Министерство науки и высшего образования Российской Федерации
Российский химико-технологический университет имени Д. И. Менделеева

Факультет информационных технологий и управления
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №9

«Хеш функции»

Выполнил студент группы КС-30 Колесников Артем Максимович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/AMKolesnikov_30_ALG

Приняли: аспирант кафедры ИКТ Пысин Максим Дмитриевич
аспирант кафедры ИКТ Краснов Дмитрий Олегович

Дата сдачи: 16.05.2022

Москва
2022

Содержание

Описание задачи

В рамках лабораторной работы необходимо было реализовать алгоритм хеширования SHA2.

Для реализованной хеш функции провести следующие тесты:

- Провести сгенерировать 1000 пар строк длинной 128 символов, отличающихся друг от друга 1,2,4,8,16 символов и сравнить хеши для пар между собой, проведя поиск одинаковых последовательностей символов в хешах и подсчитав максимальную длину такой последовательности. Результаты для каждого количества отличий нанести на график, где по оси x кол-во отличий, а по оси y максимальная длинна одинаковой последовательности.
- Провести N = 10^i(i от 2 до 6) генерацию хешей для случайно сгенерированных строк длинно 256 символов, и выполнить поиск одинаковых хешей в итоговом наборе данных, результаты привести в таблице где первая колонка это N генераций, а вторая таблица наличие и кол-во одинаковых хешей, если такие были.
- Провести по 1000 генераций хеша для строк длинной n (64, 128, 256, 512, 1024, 2048, 4096, 8192)(строки генерировать случайно для каждой серии), подсчитать среднее время и построить зависимость скорости расчета хеша от размера входных данных

Описание структуры

SHA-2 (англ. Secure Hash Algorithm Version 2 — безопасный алгоритм хеширования, версия 2) — семейство криптографических алгоритмов — однонаправленных хеш-функций, включающее в себя алгоритмы SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/256 и SHA-512/224.

Хеш-функции предназначены для создания «отпечатков» или «дайджестов» для сообщений произвольной длины. Применяются в различных приложениях или компонентах, связанных с защитой информации.

- при построении ассоциативных массивов;
- при поиске дубликатов в сериях наборов данных;
- при построении уникальных идентификаторов для наборов данных;
- при вычислении контрольных сумм от данных для последующего обнаружения в них ошибок, возникающих при хранении и/или передаче данных;
- при сохранении паролей в системах защиты в виде хеш-кода;
- при выработке электронной подписи;

По умолчанию, базовая реализация хэш-функции должна удовлетворять следующим свойствам:

- Детерминированность – т.е. функция должна, в обязательном порядке, выдавать одинаковый вывод на одинаковы ввод.
- Скорость вычисления – функция должна быстро вычисляться, чтобы ее можно было эффективно использовать для итеративных и постоянно возникающих процессов.
- Минимальное количество коллизий. – все хеширующие функции не гарантируют полное отсутствие колизий на бесконечном наборе входных данных.

Коллизии — это пересечение значений, выдаваемых хеш-функцией как результат своей работы для двух абсолютно разных вводов.

Хеш-функции семейства SHA-2 построены на основе структуры Меркла — Дамгора.

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.

Алгоритм использует следующие битовые операции:

- ‖ — конкатенация,
- + — сложение,
- and — побитовое «И»,
- xor — исключающее «ИЛИ»,
- shr (shift right) — логический сдвиг вправо,
- rotr (rotate right) — циклический сдвиг вправо.

Выполнение задачи

Я реализовал хеш функцию SHA256 на языке C++. Моя программа состоит из класса «SHA256» и функции «hashGenerator» «generatorDiffs», «findMaxLengthSameSequence», «main».

