Delft
University of
Technology

Surveying And Reproducing MD5 Fast Collision Attack Algorithms

Students: Yuhang Tian & Meng Zheng

Supervisor: Professor Kaitai Liang

IN4253ET ”HACKING LAB”-APPLIED SECURITY ANALYSIS

Table of Contents

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 Stevens 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. Stevens 3) to compare the above different implementations of collision.

Keywords: MD5, collision attack, cryptography.

Chapter 1

Introduction

1.1 Overview

This section will mainly give some fundamentals about the hash function. Then it will give a brief view of the Merkle-Damgard (MD) family with the attacks focusing on it, especially MD5. We select the attacks generated by Xiaoyun Wang, Marc Stevens and Vlastimil Klima. The selections are based on several reasons. In terms of Wang's attack, she is the first one who creates the effective scheme to find the collision which also inspires many later researchers. Therefore, we take her approach into consideration. From the perspective of Stevens' attack, he proves that the bit conditions in Wang's paper are insufficient and updates the scheme. Thus, we will also consider his method to see the differences comparing to Wang's. In Stevens' paper, he also suggests an idea of *Tunnelling* which has the ability to accelerate the searching speed. As consequence, Klima's attack will be presented in our report, as his scheme indeed speeds up the collision finding significantly. The final part will briefly demonstrate the structure of this report.

1.2 One-Way Hash Function

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 Fig. 1.1, it describes the security game version of collision resistance, where the parameters in blue colour are pre-known for the adversary whereas the parameter in red colour is not. In reality, a useful hash function is required to have that resistance.

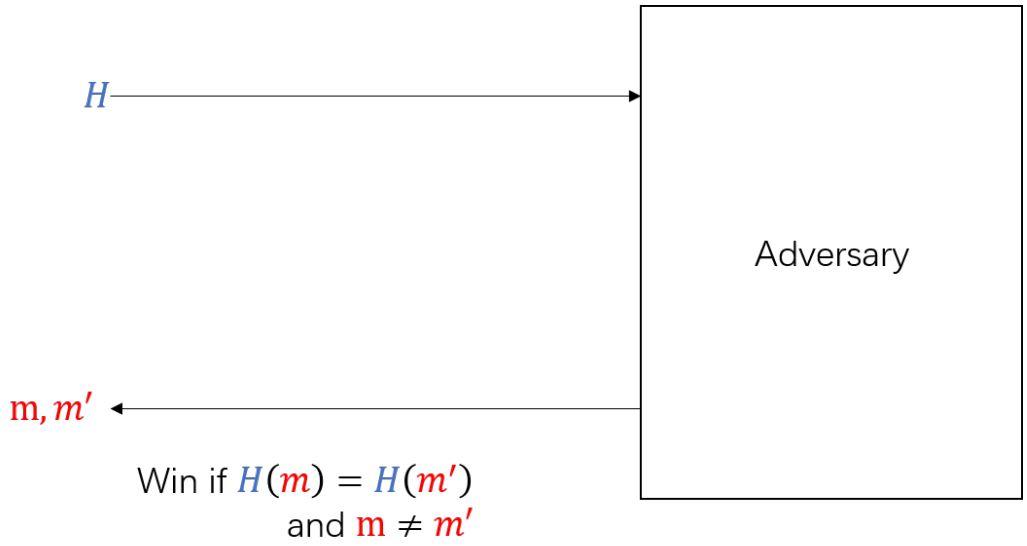


Figure 1.1: Security Game Version - Collision Resistance

1.3 Wang's Attack

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 so as to generate two identical digests at the end. Following this idea, Xiaoyun Wang et al. published a paper in 2004 and proposed an effective attacking scheme for MD5 and vanquish its collision resistance. The paper points out that using two 1024-bit messages and two con-

secutive blocks with some constraints can generate two 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.

1.4 Klima's Attack

Vlastimil Klima in 2006 generated a new scheme based on "*point of verification*" (*POV*). He substituted the original *multi-message modification methods* - try best to modify the messages to satisfy the conditions (the constraints after shifting and bits conditions) from the beginning to the end - by his tunnels, which exponentially accelerates the collision search. The original so-called "sufficient conditions" in Wang's paper were insufficient which had been improved latterly. The implementation of Klima's attack in this report uses the latest updated sufficient conditions.

1.5 Steven's Attack

Later on, in 2007, Marc Stevens made great progress who removed the prefix constraint - no requirement for identical prefix anymore - and 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 certifi-

cates 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.

1.6 Organisation of the report

This section gives a brief introduction of the background for the project. The following report will be generally divided into four sections. Firstly, it will detail the structure of the MD5 algorithm and give two demos, a python version and a java version. Following that, it will demonstrate the attacking methods of MD5. It will initially show 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, and then it will present some other modified or even advanced attacks inherited the idea from Wang. Then, the report will quantitatively compare the performances among them. Finally, it will give a summary that consists of the reflection for the project and suggestions for future work.

Chapter 2

MD5 Message-Digest Algorithm

2.1 Overview

This section will mainly present the forward computation of the MD5 message-digest algorithm. It will divide the whole algorithm into three 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.

2.2 MD5 block structure

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. 2.1.

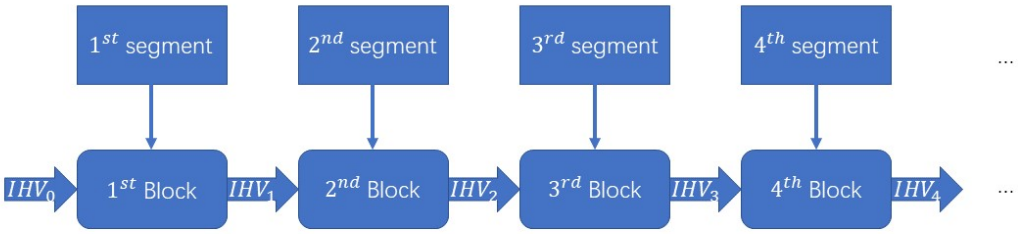


Figure 2.1: Blocks Ordering

Part III - computation: For each block, it will continue to divide the input 512-bit segment into 16 words (32 bits). A block process contains 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 which vary among steps. One-step structure has been shown in Fig. 2.2. The one-step process can be used a mathematical formula to represent as:

