Collisions for the compression function of MD5

Bert den Boer
Philips Crypto B.V.
P.O. Box 218
5600 MD Eindhoven
The Netherlands

Antoon Bosselaers
ESAT Laboratory, K.U. Leuven
Kard. Mercierlaan 94
B-3001 Heverlee, Belgium
antoon.bosselaers@esat.kuleuven.ac.be

23 July 1993

Abstract. At Crypto '91 Ronald L. Rivest introduced the MD5 Message Digest Algorithm as a strengthened version of MD4, differing from it on six points. Four changes are due to the two existing attacks on the two round versions of MD4. The other two changes should additionally strengthen MD5. However both these changes cannot be described as well-considered. One of them results in an approximate relation between any four consecutive additive constants. The other allows to create collisions for the compression function of MD5. In this paper an algorithm is described that finds such collisions.
A C program implementing the algorithm establishes a work load of finding about 2^{16} collisions for the first two rounds of the MD5 compression function to find a collision for the entire four round function. On a 33MHz 80386 based PC the mean run time of this program is about 4 minutes.

1 Introduction

The MD5 Message Digest Algorithm [Rive91, Rive92b, Schn91] introduced by Ronald L. Rivest at Crypto '91 as a strengthened version of MD4 [Rive90, Rive92a] differs from MD4 on the following points:

- A fourth round has been added.
- The second round function has been changed from the majority function $XY \vee XZ \vee YZ$ to the multiplexer function $XZ \vee Y\overline{Z}$.
- The order in which input words are accessed in rounds 2 and 3 is changed.
- The shift amounts in each round have been changed. None are the same now.
- Each step now has a unique additive constant.
- Each step now adds in the result of the previous step.

The first four changes are clearly a consequence of the two existing attacks on the two round versions of MD4 [Merk90, dBBo91]. The last two changes should additionally strengthen MD5. However both these changes can hardly be described as well-considered.

The unique additive constant in step k contains the first 32 bits of the absolute value of $\sin(k)$. This together with the following relation between four consecutive sine values

$$(\sin(k) + \sin(k+2))\sin(k+2) = (\sin(k+1) + \sin(k+3))\sin(k+1)$$

establishes an approximate relation between any four consecutive additive constants. This could be easily avoided by choosing the next 32 bits in the binary expansion of the sine values.

The last change however has more serious implications: adding in the result of the previous step allows to create collisions for the compression function of MD5. In this paper an algorithm that finds such collisions is described. This means that one of the design principles behind MD5, namely to design a collision resistant hash function based on a collision resistant compression function, is not satisfied. The entire 640-bit input of the compression function is used to produce these collisions. Therefore they do not result in an attack on the MD5 hash function, having a single and fixed 128-bit initial value. This is why they are sometimes called pseudo-collisions.

In Section 2 the necessary notation and definitions are introduced. Section 3 describes and explains the actual collision search algorithm. Section 4 contains a discussion about the optimal value for a constant of the collision search algorithm. Finally, in Section 5 some details on the implementation as well as an example collision are given.

2 Notation and definitions

The following notation will be used:

XY, $X \vee Y$, $X \oplus Y$	respectively the bitwise AND, OR and XOR of X and Y
$\overline{X}$	the bitwise complement of X
$X \lll s$, $X \ggg s$	the rotation of X to respectively the left and right by s bit positions
$V \leftarrow E$	assign to variable V the value of the expression E
MSB, LSB	respectively most and least significant bit

A *word* is defined as an unsigned 32-bit quantity taking on only nonnegative values. The word wise application of the operation $\star$ on two 4-word buffers (A_1, B_1, C_1, D_1) and (A_2, B_2, C_2, D_2) is denoted by

$$(A_1, B_1, C_1, D_1) \star (A_2, B_2, C_2, D_2) = (A_1 \star A_2, B_1 \star B_2, C_1 \star C_2, D_1 \star D_2).$$

MD5 uses the following four functions (one for each round) to process the input. They all take a 3-word input and produce a single word of output.

$$\begin{aligned} f_1(X,Y,Z) &= XY \vee \overline{X}Z & \quad f_3(X,Y,Z) &= X \oplus Y \oplus Z \\ f_2(X,Y,Z) &= XZ \vee Y\overline{Z} & \quad f_4(X,Y,Z) &= (X \vee \overline{Z}) \oplus Y \end{aligned}$$

