Chosen-prefix Collisions for MD5 and Applications

Marc Stevens[1], Arjen Lenstra[2] and Benne de Weger[3]

[1] CWI, Amsterdam, The Netherlands
[2] EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland
[3] EiPSI, Faculty of Mathematics and Computer Science, TU Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. We present a novel, automated way to find differential paths for MD5. As an application we have shown how, at an approximate expected cost of 2^{39} calls to the MD5 compression function, for any two chosen message prefixes P and P', suffixes S and S' can be constructed such that the concatenated values $P\|S$ and $P'\|S'$ collide under MD5. The practical attack potential of this construction of *chosen-prefix collisions* is of greater concern than the MD5-collisions that were published before. This is illustrated by a pair of MD5-based X.509 certificates one of which was signed by a commercial Certification Authority (CA) as a legitimate website certificate, while the other one is a certificate for a rogue CA that is entirely under our control (cf. `http://www.win.tue.nl/hashclash/rogue-ca/`). Other examples, such as MD5-colliding executables, are presented as well. More details can be found on `http://www.win.tue.nl/hashclash/ChosenPrefixCollisions/`.

1 Introduction

Cryptographic hash functions. Modern information security methods heavily rely on cryptographic hash functions, functions that map bitstrings of arbitrary length to fixed-length bitstrings in such a way that a number of security related conditions is satisfied. Commonly used cryptographic hash functions are MD5, SHA-1, and SHA-256, mapping their inputs to fixed-lengths outputs of 128, 160, and 256 bits, respectively. We refer to [21] for a description of the general design principle of these hash functions and of the security properties that cryptographic hash functions are supposed to satisfy. One of these properties is *collision resistance*: it should be practically infeasible to find two different inputs that have the same hash value. The collision resistance of MD5 is the focus of this article.

Previous results on collision attacks for MD5. In August 2004 at the rump session of the annual Crypto conference in Santa Barbara, Xiaoyun Wang (cf. [31]) presented a pair of two-block messages that collide under MD5. The details of their attack construction were presented by Wang and Yu in [34]. It describes a

manually found differential path for MD5 and introduces the concept of *near-collision blocks*: a pair of input blocks that results in specifically targeted output differences. It allows computation of a new collision in a few hours of CPU time. Improved versions of these attacks are commented on at the end of this section.

Impact of previous results. Although Wang's random looking *collision blocks* by themselves do not pose any danger, it was shown in [14] and [22] how those original collisions can be used to mislead integrity checking software and replace benign files with malicious versions without detection. Furthermore, it was immediately clear that any value can be used for the IHV (in this paper the chaining variable will be denoted by IHV, for Intermediate Hash Value) at the beginning of the two-block collision search, not just MD5's initial value as in their example collision. This freedom to choose the IHV allowed several authors to use Wang's attack construction for slightly more ulterior purposes, e.g. by inserting both collision blocks in different Postscript documents that collide under MD5 (cf. [7]).

Although none of these developments in the collision spoke in favor of continued usage of MD5, the potential for abuse of these types of collisions was limited. Initially, serious misuse was believed to be prevented by the lack of control over the contents of the collision blocks. In [18] it was shown, however, that for any pair of meaningless data (M, M') a suffix T can easily be found such that both concatenations $M\|T$ and $M'\|T$ are fully meaningful. This allows the following attack construction. First, for any meaningful common prefix P, collision blocks (M, M') may be constructed using Wang's approach such that $P\|M$ and $P\|M'$ collide under MD5. Even though $P\|M$ and $P\|M'$ can be expected to be partially meaningless, an appendage T can subsequently be calculated such that both $P\|M\|T$ and $P\|M'\|T$ are fully meaningful. Furthermore, due to the iterative structure of MD5, they also still collide under MD5. This shows that the argument that Wang's MD5-collisions are mostly harmless because of their lack of structure is in principle invalid. The above attack construction allowed the realization of two different X.509 certificates with identical Distinguished Names and identical MD5-based signatures but different public keys (cf. [18]). Such pairs of certificates theoretically violate the security of the X.509 Public Key Infrastructure, however the limitation to identical Distinguished Names does not allow abuse in practice.

Threatening abuse scenarios did not emerge due to a severe limitation of this collision attack, namely that both colliding messages must have identical IHVs at the beginning of the collision blocks. This requirement is most naturally fulfilled by making the documents identical up to that point. Therefore, we call such collisions *identical-prefix collisions*. In the above example, P would be the identical prefix.

New contributions. The most important contribution of this paper is the removal of the identical prefix condition, a result that we originally presented at Eurocrypt 2007 (cf. [29]), and of which a considerably improved version was presented at Crypto 2009 (cf. [30]). We show how *any* pair of IHVs can be made to collide

under MD5 by appending properly chosen collision blocks. More precisely, we show how, for any two chosen message prefixes P and P', suffixes S and S' can be constructed such that the concatenated values $P\|S$ and $P'\|S'$ collide under MD5. Such collisions will be called *chosen-prefix collisions* (though *different-prefix collisions* would have been appropriate as well). Our attack construction is based on a "birthday" search combined with a novel, automated way to find differential paths for MD5. It has an approximate expected cost of 2^{39} calls to the MD5 compression function. In practical terms, this translates to about a day on a standard quad-core PC per chosen-prefix collision. This notable improvement over the 6 months on thousands of PCs for a single chosen-prefix collision that we reported earlier in the Eurocrypt 2007 paper [29], was triggered by the application presented in the Crypto 2009 paper [30].

Significance and impact of the new contributions. Chosen-prefix collisions have a greater threat potential than identical-prefix ones. Using the diamond construction from Kelsey and Kohno (cf. [15]) along with chosen-prefix collisions, any number of documents of one's choice can be made to collide after extending them with relatively short and innocuously looking appendages that can easily remain hidden to the unsuspicious reader when popular document formats (such as PDF, Postscript, or MS Word) are used. We illustrate this in Section 5.3 with a Nostradamus attack to predict the winner of the 2008 US Presidential elections. Implementing a Herding attack we constructed 12 different but MD5-colliding PDF files, each predicting a different winner. Their common hash serves as commitment to our prediction of the outcome. Our prediction was correct.

Similarly, chosen-prefix collisions can be used to mislead, for instance, download integrity verification of code signing schemes, by appending collision blocks to executables. Details, and how it improves upon previous such constructions, can be found in Section 5.4.

The most convincing application of MD5-collisions, however, would target the core of the Public Key Infrastructure (PKI) and truly undermine its security (cf. Section 4.1 of [29]). The most obvious way to realize this would be by constructing *colliding certificates*, i.e., certificates for which the to-be-signed parts have the same cryptographic hash and therefore the same digital signature. This is undesirable, because the signature of one of the to-be-signed parts, as provided by a Certification Authority (CA), is also a valid signature for the other to-be-signed part. Thus, this gives rise to a pair of certificates, one of which is legitimate, but the other one is a rogue certificate.

Constructing colliding certificates that affect authenticity thereby truly undermining security, seemed to be out of reach, however. As mentioned in the Eurocrpt 2007 paper [29] that introduced chosen-prefix collisions, these collisions allow us to construct two X.509 certificates with different Distinguished Names and different public keys, but identical MD5-based signatures. This improves upon the construction from [18] mentioned above, but is hardly more threatening due to two problems. In the first place, we need full control over the prefixes of the certificates' to-be-signed parts, to be able to calculate the collision blocks. When using a 'real life' CA, however, that CA has final control

over the contents of one of the to-be-signed parts: in particular, it inserts a serial number and a validity period. Furthermore, our construction as in [29] results in 8192-bit RSA moduli, which is quite a bit longer than the 2048-bit upper bound that is enforced by some CAs.

It was pointed out to us by three of our coauthors on the Crypto 2009 follow-up paper [30], that there are circumstances where, with sufficiently high probability, the first of the above two problems can be circumvented. Naively, one would expect that a somewhat more extensive search would suffice to address the remaining problem of reducing the length of the RSA moduli to a generally acceptable 2048 bits. Substantially more extensive improvements to all stages of the original chosen-prefix construction from the Eurocrypt 2007 paper [29] were required, though. This is a nice illustration of scientific progress being driven by practical applications. Furthermore, more computational power had to be brought to bear to deal with the timing restrictions of reliably predicting the CA's contribution. Ultimately this led to the rogue CA certificate mentioned in the abstract, generated in about a day on a cluster of 215 PlayStation 3 game consoles. From the heartwarming industry reception of our rogue CA construction, which is further described in Section 5.2, it must be concluded that we finally managed to present a sufficiently convincing argument to discontinue usage of MD5 for digital signature applications.

Very brief summary of new techniques. The possibility of chosen-prefix collisions was mentioned already in [9, Section 4.2 case 1] and, in the context of SHA-1, in [4] and on `http://www.iaik.tugraz.at/content/research/krypto/sha1/`. This paper is an updated version of the Eurocrypt 2007 paper [29], incorporating the improvements mentioned above and as described in [28] and in the Crypto 2009 paper [30]. The new applications mentioned above have been realized using the following main improvements. In the first place we introduce an extended family of differential paths compared to [29]. Secondly, we use a more powerful birthday search procedure which reduces the overall complexity from 2^{50} to 2^{39} MD5 compression function calls. This procedure also introduces a time-memory trade-off and more flexibility in the birthday search complexity to allow for variability in the expected number of near-collision blocks. Finally, we have a much improved implementation exploiting the wide variety of features of a cluster of PlayStation 3 game consoles.

Outline of article. Section 3 gives a high level overview of our method to construct chosen-prefix collisions. Section 4 presents the method in full detail. The three proof of concept applications are presented in Section 5.

Other improvements of Wang's original collision finding method. Another application of our automated differential path finding method is a speedup of the identical-prefix collision attack by Wang et al. In combination with the idea of tunnels from Klima [17] collisions can be found in 2^{25} MD5 compression function calls, see [28]. Source and binary code for this improvement is available on `http://www.win.tue.nl/hashclash/`). Note that Xie, Liu and Feng [35] used

a different method for identical-prefix collisions, reaching a complexity of 2^{21} MD5 compression function calls, and that in the meantime identical-prefix collisions for MD5 can be found in 2^{16} MD5 compression function calls [30]. This new identical-prefix collision attack is used in Section 4.8 to construct very short chosen-prefix collisions with complexity of about $2^{53.2}$ MD5 compressions, where the collision-causing suffixes are only 596 bits long instead of several thousands of bits.

Summary of old and new results. In Table 1-1 we present a historical overview of the decline in complexity of MD5 and SHA-1 collision finding. For historical interest we include claims that were presented but that have never been published (indicated by "u: ..." in Table 1-1) and a claim that has been withdrawn (indicated by "w: ..." in Table 1-1) It clearly illustrates that attacks against MD5 keep getting better, and that the situation around SHA-1 is unclear. Not reflected in the table is the fact that already in 1993 it was known that there was serious trouble with MD5, based on collisions in its compression function (cf. [2], [8]). We leave any speculation about the future of SHA-1 cryptanalysis to the knowledgeable reader.

Table 1-1. Collision complexities – Historical overview.

	MD5		SHA-1	
year	identical-prefix	chosen-prefix	identical-prefix	chosen-prefix
pre-2004	2^{64} (trivial)	2^{64} (trivial)	2^{80} (trivial)	2^{80} (trivial)
2004	2^{40} [31], [34]			
2005	2^{37} [16]		2^{69} [33]	
			u: 2^{63} [32]	
2006	2^{32} [17], [27]	2^{49} [29]		u: $2^{80-\epsilon}$ [25]
2007	2^{25} [28]	2^{42} [28]	u: 2^{61} [20]	
2008	2^{21} [35]			
2009	2^{16} [30]	2^{39} [30]	w: 2^{52} [19]	

Complexity is given as the number of calls to the relevant compression function (cf. Section 2). The figures are optimized for speed, i.e., for collisions using any number of near-collision blocks. For other collision lengths the complexities may differ.

2 Merkle-Damgård and MD5

In this section we describe the Merkle-Damgård construction in Section 2.1, then we fix some notation in Section 2.2 and give a description of MD5 in Section 2.3.

2.1 Merkle-Damgård

The well known Merkle-Damgård construction describes exactly how to construct a hash function based on a compression function with fixed-size inputs in

an iterative structure as is depicted in Figure 1. Since it has been proven that the hash function is collision resistant if the underlying compression function is collision resistant, the majority of all the currently used hash functions are based on this Merkle-Damgård construction. The construction builds a hash function based on a compression function that takes two fixed-size inputs, namely a chaining value denoted by IHV and a message block, and outputs a new IHV. For instance, MD5's compression function operates on an IHV of bit length 128 and a message block consisting of 512 bits. An input message is first padded with a single 1 bit followed by a number X of 0 bits and lastly the original message length encoded in 64 bits. The number X of 0 bits to be added is defined as the lowest possible number so that the entire padded message bit length is an integer multiple of 512. The padded message is now split into N blocks of size exactly 512 bits. The hash function starts with a fixed public value for IHV_0 called the IV (Initial Value). For each subsequent message block M_i it calls the compression function with the current IHV_i and the message block M_i and stores the output as the new IHV_{i+1}. After all blocks are processed it outputs the last IHV_N after an optional finalization transform.

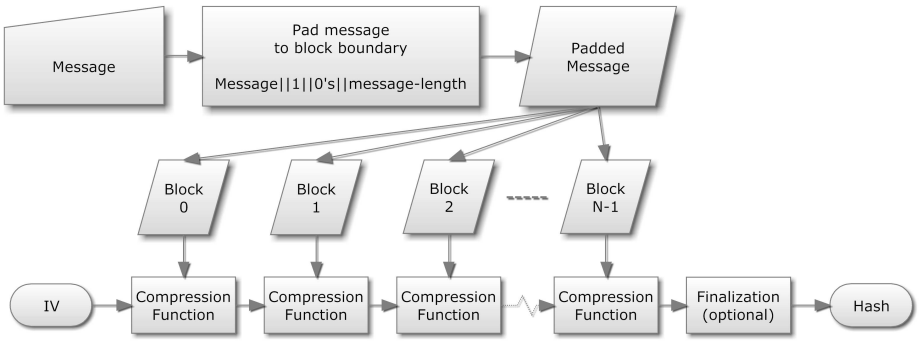

Fig. 1. Merkle-Damgård construction

2.2 Preliminaries

MD5 operates on 32-bit words $(v_{31}v_{30}\dots v_0)$ with $v_i \in \{0,1\}$, that are identified with elements $v = \sum_{i=0}^{31} v_i 2^i$ of $\mathbb{Z}/2^{32}\mathbb{Z}$ (the ring of integers modulo 2^{32}, represented by the set of least non-negative residues $\{0, 1, \dots, 2^{32}-1\}$) and referred to as 32-bit integers. In this paper we switch freely between these representations.

A *binary signed digit representation* (BSDR) for a 32-bit word X is defined as $(k_i)_{i=0}^{31}$, where

$$X = \sum_{i=0}^{31} 2^i k_i, \quad k_i \in \{-1, 0, +1\}.$$

Many different BSDRs exist for any non-zero X. The *weight* of a BSDR is the number of non-zero k_i's. A particularly useful BSDR is the Non-Adjacent Form (NAF), where no two non-zero k_i's are adjacent. The NAF is not unique since we work modulo 2^{32} (making $k_{31} = +1$ equivalent to $k_{31} = -1$), but uniqueness of the NAF can be enforced by choosing $k_{31} \in \{0, +1\}$. Among the BSDRs of an integer, the NAF has minimal weight (cf. [5]). It can easily be computed as $\mathrm{NAF}(n) = (a_i - b_i)_{i=0}^{31}$ where $a_i, b_i \in \{0, 1\}$ such that $\sum_{i=0}^{31} a_i 2^i = n + \lfloor \frac{n}{2} \rfloor \bmod 2^{32}$ and $\sum_{i=0}^{31} b_i 2^i = \lfloor \frac{n}{2} \rfloor$.

Integers are denoted in hexadecimal as, for instance, $\mathtt{1E}_{16}$ and in binary as $\mathtt{00011110}_2$. For bitstrings X and Y we use the following notation:

- $X \wedge Y$ is the bitwise AND of X and Y;
- $X \vee Y$ is the bitwise OR of X and Y;
- $X \oplus Y$ is the bitwise XOR of X and Y;
- $\overline{X}$ is the bitwise complement of X;

for $X, Y \in \mathbb{Z}/2^{32}\mathbb{Z}$:

- $X[i]$ is the i-th bit of the regular binary representation of X;
- $X + Y$ resp. $X - Y$ is the addition resp. subtraction modulo 2^{32};
- $RL(X, n)$ (resp. $RR(X, n)$) is the cyclic left (resp. right) rotation of X by n bit positions:

$$RL(\mathtt{10100100\ldots00000001}_2,\ 5) = \mathtt{100\ldots0000000110100}_2;$$

and for a 32-digit BSDR X:

- $X[\![i]\!]$ is the i-th signed bit of X;
- $RL(X, n)$ (resp. $RR(X, n)$) is the cyclic left (resp. right) rotation of X by n positions.
- $w(X)$ is the weight of X.

For chosen message prefixes P and P' we seek suffixes S and S' such that the messages $P\|S$ and $P'\|S'$ collide under MD5. In this paper any variable X related to the message $P\|S$ or its MD5 calculation, may have a corresponding variable X' related to the message $P'\|S'$ or its MD5 calculation. Furthermore, for such a 'matched' variable $X \in \mathbb{Z}/2^{32}\mathbb{Z}$ we define $\delta X = X' - X$ and $\Delta X = (X'[i] - X[i])_{i=0}^{31}$, which is a BSDR of δX. For a matched variable Z that consist of tuples of 32-bit integers, say $Z = (z_1, z_2, \ldots)$, we define δZ as $(\delta z_1, \delta z_2, \ldots)$.

2.3 Description of MD5

2.3.1 MD5 overview

MD5 follows the Merkle-Damgård construction and works as follows, cf. [26]:

1. *Padding.* Pad the message: first a '1'-bit, next the least number of '0' bits to make the bitlength equal to 448 mod 512, and finally the bitlength of the original unpadded message as a 64-bit little-endian integer. As a result the total bitlength of the padded message is $512N$ for a positive integer N.

2. *Partitioning.* Partition the padded message into N consecutive 512-bit blocks $M_1, M_2, \ldots, M_N$.
3. *Processing.* To hash a message consisting of N blocks, MD5 goes through $N+1$ states IHV_i, for $0 \leq i \leq N$, called the *intermediate hash values*. Each intermediate hash value IHV_i consists of four 32-bit words a_i, b_i, c_i, d_i. For $i = 0$ these are fixed public values:

$$(a_0, b_0, c_0, d_0) = (\texttt{67452301}_{16}, \texttt{EFCDAB89}_{16}, \texttt{98BADCFE}_{16}, \texttt{10325476}_{16}).$$

   For $i = 1, 2, \ldots, N$ intermediate hash value IHV_i is computed using the MD5 compression function described in detail below:

$$\text{IHV}_i = \text{MD5Compress}(\text{IHV}_{i-1}, M_i).$$

4. *Output.* The resulting hash value is the last intermediate hash value IHV_N, expressed as the concatenation of the hexadecimal byte strings of the four words a_N, b_N, c_N, d_N, converted back from their little-endian representation.

2.3.2 MD5 compression function

The input for the compression function $\text{MD5Compress}(\text{IHV}, B)$ consists of an intermediate hash value $\text{IHV} = (a, b, c, d)$ and a 512-bit message block B. The compression function consists of 64 *steps* (numbered 0 to 63), split into four consecutive *rounds* of 16 steps each. Each step t uses modular additions, a left rotation, and a non-linear function f_t, and involves an *Addition Constant* AC_t and a *Rotation Constant* RC_t. These are defined as follows (see also Table A-1 in Appendix A):

$$AC_t = \left\lfloor 2^{32} \left|\sin(t+1)\right| \right\rfloor \quad , \qquad 0 \leq t < 64,$$

$$(RC_t, RC_{t+1}, RC_{t+2}, RC_{t+3}) = \begin{cases} (7, 12, 17, 22) & \text{for } t = 0, 4, 8, 12, \\ (5, 9, 14, 20) & \text{for } t = 16, 20, 24, 28, \\ (4, 11, 16, 23) & \text{for } t = 32, 36, 40, 44, \\ (6, 10, 15, 21) & \text{for } t = 48, 52, 56, 60. \end{cases}$$

The non-linear function f_t depends on the round:

$$f_t(X, Y, Z) = \begin{cases} F(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z) & \text{for } 0 \leq t < 16, \\ G(X, Y, Z) = (Z \wedge X) \oplus (\overline{Z} \wedge Y) & \text{for } 16 \leq t < 32, \\ H(X, Y, Z) = X \oplus Y \oplus Z & \text{for } 32 \leq t < 48, \\ I(X, Y, Z) = Y \oplus (X \vee \overline{Z}) & \text{for } 48 \leq t < 64. \end{cases} \tag{1}$$

The message block B is partitioned into sixteen consecutive 32-bit words m_0, m_1, …, m_{15} (with little-endian byte ordering), and expanded to 64 words W_t,

for $0 \leq t < 64$, of 32 bits each (see also Table A-1 in Appendix A):

$$W_t = \begin{cases} m_t & \text{for } 0 \leq t < 16, \\ m_{(1+5t) \bmod 16} & \text{for } 16 \leq t < 32, \\ m_{(5+3t) \bmod 16} & \text{for } 32 \leq t < 48, \\ m_{(7t) \bmod 16} & \text{for } 48 \leq t < 64. \end{cases}$$

We follow the description of the MD5 compression function from [12] because its 'unrolling' of the cyclic state facilitates the analysis. For each step t the compression function algorithm maintains a working register with 4 state words Q_t, Q_{t-1}, Q_{t-2} and Q_{t-3} and calculates a new state word Q_{t+1}. With $(Q_0, Q_{-1}, Q_{-2}, Q_{-3}) = (b, c, d, a)$, for $t = 0, 1, \ldots, 63$ in succession Q_{t+1} is calculated as follows:

$$\begin{cases} F_t & = f_t(Q_t,\ Q_{t-1},\ Q_{t-2}), \\ T_t & = F_t + Q_{t-3} + AC_t + W_t, \\ R_t & = RL(T_t, RC_t), \\ Q_{t+1} & = Q_t + R_t. \end{cases} \tag{2}$$

After all steps are computed, the resulting state words are added to the intermediate hash value and returned as output:

$$\text{MD5Compress}(\text{IHV}, B) = (a + Q_{61},\ b + Q_{64},\ c + Q_{63},\ d + Q_{62}). \tag{3}$$

3 An overview of chosen-prefix collisions for MD5

Given two arbitrary chosen messages, our purpose is to find appendages such that the extended messages collide under MD5. In this section we give a summary of our method.

Given the two arbitrary messages, we first apply padding to the shorter of the two, if any, to make their lengths equal. This is unavoidable, because Merkle-Damgård strengthening, involving the message length, is applied after the last message bits have been processed. We impose the additional requirement that both resulting messages are a specific number of bits (such as 64 or 96) short of a whole number of blocks. In principle this can be avoided, but it leads to an efficient method that allows relatively easy presentation. All these requirements can easily be met, also in applications with stringent formatting restrictions.

Given this message pair, we modify a suggestion by Xiaoyun Wang (private communication) by finding a pair of k-bit values that, when appended to the last incomplete message blocks, results in a specific form of difference vector between the IHVs after application of the MD5 compression function to the extended message pair. Finding the k-bit appendages can be done using a birthday search procedure.

The specific form of difference vector between the IHVs that is aimed for during the birthday search is such that the difference pattern can relatively easily be removed compared to the more or less random difference pattern one may

expect given two arbitrarily chosen prefixes. Removing the difference pattern is done by further appending to the messages a sequence of near-collision blocks. Each pair of near-collision blocks targets a specific subpattern of the remaining differences. For each such subpattern we use an automated, improved version of Wang's original approach to construct a new differential path, as described in detail in Section 4 below, and subsequently use the differential path to construct a pair of near-collision blocks. Appending those blocks to the two messages results in a new difference vector between the new IHVs from which the targeted subpattern has been eliminated compared to the previous difference vector. The construction continues as long as differences exist. The above process is depicted in Figure 2.

