A Study of the MD5 Attacks: Insights and Improvements

John Black[1] and Martin Cochran[1] and Trevor Highland[2]

[1] University of Colorado at Boulder, USA
www.cs.colorado.edu/∼jrblack, ucsu.colorado.edu/∼cochranm
jrblack@cs.colorado.edu, cochranm@cs.colorado.edu
[2] University of Texas at Austin, USA
trevor.highland@gmail.com

Abstract. MD5 is a well-known and widely-used cryptographic hash function. It has received renewed attention from researchers subsequent to the recent announcement of collisions found by Wang et al. [16]. To date, however, the method used by researchers in this work has been fairly difficult to grasp.
In this paper we conduct a study of all attacks on MD5 starting from Wang. We explain the techniques used by her team, give insights on how to improve these techniques, and use these insights to produce an even faster attack on MD5. Additionally, we provide an "MD5 Toolkit" implementing these improvements that we hope will serve as an open-source platform for further research.
Our hope is that a better understanding of these attacks will lead to a better understanding of our current collection of hash functions, what their strengths and weaknesses are, and where we should direct future efforts in order to produce even stronger primitives.

Keywords: Cryptographic Hash Functions, Differential Cryptanalysis, MD5.

1 Introduction

Background. MD5 was the last in a succession of cryptographic hash functions designed by Ron Rivest in the early 1990s. It is a widely-used well-known 128-bit iterated hash function, used in various applications including SSL/TLS, IPSec, and many other cryptographic protocols. It is also commonly-used in implementations of timestamping mechanisms, commitment schemes, and integrity-checking applications for online software, distributed filesystems, and random-number generation. It is even used by the Nevada State Gaming Authority to ensure slot-machine ROMs have not been tampered with.

Cryptographic hash functions like MD5 do not have a sound mathematical security definition, but instead rely on the following "intuitive" notions of security: for a hash function h with domain D and range R, we require the following three properties.[1]

[1] For a more complete discussion of hash function security definitions, see [12].

Pre-image Resistance: For a given $y \in R$, it should be "computationally infeasible" to find an $x \in D$ such that $h(x) = y$.

Second Pre-image Resistance: For a given $x \in D$, it should be "computationally infeasible" to find a distinct $x' \in D$ such that $h(x) = h(x')$.

Collision Resistance: It should be "computationally infeasible" to find distinct $x, x' \in D$ such that $h(x) = h(x')$.

In all attacks described in this paper, the focus is on violating the last requirement above: that is, we wish to find collisions in MD5.

In 1993 B. den Boer and A. Bosselaers [4] found two messages that collided under MD5 with two different IVs. In 1996 H. Dobbertin [5] published an attack, without details, that found a collision in MD5 with a chosen IV different from MD5's. Finally, at CRYPTO 2004, a team of researchers from the Shandong University in Jinan China, led by Xiaoyun Wang, announced collisions in MD5 as well as collisions in a host of other hash functions including MD4, RIPEMD, and HAVAL-128. Their findings were published at EUROCRYPT in 2005 [15, 16]. The same team presented two papers at the 2005 CRYPTO conference detailing applications of their methods to the hash functions SHA0 and SHA1, with a generated collision for SHA0, and a description on how to obtain collisions in SHA1. Given the variety of hash functions attacked by this team, it seems likely that their approach may prove effective against all cryptographic hashes in the MD family, including all variants of SHA. It therefore seems worthwhile to seek a complete understanding of how this approach works, how it can be improved, and how it can be generalized.

In Wang's short talk at the CRYPTO rump session, few details were given. She presented a brief general overview of the attacks, including the exact differentials for the pairs of colliding message blocks, along with several example collisions and estimations of the time complexity for each attack. In the interim, between her talk and the publication of the team's papers [15, 16], much interest was generated in finding the methods used by the Chinese researchers, and several papers were published on the subject [6, 8, 9]. Unfortunately, some key details of the attacks are omitted from the EUROCRYPT papers, and there are several discrepancies between the analysis done in [6, 9] and the results presented by the Chinese team.

Our Contributions. This paper attempts to consolidate and summarize all relevant knowledge of the attacks on MD5 from the works cited above [6, 8, 9, 15, 16], then additionally offer new insights and further improvements to this body of work. Specifically:

- We fully explain the "multi-message modification" technique invented by Wang.
- We offer new insights on how to find other differential paths.
- We use the above insights to demonstrate how to satisfy several more conditions in round 2 of the MD5 computation, thereby significantly speeding up the search for collisions.

– We demonstrate new methods for decreasing the search complexity when finding collisions.
– We provide an "MD5 Toolkit" that uses the above optimizations to produce MD5 collisions faster than any other known implementation; it also serves as a platform for testing further improvements and new ideas.

Along the way, we correct many of the errors made by previous authors in their published analyses, and we use what, we believe, is an improvement in notation. Also, in contrast to the other publications above, we provide full source code implementing our methods as an "MD5 Toolkit." Our hope is that this toolkit will serve as a useful device for researchers wishing to explore further techniques in this line of work. For example, making further code optimizations or search optimizations, adding further conditions, or searching for differential paths in an automated way. The MD5 Toolkit can be found at `http://www.cs.colorado.edu/~jrblack/md5toolkit.tar.gz`.

Our ultimate goal as a research community is to understand as best we can the way these iterated hash functions work, and the best known attacks against them. Our hope is that the observations offered here, along with the specific improvements we make for MD5 collision-finding, will lead to progress along these lines.

Overview of the Paper. We begin by covering the notation used throughout the paper. Section 3 reviews the specification of MD5. We then give a high-level overview of the attacks and touch on the motivation and theory behind the attacks in section 4. Then we move on to the details of the attack in section 5.

