ФГБОУ ВПО «Чувашский государственный университет им. И. Н. Ульянова»

Факультет ИВТ

Кафедра Вычислительной Техники

Лабораторная работа №1

**Реализация алгоритма шифрования RSA. Оптимизация и закладки**

**Выполнил:**

Студент гр. ИВТ-21-11

Исаков Тимофей

**Проверил:**

Лобастова О. А.

Чебоксары 2014г.

**Цель работы.**

Целью работы является знакомство с принципами работы алгоритма RSA для шифрования и подписи сообщений.

**Задание.**

Необходимо реализовать генерацию ключей, операции зашифрования и расшифрования. Выполнить оптимизацию расшифрования с помощью китайской теоремы об остатках. Оценить производительность классической реализации в сравнении с использованием Китайской теоремы об остатках на различных параметрах e и d. Графически отобразить зависимость скорости работы от размера ключа.

По статье «Встраиваем бэкдор в публичный ключ RSA» выполнить внедрение закладки в генерируемые ключи. Оценить накладные расходы на внедрение закладки.

**Краткая теория.**

RSA (аббревиатура от фамилий Rivest, Shamir и Adleman) — криптографический алгоритм с открытым ключом, основывающийся на вычислительной сложности задачи факторизации больших целых чисел.

RSA-ключи генерируются следующим образом:[[14]](https://ru.wikipedia.org/wiki/RSA#cite_note-rsa-in-hac-14)

1. Выбираются два различных [случайных простых числа](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE) p и q заданного размера (например, 1024 [бита](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82) каждое).
2. Вычисляется их произведение n=p\cdot q, которое называется *модулем*.
3. Вычисляется значение [функции Эйлера](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%D0%AD%D0%B9%D0%BB%D0%B5%D1%80%D0%B0) от числа n:

\varphi(n) = (p-1)\cdot (q-1).

1. Выбирается целое число e (1 < e < \varphi(n)), [взаимно простое](https://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%BE_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%8B%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%B0) со значением функции \varphi(n). Обычно в качестве e берут простые числа, содержащие небольшое количество единичных бит в [двоичной записи](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F), например, простые [числа Ферма](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B5%D1%80%D0%BC%D0%B0) 17, 257 или 65537.
   * Число e называется *открытой экспонентой* ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *public exponent*)
   * Время, необходимое для шифрования с использованием [быстрого возведения в степень](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%B1%D1%8B%D1%81%D1%82%D1%80%D0%BE%D0%B3%D0%BE_%D0%B2%D0%BE%D0%B7%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B2_%D1%81%D1%82%D0%B5%D0%BF%D0%B5%D0%BD%D1%8C), пропорционально числу единичных бит в e.
   * Слишком малые значения e, например 3, потенциально могут ослабить безопасность схемы RSA.[[15]](https://ru.wikipedia.org/wiki/RSA#cite_note-Boneh-15)
2. Вычисляется число d, [мультипликативно](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BF%D0%BB%D0%B8%D0%BA%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) обратное к числу e по модулю \varphi(n), то есть число, удовлетворяющее [сравнению](https://ru.wikipedia.org/wiki/%D0%A1%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8E):

d\cdot e \equiv 1 \pmod{\varphi(n)}.

* + Число d называется *секретной экспонентой*. Обычно, оно вычисляется при помощи [расширенного алгоритма Евклида](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D1%88%D0%B8%D1%80%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%95%D0%B2%D0%BA%D0%BB%D0%B8%D0%B4%D0%B0).

1. Пара \left\{ e, n \right\} публикуется в качестве *открытого ключа RSA* ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *RSA public key*).
2. Пара \left\{ d, n \right\} играет роль *закрытого ключа RSA* ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *RSA private key*) и держится в секрете.

**Шифрование**: c = E(m) = m^e \mod n ~~~~ (1)

**Расшифрование**: m = D(c) = c^d \mod n ~~~~(2)

**Выполнение задания.**

Исходный код программы писался на языке программирования Python.

Самописная библиотека для работы с большими числами:

# -\*- coding: utf-8 -\*-

import random # библиотека для работы со случайными числами

import sys

import timeit

'''

Генерация простого числа

произвольной длины.

