The Compression Functions of SHA, MD2, MD4 and MD5 are not Affine*

Mahesh V. Tripunitara and Samuel S. Wagstaff, Jr.
COAST Laboratory
Purdue University
West Lafayette, IN 47907-1398
{*tripunit,ssw*}*@cs.purdue.edu*

COAST TR-98/01

Abstract

In this paper, we show that the compressions functions of SHA, MD2, MD4 and MD5 are not affine transformations of their inputs.

1 Introduction

The Secure Hash Algorithm (SHA) [1, 5, 2] and the Message Digest algorithms MD2 [4], MD4 [6] and MD5 [7], are used to produce a *message digest*, a condensed form of a message for use in digital signature and other applications. Figure 1 shows how these algorithms are used. Each step in each of the algorithms is a *compression function* (denoted by CF in the figure) which takes as inputs a (portion of a) message and the current value for a *chaining variable*. Each compression function alters the value for the chaining variable based on the input message, and the output from each compression function is this altered value for the chaining variable. The value for the chaining variable from the last application of the compression function is the output message digest. The initial value for the chaining variable for the first application of the compression function is fixed for each algorithm and each subsequent application uses the output from the previous application as input value for the chaining variable.

An important property for each such compression function to possess is for the output to not be an affine transformation of the inputs, which are a fixed size (portion of a) message and the initial value for the chaining variable. If the output from a compression function is an affine transformation of its inputs, it would be trivial to construct messages that produce a given message digest when subjected to the message digest algorithm as we show in section 1.2. Such an algorithm would be useless in a security context.

In this paper we show that the compressions functions of the message digest algorithms SHA, MD2, MD4 and MD5 as described in [1], [4], [6] and [7] respectively are indeed not affine.

1.1 Some Characteristics of the Algorithms

Each algorithm takes an arbitrary length (we use "length", "size" and "number of bits" to mean the same unless specifically noted otherwise) message as input and produces a fixed length message digest as output. The message digest is 160 bits for SHA and 128 bits for each of MD2, MD4 and MD5.

Each algorithm pads the input message such as to make the number of bits in the message plus the pad a multiple of 512 in the case of SHA, MD4 and MD5, and a multiple of 128 in the case of MD2.

*Portions of this work were supported by sponsors of the COAST Laboratory.

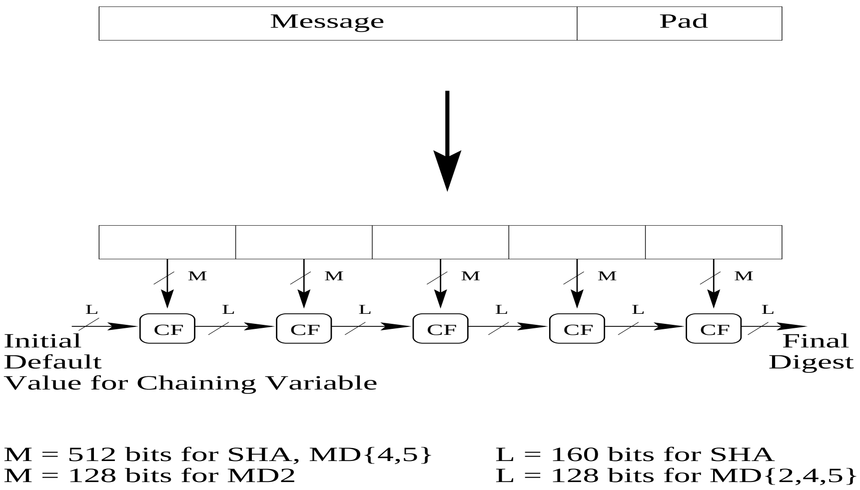

Figure 1: Use of Message Digest Algorithms

For SHA, MD4 and MD5, the padding consists of a 1 (bit) followed by enough 0's to make the number of bits of the message plus the pad (so far) 448 mod 512. At least 1 bit and at most 512 bits are thus added as pad. The length of the (original) message represented as a 64 bit number is then appended to make the entire length a multiple of 512. In the case of MD2, padding of sufficient length to make the number of bits of the padded message a multiple of 128 is added. Thus, if i bytes are added as pad, each of those bytes has the value i. At least 1 byte and at most 16 bytes are appended as pad. A 16 byte (128 bit) checksum of the message plus pad is also then appended.