Each round i $(1 \leq i \leq 4)$ consists of 16 steps, each of which contains a single application of the round function f_i. Hence each round function is used 16 times. In addition each step of round i uses one of the shift constants $si1$, $si2$, $si3$ or $si4$, each of which is used four times in each round (see Table 1). In total there are 48 steps in MD5, grouped in 4 rounds of 16 steps and numbered from 1

	Round i			
	1	2	3	4
$si1$	7	5	4	6
$si2$	12	9	11	10
$si3$	17	14	16	15
$si4$	22	20	23	21

Table 1. The 16 different shift constants of MD5

(step 1 of round 1) through 48 (step 16 of round 4). For a complete specification of MD5 the reader is referred to the original description [Rive92b, Schn91]. Note that in this original description of MD5 the designation f, g, h and i are used for respectively the round functions f_1, f_2, f_3 and f_4.

3 Description of the collision search algorithm

First the condition imposed on two inputs to produce the same image under the compression function of MD5 will be translated to a condition on the inputs to the round function of each step of MD5. Next we will show how these conditions can be (easily) met for the third and fourth round. Finally we will derive an algorithm that generates an input meeting the conditions for the first and second round.

3.1 Derivation of the round function input condition

The basis of the MD5 algorithm is a compression function G that takes as input a 4-word buffer (A, B, C, D) and a 16-word message block $(X[0], X[1], \ldots, X[15])$, and produces a 4-word output (AA, BB, CC, DD):

$$(AA, BB, CC, DD) = G((A, B, C, D), (X[0], X[1], \ldots, X[15])).$$

The idea of the collision search algorithm is to produce an input to the compression function such that complementing the MSB of each of the 4 words of the buffer (A, B, C, D) has no influence on the output of the compression function. In other words, finding an (A, B, C, D) and an $(X[0], X[1], \ldots, X[15])$ such that

$$G((A, B, C, D) \oplus (2^{31}, 2^{31}, 2^{31}, 2^{31}), (X[0], X[1], \ldots, X[15])) = G((A, B, C, D), (X[0], X[1], \ldots, X[15])). \quad (1)$$

The compression function G of MD5 consists of four 16-step rounds enclosed by a feedforward, that adds (modulo 2^{32}) to each of the 4 words A, B, C and D at the end of the fourth round the values they had at the beginning of the first round. Hence

$$G(A, B, C, D) = H(A, B, C, D) + (A, B, C, D), \quad (2)$$

where H consists of the four 16-step rounds. Substituting G in (1) by (2) together with the fact that $(A + 2^{31}) \bmod 2^{32} = A \oplus 2^{31}$ means that we are looking for an (A, B, C, D) and an $(X[0], X[1], \ldots, X[15])$ such that

$$H((A, B, C, D) \oplus (2^{31}, 2^{31}, 2^{31}, 2^{31}), (X[0], X[1], \ldots, X[15])) = \\ H((A, B, C, D), (X[0], X[1], \ldots, X[15])) \oplus (2^{31}, 2^{31}, 2^{31}, 2^{31}).$$

Consider a step of the MD5 algorithm

$$A \leftarrow B + ((A + f_i(B, C, D) + X[j] + t) \lll s),$$

where

- $1 \leq i \leq 4$,
- $X[j]$ is one of the 16 message words ($0 \leq j \leq 15$),
- t is the unique additive constant of the step,
- s is one of the 16 possible shift amounts $si1$, $si2$, $si3$ or $si4$ ($1 \leq i \leq 4$), and
- all additions are modulo 2^{32}.

The new value of the word A is obtained by adding to the result of the previous step the result of an addition rotated over s bits to the left. Complementing the MSB of each of the 4 words A, B, C and D in the right hand side of this assignment will result in a complementation of the MSB of the updated A, if the MSB of $f_i(B, C, D)$ is complemented when the MSBs of B, C and D are complemented. This observation leads to the following proposition.

Proposition 1. *Let T be a 20-word input to the compression function G and let X, Y and Z be the MSBs of the 3-word input to a round function f_i. If T produces in all steps inputs to the round functions f_i for which*

$$f_i(\overline{X}, \overline{Y}, \overline{Z}) = \overline{f_i(X, Y, Z)}$$

then the 20-word input T and the 20-word input in which the MSBs of the first four words of T are complemented have the same image under G.