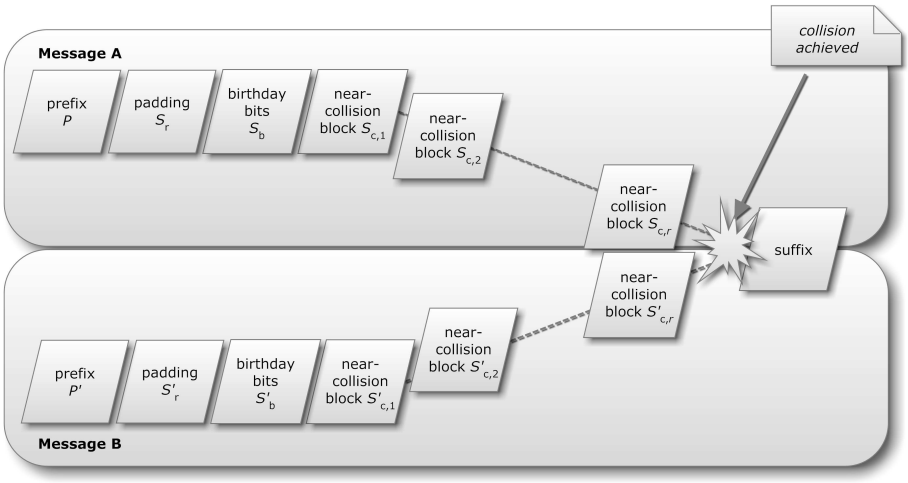

Fig. 2. Chosen-prefix collision overview

How the various steps involved in this construction are carried out and how their parameters are tuned depends on what needs to be optimized. Extensive birthday searching can be used to create difference patterns that require a small number of pairs of near-collision blocks. When combined with a properly chosen large family of differential paths, a single pair of near-collison blocks suffices to complete the collision right away. However, it may make the actual near-collision block construction quite challenging, which leads to the intuitively expected result that finding very short chosen-prefix collision-causing appendages is relatively costly. On the other side of the spectrum, fast birthday searching combined with a smaller family of differential paths leads to the need for many successive pairs of near-collision blocks, each of which can quickly be found: if one is willing to accept long chosen-prefix collision-causing appendages, the overall construction can be done quite fast. Between the two extremes almost everything can be varied: number of near-collision blocks, their construction time

given the differential path, time to find the full differential path, birthday search time and space requirements, etc., leading to a very wide variety of 'optimal' choices.

The next section contains the details of the various steps in this process, and how the steps are best glued together depending on the circumstances. Application scenarios that impose different restrictions on the chosen-prefix collisions are then presented in Section 5.

4 Chosen-prefix collision construction for MD5, details

In Section 4.1 an outline of the chosen-prefix collision construction is given; this includes a short description of the birthday search referred to in Section 3, the further details of which can be found in Section 4.2. Differential paths are introduced in Section 4.3 and Sections 4.4.1 through 4.4.6 describe how to construct partial and full differential paths. Collision finding — the search for actual near-collision blocks that satisfy a given differential path — is treated in Section 4.5 and an optional differential path preprocessing step to improve collision finding is presented in Section 4.5.3. Section 4.6 gives some details of our implementations and the complexity analysis is treated in Section 4.7. Finally, we present a practical chosen-prefix collision attack using a single near-collision block in Section 4.8.

4.1 Outline of the collision construction

A chosen-prefix collision for MD5 is a pair of messages M and M' that consist of arbitrarily chosen prefixes P and P' (not necessarily of the same length), together with constructed suffixes S and S', such that $M = P\|S$, $M' = P'\|S'$, and $\mathrm{MD5}(M) = \mathrm{MD5}(M')$. The suffixes consist of three parts: padding bitstrings $S_\mathrm{r}, S'_\mathrm{r}$, followed by 'birthday' bitstrings $S_\mathrm{b}, S'_\mathrm{b}$ both of bitlength $64 + k$, where $0 \leq k \leq 32$ is a parameter, followed by bitstrings $S_\mathrm{c}, S'_\mathrm{c}$ each consisting of a sequence of near-collision blocks. The padding bitstrings are chosen such that the bitlengths of $P\|S_\mathrm{r}$ and $P'\|S'_\mathrm{r}$ are both equal to $512n - 64 - k$ for a positive integer n. The birthday bitstrings $S_\mathrm{b}, S'_\mathrm{b}$ are determined in such a way that application of the MD5 compression function to $P\|S_\mathrm{r}\|S_\mathrm{b}$ and $P'\|S'_\mathrm{r}\|S'_\mathrm{b}$ results in IHV_n and IHV'_n, respectively and in the notation from Section 2.3.1, for which $\delta\mathrm{IHV}_n$ has a certain desirable property that is explained below.

The idea is to eliminate the difference $\delta\mathrm{IHV}_n$ in r consecutive steps, for some r, by writing $S_\mathrm{c} = S_{\mathrm{c},1}\|S_{\mathrm{c},2}\|\ldots\|S_{\mathrm{c},r}$ and $S'_\mathrm{c} = S'_{\mathrm{c},1}\|S'_{\mathrm{c},2}\|\ldots\|S'_{\mathrm{c},r}$ for r pairs of near-collision blocks $(S_{\mathrm{c},j}, S'_{\mathrm{c},j})$ for $1 \leq j \leq r$. For each pair of near-collision blocks $(S_{\mathrm{c},j}, S'_{\mathrm{c},j})$ we need to construct a differential path (see Section 4.3 for an informal definition of this term) such that the difference vector $\delta\mathrm{IHV}_{n+j}$ has lower weight than $\delta\mathrm{IHV}_{n+j-1}$, until after r pairs we have reached $\delta\mathrm{IHV}_{n+r} = (0, 0, 0, 0)$.

Fix some j and let $S_{\mathrm{c},j}$ consist of 32-bit words m_i, for $0 \leq i < 16$. We fix fifteen of the δm_i as 0 and allow only δm_{11} to be $\pm 2^{p-10 \bmod 32}$ with as yet

Table 4-1. Family of partial differential paths using $\delta m_{11} = \pm 2^{p-10 \bmod 32}$.

t	δQ_t	δF_t	δW_t	δT_t	δR_t	RC_t
31	$\mp 2^{p-10 \bmod 32}$					
32	0					
33	0					
34	0	0	$\pm 2^{p-10 \bmod 32}$	0	0	16
$35-60$	0	0	0	0	0	$\cdot$
61	0	0	$\pm 2^{p-10 \bmod 32}$	$\pm 2^{p-10 \bmod 32}$	$\pm 2^p$	10
62	$\pm 2^p$	0	0	0	0	15
63	$\pm 2^p$	0	0	0	0	21
64	$\pm 2^p$ $+ \sum_{\lambda=0}^{w'} s_\lambda 2^{p+21+\lambda \bmod 32}$					

Here $s_0, \ldots, s_{w'} \in \{-1, 0, +1\}$ and $w' = \min(w, 31 - p)$ for a fixed $w \geq 0$.
Interesting values for the parameter w are between 2 and 5.

unspecified p with $0 \leq p < 32$ (note the slight abuse of notation, since we define message block differences without specifying the message blocks themselves). This was suggested by Xiaoyun Wang because with this type of message difference the number of bitconditions over the final two and a half rounds can be kept low, which turns out to be helpful while constructing collisions. For steps $t = 34$ up to $t = 61$ the differential path is fully determined by δm_{11} as illustrated in Table 4-1. The greater variability for the steps not specified in Table 4-1 does not need to be fixed at this point. In the last two steps there is a greater degree of freedom specified by the integer $w \geq 0$ that determines which and how many IHV differences can be eliminated per pair of near-collision blocks. A larger w allows more eliminations by means of additional differential paths. The latter have, however, a smaller chance to be satisfied because they depend on more (and thus less likely) carry propagations in ΔQ_{62} and ΔQ_{63}. This effect contributes to the complexity of finding the near-collision blocks satisfying the differential paths. Varying w therefore leads to a trade-off between fewer near-collision blocks and increased complexity to find them.

This entire construction of the pair of near-collision blocks $(S_{\mathrm{c},j}, S'_{\mathrm{c},j})$ will be done in a fully automated way based on the choice of w and the values of IHV_{n+j-1} and IHV'_{n+j-1} as specified. It follows from equation (3) and the rows for $t \geq 61$ in Table 4-1 that a differential path with $\delta m_{11} = \pm 2^{p-10 \bmod 32}$ would add a tuple

$$\pm \left(0, 2^p + \sum_{\lambda=0}^{w'} s_\lambda 2^{p+21+\lambda \bmod 32}, 2^p, 2^p \right)$$

to $\delta\mathrm{IHV}_{n+j-1}$, with notation as in Table 4-1. This is set forth in more detail below. A sequence of such tuples is too restrictive to eliminate arbitrary $\delta\mathrm{IHV}_n$: although differences in the b component can be handled using a number of near-collision block pairs, only identical differences can be removed from the c and d

components and the a-component differences are not affected at all. We therefore make sure that $\delta\mathrm{IHV}_n$ has the desirable property, as referred to above, that it *can* be eliminated using these tuples. This is done in the birthday search step where birthday bitstrings S_b and S'_b are determined such that $\delta\mathrm{IHV}_n = (0, \delta b, \delta c, \delta c)$ for some δb and δc. A $\delta\mathrm{IHV}_n$ of this form corresponds to a collision $(a, c-d) = (a', c'-d')$ between $\mathrm{IHV}_n = (a, b, c, d)$ and $\mathrm{IHV}'_n = (a', b', c', d')$. With a search space of only 64 bits, such a collision can easily be found. Since the number of near-collision block pairs and the effort required to find them depends in part on the number of bit differences between δb and δc, it may pay off to lower that number at the cost of extending the birthday search space. For instance, for any k with $0 \leq k \leq 32$, a collision $(a, c-d, c-b \bmod 2^k) = (a', c'-d', c'-b' \bmod 2^k)$ with a $(64+k)$-bit search space results in $\delta c - \delta b \equiv 0 \bmod 2^k$ and thus, on average, just $(32-k)/3$ bit differences between δb and δc. Determining such S_b and S'_b can be expected to require on the order of $\sqrt{2\frac{\pi}{2}2^{64+k}} = \sqrt{\pi}2^{32+(k/2)}$ calls to the MD5 compression function. More on the birthday search in Section 4.2.

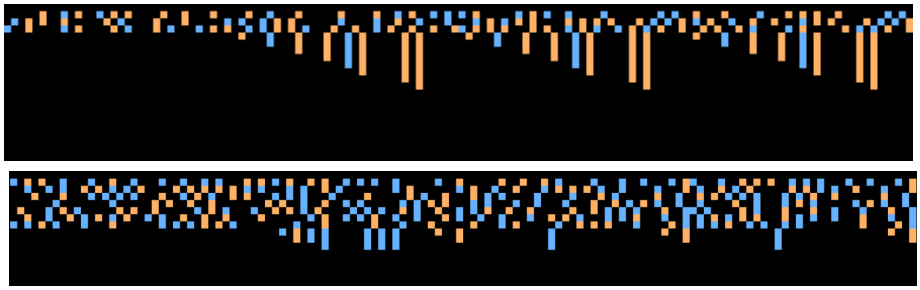

Fig. 3. Above: δIHVs for the colliding certificates with different Distinguished Names. Below: δIHVs for the colliding website certificate and the rogue CA certificate.

In the conference version [29] of this paper we used only the differential paths with $\delta Q_{64} = \pm 2^p$. This forced us to use the harder to satisfy constraint $\delta\mathrm{IHV}_n = (0, \delta c, \delta c, \delta c)$ with a search space consisting of 96 bits and an expected birthda search cost of $\sqrt{\pi}2^{48}$ MD5 compression function calls, which is the same as choosing $k = 32$ above. The top part of Figure 3 visualizes the corresponding construction of near-collision blocks for our colliding certificate example from [29]. The horizontal lines represent the NAFs of $\delta\mathrm{IHV}_i$ for $i = 0, 1, \ldots, 21$. In this example $P \| S_\mathrm{r} \| S_\mathrm{b}$ consists of 4 blocks (i.e., $n = 4$), so that three identical groups of bit differences are left at $i = 4$. As shown in Figure 3 each of these groups consists of 8 bits. The bits in each group of eight are eliminated simultaneously with the corresponding bits in the other groups of eight by 8 pairs of near-collision blocks, so that at $i = 12$ a full collision is reached. The blocks after that are identical for the two messages, so that the collision is retained.

Algorithm 4-1 Construction of pairs of near-collision blocks.

Given n-block $P\|S_{\mathrm{r}}\|S_{\mathrm{b}}$ and $P'\|S'_{\mathrm{r}}\|S'_{\mathrm{b}}$, the corresponding resulting IHV_n and IHV'_n, and a value for w, a pair of bitstrings $S_{\mathrm{c}}, S'_{\mathrm{c}}$ is constructed consisting of sequences of near-collision blocks such that $M = P\|S_{\mathrm{r}}\|S_{\mathrm{b}}\|S_{\mathrm{c}}$ and $M' = P'\|S'_{\mathrm{r}}\|S'_{\mathrm{b}}\|S'_{\mathrm{c}}$ satisfy $\mathrm{MD5}(M) = \mathrm{MD5}(M')$. This is done by performing in succession steps 1, 2 and 3 below.

1. Let $j = 0$ and let S_{c} and S'_{c} be two bitstrings of length zero.
2. Let $\delta\mathrm{IHV}_{n+j} = (0, \delta b, \delta c, \delta c)$. If $\delta c = 0$ then proceed to step 3. Let $(k_i)_{i=0}^{31} = \mathrm{NAF}(\delta c)$ and $(l_i)_{i=0}^{31} = \mathrm{NAF}(\delta b - \delta c)$. Choose any i for which $k_i \neq 0$ and let $w' = \min(w, 31 - i)$. Perform steps (a) through (f):
   (a) Increase j by 1.
   (b) Let $\delta S_{\mathrm{c},j} = (\delta m_0, \delta m_1, \ldots, \delta m_{15})$ with $\delta m_{11} = -k_i 2^{i-10 \bmod 32}$ and $\delta m_t = 0$ for $0 \leq t < 16$ and $t \neq 11$.
   (c) Given $\delta\mathrm{IHV}_{n+j-1} = \mathrm{IHV}'_{n+j-1} - \mathrm{IHV}_{n+j-1}$ and $\delta S_{\mathrm{c},j}$, construct a few differential paths based on Table 4-1 with

$$\delta Q_{61} = 0, \quad \delta Q_{64} = -k_i 2^i - \sum_{\lambda = i+21 \bmod 32}^{i+21+w' \bmod 32} l_\lambda 2^\lambda, \quad \delta Q_{63} = \delta Q_{62} = -k_i 2^i.$$

   How this is done is described in Sections 4.3 and 4.4.
   (d) Find message blocks $S_{\mathrm{c},j}$ and $S'_{\mathrm{c},j} = S_{\mathrm{c},j} + \delta S_{\mathrm{c},j}$ that satisfy one of the constructed differential paths. How this is done is described in Section 4.5. If proper message blocks cannot be found, back up to step (c) to find more differential paths.
   (e) Compute $\mathrm{IHV}_{n+j} = \mathrm{MD5Compress}(\mathrm{IHV}_{n+j-1}, S_{\mathrm{c},j})$, $\mathrm{IHV}'_{n+j} = \mathrm{MD5Compress}(\mathrm{IHV}'_{n+j-1}, S'_{\mathrm{c},j})$, and append $S_{\mathrm{c},j}$ and $S'_{\mathrm{c},j}$ to S_{c} and S'_{c}, respectively.
   (f) Repeat step 2
3. Let $\delta\mathrm{IHV}_{n+j} = (0, \delta\hat{b}, 0, 0)$. If $\delta\hat{b} = 0$ then terminate. Let $(l_i)_{i=0}^{31} = \mathrm{NAF}(\delta\hat{b})$. Choose i such that $l_i \neq 0$ and $i - 21 \bmod 32$ is minimal and let $w' = \min(w, 31 - (i - 21 \bmod 32))$. Perform steps (a) through (e) as above with $\delta m_{11} = 2^{i-31 \bmod 32}$ as opposed to $\delta m_{11} = -k_i 2^{i-10 \bmod 32}$ in step (b) and in steps (c) and (d) with

$$\delta Q_{61} = 0, \quad \delta Q_{64} = 2^{i-21 \bmod 32} - \sum_{\lambda = i}^{i+w' \bmod 32} l_\lambda 2^\lambda, \quad \delta Q_{63} = \delta Q_{62} = 2^{i-21 \bmod 32}.$$

   Perform steps (a) through (e) again with $\delta m_{11} = -2^{i-31 \bmod 32}$ in step (b) and

$$\delta Q_{61} = 0, \quad \delta Q_{64} = \delta Q_{63} = \delta Q_{62} = -2^{i-21 \bmod 32}$$

   in steps (c) and (d). Repeat step 3.

The lower part of Figure 3 visualizes the improved construction as used for the example from Section 5.2. In that example $P\|S_{\mathrm{r}}\|S_{\mathrm{b}}$ consists of 8 blocks (i.e., $n = 8$) and results in a difference vector $\delta\mathrm{IHV}_n$ of the form $(0, \delta b, \delta c, \delta c)$. For any reasonable w, e.g., $w = 2$, we then select a sequence of differential paths from the family given in Table 4-1 to eliminate $\delta\mathrm{IHV}_n$. For this example, 3 pairs of near-collision blocks sufficed to reach a collision. In the next paragraphs we show how this can be done for general $\delta\mathrm{IHV}_n$ of the form $(0, \delta b, \delta c, \delta c)$.

Let, for any such difference vector, $\delta c = \sum_i k_i 2^i$ and $\delta b - \delta c = \sum_i l_i 2^i$, where $(k_i)_{i=0}^{31}$ and $(l_i)_{i=0}^{31}$ are NAFs. If $\delta c \neq 0$, let i be such that $k_i \neq 0$. Using a differential path from Table 4-1 with $\delta m_{11} = -k_i 2^{i-10 \bmod 32}$ we can eliminate the difference $k_i 2^i$ in δc and δd and simultaneously change δb by

$$k_i 2^i + \sum_{\lambda=i+21 \bmod 32}^{i+21+w' \bmod 32} l_\lambda 2^\lambda,$$

where $w' = \min(w, 31 - i)$. Here one needs to be careful that each non-zero l_λ is eliminated only once in the case when multiple i's allow the elimination of l_λ. Doing this for all non-zero k_i's in the NAF of δc will result in a difference vector $(0, \delta\hat{b}, 0, 0)$ where $\delta\hat{b}$ may be different from δb, and where the weight $w(\mathrm{NAF}(\delta\hat{b}))$ may be smaller or larger than $w(\mathrm{NAF}(\delta b))$. More precisely, $\delta\hat{b} = \sum_{\lambda=0}^{31} e_\lambda l_\lambda 2^\lambda$, where $e_\lambda = 0$ if there exist indices i and j with $0 \leq j \leq \min(w, 31 - i)$ such that $k_i = \pm 1$ and $\lambda = 21 + i + j \bmod 32$ and $e_\lambda = 1$ otherwise.

The bits in $\delta\hat{b}$ can be eliminated as follows. Let $(\hat{l}_i)_{i=0}^{31} = \mathrm{NAF}(\delta\hat{b})$ and let j be such that $\hat{l}_j = \pm 1$ and $j - 21 \bmod 32$ is minimal. Then the difference $\sum_{i=j}^{j+w'} \hat{l}_i 2^i$ with $w' = \min(w, 31 - (j - 21 \bmod 32))$ can be eliminated from $\delta\hat{b}$ using $\delta m_{11} = 2^{j-31 \bmod 32}$, which introduces a new difference $2^{j-21 \bmod 32}$ in δb, δc and δd. This latter difference is eliminated using $\delta m_{11} = -2^{j-31 \bmod 32}$, which then leads to a new difference vector $(0, \delta\bar{b}, 0, 0)$ with $w(\mathrm{NAF}(\delta\bar{b})) < w(\mathrm{NAF}(\delta\hat{b}))$. The process is repeated until all differences have been eliminated.

Algorithm 4-1 summarizes the construction of pairs of near-collision blocks set forth above. The details of the construction are described in the sections below.

4.2 Birthday search

A birthday search on a search space V is generally performed as in [23] by iterating a properly chosen deterministic function $f : V \to V$ and by assuming that the points of V thus visited form a 'random walk', also called a trail. After approximately $\sqrt{\pi|V|/2}$ iterations one may expect to have encountered a collision, i.e., different points x and y such that $f(x) = f(y)$. As the entire trail can in practice not be stored and to take advantage of parallelism, different pseudo-random walks are generated, of which only the startpoints, lengths, and endpoints are kept. The endpoints are 'distinguished points', points with an easily recognizable bitpattern depending on $|V|$, available storage and other

characteristics. The average length of a walk is inversely proportional to the fraction of distinguished points in V. Since intersecting walks share their endpoints, they can easily be detected. The collision point can then be recomputed given the startpoints and lengths of the two colliding walks. The expected cost (i.e., number of evaluations of f) to generate the walks is denoted by C_{tr} and the expected cost of the recomputation to determine collision points is denoted by C_{coll}.

In our case the search space V and iteration function f depend on an integer parameter $k \in \{0, 1, 2, \ldots, 32\}$ as explained in Section 4.1. The birthday collision that we try to find, however, needs to satisfy several additional conditions that cannot be captured by V, f, or k: the prefixes associated with x and y in a birthday collision $f(x) = f(y)$ must be different, and the required number of pairs of near-collision blocks may be at most r when allowing differential paths with parameter w. The probability that a collision satisfies all requirements depends not only on the choice of r and w, but also on the value for k, and is denoted by $p_{r,k,w}$. As a consequence, on average $1/p_{r,k,w}$ birthday collisions have to be found.

Table 4-2. Expected birthday costs for $k = 0$.

$k=0$	$w=0$			$w=1$			$w=2$			$w=3$		
r	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
16	5.9	35.27	1MB	1.75	33.2	1MB	1.01	32.83	1MB	1.	32.83	1MB
15	7.2	35.92	1MB	2.39	33.52	1MB	1.06	32.86	1MB	1.	32.83	1MB
14	8.71	36.68	1MB	3.37	34.01	1MB	1.27	32.96	1MB	1.04	32.84	1MB
13	10.45	37.55	1MB	4.73	34.69	1MB	1.78	33.22	1MB	1.2	32.93	1MB
12	12.45	38.55	1MB	6.53	35.59	1MB	2.78	33.71	1MB	1.66	33.16	1MB
11	14.72	39.68	2MB	8.77	36.71	1MB	4.34	34.5	1MB	2.61	33.63	1MB
10	17.28	40.97	11MB	11.47	38.06	1MB	6.54	35.6	1MB	4.18	34.42	1MB
9	20.16	42.4	79MB	14.62	39.64	2MB	9.38	37.02	1MB	6.46	35.56	1MB
8	23.39	44.02	732MB	18.21	41.43	21MB	12.88	38.76	1MB	9.52	37.09	1MB
7	26.82	45.73	8GB	22.2	43.43	323MB	17.02	40.83	9MB	13.4	39.02	1MB
6	31.2	47.92	161GB	26.73	45.69	8GB	21.78	43.22	241MB	18.14	41.4	20MB
5	35.	49.83	3TB	31.2	47.92	161GB	27.13	45.89	10GB	23.74	44.2	938MB
4							34.	49.33	2TB	30.19	47.42	81GB

The columns p, C_{tr} and M denote the values of $-\log_2(p_{r,k,w})$, $\log_2(C_{\text{tr}}(r,k,w))$ and the minimum required memory M such that $C_{\text{coll}}(r,k,w,M) \leq C_{\text{tr}}(r,k,w)$, respectively. See Appendix C for more extensive tables. The values for $p_{r,k,w}$ were estimated from Algorithm 4-1.

Assuming that M bytes of memory are available and that a single trail requires 28 bytes of storage (namely 96 bits for the start- and endpoint each, and 32 for the length), this leads to the following expressions for the birthday search

costs:

$$C_{\mathrm{tr}}(r,k,w) = \sqrt{\frac{\pi \cdot |V|}{2p_{r,k,w}}}, \quad C_{\mathrm{coll}}(r,k,w,M) = \frac{2.5 \cdot 28 \cdot C_{\mathrm{tr}}(r,k,w)}{p_{r,k,w} \cdot M},$$

where $|V| = 2^{64+k}$, and the factor 2.5 is explained in Section 3 of [23].

