УДК 004

Реализация алгоритма SHA256 на C++

Вихляев Дмитрий Романович
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данной статье рассмотрен криптографический алгоритм хэширования SHA-256. Программа написана с помощью языка программирования C++. Результатом исследования станет реализация алгоритма SHA-256 и пошаговое описание всех составляющих функций.
Ключевые слова: sha-256, криптография, хэш.

Implementation of the SHA256 algorithm in C++

Vikhlyaev Dmitry Romanovich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article discusses the SHA-256 cryptographic hashing algorithm. The program is written using the C++ programming language. The result of the study will be the implementation of the SHA-256 algorithm and a step-by-step description of all the constituent functions.
Keywords: sha-256, cryptography, hash.

1 Введение

1.1 Актуальность

SHA-256 (Secure Hash Algorithm 256-bit) - это криптографический алгоритм хеш-функции, который используется для генерации односторонних криптографических хеш-кодов. Он может генерировать хеш-коды длиной 256 бит, что обеспечивает высокую степень безопасности при обработке конфиденциальной информации.

Использование SHA-256 широко распространено в различных областях, таких как криптография, безопасность информации, электронный бизнес и т.д. Он используется для создания цифровых подписей, проверки целостности файлов, хранения паролей и других конфиденциальных данных. SHA-256 также используется в блокчейне, включая биткойн и другие криптовалюты, для защиты транзакций.

SHA-256 прост в использовании, поскольку он не требует большого количества дополнительной настройки, и он более быстрый и более точный, чем его предшественники.

1.2Обзор исследований

М.С.Пантелеев описал сохранение целостности данных с помощью sha256 на c# [1]. Д.А.Окатов, Т.А.Минеева реализовали алгоритм хеширования файлов sha256 на языке программирования c# [2]. И.Е.Пустыльник, Ю.П.Преображенский рассмотрели способы защиты сообщений между сервером и приборами интернета вещей [3]. В.И.Монахов, Б.А.Стрельников, И.В.Кузьмич, О.П.Степанова разработали службу аутентификации сообщений в компьютерной сети предприятия [4]. В.В.Вихман, М.А.Панков рассмотрели криптостойкость алгоритмов многоитерационного хэширования [5].

1.3Цель исследования

Цель исследования – реализовать и раскрыть структуру всех функций алгоритма sha-256, описать причины использования и результат внутренних методов.

2 Материалы и методы

Описание алгоритма построено на последовательной работе внутренних методов. Для реализации алгоритма используете язык программирования C++.

3 Результаты и обсуждения

Семейство SHA (Secure Hash Algorithm) в настоящее время обозначает семейство из 14 различных хэш-функций: SHA-0, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256, SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256.

Все реализации относятся к одному из четырёх стандартов:

- SHA-0 – исходная версия, опубликованная в 1993 году (в настоящее время является не безопасной).
- SHA-1 – исправленная версия SHA-0, опубликованная в 1995 году (в настоящее время является не безопасной).
- 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.
- SHA-3 – последняя версия семейства SHA, изменённая версия алгоритма Keccak, которая была выбранная в 2012 году после публичного конкурса среди разработчиков.

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.

SHA-256 обладает хорошей устойчивостью к атакам первого и второго прообраза. Атака первого прообраза – это задача заключается в нахождении данных, которые хэшируются в заданный хэш. SHA-256 обеспечивает устойчивость к этому типу атак благодаря высокой сложности обратного

вычисления и большому числу возможных входных данных. SHA-256 предотвращает атаку первого прообраза, потому что она является хэш-функцией сильного типа. Это означает, что вычисление обратного значения (т.е. нахождение входного значения, которое дает заданное хэш-значение) является вычислительно невозможным без перебора всех возможных входных значений. Это делает атаку методом подбора (брутфорс) непрактичной. Атака второго прообраза – задача заключается в нахождении двух разных наборов данных, которые дают одинаковый хэш. Устойчивость SHA-256 к такой атаке обеспечивается благодаря лавинному эффекту и большому пространству хэшей, что затрудняет нахождение коллизий. SHA-256 также предотвращает атаку второго прообраза, используя функцию хэширования сильного типа и увеличивая размер выходного хэш-значения до 256 бит. Это означает, что вероятность нахождения другого входного значения, которое дает то же самое хэш-значение, что и заданное входное значение, крайне мала. Таким образом, атака второго прообраза также становится непрактичной.

