Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Основы алгоритмизации и программирования

Отчёт

по учебной практике(ознакомительной)

на тему

Реализация алгоритма хэширования SHA-256

Студент Манухо Д.А

Руководитель Ковшер Е. И

МИНСК 2024

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ ........................................................................................................... 2

1 ОПИСАНИЕ АЛГОРИТМА ........................................................................... 3

2 КОД ПРОГРАММЫ ........................................................................................ 5

3 РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ ................................................... 9

ЗАКЛЮЧЕНИЕ ................................................................................................... 10

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ ..................................... 11

ПРИЛОЖЕНИЕ А ............................................................................................. 12

**ВВЕДЕНИЕ**

Хеш-функция или функция свёртки — функция, преобразующая массив входных данных произвольного размера в выходную битовую строку определённого (установленного) размера в соответствии с определённым алгоритмом. Преобразование, выполняемое хеш-функцией, называется хешированием. Исходные (входные) данные называются входным массивом, «ключом», «сообщением». Результат преобразования (выходные данные) называется «хешем», «хеш-кодом», «хеш-суммой», «сводкой сообщения», «свёрткой».

Хеш-функции применяются в следующих случаях:

* при построении ассоциативных массивов;
* при поиске дубликатов в последовательностях наборов данных;
* при построении уникальных идентификаторов для наборов данных;
* при вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в данных ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
* при сохранении паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
* при создании (выработке) электронной подписи (на практике подпись часто создаётся не для сообщения, а для «хеш-образа» сообщения);
* и в других случаях.

В общем случае не существует однозначного соответствия между выходными входными данными Выходные данные (возвращаемые хеш-функцией значения) менее разнообразны, чем входные данные (значения входного массива). Случай, при котором хеш-функция преобразует более чем одни входные данные (один массив входных данных) в одинаковые выходные данные (сводки), называется «коллизией». Вероятность возникновения коллизий используется для оценки качества хеш-функций.

Хеш-функции семейства SHA-2 построены на основе структуры Меркла — Дамгора(разбиение входных сообщений произвольной длины на блоки фиксированной длины и работающий с ними по очереди с помощью функции сжатия, каждый раз принимая входной блок с выходным от предыдущего прохода). Исходное сообщение после дополнения разбивается на блоки, каждый блок — на 16 слов. Алгоритм пропускает каждый блок сообщения через цикл с 64 или 80 итерациями (раундами). На каждой итерации 2 слова преобразуются, функцию преобразования задают остальные слова. Результаты обработки каждого блока складываются, сумма является значением хеш-функции. Тем не менее, инициализация внутреннего состояния производится результатом обработки предыдущего блока. Поэтому независимо обрабатывать блоки и складывать результаты нельзя

**1.ОПИСАНИЕ АЛГОРИТМА**

SHA-256 (Secure Hash Algorithm 256) — это криптографическая хэш-функция, входящая в семейство алгоритмов SHA-2, разработанная Национальным институтом стандартов и технологий (NIST). Алгоритм принимает на вход сообщение произвольной длины и возвращает фиксированную 256-битную (32-байтную) строку, которая является криптографическим хэшом входного сообщения.

Этапы хэширования:

1.Подготовка сообщения (Padding)

Сообщение сначала дополняется до кратной 512 бит длины следующим образом: добавляется единичный бит 1 а также добавляются нули до тех пор, пока длина сообщения не станет на 64 бита меньше ближайшего кратного 512. Пример: если исходная длина сообщения l бит, то итоговая длина сообщения после дополнения будет (l + 1 + k + 64), где k — количество добавленных нулей.

2.Добавление длины сообщения

В конец дополненного сообщения добавляется 64-битное (8 байт) представление исходной длины сообщения. Эти 64 бита записываются в формате big-endian.

3. Инициализация хэш-значений

Устанавливаются начальные значения восьми 32-битных слов (h0, h1, h2, h3, h4, h5, h6, h7), которые являются фиксированными константами, определенными в спецификации SHA-256:

h0 = 0x6a09e667

h1 = 0xbb67ae85

h2 = 0x3c6ef372

h3 = 0xa54ff53a

h4 = 0x510e527f

h5 = 0x9b05688c

h6 = 0x1f83d9ab

h7 = 0x5be0cd19

4. Инициализация констант

Определяется массив из 64 фиксированных 32-битных констант (k0, k1, ..., k63), используемых в основном цикле алгоритма. Эти константы также заданы в спецификации SHA-256.