For $M = 70/p_{r,k,w}$ as given in the last column of Table 4-2 and in the more extensive tables in Appendix C, the two costs are equal, and the overall expected birthday costs becomes $2C_{\mathrm{tr}}(r,k,w)$. However, if the cost at run time of finding the trails exceeds the expected cost by a factor λ, then the cost to determine the resulting birthday collisions can be expected to increase by a factor λ^2. Hence, in practice it is advisable to choose M considerably larger. For $\epsilon \leq 1$, using $M = 70/(p_{r,k,w} \cdot \epsilon)$ bytes of memory will result in $C_{\mathrm{coll}} \approx \epsilon \cdot C_{\mathrm{tr}}$ and the expected overall birthday search cost will be about $(1+\epsilon) \cdot C_{\mathrm{tr}}(r,k,w)$ MD5 compressions.

4.3 Differential paths and bitconditions

In step (e) of Algorithm 4-1, MD5Compress is applied to the respective intermediate hash values IHV and IHV′ and message blocks B and B'. Here, IHV and IHV′ were constructed in such a way that δIHV has a specific structure, as set forth above. Furthermore, the blocks B and B' were constructed such that δB has a pre-specified low-weight value (cf. step (b) of Algorithm 4-1) and such that throughout the 64 steps of both calls to MD5Compress the propagation of differences between corresponding variables follows a specific precise description, as determined in step (c) of Algorithm 4-1. In this section we describe how this description, which is called a *differential path* for MD5Compress, is determined based on IHV, IHV′ and δB. According to equations (2),

$$\begin{cases} \delta F_t & = f_t(Q'_t, Q'_{t-1}, Q'_{t-2}) - f_t(Q_t, Q_{t-1}, Q_{t-2}), \\ \delta T_t & = \delta F_t + \delta Q_{t-3} + \delta W_t, \\ \delta R_t & = RL(T'_t, RC_t) - RL(T_t, RC_t), \text{ and} \\ \delta Q_{t+1} & = \delta Q_t + \delta R_t. \end{cases} \tag{4}$$

It follows that neither δF_t nor δR_t is uniquely determined given the input differences $(\delta Q_t, \delta Q_{t-1}, \delta Q_{t-2})$ and δT_t, respectively. Therefore a more flexible tool is required to describe in a succinct way a valid propagation of differences, starting from IHV $= (Q_{-3}, Q_0, Q_{-1}, Q_{-2})$, IHV′ $= (Q'_{-3}, Q'_0, Q'_{-1}, Q'_{-2})$ and δB and, in our case, resulting in the desired final differences $(\delta Q_{61}, \delta Q_{62}, \delta Q_{63}, \delta Q_{64})$ as defined in Table 4-1 and as targeted by step (c) of Algorithm 4-1.

4.3.1 Bitconditions

Differential paths are described using *bitconditions* $\mathfrak{q}_t = (\mathfrak{q}_t[i])_{i=0}^{31}$ on (Q_t, Q'_t), where each bitcondition $\mathfrak{q}_t[i]$ specifies a restriction on the bits $Q_t[i]$ and $Q'_t[i]$ possibly including values of other bits $Q_l[i]$. As we will show in this section, we can specify the values of δQ_t, δF_t for all t using bitconditions on (Q_t, Q'_t),

which also determine δT_t and $\delta R_t = \delta Q_{t+1} - \delta Q_t$ according to the difference equations (4). Thus, a differential path can be seen as a 68×32 matrix $(\mathtt{q}_t)_{t=-3}^{64}$ of bitconditions. In general, the first four rows $(\mathtt{q}_t)_{t=-3}^{0}$ are fully determined by the values of IHV and IHV$'$. Furthermore, in our specific case where δB consists of just $\delta m_{11} = \pm 2^d$, the final 34 rows $(\mathtt{q}_t)_{t=31}^{64}$ correspond to Table 4-1 and one of the choices made in step (c) of Algorithm 4-1.

Table 4-3. Differential bitconditions

$\mathtt{q}_t[i]$	condition on $(Q_t[i], Q'_t[i])$	k_i
`.`	$Q_t[i] = Q'_t[i]$	0
`+`	$Q_t[i] = 0, \quad Q'_t[i] = 1$	+1
`-`	$Q_t[i] = 1, \quad Q'_t[i] = 0$	−1

$\delta Q_t = \sum_{i=0}^{31} 2^i k_i$ and $\Delta Q_t = (k_i)$.

Table 4-4. Boolean function bitconditions

$\mathtt{q}_t[i]$	condition on $(Q_t[i], Q'_t[i])$	direct/indirect	direction
`0`	$Q_t[i] = Q'_t[i] = 0$	direct	
`1`	$Q_t[i] = Q'_t[i] = 1$	direct	
`^`	$Q_t[i] = Q'_t[i] = Q_{t-1}[i]$	indirect	backward
`v`	$Q_t[i] = Q'_t[i] = Q_{t+1}[i]$	indirect	forward
`!`	$Q_t[i] = Q'_t[i] = \overline{Q_{t-1}[i]}$	indirect	backward
`y`	$Q_t[i] = Q'_t[i] = \overline{Q_{t+1}[i]}$	indirect	forward
`m`	$Q_t[i] = Q'_t[i] = Q_{t-2}[i]$	indirect	backward
`w`	$Q_t[i] = Q'_t[i] = Q_{t+2}[i]$	indirect	forward
`#`	$Q_t[i] = Q'_t[i] = \overline{Q_{t-2}[i]}$	indirect	backward
`h`	$Q_t[i] = Q'_t[i] = \overline{Q_{t+2}[i]}$	indirect	forward
`?`	$Q_t[i] = Q'_t[i] \wedge (Q_t[i] = 1 \vee Q_{t-2}[i] = 0)$	indirect	backward
`q`	$Q_t[i] = Q'_t[i] \wedge (Q_{t+2}[i] = 1 \vee Q_t[i] = 0)$	indirect	forward

Bitconditions are denoted using symbols such as `0`, `1`, `+`, `-`, `^`, ..., as defined in Tables 4-3 and 4-4, to facilitate the representation of a differential path. A *direct* bitcondition $\mathtt{q}_t[i]$ does not involve any other indices than t and i, whereas an *indirect* bitcondition involves one of the row indices $t \pm 1$ or $t \pm 2$ as well. Table 4-3 lists *differential* bitconditions $\mathtt{q}_t[i]$, which are direct bitconditions that specify the value $k_i = Q'_t[i] - Q_t[i]$. A full row of differential bitconditions $\mathtt{q}_t = (k_i)_{i=0}^{31}$ fixes a BSDR of $\delta Q_t = \sum_{i=0}^{31} 2^i k_i$. Table 4-4 lists *boolean function* bitconditions, which are direct or indirect. They are used to resolve a possible ambiguity in

$$\Delta F_t[\![i]\!] = f_t(Q'_t[i], Q'_{t-1}[i], Q'_{t-2}[i]) - f_t(Q_t[i], Q_{t-1}[i], Q_{t-2}[i]) \in \{-1, 0, +1\}$$

that may be caused by different possible values for $Q_j[i], Q'_j[i]$ given differential bitconditions $\mathtt{q}_j[i]$. As an example, for $t = 0$ and $(\mathtt{q}_t[i], \mathtt{q}_{t-1}[i], \mathtt{q}_{t-2}[i]) = (\mathtt{.}, \mathtt{+}, \mathtt{-})$

(cf. Table 4-3) there is an ambiguity:

$$\text{if } Q_t[i] = Q'_t[i] = 0 \text{ then } \Delta F_t[\![i]\!] = f_t(0,1,0) - f_t(0,0,1) = -1,$$
$$\text{but if } Q_t[i] = Q'_t[i] = 1 \text{ then } \Delta F_t[\![i]\!] = f_t(1,1,0) - f_t(1,0,1) = +1.$$

To resolve this ambiguity the triple of bitconditions (`.`,`+`,`-`) can be replaced by (`0`,`+`,`-`) or (`1`,`+`,`-`) for the two cases given above, respectively.

All boolean function bitconditions include the constant bitcondition $Q_t[i] = Q'_t[i]$, so boolean function bitconditions do not affect δQ_t. Furthermore, the indirect boolean function bitconditions never involve bitconditions `+` or `-`, since those bitconditions can always be replaced by one of the direct ones `.`, `0` or `1`. For the indirect bitconditions we distinguish between 'forward' and 'backward' ones, because that makes it easier to resolve an ambiguity later on in our step-wise approach. In a valid (partial) differential path one can easily convert forward bitconditions into backward bitconditions and vice versa.

When all δQ_t and δF_t have been determined by bitconditions then also δT_t and $\delta R_t = \delta Q_{t+1} - \delta Q_t$ can be determined, which together describe the bitwise rotation of δT_t in each step. This does, however, not imply that the left rotate of δT_t over RC_t positions is equal to δR_t or with what probability that happens. See also Section 4.4.4.

The differential paths we constructed for several of our examples can be found at `http://www.win.tue.nl/hashclash/ChosenPrefixCollisions/`.

4.4 Differential path construction

The basic idea to construct a differential path is to construct a partial lower differential path over steps $t = 0, 1, \ldots, 11$ and a partial upper differential path over steps $t = 63, 62, \ldots, 16$, so that the Q_i involved in the partial paths meet but do not overlap. Given the two partial paths, we try to connect them over the remaining 4 steps into one full differential path which hopefully succeeds with some non-negligible probability. Using many lower and upper differential paths and trying to connect each combination of a lower and an upper differential path will eventually result in full differential paths. Constructing the partial lower path can be done by starting with bitconditions $\mathtt{q}_{-3}$, $\mathtt{q}_{-2}$, $\mathtt{q}_{-1}$, $\mathtt{q}_0$ that are equivalent to the values of IHV, IHV$'$ and then extend this step by step. Similarly the partial upper path can be constructed by extending the partial paths in Table 4-1 step by step. In both constructions the transitions between the steps must be compatible with the targeted message difference δB. To summarize, step (c) of Algorithm 4-1 in Section 4.1 consists of the following substeps:

c.1 Given IHV and IHV$'$, determine the corresponding bitconditions $(\mathtt{q}_i)_{i=-3}^{0}$.
c.2 Generate partial lower differential paths by extending $(\mathtt{q}_i)_{i=-3}^{0}$ forward up to step $t = 11$. This is explained in Sections 4.4.1 - 4.4.4.
c.3 Generate partial upper differential paths by extending the path specified by Table 4-1 backward from $t = 31$ down to $t = 16$. This is explained in Section 4.4.5.

c.4 Try to connect all pairs of lower and upper differential paths over $t = 12, 13, 14, 15$ to generate as many full differential paths as possible given the outcome of the two previous steps. This is explained in Section 4.4.6.

4.4.1 Extending differential paths forward

In general, when constructing a differential path one must first fix the message block differences $\delta m_0, \ldots, \delta m_{15}$. In our particular case this is achieved by the choice of $\delta S_{c,j}$ in step (b) of Algorithm 4-1. Suppose we have a partial differential path consisting of at least bitconditions $\mathfrak{q}_{t-1}$ and $\mathfrak{q}_{t-2}$ and that the differences δQ_t and δQ_{t-3} are known. In step c.2 of Algorithm 4-1, we want to extend this partial differential path forward with step t resulting in the difference δQ_{t+1}, bitconditions $\mathfrak{q}_t$, and additional bitconditions $\mathfrak{q}_{t-1}, \mathfrak{q}_{t-2}$ (cf. Section 4.4).

We assume that all indirect bitconditions in $\mathfrak{q}_{t-1}$ and $\mathfrak{q}_{t-2}$ are forward and involve only bits of Q_{t-1}. If we already have $\mathfrak{q}_t$ as opposed to just the value δQ_t (e.g. $\mathfrak{q}_0$ resulting from given values $\mathrm{IHV}, \mathrm{IHV}'$), then we can skip Section 4.4.2 and continue at Section 4.4.3.

4.4.2 Carry propagation

First we select bitconditions $\mathfrak{q}_t$ based on the value δQ_t. Since we want to construct differential paths with as few bitconditions as possible, but also want to be able to randomize the process, any low weight BSDR (such as the NAF) of δQ_t may be chosen, which then translates into a possible choice for $\mathfrak{q}_t$ as in Table 4-3. For instance, with $\delta Q_t = 2^8$, we may choose $\mathfrak{q}_t[8] = \text{`+'}$, or $\mathfrak{q}_t[8] = \text{`-'}$ and $\mathfrak{q}_t[9] = \text{`+'}$ (with in either case all other $\mathfrak{q}_t[i] = \text{`.'}$).

4.4.3 Boolean function

For some i, let $(a, b, c) = (\mathfrak{q}_t[i], \mathfrak{q}_{t-1}[i], \mathfrak{q}_{t-2}[i])$ be any triple of bitconditions such that all indirect bitconditions involve only $Q_t[i]$, $Q_{t-1}[i]$ or $Q_{t-2}[i]$. For any such triple (a, b, c) let U_{abc} denote the set of tuples of values $(x, x', y, y', z, z') = (Q_t[i], Q'_t[i], Q_{t-1}[i], Q'_{t-1}[i], Q_{t-2}[i], Q'_{t-2}[i])$ satisfying it:

$$U_{abc} = \left\{(x, x', y, y', z, z') \in \{0,1\}^6 \text{ satisfies bitconditions } (a, b, c)\right\}.$$

The cardinality of U_{abc} indicates the amount of freedom left by (a, b, c). Triples (a, b, c) for which $U_{abc} = \emptyset$ cannot be part of a valid differential path and are thus of no interest. The set of all triples (a, b, c) as above and with $U_{abc} \neq \emptyset$ is denoted by $\mathcal{F}_t$.

Each $(a, b, c) \in \mathcal{F}_t$ induces a set V_{abc} of possible boolean function differences $\Delta F_t[\![i]\!] = f_t(x', y', z') - f_t(x, y, z)$:

$$V_{abc} = \{f_t(x', y', z') - f_t(x, y, z) \mid (x, x', y, y', z, z') \in U_{abc}\} \subset \{-1, 0, +1\}.$$

A triple $(d, e, f) \in \mathcal{F}_t$ with $|V_{def}| = 1$ leaves no ambiguity in $\Delta F_t[\![i]\!]$ and is therefore called a *solution*. Let $\mathcal{S}_t \subset \mathcal{F}_t$ be the set of solutions.

For arbitrary $(a, b, c) \in \mathcal{F}_t$ and for each $g \in V_{abc}$, we define $W_{abc,g}$ as the subset of $\mathcal{S}_t$ consisting of all solutions that are compatible with (a, b, c) and that have g as boolean function difference:

$$W_{abc,g} = \{(d, e, f) \in \mathcal{S}_t \mid U_{def} \subset U_{abc} \wedge V_{def} = \{g\}\}.$$

For each $g \in V_{abc}$ there is always a triple $(d, e, f) \in W_{abc,g}$ consisting of direct bitconditions `01+-` that suffices, i.e., fixes a certain tuple in U_{abc}. This implies that $W_{abc,g} \neq \emptyset$. Despite this fact, we are specifically interested in bitconditions $(d, e, f) \in W_{abc,g}$ that maximize $|U_{def}|$ as such bitconditions maximize the amount of freedom in the bits of Q_t, Q_{t-1}, Q_{t-2} while fully determining $\Delta F_t[\![i]\!]$.

The direct and forward (resp. backward) boolean function bitconditions were chosen such that for all t, i and $(a, b, c) \in \mathcal{F}_t$ and for all $g \in V_{abc}$ there exists a triple $(d, e, f) \in W_{abc,g}$ consisting only of direct and forward (resp. backward) bitconditions such that

$$\{(x, x', y, y', z, z') \in U_{abc} \mid f_t(x', y', z') - f_t(x, y, z) = g\} = U_{def}.$$

These boolean function bitconditions allow one to resolve an ambiguity in an optimal way in the sense that they are sufficient *and* necessary.

If the triple (d, e, f) is not unique, then for simplicity we prefer direct over indirect bitconditions and short indirect bitconditions (`vy^!`) over long indirect ones (`whqm#?`). For given t, bitconditions (a, b, c), and $g \in V_{abc}$ we define $FC(t, abc, g) = (d, e, f)$ as the preferred triple (d, e, f) consisting of direct and forward bitconditions. Similarly, we define $BC(t, abc, g)$ as the preferred triple consisting of direct and backward bitconditions. These functions are easily determined and should be precomputed. They have been tabulated in Appendix B in Tables B-1, B-2, B-3 and B-4 grouped according to the four different round functions F, G, H, I, and per table for all 27 possible triples (a, b, c) of differential bitconditions.

To determine the differences $\delta F_t = \sum_{i=0}^{31} 2^i g_i$ we proceed as follows. For $i = 0, 1, 2, \ldots, 31$ we assume that we have valid bitconditions $(a, b, c) = (\mathtt{q}_t[i], \mathtt{q}_{t-1}[i], \mathtt{q}_{t-2}[i])$ where only c may be indirect. If it is, it must involve $Q_{t-1}[i]$. Therefore $(a, b, c) \in \mathcal{F}_t$. If $|V_{abc}| = 1$, then there is no ambiguity and $\{g_i\} = V_{abc}$. Otherwise, if $|V_{abc}| > 1$, then we choose g_i arbitrarily from V_{abc} and we resolve the ambiguity by replacing bitconditions (a, b, c) by $FC(t, abc, g_i)$. Once all g_i and thus δF_t have been determined, δT_t is determined as $\delta F_t + \delta Q_{t-3} + \delta W_t$.

Note that in the next step $t+1$ our assumptions hold again, since a is a direct bitcondition and if b is indirect then it is forward and involves a. Bitconditions a and b may be new compared to the previous step, namely if the triple (a, b, c) was replaced by $FC(t, abc, g_i)$.

4.4.4 Bitwise rotation

The integer δT_t as just determined does not uniquely determine $\delta R_t = RL(T'_t, n) - RL(T_t, n)$, where $n = RC_t$ (cf. difference equations (4)). In this section we show how to find the most likely δR_t that corresponds to a given δT_t, i.e., the v for which $|\{X \in \mathbb{Z}/2^{32}\mathbb{Z} \mid v = RL(X + \delta T_t, n) - RL(X, n)\}|$ is maximized

Any BSDR $(k_{31}, \ldots, k_{32-n}, k_{31-n}, \ldots, k_0)$ of δT_t gives rise to a candidate δR_t given by the BSDR $RL((k_i), n) = (k_{31-n}, \ldots, k_0, k_{31}, \ldots, k_{32-n})$. Two BSDRs (k_i) and (l_i) of δT_t result in the same δR_t if

$$\sum_{i=0}^{31-n} 2^i k_i = \sum_{i=0}^{31-n} 2^i l_i \quad \text{and} \quad \sum_{i=32-n}^{31} 2^i k_i = \sum_{i=32-n}^{31} 2^i l_i.$$

This suggests the following approach. We define a *partition* as a pair (α, β) of integers such that $\alpha + \beta = \delta T_t \bmod 2^{32}$, $|\alpha| < 2^{32-n}$, $|\beta| < 2^{32}$ and $2^{32-n} | \beta$. For any partition (α, β), values $k_i \in \{0, \pm 1\}$ for $0 \leq i < 32$ can be found such that

$$\alpha = \sum_{i=0}^{31-n} 2^i k_i \quad \text{and} \quad \beta = \sum_{i=32-n}^{31} 2^i k_i. \tag{5}$$

With $\alpha + \beta = \delta T_t \bmod 2^{32}$ it follows that (k_i) is a BSDR of δT_t. Conversely, with (5) any BSDR (k_i) of δT_t defines a partition, which we denote $(k_i) \equiv (\alpha, \beta)$.

The rotation of a partition (α, β) is defined as

$$RL((\alpha, \beta), n) = (2^n \alpha + 2^{n-32} \beta \bmod 2^{32}).$$

If $(k_i) \equiv (\alpha, \beta)$, this matches $RL((k_i), n)$. The latter, as seen above, is a candidate δR_t, and we find that different partitions give rise to different δR_t candidates. Thus, to find the most likely δR_t, we define

$$p_{(\alpha,\beta)} = Pr[RL((\alpha, \beta), n) = RL(X + \delta T_t, n) - RL(X, n)]$$

where X ranges over the 32-bit words, and show how $p_{(\alpha,\beta)}$ can be calculated.

Let $x = \delta T_t \bmod 2^{32-n}$ and $y = (\delta T_t - x) \bmod 2^{32}$ with $0 \leq x < 2^{32-n}$ and $0 \leq y < 2^{32}$. This gives rise to at most 4 partitions:

- $(\alpha, \beta) = (x, y)$;
- $(\alpha, \beta) = (x, y - 2^{32})$, if $y \neq 0$;
- $(\alpha, \beta) = (x - 2^{32-n}, y + 2^{32-n} \bmod 2^{32})$, if $x \neq 0$;
- $(\alpha, \beta) = (x - 2^{32-n}, (y + 2^{32-n} \bmod 2^{32}) - 2^{32})$, if $x \neq 0 \wedge y + 2^{32-n} \neq 2^{32}$.

These are all possible partitions, so we find that δT_t leads to at most 4 different possibilities for δR_t. It remains to determine $p_{(\alpha,\beta)}$ for the above partitions. For each of the 4 possibilities this is done by counting the number of 32-bit words X such that the BSDR defined by $k_i = (X + \delta T_t)[i] - X[i]$ satisfies $(k_i) \equiv (\alpha, \beta)$. Considering the $(32 - n)$ low-order bits, the probability that a given α satisfies $\alpha = \sum_{i=0}^{31-n} k_i$ follows from the number r of Y's with $0 \leq Y < 2^{32-n}$ such that $0 \leq \alpha + Y < 2^{32-n}$: if $\alpha < 0$ then $r = 2^{32-n} + \alpha$ and if $\alpha \geq 0$ then $r = 2^{32-n} - \alpha$. Hence $r = 2^{32-n} - |\alpha|$ out of 2^{32-n} Y's. Now assuming $\alpha = \sum_{i=0}^{31-n} k_i$, there is no carry to the high-order bits and the same argument can be used for $\beta / 2^{32-n}$. Hence, we conclude

$$p_{(\alpha,\beta)} = \frac{2^{32-n} - |\alpha|}{2^{32-n}} \cdot \frac{2^n - |\beta| 2^{n-32}}{2^n} = \frac{2^{32-n} - |\alpha|}{2^{32-n}} \cdot \frac{2^{32} - |\beta|}{2^{32}}.$$

Note that these probabilities, corresponding to the at most 4 partitions above, indeed add up to 1.

We have shown that all δR_t that are compatible with a given δT_t can easily be determined, including the probabilities that they occur. In Algorithm 4-1 we choose a partition (α, β) for which $p_{(\alpha,\beta)}$ is maximal and take $\delta R_t = RL((\alpha, \beta), n)$. A more straightforward approach (as previously used in practice) would be to use $\delta R_t = RL(\mathrm{NAF}(\delta T_t), n)$. This is in many cases the most likely choice, and matches our desire to minimize the number of differences in δQ_t and therefore also in δT_t and δR_t. Given δR_t, we finally determine δQ_{t+1} as $\delta Q_t + \delta R_t$.

4.4.5 Extending differential paths backward

Having dealt with the forward extension of step c.2 of Algorithm 4-1 in Sections 4.4.2, 4.4.3 and 4.4.4, we now consider the backward extension of step c.3 of Algorithm 4-1 (cf. Section 4.4). The backward construction follows the same approach as the forward one. Our description relies on the notation introduced in Section 4.4.3.

Suppose we have a partial differential path consisting of at least bitconditions $\mathfrak{q}_t$ and $\mathfrak{q}_{t-1}$ and that the differences δQ_{t+1} and δQ_{t-2} are known. In step c.3 of Algorithm 4-1 we want to extend this partial differential path backward with step t resulting in the difference δQ_{t-3}, bitconditions $\mathfrak{q}_{t-2}$, and additional bitconditions $\mathfrak{q}_t, \mathfrak{q}_{t-1}$. We assume that all indirect bitconditions in $\mathfrak{q}_t$ and $\mathfrak{q}_{t-1}$ are backward and only involve bits of Q_{t-1}.

