Федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский Политехнический университет Петра Великого Физико-Механический институт

Лабораторная 1 – Кодирование

**«Вариант 3 – Алгоритм RSA»**

Выполнил студент гр. 5030102/20101: Комаров Е. Р. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Преподаватель: Новиков Ф. А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Работа принята: Дата \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Санкт-Петербург

2024

Содержание

1. Введение
2. Алгоритм шифрования
3. Алгоритм расшифровки
4. Алгоритм генерации ключей
   1. Алгоритм генерации простых чисел
   2. Алгоритм теста Миллера-Рабина
   3. Основной алгоритм
5. Интерфейс приложения
   1. Реализация
   2. Пример использования
   3. ручной пример работы алгоритма
   4. Ограничения
6. Заключение

Введение

Алгоритм RSA (названный по фамилиям его создателей — Ривест, Шамир и Адлеман) является одним из наиболее известных и широко применяемых асимметричных криптографических алгоритмов. Он был разработан в 1977 году и до сих пор остается основным методом шифрования, обеспечивающим безопасность передачи данных в интернете, например, в протоколе HTTPS.

В основе RSA лежат фундаментальные идеи теории чисел, такие как сложность факторизации больших чисел, что делает его особенно подходящим для применения в задачах шифрования и защиты информации.

Алгоритм шифрования

Сообщение M шифруется с использованием открытого ключа. Зашифрованное сообщение C вычисляется по формуле:

Здесь M — исходное сообщение, представленное в виде числа, а C — зашифрованный текст.

def mod\_exp(base, exp, mod):  
 result = 1  
 while exp > 0:  
 if exp % 2 == 1:  
 result = (result \* base) % mod  
 base = (base \* base) % mod  
 exp //= 2  
 return result

def encrypt(message, e, n):  
 message\_int = int.from\_bytes(message.encode('utf-8'), byteorder='big')  
 cipher = mod\_exp(message\_int, e, n)  
 return cipher

Листинг 1: Реализация шифрования на Python

Алгоритм расшифровки

Получатель с помощью закрытого ключа расшифровывает сообщение, используя следующую формулу:

Здесь d — закрытая экспонента, которая хранится в секрете.

def mod\_exp(base, exp, mod):  
 result = 1  
 while exp > 0:  
 if exp % 2 == 1:  
 result = (result \* base) % mod  
 base = (base \* base) % mod  
 exp //= 2  
 return result

def decrypt(cipher, d, n):  
 decrypted\_int = mod\_exp(cipher, d, n)  
 decrypted\_bytes = decrypted\_int.to\_bytes((decrypted\_int.bit\_length() + 7) // 8, byteorder='big')  
 return decrypted\_bytes.decode('utf-8')

Листинг 2: Реализация расширения на Python

Алгоритм генерации ключей

1. Алгоритм генерации простых чисел

def generate\_prime(bits):  
 while True:   
 prime\_candidate = random.getrandbits(bits)  
 prime\_candidate |= (1 << (bits - 1)) | 1  
  
 # Проверяем, является ли число простым  
 if miller\_rabin(prime\_candidate):  
 return prime\_candidate

Листинг 3: Реализация генерации простых чисел на Python

2. Алгоритм проверки Миллера-Рабина

def mod\_exp(base, exp, mod):  
 result = 1  
 while exp > 0:  
 if exp % 2 == 1:  
 result = (result \* base) % mod  
 base = (base \* base) % mod  
 exp //= 2  
 return result   
def miller\_rabin(n, k=5):  
 if n == 2 or n == 3:  
 return True  
 if n < 2 or n % 2 == 0:  
 return False  
  
 r, d = 0, n - 1  
 while d % 2 == 0:  
 d //= 2  
 r += 1  
  
 for \_ in range(k):  
 a = random.randint(2, n - 2)  
 x = mod\_exp(a, d, n)  
 if x == 1 or x == n - 1:  
 continue  
 for \_ in range(r - 1):  
 x = mod\_exp(x, 2, n)  
 if x == n - 1:  
 break  
 else:  
 return False  
 return True

