

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего
образования «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА № 51

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

ассистент

должность, уч. степень, звание

подпись, дата

М.Н. Исаева

инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №4

Однонаправленные хеш-функции

по курсу: Криптографические методы защиты информации

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №

5911

подпись, дата

К.В. Жук

инициалы, фамилия

Санкт-Петербург 2022

1. Цель работы:

Целью нашей работы является ознакомление с алгоритмами однонаправленных хеш-функций. В данном случае рассматривается семейство алгоритмов SHA-2 (реализован алгоритм SHA-256). Так же лабораторная работа подразумевает проведение эксперимента на нахождение второго прообраза и коллизий на малом количестве бит (8, 10, 12, 14, 16).

2. Описание алгоритма:

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.

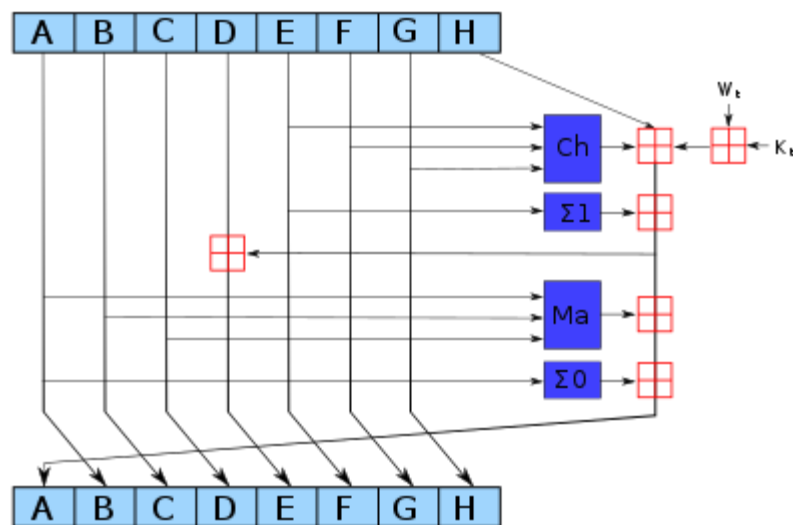


Рис.1 – Схема одной итерации алгоритмов семейства SHA-2

3. Реализация алгоритма:

3.1. Сначала инициализируются определённые константы.

- 1) Первые 32 бита дробных частей квадратных корней первых восьми простых чисел [от 2 до 19]:

```
ctx->state[0] = 0x6a09e667;  
ctx->state[1] = 0xbb67ae85;  
ctx->state[2] = 0x3c6ef372;  
ctx->state[3] = 0xa54ff53a;  
ctx->state[4] = 0x510e527f;  
ctx->state[5] = 0x9b05688c;  
ctx->state[6] = 0x1f83d9ab;  
ctx->state[7] = 0x5be0cd19;
```

Рис.2 – Первые восемь простых чисел

- 2) Первые 32 бита дробных частей кубических корней первых 64 простых чисел [от 2 до 311]:

```
unsigned int k[64] = {  
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,  
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,  
    0xe49b69c1, 0xefbe4786, 0xfc19dc6, 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  
};
```

Рис.3 – Дробные части кубических корней простых чисел

3.2.

К сообщению, хэш которого нам надо получить, добавляется 1, потом K нулевых бит так, чтобы $(L + 1 + K) \bmod 512 = 448$, где L — число бит в сообщении. Последние 8 байт отводятся на 64-битное число, которое означает длину исходного сообщения.

