As it is easy to generate MD5 collisions, it is possible for the person who created the file to create a second file with the same checksum, so this technique cannot protect against some forms of malicious tampering. In some cases, the checksum cannot be trusted (for example, if it was obtained over the same channel as the downloaded file), in which case MD5 can only provide error-checking functionality: it will recognize a corrupt or incomplete download, which becomes more likely when downloading larger files.

Historically, MD5 has been used to store a one-way hash of a password, often with key stretching.[47][48] NIST does not include MD5 in their list of recommended hashes for password storage.[49]

MD5 is also used in the field of electronic discovery, to provide a unique identifier for each document that is exchanged during the legal discovery process. This method can be used to replace the Bates stamp numbering system that has been used for decades during the exchange of paper documents. As above, this usage should be discouraged due to the ease of collision attacks.

Algorithm

MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is padded so that its length is divisible by 512. The padding works as follows: first, a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^{64}.

The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted *A*, *B*, *C*, and *D*. These are initialized to certain fixed constants. The main algorithm then uses each 512-bit message block in turn to modify the state. The processing of a message block consists of four similar stages, termed *rounds*; each round is composed of 16 similar operations based on a non-linear function *F*, modular addition, and left rotation. Figure 1 illustrates one operation within a round. There are four possible functions; a different one is used in each round:

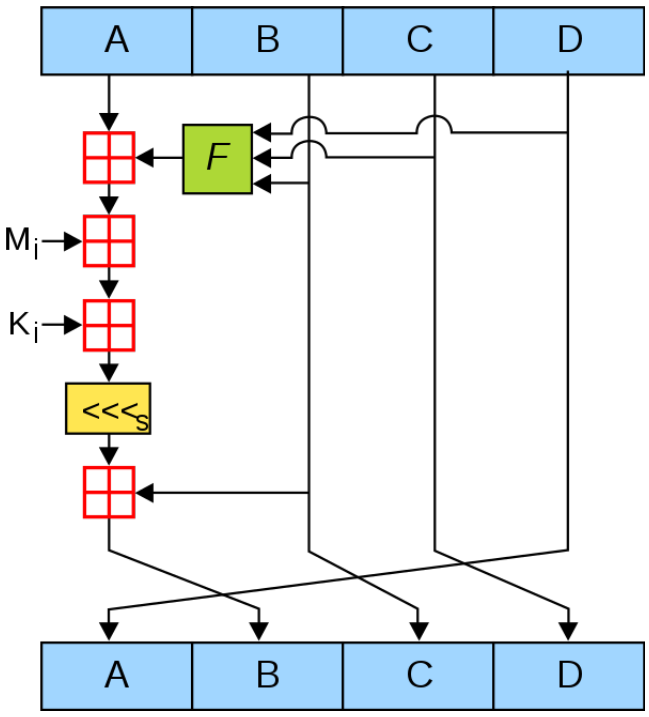

Figure 1. One MD5 operation. MD5 consists of 64 of these operations, grouped in four rounds of 16 operations. F is a nonlinear function; one function is used in each round. M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation. $<<<_s$ denotes a left bit rotation by s places; s varies for each operation. $\boxplus$ denotes addition modulo 2^{32}.

$$F(B,C,D) = (B \wedge C) \vee (\neg B \wedge D)$$
$$G(B,C,D) = (B \wedge D) \vee (C \wedge \neg D)$$
$$H(B,C,D) = B \oplus C \oplus D$$
$$I(B,C,D) = C \oplus (B \vee \neg D)$$

$\oplus, \wedge, \vee, \neg$ denote the XOR, AND, OR and NOT operations respectively.

Pseudocode

The MD5 hash is calculated according to this algorithm.[50] All values are in little-endian.

```
// : All variables are unsigned 32 bit and wrap modulo 2^32 when calculating
var int s[64], K[64]
var int i

// s specifies the per-round shift amounts
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22 }
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20 }
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21 }

// Use binary integer part of the sines of integers (Radians) as constants:
for i from 0 to 63 do
    K[i] := floor(2^32 × abs (sin(i + 1)))
end for
// (Or just use the following precomputed table):
K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
K[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
K[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
K[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
K[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
K[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
K[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
K[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
K[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
K[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }

// Initialize variables:
var int a0 := 0x67452301   // A
var int b0 := 0xefcdab89   // B
var int c0 := 0x98badcfe   // C
var int d0 := 0x10325476   // D

// Pre-processing: adding a single 1 bit
append "1" bit to message
 // Notice: the input bytes are considered as bits strings,
 //  where the first bit is the most significant bit of the byte.[51]

// Pre-processing: padding with zeros
append "0" bit until message length in bits ≡ 448 (mod 512)

// Notice: the two padding steps above are implemented in a simpler way
//  in implementations that only work with complete bytes: append 0x80
//  and pad with 0x00 bytes so that the message length in bytes ≡ 56 (mod 64).
```

