Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт компьютерных наук и кибербезопасности
Высшая школа кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА № 1

«Основы криптосистемы RSA»

по дисциплине «Криптографические методы защиты информации»

Выполнил
студент гр. 5151004/90101 Кондачков Е.Д.

Преподаватель
Ассистент Ярмак А.В.

Санкт-Петербург
2023

Оглавление

1. Цель работы

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

2. Задачи работы

1. Изучить описание криптосистемы RSA приведенное в методическом пособии к лабораторной работе;

2. Реализовать алгоритмы зашифрования и расшифрования;

3. Проверить работоспособность разработанного приложения и привести в отчете скриншоты зашифрованного сообщения и используемых параметров RSA;

4. Реализовать алгоритмы формирования и проверки цифровой подписи сообщения;

5. Провести тестирование и приложить к отчету скриншоты файла подписи.

3. Ход работы

1.1 Теоретическая часть

В соответствии с поставленными в предыдущем пункте задачами первым делом были изучены методические указания к лабораторной работе, в которых содержалась информация о построении криптосистемы RSA.

Параметрами системы является 5 основных значений, а также два ключа: открытый и закрытый. Таким образом, к параметрам относятся:

– простые числа p и q порядка 2048-4096 бит, которые в будущей реализации выбираются случайно;

– число $n = pq$;

– число e, называемое открытым показателем, которое ищется как число взаимно простое с $\phi(n) = (p-1)(q-1)$;

– число d, называемое закрытым показателем, которое вычисляется из сравнения $d \equiv e^{-1}(mod\ \phi(n))$;

– открытый ключ (public key, ключ зашифрования), получаемый отправителем и состоящий из пары (n, e);

– закрытый ключ (private key, ключ расшифрования), не разглашаемый получателем и состоящий из пары (n, d).

Эти параметры ложатся в основу криптосистемы RSA в независимости от того, зашифровываем мы или формируем подпись.

Алгоритм зашифрования и расшифрования RSA:

1. Отправитель зашифровывает открытый текст m:

$$c \equiv m^e (mod\ n)$$

2. Получатель расшифровывает зашифрованный текст c:

$$m \equiv c^d (mod\ n)$$

Данное шифрование является визуально простым и понятным, но этот подход занимает довольно много времени для больших текстов. По этой причине в лабораторной работе будет использоваться шифр AES, а его ключ, передаваемый вместе с сообщением будет зашифрован при помощи RSA.

Помимо шифрования в данной работе будет также использоваться

электронная подпись основанная на RSA. Её алгоритм таков:

1. Отправитель подписи генерирует хэш-значение h текста m (в этой работе будет использоваться алгоритм SHA256);

2. Генерируется число a, соответствующее значению хеша из предыдущего пункта;

3. Генерируется значение s: $s = h^d (mod\ n)$;

4. Полученное значение отправляется вместе с текстом m;

5. На стороне получателя подписи получается значение h' от полученного текста;

6. Производится проверка $h' = s^e (mod\ n)$.

1.2 Генерация параметров криптосистемы

Как уже было сказано ранее, первичные параметры p и q генерируются случайным образом при помощи функции $getPrime()$. Позднее из этих значений получается n и $\phi(n)$. В дальнейшем при помощи функций GCD и $inverse$ получаются e и d. Полный код генерации базовый параметров представлен на рисунке 1. Пример же сгенерированных параметров представлен на рисунке 2

```
def init_keys():
    global rsa_pq
    # choosing p and q
    p = getPrime(1024, randfunc=get_random_bytes)
    q = getPrime(1024, randfunc=get_random_bytes)
    rsa_pq = (p, q)
    e = 0
    #init base parameters
    n = p * q
    v = (p-1)*(q-1)
    while GCD(e, v) > 1:
        e = getPrime(1024, randfunc=get_random_bytes)
    d = inverse(e, v)

    return (n, e), (n, d)
```