В классе «SHA256» описан весь алгоритм хеширование SHA256.

```
class SHA256 {

public:
	SHA256();
	void update(const uint8_t* data, size_t length);
	void update(const string& data);
	uint8_t* digest();

	static string toString(const uint8_t* digest);

private:
	uint8_t  m_data[64];
	uint32_t m_blocklen;
	uint64_t m_bitlen;
	uint32_t m_state[8]; //A, B, C, D, E, F, G, H

	static constexpr 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
	};

	static uint32_t rotr(uint32_t x, uint32_t n);
	static uint32_t choose(uint32_t e, uint32_t f, uint32_t g);
	static uint32_t majority(uint32_t a, uint32_t b, uint32_t c);
	static uint32_t sig0(uint32_t x);
	static uint32_t sig1(uint32_t x);
	void transform();
	void pad();
	void revert(uint8_t* hash);
};

SHA256::SHA256() : m_blocklen(0), m_bitlen(0) {
	m_state[0] = 0x6a09e667;
	m_state[1] = 0xbb67ae85;
	m_state[2] = 0x3c6ef372;
	m_state[3] = 0xa54ff53a;
	m_state[4] = 0x510e527f;
	m_state[5] = 0x9b05688c;
	m_state[6] = 0x1f83d9ab;
	m_state[7] = 0x5be0cd19;
}
```

```
void SHA256::update(const uint8_t* data, size_t length) {
	for (size_t i = 0; i < length; i++) {
		m_data[m_blocklen++] = data[i];
		if (m_blocklen == 64) {
			transform();

			// End of the block
			m_bitlen += 512;
			m_blocklen = 0;
		}
	}
}

void SHA256::update(const string& data) {
	update(reinterpret_cast<const uint8_t*> (data.c_str()), data.size());
}

uint8_t* SHA256::digest() {
	uint8_t* hash = new uint8_t[32];

	pad();
	revert(hash);

	return hash;
}

//циклический сдвиг значения X вправо на N разрядов
uint32_t SHA256::rotr(uint32_t x, uint32_t n) {
	return (x >> n) | (x << (32 - n));
}

uint32_t SHA256::choose(uint32_t e, uint32_t f, uint32_t g) {
	return (e & f) ^ (~e & g);
}

uint32_t SHA256::majority(uint32_t a, uint32_t b, uint32_t c) {
	return (a & (b | c)) | (b & c);
}

uint32_t SHA256::sig0(uint32_t x) {
	return rotr(x, 7) ^ rotr(x, 18) ^ (x >> 3);
}

uint32_t SHA256::sig1(uint32_t x) {
	return rotr(x, 17) ^ rotr(x, 19) ^ (x >> 10);
}

void SHA256::transform() {
	uint32_t maj, xorA, ch, xorE, sum, newA, newE, m[64];
	uint32_t state[8];

	for (uint8_t i = 0, j = 0; i < 16; i++, j += 4) { // Split data in 32 bit blocks for
the 16 first words
		m[i] = (m_data[j] << 24) | (m_data[j + 1] << 16) | (m_data[j + 2] << 8) |
(m_data[j + 3]);
	}

	for (uint8_t k = 16; k < 64; k++) { // Remaining 48 blocks
		m[k] = sig1(m[k - 2]) + m[k - 7] + sig0(m[k - 15]) + m[k - 16];
	}

	for (uint8_t i = 0; i < 8; i++) {
		state[i] = m_state[i];
	}
```

```
	for (uint8_t i = 0; i < 64; i++) {
		maj = majority(state[0], state[1], state[2]);
		xorA = rotr(state[0], 2) ^ rotr(state[0], 13) ^ rotr(state[0], 22);

		ch = choose(state[4], state[5], state[6]);

		xorE = rotr(state[4], 6) ^ rotr(state[4], 11) ^ rotr(state[4], 25);

		sum = m[i] + K[i] + state[7] + ch + xorE;
		newA = xorA + maj + sum;
		newE = state[3] + sum;

		state[7] = state[6];
		state[6] = state[5];
		state[5] = state[4];
		state[4] = newE;
		state[3] = state[2];
		state[2] = state[1];
		state[1] = state[0];
		state[0] = newA;
	}

	for (uint8_t i = 0; i < 8; i++) {
		m_state[i] += state[i];
	}
}

void SHA256::pad() {

	uint64_t i = m_blocklen;
	uint8_t end = m_blocklen < 56 ? 56 : 64;

	m_data[i++] = 0x80; // Append a bit 1
	while (i < end) {
		m_data[i++] = 0x00; // Pad with zeros
	}

	if (m_blocklen >= 56) {
		transform();
		memset(m_data, 0, 56);
	}

	// Append to the padding the total message's length in bits and transform.
	m_bitlen += m_blocklen * 8;
	m_data[63] = m_bitlen;
	m_data[62] = m_bitlen >> 8;
	m_data[61] = m_bitlen >> 16;
	m_data[60] = m_bitlen >> 24;
	m_data[59] = m_bitlen >> 32;
	m_data[58] = m_bitlen >> 40;
	m_data[57] = m_bitlen >> 48;
	m_data[56] = m_bitlen >> 56;
	transform();
}

void SHA256::revert(uint8_t* hash) {
	// SHA uses big endian byte ordering
	// Revert all bytes
	for (uint8_t i = 0; i < 4; i++) {
		for (uint8_t j = 0; j < 8; j++) {
			hash[i + (j * 4)] = (m_state[j] >> (24 - i * 8)) & 0x000000ff;
		}
	}
}
```

```
string SHA256::toString(const uint8_t* digest) {
	stringstream s;
	s << setfill('0') << hex;

	for (uint8_t i = 0; i < 32; i++) {
		s << setw(2) << (unsigned int)digest[i];
	}

	return s.str();
}
```

