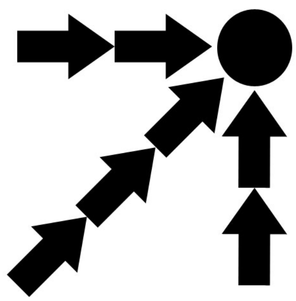
Graz University of Technology
IAIK
Institute for Applied Information Processing and Communications
Inffeldgasse 16a
A–8010 GRAZ

IAIK
TU Graz



A Tool for Message Modification: Application to MD5

Martin Holzer

A Tool for Message Modification: Application to MD5

Master Thesis

at

Graz University of Technology

submitted by

Martin Holzer

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology
A-8010 Graz, Austria

May 22, 2013

Assessor: Univ.-Prof. Dr. Vincent Rijmen
Advisor: Dipl.-Ing. Dr. techn. Martin Schläffer

Ein Tool zur Message Modification: Anwendung auf MD5

Masterarbeit

an der

Technischen Universität Graz

vorgelegt von

Martin Holzer

Institut für Angewandte Informationsverarbeitung und Kommunikation (IAIK),
Technische Universität Graz
A-8010 Graz, Austria

22. Mai 2013

Diese Arbeit ist in englischer Sprache verfasst.

Gutachter: Univ.-Prof. Dr. Vincent Rijmen
Betreuer: Dipl.-Ing. Dr. techn. Martin Schläffer

Abstract

Many of today's cryptographic protocols rely on strong hash functions. MD5 is still a popular hash algorithm. It was subject to meaningful attacks leading to the conclusion that the algorithm should not be used at all anymore. In 2005, Wang *et al.* introduced a completely new idea using differential cryptanalysis to find collisions in MD5. Since then, many improvements and new techniques have been introduced. The latest single-block collision attacks by Xie *et al.* and Stevens showed further considerable weaknesses in this hash function.

In this thesis, the most popular collision attacks will be analyzed using a non-linear toolbox. This framework enables us to check, propagate and find differential paths. Focus was laid on how this tool could be used for the specific attacks on MD5. The message modification and tunnel techniques were considerably slow for the bit-sliced approach of the tool. Hence, the time-critical parts of attacks were implemented using data structures based on whole words. Furthermore, a new differential characteristic was constructed for the partial path published by Xie *et al.* and a conforming message pair was found for up to 24 steps. We estimate the complexity of finding a collision with $2^{62.04}$ MD5 compression function evaluations.

Keywords: hash function, MD5, differential cryptanalysis, non-linear toolbox, message modification, single-block collision

Kurzfassung

Viele der heutigen kryptographischen Protokolle sind abhängig von stabilen Hash-Funktionen. Der Algorithmus MD5 ist ein sehr bekannter Vertreter. Es gab schon viele bedeutende Attacken darauf, welche offenbarten, dass MD5 nicht mehr verwendet werden sollte. Wang *et al.* veröffentlichte im Jahr 2005 eine neue innovative Attacke unter der Verwendung von differentieller Kryptanalyse. Darauf hin wurde der Angriff von vielen Kryptographen weltweit untersucht und verbessert. Die aktuellsten Attacken von Xie *et al.* und Stevens benötigen für eine Kollision nur mehr einen einzigen Nachrichtenblock. Das zeigte, dass noch immer weitere Schwächen in MD5 gefunden werden können.

In dieser Arbeit werden die elementarsten Kollisionsattacken analysiert. Dabei stellt sich die non-linear Toolbox als sehr wichtiges Hilfsmittel dar. Dieses Framework ermöglicht es einem, differentielle Pfade zu überprüfen und zu finden. Im Kontext dieser Arbeit war wichtig, herauszufinden, inwieweit dieses Programm für die spezifischen Angriffe auf MD5 benutzt werden kann. Die Techniken *Message Modification* und der Einsatz von sogenannten *Tunneln* sind auf diesem Framework langsam, da dessen interne Datenstruktur Bit-orientiert arbeitet. Um eine Beschleunigung zu erzielen, wurden die wichtigsten Kollisionsattacken mit optimierten Datenstrukturen implementiert. Interessant ist dieser Zugang vor allem bei der Kollisionsattacke auf einzelne Nachrichtenblöcke. Schließlich wurde auf Basis der partiellen differentiellen Charakteristik von Xie *et al.* ein neuer, vollständiger Pfad erzeugt und eine Lösung für 24 Schritte gefunden. Die Gesamtkomplexität wird auf $2^{62.04}$ Aufrufe der Kompressionsfunktion geschätzt.

Keywords: Hash-Funktion, MD5, Differentielle Kryptanalyse, Non-Linear Toolbox, Message Modification, Kollision mit einzelnen Nachrichtenblöcken.

Danksagung

Ich möchte mich hier bei allen Personen bedanken, die im Schaffungsprozess dieser Arbeit involviert waren.
Als erstes spreche ich meinen Dank Herrn Prof. Vincent Rijmen aus, der meine Arbeit begutachtet hat. Durch seine Vorlesungen bin ich erst auf den Geschmack gekommen, mich mit IT-Security näher auseinander zu setzen.
Ebenso gilt mein Dank meinem Betreuer, Herrn Dr. M. Schläffer, der mir mit viel Geduld und stets mit Antworten zu meinen Fragen unermüdlich zur Seite stand. Weiters möchte ich mich bei Herrn Dr. F. Mendel bedanken, der mich in der Anfangsphase meiner Arbeit mitbetreut hat.
Ich hatte viel Kontakt zu Studenten, die mir bei Prüfungen und Projekten geholfen haben. Hervorheben möchte ich besonders Christoph, Jochen und Philipp. Ich erinnere mich gerne zurück an die Zeiten im Studienzentrum, wo wir gemeinsam viel Zeit verbracht haben. Außerdem habe mit ihnen neue Freundschaften fürs Leben gefunden.
Mein abschließender Dank gebührt meinen Eltern, die mich während des gesamten Studiums stets unterstützt haben. Vor allem aber danke ich vielmals meiner Mama für die Mitkorrektur dieser Arbeit.

Martin Holzer, Mai 2013

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Contents

Chapter 1

Introduction

Today's information security relies on three important aspects: confidentiality, integrity and authenticity. The first one ensures that only allowed parties are able to access their designated message. Integrity provides a mechanism, so that involuntary modifications of a message are detectable. Authenticity guarantees sure that a message is genuine. The introduction of asymmetric encryption schemes was a big step toward providing these properties. One-way functions deliver the ability of evaluating a computation relatively easily in one direction. The reverse calculation should be very hard. In the case of asymmetric encryption, a private key is the only way of finding the inverse.

Hash functions are another type of one-way functions. In simple terms, they create a digital fingerprint of a message. Computation of this fingerprint should be easy, however, finding a message to a fingerprint should be a problem very hard to solve. Hash functions play a key role in many different cryptographic applications. If you want to check your bank account on the web, hash functions along with asymmetric cryptography provide a secure connection. Another example would be creating a signature of a digital document which is accepted similarly to a normal signature on paper.

The goal of an adversary would be the creation of another document sharing its signature with the original one. If this succeeds, no one would be able to distinguish between the genuine and the illegitimate document. Stevens [SLW07] was able to create a rogue SSL certificate which could be used to falsify the identity of the attacked web server. Hence, cryptanalysis of hash functions is highly desired.

MD5 is such a hash function, belonging to a large family of hash functions called the *MD-family*. All of them share a common method of operation. They split the message in parts of a specified length and process each block together with the result of the last block. Its successor, *SHA-1*, in particular, is used in many applications nowadays.

1.1 Related Work

In 2005, Wang *et al.* [WY05] made a breakthrough in attacking the MD-family of hash functions. For MD5, a colliding pair of messages sharing the same fingerprint could be generated in a reasonable time. This research had a great impact on cryptologists all over the world. Since then, many of them started their own analysis based on Wang's work. MD5 was largely used at that time. The weaknesses of this algorithm became apparent and this is why it was highly recommended to use stronger hash functions like *SHA-2*. Wang used a completely new approach in her attack. About two years later, Stevens [SLW07] was able to create collision with messages including a common prefix. For such an attack, a cluster of computers was necessary. They were used for creating a forged SSL certificate. One could use this technique in creating a rogue web page which appears to be genuine. Furthermore, he was able to improve Wang's approach and could reduce the runtime of creating random collisions to a few minutes [Ste06] on a normal computer. These attacks by Wang and Stevens used two message blocks for *MD5* to create a colliding hash value.

In 2010, Xie *et al.* [XF10] were able to create another collision. This time, they were able to use a message pair each having the length of exactly one block for MD5. Details on the algorithm were not published. Xie *et al.* called a competition on who would publish another one-block collision with a reward of $ 10.000. Stevens [Ste12a] was successful and made the details of his attempt public. Very recently, Xie *et al.* [XLF13] published details of their original attack.

The *Institute for Applied Information Processing and Communications* has created a toolbox, later referred to as *nltool*, which can be used to analyze and perform differential cryptanalysis. It can also be applied to MD5 including path and message search algorithms.

1.2 Contributions

The goal of this work is to apply this tool on two- and single-block-collisions. First of all, in-depth analysis of these attacks is made. The new techniques already introduced in these attacks, *i.e.* tunnels, will be subject to in-depth analysis. These methods are recreated for a better understanding of their behaviour. Measurements are made on how the *nltool* performs with its already implemented strategies on these attacks. After that, optimizations are programmed to further speed up these collision attacks on *MD5*. These optimizations include faster data structures for these specialized attacks. The attack for two-block collisions based on Stevens [Ste06] can be executed in seconds which makes the optimizations practical to measure.

The remaining work focuses on single-block-collisions using the works of Xie *et al.* [XF10] [XLF13] and Stevens [Ste12a]. Again, a combination of the *nltool* and new optimizations is made. The target is to determine which parts of these attacks can be done automatically by the toolbox. For verifying Stevens' algorithm description, the attack will be implemented and complexities are determined. Finally, we will use the best partial path given by Xie *et al.* [XLF13] and derive a full differential characteristic. This will be the base for designing our own single-block collision attack.

1.3 Outline

The work is structured as follows:
In Chapter 2, an overview over hash functions and their properties is given. Moreover, certain security constraints are clarified.
Details about MD5 and its common aspects with the *MD-family* are explained in Chapter 3. An overview of all major attacks including their complexities is given.
The basics behind the the differential attacks are explained in Chapter 4. This is an important base for the next chapters. Moreover, the *nltool* is described including its fundamental features and algorithms.
Chapter 5 deals with all well-known collisions based on two message blocks. The approaches by Wang *et al.* and Stevens are discussed extensively. We will provide implementation details on how to use the *nltool* for these kinds of attacks.
Finally, one-block collisions are evaluated in Chapter 6. The implementation of Stevens' approach is given and contributed results are presented. Furthermore, a partial path by Xie *et al.* will be completed and an attack proposal is made.
A conclusion is made in Chapter 7.

Chapter 2

Cryptographic Hash Functions

An important type of functions in cryptography are so called *one-way functions*. They map a huge domain to a fixed range of n bits. For example, the latter could be a 128 bit value. The calculations of those functions should be simple. Yet, the computation of its inverse value should be very hard. Using one-way functions, many other cryptographic primitives can be derived: pseudo-random generators [ILL89], message authentication codes and digital signature schemes [Rom90]. Concerning cryptography, so-called *hash functions* [MVO96] are a class of one-way functions.

Definition 2.0.1 (Hash function)**.** Given: Hash function h with output size n, input domain $D = \{0,1\}^m$ and range $R = \{0,1\}^n$: $h : D \rightarrow R$.

We can further distinguish between *modification detection codes* (MDC) and *message authentication codes* (MAC) [MVO96]. MDCs make use of *unkeyed* hash functions. They use only a single input parameter: the message to digest. On the other hand, MACs facilitate *keyed* hash functions. Those are defined by two distinct inputs, a message and a secret key.

Unkeyed hash functions are very important because of their use in digital signature schemes. Before using hash functions, the idea was to sign the whole document (with arbitrary size). The operation of signing is very slow. Moreover, the signature has the same size as the document itself. Rabin [Rab79] introduced the approach of signing the hash of a document and not the document itself. Using hashes for the signature scheme, computing power and signature size can be constant and independent of the size of the document. Attacking such schemes can be done in two ways: either by breaking the signature algorithm itself or by finding a different document with the same hash. If the latter succeeds, an adversary would be able to create a correctly signed document and the authenticity cannot be denied. This simple, yet powerful example shows the importance of reliable hash functions. However, the term *reliable* can be described more properly. Accordingly, more properties for hash functions can be defined [MVO96]:

Preimage resistance: For a given hash value, it is computationally infeasible to find any message with the same hash value.

Second-preimage resistance: For a given message and a hash value, it is computationally infeasible to find another message with the same value. The attack on digital signature schemes is a good example of finding a second message (or document) in order to forge a signature.

Collision resistance: It is computationally infeasible to find any two messages with the same hash value. Due to the nature of the birthday paradox (see theorem 2.2.1), the complexity of this property is significantly lower compared to preimage and second-preimage resistances. Thus, this class of resistance is subject to many attacks on hash functions. This thesis will concentrate on finding collisions.

The term *computationally infeasible* is used intentionally without any further definition. This property is used as a reference for comparisons between *easy* and *hard* problems. Moreover, attacks that were infeasible ten years ago are now possible in reasonable time.

2.1 Other applications and properties

Unkeyed hash functions using one-way functions offer even more possibilities to take advantage of:

1. *Confirmation of knowledge*
   They can be used for proving ownership of specific data. For example, someone could publish a document. Before making it available to everyone, one could make the hash value public to demonstrate the existence of document. Later on, all people can generate the fingerprint of this document and compare it with the hash value.

2. *Key derivation*
   In systems where keys have to be changed on a regular basis, new key values are calculated as a hash value of a previously used key. It is important to protect the expired keys if the current key happens to be revealed.

For one-way hash functions, supplementary definitions can be made [MVO96]:

Definition 2.1.1 (non-correlation)**.** Input and output bits should not be correlated. In this manner, an avalanche property (also used in strong block ciphers) is essential whereby every input bit affects every output bit. This rules out hashes where preimage resistance fails to imply second-preimage resistance because the function ignores a subset of input bits.

Definition 2.1.2 (near-collision resistance)**.** It should be computationally infeasible to find any two inputs whose hash values only differ in a small number of bits.

Definition 2.1.3 (partial-preimage resistance or local one-wayness)**.** The recovery of any substring should be as hard as recovering the complete input. On top of that, even for a known part of the input, finding the rest should be hard. For example, if m input bits are unknown, an average of 2^{m-1} hash operations should be necessary to recover the missing bits.

2.2 Generic Attacks

Based on these definitions, generic attacks are possible without the knowledge of details for a specific hash function. For a fixed message M of an n bit hash function h, the most naive method of finding any other colliding M' is to create random values for M' and checking if $h(M) = h(M')$. The memory complexity is constant, the probability of finding a collision is 2^{-n}.

Desired Complexities. Comparing the different attack types is interesting in terms of complexity. It is always given as the amount of hash function calls depending on the bit size n of the hash value.

Table 2.1: Ideal strengths of properties of hash functions

Property	Ideal Strength
Preimage resistance	2^n
Second-preimage resistance	2^n
Collision resistance	$2^{\frac{n}{2}}$

Considering preimage and second-preimage attacks, an adversary could precompute an extensive list of pairs $(x, h(x))$. If the list is long enough, the probability can be lowered for such attacks. For example, a 64-bit hash function is subject of this attack. The opponent could create a list containing 2^{32} value pairs. Hence, time and memory complexity are $O(2^{32})$. The probability of finding a preimage using this list is now reduced to 2^{-32}. Depending on the algorithm, this probability is low enough to be feasible on modern computers. Storing 2^{32} pairs would require 64 GB of memory.

If the complexity of an attack is below the ideal strength, the hash function is considered *broken*. As mentioned earlier, one can see the lower complexity for collision resistances compared to preimage and second-preimage attacks. This is the reason why many attacks focus on this weakness. Based on the ideal strength, general security observations can be made. If a hash function returned a hash value of $n = 64$ bits, its collision resistance would be at most 2^{32}. Depending on implementation details, this rather low complexity is not a hard problem for today's computers. In conclusion, such a hash function would be considered unsuitable for modern requirements. This is also the reason why new hash functions simply have a higher output size.

Generally speaking, requirements can be made on the bit size. Designing a hash function below these bounds is not practical because the simplest attacks (*i.e.* collision resistance) can be executed in feasible time. For a collision resistant hash function, a minimum of 160 bits is recommended. The complexity of a birthday attack is at least 2^{80}.

2.2.1 Collision attack types

An iterated hash function needs a starting value (the IV). Depending on the freedom of determining this value, three different attacks can be distinguished:

Definition 2.2.1 (Collision)**.** A normal collision uses the fixed IV which is used for the first message block to be hashed. This means that the input messages differ but not the chaining value.

Definition 2.2.2 (Semi-free-start collision)**.** In this case, a random chaining value is taken instead of an IV such that a collision is created.

Definition 2.2.3 (Free-start collision)**.** The two colliding hash function calls get two different messages *and* two different chaining values (compared to the free-start collision, where the chaining value is shared by both hash function calls).

The freedom for free-start collisions is much higher. Hence, first attempts of attacking a hash function are often based on chosen IVs. The attacks in this thesis mostly focus on normal collisions, however, different IVs are used sometimes for complexity comparisons.

2.2.2 Birthday attack

This attack (also often referred to as *square-root* attack) provides the probability of finding two random colliding input values. The birthday paradox arises from the classical occupancy distribution.

Theorem 2.2.1 (Birthday Paradox [MVO96])**.** *When drawing elements randomly, with replacement, from a set of N elements, with high probability a repeated element will be encountered after $O(\sqrt{N})$ selections.*

A naive attack could be created by saving hash values and their corresponding inputs in a list. See algorithm 1 for further details.

Yuval's [Yuv79] birthday attack generates $t = 2^{\frac{n}{2}}$ messages based on a legitimate message x. and stores them with their hash value. The time-complexity is therefore $O(t)$. Based on a fraudulent message x', messages are generated and their hash-result is looked up in the previously generated list, until a message pair is found. The draw-back is a high memory requirement.

This concludes that the birthday attack works for any hash function with a bit size n and finds results after an average of $2^{\frac{n}{2}}$ calculations. This defines the general upper bound for an attack for the collision resistance of a hash function.

Algorithm 1 Birthday attack on hash functions

INPUT: Hash function h
OUTPUT: Message pair (x, x') where $h(x) = h(x')$

```
loop
    Generate random x' and calculate h(x')
    if h(x') is in list then
        return Corresponding x of list entry and x'
    else
        Save pair (x', h(x')) in list
```

2.3 Iterated hash functions

Because of the fact that the message size can be chosen arbitrarily, some kind of iteration is necessary. The most important principle of this iteration is the one by Merkle [Mer89] and Damgård [Dam89]. Like encryption, the message is split into blocks of equal size. Each block is then processed by a *compression function*.

Definition 2.3.1 (Compression function [Dau05])**.** A function f that compresses two inputs into a single output:

$$f : \{0,1\}^m \times \{0,1\}^l \rightarrow \{0,1\}^m \qquad l > m \geq 1$$

Figure 2.1 shows the integration of the compression function in the iterated hash principle. The message blocks $x_1, \ldots, x_n$ are processed by the compression function. The intermediate hash values $H_0, \ldots, H_n$ provide that hash values from previous blocks are part of the final hash result. Using the definition 2.3.1, we can see that the intermediate hash values have to smaller size m and the message blocks have size l. Function g is sometimes used providing a last processing step before the final hash value.

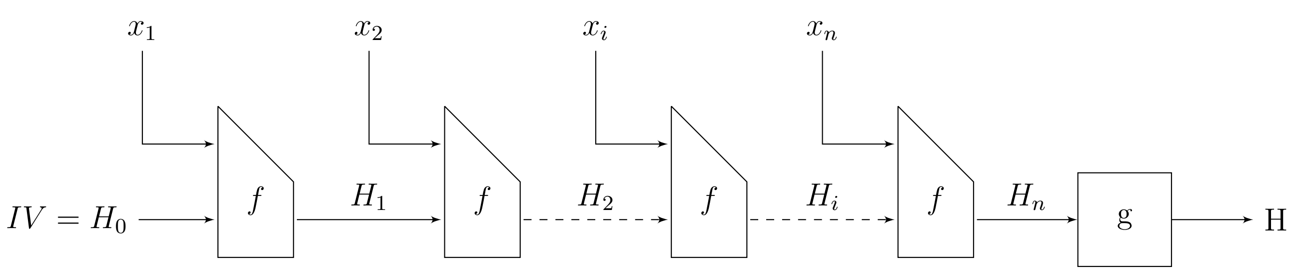


Figure 2.1: Iterative hashing principle

Due to the fact that the block size is fixed and the hash function has to be able to process messages of arbitrary size, some kind of padding is necessary to fill the last block completely. The most common way to accomplish this is by adding the length of the message as a binary value and additional bits (*i.e.* zero bits) to the end of the block. This process is called *MD-Strengthening*.

The collision resistance of the compression function has a huge impact on the strength of the complete hash algorithm. Theorem 2.3.1 states that if the compression function is collision resistant, the hash function is collision resistant as well. Hence, the compression function will be subject to in-depth analysis.

Theorem 2.3.1 (Construct collision free hash functions from fixed size collision free hash functions[Mer89])**.** *f is a fixed size collision free hash function mapping m to n bits. Then there exists a collision free hash function h mapping strings of varying length to strings of length n. $a||b$ is the concatenation of the bit string a and b. The bit blocks h_0, h_1 with bit length r are defined by: $h_1 = f(0^{r+1}||x_1)$ and $h_{i+1} = f(h_i||1||x_{i+1})$ ending with $h(x) = h_{\frac{n}{(m-r)+1}}$.*

Chapter 3

Hash Functions based on the MD-family

Dedicated hash functions are hash functions that are solely developed for the purpose of hashing. Popular examples include *MD4*, *MD5*, *SHA-1* and *RIPE-MD*. These are often used in various cryptographic standards. Hash functions of this family share common design patterns. This section will explain these principles. The step operation, which is a vital component in each of these hash functions, will be explained in detail. This chapter will conclude with a list of important attacks on the MD-family.

The hash functions of the MD-family are iterated hash functions. The message is split into blocks and each block is processed along with the intermediate hash result of the previous block. This compression function is called once for each iteration. Furthermore, the *Davies-Meyer scheme* (see Figure 3.1) is part of the compression function. This construction adds a feed-forward loop to strengthen non-invertability.

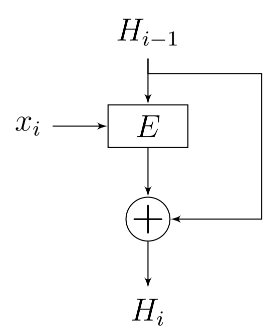


Figure 3.1: Davies-Meyer scheme [MVO96]

Figure 3.2 shows a high-level overview of the compression function. On the top left part, you can see the intermediate hash value of a previous block. If this would be the first message block, this input is the IV. The message block is split up into message words.