$$(A', B', C', D') = (D', \mathscr{B}, B', C') \tag{2.1}$$

$$\mathscr{B} = RL[f_t(B, C, D) + A + m_i + K_i, s_i] + B \tag{2.2}$$

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 adding constant, s_i is the rotation constant. 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 $temp = A', B', C', D'$ to the very

beginning $IHV_{in} = A,B,C,D$ as the output IHV_{out}:

$$IHV_{out} = (A + Q[61], B + Q[64], C + Q[63], D + Q[62]) \tag{2.3}$$

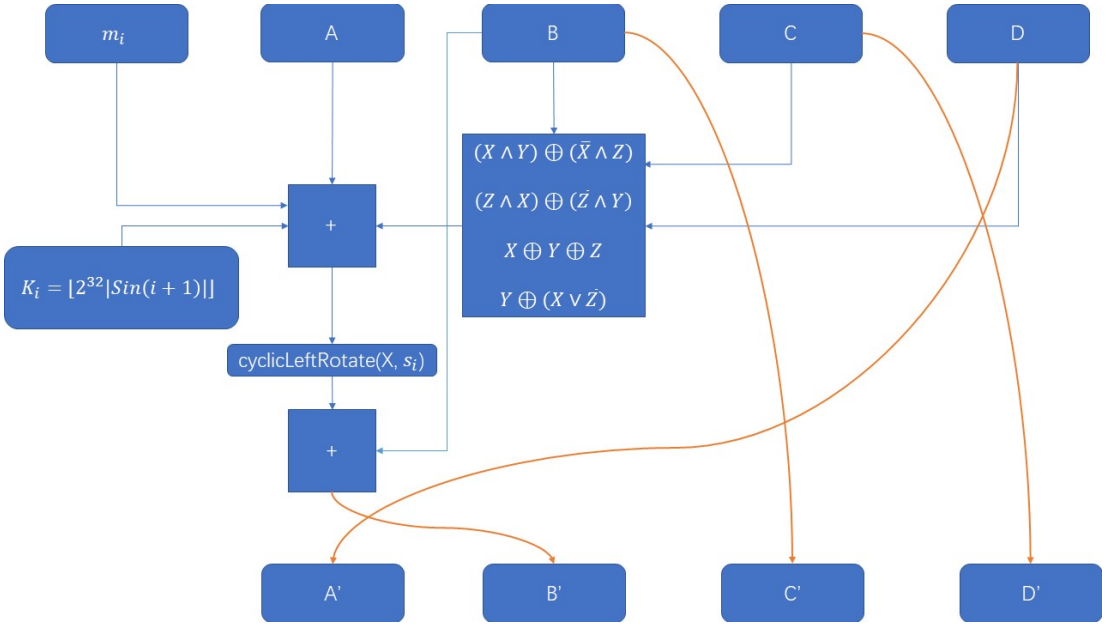


Figure 2.2: MD5 Main Structure

2.3 More Details of A Block

- Block's inputs:
  - The input 16 words will be reused in order to feed the 64 steps. The rule is shown in the following Eq. 2.4.

$$w[t] = \begin{cases} m[t] & \text{for } 0 \le t < 16 \\ m[1+5t \mod 16] & \text{for } 16 \le t < 32 \\ m[5+3t \mod 16] & \text{for } 32 \le t < 48 \\ m[7t \mod 16] & \text{for } 48 \le t < 64 \end{cases} \tag{2.4}$$

  - For the first block, the IHV should be set manually. Normally, people will use the default value (67452301, efcdab89, 98badcfe, 10325476

$)_{hex}$. However, it can be set by the user as well.

- Adding constant: The adding constant is generated by $2^{32} \times \sin(t+1)$, and if the calculating result is not an integer, it will use the floor value. It can use Eq. 2.5 to represent.

$$K[t] = \lfloor 2^{32} |sin(t+1)| \rfloor \tag{2.5}$$

- Rotation constant: The rotation constants vary from each round. They can be represented by Eq. 2.6.

$$s[t], s[t+1], s[t+2], s[t+3] = \begin{cases} 7, 12, 17, 22 & \text{for } t = 0, 4, 8, 12 \\ 5, 9, 14, 20 & \text{for } t = 16, 20, 24, 28 \\ 4, 11, 16, 23 & \text{for } t = 32, 36, 40, 44 \\ 6, 10, 15, 21 & \text{for } 48, 52, 56, 60 \end{cases} \tag{2.6}$$

- Compression function; The compression functions used for adding the non-linearity vary from each round. They can be represented by Eq. 2.7. It should be noticed that the representation is not unique.

$$f_t(X,Y,Z) = \begin{cases} F(X,Y,Z) = (X \wedge Y) \oplus (\bar{X} \wedge Z) & \text{for } 0 \leq t < 16 \\ G(X,Y,Z) = (Z \wedge X) \oplus (\bar{Z} \wedge Y) & \text{for } 16 \leq t < 32 \\ H(X,Y,Z) = X \oplus Y \oplus Z & \text{for } 32 \leq t < 48 \\ I(X,Y,Z) = Y \oplus (X \vee \bar{Z}) & \text{for } 48 \leq t < 64 \end{cases} \tag{2.7}$$

2.4 MD5 Example

To help readers understand the process clearly, this section will provide an example of hashing ”TU Delft”. Also, it will provide two demos that readers can download and play with them.

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*.

$$\begin{array}{cccc} 54552044 & 656c6674 & 80000000 & 00000000 \\ 00000000 & 00000000 & 00000000 & 00000000 \\ 00000000 & 00000000 & 00000000 & 00000000 \\ 00000000 & 00000000 & 40000000 & 00000000 \end{array}$$

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 was invented by Ronald Rivest in 1992. It met some threatens in 1996, still, 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 et al.'s elegant approaches make the MD5 algorithm become history.

