: int, C1: Point, C2: Point) -> bytes:

return self.decrypt\_bytes(private\_key, C1, C2)

def encrypt\_bytes(self, plaintext: bytes, public\_key: Point,

randfunc: Callable = None) -> Tuple[Point, Point]:

# Encode plaintext into a curve point

M = self.curve.encode\_point(plaintext)

return self.encrypt\_point(M, public\_key, randfunc)

def decrypt\_bytes(self, private\_key: int, C1: Point, C2: Point) -> bytes:

M = self.decrypt\_point(private\_key, C1, C2)

return self.curve.decode\_point(M)

def encrypt\_point(self, plaintext: Point, public\_key: Point,

randfunc: Callable = None) -> Tuple[Point, Point]:

randfunc = randfunc or urandom

# Base point G

G = self.curve.G

M = plaintext

random.seed(randfunc(1024))

k = random.randint(1, self.curve.n)

C1 = k \* G

C2 = M + k \* public\_key

return C1, C2

def decrypt\_point(self, private\_key: int, C1: Point, C2: Point) -> Point:

M = C2 + (self.curve.n - private\_key) \* C1

return M