Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**ЛАБОРАТОРНАЯ РАБОТА №11**

**Дисциплина: Криптографические протоколы**

Работу выполнила: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Д. Н. Баева

Направление подготовки: 02.03.02 Фундаментальная информатика и информационные технологии

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А. А. Крамаренко

**Цель работы:** реализовать программный продукт построения sha-256 для введенного текста.

**Ход работы:**

SHA (Алгоритмы безопасного хеширования) – это семейство криптографических хэш-функций, способных принимать сообщения произвольной длины и вычислять уникальный хэш-код фиксированной длины. Хэш-код SHA может быть использован для проверки целостности сообщения, а также для генерации цифровой подписи сообщения. На данный момент существует несколько стандартов безопасного алгоритма, каждый последующий включает более надёжные хэш-функции:

1. SHA-0 – исходная версия 160-битной хеш-функции, опубликованной в 1993 году под названием «SHA».
2. SHA-1 – исправленная версия SHA-0, опубликованная в 1995 году.
3. SHA-2 – набор криптографических хэш-функций, впервые опубликованный в 2001 году. Включает в себя несколько размеров ключа, включая SHA-256, SHA-384 и SHA-512. Считается более безопасным, чем SHA-1, и рекомендуется для использования в новых системах.
4. SHA-3 – последняя версия семейства хэш-функций SHA, ранее называвшаяся Keccak, выбранная в 2012 году после публичного конкурса среди разработчиков. Он поддерживает те же длины хэшей, что и SHA-2, но представляет собой новую хэш-функцию, которая отличается от SHA-1 и SHA-2.

SHA-256 относится к стандарту SHA-2, использует размер слова в 32 бита. Окончание 256 означает, что фиксированный размер хэша для любого сообщения равен 256 бит.

Кратко описать алгоритм можно следующими шагами:

1. **Предварительная обработка**:
   * Преобразование входных данных в двоичный вид.
   * Добавление единицы и заполнение нулями до кратности 512 битам.
   * Добавление 64-битного значения, обозначающего длину входных данных.
2. **Инициализация значений хеша (h)**:
   * Создание 8 начальных значений хеша.
3. **Обработка блоков**:
   * Разбиение входных данных на 64-байтные блоки.
   * Применение раундовых функций для каждого блока.
4. **Формирование финального хеша**:
   * Комбинирование результатов обработки блоков.
   * Получение 256-битного хеша.

Программный продукт состоит из следующих функций, которые применяются (вызываются) последовательно:

1. Функция **generate\_hash(message)** выполняет вычисление хеша SHA-256 для входного сообщения.

Сперва выполняется проверка типа сообщения и преобразование в байтовый массив: если входное сообщение — строка (str), оно преобразуется в байтовый массив с использованием кодировки UTF-8. Если сообщение уже является байтовым объектом (bytes), оно остается без изменений. Если сообщение не является байтовым массивом или строкой, вызывается исключение TypeError.

Затем следует этап дополнения (Padding): добавляется бит 0x80 (в двоичном виде: 10000000), обозначающий конец сообщения. Затем добавляются нули до тех пор, пока длина сообщения (в битах) плюс 64 не станет кратной 512. Добавляется длина сообщения (в битах) в виде 8-байтового числа.

3 этап - разбиение на блоки (Parsing): сообщение разбивается на блоки по 512 бит (64 байта). Каждый блок представляет собой 512-битный фрагмент данных.

После следует инициализация значений хеш-функции: задаются начальные значения переменных h0, h1, …, h7.

Далее вычисление хеша: для каждого блока сообщения выполняется 64 раунда. Вычисляются значения t1 и t2 на основе предыдущих значений. Обновляются переменные a, b, c, …, h.

Сперва вычисляются промежуточные значения переменных, а затем они объединяются в итоговый хеш.

Фрагмент функции представлен на рисунке 1.