Chapter 3

Collision Attack Methods

3.1 Overview

This section will provide the idea of how to build a real collision from shallow to deep. It will first introduce the basic common theory of collision attack. The theory was mainly provided by Wang who published her paper in 2004 where she unprecedentedly found an efficient scheme to build a collision attack for MD5. Besides, the theory is also the backbone of other schemes, inspiring many researchers. Following that it will sequentially show how Wang, Klima and Stevens implement their attacks to find a collision. Thirdly, it will show the significance and some relevant applications of the above-attacking schemes which are also called *identical-prefix attack*s. Then, it will briefly demonstrate a more advanced attacking scheme *chosen-prefix attack* which is created by Stevens, also with its application. Since the *chosen-prefix attack* is not a fast collision method, this report will not give many details, but readers can find an exhaustive analysis in his PhD thesis. In order to guide the readers to reproduce the results made by us, we will show more instructions in the appendix 5.

3.2 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. 3.1 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 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 $\delta m_i = 0$.

$$\delta m_{14} = 2^{31}, \delta m_{11} = 2^{15}, \delta m_{14} = 2^{31}, \delta m_{i \in other} = 0 \tag{3.1}$$

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

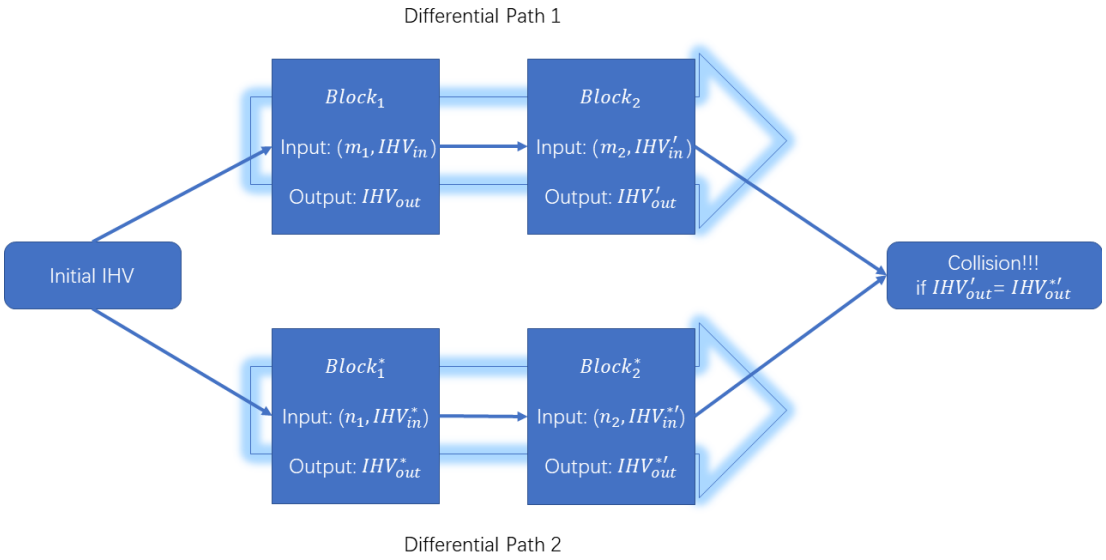


Figure 3.1: Differential Paths

2^{31} and other $\delta m_i = 0$.

$$\delta m_{14} = 2^{31}, \delta m_{11} = -2^{15}, \delta m_{14} = 2^{31}, \delta m_{i \in other} = 0 \tag{3.2}$$

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 $\delta IV_{1D} = 2^{31} + 2^{25}$.

$$\delta IHV_1 = \{2^{31}, 2^{31} + 2^{25}, 2^{31} + 2^{25}, 2^{31} + 2^{25}\} \tag{3.3}$$

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 $\delta IV_{2D} = 0$.

$$\delta IHV_2 = \{0, 0, 0, 0\} \tag{3.4}$$

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. Furthermore, the multi-message modification can happen during the verification, since there is still some freedom to adjust the output, which means that it is not essential to jump back to the initial step after failing to pass the testing.

3.3 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 complete one. The difficulty of reproducing Wang's collision attack is that the original website which presents the implementation and code expires, so we cannot find the original one. In her paper, she details the theoretical background but does not exhaustively show the idea of implementing. Fortunately, We find a paper and project website provided by Thomsen wangmd5.c who claims that the idea is retrieved from Wang.

By observing the block structure of MD5 in Fig. 2.2 along with its one-step computation function Eq. 2.2, 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 randomnesses 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:

$$m_i = RR(Q_{i+1} - Q_i, s[i]) - f_t(Q_i, Q_{i-1}, Q_{i-2}) - Q_{i-3} - K[i] \tag{3.5}$$

where RR is the Right Rotation function, Q_t is the intermediate output value of the t^{th} step. Eq. 3.5 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 preknowledge 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 below Alg. 1.

Algorithm 1 Wang Collision Algorithm

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

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

1: Randomly generate Q[3] to Q[8]

- Use constraints to test them during the generation
- If it cannot satisfy the constraint, then goes back to 1

2: Randomly generate Q[9] to Q[12]

(i) Till generate valid Q[9] to Q[11]

(ii) Generate Q[12] and if it cannot satisfy the constraint, then goes back

to i

3: Randomly generate Q[13]

4: Randomly generate Q[14] to Q[16]

- Use constraints to test them during the generation
- If it cannot satisfy the constraint, then goes back to 4

5: Calculate x[6] and x[11] by Eq. 3.5

6: Initialize ctr=0 and If ctr=0xffff, then it goes back to 1

- Randomly generate Q[17]
- Calculate Q[18] and Q[19] by Eq. 2.2 and if they satisfy the constraint, then goes to 7
- ctr++

7: Calculate x[1]

8: Exhaustively traverse Q[20] then goes back to 1

- Calculate the rest of x[ ]s and remaining Q[ ]s
- If it cannot satisfy the constraint, then goes back to 8
- If all the constraints are fulfilled, then it gives the output of block 1

The original coding language is C, and we use Java to reproduce it. In addition, the original structure of the second block is blurry, so we rearrange the structure to make it more sensible. Besides, the original code calculates x[i]s during the testing, but it is not necessary. We put them afterwards so as to avoid redundant calculations. In general, it can improve about 15% of the searching speed and significantly promote efficiency. Furthermore, we also prune our code according to the feedback by SonarQube which can automatically analyze the quality of the code and give suggestions for tuning.

It approximately costs about 15 minutes to find a pair of messages which outputs the identical digests. We provide WangMD5.cpp and WangMD5.java versions.

