Cryptanalysis of Hash Functions of the MD4-Family

MAGNUS DAUM

Dissertation zur Erlangung
des Grades eines Doktor
der Naturwissenschaften
der Ruhr-Universität Bochum
am Fachbereich Mathematik

vorgelegt von
Magnus Daum

unter der Betreuung von
Prof. Dr. Hans Dobbertin

Bochum, Mai 2005

Wenn du jetzt aufgibst
wirst du's nie versteh'n
Du bist zu weit um umzudreh'n

Vor dir der Berg
Du glaubst du schaffst es nicht
doch dreh' dich um und sieh'
wie weit du bist
im Tal der Tränen liegt auch Gold
komm lass es zu
dass du es holst

(Rosenstolz - "Wenn Du Jetzt Aufgibst")

Contents

1 Introduction 1

2 Cryptographic Hash Functions 13

2.1 Basic Definitions and Properties 13

2.1.1 Generic Attacks 16

2.2 Hashing by Iterated Compression 17

2.2.1 The MD-Design Principle 17

2.2.2 Collisions and Pseudo-Collisions 19

2.2.3 Multicollisions 21

3 Hash Functions of the MD4-Family 23

3.1 Structure 23

3.1.1 Parameters 24

3.1.2 Notation 25

3.1.3 Subfamilies 27

3.2 Message Expansion 28

3.2.1 Roundwise Permutations 28

3.2.2 Recursive Message Expansions 29

3.3 Step Operation 31

3.3.1 Bitwise Applied Boolean Functions 31

3.3.2 Structure and Notation 32

3.3.3 Specific Features 34

4 A Toolbox for Cryptanalysis 39

4.1 Links between Different Kinds of Operations 40

4.1.1 $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$ 40

4.1.2 Connections between + and $\oplus$ 42

4.1.3 Modular Addition and Bit Rotations 45

4.2 Measuring the Avalanche Effect 49

4.2.1 Different Metrics 49
4.2.2 NAF-Weight and NAF-Distance 52
4.2.3 Quantifying the Avalanche Effect 57
4.3 Difference Propagation 62
4.3.1 $\oplus$-Differences and Approximations 63
4.3.2 Finding Modular Differential Patterns 65

5 Cryptanalytic Methods 73
5.1 Structural Aspects of the Attacks 74
5.1.1 Difference Patterns 74
5.1.2 Multiblock Collisions 76
5.2 Dobbertin's Method 77
5.2.1 The Method 77
5.2.2 Results 91
5.2.3 Extensions and Further Analysis 93
5.3 Method by $\mathbb{F}_2$-linear Approximations 97
5.3.1 Original Method of Chabaud and Joux 98
5.3.2 Neutral Bits Method 100
5.3.3 Results 102
5.4 A Systematic Treatment of Wang's Method 102
5.4.1 The Method 102
5.4.2 Results 111
5.4.3 Extension of Wang's Method 112
5.5 Practical Relevance 118
5.5.1 Meaningful Collisions: The Poisoned Message Attack . 118

6 Solution Graphs 121
6.1 Dobbertin's Original Algorithm 122
6.2 T-Functions 124
6.3 Solution Graphs for Narrow T-functions 126
6.4 Algorithms for Solution Graphs 130
6.4.1 Reducing the Size 130
6.4.2 Computing Solutions 134
6.4.3 Combining Solution Graphs 136
6.5 Extensions of this Method 138
6.5.1 Including Right Shifts 138
6.5.2 Generalized Solution Graphs 138
6.5.3 Including Bit Rotations 139
6.6 Examples of Applications 140

7 Conclusion **143**
7.1 Status of Current Hash Functions 143
7.2 Perspectives 145

A Specifications **147**

Bibliography **159**

List of Symbols **165**

Index **167**

You know that the beginning is
the most important part of any work.
(Plato)

Chapter 1

Introduction

Hash functions — or rather *cryptographic* hash functions — constitute an important primitive necessary in many cryptographic applications. They are used to compress messages — to be precise, arbitrary long messages are mapped to fixed length *hash values*.

This compression is *not* supposed to preserve the complete original content of the message. Obviously, that would not be possible, as there is an infinite number of messages of arbitrary length, but only a limited number of fixed length values. Rather, the hash value of a message should be regarded as a *digital fingerprint*: like fingerprints are used to (almost) uniquely identify a person, cryptographic hash values can be used to (almost) uniquely identify a message. This means, given a message and a hash value, it is possible to decide whether the message fits to the hash value or not. Furthermore, similar to comparing the fingerprints found at a site of crime with those in a database of suspects, one can check which is the original message to some hash value, as long as there is only a limited set of messages to choose from. However, it is *not* possible to recover an original message given only its hash value, like a fingerprint alone does not tell anything about the person it belongs to.

Mathematically – but informally – speaking, hash functions should have the property that restricted to randomly chosen (not "too big") finite subsets of the domain they should be injective with high probability.

The term *hash function* is also used in a non-cryptographic framework, referring to functions which share some properties with cryptographic hash functions but are contrary concerning other important properties. However, in this thesis we are only dealing with *cryptographic* hash functions. See [Knu98] for more details and examples on *non-cryptographic* hashing.

Recently the topic of hash functions — and especially of those of the MD4-family which are the most important in practical use — has received a lot of attention in the cryptographic community and even beyond. This is due to some attacks published recently, which come close to “breaking” some hash functions which are widely used in many cryptographic standards. *Breaking* may, for example, mean to find a way of efficiently producing different messages which are mapped to the same hash value by some hash function, as that would compromise the security of many cryptographic applications in which this functions is used.

Therefore it is very important to study these functions and their cryptanalysis, not only because they are interesting cryptographical objects, but also to assess the situation with current standards building on these hash functions, which are widely used in practice.

Applications of Hash Functions. Hash functions are used in various cryptographic schemes, the most important being digital signature schemes.

A digital signature scheme is an asymmetric cryptographic scheme used to provide messages with personalized signatures which serve to prove the authenticity of the message and the sender.

Suppose that Alice wants to send a message to Bob, convincing him that she is really the sender of the message. Therefore, she would use such a scheme to compute a digital signature for her message depending on some information that is private to her, called the secret key. Upon receiving the message together with the signature, Bob can use some publicly available information, called the public key, to verify this signature.

The problem with these asymmetric schemes is, that they are quite slow and that the signature usually is about as big as the message itself. This is the reason for the application of hash functions in digital signature schemes, as illustrated in Figure 1.1: Alice does not sign the message itself, but she signs only the hash value (the digital fingerprint) of the message. This can be computed very efficiently and is usually much shorter than the message. Therefore, the computation of the signature is much faster and the signature itself is much smaller. For verifying the signature, Bob by himself computes the fingerprint of the message he received and verifies whether the signature he received is valid for this fingerprint.

However, for a hash function to be useful in a digital signature scheme, it does not suffice that it compresses messages efficiently in an arbitrary

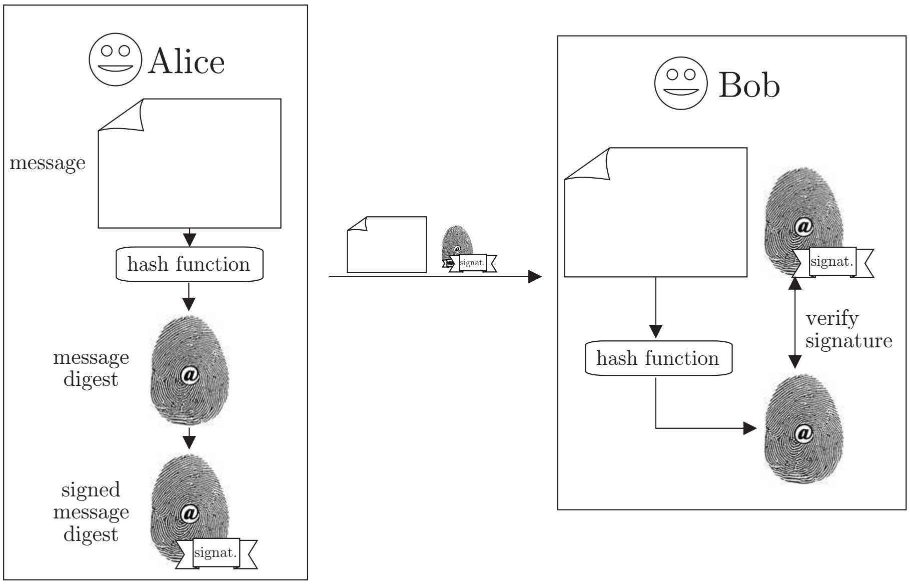

Figure 1.1: Hashing in a digital signature scheme.

manner. It is of utmost importance that it is not possible for anybody to find a *collision*. A collision consists of two messages which are mapped to the same hash value by this hash function. If this holds, then we call such a hash function *collision resistant*.

The reason for this requirement is that a malicious third person Eve could take advantage of a collision as follows (cf. Figure 1.2): Suppose Eve would be able to produce two messages which are mapped to the same hash value and whose contents differ significantly, for example two contracts about buying something but with different prices. Alice might be willing to sign the first message which seems to be a cheap offer to her. So Eve asks Alice for her digital signature on this contract and what she receives is not only a valid signature for the first but also for the second message. The signature is valid for both messages, because the verification process only refers to their common hash value. This means, Eve can replace one message by the other and claim that Alice signed the second message, in the example the contract with the much higher price. Hence, it is very important to require that hash functions are collision resistant.

Hash functions also occur as components in various other cryptographic applications (e.g. protection of pass-phrases, protocols for payment, broadcast authentication etc.) where usually their property as a (computational) *one-way* function is used. Roughly spoken, a hash function is called one-way

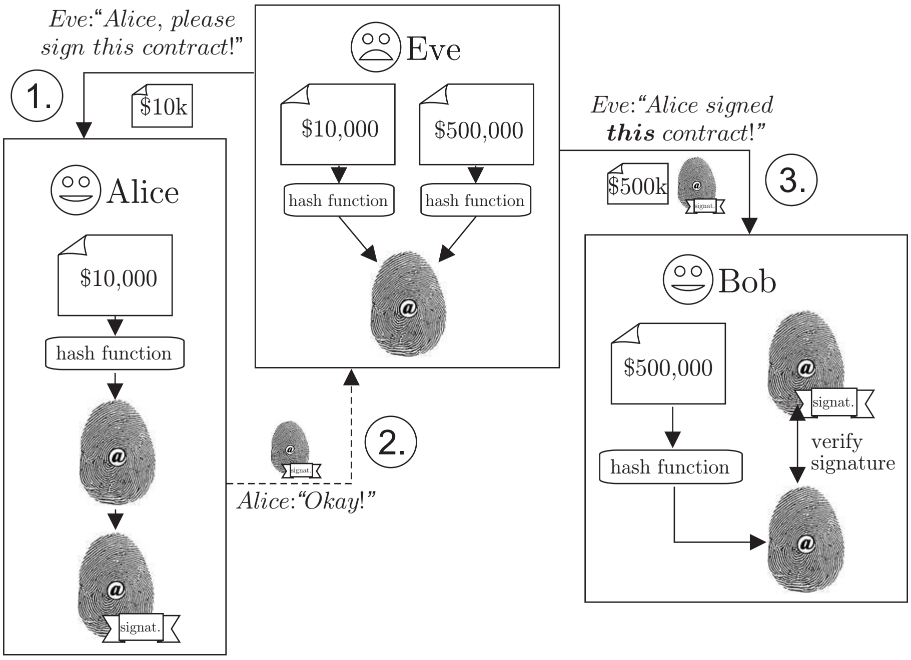

Figure 1.2: Forging a signature with a non collision resistant hash function.

if it is not possible to find a message fitting to a given hash value.

Such one-way functions are used in many cryptographical protocols, for example for commitments. In order to commit to a value one simply publishes its hash value. As long as the hash function is one-way, nobody gets to know which value was chosen. However, when the value is published, everybody can check that it is, in fact, a preimage of the previously published hash value.

In the application of digital signatures, the cryptographic defect is made even worse — in comparison to the loss of collision resistance — if one-wayness is violated (to be precise: if it would be possible to find a message different to the original one hashing to the same value): Eve does not depend on the cooperation of Alice. If she obtained any message signed by Alice, then she would be able to replace it by a different message.

Symmetric and Asymmetric Cryptography. Usually two main parts of cryptography are distinguished:

- *symmetric* cryptography, in which usually all participants share the same (secret) key, which is used e.g. for encryption as well as for decryption

- *asymmetric* cryptography, in which every participant usually has two different keys, one public key, for encryption and verification, and one secret key, for decryption and signing

In asymmetric cryptography, which is the far younger one of the two subjects, schemes are most often based on the hardness of well studied mathematical problems like factoring integers or computing discrete logarithms, for example. For these schemes there are strong connections between the security of the system and the hardness of the underlying problem, e.g. proofs saying something like: if the scheme could be broken then the underlying problem would be solved. Hence, in these schemes the role of mathematics is quite evident.

In contrast to this, the situation in symmetric cryptography is not that nice from a mathematical point of view. By exaggerating a bit we could say that in this case it often seems that security is achieved by lumping everything together and stirring up intensely. Usually the schemes are optimized for efficiency and with respect to security they try to come as close as possible to the — optimally secure but extremely inefficient — one-time-pad, e.g. by simulating random functions. Very often the designs can be described as a kind of mixing bits using concepts like *confusion* or *diffusion*. Thus, in many cases the contribution of mathematics is quite limited, usually restricted to some supporting roles, for example, finding useful criteria for good functions to be used in the schemes, solving some optimization problems, or serving as a tool in various cryptanalytic techniques.

This shows that the design philosophies in these two areas of cryptography are very different. Hash functions cannot be related directly to one of these two parts. They are applied usually in asymmetric schemes (e.g. digital signatures), but intrinsically they have a more symmetrical nature as there are no keys or other information which could be distributed asymmetrically.

In fact, there are many examples of proposals for both kinds of design philosophies, but experience has shown that the symmetric approach by including *not too much* mathematical structure, is the more successful one. There are various examples of proposals for hash functions based on a rich mathematical structure. However, usually they failed for mainly two reasons: Often such rich structures also require time-consuming computations, hence the functions are often not very efficient. But, as described in the example of application in a digital signature scheme, efficiency is a very important design criterion of a hash function.

The other reason is that the mathematical structure often causes security weaknesses as well. In other cryptographic systems there is a well-defined object which has to be protected: the secret key. In order to achieve this, often mathematical structures prove to be very useful. In contrast to this, such a well-defined object does not exist in hash functions. There, for example, one rather has to prevent that *any* collision can be found at all. Thus, a too regular structure often opens the door to an attacker to exploit it for attacks.

Nowadays hash functions are mainly built following two design criteria: high performance and security — to measure the latter criterion usually the known attacks are tried on new proposals.

The MD4-Family. Inspired by Merkle's and Damgård's articles at Crypto '89, [Dam90, Mer90], in which proposals about constructing hash functions are described, Rivest proposed MD4 (cf. [Riv91]) one year later. MD4 is a very efficient hash function based on the principles by Merkle and Damgård.

After some cryptanalysis on MD4 had been published, which did not break it but revealed certain unexpected properties raising concerns about its security, Rivest proposed the successor MD5 (cf. [Riv92b]) in 1992. It is based on MD4, sharing many design ideas, but it is much more conservative, meaning that the focus here is much more on security than on efficiency.

In parallel, the European project RIPE (RACE Integrity Primitives Evaluation) developed an independent replacement for MD4, called RIPEMD (cf. [RIP95]). This design also shares many design criteria with MD4, or rather with Extended MD4, a variant of MD4 which was already proposed in the original article by Rivest in 1990, but did not gain much attention.

In 1993, NIST proposed the secure hash standard (cf. [FIP]), including the "secure hash algorithm", SHA, a design including some new ideas but obviously still based on MD4. After becoming aware that there is a serious "design flaw" in the SHA design, NIST had to replace the standard two years later by a new standard called SHA-1. Later on the former standard has been usually referenced as SHA-0 to avoid confusion.

Dobbertin's attacks on MD4, MD5 and RIPEMD (cf. [Dob96c, Dob97, Dob98a]) influenced further designs. In 1996, Dobbertin, Bosselaers and Preneel proposed new improved versions of RIPEMD in [DBP96]. The two main proposals are RIPEMD-128, which was mainly intended to be a replacement for the old RIPEMD having the same output size, and RIPEMD-160, a design having an increased output length and incorporating also additional ideas for increasing the security. Furthermore, two larger sized variants have been proposed, RIPEMD-256 and RIPEMD-320.

In 2002, NIST published an extension of the secure hash standard (cf. [FIP02]), including three new, much more complex versions of the secure hash algorithm, called SHA-256, SHA-384 and SHA-512. Later, in 2004, a fourth variant, SHA-224, was added to the standard.

As all these hash functions share many design criteria, they are combinedly called the *MD4-family*. For an overview of the history of the MD4-family of hash functions compare Figure 1.3. The importance of the hash functions in this family originates in their very high efficiency compared to other hash functions. Therefore, in standards which are of practical importance, mostly members of the MD4-family are included.

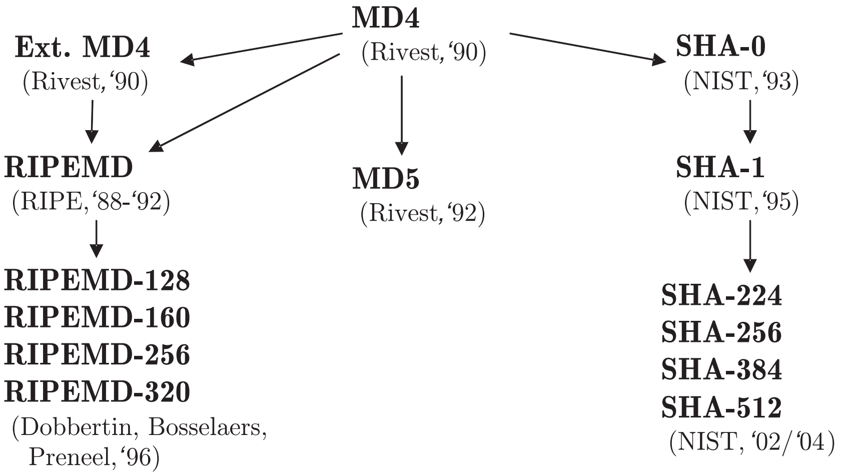

Figure 1.3: The MD4-family of hash functions.

Many of the designs of the MD4-family seem to be quite ad-hoc. However, not only the designs, but also most of the attacks on these functions proposed so far are ad-hoc attacks on specific properties of some functions, which often require some effort to be transferred to other functions, if this is possible at all.

Hence, one main subject of this thesis is to work out and structure the main techniques and methods developed for the cryptanalysis of hash functions of the MD4-family. This allows two other subjects of the thesis: to communicate these methods and describe them in an easily understandable manner, and to analyze and especially to improve upon them.

Structure of the Thesis. We start in Chapter 2 by presenting the technical background on hash functions, which is not restricted to the MD4-family. In Chapter 3 we describe the characteristic properties of the functions of this family, focussing on some important aspects. The complete detailed description of these functions can be found in Appendix A.
The main contribution of this chapter is the proposal of a new notation for

describing the hash functions of the MD4-family. The notation commonly used when describing or analyzing these functions focusses on implementation. It attaches one variable to each register used in an implementation and then usually a long list of descriptions of the steps to be processed is necessary to describe the functions. However, these steps are very similarly structured, even though their description in this notation is not, because they are applied to different registers in each step. Additionally, for the analysis one has to keep track of the contents of the registers at different times and then it is necessary to number the variables leading to a huge, unmanageable number of variables.

The new notation proposed here, makes use of a special property of the MD4-family: in each step only one register is changed and therefore we only introduce one variable per step. Additionally, this regular structure allows us to capture all steps using only one equation (and some additional parameters). Hence, this notation is much more efficient and — more important — much better adapted to the situation in cryptanalysis.

Chapter 4 provides a toolbox for cryptanalysis of MD4 type hash functions. In this chapter we cover three different aspects:

First, the connections between the different operations used in the design of the hash functions are considered. These functions include bitwise additions, additions of integers modulo 2^n and also operations like bit rotations or Boolean functions which are applied bitwise. Here the important point is that these functions are — from a mathematical point of view — rather incompatible. Therefore, in this chapter we present many observations and tools which are very useful in situations where these operations appear in combination.

An important contribution in this part is the consideration of signed bitwise differences. That means, we do *not* regard register values of n bits width *only* as elements of the vector space $\mathbb{F}_2^n$ and thus consider only bitwise differences modulo 2. We *neither* regard them *only* as elements of the ring $\mathbb{Z}_{2^n}$ and thus consider only modulo 2^n differences. Instead, we look at bitwise differences in which also the sign is taken into account, resulting in values from $\{-1, 0, 1\}$ for each bit. As shown in this section these signed bitwise differences form an important link between the other two kinds of differences enabling us to combine them in cryptanalysis.

Second, there is a section on measuring the avalanche effect. By *avalanche effect* we mean the effect, that by introducing only a little change in a computation somewhere (e.g. changing the input or some intermediate value a little), big differences are caused after only a few further steps of computation. In hash functions this effect is very important for achieving

the desired properties, because otherwise, for example, similar messages may lead to similar hash values and that might be exploited by an attacker.
To analyze the avalanche effect, we have to measure distances between intermediate values occuring in the hash function computations. Again the problem is that these intermediate values are sometimes interpreted as elements of the vector space $\mathbb{F}_2^n$ and sometimes as elements of the ring $\mathbb{Z}_{2^n}$. However, the metrics, which are commonly applied to the corresponding groups, do not fit very well to each other, i.e. there are pairs of elements which are close measured in one of the metrics and far in the other or vice versa. Therefore, we first propose a new metric which is especially adapted to the situation here and analyze its properties. Then this metric is applied to measure the avalanche for the hash functions of the MD4-family.

The third section is devoted to difference propagation. Most attacks are attacks on the collision resistance, and thus, in general, differences play an important role in these attacks, as a collision consists of two inputs having a nonzero difference which lead to outputs of the hash function having a zero difference. Like before, due to the different group operations applied in these functions, it makes sense to consider various kinds of differences here. We consider propagation of bitwise differences as well as modular differences. The main contribution here is a method, based on the observations from the first section on the connections between the different kinds of operations, which allows to analyze the propagation thoroughly.

In Chapter 5 we present the three main methods of cryptanalysis of hash functions of the MD4-family. These methods share some common aspects, e.g. a division into two parts of choosing some “differential pattern” and finding the actual collision, which are described first in this chapter. Then in the following three sections we present Dobbertin’s method, a method using $\mathbb{F}_2$-linear approximations due to Chabaud and Joux, and extended by Biham and Chen, and finally Wang’s method, supplemented with additional information to clarify the descriptions.
For Dobbertin’s method we give the first detailed description of his attack on MD5; so far only the found collision (cf. [Dob96b]) and a rough survey (cf. [Dob96c]) had been published. In addition to a thorough description we present some further analysis based on the results from Chapter 4, by applying the results on the connection of modular addition and bit rotation and the method for analyzing differential propagations. We also analyze the possibility to apply this method to other hash functions. Furthermore, we present a data structure for efficiently solving the occuring equations and representing their sets of solutions. A detailed description of the latter can be found in Chapter 6.

The attacks used to find the "breakthrough" collisions presented at the rump session of Crypto '04 (cf. [WLFY04]) are described in [WY05, WLF$^+$05] together with some general ideas of the method, which, to give credit to the main author, we refer to as *Wang's method*. In contrast to the published descriptions which focus on the actual attack rather than on the method itself, in this thesis we give a more systematic treatment of the method, based as well on [WY05, WLF$^+$05] but in particular on [WL04]. It is not only our aim to give a well-structured description of this method, but we also provide a detailed list of so-called *multi-step modifications* (to be applied in these attacks) and a proposal of an algorithm for finding differential patterns useful for this method. In [WY05, WLF$^+$05] only two examples of such multi-step modifications are given.
We conclude this chapter with some remarks on the practical relevance of the attacks including a description of how to apply the Wang attack to find colliding postscript documents displaying completely different contents. This description is based on a joint work with Stefan Lucks and has also been published in [LD05].

Chapter 6 is devoted to the description of *solution graphs*, our proposal of a new data structure, enabling us not only to solve systems of equations appearing in the attacks more efficiently but also to represent the sets of solutions in a much more compact way.

Concerning representations of the sets of solutions, solution graphs can be described as a crossing between simply enumerating all solutions — which is always possible but very inefficient — and a very efficient representation using an inherent structure of the set, like e.g. a vector space can be represented by a basis. To achieve this, solution graphs exploit "hidden" structures in the sets of solutions which are often inherent in the observed systems of equations due to their origin, e.g. coming from the attacks on MD4-family hash functions.

The main aspect which is responsible for this "hidden structure" is that most of the functions appearing in these systems of equations are T-functions, i.e. functions in which the k-th output bit depends only on the least significant k input bits. The concept of T-functions was introduced by Klimov and Shamir in a series of papers [KS02, KS03, Kli04, KS04, KS05].

