Surveying And Reproducing MD5 Fast Collision Attack Algorithms

Meng Zheng
Electrical Engineering
Delft University of Technology
Delft, Netherlands
M.Zheng-3@student.tudelft.nl

Yuhang Tian
Embedded Systems
Delft University of Technology
Delft, Netherlands
Y.Tian-13@student.tudelft.nl

***Abstract*—This paper is the survey for exploring the MD5 fast collision based on M. Stevens' paper published in 2012 where he extends the research carried out by Wang et al. and shows that the original structure of differential path cannot be proved as satisfying the "sufficient conditions". Wang's collision method has the limitation of the identical prefix, whereas Steven breaks this limitation to make the method become chosen-prefix collision attacks, which is the main contribution for him compared to primitive achievements. Our work, in this survey, is mainly 1) to explain the significance of MD5 collision, 2) to reproduce the fast collision attack algorithm for MD5, including the methods of X. Wang, V. Klima and M. Steven 3) to compare the different implementations of collision above.**

***Index Terms*—MD5, collision attack, cryptography**

I. Introduction

This section will mainly demonstrate some fundamentals about the hash function and give a brief view of the Merkle-Damgard (MD) family, especially MD5.

For a one-way hash function, the collision attack is to find two different messages M and M' that have the identical hash values $H(M) = H(M')$. In terms of the Merkle-Damgard family, like MD5, the most efficient technology to find the collisions for them is *differential cryptanalysis*. The core idea is to investigate and control the differences between two preimages when they propagate through the hash blocks and to generate two identical digests at the end. Following this idea, Xiaoyun Wang et al., published a paper in 2004 and proposed an effective attack for MD5 and achieved the collision attack. The paper points out that using two consecutive blocks with some constraints can generate two 1024-bit messages with identical hash digests - M_1 varies from M_2 ($M_1 \neq M_2$) but $MD5(M_1, IHV_i) = MD5(M_2, IHV_i)$.

MD5 structure has the property that if $MD5(x) = MD5(y)$ then $MD5(x||z) = MD5(y||z)$ where "||" denotes concatenation. According to this structure-property and combining the result provided by Wang, Dan Kaminsky in 2005 created *Stripwire* to mislead integrity checking, and so did Ondrej Mikle. This kind of attacks based on Wang's knowledge is *identical-prefix collision attack*. However, this type of attacks is quite limited as it requires that the initial IHV for those two consecutive blocks ought to be the same.

Later on, in 2007, Marc Stevens made great progress who removed that constraint - no requirement for identical prefix anymore - and he generated *chosen-prefix collision attacks* for MD5 which was also his main contribution. Without the restriction of the identical prefix, he applied this attack to forgery certificates such as rogue X.509 CA. X.509 was widely adopted to guarantee the security of the HTTPS website. The result reassured that MD5 could not be a secure certificating scheme anymore.

The following report will be generally divided into four sections. Firstly, it will detail the structure of the MD5 algorithm and give demos with both python and java versions. Following that, it will demonstrate the identical-prefix collision attack created by Wang et al. which is a brilliant attacking method different from the original methods based on brute-forcing or birthday-paradox and will show how this affects the secrecy of MD5. After that, it will present some other modified or even advanced attacks based on the idea from Wang, and compare what are the main differences among them. Finally, it will give a summary that consists of the reflection for the project and suggestions for future work.

II. MD5 Message-Digest Algorithm

This section will mainly present the forward computation of the MD5 message-digest algorithm. It will divide the whole algorithm into several parts in order to analyze it. There are already exhaustive researches and sufficient papers relevant to the computing process of MD5, thus only the core concept will be detailed.

The MD5 message-digest algorithm, MD5 algorithm for short, takes the number of N 512 bits messages including padding and an initial 128 bits IHV as inputs to generate a 128 bits fingerprint. The whole algorithm process can be separated into three parts.

Part I - padding: Initially, it uses method 1 padding - append a bit '1' to the message and then append many bits '0' - to make the length of the padded message become 448 (mod 512). After that, for the last remaining 64 bits, it appends the length of the original unpadded message. As a consequence, the message length is now $512N$ where N is an integer.

Part II - segmentation: It will divide the padded message into N 512-bit segments, for each of which it will assign a block that takes a segment and a 128-bit IHV_{in} as inputs and output a 128-bit IHV_{out}. Except for the first block which

IHV is set manually, the input IHV_{in} of a block is given by the output IHV_{out} from the former one. All the blocks are concatenated in serial like Fig. 1.

