                      US Secure Hash Algorithm 1 (SHA1)

Status of this Memo

Copyright Notice

Abstract

   The purpose of this document is to make the SHA-1 (Secure Hash
   Algorithm 1) hash algorithm conveniently available to the Internet
   community. The United States of America has adopted the SHA-1 hash
   algorithm described herein as a Federal Information Processing
   Standard. Most of the text herein was taken by the authors from FIPS
   180-1. Only the C code implementation is "original".

Acknowledgements

   Most of the text herein was taken from [FIPS 180-1]. Only the C code
   implementation is "original" but its style is similar to the
   previously published MD4 and MD5 RFCs [RFCs 1320, 1321].

   The SHA-1 is based on principles similar to those used by Professor
   Ronald L. Rivest of MIT when designing the MD4 message digest
   algorithm [MD4] and is modeled after that algorithm [RFC 1320].

   Useful comments from the following, which have been incorporated
   herein, are gratefully acknowledged:

      Tony Hansen
      Garrett Wollman

Table of Contents

1. Overview of Contents

   NOTE: The text below is mostly taken from [FIPS 180-1] and assertions
   therein of the security of SHA-1 are made by the US Government, the
   author of [FIPS 180-1], and not by the authors of this document.

   This document specifies a Secure Hash Algorithm, SHA-1, for computing
   a condensed representation of a message or a data file. When a
   message of any length < 2^64 bits is input, the SHA-1 produces a
   160-bit output called a message digest. The message digest can then,
   for example, be input to a signature algorithm which generates or
   verifies the signature for the message. Signing the message digest
   rather than the message often improves the efficiency of the process
   because the message digest is usually much smaller in size than the
   message. The same hash algorithm must be used by the verifier of a
   digital signature as was used by the creator of the digital
   signature. Any change to the message in transit will, with very high
   probability, result in a different message digest, and the signature
   will fail to verify.

   The SHA-1 is called secure because it is computationally infeasible
   to find a message which corresponds to a given message digest, or to
   find two different messages which produce the same message digest.
   Any change to a message in transit will, with very high probability,
   result in a different message digest, and the signature will fail to
   verify.

Section 2 below defines the terminology and functions used as
building blocks to form SHA-1.

2. Definitions of Bit Strings and Integers

   The following terminology related to bit strings and integers will be
   used:

   a. A hex digit is an element of the set {0, 1, ... , 9, A, ... , F}.
      A hex digit is the representation of a 4-bit string. Examples: 7
      = 0111, A = 1010.

   b. A word equals a 32-bit string which may be represented as a
      sequence of 8 hex digits. To convert a word to 8 hex digits each
      4-bit string is converted to its hex equivalent as described in
      (a) above. Example:

      1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.

   c. An integer between 0 and $2^{32} - 1$ inclusive may be represented as
      a word. The least significant four bits of the integer are
      represented by the right-most hex digit of the word
      representation. Example: the integer $291 = 2^8+2^5+2^1+2^0 =$
      $256+32+2+1$ is represented by the hex word, 00000123.

      If z is an integer, $0 \le z < 2^{64}$, then $z = (2^{32})x + y$ where $0 \le$
      $x < 2^{32}$ and $0 \le y < 2^{32}$. Since x and y can be represented as
      words X and Y, respectively, z can be represented as the pair of
      words (X,Y).

   d. block = 512-bit string. A block (e.g., B) may be represented as a
      sequence of 16 words.

3. Operations on Words

   The following logical operators will be applied to words:

   a. Bitwise logical word operations

      X AND Y = bitwise logical "and" of X and Y.

      X OR Y   = bitwise logical "inclusive-or" of X and Y.

      X XOR Y = bitwise logical "exclusive-or" of X and Y.

      NOT X   = bitwise logical "complement" of X.

Example:

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

b. The operation X + Y is defined as follows: words X and Y
   represent integers x and y, where 0 <= x < 2^32 and 0 <= y < 2^32.
   For positive integers n and m, let n mod m be the remainder upon
   dividing n by m. Compute

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

   Then 0 <= z < 2^32. Convert z to a word, Z, and define Z = X +
   Y.

c. The circular left shift operation S^n(X), where X is a word and n
   is an integer with 0 <= n < 32, is defined by

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

   In the above, X << n is obtained as follows: discard the left-most
   n bits of X and then pad the result with n zeroes on the right
   (the result will still be 32 bits). X >> n is obtained by
   discarding the right-most n bits of X and then padding the result
   with n zeroes on the left. Thus S^n(X) is equivalent to a
   circular shift of X by n positions to the left.