Note that this is made possible by adding in, in each step, the result of the previous step. This is why this attack does not work for MD4. Note also that this collision has the property that the message part of the input is the same.

Proposition 2. *The condition $f_i(\overline{X}, \overline{Y}, \overline{Z}) = \overline{f_i(X, Y, Z)}$ is met by the following 3-tuples (X, Y, Z) for respectively f_1, f_2, f_3, and f_4.*

1. $(0,0,0)$, $(1,0,0)$ *and their complements* $(1,1,1)$ *and* $(0,1,1)$,
2. $(0,0,0)$, $(0,0,1)$ *and their complements* $(1,1,1)$ *and* $(1,1,0)$,
3. *all inputs,*
4. $(0,0,0)$, $(0,1,0)$ *and their complements* $(1,1,1)$ *and* $(1,0,1)$.

Proof.

1. $\overline{XY} \vee X\overline{Z} = \overline{XY \vee \overline{X}Z}$

   $\Leftrightarrow \overline{\overline{XY} \vee X\overline{Z}} \oplus (XY \vee \overline{X}Z) = 0$

   $\Leftrightarrow (X \vee Y)(\overline{X} \vee Z) \oplus (XY \vee \overline{X}Z) = 0$

   $\Leftrightarrow (\overline{X}Y \vee XZ) \oplus (XY \vee \overline{X}Z) = 0$

   $\Leftrightarrow Y \oplus Z = 0$

2. the same as above, but with X and Z interchanged.
3. $\overline{X \oplus Y \oplus Z} = \overline{X} \oplus Y \oplus Z = \overline{X} \oplus \overline{Y} \oplus \overline{Z}$.
4. $(\overline{X} \vee Z) \oplus \overline{Y} = \overline{(X \vee \overline{Z}) \oplus Y}$

   $\Leftrightarrow (\overline{X} \vee Z) \oplus (X \vee \overline{Z}) = 0$

   $\Leftrightarrow X \oplus Z = 0$

□

3.2 Collisions for round 3 and 4

From Proposition 2 it follows that a random 20-word input to round 4 has a probability of 2^{-16} of fulfilling the condition $f_4(\overline{X}, \overline{Y}, \overline{Z}) = \overline{f_4(X, Y, Z)}$ for all 16 steps of the round. Round 3 imposes according to the same proposition no additional constraints. Due to the pseudo-random behaviour of round 3 it is save to assume that the input at the beginning of round 4 does not significantly deviate from a random one. A 20-word random input has therefore the same probability of 2^{-16} of meeting all conditions in both round 3 and 4. It remains to produce enough 20-word inputs fulfilling all conditions in the first two rounds in order to generate a collision for the compression function G.

3.3 Collisions for round 1 and 2

According to Proposition 2 the condition $f_1(\overline{X}, \overline{Y}, \overline{Z}) = \overline{f_1(X, Y, Z)}$ is met by both $(1, 1, 1)$ and $(1, 0, 0)$, and their complements. However in each step of the first round only one of A, B, C and D is updated. Therefore an appearance of $(1, 0, 0)$ in a particular step will lead to $(x, 1, 0)$ in the next step (where x is either 1 or 0). Since $f_1(\overline{x}, 0, 1)$ is not equal to $\overline{f_1(x, 1, 0)}$, $(1, 0, 0)$ cannot appear as input to the function f_1 in the course of the first round. The same applies to its complement $(0, 1, 1)$ and to the inputs $(0, 0, 1)$ and $(1, 1, 0)$ to the second round function f_2. Hence only $(1, 1, 1)$ or its complement $(0, 0, 0)$ are allowed as inputs to the first and second round functions f_1 and f_2. This input condition is met by keeping the MSBs of A, B, C and D in the first two rounds equal to one, except for the value of A at the beginning of the first round and the value of B at the end of the second round, for which there are no constraints: they are not used as input to f_1 or f_2. The idea of the algorithm is therefore to choose the 16 words $X[0], X[1], \ldots, X[15]$ in precisely such a way that all the input words to

the f_1 and f_2 function keep their MSB on one during the first two rounds. This is done in the following way.

We start halfway the first two rounds by generating random A, B, C and D values between the first and second round with MSBs equal to one. We walk through the second round making all the updated buffer words equal to a "magic value" N by specific choices for the 16 message words $X[0]$ through $X[15]$. This is called the forward walk. The best choice for N depends on the actual values of the shift constants in the first two rounds and will be discussed in Section 4. For the current values of the shift constants the best choice for N is `F8000000` (hexadecimal notation).