'''

def Zpow(a, p, m):

"""

Функция возведения в степень по модулю

принимает 3 аргумента

a - сам математический аргумент, возводимое число

p - степень

m - модуль

"""

result = 1

while p > 2: # когда степень сократится до квадрата и меньше - завершаем

if p % 2 == 0: # если степень кратна 2

a = (a \*\* 2) % m

p = p // 2 # целочисленное деление (на всякий)

else:

result = (result \* a) % m

p = p - 1

a = (a \*\* p) % m

result = (result \* a) % m

return result

def miller\_rabin\_test(m, r):

'''

Вероятностый полинимиальный тест простоты

:param x: входное число

:param r: число раундов проверки на простоту, чем больше раундов, тем лучше проверка

:return: возвращает или False - составное, или True - вероятно простое

'''

if m < 4: # 1 - не простое число

return False

if m == 3:

return True #

if m % 2 == 0:

return False # число составное, так как делится на 2

# условия выше можно убрать, так как в тест Миллера-Рабина должны даваться

# числа большие 3 и нечетные. Если эти условия будут проверяться до вызова

# функции, то будет теряться только время. Я оставляю эти условия

# для защиты от дурака

# Представим m - 1 = 2^s\*t, найдем t - нечетное.

buf\_m = m - 1

s = 0

while buf\_m % 2 == 0:

buf\_m /= 2

s += 1

t = buf\_m

for i in range(0, r): # первый цикл

a = random.randint(2, m - 2)

#x = a\*\*t % m # можно при помощи алгоритма возведения в степень по модулю

x = Zpow(a, t, m) # это алгоритм не самый оптимальный, лучше Монтгомери, но я пока не осилил,

# у меня есть исходник свой, который я давно писал, но я его не совсем понял :D

if x == 1 or x == m - 1:

continue # перейти на следующую итерацию

trigger = False # служебный

for j in range(0, s-1):

x = x\*\*2 % m

if x == 1:

return False

if x == m - 1:

trigger = True

if trigger:

continue

return False

return True # вероятно простое

def AlgEvklid(a, b, d=0, x=0, y=0):

'''

Расширенный алгоритм Евклида

:param a:

:param b:

:param d: НОД

:param x:

:param y:

:return: НОД

'''

if ( b == 0 ):

d = a

x = 1

y = 0

return y

x2 = 1

x1 = 0

y2 = 0

y1 = 1

while (b > 0):

q = a / b

r = a - q\*b

x = x2 -q\*x1

y = y2 -q\*y1

a = b

b = r

x2 = x1

x1 = x

x1 =x

y2 = y1

y1 = y

d = a

x = x2

y = y2

return d

def AlgEvklid\_ex(a, b, d=0, x=0, y=0):

'''

Расширенный алгоритм Евклида

:param a:

:param b:

:param d: НОД

:param x:

:param y:

:return: НОД

'''

if ( b == 0 ):

d = a

x = 1

y = 0

return y

x2 = 1

x1 = 0

y2 = 0

y1 = 1

while (b > 0):

q = a / b

r = a - q\*b

x = x2 -q\*x1

y = y2 -q\*y1

a = b

b = r

x2 = x1

x1 = x

x1 =x

y2 = y1

y1 = y

d = a

x = x2

y = y2

return {

'gcd': d,

'x': x,

'y': y

}

def generate\_prime(S, max\_count = 100):

'''

Генерация простого числа

:param S: простое число

:param max\_count: максимальное число итераций на поиск простого числа

:return: существенно большее простое число или None, если такое не найдено

'''

step = 0

while True:

R = random.randint(2, 4\*S+2)

if R % 2 != 0:

R += 1

N = S\*R + 1

# проверка на отсутствие малых простых делителей,

# этот пункт опускаю, так как осуществляю проверку в Рабине-Миллере

# проверка Миллер-Рабин

if miller\_rabin\_test(N, 50):

for i in range(1000):

a = random.randint(2, N-1)

if Zpow(a, N-1, N) == 1:

return N

step += 1

if step > max\_count:

return None

return

def generate\_prime\_fix\_len(bits\_len):

'''

:param bits\_len: битовая длина выходного числа

:return: prime number

'''

if bits\_len < 1:

return False

border1 = 2\*\*(bits\_len-1)

border2 = 2\*\*bits\_len-1

primes = []

fin = open('prime20k', 'r')

for num in fin:

primes.append(int(num))

while True:

is\_prime = True

num = random.randint(border1, border2)

if num % 2 == 0:

continue

for divider in primes:

if num % divider == 0:

is\_prime = False

break

if is\_prime:

if miller\_rabin\_test(num, 5):

return num

def main():

setup = "from prime\_num import generate\_prime\_fix\_len"

times = 10

i = 8

while i < 5000:

print "Length is %d bits: %s" %\

(i, sum(timeit.repeat('generate\_prime\_fix\_len(%d)'%(i), setup, timeit.default\_timer, times, 1)) / times)

i \*= 2

return

if \_\_name\_\_ == "\_\_main\_\_":

main()

Генерация RSA-ключа, шифрование, расшифрование:

# -\*- coding: utf-8 -\*-

from random import random

from prime\_num import generate\_prime, AlgEvklid, Zpow, generate\_prime\_fix\_len, AlgEvklid\_ex

import sys

import math

def generate\_rsa\_key(bits\_len):

p = None

q = None

while p is None:

p = generate\_prime\_fix\_len(bits\_len) # generate 1st prime

while q is None:

q = generate\_prime\_fix\_len(bits\_len) # generate 2nd prime

n = p\*q # modulo

phi = (p - 1) \* (q - 1) # Euler function of n

# let's start to generate RSA key

pos = 256

e = 2\*\*int(math.sqrt(bits\_len/8))

x = 0

y = 0

while AlgEvklid(phi, e, x, y) != 1:

pos <<= 1

e = pos \*\* 2 + 1

res = AlgEvklid\_ex(phi, e)

d = res['y']

if d < 0:

d += phi

d %= phi

public\_key = {

'e': e,

'n': n

}

private\_key = {

'd': d,

'n': n

}

return {

'public\_key': public\_key,

'private\_key': private\_key

}

def rsa\_encrypt(mes, key, bits):

length = len(mes)

block\_size = bits/32 # размер блока

blocks = length / block\_size # 1 блок равен 1/2 ключа

st\_pos = 0

res = ""

f = open('out', 'wb')

for i in xrange(blocks):

v = ""

f.write(str(i) + ': \n')

for i in mes[st\_pos:st\_pos+block\_size]:

num = (hex(ord(i)))[2:]

if len(num) == 1:

num = '0' + num

v += num

st\_pos += block\_size

v = '0x' + v

v = long(v, 16)

f.write(str(v) + '\n')

cr = Zpow(v, key['public\_key']['e'], key['public\_key']['n'])

wr = hex(cr)[2:-1]

while len(wr) < bits/2:

wr = '0' + wr

f.write(str(wr)+'\n\n')

res += wr

f.close()

return res

def rsa\_decrypt(res, key, bits):

length = len(res)

block\_size = bits/2 # размер блока

blocks = length / block\_size # 1 блок равен 1/2 ключа

st\_pos = 0

result = ''

f = open('out2', 'wb')

for i in xrange(blocks):

v = ""

mes = ""

f.write(str(i)+': \n')

for i in res[st\_pos:st\_pos+block\_size]:

mes += i

st\_pos += block\_size

f.write(str(mes) + '\n')

mes = '0x' + mes

mes = long(mes, 16)

answer = Zpow(mes, key['private\_key']['d'], key['private\_key']['n'])

f.write(str(answer)+'\n\n')

tmp = hex(answer)[2:-1]

if len(tmp) % 2 == 1:

tmp = '0' + str(tmp)

result += str(tmp.decode('hex'))

return result

Создание оптимизированного ключа, алгоритм расшифрования таким ключом:

# -\*- coding: utf-8 -\*-

from random import random

from prime\_num import generate\_prime, AlgEvklid, Zpow, generate\_prime\_fix\_len, AlgEvklid\_ex

import sys

import math

def generate\_opt\_rsa\_key(bits\_len):

p = None

q = None

while p is None:

p = generate\_prime\_fix\_len(bits\_len) # generate 1st prime

while q is None:

q = generate\_prime\_fix\_len(bits\_len) # generate 2nd prime

n = p\*q # modulo

phi = (p - 1) \* (q - 1) # Euler function of n

# let's start to generate RSA key

pos = 256

e = 2\*\*int(math.sqrt(bits\_len/8))

x = 0

y = 0

while AlgEvklid(phi, e, x, y) != 1:

pos <<= 1

e = pos \*\* 2 + 1

res = AlgEvklid\_ex(phi, e)

d = res['y']

if d < 0:

d += phi

d %= phi

dP = d % (p - 1)

dQ = d % (q - 1)

qinv = AlgEvklid\_ex(p, q)['y']

if qinv < 0:

qinv += p

public\_key = {

'e': e,

'n': n

}

private\_key = {

'dP': dP,

'dQ': dQ,

'p': p,

'q': q,

'qinv': qinv,

'd': d,

'n': n

}

return {

'public\_key': public\_key,

'private\_key': private\_key

}

def rsa\_decrypt\_opt(res, key, bits):

length = len(res)

block\_size = bits/2 # размер блока

blocks = length / block\_size # 1 блок равен 1/2 ключа

st\_pos = 0

result = ''

f = open('out2', 'wb')

for i in xrange(blocks):

v = ""

mes = ""

f.write(str(i)+': \n')

for i in res[st\_pos:st\_pos+block\_size]:

mes += i

st\_pos += block\_size

f.write(str(mes) + '\n')

mes = '0x' + mes

mes = long(mes, 16)

m1 = Zpow(mes, key['private\_key']['dP'], key['private\_key']['p'])

m2 = Zpow(mes, key['private\_key']['dQ'], key['private\_key']['q'])