In this chapter we especially analyze the connection to T-functions and extract a property of T-functions, which we call narrowness and which influences the efficiency of our algorithms significantly.

The main results of Chapter 6 have also been published in [Dau05].

Finally in Chapter 7 we conclude by summarizing the status of the different hash functions of the MD4-family and presenting some perspectives.

Technical details on the applied notation are given throughout the thesis where necessary and summarized in a list of symbols on page 165.

Acknowledgements. First of all I wish to thank my supervisor Hans Dobbertin for directing me to this very interesting subject and giving me valuable insight through his great experience in this area of research. He presented interesting problems to me for consideration and motivated me to descend deeply into the world of bits and bytes of hash functions — sometimes even beyond the point of no return.

Thanks to all the members of the CITS Research Group and to all other people I met at the Ruhr-University Bochum for the nice working atmosphere. Especially I would like to thank Gregor Leander and Patrick Felke, for many interesting and fruitful discussions but also for being trustworthy colleagues, and Elena Prokhorenko and Rita Schröer for technical and administrative support.

Very special thanks go to Tanja Lange for supporting me in manifold ways, for motivating and pushing me, for sharing her experience with me, for being a nice person to chat with, and, of course, for many detailed and very helpful comments.

I also gratefully acknowledge partial support by the European Union within the European Network of Excellence ECRYPT for giving me the possibility of interacting with international experts. The Bundesamt für Sicherheit in der Informationstechnik supported part of the research presented in this thesis within the project "Analyse der Hashfunktionen der SHA-Familie".

Furthermore, I really appreciated visiting the Technical University of Denmark and the University of Mannheim. I would like to thank all the people there for their hospitality and for fruitful discussions and remarks on my research, especially Lars Knudsen, Matthias Krause, Frederik Armknecht, Stefan Lucks, Dirk Stegemann and Charlotte Vikkelsø.

Also many thanks to all the students in my exercise groups who have been considerate with me despite having to suffer from various changing moods. Special thanks go to Michael Kallweit for providing his "digital fingerprint" and to Kathrin Kirchner and Anna Troost for supporting me mentally as well as actively in manifold ways.

Finally I wish to thank my parents for supporting me my whole life in everything I decided to do, and Steffi for keeping me grounded and for being the one and only person without whose love and support I would have never been able to complete this thesis.

To err is human,
but to really foul things up
requires a computer.
(Paul Ehrlich)

Chapter 2

Cryptographic Hash Functions

In this chapter we give the basic technical background on hash functions.

Therefore in Section 2.1 we start with describing general definitions and properties of hash functions, not only restricted to the MD4-family. In Section 2.2 we focus on the concept of hashing by iterated compression, which is the common basic idea of all hash functions of the MD4-family.

2.1 Basic Definitions and Properties

In this section we introduce the basic definition and some fundamental properties of cryptographic hash functions in general.

Definition 2.1 (Cryptographic Hash Function). *A* ***cryptographic hash function*** *is a mapping*

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

where $\{0,1\}^*$ *denotes the set of bit strings of arbitrary length. The image* $h(X)$ *of some message* $X \in \{0,1\}^*$ *is called the* ***hash value*** *of* X*.*

This means, hash functions are mappings which allow to compress arbitrary long messages to fixed length values.

Remark 2.2. While formally a hash function maps *arbitrary* long messages, practical hash functions most often have a limit on the length of the input. However, these limits are so high (usually 2^{64} or more) that this is never a problem in practice.

As described in Chapter 1, the most important property of a hash function is, that it can be evaluated very efficiently, which allows it to be used in various contexts, cryptographical as well as non-cryptographical. The important difference between these two applications is that *cryptographic* hash functions have to fulfill much more properties than just compression and efficient evaluation. An *ideal* cryptographic hash function would behave like a random function.[1]
Unfortunately, this goal cannot be reached in practice. We can only try to simulate real randomness by using pseudorandom processes. With respect to cryptography, a measure for the quality of this simulation is to require that it is as hard to attack the pseudorandom hash function as to attack an ideal, random function. Therefore, depending on the application, cryptographic hash functions need to fulfill some or all of the following properties:

Definition 2.3. *A hash function h is called*

- ***preimage resistant**, if, given a hash value V, it is computationally infeasible to find a message X with $h(X) = V$,*

- ***second preimage resistant**, if, given a message Y, it is computationally infeasible to find a message $X \neq Y$ with $h(X) = h(Y)$,*

- ***collision resistant**, if it is computationally infeasible to find a **collision**, that is, a pair of two different messages X and X' with $h(X) = h(X')$.*

Remark 2.4. In the literature sometimes another set of names for these properties is used: *one-way* (preimage resistant), *weak collision resistant* (second preimage resistant) and *strong collision resistant* (collision resistant). These terms are given here only for the sake of completeness and will not be used in this thesis.

The mere *existence* of collisions (and also of preimages) is unavoidable. Due to the compression from arbitrary to fixed length, in general there are very many preimages to one specific hash value, i.e. also very many messages

[1] In theoretical proofs of security for cryptographical protocols hash functions are thus often modelled as so-called *random oracles*.

which map to the same hash value. The important point of Definition 2.3 is, that two such messages cannot actually be *found*.

These properties are designed to serve the special needs of some cryptographic applications. For example, hash functions used in digital signature schemes need to be collision resistant, as otherwise there would be scenarios in which Eve could cheat Alice easily and forge her signatures (cf. Chapter 1). If preimage resistance and second preimage resistance are satisfied, we say h is *one-way*. Such one-way functions are used in many cryptographical protocols to commit to certain values (cf. Chapter 1).

Let us take a look at some relations between the properties of hash functions, which follow directly from Definition 2.3.

Theorem 2.5. *If h is collision resistant, then it is second preimage resistant.*

Remark 2.6. At first glance, second preimage resistance may seem to imply preimage resistance: if we are able to find preimages for given hash values then it is possible, given some message Y with hash value $h(Y)$, to compute some preimage X which produces the same hash value. The problem is that we cannot guarantee $X \neq Y$, which is necessary to violate *second* preimage resistance. However, for the most common hash functions (which come close to behaving like an ideal hash function, i.e. randomly), we can informally consider preimage resistance and second preimage resistance as equivalent.

The most critical of the properties is collision resistance, as it follows directly from the other two. Additionally, it seems to be the property to be violated more easily which is why most attacks on hash functions focus on collision resistance. Likewise, in this thesis we will mainly focus on the question of collision resistance. However, we will remark on the other aspects when appropriate.

Clearly, the above notions of collision resistance and one-wayness are pragmatic and not formal mathematical definitions. In particular, the question, “What does *computationally infeasible* mean precisely?”, depends very much on the context. For example, computationally feasible may mean in polynomial time (asymptotically) or simple requiring less than some specified bound of computing time or space. In practice, nowadays a bound of 2^{80} computational steps is considered to be absolutely infeasible. However, in the near future this bound will probably be increased to 2^{100}. But the question when a hash function should be considered *practically broken* is hard to quantify and perhaps more of a psychological nature.

Academically, i.e. concerning theoretical attacks which may but need not have practical implications, it is widely-used to regard a hash function as “broken”, if the best known attacks on it are more efficient than generic attacks. These are attacks which do not make any use of specific properties of a certain hash function, but could also be applied to an ideal hash function.

2.1.1 Generic Attacks

The most important generic attack on the collision resistance is the *birthday attack*. The name comes from the so-called "birthday paradox", which is, strictly speaking, not a true paradox, but simply a surprising fact. Here we only state a more general form of the paradox, which says the following (for details, e.g. on the origin of the name, the interested reader is referred to [DD06]):

Theorem 2.7 (Generalized Birthday Paradox). *Given a set of t pairwise distinct elements ($t \geq 10$), the probability that, in a sample of size $k > 1.2\sqrt{t}$ (drawn with repetition) there are two equal elements, is greater than $^1/_2$.*

Proof. A proof of this theorem can be found in many textbooks and also in [DD06]. □

For cryptographic applications, it is more convenient to consider the following corollary:

Corollary 2.8 (Birthday Collisions). *Suppose that $F : \mathcal{X} \to \mathcal{Y}$ is a random function where $\mathcal{Y}$ is a set of t values. Then one expects a collision in about $\sqrt{t}$ evaluations of F.*

It is interesting to notice that the average number of evaluations mentioned in the corollary is some kind of "worst case" (for the attacker):

Remark 2.9. If there was some bias in the considered function F, that is, it is more likely that some value Y_1 is hit than some other value Y_2, then the expected number of required evaluations would decrease. For example, consider the extreme case that all the values from $\mathcal{X}$ are mapped into only one value in $\mathcal{Y}$. Then a collision is found in exactly two evaluations of F. Thus it is an important requirement for a hash function that it maps to all possible hash values with (nearly) equal probability.

However, from the (quite theoretical) description of the birthday attack, the question is whether such attacks have any practical relevance.

Due to Yuval [Yuv79] the answer to this question is clearly "yes"; the idea is the following.
Starting from two messages X_1 and X_2, whose hash values one wishes to

collide, one produces about $2^{n/2}$ variants of each of these messages, having the same content. At first glance this seems to be impractical, but it is carried out easily by finding $n/2$ independent positions, in which the actual code of the text can be changed, without changing its contents, for example, using synonymyous expressions, exchanging tabs for spaces (or vice versa), adding some blank lines, etc. Then, by a reasoning very similar to that of the birthday paradox, there is a good chance that there are two messages X'_1 (a variant of X_1) and X'_2 (a variant of X_2) which generate the same hash value.

Practically, this can even be realized without big memory requirements by using *Floyd's Algorithm* (see e.g. [MvOV96]), a method which is also often used in Pollard's ρ-method of factoring integers.

An n-bit hash function can take on 2^n values. Thus, if we compute hash values of about $\sqrt{2^n} = 2^{n/2}$ messages we expect to find a collision. Hence, using the bound of 2^{80} steps, we conclude the length of the hash value should at least be 160 bits, if not more.

For finding (second) preimages, the only generic attacks are brute-force, i.e. randomized search algorithms. This means, generic algorithms require about 2^n hash computations to find (second) preimages. Altogether we can thus state the following:

Remark 2.10. A hash function which computes hash values of a length of n bits is considered *academically broken*, if it is possible to find collisions in less than $2^{n/2}$ computations of the hash function or to compute (second) preimages in less than 2^n hash computations.

2.2 Hashing by Iterated Compression

As presented in Definition 2.1 a hash function maps *arbitrary* long messages to values of some fixed length. The only useful (and maybe only known) way to achieve this, is to use some kind of iteration. Of course, there are many possibilities to introduce such an iteration, the most important being the *MD-design principle.*

In this section we will first describe this design principle which is the basis for all MD4-family hash functions and then we will refine the notion of security for hash functions which are based on this principle.

2.2.1 The MD-Design Principle

All the hash functions of the MD4-family follow a certain design principle due to Merkle and Damgård (see [Dam90, Mer90]). The basic idea is that hashing, like encryption, should be done blockwise in some iterative way.

The principle is based on the iteration of a compression function:

Definition 2.11 (Compression Function). *A **compression function** is a mapping*

$$g : \{0,1\}^m \times \{0,1\}^l \to \{0,1\}^m$$

with $l > m \geq 1$, which can be evaluated efficiently.
Usually the $\{0,1\}^m$ part of the domain plays the role of a parameter of the compression function and such a function g with some fixed parameter IV (initial value), is denoted by g_{IV} (and maps $\{0,1\}^l \to \{0,1\}^m$).

Thus a compression function can be described as a “small hash function”, a hash function with a fixed length input. The compression ratio provided by such functions is determined by the two parameters m and l which can serve as follows: The greater m is, the greater the security of the function can be, and the greater l is, the greater is the actual compression and thus the efficiency of a hash function built on this compression function.

The idea of the Merkle-Darmgård principle is illustrated in Figure 2.1.

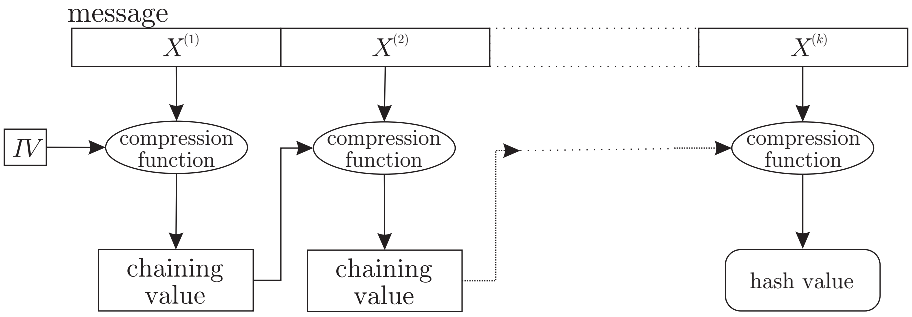

Figure 2.1: Merkle-Damgård principle, hashing by iterated compression.

Remark 2.12 (MD-Strengthening). To be able to apply it, we first must pad the message to be hashed such that its length is a multiple of m. This can be done in different ways, for example, by simply appending some zeroes, but this would allow some special attacks (see for example [MvOV96, Remark 9.32]).
To avoid these attacks other means have to be applied. There are various proposals to achieve this provably, for example by Merkle, [Mer90], and Damgård, [Dam90]. The one commonly used in the current functions is to append some block to the message during the padding which contains the binary representation of its original bitlength. Nowadays this is referred to

as *MD-strengthening.*
This method of padding is also responsible for the limit on the message length mentioned in Remark 2.2 as the binary representation of the message length has to fit into one block.

Suppose that, upon appropriate padding, a given message X is split into a sequence of k blocks of length l:

$$X = X^{(1)} \| X^{(2)} \| \ \ldots \ \| X^{(k)}$$

The hashing process is initialized with some fixed m-bit initial value IV, which is a part of the specification of the hash algorithm. The hash value of X is then computed by an iterative application of g, where the $X^{(i)}$ are taken as inputs and each output of g acts as the initial value for the next application of g:

$$\begin{array}{rcl} H_0 & := & IV, \\ H_i & := & g_{H_{i-1}}(X^{(i)}), \quad i = 1, ..., k. \end{array}$$

Therefore the H_i are also referred to as *chaining values.* The hash value of X is defined to be the output of the last application of the compression function g, i.e.

$$h(X) := H_k.$$

Hash functions which use this design principle are usually referred to as *iterated hash functions.* All the hash functions of the MD4-family are such iterated hash functions and mainly differ (besides different choices for some parameters) in the compression function they use .

Before describing detailed aspects of the various compression functions in Chapter 3, in the following section we refine the notations for properties of hash functions from Definition 2.3 to the situation of a hash function based on the MD principle.

2.2.2 Collisions and Pseudo-Collisions of the Compression Function

As compression functions can be seen as small hash functions, it is self-evident to extend the notion of collision resistance also to compression functions. But as $g_{IV}(X^{(i)})$ depends not only on the input $X^{(i)}$ but also on the parameter IV, we have to clarify the term “collision”:

Definition 2.13 (Collision of the Compression Function). *A* ***collision of the compression function*** *g consists of* one *initial value IV and two different inputs X and X' such that*

$$g_{IV}(X) = g_{IV}(X').$$

In contrast, if we also allow the IV to be modified, we speak of *pseudo-collisions*:

Definition 2.14 (Pseudo-Collision). *For a compression function g, two pairs of initial values IV, IV' and inputs X, X' for which*

$$g_{IV}(X) = g_{IV'}(X') \quad \textit{and} \quad (IV, X) \neq (IV', X').$$

constitute a ***pseudo-collision*** *of g.*

Remark 2.15. In the literature sometimes pseudo-collisions are also denoted collisions.

It is important to observe that an attack leading to collisions of the compression function is already very close to finding collisions of the hash function. What remains is to extend these attacks to the complete hash function. This could be done by extending in a way such that it is possible to prescribe the initial value in the collision of the compression function as the initial value given in the definition of the hash algorithm. Loosely speaking, we can state that collisions of the compression function are (instances of) collisions of the hash function with a *wrong initial value.*

The relation of pseudo-collisions of the compression function and collisions of the hash function is given by the fundamental theoretical result of Merkle and Damgård:

Theorem 2.16 (Merkle-Damgård Theorem). *Let g be a compression function and h be a hash function, constructed from g by using the MD-design principle with MD-strengthening. Then the following holds:*

$$g \textit{ is pseudo-collision resistant} \quad \Longrightarrow \quad h \textit{ is collision resistant}$$

Proof. Assume, we are able to find a collision of the hash function h, i.e. $X \neq X'$ with $h(X) = h(X')$.
If the lengths of X and X' are different, then so are the last blocks $X^{(k)}$ and $X'^{(k')}$ and we have found a pseudo-collision, as

$$g_{H_{k-1}}(X^{(k)}) = h(X) = h(X') = g_{H'_{k'-1}}(X'^{(k')}).$$

If $H_i = H'_i$ for all i, consider some j with $X^{(j)} \neq X'^{(j)}$, then we have a pseudo collision

$$g_{H_{j-1}}(X^{(j)}) = H_j = H'_j = g_{H'_{j-1}}(X'^{(j)}).$$

Finally, if the lengths are equal and there exists some i with $H_i \neq H'_i$, consider the maximal such i. Then we also have a pseudo collision as

$$g_{H_i}(X^{(i+1)}) = H_{i+1} = H'_{i+1} = g_{H'_i}(X'^{(i+1)}).$$

□

Unfortunately we cannot apply this theorem to verify the collision resistance of MD4-like hash functions, neither in the positive nor in the negative sense: finding pseudo-collisions of g (with independent initial values IV and IV') does not necessarily imply any hint on how to find collisions of h. We do not have the converse of the Merkle-Damgård theorem.
On the other hand, at the moment no constructions are known which would allow us to construct a practical compression function with provable pseudo-collision resistance.

2.2.3 Multicollisions

Keeping in mind the goal of an ideal hash function, which behaves randomly, multicollisions are considered, for the first time in [Mer90]. Finding a K-*collision* simply means to find K different messages which map to the same hash value, and analog definitions can be given for *K-way (second) preimages.*

The bounds on generic attacks against collision resistance and (second) preimage resistance of $2^{n/2}$ and 2^n respectively can be generalized to K-collisions and K-way (second) preimages:

Fact 2.17. *For an ideal hash function finding a K-collision requires about $2^{\frac{(K-1)n}{K}}$ hash computations while finding a K-way (second) preimage requires about $K2^n$ hash computations.*

Unfortunately for iterated hash functions (i.e. for example based on the MD-design principle), more efficient attacks are possible, as has been shown by Joux in [Jou04]. He presents a construction yielding the following result:

Theorem 2.18 ([Jou04]). *For iterated hash functions, the construction of 2^t-collisions costs (at most) t times as much as is required for ordinary 2-collisions.*

This shows that the goal of building an ideal hash function cannot be achieved by the MD-design principle. A possible solution was proposed by Lucks [Luc04], who described a modified design idea in which chaining values are used, which are longer than the hash value. Doing this makes it possible to resist generic attacks which are faster than the bounds given in Fact 2.17, but, of course, on the other hand this construction is not nearly as efficient as the MD design.

In this thesis we will focus only on the already existing hash function designs of the MD4-family (which are all based on the MD-design principle) and we will not go into the details of other proposals for the design of the iteration.

Make everything as simple as possible. . .
. . . but not simpler.
(Albert Einstein)

Chapter 3

Hash Functions of the MD4-Family

In this chapter we describe the hash functions of the MD4-family. At first we provide a framework in which all these functions can be described in a common notation. In contrast to other usual ways of describing these functions which focus on implementation, this framework is intended to offer a useful approach to cryptanalysis.

In Section 3.1 we start by describing the general structure of all compression functions of the MD4-family, along with an overview of the most important parameters of these functions. After shortly introducing the notation, we divide all functions into three subfamilies.

The actual descriptions of the compression functions can be split into two parts: the *message expansion* where the current input message block is expanded to a number of words, and the second part, where the expanded message is processed in the *step operations*. Sections 3.2 and 3.3 focus on these two main aspects of the compression functions. Here we introduce further useful notation and describe the most important aspects of all functions of the MD4-family.

3.1 Structure

The MD4-family hash functions are all based on the Merkle-Damgård design principle which is described in Section 2.2.1. Besides this they have various other commonalities in the actual construction of the compression function they use.

The basic construction is always as follows (see Figure 3.1): The compression functions use some small number of registers which are initialized with the IV. These registers are then updated in many consecutive steps which depend in some way on the message. Apart from this dependence on the message these single steps are very similar, varying only in some specific parameters.

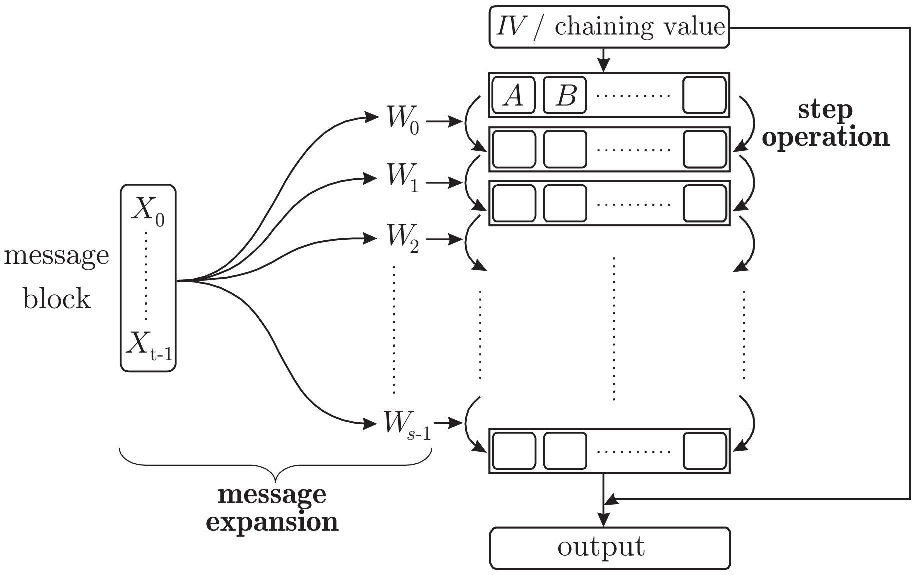

Figure 3.1: Interaction between message expansion and step operations.

The final output of the compression function is computed from the final state of the registers and the IV (usually by a simple addition). Thereby an easy way of inverting the compression function, by simply computing the single steps backwards, is blocked, as with respect to inversion this operation provides some kind of feedback. This is necessary to prevent simple meet-in-the-middle attacks.

3.1.1 Parameters

Details about the general stucture of each of the functions are described by several parameters. Table 3.2 at the end of this section provides an overview of these. The general parameters of all hash functions based on the MD-principle – namely the output length n of the hash function, the output length m of the compression function and the length l of the input message blocks – have already been introduced in Sections 2.1 and 2.2.

Nearly all the functions of the MD4-family are optimized in their design for software implementations on 32-bit processors, i.e. the word size is $w = 32$ bits.[1] The only exceptions are SHA-384 and SHA-512 where $w = 64$.

Other parameters are deduced from these: the number of input words X_i for the compression function, denoted by t, is simply given by $t = {}^{l}\!/\!_{w}$. Similarly, as the output length m of the compression function is also the length of the chaining value, which in MD4-family hash functions is used to initialize the registers, the number r of registers is $r = {}^{m}\!/\!_{w}$.

Another important parameter is the number of steps s (in the literature sometimes also referred to as *rounds*). This can be viewed as a security parameter, in the sense that the security of the function can be increased to some extent by increasing s, but, of course, for the cost of a loss of efficiency. Reducing a hash function in order to check the practicability of a theoretical attack is often done by decreasing s, as this may have the least chance on violating any design criteria of the function.

Details on the parameters for specific hash functions can be found in Table 3.1. Note that SHA-0 and RIPEMD-0 were originally proposed without the "-0" extension. Later this was added to avoid confusion. The notation "2×" in the table refers to the fact that these functions use two (mostly) independent lines of computations in parallel.

With the exception of the functions SHA-224 and SHA-384 the output length m of the compression function and the output length n of the hash function are the same, $m = n$. In SHA-224 and SHA-384 an additional final step truncates some bits decreasing the actual hash value.

3.1.2 Notation

The current input message block $X^{(j)}$ is split into t message words denoted by $X_0, \ldots, X_{t-1}$. From these message words in the message expansion, the values $W_0, \ldots, W_{s-1}$ are computed, where W_i serves as input for the i-th step in the step operation part. The actual content of the registers during the step operation is denoted R_i (for details see Section 3.3).

Notation and parameters are summarized in Table 3.2 (including notation introduced in the following sections).

