Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет безопасности информационных технологий

Дисциплина:

«Криптографические методы обеспечения информационной безопасности»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

«Асимметричные криптосистемы»

Выполнил:

Чу Ван Доан, студент группы N3347

(подпись)

Проверил:

Таранов Сергей Владимирович

(отметка о выполнении)

(подпись)

Санкт-Петербург

2025 г.

Содержание

Введение

Цель: изучить основные принципы работы асимметричных криптосистем на примере алгоритма RSA

Ход работы

1. Асимметричное шифрование

Асимметричное шифрование (asymmetric cryptography), также называемое шифрованием с открытым ключом (public key), представляет собой систему, использующую пару ключей для шифрования и дешифрования: открытый ключ (public key) используется для шифрования, а закрытый ключ (private key) — для дешифрования.

В этой системе любой может использовать открытый ключ для шифрования сообщения и отправки его получателю. Однако только владелец соответствующего закрытого ключа может расшифровать сообщение, поскольку закрытый ключ хранится в секрете.

Обычно при генерации пары ключей принимаются меры для того, чтобы восстановить закрытый ключ по открытому было чрезвычайно сложно (почти невозможно). Поэтому, даже если злоумышленник завладеет открытым ключом (что довольно просто), он не сможет получить закрытый ключ для дешифровки данных.

Тем не менее, всё не так идеально. Пока не существует алгоритма генерации ключей, который был бы полностью защищён от всех возможных атак. Современные алгоритмы могут противостоять взлому с использованием обычных персональных компьютеров, однако с развитием суперкомпьютеров и квантовых технологий асимметричное шифрование сталкивается с серьёзными вызовами.

2. Цифровая подпись

Цифровая подпись — это разновидность электронной подписи. Это вид данных, используемый для подтверждения подлинности других данных. Как это работает на практике — мы рассмотрим в следующем разделе.

Цифровая подпись использует асимметричную криптографию. В большинстве случаев она также может проверять целостность данных. Цифровая подпись во многом аналогична рукописной подписи, но процесс её настройки и использования значительно сложнее.

В следующей части мы рассмотрим, как цифровая подпись использует классический алгоритм асимметричного шифрования — RSA.

3. Алгоритм шифрования RSA

RSA — это асимметричный алгоритм шифрования, разработанный Роном Ривестом, Ади Шамиром и Леонардом Адлманом (название RSA — это аббревиатура, образованная от фамилий этих трёх авторов). Он широко используется в криптографии и технологии цифровой подписи.

В этой системе шифрования открытый ключ (public key) может быть свободно распространён среди всех. Работа алгоритма RSA основана на четырёх основных этапах: генерация ключей, распространение ключей, шифрование и расшифровка.

3.1. Генерация ключей

Основная идея генерации ключей в алгоритме RSA заключается в нахождении трёх случайных чисел e, d и n, таких что:

$$m^{ed} = m(mod)n$$

и важный момент заключается в том, что даже зная e и n, а также даже m, невозможно вычислить d.

Конкретно, ключи RSA генерируются следующим образом:

- Выбираются два простых числа p и q
- Вычисляется n = pq. Это значение будет использоваться как модуль как для открытого, так и для закрытого ключа.
- Вычисляется значение с использованием функции Кармайкла:

$\lambda(n) = НОК(\lambda(p), \lambda(q)), \lambda(q)) = НОК(p - 1, q - 1)$. Значение λ(n) сохраняется в секрете.

- Выбирается случайное число e из интервала (1, λ(n)), такое что НОД(e, λ(n) = 1, то есть ее и λ(n)— взаимно простые.
- Вычисляется число d, обратное к ее по модулю λ(n), т.е.

$d \equiv \frac{1}{e} (mod\, \lambda(n))$ или $de \equiv (mod\, \lambda(n))$

Открытый ключ — это пара (n, e), а закрытый ключ — пара (n, d). Закрытый ключ необходимо хранить в тайне, как и значения p и q, так как знание этих чисел позволяет легко восстановить ключи.

На практике часто выбирается фиксированное значение e для ускорения процесса шифрования и дешифрования, например e = 65537. Также можно использовать функцию Эйлера:

φ(n)=(p−1)(q−1)

Значение φ(n) кратно λ(n), поэтому значение d, удовлетворяющее

de≡1 (mod φ(n)) тоже удовлетворяет de ≡ 1 (mod λ(n)) Тем не менее, выбор φ(n) может быть менее безопасным, чем λ(n), если требуется высокий уровень безопасности.

