Вариант № 21. Взвод 2231

Данила Михальцов, Глеб Скрябин, Артём Зиннуров

Содержание

1 Постановка задачи

Требуется разработать и программно реализовать учебную систему электронной подписи (ЭП) на основе следующих криптографических алгоритмов:

1. Криптосистема RSA в режиме ЭП.
2. Хэш-функция SHA-224 (из семейства SHA-2).

2 Разработка системы электронной подписи на базе криптосистемы RSA

Система электронной подписи — это цифровая подпись, используемая для аутентификации личности физических или юридических лиц и обеспечения целостности электронных документов. Криптосистема RSA является одним из наиболее широко используемых алгоритмов для систем электронной подписи.

Криптосистема RSA основана на математической задаче факторизации больших простых чисел. Алгоритм генерирует пару ключей — открытый ключ и закрытый ключ — которые используются для шифрования и расшифровки сообщений. Открытый ключ доступен любому, а закрытый ключ хранится в секрете владельцем.

2.1 Работа криптосистемы RSA

Для создания электронной подписи с помощью RSA выполняются следующие шаги:

1. Подписываемый документ хэшируется с использованием безопасного алгоритма хеширования для создания дайджеста сообщения.
2. Затем дайджест сообщения шифруется с использованием закрытого ключа подписывающей стороны для создания цифровой подписи.
3. Цифровая подпись прикрепляется к документу вместе с открытым ключом подписавшего для создания подписанного документа.

Для проверки электронной подписи выполняются следующие действия:

1. Подписанный документ разделяется на документ и цифровую подпись.
2. Документ хэшируется с использованием того же алгоритма безопасного хеширования, который подписывающая сторона использовала для создания дайджеста сообщения.
3. Дайджест сообщения расшифровывается с использованием открытого ключа подписывающей стороны, чтобы получить дайджест исходного сообщения.
4. Исходный дайджест сообщения сравнивается с дайджестом сообщения, созданным из документа. Если они совпадают, подпись действительна.

2.2 Государственные стандарты Российской Федерации криптосистемы RSA

В России использование систем электронной подписи регулируется несколькими государственными стандартами, в том числе ГОСТ Р 34.10-2012 и ГОСТ Р 34.11-2012. Эти стандарты определяют требования к криптографическим алгоритмам, используемым в системах электронной подписи, включая криптосистему RSA. Они также определяют технические требования к устройствам электронной подписи и процедуры создания и проверки электронных подписей.

3 Программная реализация криптосистемы RSA на языке Python

3.1 Используемые библиотеки

```
# ! pip install pycryptodome==3.9.9
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
from Crypto.Signature import pss
import datetime
import os
import shutil
```

3.2 Подписание данных

```
def sign_data(data_file_path,
              signer_name,
              signer_private_key_file_path,
              signer_public_key_file_path):

    # Получить ключи из файлов
    signer_private_key = \
        RSA.import_key(open(signer_private_key_file_path).read())
    # Получить хеш
    hash = get_file_hash(data_file_path)
    # Зашифровать хеш закрытым ключом подписывающего
    encrypted_hash = encrypt_hash(hash, signer_private_key)
    # Получить имя2 файла с расширением из пути
    signer_public_key_file_name = \
        os.path.basename(signer_public_key_file_path)
    # Создать сертификат подписи
    certificate = \
        create_certificate(signer_name, signer_public_key_file_name)
    # Создать папку для файлов подписи
    if os.path.exists("Digital signature"):
        shutil.rmtree("Digital signature")
```

```
    os.mkdir("Digital signature")
    # Записать зашифрованный хеш в файл и
    # поместить в специальную папку для ЭЦП
    with open("Digital signature/Digital signature.pem", "wb+") as f:
        f.write(encrypted_hash)
    # Добавить сертификат
    with open("Digital signature/Digital signature certificate.pem", "a") as f:
        f.write(certificate)
    # Скопировать файл с данными в специальную папку для ЭЦП
    data_file_name = os.path.basename(data_file_path)
    shutil.copyfile(data_file_path,
                    "Digital signature/" + data_file_name)
    # Скопировать файл с открытым ключом
    # отправителя в специальную папку для ЭЦП
    shutil.copyfile(signer_public_key_file_path,
                    "Digital signature/" + signer_public_key_file_name)
    os.rename("Digital signature/" + signer_public_key_file_name,
              "Digital signature/" + "Signer public key.pem")
```

3.3 Зашифровка хеша

```
def encrypt_hash(hash, signer_private_key):
    # Создать сигнатуру по закрытому ключу подписывающего
    signature = pss.new(signer_private_key)
    # Подписать сигнатуру
    return signature.sign(hash)
```

