1. Министерство науки и высшего образования Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Институт кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА № 4**

1. «Криптосистемы на основе задачи дискретного логарифмирования»
2. по дисциплине «Криптографические методы защиты информации»
3. Выполнил
4. студент гр. 4851003/70801 Гасанов Э.А.

<*подпись*>

1. Преподаватель
2. ассистент Ярмак А.В.

<*подпись*>

1. Санкт-Петербург
2. 2021

**Цель работы**

Изучение протоколов шифрования с открытым ключом и электронной подписи, безопасность которых основана на задаче дискретного логарифмирования в конечном поле.

**Задачи**

1. Вариант задания — 4
2. Разработать программу (**П-1**), которая реализует схему подписи Эль-Гамаля. В программу, по крайней мере, должны быть добавлены две функции: подпись сообщения и проверка подписи. Программа также должна допускать возможность использования различных ключей. При подписи файла используется алгоритм хэширования SHA-256, причем результат хэширования файла участвует в алгоритме подписи как , где – хэш-функция, – сообщение, – порядок мультипликативной подгруппы простого поля.

**Ход работы**

Была написана программа на Python, реализующая схему подписи Эль-Гамаля. В сравнении с протоколом подписи RSA, подпись Эль-Гамаля несколько сложнее в реализации. В RSA используется меньше параметров, а процедуры формирования и проверки подписи выполняются за одну операцию. Однако со стороны производительности, в процедуре формирования подписи можно выполнить заранее некоторые операции, и тогда из операций, зависимых от сообщения, останутся лишь одно умножение и вычитание по модулю. При проверке подписи необходимо выполнить 3 модульных возведения в степень и 1 умножение. При этом у подписи Эль-Гамаля есть значительное преимущества перед RSA, заключающееся в возможности использования одного секретного ключа для нескольких сообщений, меняя лишь случайный показатель. Так как время генерации параметров может занимать значительное время, подпись Эль-Гамаля более производительна в случае необходимости подписывания нескольких сообщений.

Сгенерированные параметры для проведения лабораторной работы, а также хэш-образ сообщения приведены на рисунке 1:

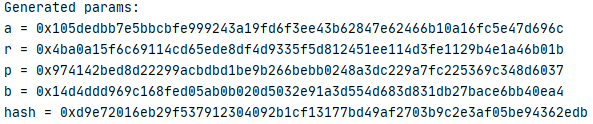


Рисунок 1 - Параметры криптосистемы

Содержимое подписываемого файла изображено на рисунке 2.

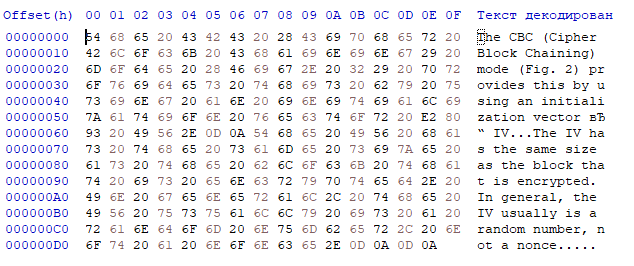


Рисунок 2 – Подписываемый файл.

На рисунке 3 представлено содержимое файла с подписью.

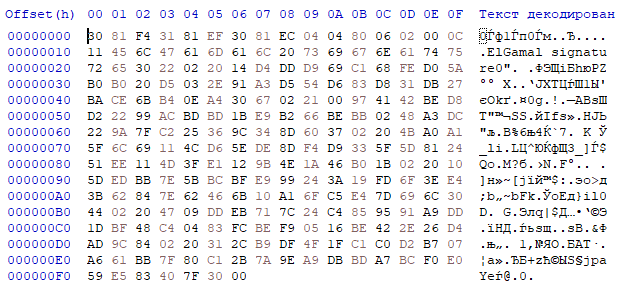


Рисунок 3 – Содержимое файла с подписью.