# if m1 < m2:

# h = key['private\_key']['qinv']\*(m1+(key['private\_key']['q']\*key['private\_key']['p'])\*key['private\_key']['p'] - m2 ) % key['private\_key']['p']

# else:

h = (key['private\_key']['qinv'] \* (m1 - m2) )% key['private\_key']['p']

answer = m2 + h \* key['private\_key']['q']

f.write(str(answer)+'\n\n')

tmp = hex(answer)[2:-1]

if len(tmp) % 2 == 1:

tmp = '0' + str(tmp)

result += str(tmp.decode('hex'))

return result

Алгоритм создания ключа с backdoor:

# -\*- coding: utf-8 -\*\_

from random import random, seed

from prime\_num import generate\_prime, AlgEvklid, Zpow, generate\_prime\_fix\_len, AlgEvklid\_ex, miller\_rabin\_test

import sys

import math

from opt\_rsa import generate\_opt\_rsa\_key

from standart\_rsa import rsa\_decrypt, rsa\_encrypt

def generate\_bad\_rsa\_key(bits\_len, payload, offset):

if bits\_len < len(payload)/10:

sys.stderr("Fatal error! Length of a key cannot be less than 256 bits")

exit()

seed(payload)

key = generate\_opt\_rsa\_key(bits\_len)

buf = hex(key['public\_key']['n'])[2:]

buf = buf[:offset] + payload + buf[offset+len(payload):]

n = long(buf, 16)

p = key['private\_key']['p']

q = n / p

while not miller\_rabin\_test(q, 30):

q += 1

n = p\*q # modulo

phi = (p - 1) \* (q - 1) # Euler function of n

# let's start to generate RSA key

pos = 256

e = 2\*\*int(math.sqrt(bits\_len/8))

x = 0

y = 0

while AlgEvklid(phi, e, x, y) != 1:

pos <<= 1

e = pos \*\* 2 + 1

res = AlgEvklid\_ex(phi, e)

d = res['y']

if d < 0:

d += phi

d %= phi

public\_key = {

'e': e,

'n': n

}

private\_key = {

'd': d,

'n': n

}

return {

'public\_key': public\_key,

'private\_key': private\_key

}

def get\_payload(n, offset, length):

return hex(n)[2:][offset:offset+length]

payload = "06"

if \_\_name\_\_ == "\_\_main\_\_":

offest =2

key = generate\_bad\_rsa\_key(int(sys.argv[1]), payload, offest)

mes = open(sys.argv[2], 'rb').read()

crypt = rsa\_encrypt(mes, key, int(sys.argv[1]))

fout = open('backdoor/rsa\_encrypt', 'wb').write(crypt)

res = rsa\_decrypt(crypt, key, int(sys.argv[1]))

fout = open('backdoor/rsa\_decrypt', 'wb').write(res)

pay = get\_payload(key['public\_key']['n'], offest, 2)

print pay

bad\_key = generate\_bad\_rsa\_key(int(sys.argv[1]), pay, offest)

res = rsa\_decrypt(crypt, bad\_key, int(sys.argv[1]))

fout = open('backdoor/rsa\_backdoor\_decrypt', 'wb').write(res)

**Пример работы программы.**

Для нахождения среднего значения проводилось 30 проходов каждой функции.

Шифрование и расшифрование без оптимизации и с оптимизацией.

Исходный текст:

The MIT License (MIT)

Copyright (c) 2015 tisOO

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

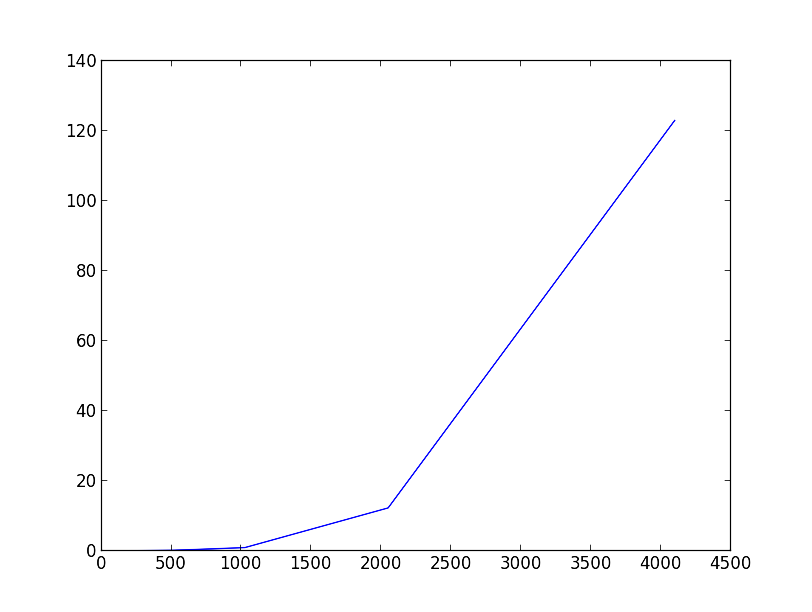
SOFTWARE.

