Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА № 51

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ассистент. |  |  |  | М. Н. Исаева |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №4 |
| ОДНОНАПРАВЛЕННЫЕ ХЕШ-ФУНКЦИИ |
| по курсу: КРИПТОГРАФИЧЕСКИЕ МЕТОДЫ ЗАЩИТЫ ИНФОРМАЦИИ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 5811 |  |  |  | Т.С.Климов |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2021

**1. Цель работы:**

Исследование однонаправленных хеш-функций.

**2. Описание задания:**

Реализовать алгоритм хеширования MD5 (вариант 2). Сравнить хеш, полученный при выполнении реализованного алгоритма и хеш из стандарта (или примеры в интернете).

Придумать слово-пароль, которое будет исходным сообщением х, найти от него хеш h(x) взять от полученного хеша первые 8 бит, обозначив последовательность у0. Далее случайным образом сгенерировать N (N зависит от двух условий, описанных ниже) сообщений, найти от каждого хеш и взять от каждого хеша первые 8 бит, получив последовательность у1, у2, ... , уN-1.

1) Нахождение второго прообраза. Необходимо найти такой уi, что уi = у0. Посчитать количество шагов, которое потребовалось, чтобы найти уi – это сложность второго прообраза.

2) Нахождение коллизий. Необходимо найти в полученной последовательности такие уi и уj, что уi = уj. Посчитать количество шагов, которое потребовалось, чтобы найти эту пару – это сложность коллизии.

Проделать оба эксперимента 1000 раз, получить средние значения сложности второго прообраза и сложности коллизии.

Повторить данные манипуляции для последовательности хешей размером 12, 16, 20 и 24 бит. Построить графики зависимости среднего значения сложности второго прообраза и коллизии от количества взятых бит. Оценить полученные графики.

**3. Описание алгоритма:**

На вход алгоритма поступает входной поток данных, хеш которого необходимо найти. Длина сообщения измеряется в битах и может быть любой (в том числе нулевой). Запишем длину сообщения в *L*. Это число целое и неотрицательное. Кратность каким-либо числам необязательна. После поступления данных идёт процесс подготовки потока к вычислениям.

Ниже приведены 5 шагов алгоритма:

**Шаг 1. Выравнивание потока:**

Сначала к концу потока дописывают единичный бит.

Затем добавляют некоторое число нулевых бит такое, чтобы новая длина потока *{\displaystyle L'}L’* стала сравнима с 448 по модулю 512, (*{\displaystyle L'=512\times N+448}L’ = 512 × N + 448*). Выравнивание происходит в любом случае, даже если длина исходного потока уже сравнима с 448.

**Шаг 2. Добавление длины сообщения:**

В конец сообщения дописывают 64-битное представление длины данных (количество бит в сообщении) до выравнивания. Сначала записывают младшие 4 байта, затем старшие. Если длина превосходит - 1{\displaystyle 2^{64}-1}, то дописывают только младшие биты (эквивалентно взятию по модулю {\displaystyle 2^{64}}). После этого длина потока станет кратной 512. Вычисления будут основываться на представлении этого потока данных в виде массива слов по 512 бит.

**Шаг 3. Инициализация буфера:**

Для вычислений инициализируются четыре переменные размером по 32 бита, начальные значения которых задаются шестнадцатеричными числами (порядок байтов little-endian):

А = 01 23 45 67; // 67452301h

В = 89 AB CD EF; // EFCDAB89h

С = FE DC BA 98; // 98BADCFEh

D = 76 54 32 10. // 10325476h

В этих переменных будут храниться результаты промежуточных вычислений. Начальное состояние ABCD называется инициализирующим вектором.

**Шаг 4. Вычисление в цикле:**

Определим функции и константы, которые понадобятся нам для вычислений.

1) Для каждого раунда потребуется своя функция от трёх параметров — слов, результатом также будет слово:

1-й раунд: FunF(X, Y, Z) = (X ∧ Y) ∨ ( ¬X ∧ Z){\displaystyle \operatorname {FunF} (X,Y,Z)=(X\wedge Y)\vee (\neg X\wedge Z)},

2-й раунд: {\displaystyle \operatorname {FunG} (X,Y,Z)=(X\wedge Z)\vee (\neg Z\wedge Y)}FunG(X, Y, Z) = (X ∧ Z) ∨ (¬Z ∧ Y){\displaystyle \operatorname {FunF} (X,Y,Z)=(X\wedge Y)\vee (\neg X\wedge Z)},