Листинг 4: Реализация проверки Миллера-Рабина на Python

**Выбор k:**

* В реальных приложениях k обычно выбирается на уровне 5 или 10. Это даёт достаточно низкую вероятность ошибки для большинства практических задач, включая криптографию.
* Например, если k=5, вероятность ошибки составит ​, что весьма приемлемо.
* Для криптографических целей чаще выбирают большее k, например, 40 или 100, чтобы сделать вероятность ошибки настолько малой, что её можно игнорировать.

**Компромисс между точностью и производительностью:**

* С увеличением k, точность теста возрастает, но также возрастает и количество вычислений, поскольку каждое испытание включает в себя возведение в степень по модулю и другие вычисления.

3. Основной алгоритм

def mod\_exp(base, exp, mod):  
 result = 1  
 while exp > 0:  
 if exp % 2 == 1:  
 result = (result \* base) % mod  
 base = (base \* base) % mod  
 exp //= 2  
 return result  
  
def miller\_rabin(n, k=5):  
 if n == 2 or n == 3:  
 return True  
 if n < 2 or n % 2 == 0:  
 return False  
  
 r, d = 0, n - 1  
 while d % 2 == 0:  
 d //= 2  
 r += 1  
  
 for \_ in range(k):  
 a = random.randint(2, n - 2)  
 x = mod\_exp(a, d, n)  
 if x == 1 or x == n - 1:  
 continue  
 for \_ in range(r - 1):  
 x = mod\_exp(x, 2, n)  
 if x == n - 1:  
 break  
 else:  
 return False  
 return True  
  
def generate\_prime(bits):  
 while True:  
 prime\_candidate = random.getrandbits(bits)  
 prime\_candidate |= (1 << (bits - 1)) | 1  
  
 if miller\_rabin(prime\_candidate):  
 return prime\_candidate   
  
def mod\_inverse(e, phi):  
 old\_r, r = e, phi  
 old\_s, s = 1, 0  
  
 while r != 0:  
 quotient = old\_r // r  
 old\_r, r = r, old\_r - quotient \* r  
 old\_s, s = s, old\_s - quotient \* s  
  
 if old\_r > 1:  
 raise Exception("e и φ(n) не взаимно просты")  
 if old\_s < 0:  
 old\_s += phi  
  
 return old\_s  
  
  
def generate\_keys(bits=1024):  
 p = generate\_prime(bits)  
 q = generate\_prime(bits)  
  
 n = p \* q  
 phi\_n = (p - 1) \* (q - 1)  
  
 e = 65537  
  
  
 d = mod\_inverse(e, phi\_n)  
  
 return (n, e, d)

**Генерация ключей**:

* Генерируются два больших случайных простых числа p и q.
* Вычисляется модуль n=p×q, который используется и в открытом, и в закрытом ключе.
* Вычисляется функция Эйлера ϕ(n)=(p−1)×(q−1), которая используется для вычисления закрытого ключа.
* Выбирается открытая экспонента e, которая должна быть взаимно простой с ϕ(n). Обычно используется стандартное значение e=65537.
* Вычисляется закрытая экспонента d, которая является мультипликативно обратным числом к e по модулю ϕ(n), то есть e×d≡1mod  ϕ(n).

**Размер ключа** — **2048 бита**:

* Минимально рекомендованный размер для ключей RSA — 1024 бита. Для современных требований безопасности обычно используется 2048 или 3072 бит, однако 1024 бита до сих пор применяются для демонстрационных или учебных целей. В данной реализации мы используем ключ длиной 1024 бита для демонстрации работы алгоритма.