Step operations. The intermediate hash is broken down into several internal registers, *i.e.* $a, b, \ldots$. Using these values including a selected message word, a step operation is applied on these registers. The number of steps and the step operation is clearly defined. Most of the time, steps are logically grouped to *rounds*. As can be seen later on, the terms *rounds* and *steps per round* are used frequently.

Message expansion. The message block itself is too small such that every step operation uses its own independed message word w_s. Depending on the hash function, there are different approaches on how these message words are distributed. In other terms, the message has to expanded to be used in the step operations.

Finally, the internal registers are added to its initial state. They are concatenated leading to the final output of the compression function.

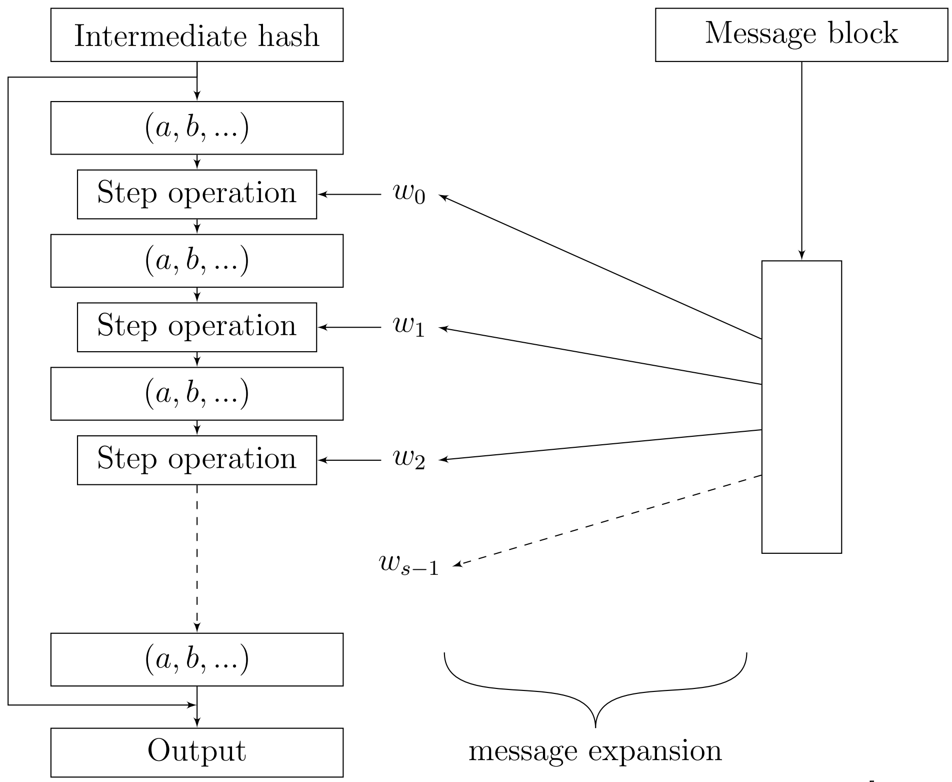


Figure 3.2: Compression function of a MD-family hash function [Dau05]

3.1 MD5

MD5 was introduced in 1992 by Rivest [Riv92] as a successor to MD4. MD5 processes 512-bit message blocks each round. The intermediate hash values consists of four 32-bit words. As we know, the compression function calls the step operation several times. In MD5, 64 step operations are executed. These are grouped into 4 rounds of 16 steps per round. Each step operation includes rotate and add operations. All those operations are based on 32-bit words. Moreover, a boolean function in each step operation is used and differs each round. It includes `AND`, `OR`, `XOR` and `NOT` bit operations. See algorithm 2 for a detailed view including the step operation. The latter is shown in a more clarified manner in Figure 3.3. The constants for each step and round can be found in Table 3.1.

Functions in MD5. Due to the behaviour of a boolean functions, they are often referred to as `IF`, `IF3`, `XOR` and `ONX`. See Table 3.2 for their operations.

Algorithm 2 The MD5 hash function [MVO96]

INPUT: Message M as bit string
OUTPUT: Hash result as an 128 bit string

Pad M such that its bit length is a multiple of 512, as follows. Append a 1-bit. Then add $(r-1)$ 0-bits for the smallest r of a bit length 64 less than the multiple of 512. Append the 64-bit representation of the length of M. n is now the number of 512-bit blocks of the formatted M. The padded input is now split into 32-bit words $m_0, \ldots, m_{16\cdot(n-1)}$.
Initialize $H = (h_0, h_1, h_2, h_3)$ with $IV = (IV_0, IV_1, IV_2, IV_3)$.
for $k \leftarrow 0$ to $(n-1)$ **do**
    Copy the 32-bit values from the current block to a temporary storage $(w_0, \ldots, w_{15})$: $w_j = m_{16\cdot k+j} \qquad 0 \leq j \leq 15$
    Initialize working values: $(a_{-4}, a_{-1}, a_{-2}, a_{-3}) \leftarrow (h_0, h_1, h_2, h_3)$
    for $i \leftarrow 0$ to 63 **do**
        Perform step operation:

$$a_i \leftarrow (a_{i-4} + F_i(a_{i-3}, a_{i-2}, a_{i-1}) + w_{z_i} + k_i) \lll s_i + a_{i-1}$$

    Update chaining values: $(h_0, h_1, h_2, h_3) \leftarrow (h_0 + a_{60}, h_1 + a_{63}, h_2 + a_{62}, h_3 + a_{61})$
return the hash value as a concatenation: $h_0||h_1||h_2||h_3$

3.2 Other hash functions

In the following section, similar hash functions (both older and newer) are briefly mentioned. A difference in their general runtime is also compared in Table 3.3.

3.2.1 MD4

MD4 [Riv91] is the predecessor of MD5. It uses three rounds instead of four. Hence only 48 step operations are performed. Moreover, the constants z_i and s_i are different to those in MD5. Because of the reduced number of rounds, only three round functions are used: f, g, h. k_i only has per round values and not per step. Finally, the compression function

Table 3.1: Constants used in MD5

Name	Values
IV	`(0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476)`
k_i	first 32 bits of the binary result of $abs(sin(i+1))$
z_i	$0 \leq i \leq 15$: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) $16 \leq i \leq 31$: (1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12) $32 \leq i \leq 47$: (5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2) $48 \leq i \leq 63$: (0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9)
s_i	$0 \leq i \leq 15$: (7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22) $16 \leq i \leq 31$: (5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20) $32 \leq i \leq 47$: (4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23) $48 \leq i \leq 63$: (6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21)

Table 3.2: Functions used in MD5

Values	Name
$0 \leq i \leq 15$: $f(x,y,z) = (x \wedge y) \vee (\neg x \wedge z)$	`IF`
$16 \leq i \leq 31$: $g(x,y,z) = (x \wedge z) \vee (y \wedge \neg z)$	`IF3`
$32 \leq i \leq 47$: $h(x,y,z) = x \oplus y \oplus z$	`XOR`
$48 \leq i \leq 63$: $k(x,y,z) = y \oplus (x \vee \neg z)$	`ONX`

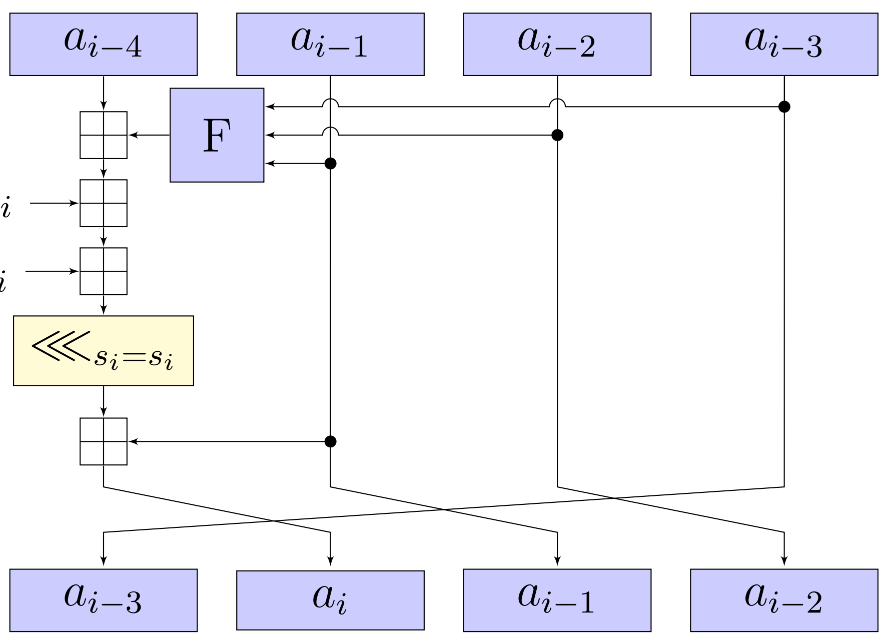


Figure 3.3: MD5 step function

lacks the add operation after the rotation. In conclusion, MD4 is less complex due to reduced rounds and a simpler step operation.

Table 3.3: Comparison of MD4-based hash functions

Name	Bitlength	Rounds $\times$ Steps per round	Relative performance
MD4	128	3×16	1.00
MD5	128	4×16	0.68
RIPEMD-128	128	4×16 two times parallel	0.39
SHA-1	160	4×20	0.28
RIPEMD-160	160	5×16 two times parallel	0.24

3.2.2 SHA-1

Due to security flaws in MD5, SHA-1 [EJ] was introduced. Instead of four intermediate hash values, five are used. Thus, the hash value is 32-bit longer resulting in a total bit size of 160 bits. The number of rounds stays the same, however, 20 steps per round are now processed. Therefore, the compression function involves 80 step operations. Each 512-bit message block is expanded to 80 message words. Each of the last 64 message words is the XOR of four words from earlier steps in this expanded block. Accordingly, the word index z_i is obsolete because each step uses its own message word w_i. Other constants used in the compression function no longer contain zero values. SHA-1 is more sophisticated than MD5 and crytographically stronger.

3.2.3 RIPEMD-160

RIPEMD-160 is also based on MD4 and adds ideas from MD4, MD5 and RIPEMD. The compression function maps 21 input-words to 5 output-words. One main principle which is also shared with RIPEMD is the parallelization of the input block. There is a so-called *left* and *right line*. Instead of three, five round functions are present. The amount of rounds and steps per round is improved.

3.3 Recent collision attacks on the MD-family

In the following subsection, our focus is laid on *collision attacks*. *Preimage attacks* were also successfully done on adapted MD5 [SA08]. These types of attacks are not considered here.

Already after introducing MD5, theoretical weaknesses were found. Den Boer and Bosselaers [dBB94] found out that the changes made to strengthen MD5 were not considered very well. They observed a relation of any four add constants $k_i, \ldots, k_{i+3}$. This enabled them to find colliding inputs for the compression function. The first two rounds of the compression function were easliy broken, for finding collisions in the fourth round, 2^{16} collisions in the first two rounds had to be created.

The idea of differential cryptanalysis was introduced by Biham and Shamir [BS91], where they applied XOR differences on the block cipher DES. This was highly applicable

due to the amount of XOR operations in the algorithm.
The first approach on MD5 using differentials was made by Berson in 1992 [Ber93]. However, he could not find a practical collision. Another type of differences, so-called modular differences were used in his work.

Wang *et al.* [WY05] uses Dobbertin's idea [Ber93] to create a colliding message pair. The attack incorporated the construction of a collision using exactly two message blocks. Hence, two calls of the compression function f are necessary. This initiated many publications that have been trying to improve this approach. Klima [Ste06] started with a technique called *tunnels*. Stevens optimized it further. These types of attacks are analysed extensivly in Chapter 5.

Xie *et al.* [XF10] published a real collision using just one message block, *i.e.* the colliding message pairs have a size of 512 bytes. This means that only one call to the compression function f is necessary to achieve a collision. They started a competition on who could publish another single block collision. About 2 years later, Stevens [Ste12a] reported a working solution. Chapter 6 discusses these attacks in detail. In 2013, Xie *et al.* [XLF13] published more details on their first single-block collision in 2010 [XLF13]. The complexity was significantly lower than the one by Stevens. They also published the fastest practical two-block collision.

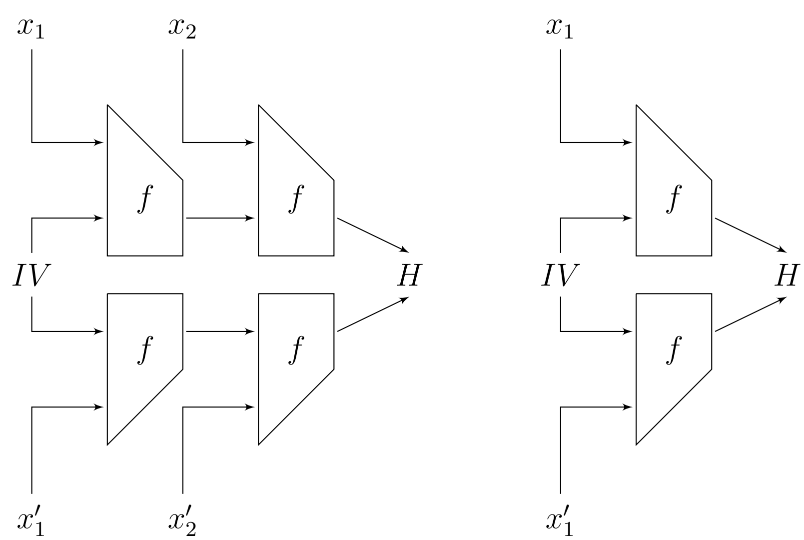


Figure 3.4: 2-block and single block collisions

Figure 3.4 shows the fundamental difference between two-block and single-block collisions. For the former, a colliding message pair (M, M') is split into $(x_1||x_2, x'_1||x'_2)$. The difference in the intermediate hash after processing the first message block is already smaller but not completely removed. It takes another iteration to achieve a true collision. For single-block collisions, the message pair (M, M') does not need any split operation and can be directly represented as (x_1, x'_1). The complexity for building attacks in that manner is much higher than for two-block collisions. More details on these attacks are given in Chapter 4. For an overview of the current achieved complexities of attacks, see Table 3.4.

The metric of the complexity is the number of compression functions calls necessary for the attack.

Table 3.4: Attacks and their complexities

Year	Name	Hash	Complexity
2005	Wang's two block collision [WY05]	MD5	2^{39}
		MD4	2^{23}
		RIPEMD	2^{30}
		SHA-0	2^{61}
2006	Klima's two block collision [Kli06]	MD5	31 seconds
2006	Stevens' two block collision [Ste06]	MD5	$2^{32.3}$
2008	Xie *et al.* two block collision [XFL08]	MD5	2^{36}
2010	Xie *et al.* one block collision [XF10] [XLF13]	MD5	2^{47}
2012	Stevens' one block collision [Ste12a]	MD5	$2^{49.81}$
2013	Xie *et al.* two block collision [XLF13]	MD5	2^{18}
2013	Xie *et al.* one block collision [XLF13]	MD5	2^{41}

MD5 is considered broken, as the complexity of many attacks on building collisions is considerably below the bound of the birthday probability (2^{64}). The last challenges are finding shorter message pairs leading to a collision.

Chapter 4

Differential Cryptanalysis of MD5

As of today, the most effective type of analysis on MD-based hash functions is *differential cryptanalysis* [Ste12b]. Compared to other approaches, two (instead of a single) messages and results of a hash function are observed simultaneously. The differences between these evaluations are analyzed. It can be examined how those differences propagate through the computation steps of a hash function. If those propagations can be controlled, it is possible to remove differences in order to lower (or even eliminate) the difference at all. Most of the time, it is used for *collision attacks*.

The term *differential cryptanalysis* has been first mentioned in the attacks against the DES (Data Encryption Standard) cipher in 1990. Biham and Shamir took advantage of `XOR` differences, since `XOR` (linear) functions are used in DES extensively [BS91]. The basic idea could be adapted to other types of differences. E.g. Berson [Ber93] is able to use modular differences on the MD5 hash function. One drawback of using a modular difference is that it cannot handle rotation operations and boolean functions very well. In 1995, Dobbertin attacked the compression function of MD5 [Dob96]. However, his attack could not be applied on MD5 itself. Later, he was successful at finding collisions on MD4 [Dob98]. Some years later, Biham *et al.* [BCJ⁺05] attacked SHA-0 and SHA-1 using both `XOR` and modular differences. This set off a new motivation for cryptanalysis on hashes based on the MD-family. Wang *et al.* [WY05] introduced a new kind of differences. Using *signed bit differences*, they were able to deliver a completely new attempt on creating collision on the MD5 hash function. Furthermore, Rechberger *et al.* [CR06] published an attack on the SHA-1 hash function using *generalized differences*.

In this section, the various types of differences and their properties will be explained. Moreover, the general approach of all differental attacks on MD5 is explained. At the end of this section, the *nltool*, which is a toolbox used for differential cryptanalysis, will be presented.

4.1 Differential Characteristics

A fundamental part in differential cryptanalysis is to define how differences at the inputs affect the differences at the outputs of a cryptographic function. Figure 4.1 shows how differences can propagate through different steps of some generic function. The input difference is ΔA and the output difference is ΔD. Between these differences, other step differences (in this example two) can be observed. It is often possible, that for the same input and output difference, various intermediate step differences are possible. Hence, ΔA and ΔD can have multiple ΔB_i and ΔC_i in this example. Depending on the depth of analysis, more step differences can be observed.

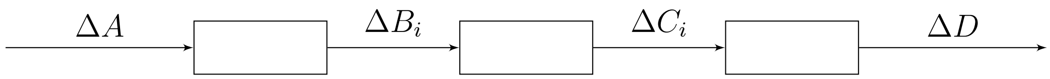


Figure 4.1: Characteristics and differentials

Differential and Differential Characteristic. The pair of input and output differences $(\Delta A, \Delta D)$ is called a *differential*. This top-level view does not give any insights into inner differences. On the contrary, a *differential characteristic* shows a sequence of differences, *i.e.* given in the form $(\Delta A, \Delta B_i, \Delta C_i, \Delta D)$. The term *differential characteristic* is often also referred to as *differential path*.

Probabilities. For all attacks, it is important to calculate the probability of a given differential or differential characteristic. The runtime of collisions depends greatly on the probability. First of all, the probability of a differential can be calculated by summing up all possible differential characteristics:

$$\begin{aligned} Pr(\Delta A \to \Delta D) = & \\ & Pr(\Delta A \to \Delta B_0 \to \Delta C_0 \to \Delta D)+ \\ & Pr(\Delta A \to \Delta B_1 \to \Delta C_1 \to \Delta D)+ \\ & \ldots = \sum_{B_i} \sum_{C_i} Pr(\Delta A, \Delta B_i, \Delta C_i, \Delta D) \end{aligned}$$

The probability of a differential is at least as high as a single differential characteristic:

$$Pr(\Delta A, \Delta D) \geq Pr(\Delta A, \Delta B_i, \Delta C_i, \Delta D)$$

Probabilities can be further broken down into rounds and steps *i.e.* MD5 [WY05]:

Definition 4.1.1 (Round differential). $(\Delta R_{i,j-1} \to \Delta R_{i,j})$ is a round differential where i is the current compression function iteration and $1 \leq j \leq 4$. Therefore, an iterated differential can be expanded to this:

$$\Delta H_i \xrightarrow{M_i, M_i'} \Delta H_{i+1} \Leftrightarrow \Delta H_i \to \Delta R_{i+1,1} \to \Delta R_{i+1,2} \to \Delta R_{i+1,3} \to \Delta R_{i+1,4} = \Delta H_{i+1}$$

The round differential itself can be split into the step differentials:

$$(\Delta R_{i,j-1} \to \Delta R_{i,j}) \Leftrightarrow \Delta R_{i,j-1} \to \Delta A_{i,j,1} \to \ldots \to \Delta A_{i,j,16} = \Delta R_{i,j-1}$$

where $(\Delta A_{i,j,t-1} \to \Delta A_{i,j,t})$ is the differential for step t in round j of iteration i.

Theorem 4.1.1 (Probabilities of full differentials, rounds and steps in MD5)**.** *The probability of an MD5 iteration is at least as high as the probabilities of the round differentials:*

$$Pr(\Delta H_i \xrightarrow{M_i, M_i'} \Delta H_{i+1}) \geq \prod_{j=1}^{4} Pr(\Delta R_{i,j-1} \to \Delta R_{i,j})$$

The same holds for step probabilities:

$$Pr(\Delta R_{i,j-1} \to \Delta R_{i,j}) \geq \prod_{t=1}^{16} Pr(\Delta A_{i,j,t-1} \to \Delta A_{i,j,t})$$

Differential characteristics are the fundamental element because they describe how differences propagate through an algorithm. In the case of MD5, the differences of each step are important because with this knowledge, collisions can be constructed. A collision can be found using a differential where the output difference is zero.

4.2 Types of Differences

The following section covers all types of differences necessary for covering the attacks on MD5. Moreover, correlations are analyzed. The most sophisticated type of difference, the *generalized difference*, will be subject to further examination. This includes its behaviour on the basic operations in the compression function of MD5.

4.2.1 XOR difference and modular difference

The first difference type used for differential cryptanalysis was the `XOR` difference [BS91]. On a bit level, we can define it by

Definition 4.2.1 (Bitwise `XOR` difference)**.**

$$\delta_X(bit_a, bit_b) = \begin{cases} 0 & \text{if } bit_a = bit_b \\ 1 & \text{if } bit_a \neq bit_b \end{cases}$$

Definition 4.2.2 (`XOR` difference)**.** Using bitwise $\oplus$-operations:

$$\Delta_X(x_1, x_2) = x_1 \oplus x_2 = \mathop{\Big|\Big|}_{i=0}^{w-1} \delta_X(x_{1,i}, x_{2,i}) = \{0,1\}^w$$

$i = 0$ denotes the least significant bit and $i = w - 1$ the most significant bit.

Later on, Berson *et al.* [Ber93] used a *modular difference* on the MD5 hash function:

Definition 4.2.3 (Modular difference or subtraction difference)**.**

$$\Delta_M(x_1, x_2) = (x_1 - x_2) \mod 2^w = \{0,1\}^w$$

These differences were very well suited for cryptographic primitives incorporating `XOR` operations. For MD5, which uses `ADD` operations extensively, other types of differences were necessary for more sophisticated cryptanalysis.

4.2.2 Signed differences

This third type of difference incorporates both `XOR` and modular difference. The exact correlation of these types will be dealt with later on in Section 4.2.4. First of all, we need to define a bitwise signed difference δ_s:

Definition 4.2.4 (Bitwise signed difference)**.**

$$\delta_S(bit_a, bit_b) = \begin{cases} 0 & \text{if } bit_a = bit_b \\ 1 & \text{if } bit_a > bit_b \\ -1 & \text{if } bit_a < bit_b \end{cases}$$

Definition 4.2.5 (Signed difference)**.**

$$\Delta_S(x_1, x_2) = \mathop{\Big|\Big|}_{i=0}^{w-1} \delta_S(x_{1,i}, x_{2,i}) = \{-1, 0, 1\}^w$$

x_1 and x_2 are binary strings with length w. $x_{1,i}$ and $x_{2,i}$ denote the bit at position i. The least significant bit is at index 0. The signed difference is a concatenation of bitwise signed differences. For each bit pair, three difference values are possible.