Хеш-функция построена на основе структуры Меркла-Дамгора. Создателями данной структуры было доказано, что если односторонняя функция сжатия f устойчива к коллизиям, то и хеш-функция, построенная на ее основе, также устойчива к коллизиям.

При поступлении в функцию сообщения произвольной длины, результатом работы функции, должно быть сообщение фиксированной длины. Для достижения такого результата входное сообщение разбивается на блоки одинакового размера. Далее в порядке очереди функция сжатия перерабатывает входной блок фиксированной длины в более короткое выходное сообщение фиксированной длины. На вход функция принимает новый блок и сообщение полученное от предыдущего преобразования. В первой итерации вместо преобразованного сообщения, подставляется сообщение фиксированного размера, называемое вектором инициализации.

Ниже представлено изображение односторонней функции сжатия (f), преобразующей два входных блока фиксированной длины в выходной блок фиксированной длины. Алгоритм начинается с начального значения или вектора инициализации (IV) (рис.1).

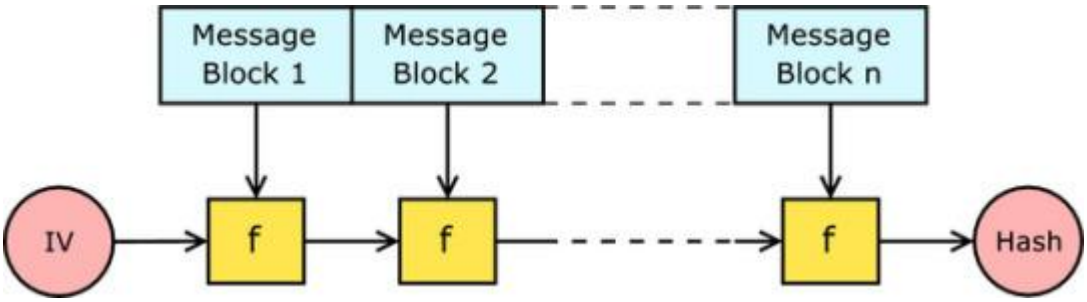

Рис. 1. Структура Меркла - Дамгора

Вектор инициализации – это некоторое заранее определённое значение фиксированного размера, в рассматриваемом алгоритме состоит из восьми констант по 32 бита (общая длина 256 бит).

- h0 = 0x6a09e667
- h1 = 0xbb67ae85
- h2 = 0x3c6ef372
- h3 = 0xa54ff53a
- h4 = 0x510e527f
- h5 = 0x9b05688c
- h6 = 0x1f83d9ab
- h7 = 0x5be0cd19

Константы в алгоритме SHA-256 были выбраны на основе квадратичных корней первых восьми простых чисел и первых 32 бит дробных частей кубических корней первых 64 простых чисел. Эти константы были выбраны таким образом, чтобы обеспечить высокую степень случайности в процессе хеширования и устойчивость к различным атакам на алгоритм. Выбор констант был основан на математических принципах и проведен после тщательного анализа различных вариантов. Пример получения h0:

Первым простым числом является – 2. Квадратный корень из 2 в двоичном 64 битном виде представлен на рисунке ниже. Необходимо найти дробную часть и выделить из неё первые 32 бита. Полученное число в шестнадцатеричном виде соответствует 0x6a09e667 (рис.2).

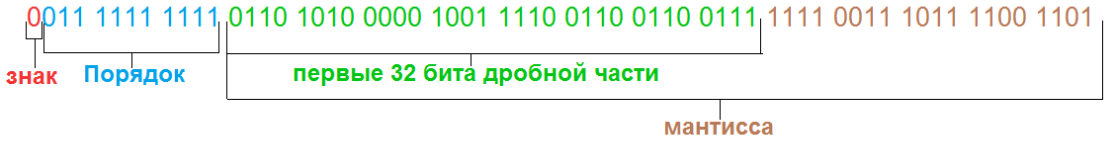

Рис. 2. Квадратный корень из 2, представленный в двоичном виде

Блоки SHA-256 имеют фиксированный размер 512 бит в порядке расположения байтов "big-endian". Если длина сообщения превышает размеры блока, создаются дополнительные блоки, до тех пор, пока сообщение полностью не разместиться. Если сообщение полностью не заполняет блок, необходимо дополнить последний блок до полного, определёнными данными (обычно нулями). Однако есть вероятность, что различные сообщения, начинающиеся одними и теми же символами, и заканчивающимися нулями или другими байтами из заполнителя, будут поступать в функцию сжатия совершенно одинаковыми блоками. В результате чего получиться одинаковая хэш-сумма. Избежать этого помогает вставка иного значения первого бита заполнителя. Так как заполнитель обычно состоит из нулей, первый бит заполнителя должен быть заменён на «1». В конце последнего блока зарезервировано 8 байт для длины исходного сообщения в битах. Алгоритм работает с каждым блоком по отдельности. Ниже приведён пример сообщения размером в 440 бит и изображение (рис.3) заполненного блока.