**Открытая экспонента e=65537**:

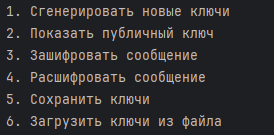
* Это стандартное значение для открытой экспоненты в RSA. e=65537— это небольшое простое число, которое оптимизирует вычисления при шифровании и достаточно велико, чтобы обеспечить безопасность. Применение более низких значений, таких как e=3, считается уязвимым для определённых атак. Значение e=65537 широко используется и обеспечивает хороший баланс между производительностью и безопасностью.

Интерфейс приложения

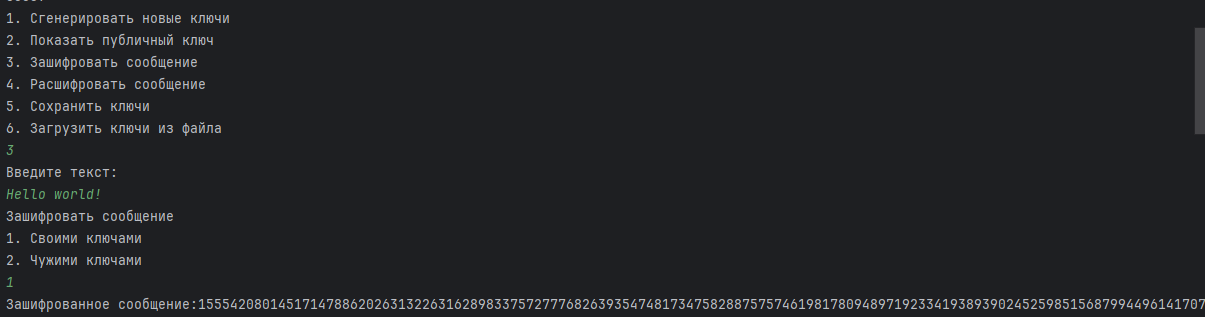
1. Реализация

import random  
  
def mod\_exp(base, exp, mod):  
 result = 1  
 while exp > 0:  
 if exp % 2 == 1:  
 result = (result \* base) % mod  
 base = (base \* base) % mod  
 exp //= 2  
 return result  
  
# Тест Миллера-Рабина для проверки числа на простоту  
def miller\_rabin(n, k=5):  
 if n == 2 or n == 3:  
 return True  
 if n < 2 or n % 2 == 0:  
 return False  
  
 # Представляем n-1 в виде d \* 2^r, где d нечетное  
 r, d = 0, n - 1  
 while d % 2 == 0:  
 d //= 2  
 r += 1  
  
 # Применяем тест k раз  
 for \_ in range(k):  
 a = random.randint(2, n - 2)  
 x = mod\_exp(a, d, n)  
 if x == 1 or x == n - 1:  
 continue  
 for \_ in range(r - 1):  
 x = mod\_exp(x, 2, n)  
 if x == n - 1:  
 break  
 else:  
 return False  
 return True  
  
# Генерация случайного простого числа  
def generate\_prime(bits):  
 while True:  
 # Генерируем случайное нечётное число с указанным количеством бит  
 prime\_candidate = random.getrandbits(bits)  
 prime\_candidate |= (1 << (bits - 1)) | 1  
  
 # Проверяем, является ли число простым  
 if miller\_rabin(prime\_candidate):  
 return prime\_candidate  
  
  
# Расширенный алгоритм Евклида для нахождения обратного по модулю  
def mod\_inverse(e, phi):  
 old\_r, r = e, phi  
 old\_s, s = 1, 0  
  
 while r != 0:  
 quotient = old\_r // r  
 old\_r, r = r, old\_r - quotient \* r  
 old\_s, s = s, old\_s - quotient \* s  
  
 if old\_r > 1:  
 raise Exception("e и φ(n) не взаимно просты")  
 if old\_s < 0:  
 old\_s += phi  
  
 return old\_s  
  
  
# Функция для генерации ключей  
def generate\_keys(bits=1024):  
 # Генерируем два случайных простых числа p и q  
 p = generate\_prime(bits)  
 q = generate\_prime(bits)  
  
