A Study of the MD5 Attacks: Insights and Improvements

J. Black [*] M. Cochran [*] T. Highland [†]

March 3, 2006

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. [14]. 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.

[*] University of Colorado at Boulder, USA E-mail: jrblack@cs.colorado.edu, Martin.Cochran@colorado.edu WWW: www.cs.colorado.edu/∼jrblack/, ucsu.colorado.edu/∼cochranm

[†] University of Texas at Austin, USA E-mail: trevor.highland@gmail.com

Contents

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]

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 the 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 [13, 14]. 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 this talk and the publication of the team's papers [13, 14], much interest was generated in finding the methods used by the Chinese researchers, and several papers were published on the subject [6–8]. Unfortunately, some key details of the attacks are omitted from the EUROCRYPT papers, and there are several discrepancies between the analysis done in [6, 8] 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, 13, 14], 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.

[1]For a more complete discussion of hash function security definitions, see [11].

- 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 mistakes made by previous authors in their published analyses, using 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 [8] 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 [8] improved the attack so that an expected 2^{33} and 2^{24} MD5 computations are needed to find the first and second, respectively, message block pairs, 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.

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, 8]. 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 $s \parallel t$ 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 [13, 14] captures this type of information by the following notation. Let x be in $[0, 2^{31} - 1]$. Then $x' = x[a_1, a_2, \dots, a_n, -b_1, -b_2, \dots, -b_m]$ 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 $i \in \{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m\}$. The complete differential tables in the appendix 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 [10].

MD5 is a hash function in the Merkle-Damgård paradigm [3, 9], where the security of the hash function reduces to the security of its compression function. The MD5 compression function, which we denote as 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, MD5$_c$: $\{0,1\}^{128} \times \{0,1\}^{512} \to \{0,1\}^{128}$. Let $H_0 \in \{0,1\}^{128}$ and let $M = (M_0, M_1, \dots, M_k)$ for some $k \geq 0$ and $|M_i| \in \{0,1\}^{512}$ for $0 \leq i \leq k$. Then MD5(M) is computed as follows. Let $H_{i+1} = \text{MD5}_c(H_i, M_i)$ for $0 \leq i \leq k$. MD5(M) is defined as H_{k+1}.

3.1 The compression function 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:

$$T_i \leftarrow \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) + Q_{i-4} + w_i + y_j$$

$$Q_i \leftarrow Q_{i-1} + (T_i \lll s_i)$$

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 [10]. 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:

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

$Q_{-1}, \dots, Q_{-4}$ 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,

$$cv_0 \leftarrow \texttt{0x01234567},\ cv_1 \leftarrow \texttt{0x89abcdef}$$

$$cv_2 \leftarrow \texttt{0xfedcba98},\ cv_3 \leftarrow \texttt{0x76543210}$$

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 [14] 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 [14] 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 [14] 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 ([14] mistakenly labels the conditions as 'sufficient'). Very little information is given in [14] as to how the conditions on the Q_i are obtained from the given differentials, 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 ← true
      **else** collision_found ← false
**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.

x	y	z	$\Delta x \Rightarrow \Delta F$	$\Delta y \Rightarrow \Delta F$	$\Delta z \Rightarrow \Delta F$
0	0	0			√
0	0	1	√		√
0	1	0	√		√
0	1	1			√
1	0	0		√	
1	0	1	√	√	
1	1	0	√	√	
1	1	1		√	

x	y	z	$\Delta x \Rightarrow \Delta G$	$\Delta y \Rightarrow \Delta G$	$\Delta z \Rightarrow \Delta G$
0	0	0		√	
0	0	1	√		
0	1	0		√	√
0	1	1	√		√
1	0	0		√	√
1	0	1	√		√
1	1	0		√	
1	1	1	√		

Figure 1: Output differences for $F = \Phi_i$, $0 \leq i < 16$ and $G = \Phi_i$, $16 \leq i < 32$

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 $0 \leq i < 32$ 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.

Before continuing, we digress slightly to answer the following question: How is this difference in bit 31 propagated through round 3? The last few steps of round two have differential of zero and the first bit difference introduced (by way of the differential in M) in round 3 is in step 34:

$$Q'_{34} \leftarrow Q_{33} + ((H(Q_{33}, Q_{32}, Q_{31}) + Q_{30} + y_{34} + m_{11} + 2^{15}) \lll 16)$$

which implies that [2]

$$Q'_{34} = Q_{34} + (2^{15} \lll 16) = Q_{34} + 2^{31} = Q_{34}[31].$$

In step 35, another bit difference is introduced because $m'_{14} = m_{14} + 2^{31}$:

$$Q'_{35} \leftarrow Q'_{34} + ((H(Q'_{34}, Q_{33}, Q_{32}) + Q_{31} + y_{35} + m_{14} + 2^{31}) \lll 23)$$

Substituting $Q_{34} + 2^{31}$ for Q'_{34} we obtain the following.

