Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

ОТЧЁТ

по лабораторной работе №4

По дисциплине «Методы защиты информации»

Выполнил:

Студент гр. 853505

Лазарева Е.В.

Проверил:

Олисейчик В.В.

Минск 2021

**1. Постановка задачи**

Необходимо реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритма Эль-Гамаля*.*

**ЗАДАНИЕ:**

1. Изучить теоретические сведения.
2. Создать программы, читающие данные из файла и шифрующие (дешифрующие) их с помощью алгоритма Эль-Гамаля.

**2. Теоретическая** **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).

**Генерация ключей:**

* Генерируется случайное простое число p.
* Выбирается целое число g — первообразный корень p.
* Выбирается случайное целое число x такое, что 1 < x < p - 1.
* Вычисляется y = g^x mod p.
* Открытым ключом является y, закрытым ключом — число x.

**Шифрование**

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

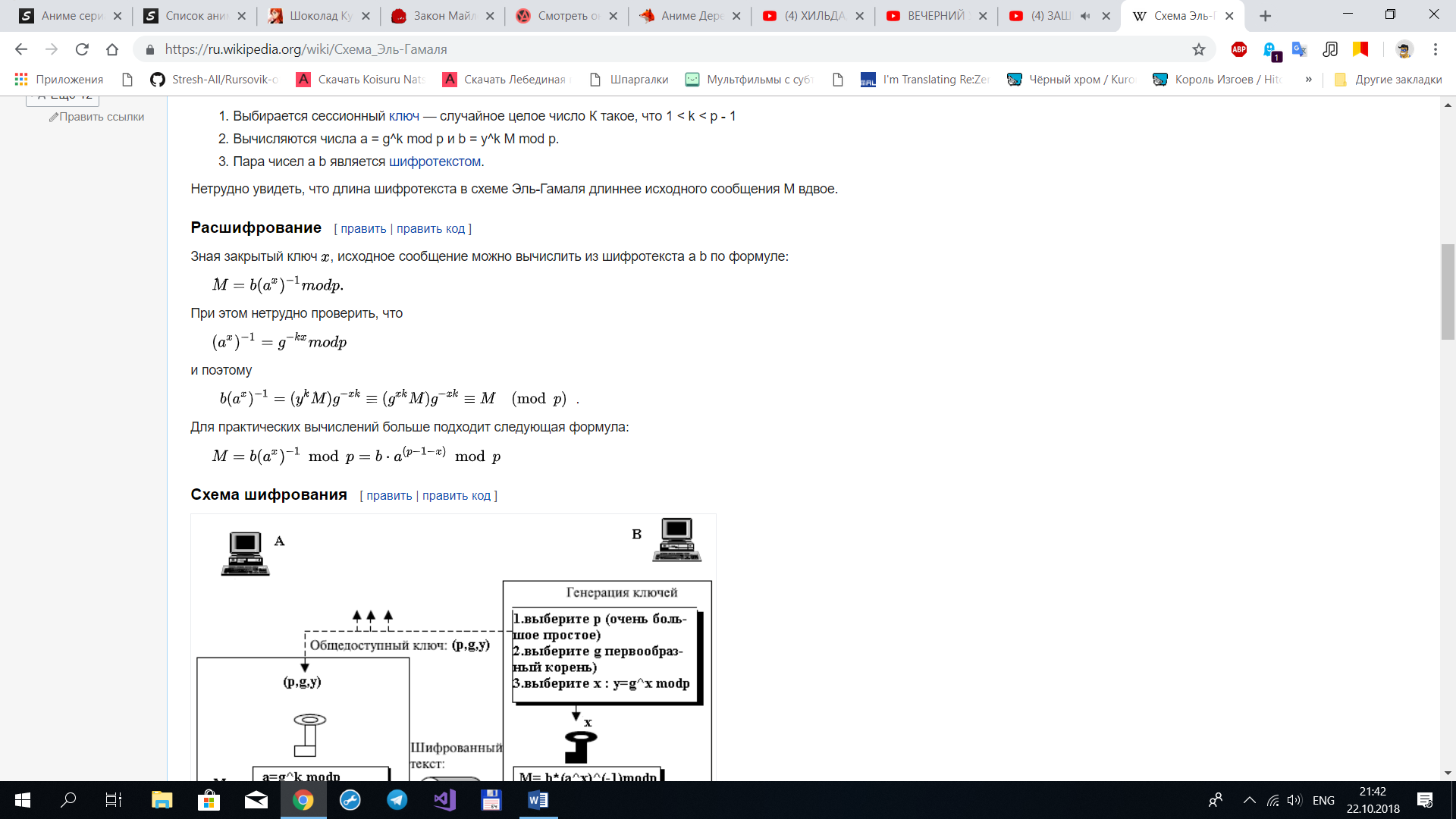
1. Выбирается сессионный ключ — случайное целое число К такое, что