[1] In some sections we also use n to denote the word size, especially used in connection with $\mathbb{Z}_{2^n}$ or $\mathbb{F}_2^n$.

	word size w (in bit)	registers	output length $n(m)$ (in bit)	steps s	reference
MD4	32	4	$4 \cdot 32 = 128$	$3 \cdot 16 = 48$	[Riv91, Riv92a]
Ext.MD4	32	2×4	$8 \cdot 32 = 256$	2×48	[Riv91]
MD5	32	4	$4 \cdot 32 = 128$	$4 \cdot 16 = 64$	[Riv92b]
RIPEMD-0 $^{(*)}$	32	2×4	$4 \cdot 32 = 128$	2×48	[RIP95]
RIPEMD-128	32	2×4	$4 \cdot 32 = 128$	2×64	[DBP96, Bos]
RIPEMD-160	32	2×5	$5 \cdot 32 = 160$	2×80	[DBP96, Bos]
RIPEMD-256	32	2×4	$8 \cdot 32 = 256$	2×64	[DBP96, Bos]
RIPEMD-320	32	2×5	$10 \cdot 32 = 320$	2×80	[DBP96, Bos]
SHA-0 $^{(*)}$	32	5	$5 \cdot 32 = 160$	80	[FIP]
SHA-1	32	5	$5 \cdot 32 = 160$	80	[FIP02]
SHA-224	32	8	$7 \cdot 32 = 224$ $(8 \cdot 32 = 256)$	64	[FIP02]
SHA-256	32	8	$8 \cdot 32 = 256$	64	[FIP02]
SHA-384	64	8	$6 \cdot 64 = 384$ $(8 \cdot 64 = 512)$	64	[FIP02]
SHA-512	64	8	$8 \cdot 64 = 512$	80	[FIP02]

Table 3.1: Parameters of the hash functions of the MD4-family. (The functions marked $^{(*)}$ have originally been proposed without the "-0" extension. This was added later to avoid confusion.)

l	length of input message blocks (in bits)
m	output length of compression function (in bits)
n	output length of hash function (in bits)
r	number of registers
s	number of steps
t	number of input message words X_i
w	word size (in bits)
σ_k	permutation used in k-th round
f_i	bitwise Boolean function applied in step i
K_i	constant used in the step operation in step i
R_i	content of register changed in step i after step i
s_i	amount of bit rotation applied in step i
W_i	input word for step i, dependent on message words
X_i	message word

Table 3.2: Notation of parameters and variables.

3.1.3 Subfamilies

There are two main distinctions which divide the hash functions of the MD4-family into subfamilies:

- The *message expansion* is either done by some *roundwise permutations* (cf. Section 3.2.1) or by a *recursive expansion* (cf. Section 3.2.2).
- The number of *parallel lines* of computation.

Depending on these two properties the functions are classified as one of the following three subfamilies:

- **MD-family:**
  MD4 and MD5 constitute this subfamily which is characterized by using roundwise permutations and only one line of computations.
- **RIPEMD-family:**
  The RIPEMD-family consists of RIPEMD-$\{0, 128, 160, 256, 320\}$ and also Extended MD4. These functions apply roundwise permutations for the message expansion but the crucial difference is that they use two parallel lines of computations.
- **SHA-family:**
  The SHA-family compression functions SHA-{0, 1, 224, 256, 384, 512} again use only one line of computations, but here the message expansion is achieved by some recursively defined function.

Remark 3.1 (HAVAL). Note that there is another interesting hash function, HAVAL (see [ZPS93]), which is quite similar in its design according to the general structure. But since there are also important differences, as for example the use of Boolean functions which are very different to those used in the other functions mentioned here (cf. Definition 3.4), we do not consider HAVAL as part of the MD4-family and therefore as part of the main focus of this thesis.
Nevertheless, some analyzes of HAVAL may give valuable insight also for MD4-family hash functions and therefore we will come back to HAVAL where appropriate.

In the following we will describe further differences between the different compression functions and other important aspects of these functions in more detail. We start with describing the two different kinds of message expansions in Section 3.2 and present some different aspects of the step operations in Section 3.3.

Detailed specifications of all the compression functions can be found in Appendix A.

3.2 Message Expansion

The message expansion part describes how the values W_i which serve as inputs for the step operations are computed from the current input message block $X^{(j)}$. Therefore the current message block $X^{(j)}$ is first split into t words $X_0 \ldots, X_{t-1}$ and then from these the s values $W_0, \ldots, W_{s-1}$ are computed.

In the MD4-family there are two different methods to do this, using "roundwise permutations" or recursively defined functions:

3.2.1 Roundwise Permutations

This kind of message expansion is used in (Extended) MD4, MD5, all the RIPEMD variants and also in HAVAL. *Roundwise permutation* means that the steps are grouped together to $^s/_t$ rounds (usually 3-5) consisting of t steps each (usually 16). Then, in every round, each of the message words is used in exactly one step as input W_i.

This means that in order to describe this message expansion exactly, it suffices to give one permutation σ_k for each round k such that

$$W_{kt+i} = X_{\sigma_k(i)}. \tag{3.1}$$

Or in other words: X_i is used as input in the steps

$$kt + \sigma_k^{-1}(i), \quad k \in \{0, \ldots, {}^s/_t - 1\}.$$

A complete overview of actual permutations used in the different hash functions of the MD4-family is given in Appendix A. In this section we will only have a short look at some examples in order to analyze some properties.

Usually the permutations are chosen in a way such that there are as few patterns as possible, e.g. such that there are no two words X_i, X_j which are applied in two consecutive steps (or two steps with some other fixed small distance) more than once. This is done to provide a good diffusion.
On the other hand many designers like to have a simple "explanation" for the chosen parameters, just to avoid some critics from claiming that there are hidden properties in permutations which are said to be chosen "randomly". For example the MD5 message expansion uses the following quite simple formulas:

$$\begin{array}{rclcrcl} \sigma_0(i) & = & i & \qquad & \sigma_2(i) & = & 3i + 5 \bmod 16 \\ \sigma_1(i) & = & 5i + 1 \bmod 16 & & \sigma_3(i) & = & 7i \bmod 16 \end{array}$$

In the design of RIPEMD-0 this concept of avoiding patterns was violated by using exactly the same permutations for both lines of computations. Consequently, there are two attacks — by Dobbertin [Dob97] (see Section 5.2) and by Wang et al. [WLFY04, WLF+05] (see Section 5.4) — which exploit this bad message expansion.

When constructing the new generation of RIPEMD-functions, the designers tried to learn from Dobbertin's attacks and avoid properties that have been exploited. They first define a permutation ρ as in Table 3.3 and another permutation π by setting $\pi(i) = 9i + 5 \bmod 16$ and then assign the permutations $\sigma_k^L := \rho^k$ to the left line of computations and $\sigma_k^R := \rho^k \pi$ to the right line.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho(i)$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8
$\pi(i)$	5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12

Table 3.3: Permutation used in the message expansion of RIPEMD-160.

Dobbertin's attack makes use of the fact that in the message expansion there is a message word X_{i_0} which is applied in steps which are quite close to one another and thus the area which is affected by differences in this word can be limited quite well (cf. Section 5.2 for details). Therefore the permutations above were chosen such that "message words which are close in the left half are at least seven positions apart in the right half" as it is said in [DBP96].

For example, one property of the chosen permutations (which can be tested simply by checking all possible choices of i) can be stated as:

Fact 3.2.

$$\begin{aligned}\min_i(\quad & \min_k((\sigma_{k+1}^L)^{-1}(i) - (\sigma_k^L)^{-1}(i)) \\ + \quad & \min_k((\sigma_{k+1}^R)^{-1}(i) - (\sigma_k^R)^{-1}(i)) \quad) \geq 13\end{aligned}$$

3.2.2 Recursive Message Expansions

In contrast to the roundwise permutations, the functions of the SHA-family use some recursive message expansions: after choosing the starting values

$$W_i = X_i, \quad i = 0, \ldots, t-1,$$

the following W_i are computed recursively from the preceding W_i.

This kind of message expansion has the advantage that the diffusion is increased, as (nearly) all the W_i depend on (nearly) all the X_i. Thus, if just one of the words of the message is changed, many of the steps in the second phase of the computation are affected. This makes it much harder to control what is happening during the step operation.

In SHA-0, the following recursive definition is used:

$$W_i = W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}, \quad i \in \{16, \dots, 79\}. \tag{3.2}$$

But there is a flaw in this definition: the k-th bit of each word W_i is only influenced by the k-th bits of the preceding W_i. This means that this expansion causes much less diffusion than desireable and also than possible with similar constructions. For example, the revised version SHA-1 uses the following message expansion which achieves much more diffusion:

$$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})^{\lll 1}, \quad i \in \{16, \dots, 79\}. \tag{3.3}$$

This means the message expansion of SHA-0 was just supplemented with a rotation by one bit, which causes the bits of these values to be mixed much more than before. However, the message expansion of SHA-0 and SHA-1 are still quite similar:

Fact 3.3. *If the input message words are restricted to $X_i \in \{0, -1\}$, then (3.2) and (3.3) behave identically.*

But there is another, more important weakness with this kind of message expansion which could have been easily avoided. This recursion is still linear with respect to $\mathbb{F}_2$ and thus much better analyzable than some non-linear recursion.

In the later SHA versions NIST decided to mix even more by using not only $\oplus$ and bit rotations, but also introducing bit shifts and modular additions in the message expansion. Therefore, two auxiliary functions

$$\begin{aligned} \sigma_0(x) &:= (x^{\ggg 7}) \oplus (x^{\ggg 18}) \oplus (x^{\gg 3}) \\ \sigma_1(x) &:= (x^{\ggg 17}) \oplus (x^{\ggg 19}) \oplus (x^{\gg 10}) \end{aligned}$$

are defined which introduce much more inter-bit diffusion than the one bit rotation of SHA-1. Additionally the use of the bit shift ($\gg$) in these functions (instead of a another rotation $\ggg$) avoids rotation symmetry which could pose a potential weakness.

SHA-256 (and thus also SHA-224, which mainly consists of applying SHA-256 and ignoring 32 output bits) uses the following expansion rule:

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, \quad i \in \{16, \dots, 63\}.$$

Hence, due the usage of modular additions instead of the $\oplus$ used before, this message expansion is no longer linear with respect to $\mathbb{F}_2$.

The message expansion of SHA-512 (and SHA-384) is similar, but uses 64-bit words and some adjusted σ_0 and σ_1, as described in Appendix A.

3.3 Step Operation

The step operations represent the main part, so to speak the heart, of every compression function. They should be easily implementable and, even more important, very efficient, but nevertheless at the same time they must provide good diffusion and be hard to analyze.

Regarding efficiency there are two main aspects in the step operations of the MD4-family to achieve this:

- Only one of the registers (or at most two) is changed in each step. This is especially important to make efficient software implementations possible, in which one cannot compute different registers in parallel.
- The step operations are built of very efficient and simple basic operations.

To achieve the last point, the single steps in the compression process of all hash functions of the MD4-family are based on the following operations on words:

- bitwise Boolean operations,
- integer additions modulo 2^w (short: *modular additions*),
- bit shifts and rotations.

These operations have been chosen, because on the one hand they can be computed very efficiently on modern computer architectures, and on the other hand the mixing of Boolean functions and addition is believed to be cryptographically strong.

3.3.1 Bitwise Applied Boolean Functions

An important part of every step operation in the MD4-family is constituted by the Boolean function, which is applied bitwise to the whole words. This means if we have a Boolean function $\varphi : \{0,1\}^3 \to \{0,1\}$ defined on *bits*, we apply it (denoted as function $f : (\{0,1\}^w)^3 \to \{0,1\}^w$) to three *words* x, y, z, by simply defining

$$[f(x,y,z)]_i = \varphi([x]_i, [y]_i, [z]_i), \quad i \in \{0,\dots,w-1\}.$$

We call such a function f *bitwise defined* and we also write $f = \varphi$ for reasons of simplicity. In detail, the step operations of the MD4-family all use the following bitwise defined Boolean functions:

Definition 3.4.

$$\begin{aligned}
\mathbf{XOR}(x,y,z) &:= x \oplus y \oplus z \\
\mathbf{MAJ}(x,y,z) &:= xy \oplus xz \oplus yz \\
\mathbf{ITE}(x,y,z) &:= xy \oplus \overline{x}z = xy \oplus xz \oplus z \\
\mathbf{ONX}(x,y,z) &:= (x \vee \overline{y}) \oplus z = xy \oplus y \oplus z \oplus 1
\end{aligned}$$

Remark 3.5. Sometimes the (non-symmetric) functions ITE and ONX are also applied with swapped parameters. We denote this e.g. by ITE_{zxy} meaning $\mathrm{ITE}_{zxy}(x,y,z) = \mathrm{ITE}(z,x,y)$.

These functions have been chosen, as they have some important properties supporting a strong avalanche effect, which means that small differences in the registers are mapped to large differences in only very few steps (cf. Section 4.2.3).

For example, these functions are *balanced* ($|f^{-1}(0)| = |f^{-1}(1)|$). They also have a quite *high non-linearity*, i.e. the correlation between one of the functions and an arbitrary linear mapping $\mathbb{F}_2^3 \to \mathbb{F}_2$ is, according to amount, always quite small.
Further, if the bits of x, y and z are independent and unbiased, then also each output bit will be independent and unbiased and last but not least, changing one bit of the inputs changes the output in exactly half of the possible cases.

A more detailed analysis of these functions is presented in Section 4.3.1.

3.3.2 Structure and Notation

In this section we will describe the general structure of the step operations and provide some useful notation to describe and analyze them.

Conventional Notation. In common descriptions of the hash functions of the MD4-family the registers are usually labelled $A, B, C, \ldots$ and the step operations are defined e.g. by

$$F(A,B,C,D) := \ldots$$

Then the algorithm is usually given in the form "first do $F(A,B,C,D)$, then $F(D,A,B,C)$, then $F(C,D,A,B)$ ..." and so on. This might be good for a simple implementation of the operations, as for reasons of efficiency these functions should be implemented with such fixed registers.

However, this notation is quite bad for cryptanalysis: in every step the applied operation is (almost) the same but that fact is hardly describable due to the fixed letters for the registers which are swapped. Moreover, if we want to introduce some dependance on the step in the notation, for example by numbering the registers $A_i, B_i, ...$, it is always hard to keep track which of these values are equal and which is the one that is changed in this step. Additionally, if we want to transfer some fact which is described for step i to the next step $i+1$, we have to change all the letters, despite the fact that we want to describe the same fact just for another step.

Proposed Notation. In the light of the above we propose a new and different notation: for this notation we use the fact, that in most step operations mainly one register is changed, which means that it suffices to introduce only one variable per step: We denote the content of the register which is changed in step i (after the change) by R_i. This means, if the step operation really changes only one register per step the contents of the registers after step i are $R_i, R_{i-1}, .., R_{i-r+1}$, where the actual assignment to the fixed registers $(A, B, C, \ldots)$ depends on the step, but usually this is not of any importance. For consistency we assign the initial or chaining value to the variables $R_{-1}, \ldots, R_{-r}$.

When there is some additional little change in another register in each step (as for example in RIPEMD-160 another register is rotated by 10 bits in each step), then this can also be easily incorporated in this notation, directly in the step operation: if in the step operation a register is used which e.g. has been updated three steps ago (and thus would normally appear as R_{i-3} in the description of the step operation) and it has also been rotated by 10 bits to the left in the meantime then it will appear as $R_{i-3}^{\lll 10}$ in the description of the step operation.

But note that in this context the actual values in the registers are not directly given by $R_i, R_{i-1}, \ldots, R_{i-r+1}$ as described above, but also include these rotated values for the registers already changed, and also the values for the initial value have to be adapted accordingly.

Using this notation it is easy to give a simple description of the step operations as one equation which holds for all steps i, for example

$$R_i = R_{i-1} + (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i}. \qquad (3.4)$$

General Structure. Like in this example the general structure of the step operations in the MD4-family is always very similar: the dominating operation is the modular addition. The addends consist of register values, the input message word W_i, some constant value K_i and a value computed by the bitwise defined function f_i applied to three register values. In addition, bit rotations, either by some fixed amount or by some varying amount denoted by s_i, are applied to some partial results.

In most cases the actual values for K_i and s_i and the actually used Boolean function f_i are step- or round-dependent. Concrete values are given in the description of the functions below and in Appendix A.

An important common property of all the step operations described here, is that they are all bijections in some sense, i.e. any two of the values R_i, R_{i-r+1}, W_i (given fixed values for $R_{i-1}, \dots, R_{i-r+2}$) uniquely determine the third one: for R_i (given R_{i-r+1} and W_i) this is clear, and for the example above this can also be seen for the other two cases considering the following two equations, which are obtained by simply transforming (3.4):

$$\begin{array}{rcl} R_{i-4} & = & (R_i - R_{i-1})^{\ggg s_i} - f_i(R_{i-1}, R_{i-2}, R_{i-3}) - W_i - K_i \\ W_i & = & (R_i - R_{i-1})^{\ggg s_i} - f_i(R_{i-1}, R_{i-2}, R_{i-3}) - K_i - R_{i-4} \end{array}$$

This is very important to note, for example, for attacks like Dobbertin's (cf. Section 5.2).

In the following, we will describe some of the important features of each step operation. For details on the definitions of the complete hash functions, see Appendix A.

3.3.3 Specific Features

MD4. The compression algorithm has three rounds each consisting of 16 steps, i.e. in total 48 steps. The step operation of the compression in MD4 is described by

$$R_i = (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i}.$$

Here the Boolean function f_i and the constants K_i are round-dependent whereas the values for s_i change for each step but on a quite regular basis:

i	$f_i(x,y,z)$	K_i
$0,..,15$	$\mathbf{ITE}(x,y,z)$	`0x00000000`
$16,..,31$	$\mathbf{MAJ}(x,y,z)$	`0x5a827999`
$32,..,47$	$\mathbf{XOR}(x,y,z)$	`0x6ed9eba1`

j	s_{4j}	s_{4j+1}	s_{4j+2}	s_{4j+3}
$0,..,3$	3	7	11	19
$4,..,7$	3	5	9	13
$8,..,11$	3	9	11	15

MD5. The design of MD5 is quite similar to that of MD4. In addition to the ITE and the XOR function used in MD4, in MD5 also ITE_{zxy} (i.e. with swapped parameters) and ONX_{xzy} are used.

The step operation itself is very similar to that of MD4 with one primary difference: after the rotation, one further register is added:

$$R_i = R_{i-1} + (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i}$$

From a cryptanalytic point of view it is interesting to note, that this extra addition — while providing a stronger avalanche effect (see Section 4.2.3) — also poses a weakness with respect to difference propagation, which is exploited in certain attacks ([dBB94, WY05], see also Section 5.4.2 and Table 5.6).

A visualization of this step operation is given in Figure 3.2, for details on K_i and s_i see Appendix A.

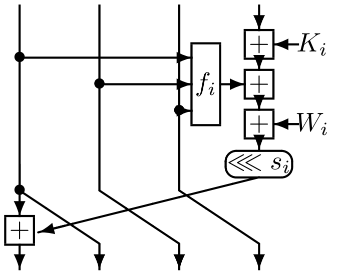

Figure 3.2: Step operation of MD5.

RIPEMD-128/RIPEMD-160. The most important difference in the compression functions of the RIPEMD functions is that the message is processed in two parallel lines. That means we have two sets of 4 (in RIPEMD-128) or 5 (in RIPEMD-160) registers, which are initialized with the same IV and updated using the same step operation, but with different constants, different amounts of rotations and, most important, different message schedules. After processing the full number of rounds, in the end the results of both lines are then combined with the IV to produce the output.

The step operation in RIPEMD-128 is the same as in MD4, and in RIPEMD-160, a very similar function

$$R_i = (R_{i-5}^{\lll 10} + f_i(R_{i-1}, R_{i-2}, R_{i-3}^{\lll 10}) + W_i + K_i)^{\lll s_i} + R_{i-4}^{\lll 10}$$

is used, in which, at the end, the value of the additional (fifth) register is added to the result. Of course, these functions, while appearing to be identical or similar to those above (for MD4/MD5), are different in the sense that in some of the steps they use other Boolean functions f_i and other constants for each step (see Appendix A).

The effect of these different designs on the avalanche effect is analyzed in Section 4.2.3.

RIPEMD-256/RIPEMD-320. These two functions use nearly the same step operation as RIPEMD-128 and RIPEMD-160 respectively. The only two differences are, that there is some limited interaction between the two parallel lines after each round (by swapping two registers each time) and the combination of the two lines in the end is omitted in order to get some output of double length (for details see Appendix A).

SHA-0/SHA-1. In SHA-0 and SHA-1 (which use identical step operations) the big difference to the MD- and RIPEMD-functions is that the bit rotations use a fixed amount and operate only on one register instead of an intermediate result:

The f_i here are ITE, MAJ and XOR, and again (as in RIPEMD-160) one additional register is updated in each step by rotating it, in this case by 30 bits. Thus we get the following description:

$$R_i := R_{i-1}^{\lll 5} + f_i(R_{i-2}, R_{i-3}^{\lll 30}, R_{i-4}^{\lll 30}) + R_{i-5}^{\lll 30} + W_i + K_i$$

SHA-224/SHA-256. The compression functions of these two hash functions are identical, apart from the prescribed IV. The only other distinction is, that when computing a hash value, in SHA-224 in the end after the last application of the compression function one truncates the output by using only the leftmost 224 bits.
This is done to offer a bigger variety of standarized sizes of hash values, because hash functions are usually a building block in some bigger cryptographical protocol and often the size of the hash value is determined by the surrounding protocol.

For these two functions the complexity of the step operation is increased by using two mechanisms: by using more registers, and by using a more complex step function.

In this more complex step operation in each step not only one but two registers are updated. Although it would be possible to describe this function using the notation proposed in Section 3.3.2, it is much more convenient to extend our notation: we simply define T_i to be the content of the other register updated in step i.

To describe the step operation, two auxiliary functions

$$\begin{aligned}\Sigma_0(x) &= (x^{\ggg 2}) \oplus (x^{\ggg 13}) \oplus (x^{\ggg 22}),\\ \Sigma_1(x) &= (x^{\ggg 6}) \oplus (x^{\ggg 11}) \oplus (x^{\ggg 25})\end{aligned}$$

are defined, which again (as already in the message expansion) provide a stronger inter-bit diffusion.

Then we can describe the step operation of SHA-224/256 by

$$\begin{aligned} T_i &= T + R_{i-4} \\ R_i &= T + \Sigma_0(R_{i-1}) + \mathrm{MAJ}(R_{i-1}, R_{i-2}, R_{i-3}) \end{aligned}$$

where T is an auxiliary variable which is defined as

$$T = T_{i-4} + \Sigma_1(T_{i-1}) + \mathrm{ITE}(T_{i-1}, T_{i-2}, T_{i-3}) + K_i + W_i$$

and used only to shorten this description.

It is interesting to note that all 64 steps of the compression function use the same Boolean function and differ only in the constant K_i and the message block W_i.

A consequence of this new, more complex, step operation is that it is slightly slower to compute than the ones described earlier, but it should also be much harder to cryptanalyze.

SHA-384/SHA-512. The situation with SHA-384 and SHA-512 is similar to that of SHA-224 and SHA-256: SHA-384 is merely a shortened version of SHA-512 with a different IV. As mentioned above, these two functions use 64-bit words instead of the common 32-bit words so as to take advantage of modern 64-bit architectures. Apart from this, the step operation is nearly the same as in SHA-256 with just some constants adjusted to the larger word size. For details, see Appendix A.

By three methods we may learn wisdom:
first, by reflection, which is noblest;
second, by imitation, which is easiest;
and third by experience, which is the bitterest.
(Confuzius)

Chapter 4

A Toolbox for Cryptanalysis

In this chapter we provide a framework necessary to describe and analyze the attacks on the hash functions of the MD4-family. Additionally, we present many tools which are either used in these attacks or useful for extending and improving them.

In Section 4.1 we analyze the relations between the different kinds of operations used in the hash functions. Therefore, after introducing some special notation in Section 4.1.1, in Section 4.1.2 we focus on the connections between the different group operations in $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$. We introduce the concept of signed bitwise differences to analyze these relations and provide tools necessary for cryptanalysis including both operations simultaneously. In Section 4.1.3 we concentrate on the relation between bit rotations and modular addition and analyze explicitly what happens when we permute the order of applying a bit rotation and a modular addition.
Section 4.2 is devoted to an analysis of the avalanche effect. As this effect is about expanding distances, in Section 4.2.1 we first analyze different metrics in particular with respect to their behaviour in connection with the identification of $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$. In Section 4.2.2 we present our own proposal of a metric, the *NAF-distance*, which is adapted to the situation that we have two different groups whose elements are identified. In Section 4.2.3 we present empirical results on the avalanche effect in the various step operations of the hash functions of the MD4-family.

Finally, Section 4.3 deals with difference propagation. In Section 4.3.1 we address the main aspects of the propagation of $\oplus$-differences, describing, in particular, their behaviour in the bitwise functions. Section 4.3.2 is devoted to the propagation of modular differences. Here we propose a method which allows to analyze the propagation of modular differences very explicitly.