Signed differences offer a better approach on dealing with propagations in MD5. In the original work by Wang *et al.* [WY05] a combination of Δ_M and Δ_X was used. However, the term *signed difference* was not mentioned at all. Further work by Xie *et al.* [XFL08] introduced signed differences.

4.2.3 Generalized Differences

Another way to define differences is using *generalized conditions* [CR06]. They will be later referred to as Δ_G. They are influenced by the idea of signed bit differences Δ_S, however, 16 possible conditions on a pair of bits can be defined (Table 4.1).

Table 4.1: Generalized conditions on pair of bits (x_1, x_2) [CR06]

$\delta_G(x_1, x_2)$	(0,0)	(1,0)	(0,1)	(1,1)
?	$\surd$	$\surd$	$\surd$	$\surd$
-	$\surd$	-	-	$\surd$
x	-	$\surd$	$\surd$	-
0	$\surd$	-	-	-
u	-	$\surd$	-	-
n	-	-	$\surd$	-
1	-	-	-	$\surd$
#	-	-	-	-

$\delta_G(x_1, x_2)$	(0,0)	(1,0)	(0,1)	(1,1)
3	$\surd$	$\surd$	-	-
5	$\surd$	-	$\surd$	-
7	$\surd$	$\surd$	$\surd$	-
A	-	$\surd$	-	$\surd$
B	$\surd$	$\surd$	-	$\surd$
C	-	-	$\surd$	$\surd$
D	$\surd$	-	$\surd$	$\surd$
E	-	$\surd$	$\surd$	$\surd$

Generalized differences are more sophisticated than signed differences because multiple cases of bit relations can be covered by such a difference.

4.2.4 Correlations between types of differences

The previously defined differences are not completely independent from each other. This section will discuss how they interact. First of all, a small example shows different values for a given message pair:

$$\begin{array}{ll}
x_1 & (0,0,0,1,1,0,0,0) \\
x_2 & (0,0,0,0,1,0,1,0) \\
\hline
\Delta_X(x_1,x_2) & (0,0,0,1,0,0,1,0) \\
\Delta_M(x_1,x_2) & (0,0,0,0,1,1,1,0) \\
\Delta_S(x_1,x_2) & (0,0,0,1,0,0,-1,0) \\
\Delta_G(x_1,x_2) & (\mathtt{0},\mathtt{0},\mathtt{0},\mathtt{n},\mathtt{1},\mathtt{0},\mathtt{u},\mathtt{0})
\end{array}$$

Signed differences can be notated in a different manner as well. This is done by using the modular difference but splitting its result into summands. Using the upper example, it can be rewritten as:

$$\Delta_S(x_1,x_2) = 0\cdot 2^7 + 0\cdot 2^6 + 0\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 - 1\cdot 2^1 + 0\cdot 2^0 = 2^4 - 2^1$$

This result can be easily converted to the modular difference Δ_M:

$$\Delta_S(x_1,x_2) = 2^4 - 2^1 = 14 = \Delta_M(x_1,x_2)$$

Further works by Wang *et al.* or Stevens only use the index and the sign for a short notation:

$$\Delta_S(x_1, x_2) = 2^4 - 2^1 = [+4, -1]$$

Correlation of Δ_X and Δ_M. We will start with the two basic difference types. The following example should show that for a given Δ_S, different results for Δ_X are possible. a and b are two 16-bit integers. We define $\Delta_M(a, b) = 2^9$ and see the variants for Δ_X:

Number of difference bits in $\Delta_X(a, b)$ is 1:

a	0000 00**1**0 0000 0000
b	0000 00**0**0 0000 0000
$\Delta_X(a, b)$	0000 00**1**0 0000 0000

Number of difference bits in $\Delta_X(a, b)$ is 2:

a	0000 0**10**0 0000 0000
b	0000 0**01**0 0000 0000
$\Delta_X(a, b)$	0000 0**11**0 0000 0000

Number of difference bits in $\Delta_X(a, b)$ is 3:

a	0000 **001**0 0000 0000
b	0000 **110**0 0000 0000
$\Delta_X(a, b)$	0000 **111**0 0000 0000

This pattern continues as the number of 1-bits in the difference grows and moves toward the most significant bit. For the purpose of this example, only 16-bit values were chosen. For MD5, only integers with 32 bits are necessary.

Correlation of Δ_X, Δ_M and Δ_S. It can be proven that a modular difference could map to a variety of `XOR` differences and also vice versa. The upper example has shown a mapping from Δ_M to many possible Δ_X. Thus, both modular difference and `XOR` difference cannot be used to accurately measure a message pair when applying differential cryptanalysis. The correlation between Δ_X, Δ_M and Δ_S can be defined here: [XFL08]

$$\Delta_X(x_1, x_2) = \sum_{i=0}^{w-1} (x_{1,i} - x_{2,i}) \bullet 2^i \mod 2^w \equiv \mathop{\Big|\Big|}_{i=0}^{w-1} (x_{1,i} - x_{2,i}) = \Delta_S$$

Correlation of Δ_S and Δ_G. These two types of differences are extensively used in the attacks in following chapters. Signed differences are employed in attacks by Wang *et al.* and Stevens. Moreover, the generalized differences are used by the *nltool* which will be covered later on in Section 4.4. Each possible outcome of the signed difference can be mapped to a generalized difference:

$$\delta_S(0) \triangleq \delta_G(-), \quad \delta_S(1) \triangleq \delta_G(\mathtt{n}), \quad \delta_S(-1) \triangleq \delta_G(\mathtt{u})$$

4.2.5 Probabilities of differences

It is important to determine the probabilities given to match a difference or condition. For a signed bit difference, we can propose the following:

$$Pr(\Delta_S x) = \prod_{i=0}^{w-1} 2^{-1} \quad \text{where} \;\; \delta_S(x_{1,i}, x_{2,i}) \neq 0$$

Example of calculating probabilities. The following example shows the calculation of a step i in the third round with the following generalized conditions. In this case, we have differences on the most significant bit only, other because of this, we shorten the values of Δ_G.

$$\begin{aligned}
\Delta_G a_{i-4} &= \Delta_G[-- \ldots -] \\
\Delta_G a_{i-3} &= \Delta_G[-- \ldots -] \\
\Delta_G a_{i-2} &= \Delta_G[\mathtt{u}- \ldots -] \\
\Delta_G a_{i-1} &= \Delta_G[\mathtt{u}- \ldots -] \\
\Delta_G a_{i} &= \Delta_G[\mathtt{u}- \ldots -]
\end{aligned}$$

The step operation with generalized differences can be written as:

$$\Delta_G a_i \xleftarrow{Pr=?} (\Delta_G f_i(\Delta_G a_{i-3}, \Delta_G a_{i-2}, \Delta_G a_{i-1}) + \Delta_G w_{z_i} + \Delta_G k_i) \lll s_i + \Delta_G a_{i-4}$$

In this step, $f_i = h$ (the third round) and no message word difference $\Delta_G w_{z_i}$ exists. The constant can also be removed.

$$\Delta_G[\mathtt{u}- \ldots -] \xleftarrow{Pr=?} \Delta h(\Delta_G[-- \ldots -], \Delta_G[\mathtt{u}- \ldots -], \Delta[\mathtt{u}- \ldots -]) + \Delta[\mathtt{u}- \ldots -]$$

Looking at the most significant bit only and the Table 4.3, there is only one outcome for the boolean function h with these bit differences:

$$\delta_G h(\delta_G[-], \delta_G[\mathtt{u}], \delta_G[\mathtt{u}]) = \delta_G[-]$$

So, the step operation can be reduced to:

$$\Delta_G[\mathtt{u}-\ldots-] \xleftarrow{Pr=?} \Delta_G[--\ldots-] + \Delta_G[\mathtt{u}-\ldots-]$$

In this case, this is always true, so the probability of this step is 1.

4.2.6 Operations on generalized differences

This section will deal with the propagations of generalized differences along all bit operations specific for MD5: add, rotate and boolean function F. Moreover, the carry expansion is explained.

Rotate

A rotation simply rotates all differences in a specified direction and shift value. For example, we use several different types of generalized differences and rotate them by four to the left:

$$\Delta_G[-\mathtt{unx0n1n}] \lll 4 = \Delta_G[\mathtt{0n1n} - \mathtt{unx}]$$

Add

Many important cases of `ADD` operations on generalized conditions are presented. We compare them by showing the same operations on signed differences. If an `ADD` operation on the most significant bit happens, the carry drops out because it is a modular addition. The following simplified cases are done in 2-bit words.

$$\begin{aligned}
\Delta_G[-\mathtt{n}] \boxplus \Delta_G[--] = \Delta_G[-\mathtt{n}] &\Leftrightarrow 2^0 + 0 = 2^0 \\
\Delta_G[-\mathtt{n}] \boxplus \Delta_G[-\mathtt{n}] = \Delta_G[\mathtt{n}-] &\Leftrightarrow 2^0 + 2^0 = 2^1 \\
\Delta_G[\mathtt{n}-] \boxplus \Delta_G[\mathtt{n}-] = \Delta_G[--] &\Leftrightarrow 2^1 + 2^1 = 2^2 \mod 4 = 0 \\
\Delta_G[-\mathtt{n}] \boxplus \Delta_G[-\mathtt{u}] = \Delta_G[--] &\Leftrightarrow 2^0 - 2^0 = 0
\end{aligned}$$

Carry Expansion

The carry expansion allows you reinterpret a difference as follows. For example, we can rewrite the signed difference 2^0 like this:

$$2^0 = 2^1 - 2^0 = 2^2 - 2^1 - 2^0 = 2^3 - 2^2 - 2^1 - 2^0$$

These equivalences can also be shown as generalized conditions (in this case 4-bit-words):

$$\Delta_G[---\mathtt{n}] \Leftrightarrow \Delta_G[--\mathtt{un}] \Leftrightarrow \Delta_G[-\mathtt{unn}] \Leftrightarrow \Delta_G[\mathtt{unnn}]$$

4.2.7 Propagations of the boolean function F

The boolean step function is used in the step operation and uses four different variants in MD5. By choosing clever inputs, generalized differences can be passed through or blocked. The main three generalized differences, '-', 'n', 'u' are shown in all variants. The table shows which outcomes are possible. For the functions in rounds one and two, see Table 4.2, for three and four, 4.3. Definitions are partly taken from the MD4 analysis of Schläffer [Sch06].

Types of outcomes. A '$\surd$' shows that this output difference happens with a probability of 100 % without any additional constraints. The symbol '$\lightning$' means a case which can never happen. Finally, there are cases where the outcome relies not on differences, but on the actual values of the bits (0 and 1). On top of that, conditions arise involving two bits being equal or not equal. Depending on the case, the probability is then lowered to 50 %.

Table 4.2: Output differences for boolean functions f (`IF`) and g (`IF3`) based on [Sch06]

$\delta_G x$	$\delta_G y$	$\delta_G z$	$\delta_G f =$`-`	$\delta_G f =$`n`	$\delta_G f =$`u`	$\delta_G g =$`-`	$\delta_G g =$`n`	$\delta_G g =$`u`
-	-	-	$\checkmark$	$\lightning$	$\lightning$	$\checkmark$	$\lightning$	$\lightning$
n	-	-	$y = z$	$y = 1, z = 0$	$y = 0, z = 1$	$z = 0$	$z = 1$	$\lightning$
u	-	-	$y = z$	$y = 0, z = 1$	$y = 1, z = 0$	$z = 0$	$\lightning$	$z = 1$
-	n	-	$x = 0$	$x = 1$	$\lightning$	$z = 1$	$z = 0$	$\lightning$
-	u	-	$x = 0$	$\lightning$	$x = 1$	$z = 1$	$\lightning$	$z = 0$
-	-	n	$x = 1$	$x = 0$	$\lightning$	$x = y$	$x = 1, y = 0$	$x = 0, y = 1$
-	-	u	$x = 1$	$\lightning$	$x = 0$	$x = y$	$x = 0, y = 1$	$x = 1, y = 0$
n	-	u	$y = 1$	$\lightning$	$y = 0$	$y = 0$	$y = 1$	$\lightning$
u	-	n	$y = 1$	$y = 0$	$\lightning$	$y = 0$	$\lightning$	$y = 1$
n	-	n	$y = 0$	$y = 1$	$\lightning$	$y = 1$	$y = 0$	$\lightning$
u	-	u	$y = 0$	$\lightning$	$y = 1$	$y = 1$	$\lightning$	$y = 0$
n	u	-	$z = 0$	$\lightning$	$z = 1$	$\lightning$	$z = 1$	$z = 0$
u	n	-	$z = 0$	$z = 1$	$\lightning$	$\lightning$	$z = 0$	$z = 1$
n	n	-	$z = 1$	$z = 0$	$\lightning$	$\lightning$	$\checkmark$	$\lightning$
u	u	-	$z = 1$	$\lightning$	$z = 0$	$\lightning$	$\lightning$	$\checkmark$
-	n	u	$\lightning$	$x = 1$	$x = 0$	$x = 1$	$x = 0$	$\lightning$
-	u	n	$\lightning$	$x = 0$	$x = 1$	$x = 1$	$\lightning$	$x = 0$
-	n	n	$\checkmark$	$\lightning$	$\lightning$	$x = 0$	$x = 1$	$\lightning$
-	u	u	$\lightning$	$\lightning$	$\checkmark$	$x = 0$	$\lightning$	$x = 1$
n	n	n	$\lightning$	$\checkmark$	$\lightning$	$\lightning$	$\checkmark$	$\lightning$
u	n	u	$\checkmark$	$\lightning$	$\lightning$	$\checkmark$	$\lightning$	$\lightning$
u	u	n	$\checkmark$	$\lightning$	$\lightning$	$\lightning$	$\lightning$	$\checkmark$
n	u	u	$\lightning$	$\lightning$	$\checkmark$	$\checkmark$	$\lightning$	$\lightning$
u	n	n	$\lightning$	$\checkmark$	$\lightning$	$\checkmark$	$\lightning$	$\lightning$
n	n	u	$\checkmark$	$\lightning$	$\lightning$	$\lightning$	$\checkmark$	$\lightning$
n	u	n	$\checkmark$	$\lightning$	$\lightning$	$\checkmark$	$\lightning$	$\lightning$
u	u	u	$\lightning$	$\lightning$	$\checkmark$	$\lightning$	$\lightning$	$\checkmark$

`IF`: $f(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$
`IF3`: $g(x, y, z) = (x \wedge z) \vee (y \wedge \neg z)$

Table 4.3: Output differences for boolean functions h (`XOR`) and k (`ONX`) based on [Sch06]

$\delta_G x$	$\delta_G y$	$\delta_G z$	$\delta_G h =$`-`	$\delta_G h =$`n`	$\delta_G h =$`u`	$\delta_G k =$`-`	$\delta_G k =$`n`	$\delta_G k =$`u`
-	-	-	$\surd$	$\lightning$	$\lightning$	$\surd$	$\lightning$	$\lightning$
n	-	-	$\lightning$	$y = z$	$y \neq z$	$z = 0$	$y = 0, z = 1$	$y = 1, z = 1$
u	-	-	$\lightning$	$x \neq y$	$y \neq z$	$z = 0$	$y = 1, z = 1$	$y = 0, z = 1$
-	n	-	$\lightning$	$x = z$	$x \neq z$	$\lightning$	$z = 1$	$z = 0$
-	u	-	$\lightning$	$x \neq y$	$x = z$	$\lightning$	$z = 0$	$z = 1$
-	-	n	$\lightning$	$x = y$	$x \neq y$	$x = 1$	$x = 0, y = 1$	$x = 0, y = 0$
-	-	u	$\lightning$	$x \neq y$	$x = y$	$x = 1$	$x = 0, y = 0$	$x = 0, y = 1$
n	-	u	$\surd$	$\lightning$	$\lightning$	$\lightning$	$y = 0$	$y = 1$
u	-	n	$\surd$	$\lightning$	$\lightning$	$\lightning$	$y = 1$	$y = 0$
n	-	n	$\surd$	$\lightning$	$\lightning$	$\surd$	$\lightning$	$\lightning$
u	-	u	$\surd$	$\lightning$	$\lightning$	$\surd$	$\lightning$	$\lightning$
n	u	-	$\surd$	$\lightning$	$\lightning$	$z = 1$	$z = 0$	$\lightning$
u	n	-	$\surd$	$\lightning$	$\lightning$	$z = 1$	$\lightning$	$z = 0$
n	n	-	$\surd$	$\lightning$	$\lightning$	$z = 1$	$\lightning$	$z = 0$
u	u	-	$\surd$	$\lightning$	$\lightning$	$z = 1$	$z = 0$	$\lightning$
-	n	u	$\surd$	$\lightning$	$\lightning$	$x = 0$	$\lightning$	$x = 1$
-	u	n	$\surd$	$\lightning$	$\lightning$	$x = 0$	$x = 1$	$\lightning$
-	n	n	$\surd$	$\lightning$	$\lightning$	$x = 0$	$\lightning$	$x = 1$
-	u	u	$\surd$	$\lightning$	$\lightning$	$x = 0$	$x = 1$	$\lightning$
n	n	n	$\lightning$	$\surd$	$\lightning$	$\lightning$	$\lightning$	$\surd$
u	n	u	$\lightning$	$\surd$	$\lightning$	$\lightning$	$\lightning$	$\surd$
u	u	n	$\lightning$	$\surd$	$\lightning$	$\surd$	$\lightning$	$\lightning$
n	u	u	$\lightning$	$\surd$	$\lightning$	$\surd$	$\lightning$	$\lightning$
u	n	n	$\lightning$	$\lightning$	$\surd$	$\surd$	$\lightning$	$\lightning$
n	n	u	$\lightning$	$\lightning$	$\surd$	$\surd$	$\lightning$	$\lightning$
n	u	n	$\lightning$	$\lightning$	$\surd$	$\lightning$	$\surd$	$\lightning$
u	u	u	$\lightning$	$\lightning$	$\surd$	$\lightning$	$\surd$	$\lightning$

`XOR`: $h(x, y, z) = x \oplus y \oplus z$
`ONX`: $k(x, y, z) = y \oplus (x \vee \neg z)$

4.3 Basic steps for differential attacks

This section describes the basic steps that are required to build a differential attack on hash functions of the MD-family.

Attacking the hash function. Colliding differential characteristics always exist. For two- or more colliding message blocks, finding such characteristics is easier because only a near-collision for the first message block has to be created. With this near-collision and therefore a free-start collision on the second block, a complete collision can be constructed. Single-block collisions are much harder, because a full collision has to be created in the first block.

Attacking the compression function. The collision attack strategy can be broken down into four basic steps [MNS12]:

1. Find a characteristic with a high probability after the first round.
2. Find a characteristic for the first round. A high probability is not always necessary
3. Message modification is a technique to increase the probability of the characteristic. This is done by fulfilling conditions for the first round.
4. The remaining task is only probabilistic and uses random values to find a complete message pair fulfilling all conditions.

Finding a good differential characteristic for the all rounds requires great effort. The overall complexity of the attack depends on the quality of the characteristic. As you can see, it is important that the probabilistic part of the attack should be as fast as possible. The following properties are important for designing a good differential path [XFL08]:

1. A differential path for the first round has to exist in order to achieve feasible collisions.
2. The path in the second round has to reduce the differences from the first round.
3. The amount of free message words before the start of the differential has a huge impact on building paths.
4. Changes of the path in the second round should not propagate back to the first round.
5. Only a small number of differentials have to occur in rounds three and four.
6. The behaviour of signed differences in each step has to be used in order to generate forward and backward propagations. This is applicable to every hash function. Differences are either desired or unwanted and have to be used accordingly.
7. The boolean function F of MD5 can be used to stop propagations of differences. See Section 4.2.7 for further details.

4.4 The Non-Linear Toolbox

The previous techniques are far from trivial. Rechberger *et al.* [CR06] and Mendel *et al.* [MNS11] have developed a tool which can find complex nonlinear differential characteristics using generalized conditions. These conditions are propagated in bit slices.

Bit sliced step operation on MD5. All state words in the toolbox are written as uppercase letters. The step operation is very complex when handling generalized propagations. Therefore, it is split up into sub steps. The following state words are used, the index always defines the current step in the compression function:

- `W0`,...,`W15` represent the message words $w_0, \dots, w_{15}$.
- `A-4`,...,`A-1` are the chaining input (which is also $a_{-4}, \dots, a_{-1}$).
- `F0`,...,`F63` are the results of the boolean function $f_0, \dots, f_{63}$.
- `B0`,...,`B63` are the results after applying the message word and the additive constant to the result of the boolean function.
- `A0`,...,`A63` are the results after the rotation and the last add operation.

Input/Output. Figure 4.2 shows a typical differential characteristic with the output of the intermediate state registers `F` and `B` as well. Later outputs will omit these registers in order to save space.

Propagations. The inputs and outputs of each step are analyzed in each step operation [CR06]. Three options are possible: The conditions contradict each other, the conditions are consistent or they are consistent if certain additional bit conditions are also fulfilled. An example would be the propagation of the expanded message words to the internal state words.

Two-Bit Conditions. Generalized conditions only concern a single bit position. However, conditions could exist concerning multiple bits. Later on, we will see that especially two-bit conditions are crucial for differential paths. Most of the time, the two-bit conditions can be traced back to the boolean function F. An example would be $f(a_{i-1}, a_{i-2}, a_{i-3})$. If a propagation of Δa_{i-1} should be stopped, Table 4.2 shows that $a_{i-2} \neq a_{i-3}$, hence leading to a two-bit condition. Such conditions are not shown in the characteristic (like Figure 4.2) but can lead to inconsistencies.

Inconsistency Checks. It is necessary to detect inconsistent differential characteristics because search algorithms could stop at this point and return to a non inconsistent state. A complete check of all conditions is considerably slow, however, doing simple tests at a later point may be enough to uncover contradictions. There are different types of checks [MNS11]:

- Two-bit condition checks analyse conditions where two bits should be equal or unequal. This is achieved by creating a linear system of equations representing the conditions and then try to solve it.
- The complete condition check requires a high computational effort. It checks every bit with the generalized difference `-` or `x` whether both imposed cases (`0`/`1` and `n`/`u`) are valid. This check should only be used on rare occasions due to its expensiveness.

4.4.1 Searching for Differential Characteristics

As mentioned earlier, the tool can also search for differential characteristics in an automated manner [MNS11]. The algorithm can be split into three phases: decision, deduction and backtracking. The same idea can be found in other topics, *i.e.* SAT solvers [GPFW96]. In the first phase, an undetermined bit is picked and set to more restricted condition as described in the algorithm. The second phase involves propagation and a check of contradiction. When a contradiction is found, backtracking is necessary in order to continue the process with a previously valid characteristic. See algorithm 3 for further details.