1 < k < p - 1

1. Вычисляются числа a = g^k mod p и b = y^k M mod p.
2. Пара чисел a b является шифротекстом.

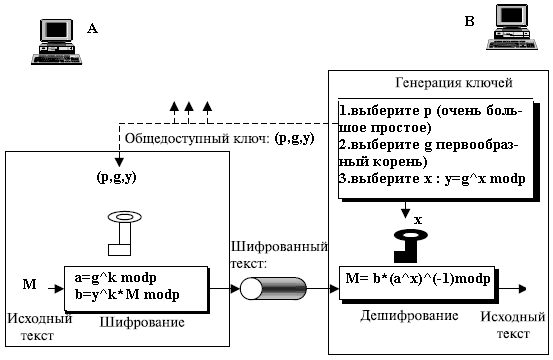
Нетрудно увидеть, что длина шифротекста в схеме Эль-Гамаля длиннее исходного сообщения M вдвое.

**Расшифрование**

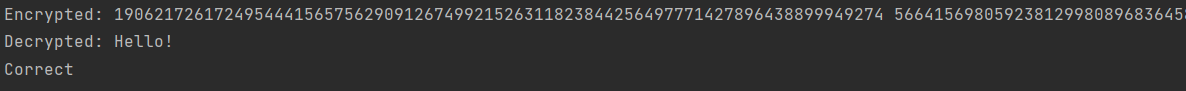


**3. Блок-схема алгоритма**

**Схема шифрования:**



1. **Пример работы программы**



**5. Код 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  
  
# computes the greatest common denominator of a and b. assumes a > b  
def gcd( a, b ):  
 while b != 0:  
 c = a % b  
 a = b  
 b = c  
 #a is returned if b == 0  
 return a  
  
#computes base^exp mod modulus  
def modexp( base, exp, modulus ):  
 return pow(base, exp, modulus)  
  
#solovay-strassen primality test. tests if num is prime  
def SS( num, confidence ):  
 #ensure confidence of t  
 for i in range(confidence):  
 #choose random a between 1 and n-2  
 a = random.randint( 1, num-1 )  
  
 #if a is not relatively prime to n, n is composite  
 if gcd( a, num ) > 1:  
 return False  
  
 #declares n prime if jacobi(a, n) is congruent to a^((n-1)/2) mod n  
 if not jacobi( a, num ) % num == modexp ( a, (num-1)//2, num ):  
 return False  
  
 #if there have been t iterations without failure, num is believed to be prime  
 return True  
  
#computes the jacobi symbol of a, n  
def jacobi( a, n ):  
 if a == 0:  
 if n == 1:  
 return 1  
 else:  
 return 0  
 #property 1 of the jacobi symbol  
 elif a == -1:  
 if n % 2 == 0:  
 return 1  
 else:  
 return -1  
 #if a == 1, jacobi symbol is equal to 1  
 elif a == 1:  
 return 1  
 #property 4 of the jacobi symbol  
 elif a == 2:  
 if n % 8 == 1 or n % 8 == 7:  
 return 1  
 elif n % 8 == 3 or n % 8 == 5:  
 return -1  
 #property of the jacobi symbol:  
 #if a = b mod n, jacobi(a, n) = jacobi( b, n )  
 elif a >= n:  
 return jacobi( a%n, n)  
 elif a%2 == 0:  
 return jacobi(2, n)\*jacobi(a//2, n)  
 #law of quadratic reciprocity  
 #if a is odd and a is coprime to n  
 else:  
 if a % 4 == 3 and n%4 == 3:  
 return -1 \* jacobi( n, a)  
 else:  
 return jacobi(n, a )  
  
