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

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. «ОСНОВЫ КРИПТОСИСТЕМЫ RSA»
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

**Цели**

Изучение криптосистемы RSA, реализация алгоритмов зашифрования и расшифрования сообщений, формирования и проверки электронной цифровой подписи.

**Задачи**

Разработать программу (**П-1**), которая:

1. выполняет зашифрование и расшифрование файла с использованием алгоритма RSA. Зашифрование файла необходимо производить с использованием вспомогательного симметричного алгоритма AES-256 в режиме CBC, реализация симметричного алгоритма не требуется. Ключ шифрования симметричного алгоритма – случайный или изначально задан в программе. Случайный ключ шифрования алгоритма AES (32 байта) необходимо представить как число для шифрования алгоритмом RSA, порядок байтов – MSB. Старшие неиспользуемые цифры (байты) числа следует считать нулевыми;
2. выполняет формирование и проверку электронной подписи с использованием алгоритма RSA. При реализации алгоритма электронной подписи лучше всего использовать хэш-алгоритм SHA-256. Поскольку хэш-образ, вычисленный по алгоритму SHA-256, может быть длиннее модуля RSA, при проверке подписи следует сравнивать  с подписью сообщения *s*. Результатом данной части работы должна быть программа, предоставляющая пользователю возможность создать и проверить электронную подпись.

**Ход работы**

Опишем использованные параметры криптосистем.

AES:

* Длина ключа – 256 бит (32 байта)
* Размер блока – 128 бит (16 байт)
* IV – 128 бит (16 байт)
* Режим шифрования – CBC (режим сцепления блоков)
* Дополнение неполного блока – Padding16

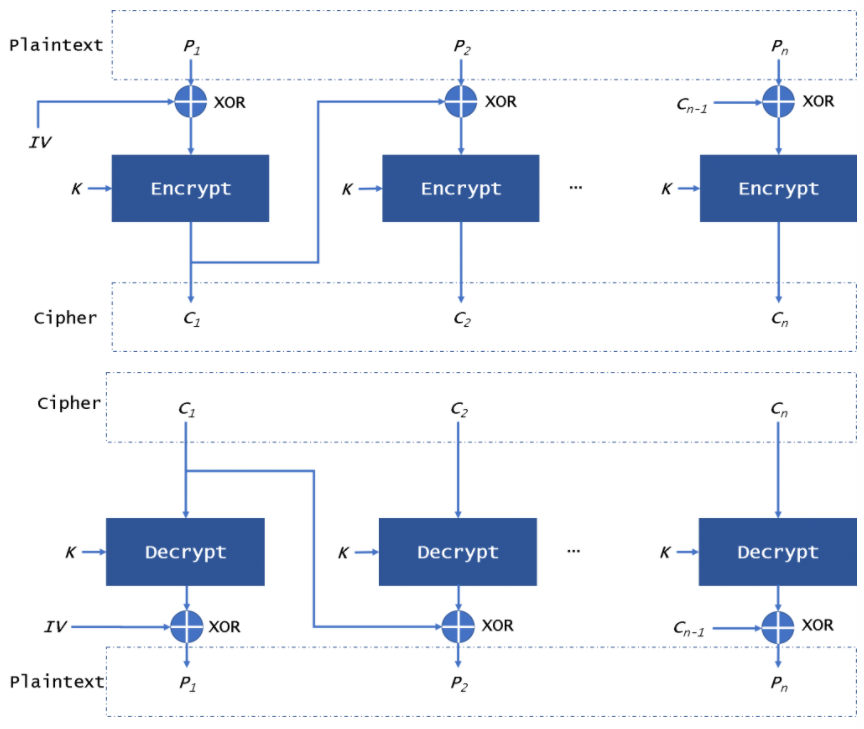


Рисунок 1 – CBC.