Сообщение:
«ABCDEFGHIJKLMNOPQRASTUVWXYZabcdifghijklmnopqrstuvwxyz01»

```
01000001 01000010 01000011 01000100
01000101 01000110 01000111 01001000
01001001 01001010 01001011 01001100
01001101 01001110 01001111 01010000
01010001 01010010 01000001 01010011
01010100 01010101 01010110 01010111
01011000 01011001 01011010 01100001
01100010 01100011 01100100 01101001
01100110 01100111 01101000 01101001
01101010 01101011 01101100 01101101
01101110 01101111 01110000 01110001
01110010 01110011 01110100 01110101
01110110 01110111 01111000 01111001
01111010 00110000 00110001 10000000
00000000 00000000 00000000 00000000
00000000 00000000 00000001 10111000
```

Рис. 3. Блок с размером сообщения 440 бит

Пример входного сообщения размером в 448 бит:
«ABCDEFGHIJKLMNOPQRASTUVWXYZabcdifghijklmnopqrstuvwxyz012»
(рис.4,5)

```
01000001 01000010 01000011 01000100
01000101 01000110 01000111 01001000
01001001 01001010 01001011 01001100
01001101 01001110 01001111 01010000
01010001 01010010 01000001 01010011
01010100 01010101 01010110 01010111
01011000 01011001 01011010 01100001
01100010 01100011 01100100 01101001
01100110 01100111 01101000 01101001
01101010 01101011 01101100 01101101
01101110 01101111 01110000 01110001
01110010 01110011 01110100 01110101
01110110 01110111 01111000 01111001
01111010 00110000 00110001 00110010
10000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

Рис. 4. Первый блок с размером сообщения 448 бит

Рис. 5. Второй блок с размером сообщения 448 бит

Из приведённых выше примеров видно, что на практике минимальный размер сообщения для использования только одного блока составляет 440 бит. Связано это с тем, что минимальной единицей, хранящейся в памяти, является байт (символы, как и другие типы данных, могут занимать место кратное 8 битам). Соответственно поэтому при добавлении ещё одного символа, в блоке не остаётся места под 64 битную информацию о длине сообщения. Из-за чего создаётся второй блок, запененный нулями.

На следующих шагах алгоритм работает индивидуально с каждым блоком по очереди.

На данном этапе все блоки имеют начальный вид, представленный в виде массива, состоящего из 16 слов (от 0 до 15) по 32 бита. Полный размер блока в SHA-256 составляет 64 слова.

Ниже представлена функция на языке программирования C++, которая формирует и заполняет блоки (рис.6).

```
//количество символов в строке
int64_t lenstr = message.size();
//раззмер поля длины в байтах
uint8_t field_len = 8;
//количество бит в строке
uint64_t bitstr = lenstr * 8;

//полезная нагрузка в байтах
uint64_t payload = message.size() + field_len;
//заполнитель в батах
uint64_t padding = 64 - payload % 64;
//количесво блоков
uint64_t block_count = payload / 64 + 1;

//инициализация блоков
std::vector<std::array <uint32_t, 64>> blocks(block_count);

//заполнение блоков
for (int64_t i = 0; i < block_count; i++)
{
    //массив каждого блока по 32 бита представляется как массив по 8 бит
    char* byte_block = (char*)&(blocks[i]);

    //обход элементов массива
    for (int j = 0 ; j < 64; j++)
    {
        if (lenstr == 0)
        {
            //заполнение блока отступами
            if (padding!=0)
            {
                *(byte_block + j) = 0x00;
                padding--;
            }
            //заполнение поля - "длины сообщения"
            else
            {
                field_len--;
                *(byte_block + j) = bitstr >> (8 * field_len);
            }
        }
        else
        {
            //заполнение массива символоми сообщения
            *(byte_block + j) = message[j + i * 64];

            lenstr--;

            //если сообщение закончилось, вставляется первый байт заполнителя с "1" в начале
            if (lenstr == 0)
            {
                j++;
                *(byte_block + j) = 0x80;
                padding--;
            }
        }
    }
}
```

Рис. 6. Создание блоков

Для большинства машин, данный код заполонит блоки в порядке "little-endian", поэтому необходимо будет конвертировать каждое слово в порядок "big-endian" (рис.7).