 # Вычисляем n и функцию Эйлера  
 n = p \* q  
 phi\_n = (p - 1) \* (q - 1)  
  
 # Выбираем e (стандартное значение 65537)  
 e = 65537  
  
 # Находим d: d \* e ≡ 1 (mod φ(n))  
 d = mod\_inverse(e, phi\_n)  
  
 return (n, e, d)  
  
# Функция для шифрования  
def encrypt(message, e, n):  
 message\_int = int.from\_bytes(message.encode('utf-8'), byteorder='big')  
 cipher = mod\_exp(message\_int, e, n)  
 return cipher  
  
# Функция для расшифрования  
def decrypt(cipher, d, n):  
 decrypted\_int = mod\_exp(cipher, d, n)  
 decrypted\_bytes = decrypted\_int.to\_bytes((decrypted\_int.bit\_length() + 7) // 8, byteorder='big')  
 return decrypted\_bytes.decode('utf-8')  
  
# Основная программа  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Генерация ключей  
 n, e, d = generate\_keys(1024)  
 while True:  
 print("1. Сгенерировать новые ключи\n2. Показать публичный ключ\n3. Зашифровать сообщение \n4. Расшифровать сообщение\n5. Сохранить ключи\n6. Загрузить ключи из файла")  
  
 a = str(input())  
 if a=="1":  
 n, e, d = generate\_keys(1024)  
 elif a=="2":  
 print(str(n) + "\n" + str(e))  
 elif a=="3":  
 print("Введите текст:")  
 message =str(input())  
 print("Зашифровать сообщение\n1. Своими ключами \n2. Чужими ключами")  
 g=str(input())  
 if (g=="1"):  
 cipher=encrypt(message, e, n)  
 print("Зашифрованное сообщение:" + str(cipher))  
 elif (g=="2"):  
 print("Введите публичные ключи")  
 pubkey1=int(input())  
 pubkey2=int(input())  
 cipher = encrypt(message,pubkey2,pubkey1)  
 print("Зашифрованное сообщение:" + str(cipher))  
 else:  
 print("Вы неправильно ввели команду")  
 elif a=="4":  
 print("Введите зашифрованный текст: ")  
 message=int(input())  
 decrypted\_message = decrypt(message,d,n)  
 print("Расшифрованное сообщение:" + str(decrypted\_message))  
 elif a=="5":  
 file = open("key.txt", "w")  
 file.write(str(n) + "\n" + str(e) + "\n" +str(d) )  
 elif a=="6":  
 file= open("key.txt", "r")  
 k=0  
 for i in file:  
 if k==0:  
 n=int(i)  
 k=1  
 elif k==1:  
 e=int(i)  
 k=2  
 elif k==2:  
 d=int(i)  
 k=3  
 else:  
 print("Вы неправильно ввели команду\n")