3.3. Далее сообщение обрабатывается кусками по 512 бит определённым образом:

```
for (i = 0, j = 0; i < 16; ++i, j += 4)
    m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8) | (data[j + 3]);
for (; i < 64; ++i)
    m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];

a = ctx->state[0];
b = ctx->state[1];
c = ctx->state[2];
d = ctx->state[3];
e = ctx->state[4];
f = ctx->state[5];
g = ctx->state[6];
h = ctx->state[7];

for (i = 0; i < 64; ++i) {
    t1 = h + EP1(e) + CH(e, f, g) + k[i] + m[i];
    t2 = EP0(a) + MAJ(a, b, c);
    h = g;
    g = f;
    f = e;
    e = d + t1;
    d = c;
    c = b;
    b = a;
    a = t1 + t2;
}
```

Рис.4 – обработка кусков сообщения

3.4. После обработки, вычисленные значение добавляются к ранее вычисленным результатам:

```
ctx->state[0] += a;
ctx->state[1] += b;
ctx->state[2] += c;
ctx->state[3] += d;
ctx->state[4] += e;
ctx->state[5] += f;
ctx->state[6] += g;
ctx->state[7] += h;
```

Рис.5 – суммирование с прошлыми результатами

- 3.5. После всех преобразований, результаты работы программы, которые находятся в переменных от h0 до h7 конкатенируются. Это итоговое значение и является хэшем изначального сообщения.

4. Примеры работы программы:

Пример 1:

SHA256

Текст (7):

sha-256

☒ SHA256 ☐ SHA224

Кодировать

Результат (64):

3128f8ac2988e171a53782b144b98a5c2ee723489c8b220cece002916fbc71e2

Рис.6 – Хэш сообщения из проверенного источника

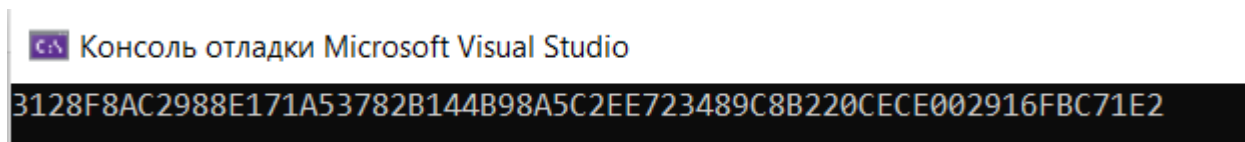


Рис.7 – Хэш сообщения из реализованной программы

Пример 2:

SHA256

Текст (7):

0000000

☒ SHA256 ☐ SHA224


Кодировать

Результат (64):

20fdf64da3cd2c78ec3c033d2ac628bacf701711fa99435ee37bef0304800dc5

Копировать Очистить

Рис.8 – Хэш сообщения из проверенного источника

 Консоль отладки Microsoft Visual Studio

20FDF64DA3CD2C78EC3C033D2AC628BACF701711FA99435EE37BEF0304800DC5

Рис.9 – Хэш сообщения из реализованной программы

Как видно из результатов, хэши совпадают, следовательно, программа реализована верно.

5. Результаты экспериментов:

Количество N для нахождения второго прообраза (1000 экспериментов):

Таблица 1 – количество попыток нахождения прообраза

Слово - пароль	8 бит	10 бит	12 бит	14 бит	16 бит
sha-256	28	984	4062	16516	16777
inskip1	425	1055	4488	16777	16777
0000000	28	1026	4254	16193	16777