3.4 Marc Stevens Fast Collision Implementation

This section will focus on the implementation of Marc Stevens fast collision algorithm, after Wang proposed her genius work on MD5 Collision theory, Marc Stevens pointed out her sufficient conditions are not sufficient at all, furthermore, he also proposed one search algorithm for finding collision, at the same time he optimized block conditions based on Wang's work. This section will briefly introduce this algorithm, and also show the work of implement based on Python. This algorithm can choose message blocks that satisfy the set of sufficient conditions for the first round deterministically, however, there are more restrictions both on Q and T, before going deeper to introduce this algorithm, one thing shall be explained here as background knowledge, this algorithm did not find messages that generated randomly, on the contrary, it first randomly generated Q which fulfil optimized sufficient requirements, then by applying function 3.5, the message can be calculated deterministically. The algorithm used for the second block is proposed by Klima, Marc proposed a new similar algorithm for the first block, both algorithms used function 3.5 to calculate deterministic messages. The details of the algorithm are shown below.

Algorithm 2 Marc Stevens Block 1 search algorithm

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

Output: 128-bit Digest $IHV_1 = \{\hat{A}, \hat{B}, \hat{C}, \hat{D}\}$

1. Choose Q[1], Q[3], . . . , Q[16] fulfilling conditions;
2. Calculate m[0], m[6], . . . , m[15];
3. Loop until counter is full:
    a. Choose Q[17] fulfilling conditions;
    b. Calculate m[1] at t = 16;
    c. Calculate Q[2] and m[2], m[3], m[4], m[5];
    d. Calculate Q[18], . . . , Q[21];

e. Check Q[17], . . . , Q[21] are fulfilling conditions, if satisfy break and set counter to zero, else back to step 3

4. Check counter, if zero go to step 5, else go back to step 1
5. loop until second counter is full:
   a. loop all possible Q[9] and Q[10] satisfying m[11] does not change
   b. Calculate m[8], m[9], m[10], m[12], m[13];
   c. Calculate Q[22], . . . , Q[64];
   d. Check Q[22], . . . , Q[64], T22, T34 and the iv-conditions for the next block are fulfilling conditions, if satisfy go to step 6 and set second counter to zero, else back to step 5
6. Check second counter, if it is zero output IHV_1 and go to second block else back to step 1

Algorithm 3 Marc Stevens Block 2 search algorithm

Input: 128-bit Initial $IHV_1 = \{A, B, C, D\}$

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

1. Choose Q[2], Q[3], . . . , Q[16] fulfilling conditions;
2. Calculate m[5], m[6], . . . , m[15];
3. Loop until counter is full:
   a. Choose Q1 fulfilling conditions;
   b. Calculate m[0],.....,m[4];
   d. Calculate Q[17], . . . , Q[21];
   e. Check Q[17], . . . , Q[21] are fulfilling conditions, if satisfy break and set counter to zero, else back to step 3;
4. Check counter, if zero go to step 5, else go back to step 1
5. loop until second counter is full:
   a. loop all possible Q[9] and Q[10] satisfying m[11] does not change

b. Calculate m[8], m[9], m[10], m[12], m[13];

c. Calculate Q[22], . . . , Q[64];

d. Check Q[22], . . . , Q[64], T22, T34 are fulfilling conditions, if satisfy go to step 6 and set second counter to zero, else back to step 5;

6. Check second counter, if it is zero output IHV_2 and collision found, else go back to step 1.

Here are also several points to address, first, this algorithm is based on 'optimized' sufficient conditions, which is addressed in Appendix A of [3], here more restrictions on T are applied, then algorithm 2 and algorithm 3 are different from Marc Steven's latest version of fast-coll, for his latest fast-coll algorithm, Tunnels were applied, which improved performance greatly, more details of Tunnels shall be shown in next section, also, Marc improved his algorithm by making changes on step 3 and 5 to improve the performance, most importantly by adding restrictions on T for both blocks, this algorithm guarantees a collision if all 'optimized' sufficient conditions are fulfilled for both blocks. Finally, we want to end up this section by pointing out several issues we tackled when implementing this algorithm with Python, first of all, as Python store data different from Java and C++ which is widely used by this algorithm, we need to add overflow with python, which means if any number is above this limitation, it shall return back with overflow control. Also, it is even harder for handling with transforms between different bases of data, as Python hold different bin and hex number for negative numbers compared with Java and C++.

3.5 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. When implementing this scheme, not many difficulties are met, as the structure itself is clear and coherent. They rigorously use 3 generations to update their code, so not many improvements can be done for the latest version.

In the previous section, multi-message modification is based on that if $M - M'$ fulfills the specific constraints C, then $MD5(M) - MD5(M')$ fulfills the corresponding constraints C^*, so the "sufficient conditions" are designed for this purpose. However, in terms of tunnelling, it creates multiple subsets of sufficient conditions to segment the original testing process, and at the same time, it guarantees that if message M fulfills those conditions, then the transformed message can also fulfill. Instead of using multi-message modification, tunnels can significantly reduce the searching time and increase the probability of meeting the specific conditions.

The pseudo code for the implementation can be found below Alg. 4. In block 1, tunnels $Q_{10}, Q_{20}, Q_{13}, Q_{14}, Q_4$ and Q_9 are employed, while in block 2, tunnels $Q_{16}, Q_{1,2}, Q_4$ and Q_9.