The remainder of the paper is devoted to detailing our insights and improvements. Specific to MD5, we offer improvements that reduce the best-known time complexity [9] by roughly a factor of three. The methods used by the Chinese team require an expected 2^{37} MD5 computations to find the first block pair of the colliding messages, and an expected 2^{30} MD5 computations to find the second block pair. Klima [9] improved the attack so that an expected 2^{33} and 2^{24} MD5 computations are needed to find the first and second message block pairs, respectively, although Klima did not implement his improved attack for finding the second block pair. Our method improves the attack so that an expected 2^{30} MD5 computations are required to find the first block pair, and we implement Klima's code for finding the second block pair.

The Wang team reported that the example collision they found for the first block took about an hour on an IBM supercomputer, and the second block pair was found in 15 seconds to 5 minutes on the same computer. Our code produces both blocks in an average of 11 minutes on a commodity PC.

Latest Results. Since the publication of our paper, more progress has been made in attacking MD5. The HashClash project[14] implements Wang's attack using parallel computing resources to produce collisions in under 1 minute. Klima [7] has invented a new technique called "tunneling" also resulting in code that produces collisions in under a minute.

2 Notation

All indices start at 0. This is in contrast to the notation used in the Wang et al. papers, as well as [6, 9]. Thus, for a 4-byte unsigned integer x, the bits are labeled from 0 to 31, with 0 referring to the least significant bit. Let $\{0,1\}^n$ denote the set of all binary strings of length n. For an alphabet Σ, let Σ^* denote the set of all strings with elements from Σ. Let $\Sigma^+ = \Sigma^* - \{\epsilon\}$ where ϵ denotes the empty string. For strings s, t, let $s \,\|\, t$ denote the concatenation of s and t. For a binary string s let $|s|$ denote the length of s. For a string s where $|s|$ is a multiple of n, let $|s|_n$ denote $|s|/n$. Given binary strings s, t such that $|s| = |t|$, let $s \oplus t$ denote the bitwise XOR of s and t. For a string M such that $|M|$ is a multiple of n, $|M|_n = k$, then we will use the notation $M = (M_0, M_1, M_2, \ldots, M_{k-1})$ such that $|M_0| = |M_1| = |M_2| = \ldots = |M_{k-1}| = n$. We will also use the notation $M = (m_0, m_2, \ldots, m_{k-1})$ such that $|m_0| = |m_2| = \ldots = |m_{k-1}| = n$. This latter notation is used when $n = |m_i| = 32$. The former notation will be used when $n = |M_i| = 512$. We may think of M as a k-tuple if it is convenient (hence the vector notation). Generally, the symbol M will be used for members of $(\{0,1\}^{512})^+$. For a set S of the form $\{A_i : a \le i \le b\}$, we will sometimes denote S as $A_{a:b}$.

XOR Differential vs. Subtraction Differential. These methods use a combination of the XOR differential and the subtraction differential, but with an emphasis on the subtraction differential. That is, for two integers $x, x' \in [0, 2^{31} - 1]$, consider the function $\Delta_X(x, x') = x \oplus x'$. This defines the XOR differential for x, x'. Alternatively, define $\Delta_S(x, x')$ as $x' - x \bmod 2^{32}$. This is the subtraction differential. The Chinese authors supply two columns of differentials in their tables of differentials for each step. One column contains the subtraction differential. Another contains what is essentially the XOR differential, but there is extra information included to indicate bit differences. For example, let $\Delta_S(x, x') = 2^2$. There are many possibilities for $\Delta_X(x, x')$ such as these three examples.

- $\Delta_X(x, x') =$ `0x00000004` (there is only one bit different between x and x', in index 2)
- $\Delta_X(x, x') =$ `0x0000000c` (bit 3 is set in x' but is not set in x, bit 2 is not set in x' but is set in x)
- $\Delta_X(x, x') =$ `0x0000fffc` (bit 15 is set in x' but is not set in x, bits 2 through 14 are not set in x' but are set in x

The differential used in [15, 16] captures this type of information by the following notation. Let x be in $[0, 2^{31} - 1]$. Then $x' = x[a_1, a_2, \ldots, a_n, -b_1, -b_2, \ldots, -b_m]$ denotes $x' = x + 2^{a_1} + 2^{a_2} + \cdots + 2^{a_n} - 2^{b_1} - 2^{b_2} \cdots - 2^{b_m} \bmod 2^{32}$. From this information one can compute both $\Delta_X(x, x')$ and $\Delta_S(x, x')$ if and only if for every index i for which x and x' differ $i \in \{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m\}$. The complete differential tables in the full version of this paper [1] use this specialized differential, but with the above property so that both Δ_X and Δ_S may be computed.

3 The MD5 Algorithm

The following is a brief description of MD5 using the notation that is used to describe the attacks later in this paper. We omit message padding in this description since it has no effect on our attacks. The full specification for MD5 can be found in [11].

MD5 is a hash function in the Merkle-Damgård paradigm [2, 10], where the security of the hash function reduces to the security of its compression function. The MD5 compression function, which we denote as $\mathrm{MD5}_c$, accepts as input a 128-bit chaining value CV which we break into four 32-bit values cv_0, cv_1, cv_2, cv_4 and a 512-bit message block M and outputs a 128-bit chaining value CV'. Formally, $\mathrm{MD5}_c : \{0,1\}^{128} \times \{0,1\}^{512} \to \{0,1\}^{128}$. Let $H_0 \in \{0,1\}^{128}$ and let $M = (M_0, M_1, \ldots, M_k)$ for some $k \geq 0$ and $|M_i| \in \{0,1\}^{512}$ for $0 \leq i \leq k$. Then $\mathrm{MD5}(M)$ is computed as follows. Let $H_{i+1} = \mathrm{MD5}_c(H_i, M_i)$ for $0 \leq i \leq k$. $\mathrm{MD5}(M)$ is defined as H_{k+1}.