3.2. Шифрование и дешифрование

В этом разделе мы рассмотрим, как выполняется шифрование с использованием открытого ключа (n, e) и дешифрование с использованием закрытого ключа (n, d).

Если у нас есть сообщение MM, его необходимо преобразовать в число m из диапазона (0, n), при этом m и n должны быть взаимно простыми. Это легко достигается с помощью методов добавления служебной информации (padding). Далее, мы шифруем m, получая c, следующим образом: $c = m^e \ (mod\ n)$

Затем зашифрованное значение c отправляется получателю.

На стороне получателя выполняется расшифровка c, чтобы восстановить m, по следующей формуле: $c^e \equiv m^{de} \equiv m \ (mod\ n)$

Из m затем можно восстановить исходное сообщение, удалив padding.

- Генерация ключей
- Выбираем два небольших простых числа:

  p = 3, q = 11
- Вычисляем n=p×q=3×11=33
- Вычисляем функцию Эйлера:

  φ(n)=(p−1)(q−1)=2×10=20
- Выбираем e = 3, такое что gcd(e,φ(n))=1
- Находим d, такое что:

  d≡e^{-1}mod φ(n)⇒d ≡ 7 mod 20 (так как 3×7 = 21 ≡ 1 mod 20)

-> Открытый ключ: (n=33,e=3)(n = 33, e = 3)

-> Закрытый ключ: (n=33,d=7)(n = 33, d = 7)

- Шифрование
- Предположим, мы хотим отправить сообщение m = 4
- Вычисляем шифртекст c по формуле:

  c≡m^emod n=4^3mod 33=64 mod 33=31

-> Шифртекст: c = 31

- Расшифровка

- Используем закрытый ключ d = 7 для расшифровки:

  $m \equiv c^d \bmod n = 31^7 \bmod 33$

  Вычисляем:

$31^7 \bmod 33$

$31 \equiv -2 \bmod 33$

$31^7 \equiv (-2)^7 mod\ 33$

$31^7 \equiv -128\ mod\ 33$

$-128\ mod\ 33 = 4$

Получаем исходное сообщение: m = 4

4. Моделирование алгоритма

4.1. Цели

- Реализация RSA
  - Генерация ключей (key generation) для RSA с двумя простыми числами (2-prime RSA, ключ 1024 бита).
  - Расширение до мульти-простого RSA (3-prime RSA).
- Оптимизация
  - Расшифровка с использованием китайской теоремы об остатках (CRT-decrypt).
  - Бинарное модульное возведение в степень (binary modular exponentiation — modexp).
- Первичный анализ безопасности
  - Атака на малые значения nnn с помощью пробного деления и метода Полларда (Pollard's Rho).
- Бенчмарк
  - Сравнение времени генерации ключей, шифрования и расшифровки в обычной форме и с использованием CRT.

4.2. Методика

- Проверка на простоту и генерация простых чисел
  - Miller–Rabin (5 раундов), чтобы гарантировать вероятность ошибки $\leq$ 0.001%.

- Генерация случайного простого числа с помощью random.getrandbits(bit_length) с установленными старшим и младшим битами = 1.

- Генерация ключей RSA

2-простое RSA:

- Выбрать два различных простых числа p, q, каждое длиной около 512 бит.
- $n = pq, \varphi(n) = (p-1)(q-1)$
- $e = 65537$ (если не подходит, то перебор следующих нечётных чисел)
- $d = e^{-1} \mod \varphi(n)$

Мульти-простое RSA:

- Аналогично, но выбрать 3 простых числа по ~341 бит, тогда $n = pqr$, $\varphi(n) = \prod(p_i - 1)$

- Шифрование / Расшифровка

  Шифрование:

$$c_i = m_i^e \mod n, \quad \text{где } m_i = \text{ord}(char)$$

  Расшифровка (обычная): $m_i = c_i^d \bmod n.$

  Расшифровка с помощью CRT:

$$m_1 = c^{d \bmod (p-1)} \bmod p, \quad m_2 = c^{d \bmod (q-1)} \bmod q$$

Затем собрать результат по китайской теореме об остатках (CRT), что экономит ~4× времени.

Бинарное модульное возведение в степень:

Реализуется быстрый алгоритм возведения в степень по модулю, сравнение с pow().

- Примитивные атаки

Trial division: перебор всех делителей от 2 до $\sqrt{n}$ (для $n \leq 32$ бит).

