Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчёт

по лабораторной работе №3

**Асимметричная криптография.**

**RSA**

|  |  |
| --- | --- |
| Выполнил:  студент группы 653502  Куликов А.Д. | Проверил:  Артемьев В.С. |

Минск 2019

# Постановка задачи

Реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритма RSA.

# Теоретическая справка

Как известно, в настоящее время одним из наиболее надежных и эффективных методов защиты информации является шифрование, представляющее собой метод преобразования информации в зашифрованный текст с той целью, чтобы доступ к ней смог получить лишь пользователь, у которого есть необходимый ключ для дешифровки.

Ранее информация шифровалась и расшифровывалась при помощи одного и того же криптографического ключа. Однако в этом случае существует проблема: как передать такой ключ получателю, чтобы он с его помощью смог расшифровать ваше сообщение? Он должен передаваться либо при личной встрече, либо по надежным защищенным каналам связи, что позволит предотвратить перехват ключа посторонними. Но это не всегда удобно и достаточно проблематично.

Данная проблема решается при помощи ассиметричной криптографии. В данном случае у пользователя имеется так называемая ключевая пара, состоящая из закрытого ключа (Private Key) и открытого ключа (Public Key). Открытый ключ предназначен для массового распространения — вы отправляете его другим пользователям, публикуете на открытых серверах ключей и т. д. для того чтобы все желающие могли зашифровать с его помощью сообщение для вас.

Интересно то, что после того, как сообщение зашифровано, расшифровать его сможет лишь владелец закрытого ключа, который находится в паре с открытым ключом, то есть только вы и никто другой, даже отправитель сообщения.

Это возможно по тому, что открытый и закрытый ключи связаны между собой по особой математической зависимости, но получить из открытого ключа закрытый не представляется возможным.

В основе ассиметричной криптографии лежит алгоритм Диффи-Хеллермана, двух ученых, сформулировавших модель криптографической системы с открытым ключом. Впоследствии три других ученых Р. Ривест, А. Шамир и Л. Адлеман создали ассиметричный алгоритм шифрования RSA (от первых букв фамилий его создателей), который сегодня получил повсеместное распространение и используется как для шифрования/дешифрования сообщений, так и для создания электронно-цифровых подписей.

Ассиметричная криптография сегодня еще известна как криптография на основе инфраструктуры открытых ключей (Public Key Infrastructure), она широко используется по всему миру как серьезными структурами и организациями (например, министерством обороны США), так и рядовыми пользователями.

## Алгоритм RSA

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

Криптосистема RSA стала первой системой, пригодной и для шифрования, и для цифровой подписи. Алгоритм используется в большом числе криптографических приложений, включая PGP, S/MIME, TLS/SSL, IPSEC/IKE и других.

Криптографические системы с открытым ключом используют так называемые односторонние функции, которые обладают следующим свойством:

– если известно x, то f(x) вычислить относительно просто;

– если известно y=f(x), то для вычисления x нет простого (эффективного) пути.

Под односторонностью понимается не теоретическая однонаправленность, а практическая невозможность вычислить обратное значение, используя современные вычислительные средства, за обозримый интервал времени.

В основу криптографической системы с открытым ключом RSA положена сложность задачи факторизации произведения двух больших простых чисел. Для шифрования используется операция возведения в степень по модулю большого числа. Для дешифрования (обратной операции) за разумное время необходимо уметь вычислять функцию Эйлера от данного большого числа, для чего необходимо знать разложение числа на простые множители.

В криптографической системе с открытым ключом каждый участник располагает как открытым ключом (англ. public key), так и закрытым ключом (англ. private key). В криптографической системе RSA каждый ключ состоит из пары целых чисел. Каждый участник создаёт свой открытый и закрытый ключ самостоятельно. Закрытый ключ каждый из них держит в секрете, а открытые ключи можно сообщать кому угодно или даже публиковать их. Открытый и закрытый ключи каждого участника обмена сообщениями в криптосистеме RSA образуют «согласованную пару» в том смысле, что они являются взаимно обратными, то есть:

Для любых допустимых пар открытого и закрытого ключей (p,s)

существуют соответствующие функции шифрования Ep(x) и расшифрования Ds(x) такие, что для любого сообщения m из M, где M — множество допустимых сообщений, m = Ds(Ep(m)) = Ep(Ds(m)).