```
void little_to_big_convert(uint32_t &word)
{
    word = (word >> 24) | ((word >> 8) & 0xff00)
        | ((word << 8) & 0xff0000) | (word << 24);
}
```

Рис. 7. Перевод байтов в порядок big-endian

На данном этапе каждый блок представляется в виде массива состоящего из 16 слов (от 0 до 15) по 32 бита.

Далее происходит расширение блока до 64 слов. Формирование каждого последующего слова происходит согласно структуре представленной ниже (рис.8,9).

1. w[n] = w[n-16] +c0+w[n-7]+c1
   - 1.1. w – массив слов
   - 1.2. n – номер индекса слова
   - 1.3. (+) – оператор сложения (не по модулю 2, происходит переполнение)
   - 1.4. c0 = right_rotate(w[n-15], 7) ^ right_rotate(w[n-15], 18) ^ right_shift(w[n-15], 3)
   - 1.5. c1 = right_rotate(w[n-2], 17) ^ right_rotate(w[n-2], 19) ^ right_shift(w[n-2], 10)
     - 1.5.1. right_rotate – функция реализующая правое вращение бит, принимающая 32-битное слово и центр вращения данного слова
     - 1.5.2. right_shift – функция побитового сдвига в право, принимающая 32-битное слово и количество сдвигаемых бит
     - 1.5.3. (^) – операция xor

Рис. 8. Структура методов расширения блока

```
uint32_t RightRotate32(uint32_t n, uint32_t d)
{
    return (n >> d) | (n << (32 - d));
}

uint32_t C0(uint32_t word)
{
    return RightRotate32(word, 7) ^ RightRotate32(word, 18) ^ (word >> 3);
}

uint32_t C1(uint32_t word)
{
    return RightRotate32(word, 17) ^ RightRotate32(word, 19) ^ (word >> 10);
}

