DigSig v1.10 for Windows Help Contents

Select one from the following list to learn more about the subject:

Startup and exiting

When you run DigSig, it tries to automatically load a file called "SIGDBASE.SDB". This is the default database that holds your digital signatures. When you exit from DigSig, this file is automatically updated with all the signatures in the main Window.

If DigSig cannot find this default database then it will give you the opportunity to use the Windows file selector to try and locate it, or to start afresh with an empty database.

Using the main window

The main DigSig window contains a list of the signatures currently in the DigSig database. One of these signatures will be highlighted, this is the *current signature*. Many of the menu options operate on the current signature.

Each entry in the list holds four items of information: the date the signature was made, the full name of the file, the comment that you gave it and the signature itself.

You can use the up and down cursor keys to change the current signature. Alternatively you can also use the mouse to point and click on a signature or you can use the vertical slider bar to change the current signature.

Often, the entries in the list are too long to fit in the window. You can use the horizontal scroll bar or the left and right cursor keys to change the part of the entries that are shown in the window.

Menu shortcut

You can use the right mouse button as a shortcut for the last used menu. Clicking the right mouse button anywhere in the window will bring up the last used menu. This is a good way of quickly getting at frequently used menu options.

Edit shortcut

You can double-click the left mouse button on a signature entry as a shortcut for the "Edit" option in the "Entry" menu.

Menu options

Select an option from the following list to learn more about it.

File menu

- Open and sign
- Add signature file
- Save signature file as
- Print signature
- Exit

Edit menu

- Copy

Entry menu

- Edit
- Remove
- Recalculate
- Verify

Method menu

- SHS
- MD4
- MD5

Open and sign (Ctrl+O)

This option is used to locate a file, calculate its digital signature and add it to the list. The Windows standard file selector will appear which you should use to choose the name of the file whose signature you wish to calculate.

The time taken to calculate the signature depends on the size of the file. For most files the time will be negligible on reasonably fast computers but please do be patient.

When the signature has been calculated a dialogue box will appear that you should use to add a comment to the signature entry. Every entry must have a comment, but it does not matter if comments are the same.

Add signature file

This option can be used to add a previously saved signature file to the database. The Windows standard file selector will appear that you can use to locate the signature file.

DigSig signature files all end with a ".SIG" extension by default.

Save signature file as

This option allows you to save the current signature to disk as an independent signature file. The Windows standard file selector will appear that you can use to choose the name of the new signature file.

You should ensure that you give all signature files an extension of ".SIG" to distinguish them from other files. The *Add signature file* option can be used to add these signature files to the database at a later date.

Print signature (Ctrl+P)

This option is used to print the current signature (just the signature, not the details). The idea of this option is that you might use it to print a signature at the bottom of a letter that you have written, for example. The recipient of the letter can then call you to verify that it was indeed you who sent the letter since nobody else could possibly have generated a letter with the same signature.

The "Print" dialogue box allows you to specify the x and y co-ordinates on the page where the signature will be printed. For your convenience, you may specify these co-ordinates in either millimetres or inches.

The co-ordinate origin is at the top left hand corner of the page. X co-ordinates are measured to the right, across the page. Y co-ordinates are measured down the page.

You should also specify the font that you want to print the signature in. This allows you to choose a style that fits the overall "look" of your page.

Exit (Alt+F4)

This option will exit from DigSig, without further questioning. The signature database will be automatically saved in the file SIGDBASE.SIG, or, if you chose the name of another file when you first ran DigSig, it will be saved in that file.

Copy (Ctrl+Ins)

This option will copy the current signature to the Windows clipboard where you can access it from other applications, such as word-processors.

Edit (Ctrl+E)

This option allows you to change the comment that you gave the file when you first calculated its signature. You may also change the date of the signature to the current date if you want.

The file name of the file is shown in the dialogue box for your information only. You may not change the file name.

Remove (Del)

This option will remove the current signature from the database, without further prompting. The signature is totally lost. If you want it back then you will have to use the "Open and sign" option to add it.

Recalculate signature (Ctrl+R)

This option will use the file name of the current signature to locate it and recalculate its digital signature. The current signature will have its signature updated automatically after the recalculation.

You must not have deleted or moved the file since the original signature was calculated or an error will occur.