After padding, SHA, MD4 and MD5 break the padded message into 512 bit chunks and use those as inputs to the compression function, in turn, as indicated in figure 1. MD2 also does the same, except that its compression function takes 128 bit chunks of the message as input. Each compression function also takes as input the value of the chaining variable which is the output from the previous application of the compression function. For the first application of the compression function, each algorithm uses a default initial value for the chaining variable.

1.2 An Attack on a Message Digest Algorithm That Uses an Affine Compression Function

In this section, we assume the existence of a message digest algorithm, A that works like SHA, MD2, MD4 and MD5 in that it takes as input a message of some length, pads the message and breaks the padded message into equal sized chunks to use as input to a compression function, CF. CF also takes as input a chaining variable that is the output from the previous application of CF (a default value for the first application). Also, we assume that CF is affine in its inputs, that is, CF can be represented as a pre-multiplication by a matrix of appropriate dimensions bitwise XOR-ed with a vector of appropriate size. Then, given a message and its corresponding message digest, we construct a second message with the same message digest. We assume that the reader is familiar with basic concepts of numerical linear algebra such as *rank of a matrix* and *Gaussian Elimination.* We refer the reader to [3] for a review of these and other related topics. Also, we adopt the following notation:

$F_2^{n,m}$: The set of $n \times m$ matrices each of whose elements are in F_2, the Galois Field with the two elements 0 and 1. m is omitted if it is 1, and n and m are omitted if they are both 1.

$\oplus$: The (bitwise) exclusive-or operator.

$|\cdot|$: A function that takes as argument a bit string or bit vector and returns its length.

$\cdot$: Used to concatenate bit strings.

A: The message digest algorithm with an affine compression function.

M, M', M'': Inputs to A (messages of some length).

D: Output from A (message digest).

CF: The compression function used in A.

m, m_i: Input message to CF, typically a portion of M, M' or M'', written as a bit vector.

v, v_i: Input chaining variable to CF written as a bit vector.

w, w_i: Output from CF written as a bit vector. The output from the final application of CF is the message digest.

L_m: The length of the input message to CF. Thus, typically, $L_m = |m_i|$ and $L_m = 512$ for SHA, MD4 and MD5 and $L_m = 128$ for MD2.

L_v: The length of the input chaining variable to and thus, the output from, CF. Thus, $L_v = |v_i| = |w_j|$ for any i, j. $L_v = 160$ for SHA and $L_v = 128$ for MD2, MD4 and MD5.

p: The padding added to the message. For SHA, MD4 and MD5, $65 \leq |p| \leq 576$ and for MD2, $136 \leq |p| \leq 256$.

C: The matrix corresponding to CF. Thus, $C \in F_2^{L_v,(L_v+L_m)}$.

C_i: The i^{th} column of C, where $1 \leq i \leq (L_v + L_m)$. Thus, $C_i \in F_2^{L_v}$ and $C = \left[\begin{array}{cccc} C_1 & C_2 & \ldots & C_{(L_v+L_m)} \end{array}\right]$.

b: The constant vector corresponding to the affine transformation, CF. Thus, b is the output on input $\mathbf{0}$ (the vector of all 0's), and $b \in F_2^{L_v}$.

Thus, CF, on inputs v, m and output w can now be expressed as:

$$w = \left[\begin{array}{cccc} C_1 & C_2 & \ldots & C_{(L_v+L_m)} \end{array}\right] \left[\begin{array}{c} v \\ m \end{array}\right] \oplus b \tag{1}$$

Assume that some message M on input to A produces D as message digest. We would like to substitute our favorite message, M' for the original message, but have the new message also produce the same output D on input to A. Of course, this is not possible in general. Therefore, starting from our favorite message M', we construct M'', a slightly transformed version of M', such that M'' on input to A, produces D as message digest. Starting with M', we carry out the attack as follows.

1. We split M' into chunks of size L_m. Thus, we produce $m_1, m_2, \ldots, m_k$ such that $|m_i| = L_m$ for $i = 1, 2, \ldots, k-1$, $|m_k| \leq L_m$ and $m_1 \cdot m_2 \cdot \ldots \cdot m_k = M'$.

2. We subject each of $m_1, m_2, \ldots, m_{k-1}$ to CF, in turn. At the first application, we use the default value for the input chaining variable and at each subsequent application, we use the output from the previous application of CF. Application of CF is carried out as indicated in (1). The output from the last such application of CF is $w_{k-1} = v_k$, which is the same as the input chaining variable to the next application of CF. Thus, we have now performed $k-1$ applications of CF.