4.1 Links between Different Kinds of Operations

For a pure description of the algorithms (as in Chapter 3) it suffices to say that all the hash functions considered here work on registers of width w. However, for a mathematical cryptanalysis it is necessary to interpret these contents of the registers as elements of some mathematical structure. Depending on which operations we are considering, the contents of such registers can canonically be interpreted as elements of the vector space $\mathbb{F}_2^n$ or of the ring of integers modulo 2^n, $\mathbb{Z}_{2^n}$, which both bring their own group structure.[1]

From a mathematical point of view the two group operations (denoted by $\oplus$ in $\mathbb{F}_2^n$ and by $+$ in $\mathbb{Z}_{2^n}$) are quite incompatible, e.g. regarding the orders of elements of these groups. However, as both operations (together with other similar functions) are used simultaneously in the design of the hash functions of the MD4-family, it is necessary to use both interpretations and also study the connections between them.

After introducing the necessary notations in Section 4.1.1, we deal with this problem by considering the links between the operations $+$ and $\oplus$ and between the corresponding differences in Section 4.1.2 and by analyzing the relation between modular additions and bit rotations in Section 4.1.3.

4.1.1 $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$

The mathematical objects playing the major roles when describing the construction of the hash functions of the MD4-family are the vector space $\mathbb{F}_2^n$ and the ring $\mathbb{Z}_{2^n}$. In a computer the elements of both sets are represented as elements of $\{0,1\}^n$, i.e. as bitstrings of length n, giving rise to the following natural identification of these two sets:

$$(x_{n-1},\ldots,x_0) \mapsto \sum_{i=0}^{n-1} x_i \cdot 2^i \tag{4.1}$$

[1] We use n instead of w here as this is commonly used in this context.

In some cases we will also use values from $\{-1, 0, 1\}$. The elements of $\{-1, 0, 1\}^n$ can also be interpreted as values from $\mathbb{Z}$ or $\mathbb{Z}_{2^n}$ via (4.1), but note that in this case it is not an identification, as the mapping is not injective. A bijective mapping from $\mathbb{Z}_{2^n}$ to a certain subset of $\{-1, 0, 1\}^{n+1}$ is introduced in Section 4.2.1.

To distinguish addition in $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$ we denote addition in $\mathbb{F}_2^n$ by $\oplus$ and addition in $\mathbb{Z}_{2^n}$ by $+$. The usual addition in $\mathbb{Z}$ is also denoted by $+$ and if it is necessary to make distinctions we write $+_n$ for addition modulo 2^n and $+_\mathbb{Z}$ for integer addition in $\mathbb{Z}$.

When two objects x, y from different sets can be identified (e.g. via (4.1)) we write $x = y$. When comparing two numbers (e.g. by $<$) which are computed modulo 2^n then we actually compare the integer representatives between 0 and $2^n - 1$.

Differences. To denote differences between x and x', with respect to the different group operations we use

$$\Delta^{\oplus}(x, x') := x \oplus x' \quad \text{and} \quad \Delta^{+}(x, x') := x - x'.$$

To abbreviate this notation, we shortly write $\Delta^{\oplus} x := \Delta^{\oplus}(x, x')$ (and $\Delta^{+} x$ respectively) where the second variable is built by simply adding $'$ to the first variable.
Since in many cases these differences will include many zeroes when written as objects of $\{0, 1\}^n$ we use the following short notation:

$$\begin{aligned}[i_1, \ldots, i_r] &:= (x_{n-1}, \ldots, x_0) \\ &\qquad \text{with } x_{i_1} = \ldots = x_{i_r} = 1, \quad x_j = 0 \text{ for all } j \notin \{i_1, \ldots, i_r\}.\end{aligned}$$

Sometimes the symmetry of $\Delta^{\oplus}$ is a problem. To capture the bitwise differences while maintaining the ordering of the inputs we introduce the concept of signed bitwise differences denoted by

$$\Delta^{\pm} x := \Delta^{\pm}(x, x') := (x_{n-1} - x'_{n-1}, \ldots, x_0 - x'_0).$$

To abbreviate values from $\{-1, 0, 1\}^n$, we introduce the notation $\overline{i_k}$ to denote $x_{i_k} = -1$ instead of $x_{i_k} = 1$, e.g.

$$\begin{aligned}[i_1, \overline{i_2}, \overline{i_3}, i_4, \overline{i_5}] &= (x_{n-1}, \ldots, x_0) \\ &\text{with } x_{i_1} = x_{i_4} = 1, x_{i_2} = x_{i_3} = x_{i_5} = -1, \; x_j = 0 \text{ for all } j \notin \{i_1, \ldots, i_5\}.\end{aligned}$$

4.1.2 Connections between + and ⊕

Modular addition and subtraction are not defined bitwise, but they can be written in a nearly bitwise manner with the help of one additional variable and a shift by one bit:

Lemma 4.1. *For the modular sum $A+B$ and the modular difference $A-B$ (modulo 2^n) it holds that*

$$\begin{array}{rcll} [A+B]_i &=& A_i \oplus B_i \oplus C_i^+ & \text{with } C^+ = \mathrm{MAJ}(A,B,C^+)^{\lll 1}, \\ [A-B]_i &=& A_i \oplus B_i \oplus C_i^- & \text{with } C^- = \mathrm{MAJ}(\overline{A},B,C^-)^{\lll 1}, \end{array}$$

where MAJ *stands for the bitwise defined majority function (cf. Definition 3.4).*

Here, C^+ and C^- are the vectors of carry bits which can be computed from the right to the left, i.e. start with $C_0 = 0$ and then successively compute $C_i = \mathrm{MAJ}(A_{i-1}, B_{i-1}, C_{i-1})$ for $i = 1, \ldots, n-1$ (or $C_i = \mathrm{MAJ}(\overline{A_{i-1}}, B_{i-1}, C_{i-1})$ respectively).

A trivial but quite usefal fact is the following observation:

Lemma 4.2. *For arbitrary $x \in \mathbb{Z}_{2^n}$ it holds that*

$$x + \overline{x} = x \oplus \overline{x} = (1,1,\ldots,1) = -1.$$

As some of the attacks described in Chapter 5 make use of ⊕- and modular differences in parallel, it is important to study the relations between such differences. These relations are especially important for the method of finding differential patterns, as described in Section 4.3.2.

From Lemma 4.1 we can deduce the following directly:

Theorem 4.3. *Let $x, x' \in \mathbb{F}_2^n$ with some fixed signed bitwise difference $\Delta^{\pm}x$. Then the ⊕-difference $\Delta^{\oplus}x$ and the modular difference $\Delta^{+}x$ are uniquely determined.*

Proof. For $\Delta^{\oplus}x$ this is obvious. For $\Delta^{+}x$ note that $\Delta^{+}x = x - x'$. Thus with Lemma 4.1 applied on $A := x$, $B := x'$ we get

$$\left[\Delta^{+}x\right]_i = x_i \oplus x'_i \oplus c_i \quad \text{with } c_i = \mathrm{MAJ}(x_{i-1} \oplus 1, x'_{i-1}, c_{i-1}), c_0 = 0.$$

Considering the three possible cases for each of the $\Delta^{\pm}x_i$ we see that the c_i are uniquely determined by $\Delta^{\pm}x$:

$$\begin{array}{rl} \Delta^{\pm}x_i = 0: & x_i = x'_i \Rightarrow c_{i+1} = \left\{ \begin{array}{c} \mathrm{MAJ}(0,1,c_i) \\ \mathrm{MAJ}(1,0,c_i) \end{array} \right\} = c_i \\ \Delta^{\pm}x_i = 1: & x_i = 1, x'_i = 0 \Rightarrow c_{i+1} = \mathrm{MAJ}(0,0,c_i) = 0 \\ \Delta^{\pm}x_i = -1: & x_i = 0, x'_i = 1 \Rightarrow c_{i+1} = \mathrm{MAJ}(1,1,c_i) = 1 \end{array}$$

As we also have $x_i \oplus x'_i = |\Delta^{\pm} x_i|$, $\Delta^{+} x$ is uniquely determined. □

In the sequel we will present a sequence of statements all based on the following lemma:

Lemma 4.4. *Let $x, x' \in \mathbb{F}_2^n$, $0 \leq k \leq n-1$ and define*

$$\begin{aligned} l_0^{(k)} &:= \max\{0 \leq j \leq n-k \mid x_i = 0 \textit{ for } k \leq i \leq k+j-1\}, \\ l_1^{(k)} &:= \max\{0 \leq j \leq n-k \mid x_i = 1 \textit{ for } k \leq i \leq k+j-1\}. \end{aligned}$$

Then it holds that

$$\Delta^{+} x = [k] \iff \begin{cases} \quad \Delta^{\pm} x = \left[k + l_0^{(k)}, \overline{k + l_0^{(k)} - 1}, \dots, \overline{k}\right], \textit{if } \ k + l_0^{(k)} < n, \\ \textit{or } \Delta^{\pm} x = \left[\overline{n-1}, \dots, \overline{k}\right], \qquad \textit{else}, \end{cases}$$

$$\Delta^{+} x = [\overline{k}] \iff \begin{cases} \quad \Delta^{\pm} x = \left[\overline{k + l_1^{(k)}}, k + l_1^{(k)} - 1, \dots, k\right], \textit{if } \ k + l_1^{(k)} < n, \\ \textit{or } \Delta^{\pm} x = [n-1, \dots, k], \qquad \textit{else}. \end{cases}$$

Proof. Consider

$$x - x' = \Delta^{+} x = \pm 2^k \Rightarrow x' = x \mp 2^k \quad \Rightarrow \quad x'_i = x_i \oplus \delta_{k,i} \oplus c^{\mp} \tag{4.2}$$

where $\delta_{i,k} = 1 \iff i = k$ (otherwise $\delta_{i,k} = 0$) and by Lemma 4.1

$$c^{+} = \mathrm{MAJ}(x, [k], c^{+})^{\lll 1} \quad \text{and} \quad c^{-} = \mathrm{MAJ}(\overline{x}, [k], c^{-})^{\lll 1}.$$

From $c_0^{+} = c_0^{-} = 0$ and the definition of $l_0^{(k)}$ and $l_1^{(k)}$ we can inductively deduce that

$$c^{+} = [k + l_1^{(k)}, \dots, k+1] \quad \text{and} \quad c^{-} = [k + l_0^{(k)}, \dots, k+1]$$

and thus by (4.2) we have $\Delta^{\oplus} x = [k + l_1^{(k)}, \dots, k]$ or $\Delta^{\oplus} x = [k + l_0^{(k)}, \dots, k]$ respectively. Combined with the definitions of $l_0^{(k)}$ and $l_1^{(k)}$ again, this completes the proof. □

This lemma shows that, when transforming a modular difference into an $\oplus$-difference, such basic modular differences as 2^k can be "expanded" to affect more than only one bit by assuring that certain conditions on the bits in the actual values hold. Further cases, which serve as important tools in attacks in which both kinds of differences are used simultaneously are given in the following corollaries:

Corollary 4.5.

$$\Delta^{+} x = [k] \ \textit{ or } \ \Delta^{+} x = [\overline{k}] \quad \Longrightarrow \quad \exists\, l \geq 0 : \ \Delta^{\oplus} x = [k + l, \dots, k].$$

Corollary 4.6. *For fixed* $\Delta^{+}x = [k]$, $0 \leq l \leq n-k-1$ *and* $x \in \mathbb{F}_2^n$ *chosen uniformly at random one has*

$$\begin{aligned} Pr(\Delta^{\pm}x = [k+l, \overline{k+l-1}, .., \overline{k}]) &= 2^{-(l+1)}, \\ Pr(\Delta^{\pm}x = [\overline{n-1}, \ldots, \overline{k}]) &= 2^{-(n-k)}. \end{aligned}$$

For fixed $\Delta^{+}x = [\bar{k}]$ *the following probabilities hold:*

$$\begin{aligned} Pr(\Delta^{\pm}x = [\overline{k+l}, k+l-1, .., k]) &= 2^{-(l+1)}, \\ Pr(\Delta^{\pm}x = [n-1, \ldots, k]) &= 2^{-(n-k)}. \end{aligned}$$

Thus in both cases ($\Delta^{+}x = [k]$ *or* $\Delta^{+}x = [\bar{k}]$) *and fixed* $0 \leq l < n-k-1$ *we have*

$$\begin{aligned} Pr(\Delta^{\oplus}x = [k+l, \ldots, k]) &= 2^{-(l+1)}, \\ Pr(\Delta^{\oplus}x = [n-1, \ldots, k]) &= 2^{-(n-k-1)}. \end{aligned}$$

Up to here, we only considered basic modular differences of the form $[k]$ or $[\bar{k}]$. However, in situations where these corollaries are applied, usually more complicated differences appear as well.

Remark 4.7. Lemma 4.4 and the Corollaries 4.5 and 4.6 can be generalized to be applicable to modular differences composed of two or more such basic differences $[k]$ or $[\bar{k}]$, as long as the affected bit ranges are disjoint. For example, a modular difference of $\Delta^{+}x = [k_0, \overline{k_1}]$, $k_0 > k_1$, corresponds to signed bitwise differences of the form

$$\Delta^{\pm}x = [k_0 + l_0, \overline{k_0 + l_0 - 1}, .., \overline{k_0}, \overline{k_1 + l_1}, k_1 + l_1 - 1, .., k_1]$$

as long as $k_0 > k_1 + l_1$, where $l_i := l_i^{(k_i)}$, $i = 0, 1$, are defined as in Lemma 4.4.

This does not cover all possible cases (as e.g. $k_0 \leq k_1 + l_1$ might be possible) but it is usually enough for the situations considered here.

Sometimes in the considered attacks the question appears whether it might be useful to exchange $\oplus$-differences for modular differences or vice versa. This approach can be of interest when some $\oplus$-differences and the corresponding modular differences are identical, which is characterized in the following lemma:

Lemma 4.8. *For all* $x, x' \in \mathbb{F}_2^n$ *the following equivalence holds:*

$$\begin{aligned} & \Delta^{\oplus}x = \Delta^{+}x \\ \iff \quad & \left[\Delta^{\oplus}x\right]_{n-2,\ldots,0} = \left[\Delta^{\pm}x\right]_{n-2,\ldots,0} \\ \iff \quad & (\Delta^{\oplus}x_i = 1 \;\Rightarrow\; x_i = 1), \textit{for all } i \in \{0, \ldots, n-2\} \end{aligned}$$

Proof. By Lemma 4.1, $\Delta^{\oplus}x = \Delta^{+}x$ is equivalent to

$$x_i \oplus x'_i = \left[\Delta^{\oplus}x\right]_i = \left[\Delta^{+}x\right]_i = x_i \oplus x'_i \oplus c_i \qquad \Longleftrightarrow \; c_i = 0$$

for all $i \in \{0, \ldots, n-1\}$ where $c_0 = 0, c_i = \text{MAJ}(\overline{x_{i-1}}, x'_{i-1}, c_{i-1})$. This is equivalent to (for $i \in \{1, \ldots, n-1\}$)

$$0 = \overline{x_{i-1}}x'_{i-1} = (x_{i-1} \oplus 1)(x_{i-1} \oplus \Delta^{\oplus}x_{i-1}) = (x_{i-1} \oplus 1)\Delta^{\oplus}x_{i-1}$$

which directly proves the claim. □

Thus the probability that (for some fixed $\oplus$-difference) these differences are equal depends directly on the Hamming weight of the given difference:

Corollary 4.9. *Let $\Delta = \Delta^{\oplus}x$ be fixed. Then for $x \in \mathbb{F}_2^n$ chosen uniformly at random it holds that*

$$Pr(\Delta^{\oplus}x = \Delta^{+}x) = 2^{-w_H(\Delta_{n-2,\ldots,0})},$$

where w_H denotes the Hamming weight (cf. Example 4.17).

4.1.3 Modular Addition and Bit Rotations

In this section we use the notation

$$A = [\![A_L|_i A_R]\!] \tag{4.3}$$

in the sense, that for $A = (a_{n-1}, \ldots, a_0)$ we have $A_L = (a_{n-1}, \ldots, a_i)$ and $A_R = (a_{i-1}, \ldots, a_0)$ or, to be more exact (e.g. for the case that A, A_L or A_R include some formulas), that

$$\lfloor A \rfloor_n = \lfloor A_L \rfloor_{n-i} \cdot 2^i + \lfloor A_R \rfloor_i$$

where we use identification (4.1) and where $\lfloor x \rfloor_k$ denotes the number $\tilde{x}$ with $0 \leq \tilde{x} \leq 2^k - 1$ and $\tilde{x} = x \bmod 2^k$. If A is given, and we want to actually define A_L and A_R by (4.3), then we suppose A_L and A_R to be the uniquely determined values between 0 and $2^{n-i} - 1$ (and $2^i - 1$ respectively).

Furthermore, we denote characteristic functions by $\mathbb{1}$, where

$$\mathbb{1}(x) = \begin{cases} 1, & \text{if } x \text{ is true,} \\ 0, & \text{if } x \text{ is false.} \end{cases}$$

Unless otherwise stated, in this section let $0 < k < n$ and A and B be two integers with $0 \leq A, B < 2^n$ and

$$A = [\![A_L|_{n-k} A_R]\!] \quad \text{and} \quad B = [\![B_L|_{n-k} B_R]\!] .$$

Then from the notation introduced above we can easily deduce the following lemma:

Lemma 4.10.

$$\begin{aligned}
A^{\lll k} &= \llbracket A_R |_k A_L \rrbracket \,, \\
A + B &= \left\llbracket A_L + B_L + c_R^+ |_{n-k} A_R + B_R \right\rrbracket \,, \\
A - B &= \left\llbracket A_L - B_L - c_R^- |_{n-k} A_R - B_R \right\rrbracket \,,
\end{aligned}$$

where

$$\begin{aligned}
c_R^+ &= \mathbb{1}(A_R +_{\mathbb{Z}} B_R \geq 2^{n-k}), \\
c_R^- &= \mathbb{1}(A_R < B_R)
\end{aligned}$$

are the carry bits coming from the right half of the computation.

Using this lemma we can exactly describe which disturbances appear when the order of applying a rotation and an addition is reversed:

Theorem 4.11.

$$(A + B)^{\lll k} - (A^{\lll k} + B^{\lll k}) = \left\llbracket -c^+ |_k c_R^+ \right\rrbracket \,,$$

where

$$\begin{aligned}
c^+ &= \mathbb{1}(A +_{\mathbb{Z}} B \geq 2^n), \\
c_R^+ &= \mathbb{1}(A_R +_{\mathbb{Z}} B_R \geq 2^{n-k})
\end{aligned}$$

are the carry bits coming from the full and from the right side addition.

Proof. With Lemma 4.10 we have

$$(A + B)^{\lll k} = \left\llbracket A_R + B_R |_k A_L + B_L + c_R^+ \right\rrbracket$$

and

$$(A^{\lll k} + B^{\lll k}) = \left\llbracket A_R + B_R + c_L^+ |_k A_L + B_L \right\rrbracket$$

with $c_L^+ = \mathbb{1}(A_L +_{\mathbb{Z}} B_L \geq 2^k)$. Thus again by Lemma 4.10

$$\begin{aligned}
&(A + B)^{\lll k} - (A^{\lll k} + B^{\lll k}) \\
&\quad = \left\llbracket -c_L^+ - \mathbb{1}(\lfloor A_L + B_L + c_R^+ \rfloor_k < \lfloor A_L + B_L \rfloor_k) \; |_k c_R^+ \right\rrbracket \,.
\end{aligned}$$

It remains to show that $c_L^+ + \mathbb{1}(\lfloor A_L + B_L + c_R^+ \rfloor_k < \lfloor A_L + B_L \rfloor_k) = c^+ = \mathbb{1}(A +_{\mathbb{Z}} B \geq 2^n)$. We have

$$A +_{\mathbb{Z}} B \geq 2^n \;\Leftrightarrow\; \left(A_L +_{\mathbb{Z}} B_L \geq 2^k\right) \vee \left(\left(A_L +_{\mathbb{Z}} B_L = (2^k - 1)\right) \wedge \left(c_R^+ = 1\right)\right)$$

and

$$\begin{aligned}&\lfloor A_L + B_L + c_R^+\rfloor_k < \lfloor A_L + B_L\rfloor_k \\ &\qquad \overset{c_R^+\in\{0,1\}}{\Longleftrightarrow} \quad \big(c_R^+ = 1\big) \wedge \big(\lfloor A_L + B_L\rfloor_k = 2^k - 1\big)\end{aligned}$$

and thus

$$\begin{aligned} c^+ &= \mathbb{1}(A +_{\mathbb{Z}} B \geq 2^n) \\ &= \mathbb{1}(A_L +_{\mathbb{Z}} B_L \geq 2^k) + \mathbb{1}(A_L + B_L = 2^k - 1) \cdot c_R^+ \\ &= \mathbb{1}(A_L +_{\mathbb{Z}} B_L \geq 2^k) + \mathbb{1}(\lfloor A_L + B_L + c_R^+\rfloor_k < \lfloor A_L + B_L\rfloor_k). \end{aligned}$$

□

This theorem describes the errors occuring when we modify modular equations by exchanging the order of an addition and a bit rotation.

In practical cases often some or all of the variables in the equations above are supposed to be chosen uniformly at random and then we are interested in the probabilities that each of the cases occurs:

Corollary 4.12. *Let $P_{\alpha,\beta}$ (with $\alpha, \beta \in \{0,1\}$) be the probability that*

$$(A + B)^{\lll k} - (A^{\lll k} + B^{\lll k}) = [\![-\alpha|_k\beta]\!]\,.$$

1. *If we suppose A to be fixed and B to be chosen uniformly at random, then the corresponding probabilities are*

$$\begin{aligned} P_{0,0} &= 2^{-n}(2^{n-k} - A_R)(2^k - A_L), \\ P_{0,1} &= 2^{-n}A_R(2^k - A_L - 1), \\ P_{1,0} &= 2^{-n}(2^{n-k} - A_R)A_L, \\ P_{1,1} &= 2^{-n}A_R(A_L + 1). \end{aligned}$$

2. *If we suppose A and B to be chosen independently and uniformly at random, then*

$$\begin{aligned} P_{0,0} &= {}^1\!/\!_4(1 + 2^{-(n-k)} + 2^{-k} + 2^{-n}), \\ P_{0,1} &= {}^1\!/\!_4(1 - 2^{-(n-k)} - 2^{-k} + 2^{-n}), \\ P_{1,0} &= {}^1\!/\!_4(1 + 2^{-(n-k)} - 2^{-k} - 2^{-n}), \\ P_{1,1} &= {}^1\!/\!_4(1 - 2^{-(n-k)} + 2^{-k} - 2^{-n}). \end{aligned}$$

Proof. The proof follows by counting the number of possible values for B (for fixed A) fulfilling the equations from Theorem 4.11 which determine α and β. Note that

$$A +_{\mathbb{Z}} B \geq 2^n \iff A_L +_{\mathbb{Z}} B_L +_{\mathbb{Z}} c_R^+ \geq 2^k.$$

□

From this theorem we can see various things: for example, if A and B are both chosen at random then the most probable difference is 0. However, if the bit rotation is somewhat medium-sized (i.e. $k \approx \frac{n}{2}$) then the probability for each of the four cases is about 1/4, but for small rotations (to the left or to the right) the probabilities for some of the cases increase noticeably (to up to about 3/8).

Additionally, if one of the values is fixed, there are many cases (e.g. with very small or very large values of A_R and/or A_L) in which one or two of the four cases are dominant, sometimes with probability ≈ 1, a fact which can often be used to simplify equations (cf. Section 5.2.3.1).

Analogously to Theorem 4.11 and Corollary 4.12 we can prove the following about modular subtraction:

Theorem 4.13.

$$(A - B)^{\lll k} - (A^{\lll k} - B^{\lll k}) = c^- 2^k - c_R^-$$

where

$$\begin{aligned} c^- &= \mathbb{1}(A < B), \\ c_R^- &= \mathbb{1}(A_R < B_R) \end{aligned}$$

are the carry bits coming from the full and from the right side subtraction.