We choose a low weight BSDR (such as the NAF) of δQ_{t-2}, which then translates into a possible choice for $\mathfrak{q}_{t-2}$ as in Table 4-3.

As in the last two paragraphs of Section 4.4.3, the differences $\delta F_t = \sum_{i=0}^{31} 2^i g_i$ are determined by assuming for $i = 0, 1, \ldots, 31$ that we have valid bitconditions $(a, b, c) = (\mathfrak{q}_t[i], \mathfrak{q}_{t-1}[i], \mathfrak{q}_{t-2}[i])$ where only a may be indirect. If it is, it must involve $Q_{t-1}[i]$. Therefore $(a, b, c) \in \mathcal{F}_t$. If $|V_{abc}| = 1$, then there is no ambiguity and $\{g_i\} = V_{abc}$. Otherwise, if $|V_{abc}| > 1$, then we choose g_i arbitrarily from V_{abc} and we resolve the ambiguity by replacing bitconditions (a, b, c) by $BC(t, abc, g_i)$.

To rotate $\delta R_t = \delta Q_{t+1} - \delta Q_t$ over $n = 32 - RC_t$ bits, we may follow the framework as set forth in Section 4.4.4 with the roles of δR_t and δT_t reversed: choose a partition (α, β) (of δR_t as opposed to δT_t) with maximal probability and determine $\delta T_t = RL((\alpha, \beta), n)$. Finally, we determine $\delta Q_{t-3} = \delta T_t - \delta F_t - \delta W_t$ to extend our partial differential path backward with step t. Note that here also (i.e., as in the last paragraph of Section 4.4.3) in the next step $t - 1$ our assumptions hold again, since c is a direct bitcondition and if b is indirect then it is backward and involves c (where b and c are new if (a, b, c) was replaced by $BC(t, abc, g_i)$).

4.4.6 Constructing full differential paths

Construction of a full differential path can be done as follows. Assume that for some δQ_{-3} and bitconditions $\mathfrak{q}_{-2}$, $\mathfrak{q}_{-1}$, $\mathfrak{q}_0$ the forward construction as described in Sections 4.4.1, 4.4.2, 4.4.3, and 4.4.4 has been carried out up to step $t = 11$ (cf. step c.2 in Section 4.4). Furthermore, assume that for some δQ_{64} and

bitconditions $\mathfrak{q}_{63}, \mathfrak{q}_{62}, \mathfrak{q}_{61}$ the backward construction as described in Section 4.4.5 has been carried out down to step $t = 16$ (cf. step c.3 in Section 4.4). For each combination of forward and backward partial differential paths thus found, this leads to bitconditions $\mathfrak{q}_{-2}, \mathfrak{q}_{-1}, \ldots, \mathfrak{q}_{11}, \mathfrak{q}_{14}, \mathfrak{q}_{15}, \ldots, \mathfrak{q}_{63}$ and differences δQ_{-3}, δQ_{12}, δQ_{13}, δQ_{64}.

It remains to try and glue together each of these combinations by finishing steps $t = 12, 13, 14, 15$ (cf. step c.4 in Section 4.4) until a full differential path is found. First, as in the backward extension in Section 4.4.5, for $t = 12, 13, 14, 15$ we set $\delta R_t = \delta Q_{t+1} - \delta Q_t$, choose the resulting δT_t by left-rotating δR_t over $n - RC_t$ bits, and determine $\delta F_t = \delta T_t - \delta W_t - \delta Q_{t-3}$.

Algorithm 4-2 Construction of $\mathcal{U}_{i+1}$ from $\mathcal{U}_i$.

Suppose $\mathcal{U}_i$ is given as $\{(\delta Q_{12}, \delta Q_{13}, \delta F_{12}, \delta F_{13}, \delta F_{14}, \delta F_{15})\}$ if $i = 0$ or if $i > 0$ constructed inductively based on $\mathcal{U}_{i-1}$ by means of this algorithm. For each tuple $(q_1, q_2, f_1, f_2, f_3, f_4) \in \mathcal{U}_i$ do the following:

1. Let $\mathcal{U}_{i+1} = \emptyset$ and $(a, b, e, f) = (\mathfrak{q}_{15}[i], \mathfrak{q}_{14}[i], \mathfrak{q}_{11}[i], \mathfrak{q}_{10}[i])$
2. For each bitcondition $d = \mathfrak{q}_{12}[i] \in \begin{cases} \{.\} & \text{if } q_1[i] = 0 \\ \{-, +\} & \text{if } q_1[i] = 1 \end{cases}$ do
3. Let $q_1' = 0, -1$ or $+1$ depending on whether $d =$ '.', '-' or '+', respectively
4. For each different $f_1' \in \{-f_1[i], +f_1[i]\} \cap V_{def}$ do
5. Let $(d', e', f') = FC(12, def, f_1')$
6. For each bitcondition $c = \mathfrak{q}_{13}[i] \in \begin{cases} \{.\} & \text{if } q_2[i] = 0 \\ \{-, +\} & \text{if } q_2[i] = 1 \end{cases}$ do
7. Let $q_2' = 0, -1$ or $+1$ depending on whether $c =$ '.', '-' or '+', respectively
8. For each different $f_2' \in \{-f_2[i], +f_2[i]\} \cap V_{cd'e'}$ do
9. Let $(c', d'', e'') = FC(13, cd'e', f_2')$
10. For each different $f_3' \in \{-f_3[i], +f_3[i]\} \cap V_{bc'd''}$ do
11. Let $(b', c'', d''') = FC(14, bc'd'', f_3')$
12. For each different $f_4' \in \{-f_4[i], +f_4[i]\} \cap V_{ab'c''}$ do
13. Let $(a', b'', c''') = FC(15, ab'c'', f_4')$
14. If $(q_1 - 2^i q_1', q_2 - 2^i q_2', f_1 - 2^i f_1', f_2 - 2^i f_2', f_3 - 2^i f_3', f_4 - 2^i f_4')$ is not in $\mathcal{U}_{i+1}$ yet, insert it in $\mathcal{U}_{i+1}$

We aim to complete the differential path by finding new bitconditions $\mathfrak{q}_{10}$, $\mathfrak{q}_{11}$, ..., $\mathfrak{q}_{15}$ that are compatible with the original bitconditions and that result in the required δQ_{12}, δQ_{13}, δF_{12}, δF_{13}, δF_{14}, δF_{15}.

An efficient way to find the missing bitconditions is to first test if they exist, and if so to backtrack to actually construct them. For $i = 0, 1, \ldots, 32$ we attempt to construct a set $\mathcal{U}_i$ consisting of all tuples $(q_1, q_2, f_1, f_2, f_3, f_4)$ of 32-bit integers with $q_j \equiv f_k \equiv 0 \bmod 2^i$ for $j = 1, 2$ and $k = 1, 2, 3, 4$ such that for all $\ell = 0, 1, \ldots, i-1$ there exist compatible bitconditions $\mathfrak{q}_{10}[\ell], \mathfrak{q}_{11}[\ell], \ldots, \mathfrak{q}_{15}[\ell]$ that

determine $\Delta Q_{11+j}[\![\ell]\!]$ and $\Delta F_{11+k}[\![\ell]\!]$ below, and such that

$$\begin{cases} \delta Q_{11+j} &= q_j + \sum_{\ell=0}^{i-1} 2^\ell \Delta Q_{11+j}[\![\ell]\!], \quad j = 1, 2, \\ \delta F_{11+k} &= f_k + \sum_{\ell=0}^{i-1} 2^\ell \Delta F_{11+k}[\![\ell]\!], \quad k = 1, 2, 3, 4. \end{cases} \tag{6}$$

From these conditions it follows that $\mathcal{U}_0$ must be chosen as $\{(\delta Q_{12}, \delta Q_{13}, \delta F_{12}, \delta F_{13}, \delta F_{14}, \delta F_{15})\}$. For $i = 1, 2, \ldots, 32$, we attempt to construct $\mathcal{U}_i$ based on $\mathcal{U}_{i-1}$ using Algorithm 4-2. Per j there are at most two q_j's and per k there are at most two f_k's that can satisfy the above relations. Thsi implies that $|\mathcal{U}_i| \leq 2^6$ for each i, $0 \leq i \leq 32$. On the other hand, for each tuple in $\mathcal{U}_i$ there may in principle be many different compatible sets of bitconditions.

As soon as we encounter an i for which $\mathcal{U}_i = \emptyset$, we know that the desired bitconditions do not exist, and that we should try another combination of forward and backward partial differential paths. If, however, we find $\mathcal{U}_{32} \neq \emptyset$ then it must be the case that $\mathcal{U}_{32} = \{(0, 0, 0, 0, 0, 0)\}$. Furthermore, in that case, every set of bitconditions that leads to this non-empty $\mathcal{U}_{32}$ gives rise to a full differential path, since equations (6) hold with $i = 32$. Thus, if $\mathcal{U}_{32} \neq \emptyset$, there exists at least one valid path $u_0, u_1, \ldots, u_{32}$ with $u_i \in \mathcal{U}_i$. For each valid path, the desired new bitconditions $(\mathfrak{q}_{15}[i], \mathfrak{q}_{14}[i], \ldots, \mathfrak{q}_{10}[i])$ are $(a', b'', c''', d''', e'', f')$, which can be found at step 13 of Algorithm 4-2.

4.5 Collision finding

Collision finding is the process of finding an actual message block pair $S_{c,j}, S'_{c,j}$ that satisfies a given $\delta S_{c,j}$ and a differential path based on a given IHV_{n+j-1}, IHV'_{n+j-1}, cf. step (d) of Algorithm 4-1. The differential paths as originally considered by Wang et al. [34] consisted of only 28 bitconditions. In that case, collision finding can now be done in the equivalent of a mere $2^{24.8}$ expected MD5 compression function calls, for arbitrary IHV [28]. For chosen-prefix collisions, however, the number of bitconditions is substantially larger, thereby complicating collision finding. For instance, in one of our earliest chosen-prefix collision constructions the differential path has 71 bitconditions on Q_{20} up to Q_{63}.

4.5.1 Tunnels

To find collisions for these more difficult differential paths, we make extensive use of so-called *tunnels* [17]. A tunnel allows one to make small changes in a certain first round Q_t, in specific bits of Q_t that are determined by the full differential path $\mathfrak{q}_{-3}, \mathfrak{q}_{-2}, \ldots, \mathfrak{q}_{64}$ under consideration, while causing changes in the second round only after some step l that depends on the tunnel. However, each tunnel implies that additional first-round bitconditions have to be taken into account in the differential path, while leaving freedom of choice for some of the bits in Q_t that may be changed. A tunnel's *strength* is the number of independent bits

that can be changed in this first round Q_t. Thus, a tunnel of strength k allows us to generate 2^k different message blocks that all satisfy the differential path up to and including step l in the second round.

Table 4-5. Collision finding tunnels for MD5.

Tunnel	Change	Affected	Extra bitconditions*
$\mathcal{T}_1$	$Q_4[b]$	$m_3..m_5, m_7, Q_{21}..Q_{64}$	$Q_5[b] = 1, Q_6[b] = 1$
$\mathcal{T}_2$	$Q_5[b]$	$m_4, m_5, m_7, m_8, Q_{21}..Q_{64}$	$Q_6[b] = 0$
$\mathcal{T}_3$	$Q_{14}[b]$	$m_{13}..m_{15}, m_6, Q_3, m_2..m_5, Q_{21}..Q_{64}$	$Q_{15}[b] = Q_{16}[b], Q_3[b]$ free†
$\mathcal{T}_4$	$Q_9[b]$	$m_8..m_{10}, m_{12}, Q_{22}..Q_{64}$	$Q_{10}[b] = 1, Q_{11}[b] = 1$
$\mathcal{T}_5$	$Q_{10}[b]$	$m_9, m_{10}, m_{12}, m_{13}, Q_{22}..Q_{64}$	$Q_{11}[b] = 0$
$\mathcal{T}_6$	$Q_8[b]$	$m_7..m_9, Q_{12}, m_{12}..m_{15}, Q_{23}..Q_{64}$	$Q_{10}[b] = 1, RR(Q_{12}, 22)[b]$ free‡
$\mathcal{T}_7$	$Q_4[b]$	$m_3, m_4, m_7, Q_{24}..Q_{64}$	$Q_5[b] = 0, Q_6[b] = 1$
$\mathcal{T}_8$	$Q_9[b]$	$m_8, m_9, m_{12}, Q_{25}..Q_{64}$	$Q_{10}[b] = 0, Q_{11}[b] = 1$

* The extra bitconditions refer only to $Q_t[b]$ and not to $Q'_t[b]$, so e.g. $Q_6[b] = 0$ is met by both $\mathtt{q}_6[b] =$ '0' and $\mathtt{q}_6[b] =$ '+'.
† Bitcondition $\mathtt{q}_3[b] =$ '.' and no other indirect bitconditions may involve $Q_3[b]$. Set $Q_3[b] = Q_{14}[b]$ to avoid carries in Q_3.
‡ Bitcondition $\mathtt{q}_{12}[b - 22 \bmod 32] =$ '.' and no other indirect bitconditions may involve $Q_{12}[b - 22 \bmod 32]$. Set $Q_{12}[b - 22 \bmod 32] = Q_8[b]$ to avoid carries in Q_{12}.

The tunnels used in our collision finding algorithm are shown in Table 4-5. For example, the first tunnel ($\mathcal{T}_1$) allows changes in bits of Q_4, in such a way that if $Q_4[b]$ is changed for some bit position b with $0 \leq b < 32$, this causes extra bitconditions $Q_5[b] = 1$ and $Q_6[b] = 1$, which have to be incorporated in the differential path. Furthermore, because tunnel $\mathcal{T}_1$ affects after the first round only Q_{21} through Q_{64} we have that $l = 20$, and $\mathcal{T}_1$ can be used to change message blocks m_3, m_4, m_5, and m_7. To determine the strength of a tunnel one first needs to incorporate the tunnel's extra bitconditions in the full differential path, and then count the remaining amount of freedom in the first round Q_t that is changed by the tunnel. Given its dependence on the differential path, a tunnel's strength can thus not be tabulated.

The most effective tunnel is $\mathcal{T}_8$. As indicated in the table, it affects after the first round only $Q_{25}, \ldots, Q_{64}$. Over these rounds, Wang's original differential paths have 20 bitconditions whereas the chosen-prefix collision differential paths that we manage to construct have approximately 27 bitconditions. It follows that, given enough tunnel strength, especially for $\mathcal{T}_7$ and $\mathcal{T}_8$, collision finding can be done efficiently.

4.5.2 Algorithm

Algorithm 4-3 Collision finding algorithm.

Given a full differential path $\mathtt{q}_{-3}, \ldots, \mathtt{q}_{64}$ consisting of only direct and backward bitconditions and the set $\mathcal{T}_1, \ldots, \mathcal{T}_8$ of tunnels from Table 4-5, perform the following steps:

1. Determine for all tunnels for which bits b the extra bitconditions as shown in Table 4-5 can be met. For each possible case, apply compatible bitconditions to enforce the extra bitconditions and change the bitconditions $\mathtt{q}_t[b]$ of the changed or affected $Q_t[b]$ in the first round from '.' to '0'.
2. Perform the steps below until a collision block has been found.
3. Select $Q_1, Q_2, Q_{13}, \ldots, Q_{16}$ such that $\mathtt{q}_1, \mathtt{q}_2, \mathtt{q}_{13}, \ldots, \mathtt{q}_{16}$ hold.
4. Compute m_1, Q_{17}.
5. If $\mathtt{q}_{17}$ holds and the rotation for $t = 16$ is successful, then proceed.
6. Store the set $\mathcal{Z}$ of all pairs (Q_1, Q_2) meeting $\mathtt{q}_1, \mathtt{q}_2$ that do not change m_1 and bits of Q_2 involved in $\mathtt{q}_3$.
7. For all $Q_3, \ldots, Q_7$ meeting $\mathtt{q}_3, \ldots, \mathtt{q}_7$ do:
8. Compute m_6, Q_{18}.
9. If $\mathtt{q}_{18}$ holds and the rotation for $t = 17$ is successful, then proceed.
10. For all $Q_8, \ldots, Q_{12}$ meeting $\mathtt{q}_8, \ldots, \mathtt{q}_{12}$ do:
11. Compute m_{11}, Q_{19}.
12. If $\mathtt{q}_{19}$ holds and the rotation for $t = 18$ is successful, then proceed.
13. For all (Q_1, Q_2) in $\mathcal{Z}$ do:
14. Compute m_0, Q_{20}.
15. If $\mathtt{q}_{20}$ holds and the rotation for $t = 19$ is successful, then proceed.
16. For all values of the bits of tunnels $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ do:
17. Set the bits to those values and compute m_5, Q_{21}.
18. If $\mathtt{q}_{21}$ holds and the rotation for $t = 20$ is successful, then proceed.
19. For all values of the bits of tunnels $\mathcal{T}_4, \mathcal{T}_5$ do:
20. Set the bits to those values and compute m_{10}, Q_{22}.
21. If $\mathtt{q}_{22}$ holds and the rotation for $t = 21$ is successful, then proceed.
22. For all values of the bits of tunnel $\mathcal{T}_6$ do:
23. Set the bits to those values and compute m_{15}, Q_{23}.
24. If $\mathtt{q}_{23}$ holds and the rotation for $t = 22$ is successful, then proceed.
25. For all values of the bits of tunnel $\mathcal{T}_7$ do:
26. Set the bits to those values and compute m_4, Q_{24}.
27. If $\mathtt{q}_{24}$ holds and the rotation for $t = 23$ is successful, then proceed.
28. For all values of the bits of tunnel $\mathcal{T}_8$ do:
29. Set the bits to those values and compute m_9, Q_{25}.
30. If $\mathtt{q}_{25}$ holds and the rotation for $t = 24$ is successful, then proceed.
31. Compute $m_0, \ldots, m_{15}$, $Q_{26}, \ldots, Q_{64}$ and $Q'_1, \ldots, Q'_{64}$.
32. If $\widehat{\delta Q_t} = Q'_t - Q_t$ agrees with $\mathtt{q}_t$ for $t = 61, 62, 63, 64$, return M, M'.

Computation of m_i and Q_i is performed at $t = i$ and $t = i - 1$, respectively.
We assume that the rotations in the first round have probability very close to 1 to be correct, and therefore do not verify them. This is further explained in Section 4.5.3.

In our construction we performed the collision finding using Algorithm 4-3. The conditions on the differential path imposed by Algorithm 4-3 can easily be met because, as mentioned in Section 4.3.1, forward and backward bitconditions in the differential path are interchangeable. Steps 10 through 15 of Algorithm 4-3 are its most computationally intensive part, in particular for the toughest differential paths in a chosen-prefix collision, so they should be optimized. Greater tunnel strength significantly reduces the time spent there, because after step 15 all tunnels are used.

In practice, in step c.4 of Algorithm 4-1 (cf. Section 4.4) we keep only those full differential paths for which tunnels $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_4$ and $\mathcal{T}_5$ satisfy a certain lower bound on their strength. Furthermore, of the full differential paths kept, we select those with the best properties such as high tunnel strength and a low number of bitconditions on $Q_{18}, \ldots, Q_{63}$.

4.5.3 Rotation bitconditions

As mentioned below Algorithm 4-3, it is assumed there that all rotations in the first round will be correct with probability very close to 1. In Algorithm 4-3, $Q_1, \ldots, Q_{16}$ are chosen in a non-sequential order and also changed at various steps in the algorithm. Ensuring correct rotations in the first round would be cumbersome and it would hardly avoid wasting time in a state where one or more rotations in the first round would fail due to the various tunnels. However, as shown in [27], if we use additional bitconditions $\mathfrak{q}_t[i]$ we can (almost) ensure correct rotations in the first round, thereby (almost) eliminating both the effort to verify rotations and the wasted computing time. This is explained below.

We use the notation introduced in Section 4.4.4. Given δT_t and δR_t it is easy to determine which partition (α, β) satisfies $RL((\alpha, \beta), RC_t) = \delta R_t$. The probability that this correct rotation holds is not necessarily $p_{(\alpha,\beta)}$ because it may be assumed that bitconditions $\mathfrak{q}_t$ and $\mathfrak{q}_{t+1}$ hold and these directly affect $R_t = Q_{t+1} - Q_t$ and thus $T_t = RR(R_t, RC_t)$. Hence, using bitconditions $\mathfrak{q}_t$ and $\mathfrak{q}_{t+1}$ we can try and increase the probability of a correct rotation in step t to (almost) 1 in the following way.

The other three partitions (of the four listed in Section 4.4.4) correspond to the incorrect rotations. Those partitions are of the form

$$(\hat{\alpha}, \hat{\beta}) = (\alpha - \lambda_0 2^{32-RC_t}, \beta + \lambda_0 2^{32-RC_t} + \lambda_{RC_t} 2^{32}), \quad \lambda_0, \lambda_{RC_t} \in \{-1, 0, +1\}$$

where either $\lambda_0 \neq 0$ or $\lambda_{RC_t} \neq 0$. They result in incorrect $\widehat{\delta R_t}$ of the form

$$\widehat{\delta R_t} = RL((\hat{\alpha}, \hat{\beta}), RC_t) = \delta R_t + \lambda_0 2^0 + \lambda_{RC_t} 2^{RC_t}.$$

They are caused by a carry when adding δT_t to T_t that does or does not propagate: from bit position $32 - RC_t - 1$ to $32 - RC_t$ for $\lambda_0 \neq 0$ and from bit position 31 to 32 for $\lambda_{RC_t} \neq 0$. Since we chose the partition (α, β) with highest probability, this usually means that we have to prevent instead of ensure those propagations in order to decrease the probability that $\lambda_0 \neq 0$ or $\lambda_{RC_t} \neq 0$.

To almost guarantee proper rotations in each step of Algorithm 4-3, additional bitconditions can be determined by hand, as shown in [27]. It was seen that

adding bitconditions on Q_t, Q_{t+1} around bit positions $31 - RC_t + i$ and lower helps preventing $\lambda_i \neq 0$. This can be automated using a limited brute-force search, separately handling the cases $\lambda_0 \neq 0$ and $\lambda_{RC_t} \neq 0$.

Let $i \in \{0, RC_t\}$. Given bitconditions $\mathfrak{q}_t, \mathfrak{q}_{t+1}$, we estimate $Pr[\lambda_i \neq 0 | \mathfrak{q}_t, \mathfrak{q}_{t+1}]$ by sampling a small set of $\widehat{Q}_t, \widehat{Q}_{t+1}$ satisfying $\mathfrak{q}_t, \mathfrak{q}_{t+1}$, e.g. of size 2^{11}, and determining the fraction where $\lambda_i = \mathrm{NAF}(\widehat{\delta R_t} - \delta R_t)[\![i]\!] \neq 0$ using

$$\widehat{\delta R_t} = RL(RR(\widehat{Q}_{t+1} - \widehat{Q}_t, RC_t) + \delta T_t, RC_t).$$

Using this approach, we estimate the probability that $\lambda_i = 0$ by selecting a small search bound B and exhaustively trying all combinations of additional bitconditions on $Q_t[b], Q_{t+1}[b]$ for $b = 31 - RC_t + i - B, \ldots, 31 - RC_t + i$. Finally, if there are any bitconditions $(\mathfrak{q}'_t, \mathfrak{q}'_{t+1})$ for which $Pr[\lambda_i \neq 0 | \mathfrak{q}'_t, \mathfrak{q}'_{t+1}]$ is negligible, we select the pair $(\mathfrak{q}'_t, \mathfrak{q}'_{t+1})$ that leads to the smallest number of additional bitconditions and for which $Pr[\lambda_0 = \lambda_{RC_t} = 0 | \mathfrak{q}_{t-1}, \mathfrak{q}'_t]$ and $Pr[\lambda_0 = \lambda_{RC_t} = 0 | \mathfrak{q}'_{t+1}, \mathfrak{q}_{t+2}]$ do not decrease significantly for step $t-1$ and $t+1$, respectively.