Algorithm 3 Search algorithm of the *nltool* [MNS11]

INPUT: Characteristic filled with undetermined bits
OUTPUT: Full determined characteristic

1 U is a set of all bits with generalized condition `'?'` or `'x'`
2 **loop** until U is empty
Decision phase
3 Pick a random bit x in U
4 **if** x is `'?'` **then**
5 Set x to `'-'`
6 **else if** x is `'x'` **then**
7 Set x to `'u'` or `'n'` in a random manner
Deduction phase
8 Compute the propagation
9 **if** a contradiction is not found **then**
10 Continue with loop and go to step 2
Backtracking phase
11 Set the characteristic back to an earlier (non-contradicting) state and go to step 2

Search configurations. We are able to control which bits are picked first in the search algorithm. So, depending on the attack, we can optimize the algorithm to focus on certain internal registers first instead of guessing completely random bits. The following options are possible:

```
-4 A: 01100111010001010010001100000001
-3 A: 00010000000110010010101010001110110
-2 A: 10011000101110101101110011111110
-1 A: 11101111111001101101010101110001001
 0 B: --------1----------0----01110110 F: 10011000101110101101110011111110
 0 A: ?----------------110011--------- W: 1?-1----01-101---1-000--11111111
 1 B: ------------------11110------0-- F: 1---1---1---1---1011101-1---1---
 1 A: -----------------00---------00-- W: -10111110111--00--11010111111-11
 2 B: ---1-----1-0-01--00-----01--00-- F: -----------0-1---010--1-----10--
 2 A: 00-----01---------1-----110-01-- W: 11110010010000100001111101111111
 3 B: --0---0-0-----011-10--11-0---101 F: ------1-------00--00---------001
 3 A: ---0-----1--0---0-------010-00-- W: 11011100001111011101-1100111101
 4 B: 0--1-0--0-1-01--0--1--01-------1 F: -------------------------1--00--
 4 A: 00---01--1-----0------------1-0- W: 0--10001011101-00--10-01001010-1
 5 B: --------0-1--1-1-10-1n-----1--0- F: 00-------1---------------10-0---
 5 A: ---------n-----0--0------------- W: 0-10--1-0011-10--101-n001001-00-
 6 B: 1--1---00-------------0-------0- F: ---------1----------------------
 6 A: 001--0--1n---0-1--0---00-------- W: 1111--110----00-1100-0000111001
 7 B: ---1-0--1---11---0-0--1--1-1---- F: 00---0---1-----0----------------
 7 A: 1111----0n-----------------0-010 W: 11111001101111111010111110111110
 8 B: 00101-00nu0-00-0-0001010-000-0-0 F: 001------n--------0-------------
 8 A: 00n1-00-nu0001010000000-000-0--0 W: 0110110001010011-000101010000010
 9 B: 1-1-111-0--11011-11-111110101011 F: 0011-0--un--------0---0---------
 9 A: 11n1----0n11111110111---111-0-01 W: 101-111-0--11--1011--01110101011
10 B: u000100nuu00-1-001100----011-1-1 F: 0011----000001010-000---000---10
10 A: 10n111101n1-1-un110011--n11---01 W: n0-0-1--0--0-1100-110011000-1-1-
11 B: nu0uuuuuuu01-0u-----1-00n11--110 F: 10n1----011-11u110001---n11-0--1
11 A: 1n111--0nu0-n-un110n010-0------- W: 010001001---1-1111--1---0--1-1-1
12 B: -------n01001-111unn1111000----0 F: 1un11---nn111-11111u11--111---01
12 A: 1u1000100n1-1-un010n11001------- W: 1001111-001110101011111-111-1--1
13 B: 00n1---1nu--n1un100n------001001 F: 101111-0100-n-un110n01--0-----01
13 A: 01nn101n1n1-10un11n1uuu00---nu-- W: 10000011---1--11100100-1-10-1-00
14 B: 00uu1110nu-0unn--111n110n0---000 F: 1u1u0-10011-1-un010nu10-0-----11
14 A: 0n11nuunuu10101u0u0u1nunu---01-- W: 00000-1010-1-01--001--10--111110
15 B: u01un10n10101nn-------n1n1110u01 F: 11nnn01nun1-1-un0101uuu0n----u--
15 A: unn1010n01n1un0nunnnnu001---01-- W: 11101-----------0--------10----1
16 B: -1101u0n0nn1u01n--1nu10nu111---- F: 0n110u0n01n-u0n1unnnn10nu---01--
16 A: n0nu0u111n---1u000100n11111000--
17 B: un011-0--001nuu-11---00nnuuuuu11 F: u0nu01unun-1-nununnn01un1---00--
17 A: n10n01-1nu---1110010nn1un001----
18 B: 101n1nu000----u1un11--1u-------- F: nn0n01111u---1un0010nn11n---0---
18 A: 001101--00----0---0n11011001---0
19 B: ------1n1un00unn-0-100------0--- F: 01n101--0u----n10000nn011001----
19 A: 0--1----01101-0-0nu1unn111un0001
20 B: u00nun1----1--u10-1u-nu001nu1001 F: 0-110---0u----0---unu1n1100n----
20 A: u------0-u010-u-1nn0010unnn1001n
21 B: 101n0011nu----10--n111uu11u1-001 F: 0-------01----0--nuu01nun1u100-1
21 A: 10-00-00-00un-1-nuu-1000010110n0
22 B: u--0-0011nuuu0u--0--0n---nu---00 F: u--0---0-001n-u-101-uu0001n10010
22 A: n--0100n10-00-1-100--0-01100nu10
23 B: n--110---011n1110u10--u1n00u11-0 F: n--0--00-000n-1-1uu-1000n1001010
23 A: n0-uu010011nu---001--1-0n0111100
24 B: 1001---n1u0101u10u-----10u0nnn0u F: n--0100n-0-00---u0u--0-010011uu0
24 A: u--101010u010-0------u-001u0111-
```

Figure 4.2: Sample differential characteristic output of `nltool` for the first 25 steps of MD5

- Select the registers where bits are picked. We can define certain state registers , *i.e.* `A5`.
- Choose how the bits are selected word-wise.
- Define which values to guess. Usually we only choose `?` or `x` bits, but this could be altered as well.
- Define the choices on the previously picked bits. For each choice, the random distribution probability can be set as a value between 0 and 1. 0 defines that the choice is never picked, 1 always uses this choice.

The default configuration simply acts on all state registers `A`,`B`,`F` and `W`. It picks all `?` and `x` bits and sets them accordingly. For `x`-bits, `u` and `n` are uniformly chosen, *i.e.* their probability is 0.5 and 0.5.

With the *nltool*, we can analyze, verify and search for differential paths. A common input/output format enables us to easily manipulate data. The main function is calculating propagations and attempting to reduce implementational effort for attacks by using automations. Custom search configurations enable you to optimize the tool for specific attacks rather than to rely on the default configuration.

Chapter 5

Two Block Collisions and Further Improvements

In 2004 Wang *et al.* [WY05] published a method to create a collision in MD5 with two message blocks. This was done by using differential characteristics and message modification. Klima [Kli06] introduced the idea of *tunnels* to speed up Wang's original collision attack. Further refinements were made by Stevens [Ste06] and Xie *et al.* [XFL08].

This chapter deals with an in-depth analysis of Wang's attack. Moreover, the principles of tunnels are explained. Using the *nltool*, differential characteristics will be shown for analysis. For a better understanding of these tunnels, we will create our own tunnel patterns. Finally, we will adapt the *nltool* to run Stevens' two-block collision attack and measure results.

5.1 Wang's original approach

Wang *et al.* [WY05] use a combination of `XOR` and modular differences for their characteristic. This results that their differential cryptanalysis use the signed bit difference. The behaviour and correlations of these differences were explained in Section 4.2. They created differential paths for the compression function of the MD5 hash function. Using this path, they constructed a set of *sufficient conditions* over the bits a_i in the first and second block. These conditions are represented as a system of conditions to guarantee that the differences exactly follow the differential path [Ste12b]. The characteristic differential for both message blocks can be seen in Figure 5.1. Because of the feed-forward behaviour of the compression function, you can see in the second block that the chaining input $a_{-4}, \ldots, a_{-1}$ exactly cancels out the last four internal register states $a_{60}, \ldots, a_{63}$ resulting on a full collision.

The following collision differential is created. M_0 and M_1 are the two message blocks. These differences were chosen because they provide a low complexity for the collision finding algorithm.

$$\Delta H_0 \xrightarrow{M_0, M_0'} \Delta H_1 \xrightarrow{M_1, M_1'} \Delta H_2 = \Delta H = 0$$

The following differences are used. The original $IV = H_0$ is used.

$$\Delta M_0 = M_0' - M_0 = (0,0,0,0,2^{31},0,0,0,0,0,0,2^{15},0,0,2^{31},0)$$
$$\Delta M_1 = M_1' - M_1 = (0,0,0,0,2^{31},0,0,0,0,0,0,-2^{15},0,0,2^{31},0)$$
$$\Delta H_1 = (2^{31},2^{31}+2^{25},2^{31}+2^{25},2^{31}+2^{25})$$

Wang uses an extensive set of conditions such that the differential holds for each step. For example, the following bit conditions have to hold for a_4:

$$a_{4,7}=0, a_{4,8}=a_{3,8}, a_{4,9}=a_{3,9}, a_{4,10}=a_{3,10}, a_{4,11}=a_{3,11}, a_{4,12}=1, a_{4,13}=a_{3,13}, a_{4,14}= a_{3,14}, a_{4,15}=a_{3,15}, a_{4,16}=a_{3,16}, a_{4,17}=a_{3,17}, a_{4,18}=a_{3,18}, a_{4,19}=a_{3,19}, a_{4,20}=1, a_{4,21}= a_{3,21}, a_{4,22}=a_{3,22}, a_{4,23}=a_{3,23}, a_{4,24}=0, a_{4,32}=0$$

The conditions of Wang either specify the binary value or a condition involving another bit. In all cases here, these two-bit conditions hold for the same bit position connected to the bit of the earlier step. With these conditions, the most naive approach would be by trying all possible values for the message words $w_0,\ldots,w_{15}$ and checking against the set of conditions. The complexity is very high due to many probabilistic fulfillings of the conditions. Using *message modification* (further described in Section 5.2), the performance and complexity can be considerably lowered.

5.2 Message modification of Wang

The message modification techniques are used by Wang to accelerate the attack by improving the probability of matching the conditions. There are two types of techniques:

1. *Basic message modification*: For a given differential ($\Delta H_i \xrightarrow{M_i,M_i'} \Delta H_{i+1}$), it modifies M_i such that the differential of the first round ($\Delta H_i \rightarrow \Delta R_{i+1,1}$) holds in a deterministic manner. Further details are discussed in Section 5.2.1.

2. *Advanced message modification*: As before, M_i is altered not only to hold $Pr(\Delta H_i \rightarrow \Delta R_{i+1,1}) = 1$ but also greatly improve the probability of the second round. See Section 5.2.2 for in-depth analysis.

In short terms, single-message modification adapts the message to hold for the first round. Multi-message does this for the second round, which is more complicated.

5.2.1 Basic message modification

The basic message modification (or also referred to by Wang as *single-message modification*) tries to modify the message words $w_0,\ldots,w_{15}$ that hold for all conditions in the first round in the differential characteristic.

For each of the 16 steps with $i = 0, ..., 15$ do

```
-4 A: --------------------------------
-3 A: --------------------------------
-2 A: --------------------------------
-1 A: --------------------------------
 0 A: -------------------------------- W: --------------------------------
 1 A: -------------------------------- W: --------------------------------
 2 A: -------------------------------- W: --------------------------------
 3 A: -------------------------------- W: --------------------------------
 4 A: ---------unnnnnnnnnnnnnnnn------ W: u-------------------------------
 5 A: n-------n----------------u------ W: --------------------------------
 6 A: nnnnnnuuu-----------unnnnnunnnnn W: --------------------------------
 7 A: --------u--unnnun--------------n W: --------------------------------
 8 A: u----------------------unn----nu W: --------------------------------
 9 A: n-----------------nu------------ W: --------------------------------
10 A: nn------------------------------ W: --------------------------------
11 A: n-----------unnnnnn----un------- W: ----------------n---------------
12 A: n-----nu------------------------ W: --------------------------------
13 A: n------------------------------- W: --------------------------------
14 A: n---------------u-----------n--- W: u-------------------------------
15 A: n-u----------------------------- W: --------------------------------
16 A: n-------------------------------
17 A: n-------------------------------
18 A: n-------------n-----------------
19 A: n-------------------------------
20 A: n-------------------------------
21 A: n-------------------------------
22 A: --------------------------------
23 A: --------------------------------
24 A: --------------------------------
25 A: --------------------------------
26 A: --------------------------------
27 A: --------------------------------
28 A: --------------------------------
29 A: --------------------------------
30 A: --------------------------------
31 A: --------------------------------
32 A: --------------------------------
33 A: --------------------------------
34 A: n-------------------------------
35 A: n-------------------------------
36 A: n-------------------------------
37 A: u-------------------------------
38 A: u-------------------------------
39 A: u-------------------------------
40 A: n-------------------------------
41 A: u-------------------------------
42 A: u-------------------------------
43 A: u-------------------------------
44 A: n-------------------------------
45 A: n-------------------------------
46 A: u-------------------------------
47 A: n-------------------------------
48 A: u-------------------------------
49 A: u-------------------------------
50 A: u-------------------------------
51 A: u-------------------------------
52 A: u-------------------------------
53 A: u-------------------------------
54 A: u-------------------------------
55 A: u-------------------------------
56 A: u-------------------------------
57 A: u-------------------------------
58 A: u-------------------------------
59 A: n-------------------------------
60 A: u-------------------------------
61 A: n-----n-------------------------
62 A: u-----n-------------------------
63 A: n-----n-------------------------
```

```
-4 A: n-------------------------------
-3 A: n-----n-------------------------
-2 A: n----nu-------------------------
-1 A: n-----n-------------------------
 0 A: u-----n------------------------- W: --------------------------------
 1 A: u-----n-------------------n----- W: --------------------------------
 2 A: unuuuuu---nuuuuu---nu---nuu----- W: --------------------------------
 3 A: u----nu-------------------nunnn- W: --------------------------------
 4 A: n------------------nuuunnu-----n W: u-------------------------------
 5 A: n---------un--un---------------- W: --------------------------------
 6 A: u--un-----------------unnn------ W: --------------------------------
 7 A: u----unnn-----unu--------------- W: --------------------------------
 8 A: u---------------------nuuu----nu W: --------------------------------
 9 A: u------------------n------------ W: --------------------------------
10 A: u------------------------------- W: --------------------------------
11 A: u-----------unnnnnn-----u------- W: ----------------u---------------
12 A: nnuuuuuu------------------------ W: --------------------------------
13 A: n------------------------------- W: --------------------------------
14 A: n---------------n-----------n--- W: u-------------------------------
15 A: n-u----------------------------- W: --------------------------------
16 A: n-------------------------------
17 A: n-------------------------------
18 A: n-------------n-----------------
19 A: n-------------------------------
20 A: n-------------------------------
21 A: n-------------------------------
22 A: --------------------------------
23 A: --------------------------------
24 A: --------------------------------
25 A: --------------------------------
26 A: --------------------------------
27 A: --------------------------------
28 A: --------------------------------
29 A: --------------------------------
30 A: --------------------------------
31 A: --------------------------------
32 A: --------------------------------
33 A: --------------------------------
34 A: u-------------------------------
35 A: n-------------------------------
36 A: n-------------------------------
37 A: u-------------------------------
38 A: u-------------------------------
39 A: u-------------------------------
40 A: n-------------------------------
41 A: u-------------------------------
42 A: u-------------------------------
43 A: u-------------------------------
44 A: u-------------------------------
45 A: u-------------------------------
46 A: n-------------------------------
47 A: u-------------------------------
48 A: n-------------------------------
49 A: n-------------------------------
50 A: n-------------------------------
51 A: n-------------------------------
52 A: n-------------------------------
53 A: n-------------------------------
54 A: n-------------------------------
55 A: n-------------------------------
56 A: n-------------------------------
57 A: n-------------------------------
58 A: n-------------------------------
59 A: u-------------------------------
60 A: n-------------------------------
61 A: u-----u-------------------------
62 A: n----un-------------------------
63 A: n-----u-------------------------
```

Figure 5.1: Characteristic of Wang's 2-block-collision [WY05]

1. Generate a random value for w_i. Calculate the corresponding a'_i. This is a normal MD5 step operation:

$$a'_i = (a_{i-4} + f_0(a_{i-1}, a_{i-2}, a_{i-3}) + w_i + k_i) \lll s_i + a_{i-1}$$

2. Check if a'_i meets all conditions for a_i. Two checks are necessary:
   (a) Check if all zero bits with a'_i match with the corresponding ones in a_i. This can be done by calculating (zero bit mask of a_i) $\wedge\, a'_i$ and comparing it with zero. If this is not the case, correct the wrong bits by applying the negated zero bit mask.
   (b) For all one bits do the same as above but use boolean `OR` operations.

3. If a correction of w_i is necessary, recalculate it:

$$w_i = (a_i - a_{i-1}) \ggg s_i - a_{i-4} - f_0(a_{i-1}, a_{i-2}, a_{i-3}) - k_i$$

4. Apply two bit conditions necessary for step $i+1$.

5.2.2 Advanced message modification

Basic message modification covers only finding message words to hold conditions for the first 16 steps. To go further, advanced techniques are introduced. Black *et al.* [BCH06] did an in-depth analysis of Wang's multi message approach.

Example. The general idea of multi-message modification is explained by an example. An in-depth analysis is made. The condition to be met is that the most significant bit of a_{16} has to be zero. First of all, we start by modifying the message word w_1 into w'_1. The rotate value in step 16 is 5. Because of this, an addition of 2^{26} is necessary, because $2^{26} \lll 5 = 2^{31}$:

$$w_1 = w_1 + 2^{26}$$

The latter is necessary for meeting the condition set in the example. Changing w_1 also affects the value of a_1 which has to be recalculated as well:

$$a_1 = (a_{-3} + f_1(a_0, a_{-1}, a_{i-2}) + w_1 + k_1) \lll 12 + a_0$$

Due to the fact that a_1 has changed (which is now represented as a'_1), the next four message words $w_2, \ldots, w_5$ have to be recalculated in a manner that other step values remain unchanged:

$$w_i = ((a_i - a_{i-1}) \ggg s_i) - a_{i-2} - f_2(a_{i-1}, a_{i-2}, a_{i-3}) - k_i \qquad 2 \le i \le 5$$

After this process, $w_1, w_2, w_3, w_4, w_5, a_1, a_{16}$ have been recalculated, whereas other step values a_i or message bits w_i stay unchanged. The necessary condition $a_{16,31} = 0$ can now hold. Wang uses other conditions that can be corrected with this technique. With these modifications, 37 conditions are undetermined in the first block and 30 conditions are undetermined in the second block for rounds 2-4. This leads to the probabilities of 2^{-37} and 2^{-30}.

5.2.3 Algorithm and results

For both message blocks, the algorithm generations a random message, uses the previously explained modification algorithms and repeats this process with an expected probability. See algorithm 4 for further details.

Algorithm 4 Wang's two-block collision attack [WY05]

Creates a collision with the following differences:
$\Delta H_0 \xrightarrow{(M_0, M_0'), 2^{-37}} \Delta H_1 \xrightarrow{(M_1, M_1'), 2^{-30}} \Delta H = 0$

loop until the first colliding block is found
  Select a random message M_0.
  Modify M_0 with basic and advanced message modification as described in 5.2
  Produce the first iteration differential with a probability of 2^{-37}.

$$\Delta M_0 \rightarrow (\Delta H_1, \Delta M_1)$$

  Test the characteristic with the compression function f on M_0 and M_0'.
loop until the first colliding block is found
  Select a random message M_1.
  Modify M_1 with basic and advanced message modification as described in 5.2
  Produce the first iteration differential with a probability of 2^{-30}.

$$(\Delta H_1, \Delta M_1) \rightarrow \Delta H = 0$$

  Test if the pair $(M_0 || M_1, M_0' || M_1')$ leads to a collision.

The overall complexity can be split into finding the first and the second colliding message block pair. For the first block, it does not exceed 2^{39} MD5 operations. The second block has a lower complexity with 2^{32} MD5 operations.

5.2.4 Errors in Wang's sufficient conditions

In 2005, Yajima and Shimoyama [YS05] tried to trace Wang's collision search algorithm. However, they were not able to find results. After extensive analysis, they found missing conditions and also corrected some of Wang's conditions.

They found errors in the 7th step. Some conditions were missing, other ones had to be altered. On top of that, sufficient conditions in the second message block were also corrected. After those adoptions, they reran the search algorithm and found many pairs for the first message blocks. It took them several hours to find one colliding message.

5.3 Using tunnels for faster results

In 2006, Klima introduced the idea of tunneling [Kli06]. It partly replaces the multi-message modification algorithm and provides faster collision search.

Limitations of message modification. Klima found out about limitations of the advanced message modifications by Wang *et al.* and introduced the idea of tunnels. The main problem of the message modification is the *point of verification*, which, in the case of the MD5 hash function, is at step 23. After the message search algorithm reaches this exact step, it is not able to change any state registers after this step. The remaining set of sufficient conditions can only be checked in a probabilistic manner. These are again the limitations [DLS11]:

1. The internal state registers $a_0, \ldots, a_{23}$ can be found in a *deterministic* manner satisfying all the conditions
2. All other registers $a_{24}, \ldots, a_{63}$ can be determined by trial and error in a *probabilistic* way

Tunnels. *Tunnels* can be used for adding more fixed conditions or to increase the probability of meeting conditions. Adding fixed conditions reduces the search size and therefore the complexity. The challenge when designing such tunnels is that dependencies between the internal state registers are rather complex. In simple terms, tunnels are very clever bit flips that do not affect anything before this *point of verification*. So they are changes at a certain step which *vanish* until step 24 and reappear afterwards in order to reduce the complexity.

Example: The tunnel in a_8. The following example shows how such a tunnel works by flipping bits in a_8. The following state registers and message words have to be adapted to allow this tunnel to work [DLS11]:

1. a_9 would be affected by changes in a_8. To circumvent this, change message word w_9.
2. a_{12} would be also be influenced by the changes in a_8. For a fix, adjust message word w_{12}.
3. The following state registers have changed: a_8, w_8, w_9 and w_{12}. When taking a look at the indices of the expanded message words used in each step, you will see that w_8 reappears in step 27, w_9 in 24 and w_{12} in 31. So this tunnel is able to affect only bits after the *point of verification*, step 23.

```
 0 A: --------------------------------
 1 A: --------------------------------
 2 A: ------------0-------0----0------
 3 A: 1-------0---1-------1----011----
 4 A: 1000100-01000000000000000010-1-1
 5 A: 0000001-011111111011110001 00-0-1
 6 A: 00000011111111101111100000100000
 7 A: 000000011--100010-0-010101000000
 8 A: 11111011---100000-1-111100111101
 9 A: 0111----000111111-01---001----00
10 A: 0010-0-0111-00011-00-0-011----10
11 A: 000---------10000001---10-------
12 A: 01----01----1111111----00---1---
13 A: 000---00----1011111----11---1---
14 A: -1100001--------10-------0000000
15 A: -01000-------------------000-000
16 A: -1------------0-----------------
17 A: --------------1-----------------
18 A: --------------0-----------------
19 A: --------------------------------
20 A: --------------------------------
21 A: --------------------------------
22 A: 0-------------------------------
23 A: 1-------------------------------
```

Figure 5.2: Sufficient conditions for the first block by Klima

4. The conditions for a_8, a_9, a_{10} only allow three bits (marked blue in Figure 5.2) to act like this. Therefore, 2^3 possible bit flips can be applied for this tunnel.