Зашифрованный файл:



Выходные файлы получились идентичны оригинальному.

**Анализ производительности**



*Генерация простых чисел для RSA-ключа*

Генерация простых чисел для RSA-ключа

Length is 8 bits: 0.00286498069763

Length is 16 bits: 0.00103189945221

Length is 32 bits: 0.00149788856506

Length is 64 bits: 0.00425062179565

Length is 128 bits: 0.0110202074051

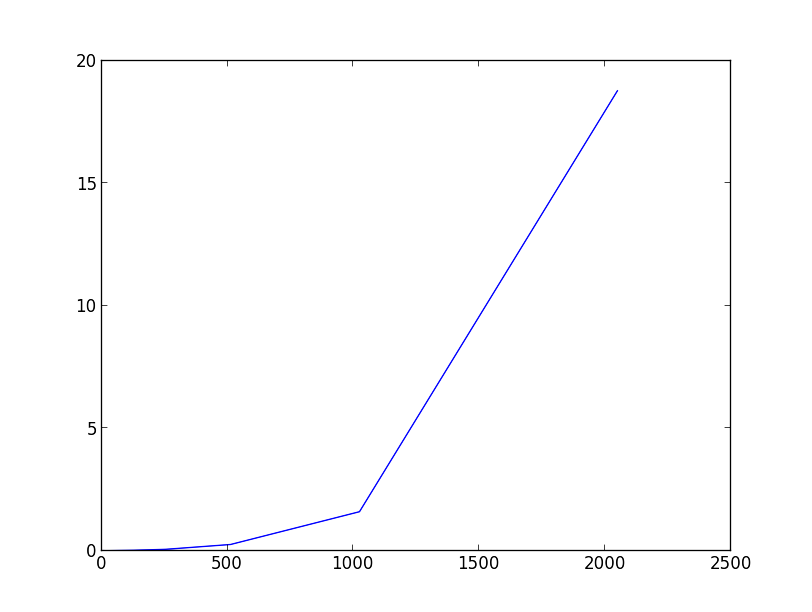
Length is 256 bits: 0.0199715137482

Length is 512 bits: 0.126261591911

Length is 1024 bits: 0.902797532082

Length is 2048 bits: 12.2377109289

Length is 4096 bits: 122.955223346

Генерация RSA-ключа, неоптимизированный алгоритм.

Length is 8 bits: 0.00462395350138

Length is 16 bits: 0.00224466323853

Length is 32 bits: 0.00299075444539

Length is 64 bits: 0.0090324083964

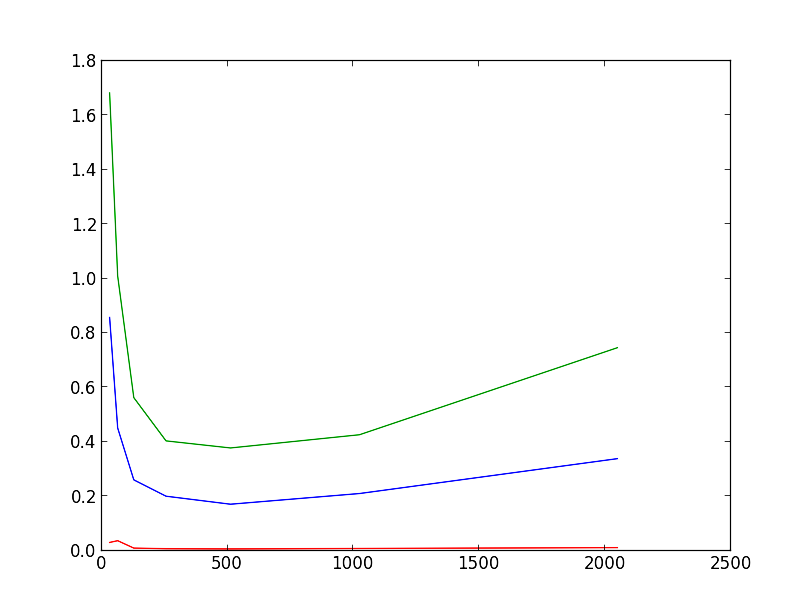
Length is 128 bits: 0.0182889461517

Length is 256 bits: 0.059876259168

Length is 512 bits: 0.255465078354

Length is 1024 bits: 1.59314599832

Length is 2048 bits: 18.77348303

**Шифрование файлов.**

Шифрование файлов размером 1072, 30016, 60032 символов.

