Network Working Group
Request for Comments: 3174
Category: Informational

D. Eastlake, 3rd
Motorola
P. Jones
Cisco Systems

September 2001

US Secure Hash Algorithm 1 (SHA1)

Алгоритм хеширования SHA1 (США)

Статус документа

Этот документ содержит информацию для сообщества Internet и не задаёт каких-либо стандартов Internet. Документ можно распространять без ограничений.

Авторские права

Аннотация

Этот документ стремится сделать алгоритм хэширования SHA-1 (Secure Hash Algorithm 1) удобным для сообщества Internet. В США описанный здесь алгоритм SHA-1 принят как федеральный стандарт обработки информации (Federal Information Processing Standard) и большая часть приведённого ниже текста заимствована из документа FIPS 180-1. «Оригинальным» является лишь код на языке C.

Благодарности

Большая часть текста документа заимствована из [FIPS 180-1]. Реализация кода C является «оригинальной», но стиль похож на опубликованные ранее в RFC для MD4 и MD5 [RFC 1320, RFC 1321].

Алгоритм SHA-1 основан на принципах, похожих на использованные профессором Ronald L. Rivest из MIT при создании алгоритма цифровых дайджестов MD4 [MD4] и промоделированные в [RFC 1320].

Спасибо Tony Hansen, Garrett Wollman за полезные комментарии.

Оглавление

1. Содержимое документа

Примечание. Приведённый ниже текст в основном заимствован из [FIPS 180-1] и заявления о безопасности SHA-1 сделаны правительством США, автором [FIPS 180-1], а не авторами этого документа.

Этот документ определяет защищённый алгоритм хэширования SHA-1 для расчёта сжатого представления (свёртки) сообщения или файла данных. Принимая на входе сообщение размером меньше 264 битов, SHA-1 даёт на выходе 160-битовый дайджест сообщения. Этот дайджест затем может быть вводом, например, алгоритма подписи, который создаёт и проверяет подпись для сообщения. Подписание дайджеста, а не самого соединения зачастую повышает эффективность процесса, поскольку размер дайджеста обычно значительно меньше размера сообщения. Проверяющий цифровую подпись должен использовать тот же алгоритм хэширования, который применялся при создании подписи. Любое изменение сообщения при передаче будет с высокой вероятностью приводить к другому дайджесту и проверка подписи приведёт к отказу.

Алгоритм SHA-1 считается защищённым, поскольку невозможно вычислительными средствами найти сообщение которое будет соответствовать данному дайджесту, или два разных сообщения, дайджесты которых совпадут. Любое изменение сообщения при передаче будет с высокой вероятностью приводить к другому дайджесту и проверка подписи приведёт к отказу.

В разделе 2 определены термины и функции, применяемые в SHA-1.

2. Определения битовых строк и целых чисел

Ниже приведены соглашения, используемые для битовых строк и целых чисел.

a. Шестнадцатеричной (hex) цифрой считается элемент из множества {0, 1, ... , 9, A, ... , F}. Такие цифры представляются 4 битами. Например, 7 = 0111, A = 1010.

b. Словом (word) является 32-битовая строка, которую можно представить последовательностью из 8 шестнадцатеричных цифр. Для преобразования word в 8 hex-цифр каждая 4-битовая строка преобразуется в символьный элемент из п. a. Например,

```
1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.
```

c. Целые числа из диапазона от 0 до 2^{32} - 1 (включительно) можно представить как word. Младшие 4 бита числа представляет самая правая hex-цифра в представлении word. Например, целое число 291 = $2^8+2^5+2^1+2^0$ = 256+32+2+1 представляется шестнадцатеричным словом 00000123.

Если z - целое число из диапазона 0 - 2^{64}, то z = (2^{32})x + y, где 0 <= x < 2^{32} и 0 <= y < 2^{32}. Поскольку x и y можно представить как слова X и Y, соответственно, z можно представить парой (X,Y).

d. block - строка из 512 битов. Блок (например, B) можно представить последовательностью из 16 word.

3. Операции со словами

Ниже указаны логические операции, применяемые к словам (word).

a) Побитовые логические операции для word.

X AND Y = побитовая логическая операция И (AND) для X и Y.