Next we check whether the choices for the message words made in the second round are also good choices for the first round, i.e., whether they keep the MSB of the buffer words in the first round on one. We therefore start at the end of the first round and walk through the first round in the reverse direction. This is called the backward walk. When we find a buffer word with zero MSB, we adapt the most significant part of the message word used in that particular step in such a way that the buffer word now approximates the magic value N. We then once again start the forward walk at the second round step where this message word is used, and check whether this change has any influence on the MSBs of the remaining buffer words of the second round. If so, we make the necessary changes to the other (i.e., least significant) part of the message words in order that the buffer words approximate once again the magic value. These least significant parts of the message words become the most significant after the rotation in the forward walk steps. Next we start once again the backward walk. This way we go to and fro until we reach the beginning of the first round, at which point we found a message block keeping the MSBs of the buffer words in the first two rounds on one.

First a description of the initialization procedure is given, which consists of a forward walk and partial backward walk.

1. *Initialize* (A, B, C, D).
   Generate random A, B, C, D values between the first and second round with MSBs equal to one.

2. *Initialize* $(X[0], X[1], \ldots, X[15])$.
   2.1 Step forwards (i.e., into round 2) and make the updated buffer words in the first six steps of round 2 (step 17 through 22) equal to the magic value N by a specific choice of the message words used in the first six steps: respectively $X[1]$, $X[6]$, $X[11]$, $X[0]$, $X[5]$ and $X[10]$.
   2.2 Do the next step (step 23) forwards making the updated value of C equal to N by a specific choice for X[15].

$$X[15] = ((N - D) \ggg s23) - C - f_2(D, A, B) - 3634488961$$

   Do the last step of the first round (step 16) backwards making the value of B at the beginning of step 16 equal to N by another specific choice

for X[15].

$$X[15] = ((B - C) \ggg s14) - N - f_1(C, D, A) - 1236535329$$

Of course we get different values for X[15] but we take the $s23$ MSBs of the backward step solution and the other $32 - s23$ bits of the forward step solution. This way both newly computed values of C (forward step) and B (backward step) are approximations of N:

$$C' = D + (C + f_2(D, A, B) + X[15] + 3634488961) \lll s23,$$

$$B' = ((B - C) \ggg s14) - X[15] - f_1(C, D, A) - 1236535329.$$

In the forward step the $s23$ bits of the backward solution become the LSBs after the rotation of the sum over $s23$ bits, in the backward step the bits of the forward solution are on the least significant positions as well.

2.3 Step forwards (steps 24 and 25) computing $X[4]$ and $X[9]$ as in step 2.1
2.4 Put $X[14]$ equal to the $s22$ MSBs of the backward solution of step 15 and the $32 - s22$ LSBs of the forward solution of step 26.
2.5 Step forwards (steps 27 and 28) computing $X[3]$ and $X[8]$ as in step 2.1
2.6 Put $X[13]$ equal to the $s21$ MSBs of the backward solution of step 14 and the $32 - s21$ LSBs of the forward solution of step 29.
2.7 Step forwards (steps 30 and 31) computing $X[2]$ and $X[7]$ as in step 2.1
2.8 Put $X[12]$ equal to the backward solution of step 12, as there are no constraints on the value of B at the end of the second round (step 32).

Next an informal and formal description of the actual algorithm is given. First we define three functions used in these descriptions. Let

- $s2[j]$ be the shift constant used in step j of the second round ($17 \leq j \leq 32$).
- $fw[i]$ be the step in the forward walk (i.e., the second round) using the input word $X[i-1]$ ($1 \leq i \leq 16$),
- $bw[j]$ be the step in the backward walk (i.e., the first round) using the message word that is used in the jth step of the forward walk ($17 \leq j \leq 32$). Hence the functions $fw[]$ and $bw[]$ are each others inverse: if $j = fw[i]$ is the step in the forward walk using $X[i-1]$, then $i = bw[j]$ is the step in the backward walk using the message word that is used in the jth step of the forward walk (i.e., $X[i-1]$).