Algorithm 4 Klima Collision Algorithm

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

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

1: **while** block 1 conditions fulfilled **do**

2: generate Q[1], Q[3], ..., Q[17] fulfilling conditions

```
3:      calculate x[0], x[1], ..., x[15] by Eq. 3.5
4:      calculate Q[2], Q[18], ..., Q[24] by Eq. 2.2
5:      for itr_Q10 ← 0 to 2^3 do
6:          Tunnel Q_10
7:          for itr_Q20 ← 0 to 2^6 do
8:              Tunnel Q_20
9:              for itr_Q13 ← 0 to 2^12 do
10:                 Tunnel Q_13
11:                 for itr_Q14 ← 0 to 2^9 do
12:                     Tunnel Q_14
13:                     for itr_Q4 ← 0 to 2^0 do
14:                         Tunnel Q_4
15:                         for itr_Q9 ← 0 to 2^3 do
16:                             Tunnel Q_9
17:                             calculate Q[25], Q[26], ..., Q[64] by Eq. 2.2
18:                             calculate IHV_1 by Eq. 2.3
19:                         end for
20:                     end for
21:                 end for
22:             end for
23:         end for
24:     end for
25: end while
26:
27: while block 2 conditions fulfilled do
28:     generate Q[1], Q[3], ..., Q[14] fulfilling conditions
29:     for itr_Q16 ← 0 to 2^25 do
30:         Tunnel Q_16
31:         for itr_Q1,Q2 ← 0 to 2^(Q1Q2_strength) do
32:             Tunnel Q_1,2
33:             for itr_Q4 ← 0 to 2^6 do
```

```
34:                 Tunnel Q_4
35:                 for itr_Q9 ← 0 to 2^8 do
36:                     Tunnel Q_9
37:                     calculate Q[25], Q[26], ..., Q[64] by Eq. 2.2
38:                     calculate IHV_2 by Eq. 2.3
39:                 end for
40:             end for
41:         end for
42:     end for
43: end while
```

The original coding language is C, and we use Java to reproduce it. In his code, he uses a function "*mask_bit*" to generate the masking bits based on the step conditions. However, we think this function is not efficient, as the masking bits are commonly used among the tunnels, so it is redundant and unnecessary to call it every time when using it, especially in the loop. Thus, we calculate them before running the program. However, the improvement is not significant, since the speed is originally fast.

We provide java version source code KlimaMD5.java and jar application KlimaMD5.jar for the testing.

3.6 Identical Prefix Attack Applications

Thereafter, people can apply those methods to find as many collisions. In the same year after Wang et al. published their paper, D. Kaminsky preemptively applied their scheme to prove that MD5 Collision adversely impacted file-oriented system auditors, like Tripwire, because Tripwire was usually configured to MD5 hashes by the users. However, in his conclusion, he claimed that the

influence was not catastrophic even thought the ”functionally identical” of MD5 was broken.

Similarly, O. Mikle in 2004, used the knowledge of MD5 Collision trying to attack digital signature with GPG. He claimed that since the scheme was not about the second pre-image attack, the harms of collision would not be serious.

According to the above declarations, 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.

3.7 Chosen Prefix Attack

The chosen prefix attack is proposed by M. Stevens and defined as the following: for any two chosen message prefixes P and P', suffixes S and S' can be constructed such that the concatenated values $P||S$ and $P'||S'$ form a chosen-prefix collision for MD5. For the perspective of identical prefix attack, it requires the initial IHV_0 - it can be the digest of the previous block - should be the same for two differential paths, which means that all the block messages before two consecutive collision blocks should be the same (this is not rigours explanation, since different messages may generate the identical digests, but it is infeasible). In contrast, the Chosen Prefix Attack does not have such requirements. The structure of the chosen prefix attack has been shown in Fig. 3.2.

The details of the chosen-prefix collision implementation will not be presented in this paper, as it is not a fast collision method, but it extends the traditional identical-prefix and further damages the secrecy of MD5. Besides, M. Stevens provides a method that only employs one single block to generate the collision.

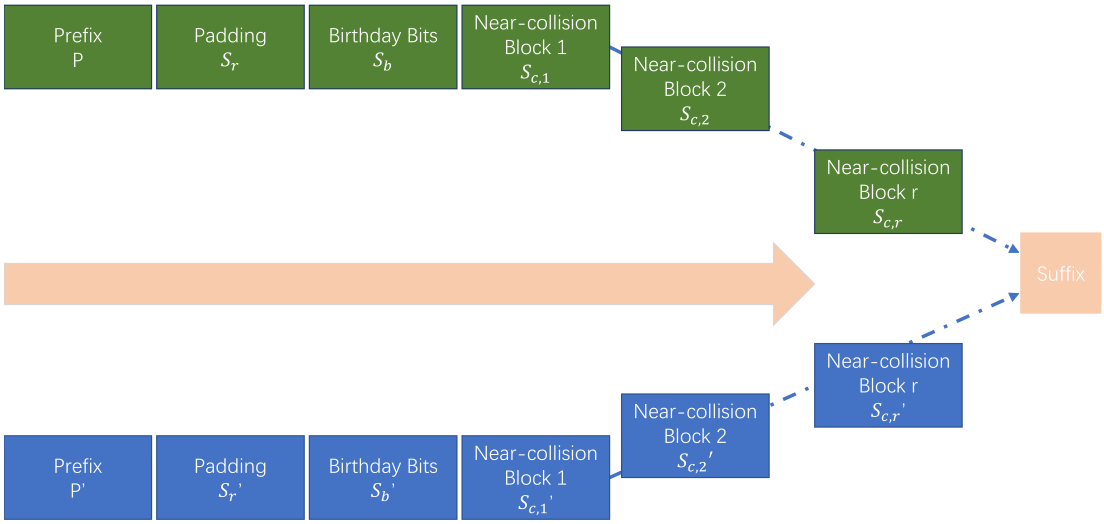


Figure 3.2: Chosen-prefix Attack Structure

3.8 Chosen Prefix Attack Application

Since the chosen prefix attack overcomes the barriers of identical prefix attack to some extent, it can be more easily and widely abused in many scenarios. For instance, inserting two images as suffixes into two different documents. The messages before the colour images act like the prefixes. The MD5 of the message in one document varies from another, but after patching with the colour images, they become the same. In other words, in this situation, $P||S$ and $P'||S'$, P is the actual message containing the correct information whereas P' is the forgery message containing the misleading information. People may point out that the colour image S is different from the image S', but in reality, it quite difficult to distinguish them using human's vision because the attacker can hide the differences in the pixels if he elaborates them carefully. The image can be a barcode or a logo that usually appear ubiquitously.

If readers are interested in such an application, the website built by M. Stevens provides the way to forgery the X.509 certificates.

Chapter 4

Attacks Comparison

4.1 Overview

This section will provide the results of the execution time for different schemes and mark their features. After that, it will also provide some self reflections of the whole project.