Pollard's Rho: итерации по функции $f(x) = x^2 + 1 \mod n$, чтобы найти общий делитель.

- Измерение производительности
  - Генерация ключа 1024-бита.

- Шифрование строки длиной ~20 символов.
- Расшифровка обычным способом и с использованием CRT.
- Измерение времени с использованием time.time().

4.3. Результат

Рисунок 1 - Результат работы программы

Рисунок 2 - Результат работы программы

Заключение

В ходе работы была реализована полная схема алгоритма RSA на языке Python, включая проверку простоты методом Миллера–Рабина, генерацию ключей (двух- и трёхпростых вариантов), шифрование и дешифрование с и без использования Китайской теоремы об остатках. Проведён примитивный криптоанализ малых модулей с помощью пробного деления и метода Полларда Rho, что подтвердило необходимость выбора достаточной битовой длины. Бенчмаркинг показал значительное ускорение расшифрования при использовании CRT (более чем в два раза) и эффективность оптимизаций. Полученные результаты демонстрируют баланс между параметрами безопасности и производительностью при практической реализации RSA.

ПРИЛОЖЕНИЕ А

Листинг А.1 – Код файла rsa.py

```
#!/usr/bin/env python3
import random
import time
from math import gcd

#
-------------------------------------------------------------------------
---
# 1. Prime Test: Miller-Rabin
#
-------------------------------------------------------------------------
---
def is_prime(n, k=5):
    if n < 2: return False
    for p in [2,3,5,7,11,13,17,19,23]:
        if n % p == 0:
            return n == p
    # write n-1 = 2^r * d
    r, d = 0, n-1
    while d % 2 == 0:
        r += 1; d //= 2
    for _ in range(k):
        a = random.randrange(2, n-1)
        x = pow(a, d, n)
        if x in (1, n-1):
            continue
        for __ in range(r-1):
            x = pow(x, 2, n)
            if x == n-1:
                break
        else:
            return False
    return True
```

```
#
-----------------------------------------------------------------------
---
# 2. Generate a random prime of given bit-length
#
-----------------------------------------------------------------------
---
def generate_prime(bit_length):
    while True:
        p = random.getrandbits(bit_length) | (1 << (bit_length-1)) | 1
        if is_prime(p):
            return p

#
-----------------------------------------------------------------------
---
# 3. Modular inverse via Extended Euclid
#
-----------------------------------------------------------------------
---
def modinv(a, m):
    def egcd(a, b):
        if b == 0:
            return (1, 0, a)
        x1, y1, g = egcd(b, a % b)
        return (y1, x1 - (a // b) * y1, g)
    x, y, g = egcd(a, m)
    if g != 1:
        raise Exception(f"No modular inverse for {a} mod {m}")
    return x % m

#
-----------------------------------------------------------------------
---
# 4. RSA Key Generation (2 primes)
#
-----------------------------------------------------------------------
---
def generate_keypair(bit_length=1024):
    p = generate_prime(bit_length // 2)
```

```
    q = generate_prime(bit_length // 2)
    while q == p:
        q = generate_prime(bit_length // 2)
    n   = p * q
    phi = (p-1)*(q-1)
    e   = 65537
    if gcd(e, phi) != 1:
        e = 3
        while gcd(e, phi) != 1:
            e += 2
    d = modinv(e, phi)
    return (e, n), (d, n), p, q

#
-------------------------------------------------------------------------
---
# 5. Multi-prime RSA Key Generation (3 primes)
#
-------------------------------------------------------------------------
---
def generate_multi_prime_keypair(bit_length=1024, k=3):
    bits = bit_length // k
    primes = [generate_prime(bits) for _ in range(k)]
    while len(set(primes)) < k:
        primes = [generate_prime(bits) for _ in range(k)]
    n = 1
    phi = 1
    for pi in primes:
        n *= pi
        phi *= (pi-1)
    e = 65537
    if gcd(e, phi) != 1:
          raise Exception("e và phi không nguyên tố cùng nhau, chọn lại
primes")
    d = modinv(e, phi)
    return (e, n), (d, n), primes

#
-------------------------------------------------------------------------
---
```

```
# 6. Binary Modular Exponentiation (modexp)
#
------------------------------------------------------------------------