After the initialization of both (A, B, C, D) and $(X[0], X[1], \ldots, X[15])$ as already described, we step backwards checking whether our choices for the $X[\cdot]$'s so far are also good choices for the backward walk i.e., whether *at the beginning* of each first round step the MSB of the buffer word being updated is equal to one. If that is not the case for step i the first $s2[fw[i]]$ (i.e., the shift constant of the step in the forward walk using $X[i-1]$) bits of $X[i-1]$ are adapted such that the value of the buffer word at the beginning of that step is, given these limitations, the best possible approximation of the magic value N. Alas, now all

values in the forward walk from step $fw[i]$ onwards change. The first changes are mild, but soon they will accumulate. But as long as the MSBs of the buffer words A, B, C and D do not change, we keep the $X[\cdot]$ values as they are. However if in step j of the forward walk the MSB of a buffer word changes, we adapt all or part of the bits of the message word used in that step (i.e., $X[bw[j]-1]$) to let the updated value of the buffer word approximate once again the magic value N. For this purpose we can use all bits of $X[bw[j]-1]$ in case, up to this point, it has not been used yet in the backward walk (i.e., if $bw[j] < i$). Otherwise we combine the forward and backward solutions for $X[bw[j]-1]$. Having completed the entire forward walk in the same way, we once again start the backward walk at step k, where $X[k-1]$ is the message word with the highest index that was changed in the forward walk, and we check whether these new choices for the $X[\cdot]$'s are also good choices for the backward walk. This way we go to and fro, until we find a solution meeting all conditions in both rounds. Below the formal description of the algorithm is given together with a flowchart in Figure 1.

3. *The actual algorithm.*
   - 3.0 Set $i \leftarrow 12$.
   - 3.1 If $i = 1$, a solution has been found as there are no constraints on the value of A at the beginning of the first round.
   - 3.2 Do step i backwards. The value at the beginning of step i of the buffer word that is updated in this step, is calculated using the known value at the end of the step and the value of $X[i-1]$ from the forward walk.
   - 3.3 If the MSB of the new value is 1, decrement i and goto 3.1.
   - 3.4 Set $j \leftarrow fw[i]$, $k \leftarrow i$ (k keeps track of the highest first round step using a message word that has been adapted during the forward walk). Adapt the $s2[j]$ MSBs of $X[i-1]$ to let the value of the buffer word *at the beginning* of first round step i approximate the magic value N.
   - 3.5 If $j = 32$, set $i \leftarrow k$ and goto 3.1, as there are no constraints on the value of B at the end of the second round.
   - 3.6 Do step j forwards.
   - 3.7 If the MSB of the updated buffer word is 1, increment j and goto 3.5.
   - 3.8 If $bw[j] < i$, compute $X[bw[j]-1]$ as in step 2.1 (i.e., if the message word used in this step has not been used yet in the backward walk, then use all the bits of this message word to make the updated value of the buffer word equal to N). Increment j and goto 3.5.
   - 3.9 Adapt the $32-s2[j]$ LSBs of $X[bw[j]-1]$ to let the updated value of the buffer word in step j approximate the magic value N (i.e., in case the message word used in this step has already been used in the backward walk).
   - 3.10 If $bw[j] > k$, set $k \leftarrow bw[j]$ (the highest first round step so far using a message word that has been changed during this forward walk, and hence the place to start a new backward walk).
   - 3.11 Increment j and goto 3.5.