3.1 The compression function $\mathbf{MD5}_c$

We now detail the compression function used in MD5. There are 64 intermediate values produced, which we will call step values and denote by Q_i for $0 \leq i < 64$. The step values are computed in the following fashion:

$$T_i \leftarrow \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) + Q_{i-4} + w_i + y_i$$
$$Q_i \leftarrow Q_{i-1} + (T_i \lll s_i)$$

Where s_i, y_i are step-dependent constants and w_i is the i-th block of the initial message expansion. For $0 \leq i < 64$, $w_i = m_j$ for some $0 \leq j < 16$. The exact message expansion can be found in [11]. By '$x + y$' we mean the addition of x and y modulo 2^{32}, and by '$x \lll y$' we mean the circular left shift of x by y bit positions (similarly, '$x \ggg y$' denotes the circular right shift of x by y bit positions).

The Φ function is defined in the following manner:

$$\begin{array}{ll}
\Phi_i(x,y,z) = F(x,y,z) = (x \wedge y) \vee (\neg x \wedge z), & 0 \leq i \leq 15 \\
\Phi_i(x,y,z) = G(x,y,z) = (x \wedge z) \vee (y \wedge \neg z), & 16 \leq i \leq 31 \\
\Phi_i(x,y,z) = H(x,y,z) = x \oplus y \oplus z, & 32 \leq i \leq 47 \\
\Phi_i(x,y,z) = I(x,y,z) = y \oplus (x \vee \neg z), & 48 \leq i \leq 63
\end{array}$$

$Q_{-1}, \ldots, Q_{-4}$ are determined by the chaining values to MD5 so that

$$Q_{-4} \leftarrow cv_0,\ Q_{-3} \leftarrow cv_3,\ Q_{-2} \leftarrow cv_2,\ Q_{-1} \leftarrow cv_1$$

The chaining values are initially set to, in big endian byte order,

$$cv_0 \leftarrow \mathtt{0x01234567},\ cv_1 \leftarrow \mathtt{0x89abcdef}$$
$$cv_2 \leftarrow \mathtt{0xfedcba98},\ cv_3 \leftarrow \mathtt{0x76543210}$$

After all 64 steps are computed, $\mathrm{MD5}_c$ computes

$$cv'_0 \leftarrow cv_0 + Q_{60},\ cv'_1 \leftarrow cv_1 + Q_{63},\ cv'_2 \leftarrow cv_2 + Q_{62},\ cv'_3 \leftarrow cv_3 + Q_{61}$$

and outputs $CV' \leftarrow cv'_0 \parallel cv'_1 \parallel cv'_2 \parallel cv'_3$.

Because of their importance later, we repeat some of our notation and terminology: for each message *block*, $\mathrm{MD5}_c$ has four *rounds*, each of which computes 16 *step values* (for a total of 64).

4 High-Level Overview

Define δ_0 as $(0,0,0,0,2^{31},0,0,0,0,0,0,2^{15},0,0,2^{31},0)$ and δ_1 as $(0,0,0,0,2^{31},0,0,0,0,0,0,-2^{15},0,0,2^{31},0)$. Let $M = (M_0, M_1)$ be a 1024-bit string such that $|M_0| = |M_1| = 512$. For any such M let $M'_0 = M_0 + \delta_0$, $M'_1 = M_1 + \delta_1$ and $M' = (M'_0, M'_1)$ where addition is done component-wise modulo 2^{32}.

The Wang attacks describe a way of efficiently finding 1024-bit strings M such that $\mathrm{MD5}(M) = \mathrm{MD5}(M')$. They do this by tracking the differences in the step values during the computation of $\mathrm{MD5}(M)$ and $\mathrm{MD5}(M')$. Formally, let Q_i denote the output of the i-th round of the MD5 compression function upon input M and let Q'_i denote the output of the i-th round of MD5 upon input M'. Then [16] supplies 128 values (64 for the first block and 64 for the second block) a_i, $0 \le i < 128$ such that if their methods find an M such that $\mathrm{MD5}(M) = \mathrm{MD5}(M')$, then $Q'_i - Q_i = a_i$ for all Q_i computed during the computation of $\mathrm{MD5}_c(M_0)$ and $\mathrm{MD5}_c(M'_0)$ and $Q'_i - Q_i = a_{i+64}$ for all Q_i computed during the computation of $\mathrm{MD5}_c(M_1)$ and $\mathrm{MD5}_c(M'_1)$. We will call the values $Q'_i - Q_i$ *differentials.* The a_i are the correct or prescribed differentials. Additionally, four extra values are given in [16] that specify the differentials for the intermediate chaining values, or the outputs of $\mathrm{MD5}_c(M_0)$ and $\mathrm{MD5}_c(M'_0)$.

It is not described in [16] or elsewhere how they chose the values for a_i, but in the next subsection we conjecture some ideas on their derivation. Regardless, Wang et al. detail methods for efficiently finding such M by determining conditions on the Q_i such that if those conditions are satisfied then the differentials hold with high probability ([16] mistakenly labels the conditions as 'sufficient'). Very little information is given in [16] as to how the conditions on the Q_i are obtained, but an excellent analysis is given by Hawkes, Paddon, and Rose in [6].