Функция «hashGenerator» ~ создает экземпляр класса SHA256 и с помощью вспомогательных функций хеширует передаваемое сообщение:

```
string hashGenerator(string msg) {

	string hash;
	SHA256 sha;
	sha.update(msg);
	uint8_t* digest = sha.digest();
	hash = sha.toString(digest);
	//cout << hash << endl;
	delete[] digest;
	return hash;
}
```

Функция «generatorDiffs» ~ на основе передаваемой строки создает такую строку, которая отличается от исходной на N символов:

```
string generatorDiffs(string str, int num_diffs) {
	for (int i = 0; i < num_diffs; i++)
	{
		while (true)
		{
			int rand_ind = numGenerator(0, str.length() - 1);
			if (str[rand_ind] != 'a') {
				str[rand_ind] = 'a';
				break;
			}
		}
	}
	return str;
}
```

Функция «findMaxLengthSameSequence» ~ ищет одинаковые последовательности символов в передаваемых строках, рассчитывает их длину и возвращает максимальную длину такой последовательности.

```
int findMaxLengthSameSequence(string str1, string str2) {

	int max_length = 0;

	for (int i = 0; i < str1.length(); i++)
	{
		int count = 0;

		for (int j = 0; j < str2.length(); j++)
		{
			if (str1[i] == str2[j])
			{
				int temp_i = i;
				int temp_j = j;
				while ((str1[temp_i] == str2[temp_j]) && (str1.length() != 
temp_i) && (str2.length() != temp_j)) {
```

```
                                temp_i++;
                                temp_j++;
                                count++;
                        }

                        if (max_length < count)
                                max_length = count;

                        count = 0;
                }
        }
    }

    return max_length;
}
```