Figure 5.2 shows all sufficient conditions for the first block. The tunnels are marked in the following colors:

a_3 with red background

a_8 with blue background

a_9 with yellow background

a_{13} with green background

The time of finding MD5 collisions was reduced to just 31 seconds on a normal laptop.

5.4 Further improvements by Stevens

Stevens [Ste06] improves the attack algorithm for finding two-block collisions of the MD5 hash function. It uses the same differential path including the set of sufficient conditions which was published by Wang *et al.* . He uses a new algorithm to deterministically fulfill the conditions for the rotations of the differential in the first round of MD5. A new algorithm for the first block is presented. For optimizations, the set of conditions will be extended.

Types of conditions. Wang uses an extensive set of equations to describe the differential path. Stevens has his own notation where he describes conditions for all state registers a_i in for both message blocks. We can map the notations to the conditions used for the *nltool* in Table 5.1.

Table 5.1: Stevens two-block conditions types on bits of a_i and its mapping to the *nltool*

Stevens' notation	*nltool* equivalence
'.'	no restriction on a bit b of a_i, δ_g '-'
'0'	is the same as δ_g '0'
'1'	is the same as δ_g '1'
'^'	two-bit condition to the previous step in bit b: $a_{i-1,b} = a_{i,b}$
'!'	two-bit condition to the previous step in bit b: $a_{i-1,b} \neq a_{i,b}$
'I','J' and 'K'	two-bit conditions connecting multiple bits where each bits with 'I' and 'J' match each other. Bits with 'K' have to be the inverse of bits with 'I'

As can be seen, only single-bit conditions are subject to input for the *nltool*. The two-bit conditions are propagated automatically.

Details on added restrictions. He adds further restrictions on modular differences. These additional rules reduce the amount of probabilistic complexities in order to accelerate the algorithm. For example, the modular difference 2^{13} is not allowed to propagate after the 14^{th} bit:

	bits 31-28:
a_{10}	00**10** ....
a_9	01**11** ....
$a_{10} - a_9$	1010 ....

The condition therefore is that the bit on position 13 is rotate to position 30 and has to be zero. Moreover, conditions $a_{10,29} = a_{10,28} = a_{9,29} = 0$ and $a_{9,28} = 1$ apply.

Like these additional restrictions, many more are defined with the focus of stopping the propagation. As previously stated, carry bits make the conditions more complicated as they propagate differently. Lets take a look at the modular difference before the shift operation of -2^7 in step 14. The propagation has to stop at the 9^{th} bit. This has to be achieved by setting $a_{15,30} = \neg a_{14,30}$. Because of that, the one-bit at position 7 rotates to 29 in step 14. The following two bits in positions 8 and 9 are also one-bits. Their rotated counterpart, however, could be `1` or `0` depending on the existence of a negative carry from bits before:

no carry:

	bits 31-29:
a_{15}	0**0**1. ....
a_{14}	0**1**1. ....
$a_{15} - a_{14}$	110. ....

neg. carry:

	bits 31-29:
a_{15}	0**0**1. ....
a_{14}	0**1**1. ....
$a_{15} - a_{14}$	101. ....

In the same manner other propagations are stopped. This technique also holds for the second block.

Algorithm. The algorithm for the first block works as follows. Using simple message modification, a_0, ..., a_{15} **except** a_1 are generated. With these values, all message words except m_6 can be calculated. This creates a sparse result which is a starting point for the next non-deterministic steps. The first loop runs until a_{16}, ..., a_{20} are satisfying the conditions. a_{16} is chosen. This can be done by generating random values for each loop iteration. From a_{16}, w_1 can be calculated. This leads to calculating the missing a_1 and the affected message words. Calculate a_{17} to a_{21}. The last loop checks all satisfying values of a_8 and a_9 with $m_1 1$ unchanged and verifies every a to step 63. Note that the conditions on the IV for the next block also have to be checked. The second block works similar. It has to be noted that a_1 remains sparse instead of a_2 at the beginning. The loops also work in a probabilistic way. See algorithm 5 for all details.

Results. For the first block, he observed an average complexity of $2^{27.6}$ MD5 compressions. For both blocks the complexity is $2^{32.25}$.

5.5 New collision differential by Xie *et al.*

In 2008, Xie [XFL08] presented a new solution based on Wang's two-block-collision approach. The use of signed differences was introduced in order to improve the performance and reduce the complexity. Their work also aimed at creating a more understandable path than Wang's. Multi-message modification was also adapted for this new path.

As defined earlier, Xie clearly distinguishes between the three differential types Δ_X, Δ_M and Δ_S (see definitions 4.2.2, 4.2.3 and 4.2.5).

They use a new collision differential, which is different to Wang's:

$$\Delta M_0 = M'_0 - M_0 = (0, 0, 0, 0, 0, 0, -2^8, 0, 0, 2^{31}, 0, 0, 0, 0, 0, 2^{31})$$
$$\Delta M_1 = M'_1 - M_1 = (0, 0, 0, 0, 0, 0, 2^8, 0, 0, 2^{31}, 0, 0, 0, 0, 0, 2^{31})$$
$$\Delta H_1 = (2^{31} - 2^{23}, 2^{31} - 2^{23}, 2^{31} - 2^{23}, 2^{31} - 2^{23})$$

Their differential path starts very late in step 6. Therefore four message words $w_1, \ldots, w_4$ are completely undetermined. This fact improves the multi-message modification because

Algorithm 5 Stevens' two-block collision attack [Ste06]

INPUT: Conditions sets for first and second block
OUTPUT: Message pair (M, M') where $MD5(M) = MD5(M')$

Use condition sets for the first block.
loop
  Choose $a_0, a_1, \ldots, a_{15}$ fulfilling conditions.
  Calculate $w_0, w_6, \ldots, w_{15}$
  loop until $a_{16}, \ldots, a_{20}$ are fulfilling conditions
    Choose a_{16} fulfulling conditions
    Calculate w_1 from step 16
    Calculate $a_2, w_2, w_3, w_4, w_5, a_{17}, a_{18}, a_{19}, a_{20}$
  loop over all possible a_8, a_9 satisfying conditions where w_{11} remains unchanged
    Calculate $w_8, w_9, w_{10}, w_{12}, w_{13}$ and $a_{21}, \ldots, a_{63}$
    Check all conditions from step 21 to 63. If all hold, **exit loop**.
$M_0 \leftarrow w_0||w_1||\ldots||w_{15}$ and $M'_0 = M_0 + \Delta M_0$
Use condition sets for the second block and intermediate hash from the first block.
loop
  Choose $a_1, \ldots, a_{15}$ fulfilling conditions.
  Calculate $w_5, \ldots, w_{15}$
  loop until $a_{16}, \ldots, a_{20}$ are fulfilling conditions
    Choose a_0 fulfulling conditions
    Calculate $w_0, \ldots, w_4$ and $a_{16}, \ldots, a_{20}$
  loop over all possible a_8, a_9 satisfying conditions where w_{11} remains unchanged
    Calculate $w_8, w_9, w_{10}, w_{12}, w_{13}$ and $a_{21}, \ldots, a_{63}$
    Check all conditions from step 21 to 63. If all hold, **exit loop**.
$M_1 \leftarrow w_0||w_1||\ldots||w_{15}$ and $M'_1 = M_1 + \Delta M_1$
return colliding message pair $(M, M') = (M_0||M_1, M'_0||M'_1)$.

these free message words can be satisfied easily in the second round. More details on the algorithm can be found in [XFL08].

For the first block, 36 conditions are defined that have to be met in a probabilistic manner. For the second block, the amount of conditions is 32. The overall complexities do not exceed 2^{36} and 2^{32} MD5 operations.

5.6 Creating our own collisions with the *nltool*

This section deals with creating own two-block collisions with the *nltool*. First of all, we create a local collision in the first round and show how to use this for the principle of tunneling. The next step is to embed Stevens' two-block collision attack into the *nltool*. All necessary modifications are explained.

5.6.1 Placing tunnels

General idea. We will try to build our own tunnel as given in Figure 5.3. We start by introducing a positive difference at an arbitrary position on some message word w_i. In this example, this difference should be canceled out immediately. Two negative differences in following message words are necessary. Then we can discover the principle of tunneling. When choosing the message words wisely, they reappear in the second round at a very late step. For a better understanding, the figure shows which message words are used in the second round.

```
-2 A: --------------------------------
-1 A: --------------------------------
 0 A: -------------------------------- W: --------------------------------
 1 A: -------------------------------- W: --------------------------------
 2 A: -------------------------------- W: --------------------------------
 3 A: -------------------------------- W: --------------------------------
 4 A: -------------------------------- W: --------------------------------
 5 A: -------------------------------- W: --------------------------------
 6 A: -------------------------------- W: --------------------------------
 7 A: -------------------------------- W: --------------------------------
 8 A: ------------------------n------- W: -------------------------------n   <- introduced positive difference
 9 A: ------------------------0------- W: ----u---------------------------   <- negative difference to cancel out a8
10 A: ------------------------1------- W: --------------------------------
11 A: -------------------------------- W: --------------------------------
12 A: -------------------------------- W: ------------------------u-------   <- 2nd neg. difference to cancel out a8
13 A: -------------------------------- W: --------------------------------
14 A: -------------------------------- W: --------------------------------
15 A: -------------------------------- W: --------------------------------
16 A: -------------------------------- W: -------------------------------- [ 1]
17 A: -------------------------------- W: -------------------------------- [ 6]
18 A: -------------------------------- W: -------------------------------- [11]
19 A: -------------------------------- W: -------------------------------- [ 0]   Length ℓ of tunnel
20 A: -------------------------------- W: -------------------------------- [ 5]   after first round
21 A: -------------------------------- W: -------------------------------- [10]
22 A: -------------------------------- W: -------------------------------- [15]
23 A: -------------------------------- W: -------------------------------- [ 4]
24 A: -------------------------------u W: ----u--------------------------- [ 9]
```

Figure 5.3: A sample tunnel with $\ell = 8$ as an *nltool* characteristic

Exact definition of this tunnel. The following section shows an example tunnel structure for a single bit difference to be canceled out. As a demonstration, three approaches are given on how to describe the tunnel:

1. Show all the information compressed in an `nltool` output. Figure 5.3 shows this output.

2. Use a block diagram and follow positive and negative differences. See Figure 5.4 for details.

3. Use formulae to define sufficient conditions for this pattern:

Formulae. In the following, differences are described as $\delta(x) = [+y]$. This means, that the generalized difference $\delta_G x$ has a positive difference 'n' in position y. $\delta(x) = [-y]$ denotes a negative difference 'u' at position y.

The input is a positive message difference at position x at step i $(0 \leq x \leq 31)$:

$$\delta(w_i) = \delta[+x]$$

The propagation therefore is:

$$\delta(a_i) = \delta[+x] \ggg s_i = \delta[+((x + s_i) \mod 32)]$$

It is important to let the boolean function F (which is f in the first round) block the input difference. Table 4.2 shows the exact behaviour of f and its constraints. Because of these conditions, extra conditions on the values of x, y, z occur.

1. Step $i+1$: $(f(\mathtt{n}, -, -) = -) \Rightarrow y = z$

2. Step $i+2$: $(f(-, \mathtt{n}, -) = -) \Rightarrow x = 0$

3. Step $i+3$: $(f(-, -, \mathtt{n}) = -) \Rightarrow x = 1$

The two single-bit conditions determine that bit x in a_{i+1} has to be 0 and bit x in a_{i+2} has to be 1 which can be observed in Figure 5.3.

The conditions necessary for the message word are

1. $\delta(w_{i+1}) = \delta[-((x + s_i) \mod 32)] \lll s_{i+1}$

2. $\delta(w_{i+4}) = \delta[-((x + s_i) \mod 32)]$

Choosing a starting point. In round 2 of MD5, the message expansion shuffles the message word indices and therefore different lengths of this pattern can be found. The indices for the message words $w_{16}, ..., w_{24}$ are (1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12). Table 5.2 shows all possible options of introducing the difference in the message words $w_0, \ldots, w_{11}$. The longest tunnel goes up to step 24 and is shown in Figure 5.3. The longest pattern can be constructed with a bit difference starting in w_8. The message word w_9 is used at step 24.

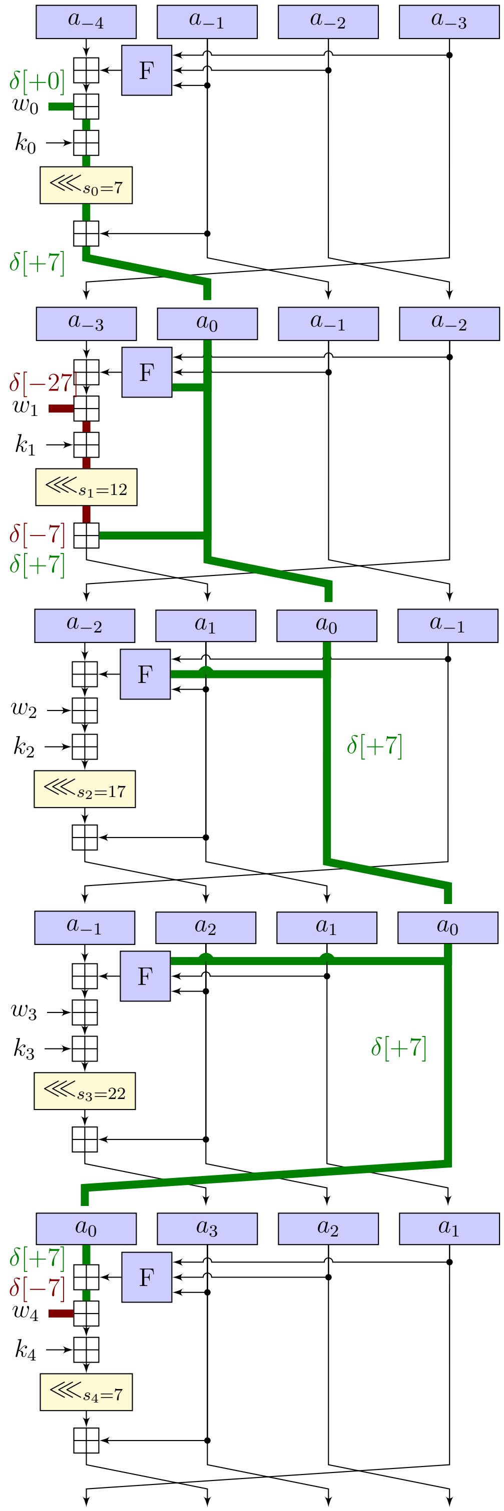


Figure 5.4: The pattern for creating a local collision for an introduced difference at $a_{0,0}$

Table 5.2: Different starting points for differences and corresponding tunnel lengths

i_{start}	i_{negpos}	$i_{negpos+4}$	Affected w_i for second round	Length ℓ
0	27	7	0,1,4	0
1	27	12	1,2,5	0
2	27	17	2,3,6	1
3	15	22	3,4,7	7
4	27	7	4,5,8	5
5	27	12	5,6,9	1
6	27	7	6,7,10	1
7	15	22	7,8,11	2
8	27	7	8,9,12	8
9	27	12	9,10,13	5
10	27	17	10,11,14	2
11	15	22	11,12,15	2

i_{start} ... Step i where $\delta(w_i) = \delta[+0]$
i_{negpos} ... Position of neg. δ in w_{i+1}
$i_{negpos+4}$ ... Position of neg. δ in w_{i+4}

5.6.2 Constructing our own two-block collisions

The goal now is to let the *nltool* create such collisions. Because of the well-documented approach, Stevens' two-block collision attack is implemented (see algorithm 5). In this case, the complete differential path including all sufficient conditions is given, so only a message search is performed. Now it has to be determined which parts of this attack could be automatically performed by the *nltool*.

Bit-wise vs. Word-wise representations

As explained before in Section 4.4, the *nltool* uses slices for representing each bit (*i.e.* the generalized condition). This is necessary to provide exact propagations of these condition types. However, Stevens' algorithm only needs a small subset of conditions (see Table 5.1). For this reason, we will use the *nltool* to parse the differential characteristics and to derive the two-bit conditions. After that, the internal state registers are represented in word-wise data structures for a much faster calculation.

Implementation Details

Single-bit conditions. Each bit for the internal state registers a_i and message words w_i are represented as a pair of 32-bit words. The first word represents the 1-bits, the second one the 0-bits. Each bit therefore has the possibility of 4 possible values. Table 5.3 shows these possibilities along with the output of the *nltool* as comparison.

Table 5.3: Internal data structure for own two-block-collisions

nltool representation of register x	`-10#`
Bitmask x' (1-bits)	`0101`
Bitmask x'' (0-bits)	`0011`

Many bit operations in MD5 can be performed easily and very efficiently with these bit masks. If none of the bits are undetermined ($x' \oplus x'' = 111\ldots111$), x' represents x and can be used for all normal calculations. For choosing a random value x to satisfy the conditions in x' and x'', simple AND and OR operations can be applied.

Two-bit conditions. In Stevens' two-block collision, two types of conditions are possible: either two bits are equal or have to be unequal. For each bit pair, its step and bit position has to be saved. In conclusion, one two-bit condition needs four indices and a type to differentiate. The two-bit conditions are parsed and stored at the beginning. After that they are used to calculate forward propagations. Each target bit is checked and inconsistencies can be recognized.

Input and output. When parsing the characteristic with the *nltool* is successful, the data structure is initialized and the single-bit conditions are set to the internal bit masks.

Two-bit conditions are only copied to the internal list and applied afterwards. After the process is finished, the message words w_i are used to create a new *nltool* characteristic which can be printed out.

Performance and Results. With these features in mind, the implementation was done for the first block of Stevens' two-block-collision. The result can be found in Figure 5.5. The runtime for the first block of Stevens two-block-collision is about 15 seconds on a laptop. The complexity for the first block is $2^{22.82} \cdot 2^{3.9} = 2^{26.72}$. Stevens measured a complexity of $2^{27.6}$ for the first block.

```
-4 A: 01100111010001010010001100000001
-3 A: 00010000000110010010101000111011 0
-2 A: 10011000101110101101110011111110
-1 A: 11101111111001101101010111000100 1
 0 A: 11110000110100111000010111110000 W: 11110110100101110110011100111101
 1 A: 11100110100000101100010001111100 W: 00000110111010110111010110011001
 2 A: 01101000110001110110010010111110 W: 10101001011101100100001101011000
 3 A: 11011101010011110110110010110101 W: 01111110000000011001110111011111
 4 A: 10001001011unnnnnnnnnnnnnn101111 W: u0100100010000001100011010001011
 5 A: n0000011n1111111101111000u001011 W: 10101001111101011011000110100000
 6 A: nnnnnnuuu11111101111unnnnnunnnnn W: 10101111100000101001010010101111
 7 A: 00000001u11unnnun10101010100000n W: 01100011010111111100101000110010
 8 A: u11110110001000001111111unn1111nu W: 00000101000001010101101010000001
 9 A: n1110110001111111nu000001110000 W: 10000010011101000010100111111011
10 A: nn100000100100011100000011000010 W: 10000000010110101010110000100111
11 A: n00101000100unnnnnn1110un1011011 W: 01010011000100 10n01110110101001 1
12 A: n10000nu11001111111001100011111 0 W: 11111101100000101011111011010000
13 A: n0010100001110111110010111111100 W: 01100111101111001101111111100010
14 A: n0100101111100000u00010001001n010 W: u0001001111101000000100001000011
15 A: n1u11101000001110111111010101101 W: 11111001101011000110000100100100
16 A: n0010000101001010011101110011011
17 A: n10111011111111101001000101010100
18 A: n0101001110100n11101000010010001
19 A: n0010000011011001000001110100100
20 A: n0110111011001000110011101011100
21 A: n1001010100011100110101110000000
22 A: 01101010100100011111010001001011
23 A: 10010110100000010011001010000011 0
24 A: 01100000001001100111101100010010 1
25 A: 11011001001011100011001100111100
26 A: 10101101111000111110011010101001
27 A: 10010100101111011000101011001001
28 A: 01110000000101110001111010110110 1
29 A: 00010100101111000010100011001100
30 A: 01010001001001000110010000101000
31 A: 11111101110110111010011000010011
32 A: 00100011111111001101101010000000
33 A: 11000011001001110010010000000001
34 A: n0011100010101100101001011010101
35 A: n1110111100101110111010000011110
36 A: n0000000011010110100011010010101
37 A: u0001001001111111101111111000110 0
38 A: u1011110101011100101010011000001
39 A: n0100101001101011010101100010110
40 A: n1011011101100100000001001001011 0
41 A: n1101111000000100010010011101011
42 A: n0011101100100100110100000001111
43 A: u0111011100100111001100101100001
44 A: u1001011100000000110111100101100 1
45 A: u1011000111001111001110001100000
46 A: u1111010011010110111110000110110
47 A: u0010000111101001101110101101110
48 A: u0000111001011111100011000101110
49 A: n1111101001100100111000010001000
50 A: u0101011100111100000001001000110 0
51 A: n0111111001101011101000111011011
52 A: u1011011101001000101111100011010
53 A: n0011111110010101100010011000110
54 A: u1001011011000001101110011110001
55 A: n1111010011110000110011111001100
56 A: u1010101110000111101010101101100
57 A: n0000100001110011000100000001001
58 A: u0000101001111110011111100111001
59 A: u0100101011110101010011110111011
60 A: u0101011111100110111011110110110
61 A: u11011n0000001010111100011000001 0
62 A: u10111n1111010010100100100000110
63 A: u01000n0011100011111010010100001
```

Inner collision

Figure 5.5: First block of 2-block-collision

Chapter 6

Single Block Collisions

Until 2010, all collision attacks on MD5 were created using a pair of two message blocks. In 2010, Xie *et al.* [XF10] were able to create a collision using just a single message block. This means, that only one call for the compression function of MD5 is necessary. Finding a suitable differential and a computationally feasible solution is much harder than for two-block collisions since all differences have to be canceled out in the first block. Previous attacks created a near collision for the first block and then removed the remaining differences in the second one. The complexity for these attacks is lower than for single-block collisions. About one year ago, Stevens [Ste12a] was able to create another single-block collision and provided more insights into his attack. In 2013, Xie *et al.* [XLF13] published details on their original single-block attack as well as improved differentials. Unfortunately, only partial differential paths were given. The only full one by Xie *et al.* can be extracted from their solution.