4.6 Implementation remarks

Our software to construct chosen-prefix collisions consists of five main components that perform the following tasks:

1. the birthday search (with a special implementation for Sony's PlayStation 3);
2. forward extension of a given set of partial lower differential paths by a given step t, saving only the paths with the fewest bitconditions;
3. backward extension of a given set of partial upper differential paths by a given step t, saving only the paths with the fewest bitconditions;
4. attempt to connect all combinations of given lower and upper differential paths;
5. coordinate the four earlier tasks by preparing the required inputfiles, collect the outputs, and search for near-collision blocks.

These tasks are carried out as described in the earlier sections. A few remarks are in order. The first task is the most computationally expensive one and consists mostly of simple applications of the MD5 compression function. It turns out that the Cell processor, contained in the PlayStation 3 game console, can be made to perform this task about 30 times faster than a regular 32-bit PC core. More details on the peculiarities of the PlayStation 3 implementation are described in Section 5.2.2.

For the second and third task we exhaustively try all limited weight BSDRs of δQ_t, all possible δF_t's, and we use the highest-probability rotation. We keep at most a preset number of paths with the lowest number of bitconditions that have a preset minimum total strength over tunnels $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_4$, and $\mathcal{T}_5$. Each of the programs is designed to execute several separate but parallel threads.

4.7 Complexity analysis

The overall complexity of the chosen-prefix collision attack depends on the parameters used for the birthday search and the construction of pairs of near-collision blocks. This involves various trade-offs and is described in this section.

The birthday search complexity depends on the parameter w (defining the family of differential paths), the upper bound on the number r of pairs of near-collision blocks, the size 2^{64+k} of the search space, and the amount of available memory M. For various parameter choices of r, k and w we have tabulated the heuristically determined expected birthday search complexities and memory requirements in Appendix C (in practice it is advisable to use a small factor more memory than required to achieve $C_{\text{coll}} \ll C_{\text{tr}}$). Given r, w and M, the optimal value for k and the resulting birthday complexity can thus easily be looked up. When r is left free, one can balance the birthday complexity and the complexity of constructing r pairs of near-collision blocks.

Each pair of near-collision blocks requires construction of a set of full differential paths followed by the actual construction of the pair of near-collision blocks. The complexity of the former construction depends on several parameter choices, such as the size of the sets of lower and upper differential paths, and the restraints used when selecting BSDRs for a δQ_t. Naturally, a higher overall quality of the resulting complete differential paths, i.e., a low number of overall bitconditions and a high total tunnel strength, generally results when more effort is put into the construction. For practical purposes we have found parameters sufficient for almost all cases (as applicable to the chosen-prefix collision attack) that have an average total complexity equivalent to roughly 2^{35} MD5 compressions.

The complexity of the collision finding, i.e., the construction of a pair of near-collision blocks, depends on the parameter w, the total tunnel strength and the number of bitconditions in the last 2.5 rounds. For small $w = 0, 1, 2$ and paths based on Table 4-1, the construction requires on average roughly the equivalent of 2^{34} MD5 compressions. Combined with the construction of the differential paths, this leads to the rough overall estimate of about $2^{35.6}$ MD5 compressions to find a single pair of near-collision blocks for a chosen-prefix collision attack.

With $w = 2$ and optimizing for overall complexity this leads to the optimal parameter choices $r = 9$ and $k = 0$. For these choices, the birthday search cost is about 2^{37} MD5 compressions and constructing the $r = 9$ pairs of near-collision blocks costs about $2^{38.8}$ MD5 compressions. The overall complexity is thus estimated at roughly $2^{39.1}$ MD5 compressions, which takes about 35 hours on a single PC-core. For this parameter choice the memory requirements for the birthday search are very low, even negligible compared to the several hundreds of MBs required for the construction of the differential paths.

With more specific demands, such as a small number r of near-collision blocks possibly in combination with a relatively low M, the overall complexity will increase. As an example, our rogue CA construction required at most $r = 3$ near-collision blocks, and using $M = 5$TB this results in an overall complexity of about 2^{49} MD5 compressions.

4.8 Single-block chosen-prefix collision

Using a different approach it is even possible to construct a chosen-prefix collision using only a single pair of near-collision blocks. Together with 84 birthday bits, the chosen-prefix collision-causing appendages are only $84+512 = 596$ bits long. This approach is based on an even richer family of differential paths that allows elimination using a single pair of near-collision blocks of a set of δIHVs that is bounded enough so that finding the near-collision blocks is still feasible, but large enough that such a δIHV can be found efficiently by a birthday search. Instead of using the family of differential paths based on $\delta m_{11} = \pm 2^i$, we use the fastest known collision attack for MD5 and vary the last few steps to find a large family of differential paths.

We first present a new collision attack for MD5 with complexity of approximately 2^{16} MD5 compressions improving upon the $2^{20.96}$ MD5 compressions required in [35]. Our starting point is the partial differential path for MD5 given in Table 4-6. It is based on message differences $\delta m_2 = 2^8$, $\delta m_4 = \delta m_{14} = 2^{31}$ and $\delta m_{11} = 2^{15}$ which is very similar to those used by Wang et al. in [34] for the first collision attack against MD5. This partial differential path can be used for a near-collision attack with complexity of approximately $2^{14.8}$ MD5 compressions.

This leads in the usual fashion to an identical-prefix collision attack for MD5 that requires approximately 2^{16} MD5 compressions, since one has to do it twice: first to add differences to δIHV and then to eliminate them again. It should be noted that usually bitconditions are required on the IHV and IHV$'$ between the two collision blocks which imply an extra factor in complexity. In the present case, however, we can construct a large set of differential paths for the second near-collision block that will cover all bitconditions that are likely to occur, thereby avoiding the extra complexity.

By properly tuning the birthday search, the same partial differential path leads to the construction of a single near-collision block chosen-prefix collision for MD5. By varying the last steps of the differential path and by allowing the collision finding complexity to grow by a factor of about 2^{26}, we have found a set $\mathcal{S}$ of about $2^{23.3}$ different $\delta\text{IHV} = (\delta a, \delta b, \delta c, \delta d)$ of the form $\delta a = -2^5$, $\delta d = -2^5 + 2^{25}$, $\delta c = -2^5 \bmod 2^{20}$ that can be eliminated. Such δIHVs can be found using an 84-bit birthday search with step function $f : \{0,1\}^{84} \to \{0,1\}^{84}$ of the form

$$f(x) = \begin{cases} \phi(\text{MD5compress}(\text{IHV}, B\|x) + \delta\widehat{\text{IHV}}) & \text{for } \sigma(x) = 0 \\ \phi(\text{MD5compress}(\text{IHV}', B'\|x)) & \text{for } \sigma(x) = 1, \end{cases}$$

where $\delta\widehat{\text{IHV}}$ is of the required form, $\sigma : x \mapsto \{0,1\}$ is a balanced selector function and $\phi(a,b,c,d) \mapsto a\|d\|(c \bmod 2^{20})$. There are $2^{128-84} = 2^{44}$ possible δIHVs of this form, of which only about $2^{23.3}$ are in the allowed set $\mathcal{S}$. It follows that a birthday collision has probability $p = 2^{23.3}/(2^{44} \cdot 2) = 2^{-21.7}$ to be useful, where the additional factor 2 stems from the fact that different prefixes are required.

A useful birthday collision can be expected after $\sqrt{\pi 2^{84}/(2p)} \approx 2^{53.2}$ MD5 compressions, requires 400MB of storage and takes about 3 days on 215 PS3s.

Table 4-6. Partial differential path for fast near-collision attack.

t	δQ_t	δF_t	δW_t	δT_t	δR_t	RC_t
26	-2^8					
27	0					
28	0					
29	0	0	2^8	0	0	9
$30-33$	0	0	0	0	0	$\cdot$
34	0	0	2^{15}	2^{15}	2^{31}	16
35	2^{31}	2^{31}	2^{31}	0	0	23
36	2^{31}	0	0	0	0	4
37	2^{31}	2^{31}	2^{31}	0	0	11
$38-46$	2^{31}	2^{31}	0	0	0	$\cdot$
47	2^{31}	2^{31}	2^8	2^8	2^{31}	23
48	0	0	0	0	0	6
49	0	0	0	0	0	10
50	0	0	2^{31}	0	0	15
$51-59$	0	0	0	0	0	$\cdot$
60	0	0	2^{31}	2^{31}	-2^5	6
61	-2^5	0	2^{15}	2^{15}	2^{25}	10
62	-2^5+2^{25}	0	2^8	2^8	2^{23}	15
63	$-2^5+2^{25}+2^{23}$	2^5-2^{23}	0	2^5-2^{23}	$2^{26}-2^{14}$	21
64	$-2^5+2^{25}+2^{23}+2^{26}-2^{14}$					

Partial differential path for $t = 29, \ldots, 63$ using message differences $\delta m_2 = 2^8$, $\delta m_4 = \delta m_{14} = 2^{31}$, $\delta m_{11} = 2^{15}$. The probability that it is satisfied is approximately $2^{-14.5}$.

The expected complexity of finding the actual near-collision blocks is bounded by about $2^{14.8+26} = 2^{40.8}$ MD5 compressions.

In Table 4-7 two 128-byte messages are given both consisting of a 52-byte chosen prefix and a 76-byte single-block chosen-prefix collision suffix and with colliding MD5 hash value `D320B6433D8EBC1AC65711705721C2E1`.

5 Applications of chosen-prefix collisions

When exploiting collisions in real world applications two major obstacles must be overcome.

- The problem of *meaningful collisions*. Given current methods, collisions require appendages consisting of unpredictable and mostly uncontrollable bitstrings. These must be hidden in the usually heavily formatted application data structure without raising suspicion.
- The problem of *realistic attack scenarios.* As we do not have effective attacks against MD5's (second) preimage resistance but only collision attacks, we cannot target existing MD5-values. In particular, the colliding data struc-

Table 4-7. Example single-block chosen-prefix collision.

Message 1

```
4F 64 65 64 20 47 6F 6C 64 72 65 69 63 68 0A 4F
64 65 64 20 47 6F 6C 64 72 65 69 63 68 0A 4F 64
65 64 20 47 6F 6C 64 72 65 69 63 68 0A 4F 64 65
64 20 47 6F D8 05 0D 00 19 BB 93 18 92 4C AA 96
DC E3 5C B8 35 B3 49 E1 44 E9 8C 50 C2 2C F4 61
24 4A 40 64 BF 1A FA EC C5 82 0D 42 8A D3 8D 6B
EC 89 A5 AD 51 E2 90 63 DD 79 B1 6C F6 7C 12 97
86 47 F5 AF 12 3D E3 AC F8 44 08 5C D0 25 B9 56
```

Message 2

```
4E 65 61 6C 20 4B 6F 62 6C 69 74 7A 0A 4E 65 61
6C 20 4B 6F 62 6C 69 74 7A 0A 4E 65 61 6C 20 4B
6F 62 6C 69 74 7A 0A 4E 65 61 6C 20 4B 6F 62 6C
69 74 7A 0A 75 B8 0E 00 35 F3 D2 C9 09 AF 1B AD
DC E3 5C B8 35 B3 49 E1 44 E8 8C 50 C2 2C F4 61
24 4A 40 E4 BF 1A FA EC C5 82 0D 42 8A D3 8D 6B
EC 89 A5 AD 51 E2 90 63 DD 79 B1 6C F6 FC 11 97
86 47 F5 AF 12 3D E3 AC F8 44 08 DC D0 25 B9 56
```

tures must be generated simultaneously, along with their shared hash, by the adversary.

In Section 5.1 several chosen-prefix collision applications are surveyed where these problems are addressed with varying degrees of success. Sections 5.2, 5.3, and 5.4 describe the three most prominent applications in more detail.

5.1 A survey of potential applications

We mention some potential applications of chosen-prefix collisions.

Digital certificates. Given how heavily they rely on cryptographic hash functions, digital certificates are the first place to look for applications of chosen-prefix collisions. Two X.509 certificates are said to collide if their to-be-signed parts have the same hash and consequently their digital signatures, as provided by the CA, are identical. In earlier work (cf. [18]) we have shown how identical-prefix collisions can be used to construct colliding X.509 certificates with different RSA moduli but identical Distinguished Names. Here the RSA moduli absorbed the random-looking near-collision blocks, thus inconspicuously and elegantly solving the meaningfulness problem. Allowing different Distinguished Names required chosen-prefix collisions, as shown in [29]. The certificates resulting from both constructions do not contain spurious bits, so superficial inspection at bit level of either of the certificates will not reveal

the existence of a sibling certificate that collides with it signature-wise. Nevertheless, for these constructions to work the entire to-be-signed parts, and thus the signing CA, must be fully under our own control, thereby limiting the practical attack potential.
A related but in detail rather different construction was carried out in collaboration with Alexander Sotirov, Jacob Appelbaum, David Molnar, and Dag Arne Osvik, as reported on `http://www.win.tue.nl/hashclash/rogue-ca/` and in [30]. Although in practice a certificate's to-be-signed part cannot be for 100% under control of the party that submits the certification request, for some commercial CAs (that still used MD5 for their digital signature generation) the entire to-be-signed part could be predicted reliably enough to make the following guess-and-check approach practically feasible: prepare the prefix of the to-be-signed part of a legitimate certification request including a guess for the part that will be included by the CA upon certification, prepare a rogue to-be-signed prefix, determine different collision-causing and identical collision-maintaining appendages to complete two colliding to-be-signed parts, and submit the legitimate one for certification. If upon receipt of the legitimate certificate the guess turns out to have been correct, then the rogue certificate can be completed by pasting the CA's signature of the legitimate data onto the rogue data: because the data collide, the signature will be equally valid for both. Otherwise, if the guess is incorrect, another attempt is made. Using this approach we managed (upon the 4th attempt) to trick a commercial CA into providing a signature valid for a rogue CA certificate. For the intricate details of the construction we refer to Section 5.2.
A few additional remarks about this construction are in order here. We created not just a rogue certificate, but a rogue CA certificate, containing identifying information and public key material for a rogue CA. The private key of this rogue CA is under our control. As the commercial CA's signature is valid for the rogue CA certificate, all certificates issued by the rogue CA are trusted by anybody trusting the commercial CA. As the commercial CA's root certificate is present in all major browsers, this gives us in principle the possibility to impersonate any certificate owner. This is certainly a realistic attack scenario. The price that we have to pay is that the meaningfulness problem is only adequately – and most certainly not elegantly – solved: as further explained in the next paragraph, one of the certificates contains a considerable number of suspicious-looking bits.
To indicate that a certificate is a CA certificate, a certain bit has to be set in the certificate's to-be-signed-part. According to the X.509v3 standard [6], this bit comes after the public key field. It is unlikely that a commercial CA will accept a certification request where the CA bit is set. Therefore, the bit must not be set in the legitimate request. For our rogue CA certificate construction, the fact that the two to-be-signed parts must contain a different bit *after* the public key field causes an incompatibility with our 'usual' colliding certificate construction as in [18] and [29]. In that construction the collision-causing appendages correspond to the high order bits of RSA moduli, and they are followed by identical collision-maintaining appendages that

transform the two appendages into valid RSA moduli. Anything following after the moduli must remain identical lest the collision property goes lost. As a consequence, the appendages on the rogue side can no longer be hidden in the public key field and some other field must be found for them. Such a field may be specially defined for this purpose, or an existing (proprietary) extension may be used. The Netscape Comment extension is a good example of the latter, as we found that it is ignored by the major certificate processing software. The upshot is, however, that as the appendages have non-negligible length, it will be hard to define a field that will not look suspicious to someone who looks at the rogue certificate at bit level.

Colliding documents. In [7] (see also [10]) it was shown how to construct a pair of PostScript files that collide under MD5, but that display different messages when viewed or printed. These constructions use identical-prefix collisions, and therefore they have to rely on the presence of both messages in each of the colliding files and on macro-functionalities of the document format used. Obviously, this raises suspicion upon inspection at bit level. With chosen-prefix collisions, one message per colliding document suffices and macro-functionalities are no longer required. For example, using a document format that allows insertion of color images (such as Microsoft Word or Adobe PDF), inserting one message per document, two documents can be made to collide by appending carefully crafted color images after the messages. A short one pixel wide line will do – for instance hidden inside a layout element, a company logo, or a nicely colored barcode – and preferably scaled down to hardly visible size (or completely hidden from view, as possible in PDF). An extension of this construction is presented in the paragraphs below and set forth in detail in Section 5.3.

Hash based commitments. Kelsey and Kohno [15] presented a method to first commit to a hash value, and next to construct faster than by a trivial pre-image attack a document with the committed hash value, and with any message of one's choice as a prefix. The method applies to any Merkle-Damgård hash function, such as MD5, that given an IHV and a suffix produces some IHV. Omitting details involving message lengths and Merkle-Damgård strengthening, the idea is to commit to a hash value based on an IHV at the root of a tree, either that IHV itself or calculated as the hash of that IHV and some suffix at the root. The tree is a complete binary tree and is calculated from its leaves up to the root, so the IHV at the root will be one of the last values calculated. This is done in such a way that each node of the tree is associated with an IHV along with a suffix that together hash to the IHV associated with the node's parent. Thus, two siblings have IHVs and suffixes that collide under the hash function. The IHVs at the leaves may be arbitrarily chosen but are, preferably, all different. Given a prefix of one's choice one performs a brute-force search for a suffix that, when appended to the prefix and along with the standard IHV, results in the IHV at one of the leaves (or nodes) of the tree. Appending the suffixes one encounters on one's way from that leave or node to the root, results in a final message with the desired prefix and committed hash value.

Originally based on a birthday search, the construction of the tree can be done more efficiently by using chosen-prefix collisions to construct sibling node suffixes based on their IHVs. For MD5, however, it remains far from feasible to carry out the entire construction in practice. In a variant that *is* feasible, one commits to a prediction by publishing its hash value. In due time one reveals the correct prediction, chosen from among a large enough preconstructed collection of documents that, due to tree-structured chosen-prefix collision appendages, all share the same published hash value. In section 5.3 we present an example involving 12 documents.

Software integrity checking. In [14] and [22] it was shown how any existing MD5 collision, such as the ones originally presented by Xiaoyun Wang at the Crypto 2004 rump session, can be abused to mislead integrity checking software that uses MD5. A similar application, using freshly made collisions, was given on `http://www.mathstat.dal.ca/~selinger/md5collision/`. See also [9]. As shown on `http://blog.didierstevens.com/2009/01/17/` this can even be done within the framework of Microsoft's Authenticode code signing program. All these results use identical-prefix collisions and, similar to the colliding PostScript application mentioned earlier, differences in the colliding inputs are used to construct deviating execution flows.

Chosen-prefix collisions allow a more elegant approach, since common operating systems ignore bitstrings that are appended to executables: the programs will run unaltered. Thus, using tree-structured chosen-prefix collision appendages as above, any number of executables can be made to have the same MD5 hash value or MD5-based digital signature. See Section 5.4 for an example.

One can imagine two executables: a 'good' one (say Word.exe) and a bad one (the attacker's Worse.exe). A chosen-prefix collision for those executables is computed, and the collision-causing bitstrings are appended to both executables. The resulting altered file Word.exe, functionally equivalent to the original Word.exe, can be offered to a code signing program such as Microsoft's Authenticode and receive an 'official' MD5-based digital signature. This signature will then be equally valid for the attacker's Worse.exe, and the attacker might be able to replace Word.exe by his Worse.exe (renamed to Word.exe) on the appropriate download site. This construction affects a common functionality of MD5 hashing and may pose a practical threat. It also allows people to get many executables signed at once and for free by getting a single executable signed, bypassing verification of any kind (e.g. authenticity, quality, compatibility, non-spyware, non-malware) by the signing party.

Computer forensics. In computer forensics so-called *hash sets* are used to quickly identify known files. For example, when a hard disk is seized by law enforcement officers, they may compute the hashes of all files on the disk, and compare those hashes to hashes in existing hash sets: a whitelist (for known harmless files such as operating system and other common software files) and a blacklist (for previously identified harmful files). Only files whose hashes do not occur in either hash set have to be inspected further. A useful feature

of this method of recognizing files is that the file name itself is irrelevant, since only the content of the file is hashed.

MD5 is a popular hash function for this application. Examples are NIST's National Software Reference Library Reference Data Set (`http://www.nsrl.nist.gov/`) and the US Department of Justice's Hashkeeper application (`http://www.usdoj.gov/ndic/domex/hashkeeper.htm`).

A conceivable, and rather obvious, attack on this application of hashes is to produce a harmless file (e.g. an innocent picture) and a harmful one (e.g. an illegal picture), and insert collision blocks that will not be noticed by common application software or human viewers. In a learning phase the harmless file might be submitted to the hash set and thus the common hash may end up on the whitelist. The harmful file will be overlooked from then on.

Peer to peer software. Hash sets are also used in peer to peer software. A site offering content may maintain a list of pairs (file name, hash). The file name is local only, and the peer to peer software uniquely identifies the file's content by means of its hash. Depending on how the hash is computed such systems may be vulnerable to a chosen-prefix attack. Software such as eDonkey and eMule use MD4 to hash the content in a two stage manner: the identifier of the content $c_1\|c_2\|\ldots\|c_n$ is $\mathrm{MD4}(\mathrm{MD4}(c_1)\|\ldots\|\mathrm{MD4}(c_n))$, where the chunks c_i are about 9 MB each. One-chunk files, i.e., files not larger than 9 MB, are most likely vulnerable; whether multi-chunk files are vulnerable is open for research. We have not worked out the details of a chosen-prefix collision attack against MD4, but this seems very well doable by adapting our methods and should result in an attack that is considerably faster than our present one against MD5.

Content addressed storage. In recent years *content addressed storage* is gaining popularity as a means of storing fixed content at a physical location of which the address is directly derived from the content itself. For example, a hash of the content may be used as the file name. See [24] for an example. Clearly, chosen-prefix collisions can be used by an attacker to fool such storage systems, e.g. by first preparing colliding pairs of files, by then storing the harmless-looking first one, and later overwriting it with the harmful second one.

Further investigations are required to assess the impact of chosen-prefix collisions. We leave it to others to study to what extent commonly used protocols and message formats such as TLS, S/MIME (CMS), IPSec and XML Signatures (see [1] and [13]) allow insertion of random looking data that may be overlooked by some or all implementations. The threat posed by identical-prefix collisions is not well understood either: their application may be more limited, but for MD5 they can be generated almost instantaneously and thus allow real-time attacks on the execution of cryptographic protocols, and, more importantly, for SHA-1 they may soon be feasible.

5.2 Creating a rogue Certification Authority certificate

This section contains an in-depth discussion of the practical dangers posed by rogue Certification Authority certificates, followed by a detailed description of how we managed to construct such a certificate.

The work reported here was carried out in close collaboration with Alexander Sotirov and Dag Arne Osvik, and was triggered by email exchanges with Alexander Sotirov, Jacob Appelbaum and David Molnar.

5.2.1 Attack potential of rogue CA certificates

In the conference version [29, Section 4.1] of this paper we daydreamed:

> "Ideally, a realistic attack targets the core of PKI: provide a relying party with trust, beyond reasonable cryptographic doubt, that the person indicated by the Distinguished Name field has exclusive control over the private key corresponding to the public key in the certificate. The attack should also enable the attacker to cover his trails."

Our dream scenario has been, mostly, realized with the construction of a rogue CA certificate. With the private key of a CA under our control, and the public key appearing in a certificate with a valid signature of a commercial CA that is trusted by all major browsers, we can create 'trusted' certificates at will. When scrutinized at bit level, however, our rogue CA certificate may look suspicious which may, ultimately, expose us. Bit level inspection is not something many users will engage in – if they know the difference between `https` and `http` to begin with – and, obviously, the software that is supposed to inspect a certificate's bits is expertly guided around the suspicious ones. So, it may be argued that our construction has a non-negligible attack potential. Below we discuss some possibilities in this direction. Upfront, however, we like to point out that our rogue CA is nothing more than a proof of concept that is incapable of doing much harm, because it expired, on purpose, in September of 2004, i.e., more than 4 years before it was created.

Any website secured using TLS can be impersonated using a rogue certificate issued by a rogue CA. This is irrespective of which CA issued the website's true certificate and of any property of that certificate (such as the hash function it is based upon – SHA-256 is not any better in this context than MD4). Combined with redirection attacks where `http` requests are redirected to rogue web servers, this leads to virtually undetectable phishing attacks.