5. Обработка блоков сообщения

Сообщение разбивается на блоки по 512 бит (64 байта). Каждый блок обрабатывается следующим образом:

6. Инициализация рабочего массива W

Рабочий массив W состоит из 64 32-битных слов. Первые 16 слов (W0 - W15) инициализируются исходными значениями блока. Остальные 48 слов (W16 - W63) вычисляются по следующей формуле:

W[i] = σ1(W[i-2]) + W[i-7] + σ0(W[i-15]) + W[i-16]

где σ0 и σ1 — побитовые операции сдвига и ротации.

7. Основной цикл (64 шага)

Основной цикл состоит из 64 шагов, в каждом из которых выполняются следующие операции:

T1 = h + Σ1(e) + Ch(e, f, g) + k[i] + W[i]

T2 = Σ0(a) + Maj(a, b, c)

h = g

g = f

f = e

e = d + T1

d = c

c = b

b = a

a = T1 + T2

где:

Σ0 и Σ1 — побитовые операции сдвига и ротации.

Ch и Maj — функции выбора и большинства, соответственно.

8. Обновление хэш-значений

После обработки каждого блока значения (h0, h1, ..., h7) обновляются следующим образом:

h0 = a + h0

h1 = b + h1

h2 = c + h2

h3 = d + h3

h4 = e + h4

h5 = f + h5

h6 = g + h6

h7 = h + h7

9. Формирование итогового хэша

После обработки всех блоков сообщения итоговое значение хэша формируется путем конкатенации всех восьми хэш-значений (h0, h1, ..., h7). Итоговое значение хэша имеет длину 256 бит (32 байта).

**2.КОД ПРОГРАММЫ**

1. Файл sha256.h

#include <stddef.h>

#include <stdint.h>

const int SHA256\_HEX\_SIZE = 64 + 1;

const int SHA256\_BYTES\_SIZE = 32;

void sha256\_hex(const void \*src, size\_t n\_bytes, char \*dst\_hex65);

void sha256\_bytes(const void \*src, size\_t n\_bytes, void \*dst\_bytes32);

typedef struct sha256 {

uint32\_t state[8];

uint8\_t buffer[64];

uint64\_t n\_bits;

uint8\_t buffer\_counter;

} sha256;

void sha256\_init(struct sha256 \*sha);

void sha256\_append(struct sha256 \*sha, const void \*data, size\_t n\_bytes);

void sha256\_finalize\_hex(struct sha256 \*sha, char \*dst\_hex65);

void sha256\_finalize\_bytes(struct sha256 \*sha, void \*dst\_bytes32);

2. Файл main.c

#include "sha256.h"

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#define ROTR(x, n) (((x) >> (n)) | ((x) << (32 - (n))))

uint32\_t step1(uint32\_t e, uint32\_t f, uint32\_t g) {

return (ROTR(e, 6) ^ ROTR(e, 11) ^ ROTR(e, 25)) + ((e & f) ^ ((~e) & g));

}

uint32\_t step2(uint32\_t a, uint32\_t b, uint32\_t c) {

return (ROTR(a, 2) ^ ROTR(a, 13) ^ ROTR(a, 22)) +

((a & b) ^ (a & c) ^ (b & c));

}

void update\_w(uint32\_t \*w, int i, const uint8\_t \*buffer) {

int j;

for (j = 0; j < 16; j++) {

if (i < 16) {

w[j] = ((uint32\_t)buffer[0] << 24) | ((uint32\_t)buffer[1] << 16) |

((uint32\_t)buffer[2] << 8) | ((uint32\_t)buffer[3]);

buffer += 4;