Corollary 4.14. *Let $P_{\alpha,\beta}$ (with $\alpha, \beta \in \{0,1\}$) be the probability that*

$$(A - B)^{\lll k} - (A^{\lll k} - B^{\lll k}) = \alpha 2^k - \beta.$$

1. *If we suppose A to be fixed and B to be chosen uniformly at random, then the corresponding probabilities are*

$$\begin{aligned} P_{0,0} &= 2^{-n}(A_R + 1)(A_L + 1), \\ P_{0,1} &= 2^{-n}(2^{n-k} - A_R - 1)A_L, \\ P_{1,0} &= 2^{-n}(A_R + 1)(2^k - A_L - 1), \\ P_{1,1} &= 2^{-n}(2^{n-k} - A_R - 1)(2^k - A_L). \end{aligned}$$

2. *If we suppose A to be chosen uniformly at random and B to be fixed, then the corresponding probabilities are*

$$\begin{aligned} P_{0,0} &= 2^{-n}(2^k - B_L)(2^{n-k} - B_R), \\ P_{0,1} &= 2^{-n}(2^k - B_L - 1)B_R, \\ P_{1,0} &= 2^{-n}B_L(2^{n-k} - B_R), \\ P_{1,1} &= 2^{-n}(B_L + 1)B_R. \end{aligned}$$

3. *If we suppose A and B to be chosen independently and uniformly at random, then*

$$\begin{aligned} P_{0,0} &= {}^1\!/\!_4(1 + 2^{-(n-k)} + 2^{-k} + 2^{-n}), \\ P_{0,1} &= {}^1\!/\!_4(1 - 2^{-(n-k)} - 2^{-k} + 2^{-n}), \\ P_{1,0} &= {}^1\!/\!_4(1 + 2^{-(n-k)} - 2^{-k} - 2^{-n}), \\ P_{1,1} &= {}^1\!/\!_4(1 - 2^{-(n-k)} + 2^{-k} - 2^{-n}). \end{aligned}$$

Proof. The basic observation for this proof is that

$$\begin{aligned} & A < B \\ \iff & (A_L < B_L \text{ or } (A_L = B_L, A_R < B_R)) \\ \iff & A_L - c_R^- < B_L. \end{aligned}$$

□

4.2 Measuring the Avalanche Effect

Usually in a hash function the step operation is designed to have a strong avalanche effect. That means that “small” differences in the registers lead to “big” differences after only a few steps.

In this section we will analyze and try to quantify the avalanche effect in the step operations used in the MD4-family. To compare these effects for different step operations we first have to specify what *small* and *big* differences are, i.e. we have to choose some useful metrics on $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$.

4.2.1 Different Metrics

We will follow the common approach to define specific metrics by weights:

Definition 4.15 (Weight). *Let G be a group. A **weight** (or **quasi-norm**) is a mapping $w : G \to \mathbb{R}_{\geq 0}$ with the following properties:*

1. $w(x) = 0 \iff x = 0$

2. $w(-x) = w(x)$

3. $w(x + y) \le w(x) + w(y)$

From this definition the following lemma directly follows:

Lemma 4.16. *Let w be a weight for a group G. Then the mapping $d : G \times G \to \mathbb{R}_{\ge 0}$ defined by $d(x, y) := w(x - y)$ is a metric on G.*

For the groups relevant here, $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$, there are two typical weights which are used most often:

Example 4.17. On $\mathbb{F}_2^n$ there is the **Hamming weight**

$$w_H(x) := \sharp\{i \mid x_i \neq 0\}$$

and the induced **Hamming distance** $d_H(x, y) := w_H(x \oplus y)$.

Example 4.18. On $\mathbb{Z}_{2^n}$ we can define the **modular weight**

$$w_M(x) := |x'|,$$

where $x \equiv x' \bmod 2^n$ and $-2^{n-1} < x' \le 2^{n-1}$, and this induces the **modular distance** $d_M(x, y) := w_M(x - y)$.

In (4.1) we introduced the interpretation of binary vectors as elements of $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$. Due to this identification we cannot apply both metrics independently. A measure for the compatibility of two metrics under some given embedding of one metric space into another is the *distortion* (cf. e.g. [Lin02]):

Definition 4.19 (Distortion). *Given two metric spaces (M_1, d_1) and (M_2, d_2) and an injective mapping $\varphi : M_1 \to M_2$, the* ***distortion*** *of φ is defined to be*

$$\text{distortion}(\varphi, d_1, d_2) = \sup_{x \neq y \in M_1} \frac{d_2(\varphi(x), \varphi(y))}{d_1(x, y)} \cdot \sup_{x \neq y \in M_1} \frac{d_1(x, y)}{d_2(\varphi(x), \varphi(y))}$$

Usually this distortion is used to compare different embeddings φ for fixed metric spaces. In our context, we want to compare different pairs of metrics for a fixed embedding. As the range of possible values for the distortion depends very much on the metrics, we define a normed version of the distortion which takes on values between 0 and 1. This is possible as we are dealing only with finite spaces here.

Definition 4.20 (Normed Distortion). *Under the conditions of Definition 4.19 where additionally the metric spaces are finite, we define the **normed distortion** to be*

$$\operatorname{distortion}^*(\varphi, d_1, d_2) := \\ \operatorname{distortion}(\varphi, d_1, d_2) \cdot \frac{\min\limits_{x_1 \neq y_1 \in M_1} d_1(x_1, y_1) \cdot \min\limits_{x_2 \neq y_2 \in M_2} d_2(x_2, y_2)}{\max\limits_{x_1, y_1 \in M_1} d_1(x_1, y_1) \cdot \max\limits_{x_2, y_2 \in M_2} d_2(x_2, y_2)}.$$

In the sequel we will now check whether the two metrics from Examples 4.17 and 4.18 fit together with respect to identification (4.1):

Proposition 4.21. *For $(\mathbb{F}_2{}^n, d_H)$, $(\mathbb{Z}_{2^n}, d_M)$, and φ denoting the identification given by (4.1) we have*

$$\operatorname{distortion}^*(\varphi, d_H, d_M) = 1.$$

Proof. From Example 4.17 and Example 4.18 we have

$$w_H(\mathbb{F}_2{}^n) \subseteq \{0, 1, 2, \ldots, n\} \quad \text{and} \quad w_M(\mathbb{Z}_{2^n}) \subseteq \{0, 1, 2, \ldots, 2^{n-1}\}.$$

Thus we have upper bounds

$$\begin{aligned} &\max_{x \neq y \in \mathbb{F}_2{}^n} \frac{d_M(\varphi(x), \varphi(y))}{d_H(x, y)} \leq \frac{2^{n-1}}{1} = 2^{n-1} \\ \text{and} \quad &\max_{x \neq y \in \mathbb{F}_2{}^n} \frac{d_H(x, y)}{d_M(\varphi(x), \varphi(y))} \leq \frac{n}{1} = n \end{aligned}$$

on the two factors of distortion(φ, d_1, d_2) and because of

$$\begin{array}{ll} d_H(0, 2^{n-1}) = 1 & d_H(0, -1) = n \\ d_M(0, 2^{n-1}) = 2^{n-1} & d_M(0, -1) = 1 \end{array}$$

they are actually achieved. □

This shows that the identification we are using has maximal distortion regarding these two metrics. Thus, it is more convenient to use some metrics which are more adapted to the situation, that we have two different representations and also two different and quite incompatible operations. Therefore in the next section we introduce the notion of the *NAF-distance.*

4.2.2 NAF-Weight and NAF-Distance

Definition 4.22 (Non-Adjacent Form). *A* ***non-adjecent form (NAF)*** *of $(x_{n-1}, \dots, x_0) \in \{0,1\}^n$ is a signed representation*

$$(\alpha_n, \dots, \alpha_0) \in \{-1, 0, 1\}^{n+1}$$

such that

$$\sum_{i=0}^{n-1} x_i 2^i = \sum_{i=0}^{n} \alpha_i 2^i \ (\textit{in } \mathbb{Z}) \quad \textit{and} \quad \alpha_i \alpha_{i+1} = 0 \textit{ for } 0 \le i \le n-1.$$

Given a binary representation $x = (x_{n-1}, \dots, x_0) \in \{0,1\}^n$, a NAF $(\alpha_n, \dots, \alpha_0)$ of x can computed by the following algorithm:

Algorithm 4.23 (Computation of NAF).

```
Input: binary representation $(x_{n-1}, \dots, x_0) \in \{0,1\}^n$ of $x$
Output: NAF $(\alpha_n, \dots, \alpha_0)$ of $x$
$i := 0$, $\tilde{x} := x$
while $i \le n$
    if $[\tilde{x}]_0 = 0$ then
        $\alpha_i = 0$
    else
        $\alpha_i = 1 - 2\,[\tilde{x}]_1$
    end if
    $\tilde{x} := (\tilde{x} - \alpha_i)^{\gg 1}$
    $i := i + 1$
end while
```

This algorithm assures the existence of a NAF representation for arbitrary integers. The following lemma gives its uniqueness:

Lemma 4.24. *For each $x = (x_{n-1}, \dots, x_0) \in \mathbb{F}_2^n$ there is exactly one NAF.*

Proof. A proof of this lemma can be found in [Rei60]. □

Remark 4.25. It is also possible to define the NAF for elements of $\mathbb{Z}_{2^n}$ (via identification (4.1)). To maintain the uniqueness, it is important to note, that to determine the NAF of some $x \in \mathbb{Z}_{2^n}$ it is necessary to first identify it with an integer between 0 and $2^n - 1$ and then compute its NAF.

Remark 4.26. As we only consider NAFs of non-negative values, the most significant nonzero α_k is always equal to 1, as otherwise

$$x = \sum_{i=0}^{k} \alpha_i 2^i = -2^k + \underbrace{\sum_{i=0}^{k-1} \alpha_i 2^i}_{\le 2^k - 1} < 0.$$

The advantage of using NAF representations is that they are still binary (just signed) and behave nicely with regard to $\oplus$-operations. Additionally, it is easy to compute the NAF of $-x$, given the NAF of x and more important they are very similar. Thus, the NAF seems to be a good candidate for basing some metrics on it which have lower distortion.

Due to the uniqueness of the NAF representation, our following proposal of a weight function makes sense:

Definition 4.27 (NAF-Weight).
*Let $(\alpha_n, \dots, \alpha_0)$ be the NAF of $(x_{n-1}, \dots, x_0) \in \mathbb{F}_2^n$. Then the **NAF-weight** of x is defined to be*

$$w_N(x) := \sharp\{\, i < n \mid \alpha_i \neq 0\}.$$

For $x \in \mathbb{Z}_{2^n}$ the NAF-Weight is defined via identification (4.1).

Remark 4.28. It is important to note that $w_N(x)$ is independent of α_n, only the number of nonzero entries in $\alpha_0, \dots, \alpha_{n-1}$ is counted.

To justify the name as a *weight*, we give the following proposition:

Proposition 4.29. *The NAF-weight w_N is a weight according to Definition 4.15 with respect to the group $\mathbb{Z}_{2^n}$ together with the modular addition $+$ as group operation.*

Proof. Property 1 is obviously fulfilled as we compute modulo 2^n and the only α_i which may be nonzero, if $w_N(x) = 0$, is α_n.
For Property 2, note that $-x = 2^n - x$ in $\mathbb{Z}_{2^n}$. Thus, if $(\alpha_n, \dots, \alpha_0)$ is the NAF of x, we have

$$-x = 2^n - \sum_{i=0}^{n} \alpha_i 2^i = (1 - \alpha_n)2^n + \sum_{i=0}^{n-1} (-\alpha_i)2^i.$$

Hence, if $\alpha_{n-1} = 0$ the NAF of $-x$ is given by $(1 - \alpha_n, -\alpha_{n-1}, \dots, -\alpha_0)$ and it follows that $w_N(-x) = w_N(x)$. The case $\alpha_{n-1} = -1$ is impossible (cf. Remark 4.26).
If $\alpha_{n-1} = 1$, then $\alpha_n = \alpha_{n-2} = 0$. Thus, we have

$$-x = 2^n - \sum_{i=0}^{n} \alpha_i 2^i = \underbrace{2^n - 2^{n-1}}_{=2^{n-1}} + \sum_{i=0}^{n-2} (-\alpha_i)2^i$$

and the NAF of $-x$ is $(0, \alpha_{n-1}, -\alpha_{n-2}, \dots, -\alpha_0)$ which also yields the same NAF-weight as x.

For Property 3, let $x, y \in \mathbb{Z}_{2^n}$ and denote their NAFs by (α_i) and (β_i) respectively. If we define $\gamma_i = \alpha_i + \beta_i$, we can detect the following properties:

$$\begin{aligned} \gamma_i &\in \{-2,-1,0,1,2\} && (4.4)\\ x+y &= \sum_{i=0}^{n} \gamma_i 2^i && (4.5)\\ \sharp\{i < n \mid \gamma_i \neq 0\} &\leq w_N(x) + w_N(y) && (4.6) \end{aligned}$$

To prove that Property 3 holds, we have to modify the γ_i in a way such that (4.4) reads

$$\gamma_i \in \{-1,0,1\} \tag{4.7}$$

and such that

$$\gamma_{i+1}\gamma_i = 0 \tag{4.8}$$

but without violating (4.5) and (4.6):

Therefore, first for all i with $|\gamma_i| = 2$ we note that $\gamma_{i+1} = 0$ due to the NAF property of (α_i) and (β_i). Hence, we can set $\gamma_{i+1} = \frac{\gamma_i}{2}$, $\gamma_i = 0$. This does not change (4.5) nor (4.6), but additionally fulfills (4.7).

Then, in a next step we replace pairs (γ_{i+1}, γ_i) with $\gamma_{i+1}\gamma_i \neq 0$ in the following way:
If $(\gamma_{i+1}, \gamma_i) = (-1, 1)$ or $(\gamma_{i+1}, \gamma_i) = (1, -1)$ we just replace it by $(0, -1)$ or $(0, 1)$ respectively. This does not violate any of the required equations.
Otherwise, (i.e. if $\gamma_{i+1} = \gamma_i$) we replace the pair by $(0, -\gamma_i)$, and to preserve (4.5) we add γ_i to γ_{i+2}. This may cause $|\gamma_{i+2}| > 1$ but similar as in the first step above we can propagate this to γ_{i+3} and if this is also too large to γ_{i+4} and so forth. If in the end γ_n gets too large this is no problem, as we compute modulo 2^n.

We repeat these steps for i from 0 to $n-1$, and after doing this we have some (γ_i) which fulfill (4.5) to (4.8), i.e. which represent a NAF for $x + y$ and as none of the steps increased the weight we can deduce Property 3 from (4.6). □

Fact 4.30. *For all $n \leq 20$, w_N is a weight according to Definition 4.15 with respect to the group $\mathbb{F}_2^n$ together with $\oplus$ as group operation.*

Proof. Properties 1 and 2 are trivially fulfilled, Property 1 for the same reasons as in Proposition 4.29 and Property 2 as in $\mathbb{F}_2^n$ we have $-x = x$.
Property 3 has (up to now only) been tested by computer simulations for values up to $n \leq 20$. □

Based on this observation, we conjecture that w_N is a weight for the group $\mathbb{F}_2^n$ for arbitrary n. By this assumption, Proposition 4.29 and due to Lemma 4.16 we can define two closely related metrics on $\mathbb{Z}_{2^n}$ and $\mathbb{F}_2^n$ respectively, which we will call the NAF-distance:

Definition 4.31 (NAF-Distance).

- *For $x, y \in \mathbb{F}_2^n$ we define the* ***$\oplus$-NAF-distance*** *of x and y*

$$d_N^{\oplus}(x,y) := w_N(x \oplus y).$$

- *For $x, y \in \mathbb{Z}_{2^n}$ we define the* ***modular NAF-distance*** *of x and y*

$$d_N^{+}(x,y) := w_N(x - y).$$

Asymptotically (for increasing n) the density of nonzero bits in NAF representations (which corresponds to the average NAF-distance of random values) is $1/3$ (cf. [MO90]). However, as we are considering quite small values of n, we prove a more precise formula for the average NAF-distance:

Proposition 4.32. *The average NAF-weight of some random $x \in \mathbb{F}_2^n$ and thus the average NAF-distance of two random $x, y \in \mathbb{F}_2^n$ is*

$$\frac{1}{3}n - (-2)^{-n} - \frac{1}{3}(-2)^{-n+1} + \frac{1}{6}\sum_{i=0}^{n-2}(-2)^{-i} \quad \approx \quad \frac{1}{3}n + \frac{1}{9}.$$

Proof. Let a_k denote the average NAF-weight of some random $x \in \mathbb{F}_2^k$. Then, clearly $a_1 = 1/2$ and $a_2 = 3/4$ as $1 = 2^0$, $2 = 2^1$ and $3 = 2^2 - 2^0$ have NAF-weight 1. For $k \geq 3$ from Algorithm 4.23 we can deduce a recursive formula for the a_k:
If $[x]_0 = 0$, then $\alpha_0 = 0$ and for the next iteration we have $\tilde{x} = x^{\gg 1}$, i.e. we expect an average weight of a_{k-1} for this x.
If $[x]_0 = 1$, then $\alpha_0 = \pm 1$ and $\tilde{x} = (x + 2\,[x]_1 - 1)^{\gg 1} = [x]_{k-1,\dots,1} + [x]_1$ is the value used in the second iteration. Obviously, this yields $\alpha_1 = 0$ and a value of $([x]_{k-1,\dots,1} + [x]_1)^{\gg 1} = [x]_{k-1,\dots,2} + [x]_1$ for the third iteration which is a value from $\mathbb{F}_2^{k-2}$ chosen uniformly at random. That means, in this case we expect an overall NAF-weight of $1 + a_{k-2}$.
As both cases appear with probability $1/2$, we get

$$a_k = \frac{1}{2}a_{k-1} + \frac{1}{2}\left(a_{k-2} + 1\right),$$

which can be transformed into the formula given in the proposition by standard techniques for solving difference equations. □

In the sequel two properties of the NAF-distance are presented, both based on corresponding computer experiments for $n \leq 20$:

Fact 4.33. *$d_N^{\oplus}(x,y) \leq 2 \cdot d_N^{+}(x,y)$ for all $x, y \in \mathbb{F}_2^n$ and all $n \leq 20$.*

Fact 4.34. *$d_N^{+}(x,y) \leq d_H(x,y)$ for all $x, y \in \mathbb{F}_2^n$ and all $n \leq 20$.*

Based on these experiments we conjecture that Fact 4.33 and Fact 4.34 hold for arbitrary n. Using this assumption we can prove the following two properties of the NAF-distance.

Proposition 4.35. *Let n be such that Fact 4.33 holds for this n, then the normed distortion of the two NAF-distances is*

$$\text{distortion}^*(\varphi, d_N^{\oplus}, d_N^{+}) = \begin{cases} 1/n, & \textit{if } n \textit{ is even,} \\ 1/n+1, & \textit{if } n \textit{ is odd,} \end{cases}$$

where φ denotes the identification given by (4.1).

Proof. Let $x = [1,3,5,\ldots]$, $y = [0,2,4,\ldots]$, then $x \oplus y = -1$ and thus $w_N(x \oplus y) = 1$. Furthermore,

$$x - y = \sum_{i=0}^{n-1} (-1)^{i+1} 2^i = \sum_{i=0}^{\lfloor n/2 \rfloor - 1} (\underbrace{2^{2i+1} - 2^{2i}}_{=2^{2i}}) \underbrace{-2^{n-1}}_{\text{only if } n \text{ odd}}$$

and thus

$$w_N(x - y) = \lceil n/2 \rceil .$$

Hence by Fact 4.33 we have $\text{distortion}(\varphi, d_N^{\oplus}, d_N^{+}) = 2 \cdot \lceil n/2 \rceil$ and as $w_N(x) \in \{0, \ldots, \lceil n/2 \rceil\}$ this means that

$$\text{distortion}^*(\varphi, d_N^{\oplus}, d_N^{+}) = \frac{2 \cdot \lceil n/2 \rceil}{\lceil n/2 \rceil^2} = \frac{2}{\lceil n/2 \rceil} = \begin{cases} 1/n, & \text{if } n \text{ is even,} \\ 1/n+1, & \text{if } n \text{ is odd.} \end{cases}$$

□

Proposition 4.36. *Let n be such that Fact 4.34 holds for this n, then the normed distortion of d_H and d_N^{+} is*

$$\text{distortion}^*(\varphi, d_H, d_N^{+}) = \begin{cases} 2/n, & \textit{if } n \textit{ is even,} \\ 2/n+1, & \textit{if } n \textit{ is odd,} \end{cases}$$

where φ denotes the identification given by (4.1).

Proof. We have $d_H(0,-1) = n$, $d_N^+(0,-1) = 1$ and thus by Fact 4.34, $\text{distortion}(\varphi, d_H, d_N^+) = n$. As $w_H(x) \in \{0,\ldots,n\}$ this means that

$$\text{distortion}^*(\varphi, d_H, d_N^+) = \frac{n}{n \cdot \lceil n/2 \rceil} = \frac{1}{\lceil n/2 \rceil} = \begin{cases} 2/n, & \text{if } n \text{ is even,} \\ 2/n+1, & \text{if } n \text{ is odd.} \end{cases}$$

□

We will use these NAF-distances mainly in the next section for quantifying the avalanche effect, but there are also some other parts in the following sections where the NAF-distance is the appropriate measure. This is the case, in particular, when we consider mainly modular differences: if we write $\Delta^+ x = [k_1, \ldots, k_s]$ allowing also $\overline{k_i}$ for denoting -1, we can write $\Delta^+ x$ using its NAF representation, and then, the NAF weight $d_N^+(x,x')$ directly corresponds to the number of entries in this notation.

4.2.3 Quantifying the Avalanche Effect

Having a strong avalanche effect could be viewed as some kind of opposite of being continuous in the sense that small differences are mapped to big differences. However, as we are working here only on discrete spaces, all functions are continuous, of course. Thus continuity itself cannot be of any help for quantifying the avalanche effect.

However, based on the idea of *Lipschitz* continuity, we can use the Lipschitz constant to define some kind of expansion factor which measures the avalanche effect. For a function $f : M \to M$ the Lipschitz constant is defined to be

$$L := \sup_{x \neq y \in M} \frac{d(f(x), f(y))}{d(x,y)}$$

and measures the distance of the images in relation to the distance of the preimages in the "worst case".

In our scenario this "worst case" is not that important. Due to the usually randomized nature of the algorithms it is often possible to ignore values behaving extremely in some sense by simply choosing other values for some random parameter. Here, it is much more important that we have a good *chance* of finding suitable values, i.e. we are interested in the *average* behaviour of the step operations. Furthermore, an important means in the attacks is to try to limit the occuring differences, i.e. usually the differences appearing during an attack are not arbitrary but most often quite small. Hence, instead of analyzing the Lipschitz constants of the involved mappings a more appropriate idea would be to look at the *average value* of

$$\frac{d(f(x), f(y))}{d(x, y)}$$

for all $x, y \in M$ with bounded distance $d(x, y) \leq \delta$.

Moreover, e.g. for $d = d_N$, the number of pairs x, y with $d(x, y) = \delta$ for small δ increases very quickly with increasing δ. Thus, the average value over all pairs with $d \leq \delta$ is dominated by those pairs with exact distance δ. Hence, it suffices to compute this average value for fixed distance $d(x, y) = \delta$.

In order to make it easier to compare the values for different functions, we additionally normalize this average value by dividing by the average distance of two randomly chosen values (cf. Proposition 4.32), which leads to the following definition:

Definition 4.37 (Avalanche Factor). *For a function $f : M \to M$ defined on some finite metric space (M, d) the* ***avalanche factor*** *of f with respect to d and an input distance of δ is*

$$\frac{\sum_{d(x,y)=\delta} d(f(x), f(y))}{\#\{(x, y) \mid d(x, y) = \delta \}} \cdot \frac{\# M^2}{\sum_{x,y} d(x, y)}.$$

To compute such avalanche factors for the step operations of the considered hash functions, we first have to extend the proposed metrics from $\mathbb{F}_2^n$ to $(\mathbb{F}_2^n)^r$ (or from $\mathbb{Z}_{2^n}$ to $(\mathbb{Z}_{2^n})^r$ resp.) as the step operations are defined on r registers. There are many ways to do this, the most natural ones being the maximum metric

$$d_{max}((x^{(1)}, \ldots, x^{(r)}), (y^{(1)}, \ldots, y^{(r)})) := \max_{1 \leq i \leq r} d(x^{(i)}, y^{(i)})$$

and the sum metric

$$d_{sum}((x^{(1)}, \ldots, x^{(r)}), (y^{(1)}, \ldots, y^{(r)})) := \sum_{i=1}^{r} d(x^{(i)}, y^{(i)}).$$

To increase the number of possible values and thus the granularity of the applied metric, we decide to use the sum metric for our analysis.

When trying to apply this to the step operations of the compression functions, of course, it is infeasible to compute the exact avalanche factors due to the large sets of pairs (x, y) with $d(x, y) = \delta$. However, as the avalanche factor is defined as an average value, it can be approximated by restricting to a randomly chosen subset of input values.

Empirical Results for some Step Operations. In this section we present our empirical results on the effects of inducing small differences. In particular, we study the behaviour over some range of steps in the various step operations.
For our analysis we had to choose two main parameters:

1. The number of *steps* we consider at the same time, i.e. as one application of the function f for which we want to compute the avalanche factor. As long as we do not want to examine the dependence on this number of steps, an appropriate choice for this parameter is the number of registers r, as after this number of steps each register is updated exactly once.