Рисунок 1 – Функция генерации параметров криптосистемы

```
In this start RSA cryptosystem using this patameters:

p = 167734144587429909058705489444703443697743271808512367825940282165070040162901630135164190
979096032135568580305199937011566125222586598579875137582220846106524789822841581749368403887 2
767918659507995395309635493948470446135695055621049066427257742170058561066421972026496669535 5
366119568282954007612982353370 7

q = 957425882162791912135485822350950826738318832406091887773404952721621576058226872067316619
921615615675040328514982415177679808111318765358020047203897716827789762004227850038051771684 1
058869167741305072352982345306982944601514584814625909784744897289010573882778277389772400174 6
937522802444446593299152248313

n = 160593011350441368921508657587250900493755478374382057561131467795099122373442506408153300
016249422829434901838598322830908011838343724118311934220825581258262949921785225076472206361 7
864215742237550482859269927562840759961183034460980848602632988277928517302704847297483158512 6
274246602635231513032193038362572035930399539729812161651344656576564990369850724601723751682
422048746188046261712846600749770720903800203061236868603578471228979018188748087238425976399 2
021429214556569252692078991026629969697570533617288169088595144598993166344109981918238468590 8
578860403869241574272795283694945567270704031387589386291

e = 169275371833834604606177624885622974850911424403206469404606326154652410425981471514265307
774324965047746164501103627631212214626385547227397265575843531091300044954616759708055641609 7
038491117784640267555312140968804971445641125711349873280088839984434996192504349926852224996 8
015298144505801307431638784617 3

d = 883232371377627764955691967215047407664073869878378881586729646305137358636577352960873 8
923367883966763322795259414603267659677755807883546158473743339945689380542064574380646754147 1
705441628668974135513039404747305737951896509922399965015783873433833958614079045169253380843 3
568772510094937191797006321510101779181803258158645947984479767362966430717272652156643904493 5
768552233763254573985909544388953192380823901179364241638915679729329550199911608452230032027 1
157194024496728074133694595935274647925134295081490971816906851267873778644393118219265552255 3
145659216696620521400566002236982685909887525365096312 05
```

Рисунок 2 – Пример сгенерированных параметров системы

1.3 Реализация алгоритмов шифрования

Алгоритм шифрования и расшифрования описаны в теоретической части. В этом пункте стоит показать реализацию данных функций и результат, даваемый функцией зашифрования (структуру ASN-файла, который должен передаваться получателю).

Как и было указано ранее, текст шифруется при помощи алгоритма AES, что продемонстрировано на рисунке 3.

```python
def encrypt_text(text):
    aes_cipher = AES.new(key=aes_key,
                         mode=AES.MODE_CBC,
                         iv=init_vector)

    encrypted_text = aes_cipher.encrypt(pad(text, 16))

    return encrypted_text, aes_key
```

Рисунок 3 – Шифрование текста алгоритмом AES

Ключ AES шифруется при помощи RSA, как на рисунке 4.

```python
def encrypt_key(open_key):
    open_key_bytes = int.from_bytes(open_key, byteorder="big")
    encrypted_key = pow(open_key_bytes, public_key[1], public_key[0])
    return encrypted_key
```

Рисунок 4 – Шифрование ключа при помощи RSA

В итоге после зашифрования получается файл, указанный на рисунке 5.

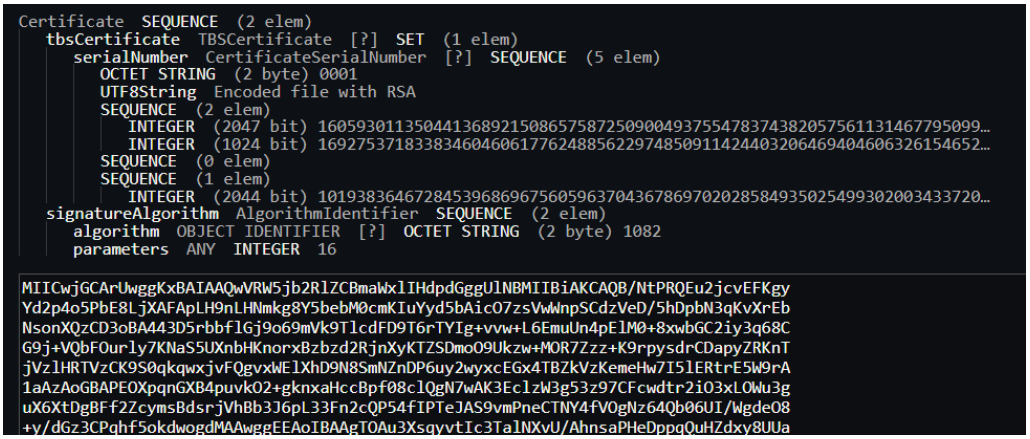

Рисунок 5 – ASN файл, полученный после зашифрования текста

В указанном выше ASN файле стоит уточнить значение полей типа $INTEGER$. Данные поля содержат (сверху вниз соотвественно):

1. Число n;
2. Число e;
3. Зашифрованный ключ AES.