Verify (Ctrl+V)

This option will compare the signature stored in the database to the signature calculated from the file itself. If the two are the same then you can be assured that the file has not changed. If the two differ then the file as certainly changed.

A message box will appear, informing you of the results of the verification.

SHS

Selecting this menu option will cause a tick mark to be placed next to its menu entry. Files that you open and sign from now on will be signed with the SHS algorithm.

About the SHS

Federal Information Processing Standards Publication YY
DRAFT 1992 January 22 DRAFT
Specifications for a SECURE HASH STANDARD (SHS)

1. Introduction

The Secure Hash Algorithm (SHA) specified in this standard is necessary to ensure the security of the Digital Signature Standard. When a message of length < 2^64 bits is input, the SHA produces a 160-bit representation of the message called the message digest. The message digest is used during generation of a signature for the message. The same message digest should be computed for the received version of the message, during the process of verifying the 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 is designed to have the following properties: it is computationally infeasible to recover a message corresponding to a given message digest, or to find two different messages which produce the same message digest.

2. Bit Strings and Integers

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

(a) hex digit = 0, 1, ... , 9, a, ... , f. A hex digit is the representation of a 4-bit string. Examples: 7 = 0111, a = 1010.

(b). word = 32-bit string b(31) b(30) ... b(0). A word may be represented as a sequence of 8 hex digits. To convert the latter to a 32-bit string, each hex digit is converted to a 4-string as in (a). Example:

a103fe23 = 1010 0001 0000 0011 1111 1110 0010 0011.

A word is also the representation of an unsigned integer between 0 and 2^32 - 1, inclusive, with the most significant bit first. Example: the hex string a103fe23 represents the decimal integer 2701393443.

(c). block = 512-bit string. A block may be represented as a sequence of 16 words.

(d). integer: each integer x in the standard will satisfy 0 <= x < 2^64. For the purpose of this standard, "integer" and "unsigned integer" are equivalent. If an integer x satisfies 0 <= x < 2^32, x may be represented as a word as in (b). If 2^32 <= x < 2^64, then x = 2^32 y + z where 0 <= y < 2^32 and 0 <= z < 2^32. Hence y and z can be represented as words A and B, respectively, and x can be represented as the pair of words (A,B).

Suppose 0 <= x < 2^32. To convert x to a 32-bit string, the following algorithm may be employed:

```
y(0) = x;
for i = 0 to 31 do
{
   b(i) = 1 if y(i) is odd, b(i) = 0 if y(i) is even;
   y(i+1) = (y(i) - b(i))/2;
}
```

Then x has the 32-bit representation b(31) b(30) ... b(0). Example:

25 = 00000000 00000000 00000000 00011001 = hex 00000019.

If 2^32 <= x < 2^64, the 2-word representation of x is obtained similarly. Example:

2^35 + 25 = 8 * 2^32 + 25 =
00000000 00000000 00000000 00001000
00000000 00000000 00000000 00011001
= hex 00000008 00000019.

Conversely, the string b(31) b(30) ... b(0) represents the integer

b(31) * 2^31 + b(30) * 2^30 + ... + b(1) * 2 + b(0).

3. Operations on Words

The following logical operators will be applied to words:

AND = bitwise logical and.

OR = bitwise logical inclusive-or.

XOR = bitwise logical exclusive-or.

~x = bitwise logical complement of x.

Example:

01101100101110011101001001111011 XOR
01100101110000010110100110110111 =
00001001011110001011101111001100.

Another operation on words is A + B. This is defined as follows: words A and B represent integers x and y, where 0 <= x < 2^32 and 0 <= y < 2^32. Compute

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

Then 0 <= z < 2^32. Convert z to a word, C, and define A + B = C.

Another function on words is S(n,X), where X is a word and n is an integer with 0 <= n < 32. This is defined by

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

In the above, X << n is obtained as follows: discard the leftmost n bits of X and then pad the result with n zeroes on the right (the result will still be 32 bits). X >> m is obtained by discarding the rightmost m bits of X and then padding the result with m 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

The SHA takes bit strings as input. Thus, for the purpose of this standard, a message will be considered to be a bit string. The length of the message is the number of bits (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.

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

(a). "1" is appended. Example: if the original message is "01010011", this is padded to "010100111".