2. The size δ of the induced input differences. As explained above for the practical applications we are mainly interested in very small values here. Thus we have decided to examine the avalanche factors for choices of δ between 1 and 5.

For the reasons given earlier, we use the NAF-distance d_N^+ to measure the distances, and thus by applying Proposition 4.32 we can estimate the second (normalizing) coefficient of the avalanche factors to be ${}^{32}\!/_{3} + {}^{1}\!/_{9} = {}^{97}\!/_{9}$ (as we have $n = 32$).

In Table 4.1 we present the avalanche factors for the step operations of MD4, MD5, RIPEMD-160 and SHA-1 for different choices of δ. SHA-0 and RIPEMD-128 are not mentioned explicitly here, as their step operations are identical to those of SHA-1 and MD4 respectively. As the step operations always include some step- or round-dependent parameters we decided to make one more distinction by at least analyzing those step operations separately, in which different bitwise functions are used. Therefore, in Table 4.1 we have, for example, MD4 ITE, MD4 MAJ and MD4 XOR, indicating that we are considering steps of the first, second or third round of MD4 respectively. For RIPEMD-160 we are considering the five different rounds (XOR, ITE, ITE$_{zxy}$, ONX, ONX$_{yzx}$), but we do not distinguish between the two lines of computation as they result in (nearly) identical avalanche factors.

One thing standing out from this table is that step operations using XOR as bitwise function have much higher avalanche factors than the other step operations of the same type. Nevertheless, it is clearly not a good idea to choose XOR for every step, as due to its determistic $\Delta^{\oplus}$ propagation, such functions would be much easier analyzed (cf. Section 5.3).
Other observations to make from this table are that MD4, as expected, has the worst avalanche effect, while MD5 is much stronger in terms of the avalanche factors — even stronger than the step operations of the other

		δ 1	3	5
MD4	MAJ	0.163	0.393	0.546
MD4	ITE	0.174	0.408	0.559
RIPEMD-160	ITE	0.256	0.522	0.666
RIPEMD-160	ITE$_{zxy}$	0.251	0.528	0.677
SHA-1	ITE	0.266	0.569	0.724
SHA-1	MAJ	0.265	0.572	0.729
RIPEMD-160	ONX	0.298	0.565	0.696
MD5	ITE	0.357	0.660	0.790
RIPEMD-160	ONX$_{yzx}$	0.385	0.658	0.775
MD4	XOR	0.385	0.668	0.790
SHA-1	XOR	0.365	0.679	0.809
MD5	ITE$_{zxy}$	0.382	0.679	0.800
MD5	ONX$_{xzy}$	0.385	0.681	0.801
RIPEMD-160	XOR	0.500	0.742	0.838
MD5	XOR	0.529	0.777	0.869
Random		1.000	1.000	1.000

Table 4.1: Avalanche factors for applying r steps of the different step operations starting with an input difference of δ.

functions.
Another interesting observation is that for ITE there is not much difference (in terms of the resulting avalanche effect) between the step operations using the "original" ITE and those using ITE$_{zxy}$. In contrast to this, for the ONX functions there is a clear difference in the step operations from RIPEMD-160 using ONX to those using ONX$_{yzx}$.
In general, we can state from these values that as long as we induce only small differences of $\delta < 5$ all these functions, which consist of an application of r steps of the step operations, are far from behaving like random functions, as those would result in an avalanche factor of 1.

From Section 3.3.2 we know that all the step operations can be reversed, i.e. we can go backwards through all the steps by computing R_{i-r} from $R_{i-r+1}, \ldots, R_i$. In Table 4.2 we compare (for the example of fixed $\delta = 3$ and a fixed number of r steps) the avalanche factors of the actual step operations with those of the reversed ones.

From this table we can see that for MD4, SHA-1 and especially for MD5 the avalanche effect is much stronger when computing in the intended di-

step operation	forw.	backw.	step operation	forw.	backw.
RIPEMD-160 (ITE)	0.522	0.590	MD4 (ITE)	0.408	0.325
RIPEMD-160 (ITE$_{zxy}$)	0.528	0.529	MD4 (MAJ)	0.393	0.303
RIPEMD-160 (ONX)	0.565	0.637	MD4 (XOR)	0.668	0.382
RIPEMD-160 (ONX$_{yzx}$)	0.658	0.710	MD5 (ITE)	0.660	0.464
RIPEMD-160 (XOR)	0.742	0.630	MD5 (ITE$_{zxy}$)	0.679	0.455
SHA-1 (ITE)	0.569	0.525	MD5 (ONX$_{xzy}$)	0.681	0.497
SHA-1 (MAJ)	0.572	0.525	MD5 (XOR)	0.777	0.526
SHA-1 (XOR)	0.679	0.401			

Table 4.2: Comparison of avalanche factors for computing forward and backward (applying r steps and starting with an input difference of $\delta = 3$).

rection than when computing backwards. Especially for the step operations using the XOR operation there is a significant difference between the two directions.

This effect can be exploited, when looking for difference patterns, because based on our observations here it is much easier to control small differences when computing backwards than forwards. Hence, when looking for such difference patterns, it is better to start at the end and go backwards than vice versa.

Interestingly, for the RIPEMD-160 step operations it is different. Apart from the steps using the XOR function, for this compression function computing backward has the stronger avalanche effect.

The dependency on the number of steps, especially compared with a step operation updating one register with a totally random value is presented in Table 4.3. For the sake of clarity, in this table we restrict to the step operations using ITE and the case of $\delta = 1$. This analysis is interesting as in some attacks probabilistic methods are used which depend on the fact that there is a significant difference (in terms of resulting differences) between a random function and the application of some steps of the step operation. From the table we can see clearly, that as long as we do not apply too many steps, say less than 10, for most of the functions there is a substantial difference to a random function. The only exception is MD5, requiring a stricter limitation (to about 7).

As the results from this section show, the step operations of the different hash functions are far from behaving randomly, as long as they are considered only over a limited number of steps. This property is exploited in various

	1	2	3	4	5	6	7	8	9	10	11	12
MD4	0.04	0.06	0.11	0.17	0.26	0.38	0.50	0.63	0.74	0.83	0.90	0.94
MD5	0.04	0.10	0.20	0.36	0.54	0.71	0.85	0.93	0.98	0.99	1.00	1.00
Rand.($r = 4$)	0.27	0.51	0.76	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
RIPEMD-160	0.03	0.06	0.10	0.16	0.26	0.37	0.50	0.63	0.75	0.84	0.91	0.96
SHA-1	0.03	0.06	0.10	0.17	0.27	0.38	0.51	0.64	0.77	0.86	0.93	0.97
Rand.($r = 5$)	0.21	0.41	0.61	0.80	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 4.3: Step dependance of the avalanche factors (using the example of step operations with $f_i = \text{ITE}$ and starting with $\delta = 1$).

techniques (cf. also Section 5.2), one of which is described in the following section.

4.3 Difference Propagation

All the usual attacks on the collision resistance of some hash function focus mainly on considering differences of values instead of the values themselves. The general problem of this approach is that the difference propagation is usually not deterministic.

To clarify this, consider processing two different messages resulting in input words W_i and W_i' respectively, and denote the contents of the registers by R_i and R_i' respectively. Then, if we *only* fix the *differences* $\Delta R_{i-r}, \ldots, \Delta R_{i-1}$ for the register values and ΔW_i for the input values used in one certain step i, the difference ΔR_i of the values computed in this step, is, in general, not uniquely determined. It also depends on the actual values in the registers and not only on their differences.

However, for "small" differences, this difference behaviour is usually quite predictable and manageable. The extreme case is having zero input differences which, of course, always lead to zero output differences, as we are dealing with deterministic operations. For nonzero differences "small" may have various meanings, but a small NAF-weight (cf. Section 4.2.2), seems to be a good interpretation (cf. Section 4.3.2).

An important part in many attacks is controlling the difference propagation. That includes the questions, which input differences result in which output differences, under which conditions does this happen and how can it be controlled.

There are two approaches of dealing with the problem of difference propagation: the first (which is treated in Section 4.3.1) is to approximate all basic operations by $\mathbb{F}_2$-linear functions such that the complete (approximated) compression function becomes $\mathbb{F}_2$-linear. Hence, in this approximated function the difference propagation with respect to $\oplus$-differences is deterministic and can be easily analyzed using linear algebra. Inherent in this approach is the problem that one needs to analyze which results hold for the actual functions and not only for the approximated function in order to get results for the original function.

The other approach is to try to analyze the actual difference behaviour directly, that is, to consider mainly modular differences (but also signed bitwise differences where appropriate) and follow their propagation step by step using e.g. considerations as described in Section 4.1. In Section 4.3.2 we propose a method for analyzing and finding such differential patterns using this approach.

4.3.1 $\oplus$-Differences and Approximations

With respect to $\oplus$-differences the situation is clear, as long as we consider $\mathbb{F}_2$-linear functions f. In this case for some input difference $\overset{\oplus}{\Delta} x$ we can compute the output difference as

$$\overset{\oplus}{\Delta} f := f(x) \oplus f(x') = f(x) \oplus f(\overset{\oplus}{\Delta} x) \oplus f(x) = f(\overset{\oplus}{\Delta} x).$$

But the design of the MD4-family hash functions, as described in Chapter 3, includes many parts, which do not behave in such a simple way (i.e. are not linear with respect to $\oplus$), e.g. the modular additions and most of the bitwise defined functions from Definition 3.4.

In a way the $\oplus$-difference propagation in modular additions was already treated in Section 4.1.2: We can say that as long as we ignore carry effects, modular addition behaves $\mathbb{F}_2$-linear and $\oplus$ is a quite useful approximation for $+$.

In the following we will concentrate on the $\oplus$-difference propagation in the bitwise functions from Definition 3.4. Obviously XOR is $\mathbb{F}_2$-linear and hence only $\mathrm{ITE}, \mathrm{MAJ}$ and ONX need to be considered. Due to their bitwise definition, it suffices to look at the underlying Boolean functions.

For Boolean functions one possible way to measure how far a function is from being linear (which is called the "non-linearity") is to look at the coefficients of the Walsh spectrum. These values describe the correlation between the considered Boolean function and arbitrary $\mathbb{F}_2$-linear functions (for details, see e.g. [Rue91]). Direct computation shows that for each function $f \in \{\mathrm{ITE}, \mathrm{MAJ}, \mathrm{ONX}\}$ the maximum Walsh coefficient is 4, which means that whatever $\mathbb{F}_2$-linear (or $\mathbb{F}_2$-affine) function $l : \mathbb{F}_2^3 \to \mathbb{F}_2$ is chosen, there are always at least two values $a, b \in \mathbb{F}_2^3$ with $f(a) \neq l(a), f(b) \neq l(b)$. In fact, this is the best one can achieve for Boolean functions defined on $\mathbb{F}_2^3$.

It is also possible to determine these optimal approximating $\mathbb{F}_2$-linear functions, which achieve the bound of only two deviating values. For the three functions considered here they are

- for ITE: $y, z, x \oplus y \oplus 1$ and $x \oplus z$,
- for MAJ: x, y, z and $x \oplus y \oplus z \oplus 1$,
- for ONX: $z \oplus 1, x \oplus z, y \oplus z \oplus 1$ and $x \oplus y \oplus z \oplus 1$.

Let us now take a short look at the actual differential behaviour of these functions and their possible approximations. In Table 4.4, we can see how (or rather if) the output of the above functions changes, when some input values are changed.

$\Delta^{\oplus}x$	$\Delta^{\oplus}y$	$\Delta^{\oplus}z$	$\Delta^{\oplus}$ITE	$\Delta^{\oplus}$MAJ	$\Delta^{\oplus}$ONX
0	0	0	0	0	0
0	0	1	$x \oplus 1$	$x \oplus y$	1
0	1	0	x	$x \oplus z$	$x \oplus 1$
0	1	1	1	$y \oplus z \oplus 1$	x
1	0	0	$y \oplus z$	$y \oplus z$	y
1	0	1	$x \oplus y \oplus z$	$x \oplus z \oplus 1$	$y \oplus 1$
1	1	0	$x \oplus y \oplus z \oplus 1$	$x \oplus y \oplus 1$	$x \oplus y$
1	1	1	$y \oplus z \oplus 1$	1	$x \oplus y \oplus 1$

Table 4.4: $\oplus$-differential propagation of Boolean functions.

This table visualizes that for each of the three functions, there is only one input difference $(\Delta^{\oplus}x, \Delta^{\oplus}y, \Delta^{\oplus}z)$ for which the output difference is always zero, and, more important, there is also exactly one input difference with a guaranteed change in the output. For all the other input differences, the output difference depends on the actual input values, and if they are choosen independently and uniformly at random, an output difference of 1 appears with probability $1/2$, as is easy to see from the table.
As mentioned already in Section 3.3.1, it is usually one of the design criteria for such functions that the differential behaviour is hard to approximate.

This is achieved here as all the approximated functions have a deterministic differential propagation.

An additional criterion derived from Table 4.4 for choosing one of the approximations for ITE, MAJ or ONX given above, could be to have a 1 in the output difference of the approximation for that input difference for which the original function has a guaranteed 1. However, as is easy to check, this already holds for all the approximations given above.

For an application of this $\oplus$-difference propagation, see Section 5.3.

4.3.2 Finding Modular Differential Patterns

In this section we show how the statements from Section 4.1 can be used to estimate the difference propagation during a few steps. We propose a method enabling us e.g. to replace methods of randomized searching for differential patterns (as for example in Dobbertin's attack, cf. Section 5.2.1) by some more theoretical considerations.

The method can be applied in both directions, going forward as well as backward through the step operation. We will describe it here for the direction of going backward.

We start by fixing some *modular* differences $\Delta^{+} R_{i-r+1}, \ldots, \Delta^{+} R_i$ for the values in all the registers after step i and also some $\Delta^{+} W_i$ for the input word for step i. If we would know all the actual register values (and not only their differences) we could compute R_{i-r} directly from the equation describing the step operation, as described in Section 3.3.2. This idea can be transferred to some extent also to the corresponding differences.

The idea of the method described here is to use some of the theorems from Section 4.1 to make estimates about what values can be expected for $\Delta^{+} R_{i-r}$ and with which probabilities (or at least lower bounds for these). To achieve this, the single operations, which compose the step operation, are considered one after the other, each time estimating (or even deducing) the resulting difference.

Considering mainly modular differences Δ^{+} has the advantage that the difference propagation for modular addition $+$ is deterministic. Thus most of the basic operations of which the step operations are built can be handled without having to estimate. The only operations to take care of are bit rotations and the bitwise defined functions.

Bit Rotations. The effect of bit rotations on modular differences can be deduced from the theorems of Section 4.1.3. For example, if we have a register R with a fixed difference $\Delta^{+} R = [t]$ (i.e. $R' = R - \Delta^{+} R = R - 2^t$), we can easily compute the probabilities for the possible differences after a rotation by s bits: we have

$$[t] = \begin{cases} \llbracket 0|_{n-s} 2^t \rrbracket , & \text{if } t < n-s, \\ \llbracket 2^{t+s-n}|_{n-s} 0 \rrbracket , & \text{if } t \geq n-s, \end{cases}$$

and thus by Corollary 4.14 (applied on $A = R$, $B = \Delta^+ R$) we obtain the following corollary.

Corollary 4.38. *Let* $0 < s < n$ *and* $\Delta^+ R = [t]$ *with* $0 \leq t < n$*, and denote* $\Delta^+(R^{\lll s}) := R^{\lll s} - R'^{\lll s}$*. Then for* R *chosen uniformly at random*

$$\begin{aligned} Pr(\Delta^+(R^{\lll s}) = [t+s]) \quad &= 1 - 2^{t+s-n}, \\ Pr(\Delta^+(R^{\lll s}) = [t+s, 0]) \quad &= 2^{t+s-n} - 2^{t-n}, \\ Pr(\Delta^+(R^{\lll s}) = [t+s, \bar{s}, 0]) &= 2^{t-n} \end{aligned}$$

if $t < n-s$ *(for* $t = 0$ *the last difference should read* $[0]$*), and*

$$\begin{aligned} Pr(\Delta^+(R^{\lll s}) = [t+s-n]) \quad &= 1 - 2^{t-n}, \\ Pr(\Delta^+(R^{\lll s}) = [\bar{s}, t+s-n]) &= 2^{t-n} \end{aligned}$$

if $t \geq n-s$.

Remark 4.39. Here it is important to note that $[t]^{\lll s} = [t + s \bmod n]$. As for $[\bar{t}]$ the analogous statement does not hold, the probabilities above do *not directly* transfer to $[\bar{t}]$ (but, of course, those can also be computed from Corollary 4.14).

Thus, as long as $n - t$ or $(n - s) - t$ are not too small, it is a good approximation only to rotate the difference. For other more complicated differences, the corresponding probabilities can be computed similarly by applying Corollary 4.14.

Bitwise Defined Functions. The bitwise operations are a little bit harder to take into account. For these operations information about the actual *bitwise* input differences is required to be able to make some estimates about the output differences. The theorems in Section 4.1.2 give a good overview and some important facts about the relation between modular and (signed) bitwise differences.

The most important fact is, that from pure $\oplus$-differences one cannot get back to unique modular differences. Thus, and as we are working mainly with modular differences here, it is much better to consider *signed* bitwise differences, as (by Theorem 4.3) from some $\Delta^{\pm}x$ it is possible to come back to a uniquely determined $\Delta^{+}x$.

Another important fact is that $\Delta^{+}x = 0 \iff \Delta^{\oplus}x = 0$. That is, as long as only registers with $\Delta^{+}x = 0$ are involved in the bitwise function there is no need to consider bitwise differences at all.

Coming back to the algorithm, we can say that as soon as some R_i, for which we assume (or even know) that $\Delta^{+}R_i \neq 0$, is used in a bitwise function, we need to make assumptions about the corresponding $\Delta^{\pm}R_i$. For small differences (according to d_N) this can be done using Corollary 4.6 (see also Remark 4.7).

Following this corollary, the most probable assumption is, that a modular difference of $[k]$ or $[\bar{k}]$ also causes only an $\oplus$-difference in the k-th bit. Usually it is convenient to stick to such assumptions as besides being the most likely they also have the advantage of keeping the weight of the differences small, which is good for the probabilities in the following steps. However, sometimes other assumptions might also be useful, for example, in order to cancel other differences (cf. also Example 5.9 in Section 5.4).

Signed Differences. After having fixed the assumptions on the input differences the next step is to estimate the output difference of the bitwise function by considering its difference propagation. The first idea to do this could be, to use Table 4.4 to estimate the most probable output differences. But there are two problems occuring, when doing this:

When making assumptions about bitwise differences coming from modular differences we have to make (at least implicitly) assumptions about signed differences and this means assumptions about some of the register bit themselves. Thus, in Table 4.4 the probabilities would not be $1/2$ for the most cases (as it might seem at first glance) but many of those would be influenced by these assumptions:

Example 4.40. Consider a (signed) input difference of $(\Delta^{\pm}x, \Delta^{\pm}y, \Delta^{\pm}z) = (-1, 1, 1)$ for the ITE-function. If we only look at *unsigned* differences, we have $(\Delta^{\oplus}x, \Delta^{\oplus}y, \Delta^{\oplus}z) = (1, 1, 1)$ which by Table 4.4 leads to an output difference of $y \oplus z \oplus 1$ which is 1 with probability $1/2$ as long as we do not consider other assumptions.
But when looking at the signed differences, we see that $\Delta^{\pm}y = \Delta^{\pm}z = 1$, which means that $y = z = 1$ (and $y' = z' = 0$) and thus the output difference is *always* 1 when taking into account this *signed* input difference.

A second problem of considering only unsigned differences occurs when we want to include the bitwise output difference in the next step: then we again require a modular difference, as the next operations to consider are modular additions. But from Table 4.4 we can only compute $\oplus$-differences and from these we cannot deduce unique modular differences. Therefore, in Table 4.5 a complete overview of the propagation of signed bitwise differences for the bitwise functions used in the MD4-family is presented.

Let us take a look at an example for an application of our method described so far:

Example 4.41. Table 4.6 describes a possible (and probable) difference propagation through the steps 25 to 21 backwards in MD5.[2]

In the top row the equation describing the step operation is shown, in this case transformed to work backwards such that R_{i-4} can be computed. Furthermore, the K_i are skipped in this description as they do not contribute to any difference.

In the following rows the differences appearing during the computation are described. For example, we want to have a collision appearing in step 25. That means, we have

$$\Delta^{+} R_{25} = \ldots = \Delta^{+} R_{22} = 0$$

and this is described by the zeroes in the first row. The two zeroes in the second row, just besides the two lines, denote that also $\Delta^{+}((R_i - R_{i-1})^{\ggg s_i})$ and $\Delta^{+}(f_i(R_{i-1}, R_{i-2}, R_{i-3}))$ equal 0, as can be easily deduced.

To introduce a difference in this step we consider $\Delta^{+} W_{25} = [9]$ (which is actually the difference used in Dobbertin's attack, cf. Section 5.2.1). Altogether this leads to $\Delta^{+} R_{21} = [\bar{9}]$ which is written in the R_{i-4} column.

In the next step the difference of $[\bar{9}]$ appears in a bitwise function. Thus, we first have to make an assumption on the corresponding bitwise difference $\Delta^{\pm} R_{21}$. Corollary 4.6 says that with probability $1/2$ we have $\Delta^{\pm} R_{21} = [\bar{9}]$ and to denote that we are talking about a bitwise difference here, we write $[\bar{9}]^{\pm}$ in the table. Then, as $f_{24} = \mathrm{ITE}_{zxy}$ from Table 4.5 we can see that again with probability $1/2$ the resulting difference is 0, as written below the corresponding line.

This directly leads to $\Delta^{+} R_{20} = 0$, and for $i = 23$ similarly we get $\Delta^{+} R_{19} = 0$.

For $i = 22$ the nonzero difference $\Delta^{+} R_{21}$ appears also in the first part with the rotation, yielding a difference of $[9]$ for $\Delta^{+}(R_i - R_{i-1})$. By applying Corollary 4.38 we can see that with probability $1 - 2^{-5}$ this difference is simply rotated, which leads to $\Delta^{+} R_{18} = [27]$.

[2] See Section 5.2.3.2 for an application of this example in Dobbertin's attack.

$\Delta^{\pm}x$	$\Delta^{\pm}y$	$\Delta^{\pm}z$	$\Delta^{\pm}$XOR	$\Delta^{\pm}$ITE	$\Delta^{\pm}$MAJ	$\Delta^{\pm}$ONX
0	0	0	0	0	0	0
0	0	1	$1-2(x\oplus y)$	$1-x$	$x\oplus y$	$-1-2(x-1)y$
0	0	-1	$2(x\oplus y)-1$	$x-1$	$-(x\oplus y)$	$1+2(x-1)y$
0	1	0	$1-2(x\oplus z)$	x	$x\oplus z$	$(1-x)(2z-1)$
0	1	1	0	1	1	$-x$
0	1	-1	0	$2x-1$	0	x
0	-1	0	$2(x\oplus z)-1$	$-x$	$-(x\oplus z)$	$(1-x)(1-2z)$
0	-1	1	0	$1-2x$	0	$-x$
0	-1	-1	0	-1	-1	x
1	0	0	$1-2(y\oplus z)$	$y-z$	$y\oplus z$	$y(1-2z)$
1	0	1	0	y	1	$y-1$
1	0	-1	0	$y-1$	0	$1-y$
1	1	0	0	$1-z$	1	0
1	1	1	1	1	1	-1
1	1	-1	-1	0	1	1
1	-1	0	0	$-z$	0	$1-2z$
1	-1	1	-1	0	1	0
1	-1	-1	1	-1	-1	0
-1	0	0	$2(y\oplus z)-1$	$z-y$	$-(y\oplus z)$	$y(2z-1)$
-1	0	1	0	$1-y$	0	$y-1$
-1	0	-1	0	$-y$	-1	$1-y$
-1	1	0	0	z	0	$2z-1$
-1	1	1	-1	1	1	0
-1	1	-1	1	0	-1	0
-1	-1	0	0	$z-1$	-1	0
-1	-1	1	1	0	-1	-1
-1	-1	-1	-1	-1	-1	1

Table 4.5: Propagation of *signed* bitwise differences.

i	R_{i-4}	$=$	$(R_i - R_{i-1})^{\ggg s_i}$	$-$	$f_i\ (R_{i-1}, R_{i-2}, R_{i-3})$	$-$	W_i	Conditions
25			$0 - 0$		$0 \quad 0 \quad 0$			
	$[\bar{9}]$	$=$	0	$-$	0	$-$	$[9]$	
24					$[\bar{9}]$ $\downarrow$ $(Pr.{=}1/2)$			$[R_{21}]_9 = 0$
			$0 - 0$		$0 \quad 0 \quad [\bar{9}]^{\pm}$			
	0	$=$	0	$-$	0 (Pr.=1/2, ITE$_{zxy}$)	$-$	0	$[R_{23} \oplus R_{22}]_9 = 0$
23			$0 - 0$		$0 \quad [\bar{9}]^{\pm} \quad 0$			
	0	$=$	0	$-$	0 (Pr.=1/2, ITE$_{zxy}$)	$-$	0	$[R_{20}]_9 = 1$
22			$0 - [\bar{9}]$		$[\bar{9}]^{\pm} \quad 0 \quad 0$			
	$[27]$	$=$	$\xrightarrow[Pr. = 1 - 2^{-5}]{s_i = 14} [27]$	$-$	0 (Pr.=1/2, ITE$_{zxy}$)	$-$	0	$[R_{19}]_9 = 0$
21					$[27]$ $\downarrow$ $(Pr.{=}1/2)$			$[R_{18}]_{27} = 1$
			$[\bar{9}] - 0$		$0 \quad 0 \quad [27]^{\pm}$			
	$[\bar{0}]$	$=$	$\xrightarrow[Pr. = 1 - 2^{-23}]{s_i = 9} [\bar{0}]$	$-$	0 (Pr.=1/2, ITE$_{zxy}$)	$-$	0	$[R_{19} \oplus R_{20}]_{27} = 0$

Table 4.6: Example for computing a difference propagation.