X OR Y = побитовая логическая операция ИЛИ (OR) для X и Y.

X XOR Y = побитовая логическая операция Исключительное-ИЛИ (XOR) для X и Y.

NOT X = побитовое логическое «дополнение» X.

Пример

```
          01101100101110011101001001111011
XOR       01100101110000010110100110110111
          --------------------------------
  =       00001001011110001011101111001100
```

b) Операция X + Y определяется следующим образом: слова X и Y представляют целые числа x и y из диапазона от 0 до 2^{32} - 1 (включительно). Пусть для положительных чисел n и значение n mod m является остатком от деления n на m. Тогда

```
z = (x + y) mod 2^32.
```

При этом 0 <= z < 2^{32}. Значение z преобразуется в word (Z) и определяется Z = X + Y.

c) Операция циклического сдвига влево $S^{n(X)}$, где X - слово, а n - целое число от 0 до 31, определяется выражением

```
S^n(X) = (X << n) OR (X >> 32-n).
```

В приведённом примере X << n выполняется следующим образом: отбрасываются n битов X слева, оставшиеся биты переносятся на n позиций влево, а справа добавляется n нулей (результат остаётся 32-битовым). X >> n выполняется путём отбрасывания n справа, переноса оставшихся битов X на n позиций вправо и заполнение нулями n битов слева. Таким образом, Sn(X) эквивалентно циклическому сдвигу X на n позиций влево.

4. Дополнение сообщений

SHA-1 применяется для расчёта дайджеста сообщения или файла данных, переданного на вход алгоритма. Сообщение или файл данных следует рассматривать как строку. Размер сообщения определяется числом битов в нем (пустое сообщение имеет размер 0). Если число битов сообщения кратно 8, для компактности сообщение можно представить шестнадцатеричными цифрами. Чтобы сделать размер сообщения кратный 512 битам, применяется заполнение. При расчёте дайджеста SHA-1 последовательно обрабатывает 512-битовые блоки. Процесс заполнения описан ниже. В итоге в конце сообщения помещается "1", затем m нулей и 64-битовое целое число (размер исходного сообщения), чтобы размер сообщения составлял 512 * n. Дополненное сообщение обрабатывается SHA-1 как цепочка из n блоков по 512 битов.

Предположим, что сообщение имеет размер l < 2^{64}. До передачи в SHA-1, сообщение дополняется, как показано ниже.

a) В конец добавляется 1. Например, исходное сообщение 01010000 станет 010100001.

b) В конец добавляются нули (0), число которых зависит от размера исходного сообщения. Последние 64 бита последнего 512-битового блока резервируются для размера исходного сообщения. Предположим, что исходное сообщение имеет вид

```
01100001 01100010 01100011 01100100 01100101.
```

После п. a) оно приобретёт форму

```
01100001 01100010 01100011 01100100 01100101 1
```

Так как l = 40, число битов в приведённой выше строке составит 41 и в конце сообщения будет добавлено 407 нулей, после чего сообщение достигнет размера 448 битов и примет вид (в шестнадцатеричной форме)

```
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000
```

c) Создаётся 64-битовое (два word) представление l - числа битов в исходном сообщении. Если $l < 2^{32}$, первое слово заполняется нулями. Оба полученных слова добавляются в конец сообщения.

Например, для сообщения из b) размер l = 40 (до заполнения). 64-битовое представление числа 40 имеет вид 00000000 00000028 и окончательный блок (сообщение) становится

```
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000028
```

Дополненное сообщение содержит 16 * n слов, где n > 0. Дополненное сообщение рассматривается как последовательность из n блоков M(1) , M(2).