3-й раунд: {\displaystyle \operatorname {FunH} (X,Y,Z)=X\oplus Y\oplus Z}FunH(X, Y, Z) = X ⊕ Y ⊕ Z,

4-й раунд:{\displaystyle \operatorname {FunI} (X,Y,Z)=Y\oplus (\neg {Z}\vee X)} FunI(X, Y, Z) = Y ⊕ (¬Z ∨ X){\displaystyle \operatorname {FunF} (X,Y,Z)=(X\wedge Y)\vee (\neg X\wedge Z)},

2) Определим таблицу констант T{\displaystyle T[1\ldots 64]}[1…64] — 64-элементная таблица данных, построенная следующим образом:  .{\displaystyle T[n]=\operatorname {int} (2^{32}\cdot |\sin n|)}

3) Каждый 512-битный блок проходит 4 этапа вычислений по 16 раундов. Для этого блок представляется в виде массива *X* из 16 слов по 32 бита. Все раунды однотипны и имеют вид: [abcd k s i], определяемый как

{\displaystyle a=b+((a+\operatorname {Fun} (b,c,d)+X[k]+T[i])\lll s)},

где ***k*** — номер 32-битного слова из текущего 512-битного блока сообщения, и **{\displaystyle \ldots \lll s}…<<< *s*** — циклический сдвиг влево на ***s*** бит полученного 32-битного аргумента. Число ***s*** задается отдельно для каждого раунда.

Заносим в блок данных элемент *n* из массива 512-битных блоков. Сохраняются значения A, B, C и D, оставшиеся после операций над предыдущими блоками (или их начальные значения, если блок первый).

AA = A

BB = B

CC = C

DD = D

**Этап 1**

/\* [abcd k s i] a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). \*/

[ABCD 0 7 1][DABC 1 12 2][CDAB 2 17 3][BCDA 3 22 4]

[ABCD 4 7 5][DABC 5 12 6][CDAB 6 17 7][BCDA 7 22 8]

[ABCD 8 7 9][DABC 9 12 10][CDAB 10 17 11][BCDA 11 22 12]

[ABCD 12 7 13][DABC 13 12 14][CDAB 14 17 15][BCDA 15 22 16]

**Этап 2**

/\* [abcd k s i] a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). \*/

[ABCD 1 5 17][DABC 6 9 18][CDAB 11 14 19][BCDA 0 20 20]

[ABCD 5 5 21][DABC 10 9 22][CDAB 15 14 23][BCDA 4 20 24]

[ABCD 9 5 25][DABC 14 9 26][CDAB 3 14 27][BCDA 8 20 28]

[ABCD 13 5 29][DABC 2 9 30][CDAB 7 14 31][BCDA 12 20 32]

**Этап 3**

/\* [abcd k s i] a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). \*/

[ABCD 5 4 33][DABC 8 11 34][CDAB 11 16 35][BCDA 14 23 36]

[ABCD 1 4 37][DABC 4 11 38][CDAB 7 16 39][BCDA 10 23 40]

[ABCD 13 4 41][DABC 0 11 42][CDAB 3 16 43][BCDA 6 23 44]

[ABCD 9 4 45][DABC 12 11 46][CDAB 15 16 47][BCDA 2 23 48]

**Этап 4**

/\* [abcd k s i] a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). \*/

[ABCD 0 6 49][DABC 7 10 50][CDAB 14 15 51][BCDA 5 21 52]

[ABCD 12 6 53][DABC 3 10 54][CDAB 10 15 55][BCDA 1 21 56]

[ABCD 8 6 57][DABC 15 10 58][CDAB 6 15 59][BCDA 13 21 60]

[ABCD 4 6 61][DABC 11 10 62][CDAB 2 15 63][BCDA 9 21 64]

Суммируем с результатом предыдущего цикла:

A = AA + A

B = BB + B

C = CC + C

D = DD + D

После окончания цикла необходимо проверить, есть ли ещё блоки для вычислений. Если да, то переходим к следующему элементу массива (*n* + 1) и повторяем цикл.