The next step is again very similar, with the exception that in the rotation we have to deal with some other difference. By applying Corollary 4.14 we see that here with probability $1 - 2^{-23}$ this leads to a difference of $[\bar{0}]$.

Further — and more complex — examples, showing more advantages of the method proposed here and especially of the use of signed differences, can be found in Chapter 5 (cf. Examples 5.8 and 5.9).

One problem with this method is, of course, that we look at the single operations independently. That means, we do not notice if there are some dependencies causing the combined probabilities to be different. To some extent this can be dealt with by paying attention to and noting the conditions which are actually responsible for the probabilities. In the table we take this into account in the column "conditions". By comparing and combining these conditions one can find and analyze some dependencies.

In the example given here, at least the conditions coming from the bitwise functions are independent (we ignore the conditions from the rotation for the sake of readability) leading to an overall probability of about 2^{-6}.

In order to increase these probabilities, we might also think of other assumptions for the bitwise probabilities. Following Corollary 4.6 we could also assume that the modular difference of $[\bar{9}]$ yields a bitwise difference of $[\overline{9+l}, 9+l-1, \ldots, 9]^{\pm}$ for some $l > 0$ which is the case with probability $2^{-(l+1)}$. Then for f_{24} we would also get $l+1$ bitwise conditions such that the overall probability for these two steps is $2^{-2(l+1)}$. As all these assumptions lead to the same difference of $\Delta^{+}R_{20}$=0, we can combine them obtaining an overall probability for this difference of $2^{-2} + 2^{-4} + 2^{-6} + \ldots \approx \frac{1}{3}$.

When looking for such differential paths as described in this section, one always has to make decisions, mainly which output difference to choose or which assumption to make on a bitwise difference. Usually these decisions are based on heuristics: in Example 4.41 we always decided to simply take the most probable case, which is in general a quite good decision, as we are looking for probable differential paths.

Another useful heuristic is to try to keep the overall NAF-weight of the differences low, as this provides better probabilities for upcoming steps. This can be achieved e.g., by choosing differences which cancel other differences. These effects are illustrated, for instance, in Example 5.9 using a pattern which was actually applied in the Wang attack on MD4.

In Section 5.4.3 an idea is presented which is based on the method described here and which may provide some automatic way of finding differential patterns like those which are used in the Wang attack (cf. Section 5.4).

One machine can do the work of fifty ordinary men.
No machine can do the work of one extraordinary man.
(Elbert Hubbard)

Chapter 5

Cryptanalytic Methods

In this section we will describe the most important techniques of cryptanalysis to find collisions for hash functions of the MD4-family. Therefore we use the notation proposed in Chapter 3 and classify the attacks in a common scheme.

We start with a short historical overview of attacks on the MD4-family: The first analyzes of MD4 and MD5 were made by Merkle (unpublished), den Boer and Bosselaers [dBB92] and [dBB94], and by Vaudenay [Vau95].

Then Dobbertin developed a new general technique to attack MD4-like hash functions and applied it to RIPEMD-0, MD4, Extended MD4 and MD5 in a series of papers [Dob95, Dob96a, Dob96c, Dob97, Dob98a]. Later this technique was used by Kasselman and Penzhorn [KP00] to attack a reduced version of HAVAL and by van Rompay et al. [RBPV03] to break the smallest full version of HAVAL.

In 1998 Chabaud and Joux [CJ98] used a different method to show how SHA-0 could be broken in theory. Early 2004 this approach was extended by Biham and Chen who proposed the *neutral bits* technique in [BC04a]. They used it to produce near-collisions of SHA-0 (also in [BC04a]) and announced that it is feasible to find collisions for a version of SHA-1 which is reduced to 53 steps, skipping the first 20 steps in which the ITE-function is used (cf. [BC04b]). Independently, Rijmen and Oswald found a way to attack a 53-step version of SHA-1 in theory (described in detail in [RO05]), also skipping the ITE-steps. Joux et al. [JCJL04] applied the neutral-bit technique to produce real collisions of SHA-0. Their results are collected in [BCJ$^+$05].

Inspired by Dobbertin's MD5 collision, Wang developed another technique of attack. This way Wang et al. [WLFY04, WLF$^+$05, WY05] succeeded in producing real collisions for MD4, MD5, RIPEMD-0 and HAVAL. Later Wang, Yin and Yu [WYY05] extended this approach, combining it with ideas from the other approaches to present results on SHA-0 and SHA-1.

In this chapter we start by describing some structural aspects which are common to all the attacks mentioned above. In the following sections we describe in detail the three main methods due to Dobbertin, Chabaud/Joux and Wang, analyzing them and presenting some extensions and improvements. We conclude this chapter with some remarks on the practical relevance of the attacks and a proposal how to extend the Wang attack to find meaningful collisions, e.g. for postscript documents.

5.1 Structural Aspects of the Attacks

All the current techniques have in common that they can be divided into at least two parts. In the first part the general "attack strategy", a *difference pattern*, is chosen or determined. In this part many steps are usually done by hand and are of a more theoretical nature.
In contrast to this, the second part of determining the actual collisions, requires usually a lot of time-consuming computations. The techniques used here include "brute-force"-like searches, but also more sophisticated algorithms.

5.1.1 Difference Patterns

All the methods described here are attacks on the collision resistance. This means, we are looking for two messages X and X' which produce the same hash value. Therefore we have to correlate the computations that are done when computing the hash value of X and the computations for the hash value of X'.

Definition 5.1 (Difference Pattern).

- *A* ***difference pattern*** *is a sequence of differences, where each difference corresponds to one step in the computations. Each of the difference values is defined as a difference of one value appearing in the computation for X and the corresponding value from the computation for X'.*

- *We distinguish between **input differences**, i.e. differences in the messages or rather in the values W_i after the message expansion, and **output differences**, that is, differences appearing in the register values R_i after applying the step operations.*

- *Another important distinction is that between **modular differences** Δ^+ and **$\oplus$-differences** $\Delta^\oplus$.*

Usually one tries to choose the input differences such that the output differences behave in some special way. Clearly, the most important goal in doing so is that the last output differences, which correspond to the output value of the compression function, are zero differences, because that is what constitutes a collision. But generally more restrictions than this have to be imposed in order to find actual collisions in the end, and this is where the methods described here differ significantly.

Modular Differences or $\oplus$-Differences? In some sense it is more natural to consider modular differences, because most of the basic operations, composing the step operations, are modular additions. Especially the last operations applied in each step, which have the most explicit influence on the output differences, are modular additions. But, as described in Chapter 4, one cannot analyze the function completely by considering only modular additions, because there are many operations which are not compatible with such differences and it is not possible to deduce $\oplus$-differences uniquely from the modular differences (cf. e.g. Corollary 4.6).

On the other hand, it is also not possible to analyze these functions completely by considering only $\oplus$-differences. When using mainly $\oplus$-differences, the operations, which are not compatible with this kind of differences, are usually approximated by other functions which are compatible (cf. Section 4.3.1). Then the algebraic structure of a vector space can be used and it is quite easy to analyze this approximative function with linear algebra or coding theory techniques. However, one has the problem of transferring the results for the approximation to the real function.

Regarding the different attack methods described in this chapter, the choice of which differences to use is one of the main distinctions: Dobbertin (see Section 5.2) mainly uses modular differences, while Chabaud and Joux (see Section 5.3) rely heavily on $\oplus$-differences and approximations of the functions. Finally Wang et al. (see Section 5.4) again prefer using modular differences but also include $\oplus$-differences in their attack where appropriate.

5.1.2 Multiblock Collisions

Let us recall the theorem of Merkle and Damgård (Theorem 2.16): loosely speaking it states that we can construct a secure hash-function by constructing a secure compression function and applying the MD-design principle including MD-strengthening. Thus when trying to attack a hash function it might be the first idea to focus on the compression function.

And really, this is what was done in most of the attacks on hash functions of the MD4-family before 2004. These attacks all tried to find (pseudo-)collisions for the compression function. But in these attacks the only idea to generalize such an attack on the compression function to the full hash function was to avoid *pseudo*-collisions (i.e. find real collisions of the compression function with the same IV) and then additionally to include the prescribed IV of the hash function in some way in the attack. But this was not always possible, as in these attacks there was not much freedom left (cf. e.g. Dobbertin's attack, Section 5.2).

However, the MD-theorem requires *pseudo*-collision resistance. Hence, also pseudo-collisions might be helpful for constructing collisions of the hash function. In fact, the latest attacks really use specially related pseudo-collisions for constructing collisions of the hash function by a concatenation of such pseudo-collisions. Therefore we need to introduce the concept of consider near-collisions first:

Definition 5.2 (Near-Collision). *A pair of messages X and X' is called a* ***near-collision*** *of a compression function g, if the distance of the images $g(X)$ and $g(X')$ is very small.*

Remark 5.3. In this context the distance is usually measured by applying d_H which is the right measure if considering situations in which you simply ignore or remove some bits. If the goal is to find differences to be cancelled by a modular addition later on (as described below), we suggest to use the NAF-weight d_N^+ instead, as this is better adapted to this situation (cf. Section 4.2).

As long as it is possible to control the differences in some way, one can try to combine such (pseudo-)near-collisions. For example, if it is possible to find a near-collision $X^{(1)}, X'^{(1)}$ and a pseudo-collision $(IV, X^{(2)}), (IV', X'^{(2)})$ where $IV = g(X^{(1)})$ and $IV' = g(X'^{(1)})$, then these can be combined to a real collision $X^{(1)} \| X^{(2)}, X'^{(1)} \| X'^{(2)}$.

Thus, in this way an attacker gains much more freedom than by simply attacking one application of the compression function.

Examples of these kinds of attacks using multiblock collisions are the independent attacks on SHA-0 by Joux et al. [JCJL04] and on MD5 by Wang et al. [WLFY04, WY05].

5.2 Dobbertin's Method

In a series of papers [Dob95, Dob96a, Dob96c, Dob97, Dob98a], Dobbertin introduced a new method of cryptanalysis for hash functions of the MD4-family and attacked the functions MD4, Extended MD4, RIPEMD-0 and MD5 by it. Later this technique was also used by others, e.g. by Kasselman and Penzhorn [KP00] and by van Rompay et al. [RBPV03], in both cases to attack HAVAL.

The basic idea of the attack, which is an attack on the collision resistance of the compression function, is to describe the whole compression function by a system of equations. As variables these equations include the contents of the registers after various steps and the message words, while the equations are mainly derived from the step operation and the message expansion.

In general such a system of equations would be much too big and complicated to be solved. The crucial idea of Dobbertin's attack was to use several (partly very specialized) constraints to extremely simplify this system of equations such that it becomes solvable. This is possible, as potentially there are many messages mapping to the same hash value and thus the original system of equations is strongly underdefined. Additionally, the method includes many techniques which can be applied to solve these very special kinds of equations.

In the following section we describe several aspects of the method using the example of the attack on MD5, as this attack has not been published in detail yet (in contrast to those on MD4, see [Dob98a], and RIPEMD-0, see [Dob97]). The details of this attack have been reconstructed by private communication and some of Dobbertin's original notes and programs.

5.2.1 The Method

Dobbertin's method is an attack on only one application of the compression function. It uses mainly modular differences and consists of a very simple input difference pattern and only little restrictive output difference pattern:

We assume that for the colliding message pair X, X' all words coincide, with one exception, say X_{i_0} and X'_{i_0}. More precisely we require

$$\Delta^{+} X_i = 0, i \neq i_0 \quad \text{and} \quad \Delta^{+} X_{i_0} \neq 0 \text{ (with } d_N^{+}(X_{i_0}) = 1). \tag{5.1}$$

This implies, due to the message expansion by roundwise permutation used in MD5 (cf. Section 3.2.1), that only four of the W_i, namely for $i = 16k + \sigma_k^{-1}(i_0), k \in \{0, \dots, 3\}$, differ from the corresponding W'_i. Thus, only the step operations using these W_i (W'_i resp.) differ when processing X', in contrast to processing X. For the sake of readability, in this section we will denote these steps by $p_k := 16k + \sigma_k^{-1}(i_0)$.

Thus, in the first $\sigma_0^{-1}(i_0)$ steps, exactly the same step operations are used to update the same register values, meaning that the first nonzero output differences do not appear before step p_0. Similarly, the last $15 - \sigma_3^{-1}(i_0)$ steps are the same for both cases, implying that finding a collision after 64 steps is equivalent to finding a collision in step p_3. In other words, from the chosen input difference pattern we can conclude that the output difference pattern has zeroes at the first $\sigma_0^{-1}(i_0)$ and at the last $15 - \sigma_3^{-1}(i_0)$ steps.

To illustrate this behaviour, in Figure 5.1 the sequences of the contents of the registers are represented by strings starting both from the same initial value, splitting up at step p_0, coming together again in the step p_3, and ending both in the same output value.

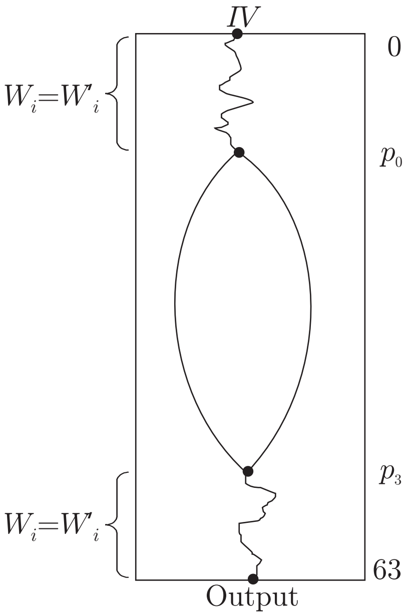

Figure 5.1: Overview of the attack on the compression function of MD5.

This shows that just by the simple restriction (5.1) on the input difference pattern, the avalanche effect is limited from taking effect over 64 rounds to only 37 rounds.

In the sequel we will describe how the actual system of equations is set up and which other constraints are used to simplify it.

5.2.1.1 Constructing the System of Equations

The variables included in the equations are R_i for the contents of the registers and the message words X_i, or rather the words W_i after the message expansion. The variables related to processing the second message X' are denoted by R_i', W_i', ... respectively.

The equations themselves are derived from the message expansion, the step operations and, of course, from the necessity to find a collision. In detail we have the following equations:

$$\begin{aligned} R_i &= R_{i-1} + (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i} && (5.2) \\ R_i' &= R_{i-1}' + (R_{i-4}' + f_i(R_{i-1}', R_{i-2}', R_{i-3}') + W_i' + K_i)^{\lll s_i} && (5.3) \end{aligned}$$

for $i = 0, \ldots, 63$ from the step operations (cf. Section 3.3.3),

$$\begin{aligned} X_i &= W_{\sigma_0^{-1}(i)} = W_{16+\sigma_1^{-1}(i)} = W_{32+\sigma_2^{-1}(i)} = W_{48+\sigma_3^{-1}(i)} && (5.4) \\ X_i' &= W'_{\sigma_0^{-1}(i)} = W'_{16+\sigma_1^{-1}(i)} = W'_{32+\sigma_2^{-1}(i)} = W'_{48+\sigma_3^{-1}(i)} && (5.5) \end{aligned}$$

for $i = 0, \ldots, 15$ from the message expansion (cf. Section 3.2.1),

$$\Delta^+ R_{60} = \Delta^+ R_{61} = \Delta^+ R_{62} = \Delta^+ R_{63} = 0 \qquad (5.6)$$

for having a collision in the end and

$$\Delta^+ X_i = 0, i \neq i_0 \quad \text{and} \quad \Delta^+ X_{i_0} = const. \neq 0 \qquad (5.7)$$

from the chosen input difference pattern.

As the goal of Dobbertin's attack is not to find *pseudo*-collisions, but real collisions of the compression function, we also require that both computations (for X and for X') start with the same IV, i.e.

$$\Delta^+ R_{-4} = \Delta^+ R_{-3} = \Delta^+ R_{-2} = \Delta^+ R_{-1} = 0. \qquad (5.8)$$

If additionally we would want to extend the attack to finding *collisions of the hash function*, we would have to fix the values not only for these $\Delta^+ R_i$, but especially also for the register values $R_{-4}, \ldots, R_{-1}$ (namely to the original prescribed initial value IV of MD5).

Applying a First Constraint. As illustrated in Figure 5.1, the first and the last steps are identical for processing X and X' and thus also the equations (5.2) and (5.3) should be the same for these steps. This can be deduced formally, directly from the equations, using the following lemma.

Lemma 5.4. *If $\Delta^{\!+} R_j = 0$ for $j \in \{i-4,\dots,i-1\}$ then*

$$\Delta^{\!+} R_i = 0 \iff \Delta^{\!+} W_i = 0.$$

If $\Delta^{\!+} R_j = 0$ for $j \in \{i-3,\dots,i\}$ it holds that

$$\Delta^{\!+} R_{i-4} = 0 \iff \Delta^{\!+} W_i = 0.$$

Proof. The first claim directly follows from the step operation equations (5.2), (5.3). The second one can be seen by transforming (5.2) to

$$R_{i-4} = (R_i - R_{i-1})^{\ggg s_i} - f_i(R_{i-1}, R_{i-2}, R_{i-3}) - W_i - K_i \tag{5.9}$$

and (5.3) analogously. □

Thus starting from (5.8) we inductively get that $\Delta^{\!+} R_i = 0$ for $i \in \{-4,\dots,p_0-1\}$. Using the second part of the lemma, the same argument can also be applied "backwards". In this case from (5.6) we get that $\Delta^{\!+} R_i = 0$ for $i \in \{p_3-3,\dots,63\}$.

This means that (5.2) and (5.3) are identical for $i \in \{0,\dots,p_0-1, p_3+1,\dots,63\}$ and thus for these i we can omit (5.3).

Removing the Dependance on W_i. Indeed, for the other steps i we cannot ignore a complete equation, but it is possible to remove at least the dependance on W_i from one of the two equations, by subtracting (5.3) from (5.2): therefore we first transform both equations by subtracting R_{i-1} and R'_{i-1} respectively, and by rotating by s_i bits to the right before we compute the difference of the transformed equations which results in

$$\begin{aligned} &f_i(R_{i-1}, R_{i-2}, R_{i-3}) - f_i(R'_{i-1}, R'_{i-2}, R'_{i-3}) \\ &\quad = \;((R_i - R_{i-1})^{\ggg s_i}) - ((R'_i - R'_{i-1})^{\ggg s_i}) + \underbrace{R'_{i-4} - R_{i-4}}_{-\Delta^{\!+} R_{i-4}} + \underbrace{W'_i - W_i}_{-\Delta^{\!+} W_i}. \end{aligned} \tag{5.10}$$

In fact, this equation does not include W_i (or W'_i) anymore, as by (5.7) together with (5.4) and (5.5) all $\Delta^{\!+} W_i$ are constant.

This shows that just by adding the constraint (5.7) we can get rid of many equations and reduce the size of the system to solve noticeably.
Even more, since any collision in step p_3 will lead to a collision in step 63, we do not need to consider (5.2) and (5.3) for $i > p_3$ at all, if we replace (5.6) by

$$\Delta^{\!+} R_{p_3-3} = \Delta^{\!+} R_{p_3-2} = \Delta^{\!+} R_{p_3-1} = \Delta^{\!+} R_{p_3} = 0. \tag{5.11}$$

Similarly, as long as we are only interested in collisions for the compression function and not necessarily for the hash function, we can replace (5.8) by

$$\Delta^{+} R_{p_0-4} = \Delta^{+} R_{p_0-3} = \Delta^{+} R_{p_0-2} = \Delta^{+} R_{p_0-1} = 0 \tag{5.12}$$

and then also ignore (5.2) and (5.3) for $i < p_0$, using the following idea: once we have found a solution for the remaining equations we use the included values for the W_i to compute backwards through these steps till we come to some values for $R_{-4}, R_{-3}, R_{-2}, R_{-1}$ which we take as the common initial value for our collision of the compression function.

Inner Collisions. In order to increase the number of steps for which we do not have to take into account both equations (5.2) and (5.3) and to restrict the avalanche effect much more, Dobbertin uses another constraint, he considers *inner collisions*.

As inner collision he denotes a range of steps $q_0, \ldots, q_1$ such that at the beginning there is a zero difference in all registers, i.e. $\Delta^{+} R_{q_0-4} = \ldots = \Delta^{+} R_{q_0-1} = 0$ and at the end there is a collision, i.e. $\Delta^{+} R_{q_1-3} = \ldots = \Delta^{+} R_{q_1} = 0$, but there are nonzero differences in between.

Due to Lemma 5.4 such steps q_0, q_1 can only appear at positions i with $\Delta^{+} W_i \neq 0$, i.e. in the steps p_0, p_1, p_2, p_3 in the MD5 attack.

Using this terminology we can say that by (5.11) and (5.12) we require an inner collision from step p_0 to step p_3. In order to restrict the avalanche effect more, Dobbertin adds another constraint of having not only one but two inner collisions, from step p_0 to p_1 and from p_2 to p_3. This is easily achieved by requiring

$$\Delta^{+} R_{p_1-3} = \Delta^{+} R_{p_1-2} = \Delta^{+} R_{p_1-1} = \Delta^{+} R_{p_1} = 0. \tag{5.13}$$

This leads to a situation (as shown in Figure 5.2) where in steps $p_1 + 1, \ldots, p_2 - 1$ exactly the same things are happening (the output difference pattern has again zeroes in these steps).

With the help of this constraint we get another range of steps in which we can ignore (5.3) and thus simplify the system of equations once again.

Remark 5.5. A very important effect of this constraint is that the parts in which the state sequences drift apart get much shorter. Thus the avalanche effect is also reduced significantly, because, as we know from Section 4.2.3, it takes some more steps for it to have the full effect. Hence, at least intuitively this should make the equations easier to solve.

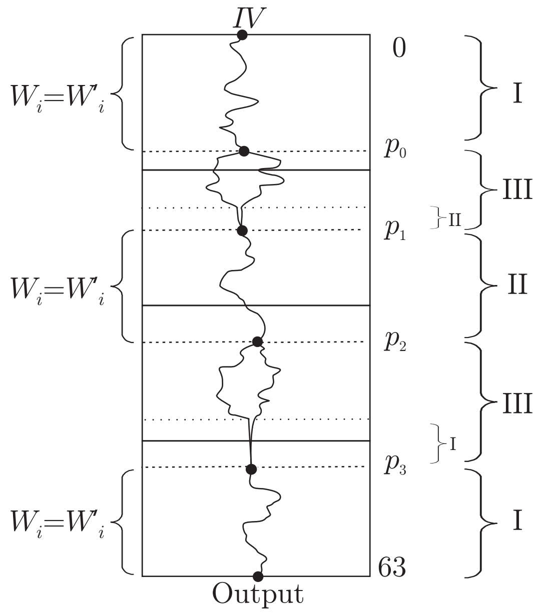

Figure 5.2: Overview of the attack on the compression function of MD5.

This can also be seen directly from the equations, as for steps close to the edge of an inner collision the equations get much simpler, e.g. for $i = p_1 - 1$, (5.10) reduces to

$$f_i(R_{i-1}, R_{i-2}, R_{i-3}) - f_i(R_{i-1}, R_{i-2}, R'_{i-3}) = -\Delta^{+} R_{i-4},$$

and for $i = p_1$ it becomes even simpler:

$$0 = R'_{i-4} - R_{i-4} - 1, \quad \text{i.e. } \Delta^{+} R_{i-4} = -1.$$

Choosing i_0. Let us summarize the reduced the system of equations which originally consisted of (5.2)-(5.8):

We still explicitly require the conditions on the input difference pattern and the message expansion, i.e. (5.4), (5.5) and (5.7). After adding (5.13) to achieve two separate inner collisions, the output difference fulfills the condition

$$\Delta^{+} R_i = 0, \text{ for } i \in \{-4, \ldots, p_0 - 1, p_1 - 3, \ldots, p_2 - 1, p_3 - 3, \ldots, 63\}. \quad (5.14)$$

For the equations coming from the step operations we have to distinguish three different cases (cf. Figure 5.2):

(I) $i \in \{0, \dots, p_0 - 1\} \cup \{p_3 + 1, \dots, 63\}$: *(Beginning and Ending)*
In these two areas the step operations behave identically during processing X and X' respectively, and as shown earlier we can ignore (5.2) and (5.3) for these steps completely.