---
def modexp(base, exponent, mod):
    result = 1
    base %= mod
    while exponent > 0:
        if exponent & 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exponent >>= 1
    return result

#
------------------------------------------------------------------------
---
# 7. Encrypt/Decrypt
#
------------------------------------------------------------------------
---
def encrypt(pubkey, message):
    e, n = pubkey
    return [pow(ord(c), e, n) for c in message]

def decrypt(privkey, ciphertext):
    d, n = privkey
    return ''.join(chr(pow(c, d, n)) for c in ciphertext)

#
------------------------------------------------------------------------
---
# 8. CRT Decrypt
#
------------------------------------------------------------------------
---
def crt_decrypt(privkey, ciphertext, p, q):
    d, n = privkey
    dp = d % (p-1)
    dq = d % (q-1)
```

```
    q_inv = modinv(q, p)
    msg = ""
    for c in ciphertext:
        m1 = pow(c, dp, p)
        m2 = pow(c, dq, q)
        h  = (q_inv * (m1 - m2)) % p
        m  = m2 + h * q
        msg += chr(m)
    return msg

#
------------------------------------------------------------------------
---
# 9. Primitive Attacks: Trial Division & Pollard's Rho
#
------------------------------------------------------------------------
---
def trial_division(n):
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return (i, n//i)
    return None

def pollards_rho(n):
    if n % 2 == 0:
        return 2
    def f(x): return (x*x + 1) % n
    x = y = 2
    d = 1
    while d == 1:
        x = f(x)
        y = f(f(y))
        d = gcd(abs(x-y), n)
    return d if d != n else None

#
------------------------------------------------------------------------
---
# 10. Demo & Benchmark
```

```
#
----------------------------------------------------------------------
---
def main():
    # --- 2-prime RSA demo ---
    print("===== 2-Prime RSA Demo =====")
    pub, priv, p, q = generate_keypair(1024)
    e, n = pub; d, _ = priv
    print("Public key (e,n):", pub)
    print("Private key (d,n):", priv)
    print("Primes p, q:", p, q, "\n")

    plaintext = "Hello, ITMO University!"
    print("Plaintext:", plaintext)
    ciphertext = encrypt(pub, plaintext)
    print("Ciphertext:", ciphertext, "\n")

    decrypted = decrypt(priv, ciphertext)
    print("Decrypted (normal):", decrypted)
    decrypted_crt = crt_decrypt(priv, ciphertext, p, q)
    print("Decrypted (CRT):  ", decrypted_crt, "\n")

    # --- Verify modexp matches pow for first char ---
    m0 = ord(plaintext[0])
    c0 = ciphertext[0]
    print("modexp check:", modexp(m0, e, n)==c0, "(should be True)\n")

    # --- Multi-prime RSA demo ---
    print("===== Multi-Prime RSA Demo (3 primes) =====")
    pub2, priv2, primes = generate_multi_prime_keypair(1024, k=3)
    e2, n2 = pub2; d2, _ = priv2
    print("Public key (e,n):", pub2)
    print("Private key (d,n):", priv2)
    print("Primes:", primes, "\n")

    pt2 = "Multi-Prime RSA!"
    print("Plaintext:", pt2)
    ct2 = encrypt(pub2, pt2)
    print("Ciphertext:", ct2, "\n")
```

```
    dt2 = decrypt(priv2, ct2)
    print("Decrypted:", dt2, "\n")

    # --- Primitive attacks on small n ---
    print("===== Primitive Attacks on small n =====")
    pub_small, priv_small, p_s, q_s = generate_keypair(32)
    n_s = pub_small[1]
    print("Small n:", n_s)
    td = trial_division(n_s)
    print("Trial division factors:", td)
    pr = pollards_rho(n_s)
    print("Pollard's Rho factor:", pr, "\n")

    # --- Benchmark performance ---
    print("===== Benchmark (1024-bit) =====")
    t0 = time.time()
    pub_b, priv_b, p_b, q_b = generate_keypair(1024)
    t_key = time.time() - t0
    print(f"Keygen time: {t_key:.3f} s")

    msg = "Benchmark Test"
    t1 = time.time()
    ct_b = encrypt(pub_b, msg)
    t_enc = time.time() - t1
    print(f"Encrypt time: {t_enc:.3f} s")

    t2 = time.time()
    _ = decrypt(priv_b, ct_b)
    t_dec = time.time() - t2
    print(f"Decrypt time (normal): {t_dec:.3f} s")

    t3 = time.time()
    _ = crt_decrypt(priv_b, ct_b, p_b, q_b)
    t_crt = time.time() - t3
    print(f"Decrypt time (CRT):    {t_crt:.3f} s")

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