4.2 Comparison Results

Table 4.1: Comparison of MD5 Collision Schemes

Name	Searching Time	Feature
Wang Modified Version	15 minutes	multi-message modification
Klima Version	1.5 minutes	Tunnels, PoV
Stevens CPC Version	1 hours	prefixes can be chosen

The above testing results are based on CPU AMD 4900H. According to Tab. 4.1, the fast algorithm to find a collision pair of MD5 is the one provided by Klima equipped with tunnels. It should be mentioned that Klima's method also owns the least time variance. It sometimes spends a half-hour to obtain the result using Wang's scheme, but sometimes costs less than ten minutes if lucky, whereas Klima's version often costs approximately 2 minutes. Stevens' versions

are fancy that can choose the prefix or even use a single block to obtain the collision, but the trade-off is spending more time.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This report firstly presents how the MD5 message-digest algorithm works and provides the demo for the readers to test. Then, it mainly provides four methods for finding the MD5 collision. The initial one is provided by Wang et al. which is also the first version to fast search the collision pair. The second one is the improved version with sufficient conditions provided by Stevens. The third one is created by Klima who implements the tunnels using PoV to accelerate the searching speed and increase the collision probability. The fourth one is the chosen-prefix attack invented by Steven. In the same section, it also provides some abusive scenarios of those attacking application. Finally, it compares the performances of different methods and gives a guideline for users who want to reproduce the results by themselves.

5.2 Future Work

In this paper, we only focus on the fast collision algorithm, so there are many perspectives that can be explored in the near future.

- Since we only reproduce the fast collision algorithm of MD5 and do not do much work for other advanced attacks, we can extend the research to a much wider range like the chosen-prefix one and their applications.
- In addition, the idea of MD5 collision can be mimicked to SHA-1 as well. Wang also proposed this idea in 2005.

These can be suggestions for future work.

5.3 Review and Reflection

The total journey of this project is about 8 weeks, during which we have reviewed much previous knowledge and learned lots of fresh ones as well. In this section, we will reflect on our project in general.

Firstly, to prepare for the survey, we reviewed the earlier knowledge about the hash function and Merkle–Damgard family from IN4191. After that, we started reading M. Stevens' PhD thesis, having a quite high quality, where almost all the things could be found, including but not limited to the developing history, preliminary, algorithms and implementations. The comprehensive materials contributed to our project a lot. In order to implement the earliest MD5 collision scheme, we browsed X. Wang's project website, but unfortunately, the website had expired and only published papers could be found. After reading the paper, we acquired the knowledge of bit conditions and got an intuitive understanding of how bit conditions could cancel the differences in the messages using two consecutive blocks, but the multi-message modification technique was still obscure to us. Luckily, we found a website provided by S. Thomsen where he reproduced the experiment using Wang's implementation. Using his idea, we made our first MD5 collider. Since the code was generated from the early version, there is a lot of room to be improved. We adopted SonarQube to assist

us to find any places that we could make an improvement. We adjusted the code structure to make it more efficient and readable. Later on, through M. Stevens' paper, we knew that the bit conditions of Wang's method were insufficient, thus, it might take a quite long time to process as it was unstable. Therefore, later on, we updated the bit conditions according to Stevens' version. Compared with Wang's version, it would not search more than 40 minutes to find a collision anymore. In precise, the variation of the Stevens' version was smaller and it improved the worst-case execution time. Stevens mentioned tunnels in his paper but not detailed their implementation. As consequence, we needed to seek someone else who had implemented it. Then, we found that V. Klima achieved this function in his implementation. Through reading his paper and code, we knew that the implementation of the tunnel was based on PoV. It left more freedom for the modification. Due to this flexibility, the constraints were more easily to be satisfied. In other words, it increased the probability of finding a collision pair, so it needed to be added one more forward MD5 calculation at the end of each block to verify whether the finding results were correct - it increased the probability but not guaranteed it. After we reproduced the collision attacks, we also tried to find if they would be abused in some scenarios. Then, we noticed many people commented that the identical-prefix attack could not adversely impact the MD5 significantly. In this situation, we continued to study more about the chosen prefix attack. This method was a more advanced one that Marc used to forgery the CA, but the trade-off was time. It would consume quite an amount of time to complete one collision. Since it was not a fast collision method, we listed it as the future plan for further studying, so did SHA-1. This is the end of our project, but we hope this is not the end of the journey to pursue security knowledge.

Bibliography

[1] Stevens, M., Lenstra, A. K., & De Weger, B. (2012). *Chosen-prefix collisions for MD5 and applications*. International Journal of Applied Cryptography, 2(4), 322-359.

[2] Stevens, M. (2012). *Attacks on hash functions and applications*. Mathematical Institute, Faculty of Science, Leiden University, 3.

[3] Stevens, M. (2006). *Fast Collision Attack on MD5*. IACR Cryptol. ePrint Arch., 2006, 104.

[4] Wang, X., & Yu, H. (2005, May). *How to break MD5 and other hash functions*. In Annual international conference on the theory and applications of cryptographic techniques (pp. 19-35). Springer, Berlin, Heidelberg.

[5] Klima, V. (2006). *Tunnels in Hash Functions: MD5 Collisions Within a Minute*. IACR Cryptol. ePrint Arch., 2006, 105.

[6] Klima, V. (2005). *Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications*. IACR Cryptol. ePrint Arch., 2005, 102.

[7] Hawkes, P., Paddon, M., & Rose, G. G. (2004). *Musings on the Wang et al. MD5 Collision*. IACR Cryptol. ePrint Arch., 2004, 264.

[8] Smart, N. P., & Smart, N. P. (2016). *Cryptography made simple*. Springer.

[9] Mikle, O. (2004). *Practical Attacks on Digital Signatures Using MD5 Message Digest*. IACR Cryptol. ePrint Arch., 2004, 356.

[10] Kaminsky, D. (2005). *MD5 to be considered harmful someday*. In Aggressive Network Self-Defense (pp. 323-337). Syngress.

[11] Kashyap, N. D. (2006). *A meaningful MD5 hash collision attack*.

Appendix A

Program Instructions

This section will guide the readers to reproduce the results like ours. The project website is github.com/Timo9Madrid7/MD5-Collision. You can download the zip file manually or use git to clone the project onto your local machine. In TimoMD5 branch, there is a file named "code" in which the README.md provides the structure of the "code" file.