File 1. Length is 32 bits: 0.0305180549622

File 2. Length is 32 bits: 0.857126951218

File 3. Length is 32 bits: 1.68183994293

File 1. Length is 64 bits: 0.0369710922241

File 2. Length is 64 bits: 0.450930118561

File 3. Length is 64 bits: 1.00915384293

File 1. Length is 128 bits: 0.00973892211914

File 2. Length is 128 bits: 0.260397911072

File 3. Length is 128 bits: 0.562474012375

File 1. Length is 256 bits: 0.00766682624817

File 2. Length is 256 bits: 0.20038819313

File 3. Length is 256 bits: 0.403558015823

File 1. Length is 512 bits: 0.00682401657104

File 2. Length is 512 bits: 0.170945167542

File 3. Length is 512 bits: 0.377629995346

File 1. Length is 1024 bits: 0.00849485397339

File 2. Length is 1024 bits: 0.210194826126

File 3. Length is 1024 bits: 0.426136016846

File 1. Length is 2048 bits: 0.0114049911499

File 2. Length is 2048 bits: 0.338324069977

File 3. Length is 2048 bits: 0.745917081833

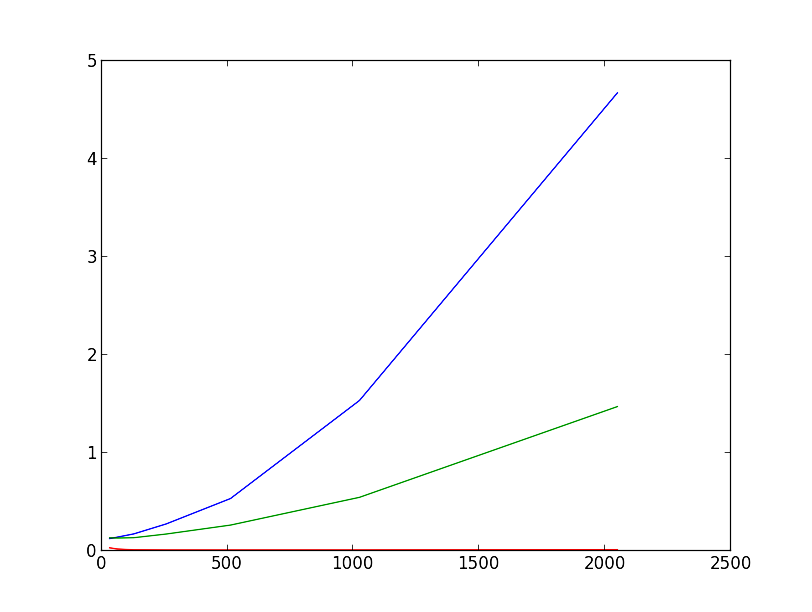
***Сравнительный анализ процесс шифрования, расшифрования.***

1 — шифрование

2 — расшифрование

3 — оптимальное расшифрование

*Сравнительный анализ процессов шифрования и расшифрования на файле размером 1072 символа.*