Wang's method for finding an M of the correct form can be described in pseudocode as the following:

Algorithm Find_Collision
while collision_found is false **do**:

1. Use random seeds and deterministic methods to find M which satisfies most conditions on Q_i
2. Compute all Q_i and Q'_i to check to see if differentials are correct
3. **if** (rest_of_differentials_hold) **then** collision_found $\leftarrow$ true
   else collision_found $\leftarrow$ false

x	y	z	$\Delta x \Rightarrow \Delta F$	$\Delta y \Rightarrow \Delta F$	$\Delta z \Rightarrow \Delta F$	x	y	z	$\Delta x \Rightarrow \Delta G$	$\Delta y \Rightarrow \Delta G$	$\Delta z \Rightarrow \Delta G$
0	0	0			$\surd$	0	0	0		$\surd$	
0	0	1	$\surd$		$\surd$	0	0	1	$\surd$		
0	1	0	$\surd$		$\surd$	0	1	0		$\surd$	$\surd$
0	1	1			$\surd$	0	1	1	$\surd$		$\surd$
1	0	0		$\surd$		1	0	0		$\surd$	$\surd$
1	0	1	$\surd$	$\surd$		1	0	1	$\surd$		$\surd$
1	1	0	$\surd$	$\surd$		1	1	0		$\surd$	
1	1	1		$\surd$		1	1	1	$\surd$		

Fig. 1. Output differences for $F = \Phi_i$, $0 \le i < 16$ and $G = \Phi_i$, $16 \le i < 32$

end do
return M

We also note here that the above pseudocode is actually done once for each block of M. First a 512-bit block M_0 is found that satisfies all first-block differentials, then block M_1 is found.

4.1 Finding the differentials and conditions

Generating Message Differentials. The derivation of the message and step value differentials used by Wang remains unexplained. We attempt here to conjecture how these were derived, although we stress that this is pure speculation and guesswork.

We begin by noting the following three things:

- The Φ function for round three is just the bitwise XOR of the inputs and is therefore linear - any change in one of the inputs necessarily changes the output in the same bits (formally, for any six 32-bit unsigned integers u, v, w, x, y, z, $H(x \oplus u, y \oplus v, z \oplus w) = H(u, v, w) \oplus H(x, y, z)$). On a related note, as can be seen in figure 1, Φ_i for $0 \le i < 32$ has some 'absorbing' properties. That is, it is common that bit changes in the input do not change the output.
- The differential is 0 for the last few step values in round 2 and the first few step values for round 3.
- The differential is 2^{31} for almost all step values in rounds 3 and 4.

The full version of this paper contains a detailed description of how this difference in bit 31 is propagated in round three through the introduction of differences via the message words and careful manipulation of certain properties of addition modulo 2^{32} and the function H. Basically, addition of 2^{31} modulo 2^{32} operates the same as XOR. For now, however, let us note that it appears that the message differentials were chosen for this exact reason - propagating a single bit difference through most of round three.

The above analysis leads us to believe the following course of action was used in determining the message and step value differentials:

- Assume that whatever message differences are introduced in the first and second rounds can be absorbed by the Φ_i functions so that there are no differences in the step values used in the first step of round 3.
- Pick message differences so that the difference in bit 31 cascades through the step values. This involves:
  - Picking blocks in the initial message expansion m_a, m_b, m_c, such that $m_a = w_i$, $m_b = w_{i+1}$, $m_c = w_{i+3}$, $32 \leq i < 45$.
  - Let the differential be $m'_b = m_b + 2^{31}$, $m'_c = m_c + 2^{31}$ and $m'_a = m_a + 2^{31-s_i}$ where s_i is the shift value for round i.
- Find a differential path through the first and second rounds, using the message differentials chosen above, so that the difference for the last four step values in round 2 is zero.
- Find sufficient conditions on the step values to guarantee the differential path (the work done in [6] is an excellent resource on this step).
- For the above step try to minimize 2nd round conditions to avoid complicated multi-message modification techniques.

This third to last step is still surrounded in mystery, but one can see that by the properties of the Φ_i functions for rounds 1 and 2 that the task is possible. Although the step update function for MD4 and RIPEMD is different than that of MD5, the Wang et al. attacks [15] on those functions support the above analysis. That is, there is no difference in the step values for the last few steps of round 2 and the message differentials appear to have been chosen to minimize differences in round three by exploiting the linearity of bit 31. Again, we stress that this analysis is guesswork and we eagerly await a full exposition by the authors of [15, 16].

Fulfilling the Conditions. Conditions on Q_i are conditions on the individual bits of Q_i. For example, for the first block near-collision of MD5 to guarantee the differential they require that the 8-th least significant bit of Q_4 is zero. There are a total of 290 conditions on the round values for the first block attack, and there a total of 310 conditions for step values in the second block.

However, most of these conditions occur in the first and second rounds. This is important because during the first round, one can easily change M so that all the conditions are satisfied because at that point one has complete control over M and any changes do not affect prior computation. The Chinese team denoted these types of changes as “single-message modifications” or “single-step modifications.” We will adopt and use the former terminology. Some round two conditions may also be corrected by other methods, which we will refer to as “multi-message modifications,” but these methods are considerably more complicated because one has to be sure, because of the initial message expansion, that changes to M do not affect the computation of earlier rounds.

We present efficient methods [9, 16] which satisfy all but 30 conditions for the first block, and all but 24 conditions for the second block. The remaining conditions are satisfied in a probabilistic manner. On the assumption that each condition is satisfied with probability 1/2, an expected 2^{30} (2^{24}, resp) messages need to be generated before a message M is found which satisfies all the first (second, resp) block conditions. This estimate is actually a tad low, because it does not account for the fact that the conditions on the Q_i are necessary, but not sufficient, for the step differentials to hold, even in the later rounds where the differentials cannot be satisfied deterministically.

5 The Dirty Details

In the full version [1] this section is intended as a detailed step-by-step guide to writing code that implements the MD5 attacks. Here, however, we focus only on some detailed examples of the method known as multi-message modification. It is this method which is perhaps the most unexplained and crucial step of the Chinese methods. We give a general overview of the method and go through details for the first block multi-message modifications based on Klima's paper [9]. In the appendix we go over some new, more complex methods in detail. A more general and comprehensive approach to this technique is presented in [3].