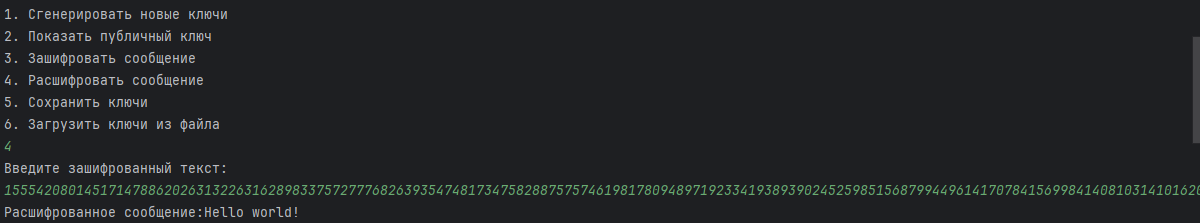
2.Пример Использования



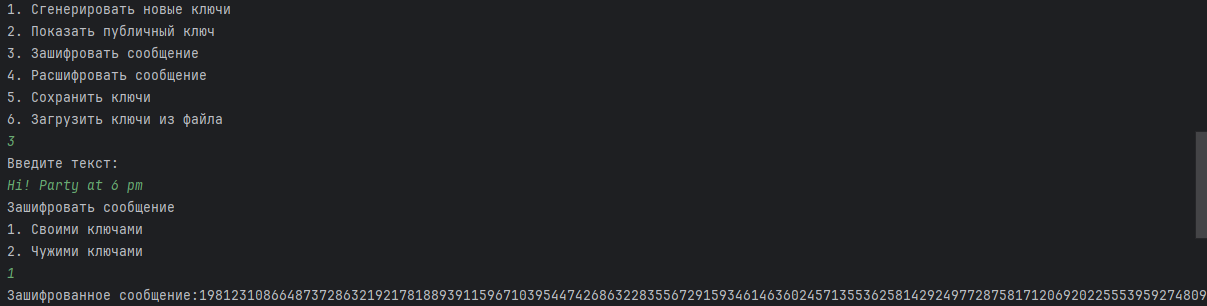
Изображение 1 – Пользовательский интерфейс



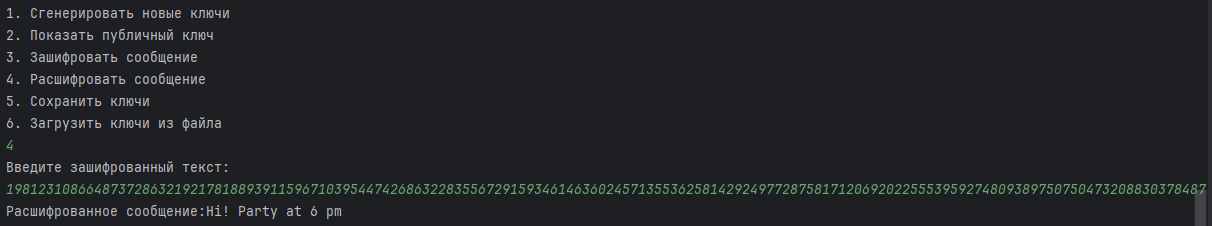
Изображение 2 – Пример 1, Работа Шифрования сообщения



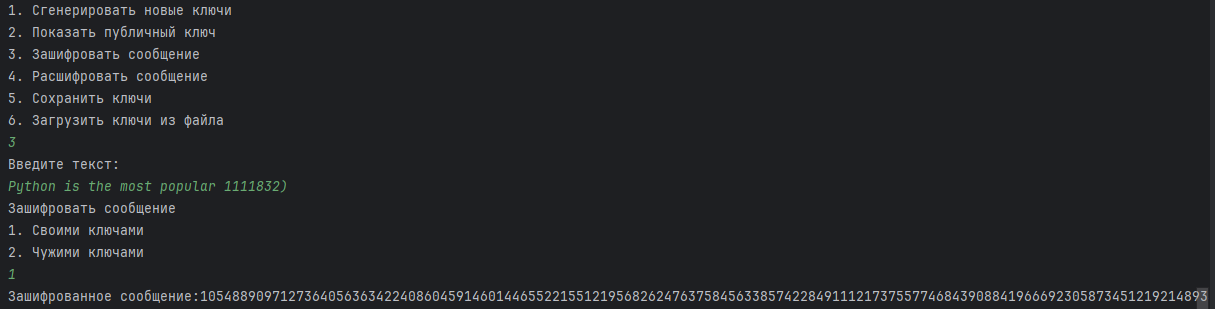
Изображение 3 – Пример 1, Работа Расшифровывания сообщения



