Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчёт

по лабораторной работе №4

**Ассиметричная криптография. Алгоритм Эль-Гамаля**

Выполнил:

Зеневич А. В.

Проверил:

Артемьев В. С.

Минск 2019

Содержание

[1. Постановка задачиPAGEREF \_Toc506485975 \hError: Reference source not found](#_Toc506485975)

2

2. Краткие теоретические сведения

3[. Результаты выполненияPAGEREF \_Toc506485977 \hError: Reference source not found](#_Toc506485977)

4[. ВыводыPAGEREF \_Toc506485978 \hError: Reference source not found](#_Toc506485978)

[ПРИЛОЖЕНИЕ 1. Исходный код программы.Error: Reference source not foundError: Reference source not foundPAGEREF \_Toc506485979 \hError: Reference source not found](#_Toc506485979)

# 1. Постановка задачи

1. Изучить теоретические сведения.

2. Создать программу, генерирующую ключи, а также шифрующую (дешифрующую) сообщения алгоритмом Эль-Гамаля.

**2. Краткие теоретические сведения**

Схема Эль-Гамаля (Elgamal) — криптосистема с открытым ключом, основанная на трудности вычисления дискретных логарифмов в конечном поле. Криптосистема включает в себя алгоритм шифрования и алгоритм цифровой подписи. Схема Эль-Гамаля лежит в основе бывших стандартов электронной цифровой подписи в США (DSA) и России (ГОСТ Р 34.10-94).

Схема была предложена Тахером Эль-Гамалем в 1985 году. Эль-Гамаль разработал один из вариантов алгоритма Диффи-Хеллмана. Он усовершенствовал систему Диффи-Хеллмана и получил два алгоритма, которые использовались для шифрования и для обеспечения аутентификации. В отличие от RSA алгоритм Эль-Гамаля не был запатентован и, поэтому, стал более дешевой альтернативой, так как не требовалась оплата взносов за лицензию. Считается, что алгоритм попадает под действие патента Диффи-Хеллмана.

**Алгоритм создания ключей**

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(mod p).

5.Открытым ключом является y, закрытым ключом — число x.

**Алгоритм шифрования:**

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

1. Выбирается сессионный ключ —случайное целое число k такое, что 1<k<p-1.

2. Вычисляются числа a=g^k (mod p) и b=y^(k)\*M (mod p).

3. Пара чисел (a,b) является шифротекстом.

Нетрудно увидеть, что длина шифротекста в схеме Эль-Гамаля длиннее исходного сообщения M вдвое.

**Алгоритм расшифрования:**

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

M=b(a^x)^(-1) (mod p)

При этом нетрудно проверить, что

(a^x)^(-1) (mod p) =g^(-kx) (mod p)

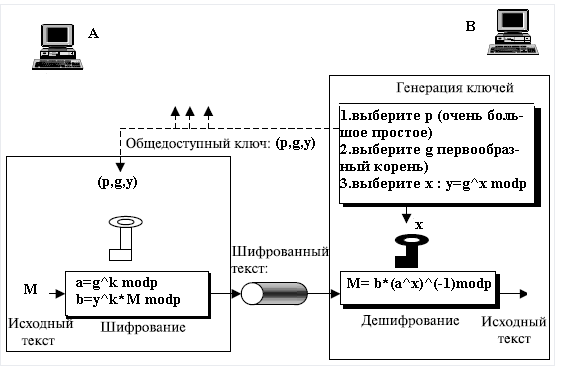
и поэтому

b(a^x)^(-1)=(y^k)M)g^(-xk) = (g^(xk)M)g^(-xk) = M (mod p).

Для практических вычислений больше подходит следующая формула:

M=b(a^x)^(-1)=ba^(p-1-x) (mod p)

**Схема алгоритма**



# Результаты выполнения

# 

# Выводы

# В результате выполнения лабораторной работы была получена реализация алгоритма Эль-Гамаля, а также алгоритм генерации ключей для данного алгоритма. Данный алгоритм относится к семейству асимметричных алгоритмов.