(b). If necessary, "0"s are then appended until the number of bits in the padded message is congruent to 448 modulo 512. 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.

The number of bits in the above is 41; we pad with 407 "0"s to make the length of the padded message congruent to 448 modulo 512. This gives (in hex)

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

Note that the padding is arranged so that at this point the padded message contains 16s + 14 words, for some s >= 0.

(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 final padded message will contain 16N words for some N > 0. Example: in (c) above, the final padded message has N = 1. The final padded message may be regarded as a sequence of N blocks M(1), M(2), ... , M(N), where each M(i) contains 16 words and M(1) is leftmost.

5. Functions Used

A sequence of logical functions f(0,x,y,z), ... , f(79,x,y,z) is used in the SHA. Each f operates on three 32-bit words {x,y,z} and produces a 32-bit word as output. f(t,x,y,z) is defined as follows: for words x,y,z,

(0 <= t <= 19) f(t,x,y,z) = (x AND y) OR (~x AND z)
(20 <= t <= 39) f(t,x,y,z) = x XOR y XOR z
(40 <= t <= 59) f(t,x,y,z) = (x AND y) OR (x AND z) OR (y AND z)
(60 <= t <= 79) f(t,x,y,z) = x XOR y XOR z

6. Constants Used

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

(0 <= t <= 19) K(t) = 5a827999

(20 <= t <= 39) K(t) = 6ed9eba1

(40 <= t <= 59) K(t) = 8f1bbcdc

(60 <= t <= 79) K(t) = ca62c1d6

7. Computing The Message Digest

The message digest is computed using the final padded message. The computation uses 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 {hj} 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 leftmost word.

(b). For t = 16 to 79 let W(t) = 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.

The 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:

(aa). Divide M(i) into 16 words W[0], ... , W[15], where W[0] is the leftmost word.

(bb). Let A = h0, B = h1, C = h2, D = h3, E = h4.

(cc). For t = 0 to 79 do
s = t AND MASK;
f (t >= 16) W[s] = 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;

(dd). Let h0 = h0 + A, h1 = h1 + B, h2 = h2 + C, h3 = h3 + D, h4 = h4 + E.

Both (a) - (d) and (aa) - (dd) yield the same message digest. Although using (aa) - (dd) 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 (cc). Other computation methods which give identical results may be implemented in conformance with the standard.

MD4

Selecting this menu option will cause a tick mark to be placed next to its menu entry. Files that you open and sign from now on will be signed with the MD4 algorithm.

About MD4

The MD4 Message Digest Algorithm
by Ronald L. Rivest

MIT Laboratory for Computer Science, Cambridge, Mass. 02139
and
RSA Data Security, Inc., Redwood City, California 94065
(Version 2/17/90 -- Revised)

Abstract

This note describes the MD4 message digest algorithm. The algorithm takes as input an input message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. The MD4 algorithm is thus ideal for digital signature applications, where a large file must be "compressed" in a secure manner before being signed with the RSA public-key cryptosystem.

The MD4 algorithm is designed to be quite fast on 32-bit machines. On a SUN Sparc station, MD4 runs at 1,450,000 bytes/second. On a DEC MicroVax II, MD4 runs at approximately 70,000 bytes/second. On a 20MHz 80286, MD4 runs at approximately 32,000 bytes/second. In addition, the MD4 algorithm does not require any large substitution tables; the algorithm can be coded quite compactly.

The MD4 algorithm is being placed in the public domain for review and possible adoption as a standard.

(Note: The document supersedes an earlier draft. The algorithm described here is a slight modification of the one described in the draft.)

I. Terminology and Notation

In this note a "word" is a 32-bit quantity and a byte is an 8-byte quantity. A sequence of bits can be interpreted in a natural manner as a sequence of bytes, where each consecutive group of 8 bits is interpreted as a byte with the high-order (most significant) bit of each byte listed first. Similarly, a

sequence of bytes can be interpreted as a sequence of 32-bit words, where each consecutive group of 4 bytes is interpreted as a word with the low-order (least significant) byte given first.

Let x_i denote "x sub i". If the subscript is an expression, we surround it in braces, as in x_{i+1}. Similarly, we use ^ for superscripts (exponentiation), so that x^i denotes x to the i-th power.