RSA-ключи генерируются следующим образом:

1. Выбираются два различных случайных простых числа p и q заданного размера (например, 1024 бита каждое).
2. Вычисляется их произведение n=p\*q, которое называется модулем.
3. Вычисляется значение функции Эйлера от числа n:

phi(n)=(p-1)\*(q-1)

1. Выбирается целое число e (1<e<phi(n)), взаимно простое со значением функции phi (n). Число e называется открытой экспонентой (англ. public exponent). Обычно в качестве e берут простые числа, содержащие небольшое количество единичных бит в двоичной записи, например, простые из чисел Ферма: 17, 257 или 65537, так как в этом случае время, необходимое для шифрования с использованием быстрого возведения в степень будет меньше. Слишком малые значения e, например 3, потенциально могут ослабить безопасность схемы RSA.
2. Вычисляется число d, мультипликативно обратное к числу e по phi(n), то есть число, удовлетворяющее сравнению:

d\*e =1 (mod phi(n)). Число d называется секретной экспонентой. Обычно оно вычисляется при помощи расширенного алгоритма Евклида.

1. Пара (e,n) публикуется в качестве открытого ключа RSA (англ. RSA public key).
2. Пара (d,n) играет роль закрытого ключа RSA (англ. RSA private key) и держится в секрете.

# Блок-схема алгоритма

Предположим, Боб хочет послать Алисе сообщение m.

Сообщениями являются целые числа в интервале от 0 до n-1.

A screenshot of a cell phone

Description automatically generated

Алгоритм шифрования:

1. Взять открытый ключ (e,n) Алисы
2. Взять открытый текст m
3. Зашифровать сообщение с использованием открытого ключа Алисы: c=E(m)=me mod n

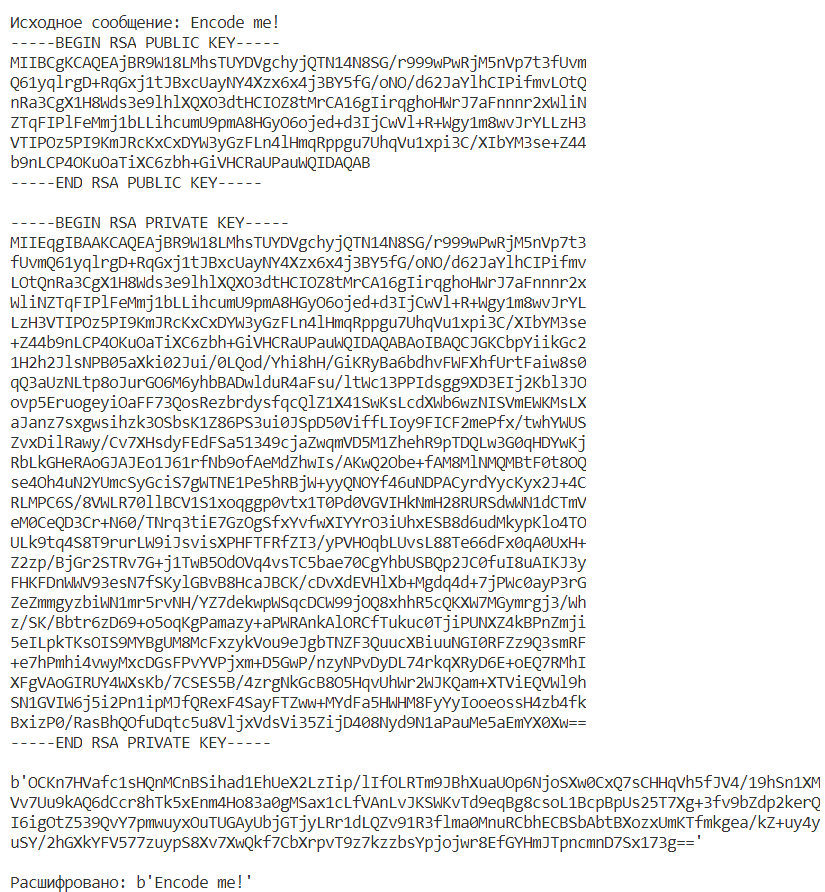
Алгоритм расшифрования:

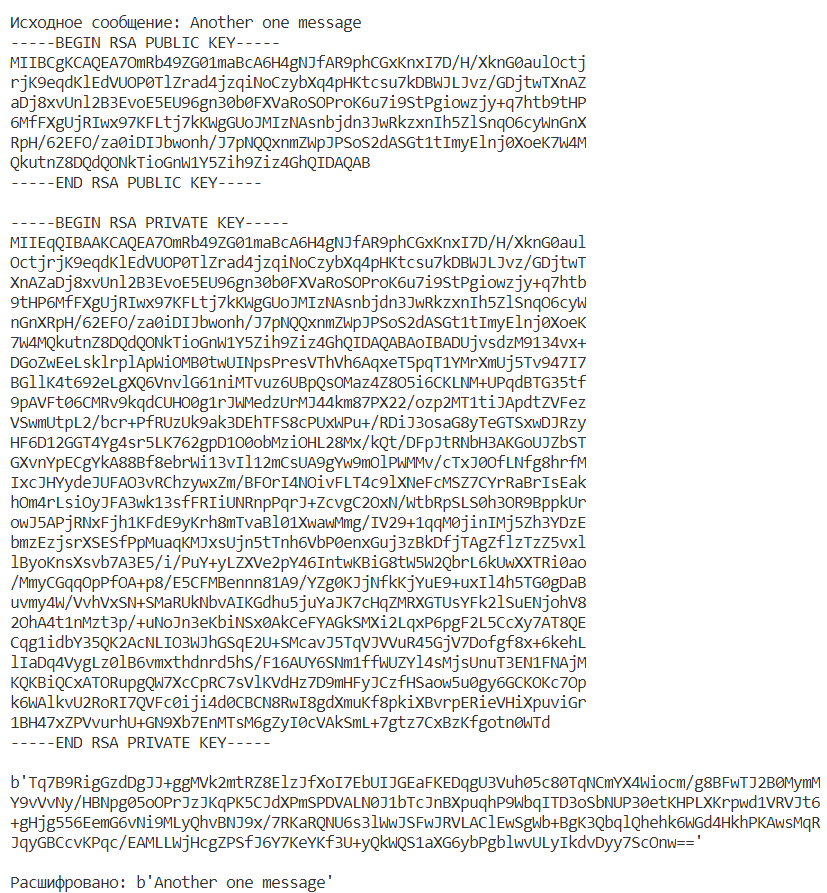
1. Принять зашифрованное сообщение c.
2. Взять свой закрытый ключ (d,n).
3. Применить закрытый ключ для расшифрования сообщения: m=D(c)= cd mod n.

Данная схема на практике не используется по причине того, что она не является практически надёжной (semantically secured). Действительно, односторонняя функция E(m) является детерминированной — при одних и тех же значениях входных параметров (ключа и сообщения) выдаёт одинаковый результат. Это значит, что не выполняется необходимое условие практической (семантической) надёжности шифра.

Наиболее используемым в настоящее время является смешанный алгоритм шифрования, в котором сначала шифруется сеансовый ключ, а потом уже с его помощью участники шифруют свои сообщения симметричными системами. После завершения сеанса сеансовый ключ, как правило, уничтожается.

# Результаты выполнения программы





# Приложение

**key.py**

import typing

import rsa.prime

import rsa.pem

import rsa.common

import rsa.randnum

import rsa.core

DEFAULT\_EXPONENT = 65537

class AbstractKey:

    \_\_slots\_\_ = ('n', 'e')

    def \_\_init\_\_(self, n: int, e: int) -> None:

        self.n = n

        self.e = e

    @classmethod

    def \_load\_pkcs1\_pem(cls, keyfile: bytes) -> 'AbstractKey':

        pass

    @classmethod

    def \_load\_pkcs1\_der(cls, keyfile: bytes) -> 'AbstractKey':

        pass

    def \_save\_pkcs1\_pem(self) -> bytes:

        pass

    def \_save\_pkcs1\_der(self) -> bytes:

        pass

    @classmethod

    def load\_pkcs1(cls, keyfile: bytes, format='PEM') -> 'AbstractKey':

        methods = {

            'PEM': cls.\_load\_pkcs1\_pem,

            'DER': cls.\_load\_pkcs1\_der,

        }

        method = cls.\_assert\_format\_exists(format, methods)

        return method(keyfile)