Шифрование происходит в обратном порядке идентичными методами, поэтому их реализация здесь показана не будет

1.4 Реализация алгоритмов формирования и проверки электронной цифровой подписи

Как и в случаен с алгоритмами шифрования, описание формирования и проверки подписи было дано ранее. По этой причине в этой части так же, как и в предыдущей, буду представлены фактические реализации методов формирования и проверки, а также вид файла подписи.

Функция формирования выглядит как на рисунке 6.

```
def form_sign(text):
    hash = SHA256.new(text)
    int_hash = int(hash.hexdigest(),16)
    sign = pow(int_hash, private_key[1], private_key[0])
    return sign
```

Рисунок 6 – Функция формирования подписи

Файл, получаемый в ходе формирования подписи, выглядит подобно примеру на рисунке 7.

```
Certificate  SEQUENCE  (2 elem)
   tbsCertificate  TBSCertificate  [?]  SET  (1 elem)
      serialNumber  CertificateSerialNumber  [?]  SEQUENCE  (5 elem)
         OCTET STRING  (2 byte) 0040
         UTF8String  EDS with RSA
         SEQUENCE  (2 elem)
            INTEGER  (2047 bit) 160593011350441368921508657587250900493755478374382057561131467795099…
            INTEGER  (1024 bit) 169275371833834604606177624885622974850911424403206469404606326154652…
         SEQUENCE  (0 elem)
         SEQUENCE  (1 elem)
            INTEGER  (2046 bit) 562831666750333406208493954906718655805005322239008099192221162680020…
   signatureAlgorithm  AlgorithmIdentifier  SEQUENCE  (0 elem)
```

```
MIICsjGCAqwwggKoBAIAQAwMRURTIHdpdGggUlNBMIIBiAKCAQB/NtPRQEu2jcvEFKgyYd2p4o5PbE8L
jXAFApLH9nLHNmkg8Y5bebM0cmKIuYyd5bAicO7zsVwWnpSCdzVeD/5hDpbN3qKvXrEbNsonXQzCD3oB
A443D5rbbflGj9o69mVk9TlcdFD9T6rTYIg+vvw+L6EmuUn4pElM0+8xwbGC2iy3q68CG9j+VQbFOurl
y7KNaS5UXnbHKnorxBzbzd2RjnXyKTZSDmoO9Ukzw+MOR7Zzz+K9rpysdrCDapyZRKnTjVzlHRTVzCK9
S0qkqwxjvFQgvxWElXhD9N8SmNZnDP6uy2wyxcEGx4TBZkVzKemeHw7I5lERtrE5W9rA1aAzAoGBAPEO
XpqnGXB4puvkO2+gknxaHccBpf08clQgN7wAK3EclzW3g53z97CFcwdtr2iO3xLOWu3guX6XtDgBFf2Z
cymsBdsrjVhBb3J6pL33Fn2cQP54fIPTeJAS9vmPneCTNY4fVOgNz64Qb06UI/WgdeO8+y/dGz3CPqhf
5okdwogdMAAwggEEAoIBACyVuSH0J4LvMSSO5DFTdhna/z90XN1KnMRSfg9j+AM+5KIYo89LQAsdt9pA
```

Рисунок 7 – Файл с подписью

Процедура проверки подписи показана на рисунке 8.

```python
def check_sign( sign, text):
    hash = int(SHA256.new(text).hexdigest(), 16)%public_key[0]
    checking_hash = pow(sign, public_key[1], public_key[0])
    if hash == checking_hash:
        print('Подпись принимается')
    else:
        print('Подпись неверна')
```

Рисунок 8 – Процедура проверки подписи

4. Ответы на контрольные вопросы

1. Какие задачи положены в основу безопасности системы RSA?

В основу безопасности системы RSA положена задача факторизации большого простого числа. Зная открытый ключ (n, e) и разложение показателя $n = pq$, злоумышленник сможет рассчитать функцию Эйлера $\varphi(n)$, затем вычислить секретный показатель $d \equiv e^{-1}(mod\ \varphi(n))$ и расшифровать сообщение $m \equiv cd\ (mod\ n)$.

2. Показать, что схема RSA работает корректно для любого сообщения $m \in Z/nZ$.

При подборе параметров криптосистемы RSA требуется выполнение равенства:

$$ed \equiv 1(mod\ \varphi(n)).$$

То есть это означает, что числа e и d являются взаимно простыми по модулю функции Эйлера $\varphi(n) = (p-1)(q-1)$.