In this chapter we will analyse Xie's and Stevens' attacks and compare their differential paths. Moreover, Stevens' attack will be embedded in the *nltool* and a more detailed analysis is made on runtimes and path probabilities. We will run his attack and try find new single-block collisions with his approach. After that we use the best partial path by Xie *et al.* and derive a full one using the nltool. Finally, we will use the nltool to create a partial solution with our custom built differential.

6.1 First result by Xie *et al.*

In 2010, Xie *et al.* [XF10] published a colliding message pair with only 512-bits each. This means, that only a single message block is necessary to create this collision. They use the following message differences:

$$\Delta M_0 = M_0' - M_0 = (0, 0, 0, 0, 0, 2^{10}, 0, 0, 0, 0, 2^{31}, 0, 0, 0, 0, 0)$$

A challenge was also called for finding a different single-block collision with a reward of 10,000 USD. Details on the algorithm were not described due to "security reasons". The full path of their differential is shown in Figure 6.1.

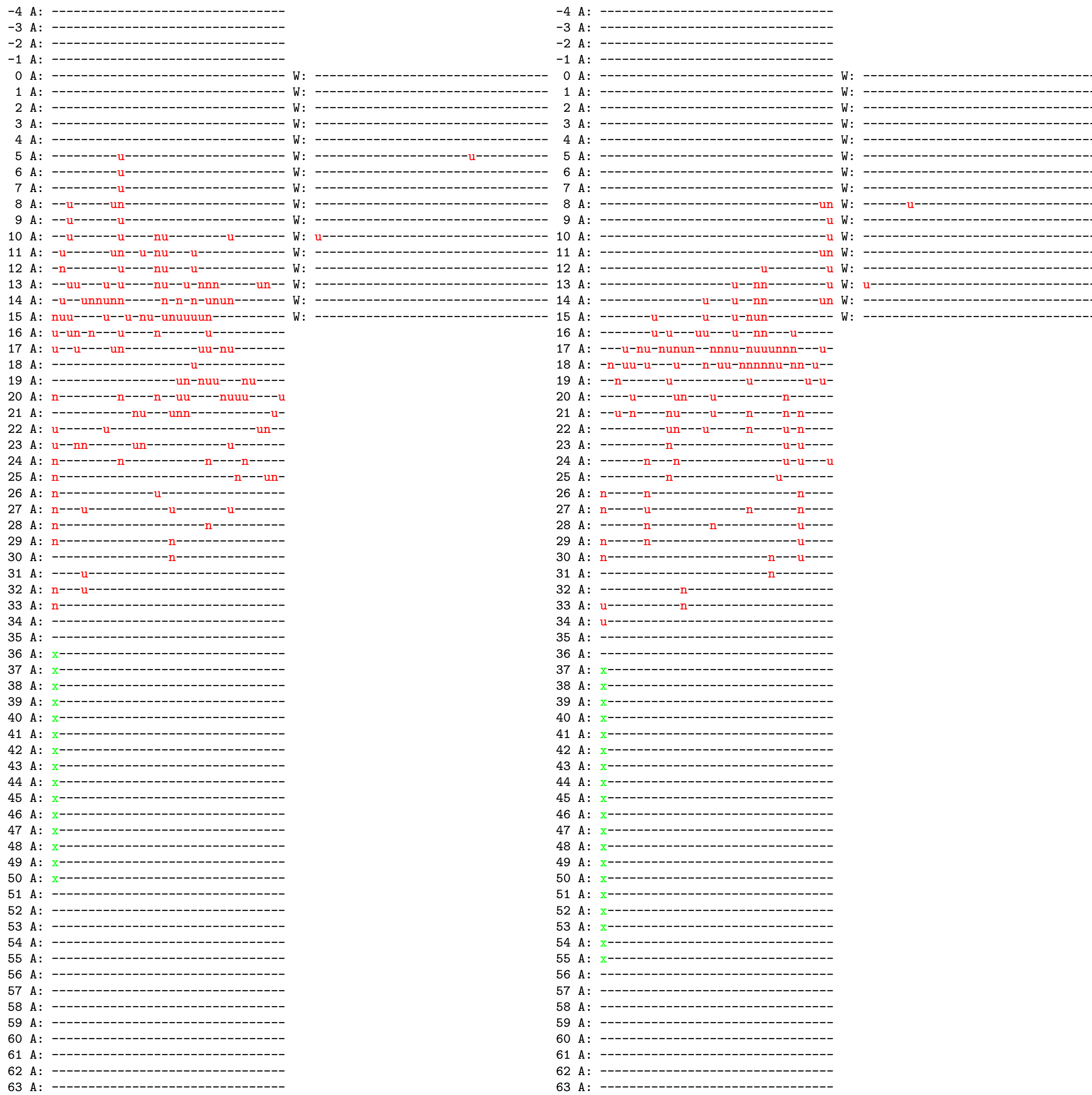


Figure 6.1: Single-block collision differential conditions by Xie *et al.* [XF10] and Stevens [Ste12a]

Table 6.1: Single-block collision message pairs by Xie *et al.* [Ste12a]

M_0	0x6165300e,0x87a79a55,0xf7c60bd0,0x34febd0b,0x6503cf04,0x854f709e, 0xfb0fc034,0x874c9c65,0x2f94cc40,0x15a12deb,0x5c15f4a3,0x490786bb, 0x6d658673,0xa4341f7d,0x8fd75920,0xefd18d5a
M'_0	0x6165300e,0x87a79a55,0xf7c60bd0,0x34febd0b,0x6503cf04,0x854f749e, 0xfb0fc034,0x874c9c65,0x2f94cc40,0x15a12deb,0xdc15f4a3,0x490786bb, 0x6d658673,0xa4341f7d,0x8fd75920,0xefd18d5a
MD5	0xf999c8c9,0xf7939ab6,0x84f3c481,0x1457cb23

6.2 Stevens' response to this challenge

Stevens was the first one who successfully published a different result for a single block collision. [Ste12a]. This section does an in-depth analysis of his attack. He used the following message differences:

$$\Delta M_0 = M'_0 - M_0 = (0, 0, 0, 0, 0, 0, 0, 0, 2^{25}, 0, 0, 0, 0, 2^{31}, 0, 0)$$

Table 6.2: Single-block collision message pairs by Stevens [Ste12a]

M_0	0xff68c94d,0x205ce30e,0x77d47295,0x8715727b,0xb2a76fd3,0xb756dc1b, 0x78c03d4a,0x18957b3e,0x00a2bfaf,0xf34b28a8,0x554b8e6e,0x75425fb3, 0x6749d893,0x55d1a06d,0xfb60835d,0xa2fe075f
M'_0	0xff68c94d,0x205ce30e,0x77d47295,0x8715727b,0xb2a76fd3,0xb756dc1b, 0x78c03d4a,0x18957b3e,0x02a2bfaf,0xf34b28a8,0x554b8e6e,0x75425fb3, 0x6749d893,0xd5d1a06d,0xfb60835d,0xa2fe075f
MD5	0x3a3ee800, 0x9d58b51c, 0xb025b4fe, 0xc9219195

Stevens' attack uses the message differences w_8 at bit 25 and w_{13} at bit 31. They were chosen because they have similar properties like the differences used by Xie's single-block collision attack. These differences brought up a partial differential path. Then he used his differential path construction algorithm from an earlier work [SLW07] to create a full differential path (see Figure 6.1). The amount of bit conditions for the first round of MD5 were kept very low. In particular, only a small number of conditions are set on a_1, a_7, a_8, a_{11} and a_{12}. Moreover, all conditions from a_{13} to a_{21} can be fulfilled easily. These paths are only possible when the differential has a complete zero difference in its chaining input value. Only a single-block collision attack with an identical prefix can have this property. Using these condition sets he used a new algorithm for finding collisions.

The algorithm is described in algorithm 7. Figure 6.2 shows an overview of the main parts. For better understanding, he splits it into four parts. Starting with steps 13 to 20, a_i can be set randomly satisfying all necessary conditions. When taking a look at the indices for the message words in the second round, those can be easily determined. In the precomputation phase, a list of tuples values for $a_1, \ldots, a_6, a_{12}$ satisfying conditions in

2. Steps 4-7

3. Steps 8-11

1. Steps 2-3

4. Steps 12-24

Tunnels T_4, T_9, T_{14}

```
-4 A: 01100111010001010010001100000001
-3 A: 00010000001100100101010001110110
-2 A: 10011000101110101101110011111110
-1 A: 11101111110011011010101110001001
 0 A: -101100---------------------0--- W: ---1111-0110100-------0------10-
 1 A: -------------------------------- W: ------------------1-0-----------
 2 A: 010-0-00-0001---01-1---000-000-- W: --------------------------------
 3 A: 00000000000000000000000000000000 W: --------------------------------
 4 A: 11101011011110001101000111011100 W: --------------------------------
 5 A: ---1-1--1----111--1-111---1---11 W: --------------1----------0------
 6 A: -------------------------------0 W: 1---------------101111100--10110
 7 A: -------------------------------0 W: --------------------------------
 8 A: 0000000000000000000000-00-0000un W: ------u-------------------------
 9 A: 0000000000000000000000-00-00000u W: --------------------------------
10 A: 1111111111111111111111011-11111u W: -------------------------------0
11 A: ------------------1--00-------un W: ----0-----------010-11111011-01-
12 A: ------------------1-00u-------0u W: --------------------------------
13 A: 010-0-00-0001---01u10nn000-0001u W: u-------------------------------
14 A: 000-0-00-00001u-00u00nn0001000un W: --------------------------------
15 A: 0001010u100000u100u0nun000100001 W: -------------------------------1
16 A: -0--10-u0u000uu010u10nn101u0-000 W: [ 1]
17 A: -01u1nu1nunun01nnnu0nuuunnn010u- W: [ 6]
18 A: -n0uu-u100u011n1uu1nnnnnu1nn0u0- W: [11]
19 A: --n10-0-1u010-11011-u1000011u-u- W: [ 0]
20 A: --11u-1--1un--1u11--00100n10---- W: [ 5]
21 A: --u-n----nu---1u----n----n-n---- W: [10]
22 A: ----1-0--un---u0----n----u-n---0 W: [15]
23 A: ----1-0--n0----0----0---1u-u---1 W: [ 4]
24 A: 0-----n--0n---------1---1u-u---u W: [ 9]
25 A: ------0--n----------1---u1-1---- W: [14]
26 A: n-----n--------0----1----1-n---- W: [ 3]
27 A: n-----u--------1----n------n---- W: [ 8]
28 A: 0-----n--------n----1--0---u---- W: [13]
29 A: n-----n--------0----0------u---- W: [ 2]
30 A: n-----1--------1-------n---u---- W: [ 7]
31 A: ------0----------------n-------- W: [12]
32 A: -----------n--------------------
33 A: u----------n--------------------
34 A: u-------------------------------
35 A: --------------------------------
36 A: --------------------------------
37 A: x-------------------------------
38 A: x-------------------------------
39 A: x-------------------------------
40 A: x-------------------------------
41 A: x-------------------------------
42 A: x-------------------------------
43 A: x-------------------------------
44 A: x-------------------------------
45 A: x-------------------------------
46 A: x-------------------------------
47 A: x-------------------------------
48 A: x-------------------------------
49 A: x-------------------------------
50 A: x-------------------------------
51 A: x-------------------------------
52 A: x-------------------------------
53 A: x-------------------------------
54 A: x-------------------------------
55 A: x-------------------------------
56 A: 1-------------------------------
57 A: 0-------------------------------
58 A: --------------------------------
59 A: --------------------------------
60 A: --------------------------------
61 A: --------------------------------
62 A: --------------------------------
63 A: --------------------------------
```

Step numbers refer to Algorithm 7.

Figure 6.2: High level overview of Stevens' [Ste12a] collision attack (full differential).

steps 1, 5, 6 and 16 is generated. The index of this list is the values of a_6 and a_{12}. In the main loop, all values satisfying conditions in step 7 to 11 are iterated. The lookup-table is then used for resolving indirect conditions between steps 6 to 7 and 11 to 12. From that moment, all message words $w_0, \ldots, w_{15}$ are resolved. The algorithm has progressed to step 22. Now, tunnels are used to go further and modify message words in order to hold bit conditions. Three distinct tunnels, later referred to as T_4, T_9 and T_{14} are used to find values up to step 28 fulfilling all conditions. From this point on, no further message modification is possible. All remaining steps are calculated and a collision check is made. All conditions from this point have to be fulfilled in a probabilistic manner.

Algorithm 6 Steven's single block collision algorithm [Ste12a]

INPUT: $IV = (IV_0, IV_1, IV_2, IV_3)$, where $IV_i = \{0,1\}^{32}$ and bitconditions from Figure 6.1

OUTPUT: Message pair (M, M'), where $f(IV, M) = f(IV, M')$ and $M, M' = \{0,1\}^{512}$

1 Set IV to $(a_{-4}, a_{-3}, a_{-2}, a_{-1})$.
2 Create random values for $(a_{13}, \ldots, a_{20})$ satisfying conditions
3 Calculate $m_6, m_{11}, m_0, m_5, a_0$
4 **for all** (a_2, a_3, a_4, a_5) satisfying conditions **do** $\triangleright$ create lookup table
5   Calculate a_6, a_1, m_1, a_{12}
6   **if** (a_6, a_1, a_{12}) satisfy conditions **then**
7   Append tuple $(a_6 \wedge b_7, a_{12} \wedge b_{12}), (a_1, a_2, a_5, a_6, a_{12})$ to lookup table
8 **for all** $(a_8, a_9, a_{10}, a_{11})$ satsfying conditions **do** $\triangleright$ main loop
9   Calculate a_7
10   **if** a_7 satisfies conditions **then**
11   **for all** $(a_1, a_2, , a_5, a_6, a_{12})$ at index $(a_6 \wedge b_7, a_{12} \wedge b_{12})$ in lookup table **do**
12   Calculate all message words $w_0, \ldots, w_{15}$ and a_{21}, a_{22}
13   **if** a_{21}, a_{22} satisfy conditions **then**
14   **for all** values of tunnel T_4 **do**
15   Calculate w_4, a_{23}
16   **if** a_{23} satisfies conditions **then**
17   **for all** values of tunnel T_9 **do**
18   Calculate w_9, a_{24}
19   **if** a_{24} satisfies conditions **then**
20   **for all** values of tunnel T_{14} **do**
21   Calculate $w_{14}, w_3, w_8, w_{13}, a_{25}, a_{26}, a_{27}, a_{28}$
22   **if** $a_{25}, a_{26}, a_{27}, a_{28}$ satisfy conditions **then**
23   Calculate M from $w_0, \ldots, w_{15}$ and M'
24   **if** $f(IV, M) = f(IV, M')$ **then return** (M, M')
25 Start again from step 1.

6.2.1 Types of conditions

In Stevens' notation for the single-block attack, he uses a different set of condition types compared to his two-block collision attack (see Table 5.1).

Table 6.3: Stevens single-block conditions types on bits of a_i and its mapping to the *nltool*

Stevens' notation	*nltool* equivalence
'`.`'	no restriction on a bit b of a_i, δ_G '`-`'
'`0`'	is the same as δ_G '`0`'
'`1`'	is the same as δ_G '`1`'
'`+`'	is the same as δ_G '`n`'
'`-`'	is the same as δ_G '`u`'
'`^`'	two-bit condition to the previous step in bit b: $a_{i-1,b} = a_{i,b}$
'`!`'	two-bit condition to the previous step in bit b: $a_{i-1,b} \neq a_{i,b}$

6.2.2 Tunnels

Tunnel T_4 affects 13 bits in a_3. All possible values of those bits are iterated. The involved registers based on the changes of a_3 are recalculated. Figure 6.3 shows all related registers and bits.

$$w_i \leftarrow (a_i - a_{i-1}) \ggg s_i - a_{i-1} - f(a_{i-1}, a_{i-2}, a_{i-3}) - k_i \qquad i = (3, 4, 7)$$
$$a_{23} \leftarrow (g(a_{21}, a_{22}, a_{23}) + w_4 + k_{23}) \lll s_{23} + a_{22}$$

```
 3 A: 00000000000000000000000000000000 W: --------------------------------
 4 A: 11101011011110001101000111011100 W: --------------------------------
 5 A: ---1-1--1----111--1-111---1---11 W: --------------1----------0------
 6 A: -------------------------------0 W: 1---------------101111100--10110
 7 A: -------------------------------0 W: --------------------------------
[...]
23 A: ----1-0--n0----0----0---1u-u---1 W: --------------------------------[ 4]
```

Figure 6.3: Tunnel T_4 of Stevens' single-block collision

The same effects can be used for tunnel T_9 (Figure 6.4) which flips 30 bits in a_8. w_8, w_9, w_{12} and a_{24} are affected and are calculated the same way is in T_4.

Tunnel T_{14} (Figure 6.5) is more complex because steps 13 and 2 are involved. Like before, three message words are influenced. w_{13} and w_{14} can be calculated. The message word used in step 17 is w_6. In this case, we have to deal with feedback. w_6 is used in step 6 as well. a_2 can propagate to this message word. Therefore, this tunnel also has to iterate over the same bits in a_2.

```
 8 A: 0000000000000000000000-00-0000un W: ------u-------------------------
 9 A: 0000000000000000000000-00-00000u W: --------------------------------
10 A: 1111111111111111111111011-11111u W: -------------------------------0
11 A: ------------------1--00-------un W: ----0-----------010-11111011-01-
12 A: ------------------1-00u-------0u W: --------------------------------
[...]
24 A: 0-----n--0n---------1---1u-u---u W: --------------------------------[ 9]
```

Figure 6.4: Tunnel T_9 of Stevens' single-block collision

```
 2 A: 010-0-00-0001---01-1---000-000-- W: --------------------------------
 3 A: 00000000000000000000000000000000 W: --------------------------------
 4 A: 11101011011110001101000111011100 W: --------------------------------
 5 A: ---1-1--1----111--1-111---1---11 W: --------------1----------0------
 6 A: -------------------------------0 W: 1---------------101111100--10110
[...]
13 A: 010-0-00-0001---01u10nn000-0001u W: u-------------------------------
14 A: 000-0-00-00001u-00u00nn0001000un W: --------------------------------
15 A: 0001010u100000u100u0nun000100001 W: -------------------------------1
16 A: -0--10-u0u000uu010u10nn101u0-000 W: ------------------1-0----------- [ 1]
17 A: -01u1nu1nunun01nnnu0nuuunnn010u- W: 1---------------101111100--10110[ 6]
[...]
25 A: ------0--n----------1---u1-1---- W: --------------------------------[14]
```

Figure 6.5: Tunnel T_{14} of Stevens' single-block collision

6.2.3 Complexity and results

Stevens measured the time of reaching step 28 with meeting all the conditions necessary from a_{-4} to a_{28}. This experimental calculation lead to a complexity of about $2^{15.96}$ MD5 compression operations. The probability from this step to a real collision is about $2^{-33.85}$. This was also experimentally verified. Accordingly, the overall complexity is $2^{49.81}$ MD5 compressions. The measurements were made on an Intel Core 2 Q9550 CPU. Stevens and his team estimated the runtime to about five weeks. Fortunately, the collision was found two weeks earlier.

6.3 Comparison of Xie's and Stevens' differential path

The differential path of Xie *et al.* (Figure 6.1) starts earlier with many conditions starting at step 12 continuing to 23. Their path has 154 conditions on positive or negative differences. On the contrary, Stevens' path (Figure 6.1) starts later, the major amount of conditions can be found between steps 17 and 20. The amount of positive or negative differences conditions is slightly lower (145). Xie's path is able to stop early with the last difference at step 50. Stevens' characteristic ends five steps later. Both share nearly the same path differences at the most significant bit. Unfortunately, no further information (*i.e.* message modification steps) can be deducted from Xie's differential path. Because of the number of conditions, the probability of Stevens' path seems slightly higher.

6.4 Fast collision attack by Xie *et al.*

In 2013, Xie *et al.* [XLF13] published details about their previously called competition in 2010 [XF10]. In their work, they presented a new method of choosing the best input differences for creating colliding messages in MD5. Two classes of sufficient conditions were defined in their work. They used strong conditions and weak conditions. This decision was made by the necessary effort to satisfy the conditions. Moreover, a proof was made on the existence of strong conditions only in steps 0 to 23. They used their findings to select ideal message differences. An implementation of a two-block collision was done with 2^{18} MD5 compressions. For single-block collisions, they proposed an attack with only 2^{41} MD5 compressions. This section deals with the details of condition strength and selecting proper message differences. Focus will be laid on single-block collisions and details on their two-block collision are omitted.

6.4.1 Weaknesses and Condition Strengths

Xie *et al.* identified two distinct shortcomings of the MD5 hash function.

Message Expansion. Message modification is not applicable to the internal state registers after a_{25}. For steps $16, \ldots, 25$, the following message words are necessary:
$w_0, w_1, w_4, w_5, w_6, w_9, w_{10}, w_{11}, w_{14}, w_{15}$. After step 25, only the remaining message words can be used and further message modification fails. Due to this fact, they defined the sufficient conditions until step 25 as weak. The remaining conditions are strong.

Difference Inheritence. In the steps $32, \ldots, 47$, the boolean function `XOR` is used. They proved that the MSB path can hold with four consecutive differences with a probability of 1 if no message word difference exists.

6.4.2 Single-Block Collisions

Xie *et al.* presented three message differences including their estimated complexity for a single-block collision attack. They even used the difference used by Stevens [Ste12a] and improved it. Partial differential paths for steps $22, \ldots, 63$ were given for each variant. Table 6.4 gives an overview.

Table 6.4: Xie *et al.* message difference variants and their complexities for a collision attack [XLF13]

Message Difference	Complexity in MD5 compressions
$\Delta w_5 = 2^{10}, \Delta w_{10} = 2^{31}$	2^{47} as in [XF10], improved to 2^{42}
$\Delta w_5 = 2^{10}, \Delta w_{10} = 2^{31}, \Delta w_{14} = 2^{31}$	2^{41}
$\Delta w_7 = 2^{31}, \Delta w_8 = 2^{25}, \Delta w_{13} = 2^{31}$	2^{46}

Errors in their differentials. We analysed the partial differential path for the message difference ($\Delta w_5 = 2^{10}, \Delta w_{10} = 2^{31}, \Delta w_{14} = 2^{31}$) with the nltool. Unfortunately, the given path had inconsistencies. Moreover, the modular differences were not matching the signed differences in their work. We corrected the signs and the resulting partial differential path can be found in Figure 6.6. Table 6.5 shows the corrected signed differences. The other differential paths also were not consistent and some manual correction would have been necessary. However, we only laid focus on the differential with the lowest complexity for the collision attack.