void ExpensionBlock(std::array<uint32_t, 64> &block)
{
    for (int i = 16; i < 64; i++)
    {
        block[i] = block[i - 16] + C0(block[i - 15]) + block[i - 7] + C1(block[i - 2]);
    }
}
```

Рис. 8. Реализация функций для расширения блока

Алгоритм пропускает каждый блок сообщения через цикл с 64 итерациями (раундами). В каждом раунде учувствуют слова из блока попарно с новым массивом констант по порядку.

Новые константы схожи по построению с начальным значением хэша, но на этот раз это первые 32 бита дробных частей кубических корней первых 64 простых чисел от 2 до 311.

Помимо них в каждом раунде принимают участие переменные, на старте инициализированные начальным значением хэша:

a = h0, b = h1, c = h2, d = h3, e = h4, f = h5, g = h6, h = h7

В конце каждого раунда значение этих переменных будет отличаться от их значения в начале раунда, благодаря множественным преобразованиям.

Преобразования, используемые на каждой итерации цикла для одного блока – это и есть функция сжатия Sha-256.

Преобразование переменных на каждой итерации одинаковые и осуществляется согласно структуре приведённой ниже (рис.9).

1. h = g
2. g = f
3. f = e
4. e = d + Temp1
5. d = c
6. c = b
7. b = a
8. a = Temp1 + Temp2
    1. Temp1 = h + Σ1 + Choice + k[i] + w[i]
    2. Temp2 = Σ0 + Majority
        1. k – массив констант
        2. w – массив слов из блока
        3. 1.3. i – итерация цикла,
        4. Σ0 = right_rotate(a, 2) ^ right_rotate(a, 13) ^ right_rotate(a, 22)
        5. Σ1= right_rotate(a, 6) ^ right_rotate(a, 11) ^ right_rotate(a, 25)
        6. Choice = (e & f) ^ ((!e) & g)
        7. Majority = (a & b) ^ (a & c) ^ (b & c),
            1. ! – инверсия бит
            2. & – побитовый оператор «и»

Рис. 9. Иерархическая структура методов в функции сжатия

Функция CH (Choose) принимает на вход три 32-битных слова (x, y и z) и выводит 32-битное слово. Он применяет логическую операцию «И» к первым двум словам и логическую операцию «НЕ» к первому слову, а затем применяет логическую операцию «ИЛИ» к результатам. Другими словами, функция CH выводит результат выбора либо второго слова, либо отрицания первого слова, в зависимости от значения третьего слова.

Функция MAJ (Majority) также принимает на вход три 32- битных слова (x, y и z) и выводит 32-битное слово. Он применяет логическую операцию «И» ко всем трем словам, а затем применяет логическую операцию «ИСКЛЮЧАЮЩЕЕ ИЛИ» ко второму и третьему словам. Другими словами, функция MAJ выводит результат наибольшего значения среди трех входных слов.

Функция SIG0 обозначает "первую" функцию для вычисления малых букв сигма. Применяется к словам сообщения в основном цикле алгоритма для генерации новых слов. На вход принимает 32-битное слово. ROTR и SHR – это операции циклического сдвига вправо и логического сдвига вправо соответственно. XOR — это операция исключающего ИЛИ. Цель SIG0 заключается в создании "смешивания" и "распространении" битов входного слова, что увеличивает сложность обратного проектирования хеш-функции и повышает криптографическую стойкость алгоритма.

Функция "sig1" используется для создания нового значения промежуточного хеша перед следующим раундом. Она принимает на вход три 32-битных целых числа (A, B и C), которые представляют собой промежуточное состояние хеш-функции, и вычисляет новое значение промежуточного хеша. Она используется как один из способов обеспечения криптографической стойкости, расширяя информацию о сообщении и обеспечивая хорошее равномерное распределение бит в промежуточном хеше, что делает его более стойким к атакам.

Лавинный эффект является ключевым свойством криптографических хэш-функций, включая SHA-256. Этот эффект означает, что даже малейшие изменения во входных данных приводят к существенному изменению результата хэш-функции. В SHA-256 лавинный эффект достигается с помощью вышеперечисленных методов. Битовые операции, такие как сдвиги, вращение и исключающее ИЛИ (XOR), которые применяются к данным и константам в процессе обработки блоков. Применение нелинейных функций, таких как Sigma и Sum, которые выполняют взаимодействие битов соседних блоков. Процесс преобразования данных включает множество раундов, что увеличивает сложность алгоритма и обеспечивает его устойчивость к атакам. Использование операций XOR, AND и OR для комбинирования битов входных данных. Применение цикла функций, которые многократно применяются к блокам входных данных, чтобы создать окончательный хэш-значение. Каждый цикл функций включает в себя несколько этапов, включая перемешивание (mixing), подстановку (substitution) и перестановку (permutation) битов. Использование нескольких раундов (64 в случае SHA-256), чтобы усилить лавинный эффект и создать более сложную функцию.

Значения, которые были получены на последней итерации цикла, являются хэшем данного блока, они становятся новыми значениями вектора инициализации для следующего блока. h0+=a, h1+=b, h2+=c, h3+=d, h4+=e, h5+=f, h6+=g, h7+=h.

Если данный блок был последним, то сумма элементов этого вектора станет итоговым хэшем сообщения.

h0+ h1+ h2+ h3+ h4+ h5+ h6+ h7=hash_sha-256

На рисунках 10, 11 представлены реализации функции сжатия и вспомогательных методов.

```
void compress(std::array<uint32_t, 64> w, std::array<uint32_t, 8> &H)
{
    const std::array<uint32_t, 64> 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
    };

    uint32_t a = H[0];
    uint32_t b = H[1];
    uint32_t c = H[2];
    uint32_t d = H[3];
    uint32_t e = H[4];
    uint32_t f = H[5];
    uint32_t g = H[6];
    uint32_t h = H[7];

    for (size_t i = 0; i < 64; i++)
    {
        uint32_t Temp1 = h + Sum1(e) + Choice(e,f,g) + K[i] + w[i];
        uint32_t Temp2 = Sum0(a) + Majority(a, b, c);

        h = g;
        g = f;
        f = e;
        e = d + Temp1;
        d = c;
        c = b;
        b = a;
        a = Temp1 + Temp2;
    }

    H[0]+=a;
    H[1]+=b;
    H[2]+=c;
    H[3]+=d;
    H[4]+=e;
    H[5]+=f;
    H[6]+=g;
    H[7]+=h;
}
```

Рис. 10. Функция сжатия

```
uint32_t Sum0(uint32_t a)
{
    return RightRotate32(a, 2) ^ RightRotate32(a, 13) ^ RightRotate32(a, 22);
}
uint32_t Sum1(uint32_t e)
{
    return RightRotate32(e, 6) ^ RightRotate32(e, 11) ^ RightRotate32(e, 25);
}

uint32_t Choice(uint32_t e, uint32_t f, uint32_t g)
{
    return (e & f) ^ ((~e) & g);
}

uint32_t Majority(uint32_t a, uint32_t b, uint32_t c)
{
    return (a & b) ^ (a & c) ^ (b & c);
}
```

Рис. 11. Вспомогательные функции

Функция вызова всех методов представлена на рисунке 12.