5.1 Multi-message modification

One of the key ideas in the Chinese papers is that of multi-message modification. This is where after the satisfaction of all first-round conditions has occurred, one may alter several message blocks together to satisfy second round conditions while leaving all first-round conditions satisfied. Despite the importance of these methods for decreasing the time complexity of the attack, the description in [15, 16] is either completely omitted or brief and truncated. We seek here to fully explain the mystery of multi-message modification techniques by covering the general ideas behind the method and then walking through a few examples in detail in the appendix.

General Idea. In [16] the method of multi-message modification is given, almost entirely, in a table similar to the following:

			Modify m_i	$a^{new}, b^{new}, c^{new}, d^{new}$
1	m_1	12	$m_1 \leftarrow m_1 + 2^{26}$	d_1^{new}, a_1, b_0, c_0
2	m_2	17	$m_2 \leftarrow ((c_1 - d_1^{new}) \ggg 17) - c_0 - \Phi_2(d_1^{new}, a_1, b_0) - y_2$	c_1, d_1^{new}, a_1, b_0
3	m_3	22	$m_3 \leftarrow ((b_1 - c_1) \ggg 22) - b_0 - \Phi_3(c_1, d_1^{new}, a_1) - y_3$	b_1, c_1, d_1^{new}, a_1
4	m_4	7	$m_4 \leftarrow ((a_2 - b_1) \ggg 7) - a_1 - \Phi_4(b_1, c_1, d_1^{new}) - y_4$	a_2, b_1, c_1, d_1^{new}
5	m_5	12	$m_5 \leftarrow ((d_2 - a_2) \ggg 12) - d_1^{new} - \Phi_5(a_2, b_1, c_1) - y_5$	d_2, a_2, b_1, c_1

The table is a guide to correcting the condition on $Q_{16,31}$, or $a_{5,32}$ in the notation from [16]. The condition is that this bit must be 0. The first column denotes the step number. The second column and third columns denote the message word and the shift value, respectively, used in the computation of the step value. The column under the heading "Modify m_i" details the update needed to correct the step value or message word in that step. The last column lists updates to step variables, if any, after the modification for that step.

How does this all work? Let's walk through the table. Although not shown in the table, the shift value for round 16 is 5. Therefore, the addition of 2^{26} to m_1 has the net effect of adding 2^{31} to $Q_{16,31}$, which corrects for the condition in question. However, this change to m_1 also changes the value of a step value computed earlier: $Q_1(= d_1)$. Therefore we must recompute d_1 with the new value of m_1 to obtain d_1^{new} (this is not explicitly shown in the above table, but we will come to this in a bit). The other rows of the table detail how to assimilate the changes in d_1 so that none of the other step values are changed (but the message bits are). Note that we can still change other message bits because in step 16 only one message block has been used to compute more than one step value. Namely, m_1. At the end of the process, $m_1, m_2, m_3, m_4, m_5, Q_1$, and Q_{16} have been changed, but all other step values and message bits remain the same. The change in Q_{16} was to remedy the incorrect condition, and the other changes were necessary to absorb the changes to m_1 and Q_1.

Furthermore, in the paper by Wang et al. discussing their attack on MD4, the table denotes that to update d_1 to d_1^{new} all one needs to do is add 2^{26} shifted by the appropriate amount (in this case 12, so that $d_1^{new} = d_1 + 2^6$). This does not always produce the correct value because shifting and carry expansion do not commute. The safest way to compute d_1^{new} is to just re-do the step value computation. A complete table with the updated computation is given below.

			Modify m_i	$a^{new}, b^{new}, c^{new}, d^{new}$
1	m_1	12	$m_1^{new} \leftarrow m_1 + 2^{26}$ $d_1^{new} \leftarrow a_1 + ((\Phi_1(a_1, b_0, c_0) + d_0 + y_1 + m_1^{new}) \lll 12)$	d_1^{new}, a_1, b_0, c_0
2	m_2	17	$m_2^{new} \leftarrow ((c_1 - d_1^{new}) \ggg 17) - c_0 - \Phi_2(d_1^{new}, a_1, b_0) - y_2$	c_1, d_1^{new}, a_1, b_0
3	m_3	22	$m_3^{new} \leftarrow ((b_1 - c_1) \ggg 22) - b_0 - \Phi_3(c_1, d_1^{new}, a_1) - y_3$	b_1, c_1, d_1^{new}, a_1
4	m_4	7	$m_4^{new} \leftarrow ((a_2 - b_1) \ggg 7) - a_1 - \Phi_4(b_1, c_1, d_1^{new}) - y_4$	a_2, b_1, c_1, d_1^{new}
5	m_5	12	$m_5^{new} \leftarrow ((d_2 - a_2) \ggg 12) - d_1^{new} - \Phi_5(a_2, b_1, c_1) - y_5$	d_2, a_2, b_1, c_1

So this is the gist of multi-message modification, but this simple trick does not handle all cases, and unfortunately the details to some of the trickier modifications are not to be found in the Chinese papers. In the appendix we go through an example of a slightly more complex multi-message modification, in addition to attempting to explain the motivation for each step in the method. We hope that by doing so the reader gains a deeper understanding of the (as yet more-or-less unexplained) method.

5.2 1st block multi-message modification

Here we present our methods for finding the first block pair, based on the methods found in [9, 16]. Before detailing our new methods for satisfying three extra conditions, we review and correct the collision-finding pseudocode in Klima's paper [9].

