Введение:

Электронная цифровая подпись (ЭЦП) - это метод аутентификации и обеспечения целостности электронных документов и сообщений. Она используется для подтверждения авторства и целостности информации, передаваемой по электронным каналам связи.

Принцип работы ЭЦП основан на использовании криптографических алгоритмов и ключей. Когда отправитель хочет подписать электронный документ или сообщение, он использует свой личный ключ для создания уникальной цифровой подписи. Личный ключ является конфиденциальной информацией и должен быть известен только отправителю. Затем получатель, чтобы проверить подпись, использует публичный ключ отправителя, который доступен всем. Публичный ключ используется для расшифровки цифровой подписи и сравнения ее с оригинальным содержимым документа или сообщения. Если они совпадают, то подпись считается действительной и можно быть уверенным в авторстве и целостности информации.

RSA (Rivest-Shamir-Adleman) - это асимметричный алгоритм шифрования с открытым ключом, который используется для электронной цифровой подписи (ЭЦП) и шифрования данных. Он назван по фамилиям его создателей: Рональда Райвеста, Ади Шамира и Леонарда Адлемана.

Код:

# Функция для проверки числа на простоту

def is\_prime(n):

    if n <= 1:

        return False

    for i in range(2, int(n\*\*0.5) + 1):

        if n % i == 0:

            return False

return True

1) Функция `is\_prime(n)` проверяет, является ли число `n` простым числом. Она возвращает `True`, если число простое, и `False`, если число составное.

# Функция для генерации простых чисел заданной длины

def generate\_large\_prime(keysize=1024):

    while True:

        num = random.getrandbits(keysize)

        if is\_prime(num):

            return num

2) Функция `generate\_large\_prime(keysize=1024)` генерирует случайное простое число заданной длины `keysize`. Она использует функцию `is\_prime(n)` для проверки простоты сгенерированного числа и возвращает это число, если оно простое.

# Функция для вычисления НОД

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a

3) Функция `gcd(a, b)` вычисляет наибольший общий делитель (НОД) для двух чисел `a` и `b`. Она использует алгоритм Евклида для нахождения НОД и возвращает его значение.

# Функция для вычисления обратного по модулю

def multiplicative\_inverse(e, phi):

    d, x1, x2, y1 = 0, 0, 1, 1

    temp\_phi = phi

    while e > 0:

        temp1 = temp\_phi // e

        temp2 = temp\_phi - temp1 \* e

        temp\_phi, e = e, temp2

        x = x2 - temp1 \* x1

        y = d - temp1 \* y1

        x2, x1 = x1, x

        d, y1 = y1, y

    if temp\_phi == 1:

        return d + phi

4) Функция `multiplicative\_inverse(e, phi)` вычисляет обратное число по модулю `e` для числа `phi`. Она использует расширенный алгоритм Евклида и возвращает найденное обратное число.

# Функция для генерации ключей

def generate\_keypair(keysize):

    p = generate\_large\_prime(keysize)

    q = generate\_large\_prime(keysize)

    n = p \* q

    phi = (p-1) \* (q-1)

5) Функция `generate\_keypair(keysize)` генерирует пару ключей (открытый и закрытый) заданного размера `keysize`. Она использует функции `generate\_large\_prime(keysize)`, `gcd(a, b)` и `multiplicative\_inverse(e, phi)` для генерации простых чисел и вычисления необходимых значений. Возвращает пару ключей в формате ((e, n), (d, n)).

    e = random.randrange(1, phi)

    g = gcd(e, phi)

    while g != 1:

        e = random.randrange(1, phi)

        g = gcd(e, phi)

    d = multiplicative\_inverse(e, phi)

    return ((e, n), (d, n))

# Функция для быстрого возведения в степень по модулю

def fast\_modular\_exponentiation(base, exponent, modulus):

    result = 1

    base = base % modulus

    while exponent > 0:

        if exponent % 2 == 1:

            result = (result \* base) % modulus

        exponent = exponent >> 1

        base = (base \* base) % modulus

    return result

6) Функция `fast\_modular\_exponentiation(base, exponent, modulus)` реализует быстрое возведение в степень по модулю. Она используется для эффективного вычисления шифрования и дешифрования.

# Функция для шифрования

def encrypt(pk, plaintext):

    key, n = pk

    cipher = [fast\_modular\_exponentiation(ord(char), key, n) for char in plaintext]

    return cipher

7) Функция `encrypt(pk, plaintext)` выполняет шифрование заданного текста `plaintext` с использованием открытого ключа `pk`. Она преобразует каждый символ текста в численное значение, затем применяет быстрое возведение в степень по модулю для шифрования каждого символа. Возвращает зашифрованный текст в виде списка чисел.

# Функция для дешифрования

def decrypt(pk, ciphertext):

    key, n = pk

    plain = [chr(fast\_modular\_exponentiation(char, key, n)) for char in ciphertext]

    return ''.join(plain)

8) Функция `decrypt(pk, ciphertext)` выполняет дешифрование заданного зашифрованного текста `ciphertext` с использованием закрытого ключа `pk`. Она применяет быстрое возведение в степень по модулю для каждого числа из зашифрованного текста, затем преобразует числа в символы. Возвращает дешифрованный текст.

# Функция для генерации случайных сообщений заданной длины

def generate\_random\_message(length):

    letters = string.ascii\_letters + string.digits + string.punctuation

    return ''.join(random.choice(letters) for i in range(length))

8) Функция `generate\_random\_message(length)` генерирует случайное сообщение заданной длины `length`. Она использует библиотеку `string` для определения допустимых символов в сообщении и функцию `random.choice()` для выбора случайных символов. Возвращает случайное сообщение.

# Функция для тестирования алгоритма RSA

def test\_rsa(keysize, num\_tests):

    public, private = generate\_keypair(keysize)

    for i in range(num\_tests):

        message = generate\_random\_message(random.randint(1, 100))

        encrypted\_msg = encrypt(public, message)

        decrypted\_msg = decrypt(private, encrypted\_msg)

        if message != decrypted\_msg:

            print(f"Тест {i+1} провален.")

            print(f"Оригинальное сообщение: {message}")

            print(f"Расшифрованное сообщение: {decrypted\_msg}")

            return

        else:

            print(f"Тест {i+1} пройден успешно.")

    print(f"Все {num\_tests} тестов пройдены успешно.")

# Вызов функции тестирования

if \_\_name\_\_ == '\_\_main\_\_':

    test\_rsa(1024, 10)

Вывод:

В ходе моего изучения я успешно рассмотрел ключевые аспекты разработки программного обеспечения для шифрования и цифровой подписи с использованием алгоритма RSA на языке Python. Были выполнены следующие задачи:

1. **Разработка алгоритма RSA**: Создана программа, реализующая генерацию ключей, шифрование и дешифрование сообщений без использования сторонних библиотек.
2. **Метод последовательного возведения в квадрат и умножения**: Интегрирован метод для ускорения процесса шифрования.
3. **Тестирование**: Разработана функция для тестирования программы на 10 наборах тестовых данных, что позволяет проверить корректность работы алгоритма.
4. **Сохранение результатов**: Добавлена возможность сохранения зашифрованных и расшифрованных сообщений в файлы для последующего анализа.
5. **Вывод результатов**: Реализован вывод оригинальных, зашифрованных и расшифрованных сообщений в консоль для наглядности.