**Шаг 5. Результат вычислений:**

Результат вычислений находится в буфере ABCD, это и есть хеш. Если выводить побайтово, начиная с младшего байта A и заканчивая старшим байтом D, то мы получим MD5-хеш. 1, 0, 15, 34, 17, 18…

**Псевдокод алгоритма:**

*// Все переменные — 32-битные беззнаковые целые. Все сложения выполняются по модулю 2^32.*

**var** int s[64], K[64]

**var** int i

*// s обозначает величины сдвигов для каждой операции:*

s[ 0..15] := *{ 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22 }*

s[16..31] := *{ 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20 }*

s[32..47] := *{ 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23 }*

s[48..63] := *{ 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21 }*

*// Определяем таблицу констант следующим образом*

**for** i from 0 **to** 63 **do**

K[i] := floor(2^32 × abs (sin(i + 1)))

**end** **for**

*// (Или просто используем заранее подсчитанные значения):*

K[ 0.. 3] := *{ 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }*

K[ 4.. 7] := *{ 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }*

K[ 8..11] := *{ 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }*

K[12..15] := *{ 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }*

K[16..19] := *{ 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }*

K[20..23] := *{ 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }*

K[24..27] := *{ 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }*

K[28..31] := *{ 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }*

K[32..35] := *{ 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }*

K[36..39] := *{ 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }*

K[40..43] := *{ 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }*

K[44..47] := *{ 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }*

K[48..51] := *{ 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }*

K[52..55] := *{ 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }*

K[56..59] := *{ 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }*

K[60..63] := *{ 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }*

*// Инициализация переменных:*

**var** int a0 := 0x67452301 *// A*

**var** int b0 := 0xefcdab89 *// B*

**var** int c0 := 0x98badcfe *// C*

**var** int d0 := 0x10325476 *// D*

*// Подготовка: добавляем бит "1" в конец сообщения.*

append "1" bit **to** message

*// Заметка: входные байты представлены строкой из бит,*

*// причем первый бит — старший (big-endian).*

*// Подготовка: дописываем нулевые биты, пока длина сообщения не станет сравнима с 448 по модулю 512*

append "0" bit **until** message length **in** bits ≡ 448 (**mod** 512)

*// Дописываем остаток от деления изначальной длины сообщения на 2^64*

append original length **in** bits **mod** 2^64 **to** message

*// Разбиваем подготовленное сообщение на 512-битные "куски":*

**for** each 512-bit chunk **of** padded message **do**

*// и работаем с каждым по отдельности*

**break** chunk into sixteen 32-bit words M[j], 0 ≤ j ≤ 15 *// разбиваем "кусок" на 16 блоков по 32 бита*

*// Инициализируем переменные для текущего куска:*

**var** int A := a0

**var** int B := b0

**var** int C := c0

**var** int D := d0

*// Основные операции:*

**for** i from 0 **to** 63 **do**

**var** int F, g

**if** 0 ≤ i ≤ 15 **then**

F := (B **and** C) **or** ((**not** B) **and** D)

g := i

**else** **if** 16 ≤ i ≤ 31 **then**

F := (D **and** B) **or** ((**not** D) **and** C)

g := (5×i + 1) **mod** 16

**else** **if** 32 ≤ i ≤ 47 **then**

F := B **xor** C **xor** D

g := (3×i + 5) **mod** 16

**else** **if** 48 ≤ i ≤ 63 **then**

F := C **xor** (B **or** (**not** D))

g := (7×i) **mod** 16

F := F + A + K[i] + M[g] *// M[g] — 32-битный блок*

A := D

D := C

C := B

B := B + (F <<< s[i]) *// Выполняем битовый сдвиг*

**end** **for**

*// Прибавляем результат текущего "куска" к общему результату*

a0 := a0 + A

b0 := b0 + B

c0 := c0 + C

d0 := d0 + D

**end** **for**

**var** char digest[16] := a0 append b0 append c0 append d0 *// (Результат в формате little-endian)*

**4. Описание реализации:**

Язык реализации: Java.

Функции класса:

public static byte[] hash(byte[] input) – Хэширование байтового входа произвольной длины. Выход – байты хэша текста

public static String byteToHex(byte num) – используется для преобразования байта в читаемый вид HEX

private static int FF(int a, int b, int c, int d, int x, int s) – ф-я 1го раунда

private static int GG(int a, int b, int c, int d, int x, int s) ) – ф-я 2го раунда

private static int HH(int a, int b, int c, int d, int x, int s) ) – ф-я 3го раунда

Функции анализа:

public static long secondPreimage(String source, int bit, int rounds) – подбор rounds вторых прообразов длины bit для входной строки source

public static long collisions(int bit, int rounds) – поиск rounds коллизий первых bit бит

