Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9

Выполнил студент группы КС-30 Суханова Евгения Валерьевна
Ссылка на репозиторий:
https://github.com/MUCTR-IKT-CPP/EVSuhanova_30/blob/main/Algoritms/laba9.cpp

Приняли: Пысин Максим Дмитриевич
Краснов Дмитрий Олегович

Дата сдачи: 24.04.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать алгоритм RIPEMD-160.

Для реализованной хеш функции провести следующие тесты:

1. Провести генерацию 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 длина одинаковой последовательности.
2. Провести N = 10^i(i от 2 до 6) генерацию хешей для случайно сгенерированных строк длиной 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.
3. 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 метода/модели.

RIPEMD-160 (от англ. RACE Integrity Primitives Evaluation Message Digest) — криптографическая хеш-функция, разработанная в Католическом университете Лувена Хансом Доббертином (Hans Dobbertin), Антоном Босселарсом (Antoon Bosselaers) и Бартом Пренелом (Бартом Пренелем). Для произвольного входного сообщения функция генерирует 160-разрядное хеш-значение, называемое дайджестом сообщения. RIPEMD-160 является улучшенной версией RIPEMD, которая, в свою очередь, использовала принципы MD4 и по производительности сравнима с более популярной SHA-1.

Также существуют 128-, 256- и 320-битные версии этого алгоритма, которые, соответственно, называются RIPEMD-128, RIPEMD-256 и RIPEMD-320. 128-битная версия представляет собой лишь замену оригинальной RIPEMD, которая также была 128-битной и в которой были найдены уязвимости. 256- и 320-битные версии отличаются удвоенной длиной дайджеста, что уменьшает вероятность коллизий, но при этом функции не являются более криптостойкими.

RIPEMD-160 была разработана в открытом академическом сообществе, в отличие от SHA-1 и SHA-2, которые были созданы NSA. С другой стороны, RIPEMD-160 на практике используется несколько реже, чем SHA-1.

Использование RIPEMD-160 не ограничено какими-либо патентами.

Алгоритм хэширования:

1. Добавление недостающих битов
2. Добавление длины сообщения
3. Определение действующих функций и констант
4. Выполнение алгоритма хеширования

Выполнение задачи.

Генерируем строку из N символов:

```
string generationString(size_t len) {
    string str = "";
    for (int i = 0; i < len; i++) {
        str += (char)(97 + rand() % 26);
    }
    return str;
}
```

Меняем в исходной строке M символов:

```
string changeString(string str, int count) {
    int* arr = new int[count];
    for (int i = 0; i < count; i++) {
        arr[i] = rand() % str.length();
        if (find(_First: arr, _Last: arr + i, _Val: arr[i]) != arr + i) {
            i--;
        }
    }
    for (int i = 0; i < count; i++) {
        str[arr[i]] = (char)(((toascii(str[arr[i]]) - 'a' + rand() % 25) % 26) + 'a');
    }
    delete[] arr;
    return str;
}
```

Хэш-функция:

I. Нелинейные побитовые функции:

- $f(j;x;y;z) = x \otimes y \otimes z \quad (0 \le j \le 15)$
- $f(j;x;y;z) = (x \wedge y) \vee (\neg x \wedge z) \quad (16 \le j \le 31)$
- $f(j;x;y;z) = (x \vee \neg y) \otimes z \quad (32 \le j \le 47)$
- $f(j;x;y;z) = (x \wedge z) \vee (y \wedge \neg z) \quad (48 \le j \le 63)$
- $f(j;x;y;z) = x \otimes (y \vee \neg z) \quad (64 \le j \le 79)$

```
#define F(x, y, z) ((x) ^ (y) ^ (z))
#define G(x, y, z) (((x) & (y)) | (~(x) & (z)))
#define H(x, y, z) (((x) | ~(y)) ^ (z))
#define IQ(x, y, z) (((x) & (z)) | ((y) & ~(z)))
#define J(x, y, z) ((x) ^ ((y) | ~(z)))
```