} else {

uint32\_t a = w[(j + 1) & 15];

uint32\_t b = w[(j + 14) & 15];

uint32\_t s0 = (ROTR(a, 7) ^ ROTR(a, 18) ^ (a >> 3));

uint32\_t s1 = (ROTR(b, 17) ^ ROTR(b, 19) ^ (b >> 10));

w[j] += w[(j + 9) & 15] + s0 + s1;

}

}

}

void sha256\_block(struct sha256 \*sha) {

uint32\_t \*state = sha->state;

static const uint32\_t k[8 \* 8] = {

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 = state[0];

uint32\_t b = state[1];

uint32\_t c = state[2];

uint32\_t d = state[3];

uint32\_t e = state[4];

uint32\_t f = state[5];

uint32\_t g = state[6];

uint32\_t h = state[7];

uint32\_t w[16];

int i, j;

for (i = 0; i < 64; i += 16) {

update\_w(w, i, sha->buffer);

for (j = 0; j < 16; j += 4) {

uint32\_t temp;

temp = h + step1(e, f, g) + k[i + j + 0] + w[j + 0];

h = temp + d;

d = temp + step2(a, b, c);

temp = g + step1(h, e, f) + k[i + j + 1] + w[j + 1];

g = temp + c;

c = temp + step2(d, a, b);

temp = f + step1(g, h, e) + k[i + j + 2] + w[j + 2];

f = temp + b;

b = temp + step2(c, d, a);

temp = e + step1(f, g, h) + k[i + j + 3] + w[j + 3];

e = temp + a;

a = temp + step2(b, c, d);

}

}

state[0] += a;

state[1] += b;

state[2] += c;

state[3] += d;

state[4] += e;

state[5] += f;

state[6] += g;

state[7] += h;

}

void sha256\_init(struct sha256 \*sha) {

sha->state[0] = 0x6a09e667;

sha->state[1] = 0xbb67ae85;

sha->state[2] = 0x3c6ef372;

sha->state[3] = 0xa54ff53a;

sha->state[4] = 0x510e527f;

sha->state[5] = 0x9b05688c;

sha->state[6] = 0x1f83d9ab;

sha->state[7] = 0x5be0cd19;

sha->n\_bits = 0;

sha->buffer\_counter = 0;

}

void sha256\_append\_byte(struct sha256 \*sha, uint8\_t byte) {

sha->buffer[sha->buffer\_counter++] = byte;

sha->n\_bits += 8;

if (sha->buffer\_counter == 64) {

sha->buffer\_counter = 0;

sha256\_block(sha);

}

}

void sha256\_append(struct sha256 \*sha, const void \*src, size\_t n\_bytes) {

const uint8\_t \*bytes = (const uint8\_t \*)src;

size\_t i;

for (i = 0; i < n\_bytes; i++) {

sha256\_append\_byte(sha, bytes[i]);

}

}

void sha256\_finalize(struct sha256 \*sha) {

int i;

uint64\_t n\_bits = sha->n\_bits;

sha256\_append\_byte(sha, 0x80);

while (sha->buffer\_counter != 56) {

sha256\_append\_byte(sha, 0);

}

for (i = 7; i >= 0; i--) {

uint8\_t byte = (n\_bits >> 8 \* i) & 0xff;

sha256\_append\_byte(sha, byte);

}

}

void sha256\_finalize\_hex(struct sha256 \*sha, char \*dst\_hex65) {

int i, j;

sha256\_finalize(sha);

for (i = 0; i < 8; i++) {

for (j = 7; j >= 0; j--) {

uint8\_t nibble = (sha->state[i] >> j \* 4) & 0xf;

\*dst\_hex65++ = "0123456789abcdef"[nibble];

}

}

\*dst\_hex65 = '\0';

}

void sha256\_finalize\_bytes(struct sha256 \*sha, void \*dst\_bytes32) {

uint8\_t \*ptr = (uint8\_t \*)dst\_bytes32;

int i, j;

sha256\_finalize(sha);

for (i = 0; i < 8; i++) {

for (j = 3; j >= 0; j--) {

\*ptr++ = (sha->state[i] >> j \* 8) & 0xff;