4. Message Padding

   SHA-1 is used to compute a message digest for a message or data file
   that is provided as input. The message or data file should be
   considered to be a bit string. The length of the message is the
   number of bits in the message (the empty message has length 0). If
   the number of bits in a message is a multiple of 8, for compactness
   we can represent the message in hex. The purpose of message padding
   is to make the total length of a padded message a multiple of 512.
   SHA-1 sequentially processes blocks of 512 bits when computing the
   message digest. The following specifies how this padding shall be
   performed. As a summary, a "1" followed by m "0"s followed by a 64-
   bit integer are appended to the end of the message to produce a
   padded message of length 512 * n. The 64-bit integer is the length
   of the original message. The padded message is then processed by the
   SHA-1 as n 512-bit blocks.

Suppose a message has length l < 2^64. Before it is input to the
SHA-1, the message is padded on the right as follows:

a. "1" is appended. Example: if the original message is "01010000",
   this is padded to "010100001".

b. "0"s are appended. The number of "0"s will depend on the original
   length of the message. The last 64 bits of the last 512-bit block
   are reserved

   for the length l of the original message.

   Example: Suppose the original message is the bit string

      01100001 01100010 01100011 01100100 01100101.

   After step (a) this gives

      01100001 01100010 01100011 01100100 01100101 1.

   Since l = 40, the number of bits in the above is 41 and 407 "0"s
   are appended, making the total now 448. This gives (in hex)

      61626364 65800000 00000000 00000000
      00000000 00000000 00000000 00000000
      00000000 00000000 00000000 00000000
      00000000 00000000.

c. Obtain the 2-word representation of l, the number of bits in the
   original message. If l < 2^32 then the first word is all zeroes.
   Append these two words to the padded message.

   Example: Suppose the original message is as in (b). Then l = 40
   (note that l is computed before any padding). The two-word
   representation of 40 is hex 00000000 00000028. Hence the final
   padded message is hex

      61626364 65800000 00000000 00000000
      00000000 00000000 00000000 00000000
      00000000 00000000 00000000 00000000
      00000000 00000000 00000000 00000028.

   The padded message will contain 16 * n words for some n > 0.
   The padded message is regarded as a sequence of n blocks M(1) ,
   M(2), first characters (or bits) of the message.

5. Functions and Constants Used

   A sequence of logical functions f(0), f(1),..., f(79) is used in
   SHA-1. Each f(t), 0 <= t <= 79, operates on three 32-bit words B, C,
   D and produces a 32-bit word as output. f(t;B,C,D) is defined as
   follows: for words B, C, D,

      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).

   A sequence of constant words K(0), K(1), ... , K(79) is used in the
   SHA-1. In hex these are given by

      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. Computing the Message Digest

   The methods given in 6.1 and 6.2 below yield the same message digest.
   Although using method 2 saves sixty-four 32-bit words of storage, it
   is likely to lengthen execution time due to the increased complexity
   of the address computations for the { W[t] } in step (c). There are
   other computation methods which give identical results.

6.1 Method 1

   The message digest is computed using the message padded as described
   in section 4. The computation is described using two buffers, each
   consisting of five 32-bit words, and a sequence of eighty 32-bit
   words. The words of the first 5-word buffer are labeled A,B,C,D,E.
   The words of the second 5-word buffer are labeled H0, H1, H2, H3, H4.
   The words of the 80-word sequence are labeled W(0), W(1),..., W(79).
   A single word buffer TEMP is also employed.

   To generate the message digest, the 16-word blocks M(1), M(2),...,
   M(n) defined in section 4 are processed in order. The processing of
   each M(i) involves 80 steps.

Before processing any blocks, the H's are initialized as follows: in
hex,

    H0 = 67452301

    H1 = EFCDAB89

    H2 = 98BADCFE

    H3 = 10325476

    H4 = C3D2E1F0.

Now M(1), M(2), ... , M(n) are processed. To process M(i), we
proceed as follows:

    a. Divide M(i) into 16 words W(0), W(1), ... , W(15), where W(0)
       is the left-most word.

    b. For t = 16 to 79 let

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

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

    d. For t = 0 to 79 do

       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. Let H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4
       + E.

After processing M(n), the message digest is the 160-bit string
represented by the 5 words

       H0 H1 H2 H3 H4.