A.1 Reproducing MD5 Algorithm Process

We provide two code for testing MD5 message-digest algorithm. The one is the Python version and another one is Java version. In terms of the Python version, it requires Jupyter Notebook in order to run, but the code can also be copied from the file and paste to other Python IDE for testing. For the Java version, it requires JDK in order to run. You can find MD5-Generator.jar in "jar" file. Simply using the command *java -jar MD5-Generator.jar* can run and test it. The corresponding source code can be viewed in MD5Encryption.java file. The following figures (Fig. A.1) show the testing result, using "TU Delft" as an example.

```
# Example
message = "TU Delft"
encode_message = encode_md5(message)
hash_message, _ = hash_md5(encode_message)
print(hash_message)

467764e7f10c77695db73ab3d414c65d
```

2 MD5 by HashLib

```
import hashlib
md5_object = hashlib.md5()
md5_object.update(b"TU Delft")
md5_result = md5_object.hexdigest()
print(md5_result)

467764e7f10c77695db73ab3d414c65d
```

(a) MD5 Algorithm - Python Example

```
D:\programmer\github\MD5-Collision\code\java\jar>java -jar MD5-Generator.jar
Please input a <string> m:
m = TU Delft
MD5(m) = 467764e7f10c77695db73ab3d414c65d
```

(b) MD5 Algorithm - Java Example

Figure A.1: MD5 Algorithm

A.2 Reproducing MD5 Collision Process (Wang Version)

We provide two code for testing Wang's collision. The one is CPP version which can be opened through Visual Studio and another one is Java version. We highly recommend readers to use the latter one, as it is much simpler. The source code of the jar application can be found in WangCol.java. The following figure (Fig. A.2(a)) shows the example of running the jar application.

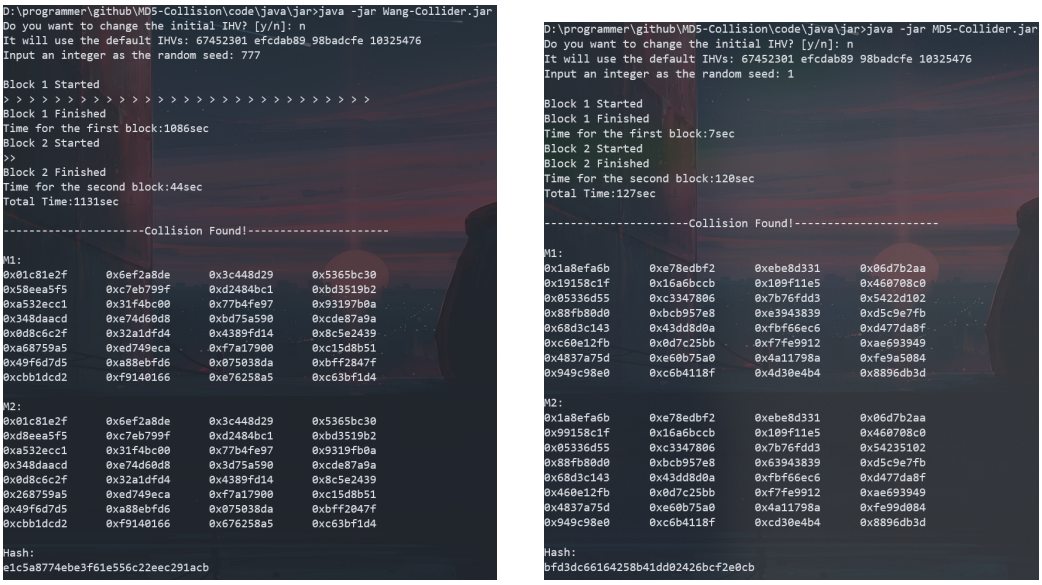
A.3 Reproducing MD5 Collision Process (Stevens)

For Stevens collision searching algorithm, we reproduced this algorithm based on Python , to test this algorithm, after downloading this file, you can press run to start ruining, in the end after all process finished, two blocks of messages and their corresponding Hash values shall be shown on the screen.

A.4 Reproducing MD5 Collision Process (Klima)

We provide two code for testing Klima's collision. The one is CPP version and another one is Java version. We still highly recommend readers to use the

latter one, as it is much simpler. The source code of the jar application can be found in MD5Collision.java. The following figure (Fig. A.2(b)) shows the example of running the jar application.



(a) Wang Collision Attack (b) Klima Collision Attack

Figure A.2: Attacking Algorithms

A.5 Basic Instructions For SonarQube

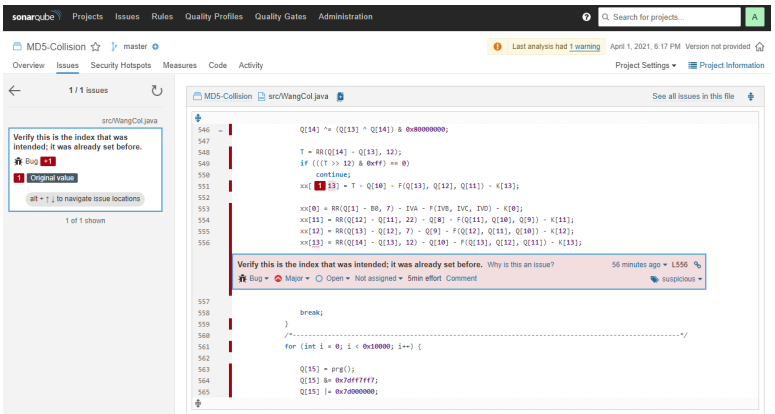


Figure A.3: Sonar Analysis

In the report, we mentioned one useful tool named SonarQube which has the ability to ”catch tricky bugs to prevent undefined behaviour from impacting end-users”. The free-trail version can be download from sonarqube.org. After down-

loading, users can enter into the file "*./bin/windows-x86-64*" and click "*Start-Sonar.bat*" to run it. It will automatically open a new terminal and run. Users, then, move to the project file and use "*javac*" to compile the ".java" files first to get ".class" files. After that, users can open "local host:9000" on the website to create a new project following the guideline. When using its default command, "*-D"sonar.java.binaries=D:#path contains .class files#""* should be added behind it. The above figure Fig. A.3 shows an example of the output result.