VI. Набор для битового поворота влево (операция rol)

- s(0..15) = 11; 14; 15; 12; 5; 8; 7; 9; 11; 13; 14; 15; 6; 7; 9; 8
- s(16..31) = 7; 6; 8; 13; 11; 9; 7; 15; 7; 12; 15; 9; 11; 7; 13; 12
- s(32..47) = 11; 13; 6; 7; 14; 9; 13; 15; 14; 8; 13; 6; 5; 12; 7; 5
- s(48..63) = 11; 12; 14; 15; 14; 15; 9; 8; 9; 14; 5; 6; 8; 6; 5; 12
- s(64..79) = 9; 15; 5; 11; 6; 8; 13; 12; 5; 12; 13; 14; 11; 8; 5; 6
- s'(0..15) = 8; 9; 9; 11; 13; 15; 15; 5; 7; 7; 8; 11; 14; 14; 12; 6
- s'(16..31) = 9; 13; 15; 7; 12; 8; 9; 11; 7; 7; 12; 7; 6; 15; 13; 11
- s'(32..47) = 9; 7; 15; 11; 8; 6; 6; 14; 12; 13; 5; 14; 13; 13; 7; 5
- s'(48..63) = 15; 5; 8; 11; 14; 14; 6; 14; 6; 9; 12; 9; 12; 5; 15; 8
- s'(64..79) = 8; 5; 12; 9; 12; 5; 14; 6; 8; 13; 6; 5; 15; 13; 11; 11

```
/* round 1 */
FF(aa, bb, cc, dd, ee, X[0], 11);
FF(ee, aa, bb, cc, dd, X[1], 14);
FF(dd, ee, aa, bb, cc, X[2], 15);
FF(cc, dd, ee, aa, bb, X[3], 12);
FF(bb, cc, dd, ee, aa, X[4], 5);
FF(aa, bb, cc, dd, ee, X[5], 8);
FF(ee, aa, bb, cc, dd, X[6], 7);
FF(dd, ee, aa, bb, cc, X[7], 9);
FF(cc, dd, ee, aa, bb, X[8], 11);
FF(bb, cc, dd, ee, aa, X[9], 13);
FF(aa, bb, cc, dd, ee, X[10], 14);
FF(ee, aa, bb, cc, dd, X[11], 15);
FF(dd, ee, aa, bb, cc, X[12], 6);
FF(cc, dd, ee, aa, bb, X[13], 7);
FF(bb, cc, dd, ee, aa, X[14], 9);
FF(aa, bb, cc, dd, ee, X[15], 8);
```

V. Исходные значения слов дайджеста

- h0 = 67452301x;
- h1 = EFCDAB89x;
- h2 = 98BADCFEx;
- h3 = 10325476x;
- h4 = C3D2E1F0x;

```
int j;
uint32_t digest[5] = { 0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xc3d2e1f0UL };
```

Комбинирование результата:

```
/* combine results */
ddd += cc + MDbuf[1];
MDbuf[1] = MDbuf[2] + dd + eee;
MDbuf[2] = MDbuf[3] + ee + aaa;
MDbuf[3] = MDbuf[4] + aa + bbb;
MDbuf[4] = MDbuf[0] + bb + ccc;
MDbuf[0] = ddd;
```

Функция ROL:

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

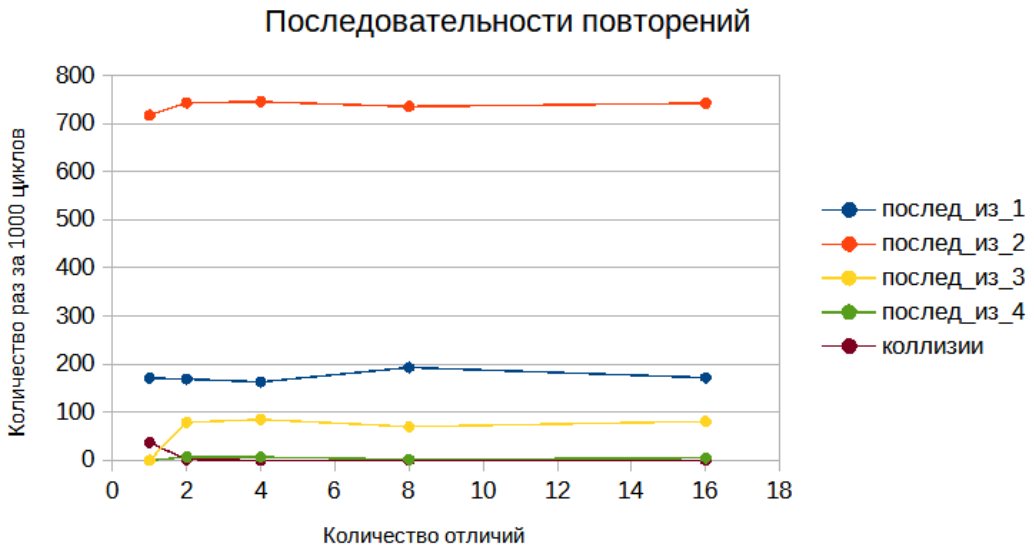
Представление шестнадцатеричного числа в виде строки:

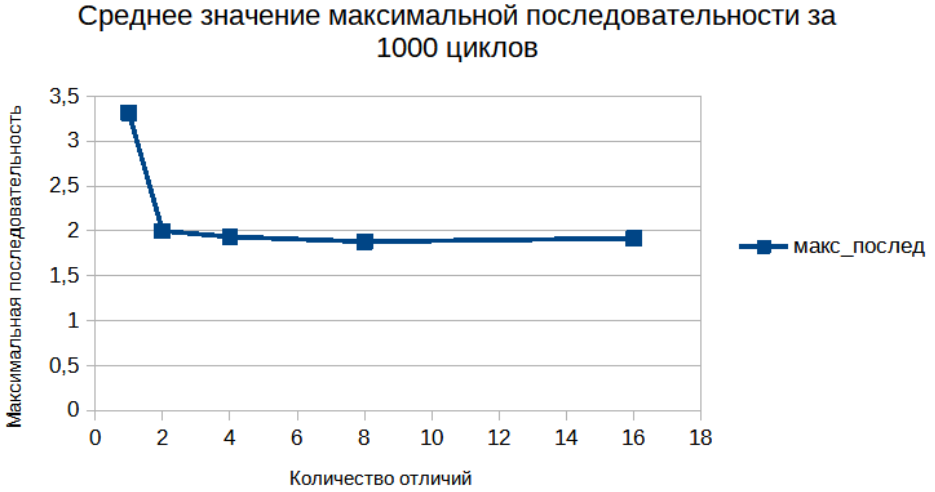
```
string uint8_to_hex_string(const uint8_t* v, const size_t s) {
    stringstream ss;

    ss << hex << setfill(_Ch: '0');

    for (int i = 0; i < s; i++) {
        ss << hex << setw(2) << static_cast<int>(v[i]);
    }

    return ss.str();
}
```