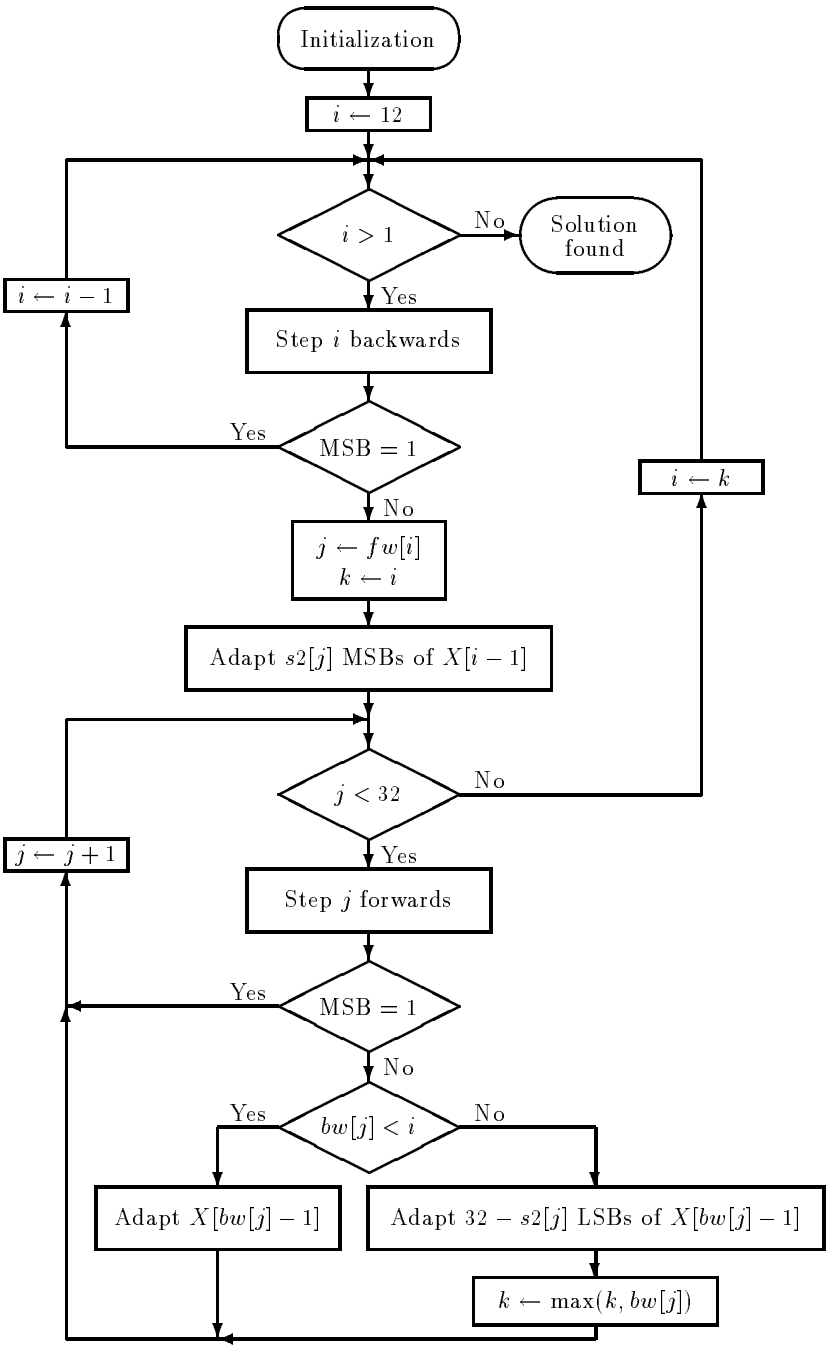

Fig. 1. Flowchart of the collision search algorithm

There is of course a real danger for the algorithm to get in an endless loop. Therefore we count the number of times an $X[\cdot]$ value has been adapted. If that number becomes larger than a certain value, we stop and try another initial value for the 4-word buffer (A, B, C, D) at the end of the first round. Computer simulations show that the algorithm either converges very quickly to a solution or gets stuck into an endless loop, so that this value can be chosen quite small (e.g., 300). The closer the shifts in the second round are to 16, the smaller the probability to get into such an endless loop, since then nearly the same number of bits of the forward and the backward solution are used. The part of the backward step solution of $X[\cdot]$ will therefore change the MSB of the forward step buffer with a relatively small probability, and vice versa. However for the steps using the second round shift $s21$ the situation is totally different: here only five bits of the backward step solution are used, making it quite probable that the MSB of the backward step buffer gets changed by the part of the forward step solution. A good choice for the magic value can reduce the probability that this happens to a minimum.

4 Choice of the magic value

The MSB of the magic value N must of course be one, as it is intended to be the intermediate value of the buffer words A, B, C and D in the first two rounds. Moreover at least one other bit of N must be nonzero to allow small negative changes to N without affecting its MSB. The more significant this bit is, the less susceptible N becomes to a change of its MSB as a result of a subtraction. The critical steps in this regard are the first round steps 2, 6, 10 and 14, using respectively message words $X[1]$, $X[5]$, $X[9]$ and $X[13]$. In the second round these message words are used in combination with the shift constant $s21 = 5$, which means that only 5 bits of the backward walk solution are used to let the backward walk buffer word approximate the magic value. The magic value should therefore be greater or equal to `0x88000000`, i.e., all 32-bit values with at least two of the five MSBs on one and the 27 LSBs on zero are 'good' magic values. Computer simulations show that the best choice for N is `0xF8000000`: for only about 0.15% of all initial values the algorithm gets caught in an endless loop (see Figure 2).