    @staticmethod

    def \_assert\_format\_exists(file\_format: str, methods: typing.Mapping[str, typing.Callable]) \

            -> typing.Callable:

        try:

            return methods[file\_format]

        except KeyError:

            formats = ', '.join(sorted(methods.keys()))

            raise ValueError('Unsupported format: %r, try one of %s' % (file\_format,

                                                                        formats))

    def save\_pkcs1(self, format='PEM') -> bytes:

        methods = {

            'PEM': self.\_save\_pkcs1\_pem,

            'DER': self.\_save\_pkcs1\_der,

        }

        method = self.\_assert\_format\_exists(format, methods)

        return method()

class PublicKey(AbstractKey):

    \_\_slots\_\_ = ('n', 'e')

    @classmethod

    def \_load\_pkcs1\_der(cls, keyfile: bytes) -> 'PublicKey':

        from pyasn1.codec.der import decoder

        from rsa.asn1 import AsnPubKey

        (priv, \_) = decoder.decode(keyfile, asn1Spec=AsnPubKey())

        return cls(n=int(priv['modulus']), e=int(priv['publicExponent']))

    def \_save\_pkcs1\_der(self) -> bytes:

        from pyasn1.codec.der import encoder

        from rsa.asn1 import AsnPubKey

        asn\_key = AsnPubKey()

        asn\_key.setComponentByName('modulus', self.n)

        asn\_key.setComponentByName('publicExponent', self.e)

        return encoder.encode(asn\_key)

    @classmethod

    def \_load\_pkcs1\_pem(cls, keyfile: bytes) -> 'PublicKey':

        der = rsa.pem.load\_pem(keyfile, 'RSA PUBLIC KEY')

        return cls.\_load\_pkcs1\_der(der)

    def \_save\_pkcs1\_pem(self) -> bytes:

        der = self.\_save\_pkcs1\_der()

        return rsa.pem.save\_pem(der, 'RSA PUBLIC KEY')

class PrivateKey(AbstractKey):

    \_\_slots\_\_ = ('n', 'e', 'd', 'p', 'q', 'exp1', 'exp2', 'coef')

    def \_\_init\_\_(self, n: int, e: int, d: int, p: int, q: int) -> None:

        AbstractKey.\_\_init\_\_(self, n, e)

        self.d = d

        self.p = p

        self.q = q

        self.exp1 = int(d % (p - 1))

        self.exp2 = int(d % (q - 1))

        self.coef = rsa.common.inverse(q, p)

    def blinded\_decrypt(self, encrypted: int) -> int:

        return rsa.core.decrypt\_int(encrypted, self.d, self.n)

    def blinded\_encrypt(self, message: int) -> int:

        return rsa.core.encrypt\_int(message, self.d, self.n)

    @classmethod

    def \_load\_pkcs1\_der(cls, keyfile: bytes) -> 'PrivateKey':

        from pyasn1.codec.der import decoder

        (priv, \_) = decoder.decode(keyfile)

        if priv[0] != 0:

            raise ValueError('Unable to read this file, version %s != 0' % priv[0])

        as\_ints = map(int, priv[1:6])

        key = cls(\*as\_ints)

        return key

    def \_save\_pkcs1\_der(self) -> bytes:

        from pyasn1.type import univ, namedtype

        from pyasn1.codec.der import encoder

        class AsnPrivKey(univ.Sequence):

            componentType = namedtype.NamedTypes(

                namedtype.NamedType('version', univ.Integer()),

                namedtype.NamedType('modulus', univ.Integer()),

                namedtype.NamedType('publicExponent', univ.Integer()),

                namedtype.NamedType('privateExponent', univ.Integer()),

                namedtype.NamedType('prime1', univ.Integer()),

                namedtype.NamedType('prime2', univ.Integer()),

                namedtype.NamedType('exponent1', univ.Integer()),

                namedtype.NamedType('exponent2', univ.Integer()),

                namedtype.NamedType('coefficient', univ.Integer()),

            )

        asn\_key = AsnPrivKey()

        asn\_key.setComponentByName('version', 0)

        asn\_key.setComponentByName('modulus', self.n)

        asn\_key.setComponentByName('publicExponent', self.e)

        asn\_key.setComponentByName('privateExponent', self.d)

        asn\_key.setComponentByName('prime1', self.p)

        asn\_key.setComponentByName('prime2', self.q)

        asn\_key.setComponentByName('exponent1', self.exp1)

        asn\_key.setComponentByName('exponent2', self.exp2)

        asn\_key.setComponentByName('coefficient', self.coef)

        return encoder.encode(asn\_key)