  
#finds a primitive root for prime p  
#this function was implemented from the algorithm described here:  
#http://modular.math.washington.edu/edu/2007/spring/ent/ent-html/node31.html  
def find\_primitive\_root( p ):  
 if p == 2:  
 return 1  
 #the prime divisors of p-1 are 2 and (p-1)/2 because  
 #p = 2x + 1 where x is a prime  
 p1 = 2  
 p2 = (p-1) // p1  
  
 #test random g's until one is found that is a primitive root mod p  
 while( 1 ):  
 g = random.randint( 2, p-1 )  
 #g is a primitive root if for all prime factors of p-1, p[i]  
 #g^((p-1)/p[i]) (mod p) is not congruent to 1  
 if not (modexp( g, (p-1)//p1, p ) == 1):  
 if not modexp( g, (p-1)//p2, p ) == 1:  
 return g  
  
#find n bit prime  
def find\_prime(numBits, confidence):  
 #keep testing until one is found  
 while(1):  
 #generate potential prime randomly  
 p = random.randint( 2\*\*(numBits-2), 2\*\*(numBits-1) )  
  
 #keep doing this if the solovay-strassen test fails  
 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))  
  
 #if p is prime compute p = 2\*p + 1  
 #if p is prime, we have succeeded; else, start over  
 p = p \* 2 + 1  
 if SS(p, confidence):  
 return p  
  
#generates public key K1 (p, g, y) and private key K2 (p, g, x)  
def generate\_keys(numBits=256, confidence=32):  
 #p is the prime  
 #g is the primitve root  
 #x is random in (0, p-1) inclusive - private key  
 #y = g ^ x mod p - public key  
 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}  
  
  
#encrypts a string plainText using the public key k  
def encrypt(key, plainText):  
 z = bytearray(plainText, 'utf-8')  
  
 #cipher\_pairs list will hold pairs (a, b) corresponding to each byte in z  
 cipher\_pairs = []  
 #i is an integer in z  
 for i in z:  
 #pick random y from (1, p-1)  
 k = random.randint(2, key.p - 1)  
 #a = g^y mod p  
 a = modexp( key.g, k, key.p )  
 #d = ih^y mod p  
 b = (i\*modexp( key.y, k, key.p)) % key.p  
 #add the pair to the cipher pairs list  
 cipher\_pairs.append( [a, b] )  
  
 encryptedStr = ""  
 for pair in cipher\_pairs:  
 encryptedStr += str(pair[0]) + ' ' + str(pair[1]) + ' '  
  
 return encryptedStr  
  
#performs decryption on the cipher pairs found in Cipher using private key K2  
def decrypt(key, cipher):  
 #decrpyts each pair and adds the decrypted byte to list of plaintext bytes  
 plaintext = []  
  
 cipherArray = cipher.split()  
 if (not len(cipherArray) % 2 == 0):  
 return "Malformed Cipher Text"  
 for i in range(0, len(cipherArray), 2):  
 #a = first number in pair  
 a = int(cipherArray[i])  
 #b = second number in pair  
 b = int(cipherArray[i+1])  
  
 #s = c^x mod p  
 s = modexp( a, key.x, key.p )  
 #plaintext integer = ds^-1 mod p  
 plain = (b\*modexp( s, key.p-2, key.p)) % key.p  
 #add plain to list of plaintext integers  
 plaintext.append( plain )  
  
 decryptedText = bytearray(plaintext).decode('utf-8')  
  
 return decryptedText  
  
keys = generate\_keys()  
priv = keys['privateKey']  
pub = keys['publicKey']  
message = "Hello!"  
cipher = encrypt(pub, message)  
print("Encrypted:", cipher)  
plain = decrypt(priv, cipher)  
print("Decrypted:", plain)  
  
print("Correct" if plain == message else "Incorrect")

**6. Вывод**

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 дешёвой альтернативой, так как не требовалась оплата взносов за лицензию. Считается, что алгоритм попадает под действие патента Диффи-Хеллмана.