**Вывод**

Был изучен протокол подписи Эль-Гамаля, а также был рассмотрен протоколом шифрования Эль-Гамаля и протокол бесключевого шифрования Месси–Омуры. Также были изучены задачи, положенные в основы этих протоколов, некоторые уязвимости, и недостатки.

**Контрольные вопросы**

1. **Пусть G – конечная циклическая группа с заданной образующей. Порядок группы G известен и делится на простое число r. Как найти образующую циклической подгруппы порядка r?**

Если порядок группы , то циклическая группа порядка будет иметь следующий вид: с образующей .

Например,

G=<Z10,\*> - мультипликативная группа.

Z={***1*** ,2, ***3*** ,4,5,6, ***7*** ,8, ***9*** ,10}

Подчёркнутые числа – взаимно простые с 10. Они образуют циклическую группу.

Теперь ищем генераторы, так как они выдают всю циклическую группу:

Каждый из подчеркнутых создаёт циклическую подгруппу

30 mod(10)=1 70 mod(10)=1

31 mod(10)=3 71 mod(10)=7

32 mod(10)=9 72 mod(10)=9

33 mod(10)=7 73 mod(10)=3

1. **Покажите, что задача Месси–Омуры сводится к задаче дискретного логарифмирования.**

Пусть атакующему известны . Тогда для того, чтобы найти , необходимо найти . Зная , найти t можно следующим образом: . Таким образом, задача Месси-Омуры сводится к задаче дискретного логарифмирования.

1. **Подвержен ли протокол шифрования Эль-Гамаля к атаке на основе подобранных шифртекстов?**

Пусть атакующий может подать на вход дешифратора любой шифртекст, отличный от того, который он хочет расшифровать на самом деле. Тогда по имеющемуся шифртексту атакующий может сформировать шифртекст , подать его на вход дешифратору, и получить на выходе, откуда нетрудно восстановить желаемое сообщение .

1. **Перечислите задачи, положенные в основу схемы подписи Эль-Гамаля. Почему важно, чтобы период генератора случайных чисел, используемого в схеме подписи, был достаточно большим?**

В основу схемы подписи Эль-Гамаля положены задачи дискретного логарифмирования и предсказания следящего значения генератора псевдослучайных чисел.

Если период генератора случайных числе слишком мал, то у нарушителя могут оказаться 2 разных сообщения (m1,m2), подписанных с использованием одного и того же случайного значения , тогда и *w* повторится . Тогда нарушителю не составит труда восстановить показатель :

Зная нетрудно восстановить :

**Листинг**

Lab4.py

from Cryptodome.Hash import SHA256  
import random  
from random import randint  
from Cryptodome.Util.number import inverse  
from Cryptodome.Math.Primality import generate\_probable\_safe\_prime as gen\_safe\_prime  
  
import logging  
import sys  
from datetime import datetime  
from logging import info  
from ASN1 import \*  
  
logging.basicConfig(stream=sys.stdout, level=logging.INFO, format=**"%(message)s"**)  
  
x = 9172665889235  
  
  
def gen\_params(bits):  
 info(**"Generating prime p = 2\*q+1"**)  
 *#p = int(gen\_safe\_prime(exact\_bits=bits))* p = 68414545855766474151819781256810622433395076827137416113133938830426317348919  
 r = (p - 1) // 2  
 info(**"Composing generator of cyclic group"**)  
 while True:  
 t = randint(2, p)  
 a = pow(t, 2, p)  
 if a != 1 and a != 2:  
 break  
 return a, r, p  
  