    @classmethod

    def \_load\_pkcs1\_pem(cls, keyfile: bytes) -> 'PrivateKey':

        der = rsa.pem.load\_pem(keyfile, b'RSA PRIVATE KEY')

        return cls.\_load\_pkcs1\_der(der)

    def \_save\_pkcs1\_pem(self) -> bytes:

        der = self.\_save\_pkcs1\_der()

        return rsa.pem.save\_pem(der, b'RSA PRIVATE KEY')

def find\_p\_q(nbits: int, getprime\_func=rsa.prime.getprime, accurate=True) -> typing.Tuple[int, int]:

    total\_bits = nbits \* 2

    # Make sure that p and q aren't too close or the factoring programs can

    # factor n.

    shift = nbits // 16

    pbits = nbits + shift

    qbits = nbits - shift

    # Choose the two initial primes

    p = getprime\_func(pbits)

    q = getprime\_func(qbits)

    def is\_acceptable(p, q):

        if p == q: return False

        if not accurate: return True

        # Make sure we have just the right amount of bits

        found\_size = rsa.common.bit\_size(p \* q)

        return total\_bits == found\_size

    # Keep choosing other primes until they match our requirements.

    change\_p = False

    while not is\_acceptable(p, q):

        # Change p on one iteration and q on the other

        if change\_p: p = getprime\_func(pbits)

        else: q = getprime\_func(qbits)

        change\_p = not change\_p

    # We want p > q as described on

    # http://www.di-mgt.com.au/rsa\_alg.html#crt

    return max(p, q), min(p, q)

def calculate\_keys\_custom\_exponent(p: int, q: int, exponent: int) -> typing.Tuple[int, int]:

    phi\_n = (p - 1) \* (q - 1)

    try:

        d = rsa.common.inverse(exponent, phi\_n)

    except rsa.common.NotRelativePrimeError as ex:

        raise rsa.common.NotRelativePrimeError(

            exponent, phi\_n, ex.d,

            msg="e (%d) and phi\_n (%d) are not relatively prime (divider=%i)" %

                (exponent, phi\_n, ex.d))

    if (exponent \* d) % phi\_n != 1:

        raise ValueError("e (%d) and d (%d) are not mult. inv. modulo "

                         "phi\_n (%d)" % (exponent, d, phi\_n))

    return exponent, d

def gen\_keys(nbits: int,

             getprime\_func: typing.Callable[[int], int],

             accurate=True,

             exponent=DEFAULT\_EXPONENT) -> typing.Tuple[int, int, int, int]:

    # Regenerate p and q values, until calculate\_keys doesn't raise a

    # ValueError.

    while True:

        (p, q) = find\_p\_q(nbits // 2, getprime\_func, accurate)

        try:

            (e, d) = calculate\_keys\_custom\_exponent(p, q, exponent=exponent)

            break

        except ValueError:

            pass

    return p, q, e, d

def newkeys(nbits: int, accurate=True, poolsize=1, exponent=DEFAULT\_EXPONENT) \

        -> typing.Tuple[PublicKey, PrivateKey]:

    if nbits < 16:

        raise ValueError('Key too small')

    if poolsize < 1:

        raise ValueError('Pool size (%i) should be >= 1' % poolsize)

    # Determine which getprime function to use

    if poolsize > 1:

        from rsa import parallel

        def getprime\_func(nbits):

            return parallel.getprime(nbits, poolsize=poolsize)

    else:

        getprime\_func = rsa.prime.getprime

    (p, q, e, d) = gen\_keys(nbits, getprime\_func, accurate=accurate, exponent=exponent)

    n = p \* q

    return (

        PublicKey(n, e),

        PrivateKey(n, e, d, p, q)

    )

\_\_all\_\_ = ['PublicKey', 'PrivateKey', 'newkeys']