# ПРИЛОЖЕНИЕ 1. ИСХОДНЫЙ ТЕКСТ ПРОГРАММЫ

import random

import math

import sys

class PrivateKey(object):

def \_\_init\_\_(self, p=None, g=None, x=None, numBits=0):

self.p = p

self.g = g

self.x = x

self.numBits = numBits

class PublicKey(object):

def \_\_init\_\_(self, p=None, g=None, y=None, numBits=0):

self.p = p

self.g = g

self.y = y

self.numBits = numBits

def gcd( a, b ):

while b != 0:

c = a % b

a = b

b = c

return a

def modexp( base, exp, modulus ):

return pow(base, exp, modulus)

def SS( num, confidence ):

for i in range(confidence):

a = random.randint( 1, num-1 )

if gcd( a, num ) > 1:

return False

if not jacobi( a, num ) % num == modexp ( a, (num-1)//2, num ):

return False

return True

def jacobi( a, n ):

if a == 0:

if n == 1:

return 1

else:

return 0

elif a == -1:

if n % 2 == 0:

return 1

else:

return -1

elif a == 1:

return 1

elif a == 2:

if n % 8 == 1 or n % 8 == 7:

return 1

elif n % 8 == 3 or n % 8 == 5:

return -1

elif a >= n:

return jacobi( a%n, n)

elif a%2 == 0:

return jacobi(2, n)\*jacobi(a//2, n)

else:

if a % 4 == 3 and n%4 == 3:

return -1 \* jacobi( n, a)

else:

return jacobi(n, a )

def find\_primitive\_root( p ):

if p == 2:

return 1

p1 = 2

p2 = (p-1) // p1

while( 1 ):

g = random.randint( 2, p-1 )

if not (modexp( g, (p-1)//p1, p ) == 1):

if not modexp( g, (p-1)//p2, p ) == 1:

return g

def find\_prime(numBits, confidence):

while(1):

p = random.randint( 2\*\*(numBits-2), 2\*\*(numBits-1) )

while( not SS(p, confidence) ):

p = random.randint( 2\*\*(numBits-2), 2\*\*(numBits-1) )

while( p % 2 == 0 ):

p = random.randint(2\*\*(numBits-2), 2\*\*(numBits-1))

p = p \* 2 + 1

if SS(p, confidence):

return p

def generate\_keys(numBits=256, confidence=32):

p = find\_prime(numBits, confidence)

g = find\_primitive\_root(p)

g = modexp( g, 2, p )

x = random.randint( 2, p - 1)

y = modexp( g, x, p )

publicKey = PublicKey(p, g, y, numBits)

privateKey = PrivateKey(p, g, x, numBits)

return {'privateKey': privateKey, 'publicKey': publicKey}

def encrypt(key, plainText):

z = bytearray(plainText, 'utf-8')

cipher\_pairs = []

for i in z:

k = random.randint(2, key.p - 1)

a = modexp( key.g, k, key.p )

b = (i\*modexp( key.y, k, key.p)) % key.p

cipher\_pairs.append( [a, b] )

encryptedStr = ""

for pair in cipher\_pairs:

encryptedStr += str(pair[0]) + ' ' + str(pair[1]) + ' '

return encryptedStr

def decrypt(key, cipher):

plaintext = []

cipherArray = cipher.split()

if (not len(cipherArray) % 2 == 0):

return "Malformed Cipher Text"

for i in range(0, len(cipherArray), 2):

a = int(cipherArray[i])

b = int(cipherArray[i+1])

s = modexp( a, key.x, key.p )

plain = (b\*modexp( s, key.p-2, key.p)) % key.p

plaintext.append( plain )

decryptedText = bytearray(plaintext).decode('utf-8')

return decryptedText