В функции «main» проводятся серии тестов алгоритма хеширования SHA256. В тесте №1 генерируется 1000 пар строк длинной 128 символов, отличающихся друг от друга 1,2,4,8,16 символов и сравниваются хеши для пар между собой, проводится поиск одинаковых последовательностей символов в хешах и рассчитывается максимальную длину такой последовательности. В тесте №2 проводится N = 10^i(i от 2 до 6) генераций хешей для случайно сгенерированных строк длинно 256 символов, и выполняется поиск одинаковых хешей в итоговом наборе данных. В тесте №3 проводится по 1000 генераций хеша для строк длинной n (64, 128, 256, 512, 1024, 2048, 4096, 8192) и подсчитывается среднее время генерации хеша. Все полученные результаты выводятся в файл «Time.txt».

```
int main()
{
    srand(time(0));
    setlocale(LC_ALL, "Rus");
    cout << "1" << endl;

    //TEST 1

    int NUM_PAIR = 1000;
    int str_length_test1 = 128;

    vector<int> max_length_1_dif(NUM_PAIR);
    vector<int> max_length_2_dif(NUM_PAIR);
    vector<int> max_length_4_dif(NUM_PAIR);
    vector<int> max_length_8_dif(NUM_PAIR);
    vector<int> max_length_16_dif(NUM_PAIR);

    for (int i = 0; i < NUM_PAIR; i++)
    {
        string str, str_1_dif, str_2_dif, str_4_dif, str_8_dif, str_16_dif, hash;

        for (int i = 0; i < str_length_test1; i++)
        {
            int num = numGenerator(0, 9);
            str += to_string(num);
        }

        hash = hashGenerator(str);

        str_1_dif = generatorDiffs(str, 1);
        max_length_1_dif[i] = findMaxLengthSameSequence(hash,
hashGenerator(str_1_dif));

        str_2_dif = generatorDiffs(str, 2);
        max_length_2_dif[i] = findMaxLengthSameSequence(hash,
hashGenerator(str_2_dif));
```

```
            str_4_dif = generatorDiffs(str, 4);
            max_length_4_dif[i] = findMaxLengthSameSequence(hash,
hashGenerator(str_4_dif));

            str_8_dif = generatorDiffs(str, 8);
            max_length_8_dif[i] = findMaxLengthSameSequence(hash,
hashGenerator(str_8_dif));

            str_16_dif = generatorDiffs(str, 16);
            max_length_16_dif[i] = findMaxLengthSameSequence(hash,
hashGenerator(str_16_dif));
        }
        cout << "2" << endl;

        //TEST 2

        int NUM_GENERATIONS_TEST2 = 5;
        int str_length_test2 = 256;
        vector<vector<string>> hash_array(NUM_GENERATIONS_TEST2);
        int power = 2;
        vector<int> countSameHash(NUM_GENERATIONS_TEST2);

        for (int i = 0; i < NUM_GENERATIONS_TEST2; i++)
        {
            string str;
            long N = pow(10, power);

            for (int j = 0; j < N; j++)
            {
                for (int i = 0; i < str_length_test2; i++)
                {
                    int num = numGenerator(0, 9);
                    str += to_string(num);
                }

                hash_array[i].push_back(hashGenerator(str));
            }

            power++;
        }

        for (int i = 0; i < NUM_GENERATIONS_TEST2; i++)
        {
            int count = 0;

            for (int j = 0; j < hash_array[i].size(); j++) {
                for (int k = j + 1; k < hash_array[i].size(); k++) {

                    if (hash_array[i][j] == hash_array[i][k])
                        count++;
                }
            }

            countSameHash[i] = count;
        }

        cout << "3" << endl;

        //TEST 3

        vector<int> N = { 64, 128, 256, 512, 1024, 2048, 4096, 8192 };
        int NUM_GENERATIONS_TEST3 = 1000;
        vector<vector<double>> time_generate_hash(N.size());
        vector<double> sum_time_generate_hash(N.size());
```

```
	for (int i = 0; i < N.size(); i++)
	{
		for (int j = 0; j < NUM_GENERATIONS_TEST3; j++)
		{
			string str;
			for (int k = 0; k < N[i]; k++)
			{
				int num = numGenerator(0, 9);
				str += to_string(num);
			}

			chrono::high_resolution_clock::time_point start =
chrono::high_resolution_clock::now();
			hashGenerator(str);
			chrono::high_resolution_clock::time_point end =
chrono::high_resolution_clock::now();
			chrono::duration<double, milli> milli_diff_gen_hash = end - start;

			time_generate_hash[i].push_back(milli_diff_gen_hash.count());
			sum_time_generate_hash[i] += milli_diff_gen_hash.count();
		}
	}

	ofstream tout("Time.txt");

	if (tout.is_open()) {

		tout << "Тест 1" << endl;
		tout << setw(13) << "Число отличий" << setw(50) << "Макс длина одинаковой
последовательности" << endl;
		tout << setw(13) << "1" << setw(50) << *max_element(max_length_1_dif.begin(),
max_length_1_dif.end()) << endl;
		tout << setw(13) << "2" << setw(50) << *max_element(max_length_2_dif.begin(),
max_length_2_dif.end()) << endl;
		tout << setw(13) << "4" << setw(50) << *max_element(max_length_4_dif.begin(),
max_length_4_dif.end()) << endl;
		tout << setw(13) << "8" << setw(50) << *max_element(max_length_8_dif.begin(),
max_length_8_dif.end()) << endl;
		tout << setw(13) << "16" << setw(50) <<
*max_element(max_length_16_dif.begin(), max_length_16_dif.end()) << endl;

		tout << "\n\nТест 2" << endl;
		tout << setw(15) << "Число генераций" << setw(30) << "Число одинаковых хешей"
<< endl;
		power = 2;
		for (int i = 0; i < NUM_GENERATIONS_TEST2; i++)
		{
			int N = pow(10, power);
			tout << setw(15) << N << setw(30) << countSameHash[i] << endl;
			power++;
		}

		tout << "\n\nТест 3" << endl;
		tout << setw(12) << "Длина строки" << setw(40) << "Среднее время расчета
хеша(мс)" << endl;
		for (int i = 0; i < N.size(); i++)
		{
			tout << setw(12) << N[i] << setw(40) << sum_time_generate_hash[i] /
NUM_GENERATIONS_TEST3 << endl;
		}

		tout.close();
	}
}
```