6.2 Method 2

The method above assumes that the sequence W(0), ... , W(79) is
implemented as an array of eighty 32-bit words. This is efficient
from the standpoint of minimization of execution time, since the
addresses of W(t-3), ... ,W(t-16) in step (b) are easily computed.
If space is at a premium, an alternative is to regard { W(t) } as a

circular queue, which may be implemented using an array of sixteen
32-bit words W[0], ... W[15]. In this case, in hex let

MASK = 0000000F. Then processing of M(i) is as follows:

    a. Divide M(i) into 16 words W[0], ... , W[15], where W[0] is the
       left-most word.

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

    c. For t = 0 to 79 do

       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. Let H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4
       + E.

7. C Code

   Below is a demonstration implementation of SHA-1 in C. Section 7.1
   contains the header file, 7.2 the C code, and 7.3 a test driver.

7.1 .h file

```
/*
 *  sha1.h
 *
 *  Description:
 *      This is the header file for code which implements the Secure
 *      Hashing Algorithm 1 as defined in FIPS PUB 180-1 published
 *      April 17, 1995.
 *
 *      Many of the variable names in this code, especially the
 *      single character names, were used because those were the names
 *      used in the publication.
 *
 *      Please read the file sha1.c for more information.
 *
 */
```

```
#ifndef _SHA1_H_
#define _SHA1_H_

#include <stdint.h>
/*
 * If you do not have the ISO standard stdint.h header file, then you
 * must typdef the following:
 *     name               meaning
 *  uint32_t         unsigned 32 bit integer
 *  uint8_t          unsigned 8 bit integer (i.e., unsigned char)
 *  int_least16_t    integer of >= 16 bits
 *
 */

#ifndef _SHA_enum_
#define _SHA_enum_
enum
{
    shaSuccess = 0,
    shaNull,            /* Null pointer parameter */
    shaInputTooLong,    /* input data too long */
    shaStateError       /* called Input after Result */
};
#endif
#define SHA1HashSize 20

/*
 *  This structure will hold context information for the SHA-1
 *  hashing operation
 */
typedef struct SHA1Context
{
    uint32_t Intermediate_Hash[SHA1HashSize/4]; /* Message Digest  */

    uint32_t Length_Low;             /* Message length in bits     */
    uint32_t Length_High;            /* Message length in bits     */

                                 /* Index into message block array  */
    int_least16_t Message_Block_Index;
    uint8_t Message_Block[64];       /* 512-bit message blocks      */

    int Computed;                 /* Is the digest computed?        */
    int Corrupted;                /* Is the message digest corrupted? */
} SHA1Context;

/*
 *  Function Prototypes
 */
```

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

#endif
```

7.2 .c file

```
/*
 *  sha1.c
 *
 *  Description:
 *      This file implements the Secure Hashing Algorithm 1 as
 *      defined in FIPS PUB 180-1 published April 17, 1995.
 *
 *      The SHA-1, produces a 160-bit message digest for a given
 *      data stream.  It should take about 2**n steps to find a
 *      message with the same digest as a given message and
 *      2**(n/2) to find any two messages with the same digest,
 *      when n is the digest size in bits.  Therefore, this
 *      algorithm can serve as a means of providing a
 *      "fingerprint" for a message.
 *
 *  Portability Issues:
 *      SHA-1 is defined in terms of 32-bit "words".  This code
 *      uses <stdint.h> (included via "sha1.h" to define 32 and 8
 *      bit unsigned integer types.  If your C compiler does not
 *      support 32 bit unsigned integers, this code is not
 *      appropriate.
 *
 *  Caveats:
 *      SHA-1 is designed to work with messages less than 2^64 bits
 *      long.  Although SHA-1 allows a message digest to be generated
 *      for messages of any number of bits less than 2^64, this
 *      implementation only works with messages with a length that is
 *      a multiple of the size of an 8-bit character.
 *
 */
```

```
#include "sha1.h"

/*
 *  Define the SHA1 circular left shift macro
 */
#define SHA1CircularShift(bits,word) \
                (((word) << (bits)) | ((word) >> (32-(bits))))

/* Local Function Prototyptes */
void SHA1PadMessage(SHA1Context *);
void SHA1ProcessMessageBlock(SHA1Context *);