Тогда существует коэффициент k, при котором справедливо равенство:

$$ed = 1 + k\ \varphi(n).$$

В схеме RSA расшифрование будет иметь следующий вид:

$$m = m^{ed} = m^{1+k\varphi(n)} = m\left(m^{k(p-1)(q-1)}\right) = m\left(m^{(p-1)}\right)^{k(q-1)}$$
$$\equiv m * 1^{k(q-1)}(mod\ p)$$

Аналогично:

$$m \equiv m^{ed}(mod\ q)$$

По КТО:

$$m \equiv m^{ed}(mod\ n)$$

3. Доказать, что задача разложения числа n на множители и задача вычисления функции Эйлера $\phi(n)$ полиномиально эквивалентны.

Пускай существует разложение $n = pq$, где p и q – простые числа. Тогда функция Эйлера вычисляется следующим образом:

$$\varphi(n) = (p-1)(q-1)$$

Пускай известны значения n и $\varphi(n)$. Тогда будет существовать система

уравнения следующего вида:

$$\begin{cases} n = pq; \\ \varphi(n) = (p-1)(q-1) = n - (p+q) + 1; \end{cases}$$

$$\begin{cases} pq = n; \\ p + q = n - \varphi(n) + 1; \end{cases}$$

Тогда p и q являются корнями некоторого квадратичного уравнения (по т. Виета) следующего вида:

$$\mathrm{x}^2 - (\mathrm{n} - \varphi(\mathrm{n}) + 1)\mathrm{x} + \mathrm{n} = 0$$

$$q = \frac{-(n - \varphi(n) + 1) \pm \sqrt{D}}{2}, \text{где } D = (n - \varphi(n) + 1)^2 - 4n$$

Отсюда:

$$p = n - \varphi(n) + 1 - \frac{-(n - \varphi(n) + 1) \pm \sqrt{D}}{2}$$

Для вычисления p и q по известным n и $\varphi(n)$ в большинстве используются «быстрые» операции – сложение и деление на 2, и только по одному разу происходит вычисление квадратного корня и возведение в квадрат. Однако из-за однократности этих операций ими можно пренебречь. В свою очередь, при вычислении функции Эйлера также используются операции произведения и сложения.

Таким образом, в силу использования одинаковых по сложности операций при вычислении функции Эйлера и при вычислении множителей числа можно говорить о том, что данные операции являются полиномиально эквивалентными.

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

Гомоморфизм определяется следующим образом:

$$\varphi: G \to G^*:$$

$$\varphi(a \circ b) = \varphi(a) * \varphi(b)$$

В случае RSA $\varphi = m^e (mod\ n)$, где m – исходное сообщение, e – открытый показатель. Тогда:

$$\varphi(m_1) * \varphi(m_2) = m_1^e * m_2^e (mod\ n) = (m_1 * m_2)^e (mod\ n) = \varphi(m_1 * m_2)$$

5. Выводы и заключение

В ходе работы была изучена криптосистема RSA. Как было выявлено, система не подходит для шифрования больших сообщений в связи с сложностью операции возведение в степень по модулю. В связи с этим связывание RSA с другими алгоритмами шифрования выглядит мудро.

Сама криптосистема проста в реализации, что определенно является её преимуществом. Но этот признак вызывает и дополнительные опасности при использовании системы. Например, при нахождении e и d может возникнуть ситуация, когда один из параметров является недостаточно большим и может быть подобран.

Другая проблема или, вернее, особенность системы – необходимость наличия у всех абонентов различных n, чтобы обеспечить невозможность нахождения параметров одного абонента, используя параметры другого.

Данная криптосистема, как и многие другие не лишена преимуществ и недостатков. В следствие своего раннего появления, её можно и нужно считать хорошей опорой многих алгоритмов.