RSA:

* Размер простых множителей p и q, сгенерированных онлайн – 256 бит
* Размер модуля n – 512 бит
* Размер открытой(e) и секретной(d) экспонент – 512 бит

Используется односторонняя блочно-итерационная функция хэширования SHA-256 (32 байта), основанная на конструкции Меркля-Домгарда. Размер блока 64 байта,

Ключ для симметричного алгоритма генерируется псевдослучайно, то есть выбирается seed(начальное значение для ГПСЧ) на основе системного времени. Тогда ключ и IV имеют вид:

|  |
| --- |
| AES key = 9d3d25776f8d1fc04288c15e43649f033acead85656d693276079d817c427755  AES IV = 92718f1e1d5103a2ea3ed8f05045460c |

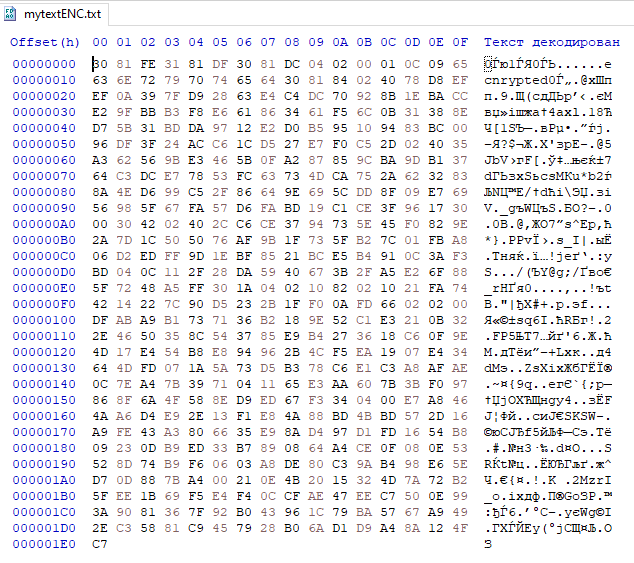


Рисунок 2 – Содержание зашифрованного файла в шестнадцатеричном виде.

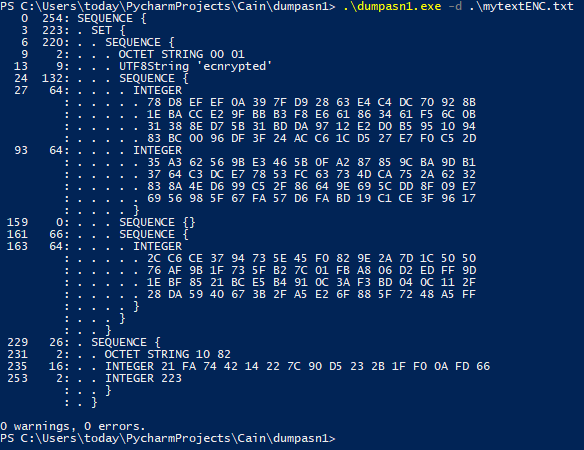


Рисунок 3 – Утилита dumpasn1.

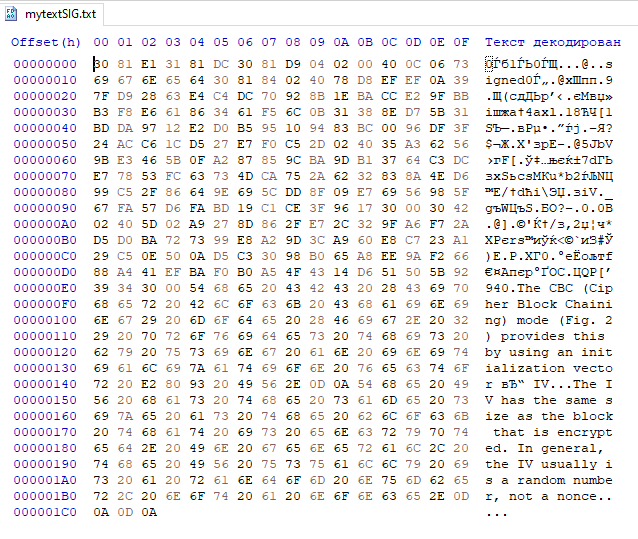


Рисунок 4 – Файл с подписью RSA.