$$\begin{aligned} Q'_{35} &\leftarrow 2^{31} + Q_{34} + ((H(Q_{34} + 2^{31}, Q_{33}, Q_{32}) + Q_{31} + y_{35} + m_{14} + 2^{31}) \lll 23) \\ &= 2^{31} + Q_{34} + ((H(Q_{34}, Q_{33}, Q_{32}) + 2^{31} + Q_{31} + y_{35} + m_{14} + 2^{31}) \lll 23) \\ &= 2^{31} + Q_{34} + ((H(Q_{34}, Q_{33}, Q_{32}) + Q_{31} + y_{35} + m_{14}) \lll 23) = Q_{35} + 2^{31} = Q_{35}[31] \end{aligned}$$

[2]This computation does not always hold because shifting and carry expansion do not commute. In [14], this is codified in the condition $\phi_{34,32} = 0$. We will assume here that the computation holds (or that Wang's condition is satisfied).

In step 36, no difference is introduced by the message word used. The step value is computed as follows.

$$\begin{aligned} & Q'_{36} \leftarrow Q'_{35} + ((H(Q'_{35}, Q'_{34}, Q_{33}) + Q_{32} + y_{36} + m_1) \lll 4) \\ = \quad & 2^{31} + Q_{35} + ((H(Q_{35} + 2^{31}, Q_{34} + 2^{31}, Q_{33}) + Q_{32} + y_{36} + m_1) \lll 4) \\ = \quad & 2^{31} + Q_{35} + ((H(Q_{35}, Q_{34}, Q_{33}) + (2^{31} \oplus 2^{31}) + Q_{32} + y_{36} + m_1) \lll 4) \\ = \quad & 2^{31} + Q_{35} + ((H(Q_{35}, Q_{34}, Q_{33}) + Q_{32} + y_{36} + m_1) \lll 4) = Q_{36} + 2^{31} = Q_{36}[31] \end{aligned}$$

Another bit difference is introduced by the message word in step 37.

$$\begin{aligned} & Q'_{37} \leftarrow Q'_{36} + ((H(Q'_{36}, Q'_{35}, Q'_{34}) + Q_{33} + y_{37} + m_4 + 2^{31}) \lll 11) \\ = \quad & 2^{31} + Q_{36} + ((H(Q_{36} + 2^{31}, Q_{35} + 2^{31}, Q_{34} + 2^{31}) + Q_{33} + y_{37} + m_4 + 2^{31}) \lll 11) \\ = \quad & 2^{31} + Q_{36} + ((H(Q_{36}, Q_{35}, Q_{34}) + 2^{32} + 2^{31} + Q_{33} + y_{37} + m_4 + 2^{31}) \lll 11) \\ = \quad & 2^{31} + Q_{36} + ((H(Q_{36}, Q_{35}, Q_{34}) + Q_{33} + y_{37} + m_4) \lll 11) = Q_{37} + 2^{31} = Q_{37}[31] \end{aligned}$$

One can then easily verify that if no further bit differences are introduced via the message words, for step values Q_i in round 3 with $i \geq 38$, $Q'_i = Q_i + 2^{31}$. Thus, the bit 31 cascades down the step values without further conditions.[3] Using the linearity of H and addition of 2^{31} modulo 2^{32} is a clever way to minimize the differences in the step values in round 3, where the Φ function does not have the difference absorbing properties of the Φ functions of rounds 1 and 2.

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 [13] 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 [13, 14].

[3]This may have been inspired by [5], as is mentioned obliquely in the introduction in [14]. However, Dobbertin's message differential has hamming weight of only one, so other bit differences are introduced in round 3.

x	y	z	$\Delta x \Rightarrow \Delta H$	$\Delta y \Rightarrow \Delta H$	$\Delta z \Rightarrow \Delta H$
0	0	0	$\surd$	$\surd$	$\surd$
0	0	1	$\surd$	$\surd$	$\surd$
0	1	0	$\surd$	$\surd$	$\surd$
0	1	1	$\surd$	$\surd$	$\surd$
1	0	0	$\surd$	$\surd$	$\surd$
1	0	1	$\surd$	$\surd$	$\surd$
1	1	0	$\surd$	$\surd$	$\surd$
1	1	1	$\surd$	$\surd$	$\surd$

x	y	z	$\Delta x \Rightarrow \Delta I$	$\Delta y \Rightarrow \Delta I$	$\Delta z \Rightarrow \Delta I$
0	0	0		$\surd$	$\surd$
0	0	1	$\surd$	$\surd$	$\surd$
0	1	0		$\surd$	$\surd$
0	1	1	$\surd$	$\surd$	$\surd$
1	0	0		$\surd$	
1	0	1	$\surd$	$\surd$	
1	1	0		$\surd$	
1	1	1	$\surd$	$\surd$	

Figure 2: Output differences for $H = \Phi_i$, $32 \leq i < 48$ and $I = \Phi_i$, $48 \leq i < 64$

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 [8, 14] 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

5.1 Outline of method

Once we are convinced that the attack works in theory, we proceed to implementation. The code we wrote as part of this project uses the following rough outline [8, 14]:

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

1. Select values $Q_{0:15}$ arbitrarily.
2. Modify $Q_{0:15}$ to satisfy all first round conditions and differentials (deterministic).
3. Compute $M_{0:15}$ from these values of $Q_{0:15}$ (deterministic).
4. Satisfy all possible second round conditions and differentials using multi-message modification methods (probabilistic, but computationally insignificant to next step).

```
    5. Check to see if all other conditions and differentials are satisfied (probabilistic).
    6. if (all differentials satisfied) then collision_found ← true
    7. else collision_found ← false
end do
return $M$
```

There are several speed-ups to the above pseudocode. The main one is this: instead of selecting new random values $Q_{0:15}$ every time a condition isn't satisfied, just change M slightly so that all first round and most second round conditions are still satisfied.