5. Функции и константы

В SHA-1 применяется последовательность логических функций f(0), f(1),..., f(79). Каждая функция f(t) (0 <= t <= 79) работает с тремя 32-битовыми словами (B, C, D) и даёт на выход 32-битовое слово (word). Функция f(t;B,C,D) для B, C, D определена ниже.

```
f(t;B,C,D) = (B AND C) OR ((NOT B) AND D)          ( 0 <= t <= 19)
f(t;B,C,D) = B XOR C XOR D                         (20 <= t <= 39)
f(t;B,C,D) = (B AND C) OR (B AND D) OR (C AND D)   (40 <= t <= 59)
f(t;B,C,D) = B XOR C XOR D                         (60 <= t <= 79).
```

В SHA-1 применяется последовательность констант (word) K(0), K(1), ... , K(79), показанных ниже.

```
K(t) = 5A827999          ( 0 <= t <= 19)
K(t) = 6ED9EBA1          (20 <= t <= 39)
K(t) = 8F1BBCDC          (40 <= t <= 59)
K(t) = CA62C1D6          (60 <= t <= 79)
```

6. Расчёт дайджеста сообщения

Методы, описанные в параграфах 6.1 и 6.2, дают одинаковый результат. Метод 2 позволяет сэкономить 64 32-битовых слова памяти, но это увеличивает время расчёта в результате роста сложности вычисления адресов для { W[t] } в c). Имеются иные методы расчёта с такими же результатами.

6.1. Метод 1

Дайджест сообщения рассчитывается с использованием дополнения, описанного в разделе 4. расчёт описывается с использованием двух буферов, каждый из которых включает пять 32-битовых слов, и последовательности из 80 32-битовых слов. Слова первого буфера обозначены A, B, C, D, E, второго - H0, H1, H2, H3, H4, слова 80-словной последовательности - W(0), W(1),..., W(79). Используется также буфер TEMP типа word.

Для генерации дайджеста блоки по 16 слов M(1), M(2),..., M(n), определённые в разделе 4, обрабатываются по порядку. Обработка каждого M(i) включает 80 шагов. Перед обработкой блоков переменные H инициализируются, как показано ниже.

```
H0 = 67452301
H1 = EFCDAB89
H2 = 98BADCFE
H3 = 10325476
H4 = C3D2E1F0.
```

Затем обрабатываются блоки M(1), M(2), ... , M(n). Обработка блока M(i) описана ниже.

a) M(i) делится на 16 слов (word) W(0), W(1), ... , W(15), где W(0) является первым слева.

b) Для t от 16 до 79 (включительно)

```
W(t) = S^1(W(t-3) XOR W(t-8) XOR W(t-14) XOR W(t-16))
```

c) Пусть A = H0, B = H1, C = H2, D = H3, E = H4.

d) Для t от 0 до 79 (включительно)

```
TEMP = S^5(A) + f(t;B,C,D) + E + W(t) + K(t);
E = D;  D = C;  C = S^30(B);  B = A; A = TEMP;
```

e) Принимается H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E.

После обработки M(n) дайджест сообщения будет представлять 160-битовую строку из 5 слов (word)

```
H0 H1 H2 H3 H4.
```

6.2. Метод 2

Описанный выше метод предполагает, что последовательность W(0), ... , W(79) реализована как массив из восьмидесяти 32-битовых слов. Это эффективно с точки зрения времени выполнения, поскольку адреса W(t-3), ... ,W(t-16) на этапе b) рассчитываются легко. Если важнее экономия памяти, можно рассматривать { W(t) } как циклическую очередь, которую можно реализовать в виде массива из 16 32-битовых слов W[0], ... W[15]. Пусть MASK = 0000000F. Тогда обработка M(i) будет иметь представленный ниже вид

a) M(i) делится на 16 слов W[0], ... , W[15], где W(0) является первым слева.

b) Пусть A = H0, B = H1, C = H2, D = H3, E = H4.

c) Для t от 0 до 79 (включительно)

```
s = t AND MASK;
if (t >= 16) W[s] = S^1(W[(s + 13) AND MASK] XOR W[(s + 8) AND
MASK] XOR W[(s + 2) AND MASK] XOR W[s]);
TEMP = S^5(A) + f(t;B,C,D) + E + W[s] + K(t);
```

```
        E = D; D = C; C = S^30(B); B = A; A = TEMP;
```

d) Принимается H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E.

7. Код C

Ниже приведена демонстрационная реализация SHA-1 на языке C. В параграфе 7.1 содержится заголовочный файл, в 7.2 - код C, а в 7.3 - тестовый драйвер.