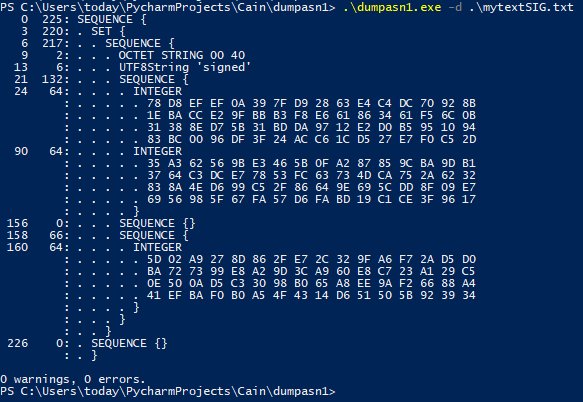


Рисунок 5 – dumpasn1 файла с подписью.

### Контрольные вопросы

1. Какие задачи положены в основу безопасности системы RSA?

Безопасность криптосистемы RSA обоснована тем, что по заданным целым числам легко найти их произведение, а разложить данное целое число на простые множители – трудно.

1. Показать, что схема RSA работает корректно для любого сообщения .

Пусть n=pq, где p и q – это простые числа. Согласно теореме Эйлера, для любого числа m (для которого выполняется НОД(m,n)=1 ) справедливо

Умножим обе его части на m, тогда получаем сравнение

.

Это сравнение справедливо для любого целого m.

Перепишем соотношение для открытого и секретного показателей в виде для некоторого целого k. Тогда

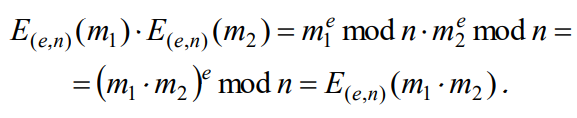
1. Доказать, что задача разложения числа  на множители и задача вычисления функции Эйлера  полиномиально эквивалентны.

Если известно разложение числа , то со сложностью можно вычислить функцию Эйлера как . Если известны значения n и , то ,зная, что n – это произведение двух различных простых чисел, из соотношения

получаем сумма , а произведение . Тогда по теореме Виета p и q являются целочисленными корнями квадратного уравнения , сложность решения которого

1. Показать, что схема RSA обладает свойством гомоморфности относительно операции умножения.

Криптосистема RSA обладает мультипликативным гомоморфизмом. Для любых открытых текстов ∈ , и открытого ключа (e,n) выполняется равенство:



**Вывод**

В ходе выполнения данной лабораторной работы были изучены криптосистема RSA, реализация алгоритмов зашифрования и расшифрования сообщений, формирования и проверки электронной цифровой подписи.

Размер ключа в алгоритме RSA связан с размером модуля n. Два числа p и q, произведением которых является модуль, должны иметь приблизительно одинаковую длину поскольку в этом случае найти сомножители (факторы) сложнее, чем в случае, когда длина чисел значительно различается. Оптимальный размер модуля определяется требованиями безопасности: модуль большего размера обеспечивает большую безопасность, но и замедляет работу алгоритма RSA. Длина модуля выбирается в первую очередь на основе значимости защищаемых данных.

В 1999 году 512-битный ключ был вскрыт за семь месяцев и это означает, что 512-битные ключи уже не обеспечивают достаточную безопасность за исключением очень краткосрочных задач безопасности. Это означает, что выбранный мною размер ключа является слабым местом и её результатами можно пользоваться для очень краткосрочных задач безопасности.

В настоящее время Лаборатория RSA рекомендует для обычных задач ключи размером 1024 бита, а для особо важных задач – 2048 битов.