5.2 "Sufficient" conditions

The conditions presented in [14] as sufficient for all first-round differentials to hold, are not. Thus, it is necessary to satisfy some of the first round differentials probabilisitically; this step is computationally insignificant compared to fulfilling the second-, third-, and fourth-round conditions and does not affect the overall runtime. In practice, this means that our code randomly chooses M, performs the first round single-message modifications, then checks to see if the differentials are satisfied. If so, great - the code proceeds to the second round multi-message modifications. If not, a new random M is chosen and the process is repeated.

5.3 Single-message modification

Single-message modification is the process by which a message M is modified such that all of the first round conditions hold. The term originates with the Wang papers on MD4 and MD5 [13, 14] in which the same method is also referred to as "single-step modification." It is probably named so because to correct the conditions on any one Q_i, $0 \leq i < 16$, only one message block m_i needs to be modified.

The method of single-message modification we present here is slightly different from that given in the Wang et al. papers, for two reasons. One is that there are some details omitted in the Wang papers (perhaps for simplicity's sake) on the method of single-message modification; if one uses those papers as a guide, inevitably the code will produce incorrect and/or otherwise useless output because some of their instructions are based on the incorrect assumption that shifting and carry expansion commute. We also feel that the methods we present are easier to initially understand.

The idea behind single-message modification is simple. Select the step value so that all conditions hold and recompute the message word from the chosen step value. We present some pseudocode to perform step (2) from the program outline in subsection 5.1. For i going from 0 to 15 do the following:

1. Change Q_i to satisfy conditions by simple bit-flipping.
2. Calculate m_i.

$$m_i \leftarrow ((Q_i - Q_{i-1}) \ggg s_i) - T_i - Q_{i-4} - \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \tag{1}$$

   Note that this is just simple algebraic manipulation of the step update function.

At the end of this method all step values are correctly computed from the changed message words and all first round conditions are satisfied. If we are using the original set of conditions, the first round differentials may not be satisfied. As noted in the previous subsection, we satisfy these differentials probabilistically and therefore may need to run the single-message modifications over many choices of M before all first conditions and differentials hold. Although not deterministic, this method has time complexity vastly less than that of the overall collision-finding program.

5.4 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 [13, 14] 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 (new) examples in detail.

General Idea. In [14] 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 [14]. 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 next section 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.

5.5 1st block multi-message modification

Here we present our methods for finding the first block pair, based on the methods found in [8, 14]. Before detailing our new methods for satisfying three extra conditions, we review and correct the collision-finding pseudocode in Klima's paper [8].

1st Block Collision-Finding Program Outline. Klima is able to satisfy four extra conditions from [14] through some clever probabilistic multi-message modifications. The following outline is nearly identical to that which is presented in Klima's full paper [8]. 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.[4]

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.

6 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 5.4 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 [13, 14], 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.

6.1 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:

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

[4]We implement this by setting a reasonable upper limit on the number of random selections of Q_{16} which are chosen and tested before we select new values for $Q_{2:15}$.

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:

$$m_{12} \leftarrow ((Q_{12} - Q_{11}) \ggg s_{12}) - F(Q_{11}, Q_{10}, Q_9) - Q_8 - y_{12}$$

$$m_{13} \leftarrow ((Q_{13} - Q_{12}) \ggg s_{13}) - F(Q_{12}, Q_{11}, Q_{10}) - Q_9 - y_{13}$$

$$m_{14} \leftarrow ((Q_{14} - Q_{13}) \ggg s_{14}) - F(Q_{13}, Q_{12}, Q_{11}) - Q_{10} - y_{14}$$

$$m_{15} \leftarrow ((Q_{15} - Q_{14}) \ggg s_{15}) - F(Q_{14}, Q_{13}, Q_{12}) - Q_{11} - y_{15}$$

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.

New Modifications for Correcting the Condition on Q_{21}. When used with Klima's methods, the above modifications reduce the number of probabilistically-satisfied conditions to 31. We now detail another multi message modification to reduce that number to 30.

The method we use here is based on the multi message modifications explained above to correct for the conditions on Q_{16}. Essentially, we will make a change to one of the message words to correct the condition and then change the other necessary message words and step values in order to 'absorb' the changes. Unfortunately, this method is not quite as straightforward as the method to correct conditions on Q_{16} as detailed in subsection 5.4.