/*
 *  SHA1Reset
 *
 *  Description:
 *      This function will initialize the SHA1Context in preparation
 *      for computing a new SHA1 message digest.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to reset.
 *
 *  Returns:
 *      sha Error Code.
 *
 */
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
 *
 *  Description:
 *      This function will return the 160-bit message digest into the
 *      Message_Digest array  provided by the caller.
 *      NOTE: The first octet of hash is stored in the 0th element,
 *            the last octet of hash in the 19th element.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to use to calculate the SHA-1 hash.
 *      Message_Digest: [out]
 *          Where the digest is returned.
 *
 *  Returns:
 *      sha Error Code.
 *
 */
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)
        {
            /* message may be sensitive, clear it out */
            context->Message_Block[i] = 0;
        }
        context->Length_Low = 0;    /* and clear length */
        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
 *
 *  Description:
 *      This function accepts an array of octets as the next portion
 *      of the message.
 *
 *  Parameters:
 *      context: [in/out]
 *          The SHA context to update
 *      message_array: [in]
 *          An array of characters representing the next portion of
 *          the message.
 *      length: [in]
 *          The length of the message in message_array
 *
 *  Returns:
 *      sha Error Code.
 *
 */
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)
        {
            /* Message is too long */
            context->Corrupted = 1;
        }
    }

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

    message_array++;
    }

    return shaSuccess;
}

/*
 *  SHA1ProcessMessageBlock
 *
 *  Description:
 *      This function will process the next 512 bits of the message
 *      stored in the Message_Block array.
 *
 *  Parameters:
 *      None.
 *
 *  Returns:
 *      Nothing.
 *
 *  Comments:
```

```
 *      Many of the variable names in this code, especially the
 *      single character names, were used because those were the
 *      names used in the publication.
 *
 *
 */
void SHA1ProcessMessageBlock(SHA1Context *context)
{
    const uint32_t K[] =    {        /* Constants defined in SHA-1   */
                            0x5A827999,
                            0x6ED9EBA1,
                            0x8F1BBCDC,
                            0xCA62C1D6
                            };
    int           t;                 /* Loop counter                 */
    uint32_t      temp;              /* Temporary word value         */
    uint32_t      W[80];             /* Word sequence                */
    uint32_t      A, B, C, D, E;     /* Word buffers                 */

    /*
     *  Initialize the first 16 words in the array 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
 *
```

```
 *   Description:
 *       According to the standard, the message must be padded to an even
 *       512 bits.  The first padding bit must be a '1'.  The last 64
 *       bits represent the length of the original message.  All bits in
 *       between should be 0.  This function will pad the message
 *       according to those rules by filling the Message_Block array
 *       accordingly.  It will also call the ProcessMessageBlock function
 *       provided appropriately.  When it returns, it can be assumed that
 *       the message digest has been computed.
 *
 *   Parameters:
 *       context: [in/out]
 *           The context to pad
 *       ProcessMessageBlock: [in]
 *           The appropriate SHA*ProcessMessageBlock function
 *   Returns:
 *       Nothing.
 *
 */

void SHA1PadMessage(SHA1Context *context)
{
    /*
     *  Check to see if the current message block is too small to hold
     *  the initial padding bits and length.  If so, we will pad the
     *  block, process it, and then continue padding into a second
     *  block.
     */
    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;
        }
    }

    /*
     *  Store the message length as the last 8 octets
     */
    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 Test Driver

   The following code is a main program test driver to exercise the code
   in sha1.c.

```
/*
 *  sha1test.c
 *
 *  Description:
 *      This file will exercise the SHA-1 code performing the three
 *      tests documented in FIPS PUB 180-1 plus one which calls
 *      SHA1Input with an exact multiple of 512 bits, plus a few
 *      error test checks.
 *
 *  Portability Issues:
 *      None.
 *
 */

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

/*
 *  Define patterns for testing
 */
#define TEST1    "abc"
#define TEST2a   "abcdbcdecdefdefgefghfghighijhi"
```

```
#define TEST2b  "jkijkljklmklmnlmnomnopnopq"
#define TEST2   TEST2a TEST2b
#define TEST3   "a"
#define TEST4a  "01234567012345670123456701234567"
#define TEST4b  "01234567012345670123456701234567"
    /* an exact multiple of 512 bits */
#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];

    /*
     *  Perform SHA-1 tests
     */
    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;    /* out of for j loop */
        }

        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;    /* out of for i loop */
            }
        }

        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]);
    }

    /* Test some error returns */
    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. Security Considerations

   This document is intended to provide convenient open source access by
   the Internet community to the United States of America Federal
   Information Processing Standard Secure Hash Function SHA-1 [FIPS
   180-1]. No independent assertion of the security of this hash
   function by the authors for any particular use is intended.

References

   [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.

Authors' Addresses

   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