**Листинг**

import logging  
import sys  
from datetime import datetime  
from logging import info  
from Cryptodome.Cipher import AES  
from Cryptodome.Hash import SHA256  
from Cryptodome.Util.Padding import pad, unpad  
from cypari import pari  
  
from gen\_params import \*  
from ASN1 import \*  
  
logging.basicConfig(stream=sys.stdout, level=logging.INFO, format=**"%(message)s"**)  
  
RSA\_size = 512  
d = 0x77E693EAA7C0DFD69AEB21130E0DF891178FA230CC906D095D06A1830164E4EF6375295EAA6A19FAD30E7BB4972FFBDB71A937AA2CEE3BC1ADA1C57B30A217A7  
e = 0x35A362569BE3465B0FA287859CBA9DB13764C3DCE77853FC63734DCA752A6232838A4ED699C52F86649E695CDD8F09E76956985F67FA57D6FABD19C1CE3F9617  
n = 0x78D8EFEF0A397FD92863E4C4DC70928B1EBACCE29FBBB3F8E661863461F56C0B31388ED75B31BDDA9712E2D0B595109483BC0096DF3F24ACC61CD527E7F0C52D  
  
def len\_in\_bytes(a\_: int):  
 return math.ceil(math.floor(math.log2(n)) / 8)  
  
  
class RSA:  
 def \_\_init\_\_(self, N: int):  
 self.N = N  
  
 def genKeys(self):  
 p = gen\_prime(self.N)  
 q = gen\_prime(self.N)  
 e, d, n = gen\_keys(p, q)  
 info(**f"n =** {n}\n**"  
 f"p =** {p}\n**"  
 f"q =** {q}\n**"  
 f"e =** {e}\n**"  
 f"d =** {d}\n**"**)  
 return n, e, d  
  
 def encrypt(self, m, n, e):  
 res = bytes()  
 bs\_enc = len\_in\_bytes(n)  
 bs\_pl = bs\_enc - 1  
 for i in range(0, len(m), bs\_pl):  
 t = int.from\_bytes(m[i:i + bs\_pl], **'big'**)  
 c = pow(t, e, n)  
 res += int(c).to\_bytes(bs\_enc, **'big'**)  
 return res  
  
 def decrypt(self, m, n, d, m\_last\_size):  
 res = bytes()  
 bs\_enc = len\_in\_bytes(n)  
 bs\_pl = bs\_enc - 1  
 for i in range(0, len(m), bs\_enc):  
 t = int.from\_bytes(m[i:i + bs\_enc], **'big'**)  
 c = pow(t, d, n)  
 res += int(c).to\_bytes(bs\_pl, **'big'**)  
 m\_last\_size = bs\_pl if m\_last\_size % bs\_pl == 0 else m\_last\_size % bs\_pl  
 res = res[:-bs\_pl] + res[-m\_last\_size:]  
 return res  
 def decrypt1(self, m, n, d):  
 res = bytes()  
 bs\_pl = math.ceil(math.floor(math.log2(n)) / 8) - 1  
 bs\_enc = math.ceil(math.floor(math.log2(n)) / 8)  
 for i in range(0, len(m), bs\_enc):  
 t = int.from\_bytes(m[i:i + bs\_enc], **'big'**)  
 c = (pari(t).Mod(pari(n)) \*\* pari(d)).lift()  
 res += int(c).to\_bytes(bs\_pl, **'big'**)  
 return res  
  
 def sign(self, m, n, d):  
 h = SHA256.new() *#вычисляем хэш образ* h.update(m)  
 r = h.digest()  
 return self.encrypt(r, n, d) *# зашифр хэш* def verify(self, s, m, n, e):  
 r = self.decrypt1(s, n, e) *#предполгагаемый хэш-образ путём рашифровки* h = SHA256.new() *#вычисл хэш-образ* h.update(m)  
 return int.from\_bytes(r, **'big'**) == int.from\_bytes(h.digest(), **'big'**) *#сравниваем*def encAES(m):  
 m = pad(m, 16)  
 key = random.randint(1, 2 \*\* 256).to\_bytes(32, **'big'**)  
 iv = random.randint(1, 2 \*\* 128).to\_bytes(16, **'big'**)  
 cipher = AES.new(key, AES.MODE\_CBC, iv)  
 ciphertext = cipher.encrypt(m)  
 return ciphertext, key, iv  
  