The first reason for the added complexity is this: at step 21, we have used a number of message words in the second round. Namely, $m_1, m_6, m_{11}, m_0, m_5$, and m_{10}. Any modification methods must not change these message words, otherwise earlier second-round computations will be affected. Our methods get around this by setting up conditions on first-round step values so that we use the properties of the Φ function to avoid changing these message words.

The other reason we can't use the same method as for correcting conditions on Q_{16} is that in this case the methods affect conditions on earlier step values. Specifically, here is the problem we encounter if we try to use the same method: we are trying to correct the condition $Q_{21,31} = 0$. The shift value for round 21 is 9, so to correct the condition we can change m_{10} so that $m_{10}^{new} \leftarrow m_{10} + 2^{22}$. However, m_{10} is used in the computation of Q_{10}, a round for which the shift value is 17. Thus, when we update the value of Q_{10}, we essentially update the value $Q_{10}^{new} \leftarrow Q_{10} + 2^{7}$. This is a problem because there is a condition on $Q_{10,7}$ which we change. Furthermore, in order to absorb the changes to Q_{10} we would almost certainly have to modify m_{11}, which would affect earlier round-two computations. So it appears we have to do something different.

The trick in this case is to instead fix the condition by manipulating the values of bit 22 of Q_7, Q_8, Q_9, and Q_{10} so that we can modify m_{10} without modifying any other bit values of Q_{10} and without modifying m_{11}.

Here is the method:

- Add the following first-round conditions: $Q_{7,22} = 0, Q_{8,22} = 1, Q_{9,22} = 0$ and $Q_{10,22} = 0$.
- If the condition on $Q_{21,31}$ is not satisfied, do the following:
  - Flip bit 22 of Q_9 to 1 (denote change as $Q_9^{new} \leftarrow Q_9 + 2^{22}$).
  - Recompute m_9 to reflect this change

$$m_9^{new} \leftarrow ((Q_9^{new} - Q_8) \ggg s_9) - F(Q_8, Q_7, Q_6) - Q_5 - y_9$$

  - Recompute m_{10}

$$m_{10}^{new} \leftarrow ((Q_{10} - Q_9^{new}) \ggg 17) - F(Q_9^{new}, Q_8, Q_7) - Q_6 - y_{10}$$

  - Recompute m_{12} and m_{13}

$$m_{12}^{new} \leftarrow ((Q_{12} - Q_{11}) \ggg s_{12}) - F(Q_{11}, Q_{10}, Q_9^{new}) - Q_8 - y_{12}$$
$$m_{13}^{new} \leftarrow ((Q_{13} - Q_{12}) \ggg s_{13}) - F(Q_{12}, Q_{11}, Q_{10}) - Q_9^{new} - y_{13}$$

  - Recompute Q_{21}

$$Q_{21}^{new} \leftarrow Q_{20} + ((G(Q_{20}, Q_{19}, Q_{18}) + Q_{17} + y_{21} + m_{10}^{new}) \lll s_{21})$$

This process looks similar to the technique present earlier in subsection 5.4, but the following question naturally arises: Why doesn't m_{11} need to be modified? The trick is looking at the values of F in these rounds before and after the change to $Q_{9,22}$. We're not really concerned with how changing $Q_{9,22}$ affects m_9, m_{12}, or m_{13}, but we would like m_{10}^{new} to look something like $m_{10} + 2^{22}$ and we would like m_{11} to not be affected *at all.* In the computation of m_{10}^{new},

$$m_{10}^{new} \leftarrow ((Q_{10} - Q_9^{new}) \ggg 17) - F(Q_9^{new}, Q_8, Q_7) - Q_6 - y_{10}$$

the change to $Q_{9,22}$ can affect m_{10}^{new} twice. Note that for the values on bits 22 of Q_9, Q_8, Q_7, bit 22 of $F(Q_9, Q_8, Q_7)$ is 0. However, bit 22 of $F(Q_9^{new}, Q_8, Q_7)$ is 1.

So, substituting $Q_9 + 2^{22}$ for Q_9^{new} we obtain that

$$\begin{aligned} m_{10}^{new} &\leftarrow ((Q_{10} - Q_9 - 2^{22}) \ggg 17) - F(Q_9, Q_8, Q_7) - 2^{22} - Q_6 - y_{10} \\ &= ((Q_{10} - Q_9) \ggg s_{10}) - 2^5 - F(Q_9, Q_8, Q_u) - 2^{22} - Q_6 - y_{10} \\ &= m_{10} - 2^{22} - 2^5 \end{aligned}$$

This change to m_{10} will almost certainly have the desired effect on Q_{21} because $Q_{21}^{new} = Q_{21} - 2^{31} - 2^{14}$. The factor of 2^{14} will only affect bit 31 of Q_{21} if bits 14 through 30 of Q_{21} are zero, which occurs with probability $1/2^{17}$.