3.4 Создание сертификата цифровой подписи

```
 def create_certificate(signer_name, signer_public_key_file_name):
    certificate = f"Дата формирования подписи:" \
                  f"{datetime.datetime.now().strftime('%d-%m-%Y %H:%M')};" \
                  f"\nФИО подписывающего: " \
                  f"{signer_name};\nИмя файла открытого ключа подписи: " \
                  f"{signer_public_key_file_name}"

    return certificate
```

3.5 Проверка подписи

```
def verification_of_data_digital_signature(encrypted_data_file_path,
                                           digital_signature_file_path,
                                           signer_public_key_file_path):

    # Получить ключи из файлов
    signer_public_key = \
        RSA.import_key(open(signer_public_key_file_path).read())
```

```
    # Получить зашифрованный хеш из подписи
    with open(digital_signature_file_path, "rb") as f:
        encrypted_hash = f.read()
    # Получить хеш оригинального файла
    original_file_hash = get_file_hash(encrypted_data_file_path)
    # Сравнить отправленный хеш из подписи и хеш файла
    is_hashes_same = verify_hash(encrypted_hash,
                                 original_file_hash,
                                 signer_public_key)
    return is_hashes_same


# Подтвердить правильность хешей
def verify_hash(encrypted_hash, original_hash, signer_public_key):
    # Создать верификатор по открытому ключу подписывающего
    verifier = pss.new(signer_public_key)
    try:
        # Проверить два хеша на идентичность
        verifier.verify(original_hash, encrypted_hash)
        return True
    except(ValueError, TypeError):
        return False


# Получить хеш
def get_file_hash(data_file_path):
    # Создать переменную хеш-функции SHA256
    hash = SHA256.new()
    # Объясить блок оптимального размера
    block = bytearray(128 * 1024)
    # Создать memoryview для предоставления внутренних
    # данных буферного объекта без копирования
    memory_view = memoryview(block)
    # Открыть файл в двоичном режиме ('b') для чтения ('r').
    # Отключить двойную буферизацию, так как уже используется
    # оптимальный размер блока
    with open(data_file_path, 'rb', buffering=0) as f:
        # iter итерирует то, что в скобках. Второй аргумент для
        # того, чтобы после каждого вызова функции из первого
        # аргумента сравнивать её возвращаемое значение со вторым
        # аргументом и если они равны, то прекращать итерирование
        for n in iter(lambda: f.readinto(memory_view), 0):
            # Обновить хеш
            hash.update(memory_view[:n])
    return hash
```

```
# Сгенерировать закрытый и открытый ключи
def generate_keys():
    key = RSA.generate(2048)
    return key.export_key(), key.publickey().export_key()


# Сохранить закрытый и открытый ключи в файлы
def save_keys(private_key, public_key):
    file = open("Keys/private_key.pem", "wb+")
    file.write(private_key)
    file.close()
    file = open("Keys/public_key.pem", "wb+")
    file.write(public_key)
    file.close()
```

3.6 Демонстрация работы криптосистемы RSA

3.6.1 Генерация ключей

```
private_key, public_key = generate_keys()
save_keys(private_key, public_key)
print("Ключи сгенерированы.")
```

3.6.2 Подписание данных

```
data_file_path = 'sample_data/datafile_example.csv'
private_key_path = 'Keys/private_key.pem'
public_key_path = 'Keys/public_key.pem'
signer_name = 'Михальцов Д.А.'
sign_data(data_file_path,
          signer_name,
          private_key_path,
          public_key_path)

print("\nДанные успешно подписаны.")
```

3.6.3 Проверка подписи

```
data_file_path = 'sample_data/datafile_example.csv'
digital_signature_path = 'Digital signature/Digital signature.pem'
public_key_path = 'Keys/public_key.pem'
verification = verification_of_data_digital_signature(
    data_file_path,
    digital_signature_path,
    public_key_path)
```

```
if verification:
    print("\nВерификация успешна. Хеши равны. " \
          "Данные не были изменены или подменены.")
else:
    print("\nВнимание! Верификация провалилась. " \
          "Хеши не равны. Данные, вероятно, не соответствуют")
```

4 Разработка системы электронной подписи с использованием криптографического алгоритма SHA-224

Система электронной подписи является важнейшим элементом обеспечения безопасности и подлинности электронных документов. Одним из самых надежных криптографических алгоритмов, используемых для электронных подписей, является хэш-функция SHA-224 из семейства SHA-2. В этом документе мы обсудим, как работает этот алгоритм и какие государственные стандарты РФ соблюдаются для этой технологии.