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

// Process the message in successive 512-bit chunks:
for each 512-bit chunk of padded message do
    break chunk into sixteen 32-bit words M[j], 0 ≤ j ≤ 15
    // Initialize hash value for this chunk:
    var int A := a0
    var int B := b0
    var int C := c0
    var int D := d0
    // Main loop:
    for i from 0 to 63 do
        var int F, g
        if 0 ≤ i ≤ 15 then
            F := (B and C) or ((not B) and D)
            g := i
        else if 16 ≤ i ≤ 31 then
            F := (D and B) or ((not D) and C)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
            F := B xor C xor D
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
            F := C xor (B or (not D))
            g := (7×i) mod 16
        // Be wary of the below definitions of a,b,c,d
        F := F + A + K[i] + M[g]  // M[g] must be a 32-bits block
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
    end for
    // Add this chunk's hash to result so far:
    a0 := a0 + A
    b0 := b0 + B
    c0 := c0 + C
    d0 := d0 + D
end for

var char digest[16] := a0 append b0 append c0 append d0 // (Output is in little-endian)
```

Instead of the formulation from the original RFC 1321 shown, the following may be used for improved efficiency (useful if assembly language is being used – otherwise, the compiler will generally optimize the above code. Since each computation is dependent on another in these formulations, this is often slower than the above method where the nand/and can be parallelised):

```
( 0 ≤ i ≤ 15): F := D xor (B and (C xor D))
(16 ≤ i ≤ 31): F := C xor (D and (B xor C))
```

MD5 hashes

The 128-bit (16-byte) MD5 hashes (also termed *message digests*) are typically represented as a sequence of 32 hexadecimal digits. The following demonstrates a 43-byte ASCII input and the corresponding MD5 hash:

```
MD5("The quick brown fox jumps over the lazy dog") =
9e107d9d372bb6826bd81d3542a419d6
```

Even a small change in the message will (with overwhelming probability) result in a mostly different hash, due to the avalanche effect. For example, adding a period to the end of the sentence:

```
MD5("The quick brown fox jumps over the lazy dog.") =
e4d909c290d0fb1ca068ffaddf22cbd0
```

The hash of the zero-length string is:

```
MD5("") =
d41d8cd98f00b204e9800998ecf8427e
```

The MD5 algorithm is specified for messages consisting of any number of bits; it is not limited to multiples of eight bits (octets, bytes). Some MD5 implementations such as md5sum might be limited to octets, or they might not support *streaming* for messages of an initially undetermined length.

On Following Page: Wikipedia's Discussion of Cryptographic Hash Properties

A cryptographic hash function must be able to withstand all known types of cryptanalytic attack. In theoretical cryptography, the security level of a cryptographic hash function has been defined using the following properties:

Pre-image resistance
: Given a hash value h, it should be difficult to find any message m such that $h = \mathrm{hash}(m)$. This concept is related to that of a one-way function. Functions that lack this property are vulnerable to preimage attacks.

Second pre-image resistance
: Given an input m_1, it should be difficult to find a different input m_2 such that $\mathrm{hash}(m_1) = \mathrm{hash}(m_2)$. This property is sometimes referred to as *weak collision resistance*. Functions that lack this property are vulnerable to second-preimage attacks.

Collision resistance
: It should be difficult to find two different messages m_1 and m_2 such that $\mathrm{hash}(m_1) = \mathrm{hash}(m_2)$. Such a pair is called a cryptographic hash collision. This property is sometimes referred to as *strong collision resistance*. It requires a hash value at least twice as long as that required for pre-image resistance; otherwise collisions may be found by a birthday attack.[4]

Collision resistance implies second pre-image resistance but does not imply pre-image resistance.[5] The weaker assumption is always preferred in theoretical cryptography, but in practice, a hash-function which is only second pre-image resistant is considered insecure and is therefore not recommended for real applications.

Informally, these properties mean that a malicious adversary cannot replace or modify the input data without changing its digest. Thus, if two strings have the same digest, one can be very confident that they are identical. Second pre-image resistance prevents an attacker from crafting a document with the same hash as a document the attacker cannot control. Collision resistance prevents an attacker from creating two distinct documents with the same hash.

A function meeting these criteria may still have undesirable properties. Currently, popular cryptographic hash functions are vulnerable to *length-extension* attacks: given $\mathrm{hash}(m)$ and $\mathrm{len}(m)$ but not m, by choosing a suitable m' an attacker can calculate $\mathrm{hash}(m \parallel m')$, where $\parallel$ denotes concatenation.[6] This property can be used to break naive authentication schemes based on hash functions. The HMAC construction works around these problems.

In practice, collision resistance is insufficient for many practical uses. In addition to collision resistance, it should be impossible for an adversary to find two messages with substantially similar digests; or to infer any useful information about the data, given only its digest. In particular, a hash function should behave as much as possible like a random function (often called a random oracle in proofs of security) while still being deterministic and efficiently computable. This rules out functions like the SWIFFT function, which can be rigorously proven to be collision-resistant assuming that certain problems on ideal lattices are computationally difficult, but, as a linear function, does not satisfy these additional properties.[7]