Now we just have to make sure that this change to $Q_{9,22}$ doesn't affect m_{11}. Let's look at how m_{11} might be affected.

$$m_{11} \leftarrow ((Q_{11} - Q_{10}) \ggg s_{11}) - F(Q_{10}, Q_9^{new}, Q_8) - Q_7 - y_{11}$$

It is clear that m_{11} will only change if $F(Q_{10}, Q_9, Q_8) \neq F(Q_{10}, Q_9^{new}, Q_8)$. We only have to check if bit 22 changes. Because bit 22 of Q_{10} is 0, bit 22 of $F(Q_{10}, Q_9, Q_8)$ and $F(Q_{10}, Q_9^{new}, Q_8)$ are both equal to $Q_{8,22} = 1$. Thus m_{11} is not changed.

Performance. We ran code based on the work done by [12], modified with our extra methods, to find the first block 80 times run on a desktop 3.0 GHz processor. The total time to find all 80 blocks was just under ten hours seven minutes, giving an average time per block of 455 seconds. We also found a second block, one for each first block found with corresponding IVs, in a total time of just under four hours 23 minutes, giving an average time of 197 seconds per block found. Overall, full two-block collisions were found, on average, in under 11 minutes. This is a dramatic improvement over the timings given by Klima, even after correcting for discrepancies in hardware.

6.2 2nd block multi-message modification

The following is a method that is very similar to one described in the Klima paper with complexity 2^{24}. The complexity turns out to be closer to 2^{26} due to the insufficiency of the conditions on the step values.

This method satisfies conditions and differentials through $Q_{0:20}$. The rest of the conditions and differentials are satisfied probabilistically. There are 24 remaining conditions on $Q_{21:63}$, along with conditions on $T_{22,17}$, $T_{34,15}$, and $T_{61,15}$.

1. Randomly select values for $Q_{2:15}$, which satisfy conditions.
2. Compute $m_{6:15}$.
3. Select Q_0 and Q_1 at random until conditions and differentials for $Q_{16:20}$ are satisfied. This involves computing $m_{0:5}$ from these chosen values, then computing $Q_{16:20}$. Sometimes it is necessary to return to step one.
4. Randomly select values for $Q_{7:10}$, which satisfy conditions (single-message modifications).
5. Calculate Q_{11}. (m_{11} is not modified.)
6. If all conditions and differentials through step 20 are not satisfied return to step 4.
7. Cycle though all of the possible bit changes for Q_8 and Q_9 using gray code.[5] note the following:
   - $\Phi_{11} = (Q_{10} \wedge Q_9) \vee (\neg Q_{10} \wedge Q_8)$
   - If $Q_{10,i} = 1$ and $i \in I$ we can modify $Q_{8,i}$ while preserving $x[11]$.
   - $I = \{2, 3, 4, 5, 10, 11, 13, 14, 18, 19, 20, 21, 22, 29, 30\}$
   - If $Q_{10,j} = 0$ and $j \in J$ we can modify $Q_{9,j}$ while preserving $x[11]$.
   - $J = \{2, 3, 4, 5, 10, 11, 20, 21, 22, 27, 28, 29, 30\}$
8. Return to step 4 until a collision has been found.

[5]This is just a method by which only one bit changes with each iteration over the possible values, done for efficiency's sake.

This method modifies $Q_{7:11}$, while leaving m_{11} unmodified. Since m_1, m_6, m_{11}, m_0, m_5 are unchanged, we know the conditions for $Q_{16:20}$ remain satisfied.

Performance for Second Block Multi-Message Modification. At first the second block multi-message modification performed quickly, but was rather inconsistent. Often the program had difficulty completing steps 4-6 quickly. We were able to remedy this by returning to step one after a prescribed number of iterations. Other times the program would simply be unable to find a collision in a reasonable amount of time. After a number of trial runs we realized that a majority of collisions were found when step 7 had been reached less than 4096 times and of those that were found when step seven had been reached more than 4096 times, many were found after step seven had been reached many more than 4096 times. Thus if we detect that step seven has been reached more than 4096 times we select a new random message.

Once our program performed consistently, we performed tests on the second block multi-message modification using a Dell Latitude D610 running at 1.6 GHz. We were able to find 85 second block messages in 4 hours 29 minutes. The mean running time was 3 minutes 9 seconds, with a range from 5 seconds to 10 minutes. This method for second block multi-message modification has close to the same running time as the method used by the Chinese team using a IBM P690. The method we used for second block multi-message modification turns out to be 25-50 times faster than the method used by the Chinese Team, based on the performance difference of the IBM P690 and a 1.6 GHz Pentium M [8].

7 A New Method

At the end of the previous section we discussed how some analysis of trends in the running time of the program to find the second blocks could lead to decreased time complexity. This suggests a general technique to reduce the search space which may be incorporated into the methods already presented.

In the example covered in the previous section, we noted that, for whatever reason, certain sets of values for the step values were more likely to find an acceptable second block than others, and that there is a simple way to test to see if a set of values is not in this desired group. The general technique can be described simply as this, 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.

Another Example. The methods presented in section 6 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 5.5, 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.

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 two conditions 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.

Acknowledgements

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. The authors wish to thank the anonymous FSE 2006 reviewers for their helpful comments.

References

[1] Brassard, G., Ed. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings* (1990), vol. 435 of *Lecture Notes in Computer Science*, Springer.