4.1 Работа алгоритма SHA-224

Алгоритм SHA-224 — это односторонняя криптографическая хеш-функция, которая генерирует уникальный вывод фиксированного размера, также известный как дайджест, для любого заданного ввода. Этот алгоритм работает в следующих шагах:

1. Заполнение: Входное сообщение дополняется серией битов, чтобы соответствовать требуемой длине в 512 бит.
2. Инициализация дайджеста сообщения: 224-битный дайджест сообщения инициализируется определенными значениями.
3. Расчет расписания сообщений: сообщение делится на блоки по 512 бит, и расписание сообщений рассчитывается для каждого блока.
4. Функция сжатия: дайджест сообщения и расписание сообщений объединяются с использованием функции сжатия для создания окончательного дайджеста сообщения.
5. Вывод: создается окончательный дайджест сообщения, представляющий собой 224-битное хэш-значение.

4.2 Государственные стандарты Российской Федерации алгоритма SHA-224

Алгоритм SHA-224 включен в государственные стандарты Российской Федерации для электронных подписей. На данную технологию соблюдаются следующие государственные стандарты:

1. ГОСТ Р 34.10-2012: Настоящий стандарт устанавливает требования к алгоритмам цифровой подписи. Он включает алгоритм SHA-224 как одну из утвержденных хеш-функций для электронных подписей.
2. ГОСТ Р 34.11-2012: Настоящий стандарт устанавливает требования к криптографической защите информации с использованием хеш-функций. Он включает алгоритм SHA-224 как одну из утвержденных хеш-функций для электронных подписей.
3. ФЗ-63: Это федеральный закон, устанавливающий правовую базу электронной подписи в Российской Федерации. Он требует, чтобы электронные подписи основывались на утвержденных криптографических алгоритмах, включая SHA-224.

5 Программная реализация алгоритма SHA-224 на языке Python

5.1 Вспомогательные функции

```
# константа для суммы по mod 2**32
VAL_MOD = 2 ** 32

def set_zero_in_end(bites, len_str):
    """ метод для вставки нулей в конец битовой последовательности"""
    while len(bites) % len_str != 0:
        bites += '0'
    return bites

def set_zero_in_start(bites, len_str):
    """ метод для вставки нулей в начало битовой последовательности"""
    while len(bites) % len_str != 0:
        bites = '0' + bites
    return bites

def set_len_str_in_end(bits, bits_len, max_mod=512):
    """ метод для вставки в конец битовой последовательности текста
        длины сообщения"""
    while (len(bits) + len(bits_len)) % max_mod != 0:
        bits += '0'
    bits += bits_len
    return bits

def hex_to_bin(hex_num, len_bits=32):
    """ метод для перевода числа из 16-ричного представления в бинарное"""
    bits = bin(hex_num)[2:]
    # добавляем в начало нули для получения нужной длины последовательности
    while len(bits) < len_bits:
        bits = '0' + bits
```

```
    return bits

def str_to_bin(text):
    """ метод для перевода строки в бинарное представление """
    binary = ''
    # приводим текст к виде списка байт
    byte_array = bytearray(text, "utf8")
    # каждый байт преобразуем в битовую строчку
    for byte in byte_array:
        binary_repr = bin(byte)[2:]
        while len(binary_repr) < 8:
            binary_repr = '0' + binary_repr
        binary += binary_repr
    return binary

def rotate_right(list_bits, count):
    """ метод для циклического сдвига вправо бинарной последовательности"""
    for _ in range(count):
        list_bits = list_bits[-1] + list_bits[:-1]
    return list_bits

def shift_right(list_bits, count):
    """ метод для логического сдивга вправо бинарной последовательности"""
    for _ in range(count):
        list_bits = '0' + list_bits[:-1]
    return list_bits

def xor(list_bits1, list_bits2):
    """ метод для вычисления исключающего или"""
    # дополняем последовательности до одинаковой длины
    max_len = max(len(list_bits1), len(list_bits2))
    list_bits1 = '0' * (max_len - len(list_bits1)) + list_bits1
    list_bits2 = '0' * (max_len - len(list_bits2)) + list_bits2
    rez_bits = []
    # xor: 1 если хотя бы один бит 1, но не оба вместе, иначе 0
    for i in range(max_len):
        new_bit = '1' if list_bits1[i] != list_bits2[i] else '0'
        rez_bits.append(new_bit)
    return ''.join(rez_bits)

def log_and(bits1, bits2):
    """ метод для вычисления логического И """
    max_len = max(len(bits1), len(bits2))
    bits1 = '0' * (max_len - len(bits1)) + bits1
    bits2 = '0' * (max_len - len(bits2)) + bits2
```

```
    rez_bits = []
    # and: 1, если оба бита = 1, иначе 0
    for i in range(max_len):
        new_bit = '1' if bits1[i] == bits2[i] == '1' else '0'
        rez_bits.append(new_bit)
    return ''.join(rez_bits)

def summator(*list_bits):
    """ метод для суммирования битовых последовательностей"""
    summa = 0
    # для получения суммы всё переводится к обычным 10-чным числам
    for bit in list_bits:
        summa += int(bit, base=2)
    # в sha-256 сумма берется по mod 2**32
    summa = summa % VAL_MOD
    # приводим обратно к битовой последовательности
    binary = bin(summa)[2:]
    while len(binary) < 32:
        binary = '0' + binary
    return binary

def log_not(bits):
    """ метод для получения отрицания от битовой последовательности"""
    binary = ''
    for bit in bits:
        binary += '1' if bit == '0' else '0'
    return binary
```