Let the symbol "+" denote addition of words (i.e., modulo-2^32 addition). Let X <<< s denote the 32-bit value obtained by circularly shifting (rotating) X left by s bit positions. Let not(X) denote the bit-wise complement of X, and let X v Y denote the bit-wise OR of X and Y. Let X xor Y denote the bit-wise XOR of X and Y, and let XY denote the bit-wise AND of X and Y.

II. MD4 Algorithm Description

We begin by supposing that we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of 8, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

m_0 m_1 ... m_{b-1} .

The following five steps are performed to compute the message digest of the message.

Step 1. Append padding bits

The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512 (in which case 512 bits of padding are added).

Padding is performed as follows: a single "1" bit is appended to the message, and then enough zero bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512.

Step 2. Append length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let M[0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

Step 3. Initialize MD buffer

A 4-word buffer (A,B,C,D) is used to compute the message digest. Here each of A,B,C,D are 32-bit registers. These registers are initialized to the following values (in hexadecimal, low-order bytes first):

word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10

Step 4. Process message in 16-word blocks

We first define three auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

f(X,Y,Z) = XY v not(X)Z
g(X,Y,Z) = XY v XZ v YZ
h(X,Y,Z) = X xor Y xor Z

In each bit position f acts as a conditional: if x then y else z. (The function f could have been defined using + instead of v since XY and not(X)Z will never have 1's in the same bit position.) In each bit position g acts as a majority function: if at least two of x, y, z are on, then g has a one in that bit position, else g has a zero. It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of f(X,Y,Z) will be independent and unbiased, and similarly each bit of g(X,Y,Z) will be independent and unbiased. The function h is the bit-wise "xor" or "parity" function; it has properties similar to those of f and g.

Do the following:

```
For i = 0 to N/16-1 do          /* process each 16-word block */
    For j = 0 to 15 do:         /* copy block i into X */
        Set X[j] to M[i*16+j].
    end                         /* of loop on j */
    Save A as AA, B as BB, C as CC, and D as DD.

    [Round 1]
        Let [A B C D i s] denote the operation
        A = (A + f(B,C,D) + X[i]) <<< s    .
        Do the following 16 operations:
        [A B C D 0 3]
        [D A B C 1 7]
        [C D A B 2 11]
        [B C D A 3 19]
        [A B C D 4 3]
        [D A B C 5 7]
        [C D A B 6 11]
        [B C D A 7 19]
        [A B C D 8 3]
        [D A B C 9 7]
        [C D A B 10 11]
        [B C D A 11 19]
        [A B C D 12 3]
        [D A B C 13 7]
        [C D A B 14 11]
        [B C D A 15 19]

    [Round 2]
        Let [A B C D i s] denote the operation
                A = (A + g(B,C,D) + X[i] + 5A827999) <<< s .
```

(The value 5A..99 is a hexadecimal 32-bit constant, written with the high-order digit first. This constant represents the square root of 2. The octal value of this constant is 013240474631.
See Knuth, The Art of Programming, Volume 2 (Seminumerical Algorithms), Second Edition (1981), Addison-Wesley. Table 2, page 660.)

```
        Do the following 16 operations:
        [A B C D 0   3]
        [D A B C 4   5]
        [C D A B 8   9]
        [B C D A 12 13]
```

[A B C D 1 3]
[D A B C 5 5]
[C D A B 9 9]
[B C D A 13 13]
[A B C D 2 3]
[D A B C 6 5]
[C D A B 10 9]
[B C D A 14 13]
[A B C D 3 3]
[D A B C 7 5]
[C D A B 11 9]
[B C D A 15 13]

[Round 3]

Let [A B C D i s] denote the operation
A = (A + h(B,C,D) + X[i] + 6ED9EBA1) <<< s

(The value 6E..A1 is a hexadecimal 32-bit constant, written with the high-order digit first. This constant represents the square root of 3. The octal value of this constant is 015666365641.
See Knuth, The Art of Programming, Volume 2 (Seminumerical Algorithms), Second Edition (1981), Addison-Wesley. Table 2, page 660.)