7.1. Файл .h

```
/*
 *  sha1.h
 *
 *  Описание
 *      Это файл заголовков для кода, реализующего алгоритм SHA-1,
 *      определённый в FIPS PUB 180-1, опубликованном 17 апреля 1995 г.
 *
 *      Многие из имён переменных в этом коде заимствованы из
 *      публикации алгоритма.
 *
 *      Дополнительные сведения приведены в файле sha1.c.
 *
 */

#ifndef _SHA1_H_
#define _SHA1_H_

#include <stdint.h>
/*
 * если вы не используете стандартный файл ISO stdint.h, нужно задать
 * приведённые ниже определения.
 *    имя              назначение
 *  uint32_t         32-битовое целое число без знака
 *  uint8_t          8-битовое целое число без знака (т. е. unsigned char)
 *  int_least16_t    целое число размером 16 битов или больше
 *
 */

#ifndef _SHA_enum_
#define _SHA_enum_
enum
{
    shaSuccess = 0,
    shaNull,            /* параметр Null-указателя */
    shaInputTooLong,    /* слишком много входных данных */
    shaStateError       /* Вызывать Input после Result */
};
#endif
#define SHA1HashSize 20

/*
 *  В этой структуре хранятся контекстные данные для операции
 *  хэширования SHA-1.
 */
typedef struct SHA1Context
{
    uint32_t Intermediate_Hash[SHA1HashSize/4]; /* дайджест сообщения */

    uint32_t Length_Low;            /* Размер сообщения в битах */
    uint32_t Length_High;           /* Размер сообщения в битах */


    int_least16_t Message_Block_Index; /* Индекса массива блоков */
    uint8_t Message_Block[64];      /* 512-битовые блоки сообщения */

    int Computed;               /* Дайджест рассчитан? */
    int Corrupted;              /* Дайджест повреждён? */
} SHA1Context;

/*
 *  Прототипы функций
 */

int SHA1Reset(  SHA1Context *);
int SHA1Input(  SHA1Context *,
                const uint8_t *,
                unsigned int);
int SHA1Result( SHA1Context *,
                uint8_t Message_Digest[SHA1HashSize]);

#endif
```