1st Block Collision-Finding Program Outline. Klima is able to satisfy four extra conditions from [16] through some clever probabilistic multi-message modifications. The following outline is nearly identical to that which is presented in Klima's full paper [9]. There are a couple of mistakes in Klima's multi-message modification methods as presented in his paper, however. A few of the steps are out of order and some crucial steps are omitted. Here is how the code should look (using our notation with shifted indices):

1. We choose $Q_{2:15}$ fulfilling conditions.
2. We compute $m_{6:15}$: For i going from 6 to 15 do

$$m_i \leftarrow ((Q_i - Q_{i-1}) \ggg s_i) - F(Q_{i-1}, Q_{i-2}, Q_{i-3}) - Q_{i-4} - y_i$$

3. We change Q_{16} until conditions $Q_{16:18}$ are fulfilled. Sometimes this is not possible (because the values of $Q_{12}, Q_{13}, Q_{14}, Q_{15}$ do not allow the conditions on Q_{17} and Q_{18} to hold), and it becomes necessary to change $Q_{2:15}$.

$$Q_{17} \leftarrow Q_{16} + ((G(Q_{16}, Q_{15}, Q_{14}) + Q_{13} + m_6 + y_{17}) \lll s_{17})$$
$$Q_{18} \leftarrow Q_{17} + ((G(Q_{17}, Q_{16}, Q_{15}) + Q_{14} + m_{11} + y_{18}) \lll s_{18})$$

4. All conditions $Q_{2:18}$ are fulfilled now. Moreover, we have free value m_0.
5. We choose Q_{19} arbitrarily, but fulfilling the one condition for it. Then we compute m_0:

$$m_0 \leftarrow ((Q_{19} - Q_{18}) \ggg s_{19}) - G(Q_{18}, Q_{17}, Q_{16}) - Q_{15} - y_{19}$$

6. Compute Q_0 from new value of m_0:

$$Q_0 \leftarrow Q_{-1} + ((F(Q_{-1}, Q_{-2}, Q_{-3}) + Q_{-4} + m_0 + y_0) \lll s_0)$$

7. Compute m_1:

$$m_1 \leftarrow ((Q_{16} - Q_{15}) \ggg s_{16}) - G(Q_{15}, Q_{14}, Q_{13}) - Q_{12} - y_{16}$$

8. Compute Q_1 from new values of m_1, Q_0:

$$Q_1 \leftarrow Q_0 + ((F(Q_0, Q_{-1}, Q_{-2}) + Q_{-3} + m_1 + y_1) \lll s_1)$$

9. Compute $m_{2:5}$: For i going from 2 to 5 do

$$m_i \leftarrow ((Q_i - Q_{i-1}) \ggg s_i) - F(Q_{i-1}, Q_{i-2}, Q_{i-3}) - Q_{i-4} - y_i$$

For step 3, we chose to satisfy the conditions on $Q_{16:18}$ probabilistically, by simply randomly selecting Q_{16} and checking to see whether the other conditions were satisfied. There are only 9 conditions for these three chaining variables, so this can be done quickly. Sometimes no selection of Q_{16} will satisfy the conditions, so in this case our code simply begins anew by randomly selecting another $Q_{2:15}$ such that the first round conditions are satisfied.[2]

[2] We implement this by setting a reasonable upper limit on the number of random selections of Q_{16} which are chosen and tested.

After step 9, we continue the computation, checking to see if the remaining conditions are satisfied (each condition is expected to be satisfied with probability near 1/2, so we expect to iterate over the above pseudocode 2^{30} times before we find a suitable first block pair). If a condition isn't satisfied, then we have to choose a new message. We do this efficiently by iterating over all possible 2^{31} values of Q_{19} in step 5 (simply incrementing Q_{19} after each failed attempt is the fastest way). If we exhaust all possible values for Q_{19} without finding a suitable message, we return to step 3 and select another value for Q_{16}. In this manner we avoid significant unnecessary computation.

Performance. We ran code based on the work done by [13], modified with our extra methods, to find the first block 80 times run on a desktop 3.0 GHz processor. Overall, full two-block collisions were found, on average, in under 5 minutes. This is a dramatic improvement over the timings given by Klima, even after correcting for discrepancies in hardware.

6 A New Method

In our research we found the following optimizing heuristic, which was verified experimentally but not analytically:

> Relying on the fact that the step update function is not very random, we can attempt to identify patterns in the step values which tend to yield solutions. Using this knowledge, we narrow our search space to use only step values which fall within these patterns.

An Example. The methods presented in the appendix provide an analytic method to satisfy conditions on Q_{20} with probability near 15/16 (approximately 94% of the time). Although the analytic solution works, it nearly doubles the computation over the main iterative loop, thus weakening its positive impact on the running time of our collision-finding program. However, we were also able to obtain an equally or perhaps more effective (satisfied conditions around 97% of the time on tests) method for satisfying conditions on Q_{20}.

The pattern is simple. While iterating through values of Q_{19}, as in step 5 of the pseudocode in section 5.2, there are distinct patterns in which values of Q_{19} automatically satisfy conditions on Q_{20}. That is, ignoring our new multi-message modifications for Q_{20}, we tried to identify which values of Q_{19} led to conditions on Q_{20} being satisfied. We found that values of Q_{19} which satisfied conditions on Q_{20} often occurred sequentially in blocks of 128 followed by 128 consecutive values of Q_{19} which *didn't* satisfy conditions on Q_{20}. There were exceptions to this pattern, but the correlation was strong enough to reduce the time complexity. This suggests a new method for satisfying conditions on Q_{20} that requires much less computation:

- While iterating over values of Q_{19}, check to see if the conditions on Q_{20} are satisfied. If not, add 127 to Q_{19} and continue.