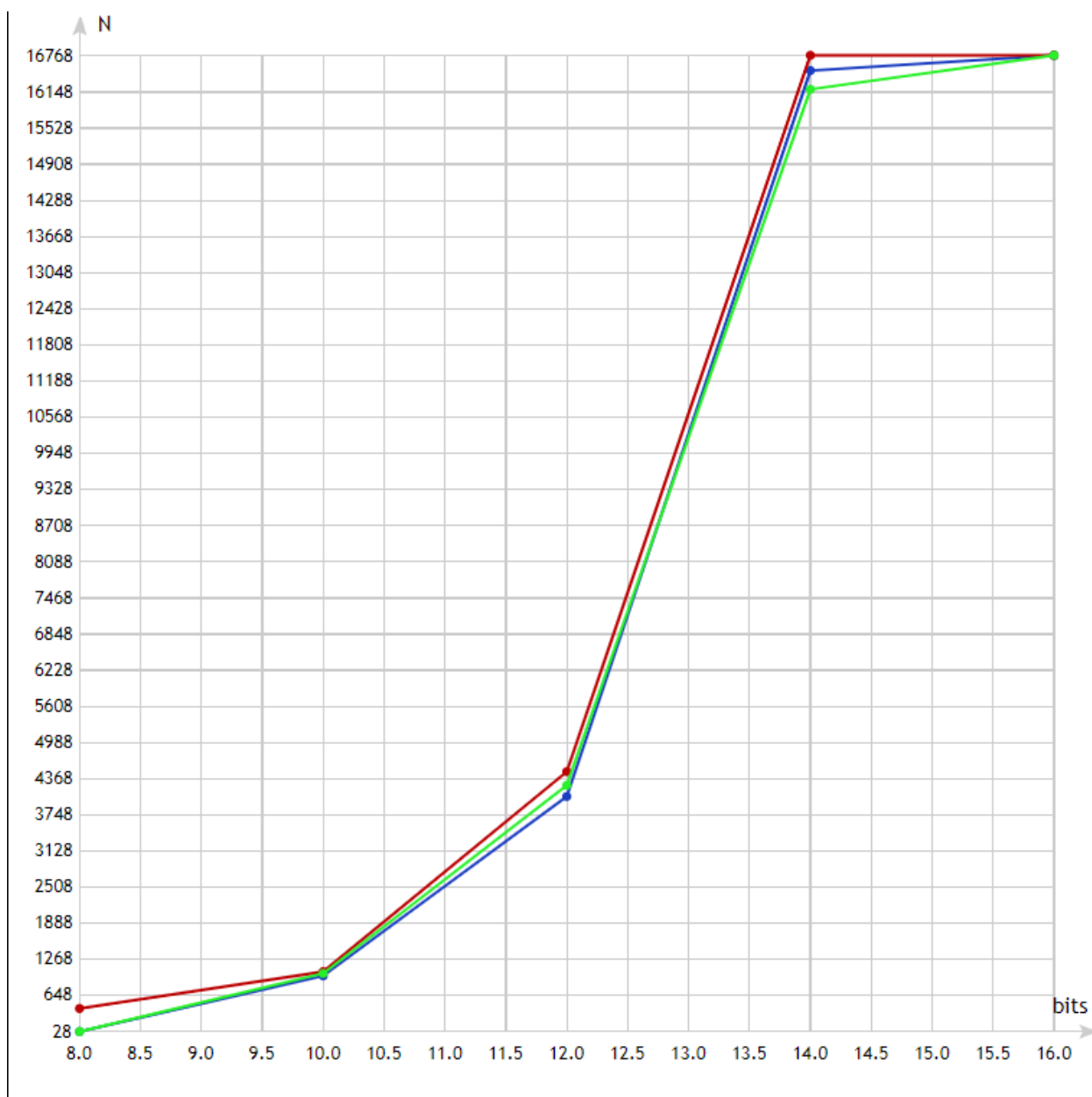


Рис.10 – График нахождения прообразов для 3 паролей

Синий – первый пароль
Красный – второй пароль
Зелёный – третий пароль

Таблица 2 – количество попыток нахождения коллизии

Слово - пароль	8 бит	10 бит	12 бит	14 бит	16 бит
sha-256	22	32	66	34	322
inskip1	23	34	61	34	333
0000000	22	34	67	35	325