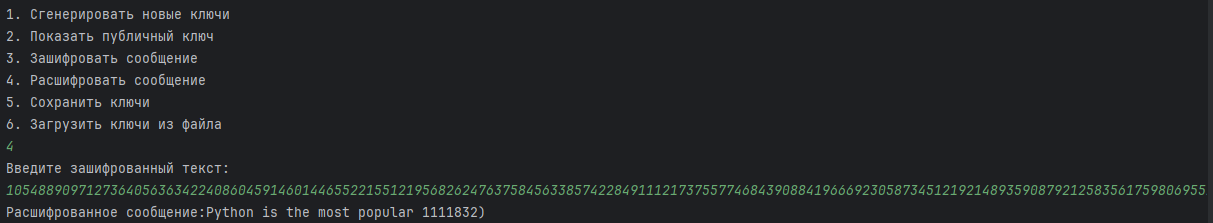
Изображение 4 – Пример 2, Работа Шифрования сообщения



Изображение 5 – Пример 2, Работа Расшифровывания сообщения



Изображение 6 – Пример 3, Работа Шифрования сообщения



Изображение 7 – Пример 3, Работа Расшифровывания сообщения

3. Ручной пример работы алгоритма, представленного выше.

**Шаг 1: Генерация ключей**

1. **Генерация простых чисел**: Мы генерируем два случайных простых числа p и q. Для простоты, допустим, что эти числа равны:
   * p=61
   * q=53
2. **Вычисление n**:

n=p×q=61×53=3233

1. **Вычисление функции Эйлера ϕ(n)**:

ϕ(n)=(p−1)×(q−1)=(61−1)×(53−1)=60×52=3120

1. **Выбор eee**:
   * По умолчанию, используется стандартное значение e=65537
2. **Проверка взаимной простоты e и ϕ(n)**: Необходимо убедиться, что gcd(65537,3120)=1. Поскольку они взаимно просты, продолжаем.
3. **Нахождение d (обратного элемента по модулю)**:
   * d находим с помощью расширенного алгоритма Евклида. В результате получаем:

d=2753

Теперь у нас есть ключи:

* Открытый ключ: (n=3233, e=65537)
* Закрытый ключ: d=2753

**Шаг 2: Шифрование сообщения**

Предположим, мы хотим зашифровать сообщение "Hi". Преобразуем его в число.

1. **Преобразование сообщения "Hi" в число**: В коде сообщение конвертируется в байты, а затем в целое число. Используя кодировку UTF-8:
   * "H" = 72
   * "i" = 105
   * Следовательно, message=18537(в 16-ричной системе это 0x48690).
2. **Шифрование с помощью открытого ключа**: Применяем алгоритм быстрого возведения в степень по модулю:

cipher=mod\_exp(18537,65537,3233)

Ручное вычисление:

1853765537 mod 3233=2201

Зашифрованное сообщение: cipher=2201.

**Шаг 3: Расшифрование сообщения**

Теперь расшифруем сообщение с помощью закрытого ключа d=2753.

1. **Расшифрование**:

decrypted\_message=mod\_exp(2201,2753,3233)

В результате вычислений получаем:

22012753 mod 3233=18537

1. **Преобразование числа обратно в текст**: Преобразуем 18537 обратно в байты:
   * 18537 — это 0x4869 в шестнадцатеричной системе, что соответствует символам "Hi" в кодировке UTF-8.

Расшифрованное сообщение: "Hi".

4. Ограничения

Защита от “дурака” в коде присутствует только на выбор режима работы. Основное ограничение присутствует на расшифровку сообщения и объём занимаемой памяти первоначального сообщения не должен превышать объём ключа, в моём примере это 256 байт, иначе происходит переполнение и результат не будет валидным. Для решения этой проблемы, при шифровании сообщения превышающего размер ключа, используют разбиение сообщения на части.

Заключение:

Алгоритм RSA стал основой для современной асимметричной криптографии и обеспечения безопасности передачи данных. Хотя его применение требует значительных вычислительных ресурсов и времени, он играет ключевую роль в обеспечении конфиденциальности и аутентификации в интернете. Текущие алгоритмы шифрования, используемые в браузерах, электронных кошельках, VPN и цифровых подписях, активно применяют RSA или его модификации для обеспечения безопасности в цифровом мире.