**File 1 ecrypt. Length is 64 bits: 0.0299260616302

File 1 decrypt. Length is 64 bits: 0.124141931534

File 1 decrypt\_opt. Length is 64 bits: 0.132165908813

File 1 ecrypt. Length is 128 bits: 0.0176479816437

File 1 decrypt. Length is 128 bits: 0.139548063278

File 1 decrypt\_opt. Length is 128 bits: 0.129881858826

File 1 ecrypt. Length is 256 bits: 0.00956606864929

File 1 decrypt. Length is 256 bits: 0.171954870224

File 1 decrypt\_opt. Length is 256 bits: 0.133167982101

File 1 ecrypt. Length is 512 bits: 0.00731801986694

File 1 decrypt. Length is 512 bits: 0.273061037064

File 1 decrypt\_opt. Length is 512 bits: 0.170399904251

File 1 ecrypt. Length is 1024 bits: 0.00707578659058

File 1 decrypt. Length is 1024 bits: 0.532961130142

File 1 decrypt\_opt. Length is 1024 bits: 0.261906862259

File 1 ecrypt. Length is 2048 bits: 0.00755214691162

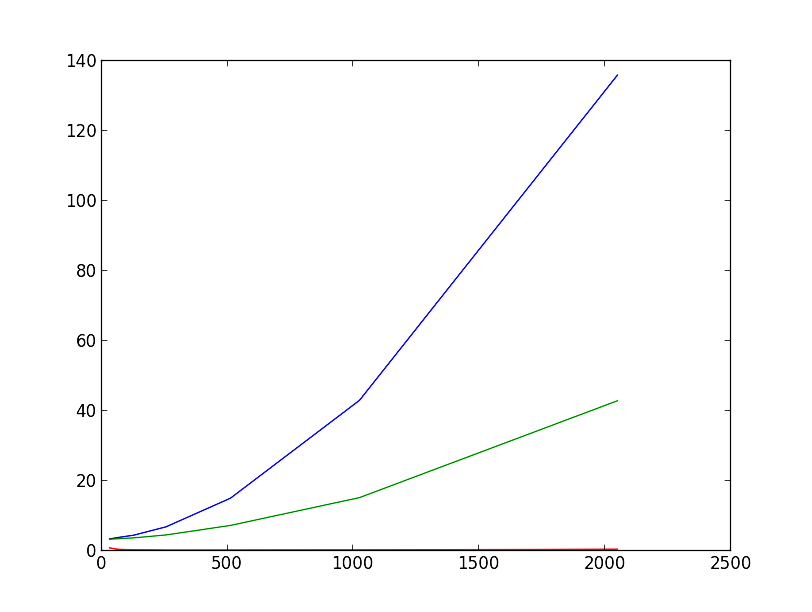
File 1 decrypt. Length is 2048 bits: 1.53342199326

File 1 decrypt\_opt. Length is 2048 bits: 0.545120954514

File 1 ecrypt. Length is 4096 bits: 0.0110700130463

File 1 decrypt. Length is 4096 bits: 4.67256903648

File 1 decrypt\_opt. Length is 4096 bits: 1.47117805481

*Сравнительный анализ процессов шифрования и расшифрования на файле размером 30016 символов.*

File 2 ecrypt. Length is 64 bits: 0.816342115402

File 2 decrypt. Length is 64 bits: 3.37858605385

File 2 decrypt\_opt. Length is 64 bits: 3.33498311043

File 2 ecrypt. Length is 128 bits: 0.433894872665

File 2 decrypt. Length is 128 bits: 3.77844595909

File 2 decrypt\_opt. Length is 128 bits: 3.41068196297

File 2 ecrypt. Length is 256 bits: 0.278341054916

File 2 decrypt. Length is 256 bits: 4.45240402222

File 2 decrypt\_opt. Length is 256 bits: 3.68621492386

File 2 ecrypt. Length is 512 bits: 0.175554990768

File 2 decrypt. Length is 512 bits: 6.8374710083

File 2 decrypt\_opt. Length is 512 bits: 4.51468205452

File 2 ecrypt. Length is 1024 bits: 0.185961961746

File 2 decrypt. Length is 1024 bits: 15.0281238556

File 2 decrypt\_opt. Length is 1024 bits: 7.23954582214

File 2 ecrypt. Length is 2048 bits: 0.237818956375

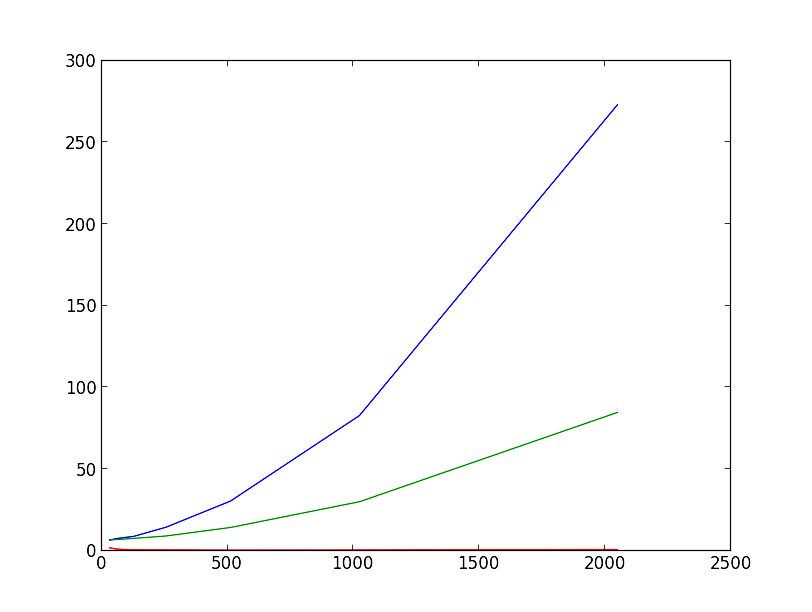
File 2 decrypt. Length is 2048 bits: 42.976388216

File 2 decrypt\_opt. Length is 2048 bits: 15.1831071377

File 2 ecrypt. Length is 4096 bits: 0.450448989868

File 2 decrypt. Length is 4096 bits: 135.894798994

File 2 decrypt\_opt. Length is 4096 bits: 42.8484909534

*Сравнительный анализ процессов шифрования и расшифрования на файлах размером 60032 символа.*