Instead of using a single magic value for the entire first two rounds, we can of course use different magic values for each step. As we have shown in the case of a single magic value, the number of nonzero MSBs of the best magic value is related to the shift constant used in a particular step, i.e., to the number of bits of the message word used in that step that can be changed to approximate this magic value. Therefore it makes no sense to choose more than eight different magic values: four for the forward walk and four for the backward walk. Computer simulations show that in doing so the number of endless loops can be reduced to about 0.02%, but the mean time to find a solution increases by about 25%. As the figure of 0.15% endless loops is very much acceptable, we decided to stick to a single magic value of `0xF8000000`.

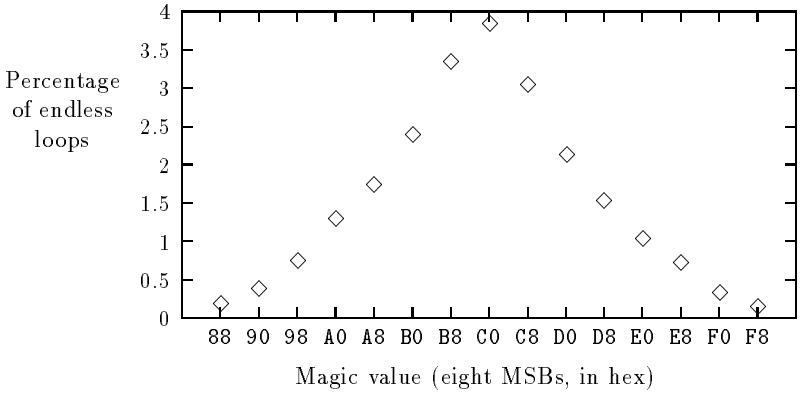

Fig. 2. Percentage of endless loops for the different 'good' magic values. Only the 8 MSBs of each magic value are indicated, the 24 LSBs are all zero.

5 Implementation

A C program has been written implementing the algorithm. It establishes a work load of finding about 2^{16} collisions for the first two rounds of the MD5 compression function to find a collision for the entire four round function. On a 33MHz 80386 based PC using a 32-bit compiler the mean time to find such a collision is about 4 minutes (2^{16} trials). However the variance is quite dramatic. Times have been observed ranging from about 1 second (317 trials) to more than 25 minutes (396324 trials). As an example, the following two 20-word inputs consisting of the common 16-word message part (hexadecimal notation)

```
5FFBB485 B73256D8 19DF08E4 11054A66 22C00E98 450A05C4 5F53A940 9DDC1CF8
DADAB3DB 8A43597A 4CA51993 E7DB12E5 1F1C0317 9A3BAAD6 B275B7BB 0F09CFD5
```

and respectively the 4-word input buffers `I1` and `I2`

```
I1: 399E49D4 876C9442 F7DFE793 83D49001
I2: B99E49D4 076C9442 77DFE793 03D49001
```

are both compressed to the same 4-word output buffer

```
F80668D5 F8AB5C93 C93998F5 D007A636
```

References

[Rive90] R.L. Rivest, "The MD4 message digest algorithm," *Advances in Cryptology, Proc. Crypto'90, LNCS 537*, S. Vanstone, Ed., Springer-Verlag, 1991, pp. 303–311.

[Merk90] R.C. Merkle, Unpublished result, 1990.

[dBBo91] B. den Boer and A. Bosselaers, "An attack on the last two rounds of MD4," *Advances in Cryptology, Proc. Crypto'91, LNCS 576*, J. Feigenbaum, Ed., Springer-Verlag, 1992, pp. 194–203.

[Rive91] R.L. Rivest, "The MD5 message digest algorithm," *Presented at the rump session of Crypto'91.*

[Schn91] B. Schneier, "One-way hash functions," *Dr. Dobb's Journal*, Vol. 16, No. 9, 1991, pp. 148–151.

[Rive92a] R.L. Rivest, "The MD4 message-digest algorithm," *Request for Comments (RFC) 1320*, Internet Activities Board, Internet Privacy Task Force, April 1992.

[Rive92b] R.L. Rivest, "The MD5 message-digest algorithm," *Request for Comments (RFC) 1321*, Internet Activities Board, Internet Privacy Task Force, April 1992.