Полученные результаты

Поиск максимальной последовательности одинаковых символов в хэшах:

N	1	2	4	8	16
1	2	2	2	2	1
2	1	2	2	2	2
3	1	2	3	2	2
4	2	2	2	2	2
5	2	1	2	2	4
6	40	1	2	1	2
7	2	2	3	2	2
8	1	2	2	3	1
9	2	3	1	3	2
10	2	1	2	2	1
11	2	2	2	2	2
12	2	2	2	1	2
13	1	1	2	2	3
14	2	1	2	2	1
15	2	2	2	2	2
16	2	2	2	2	2
17	2	1	2	2	3
18	40	2	2	2	2
19	2	2	2	2	3
20	2	1	2	1	1



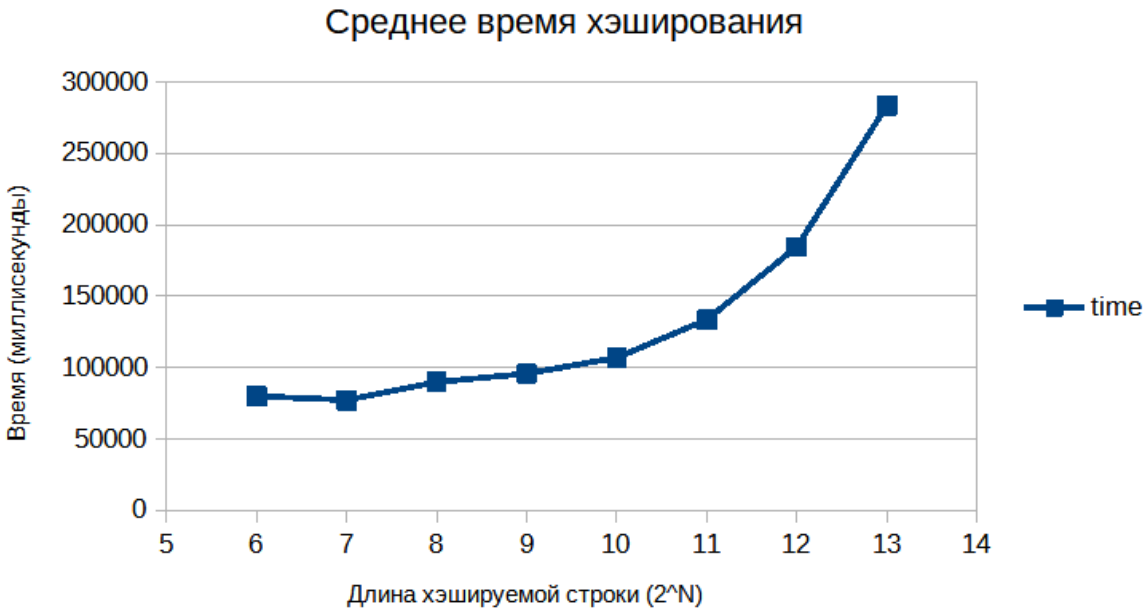


Количество коллизий:

N	count
100	0
1000	0
10000	0
100000	0
1000000	0

Среднее время хэширования:

N	time
6	80070
7	77292
8	89897,6
9	95845,3
10	106946
11	133575
12	184558
13	283318



Заключение.

Из полученных результатов можно увидеть, что данная хэш-функция выполняет довольно быстро, что является большим плюсом, так как нам необходимо хэшировать как можно быстрее. Помимо этого, можно заметить, что коллизии встречаются довольно редко, что также является преимуществом. Это связано с тем, что для хэшированию любой строки, алгоритм создает индивидуальный 160-разрядное хэш-значение, которое называется дайджестом сообщения. С помощью него и хэшируется сообщение.

Однако коллизии все же встречаются. Особенно это заметно, но достаточно коротких сообщениях, с минимальным количеством отличий. Но все же при более длинных сообщениях или при большем количестве отличий у строки коллизии вообще не встречаются.

Если говорить про время работы хэш-функции несложно заметить, что при увеличении длины строки увеличивается и время работы. Однако стоит отметить, что среднее время хэширования самого длинного из представленных сообщения занимает не так уж и много времени.