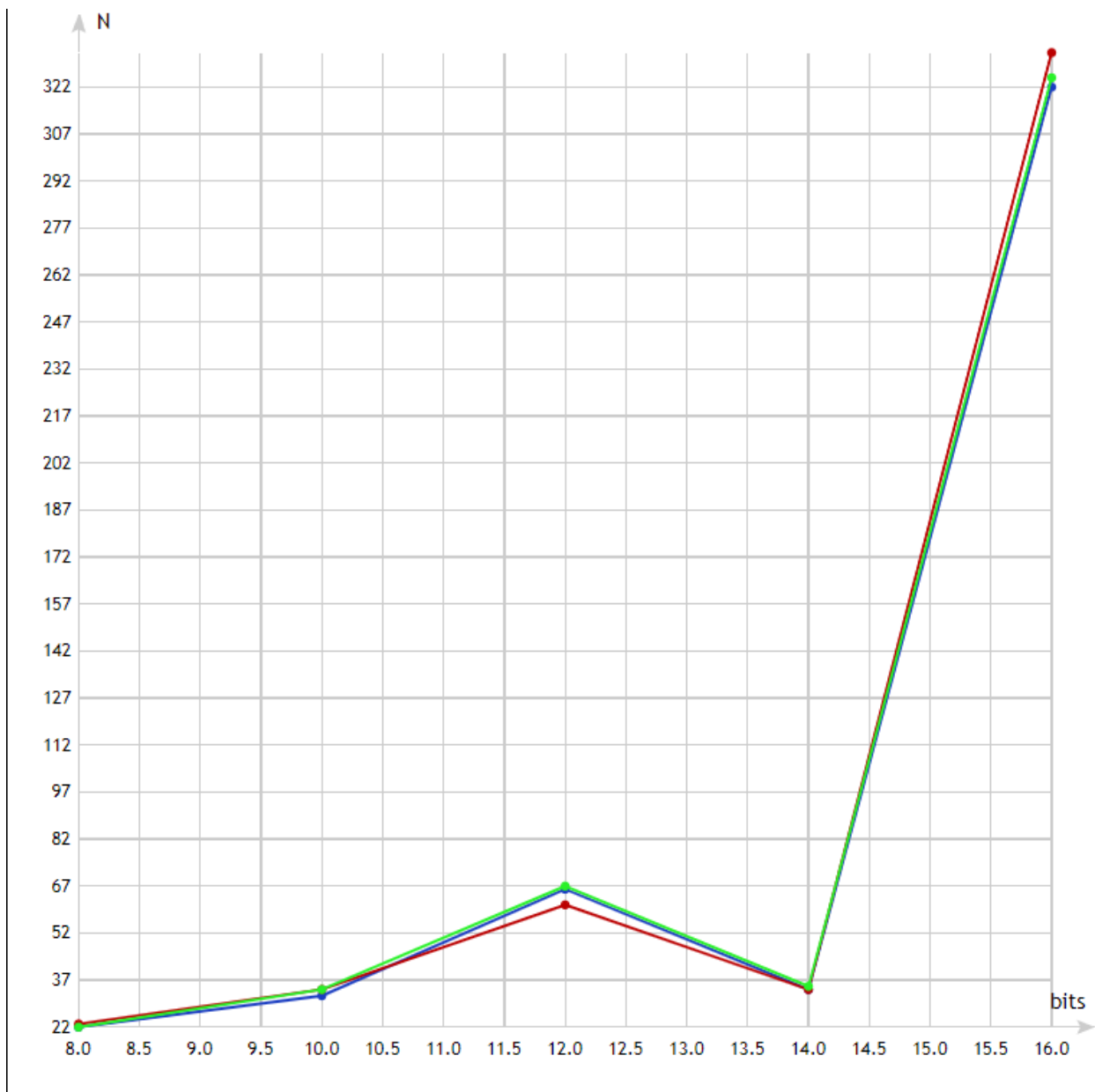


Рис.11 - График нахождения коллизий для 3 паролей

*Синий – первый пароль
Красный – второй пароль
Зелёный – третий пароль*

Исходя из результатов экспериментов, можно сделать вывод, что чем больше бит в хэше, тем труднее найти прообразы и коллизии.

6. Вывод:

Алгоритмы SHA-2 актуальны и сейчас, они используются в сертификате SSL при установлении защищённого соединения на сайтах. Также семейство активно используется в майнинге криптовалют.

Особых уязвимостей семейства обнаружено не было (кроме усеченных версий). Тем не менее в 2012 году NIST принял в качестве стандарта SHA-3.

7. Литература:

- 1) А.Л. Чмора "Современная прикладная криптография"
- 2) Б.Я. Рябко "Основы современной криптографии для специалистов в ИТ"
- 3) Баричев, Серов "Основы современной криптографии"