3. If A works like SHA, MD4 or MD5, the message is first padded with at least one bit so that it is of length 448 mod L_m, and then with the length of the message represented as a 64 bit number (of course, $L_m = 512$ for SHA, MD4 and MD5). Our strategy is to append a suitable "noise" to the message. We now need to consider one of several cases as described below.

   (a) If the next (k^{th}) application of CF is to be the last, we require the output from the application to be D. Thus, we need to find a "noise" n such that:

$$\left[\begin{array}{cccc} C_1 & C_2 & \ldots & C_{(L_v+L_m)} \end{array}\right] \left[\begin{array}{c} v_k \\ m_k \\ n \\ p \end{array}\right] = D \oplus b \tag{2}$$

To be able to find such an n, we need the following condition to be satisfied:

$$rank\left(\left[\begin{array}{cccc} C_{(L_v+|m_k|+1)} & C_{(L_v+|m_k|+2)} & \ldots & C_{(L_v+L_m-|p|)} \end{array}\right]\right) = rank(C) \tag{3}$$

We may assume that p is of minimum length, that is $|p| = 65$. If (3) is true, we are left with a consistent system of linear equations in the components (bits) of n, for any D. That is, if $n = \left[\begin{array}{cccc} n_1 & n_2 & \ldots & n_t \end{array}\right]^T$, where $t = (L_m - |m_k| - |p|)$ and each $n_i \in F_2$, $i = 1, 2, \ldots, t$, then (2) can be rewritten as:

$$\begin{aligned} n_1 C_{(L_v+|m_k|+1)} \oplus n_2 C_{(L_v+|m_k|+2)} \oplus \ldots \oplus n_t C_{(L_v+L_m-|p|)} = \\ D \oplus b \oplus \left(\left[\begin{array}{cccc} C_1 & C_2 & \ldots & C_{L_v} \end{array}\right] v_k\right) \oplus \left(\left[\begin{array}{cccc} C_{(L_v+1)} & C_{(L_v+2)} & \ldots & C_{(L_v+|m_k|)} \end{array}\right] m_k\right) \oplus \\ \left(\left[\begin{array}{cccc} C_{(L_v+L_m-|p|+1)} & C_{(L_v+L_m-|p|+2)} & \ldots & C_{(L_v+L_m)} \end{array}\right] p\right) \end{aligned} \tag{4}$$

We now pick $rank(C)$ non-trivial equations in $n_1, n_2, \ldots, n_t$ from (4) and solve them using Gaussian Elimination. Note that we are guaranteed that the system of equations (4) will be consistent because of (3). If $rank(C) < |n|$, then $|n| - rank(C)$ of $n_1, n_2, \ldots, n_t$ are chosen arbitrarily.

(b) If (3) cannot be satisfied (perhaps because $|n|$ is too small), we first pick an arbitrary "noise" n of length $L_m - |m_k|$ and compute:

$$w_k = v_{k+1} = C \left[\begin{array}{c} v_k \\ m_k \\ n \end{array}\right] \oplus b \tag{5}$$

Now, we attempt to find an additional "noise" n' such that:

$$\left[\begin{array}{cccc} C_1 & C_2 & \ldots & C_{(L_v+L_m)} \end{array}\right] \left[\begin{array}{c} v_{k+1} \\ n' \\ p \end{array}\right] = D \oplus b \tag{6}$$

Again, we may assume that p is of minimum length. For n' in (6) to be found, we need for the following condition to hold:

$$rank\left(\left[\begin{array}{cccc} C_{(L_v+1)} & C_{(L_v+2)} & \ldots & C_{(L_v+L_m-|p|)} \end{array}\right]\right) = rank(C) \tag{7}$$

If the condition (7) is satisfied, n' can be computed by solving the system of linear equations suggested by (6), similar to the previous case.