Do the following 16 operations:
[A B C D 0 3]
[D A B C 8 9]
[C D A B 4 11]
[B C D A 12 15]
[A B C D 2 3]
[D A B C 10 9]
[C D A B 6 11]
[B C D A 14 15]
[A B C D 1 3]
[D A B C 9 9]
[C D A B 5 11]
[B C D A 13 15]
[A B C D 3 3]
[D A B C 11 9]
[C D A B 7 11]
[B C D A 15 15]

Then perform the following additions:

A = A + AA
B = B + BB
C = C + CC
D = D + DD

(That is, each of the four registers is incremented by the value it had
before this block was started.)

end /* of loop on i */

Step 5. Output

The message digest produced as output is A,B,C,D. That is, we begin with the low-order byte of A, and end with the high-order byte of D. This completes the description of MD4.

III. Extensions

If more than 128 bits of output are required, then the following procedure is recommended to obtain a 256-bit output. (There is no provision made for obtaining more than 256 bits.)

Two copies of MD4 are run in parallel over the input. The first copy is standard as described above. The second copy is modified as follows.

The initial state of the second copy is:

word A: 00 11 22 33
word B: 44 55 66 77
word C: 88 99 aa bb
word D: cc dd ee ff

The magic constants in rounds 2 and 3 for the second copy of MD4 are changed from sqrt(2) and sqrt(3) to cuberoot(2) and cuberoot(3):

	Octal	Hex
Round 2 constant	012050505746	50a28be6
Round 3 constant	013423350444	5c4dd124

Finally, after every 16-word block is processed (including the last block), the values of the A registers in the two copies are exchanged.

The final message digest is obtaining by appending the result of the second copy of MD4 to the end of the result of the first copy of MD4.

IV. Summary

The MD4 message digest algorithm is simple to implement, and provides a "fingerprint" or message digest of a message of arbitrary length.

It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^64 operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^128 operations. The MD4 algorithm has been carefully scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort. The level of security provided by MD4 should be sufficient for implementing very high security hybrid digital signature schemes based on MD4 and the RSA public-key cryptosystem.

V. Acknowledgements

I'd like to thank Don Coppersmith, Burt Kaliski, Ralph Merkle, and Noam Nisan for numerous helpful comments and suggestions.

MD5

Selecting this menu option will cause a tick mark to be placed next to its menu entry. Files that you open and sign from now on will be signed with the MD5 algorithm.

About MD5

1. Executive Summary

This document describes the MD5 message-digest algorithm. The algorithm takes as input an input message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem such as RSA.

The MD5 algorithm is designed to be quite fast on 32-bit machines. In addition, the MD5 algorithm does not require any large substitution tables; the algorithm can be coded quite compactly.

The MD5 algorithm is an extension of the MD4 message digest algorithm [1,2]. MD5 is slightly slower than MD4, but is more "conservative" in design. MD5 was designed because it was felt that MD4 was perhaps being adopted for use more quickly than justified by the existing critical review; because MD4 was designed to be exceptionally fast, it is "at the edge" in terms of risking successful cryptanalytic attack. MD5 backs off a bit, giving up a little in speed for a much greater likelihood of ultimate security. It incorporates some suggestions made by various reviewers, and contains additional optimizations.

The MD5 algorithm is being placed in the public domain for review and possible adoption as a standard.

A version of this document including the C source code in the appendix is available by FTP from RSA.COM in the file "pub/md5.doc".

This document may be referred to, unofficially, as Internet draft [MD5-A].

For OSI-based applications, MD5's object identifier is md5 OBJECT IDENTIFIER ::=

{iso(1) member-body(2) US(840) rsadsi(113549) digestAlgorithm(2) 5}

In the X.509 type AlgorithmIdentifier [3], the parameters for MD5 should have type NULL.

2. Terminology and Notation

In this document a "word" is a 32-bit quantity and a "byte" is an eight-bit quantity. A sequence of bits can be interpreted in a natural manner as a sequence of bytes, where each consecutive group of eight bits is interpreted as a byte with the high-order (most significant) bit of each byte listed first. Similarly, a sequence of bytes can be interpreted as a sequence of 32-bit words, where each consecutive group of four bytes is interpreted as a word with the low-order (least significant) byte given first.

Let x_i denote "x sub i". If the subscript is an expression, we surround it in braces, as in x_{i+1}. Similarly, we use ^ for superscripts (exponentiation), so that x^i denotes x to the i-th power.