Restricting the values of Q_{19} in this manner yields an algorithm for which around 97% of all used values of Q_{19} satisfy conditions on Q_{20}, with the additional benefit that very little overhead is needed compared to the other multi-message modification method.

Several other examples are discussed in the full version of this paper[1].

The Method. We have not attempted to systematically identify and define this approach within our own work. It was merely that through a casual analysis of data patterns observed during coding, we noticed this phenomenon. Nonetheless, it seems that we used the following rough methodology:

– Record values for intermediate step values as well as result (Were all differentials satisfied with these values? If not, how many were satisfied?). Do this for many random choices of M.
– Attempt to find simple patterns in these step values which will yield a good heuristic.

This procedure can have broad or narrow scope. With the above example, we looked at consecutive values of Q_{19} and checked only one condition on Q_{20}. Broadening the scope may yield results, or it may decrease the chances that simple patterns will be easy to find.

The above technique seems possible to automate so that no human interaction is necessary and we believe this is a possible avenue for future research. We suspect that artificial intelligence techniques could be especially useful with this sort of analysis. The main drawback to this method is that one might not be able to easily understand *why* these patterns of data exist. In general, it seems preferable to do as much analysis as possible, but it seems likely that an automated tool to detect these kinds of patterns may be used with great success after analysis becomes too cumbersome or fruitless.

7 The Full Version

We encourage the interested reader to look at our full version [1], which can be found at `http://www.cs.colorado.edu/~jrblack/papers.html`. This version of the paper contains the following additional content:

– Full, complete, updated tables of the 1st and 2nd block conditions and differentials that use our notation.
– Many more details for those wishing to implement the attacks themselves, correcting errors in earlier papers. This includes discussion of the single-message modification technique.
– A detailed step-by-step guide to the 2nd block multi-message modification methods.
– More examples of the technique presented in section 6.
– An in-depth presentation of additional new multi-message modifications for the 1st block.
– A more detailed explanation of the derivation of Wang's differentials.
– Discussion on the performance of various methods.

Acknowledgements

Many thanks to Matt Robshaw and the anonymous FSE reviewers for their comments and to Vincent Rijmen for his work organizing the event and helping coordinate travel. John Black's and Martin Cochran's work was supported by NSF CAREER-0240000 and NSF-0524118. Trevor Highland's work was supported by NSF REU-0244168; his travel to FSE was in part supported by the IACR.

References

1. Black, J., Cochran, M., and Highland, T. A study of the MD5 attacks: Insights and improvements (full version). Manuscript available at `http://www.cs.colorado.edu/~jrblack/papers.html`.
2. Damgård, I. A design principle for hash functions. In *CRYPTO* (1989), vol. 435 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 416–427.
3. Daum, M. Cryptanalysis of hash functions of the MD4 family. Dissertation available at `http://www.cits.rub.de/imperia/md/content/magnus/dissmd4.pdf`.
4. den Boer, B., and Bosselaers, A. Collisions for the compression function of MD5. In *EUROCRYPT* (1993), vol. 765 of *Lecture Notes in Computer Science*, Springer, pp. 293–304.
5. Dobbertin, H. Cryptanalysis of MD5 compress. Presented at the rump session of EUROCRYPT '96.
6. Hawkes, P., Paddon, M., and Rose, G. G. Musings on the Wang et al. MD5 collision, October 2004. See `http://eprint.iacr.org/2004/264`.
7. Klima, V. Tunnels in hash functions: MD5 collisions within a minute. See `http://eprint.iacr.org/2006/105`.
8. Klima, V. Finding MD5 collisions: A toy for a notebook, March 2005. See `http://eprint.iacr.org/2005/075`.
9. Klima, V. Finding MD5 collisions on a notebook PC using multi-message modifications. In *International Scientific Conference Security and Protection of Information* (May 2005).
10. Merkle, R. C. One way hash functions and DES. In *CRYPTO* (1989), vol. 435 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 428–446.
11. Rivest, R. The MD5 message-digest algorithm. *RFC 1321* (April 1992).
12. Rogaway, P., and Shrimpton, T. Cryptographic hash-function basics: Definitions, implications and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption (FSE 2004)* (2004), vol. 3017 of *Lecture Notes in Computer Science*, Springer, pp. 371–388.
13. Stach, P., and Liu, V. MD5 collision generation. Code available at `http://www.stachliu.com/collisions.html`.
14. Stevens, M. HashClash. See `http://www.win.tue.nl/hashclash/`.
15. Wang, X., Lai, X., Feng, D., Chen, H., and Yu, X. Cryptanalysis of the hash functions MD4 and RIPEMD. In *EUROCRYPT* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.
16. Wang, X., and Yu, H. How to break MD5 and other hash functions. In *EUROCRYPT* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer, pp. 19–35.

A New Multi-Message Modification Methods

We now cover the details of our methods which reduce the overall complexity of the attack to an expected 2^{30} MD5 computations. In subsection 5.1 we covered the basic idea behind multi-message modifications and went over a simple example. However, not all second-round conditions can be handled as easily. In previous papers [15, 16], these more complicated methods are not described at all. Therefore our approach will be to walk through our techniques in detail, attempting to explain our methodology at each step so that the reader gains not only an understanding of our techniques, but hopefully insight into the general technique of multi-message modifications as well.

A.1 New multi-message modifications for correcting $Q_{20,17}$ and $Q_{20,31}$.

These modifications take into account all the modifications that Klima has done to correct the conditions on $Q_{0:19}$. These methods satisfy the two conditions on Q_{20} or a_6 (Wang's notation).