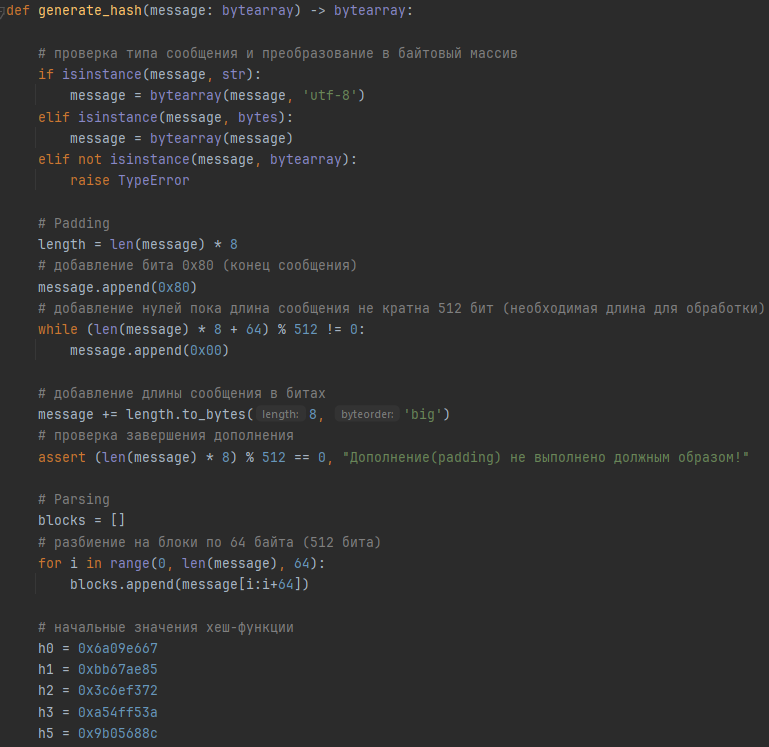


Рисунок 1 – Вычисление хэша SHA-256.

Затем следуют функции, выполняющие вспомогательную роль в вычислении хэша – выполнение циклического сдвига на разное количество битов.

1. Функция **sigma0(num: int)** принимает целое число num в качестве аргумента.

Сначала она выполняет циклический сдвиг вправо (\_rotate\_right) для num на 7 битов. Затем производит такой же сдвиг на 18 битов. Далее выполняется операция XOR (побитовое исключающее ИЛИ) между результатами обоих сдвигов. Наконец, выполняется сдвиг num на 3 бита вправо.

Результатом функции является полученное значение num.

1. Функция **sigma1(num: int)** также принимает целое число num в качестве аргумента.

Сначала она выполняет циклический сдвиг вправо для num на 17 битов. Затем производит такой же сдвиг на 19 битов. Далее выполняется операция XOR между результатами обоих сдвигов. Наконец, выполняется сдвиг num на 10 битов вправо.

Результатом функции также является полученное значение num.

1. Функция **\_capsigma0(num: int)** принимает целое число num в качестве аргумента.

Сначала она выполняет циклический сдвиг вправо (\_rotate\_right) для num на 2 бита. Затем производит такой же сдвиг на 13 битов. Далее выполняется операция XOR (побитовое исключающее ИЛИ) между результатами обоих сдвигов. Наконец, производится еще один циклический сдвиг num на 22 бита вправо.

Результатом функции является полученное значение num.

1. Функция \_**capsigma1(num: int)** также принимает целое число num в качестве аргумента.

Сначала она выполняет циклический сдвиг вправо для num на 6 битов. Затем производит такой же сдвиг на 11 битов. Далее выполняется операция XOR между результатами обоих сдвигов. Наконец, производится еще один циклический сдвиг num на 25 битов вправо.

Результатом функции также является полученное значение num.

Функции циклического сдвига представлены на рисунке 2.

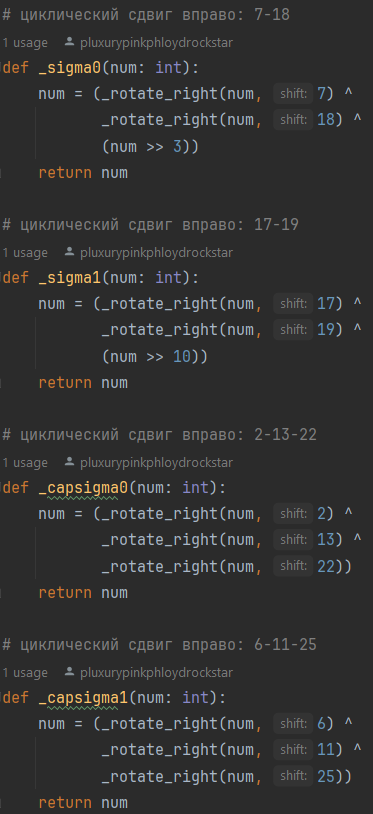


Рисунок 2 – Функции циклического сдвига.