Let the symbol "+" denote addition of words (i.e., modulo-2^32 addition). Let X <<< s denote the 32-bit value obtained by circularly shifting (rotating) X left by s bit positions. Let not(X) denote the bit-wise complement of X, and let X v Y denote the bit-wise OR of X and Y. Let X xor Y denote the bit-wise XOR of

X and Y, and let XY denote the bit-wise AND of X and Y.

3. MD5 Algorithm Description

We begin by supposing that we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of eight, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

m_0 m_1 ... m_{b-1}

The following five steps are performed to compute the message digest of the message.

3.1 Step 1. Append Padding Bits

The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512 (in which case 512 bits of padding are added).

Padding is performed as follows: a single "1" bit is appended to the message, and then enough zero bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512.

3.2 Step 2. Append Length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let M[0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

3.3 Step 3. Initialize MD Buffer

A four-word buffer (A,B,C,D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal, low-order bytes first):

word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10

3.4 Step 4. Process Message in 16-Word Blocks

We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

F(X,Y,Z) = XY v not(X) Z
G(X,Y,Z) = XZ v Y not(Z)
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X v not(Z))

In each bit position F acts as a conditional: if X then Y else Z. (The function F could have been defined using + instead of v since XY and not(X)Z will never have 1's in the same bit position.) It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of F(X,Y,Z) will be

independent and unbiased.

The functions G, H, and I are similar to the function F, in that they act in "bitwise parallel" to produce their output from the bits of X, Y, and Z, in such a manner that if the corresponding bits of X, Y, and Z are independent and unbiased, then each bit of G(X,Y,Z), H(X,Y,Z), and I(X,Y,Z) will be independent and unbiased. Note that the function H is the bit-wise "xor" or "parity" function of its inputs.