But any application involving a Certification Authority that provides MD5-based certificates with sufficiently predictable serial number and validity period may be vulnerable. In contexts different from TLS this may include signing or encryption of e-mail or software, non-repudiation services, etc.

As pointed out earlier, bit-level inspection of our rogue CA certificate will reveal a relatively large number of bits that may look suspicious – and that *are* suspicious. This could have been avoided if we had chosen to create a rogue certificate for a regular website, as opposed to a rogue CA certificate, because in

that case we could have hidden all collision causing bits inside the public keys. Nevertheless, even if each resulting certificate by itself looks unsuspicious, as soon as a dispute arises, the rogue certificate's legitimate sibling can be located with the help of the CA, and the fraud becomes apparent by putting the certificates alongside, thus exposing the party responsible for the fraud.

Our attack relies on our ability to predict the content of the certificate fields inserted by the CA upon certification: if our prediction is correct with non-negligible probability, a rogue certificate can be generated with the same non-negligible probability. Irrespective of the weaknesses, known or unknown, of the cryptographic hash function used for digital signature generation, our type of attack becomes effectively impossible if the CA adds a sufficient amount of fresh randomness to the certificate fields before the public key fields. Relying parties, however, cannot verify this randomness and also the trustworthiness of certificates should not crucially depend on such secondary and circumstantial aspects. We would be in favor of a more fundamental solution – along with a strong cryptographic hash function – possibly along the lines as proposed in [11]. Generally speaking, it is advisable not to sign data that is completely determined by some other party. Put differently, a signer should always make a few trivial and unpredictable modifications before digitally signing a document provided by someone else.

Based on our previous work [29], the issue in the previous paragraph was recognized and the possibility of the attack presented in this paper anticipated in the catalogue [3] of algorithms suitable for the German Signature Law ('Signaturgesetz'). This catalogue includes conditions and time frames for cryptographic hash algorithms to be used in legally binding digital signatures in Germany. One of the changes introduced in the 2008 version of the catalog is an explicit condition on the usage of SHA-1: only until 2010, and only for so-called "qualified certificates" that contain at least 20 bits of entropy in their serial numbers. We are grateful to Prof. Werner Schindler of the BSI for bringing this to our attention and for confirming that this change was introduced to thwart exactly the type of rogue certificates that we present here for MD5. Currently the complexity of identical-prefix collisions for SHA-1 is rumored to hover around 2^{52} compression function calls. It is unlikely it will again inch up from there, but may instead tumble down even further: given the flux in SHA-1 cryptanalysis, 20 bits of additional entropy may not suffice for much longer to even approach SHA-1's intended 80 bits of security.

We stress that our attack on MD5 is not a preimage or second preimage attack. We cannot create a rogue certificate having a signature in common with a certificate that was not especially crafted using our chosen-prefix collision. In particular, we cannot target any existing, independently created certificate and forge a rogue certificate that shares its digital signature with the digital signature of the targeted certificate. However, given any certificate with an MD5-based digital signature, a relying party cannot easily recognize if it is trustworthy or, on the contrary, crafted by our method. Therefore we repeat our urgent recommendation not to use MD5 for new X.509 certificates. How existing MD5

certificates should be handled is a subject of further research. We also urgently recommend to reconsider usage of MD5 in other applications. Proper alternatives are available; but compatibility with existing applications is obviously another matter. Given potential developments related to SHA-1 (see [4] and `http://www.iaik.tugraz.at/content/research/krypto/sha1/`) we feel that usage of SHA-1 in certificate generation should be reassessed as well.

5.2.2 Certificate construction

Our first colliding X.509 certificate construction was based on an identical-prefix collision, and resulted in two certificates with different public keys, but identical Distinguished Name fields [18]. As a first application of chosen-prefix collisions we showed how the Distinguished Name fields could be chosen differently as well [29]. In this section we describe the details of a colliding certificate construction that goes one step further by also allowing different "basic constraints" fields. This allows us to construct one of the certificates as an ordinary website certificate, but the other one as a CA certificate. As already pointed out in Section 5.1, this additional difference required a radical departure from our traditional construction method from [18] and [29]. Furthermore, unlike our previous colliding certificate constructions where the CA was under our control, a commercial CA provided the digital signature for the (legitimate) website certificate. This required us to sufficiently accurately predict its serial number and validity period well before the certification request was submitted to the signing CA.

We exploited the following weaknesses of the commercial CA that carried out the legitimate certification request:

- Its usage of the cryptographic hash function MD5 to generate digital signatures for new certificates.
- Its fully automated way to process online certification requests that fails to recognize anomalous behavior of requesting parties.
- Its usage of sequential serial numbers and its usage of validity periods that are determined entirely by the date and time in seconds at which the certification request is processed.
- Its failure to enforce, by means of the "basic constraints" field in its own certificate, a limit on the length of the chain of certificates that it can sign.

The first three points are further discussed below. The last point, if properly handled, could have crippled our rogue CA certificate but does not affect its construction. A certificate contains a "basic constraints" field where a bit is set to indicate if the certificate is a CA certificate. With the bit set, a "path length constraint" subfield may be present, specifying an integer that indicates how many CAs may occur in the chain between the CA certificate in question and end-user certificates. The commercial CA that we interacted with failed to use this option in its own certificate, implying that any number of intermediate CAs is permitted. If the "path length constraint" would have been present and set at 0 (zero), then our rogue CA certificate could still have been constructed. But whether or not the rogue CA certificate or certificates signed by it can

then also be used depends on (browser-)software actually checking the "path length constraint" subfields in chains of certificates. Thus a secondary "defense in depth" mechanism was present that could have foiled our attack, but failed to do so simply because it was not used.

Before describing the construction of the colliding certificates, we briefly discuss the parameter choices used for the chosen-prefix collision search. The 2048-bit upper bound on the length of RSA moduli, as enforced by some CAs, combined with other limitations of our certificate construction, implied we could allow for at most 3 near-collision blocks. Opting for the least difficult possibility (namely, 3 near-collision blocks), we had to decide on values for k and the aimed for value for w, determining the costs of the birthday search and the near-collision block constructions (cf. Sections 4.2 and 4.1), respectively. Obviously, our choices were influenced by our computational resources, namely a cluster of 215 PlayStation 3 (PS3) game consoles. When running Linux on a PS3, applications have access to 6 Synergistic Processing Units (SPUs), a general purpose CPU, and about 150MB of RAM per PS3. For the birthday search, the 6×215 SPUs are computationally equivalent to approximately 8600 regular 32-bit cores, due to each SPU's 4×32-bit wide SIMD architecture. The other parts of the chosen-prefix collision construction are not suitable for the SPUs, but we were able to use the 215 PS3 CPUs for the construction of the actual near-collision blocks. With these resources, the choice $w = 5$ still turned out to be acceptable despite the 1000-fold increase in the cost of the actual near-collision block construction. This is the case even for the hard cases with many differences between IHV and IHV$'$: as a consequence the differential paths contain many bitconditions, which leaves little space for the tunnels, thereby complicating the near-collision block construction.

For the targeted 3 near-collision blocks, the entries for $w = 5$ in the first table in Appendix C show the time-memory tradeoff when the birthday search space is varied with k. With 150MB at our disposal per PS3, for a total of about 30GB, we decided to use $k = 8$ as this optimizes the overall birthday complexity for the plausible case that the birthday search takes $\sqrt{2}$ times longer than expected. The resulting overall chosen-prefix collision construction takes on average less than a day on the PS3-cluster. In theory we could have used 1TB (or more) of hard drive space, in which case it would have been optimal to use $k = 0$ for a birthday search of about 20 PS3 days.

We summarize the construction of the colliding certificates in the sequence of steps below, and then describe each step in more detail.

1. Construction of templates for the two to-be-signed parts, as outlined in Figure 4. Note that we distinguish between a 'legitimate' to-be-signed part on the left hand side, and a 'rogue' to-be-signed part on other side.
2. Prediction of serial number and validity period for the legitimate part, thereby completing the chosen prefixes of both to-be-signed parts.
3. Computation of the two different collision-causing appendages.
4. Computation of a single collision-maintaining appendage that will be appended to both sides, thereby completing both to-be-signed parts.

5. Preparation of the certification request for the legitimate to-be-signed part.
6. Submission of the certification request and receipt of the new certificate.
7. If serial number and validity period of the newly received certificate are as predicted, then the rogue certificate can be completed. Otherwise return to Step 2.

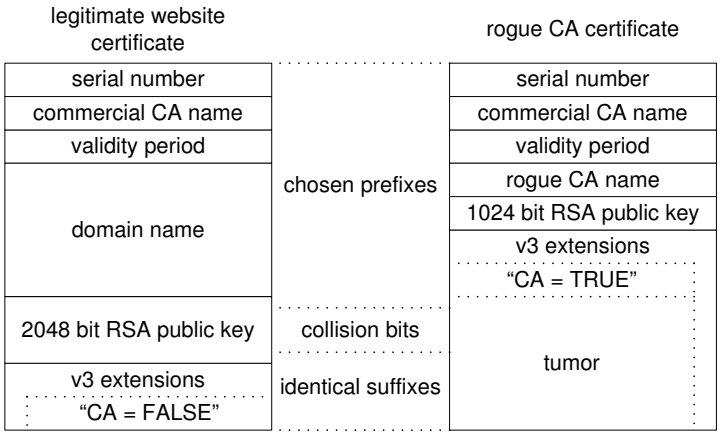

Fig. 4. The to-be-signed parts of the colliding certificates.

Step 1. Templates for the to-be-signed parts. In this step all bits are set in the two to-be-signed parts, except for bits that will be determined in later steps. For the latter bits space will be reserved here. On the legitimate side the parts to be filled in later are the predictions for the serial number and validity period, and most bits of the public key. On the rogue side the largest part of the content of an extension field of the type "Netscape Comment" is for the moment left undetermined. The following roughly describes the sequence of steps.

- On the legitimate side, the chosen prefix contains space for serial number and validity period, along with the exact Distinguished Name of the commercial CA where the certification request will be submitted. This is followed by a subject Distinguished Name that contains a legitimate website domain name (owned by one of us) consisting of as many characters as allowed by the commercial CA (in our case 64), and concluded by the first 208 bits of an RSA modulus, the latter all chosen at random after the leading '1'-bit. These sizes were chosen in order to have as many corresponding bits as possible on the rogue side, while fixing as few bits as possible of the RSA modulus on the legitimate side (see Step 4 for the reason why).
- The corresponding bits on the rogue side contain an arbitrarily chosen serial number, the same commercial CA's Distinguished Name, an arbitrarily chosen validity period (actually chosen as indicating "August 2004", to avoid

abuse of the rogue certificate), a short rogue CA name, a 1024-bit RSA public key generated using standard software, and the beginning of the X.509v3 extension fields. One of these fields is the "basic constraints" field, a bit that we set to indicate that the rogue certificate will be a CA certificate (in Figure 4 this bit is denoted by "CA=TRUE").
- At this point the entire chosen prefix is known on the rogue side, but on the legitimate side predictions for the serial number and validity period still need to be inserted. That will be done in Step 2.
- The various field sizes were selected so that on both sides the chosen prefixes are now 96 bits short of the same MD5 block boundary. On both sides these 96 bit positions are reserved for the birthday bits. As only $64 + k = 72$ birthday search bits per side will be needed (and appended in Step 3) the first 24 bits at this point are set to 0. On the legitimate side these 96 bits are part of the RSA modulus, on the rogue side they are part of an extension field of the type "Netscape Comment", denoted as 'tumor' in Figure 4.
- From here on forward, everything that goes to the rogue side is part of the "Netscape Comment" field, as it is not meaningful for the rogue CA certificate but only appended to cause and maintain a collision with bits added to the legitimate side. On the legitimate side we first make space for 3 near-collision blocks of 512 bits each (calculated in Step 3) and for 208 bits used to complete a 2048-bit RSA modulus (determined in Step 4), and then set the RSA public exponent (for which we took the common choice 65537) and the X.509v3 extensions including the bit indicating that the legitimate certificate will be an end-user certificate (in Figure 4 denoted by "CA=FALSE").

Step 2. Prediction of serial number and validity period. Based on repeated certification requests submitted to the targeted commercial CA, it turned out that the validity period can very reliably be predicted as the period of precisely one year plus one day, starting exactly six seconds after a request is submitted. So, to control that field, all we need to do is select a validity period of the right length, and submit the legitimate certification request precisely six seconds before it starts. Though occasional accidents may happen in the form of one-second shifts, this was the easy part.

Predicting the serial number is harder but not impossible. In the first place, it was found that the targeted commercial CA uses sequential serial numbers. Being able to predict the next serial number, however, is not enough: the construction of the collision can be expected to take at least a day, before which the serial number and validity period have to be fixed, and only after which the to-be-signed part of the certificate will be entirely known. As a consequence, there will have been a substantial and uncertain increment in the serial number by the time the collision construction is finished. So, another essential ingredient of our construction was the fact that the CA's weekend workload is quite stable: it was observed during several weekends that the increment in serial number over a weekend does not vary a lot. This allowed us to pretty reliably predict Monday morning's serial numbers on the Friday afternoon before. Thus, on Friday afternoon we selected

a number at the high end of the predicted range for the next Monday morning, and inserted it in the legitimate to-be-signed part along with a validity period starting that same Monday morning at the time corresponding to our serial number prediction. See Step 6 how we then managed, after the weekend, to target precisely the selected serial number and validity period.

Step 3. Computation of the collision. At this point both chosen prefixes have been fully determined so the chosen-prefix collision can be computed: first the 72 birthday bits per side, calculated in parallel on the 1290 SPUs of a cluster of 215 PS3s, followed by the calculation of 3 pairs of 512-bit near-collision blocks on the 215 PS3 CPUs. The entire calculation takes on average about a day.

Given that we had a weekend available, and that the calculation can be expected to take just a day, we sequentially processed a number of chosen-prefixes, each corresponding to different serial numbers and validity periods (targeting both Monday and Tuesday mornings). So, a near-collision block calculation on the CPUs would always run simultaneously with a birthday search on the SPUs for the 'next' attempt.

Step 4. Finishing the to-be-signed parts. At this point the legitimate and rogue sides collide under MD5, so that from here on only identical bits may be appended to both sides.

With $208 + 24 + 72 + 3 * 512 = 1840$ bits set, the remaining $2048 - 1840 = 208$ bits need to be set for the 2048-bit RSA modulus on the legitimate side. Since in the next step the RSA private exponent corresponding to the RSA public exponent is needed, the full factorization of the RSA modulus needs to be known, and the factors must be compatible with the choice of the RSA public exponent. Common CAs (including our targeted commercial CA) do not check for compositeness of RSA moduli in certification requests, implying that we could simply have added 208 bits to make the RSA modulus a prime. We found that approach unsatisfactory, and opted for the rather crude but trivial to program method sketched below that results in a 224-bit prime factor with a prime 1824-bit cofactor. Given that at the time this work was done the largest factor found using the elliptic curve integer factorization method was 222 bits long, a 224-bit smallest prime factor keeps the resulting modulus out of reach of common factoring efforts. We could have used a relatively advanced lattice-based method to try and squeeze in a 312-bit prime factor along with a prime 1736-bit cofactor. Given only 208 bits of freedom to select a 2048-bit RSA modulus, it is unlikely that a more balanced solution can efficiently be found. Thus the reason why as few bits as possible should be fixed in Step 1, is that it allows us to construct a slightly less unbalanced RSA modulus.

Let N be the 2048-bit integer consisting of the 1840 already determined bits of the RSA modulus-to-be, followed by 208 one bits. We select a 224-bit integer p at random until $N \bmod p$ is less than 2^{208}, and keep doing this until both p and $q = \lfloor N/p \rfloor$ are prime and the RSA public exponent is coprime to $(p-1)(q-1)$. Once such primes p and q have been found, the number pq will be the legitimate side's RSA modulus, the leading 1840 bits of which are already present in the

legitimate side's to-be-signed part, and the 208 least significant bits of which are inserted in both to-be-signed parts.

To analyse the required effort somewhat more in general, 2^{k-208} integers of k bits (with $k > 208$) need to be selected on average for pq to have the desired 1840 leading bits. Since an ℓ-bit integer is prime with probability approximately $1/\log(2^\ell)$, a total of $k(2048-k)2^{k-208}(\log 2)^2$ attempts may be expected before a suitable RSA modulus is found. The coprimality requirement is a lower order effect that we disregard. Note that for $k(k-2048)(\log 2)^2$ of the attempts the k-bit number p has to be tested for primality, and that for $(2048-k)\log 2$ of those q needs to be tested as well (on average, obviously). For $k = 224$ this turned out to be doable in a few minutes on a standard PC.

This completes the to-be-signed parts on both sides. Now it remains to be hoped that the legitimate part that actually will be signed corresponds, bit for bit, with the legitimate to-be-signed part that we concocted.

Step 5. Preparing the certification request. Using the relevant information from the legitimate side's template, i.e., the subject Distinguished Name and the public key, a PKCS#10 Certificate Signing Request is prepared. The CA requires proof of possession of the private key corresponding to the public key in the request. This is done by signing the request using the private key – this is the sole reason that we need the RSA private exponent.

Step 6. Submission of the certification request. The targeted legitimate to-be-signed part contains a very specific validity period that leaves no choice for the moment at which the certification request needs to be submitted to the CA. Just hoping that at that time the serial number would have precisely the predicted value is unlikely to work, so a somewhat more elaborate approach is used. About half an hour before the targeted submission moment, the same request is submitted, and the serial number in the resulting certificate is inspected. If it is already too high, the entire attempt is abandoned. Otherwise, the request is repeatedly submitted, with a frequency depending on the gap that may still exist between the serial number received and the targeted one, and taking into account possible certification requests by others. In this way the serial number is slowly nudged toward the right value at the right time. Although there is nothing illegal about repeated certification requests, it should be possible for a CA to recognize the somewhat anomalous behavior sketched above and to take appropriate countermeasures (such as random delays or jumps in serial numbers) if it occurs.

Various types of accidents may happen, of course, and we experienced some of them, such as another CA customer 'stealing' our targeted serial number just a few moments before our attempt to get it, thereby wasting that weekend's calculations. But, after the fourth weekend it worked as planned, and we managed to get an actually signed part that exactly matched our predicted legitimate to-be-signed part.

Step 7. Creation of the rogue certificate. Given the perfect match between the actually signed part and the hoped for one, and the MD5 collision between the latter and the rogue side's to-be-signed part, the MD5-based digital signature

present in the legitimate certificate as provided by the commercial CA is equally valid for the rogue side. To finish the rogue CA certificate it suffices to copy the digital signature to the right spot in the rogue CA certificate.

The full details of the above construction, including both certificates, can be found on `http://www.win.tue.nl/hashclash/rogue-ca/`.

5.3 Nostradamus attack

In the original Nostradamus attack from [15] one first commits to a certain hash value, and afterwards for any message constructs a document that not only contains that message but that also has under MD5 the committed hash value. So far, this attack is, in its full generality, infeasible for MD5 because space and time requirements are beyond what can be handled at his point. It is easily doable, though, if a limited size message space has been defined upfront.

Suppose there are messages $m_1, m_2, \ldots, m_r$, then using $r-1$ chosen-prefix collisions we can construct r suffixes $s_1, s_2, \ldots, s_r$ such that the r documents $d_i = m_i \| s_i$ all have the same hash. After committing to the common hash, afterwards any of the r documents $d_1, d_2, \ldots, d_r$ can be shown, possibly to achieve some malicious goal. The other documents will remain hidden and their contents, i.e., the m_i-parts, cannot be derived – with overwhelming probability – from the single published document or from the common hash value.

To show the practicality of this variant, we have made an example consisting of 12 different PDF documents with a common MD5-hash, where each document predicts a different outcome of the 2008 US presidential elections. The PDF format is convenient for this purpose because it allows insertion of extra image objects that are unreferenced in the resulting document and thus invisible in any common PDF reader. See the next section for more on the PDF related details of the construction and `http://www.win.tue.nl/hashclash/Nostradamus/` for the actual documents, one of which correctly predicted the outcome one year before the elections took place. For each of the 11 collisions required for this example we used a 64-bit birthday search (on a single PS3) aiming for about 11 near-collision blocks (constructed on a quad-core PC). It took less than 2 days per chosen-prefix collision. Since we performed those computations our methods have improved as described in this paper, so this attack would now run much faster.

5.3.1 PDF construction

Given the structure of PDF documents it is not entirely straightforward how to insert different chosen-prefix collision blocks, while keeping the parts following those blocks identical in order to maintain the collision. The relevant details of both the PDF structure and our construction are covered here.

A PDF document is built up from the following 4 consecutive parts: a fixed header, a part consisting of an arbitrary number of numbered objects, an object lookup table and, finally, a trailer. The trailer specifies the number of objects, which of the objects is the unique root object (containing the document content)

and which is the info object (containing the document's meta information such as authors and title etc.), and contains a filepointer to the start of the object lookup table.

Given a file containing a PDF document, additional numbered objects can be inserted, as long as they are added to the object lookup table and the corresponding changes are made to the number of objects and the filepointer in the trailer. A template for an image object is given in Table 5-1. With the exception of the binary image, the format is entirely text based. The binary image is put between single line-feed characters (ASCII code 10) and the result is encapsulated by the keywords `stream` and `endstream`. The keyword `/Length` must specify the byte length of the image. As the image is uncompressed and each pixel requires three bytes ('`RGB`'), the image byte length must be three times the product of the specified width and height. The object number (`42` in the example object header) must be set to the next available object number.

Table 5-1. A numbered image object in the PDF format.

Part	Contents
object header	`42 0 obj`
image header	`<< /ColorSpace /DeviceRGB /Subtype /Image`
image size	`/Length 9216 /Width 64 /Height 48 /BitsPerComponent 8`
image contents	`>> stream...endstream`
object footer	`endobj`

When constructing colliding PDF files they must be equal after the collision-causing data (cf. the "suffix" in Figure *yy*). The object lookup tables and trailers for all files must therefore be the same. This was achieved as follows:

- As all documents must have the same number of objects, dummy objects are inserted where necessary.
- Since all root objects must have the same object number, they can be copied if necessary to objects with the next available object number.
- The info objects are treated in the same way as the root objects.
- To make sure that all object lookup tables and filepointers are identical, the objects can be sorted by object number and if necessary padded with spaces after their `obj` keyword to make sure that all objects with the same object number have the same file position and byte length in all files.
- Finally, the object lookup tables and trailers need to be adapted to reflect the new situation – as a result they should be identical for all files.

Although this procedure works for basic PDF files (such as PDF version 1.4 as we produced using pdflatex), it should be noted that the PDF document format allows additional features that may cause obstructions, the details of which are irrelevant for this article.

Given r LaTeX files with the desired subtle differences (such as names of r different candidates), r different PDF files are produced using a version of LaTeX

that is suitable for our purposes (cf. above). In all these files a hidden image object with a fixed object number is then inserted, and the approach sketched above is followed to make the lookup tables and trailers for all files identical. To ensure that the files are identical after the hidden image contents, their corresponding objects were made the last objects in the files. This then leads to r chosen prefixes consisting of the leading parts of the PDF files up to and including the keyword `stream` and the first line-feed character. After determining $r - 1$ chosen-prefix collisions resulting in r collision-causing appendages, the appendages are put in the proper binary image parts, after which all files are completed with a line-feed character, the keywords `endstream` and `endobj`, and the identical lookup tables and trailers.

Note that the `Length` etc. fields have to be set before collision finding, and that the value of `Length` will grow logarithmically with r and linearly in the number of near-collision blocks one is aiming for.

5.4 Colliding executables

Using the same set-up as used for the Nostradamus attack reported in Section 5.3, i.e., 64-bit birthday searching on a PS3 followed by the construction of about 12 near-collision blocks on a quad-core PC, it took us less than 2 days to create two different Windows executables with the same MD5 hash.

Initially both 40960 bytes large, 13×64 bytes had to be appended to each executable, for a resulting size of just 41792 bytes each, to let the files collide under MD5 without changing their functionality. See `http://www.win.tue.nl/hashclash/SoftIntCodeSign/` for details. As noted above, it has been shown on `http://blog.didierstevens.com/2009/01/17/` that this attack can be elevated to one on a code signing scheme.