Далее следуют функции, выполняющие необходимые логические операции.

1. Функция **\_ch(x: int, y: int, z: int)** принимает в качестве аргументов три целых числа: x, y и z.

Она выполняет следующие операции: побитовое И между x и y, побитовое ИЛИ между инвертированным x и z и побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ между результатами первых двух операций.

Результатом функции является полученное значение.

1. Функция **\_maj(x: int, y: int, z: int)** также принимает три целых числа в качестве аргументов: x, y и z.

Она выполняет следующие операции: побитовое И между x и y, побитовое И между x и z, побитовое И между y и z и побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ между результатами первых трех операций.

Результатом функции также является полученное значение.

Функции логических операций представлены на рисунке 3.

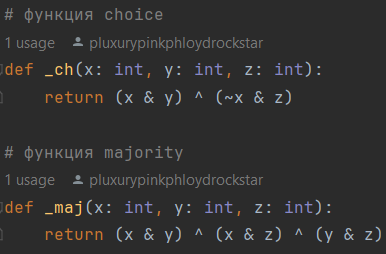


Рисунок 3 – Функции логических операций.

8) Функция **\_rotate\_right(num: int, shift: int, size: int = 32)** выполняет циклический сдвиг вправо для целого числа num и лежит в основе всех вышеописанных функций циклического сдвига.

Она принимает следующие параметры: num:Целое число, которое нужно сдвинуть, shift: Количество битов, на которое нужно выполнить сдвиг и size: Размер (в битах) целого числа (по умолчанию 32 бита).

Для выполнения сдвига: сначала выполняется сдвиг num вправо на shift битов с помощью оператора >>. Затем выполняется сдвиг num влево на size - shift битов с помощью оператора <<.

Результатом функции является побитовое ИЛИ между результатами обоих сдвигов.

В конце программы выполняются следующие действия: пользователем вводится текст и вызывается функция генерации хэш-кода (рисунок 4).

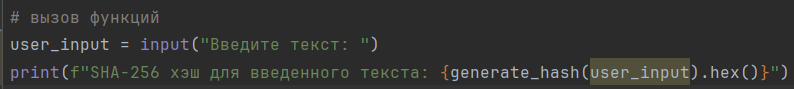


Рисунок 4 – Вызов соответствующих функций.

**Результаты работы программы:**

На рисунке 5 представлен результат работы вышеописанной программы.

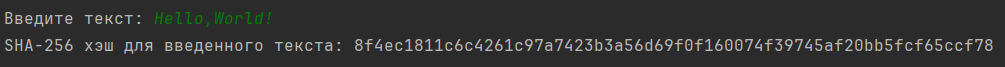


Рисунок 5– Результат работы программы.

На рисунке 6 представлен результат программы, написанной с помощью встроенной библиотеки языка на аналогичном тексте.



Рисунок 6 – Результат работы программы библиотечной реализации.

Полученные результаты совпадают, что свидетельствует о правильности работы программы.

**Листинг программы**

Файл LW\_11.py

# построение SHA-256 для введённого текста (алгоритмическая реализация)

# таблица констант

K = [

0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,

0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,

0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,

0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,

0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,

0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,

0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,

0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

]

# вычисление хэша SHA-256

def generate\_hash(message: bytearray) -> bytearray:

# проверка типа сообщения и преобразование в байтовый массив

if isinstance(message, str):

message = bytearray(message, 'utf-8')

elif isinstance(message, bytes):

message = bytearray(message)

elif not isinstance(message, bytearray):

raise TypeError

# Padding

length = len(message) \* 8

# добавление бита 0х80 (конец сообщения)

message.append(0x80)

# добавление нулей пока длина сообщения не кратна 512 бит (необходимая длина для обработки)

while (len(message) \* 8 + 64) % 512 != 0:

message.append(0x00)

# добавление длины сообщения в битах

message += length.to\_bytes(8, 'big')

# проверка завершения дополнения

assert (len(message) \* 8) % 512 == 0, "Дополнение(padding) не выполнено должным образом!"