Do the following:

```
/* Process each 16-word block. */

        For i = 0 to N/16-1 do

/* Copy block i into X. */

        For j = 0 to 15 do
          Set X[j] to M[i*16+j].
        end                             /* of loop on j */

/* Save A as AA, B as BB, C as CC, and D as DD. */

        AA = A
        BB = B
        CC = C
        DD = D

/* Round 1. */
/* Let FF(a,b,c,d,X[k],s,t) denote the operation
    a = b + ((a + F(b,c,d) + X[k] + t) <<< s). */

/* Here the additive constants t are chosen as follows: In step i, the additive constant is the integer part of
4294967296 times abs(sin(i)), where i is in radians. */

/* Let S11 = 7, S12 = 12, S13 = 17, and S14 = 22. */
/* Do the following 16 operations. */

        FF (a, b, c, d, X[ 0], S11, 3614090360); /* Step 1 */
        FF (d, a, b, c, X[ 1], S12, 3905402710); /* 2 */
        FF (c, d, a, b, X[ 2], S13,   606105819); /* 3 */
        FF (b, c, d, a, X[ 3], S14, 3250441966); /* 4 */
        FF (a, b, c, d, X[ 4], S11, 4118548399); /* 5 */
        FF (d, a, b, c, X[ 5], S12, 1200080426); /* 6 */
        FF (c, d, a, b, X[ 6], S13, 2821735955); /* 7 */
        FF (b, c, d, a, X[ 7], S14, 4249261313); /* 8 */
        FF (a, b, c, d, X[ 8], S11, 1770035416); /* 9 */
        FF (d, a, b, c, X[ 9], S12, 2336552879); /* 10 */
        FF (c, d, a, b, X[10], S13, 4294925233); /* 11 */
        FF (b, c, d, a, X[11], S14, 2304563134); /* 12 */
        FF (a, b, c, d, X[12], S11, 1804603682); /* 13 */
        FF (d, a, b, c, X[13], S12, 4254626195); /* 14 */
        FF (c, d, a, b, X[14], S13, 2792965006); /* 15 */
        FF (b, c, d, a, X[15], S14, 1236535329); /* 16 */

/* Round 2. */
/* Let GG(a,b,c,d,X[k],s,t) denote the operation
    a = b + ((a + G(b,c,d) + X[k] + t) <<< s). */
```

```
/* Let S21 = 5, S22 = 9, S23 = 14, and S24 = 20. */
/* Do the following 16 operations. */

        GG (a, b, c, d, X[ 1], S21, 4129170786); /* 17 */
        GG (d, a, b, c, X[ 6], S22, 3225465664); /* 18 */
        GG (c, d, a, b, X[11], S23,  643717713); /* 19 */
        GG (b, c, d, a, X[ 0], S24, 3921069994); /* 20 */
        GG (a, b, c, d, X[ 5], S21, 3593408605); /* 21 */
        GG (d, a, b, c, X[10], S22,   38016083); /* 22 */
        GG (c, d, a, b, X[15], S23, 3634488961); /* 23 */
        GG (b, c, d, a, X[ 4], S24, 3889429448); /* 24 */
        GG (a, b, c, d, X[ 9], S21,  568446438); /* 25 */
        GG (d, a, b, c, X[14], S22, 3275163606); /* 26 */
        GG (c, d, a, b, X[ 3], S23, 4107603335); /* 27 */
        GG (b, c, d, a, X[ 8], S24, 1163531501); /* 28 */
        GG (a, b, c, d, X[13], S21, 2850285829); /* 29 */
        GG (d, a, b, c, X[ 2], S22, 4243563512); /* 30 */
        GG (c, d, a, b, X[ 7], S23, 1735328473); /* 31 */
        GG (b, c, d, a, X[12], S24, 2368359562); /* 32 */

/* Round 3. */
/* Let HH(a,b,c,d,X[k],s,t) denote the operation
    a = b + ((a + H(b,c,d) + X[k] + t) <<< s). */
/* Let S31 = 4, S32 = 11, S33 = 16, and S34 = 23. */
/* Do the following 16 operations. */

        HH (a, b, c, d, X[ 5], S31, 4294588738); /* 33 */
        HH (d, a, b, c, X[ 8], S32, 2272392833); /* 34 */
        HH (c, d, a, b, X[11], S33, 1839030562); /* 35 */
        HH (b, c, d, a, X[14], S34, 4259657740); /* 36 */
        HH (a, b, c, d, X[ 1], S31, 2763975236); /* 37 */
        HH (d, a, b, c, X[ 4], S32, 1272893353); /* 38 */
        HH (c, d, a, b, X[ 7], S33, 4139469664); /* 39 */
        HH (b, c, d, a, X[10], S34, 3200236656); /* 40 */
        HH (a, b, c, d, X[13], S31,  681279174); /* 41 */
        HH (d, a, b, c, X[ 0], S32, 3936430074); /* 42 */
        HH (c, d, a, b, X[ 3], S33, 3572445317); /* 43 */
        HH (b, c, d, a, X[ 6], S34,   76029189); /* 44 */
        HH (a, b, c, d, X[ 9], S31, 3654602809); /* 45 */
        HH (d, a, b, c, X[12], S32, 3873151461); /* 46 */
        HH (c, d, a, b, X[15], S33,  530742520); /* 47 */
        HH (b, c, d, a, X[ 2], S34, 3299628645); /* 48 */

/* Round 4. */
/* Let II(a,b,c,d,X[k],s,t) denote the operation
    a = b + ((a + I(b,c,d) + X[k] + t) <<< s). */

/* Let S41 = 6, S42 = 10, S43 = 15, and S44 = 21. */
/* Do the following 16 operations. */

        II (a, b, c, d, X[ 0], S41, 4096336452); /* 49 */
        II (d, a, b, c, X[ 7], S42, 1126891415); /* 50 */
        II (c, d, a, b, X[14], S43, 2878612391); /* 51 */
        II (b, c, d, a, X[ 5], S44, 4237533241); /* 52 */
        II (a, b, c, d, X[12], S41, 1700485571); /* 53 */
```

```
II (d, a, b, c, X[ 3], S42, 2399980690); /* 54 */
II (c, d, a, b, X[10], S43, 4293915773); /* 55 */
II (b, c, d, a, X[ 1], S44, 2240044497); /* 56 */
II (a, b, c, d, X[ 8], S41, 1873313359); /* 57 */
II (d, a, b, c, X[15], S42, 4264355552); /* 58 */
II (c, d, a, b, X[ 6], S43, 2734768916); /* 59 */
II (b, c, d, a, X[13], S44, 1309151649); /* 60 */
II (a, b, c, d, X[ 4], S41, 4149444226); /* 61 */
II (d, a, b, c, X[11], S42, 3174756917); /* 62 */
II (c, d, a, b, X[ 2], S43,  718787259); /* 63 */
II (b, c, d, a, X[ 9], S44, 3951481745); /* 64 */
```