5.2 Список первоначальных значений хеш-функции

```
# первые 32 бита дробных частей квадратных корней первых восьми простых чисел
h0 = hex_to_bin(0xC1059ED8)
h1 = hex_to_bin(0x367CD507)
h2 = hex_to_bin(0x3070DD17)
h3 = hex_to_bin(0xF70E5939)
h4 = hex_to_bin(0xFFC00B31)
h5 = hex_to_bin(0x68581511)
h6 = hex_to_bin(0x64F98FA7)
h7 = hex_to_bin(0xBEFA4FA4)
```

5.3 Таблица констант

```
# первые 32 бита дробных частей кубических корней первых 64 простых чисел
constants = [
    hex_to_bin(0x428A2F98), hex_to_bin(0x71374491), hex_to_bin(0xB5C0FBCF),
    hex_to_bin(0xE9B5DBA5), hex_to_bin(0x3956C25B), hex_to_bin(0x59F111F1),
```

```
    hex_to_bin(0x923F82A4), hex_to_bin(0xAB1C5ED5), hex_to_bin(0xD807AA98),
    hex_to_bin(0x12835B01), hex_to_bin(0x243185BE), hex_to_bin(0x550C7DC3),
    hex_to_bin(0x72BE5D74), hex_to_bin(0x80DEB1FE), hex_to_bin(0x9BDC06A7),
    hex_to_bin(0xC19BF174), hex_to_bin(0xE49B69C1), hex_to_bin(0xEFBE4786),
    hex_to_bin(0x0FC19DC6), hex_to_bin(0x240CA1CC), hex_to_bin(0x2DE92C6F),
    hex_to_bin(0x4A7484AA), hex_to_bin(0x5CB0A9DC), hex_to_bin(0x76F988DA),
    hex_to_bin(0x983E5152), hex_to_bin(0xA831C66D), hex_to_bin(0xB00327C8),
    hex_to_bin(0xBF597FC7), hex_to_bin(0xC6E00BF3), hex_to_bin(0xD5A79147),
    hex_to_bin(0x06CA6351), hex_to_bin(0x14292967), hex_to_bin(0x27B70A85),
    hex_to_bin(0x2E1B2138), hex_to_bin(0x4D2C6DFC), hex_to_bin(0x53380D13),
    hex_to_bin(0x650A7354), hex_to_bin(0x766A0ABB), hex_to_bin(0x81C2C92E),
    hex_to_bin(0x92722C85), hex_to_bin(0xA2BFE8A1), hex_to_bin(0xA81A664B),
    hex_to_bin(0xC24B8B70), hex_to_bin(0xC76C51A3), hex_to_bin(0xD192E819),
    hex_to_bin(0xD6990624), hex_to_bin(0xF40E3585), hex_to_bin(0x106AA070),
    hex_to_bin(0x19A4C116), hex_to_bin(0x1E376C08), hex_to_bin(0x2748774C),
    hex_to_bin(0x34B0BCB5), hex_to_bin(0x391C0CB3), hex_to_bin(0x4ED8AA4A),
    hex_to_bin(0x5B9CCA4F), hex_to_bin(0x682E6FF3), hex_to_bin(0x748F82EE),
    hex_to_bin(0x78A5636F), hex_to_bin(0x84C87814), hex_to_bin(0x8CC70208),
    hex_to_bin(0x90BEFFFA), hex_to_bin(0xA4506CEB), hex_to_bin(0xBEF9A3F7),
    hex_to_bin(0xC67178F2)]
```

