

Московский Государственный Университет  
Военный учебный центр  
ВУС «Защита информационных технологий»

Практическое задание  
Изучение механизмов функционирования систем электронной подписи

Фомин С.А.

Взвод 2231

Москва, март 2021

## Оглавление

Постановка задачи .....	3
Описание алгоритмов.....	3
Особенности программной реализации .....	4
Результаты тестирования программы.....	6
Руководство пользователя .....	7

## Постановка задачи

Требуется разработать и программно реализовать учебную систему электронной подписи (ЭП) на основе следующих криптографических алгоритмов (Вариант № 23):

1. Криптосистема RSA в режиме ЭП.
2. Хэш-функция SHA-1.

## Описание алгоритмов

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

RSA-ключи генерируются следующим образом:

1. Выбираются два различных случайных простых числа  $p$  и  $q$  заданного размера (например, 128 бита каждое);
2. Вычисляется их произведение  $n = pq$ , которое называется модулем;
3. Вычисляется значение функции Эйлера от числа  $n$ :

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

4. Выбирается целое число  $e$  ( $1 < e < \varphi(n)$ ), взаимно простое со значением функции  $\varphi(n)$ ;  
Число  $e$  называется открытой экспонентой;  
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;
5. Вычисляется число  $d$ , мультипликативно обратное к числу  $e$  по модулю  $\varphi(n)$ , то есть число, удовлетворяющее соответствующему сравнению:

$$ed = 1 \bmod \varphi(n)$$

(число  $d$  называется секретной экспонентой, обычно оно вычисляется при помощи расширенного алгоритма Евклида);

6. Пара  $(e, n)$  публикуется в качестве открытого ключа RSA;
7. Пара  $(d, n)$  играет роль закрытого ключа RSA и держится в секрете.

Шифрование сообщения  $m$  осуществляется по формуле:

$$c = m^e \bmod n$$

Расшифрование шифра  $c$  осуществляется по формуле:

$$m = c^d \bmod n$$

**SHA-1** (Secure Hash Algorithm 1) — алгоритм криптографического хеширования. Описан в RFC 3174. Для входного сообщения произвольной длины алгоритм генерирует 160-битное (20 байт) хеш-значение, называемое также дайджестом сообщения, которое обычно отображается как шестнадцатиричное число, длиной в 40 цифр. Используется во многих криптографических приложениях и протоколах.

SHA-1 реализует хеш-функцию, построенную на идее функции сжатия. Входами функции сжатия являются блок сообщения длиной 512 бит и выход предыдущего блока сообщения. Выход представляет собой значение всех хеш-блоков до этого момента. Иными словами хеш-блок  $M_i$  равен  $h_i = f(M_i, h_{i-1})$ . Хеш-значением всего сообщения является выход последнего блока. Функция  $f$  реализует многократные преобразования над битами входного числа.

## Особенности программной реализации

Для реализации алгоритмов был выбран язык программирования Python, так как он поддерживает работу с большими числами без установки дополнительных библиотек.

Для эффективной реализации алгоритма RSA необходимо реализовать несколько дополнительных алгоритмов.

Алгоритм быстрого возведения в степень и расширенный алгоритм Евклида:

```
def _pow(self, a, n, *, mod):
    if n == 0:
        return 1
    if n & 0b1:
        return self._pow(a, n - 1, mod=mod) * a % mod
    tmp = self._pow(a, n // 2, mod=mod)
    return (tmp * tmp) % mod
```

```
def gcd_params(a, b):
    if a == 0:
        return b, 0, 1
    gcd, x, y = gcd_params(b % a, a)
    return gcd, y - (b // a) * x, x
```

## Алгоритм генерации простых чисел заданного размера на основе теста Миллера-Рабина:

```
def _pow_test(self, a, b, *, mod):
    s = 0
    t = b
    while not t & 0x1:
        s += 1
        t >>= 1

    tmp = self._pow(a, t, mod=mod)
    if tmp == 1:
        return True
    for _ in range(s):
        if tmp == b:
            return True
        tmp = tmp * tmp % mod
    return False

def RabinMillerWitness(self, witness, possible):
    """
    Return True if possible may be prime.
    Return False if it is composite.
    """
    return self._pow_test(witness, possible - 1, mod=possible)

def generate_prime(self, size):
    """Generate an integer of size bits that is probably prime."""
    steps = max(10 * size, 64)
    find_prime = False
    while not find_prime:
        possible_prime = randint(2 ** (size-1), 2 ** size - 1) | 0x1
        find_prime = True
        for small_prime in self.small_primes:
            if not possible_prime % small_prime:
                find_prime = False
                break
        else:
            for _ in range(steps):
                witness = randint(2, possible_prime) | 0x1
                if not self.RabinMillerWitness(witness, possible_prime):
                    find_prime = False
                    break
    return possible_prime

def encrypt(self, message_hash: int, e: int):
    """
    Encrypt number with RSA algorithm with (e, n) public key.

    :param message: number for encrypting
    :param e: first part of public key
    :return: encrypted number
    """
    return self._pow(message_hash, e, mod=self.n)

def decrypt(self, code: int, d: int):
    """
    Decrypt number with RSA algorithm with (d) private key.

    :param code: number for decrypting
    :param d: private key
    :return: decrypted number
    """
    return self._pow(code, d, mod=self.n)
```