}

}

}

void sha256\_hex(const void \*src, size\_t n\_bytes, char \*dst\_hex65) {

struct sha256 sha;

sha256\_init(&sha);

sha256\_append(&sha, src, n\_bytes);

sha256\_finalize\_hex(&sha, dst\_hex65);

}

void sha256\_bytes(const void \*src, size\_t n\_bytes, void \*dst\_bytes32) {

struct sha256 sha;

sha256\_init(&sha);

sha256\_append(&sha, src, n\_bytes);

sha256\_finalize\_bytes(&sha, dst\_bytes32);

}

int main(int argc, char \*argv[]) {

if (argc != 2) {

printf("Usage: %s <input\_string>\n", argv[0]);

return 1;

}

const char \*text = argv[1];

char hex[SHA256\_HEX\_SIZE];

sha256\_hex(text, strlen(text), hex);

FILE \*file = fopen("hash.txt", "w");

if (file == NULL) {

printf("Error opening file!\n");

return 1;

}

fprintf(file, "%s", hex);

fprintf(file, "\n");

fclose(file);

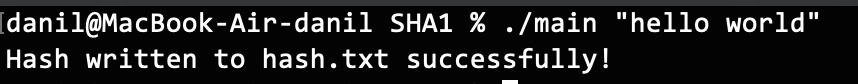
printf("Hash written to hash.txt successfully!\n");

return 0;

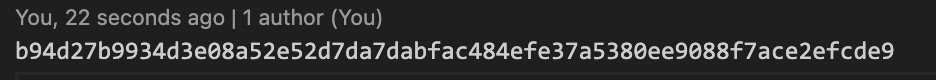
}

**3 РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ**

При запуске программы мы в консоли мы видим сообщение об успешном завершении программы, что наши выходные данные (хэш) записан в файл hash.txt



В самом файле мы видим захэшированный текст по алгоритму SHA-256



Далее можно использовать этот хэш уже из файла в любых целях разработчиков и программистов.

**ЗАКЛЮЧЕНИЕ**

В ходе данной работы был реализован алгоритм

Алгоритмы хэширования, включая SHA-256, находят широкое применение в различных областях информационных технологий. Основные сферы применения включают:

Информационная безопасность:

* Целостность данных: Хэш-функции используются для проверки целостности данных, предотвращая их несанкционированное изменение. Это особенно важно в контексте хранения и передачи данных.
* Аутентификация: SHA-256 применяется для создания цифровых подписей, которые обеспечивают проверку подлинности отправителя сообщения или документа.

Криптография:

* Создание и проверка цифровых сертификатов: Хэш-функции используются в процессе создания и проверки цифровых сертификатов, которые подтверждают подлинность веб-сайтов и программного обеспечения.

Блокчейн и криптовалюты:

* Майнинг и консенсус: В блокчейн-технологиях, таких как Bitcoin, SHA-256 используется в процессе майнинга для создания новых блоков и достижения консенсуса в сети.
* Хэширование транзакций: Все транзакции в блокчейне хэшируются с использованием SHA-256 для обеспечения их неизменности и безопасности.

Передача данных:

* Проверка целостности файлов: При передаче файлов через интернет хэш-функции используются для проверки целостности загружаемых файлов, сравнивая хэш полученного файла с хэшем оригинала.
* Электронная почта: В системах электронной почты хэширование используется для проверки подлинности отправителя и целостности сообщения.

Хранение данных:

* Обезличивание данных: Хэш-функции могут использоваться для обезличивания данных в базах данных, что особенно важно в контексте защиты персональных данных.
* Управление версиями файлов: В системах контроля версий хэш-функции применяются для идентификации и отслеживания изменений в файлах.

Алгоритм SHA-256 благодаря своей устойчивости к коллизиям и высокой вычислительной эффективности остается одним из наиболее надежных и широко используемых инструментов в области информационной безопасности и криптографии.

**СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ**

SHA2 [Электронный ресурс]. – https://ru.wikipedia.org/wiki/SHA-2

Реализация алгоритма SHA-256 https://habr.com/ru/articles/729260/

SHA-256 https://habr.com/ru/companies/selectel/articles/530262/

**ПРИЛОЖЕНИЕ А**

Блок-схема алгоритма