Appendix B

Bit Conditions

step	bit conditions from Q[31] to Q[0]	#Restraints
1	. . . . .	0
2	. . . . .	0
3	. . . . . 0 . . . . . 0 . . . . . 0	3
4	1 . . . . . 0 ∧ ∧ ∧ 1 ∧ ∧ ∧ ∧ ∧ ∧ ∧ 1 ∧ ∧ ∧ ∧ 0 1 1	21
5	1 0 0 0 1 0 0 . 0 1 . . 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 . 1 . 1	27
6	0 0 0 0 0 0 1 ∧ 0 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 1 0 0 ∧ 0 ∧ 1	32
7	0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0 0 0 1 0 0 0 0 0	32
8	0 0 0 0 0 0 0 1 1 . . 1 0 0 0 1 0 . 0 . 0 1 0 1 0 1 0 0 0 0 0 0	28
9	1 1 1 1 1 0 1 1 . . . 1 0 0 0 0 0 . 1 ∧ 1 1 1 1 0 0 1 1 1 1 0 1	28
10	0 1 1 1 . . . . 0 . . 1 1 1 1 1 1 1 0 1 . . . 0 0 1 . . . . 0 0	19
11	0 0 1 0 0 0 0 0 0 1 . . . 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 1 0	29
12	0 0 0 . . . . 0 0 . . . . 1 0 0 0 0 0 0 1 . . . 1 0	15
13	0 1 . . . . . 0 1 . . . . 1 1 1 1 1 1 1 . . . . 0 0 . . . 1 . . .	14
14	0 . 0 . . . . 0 0 . . . . 1 0 1 1 1 1 1 . . . . 1 1 . . . 1 . . .	14
15	0 . 1 . . . . 0 1 . . . . 0 1 . . . . 0 . . .	7
16	0 ! 1 . . . . . !	4
17	0 ! . . . . . 0 . ∧ . . . . . ∧ . . .	5
18	0 . ∧ 1	3
19	0 . . . . . 0	2
20	0 . . . . . !	2
21	0 . . . . . ∧	2
22	0	1
23	0	1
24	1	1
25-45	. . . . .	0
46	I	0
47	J	0
48	I	1
49	J	1
50	K	1
51	J	1
52	K	1
53	J	1
54	K	1
55	J	1
56	K	1
57	J	1
58	K	1
59	J	1
60	I	1
61	J	1
62	I	1
63	J	1
64	. . . . .	0

Table B.1: Block 1 Sufficient Conditions

step	bit conditions from Q[31] to Q[0]	#Restraints
-2	. . . . . . 0	1
-1	∧ 0 1	3
0	∧ . . . . 0 0 . . . . 0	4
1	! . . . 0 1 0 . . . 1 . . . . 0 . . . . 0 . . . . 1 0	9
2	∧ ∧ ∧ ∧ 1 1 0 . . . 0 ∧ ∧ ∧ ∧ 0 1 . . ∧ 1 . . . ∧ 1 0 . . 0 0 .	21
3	∧ 0 1 1 1 1 1 . . . 0 1 1 1 1 1 0 . . 0 1 . . 1 0 1 1 ∧ ∧ 1 1 .	24
4	∧ 0 1 1 1 0 1 . . . 0 0 0 1 0 0 . . . 0 0 ∧ ∧ 0 0 0 0 1 0 0 0 ∧	26
5	! 1 0 0 1 0 . . . . 1 0 1 1 1 1 . . . 0 1 1 1 0 0 1 0 1 0 0 0 0	25
6	∧ . . 0 0 1 0 . 1 . 1 0 . . 1 0 1 1 . 0 1 1 0 0 0 1 0 1 0 1 1 0	25
7	! . . 1 0 1 1 ∧ 1 . 0 0 . . 0 1 1 0 . 1 1 1 1 0 0 0 1	21
8	∧ . . 0 0 1 0 0 0 . 1 1 . . 1 0 1 . . . . . 1 1 1 1 1 . . . ∧ 0	19
9	∧ . . 1 1 1 0 0 0 . . . . . 0 1 0 . . ∧ . . 0 1 1 1 0 . . . 0 1	18
10	∧ . . . . 1 1 1 1 . . . 1 0 1 1 1 1 0 0 1 . 1 1 1 1 . . . . 0 0	20
11	∧ . . 0 0 . . . . 1 1 0 1 1 1 0 0 0 . 1 1 1 1 0 . . . 1 1	19
12	∧ ∧ ∧ 0 0 ∧ ∧ ∧ . . . . 1 0 0 0 0 0 0 1 . . . . . 1	17
13	! 0 1 1 1 1 1 1 0 . . . 1 1 1 1 1 1 1 . . . . . 0 . . . 1 . . .	18
14	∧ 1 0 0 0 0 0 0 1 . . . 1 0 1 1 1 1 1 . . . . . 1 . . . 1 . . .	18
15	0 1 1 1 1 1 0 1 . . . . 0 0 . . . . 0 . . .	11
16	0 . 1 0 . . . . !	4
17	0 ! . . . . 0 . ∧ . . . . ∧ . . .	5
18	0 . ∧ 1	3
19	0 . . . . 0	2
20	0 . . . . !	2
21	0 . . . . ∧	2
22	0	1
23	0	1
24	1	1
25-45	. . . .	0
46	I	0
47	J	0
48	I	1
49	J	1
50	K	1
51	J	1
52	K	1
53	J	1
54	K	1
55	J	1
56	K	1
57	J	1
58	K	1
59	J	1
60	I	1
61	J	1
62	I	1
63	J	1
64	. . . .	0

Table B.2: Block 2 Sufficient Conditions

$$f_t[b] = \begin{cases} . & \text{if there is no restriction} \\ 0 \text{ or } 1 & \text{if } Q_t[b] \text{ must be the value 0 or 1} \\ \wedge & \text{if } Q_t[b] \text{ must be equal to } Q_{t-1}[b] \\ ! & \text{if } Q_t[b] \text{ must not be equal to } Qt_1[b] \end{cases} \tag{B.1}$$

$I, J, K \in 0, 1$ and $K = \bar{I}$

The tables are retrieved from paper [3] because they are not presented in his PhD thesis [2]. For your convenience, we present them here. For more details, readers can read the paper [2] where the old tables which conditions are insufficient and the tables for chosen-prefix attack are exhaustively presented.