Используя эти алгоритмы, реализовать алгоритм шифрования и расшифрования RSA довольно просто (функции *encrypt* и *decrypt*).

В реализации алгоритма хеширования SHA-1 цикл по блокам сообщения многократно производит битовые операции над числами и формирует итоговый хеш:

```
for block in self._get_block(message):
    w = []
    for i in range(16):
        w.append(int.from_bytes(block[i*4 : (i+1)*4], 'big'))
    for i in range(16, 80):
        w.append(self._left_cyclic_rotate(
            w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16],
            1
        ))

    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    for i in range(80):
        f = self.F[i // 20]
        k = self.K[i // 20]

        temp = (self._left_cyclic_rotate(a, 5)
            + f(b, c, d) + e + k + w[i]) & self.BASE
        e = d
        d = c
        c = self._left_cyclic_rotate(b, 30)
        b = a
        a = temp

    h0 = (h0 + a) & self.BASE
    h1 = (h1 + b) & self.BASE
    h2 = (h2 + c) & self.BASE
    h3 = (h3 + d) & self.BASE
    h4 = (h4 + e) & self.BASE

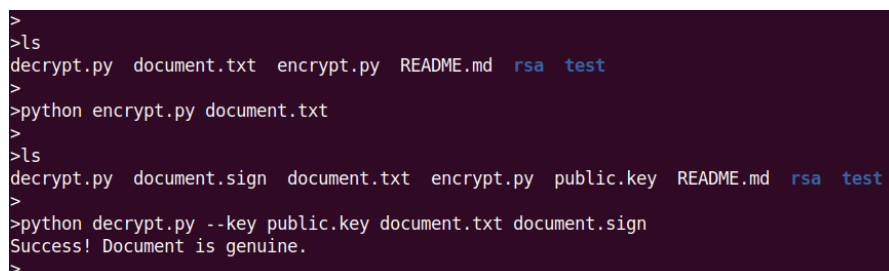
    binary_hash = f'{h0:032b}{h1:032b}{h2:032b}{h3:032b}{h4:032b}'
    return int(binary_hash, 2)
```

Полный код программы доступен в репозитории: <https://github.com/TotalChest/RSA>

## Результаты тестирования программы

Программа реализована в виде утилиты командной строки

Результаты работы можно видеть на следующих изображениях (положительный и отрицательный примеры):



```
>
>ls
decrypt.py document.txt encrypt.py README.md rsa test
>
>python encrypt.py document.txt
>
>ls
decrypt.py document.sign document.txt encrypt.py public.key README.md rsa test
>
>python decrypt.py --key public.key document.txt document.sign
Success! Document is genuine.
>
```

```

>
>ls
decrypt.py document.txt encrypt.py README.md rsa test
>
>python encrypt.py document.txt
>
>ls
decrypt.py document.sign document.txt encrypt.py public.key README.md rsa test
>
>echo "Hello World" > document.txt
>
>python decrypt.py --key public.key document.txt document.sign
Fail! Document is not genuine.
>

```

1. После работы программы *encrypt.py* в текущей директории появляются два новых файла: электронная подпись документа и открытый ключ.
2. После работы программы *decrypt.py* с переданными ей параметрами происходит проверка подлинности документа

## Руководство пользователя

Для зашифровки документа используется Python скрипт *encrypt.py* с переданным ему документом в качестве параметра командной строки

```

>python encrypt.py --help
usage: encrypt.py [-h] document

Sign a document with the digital signature (RSA).

positional arguments:
  document      Path to the document

optional arguments:
  -h, --help    show this help message and exit
>

```

Для расшифровки документа используется Python скрипт *decrypt.py*, в который нужно передать открытый ключ, документ и электронную подпись документа

```

>python decrypt.py --help
usage: decrypt.py [-h] --key KEY document signature

Check the digital signature of a document (RSA).

positional arguments:
  document      Path to the document
  signature      Path to the signature

optional arguments:
  -h, --help    show this help message and exit
  --key KEY      Path to the public key
>

```