[2] Cramer, R., Ed. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer.

[3] Damgård, I. A design principle for hash functions. In Brassard [1], pp. 416–427.

[4] den Boer, B., and Bosselaers, A. Collisions for the compressin function of MD5. In *EUROCRYPT* (1993), 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. http://eprint.iacr.org/2004/264.

[7] Klima, V. Finding MD5 collisions: a toy for a notebook, March 2005. Draft available as http://eprint.iacr.org/2005/075.

[8] Klima, V. Finding MD5 collisions on a notebook PC using multi-message modifications. In *International Scientific Conference Security and Protection of Information* (May 2005).

[9] Merkle, R. C. One way hash functions and DES. In Brassard [1], pp. 428–446.

[10] Rivest, R. The MD5 message-digest algorithm. *RFC 1321*, 37 (April 1992).

[11] 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), Lecture Notes in Computer Science, Springer.

[12] Stach, P., and Liu, V. MD5 collision generation. Code available at http://www.stachliu.com/collisions.html.

[13] Wang, X., Lai, X., Feng, D., Chen, H., and Yu, X. Cryptanalysis of the hash functions MD4 and RIPEMD. In Cramer [2], pp. 1–18.

[14] Wang, X., and Yu, H. How to break MD5 and other hash functions. In Cramer [2], pp. 19–35.

A Tables

We have re-created the tables from [13, 14], but using our notation. They can be found in figures 3, 4, 5, and 6. We have left out the "missing" conditions reported in other papers because as of the writing of this paper the extra conditions still do not make the entire set sufficient with regard to guaranteeing the differentials, and the overall runtime is unaffected either way.

```
 0: ................................     32: ................................
 1: ................................     33: ................................
 2: ............0.......0....0......     34: ................................
 3: 1.......0222122222221222220......     35: ................................
 4: 1...1.0.010000000000000000001..1.1     36: ................................
 5: 02220212011111111011110001022021     37: ................................
 6: 00000011111111101111100000100000     38: ................................
 7: 000000011..100010.0.010101000000     39: ................................
 8: 11111011...100000.12111100111101     40: ................................
 9: 01......0..111111.01...001....00     41: ................................
10: 00..........00011200...011....10     42: ................................
11: 00....22....10000001...10.......     43: ................................
12: 01....01....1111111....00...1...     44: ................................
13: 0.0...00....1011111....11...1...     45: ................................
14: 0.1...01........1...........0...     46: ................................
15: 0.1.............................     47: 3...............................
16: 0.............0.2...........2...     48: 3...............................
17: 0.2...........1.................     49: 5...............................
18: 0.............0.................     50: 3...............................
19: 0...............................     51: 3...............................
20: 0.............2.................     52: 3...............................
21: 0...............................     53: 3...............................
22: 0...............................     54: 3...............................
23: 4...............................     55: 3...............................
24: ................................     56: 3...............................
25: ................................     57: 3...............................
26: ................................     58: 3...............................
27: ................................     59: 5.....0.........................
28: ................................     60: 3....01.........................
29: ................................     61: 3...............................
30: ................................     62: 2...............................
31: ................................     63: ................................

aa: ................................     dd: ......0.........................
cc: 2....01.........................     bb: 2....00...................0.....
```

Figure 3: Summary of First Block Conditions. Key - 0: bit equals 0, 1: bit equals 1, 2: bit equals bit of same index from previous step, 3: bit equals bit of same index from two steps ago, 4: bit not equal to bit of same index from previous step, 5: bit not equal to bit of same index from two steps ago

```
 0: 1...010...1.........0.....0.....     32: ................................
 1: 1222110...0222220..21...220..00.     33: ................................
 2: 1011111...011111...01..10112211.     34: 6...............................
 3: 1011101...000100...0022000010002     35: ................................
 4: 010010....101111...0111001010000     36: ................................
 5: 0..0010...10..10...0110001010110     37: ................................
 6: 1..101122.00..012..1111000.....1     38: ................................
 7: 1..001000.11..101.....1111....20     39: ................................
 8: 1..111000.....010..2..0111....01     40: ................................
 9: 1....1111....0111..0..1111....00     41: ................................
10: 1...........21011220..1111....11     42: ................................
11: 12222222....10000001....1.......     43: ................................
12: 00111111....1111111.....0...1...     44: ................................
13: 01000000....1011111.....1...1...     45: ................................
14: 01111101........0...........0...     46: ................................
15: 0.1.............................     47: 3...............................
16: 0.............0.2...........2...     48: 3...............................
17: 0.2...........1.................     49: 5...............................
18: 0.............0.................     50: 3...............................
19: 0...............................     51: 3...............................
20: 0.............2.................     52: 3...............................
21: 0...............................     53: 3...............................
22: 0...............................     54: 3...............................
23: 4...............................     55: 3...............................
24: ................................     56: 3...............................
25: ................................     57: 3...............................
26: ................................     58: 3...............................
27: ................................     59: 5...............................
28: ................................     60: 3.....1.........................
29: ................................     61: 3.....1.........................
30: ................................     62: 3.....1.........................
31: ................................     63: ......1.........................
```