As usual, the following remarks apply:

- An existing executable with a known and published hash value not resulting from this construction cannot be targeted by this attack (cf. [9]): our attack is not a preimage or second preimage attack. In order to attack a software integrity protection or code signing scheme using this approach, the attacker must be able to manipulate the files before they are hashed (and, possibly, signed). Given the level of access required to realize the attack an attacker can probably do more harm in other simpler and more traditional ways.
- On the other hand, there is no guarantee that a downloaded file with the proper hash or correct signature is not the evil sibling of the intended file. Especially when software integrity verification takes place under the hood, users may be lured into installing – and trusting – malware. Until a tool is available that would also be able to distinguish potentially malicious MD5-based certificates, all a relying party can do is resorting to bit-level inspection of each executable; the latter requires more expertise than most users can be expected to have, in particular if the collision blocks are hidden at a less conspicious place than at the very end of the executable.
- Any number r of executables can be made to collide, at the cost of $r - 1$ chosen-prefix collisions and an $O(\log r)$-byte appendage to each of the r original executables.

A countermeasure thwarting our attack would be the inclusion of a self-checking component in software, i.e., where the software would check the integrity of its own executable as the first step of the execution. It is better, however, not to rely on cryptographic primitives such as MD5 that fail to meet their design criteria.

Acknowledgements

This work benefited greatly from suggestions by Xiaoyun Wang. We are grateful for comments and assistance received from the anonymous Eurocrypt 2007 reviewers and other anonymous reviewers, and from Jacob Appelbaum, Joppe Bos, Stuart Haber, Paul Hoffman, Pascal Junod, Vlastimil Klima, David Molnar, Dag Arne Osvik, Bart Preneel, NBV, Werner Schindler, Gido Schmitz, Alexander Sotirov, Eric Verheul, and Yiqun Lisa Yin. The second author gratefully acknowledges Alcatel-Lucent Bell Laboratories.

This work has been supported in part by the Swiss National Science Foundation under grant number 206021-117409, by EPFL DIT, and by the European Commission through the EU ICT program ECRYPT II.

References

1. Steven M. Bellovin and Eric K. Rescorla, *Deploying a New Hash Algorithm*, in: Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA, The Internet Society, 2006. Available from `http://www.isoc.org/isoc/conferences/ndss/06/proceedings/papers/deploying_new_hash_algorithm.pdf`.
2. B. den Boer and A. Bosselaers, *Collisions for the compression function of MD5*, Eurocrypt '93, Springer LNCS 765, pp. 293–304, 1994.
3. Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen, *Bekanntmachung zur elektronischen Signatur nach dem Signaturgesetz und der Signaturverordnung (Übersicht über geeignete Algorithmen)*, December 2007, Bundesanzeiger Nr. 19, p. 376, 2008. Available from `http://www.bundesnetzagentur.de/media/archive/12198.pdf`.
4. Christophe de Cannière and Christian Rechberger, *Finding SHA-1 Characteristics: General results and applications*, AsiaCrypt 2006, Springer LNCS 4284, pp. 1–20, 2006.
5. W. Clark and J. Liang, *On arithmetic weight for a general radix representation of integers*, IEEE Transactions on Information Theory, **19**(6), 823 – 826, 1973.
6. D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, W. Polk, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF RFC 5280, May 2008, `http://www.ietf.org/rfc/rfc5280.txt`.
7. M. Daum and S. Lucks, *Attacking Hash Functions by Poisoned Messages, "The Story of Alice and her Boss"*, June 2005, `http://th.informatik.uni-mannheim.de/People/lucks/HashCollisions/`.
8. H. Dobbertin, *Cryptanalysis of MD5 Compress*, EuroCrypt'96 Rump session, May 1996, `http://www-cse.ucsd.edu/~bsy/dobbertin.ps`.
9. P. Gauravaram, A. McCullagh and E. Dawson, *Collision Attacks on MD5 and SHA-1: Is this the "Sword of Damocles" for Electronic Commerce?*, AusCERT 2006 R&D Stream, May 2006, `http://www2.mat.dtu.dk/people/P.Gauravaram/AusCert-6.pdf`.
10. M. Gebhardt, G. Illies and W. Schindler, *A Note on Practical Value of Single Hash Collisions for Special File Formats*, NIST First Cryptographic Hash Workshop, October/November 2005, `http://csrc.nist.gov/groups/ST/hash/documents/Illies_NIST_05.pdf`.
11. S. Halevi and H. Krawczyk, *Strengthening Digital Signatures via Randomized Hashing*, Crypto 2006, Springer LNCS 4117, pp. 41–59, 2006.
12. Philip Hawkes, Michael Paddon and Gregory G. Rose, *Musings on the Wang et al. MD5 Collision*, Cryptology ePrint Archive, Report 2004/264, `http://eprint.iacr.org/2004/264`.
13. P. Hoffman and B. Schneier, *Attacks on Cryptographic Hashes in Internet Protocols*, IETF RFC 4270, November 2005, `http://www.ietf.org/rfc/rfc4270.txt`.
14. D. Kaminsky, *MD5 to be considered harmful someday*, December 2004, `http://www.doxpara.com/research/md5/md5_someday.pdf`.
15. J. Kelsey and T. Kohno, *Herding Hash Functions and the Nostradamus Attack*, Eurocrypt 2006, Springer LNCS 4004, pp. 183–200, 2006.
16. Vlastimil Klima, *Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications*, Cryptology ePrint Archive, Report 2005/102, `http://eprint.iacr.org/2005/102`.

17. Vlastimil Klima, *Tunnels in Hash Functions: MD5 Collisions Within a Minute*, Cryptology ePrint Archive, Report 2006/105, http://eprint.iacr.org/2006/105.
18. A.K. Lenstra and B.M.M. de Weger, *On the possibility of constructing meaningful hash collisions for public keys*, ACISP 2005, Springer LNCS 3574, pp. 267–279, 2005. Full version available from http://www.win.tue.nl/~bdeweger/CollidingCertificates/ddl-full.pdf.
19. Cameron McDonald, Philip Hawkes and Josef Pieprzyk, *SHA-1 collisions now* 2^{52}, Eurocrypt 2009 Rump session, http://eurocrypt2009rump.cr.yp.to/837a0a8086fa6ca714249409ddfae43d.pdf.
20. Florian Mendel, Christian Rechberger and Vincent Rijmen, *Update on SHA-1*, Crypto 2007 Rump session, http://rump2007.cr.yp.to/09-rechberger.pdf.
21. Alfred Menezes, Paul C. van Oorschot and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.
22. O. Mikle, *Practical Attacks on Digital Signatures Using MD5 Message Digest*, Cryptology ePrint Archive, Report 2004/356, http://eprint.iacr.org/2004/356.
23. Paul C. van Oorschot and Michael J. Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12**(1), 1–28, 1999.
24. Robert Primmer and Carl D'Halluin, *Collision and Preimage Resistance of the Centera Content Address*, Technical Report, EMC Corporation, 2005, http://www.robertprimmer.com/rjp/tech/mine/CenteraCollisionProbs.pdf.
25. Christian Rechberger, unpublished result, 2006.
26. R. Rivest, *The MD5 Message-Digest Algorithm*, IETF RFC 1321, April 1992, http://www.ietf.org/rfc/rfc1321.txt.
27. Marc Stevens, *Fast Collision Attack on MD5*, Cryptology ePrint Archive, Report 2006/104, http://eprint.iacr.org/2006/104.
28. Marc Stevens, *On Collisions for MD5*, TU Eindhoven MSc thesis, June 2007. Available from http://www.win.tue.nl/hashclash/On%20Collisions%20for%20MD5%20-%20M.M.J.%20Stevens.pdf.
29. Marc Stevens, Arjen Lenstra and Benne de Weger, *Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities*, in M. Naor (Ed.), Eurocrypt 2007, Springer LNCS 4515, pp. 1–22, 2007.
30. Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik and Benne de Weger, *Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate*, Crypto 2009, Springer LNCS 5677, 2009.
31. X. Wang, D. Feng, X. Lai and H. Yu, *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Cryptology ePrint Archive, Report 2004/199, http://eprint.iacr.org/2004/199.
32. Xiaoyun Wang, Andrew Yao and Frances Yao, *New Collision Search for SHA-1*, Crypto 2005 Rump session, http://www.iacr.org/conferences/crypto2005/r/2.pdf.
33. Xiaoyun Wang, Yiqun Lisa Yin and Hongbo Yu, *Finding Collisions in the Full SHA-1*, Crypto 2005, Springer LNCS 3621, pp. 17–36, 2005.
34. X. Wang and H. Yu, *How to Break MD5 and Other Hash Functions*, Eurocrypt 2005, Springer LNCS 3494, pp. 19–35, 2005.
35. Tao Xie, FanBao Liu and DengGuo Feng, *Could The 1-MSB Input Difference Be The Fastest Collision Attack For MD5?*, Cryptology ePrint Archive, Report 2008/391, http://eprint.iacr.org/2008/391.

A MD5 compression function constants

Table A-1. MD5 Addition and Rotation Constants and message block expansion.

t	AC_t	RC_t	W_t
0	$\texttt{d76aa478}_{16}$	7	m_0
1	$\texttt{e8c7b756}_{16}$	12	m_1
2	$\texttt{242070db}_{16}$	17	m_2
3	$\texttt{c1bdceee}_{16}$	22	m_3
4	$\texttt{f57c0faf}_{16}$	7	m_4
5	$\texttt{4787c62a}_{16}$	12	m_5
6	$\texttt{a8304613}_{16}$	17	m_6
7	$\texttt{fd469501}_{16}$	22	m_7
8	$\texttt{698098d8}_{16}$	7	m_8
9	$\texttt{8b44f7af}_{16}$	12	m_9
10	$\texttt{ffff5bb1}_{16}$	17	m_{10}
11	$\texttt{895cd7be}_{16}$	22	m_{11}
12	$\texttt{6b901122}_{16}$	7	m_{12}
13	$\texttt{fd987193}_{16}$	12	m_{13}
14	$\texttt{a679438e}_{16}$	17	m_{14}
15	$\texttt{49b40821}_{16}$	22	m_{15}

t	AC_t	RC_t	W_t
16	$\texttt{f61e2562}_{16}$	5	m_1
17	$\texttt{c040b340}_{16}$	9	m_6
18	$\texttt{265e5a51}_{16}$	14	m_{11}
19	$\texttt{e9b6c7aa}_{16}$	20	m_0
20	$\texttt{d62f105d}_{16}$	5	m_5
21	$\texttt{02441453}_{16}$	9	m_{10}
22	$\texttt{d8a1e681}_{16}$	14	m_{15}
23	$\texttt{e7d3fbc8}_{16}$	20	m_4
24	$\texttt{21e1cde6}_{16}$	5	m_9
25	$\texttt{c33707d6}_{16}$	9	m_{14}
26	$\texttt{f4d50d87}_{16}$	14	m_3
27	$\texttt{455a14ed}_{16}$	20	m_8
28	$\texttt{a9e3e905}_{16}$	5	m_{13}
29	$\texttt{fcefa3f8}_{16}$	9	m_2
30	$\texttt{676f02d9}_{16}$	14	m_7
31	$\texttt{8d2a4c8a}_{16}$	20	m_{12}

t	AC_t	RC_t	W_t
32	$\texttt{fffa3942}_{16}$	4	m_5
33	$\texttt{8771f681}_{16}$	11	m_8
34	$\texttt{6d9d6122}_{16}$	16	m_{11}
35	$\texttt{fde5380c}_{16}$	23	m_{14}
36	$\texttt{a4beea44}_{16}$	4	m_1
37	$\texttt{4bdecfa9}_{16}$	11	m_4
38	$\texttt{f6bb4b60}_{16}$	16	m_7
39	$\texttt{bebfbc70}_{16}$	23	m_{10}
40	$\texttt{289b7ec6}_{16}$	4	m_{13}
41	$\texttt{eaa127fa}_{16}$	11	m_0
42	$\texttt{d4ef3085}_{16}$	16	m_3
43	$\texttt{04881d05}_{16}$	23	m_6
44	$\texttt{d9d4d039}_{16}$	4	m_9
45	$\texttt{e6db99e5}_{16}$	11	m_{12}
46	$\texttt{1fa27cf8}_{16}$	16	m_{15}
47	$\texttt{c4ac5665}_{16}$	23	m_2

t	AC_t	RC_t	W_t
48	$\texttt{f4292244}_{16}$	6	m_0
49	$\texttt{432aff97}_{16}$	10	m_7
50	$\texttt{ab9423a7}_{16}$	15	m_{14}
51	$\texttt{fc93a039}_{16}$	21	m_5
52	$\texttt{655b59c3}_{16}$	6	m_{12}
53	$\texttt{8f0ccc92}_{16}$	10	m_3
54	$\texttt{ffeff47d}_{16}$	15	m_{10}
55	$\texttt{85845dd1}_{16}$	21	m_1
56	$\texttt{6fa87e4f}_{16}$	6	m_8
57	$\texttt{fe2ce6e0}_{16}$	10	m_{15}
58	$\texttt{a3014314}_{16}$	15	m_6
59	$\texttt{4e0811a1}_{16}$	21	m_{13}
60	$\texttt{f7537e82}_{16}$	6	m_4
61	$\texttt{bd3af235}_{16}$	10	m_{11}
62	$\texttt{2ad7d2bb}_{16}$	15	m_2
63	$\texttt{eb86d391}_{16}$	21	m_9

B Boolean Function Bitconditions

The 4 tables in this appendix correspond to rounds 1 through 4, respectively, i.e., $0 \le t < 16$, $16 \le t < 32$, $32 \le t < 48$ and $48 \le t < 64$. The 'abc' in each of the first columns denotes the three differential bitconditions $(\mathtt{q}_t[i], \mathtt{q}_{t-1}[i], \mathtt{q}_{t-2}[i])$ for the relevant t and $0 \le i \le 31$, with each table containing all 27 possible triples. Columns $2, 3, 4$ contain forward bitconditions $FC(t, abc, g)$ for $g = 0, +1, -1$, respectively, and columns $5, 6, 7$ contain backward bitconditions $BC(t, abc, g)$ for those same g's, respectively. The parenthesized number next to a triple def is $|U_{def}|$, the amount of freedom left. An entry is left empty if $g \notin V_{abc}$. See section 4.4.3 for more details.

B.1 Bitconditions applied to boolean function *F*

Table B-1. Round 1 ($0 \le t < 16$) bitconditions applied to boolean function F:

$$F(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z)$$

DB	Forward bitconditions			Backward bitconditions		
abc	$g = 0$	$g = +1$	$g = -1$	$g = 0$	$g = +1$	$g = -1$
... (8)	... (8)			... (8)		
..+ (4)	1.+ (2)	0.+ (2)		1.+ (2)	0.+ (2)	
..- (4)	1.- (2)		0.- (2)	1.- (2)		0.- (2)
.+. (4)	0+. (2)	1+. (2)		0+. (2)	1+. (2)	
.++ (2)		.++ (2)			.++ (2)	
.+- (2)		1+- (1)	0+- (1)		1+- (1)	0+- (1)
.-. (4)	0-. (2)		1-. (2)	0-. (2)		1-. (2)
.-+ (2)		0-+ (1)	1-+ (1)		0-+ (1)	1-+ (1)
.-- (2)			.-- (2)			.-- (2)
+.. (4)	+.V (2)	+10 (1)	+01 (1)	+^. (2)	+10 (1)	+01 (1)
+.+ (2)	+0+ (1)	+1+ (1)		+0+ (1)	+1+ (1)	
+.- (2)	+1- (1)		+0- (1)	+1- (1)		+0- (1)
++. (2)	++1 (1)	++0 (1)		++1 (1)	++0 (1)	
+++ (1)		+++ (1)			+++ (1)	
++- (1)	++- (1)			++- (1)		
+-. (2)	+-0 (1)		+-1 (1)	+-0 (1)		+-1 (1)
+-+ (1)	+-+ (1)			+-+ (1)		
+-- (1)			+-- (1)			+-- (1)
-.. (4)	-.V (2)	-01 (1)	-10 (1)	-^. (2)	-01 (1)	-10 (1)
-.+ (2)	-1+ (1)	-0+ (1)		-1+ (1)	-0+ (1)	
-.- (2)	-0- (1)		-1- (1)	-0- (1)		-1- (1)
-+. (2)	-+0 (1)	-+1 (1)		-+0 (1)	-+1 (1)	
-++ (1)		-++ (1)			-++ (1)	
-+- (1)	-+- (1)			-+- (1)		
--. (2)	--1 (1)		--0 (1)	--1 (1)		--0 (1)
--+ (1)	--+ (1)			--+ (1)		
--- (1)			--- (1)			--- (1)

B.2 Bitconditions applied to boolean function *G*

Table B-2. Round 2 ($16 \leq t < 32$) bitconditions applied to boolean function G:

$$G(X, Y, Z) = (Z \wedge X) \oplus (\overline{Z} \wedge Y)$$

DB	Forward bitconditions			Backward bitconditions		
abc	$g = 0$	$g = +1$	$g = -1$	$g = 0$	$g = +1$	$g = -1$
... (8)	... (8)			... (8)		
..+ (4)	.V+ (2)	10+ (1)	01+ (1)	^.+ (2)	10+ (1)	01+ (1)
..- (4)	.V- (2)	01- (1)	10- (1)	^.- (2)	01- (1)	10- (1)
.+. (4)	.+1 (2)	.+0 (2)		.+1 (2)	.+0 (2)	
.++ (2)	0++ (1)	1++ (1)		0++ (1)	1++ (1)	
.+- (2)	1+- (1)	0+- (1)		1+- (1)	0+- (1)	
.-. (4)	.-1 (2)		.-0 (2)	.-1 (2)		.-0 (2)
.-+ (2)	1-+ (1)		0-+ (1)	1-+ (1)		0-+ (1)
.-- (2)	0-- (1)		1-- (1)	0-- (1)		1-- (1)
+.. (4)	+.0 (2)	+.1 (2)		+.0 (2)	+.1 (2)	
+.+ (2)	+1+ (1)	+0+ (1)		+1+ (1)	+0+ (1)	
+.- (2)	+0- (1)	+1- (1)		+0- (1)	+1- (1)	
++. (2)		++. (2)			++. (2)	
+++ (1)		+++ (1)			+++ (1)	
++- (1)		++- (1)			++- (1)	
+-. (2)		+-1 (1)	+-0 (1)		+-1 (1)	+-0 (1)
+-+ (1)	+-+ (1)			+-+ (1)		
+-- (1)	+-- (1)			+-- (1)		
-.. (4)	-.0 (2)		-.1 (2)	-.0 (2)		-.1 (2)
-.+ (2)	-0+ (1)		-1+ (1)	-0+ (1)		-1+ (1)
-.- (2)	-1- (1)		-0- (1)	-1- (1)		-0- (1)
-+. (2)		-+0 (1)	-+1 (1)		-+0 (1)	-+1 (1)
-++ (1)	-++ (1)			-++ (1)		
-+- (1)	-+- (1)			-+- (1)		
--. (2)			--. (2)			--. (2)
--+ (1)			--+ (1)			--+ (1)
--- (1)			--- (1)			--- (1)

B.3 Bitconditions applied to boolean function *H*

Table B-3. Round 3 ($32 \leq t < 48$) bitconditions applied to boolean function H:

$$H(X, Y, Z) = X \oplus Y \oplus Z$$

DB	Forward bitconditions			Backward bitconditions		
abc	$g = 0$	$g = +1$	$g = -1$	$g = 0$	$g = +1$	$g = -1$
... (8)	... (8)			... (8)		
..+ (4)		.V+ (2)	.Y+ (2)		^.+ (2)	!.+ (2)
..- (4)		.Y- (2)	.V- (2)		!.- (2)	^.- (2)
.+. (4)		.+W (2)	.+H (2)		m+. (2)	#+. (2)
.++ (2)	.++ (2)			.++ (2)		
.+- (2)	.+- (2)			.+- (2)		
.-. (4)		.-H (2)	.-W (2)		#-. (2)	m-. (2)
.-+ (2)	.-+ (2)			.-+ (2)		
.-- (2)	.-- (2)			.-- (2)		
+.. (4)		+.V (2)	+.Y (2)		+^. (2)	+!. (2)
+.+ (2)	+.+ (2)			+.+ (2)		
+.- (2)	+.- (2)			+.- (2)		
++. (2)	++. (2)			++. (2)		
+++ (1)		+++ (1)			+++ (1)	
++- (1)			++- (1)			++- (1)
+-. (2)	+-. (2)			+-. (2)		
+-+ (1)			+-+ (1)			+-+ (1)
+-- (1)		+-- (1)			+-- (1)	
-.. (4)		-.Y (2)	-.V (2)		-!. (2)	-^. (2)
-.+ (2)	-.+ (2)			-.+ (2)		
-.- (2)	-.- (2)			-.- (2)		
-+. (2)	-+. (2)			-+. (2)		
-++ (1)			-++ (1)			-++ (1)
-+- (1)		-+- (1)			-+- (1)	
--. (2)	--. (2)			--. (2)		
--+ (1)		--+ (1)			--+ (1)	
--- (1)			--- (1)			--- (1)

B.4 Bitconditions applied to boolean function *I*

Table B-4. Round 4 ($48 \leq t < 64$) bitconditions applied to boolean function I:

$$I(X, Y, Z) = Y \oplus (X \vee \overline{Z})$$

DB	Forward bitconditions			Backward bitconditions		
abc	$g = 0$	$g = +1$	$g = -1$	$g = 0$	$g = +1$	$g = -1$
`...` (8)	`...` (8)			`...` (8)		
`..+` (4)	`1.+` (2)	`01+` (1)	`00+` (1)	`1.+` (2)	`01+` (1)	`00+` (1)
`..-` (4)	`1.-` (2)	`00-` (1)	`01-` (1)	`1.-` (2)	`00-` (1)	`01-` (1)
`.+.` (4)		`0+1` (1)	`.+Q` (3)		`0+1` (1)	`?+.` (3)
`.++` (2)	`0++` (1)		`1++` (1)	`0++` (1)		`1++` (1)
`.+-` (2)	`0+-` (1)		`1+-` (1)	`0+-` (1)		`1+-` (1)
`.-.` (4)		`.-Q` (3)	`0-1` (1)		`?-.` (3)	`0-1` (1)
`.-+` (2)	`0-+` (1)	`1-+` (1)		`0-+` (1)	`1-+` (1)	
`.--` (2)	`0--` (1)	`1--` (1)		`0--` (1)	`1--` (1)	
`+..` (4)	`+.0` (2)	`+01` (1)	`+11` (1)	`+.0` (2)	`+01` (1)	`+11` (1)
`+.+` (2)	`+.+` (2)			`+.+` (2)		
`+.-` (2)		`+0-` (1)	`+1-` (1)		`+0-` (1)	`+1-` (1)
`++.` (2)	`++1` (1)		`++0` (1)	`++1` (1)		`++0` (1)
`+++` (1)			`+++` (1)			`+++` (1)
`++-` (1)	`++-` (1)			`++-` (1)		
`+-.` (2)	`+-1` (1)	`+-0` (1)		`+-1` (1)	`+-0` (1)	
`+-+` (1)		`+-+` (1)			`+-+` (1)	
`+--` (1)	`+--` (1)			`+--` (1)		
`-..` (4)	`-.0` (2)	`-11` (1)	`-01` (1)	`-.0` (2)	`-11` (1)	`-01` (1)
`-.+` (2)		`-1+` (1)	`-0+` (1)		`-1+` (1)	`-0+` (1)
`-.-` (2)	`-.-` (2)			`-.-` (2)		
`-+.` (2)	`-+1` (1)		`-+0` (1)	`-+1` (1)		`-+0` (1)
`-++` (1)	`-++` (1)			`-++` (1)		
`-+-` (1)			`-+-` (1)			`-+-` (1)
`--.` (2)	`--1` (1)	`--0` (1)		`--1` (1)	`--0` (1)	
`--+` (1)	`--+` (1)			`--+` (1)		
`---` (1)		`---` (1)			`---` (1)	

C Birthday Cost

In this appendix notation and variables are as in Section 4.2. The columns p, C_{tr} and M denote the values $-\log_2(p_{r,k,w})$, $\log_2(C_{\mathrm{tr}}(r,k,w))$ and the minimum required memory such that $C_{\mathrm{coll}}(r,k,w,M) \leq C_{\mathrm{tr}}(r,k,w)$, respectively.