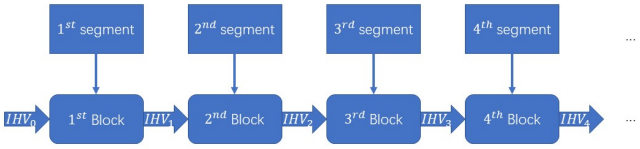

Fig. 1. Blocks Ordering

Part III - computation: For each block, it will continue to divide the input 512-bit segment into 16 words (32 bits). It will run 4 rounds and each round consists of 16 steps; as a result, there are 64 steps in total. Each step includes some bit-wise operations and the operations may vary among steps. One-step structure has been shown in Fig. 2. One-step process can be used mathematical formula to represent as:

$$(A', B', C', D') = (D', \mathcal{B}, B', C') \quad (1)$$

$$\mathcal{B} = RL[f_t(B, C, D) + A + m_i + K_i, s_i] + B \quad (2)$$

where RL is the Left Rotation function, f_t is the compression function varying from each other every 16 steps, m_i is the i^{th} word, K_i is the addition constant, s_i is the rotation constant. More details can be found in the appendix. If the current block is not the last, it will propagate the output to the next block as the input for the next on. Otherwise, in the final step of a block, it will add the current A', B', C', D' to the very beginning $IHV_{in} = A, B, C, D$ as the output IHV_{out}:

$$IHV_{out} = (A + Q61, B + Q64, C + Q63, D + Q62) \quad (3)$$

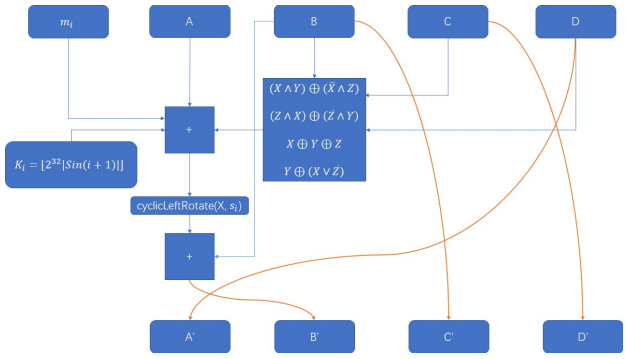

Fig. 2. MD5 Main Structure

For instance, the pre-image "TU Delft" is going to be computed by MD5. Before being fed into the blocks directly, it needs to be encoded and its original length should be calculated before the padding.

$$Encode(\text{"}TUDelft\text{"}) = 54552044\ 656c6674\ _{hex}$$
$$|Encode(\text{"}TUDelft\text{"})| = 40\ _{hex}$$

Then it should be padded as mentioned in part II. It should be noticed that the appending follows the *little-endian*.

$$54552044\ 656c6674\ 80000000\ 00000000$$
$$00000000\ 00000000\ 00000000\ 00000000$$
$$00000000\ 00000000\ 00000000\ 00000000$$
$$00000000\ 00000000\ 40000000\ 00000000$$

Since the total length after padding is merely 512 bits, it only requires one block to contain. After 64 steps computation shown in part III, the 128 bits digest can be obtained. The code for the whole process is provided: MD5.ipynb (requires Jupyter Notebook) and MD5.java (requires JDK).

$$467764e7\ f10c7769\ 5db73ab3\ d414c65d\ _{hex}$$

The MD5 algorithm invented by Ronald Rivest in 1992. Although it met some threatens in 1996, it had been used for approximately 12 years before Wang et al. found an effective way to find a collision. In the next section, it will show how Wang's elegant approach makes the MD5 algorithm become history.

III. Identical-Prefix Attack

This section will mainly introduce the method created by Wang et al., who published her paper in 2004 where she unprecedentedly found an efficient scheme to build a collision attack for MD5. After that, it will provide the idea of how to build a real collision from shallow to deep.

A. Wang Collision Theory

The theorem behind the identical-prefix attack invented by Wang et al. is *differential path*. Taking two messages $M = \{m_0, m_1, ..., m_{n-1}\}$ and $M' = \{m'_0, m'_1, ..., m'_{n-1}\}$ ($\exists i \in [0, k)\ s.t.\ m_i \neq m'_i$ and $\forall i \in [k, n)\ s.t.\ m_i = m'_i$) and identical prefix $IHV_0 = IHV'_0$ as the initial inputs, the initial difference between IHVs is 0, denoted as $\Delta IHV_0 = 0$. When these two different messages propagate through two different paths consisting of MD5 blocks, we record the intermediate IHVs of these two paths as:

$$\Delta IHV_0 \overset{m_0, m'_0}{\longrightarrow} \Delta IHV_1 \overset{m_1, m'_1}{\longrightarrow} \Delta IHV_2 \rightarrow$$
$$... \rightarrow \Delta IHV_{n-1} \overset{m_{n-1}, m'_{n-1}}{\longrightarrow} \Delta IHV_n$$

If at a position where the intermediate IHVs become no differences again that $\Delta IHV_k = 0\ (k \neq 0)$, from then on, all the remaining blocks' outputs will be identical $\Delta IHV_i = 0\ (i \geq k)$ due to the property of MD5 mentioned in section I. If any two non-identical messages found can generate this phenomenon, then the *differential path collision* is found. Based on this differential path idea, Wang uses two consecutive blocks, fabulously, the latter of which can eliminate the differences caused by the former one, like Fig. 3 shown.