7.2. Файл .c

```
/*
 *  sha1.c
```

```
 *
 *  Описание
 *      Этот файл реализует SHA-1, определённый в
 *      FIPS PUB 180-1, опубликованном 17 апреля 1995 г.
 *
 *      SHA-1 создаёт 160-битовый дайджест для представленного
 *      потока данных. Требуется около 2**n шагов чтобы найти
 *      сообщение с таким же дайджестом как у данного сообщения
 *      и около 2**(n/2) на нахождения двух сообщений с одинаковым
 *      дайджестом, где n - размер дайджеста в битах. Поэтому
 *      алгоритм можно применять для создания отпечатка сообщений
 *      (fingerprint).
 *
 *  Проблемы переносимости
 *      SHA-1 определён в терминах 32-битовых слов (word). Этот код
 *      использует файл <stdint.h> (включён из sha1.h) для определения
 *      32- и 8-битовых целых чисел без знака. Если компилятор C не
 *      поддерживает 32-битовые целые числа без знака, этот код не
 *      подойдёт.
 *
 *  Предостережения
 *      SHA-1 разработан для сообщений размером меньше 2^64 битов.
 *      Хотя SHA-1 позволяет создавать дайджесты для сообщений с
 *      любым числом битов меньше 2^64, данная реализация работает
 *      лишь с сообщениями, размер которых кратен 8 битам.
 *
 */

#include "sha1.h"

/*
 *  Определение макроса циклического сдвига влево.
 */
#define SHA1CircularShift(bits,word) \
                (((word) << (bits)) | ((word) >> (32-(bits))))

/* Локальные прототипы функций */
void SHA1PadMessage(SHA1Context *);
void SHA1ProcessMessageBlock(SHA1Context *);

/*
 *  SHA1Reset
 *
 *  Описание
 *      Эта функция инициализирует SHA1Context при подготовке к
 *      расчёту нового дайджеста SHA1.
 *
 *  Параметры
 *      context: [in/out]
 *          Контекст для инициализации.
 *
 *  Возвращает код ошибки.
 *
 */
int SHA1Reset(SHA1Context *context)
{
    if (!context)
    {
        return shaNull;
    }

    context->Length_Low             = 0;
    context->Length_High            = 0;
    context->Message_Block_Index    = 0;

    context->Intermediate_Hash[0]   = 0x67452301;
    context->Intermediate_Hash[1]   = 0xEFCDAB89;
    context->Intermediate_Hash[2]   = 0x98BADCFE;
    context->Intermediate_Hash[3]   = 0x10325476;
    context->Intermediate_Hash[4]   = 0xC3D2E1F0;

    context->Computed   = 0;
    context->Corrupted  = 0;

    return shaSuccess;
}

/*
 *  SHA1Result
 *
 *  Описание
 *      Эта функция возвращает 160-битовый дайджест сообщения в
 *      массиве Message_Digest, предоставленном вызывающим.
 *      Примечание. Первый октет хэша сохраняется в элементе 0,
 *                  последний - в элементе массива 19.
 *
```

```
 *  Параметры
 *      context: [in/out]
 *          Контекст для расчёта хэша SHA-1.
 *      Message_Digest: [out]
 *          Возвращаемый функцией дайджест.
 *
 *  Возвращает код ошибки.
 *
 */
int SHA1Result( SHA1Context *context,
                uint8_t Message_Digest[SHA1HashSize])
{
    int i;

    if (!context || !Message_Digest)
    {
        return shaNull;
    }

    if (context->Corrupted)
    {
        return context->Corrupted;
    }

    if (!context->Computed)
    {
        SHA1PadMessage(context);
        for(i=0; i<64; ++i)
        {
            /* Сообщение может быть секретным, переменная очищается */
            context->Message_Block[i] = 0;
        }
        context->Length_Low = 0;    /* Размер очистки */
        context->Length_High = 0;
        context->Computed = 1;
    }

    for(i = 0; i < SHA1HashSize; ++i)
    {
        Message_Digest[i] = context->Intermediate_Hash[i>>2]
                            >> 8 * ( 3 - ( i & 0x03 ) );
    }

    return shaSuccess;
}

/*
 *  SHA1Input
 *
 *  Описание
 *      Функция принимает следующую часть сообщения как массив октетов.
 *
 *  Параметры
 *      context: [in/out]
 *          Контекст SHA для обновления
 *      message_array: [in]
 *          Массив символов следующей части сообщения.
 *      length: [in]
 *          Размер сообщения в message_array
 *
 *  Возвращает код ошибки.
 *
 */
int SHA1Input(    SHA1Context    *context,
                  const uint8_t  *message_array,
                  unsigned       length)
{
    if (!length)
    {
        return shaSuccess;
    }

    if (!context || !message_array)
    {
        return shaNull;
    }

    if (context->Computed)
    {
        context->Corrupted = shaStateError;

        return shaStateError;
    }

    if (context->Corrupted)
    {
```

```
        return context->Corrupted;
    }
    while(length-- && !context->Corrupted)
    {
    context->Message_Block[context->Message_Block_Index++] =
                    (*message_array & 0xFF);

    context->Length_Low += 8;
    if (context->Length_Low == 0)
    {
        context->Length_High++;
        if (context->Length_High == 0)
        {
            /* Слишком длинное сообщение */