5.4 Подготовка данных для хеширования

```
# сообщение для хеширования
msg = "Euler is held to be one of the greatest mathematicians in history."
# переводим сообщение в битовую последовательность
m = str_to_bin(msg)
# добавляем в конец '1'
m = m + "1"
# добавляем в конец исходную длину сообщения в виде 64 битной последовательности
bits_len = set_zero_in_start(bin(len(msg) * 8)[2:], 64)
# добавляем в конец длину строки
m = set_len_str_in_end(m, bits_len)
print(len(m))
print(m)

for i in range(0, len(m), 512):
    part = m[i:i + 512]
    parts = []
    # разбиваем исходное сообщение на 16 кусочкой длиной 32 бита
    for j in range(0, 512, 32):
        parts.append(part[j:j + 32])

# генерируем дополнительные 48 слов для хеширования
for k in range(16, 64):
    # sigma0 = right_rotate(parts[k-15], 7) xor right_rotate(parts[k-15], 18) xor shift_right(p
```

```
    rr1 = rotate_right(parts[k - 15], 7)
    rr2 = rotate_right(parts[k - 15], 18)
    sr = shift_right(parts[k - 15], 3)
    x1 = xor(rr1, rr2)
    s0 = xor(x1, sr)
    # sigma1 = right_rotate(parts[k-2], 17) xor right_rotate(parts[k-2], 19) xor shift_rig
    rr1 = rotate_right(parts[k - 2], 17)
    rr2 = rotate_right(parts[k - 2], 19)
    sh = shift_right(parts[k - 2], 10)
    x1 = xor(rr1, rr2)
    s1 = xor(x1, sh)
    # новое слово: parts[k - 16] + sigma0 + parts[k - 7] + sigma1
    new_part = summator(parts[k - 16], s0, parts[k - 7], s1)
    parts.append(new_part)

# инициализируем дополнительные переменные
a = h0
b = h1
c = h2
d = h3
e = h4
f = h5
g = h6
h = h7
```

5.5 Работа алгоритма хеширования

```
# весь алгоритм хеширования выполняется 64 раза
for k in range(64):
    # sigma1 = rotate_right(e, 6) xor rotate_right(e, 11)
    # xor rotate_right(e, 25)
    rr1 = rotate_right(e, 6)
    rr2 = rotate_right(e, 11)
    rr3 = rotate_right(e, 25)
    x1 = xor(rr1, rr2)
    s1 = xor(x1, rr3)
    # ch = log_and(e, f) xor log_and(log_not(e), g)
    rr1 = log_and(e, f)
    rr3 = log_and(log_not(e), g)
    ch = xor(rr1, rr3)
    # temp1 = h + s1 + ch + constants[k] + parts[k]
    t1 = summator(h, s1, ch, constants[k], parts[k])
    # sigma0 = rotate_right(a, 2) xor rotate_right(a, 13)
    # xor rotate_right(a, 22)
    rr1 = rotate_right(a, 2)
    rr2 = rotate_right(a, 13)
```

```
    rr3 = rotate_right(a, 22)
    x1 = xor(rr1, rr2)
    s0 = xor(x1, rr3)
    # maj = log_and(a, b) xor log_and(a, c) xor log_and(b, c)
    rr1 = log_and(a, b)
    rr2 = log_and(a, c)
    rr3 = log_and(b, c)
    x1 = xor(rr1, rr2)
    maj = xor(x1, rr3)
    # temp2 = sigma0 + maj
    t2 = summator(s0, maj)
    # теперь необходимо изменить значения временных переменных
    h = g
    g = f
    f = e
    e = summator(d, t1)
    d = c
    c = b
    b = a
    a = summator(t1, t2)
```

5.6 Сборка результата работы алгоритма

```
# в конце необходимо изменить начальные переменные для хранения хеша
h0 = summator(h0, a)
h1 = summator(h1, b)
h2 = summator(h2, c)
h3 = summator(h3, d)
h4 = summator(h4, e)
h5 = summator(h5, f)
h6 = summator(h6, g)
h7 = summator(h7, h)

sha224text = (hex(int(h0, base=2))[2:] +
              hex(int(h1, base=2))[2:] +
              hex(int(h2, base=2))[2:] +
              hex(int(h3, base=2))[2:] +
              hex(int(h4, base=2))[2:] +
              hex(int(h5, base=2))[2:] +
              hex(int(h6, base=2))[2:])

print(f"sha224text = {sha224text},\nlen = {len(sha224text)}")

# >>> sha224text = e7af93727a21afd0e4215af076761e1a342bdbca9ea02de993527ef4,
# >>> len = 56
```