In order to use the latter block to cancel the variations caused by the former one, there are some restrictions for the two 1024-bit messages also with some sufficient conditions for the intermediate IVs - the IV represents the temporary output

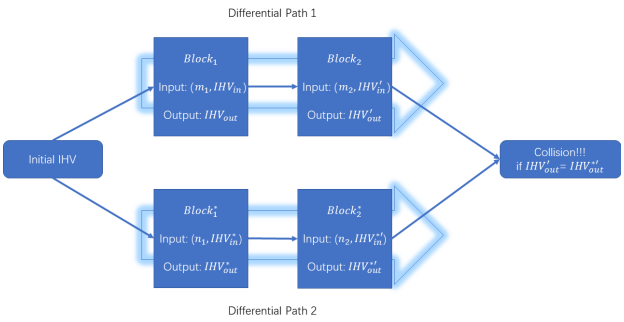

Fig. 3. Differential Paths

for each step. 1) For two 512-bit messages of the first block, $\delta m_4 = 2^{31}$, $\delta m_{11} = 2^{15}$, $\delta m_{14} = 2^{31}$ and other $\delta m_i = 0$.

$$\delta m_{14} = 2^{31}, \delta m_{11} = 2^{15}, \delta m_{14} = 2^{31}, \delta m_{i \in other} = 0 \quad (4)$$

2) For the second block two 512-bit messages, $\delta m_4 = 2^{31}$, $\delta m_{11} = 2^{-15}$, $\delta m_{14} = 2^{31}$ and other $\delta m_i = 0$.

$$\delta m_{14} = 2^{31}, \delta m_{11} = -2^{15}, \delta m_{14} = 2^{31}, \delta m_{i \in other} = 0 \quad (5)$$

3) For the output IHV_1 and IHV_1' of the first two blocks on different paths - $\delta IHV_1 = \{\delta IV_{1A}, \delta IV_{1B}, \delta IV_{1C}, \delta IV_{1D}\}$ represents their differences, it requires that $\delta IV_{1A} = 2^{31}$, $\delta IV_{1B} = 2^{31}+2^{25}$, $\delta IV_{1C} = 2^{31}+2^{25}$ and $\delta IV_{1D} = 2^{31}+2^{25}$.

$$\delta IHV_1 = \{2^{31}, 2^{31}+2^{25}, 2^{31}+2^{25}, 2^{31}+2^{25}\} \quad (6)$$

4*) For the output IHV_2 and IHV_2' of the second two blocks, $\delta IV_{2A} = 0$, $\delta IV_{2B} = 0$, $\delta IV_{2C} = 0$ and $\delta IV_{2D} = 0$.

$$\delta IHV_2 = \{0, 0, 0, 0\} \quad (7)$$

These four requirements are preliminary to generate the cancellation and get the *differential path collision* at the end. The more specific conditions for each step's intermediate IV has been shown in the appendix.

The conditions mentioned above can increase the probability of finding a collision successfully at the end. Compared with the brute-force attack which result can only be verified at the final stage, the sufficient conditions between the steps can determine the running program whether it needs to restart searching halfway when any of the conditions are not fulfilled.

B. Wang Collision Implementation

Starting from this paragraph, it will demonstrate more details about how to implement a collision algorithm from a draft intuitive version to a better and completed one.

By observing the block structure of MD5 in Fig. 2 along with its one-step computation function Eq. 1, we can find that some parameters like addition constant, rotation constant and compress function are fixed for each step, and the input word m_i acts like a variable which determines the output $\{A, B, C, D\}$. Furthermore, most of the operations are added to B whereas A, C and D are only changed the positions after each step. That is the reason why only B has a condition table.

Since we have known all the restrictions/conditions on Q_t (the t^{th} step B, here $t = i$), intuitively, we can randomly generate a m_i corresponding for a step and make a loop until it finds a suitable m_i that generates the B satisfying Q_t for this step. It looks feasible, but beginning from the second block, there are two inputs which are random word m_i and intermediate value IV_{i-1} generated from the former random m_{i-1}.