Table 6.5: Corrections for partial differential path ($\Delta w_5 = 2^{10}, \Delta w_{10} = 2^{31}, \Delta w_{14} = 2^{31}$) by Xie *et al.* [XLF13]

Step	Given Δ_S by Xie *et al.*	Corrected Δ_S
24	$\Delta_S[-2, -5, -10, -22, 31]$	$\Delta_S[-2, -5, -10, 22, -31]$
25	$\Delta_S[1, -6, 18, 31]$	$\Delta_S[1, -6, -18, -31]$
26	$\Delta_S[31]$	$\Delta_S[-31]$
27	$\Delta_S[7, 15, 31]$	$\Delta_S[7, 15, -31]$
28	$\Delta_S[-10, 27, 31]$	$\Delta_S[-10, 37, -31]$
29	$\Delta_S[-15, 31]$	$\Delta_S[-15, -31]$
30	$\Delta_S[-15, 31]$	$\Delta_S[-15, -31]$

Response to Stevens' attack. Stevens' attack [Ste12a] uses two differing bits in w_8 and w_{13} resulting in a complexity of 2^{50}. Xie *et al.* were able to lower this complexity by introducing a difference in w_7. Moreover, they claimed that Stevens' collision attack is not a completely new one but is derived from their original attack. On top of that, Stevens attack featured a higher complexity than the original single-block attack by Xie *et al.* Hence, the results by Stevens did not completely satisfy their challenge. Therefore he got only half of the awarded money.

Details of the algorithm. Unfortunately, no further details about any implementation of the single-block collision attacks were given by Xie *et al.* . The details on the calculation of the complexities for the different paths were also omitted because they could be derived from their work on the complexities of the two-block collision attack.

In conclusion, only the partial differential paths could be used for further analysis. The collision attack algorithm in their work is only applicable to two-block collisions. Moreover, no additional resulting message pairs were given to check the validity of the paths.

6.5 Constructing single-block attacks with the *nltool*

This section describes all means that were necessary to embed Stevens' attack into the toolbox. After that, our own collision searches will be run and its results documented. A probabilistic analysis will be made on Stevens' path and compared to actual runtimes.

6.5.1 Using a custom search configuration

The first attempt on reconstructing Stevens' attack with the *nltool* was by using a custom search configuration and do the attack with the *nltool* completely automatically. With this configuration we can override parameters for the default search algorithm (see Section 4.4.1). The following configuration was created:

Algorithm 7 Adapted search configuration of *nltool* for Stevens' single block collision

1 Guess words $a_{13}, \ldots, a_{20}$ with the following behaviour:
   Set all bits with a generalized difference of '`-`' to '`0`' or '`1`' randomly.
   The probability is 2^{-1} for both selections. Do a complete check after setting all bits.
2 Guess words $a_2, \ldots, a_5$ with the same settings as above.
3 Guess words $a_8, \ldots, a_{11}$ with the same settings as above.

Results. The integrated search algorithm of the *nltool* was run for several limited steps. Table 6.6 shows the runtime and the complexity in MD5 operations.

Table 6.6: Complexities of modified *nltool* search configuration for Stevens' single-block collision

Step	Average runtime	Complexity
20	0.3 s	$2^{21.08}$
21	22 s	$2^{27.27}$
22	2 min	$2^{29.27}$
23	1.5 h	$2^{35.21}$
24	> 7 days	$> 2^{42.02}$

Measurements made on Intel Core i5-2520 CPU @ 2.5 GHz
System is capable of doing $7.45 \cdot 10^6$ ($=2^{22.82}$) MD5 compressions per second.

Reaching step 29 is not possible within a feasible time. Changing the algorithm to setting the words in different orders had a huge negative impact on the runtime. Implementing

```
-4 A:  --------------------------------
-3 A:  --------------------------------
-2 A:  --------------------------------
-1 A:  --------------------------------
 0 A:  ???????????????????????????????? W:  --------------------------------
 1 A:  ???????????????????????????????? W:  --------------------------------
 2 A:  ???????????????????????????????? W:  --------------------------------
 3 A:  ???????????????????????????????? W:  --------------------------------
 4 A:  ???????????????????????????????? W:  --------------------------------
 5 A:  ???????????????????????????????? W:  ---------------------n----------
 6 A:  ???????????????????????????????? W:  --------------------------------
 7 A:  ???????????????????????????????? W:  --------------------------------
 8 A:  ???????????????????????????????? W:  --------------------------------
 9 A:  ???????????????????????????????? W:  --------------------------------
10 A:  ???????????????????????????????? W:  n-------------------------------
11 A:  ???????????????????????????????? W:  --------------------------------
12 A:  ???????????????????????????????? W:  --------------------------------
13 A:  ???????????????????????????????? W:  --------------------------------
14 A:  ???????????????????????????????? W:  n-------------------------------
15 A:  ???????????????????????????????? W:  --------------------------------
16 A:  ????????????????????????????????
17 A:  ????????????????????????????????
18 A:  ????????????????????????????????
19 A:  ????????????????????????????????
20 A:  ????????????????????????????????
21 A:  ????????????????????????????????
22 A:  n------n-------------------n----
23 A:  n---n-------n-------------------
24 A:  u--------n-----------u----u--u--
25 A:  u------------u-----------u----n-
26 A:  u-------------------------------
27 A:  u---------------n-------n-------
28 A:  u---n----------------u----------
29 A:  u---------------u---------------
30 A:  u---------------u---------------
31 A:  x---n---------------------------
32 A:  x---n---------------------------
33 A:  x-------------------------------
34 A:  --------------------------------
35 A:  --------------------------------
36 A:  x-------------------------------
37 A:  x-------------------------------
38 A:  x-------------------------------
39 A:  x-------------------------------
40 A:  x-------------------------------
41 A:  x-------------------------------
42 A:  x-------------------------------
43 A:  x-------------------------------
44 A:  x-------------------------------
45 A:  x-------------------------------
46 A:  x-------------------------------
47 A:  x-------------------------------
48 A:  x-------------------------------
49 A:  x-------------------------------
50 A:  x-------------------------------
51 A:  --------------------------------
52 A:  --------------------------------
53 A:  --------------------------------
54 A:  --------------------------------
55 A:  --------------------------------
56 A:  --------------------------------
57 A:  --------------------------------
58 A:  --------------------------------
59 A:  --------------------------------
60 A:  --------------------------------
61 A:  --------------------------------
62 A:  --------------------------------
63 A:  --------------------------------
```

Figure 6.6: Corrected partial differential path by Xie *et al.* [XLF13] with collision complexity 2^{41}

tunnels in the integrated data structure of the *nltool* does not have high performance because the *nltool* works with bit slices instead of whole words. Unfortunately, the tool itself is not able to complete this attack in practical time.

6.5.2 Optimizing the *nltool* for Stevens' attack

We were unable to run Stevens attack with the *nltool* in an automatic manner. Like the implementation of Stevens' two block collision, we will also embed Stevens' algorithm in the *nltool*. The complete attack algorithm will be implemented and no automated message search is used by the toolbox. Due to the fact that the runtime of the message search algorithm is expected to be longer (Stevens measured complexity was $2^{49.81}$), the target was to measure the progress of the attack. This was done by checking how many steps of the complete differential were reached. From step 28 on, no further message modification or tunnel strategy can be used, the remaining conditions are all matched in a probabilistic manner. In order to make a reasonable complexity analysis even before the algorithm finishes, the expected step probabilities and the steps where $\Delta_S a_i$ matches were calculated.

Basic data structures. First of all, we save all internal state registers in a 32-bit word. All MD5 related step operations can be applied very quickly. To get started, we need the following data structures to represent all related conditions (see table).

- Like before, we need words storing the generalized conditions '`0`' and '`1`'.
- Two-bit conditions only concern bits of $a_{i,j}$ and the step beforehand, $a_{i-1,j}$. For a faster approach, we use a bitmask where one-bits represent an active condition. Two masks are necessary, one for $a_{i,j} = a_{i-1,j}$ and $a_{i,j} \neq a_{i-1,j}$.

Table 6.7: Internal data structure for own single-block-collisions

Stephens' representation of register x	`.10^!`
Bitmask x' (1-bits)	`01000`
Bitmask x'' (0-bits)	`00100`
Bitmask for two-bit-equal conditions	`00010`
Bitmask for two-bit-unequal conditions	`00001`

Implementation details of the algorithm. The algorithm starts by setting $a_{13}, \ldots, a_{20}$ to values that satisfy the given conditions. This can be done by generating random 32-bit values and applying the bitmasks for single-bit conditions. For each step, we can also apply two-bit conditions as a forward propagation. With these values, including the chaining input, some message words and state registers can be calculated (see algorithm 7).

Lookup table. The next step is to compute the lookup table. A loop over all possible values in the state registers 2 to 5 are made. The best way to achieve this is by using a counter and applying the masks for 1 and 0-bits on it. The lookup table itself is a tuple where we have an index which consists of two words and the payload existing of five internal state registers. As described in the algorithm, the two words for the index are a combination of state registers and variables referred to as B_7 and B_{12}. These words represent the bits for two-bit conditions which we already have. The size of the initially created lookup table is saved for analysis. The lookup table itself is implemented as a hashmap where the index is a pair of two 32-bit words and the value is a list of a struct with the state words. The C++ construct `std::map` was used to create such a data structure.

Tunnels. The main loop starts by iterating steps 8 to 11 satisfying the conditions. In the inner loop, a lookup to the table is made. Now all message words $w_0, \dots, w_{15}$ are determined and step 22 is reached. The first tunnel T_4 has the bitmask `0x14872e23` as given by Stevens where each active bit defines flipable bits of the tunnel. A detailed explanation of these tunnels is given in Section 6.2.2. T_4 iterates over 2^{13} possible values. Again we can iterate by applying the mask at step 3 and check, if the next step satisfies. The same can be applied to T_9 with the bitmask `0xfffffdbc` which has 2^{28} possibilities for bit flips. T_{14} (`0xeb78d1dc`, 2^{19} possibilities) is more complicated because it affects both steps 13 and 3. It first iterates over step 13 and modifies w_{14}. Then it iterates over step 3 and checks whether both values fit for a_{25}.

Inner loop. The most inner loop is the step before the compression function is applied and the remaining conditions can only be fulfilled in a probabilistic way. This inner loop is exactly the point where a solution for the first 29 has been found.

Step probabilities. For a better understanding of the path, the probabilities of these steps are calculated. This can be done by checking the set of exhaustive conditions of the boolean functions h and k in Table 4.3. Figure 6.1 shows the path.

Results. The algorithm was run for several days. The consistency check revealed no errors when reaching step 28 except for steps 3,8,13. These are the tunnel iterations, in which the bits are flipped. The original set of sufficient conditions shows this set of bits flipped to a zero value. Therefore, this wrong consistency allows you to check if the tunnel bits really work and all other conditions are met perfectly.

The results of our own implementation were completely unsatisfactory. For this reason, we took a look on the provided implementation of Stevens himself. Shortly after publishing his paper, he put the source code of this algorithm online. This was done in the hope that additional details that were not mentioned in his work are revealed.

6.5.3 Analysis of Stevens' Implementation

Stevens published his implementation. This section will deal with its details. He uses several techniques that were not directly documented in his work.

Basic data structures. His implementation uses different bit masks and arrays that are defined in the code. For single-bit conditions, he uses a mask for selecting the active bits and the value mask itself. Moreover, he uses a mask for defining two-bit conditions. His code clearly indicates two different mask arrays, however, only one array is used. This array incorporates both two-bit equal and two-bit unequal conditions. Finally, he uses three arrays for storing various modular differences. These are $\Delta_S a_i$ and two others, $\Delta_S T_i$ and $\Delta_S R_i$.

ΔT_i **and** ΔR_i. In Stevens notation, T_i is a substep result in the step operation representing the value before the rotation operation. ΔT_i defines the difference at the step. ΔR_i is the difference after the rotation operation using ΔT_i from before. These two differences are extensively used as an additional integrity check. These checks are done in multiple stages in the algorithm starting at setting $a_{13}, \ldots, a_{20}$ satisfying conditions. The values for $\Delta_S T_i$ and $\Delta_S R_i$ are only given partially in his work from steps 25 to 63. In his implementation, however, the values for the steps before are also determined. He checks ΔT_i and ΔS_i for the steps $13, \ldots, 28$.

Lookup Table. The iteration works in reverse by starting with all possible values for a_5 and then iterating over a_2. Since a_3, a_4 are clearly fully determined, they are calculated beforehand and do not need any further iteration. In theory, 32 bits can be filled in the lookup table, hence leading to 2^{32} structs holding six 32-bit values. Storing this amount of data would take up about 768 GB without the indices. Due to constraints set on the bits, the actual number is much lower. In Stevens' implementation, he actually only defines a lower bound and recreates the table if the number of entries is below 2^{24}.

Optimized calculations. Many parts of the step operation, *i.e.* the result of the boolean function, do not change when iterating over the current state word a_i. Therefore, these values are only calculated once which speeds up the process by only performing necessary calculations.

Tunnels. The tunnels T_4 and T_9 are applied as described in the paper. However, tunnel T_{14} utilizes two different bit masks, once `(0xeb89d1dc XOR 0x0b70001c)` and `0x0b70001c`. The first tunnel iterates over all values in a_{13} satisfying a_{25}. In this loop, there is another iteration over the second tunnel value calculating a_{25} and a_2. The values for a_{13} again are a combination of both tunnel iterations. The bit mask `0x0b70001c` is never mentioned in his paper. It is assumed that he selected some bits for the second part of the tunnel and excludes these bits in the first tunnel iteration (because of the `XOR` operation).

Unused parts of code. Without a doubt, the source code he published is working. However, many structures, arrays, variables and even functions stay only defined and are not used at all. It seems that he did a lot of testing during the development process and the code was not cleaned up. Especially one array, which seemed to be used to another kind of two-bit conditions is only defined with zero values and is not referenced at any point of the program. An assumption would be that he wanted to split the two-bit conditions in two arrays, but then merged it into one.

Recording the progress. The last step in the inner loop now creates the message by putting the message words together and create a second message with the two introduced differences. The compression function is applied twice and a collision check is made. We record how far collision attack gets. Therefore we compute the modular differences Δ_M of $a_{32}, \ldots, a_{63}$ and compare it to the differential path it should follow.

Overhead on checking differences per step. Normally, the algorithm would perform the compression function after step 28 and no further checking is made. For a detailed analysis, every modular difference after step 28 is checked and counted. For this reason, the internal registers from $a_{32}, \ldots, a_{59}$ are stored to check the differences. Measurements were made on how this affects the overall runtime. The overhead is about 9.25 %.

Results. With Stevens' implementation, the runtime on our cluster was estimated to about 44 days and 16 hours on 40 CPUs. Fortunately, we already found a valid collision in roughly 15 days. Figure 6.7 shows the full differential. Table 6.8 shows all relevant information about the environment and the achieved steps. Table 6.9 shows the cumulated step information for rounds 3 and 4 including the achieved progress. Table 6.10 shows the inner loop count and the lookup table sizes for each instance.

Table 6.8: Summary of all measured results of the single-block collision attack

Number of parallel processes (CPU)	40
CPU information	Intel Xeon 2.5 GHz
MD5 operations/s per process	$2^{22.61}$
MD5 operations/s overall	$2^{27.93}$
Collision found in	15 days, 18 hours
Overall runtime per process	23 days, 4 hours
Complexity for found collision	$2^{48.3}$
Average lookup table size	$38337305.6 \approx 2^{25.19}$
Inner loops overall (= first 29 steps reached)	$6981615616 \approx 2^{32.7}$

Table 6.9: Step-by-step progress measurements of the algorithm

i	$\Delta_M a_i$	$\Delta_M w_i$	$Pr(\Delta_G a_{i-1} \to a_i)$	Measured steps reached
32	-2^{20}	0	2^{-1}	239430
33	$-2^{20}+2^{31}$	2^{25}	2^{-2}	37598
34	2^{31}	0	2^{-2}	8654
35	0	0	2^{-1}	3878
36	0	0	2^{-1}	1841
37	$\pm 2^{31}$	0	2^{-1}	900
38	$\pm 2^{31}$	0	1	900
39	$\pm 2^{31}$	0	1	901
40	$\pm 2^{31}$	2^{31}	1	901
41	$\pm 2^{31}$	0	1	900
42	$\pm 2^{31}$	0	1	904
43	$\pm 2^{31}$	0	1	900
44	$\pm 2^{31}$	0	1	901
45	$\pm 2^{31}$	0	1	900
46	$\pm 2^{31}$	0	1	900
47	$\pm 2^{31}$	0	1	901
48	$\pm 2^{31}$	0	2^{-1}	441
49	$\pm 2^{31}$	0	2^{-1}	225
50	$\pm 2^{31}$	0	2^{-1}	100
51	$\pm 2^{31}$	0	2^{-1}	58
52	$\pm 2^{31}$	0	2^{-1}	24
53	$\pm 2^{31}$	0	2^{-1}	15
54	$\pm 2^{31}$	0	2^{-1}	12
55	$\pm 2^{31}$	0	2^{-1}	9
56	0	2^{25}	2^{-1}	3
57	0	0	0	3
58	0	0	0	5
59	0	2^{31}	2^{-1}	2
60	0	0	0	4
61	0	0	0	1
62	0	0	0	3
63	0	0	0	7

Table 6.10: Per process lookup table sizes and inner loop counts

Process number	Lookup table size	Inner loop count
1	29360128	152461312
2	23068672	171577344
3	41706752	191696896
4	33554432	172539904
5	34504704	163880960
6	35586048	181481472
7	41943040	151117824
8	58720256	207646720
9	37745920	165404672
10	37748736	146509824
11	53084160	157913088
12	20971520	181751808
13	25165824	147066880
14	51904512	167878656
15	36610560	165380096
16	21233664	166887424
17	75497472	192712704
18	61341696	154234880
19	50331648	149143552
20	23068672	171212800
21	40894464	154382336
22	49152000	170242048
23	56623104	178888704
24	19922944	159469568
25	16777216	196608000
26	39911424	214409216
27	45613056	191864832
28	34078720	154415104
29	40894464	170217472
30	46137344	216784896
31	24117248	190218240
32	22528000	215404544
33	38010880	154955776
34	20447232	173670400
35	58720256	161759232
36	18743296	188456960
37	75497472	160149504
38	16777216	203620352
39	29360128	215072768
40	46137344	152526848

```
-4 A: 01100111010001010010001100000001
-3 A: 00010000001100100101010001110110
-2 A: 10011000101110101101110011111110
-1 A: 11101111110011011010101110001001
 0 A: 11011001100010101001000101010011 W: 10111110011010001101010101010100
 1 A: 11100010011001011100000100010110 W: 00111001011111011011010000111001
 2 A: 01101110000000001011111111010010 W: 11110100111110100100110001011001
 3 A: 00010100000000001001001000010000 1 W: 01011100011110110100001100001110
 4 A: 11101011011110001101000111011100 W: 11000010010000110110100101000011
 5 A: 10110100100011110011111111111111 W: 10110100010011101101110000100100
 6 A: 11011100110111011011100001101100 W: 10000101111111010111110001100110
 7 A: 10111011101100101000010110011000 W: 10001010001111111000000001011101
 8 A: 1011111100100101010110001100111un W: 011110u0011011111100000010000011
 9 A: 0000000000000000000000010000000u W: 01010111011001110011010101010100
10 A: 1111111111111111111111101111111u W: 01100101011011111110011001001010
11 A: 010001000111000111110000011011un W: 10000010101110100101111110111010
12 A: 101101000111101100100u01010110u W: 00001111101110001011011000000101
13 A: 011111101000001110u10nn11111001u W: u0100110110111100010111101110110
14 A: 00000100100001u000u00nn0001000un W: 01011100001010110010000011000111
15 A: 0001010u100000u100u0nun000100001 W: 10110000111011101110000100000101
16 A: 0010100u0u000uu010u10nn101u00000
17 A: 101u1nu1nunun01nnnu0nuuunnn010u1
18 A: 0n0uu1u100u011n1uu1nnnnnu1nn0u00
19 A: 10n101011u0100110110u1000011u0u0
20 A: 1011u11001un001u110000100n100011
21 A: 10u1n0100nu1111u1000n1111n1n0110
22 A: 001111011un110u01110n0000u1n0010
23 A: 101011001n0100101010010 01u1u1011
24 A: 010111n010n0011111101100 1u1u011u
25 A: 101001011n11001001101001u1010011
26 A: n10011n000100110000010101 10n0011
27 A: u00011u1000010111010n011111n1011
28 A: 110110n10001000n11001010100u1010
29 A: n01001n10001110001100111010u1111
30 A: n11000110111000110001 10n000u1010
31 A: 1010000100001010100111 0n11001111
32 A: 11101101001n00000001010101110110
33 A: u0110110010n00000100100000101000
34 A: u1000100110001111001011001011100
35 A: 00010111000101001110111110111100
36 A: 00100110000100101101001101101111
37 A: n1000111101011111111000101000000
38 A: n1111111010111001010110110101100
39 A: n1111100000001001110000001100000 0
40 A: n0100100011110111001111000110011
41 A: n1101011100010110000011011110010
42 A: u1100100110010111110101110110010
43 A: u0110100001001001101000100011000
44 A: u1000101111000101101001010100001
45 A: n0101111110110100001001001011010
46 A: u0001011011011000010010111010011
47 A: n0001010110100111011010111111111
48 A: u0001110111111000100111101110001
49 A: n1010010111011111001001100110110
50 A: u1010010001100111001100101010101
51 A: n1111010110111110100101110100000
52 A: u1111010001101001100100011101100
53 A: n1100111010101010101101111111010
54 A: u0111100111011010000011001100100
55 A: n0000011110011111001100111011101
56 A: 11010111110000000111011110011111
57 A: 00010001111010001111111101001001
58 A: 00111001010100100000001110101101 1
59 A: 10011110000101100011000100000100
60 A: 11000101001110111100010101011010
61 A: 11100101101101001001000000001000
62 A: 00000110101111001100110011111101
63 A: 00100010100101000101011010010010
```

Figure 6.7: Single block collision - Full single-block collision differential characteristic

6.5.4 Create a collision attack with the partial path by Xie *et al.*

Although not many details on the attacks by Xie *et al.* [XLF13] were given, we will use the information about their best partial differential path (see Figure 6.6) with a complexity of 2^{41} as a starting point to recreate the attack. We will use the nltool to derive a full differential with the set of sufficient conditions and then use the established techniques like message modification and tunneling to create a collision attack algorithm.

Deriving a full differential path. As Figure 6.6 presents, the conditions of steps $5, \ldots, 21$ are not given by Xie *et al.* We will use a specific search configuration to fill the remaining parts. The naive approach by just randomly filling all '`?`' bits is not appropriate since certain design constraints are desired. Figure 6.8 shows the parameters that were chosen to create a full differential characteristic. First of all, it is important to keep certain steps sparse with a low number of conditions. The corresponding message words can therefore be fulfilled easily. Of course, this principle would be desired for the whole path, however, certain steps have to contain a higher amount of conditions. The search using the nltool can be broken down into the following phases:

1. Top-down search for the internal registers $a_5, \ldots, a_{10}$. All '`?`' bits will be replaced by a '`x`' bit, if possible. After that, we want to get rid of those '`x`' bits by randomly checking '`u`' or '`n`'. As the arrow in Figure 6.8 suggests, each word is processed until stepping to the next step. The number of conditions should be low. The nltool will find many solutions in a short time, we will select the one with the lowest number of conditions.