**5. Примеры работы программы:**

**5.1** Проверка совпадения хеша, полученного при выполнении реализованного алгоритма и хеша из стандарта:

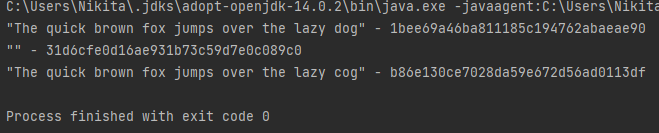


Рис 1. Примеры работы программы

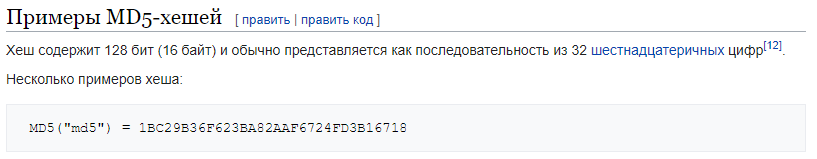


Рис 2. Примеры из стандарта

Из примеров видно, что результат работы программы совпадает с подлинными примерами.

**5.2** Поиск второго прообраза и коллизий для длин последовательностей 8, 12, 16, 20 и 24 бит:

Рис 3. Результат работы программы

**5.3** Поиск средней сложности второго прообраза и коллизий (1000 итераций):

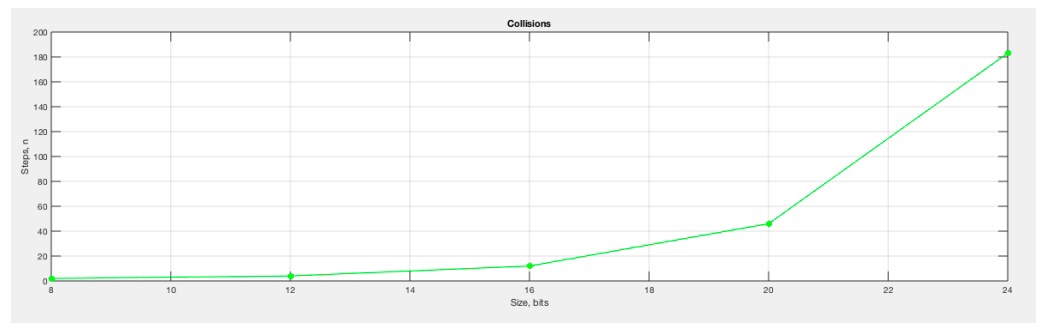
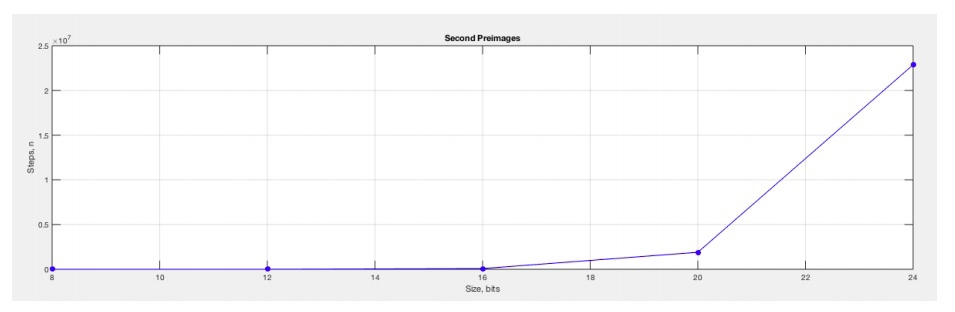


Рис 4. Средняя сложность второго прообраза и коллизий для 1000 экспериментов

Из графиков средней сложности второго прообраза и коллизии видно, что сложность второго прообраза гораздо больше. Сложность нахождения второго прообраза ~ , где n – количество бит, которые мы сравниваем. Сложность нахождения коллизии ~ .

**6. Вывод:**

Был реализован алгоритм вычисления хешей данных MD5. Результаты хеширования тестовых данных сопоставлены с подлинными. Выполнен поиск второго прообраза и коллизии и высчитаны средние сложности их нахождения. По результатам проведенных тестов можно сделать вывод о том, что атака по нахождению коллизии гораздо выгоднее чем атака на второй прообраз.

Ранее считалось, что MD5 позволяет получать относительно надёжный идентификатор для блока данных. На данный момент данная хеш-функция не рекомендуется к использованию, так как существуют способы нахождения коллизий с приемлемой вычислительной сложностью.

Свойство уникальности хеша широко применяется в разных областях. С помощью MD5 проверяли целостность и подлинность скачанных файлов — так, некоторые программы поставляются вместе со значением контрольной суммы. Например, пакеты для инсталляции свободного ПО.