Outline of Method. We set up a few conditions on $Q_{16:19}$ so that flipping a couple of bits of Q_{18} and Q_{19} does not affect earlier computations but with high probability satisfies the two conditions on Q_{20}. For example, bit 31 of Q_{20} must be set to 0. Let's say it is 1. Note how Q_{20} is computed:

$$Q_{20} \leftarrow Q_{19} + ((G(Q_{19}, Q_{18}, Q_{17}) + Q_{16} + m_5 + y_{20}) \lll 5)$$

We set conditions on $Q_{17:19}$ so that by default the value of the 26th bit of $G(Q_{19}, Q_{18}, Q_{17})$ is 0, but that if we flip the 26th bits of both Q_{19} and Q_{18} then the value of the 26th bit $G(Q_{19}, Q_{18}, Q_{17})$ changes to 1. To derive such conditions, one has to look at how the function G is computed, but it can easily be verified that if the 26th bits of Q_{18} and Q_{19} are 0, then the 26th bit of $G(Q_{19}, Q_{18}, Q_{17})$ will be zero, and if the 26th bits of Q_{19} and Q_{18} are flipped to 1, then the 26th bit of $G(Q_{19}, Q_{18}, Q_{17})$ will also be flipped. Flipping the 26th bit of $G(Q_{19}, Q_{18}, Q_{17})$ in this manner has the net effect of adding $2^{31} + 2^{26}$ to the value of Q_{20} because we have added 2^{26} to Q_{19}, which occurs twice in the computation of Q_{20} (once in the computation of $G()$ and once by addition to $T_{19} \lll 5$). Adding 2^{31} flips the most significant bit of Q_{20}, like we wanted, and the addition of 2^{26}, which we cannot really avoid, will only re-flip the most significant bit of Q_{20} if the next 5 most significant bits of Q_{20} were originally set, which occurs with probability 1/32.

At this point the observant reader may ask "Why did we have to flip the 26th bits of both Q_{19} and Q_{18}?". "Why not just flip the 26th bit of Q_{19}?" Here's why: Remember back in Klima's code how m_0 was computed:

$$m_0 \leftarrow ((Q_{19} - Q_{18}) \ggg s_{19}) - G(Q_{18}, Q_{17}, Q_{16}) - Q_{15} - y_{19}$$

If we just changed Q_{19}, we would have to re-compute m_0, which would affect the computation of m_5 a couple of steps later, and Q_{20} would likewise be affected.

In fact, changing m_0 by one bit in this way can change m_5 by a bunch of bits, so we must be very careful so we don't have to modify it for our methods to work. By changing both Q_{19} and Q_{18} in the same way, the changes cancel each other out and m_0 is not changed, so long as $G(Q_{18}, Q_{17}, Q_{16})$ is not affected by these changes. It can easily be verified that requiring the condition $Q_{16,26} = 0$ satisfies this goal (1 more condition). It is important to note that these added conditions do not significantly affect the performance of the overall code because they are satisfied in step 3 of Klima's code. Instead of 9 conditions to probabilistically satisfy in step 3, we now have 11, which is still tiny in comparison to the overall runtime.

Okay. So we've sneakily changed Q_{19} and Q_{18} so that m_0 is unaffected and a condition on Q_{20} is fulfilled with high probability. Now we have to fix everything else. We recompute the value of m_{11} from the new value of Q_{18} by the following:

$$m_{11} \leftarrow ((Q_{18} - Q_{17}) \ggg s_{18}) - G(Q_{17}, Q_{16}, Q_{15}) - Q_{14} - y_{18}$$

And we now have to recompute Q_{11} from m_{11}.

$$Q_{11} \leftarrow Q_{10} + ((F(Q_{10}, Q_9, Q_8) - Q_7 - y_{11}) \lll s_{11})$$

Luckily the changes we made to m_{11} don't affect any of the 15 conditions on Q_{11}, so long as bit 2 of Q_{11} is originally set to 0 (so that we don't have to worry about carries). So we add this condition to the list - again, it is fulfilled "for free" by the single-message modification methods presented in the Wang papers (or by the fact that Q_{11} is initially arbitrarily chosen in the Klima paper).

The only thing left to do is to recompute $m_{12:15}$ to absorb the changes in Q_{11}. This can be done without changing any of the other Q variables by simply recomputing $m_{12:15}$ as we did earlier:

$$\begin{aligned}
m_{12} &\leftarrow ((Q_{12} - Q_{11}) \ggg s_{12}) - F(Q_{11}, Q_{10}, Q_9) - Q_8 - y_{12} \\
m_{13} &\leftarrow ((Q_{13} - Q_{12}) \ggg s_{13}) - F(Q_{12}, Q_{11}, Q_{10}) - Q_9 - y_{13} \\
m_{14} &\leftarrow ((Q_{14} - Q_{13}) \ggg s_{14}) - F(Q_{13}, Q_{12}, Q_{11}) - Q_{10} - y_{14} \\
m_{15} &\leftarrow ((Q_{15} - Q_{14}) \ggg s_{15}) - F(Q_{14}, Q_{13}, Q_{12}) - Q_{11} - y_{15}
\end{aligned}$$

That's it. At the end of everything we have changed Q_{19} and Q_{18} so that one condition on Q_{20} has been changed with probability 31/32, m_{11} and Q_{11} have been changed, but without affecting the conditions on Q_{11}, and m_{12-15} have been changed to absorb the changes in Q_{11} so that no other Q values are affected.

The exact same method can be used to correct the condition on the 17th bit of Q_{20} (just shift all bit values above by 14). There are a total of 8 new conditions that this method requires, but they are all more or less "free."

It is possible that the above methods fail to correct the specified conditions, but the probability that this happens is bounded above by $1/32 + 1/32 = 1/16$.

After each iteration, our code goes back to starting values for $m_{11:15}$, Q_{11}, and Q_{18}, because we need the correct bits of Q_{11} and Q_{18} to be set so that flipping them to satisfy Q_{20} can occur safely.