            context->Corrupted = 1;
        }
    }

    if (context->Message_Block_Index == 64)
    {
        SHA1ProcessMessageBlock(context);
    }

    message_array++;
    }

    return shaSuccess;
}

/*
 *  SHA1ProcessMessageBlock
 *
 *  Описание
 *      Функция обрабатывает следующие 512 битов сообщения,
 *      хранящиеся в массиве Message_Block.
 *
 *  Параметров нет.
 *
 *  Функция не возвращает значения.
 *
 *  Комментарий
 *      Многие имена переменных в этом коде, особенно 1-символьные,
 *      заимствованы из спецификации алгоритма.
 *
 *
 */
void SHA1ProcessMessageBlock(SHA1Context *context)
{
    const uint32_t K[] =    {       /* Константы, заданные в SHA-1 */
                            0x5A827999,
                            0x6ED9EBA1,
                            0x8F1BBCDC,
                            0xCA62C1D6
                            };
    int           t;                /* Счётчик циклов */
    uint32_t      temp;             /* Временное значение слова */
    uint32_t      W[80];            /* Последовательность слов */
    uint32_t      A, B, C, D, E;    /* Буферы для слов */

    /*
     *  Инициализация первых 16 слов массива W
     */
    for(t = 0; t < 16; t++)
    {
        W[t] = context->Message_Block[t * 4] << 24;
        W[t] |= context->Message_Block[t * 4 + 1] << 16;
        W[t] |= context->Message_Block[t * 4 + 2] << 8;
        W[t] |= context->Message_Block[t * 4 + 3];
    }

    for(t = 16; t < 80; t++)
    {
       W[t] = SHA1CircularShift(1,W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
    }

    A = context->Intermediate_Hash[0];
    B = context->Intermediate_Hash[1];
    C = context->Intermediate_Hash[2];
    D = context->Intermediate_Hash[3];
    E = context->Intermediate_Hash[4];

    for(t = 0; t < 20; t++)
    {
        temp =  SHA1CircularShift(5,A) +
                ((B & C) | ((~B) & D)) + E + W[t] + K[0];
        E = D;
```

```
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 20; t < 40; t++)
    {
        temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 40; t < 60; t++)
    {
        temp = SHA1CircularShift(5,A) +
               ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 60; t < 80; t++)
    {
        temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[3];
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    context->Intermediate_Hash[0] += A;
    context->Intermediate_Hash[1] += B;
    context->Intermediate_Hash[2] += C;
    context->Intermediate_Hash[3] += D;
    context->Intermediate_Hash[4] += E;

    context->Message_Block_Index = 0;
}


/*
 *  SHA1PadMessage
 *
 *  Описание
 *      В соответствии со стандартом сообщение должно дополняться до
 *      размера, кратного 512 битам. Первый бит дополнения имеет значение 1.
 *      В последних 64 битах дополнения указывается размер исходного
 *      сообщения. Остальные биты имеют значение 0. Эта функция дополняет
 *      сообщение в соответствии с указанными выше правилами путём заполнения
 *      массива Message_Block. Функция также вызывает ProcessMessageBlock.
 *      При возврате предполагается что дайджест рассчитан.
 *
 *  Параметры
 *      context: [in/out]
 *          Контекст для дополнения
 *      ProcessMessageBlock: [in]
 *          Подходящая функция SHA*ProcessMessageBlock.
 *
 *  Функция не возвращает значения.
 *
 */

void SHA1PadMessage(SHA1Context *context)
{
    /*
     *  Проверяет, не слишком ли мал текущий блок сообщения для размещения
     *  начальных битов заполнения и размера. Если блок мал, он дополняется,
     *  обрабатывается, а затем заполнение продолжается в другом блоке.
     */
    if (context->Message_Block_Index > 55)
    {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
        while(context->Message_Block_Index < 64)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }

        SHA1ProcessMessageBlock(context);
```

```
        while(context->Message_Block_Index < 56)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }
    }
    else
    {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
        while(context->Message_Block_Index < 56)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }
    }

    /*
     *  Сохранение размера сообщения в последних 8 октетах.
     */
    context->Message_Block[56] = context->Length_High >> 24;
    context->Message_Block[57] = context->Length_High >> 16;
    context->Message_Block[58] = context->Length_High >> 8;
    context->Message_Block[59] = context->Length_High;
    context->Message_Block[60] = context->Length_Low >> 24;
    context->Message_Block[61] = context->Length_Low >> 16;
    context->Message_Block[62] = context->Length_Low >> 8;
    context->Message_Block[63] = context->Length_Low;

    SHA1ProcessMessageBlock(context);
}
```