Figure 4: Summary of Second Block Conditions. Key - 0: bit equals 0, 1: bit equals 1, 2: bit equals bit of same index from previous step, 3: bit equals bit of same index from two steps ago, 4: bit not equal to bit of same index from previous step, 5: bit not equal to bit of same index from two steps ago

Step	The output in i-th step for M_0	w_i	s_i	Δw_i	The output in the i-th step for M_0'
3	Q_3 (b_1)	m_3	22		
4	Q_4 (a_2)	m_4	7	2^{31}	$Q_4[7,\dots,22,-23]$
5	Q_5 (d_2)	m_5	12		$Q_5[-6,23,31]$
6	Q_6 (c_2)	m_6	17		$Q_6[6,7,8,9,10,-11,-23,-24,-25,$ $26,27,28,29,30,31,0,1,2,3,4,-5]$
7	Q_7 (b_2)	m_7	22		$Q_7[0,15,-16,17,18,19,-20,-23]$
8	Q_8 (a_3)	m_8	7		$Q_8[-0,1,6,7,-8,-31]$
9	Q_9 (d_3)	m_9	12		$Q_9[-12,13,31]$
10	Q_{10} (c_3)	m_{10}	17		$Q_{10}[30,31]$
11	Q_{11} (b_3)	m_{11}	22	2^{15}	$Q_{11}[7,-8,13,\dots,18,-19,31]$
12	Q_{12} (a_4)	m_{12}	7		$Q_{12}[-24,25,31]$
13	Q_{13} (d_4)	m_{13}	12		$Q_{13}[31]$
14	Q_{14} (c_4)	m_{14}	17	2^{31}	$Q_{14}[3,-15,31]$
15	Q_{15} (b_4)	m_{15}	22		$Q_{15}[-29,31]$
16	Q_{16} (a_5)	m_1	5		$Q_{16}[31]$
17	Q_{17} (d_5)	m_6	9		$Q_{17}[31]$
18	Q_{18} (c_5)	m_{11}	14	2^{15}	$Q_{18}[17,31]$
19	Q_{19} (b_5)	m_0	20		$Q_{19}[31]$
20	Q_{20} (a_6)	m_5	5		$Q_{20}[31]$
21	Q_{21} (d_6)	m_{10}	9		$Q_{21}[31]$
22	Q_{22} (c_6)	m_{15}	14		Q_{22}
23	Q_{23} (b_6)	m_4	20	2^{31}	Q_{23}
24	Q_{24} (a_7)	m_9	5		Q_{24}
25	Q_{25} (d_7)	m_{14}	9	2^{31}	Q_{25}
26	Q_{26} (c_7)	m_3	14		Q_{26}
⋮	⋮	⋮	⋮	⋮	⋮
33	Q_{33} (d_9)	m_8	11		Q_{33}
34	Q_{34} (c_9)	m_{11}	16	2^{15}	$Q_{34}[\pm 31]$
35	Q_{35} (b_9)	m_{14}	23	2^{31}	$Q_{35}[\pm 31]$
36	Q_{36} (a_{10})	m_1	4		$Q_{36}[\pm 31]$
37	Q_{37} (d_{10})	m_4	11	2^{31}	$Q_{37}[\pm 31]$
38	Q_{38} (c_{10})	m_7	16		$Q_{38}[\pm 31]$
⋮	⋮	⋮	⋮	⋮	⋮
44	Q_{44} (a_{12})	m_9	4		$Q_{44}[\pm 31]$
45	Q_{45} (d_{12})	m_{12}	11		$Q_{45}[31]$
46	Q_{46} (c_{12})	m_{15}	16		$Q_{46}[31]$
47	Q_{47} (b_{12})	m_2	23		$Q_{47}[31]$
48	Q_{48} (a_{13})	m_0	6		$Q_{48}[31]$
49	Q_{49} (d_{13})	m_7	10		$Q_{49}[-31]$
50	Q_{50} (c_{13})	m_{14}	15	2^{31}	$Q_{50}[31]$
51	Q_{51} (b_{13})	m_5	21		$Q_{51}[-31]$
⋮	⋮	⋮	⋮	⋮	⋮
57	Q_{57} (d_{15})	m_{15}	10		$Q_{57}[-31]$
58	Q_{58} (c_{15})	m_6	15		$Q_{58}[31]$
59	Q_{59} (b_{15})	m_{13}	21		$Q_{59}[31]$
60	Q_{60} (a_{16})	m_4	6	2^{31}	$Q_{60}[31]$
61	Q_{61} (d_{16})	m_{11}	10	2^{15}	$Q_{61}[25,31]$
62	Q_{62} (c_{16})	m_2	15		$Q_{62}[-25,26,31]$
63	Q_{63} (b_{16})	m_9	21		$Q_{63}[25,31]$
	$Q_{64}=Q_{60}+Q_{-4}$				$Q_{64}[31]$
	$Q_{65}=Q_{61}+Q_{-3}$				$Q_{65}[25,31]$
	$Q_{66}=Q_{62}+Q_{-2}$				$Q_{66}[-25,26,31]$
	$Q_{67}=Q_{63}+Q_{-1}$				$Q_{67}[25,-31]$