```
std::array<uint32_t, 8> Hash(std::string message)
{
    //вектор инициализации
    std::array<uint32_t, 8> h = {
        0x6a09e667,0xbb67ae85,0x3c6ef372,0xa54ff53a,
        0x510e527f,0x9b05688c,0x1f83d9ab,0x5be0cd19
    };

    auto blocks = CreateBlock(message);

    for (auto& block : blocks)
    {
        compress(block, h);
    }

    return h;
}
```

Рис. 12. Главная функция хэширования

В качестве примера для наблюдения изменения значений хэша взято сообщение длинной 448 бит «ABCDEFGHIJKLMNOPQRASTUVWXYZabcdifghijklmnopqrstuvwxyz012»

Вектор инициализации, поступающий во второй блок, содержит следующие значения:

- ho= 0x6b21d0db
- h1= 0x78b657db
- h2= 0x0e59599a

- h3= 0xd0d73fa5
- h4= 0x5f3a6d2d
- h5= 0xabf6e8d5
- h6= 0x1d443f62
- h7= 0x227abf9a

После обработки второго блока (последнего для данного сообщения), значения хэша содержат следующие значения:

- ho= 0x8da42cf0
- h1= 0x8db5e96a
- h2= 0x775d9620
- h3= 0x2fd22673
- h4= 0x16604e5e
- h5= 0xcc0cdb2d
- h6= 0x92ff4d60
- h7= 0xc65d3e36

Итоговый хэш представляется как склеивание всех значений 0x8da42cf08db5e96a775d96202fd2267316604e5ecc0cdb2d92ff4d60c65d3e36.

Выводы

Таким образом, была подробно описана программная реализация алгоритма sha-256. Также рассмотрены функции и их значимость в данном алгоритме.

Библиографический список

1. Пантелеев М.С. Целостность данных и реализация sha256 на c# // Современные тенденции развития науки и технологий. 2016. № 12-4. С. 71-72.
2. Окатов Д.А., Минеева Т.А. Реализация алгоритма хеширования файлов sha256 на языке программирования c# // Тенденции развития науки и образования. 2022. № 82-2. С. 42-45.
3. Пустыльник И.Е., Преображенский Ю.П. Защита сообщений между сервером и приборами интернета вещей // Вестник Воронежского института высоких технологий. 2019. № 2 (29). С. 40-45.
4. Монахов В.И., Стрельников Б.А., Кузьмич И.В., Степанова О.П. Служба аутентификации сообщений в компьютерной сети предприятия // В сборнике: Современные задачи инженерных наук. сборник научных трудов Международного научно-технического симпозиума. 2017. С. 113-117.
5. Вихман В.В., Панков М.А. Криптостойкость алгоритмов многоитерационного хэширования // В сборнике: Информационные технологии в экономике, образовании и бизнесе. Институт управления и социально-экономического развития, Саратовский государственный

технический университет, Richland College (Даллас, США). 2014. С. 46-56.
6. Алтухова В.А., Анфилова Е.Б., Тезик К.А. Алгоритмы хэширования ГОСТ Р 34.11-2012 И SHA-3 // В сборнике: Научное и образовательное пространство: перспективы развития. сборник материалов II Международной научно-практической конференции. 2016. С. 259-260.
7. Логвинов Д.В., Савкин С.С. Сравнительный анализ популярных алгоритмов хэширования // В сборнике: Вызовы глобализации и развитие цифрового общества в условиях новой реальности. сборник материалов IV Международной научно-практической конференции. Москва, 2022. С. 119-122.
8. Магамедова А.З., Магамедова Д.М. Хеш-функции в теории и на практике // В сборнике: Информационные технологии в бизнесе и образовании. I Студенческая научно-практическая конференция. Грозный, 2020. С. 43-46.