2. Bottom-up search for $a_{21}, \ldots, a_{14}$. The approach is the same as in the first phase. The runtime of this phase is also expected to be rather low. We will also find solutions with a varying number of conditions and select the one with the lowest complexity.

3. Randomly guessing all bits in $a_{11}, \ldots, a_{13}$. This process is assumed to take longer since the conditions that have to be met here have to fit with the previous and next steps.

The first two phases found many suitable candidates in minutes. Phase three found solutions after roughly 22 hours. The overall complexity for all three phases is an equivalent of $2^{39.56}$ MD5 compression evaluations.

Setting tunnel bits and creating sufficient conditions. To reach a set of sufficient conditions, we need to determine, which bits are suitable to be used for the tunneling. The same tunnels, T_4, T_9, T_{14}, are integrated as used by Stevens [Ste12a]. By testing each bit by placing a pattern of the affected state registers a_3, a_8, a_{13}, we can determine the tunnel masks. After setting the tunnel bits, we have derived a set of sufficient conditions, which can be found in Figure 6.10. For tunnel T_4, the mask for bit flipping is `0xfbfffffe` and for tunnel T_9, `0x30007dde`.

```
                      -4 A:  --------------------------------
                      -3 A:  --------------------------------
                      -2 A:  --------------------------------
                      -1 A:  --------------------------------
                       0 A:  ???????????????????????????????? W:  --------------------------------
Zero differences       1 A:  ???????????????????????????????? W:  --------------------------------
Steps 0-4              2 A:  ???????????????????????????????? W:  --------------------------------
                       3 A:  ???????????????????????????????? W:  --------------------------------
                       4 A:  ???????????????????????????????? W:  --------------------------------
1.                     5 A:  ???????????????????????????????? W:  ---------------------n----------
Sparse diff.           6 A:  ???????????????????????????????? W:  --------------------------------
Steps 5-10             7 A:  ???????????????????????????????? W:  --------------------------------
                       8 A:  ???????????????????????????????? W:  --------------------------------
                       9 A:  ???????????????????????????????? W:  --------------------------------
                      10 A:  ???????????????????????????????? W:  n-------------------------------
3.                    11 A:  ???????????????????????????????? W:  --------------------------------
Steps 11-13           12 A:  ???????????????????????????????? W:  --------------------------------
                      13 A:  ???????????????????????????????? W:  --------------------------------
                      14 A:  ???????????????????????????????? W:  n-------------------------------
                      15 A:  ???????????????????????????????? W:  --------------------------------
2.                    16 A:  ????????????????????????????????
More density          17 A:  ????????????????????????????????
Steps 14-21           18 A:  ????????????????????????????????
                      19 A:  ????????????????????????????????
                      20 A:  ????????????????????????????????
                      21 A:  ????????????????????????????????
                      22 A:  n------n-------------------n----
                      23 A:  n---n-------n-------------------
                      24 A:  u--------n-----------u----u--u--
                      25 A:  u------------u-----------u----n-
                      26 A:  u-------------------------------
                      27 A:  u---------------n-------n-------
                      28 A:  u---n----------------u----------
                      29 A:  u---------------u---------------
                      [...]
```

Figure 6.8: Path design principles for partial differential by Xie *et al.* [XLF13]

Comparison with other single-block differential paths. The full differential paths by Stevens and Xie *et al.* (see Figure 6.1) contain 154 respectively 145 bits with the generalized conditions '`u`','`n`' and '`x`'. The number of our self created path in Figure 6.10 is 142 and therefore slightly lower. Our path is very similar to the one by Xie *et al.* which also displays a high density of conditions between steps 14 and 16 whereas Stevens' differential is much more sparse at this point and has a high amount of conditions between steps 17 and 19. The steps with the highest densities are 12 and 13.

Constructing a collision attack. The last step is to design an algorithm which is adapted to our own differential characteristic. The same way as custom configurations in the nltool were used for deriving a full path, we can use it to fill the remaining bits with the generalized condition '`-`' with '`0`' or '`1`'. Of course, the naive approach would be to fill up all bits in a completely random manner. However, we will adapt a step-by-step approach, which was also used in previous single and two-block attacks. It has to be considered that the performance is much slower compared to the word-wise approach, nevertheless intermediate results should be found.

Finding a step reduced collision. With the set of sufficient conditions, we can design the collision attack which replaces all remaining '`-`' bits with '`0`' or '`1`'. Stevens' algorithm for choosing the message words is used (see Algorithm 7). All these searches until step 23 can be done with the nltool by applying a custom search configuration, which was also used for the path search. The search satisfies each internal state register completely until moving on. The message search will be split into three phases as displayed in Figure 6.10. First, the process starts with the registers a_{13} to a_{20}. After that, Stevens' algorithm uses a lookup table, which iterates $a_1, \ldots, a_5$. In our search variant, these registers are set the same way as in the first phase. Phase 3 fills up the remaining registers $a_6, \ldots, a_{12}$. It is important to reach partial solutions until step 22. From there on, tunnels can be used to satisfy further steps. Table 6.11 shows that the complexity for reaching step 22 is $2^{31.04}$.

Table 6.11: Complexities of modified *nltool* search configuration for our own single-block collision attack

Step	Runtime per process	Number of CPUs	Complexity
20	5 s	1	$2^{25.15}$
21	70 s	1	$2^{28.95}$
22	5 min	2	$2^{31.04}$
23	29 h	8	$2^{42.28}$

Iterate through tunnels and check modular differences. The tunnel mask was determined to ensure that we can flip certain bits to go even further when satisfying conditions. We use our modified word-based data structures to iterate through tunnel bits. The validity check only includes verifying the modular difference and using the fast reference implementation. Figure 6.9 shows two results where tunneling was successful. By flipping certain bits in a_3, we could ensure that all conditions including a_{23} can hold. When iterating through values of a_3, the message words w_3, w_4, w_7 have to be updated. The same applies for modifying a_8.

Estimated complexity for full collision. Figure 6.10 shows the full differential path. A full solution for 24 steps can be found at Figure 6.9. When looking at the steps 23 to 50, we have 43 remaining differences. As we take a look on Table 6.9 from the previous attack, we see that the MSB path from step 36 to 47 holds with a probability of 1. Therefore we have 31 remaining conditions, the estimation for a full collision attack is not higher than $2^{62.04}$.

```
-4 A: 01100111010001010010001100000001
-3 A: 00010000000110010010101000111 0110
-2 A: 10011000101110101101110011111110
-1 A: 11101111110011011010101110001001
 0 A: 10011001011100011011010011010000 W: 10110111111010001010001110011011
 1 A: 11111011011101001100010001110101 W: 01110111100100000010101110110110
 2 A: 00000000000000010000000000000000 W: 11000010111100001000010010010101
 3 A: 00000010010010000000000000000000 W: 11010001000000111101000011000010
 4 A: 11100000001000000111111111111111 W: 01110111100110010110100100001011
 5 A: 10001011nu101011000100101000 1001 W: 11100101011011010010 1n0000000010
 6 A: 01100101 1n11101111111101111100011 W: 01001100001111000001010010001100
 7 A: 10101001101110111 11n11000000 1111 W: 01111110110001110000010101110001
 8 A: 00u1111111000001000n000000100100 W: 10111101011011100111011011001110
 9 A: 01u00000110000001010000000100n11 W: 01111111100001100000100111111010
10 A: 10111001010100 0n11001010n0011n00 W: n0000101010001000000100010001000
11 A: 0010001110unn11n1001011nu1uuun00 W: 11111010101100100111101101110 00
12 A: 001n1uuuu01nuuu0uuuuuun01n001u10 W: 00011111100001100101110101101010
13 A: 0u11un101u000unu101111nn01nunuuu W: 11101010010001010000100100011101
14 A: 0n011n1111101110111n10u000101001 W: n0001011011110001000010001000000
15 A: 011u00101u1010u1101100001u100011 W: 01000110111110111000110110111011
16 A: 1001110000010 01uu01100u100000101 W: [ 1]
17 A: n010u00n1000000110n0000u00000011 W: [ 6]
18 A: 00101n0111001111111 1uu0000010001 W: [11]
19 A: 1110110010000011uu00110011u11101 W: [ 0]
20 A: 0nn11100u0nnn0n010111011un1n1101 W: [ 5]
21 A: n01u0n01001011100un100u11010011n W: [10]
22 A: n010101n0010101010110010000n0001 W: [15]
23 A: n110n1110110n0100000010000000011 W: [ 4]
```

```
-4 A: 01100111010001010010001100000001
-3 A: 00010000000110010010101000111 0110
-2 A: 10011000101110101101110011111110
-1 A: 11101111110011011010101110001001
 0 A: 10011001011100011011010011010000 W: 10110111111010001010001110011011
 1 A: 11111011011101001100010001110101 W: 01110111100100000010101110110110
 2 A: 00000000000000010000000000000000 W: 11000010111100001000010010010101
 3 A: 00000000010000010000000000000000 W: 10110101000000111101000010111010
 4 A: 11100000001000000111111111111111 W: 01110101100111000111011100001011
 5 A: 10001011nu101011000100101000 1001 W: 11100101011011010010 1n0000000010
 6 A: 01100101 1n11101111111101111100011 W: 01001100001111000001010010001100
 7 A: 10101001101110111 11n11000000 1111 W: 10000000110011100000010101110001
 8 A: 00u1111111000001000n000000100100 W: 10111101011011100111011011001110
 9 A: 01u00000110000001010000000100n11 W: 01111111100001100000100111111010
10 A: 10111001010100 0n11001010n0011n00 W: n0000101010001000000100010001000
11 A: 0010001110unn11n1001011nu1uuun00 W: 11111010101100100111101101110 00
12 A: 001n1uuuu01nuuu0uuuuuun01n001u10 W: 00011111100001100101110101101010
13 A: 0u11un101u000unu101111nn01nunuuu W: 11101010010001010000100100011101
14 A: 0n011n1111101110111n10u000101001 W: n0001011011110001000010001000000
15 A: 011u00101u1010u1101100001u100011 W: 01000110111110111000110110111011
16 A: 1001110000010 01uu01100u100000101 W: [ 1]
17 A: n010u00n1000000110n0000u00000011 W: [ 6]
18 A: 00101n0111001111111 1uu0000010001 W: [11]
19 A: 1110110010000011uu00110011u11101 W: [ 0]
20 A: 0nn11100u0nnn0n010111011un1n1101 W: [ 5]
21 A: n01u0n01001011100un100u11010011n W: [10]
22 A: n010101n0010101010110010000n0001 W: [15]
23 A: n100n1110110n0011110010000110100 W: [ 4]
```

Figure 6.9: Example of tunneling for T_4

```
-4 A: 01100111010001010010001100000001
-3 A: 00010000000110010010101000111011 0
-2 A: 10011000101110101101110011111110
-1 A: 11101111111001101101010111000100 1
 0 A: -------------------------------- W: --------------------------------
 1 A: -------------------------------- W: --------------------------------
 2 A: 000000000-00000-0000000000000000 W: --------------------------------
 3 A: 00000000010000010000000000000000 W: --------------------------------
 4 A: ---00000010000001111111111111111 W: --------------------------------
 5 A: --------nu---------1------------ W: ---------------------n----------
 6 A: --1-----1n---------1------------ W: --------------------------------
 7 A: --1-----10---------n---------1-- W: --------------------------------
 8 A: --u------1000001000n00000-100100 W: --------------------------------
 9 A: --u------10000001---00000-100n11 W: --------------------------------
10 A: ---1-00---01000n1--0--10n0011n-- W: n00-----------------------------
11 A: -010001---unn11n------1nu1uuun0- W: --------------------------------
12 A: -0-n1uuuu-1nuuu0uuuuuun01n001u1- W: ----------00---------1-------01-
13 A: -u-1un101u0-0unu-01111nn01nunuuu W: ------------0-------------01-10-
14 A: -n-11n1111-01110111n1-u0-0101-01 W: n---------1110--------000-------
15 A: 01-u00-0-u---0u11-1---00-u100-11 W: ---0---------------------011----
16 A: 100111-0-------uu-1-00u1-------- W: [ 1]
17 A: n010u0-n--------10n-00-u--0----- W: [ 6]
18 A: 001--n-11-001-1-111-uu--0001---- W: [11]
19 A: 1110-1-01-000-1-uu0---0-11u1---- W: [ 0]
20 A: -nn1-1-0u-nnn-n-101---1-un-n---- W: [ 5]
21 A: n--u0n-1----1---0un---u-10-0---n W: [10]
22 A: n--01--n-0--1----0---0--000n-0-1 W: [15]
23 A: n--0n--1-1--n----1---1---01--100 W: [ 4]
24 A: u------0-n--11-------u--01u--u1- W: [ 9]
25 A: u-----------0u--0----0--0u1---n- W: [14]
26 A: u---0-----------1----0--1-1----- W: [ 3]
27 A: u---1-----------n----1--n------- W: [ 8]
28 A: u---n-----------0----u---------- W: [13]
29 A: u---0-----------u--------------- W: [ 2]
30 A: u---1-----------u--------------- W: [ 7]
31 A: x---n--------------------------- W: [12]
32 A: x---n---------------------------
33 A: x-------------------------------
34 A: --------------------------------
35 A: --------------------------------
36 A: x-------------------------------
37 A: x-------------------------------
38 A: x-------------------------------
39 A: x-------------------------------
40 A: x-------------------------------
41 A: x-------------------------------
42 A: x-------------------------------
43 A: x-------------------------------
44 A: x-------------------------------
45 A: x-------------------------------
46 A: x-------------------------------
47 A: x-------------------------------
48 A: x-------------------------------
49 A: x-------------------------------
50 A: x-------------------------------
51 A: 1-------------------------------
52 A: 0-------------------------------
53 A: --------------------------------
54 A: --------------------------------
55 A: --------------------------------
56 A: --------------------------------
57 A: --------------------------------
58 A: --------------------------------
59 A: --------------------------------
60 A: --------------------------------
61 A: --------------------------------
62 A: --------------------------------
63 A: --------------------------------
```

2.

3.

1.

Tunnels T_4, T_9, T_{14}

Figure 6.10: Set of sufficient conditions of self created differential path based on the partial path by Xie *et al.* [XLF13] and message collision phases

Chapter 7

Conclusion

Attacking deprecated hash functions like MD5 has its charm because its principles and ideas can be used for other hash functions, *i.e.* SHA-1, as well. MD5 is still used in many protocols like SSL certificates. Many theoretical attacks have been published that are a proof of concept. Moreover, also meaningful attacks exist like the attack by Stevens *et al.* [SLW07] which generates a rogue certificate. The first part of this work focuses on hash functions and the widely-used MD-family. Wang *et al.* achieved a major breakthrough in the analysis of MD5. This approach was the base of many subsequent attacks by different cryptologists all over the world. Klima introduced the idea of using tunnels which reduced the complexity of Wang's attack. Furthermore, it was greatly improved by Stevens. Those collisions attacks relied on two message blocks. In 2010, Xie *et al.* [XF10] were to first to create a collision with only a single message block. They called a competition on who would find another independent solution. Stevens [Ste12a] was the first one and published many details on his attack. Very recently in 2013, Xie *et al.* [XLF13] presented results containing details about their first single-block attack of 2010 and further improvements. Their best attack complexities are considerably below Stevens' attack. Unfortunately, Xie *et al.* only revealed partial differential paths and no full one.

In this thesis, we have analyzed all essential attacks including the original approach by Wang *et al.* Furthermore, the concepts of *message modification* and *tunnels* were explained. A detailed analysis of the differential paths was made using the *nltool.* A tunnel pattern was explained in detail both using the *nltool* and manual calculations. Focus was laid on what the toolbox could do automatically and which parts of the attacks required additional implementation.

All of the attacks handle the internal states of MD5 in a word-wise manner to allow fast operations (like the MD5 step operation) to be performed on all bits simultaneously. The approach of the *nltool* is completely different as it handles the state registers and its conditions on a bit sliced principle. This significantly slows down many attacks as they are optimized on the word-wise approach. The two-block collision attack by Stevens [Ste06] was partially reimplemented using the word-wise approach and embedded into the *nltool.*

The final part of this thesis addresses single-block attacks. We used the nltool to compare the differential paths of Xie *et al.* and Stevens. Furthermore, the automated search algorithm by the toolbox was used to recreate a single-block collision based on Stevens' differential path. The complexity exceeded computational feasibility. Hence, Stevens attack was completely embedded into the nltool. First attempts of the implementation failed due to exceeding complexity of our implementation. Therefore, Stevens reference implementation was analysed and more details which were not given in his work were revealed. We then verified the algorithm by running a single-block collision attack successfully with a complexity of $2^{48.3}$ which verified Stevens' given complexity.

Finally, the best partial differential path was taken from the work of Xie *et al.* in 2013. The nltool was used to derive a full working differential characteristic by running a custom search configuration. This enabled us to even control the density of the path at certain steps. For finding actual message values with our custom derived path, the nltool was run with choosing step-by-step values in the same manner as in Stevens' attack. A partial solution for 23 steps was found. We then used this partial solution for tunnel processing which includes bit flipping and then calculating the MD5 steps with checking the modular differences. Step 24 was reached with a complexity of $2^{42.28}$. We estimate the complexity of a full collision attack with $2^{62.04}$.

Future work may include finding a better path with less conditions for a single-block collision. The adoptions for the *nltool* using optimized bit masks for message modification techniques could be used for other attacks on different hash functions of the MD-family.

List of Algorithms

List of Figures

List of Tables

Bibliography

[BCH06] John Black, Martin Cochran, and Trevor Highland. A Study of the MD5 Attacks: Insights and Improvements. In *FSE*, volume 4047 of *LNCS*, pages 262–277. Springer, 2006.

[BCJ+05] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 36–57. Springer, 2005.

[Ber93] Thomas Berson. Differential Cryptanalysis Mod 232 with Applications to MD5. In RainerA. Rueppel, editor, *Advances in Cryptology EUROCRYPT 92*, volume 658 of *Lecture Notes in Computer Science*, pages 71–80. Springer Berlin Heidelberg, 1993.

[BS91] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In *CRYPTO'91*, 1991.

[CJ98] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *LNCS*, pages 56–71. Springer, 1998.

[CR06] Christophe De Cannire and Christian Rechberger. Finding SHA-1 Characteristics: General Results and Applications, 2006.

[Dam89] Ivan Bjerre Damgård. A design principle for hash functions. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 416–427, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[Dau05] Magnus Daum. *Cryptanalysis of Hash Functions of the MD4-family*. 2005.

[dBB94] Bert den Boer and Antoon Bosselaers. Collisions for the compression function of MD5. In *Workshop on the theory and application of cryptographic techniques on Advances in cryptology*, EUROCRYPT '93, pages 293–304, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[DLS11] John Floren Daniel Liu and Tim Sperr. Attacking MD5: Tunneling & Multi-Message Modification. `http://www.jayway.com/wordpress/wp-content/uploads/2011/11/Attacking-MD5.pdf`, 2011. [Online, accessed 13-04-2013].

[Dob96] Hans Dobbertin. Cryptanalysis of MD5 Compress, May 1996.

[Dob98] Hans Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[EJ] Donald E. Eastlake and Paul E. Jones. US Secure Hash Algorithm 1 (SHA1). `http://www.ietf.org/rfc/rfc3174.txt?number=3174`.

[GPFW96] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1996.

[ILL89] Russell Impagliazzo, Leonid A. Levin, and Michael Luby. Pseudo-random Generation from one-way functions (Extended Abstracts). In *STOC*, pages 12–24, 1989.

[Jou04] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *LNCS*, pages 306–316. Springer, 2004.

[Kli06] Vlastimil Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105, 2006. `http://eprint.iacr.org/`.

[Mer89] Ralph C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[MNS11] Florian Mendel, Tomislav Nad, and Martin Schläffer. Finding SHA-2 Characteristics: Searching Through a Minefield of Contradictions. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *LNCS*, pages 288 – 307. Springer, 2011.

[MNS12] Florian Mendel, Tomislav Nad, and Martin Schläffer. Collision Attacks on the Reduced Dual-Stream Hash Function RIPEMD-128. In Anne Canteau, editor, *Fast Software Encryption*, volume 7549 of *LNCS*, pages 226 – 243. Springer, 2012.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[Rab79] Michael O. Rabin. Digitalized Signatures and Public-Key Functions as Intractable as Factorization. Technical report, Cambridge, MA, USA, 1979.

[Riv91] RonaldL. Rivest. The MD4 Message Digest Algorithm. In AlfredJ. Menezes and ScottA. Vanstone, editors, *Advances in Cryptology-CRYPT0 90*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer Berlin Heidelberg, 1991.

[Riv92] R. Rivest. The MD5 Message-Digest Algorithm, 1992.

[Rom90] John Rompel. One-way functions are necessary and sufficient for secure signatures, 1990.

[SA08] Yu Sasaki and Kazumaro Aoki. Preimage Attacks on Step-Reduced MD5. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *ACISP*, volume 5107 of *LNCS*, pages 282–296. Springer, 2008.

[Sch06] Martin Schläffer. Cryptanalysis of MD4, 2006.

[SLW07] Marc Stevens, Arjen Lenstra, and Benne Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2007.

[SLW12] Marc Stevens, Arjen K. Lenstra, and Benne De Weger. Chosen-prefix collisions for MD5 and applications. *Int. J. Appl. Cryptol.*, 2(4):322–359, July 2012.

[Ste06] Marc Stevens. Fast collision attack on md5. Cryptology ePrint Archive, Report 2006/104, 2006. `http://eprint.iacr.org/`.

[Ste12a] Marc Stevens. Single-block collision attack on MD5. Cryptology ePrint Archive, Report 2012/040, 2012. `http://eprint.iacr.org/`.

[Ste12b] M.M.J. Stevens. *Attacks on Hash Functions and Applications*. 2012.

[WY05] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, *Advances in Cryptology EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 561–561. Springer Berlin / Heidelberg, 2005. 10.1007/11426639-2.

[XF09] Tao Xie and Dengguo Feng. How To Find Weak Input Differences For MD5 Collision Attacks. Cryptology ePrint Archive, Report 2009/223, 2009. `http://eprint.iacr.org/`.

[XF10] Tao Xie and Dengguo Feng. Construct MD5 Collisions Using Just A Single Block Of Message. Cryptology ePrint Archive, Report 2010/643, 2010. `http://eprint.iacr.org/`.

[XFL08] Tao Xie, Dengguo Feng, and Fanbao Liu. A New Collision Differential For MD5 With Its Full Differential Path. *IACR Cryptology ePrint Archive*, 2008:230, 2008.

[XLF13] Tao Xie, Fanbao Liu, and Dengguo Feng. Fast Collision Attack on MD5. Cryptology ePrint Archive, Report 2013/170, 2013. `http://eprint.iacr.org/`.

[YS05] Jun Yajima and Takeshi Shimoyama. Wang's sufficient conditions of MD5 are not sufficient, 2005. jyajima@labs.fujitsu.com 13005 received 10 Aug 2005, last revised 10 Aug 2005.

[Yuv79] Gideon Yuval. How to Swindle Rabin. *Cryptologia*, 3:187–189, 1979.