7.3. Тестовый драйвер

Ниже приведён текст программы тестового драйвера для кода sha1.c.

```
/*
 *  sha1test.c
 *
 *  Описание
 *      Этот файл служит для выполнения кода SHA-1 в трёх тестах,
 *      указанных в FIPS PUB 180-1, а также дополнительного теста
 *      с вызовом SHA1Input для сообщения размером, кратным 512 битам,
 *      а также проверки кодов ошибок.
 *
 *  Проблем переносимости нет.
 *
 */

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "sha1.h"

/*
 *  Определение шаблонов для тестирования.
 */
#define TEST1   "abc"
#define TEST2a  "abcdbcdecdefdefgefghfghighijhi"
#define TEST2b  "jkijkljklmklmnlmnomnopnopq"
#define TEST2   TEST2a TEST2b
#define TEST3   "a"
#define TEST4a  "01234567012345670123456701234567"
#define TEST4b  "01234567012345670123456701234567"
    /* Кратно 512 битам */
#define TEST4   TEST4a TEST4b
char *testarray[4] =
{
    TEST1,
    TEST2,
    TEST3,
    TEST4
};
long int repeatcount[4] = { 1, 1, 1000000, 10 };
char *resultarray[4] =
{
    "A9 99 3E 36 47 06 81 6A BA 3E 25 71 78 50 C2 6C 9C D0 D8 9D",
    "84 98 3E 44 1C 3B D2 6E BA AE 4A A1 F9 51 29 E5 E5 46 70 F1",
    "34 AA 97 3C D4 C4 DA A4 F6 1E EB 2B DB AD 27 31 65 34 01 6F",
    "DE A3 56 A2 CD DD 90 C7 A7 EC ED C5 EB B5 63 93 4F 46 04 52"
};

int main()
{
    SHA1Context sha;
    int i, j, err;
    uint8_t Message_Digest[20];

    /*
     *  Выполнение тестов SHA-1
```

```
     */
    for(j = 0; j < 4; ++j)
    {
        printf( "\nTest %d: %d, '%s'\n",
                j+1,
                repeatcount[j],
                testarray[j]);

        err = SHA1Reset(&sha);
        if (err)
        {
            fprintf(stderr, "SHA1Reset Error %d.\n", err );
            break;    /* Выход из цикла j */
        }

        for(i = 0; i < repeatcount[j]; ++i)
        {
            err = SHA1Input(&sha,
                  (const unsigned char *) testarray[j],
                  strlen(testarray[j]));
            if (err)
            {
                fprintf(stderr, "SHA1Input Error %d.\n", err );
                break;    /* Выход из цикла i */
            }
        }

        err = SHA1Result(&sha, Message_Digest);
        if (err)
        {
            fprintf(stderr,
            "SHA1Result Error %d, could not compute message digest.\n",
            err );
        }
        else
        {
            printf("\t");
            for(i = 0; i < 20 ; ++i)
            {
                printf("%02X ", Message_Digest[i]);
            }
            printf("\n");
        }
        printf("Should match:\n");
        printf("\t%s\n", resultarray[j]);
    }

    /* проверка кодов ошибок */
    err = SHA1Input(&sha,(const unsigned char *) testarray[1], 1);
    printf ("\nError %d. Should be %d.\n", err, shaStateError );
    err = SHA1Reset(0);
    printf ("\nError %d. Should be %d.\n", err, shaNull );
    return 0;
}
```

8. Вопросы безопасности

Документ адресован сообществу Internet и предоставляет открытый код функции защищённого хэширования SHA-1 [FIPS 180-1]. Авторы не делают каких-либо заявлений о безопасности этой хэш-функции для конкретного применения.

Литература

[FIPS 180-1] "Secure Hash Standard", United States of American, National Institute of Science and Technology, Federal Information Processing Standard (FIPS) 180-1, April 1993.

[MD4] "The MD4 Message Digest Algorithm," Advances in Cryptology - CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 303-311.

[RFC 1320] Rivest, R., "The MD4 Message-Digest Algorithm", RFC 1320, April 1992.

[RFC 1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC 1750] Eastlake, D., Crocker, S. and J. Schiller, "Randomness Requirements for Security", RFC 1750, December 1994.

Адреса авторов

Donald E. Eastlake, 3rd
Motorola
155 Beaver Street
Milford, MA 01757 USA
Phone: +1 508-634-2066 (h)
+1 508-261-5434 (w)
Fax: +1 508-261-4777
EMail: Donald.Eastlake@motorola.com

Paul E. Jones

Cisco Systems, Inc.
7025 Kit Creek Road
Research Triangle Park, NC 27709 USA
Phone: +1 919 392 6948
EMail: paulej@packetizer.com

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru

Полное заявление авторских прав

Подтверждение

Финансирование функций RFC Editor обеспечено Internet Society.