Результат работы программы:

```
Тест 1
Число отличий          Макс длина одинаковой последовательности
            1                                                 5
            2                                                 5
            4                                                 5
            8                                                 5
           16                                                 5


Тест 2
Число генераций        Число одинаковых хешей
          100                                0
         1000                                0
        10000                                0
       100000                                0


Тест 3
Длина строки          Среднее время расчета хеша(мс)
          64                                0.271596
         128                                 0.36976
         256                                0.432672
         512                                0.615584
        1024                                0.985446
        2048                                 1.69176
        4096                                 3.16085
        8192                                 6.08776
```

Рисунок 1 - Результат из файла "Time.txt"

Рисунок 2 – График зависимости максимальной длины одинаковой последовательности символов в паре хешей, сгенерированных из строк отличающихся на N символов.

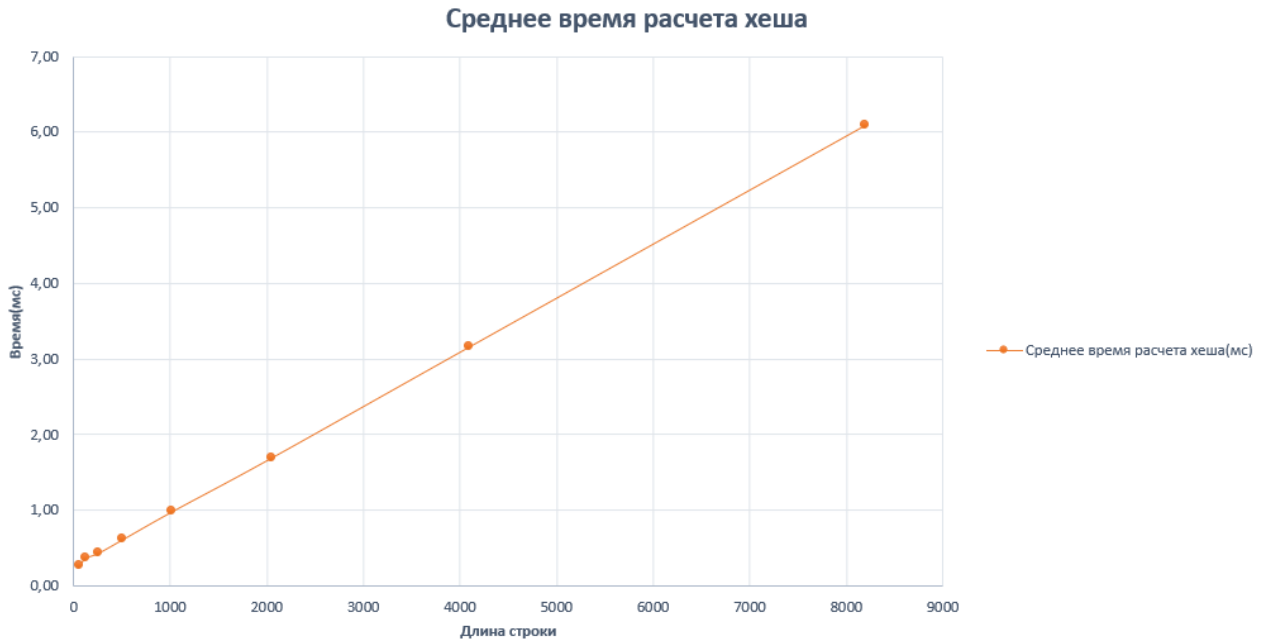

Рисунок 3 – График зависимости среднего времени расчета хеша за 1000 генераций от длины хешируемоей строки.

Заключение

В этой лабораторной работе я познакомился с алгоритмами хеширования и самостоятельно реализовал алгоритм хеширования SHA256, а после ~ провел для него несколько тестов. Анализируя график «Максимальной длины одинаковой последовательности», не трудно заметить, что **вероятность встретить одинаковую последовательность в паре хешей**, генерируемых из строк, которые различаются только на некоторое количество символов ~ **не зависит от числа различий**. Даже если исходная строка будет отличаться всего на 1 символ ~ будет сгенерирован совершенно другой хеш, не похожий на исходный. Это связано с так называемым «лавинным эффектом». По результатам теста №2 наглядно понятно, что встреча колизий достаточно мала, но это не гарантируют полное отсутствие колизий на бесконечном наборе входных данных. Из теста №3 мы можем заметить, что среднее время генерации хеша линейно зависит от длины хешируемой строки. Чем больше строка ~ тем больше время генерации хеша.