File 3 ecrypt. Length is 64 bits: 1.7978811264

File 3 decrypt. Length is 64 bits: 6.55669307709

File 3 decrypt\_opt. Length is 64 bits: 6.62704706192

File 3 ecrypt. Length is 128 bits: 0.987148046494

File 3 decrypt. Length is 128 bits: 7.57838797569

File 3 decrypt\_opt. Length is 128 bits: 6.90124988556

File 3 ecrypt. Length is 256 bits: 0.561812877655

File 3 decrypt. Length is 256 bits: 8.81195402145

File 3 decrypt\_opt. Length is 256 bits: 7.57791590691

File 3 ecrypt. Length is 512 bits: 0.472347021103

File 3 decrypt. Length is 512 bits: 14.4172840118

File 3 decrypt\_opt. Length is 512 bits: 9.04093289375

File 3 ecrypt. Length is 1024 bits: 0.341695070267

File 3 decrypt. Length is 1024 bits: 30.4248960018

File 3 decrypt\_opt. Length is 1024 bits: 14.252974987

File 3 ecrypt. Length is 2048 bits: 0.418030023575

File 3 decrypt. Length is 2048 bits: 82.7859561443

File 3 decrypt\_opt. Length is 2048 bits: 30.0108230114

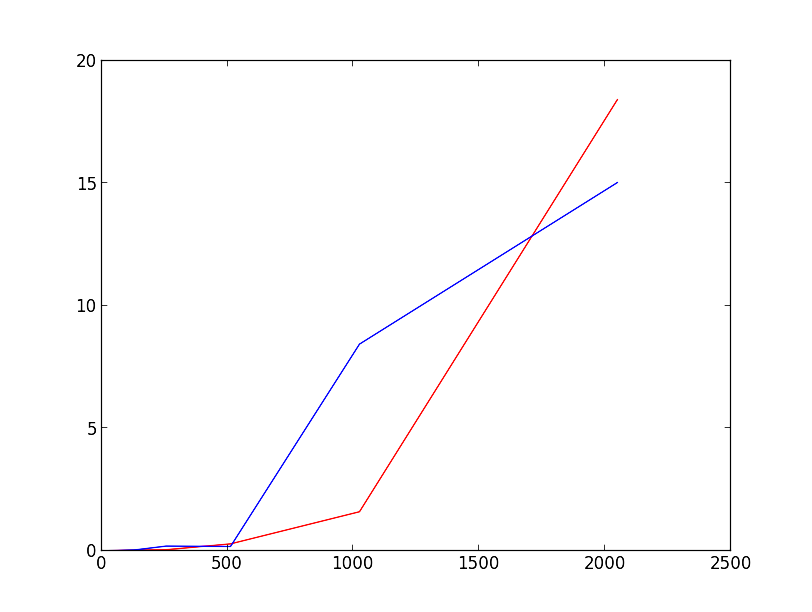
File 3 ecrypt. Length is 4096 bits: 0.772625923157

File 3 decrypt. Length is 4096 bits: 273.035149097

File 3 decrypt\_opt. Length is 4096 bits: 84.6766319275

**Работа с закладкой.**

Сравним скорость генерации простого RSA-ключа и скорость генерации ключа с закладкой.

В данном случае не использовался сложный генератор псевдослучайных чисел для создания мастер-ключа.

*1 — чистый RSA-ключ*

*2 — содержит в себе backdoor.*

*10 проходов.*

RSA-key length 64 generation: 0.00292160511017

Backdoor RSA-key length 64 generation: 0.00458030700684

RSA-key length 128 generation: 0.00881099700928

Backdoor RSA-key length 128 generation: 0.00861284732819

RSA-key length 256 generation: 0.019899892807

Backdoor RSA-key length 256 generation: 0.0291929960251

RSA-key length 512 generation: 0.0418857812881

Backdoor RSA-key length 512 generation: 0.189254617691

RSA-key length 1024 generation: 0.280648088455

Backdoor RSA-key length 1024 generation: 0.174038743973

RSA-key length 2048 generation: 1.59208962917

Backdoor RSA-key length 2048 generation: 8.4333637476

RSA-key length 4096 generation: 18.4147756815

Backdoor RSA-key length 4096 generation: 15.0294500113

**Выводы.**

При использовании алгоритма RSA для шифрования файлов необходимо учитывать, что размер зашифрованных файлов может сильно превосходить размер исходных файлов (всё зависит от размера выбранного блока). При шифровании больших файлов алгоритм работает крайне медленно, что является большим минусом при шифровании больших объёмов информации. Кроме того, является существенным минусом скорость генерации ключа. Небольшие ключи порядка 512-1024 бит генерируются быстро, но уже на данный момент не являются криптостойкими.

Большим минусом RSA является псевдослучайность чисел, которые используются при генерации ключа. К тому же RSA подвержен внедрению закладок со стороны удостоверяющего центра или иной организации, которая выдает ключи. Поэтому ключам, сгенерированным третьей стороной, невозможно полностью доверять.