  
def sign(m, x, r, a, p):  
 h = SHA256.new()  
 h.update(m) *# хэширование сообщения m* m = int.from\_bytes(h.digest(), **'big'**)  
 info(**f"hash =** {hex(m)}\n**hash mod r =** {hex(m % r)}**"**)  
 m %= r  
 k = randint(2, r) *# случайное к* w = pow(a, k, p) *# одна из составляющих подписей* s = (m - x \* w) \* inverse(k, r) % r  
 return w, s *# возвращает подпись*def verify(m, r, w, s, a, b, p):  
 if w >= p:  
 info(**"w < p -> False!"**)  
 return False  
 h = SHA256.new()  
 h.update(m)  
 m = int.from\_bytes(h.digest(), **'big'**) % r  
 t1 = pow(a, m, p) *# правая сторона тождества* t2 = pow(b, w, p) \* pow(w, s, p) % p *# левая сторона* print(math.gcd(t1,r))  
 print(math.gcd(t2,r))  
 print(math.gcd(pow(w, s, p),r))  
 return t1 == t2  
  
  
def P1sign(read\_name, save\_name):  
 f = open(read\_name, **"rb"**)  
 data = f.read()  
 f.close()  
 a, r, p = gen\_params(256) *# сгенерили образующую а,  
 # p = 149214232506424542174925348284318841489945784970035027525113607508250685338943431922983980532462290515556139731350769228904305416335969980621479083303784645874550630372919514547214912087146613377262988211510755586536793719098461155338440558735274461233340699138678962102192527578531895495805504891535016220599  
 # a = 7276582732350290014139280209659124688176361143398994406668808061152388050357762391613222569669363706153929813820735293553193864992684833109448833458857456274133569893554979785185634425526213796602053005245010795199265696992527476047263763531743075431526426764050482379825131507696299163160776985128722713092* x = 6902116623965059036991841364217864089635908055409966294862010731099038115552302557353630113028819120607649543936478104515944227249833886912323902003382504218213674797323855487989717715875035177610197291529683964168348765665739367203444292834411983570773063089525811202221952509212082057996473260654795522599  
 k = 10104442707181456502351613367843085645718247806286484240664997697569108436007331720799502123445002588509883391152906374053320643848807328088070167813690293125658477839845604521131666389973488951040068402327240066786775464825066221373584537980025203630186207238298261129950550269490246350251573895066741802881  
 b = pow(a, x, p) *# открытый ключ, один из трёх b,a,p* r = (p-1)//2  
 test = pow(a,r,p)  
 print(test)  
 info(**f"Generated params:**\n**a =** {hex(a)}\n**r =** {hex(r)}\n**p =** {hex(p)}\n**b =** {hex(b)}**"**)  
 w, s = sign(data, x, r, a, p)  
 data = packELsignASN1(w, s, b, p, r, a, **"ElGamal signature"**)  
 f = open(save\_name, **"wb"**)  
 f.write(data)  
 f.close()  
 info(**f"File** {read\_name} **was signed and saved to** {save\_name}**"**)  
  
  
def P1ver(sign\_name, data\_name):  
 f = open(sign\_name, **"rb"**)  
 signature = f.read()  
 f.close()  
 header, junk = parseASN1(signature)  
 alg = header[0]  
 if alg == b'\x80\x06\x02\x00':  
 info(**"ElGamal signature detected!"**)  
 str\_id = header[1]  
 b = header[2] *# открытый ключ, одна из троек открытого ключа* p = header[3] *# простое число, характеристика поля* r = header[4] *# порядок группы* a = header[5] *# генераор, образующая* w = header[6] *# подпись* s = header[7] *# подпись* f = open(data\_name, **"rb"**)  
 data = f.read()  
 f.close()  
 res = verify(data, r, w, s, a, b, p)  
 info(**f"Signature** {**''** if res == True else **'not '**}**verified!"**)  
 else:  
 info(**"Unknown algorithm! Terminating..."**)  
 exit(0)  
  
  
def main():  
 random.seed(datetime.now())  
 *# rsa = RSA(RSA\_size)  
 # rsa.genKeys()* if len(sys.argv) > 2:  
 if sys.argv[1] == **"sign"** and len(sys.argv) == 4:  
 P1sign(sys.argv[2], sys.argv[3])  
 elif sys.argv[1] == **"verify"** and len(sys.argv) == 4:  
 P1ver(sys.argv[2], sys.argv[3])  
 else:  
 info(**"Unknown cmd args! Terminating..."**)  
 exit(0)  
  