(c) If (7) cannot be satisfied, we first choose some n with $|n| = L_m - |m_k|$ and compute $w_k = v_{k+1}$ as indicated in (5). Similar to the previous case, we seek to incorporate additional "noise" n' and n'' with $|n'| = L_m$ and $|n''| = L_m - |p|$ such that:

$$\begin{aligned} C \left[\begin{array}{c} v_{k+1} \\ n' \end{array}\right] &= w_{k+1} \oplus b = v_{k+2} \oplus b \\ C \left[\begin{array}{c} v_{k+2} \\ n'' \\ p \end{array}\right] &= D \oplus b \end{aligned} \tag{8}$$

For us to be able to find appropriate n' and n'', we require the following to be true:

$$rank\left(\left[\begin{array}{ccc} C_1 & \ldots & C_{(L_v+L_m-|p|)} \end{array}\right]\right) = rank\left(\left[\begin{array}{ccc} C_{(L_v+1)} & \ldots & C_{(L_v+L_m)} \end{array}\right]\right) = rank(C) \tag{9}$$

(d) If (9) cannot be satisfied, this implies that some bits in $D \oplus b$ depend solely on either the input chaining variable or the last few bits that correspond to the padding p in the last application of CF. For the bits in $D \oplus b$ that are a function of only the input chaining variable, all we need to make sure is that M'' involves the same number of applications of CF as M in producing a message digest, that is, $|M'' + padding| = |M + padding|$. Similarly, for the bits in $D \oplus b$ that are a function of the padding, we only need to make sure that the padding added for M'' is the same as that for M, which will be the case if $|M''| = |M|$. This restricts our choices for M', from which M'' is constructed. Note that the condition $|M''| = |M|$ is sufficient, but not necessary.

Thus, if A is similar to SHA, MD4 or MD5, we have shown how to construct a new message M'' that results in the same message digest D as a given message M.

4. The cases for when A works like MD2 are similar.

Thus, we have shown that a message digest algorithm that employs a compression function that is affine in its inputs can be easily subverted. In the next section, we confirm that fortunately, the widely used message digest algorithms SHA, MD2, MD4 and MD5 do not employ compression functions that are affine in their inputs.

2 The Compression Functions of SHA, MD2, MD4 and MD5 are not Affine

The input to each compression function can be perceived as a (bit) vector that is the member of a vector space of dimension $L_v + L_m$, where $L_m = 512$ for SHA, MD4 and MD5, $L_m = 128$ for MD2, $L_v = 160$ for SHA and $L_v = 128$ for MD2, MD4 and MD5. The output of the compression function is a member of a vector space of dimension L_v. We show that each compression function, CF, is not an affine transformation by picking three inputs x_1, x_2, x_3 such that the property $CF(x_1) \oplus CF(x_2) \oplus CF(x_3) = CF(x_1 \oplus x_2 \oplus x_3)$ is violated. Appendix A shows such sets of three inputs for each of the algorithms. Since that property must hold for a transformation to be affine, and since we can find x_1, x_2, x_3 that violate that property for each of the compression functions of SHA, MD2, MD4 and MD5, we have proved the following theorem.

Theorem 2.1 *The compression functions of SHA, MD2, MD4 and MD5 are not affine.*

References

[1] Proposed Federal Information Processing Standard for Secure Hash Standard. *Federal Register*, 57(21), January 1992.

[2] Proposed Revision of Federal Information Processing Standard (FIPS) 180, Secure Hash Standard. *Federal Register*, 59(131), July 1994.

[3] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, second edition, 1989.

[4] B. Kaliski. *RFC-1319 The MD2 Message-Digest Algorithm.* Network Working Group, April 1992.

[5] National Institute of Standards and Technology. *NIST FIPS PUB 180, Secure Hash Standard.* U.S. Department of Commerce, May 1994.

[6] R. Rivest. *RFC-1320 The MD4 Message-Digest Algorithm.* Network Working Group, April 1992.

[7] R. Rivest. *RFC-1321 The MD5 Message-Digest Algorithm.* Network Working Group, April 1992.

A Inputs to the Compression Functions that Show that they are not Affine

This section lists the sets of inputs x_1, x_2, x_3 for each of the compression functions, CF, of SHA, MD2, MD4 and MD4 such that the property $CF(x_1) \oplus CF(x_2) \oplus CF(x_3) = CF(x_1 \oplus x_2 \oplus x_3)$ is violated. Note that each x_i is 672 bits (twenty one 32-bit words) for SHA, 256 bits (eight 32-bit words) for MD2, and 640 bits (twenty 32-bit words) for MD4 and MD5.