/* Then perform the following additions. (That is, increment each of the four registers by the value it had before this block was started.) */

```
A = A + AA
B = B + BB
C = C + CC
D = D + DD

end                 /* of loop on i */
```

3.5 Step 5. Output

The message digest produced as output is A, B, C, D. That is, we begin with the low-order byte of A, and end with the high-order byte of D.

This completes the description of MD5.

4. Summary

The MD5 message-digest algorithm is simple to implement, and provides a "fingerprint" or message digest of a message of arbitrary length. It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^64 operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^128 operations. The MD5 algorithm has been carefully scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort.

5. Summary of Differences Between MD4 and MD5

The following are the differences between MD4 and MD5:

A fourth round has been added.

Each step now has a unique additive constant.

The function g in round 2 was changed from (XY v XZ v YZ) to (XZ v Y not(Z)) to make g less symmetric.

Each step now adds in the result of the previous step. This promotes a faster "avalanche effect".

The order in which input words are accessed in rounds 2 and 3 is changed, to make these patterns less like each other.

The shift amounts in each round have been approximately optimized, to yield a faster "avalanche effect". The shifts in different rounds are distinct.

6. Security Considerations

The level of security discussed in this memo is considered to be sufficient for implementing very high security hybrid digital-signature schemes based on MD5 and a public-key cryptosystem.

References

[1] Rivest, R.L., The MD4 Message Digest Algorithm (RFC 1186), October 1990.

[2] Rivest, R.L., The MD4 message digest algorithm, presented at CRYPTO '90 (Santa Barbara, CA, August 11-15, 1990).

[3] CCITT, The Directory---Authentication Framework (Recommendation X.509), 1988.

Authors' Addresses

Ronald L. Rivest
Massachusetts Institute of Technology
Laboratory for Computer Science
NE43-324
545 Technology Square
Cambridge, MA 02139-1986
Phone: (617) 253-5880
EMail: rivest@theory.lcs.mit.edu

Steve Dusse
RSA Data Security, Inc.
10 Twin Dolphin Drive
Redwood City, CA 94065
Phone: (415) 595-8782
EMail: dusse@rsa.com

What is DigSig ?

DigSig is a windows program that calculates and stores the digital signatures of files. So what is a digital signature, and why is it useful to you ?

A *digital signature* is a very large number based on the contents of a file. No two files will generate the same signature, and it is this property that is used to great advantage in DigSig. There are a number of methods that can be used to generate digital signatures, with differing levels of security and speed of execution. DigSig currently implements three of the most popular, SHS, MD4 and MD5.

For instance, you may wish to securely send a file to someone. If you both have a copy of DigSig then the recipient of the file can generate the digital signature and call you to see if it is correct. Anyone intercepting the file would have to alter the file such that it still produces the same signature, and that is not possible.

Another use could be to electronically "sign" letters that you send out. The printed signature uniquely identifies the correspondence far better than your hand-written signature ever could. The recipient could call you with a question such as "Did you send me a letter with signature ..." and you could say "Yes" or "No" with complete certainty.

These are just two uses that I could think of, I'm sure you can think of more !

Shareware information

DigSig is shareware. This means that if you like it, and continue to use it then you should send the registration fee to me, the author. In return I will send you a disk containing the latest version of the software and the complete "C" source code. You will also get a printed manual.

The registration fee is £10 (ten pounds sterling) and should be sent to me at the following address:

Andrew Brown
28 Ashburn Drive
Wetherby
West Yorkshire
LS22 5RD
ENGLAND (UK)

I accept cash and cheques drawn on a UK bank. I also accept International Postal/Money Orders made out in sterling.

Registered users are entitled to unlimited free postal support direct to me, the author of DigSig. Suggestions for further programs or improvements and additions to this one are also most welcome.

Changes log

version 1.10, September 1993

MD4 and MD5 methods added

Complete explanation of each method included in the help text