  
def decAES(m, key, iv):  
 cipher = AES.new(key, AES.MODE\_CBC, iv)  
 plaintext = cipher.decrypt(m)  
 return unpad(plaintext, 16)  
  
  
def P1enc(read\_name, save\_name):  
 f = open(read\_name, **"rb"**)  
 data = f.read()  
 f.close()  
 length = len(data)  
 data, key, iv = encAES(data)  
 rsa = RSA(RSA\_size)  
 *#n, e, d = rsa.genKeys()* enc\_key = rsa.encrypt(key, n, e)  
 data = packencASN1(n, e, int.from\_bytes(enc\_key, **'big'**), int.from\_bytes(iv, **'big'**), length, **"ecnrypted"**) + data  
 f = open(save\_name, **"wb"**)  
 f.write(data)  
 f.close()  
 info(**f"File** {read\_name} **was encrypted and saved to** {save\_name}**"**)  
 info(  
 **f"Used params:**\n**RSA modulus size =** {RSA\_size}\n**n =** {hex(n)}\n**e =** {hex(e)}\n**n =** {hex(n)}\n**d =** {hex(d)}\n**AES key =** {key.hex()}\n**AES IV =** {iv.hex()}**"**)  
  
  
def P1sign(read\_name, save\_name):  
 f = open(read\_name, **"rb"**)  
 data = f.read()  
 f.close()  
 rsa = RSA(RSA\_size)  
 *# n, e, d = rsa.genKeys()* s = rsa.sign(data, n, d) *#зашифр хэш* data = packsignASN1(n, e, int.from\_bytes(s, **'big'**), **"signed"**) + data  
 f = open(save\_name, **"wb"**)  
 f.write(data)  
 f.close()  
 info(**f"File** {read\_name} **was signed and saved to** {save\_name}**"**)  
  
  
def P1dec(m, length, n, k, iv):  
 rsa = RSA(RSA\_size)  
 k = rsa.decrypt(k.to\_bytes(int(math.ceil(k.bit\_length() / 8)), **'big'**), n, d, 32)  
 iv = iv.to\_bytes(16, **'big'**)  
 m = decAES(m, k, iv)  
 return m  
  
  
def P1ver(s, m, n, e):  
 rsa = RSA(RSA\_size)  
 return rsa.verify(s.to\_bytes(int(math.ceil(s.bit\_length() / 8)), **'big'**), m, n, e)  
  
  
def P1dec\_ver(read\_name, save\_name):  
 f = open(read\_name, **"rb"**)  
 data = f.read()  
 f.close()  
 header, data = parseASN1(data)  
 alg = header[0]  
 if alg == b'\x00\x01':  
 info(**"RSA-AES algorith detected!"**)  
 str\_id = header[1]  
 n = header[2]  
 e = header[3]  
 k = header[4]  
 iv = header[6]  
 length = header[7]  
 data = P1dec(data, length, n, k, iv)  
 f = open(save\_name, **"wb"**)  
 f.write(data)  
 f.close()  
 info(**f"File decrypted and saved to** {save\_name}**!"**)  
 elif alg == b'\x00\x40':  
 info(**"RSA-SHA256 algorithm detected!"**)  
 str\_id = header[1]  
 n = header[2]  
 e = header[3]  
 s = header[4]  
 res = P1ver(s, data, n, e)  
 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:  
 P1dec\_ver(sys.argv[2], sys.argv[3])  
 elif sys.argv[1] == **"encrypt"** and len(sys.argv) == 4:  
 P1enc(sys.argv[2], sys.argv[3])  
 elif sys.argv[1] == **"decrypt"** and len(sys.argv) == 4:  
 P1dec\_ver(sys.argv[2], sys.argv[3])  
 else:  
 info(**"Unknown cmd args! Terminating..."**)  
 exit(0)  
  
  
main()