Приложение А

Листинг разработанной программы на языке Python3:

```
from rsa import RSA
from asn import ASN
from Crypto.Cipher import AES
from Crypto.Util.number import getPrime, inverse, GCD
from Crypto.Random import get_random_bytes
from Crypto.Hash import SHA256
from Crypto.Util.Padding import pad, unpad

# init keys
public_key = (0, 0)
private_key = (0, 0)
aes_key = get_random_bytes(32)
init_vector = get_random_bytes(16)
rsa_pq = (0, 0)



def get_text(filename = None):

    if filename == None:
        filename = input("Enter the path to the file: ")
    text = b''
    with open(filename, "rb") as file:
        for line in file:
            text += line

    return text, filename


#------------------ENCRYPTION BLOCK(#)------------------#
def encrypt_text(text):
    aes_cipher = AES.new(key=aes_key,
                         mode=AES.MODE_CBC,
                         iv=init_vector)

    encrypted_text = aes_cipher.encrypt(pad(text, 16))

    return encrypted_text, aes_key

def encrypt_key(open_key):
    open_key_bytes = int.from_bytes(open_key, byteorder="big")
    encrypted_key = pow(open_key_bytes, public_key[1], public_key[0])
    return encrypted_key

def encrypt_file(): # #
    text, filename = get_text()
    encrypted_text, open_key = encrypt_text(text)
    encrypted_key = encrypt_key(open_key)
    asn_text = ASN.encrypt_rsa_cipher(b'\x00\x01',
                b'Encoded file with RSA',
                public_key[0],
                public_key[1],
```

```
            encrypted_key,
            b'\x10\x82',
            len(encrypted_text),
            encrypted_text)
    with open("#" + filename, "wb") as enc:
        enc.write(asn_text)


#------------------DECRYPTION BLOCK(~)------------------#
def decrypt_key(encrypted_key):
    open_key_bytes = pow(encrypted_key, private_key[1], private_key[0])
    open_key = open_key_bytes.to_bytes(32, byteorder="big")
    return open_key

def decrypt_text(text, key):
    aes_cipher = AES.new(key=key,
                         mode=AES.MODE_CBC,
                         iv=init_vector)

    decrypted_text = aes_cipher.decrypt(text)
    decrypted_text = unpad(decrypted_text, 16)

    return decrypted_text

def decrypt_file():
    text, filename = get_text()
    _, _, encrypted_key, encrypted_text = ASN.decrypt_rsa_cipher(text)
    open_key = decrypt_key(encrypted_key)
    open_text = decrypt_text(encrypted_text, open_key)

    with open("~" + filename, "wb") as dec:
        dec.write(open_text)


#------------------CREATING BLOCK(^)------------------#
def form_sign(text):
    hash = SHA256.new(text)
    int_hash = int(hash.hexdigest(),16)
    sign = pow(int_hash, private_key[1], private_key[0])
    return sign

def create_signature():# ^
    text, filename = get_text()
    sign = form_sign(text)
    asn_text = ASN.encrypt_rsa_eds(b'\x00\x40',
            b'EDS with RSA',
            public_key[0],
            public_key[1],
            sign)
    with open("^" + filename, "wb") as enc:
        enc.write(asn_text)


#------------------VERIFYING BLOCK($)------------------#
def check_sign( sign, text):
    hash = int(SHA256.new(text).hexdigest(), 16)%public_key[0]
```

```
    checking_hash = pow(sign, public_key[1], public_key[0])
    if hash == checking_hash:
        print('Подпись принимается')
    else:
        print('Подпись неверна')

def verify_signature():#
    text, filename = get_text()
    sign_asn, _ = get_text('^' + filename)
    _,_,sign = ASN.decrypt_rsa_eds(sign_asn)
    check_sign(sign, text)

#------------------VIEWING BLOCK------------------#
def view_parameters():
    print('''In this start RSA cryptosystem using this patameters:''')
    print(f'''
p = {rsa_pq[0]}\n
q = {rsa_pq[1]}\n
n = {public_key[0]}\n
e = {public_key[1]}\n
d = {private_key[1]}\n
            ''')



def init_keys():
    global rsa_pq
    # choosing p and q
    p = getPrime(1024, randfunc=get_random_bytes)
    q = getPrime(1024, randfunc=get_random_bytes)
    rsa_pq = (p, q)
    e = 0
    #init base parameters
    n = p * q
    v = (p-1)*(q-1)
    while GCD(e, v) > 1:
        e = getPrime(1024, randfunc=get_random_bytes)
    d = inverse(e, v)

    return (n, e), (n, d)

def main():
    global public_key, private_key
    public_key, private_key = init_keys()

    while True:
        print('''
Warning: You will not be able to decrypt the file or verify
the signature if the action was  not performed at the current start of the program.

Available fighters:
1. Robin Hood (encrypt file)
2. James Bond (decrypt file)
3. Spider-man (create signature)
4. Captain Nemo (verify signature)
5. Albert Einstein (view parameters)
```

```
''' )
        command = input("Choose your fighter: ")

        if command == '1' or command == 'Robin Hood':
            encrypt_file()
        elif command == '2' or command == 'James Bond':
            decrypt_file()
        elif command == '3' or command == 'Spider-man':
            create_signature()
        elif command == '4' or command == 'Captain Nemo':
            verify_signature()
        elif command == '5' or command == 'Albert Einstein':
            view_parameters()
        else:
            print("incorrect input, try again")


if __name__ == '__main__':
    main()
```