SHA:

```
0x17F3C1BE 0x0B46E0A5 0x3680F780 0x02AC1A10 0x0CEF526F
0x33712CAB 0x04D7CFC0 0x0E1091E6 0x1E8DA02C 0x3E6670E6
0x14413E48 0x3240A248 0x09878EA3 0x335CB395 0x2D4065A1
0x2AB62683 0x13DA8F1A 0x0341E7F1 0x2FFD33B2 0x02013AA0
0x2F8A37F6

0x097911B3 0x2306F13B 0x1E8796E5 0x355FCC37 0x249B588C
0x1A11C15C 0x2746DB51 0x37ABB2F8 0x27380BAC 0x34B8910E
0x008FCB22 0x2D035EF2 0x2094153E 0x353C76A6 0x20CAF851
0x3301A5F7 0x3D6B5DE9 0x2F430157 0x3D36DF12 0x29098879
0x00ADA04F

0x0E9AF8AF 0x0EA77156 0x23026903 0x1443AC8F 0x35F48C17
0x2E46924B 0x341933BD 0x31C99BC4 0x35AAF255 0x0A5501D9
0x118275EA 0x2353F386 0x1A7A790B 0x32E93825 0x0D36E56B
0x1C29B966 0x14490438 0x3EAEB10F 0x28B7583B 0x298BF674
0x34E17DB4
```

MD2:

```
0x21A87D98 0x16D193D6 0x17CF5CF4 0x021143CC
0x046338A4 0x0A456E11 0x3F32766D 0x0C81184A

0x0263811F 0x0D779C30 0x2635F05B 0x06643243
0x1A9EE125 0x12DD7B2E 0x1CC6CB2D 0x2D7DB3F9

0x06695EDD 0x097700F1 0x35113822 0x36FDFB57
0x273671FA 0x20966DCB 0x0D3DE6C2 0x0F446B76
```

MD4:

```
0x3A804761 0x1DB19C8C 0x3AA1090E 0x3EFB8011 0x24105FBE
0x021553AB 0x3A78F23C 0x2D9E0908 0x345558D6 0x2C3B7CCA
0x080E10E9 0x39E112EF 0x2CF89D31 0x18032E02 0x2E325ED2
0x16A2ED03 0x0D61F325 0x203ADC17 0x13532243 0x168F03EF

0x0C76AA54 0x2E5B83FA 0x0F3DF395 0x17E0234A 0x0A5A792A
0x224F0447 0x38DBF2A9 0x281DE716 0x3A2D8A62 0x322D26C7
0x2BF5AC70 0x1FDD5D41 0x03D13A81 0x01F959ED 0x06F2B065
0x3683D725 0x1DA10557 0x33671A5B 0x2096560E 0x0E02EB09

0x1E3A2383 0x2D978C5C 0x35C3B281 0x3699A5A0 0x063557EE
0x1CD9C56A 0x0F3B3DDB 0x2A96EB87 0x01E9ED4D 0x2A6A45A8
```

```
   0x1F5AA8C9 0x02160AC9 0x232CE3A1 0x2C342166 0x17DF6201
   0x1EC1FC5D 0x13124DB9 0x0C905AA1 0x37A34BC8 0x0F91D5A5

MD5:

   0x27B2ECA4 0x330F00C1 0x048888EF 0x1E8BA477 0x1B0DA262
   0x31C8DD10 0x005C00A3 0x1237F9A0 0x1F8424CD 0x02AD7B37
   0x13814CEB 0x05489ABA 0x2A903332 0x35BAD080 0x08E48500
   0x01E839AC 0x00C14D98 0x06F4F74D 0x15EAD09B 0x38A46337

   0x027E9045 0x1924B69F 0x332B7B6D 0x2731FE78 0x10C3FF3B
   0x089B7991 0x0D36F8A6 0x172B3C6B 0x09E341B5 0x15ED36DB
   0x0D515671 0x34D17764 0x1E454DAC 0x28B64852 0x3A36A175
   0x3DCB0296 0x3D270356 0x2A58B265 0x30D83854 0x07E5698F

   0x195EB8A4 0x38BBDBA0 0x351C0F2D 0x1FD19FB8 0x0614B4C1
   0x310D342A 0x1AA1631B 0x3FE99FD7 0x2B899B98 0x1280CD49
   0x05795DB6 0x1EEC7B8F 0x13D4494D 0x390FE0DB 0x02EDA88F
   0x16BD5ADC 0x0F9F9DC1 0x303AC8DC 0x3A3EF0B6 0x23286B99
```