# Parsing

blocks = []

# разбиение на блоки по 64 байта (512 бита)

for i in range(0, len(message), 64):

blocks.append(message[i:i+64])

# начальные значения хеш-функции

h0 = 0x6a09e667

h1 = 0xbb67ae85

h2 = 0x3c6ef372

h3 = 0xa54ff53a

h5 = 0x9b05688c

h4 = 0x510e527f

h6 = 0x1f83d9ab

h7 = 0x5be0cd19

# вычисление хэша

for message\_block in blocks:

message\_schedule = []

# 64 раунда

for t in range(0, 64):

if t <= 15:

# копирование первых 16 слов из блока сообщения

message\_schedule.append(bytes(message\_block[t\*4:(t\*4)+4]))

else:

# вычисление остальных сообщения (48) на основе предыдущих

term1 = \_sigma1(int.from\_bytes(message\_schedule[t-2], 'big'))

term2 = int.from\_bytes(message\_schedule[t-7], 'big')

term3 = \_sigma0(int.from\_bytes(message\_schedule[t-15], 'big'))

term4 = int.from\_bytes(message\_schedule[t-16], 'big')

# представление в виде 4-х байтового сообщения

schedule = ((term1 + term2 + term3 + term4) % 2\*\*32).to\_bytes(4, 'big')

message\_schedule.append(schedule)

# проверка длины блока сообщения

assert len(message\_schedule) == 64

# рабочие переменные

a = h0

b = h1

c = h2

d = h3

e = h4

f = h5

g = h6

h = h7

# преобразование переменных на итерациях от t=0 до 63

for t in range(64):

t1 = ((h + \_capsigma1(e) + \_ch(e, f, g) + K[t] +

int.from\_bytes(message\_schedule[t], 'big')) % 2\*\*32)

t2 = (\_capsigma0(a) + \_maj(a, b, c)) % 2\*\*32

# обновление переменных

h = g

g = f

f = e

e = (d + t1) % 2\*\*32

d = c

c = b

b = a

a = (t1 + t2) % 2\*\*32

# вычисление промежуточного кеша

h0 = (h0 + a) % 2\*\*32

h1 = (h1 + b) % 2\*\*32

h2 = (h2 + c) % 2\*\*32

h3 = (h3 + d) % 2\*\*32

h4 = (h4 + e) % 2\*\*32

h5 = (h5 + f) % 2\*\*32

h6 = (h6 + g) % 2\*\*32

h7 = (h7 + h) % 2\*\*32

# итоговый хэш сообщения - сумма элементов последнего блока сообщения

return ((h0).to\_bytes(4, 'big') + (h1).to\_bytes(4, 'big') +

(h2).to\_bytes(4, 'big') + (h3).to\_bytes(4, 'big') +

(h4).to\_bytes(4, 'big') + (h5).to\_bytes(4, 'big') +

(h6).to\_bytes(4, 'big') + (h7).to\_bytes(4, 'big'))

# вспомогательные функции

# операция циклического сдвига

def \_rotate\_right(num: int, shift: int, size: int = 32):

return (num >> shift) | (num << size - shift)

# циклический сдвиг вправо: 7-18

def \_sigma0(num: int):

num = (\_rotate\_right(num, 7) ^

\_rotate\_right(num, 18) ^

(num >> 3))

return num

# циклический сдвиг вправо: 17-19

def \_sigma1(num: int):

num = (\_rotate\_right(num, 17) ^

\_rotate\_right(num, 19) ^

(num >> 10))

return num

# циклический сдвиг вправо: 2-13-22

def \_capsigma0(num: int):

num = (\_rotate\_right(num, 2) ^

\_rotate\_right(num, 13) ^

\_rotate\_right(num, 22))

return num

# циклический сдвиг вправо: 6-11-25

def \_capsigma1(num: int):

num = (\_rotate\_right(num, 6) ^

\_rotate\_right(num, 11) ^

\_rotate\_right(num, 25))

return num

# функция choice

def \_ch(x: int, y: int, z: int):

return (x & y) ^ (~x & z)

# функция majority

def \_maj(x: int, y: int, z: int):

return (x & y) ^ (x & z) ^ (y & z)

# вызов функций

user\_input = input("Введите текст: ")

print(f"SHA-256 хэш для введенного текста: {generate\_hash(user\_input).hex()}")