$r=3$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0												
4										34.01	51.33	2TB
8							33.42	53.03	748GB	31.31	51.98	174GB
12	34.01	55.33	2TB	32.42	54.53	374GB	30.55	53.6	103GB	28.24	52.44	21GB
16	31.	55.83	141GB	29.65	55.15	55GB	27.36	54.01	12GB	25.6	53.13	4GB
20	27.51	56.08	13GB	26.18	55.42	5GB	24.53	54.59	2GB	23.26	53.96	673MB
24	24.33	56.49	2GB	23.35	56.	714MB	22.17	55.41	315MB	21.19	54.92	160MB
28	21.11	56.88	152MB	20.56	56.6	103MB	19.98	56.32	70MB	19.57	56.11	52MB
32	17.88	57.26	17MB	17.88	57.27	17MB	17.89	57.27	17MB	17.88	57.27	17MB

$r=3$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0				31.68	48.17	225GB	30.25	47.45	84GB	28.01	46.33	18GB
4	32.2	50.43	323GB	29.92	49.29	67GB	28.06	48.36	19GB	26.2	47.43	6GB
8	28.83	50.74	32GB	27.33	49.99	11GB	25.88	49.26	5GB	24.47	48.56	2GB
12	26.63	51.64	7GB	25.14	50.9	3GB	23.96	50.3	2GB	22.94	49.8	537MB
16	24.31	52.48	2GB	23.27	51.96	675MB	22.49	51.57	394MB	21.86	51.26	255MB
20	22.28	53.46	340MB	21.62	53.13	215MB	21.14	52.9	155MB	20.73	52.69	117MB
24	20.53	54.59	102MB	20.01	54.33	71MB	19.65	54.15	55MB	19.38	54.01	46MB
28	19.25	55.95	42MB	19.02	55.83	36MB	18.82	55.74	31MB	18.65	55.65	28MB
32	17.88	57.27	17MB	17.88	57.27	17MB	17.88	57.27	17MB	17.88	57.27	17MB

$r=4$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0							34.	49.33	2TB	30.19	47.42	81GB
4				33.42	51.04	749GB	30.36	49.51	90GB	27.59	48.12	14GB
8	35.	53.83	3TB	30.3	51.48	87GB	27.21	49.93	11GB	24.87	48.76	2GB
12	29.58	53.12	53GB	27.53	52.09	13GB	24.59	50.62	2GB	22.47	49.56	388MB
16	26.26	53.45	6GB	24.36	52.51	2GB	22.06	51.36	292MB	20.38	50.51	91MB
20	23.16	53.91	628MB	21.5	53.08	199MB	19.72	52.19	58MB	18.54	51.6	26MB
24	20.25	54.45	84MB	19.09	53.87	38MB	17.8	53.23	16MB	16.86	52.76	8MB
28	17.26	54.95	11MB	16.63	54.64	7MB	16.02	54.34	5MB	15.6	54.13	4MB
32	14.29	55.47	2MB	14.29	55.47	2MB	14.29	55.47	2MB	14.29	55.47	2MB

$r=4$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	26.98	45.81	9GB	24.45	44.55	2GB	22.14	43.4	310MB	20.33	42.49	88MB
4	24.95	46.8	3GB	22.82	45.73	493MB	21.04	44.84	144MB	19.55	44.1	52MB
8	22.63	47.64	432MB	20.92	46.79	133MB	19.58	46.12	53MB	18.56	45.61	26MB
12	20.67	48.66	112MB	19.41	48.03	47MB	18.45	47.55	24MB	17.71	47.18	15MB
16	19.08	49.86	37MB	18.19	49.42	21MB	17.56	49.1	13MB	17.08	48.86	10MB
20	17.66	51.16	14MB	17.09	50.87	10MB	16.7	50.67	8MB	16.39	50.52	6MB
24	16.25	52.45	6MB	15.82	52.24	4MB	15.54	52.09	4MB	15.33	51.99	3MB
28	15.31	53.98	3MB	15.09	53.87	3MB	14.93	53.79	3MB	14.78	53.72	2MB
32	14.29	55.47	2MB	14.29	55.47	2MB	14.29	55.47	2MB	14.29	55.47	2MB

$r=5$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	35.	49.83	3TB	31.2	47.92	161GB	27.13	45.89	10GB	23.74	44.2	938MB
4	33.42	51.04	749GB	28.47	48.56	25GB	24.63	46.64	2GB	21.58	45.12	210MB
8	28.61	50.63	27GB	25.61	49.13	4GB	22.	47.33	280MB	19.39	46.02	46MB
12	25.43	51.04	3GB	22.74	49.7	468MB	19.66	48.15	56MB	17.53	47.09	13MB
16	22.36	51.51	360MB	20.02	50.34	72MB	17.59	49.12	14MB	15.95	48.3	5MB
20	19.38	52.01	46MB	17.48	51.07	13MB	15.67	50.16	4MB	14.55	49.6	2MB
24	16.68	52.66	7MB	15.35	52.	3MB	14.06	51.36	2MB	13.17	50.91	1MB
28	13.92	53.29	2MB	13.22	52.93	1MB	12.61	52.63	1MB	12.21	52.43	1MB
32	11.2	53.92	1MB	11.2	53.93	1MB	11.2	53.92	1MB	11.2	53.93	1MB

$r=5$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	20.53	42.59	102MB	18.03	41.34	18MB	16.17	40.41	5MB	14.92	39.79	3MB
4	18.94	43.79	34MB	17.	42.82	9MB	15.57	42.11	4MB	14.53	41.59	2MB
8	17.27	44.96	11MB	15.79	44.22	4MB	14.75	43.7	2MB	14.01	43.33	2MB
12	15.92	46.28	5MB	14.84	45.75	2MB	14.09	45.37	2MB	13.56	45.11	1MB
16	14.8	47.73	2MB	14.06	47.35	2MB	13.55	47.1	1MB	13.18	46.92	1MB
20	13.79	49.22	1MB	13.31	48.98	1MB	12.99	48.82	1MB	12.76	48.7	1MB
24	12.64	50.64	1MB	12.29	50.47	1MB	12.07	50.36	1MB	11.91	50.28	1MB
28	11.95	52.3	1MB	11.76	52.2	1MB	11.62	52.14	1MB	11.5	52.07	1MB
32	11.2	53.92	1MB	11.2	53.93	1MB	11.2	53.92	1MB	11.2	53.93	1MB

$r=6$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	31.2	47.92	161GB	26.73	45.69	8GB	21.78	43.22	241MB	18.14	41.4	20MB
4	28.18	48.42	20GB	23.89	46.27	2GB	19.56	44.11	52MB	16.46	42.55	6MB
8	24.66	48.66	2GB	21.17	46.91	158MB	17.37	45.01	12MB	14.79	43.72	2MB
12	21.67	49.16	224MB	18.6	47.62	27MB	15.43	46.04	3MB	13.4	45.03	1MB
16	18.82	49.74	31MB	16.21	48.43	6MB	13.74	47.2	1MB	12.23	46.44	1MB
20	16.03	50.34	5MB	13.97	49.31	2MB	12.2	48.43	1MB	11.18	47.92	1MB
24	13.54	51.1	1MB	12.11	50.38	1MB	10.86	49.75	1MB	10.04	49.35	1MB
28	11.03	51.84	1MB	10.28	51.47	1MB	9.69	51.17	1MB	9.33	50.99	1MB
32	8.56	52.6	1MB	8.56	52.6	1MB	8.56	52.6	1MB	8.56	52.6	1MB

$r=6$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	15.12	39.88	3MB	13.05	38.85	1MB	11.73	38.19	1MB	10.91	37.78	1MB
4	14.05	41.35	2MB	12.44	40.55	1MB	11.39	40.02	1MB	10.7	39.68	1MB
8	12.92	42.79	1MB	11.73	42.19	1MB	10.95	41.8	1MB	10.44	41.54	1MB
12	12.01	44.33	1MB	11.14	43.9	1MB	10.57	43.61	1MB	10.2	43.42	1MB
16	11.25	45.95	1MB	10.64	45.64	1MB	10.24	45.45	1MB	9.98	45.32	1MB
20	10.53	47.59	1MB	10.14	47.39	1MB	9.89	47.27	1MB	9.72	47.19	1MB
24	9.59	49.12	1MB	9.31	48.98	1MB	9.14	48.9	1MB	9.04	48.85	1MB
28	9.09	50.87	1MB	8.93	50.79	1MB	8.82	50.74	1MB	8.73	50.69	1MB
32	8.56	52.6	1MB	8.56	52.6	1MB	8.56	52.6	1MB	8.56	52.6	1MB

$r=7$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	26.82	45.73	8GB	22.2	43.43	323MB	17.02	40.83	9MB	13.4	39.02	1MB
4	24.02	46.34	2GB	19.68	44.16	56MB	15.16	41.9	3MB	12.18	40.41	1MB
8	21.1	46.88	151MB	17.23	44.94	11MB	13.37	43.01	1MB	10.97	41.81	1MB
12	18.32	47.49	22MB	14.96	45.8	3MB	11.82	44.24	1MB	9.98	43.31	1MB
16	15.67	48.16	4MB	12.87	46.76	1MB	10.48	45.56	1MB	9.13	44.89	1MB
20	13.1	48.88	1MB	10.93	47.79	1MB	9.26	46.95	1MB	8.35	46.5	1MB
24	10.82	49.74	1MB	9.32	48.99	1MB	8.15	48.4	1MB	7.43	48.04	1MB
28	8.56	50.6	1MB	7.78	50.22	1MB	7.23	49.94	1MB	6.91	49.78	1MB
32	6.34	51.5	1MB	6.34	51.5	1MB	6.34	51.5	1MB	6.34	51.5	1MB

$r=7$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	10.8	37.73	1MB	9.25	36.95	1MB	8.35	36.5	1MB	7.84	36.25	1MB
4	10.13	39.39	1MB	8.9	38.78	1MB	8.17	38.41	1MB	7.74	38.19	1MB
8	9.42	41.03	1MB	8.5	40.57	1MB	7.94	40.3	1MB	7.61	40.13	1MB
12	8.82	42.74	1MB	8.15	42.4	1MB	7.74	42.19	1MB	7.48	42.07	1MB
16	8.31	44.48	1MB	7.84	44.24	1MB	7.55	44.1	1MB	7.37	44.01	1MB
20	7.82	46.23	1MB	7.51	46.08	1MB	7.32	45.99	1MB	7.21	45.93	1MB
24	7.06	47.86	1MB	6.84	47.75	1MB	6.72	47.69	1MB	6.66	47.65	1MB
28	6.71	49.68	1MB	6.58	49.62	1MB	6.5	49.58	1MB	6.43	49.54	1MB
32	6.34	51.5	1MB	6.34	51.5	1MB	6.34	51.5	1MB	6.34	51.5	1MB

$r=8$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	23.39	44.02	732MB	18.21	41.43	21MB	12.88	38.76	1MB	9.52	37.09	1MB
4	20.57	44.61	105MB	15.94	42.29	5MB	11.39	40.02	1MB	8.69	38.67	1MB
8	17.91	45.28	17MB	13.77	43.21	1MB	9.99	41.32	1MB	7.86	40.26	1MB
12	15.35	46.	3MB	11.78	44.22	1MB	8.79	42.72	1MB	7.17	41.91	1MB
16	12.91	46.78	1MB	10.	45.32	1MB	7.75	44.2	1MB	6.59	43.62	1MB
20	10.56	47.61	1MB	8.35	46.5	1MB	6.81	45.73	1MB	6.03	45.34	1MB
24	8.49	48.57	1MB	6.97	47.81	1MB	5.91	47.28	1MB	5.29	46.97	1MB
28	6.48	49.56	1MB	5.71	49.18	1MB	5.21	48.93	1MB	4.93	48.79	1MB
32	4.54	50.6	1MB	4.54	50.6	1MB	4.54	50.6	1MB	4.54	50.6	1MB

$r=8$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	7.45	36.05	1MB	6.37	35.51	1MB	5.8	35.23	1MB	5.5	35.08	1MB
4	7.06	37.85	1MB	6.18	37.42	1MB	5.71	37.18	1MB	5.45	37.05	1MB
8	6.63	39.64	1MB	5.96	39.31	1MB	5.6	39.12	1MB	5.39	39.02	1MB
12	6.26	41.46	1MB	5.77	41.21	1MB	5.49	41.07	1MB	5.34	40.99	1MB
16	5.94	43.29	1MB	5.59	43.12	1MB	5.39	43.02	1MB	5.28	42.97	1MB
20	5.61	45.13	1MB	5.38	45.01	1MB	5.25	44.95	1MB	5.18	44.92	1MB
24	5.01	46.83	1MB	4.85	46.75	1MB	4.77	46.71	1MB	4.73	46.69	1MB
28	4.78	48.71	1MB	4.68	48.67	1MB	4.62	48.64	1MB	4.58	48.62	1MB
32	4.54	50.6	1MB	4.54	50.6	1MB	4.54	50.6	1MB	4.54	50.6	1MB

$r=9$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	20.16	42.4	79MB	14.62	39.64	2MB	9.38	37.02	1MB	6.46	35.56	1MB
4	17.56	43.1	13MB	12.63	40.64	1MB	8.26	38.45	1MB	5.93	37.29	1MB
8	15.09	43.87	3MB	10.75	41.7	1MB	7.2	39.92	1MB	5.41	39.03	1MB
12	12.73	44.69	1MB	9.06	42.86	1MB	6.3	41.47	1MB	4.96	40.81	1MB
16	10.51	45.58	1MB	7.57	44.11	1MB	5.53	43.09	1MB	4.57	42.61	1MB
20	8.39	46.52	1MB	6.2	45.43	1MB	4.83	44.74	1MB	4.2	44.42	1MB
24	6.53	47.59	1MB	5.05	46.85	1MB	4.12	46.39	1MB	3.63	46.14	1MB
28	4.77	48.71	1MB	4.05	48.35	1MB	3.61	48.13	1MB	3.4	48.02	1MB
32	3.14	49.9	1MB	3.14	49.9	1MB	3.14	49.9	1MB	3.14	49.9	1MB

$r=9$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	4.95	34.8	1MB	4.25	34.45	1MB	3.92	34.28	1MB	3.76	34.21	1MB
4	4.73	36.69	1MB	4.15	36.4	1MB	3.87	36.26	1MB	3.74	36.2	1MB
8	4.49	38.57	1MB	4.04	38.35	1MB	3.82	38.24	1MB	3.72	38.18	1MB
12	4.28	40.47	1MB	3.94	40.3	1MB	3.78	40.21	1MB	3.69	40.17	1MB
16	4.09	42.37	1MB	3.85	42.25	1MB	3.73	42.19	1MB	3.67	42.16	1MB
20	3.88	44.27	1MB	3.72	44.19	1MB	3.64	44.15	1MB	3.6	44.13	1MB
24	3.42	46.04	1MB	3.32	45.99	1MB	3.27	45.96	1MB	3.25	45.95	1MB
28	3.28	47.97	1MB	3.21	47.93	1MB	3.18	47.92	1MB	3.16	47.9	1MB
32	3.14	49.9	1MB	3.14	49.9	1MB	3.14	49.9	1MB	3.14	49.9	1MB

$r=10$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	17.28	40.97	11MB	11.47	38.06	1MB	6.54	35.6	1MB	4.18	34.42	1MB
4	14.87	41.76	3MB	9.77	39.21	1MB	5.73	37.19	1MB	3.87	36.26	1MB
8	12.6	42.63	1MB	8.18	40.42	1MB	4.97	38.81	1MB	3.56	38.11	1MB
12	10.45	43.55	1MB	6.79	41.72	1MB	4.34	40.49	1MB	3.3	39.97	1MB
16	8.45	44.55	1MB	5.57	43.11	1MB	3.8	42.22	1MB	3.06	41.86	1MB
20	6.56	45.61	1MB	4.48	44.56	1MB	3.31	43.98	1MB	2.83	43.74	1MB
24	4.92	46.79	1MB	3.53	46.09	1MB	2.78	45.71	1MB	2.42	45.53	1MB
28	3.44	48.04	1MB	2.78	47.72	1MB	2.44	47.54	1MB	2.28	47.47	1MB
32	2.13	49.39	1MB	2.13	49.39	1MB	2.13	49.39	1MB	2.13	49.39	1MB

$r=10$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	3.17	33.91	1MB	2.77	33.71	1MB	2.6	33.62	1MB	2.53	33.59	1MB
4	3.06	35.85	1MB	2.72	35.69	1MB	2.58	35.62	1MB	2.52	35.59	1MB
8	2.94	37.8	1MB	2.68	37.66	1MB	2.56	37.61	1MB	2.51	37.58	1MB
12	2.83	39.74	1MB	2.63	39.64	1MB	2.54	39.6	1MB	2.5	39.58	1MB
16	2.73	41.69	1MB	2.59	41.62	1MB	2.52	41.59	1MB	2.49	41.57	1MB
20	2.61	43.63	1MB	2.51	43.58	1MB	2.47	43.56	1MB	2.45	43.55	1MB
24	2.28	45.47	1MB	2.22	45.44	1MB	2.2	45.42	1MB	2.19	45.42	1MB
28	2.2	47.43	1MB	2.16	47.41	1MB	2.15	47.4	1MB	2.14	47.39	1MB
32	2.13	49.39	1MB	2.13	49.39	1MB	2.13	49.39	1MB	2.13	49.39	1MB

$r=11$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	14.72	39.68	2MB	8.77	36.71	1MB	4.34	34.5	1MB	2.61	33.63	1MB
4	12.5	40.58	1MB	7.36	38.	1MB	3.8	36.23	1MB	2.45	35.55	1MB
8	10.43	41.54	1MB	6.06	39.36	1MB	3.3	37.98	1MB	2.29	37.47	1MB
12	8.49	42.57	1MB	4.94	40.8	1MB	2.89	39.77	1MB	2.15	39.4	1MB
16	6.7	43.68	1MB	3.98	42.32	1MB	2.54	41.59	1MB	2.02	41.34	1MB
20	5.06	44.86	1MB	3.15	43.9	1MB	2.22	43.44	1MB	1.89	43.27	1MB
24	3.64	46.15	1MB	2.42	45.54	1MB	1.86	45.25	1MB	1.63	45.14	1MB
28	2.44	47.54	1MB	1.91	47.28	1MB	1.66	47.16	1MB	1.56	47.11	1MB
32	1.49	49.07	1MB	1.49	49.07	1MB	1.49	49.07	1MB	1.49	49.07	1MB

$r=11$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	2.02	33.33	1MB	1.82	33.24	1MB	1.75	33.2	1MB	1.73	33.19	1MB
4	1.97	35.31	1MB	1.81	35.23	1MB	1.75	35.2	1MB	1.72	35.19	1MB
8	1.92	37.29	1MB	1.79	37.22	1MB	1.74	37.2	1MB	1.72	37.19	1MB
12	1.87	39.26	1MB	1.77	39.21	1MB	1.73	39.19	1MB	1.72	39.19	1MB
16	1.83	41.24	1MB	1.75	41.2	1MB	1.73	41.19	1MB	1.72	41.18	1MB
20	1.76	43.21	1MB	1.71	43.18	1MB	1.7	43.17	1MB	1.69	43.17	1MB
24	1.55	45.1	1MB	1.52	45.09	1MB	1.52	45.08	1MB	1.51	45.08	1MB
28	1.52	47.09	1MB	1.5	47.08	1MB	1.49	47.07	1MB	1.49	47.07	1MB
32	1.49	49.07	1MB	1.49	49.07	1MB	1.49	49.07	1MB	1.49	49.07	1MB

$r=12$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	12.45	38.55	1MB	6.53	35.59	1MB	2.78	33.71	1MB	1.66	33.16	1MB
4	10.43	39.54	1MB	5.39	37.02	1MB	2.45	35.55	1MB	1.59	35.12	1MB
8	8.55	40.6	1MB	4.37	38.51	1MB	2.15	37.4	1MB	1.52	37.09	1MB
12	6.81	41.73	1MB	3.51	40.08	1MB	1.91	39.28	1MB	1.46	39.06	1MB
16	5.24	42.95	1MB	2.8	41.72	1MB	1.71	41.18	1MB	1.41	41.03	1MB
20	3.85	44.25	1MB	2.2	43.43	1MB	1.54	43.09	1MB	1.35	43.	1MB
24	2.66	45.66	1MB	1.69	45.17	1MB	1.32	44.98	1MB	1.2	44.93	1MB
28	1.75	47.2	1MB	1.37	47.01	1MB	1.23	46.94	1MB	1.18	46.92	1MB
32	1.15	48.9	1MB	1.15	48.9	1MB	1.15	48.9	1MB	1.15	48.9	1MB

$r=12$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	1.38	33.02	1MB	1.3	32.98	1MB	1.28	32.97	1MB	1.28	32.96	1MB
4	1.36	35.01	1MB	1.3	34.98	1MB	1.28	34.97	1MB	1.28	34.96	1MB
8	1.35	37.	1MB	1.3	36.97	1MB	1.28	36.97	1MB	1.28	36.96	1MB
12	1.33	38.99	1MB	1.29	38.97	1MB	1.28	38.96	1MB	1.27	38.96	1MB
16	1.31	40.98	1MB	1.29	40.97	1MB	1.28	40.96	1MB	1.27	40.96	1MB
20	1.29	42.97	1MB	1.27	42.96	1MB	1.26	42.96	1MB	1.26	42.95	1MB
24	1.17	44.91	1MB	1.16	44.91	1MB	1.16	44.91	1MB	1.16	44.91	1MB
28	1.16	46.91	1MB	1.16	46.9	1MB	1.15	46.9	1MB	1.15	46.9	1MB
32	1.15	48.9	1MB	1.15	48.9	1MB	1.15	48.9	1MB	1.15	48.9	1MB

$r=13$	$w=0$			$w=1$			$w=2$			$w=3$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	10.45	37.55	1MB	4.73	34.69	1MB	1.78	33.22	1MB	1.2	32.93	1MB
4	8.62	38.64	1MB	3.86	36.26	1MB	1.62	35.13	1MB	1.18	34.91	1MB
8	6.93	39.79	1MB	3.09	37.87	1MB	1.47	37.06	1MB	1.15	36.9	1MB
12	5.41	41.03	1MB	2.47	39.56	1MB	1.35	39.	1MB	1.13	38.89	1MB
16	4.06	42.35	1MB	1.98	41.31	1MB	1.26	40.95	1MB	1.12	40.88	1MB
20	2.91	43.78	1MB	1.59	43.12	1MB	1.18	42.92	1MB	1.1	42.87	1MB
24	1.96	45.31	1MB	1.27	44.96	1MB	1.08	44.87	1MB	1.04	44.85	1MB
28	1.33	46.99	1MB	1.11	46.88	1MB	1.05	46.85	1MB	1.03	46.84	1MB
32	1.03	48.84	1MB	1.03	48.84	1MB	1.03	48.84	1MB	1.03	48.84	1MB

$r=13$	$w=4$			$w=5$			$w=6$			$w=7$		
k	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M	p	C_{tr}	M
0	1.1	32.88	1MB	1.08	32.87	1MB	1.08	32.86	1MB	1.08	32.86	1MB
4	1.1	34.87	1MB	1.08	34.87	1MB	1.08	34.86	1MB	1.08	34.86	1MB
8	1.09	36.87	1MB	1.08	36.87	1MB	1.08	36.86	1MB	1.08	36.86	1MB
12	1.09	38.87	1MB	1.08	38.87	1MB	1.08	38.86	1MB	1.08	38.86	1MB
16	1.09	40.87	1MB	1.08	40.86	1MB	1.08	40.86	1MB	1.08	40.86	1MB
20	1.08	42.86	1MB	1.07	42.86	1MB	1.07	42.86	1MB	1.07	42.86	1MB
24	1.03	44.84	1MB	1.03	44.84	1MB	1.03	44.84	1MB	1.03	44.84	1MB
28	1.03	46.84	1MB	1.03	46.84	1MB	1.03	46.84	1MB	1.03	46.84	1MB
32	1.03	48.84	1MB	1.03	48.84	1MB	1.03	48.84	1MB	1.03	48.84	1MB