  
main()

gen\_params.py

import math  
import random  
  
from Cryptodome.Util import number  
from Cryptodome.Util.number import inverse  
from sympy.ntheory.factor\_ import factorint  
from Cryptodome.Math.Primality import generate\_probable\_safe\_prime as gen\_safe\_prime  
  
esizes = {512: 10 \*\* 8, 768: 10 \*\* 10, 1024: 2 \* 10 \*\* 11, 2048: 3 \* 10 \*\* 15, 4096: 10 \*\* 21, 8192: 10 \*\* 29}  
  
  
def get\_esize(n):  
 return esizes[n] if n in esizes else esizes[min(esizes.keys(), key=lambda k: abs(k - n))]  
  
  
def gen\_prime\_test(bits):  
 p = gen\_safe\_prime(exact\_bits=bits)  
 print(**f"p is very good param** {p}**"**)  
 return  
  
  
def gen\_prime(bits):  
 while True:  
 p = number.getStrongPrime(bits)  
 factor = factorint(p - 1, 2 \*\* 15)  
 print(**"HERE"**)  
 if len(factor) <= 3:  
 return p  
  
  
def gen\_relative\_prime(low, up):  
 a = random.randint(low, up)  
 while a % 2 == 0 or math.gcd(a, up) != 1:  
 a += 1  
 if a >= up:  
 a = random.randint(low, up)  
 return a  
  
  
def gen\_keys(p, q):  
 n = p \* q  
 phi = (p - 1) \* (q - 1)  
 low = get\_esize(n)  
 limit = int(n \*\* (1 / 4) \* 1 / 3)  
 while True:  
 e = gen\_relative\_prime(low, phi)  
 d = inverse(e, phi)  
 if d > limit:  
 return e, d, n

ASN1.py

import logging  
from logging import info  
import math  
import sys  
  
logging.basicConfig(stream=sys.stdout, level=logging.INFO, format=**"%(message)s"**)  
  
def ASN1packlen(length: int):  
 r = b''  
 if length < 128:  
 r += length.to\_bytes(1, **'big'**)  
 else:  
 t = int(math.ceil(length.bit\_length() / 8))  
 r += (t + 0x80).to\_bytes(1, **'big'**) + length.to\_bytes(t, **'big'**)  
 return r  
  
  
def ASN1packstr(a: str):  
 a = str.encode(a, **'utf-8'**)  
 length = len(a)  
 r = b'\x0c'  
 r += ASN1packlen(length)  
 r += a  
 return r  
  
  
def ASN1packbytes(a: bytes):  
 length = len(a)  
 r = b'\x04'  
 r += ASN1packlen(length)  
 r += a  
 return r  
  
  
def ASN1packint(a: int):  
 length = int(math.ceil(a.bit\_length() / 8))  
 r = b'\x02'  
 if a & 2 \*\* (length \* 8 - 1) != 0:  
 length += 1  
 r += ASN1packlen(length)  
 r += a.to\_bytes(length, **'big'**)  
 return r  
  
  
def ASN1packseq(A: list):  
 r = b''  
 length = 0  
 for a in A:  
 r += a  
 length += len(a)  
 r = b'\x30' + ASN1packlen(length) + r  
 return r  
  
  
def ASN1packset(A: list):  
 r = b''  
 length = 0  
 for a in A:  
 r += a  
 length += len(a)  
 r = b'\x31' + ASN1packlen(length) + r  
 return r  
  
  
def packsignASN1(n, e, s, str\_id):  
 str\_id = ASN1packstr(str\_id)  
 n = ASN1packint(n)  
 e = ASN1packint(e)  
 RSAblock = ASN1packseq(  
 [ASN1packbytes(b'\x00\x40'), str\_id, ASN1packseq([n, e]), ASN1packseq([]), ASN1packseq([ASN1packint(s)])])  
 return ASN1packseq([ASN1packset([RSAblock]), ASN1packseq([])])  
  