In order to avoid introducing two randomness at one time, we can use the "inverse way". As the conditions of Q_ts are predefined, we can use *pseudo number generator* to generate the Q_t which suffices the conditions, and use this formula to reversely compute the word m_i:

$$m_i = RR(Q_{i+1} - Q_i, s[i]) - f_t(Q_i, Q_{i-1}, Q_{i-2}) - Q_{i-3} - K[i] \quad (8)$$

where RR is the Right Rotation function, Q_t is the intermediate output value of the t^{th} step. Eq. 8 reveals that the 16 words $(m_0, m_2, .., m_{15})$ can be determined only by the initial seventeen intermediate output $(Q_0, Q_1, ..., Q_{16})$. For each m_t, it needs 5 Qs to decide. For instance, calculating m_6 needs foreknowledge about Q_7, Q_6, Q_5, Q_4 and Q_3. Therefore, we can randomly generate satisfying $Q_t, 3 \leq t \leq 16$ (In terms of Wang's version, there is no restrictions for Q_1 and Q_2) and reversely compute the corresponding m_t, then use the known Q_t and calculated m_t to forward compute the remaining Q_ts. The pseudo codes for the two blocks are provided in the appendix, Alg. 1.

The original code of Wang's version cannot be found through the internet, but there exist some duplicates made by other researchers. The above pseudo-code is made according to S. Thomsen's wangmd5.c who claims that the idea is retrieved from Wang. We provide WangMD5.cpp and WangMD5.java versions. It approximately costs 20 to 30 minutes to find a pair of messages which outputs the identical digests. In fact, The structure of the code can be further improved. updated version.

C. Implementation Along with Tunnels

Other than improving the structure, in 2006, V. Klima generates a new idea of improving the previous method of finding collisions. He substitutes the multi-message modification in the original code by tunnels which exponentially increases the finding speed of collision. Using his method, it costs an average of fewer than 2 minutes to find a collision pair.

identical-prefix attack application

Thereafter, people can apply this method to find as many collisions. However, Wang's method has two shortages. The first one is that the collision messages are randomly generated which are always meaningless ones. The second one is that the colliding messages must be generated simultaneously along differential paths.

You can start writing from here

IV. Appendix

Algorithm 1 Wang Collision Algorithm

Input: 128-bit Initial $IHV_0 = \{A, B, C, D\}$
Output: 128-bit Digest $IHV_2 = \{\hat{A}, \hat{B}, \hat{C}, \hat{D}\}$

while Block 1 conditions fulfilled **do**
  while conditions fulfilled **do**
    generate Q[3], Q[4], ..., Q[16]
    testing several of them each time, and for each time,
    if test not passed **then**
      continue
    end if
  end while
  calculate x[6] to x[15] by Eq. 8
  for $i \leftarrow 0$ to $ffff_{hex}$ **do**
    generate Q[17]
    calculate Q[18],[Q19] by Eq. 1
    if test passed **then**
      break
    end if
  end for
  calculate x[1] by Eq. 8
  Q[20] $\leftarrow$ Q[19]&80000000_{hex}
  for Q[20]$\neq$!$Q[19]\&80000000_{hex}$ **do**
    calculate x[0],x[1],x[2],x[3],x[4],x[5],Q[1],Q[2]
    calculate Q[21] to Q[64], for each time of calculation,
    if test not passed **then**
      continue
    end if
  end for
  $IHV_1 = \{IHV_{0A} + Q[61], IHV_{0B} + Q[64], IHV_{0C} + Q[63], IHV_{0D} + Q[62]\}$
end while

while Block 2 conditions fulfilled **do**
  while conditions fulfilled **do**
    generate Q[1] and Q[2]
    if test not passed **then**
      continue
    end if
    calculate xx[1]
    ... (the same for Q[3] to Q[12])
    generate Q[13]
    if test not passed **then**
      continue
    end if
    calculate xx[13]
  end while
  for $i \leftarrow 0$ to 10000_{hex} **do**
    generate Q[15]
    if test not passed **then**
      continue
    end if
    calculate xx[14]
    generate Q[16]
    calculate xx[15]
    calculate Q[17]
    if test not passed **then**
      modify xx[1]
      update Q[2]
      recalculate Q[17]
      if Q[2] condition not fulfilled **then**
        break
      end if
      recalculate xx[2],xx[3],xx[4],xx[5]
    end if
    ... (the same for Q[18] and Q[19] calculate Q[20] to Q[64], for each time of calculation,
    if test not passed **then**
      continue
    end if
    break
  end for
  $IHV_2 = \{IHV_{1A} + Q[61], IHV_{1B} + Q[64], IHV_{1C} + Q[63], IHV_{1D} + Q[62]\}$
end while