Figure 5: The Differential Characteristics for the First Block

Step	The output in i-th step for M_1	w_i	s_i	Δw_i	The output in the i-th step for M_1'
IV					$aa_0[31]$, $dd_0[25,31]$, $cc_0[-25,26,31]$, $bb_0[25,-31]$
0	Q_0 (a_1)	m_0	7		$Q_0[25,-31]$
1	Q_1 (d_1)	m_1	12		$Q_1[5,25,-31]$
2	Q_2 (c_1)	m_2	17		$Q_2[-5,-6,7,-11,12,$ $-16,\ldots,-20,21,-25,\ldots,-29,30,-31]$
3	Q_3 (b_1)	m_3	22		$Q_3[1,2,3,-4,5,-25,26,-31]$
4	Q_4 (a_2)	m_4	7	2^{31}	$Q_4[0,-6,7,8,-9,-10,-11,12,31]$
5	Q_5 (d_2)	m_5	12		$Q_5[16,-17,20,-21,31]$
6	Q_6 (c_2)	m_6	17		$Q_6[6,7,8,-9,27,-28,-31]$
7	Q_7 (b_2)	m_7	22		$Q_7[-15,16,-17,23,24,25,-26,-31]$
8	Q_8 (a_3)	m_8	7		$Q_8[-0,1,-6,-7,-8,9,-31]$
9	Q_9 (d_3)	m_9	12		$Q_9[12,-31]$
10	Q_{10} (c_3)	m_{10}	17		$Q_{10}[-31]$
11	Q_{11} (b_3)	m_{11}	22	-2^{15}	$Q_{11}[-7,13,14,15,16,17,18,-19,-31]$
12	Q_{12} (a_4)	m_{12}	7		$Q_{12}[-24,\ldots,-29,30,31]$
13	Q_{13} (d_4)	m_{13}	12		$Q_{13}[31]$
14	Q_{14} (c_4)	m_{14}	17	2^{31}	$Q_{14}[3,15,31]$
15	Q_{15} (b_4)	m_{15}	22		$Q_{15}[-29,31]$
16	Q_{16} (a_5)	m_1	5		$Q_{16}[31]$
17	Q_{17} (d_5)	m_6	9		$Q_{17}[31]$
18	Q_{18} (c_5)	m_{11}	14	-2^{15}	$Q_{18}[17,31]$
19	Q_{19} (b_5)	m_0	20		$Q_{19}[31]$
20	Q_{20} (a_6)	m_5	5		$Q_{20}[31]$
21	Q_{21} (d_6)	m_{10}	9		$Q_{21}[31]$
22	Q_{22} (c_6)	m_{15}	14		$Q_{22}[31]$
23	Q_{23} (b_6)	m_4	20	2^{31}	$Q_{23}[31]$
24	Q_{24} (a_7)	m_9	5		Q_{24}
25	Q_{25} (d_7)	m_{14}	9	2^{31}	Q_{25}
26	Q_{26} (c_7)	m_3	14		Q_{26}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
33	Q_{33} (d_9)	m_8	11		Q_{33}
34	Q_{34} (c_9)	m_{11}	16	-2^{15}	$Q_{34}[\pm 31]$
35	Q_{35} (b_9)	m_{14}	23	2^{31}	$Q_{35}[\pm 31]$
36	Q_{36} (a_{10})	m_1	4		$Q_{36}[\pm 31]$
37	Q_{37} (d_{10})	m_4	11	2^{31}	$Q_{37}[\pm 31]$
38	Q_{38} (c_{10})	m_7	16		$Q_{38}[\pm 31]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
48	Q_{48} (a_{13})	m_0	6		$Q_{48}[31]$
49	Q_{49} (d_{13})	m_7	10		$Q_{49}[-31]$
50	Q_{50} (c_{13})	m_{14}	15	2^{31}	$Q_{50}[31]$
51	Q_{51} (b_{13})	m_5	21		$Q_{51}[-31]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
58	Q_{58} (c_{15})	m_6	15		$Q_{58}[31]$
59	Q_{59} (b_{15})	m_{13}	21		$Q_{59}[31]$
60	Q_{60} (a_{16})	m_4	6	2^{31}	$Q_{60}[31]$
61	Q_{61} (d_{16})	m_{11}	10	-2^{15}	$Q_{61}[31]$
62	Q_{62} (c_{16})	m_2	15		$Q_{62}[31]$
63	Q_{63} (b_{16})	m_9	21		$Q_{63}[31]$
	$Q_{64}=Q_{60}+Q_{-4}$				Q_{64}
	$Q_{65}=Q_{61}+Q_{-3}$				Q_{65}
	$Q_{66}=Q_{62}+Q_{-2}$				Q_{66}
	$Q_{67}=Q_{63}+Q_{-1}$				Q_{67}

Figure 6: The Differential Characteristics for the Second Block