def packELsignASN1(w, s, b, p, r, a, str\_id):  
 str\_id = ASN1packstr(str\_id) *# строка со словами подпись эль-гамаль* w = ASN1packint(w) *# часть подписи* s = ASN1packint(s) *#другая часть подписи* b = ASN1packint(b) *# одна из тройки открытого ключа, ОТКРЫТЙ КЛЮЧ* p = ASN1packint(p) *# простая харктеристика поля G, вторая из тройки открытого ключа* r = ASN1packint(r) *# порядок группы r* a = ASN1packint(a) *# образующая a, генератор , часть открытого ключа* EL\_signature = ASN1packseq([w, s])  
 EL\_key = ASN1packseq([b])  
 EL\_params = ASN1packseq([p, r, a])  
 EL\_block = ASN1packseq(  
 [ASN1packbytes(b'\x80\x06\x02\x00'), str\_id, EL\_key, EL\_params, EL\_signature])  
 return ASN1packseq([ASN1packset([EL\_block]), ASN1packseq([])])  
  
  
def packencASN1(n, e, k, iv, mlen, str\_id):  
 str\_id = ASN1packstr(str\_id)  
 n = ASN1packint(n)  
 e = ASN1packint(e)  
 RSAblock = ASN1packseq(  
 [ASN1packbytes(b'\x00\x01'), str\_id, ASN1packseq([n, e]), ASN1packseq([]), ASN1packseq([ASN1packint(k)])])  
 AESblock = ASN1packseq([ASN1packbytes(b'\x10\x82'), ASN1packint(iv), ASN1packint(mlen)])  
 return ASN1packseq([ASN1packset([RSAblock]), AESblock])  
  
  
def parseASN1impl(m: bytes):  
 ret = []  
 while len(m) > 0:  
 elem\_type = m[0]  
 m = m[1:]  
 if int(m[0]) < 128:  
 length = int(m[0])  
 elem = m[1:length + 1]  
 m = m[length + 1:]  
 else:  
 ll = int(m[0]) - 128  
 length = int.from\_bytes(m[1:ll + 1], **'big'**)  
 elem = m[ll + 1:ll + 1 + length]  
 m = m[ll + 1 + length:]  
 if elem\_type == 0x02:  
 elem = int.from\_bytes(elem, **'big'**)  
 elif elem\_type == 0x0c:  
 elem = elem.decode(**'utf-8'**)  
 elif elem\_type == 0x04:  
 elem = elem  
 elif elem\_type == 0x30 or elem\_type == 0x31:  
 elem = parseASN1impl(elem)  
 if elem:  
 ret += elem  
 continue  
 else:  
 info(**"ASN1: Unsuppotred type! Terminating..."**)  
 exit(0)  
 ret.append(elem)  
 if len(m) > 0:  
 info(**"ASN1: Extra data detected! Terminating..."**)  
 exit(0)  
 else:  
 return ret  
  
  
def parseASN1(m: bytes):  
 ret = []  
 elem\_type = m[0]  
 m = m[1:]  
 if int(m[0]) < 128:  
 length = int(m[0])  
 elem = m[1:length + 1]  
 m = m[length + 1:]  
 else:  
 ll = int(m[0]) - 128  
 length = int.from\_bytes(m[1:ll + 1], **'big'**)  
 elem = m[ll + 1:ll + 1 + length]  
 m = m[ll + 1 + length:]  
 if elem\_type == 0x02:  
 elem = int.from\_bytes(elem, **'big'**)  
 elif elem\_type == 0x0c:  
 elem = elem.decode(**'utf-8'**)  
 elif elem\_type == 0x04:  
 elem = elem  
 elif elem\_type == 0x30 or elem\_type == 0x31:  
 elem = parseASN1impl(elem)  
 if elem:  
 ret += elem  
 else:  
 info(**"ASN1: Unsuppotred type!"**)  
 exit(0)  
 if len(m) > 0:  
 return ret, m  
 else:  
 return ret, None