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

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

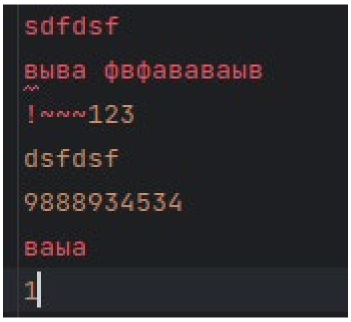
```
sdfdsf
выва фвфавваваыв
!~~~123
dsfdsf
9888934534
ваыа
1
```

Рисунок 1 – Входные данные

Результат шифрования программы:

```
    Point(X=114690805279596067248710427517217645281359064863700133589203811495228323104696,
Y=785630922798325709199482838359837410694575314341565684488134232024801365363338, Curve=P256)
    Point(X=82349876285170335880269068586725301071600765460480040487409086920174302316025,
Y=1156869260313769287086562027057608067713918007065709748258782809854028701860, Curve=P256)
```

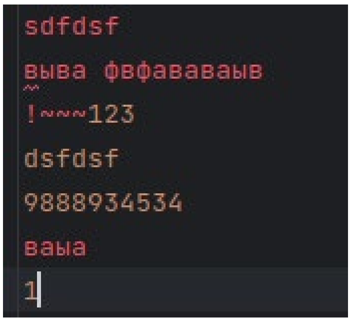
```
sdfdsf
выва фвфавваваыв
!~~~123
dsfdsf
9888934534
ваыа
1
```

Рисунок 2 – Выходные данные

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Схема Эль-Гамаля:

1. Генерируется случайное простое число p.
2. Выбирается целое число g — первообразный корень p.
3. Выбирается случайное целое число x такое, что $(1 < x < p - 1)$.
4. Вычисляется $y = g^x \bmod p$.
5. Открытым ключом является (y, g, p), закрытым ключом — число x.

Сообщение M должно быть меньше числа p. Сообщение шифруется следующим образом:

1. Выбирается сессионный ключ — случайное целое число, взаимно простое с $(p - 1)$, k такое, что $1 < k < p -$
2. Вычисляются числа $a = g^k \bmod p$ и $b = y^k M \bmod p$.
3. Пара чисел (a, b) является шифротекстом.

Нетрудно заметить, что длина шифротекста в схеме Эль-Гамаля вдвое больше исходного сообщения M.

Зная закрытый ключ x, исходное сообщение можно вычислить из шифротекста (a, b) по формуле:

$$M = b(a^x)^{-1} \bmod p.$$

При этом нетрудно проверить, что

$$(a^x)^{-1} = g^{-kx} \bmod p$$

и поэтому

$$b(a^x)^{-1} = (y^k M)g^{-xk} \equiv (g^{xk} M)g^{-xk} \equiv M \pmod p.$$

Для практических вычислений больше подходит следующая формула:

$$M = b(a^x)^{-1} = ba^{(p-1-x)} \pmod p$$

Алгоритм работы схемы Эль-Гамаля представлен на рисунке 2.

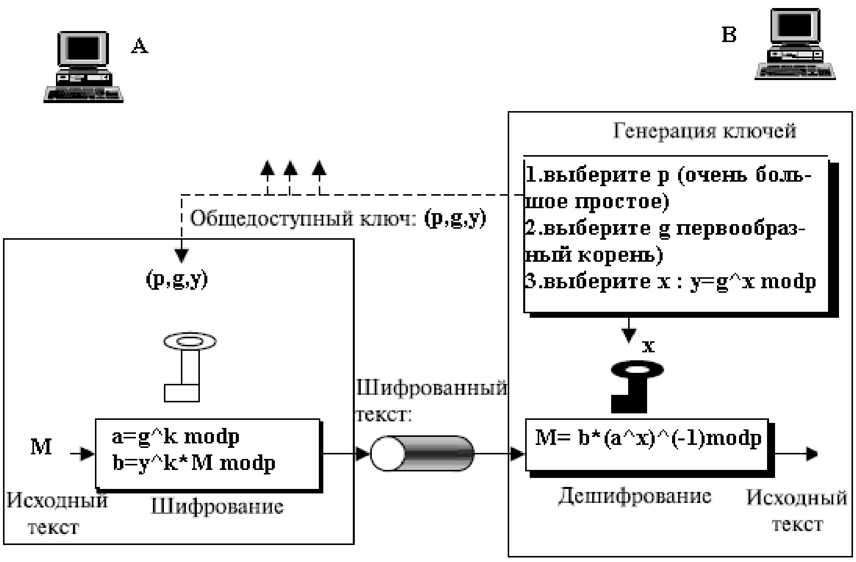

Рисунок 2 – алгоритм работы схемы Эль-Гамаля

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=41058363725152142129326129780047268409114441015993725554835256314039
467401291,

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
```