(II) $i \in \{p_1 + 1, \dots, p_2 - 1\}$: *(Connection)*
For these steps i, (5.2) and (5.3) are identical and thus it suffices to look at only one of these two equations.

(III) $i \in \{p_0, \dots, p_1\} \cup \{p_2, \dots, p_3\}$: *(Inner Collisions)*
In these two areas there is at least one nonzero difference for the two equations and we have to consider one equation of type (5.2) and one of type (5.10) per step, which (in contrast to using (5.3)) has the advantage that it does not include any W_i, W_i'.

Thus the number of equations to consider, depends significantly on the p_i and thus on the choice of i_0. Considering the definition of the permutations σ_k for MD5 (see Appendix A), we get the sizes for these ranges of steps as shown in Table 5.1.

i_0	p_1-p_0+1	p_3-p_2+1	p_2-p_1-1	♯Eqs.
0	20	8	21	77
1	16	20	19	83
2	28	16	17	105
3	24	12	15	87
4	20	14	13	81
5	16	20	11	83
6	12	16	25	81
7	24	12	7	79
8	20	24	5	93
9	16	20	19	91
10	12	16	17	73
11	17	28	6	96
12	20	8	13	69
13	16	20	11	83
14	12	16	9	65
15	8	12	23	63

Table 5.1: Number of equations to solve depending on choice of i_0.

From this table the best choice seems to be $i_0 = 15$, as in this case we have only $2 \cdot (8 + 12) + 23 = 63$ equations to consider. Dobbertin's choice instead was $i_0 = 14$, which gives $p_0 = 14$, $p_1 = 25$, $p_2 = 35$, $p_3 = 50$ and this yields only 2 equations more and might be better for his approach to actually solve the equations (see below).

Inner Almost Collisions. Another important approach used in this technique is to look for "inner almost collisions", at the end of the inner collisions.

Inner almost collision means that we require some specific very small differences $\Delta^+ R_i$ for the last steps of the actual inner collisions. These differences are chosen as starting points of output difference patterns which lead to collisions with high probability for random register values R_i and random W_i for these steps. In Figure 5.2 this idea is indicated by the dotted lines and we denote the steps by $\tilde{p}_1$ and $\tilde{p}_3$.

With this additional constraint we can again reduce the number of equations: we simply ignore (5.10), which describes the differences, for the steps $\tilde{p}_1 + 1, \dots, p_1, \tilde{p}_3 + 1, \dots, p_3$ completely, because the difference pattern is fulfilled with high probability anyway. If the part is at the beginning or at the end, we can additionally ignore (5.2) for these steps, as the actual register values are of no interest in this case. This means, in the described attack we can then consider steps $\tilde{p}_1 + 1, \dots, p_1$ as belonging to the connection part II and $\tilde{p}_3 + 1, \dots, p_3$ even as belonging to case I.

Of course, ignoring these equations means, that not every solution we get yields a collision. However, if the difference pattern holds with probability P for random register and message values, we just need to find about $^1/_P$ solutions for our system of equations to have a good chance of finding a collision.

To actually find such difference patterns, Dobbertin's idea was to do some randomized backtracking on the steps immediately before the inner collisions to decide which differences in the registers are the most likely to lead to the desired inner collision. As we know from our analysis in Section 4.2.3 doing this backwards from the end of the inner collision was a good choice as the differences can be controlled more easily going backwards than going forward.

Using such experiments he decided that $\tilde{p}_1 = 21$ and $\tilde{p}_3 = 42$ were good choices. For reasons given below he also chose $\Delta^+ X_{14} (= \Delta^+ W_{25} = \Delta^+ W_{50}) = [9]$ and from this input difference he computed the most probable output patterns to be

$$(\Delta^+ R_{18}, \dots, \Delta^+ R_{25}) = ([27], 0, 0, [\bar{9}], 0, 0, 0, 0)$$

and

$$\begin{aligned}
&(\Delta^+ R_{39}, \dots, \Delta^+ R_{50}) \\
&\quad = \quad ([31, \overline{19}, \overline{14}], [\overline{30}, \overline{14}], [\overline{30}, \bar{9}], [18, 9], [18], 0, [\bar{9}], [\bar{9}], 0, 0, 0, 0).
\end{aligned}$$

To make this idea work and to be able to solve the resulting systems of equations, some more constraints than described were required, but they depend very much on the specific situations in the techniques used to solve the equations, which we will describe in the next section.

5.2.1.2 Computational Part of the Attack

After deducing and simplifying the system of equations as described in the previous section (summarized on page 82) and after deciding about i_0 and the difference patterns for steps $\tilde{p}_1 \dots p_1$ and steps $\tilde{p}_3 \dots p_3$ what remains is to actually solve the system of equations.

The idea for this *computational part* is, roughly spoken, to split the system of equations into three parts, solve them (nearly) independently by specialized techniques and put the solutions together again. Of course, this is only the rough idea, as there are no really independent parts, but the overlap between the following three parts is very limited and manageable (cf. again Figure 5.2):

(i) *First inner collision* consisting of
steps $p_0, \dots, \tilde{p}_1$ (case III) and steps $\tilde{p}_1 + 1, \dots, p_1$ (case II):
includes only $R_{p_0-3}, \dots, R_{p_1}$, $\Delta^+ R_{p_0}, \dots, \Delta^+ R_{\tilde{p}_1}$ and $W_{p_0}, \dots, W_{p_1}$

(ii) *Second inner collision*
consisting of steps $p_2, \dots, \tilde{p}_3$ (case III):
includes only $R_{p_2-3}, \dots, R_{\tilde{p}_3}$, $\Delta^+ R_{p_2}, \dots, \Delta^+ R_{\tilde{p}_3}$ and $W_{p_2}, \dots, W_{\tilde{p}_3}$

(iii) *Connection part*
consisting of steps $p_1 + 1, \dots, \tilde{p}_2 - 1$ (case II):
includes only $R_{p_1-2}, \dots, R_{\tilde{p}_2-1}$ and $W_{p_1+1}, \dots, W_{p_2-1}$

Besides some few intersections on the "edges" of (iii) ($R_{p_1-2}, \dots, R_{p_1}$ with (i) and $R_{p_2-3}, \dots, R_{p_2-1}$ with (ii)) the only overlappings of these systems are in the W_i due to the message expansion (5.4) and (5.5). With Dobbertin's choices of $i_0 = 14$, $\tilde{p}_1 = 21$ and $\tilde{p}_3 = 42$ the X_i appearing in the three parts are:

(i) $(W_{14}, \dots, W_{21}) = (X_{14}, X_{15}, X_1, X_6, X_{11}, X_0, X_5, X_{10})$
$(W_{22}, \dots, W_{25}) = (X_{15}, X_4, X_9, X_{14})$

(ii) $(W_{35}, \dots, W_{42}) = (X_{14}, X_1, X_4, X_7, X_{10}, X_{13}, X_0, X_3)$

(iii) $(W_{26}, \dots, W_{34}) = (X_3, X_8, X_{13}, X_2, X_7, X_{12}, X_5, X_8, X_{11})$

Thus the overlappings in the variables appearing in the systems of equations are only:

$$\text{(i)} \leftrightarrow \text{(ii)} \quad : \quad X_0, X_1, X_4, X_{10}, X_{14} \tag{5.15}$$

$$\text{(i)} \leftrightarrow \text{(iii)} \quad : \quad R_{22}, R_{23}, R_{24}, R_{25} \text{ and } X_5, X_{11} \tag{5.16}$$

$$\text{(ii)} \leftrightarrow \text{(iii)} \quad : \quad R_{31}, R_{32}, R_{33}, R_{34} \text{ and } X_3, X_7, X_{13} \tag{5.17}$$

The course of the remaining steps in Dobbertin's attack is as follows: try to find solutions for the inner collisions (i.e. for (i) and (ii)) in which the overlapping variables are equal. This can be done rather independently for (i) and (ii) respectively, e.g. by fixing some of the overlapping values initially to some random values and other techniques (see below). Then from such a fixed common solution the last part of the attack is to connect both parts by solving (iii).
Additionally, if one is looking for a collision of the hash function (and not only for the compression function), one has to solve an additional system for the steps $0, \ldots, p_0 - 1$ analogous to (iii), which connects the first inner collision with the original prescribed IV.

An important idea, applied in many steps here, is to use some evolutionary principles in the algorithms: often in the methods applied there are many parameters to choose for the algorithms, e.g. how many bits to change when introducing some random change or which cost functions to choose for some hill climbing technique. Usually these choices have to be based on heuristics. Dobbertin often replaced this simple choosing of the parameters by some evolutionary process on a meta-level, making experiments with different sets of parameters and selecting the best sets of parameters by some trial-and-error process. This way he was able to find really appropriate choices of parameters for a successful solution.

Finding Inner Collisions. In this part we try to solve (i) and (ii), i.e.

$$\begin{aligned} & f_i(R_{i-1}, R_{i-2}, R_{i-3}) - f_i(R'_{i-1}, R'_{i-2}, R'_{i-3}) \\ = \; & ((R_i - R_{i-1})^{\ggg s_i}) - ((R'_i - R'_{i-1})^{\ggg s_i}) - \Delta^{+} R_{i-4} - \Delta^{+} W_i \end{aligned} \tag{5.18}$$

for $i \in \{14, \ldots, 21, 35, \ldots, 42\}$, and

$$R_i \;=\; R_{i-1} + (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i} \tag{5.19}$$

for $i \in \{14, \ldots, 25, 35, \ldots, 50\}$ simultaneously.

The first crucial idea for solving these systems is to realize that each of the W_i is determined by five consecutive R_j ($R_{i-4}, \ldots, R_i$ by (5.19)). Vice versa, as already described in Section 3.3.2, an arbitrary W_i can be reached by adjusting one of the values R_i, R_{i-4} or R_{i-1}.

Thus it is possible to concentrate on the equations given by (5.18) trying to find solutions for the appearing R_i. These values then lead to a solution of the complete subsystem (i) or (ii) respectively (including (5.19) by adjusting the W_i).

The only problem when doing this independently for (i) and (ii) is that some of the W_i computed at the end are connected via the message expansion (cf. (5.15)). In the sequel we will now describe different techniques and ideas used by Dobbertin to avoid these dependencies and solve the remaining systems of equations.

Let us first consider $W_{23} = X_4 = W_{37}$. Here the overlap is very small. In (i) because of the inner almost collision in step $\tilde{p}_1 = 21$ the values for $R_{22}, \dots, R_{25}$ can be chosen nearly arbitrarily (or at least the probability of success is good when choosing them randomly.). Hence, by the remark above, this also holds for $W_{22}, \dots, W_{25}$ and after we have found a solution for (ii) (determining $X_4 = W_{37}$) we have a good chance of being able to adjust one of the register values, say R_{23}, such that (5.18) is fulfilled for $i = 23$.

For $X_{14} = W_{14} = W_{25} = W_{35}$ the situation is similar. As R_{10}, R_{25} and R_{31} each appear only in one equation of type (5.19) these three equations can be solved by adjusting these three R_i to any value given for X_{14}.

Thus the overlap is reduced to X_0, X_1 and X_{10} which will be solved later.

Solving (i). Let us now take a look at the important part of the system of equations to be solved for (i), i.e. (5.18) for $i \in \{14, \dots, 21\}$:

$$\begin{aligned}
0 &= (R_{14}-R_{13})^{\ggg 17} - (R'_{14}-R_{13})^{\ggg 17} - \Delta^{+} W_{14} && (5.20)\\
f_{15}(R_{14}, R_{13}, R_{12}) - f_{15}(R'_{14}, R_{13}, R_{12}) &= (R_{15}-R_{14})^{\ggg 22} - (R'_{15}-R'_{14})^{\ggg 22} && (5.21)\\
f_{16}(R_{15}, R_{14}, R_{13}) - f_{16}(R'_{15}, R'_{14}, R_{13}) &= (R_{16}-R_{15})^{\ggg 5} - (R'_{16}-R'_{15})^{\ggg 5} && (5.22)\\
f_{17}(R_{16}, R_{15}, R_{14}) - f_{17}(R'_{16}, R'_{15}, R'_{14}) &= (R_{17}-R_{16})^{\ggg 9} - (R'_{17}-R'_{16})^{\ggg 9} && (5.23)\\
f_{18}(R_{17}, R_{16}, R_{15}) - f_{18}(R'_{17}, R'_{16}, R'_{15}) &= (R_{18}-R_{17})^{\ggg 14} - (R'_{18}-R'_{17})^{\ggg 14} - \Delta^{+} R_{14} && (5.24)\\
f_{19}(R_{18}, R_{17}, R_{16}) - f_{19}(R'_{18}, R'_{17}, R'_{16}) &= (R_{19}-R_{18})^{\ggg 20} - (R_{19}-R'_{18})^{\ggg 20} - \Delta^{+} R_{15} && (5.25)\\
f_{20}(R_{19}, R_{18}, R_{17}) - f_{20}(R_{19}, R'_{18}, R'_{17}) &= \quad -\Delta^{+} R_{16} && (5.26)\\
f_{21}(R_{20}, R_{19}, R_{18}) - f_{21}(R_{20}, R_{19}, R'_{18}) &= (R_{21}-R_{20})^{\ggg 9} - (R'_{21}-R_{20})^{\ggg 9} - \Delta^{+} R_{17} && (5.27)
\end{aligned}$$

In this system, those R'_i for which we know that $\Delta^{+} R_i = 0$, have already been replaced by R_i simplifying the system a lot. Next, Dobbertin first transforms some of the equations some more, especially also by fixing some values before using probabilistic techniques to find actual solutions.

For example, the first idea is to fix $R_{17} = -1, R'_{17} = 0$. Then, as $f_{20} = \mathrm{ITE}_{zxy}$, (5.26) reduces to

$$R_{19} - R'_{18} = -\Delta^{+} R_{16}. \tag{5.28}$$

Another idea, which Dobbertin often uses for the transformations, is to approximate the terms

$$((R_i - R_{i-1})^{\ggg s_i}) - ((R'_i - R'_{i-1})^{\ggg s_i})$$

on the right hand side of the equations by

$$\Delta^{+} R_i^{\ggg s_i} - \Delta^{+} R_{i-1}^{\ggg s_i}.$$

This approximation can be analyzed by applying the theorems of Section 4.1.3 (cf. Section 5.2.3.1). For example, we can apply Theorem 4.13 to the right hand side of (5.27) to get

$$\begin{array}{rl} & (R_{21} - R_{20})^{\ggg 9} - (R'_{21} - R_{20})^{\ggg 9} - \Delta^{+} R_{17} \\ = & (\underbrace{R'_{21} - R_{20}}_{=:A} - \underbrace{[9]}_{=:B})^{\lll 23} - (R'_{21} - R_{20})^{\lll 23} + 1 \\ = & -([9]^{\lll 23}) + c^{-} \cdot 2^{23} - c^{-}_{R} + 1 = c^{-} \cdot 2^{23} - c^{-}_{R} \end{array}$$

and as we have $B = 2^9$ for most choices of A it holds that $c^{-} = c^{-}_{R} = 0$. As also the left hand side of (5.27) is equal to zero most often, since $\Delta^{+} R_{18}$ has very small weight, (5.27) holds with high probability.

This was probably also the (or at least one important) reason for choosing $\Delta^{+} X_{14} = [9]$ instead of some other $[k]$, which would have resulted in similar differential patterns for the inner almost collisions, but other values for this equation.

Similarly we could deduce some information on $\Delta^{+} R_{14}$ from (5.20). But here Dobbertin chose to introduce additional restrictions by fixing

$$\begin{array}{rclcrclcl} R_{13} & = & [\overline{25}, 16], & \quad & R_{14} & = & [25, \overline{16}] & = & [24, \ldots, 16], \\ & & & & R'_{14} & = & [\overline{25}, 16, \overline{0}] & = & [31, \ldots, 25, 15, \ldots, 0], \end{array}$$

which solves (5.20) and simplifies (5.21) a lot: as $R'_{14} = \overline{R_{14}}$ the left hand side of (5.21) depends only on and, more important, is strongly influenced by R_{12}. Hence, by approximating the right hand side by $\Delta^{+} R_{15}^{\ggg 22} - \Delta^{+} R_{14}^{\ggg 22}$ we can create nearly random values for $\Delta^{+} R_{15}$ which solve (5.21) by choosing random values for R_{12}, as $\Delta^{+} R_{14}$ is already fixed.

Now it remains to solve (5.22) to (5.25) together with (5.28). Here, we do not describe every step in detail, roughly spoken it is done equation by equation: first some of the values appearing in these equations are prescribed (usually randomly) such that there is only one variable left in an equation. Then solutions for this variable are determined, the solution is put into the next equation and so on till all the equations are solved.

The solutions of single equations could obviously be found by an exhaustive search, as there is only one variable with 2^{32} possible values to choose, but Dobbertin used a specialized algorithm for doing this which is much more effective. This algorithm is described in detail in Section 6.1.

Continuous Approximation. Unfortunately it is not possible to follow this strategy till all equations are solved. After doing this for some equations, (5.24) is left to solve but without any variable to choose. However,

many variables have been chosen randomly so far and Dobbertin uses this fact in a method he calls "continuous approximation"(cf. [Dob97, Dob98a]): the idea of this method is to start from those values, which solve all equations but (5.24), as base values. Then make only very little changes to some of the values which have been randomly chosen and compute the other values as before such that all the other equations are fulfilled. Following the analysis from Section 4.2.3 this should also imply only little changes in the other variables due to the small avalanche factor of the step operations.
Then check whether the distance of the two sides of (5.24) became smaller; if so, keep the changed values as new base values and otherwise discard the changes and try other changes. This way it is possible to approach a solution of all equations in a random way. If at any time one runs into a dead-end and does not find any improvements for some time, simply start all over again with new randomly chosen base values.

An important ingredient here is the evolutionary process of choosing the parameters, for example how to introduce the random changes or when to give up and start from the scratch. This way it is possible to find solutions for (5.20) to (5.27) consisting of $R_{12}, \ldots, R_{21}$, which yield $W_{16}, \ldots, W_{21} = (X_1, X_6, X_{11}, X_0, X_5, X_{10})$. However, as (5.27) holds with high probability independent of the actual value for R_{21} it is possible to adjust the solution to some nearly arbitrary value for $X_{10} = W_{21}$. Thus, we have also solved the overlap in the variable X_{10}.

The only two overlapping X_i left are X_0 and X_1. This problem is handled by simply using the values computed for these two variables, as inputs in the next step, solving (ii).

Solving (ii). The technique for finding the second inner collision is similar to that of the first. Here only very few transformations of the equations are used, the main part being a stepwise solution of the equations starting from the last equation, (5.18) for $i = 42$, and going backwards. As above for solving the single equations the algorithm described in Section 6.1 is applied, solutions of one equation are used as inputs for the next, rewinding some steps when reaching some dead-end, i.e. some equations without solutions.

An interesting fact used here, is that once one solution of this system is found, many similar solutions can be found easily by introducing little changes to few variables (due to the small avalanche factor of the step operation, cf. Section 4.2.3). This was necessary as including the fixed value for X_0 from the solution of (i) posed a problem. Thus the system was first solved without paying attention to the prescribed value for $W_{41} = X_0$ and later this was fixed by producing many similar solutions as just described.

In fact, the overlapping variable X_1 was easier to handle, similar as X_{10} in (i): it only appears in step 36, i.e. already in the second step in this part, and by adjusting R_{32} (which has only minimal influence on the solutions of the other considered equations) this overlap can be solved.

From the overall description of finding the two inner collisions (i.e. solving (i) and (ii)) it can be seen, why the introduction of inner almost collisions is a big gain for this method: without these reductions, there would be many more overlappings which could not be removed that easily as it is the case here. And with more overlappings it would be very hard to solve the systems, as can be easily seen from the description above. Further, Dobbertin's choice of $i_0 = 14$ instead of $i_0 = 15$ (which would have produced less equations) can be justified by this, as $i_0 = 14$ resulted in this very well fitting scheme of overlappings.

Solving (iii) (i.e. Connecting the two Inner Collisions). After having fixed a common solution for (i) and (ii) it remains to find a solution for (iii), i.e. for

$$R_i \quad = \quad R_{i-1} + (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i} \qquad (5.29)$$

for $i \in \{26, \ldots, 34\}$ where $R_{22}, \ldots, R_{25}, R_{31}, \ldots, R_{34}$ and also

$$(W_{26}, W_{28}, W_{30}, W_{32}, W_{34}) = (X_3, X_{13}, X_7, X_5, X_{11})$$

are already fixed. This means, besides $R_{26}, \ldots, R_{30}$ we can only choose the three variables

$$X_8 = W_{27} = W_{33}, \quad X_2 = W_{29}, \quad X_{12} = W_{31}$$

arbitrarily. This is one variable less than necessary to have at least a "fair" chance of the system to be solvable. Hence, the question is, where to get the necessary degrees of freedom from.

One degree of freedom can be gained from the fact that $X_9 = W_{24}$ so far only appears in the inner almost collision part (type II) and thus this can also be chosen nearly arbitrarily without violating (i). This can be included indirectly in the solving process for (iii) by adjusting R_{24} accordingly.

That way we have as many variables left as we have equations to solve. However, to have a good probability of finding solvable systems we need at least one degree of freedom more to be able to introduce some randomness in the process easily.

This can be achieved by using the variable X_{15}. So far, this appears only in two steps: in step 15 where, as described above, it has very little influence on the solutions and can also be chosen nearly arbitrarily, and in step 22 in the inner almost collision part (type II). Here it can also be chosen nearly arbitrarily, but changes $R_{22}, \ldots, R_{25}$ significantly, thus having a strong impact on the connection part (iii).

The last problem to be solved is the double appearance of X_8 in (5.29) for $i = 27$ and $i = 33$. This can again be dealt with by the continuous approximation technique described above.

Thus the connection part can be summarized as: start with a fixed solution for (i) and (ii) and adjust it to a randomly chosen X_{15}. Then fix the remaining free variables $R_{26}, \ldots, R_{30}, X_2, X_8, X_{12}$ and also R_{24} (for X_9) in a well-chosen order (also including some random choices) to solve the equations by continuous approximation. If this seems to be impossible, start again by choosing a new random value for X_{15}.

To actually complete the process of finding collisions one has to repeat this algorithm with random parameters, in order to finally find a solution of the systems of equations, which also fulfills the equations that were omitted due to the inner almost collision part.

5.2.2 Results

In this section we summarize the actual results, Dobbertin achieved by applying this technique to the functions MD4, MD5 and RIPEMD-0:

MD5. As described in Section 5.2.1, with this algorithm it is possible to produce collisions for the compression function of MD5 efficiently. To be more precise, this means, after doing some precomputations, with a usual desktop PC it is possible to produce such collisions every few minutes nowadays. An example collision can be found in [Dob96c].

Unfortunately, in the algorithm as described here, there is not enough freedom left to also solve the additional equations for steps $0, \ldots, p_0 - 1$ in order to produce a collision of the *hash* function.

MD4. For MD4 this method is a little bit simpler to apply. MD4 consists of only 48 steps, that is three rounds. Hence the message word X_{i_0}, in which the difference is injected, is only applied in three steps denoted p_0, p_1 and p_2.

Dobbertin's idea in this case was to use an inner almost collision in step p_1 together with a differential pattern for steps $p_1, \ldots, p_2$. Thus for collisions of the compress function there is only one part of type III left (for the steps $p_0, \ldots, p_1$). Hence it is not surprising, that there is enough freedom left to solve an additional part of type II (for steps $0, \ldots, p_0 - 1$) to also include the original prescribed IV and find collisions of the hash function.

The actually used difference pattern for steps $p_1, \ldots, p_2$ has a probability of success of about 2^{-22}. Counting the number of computational steps necessary to compute a new set of values for which this pattern can be tested, this leads to the conclusion that the overall attack has a complexity which is approximately equal to that of computing 2^{20} hash values.[1] The details of the attack are described in [Dob98a].

Additionally, it is described in [Dob98a] how the attack can be easily transferred to Extended MD4, showing that this hash function is not collision resistant either.

Meaningful Collisions. Dobbertin also extended this attack to find "meaningful" collisions, in this case, collisions consisting of ASCII text. Of course, finding meaningful collisions is much more complex, as there are many restrictions on the message value, but it is still possible to do it efficiently. One example is given in Figure 5.3. Here the asterisks represent 20 "random" bytes, which form a header. These two contracts generate the same MD4 hash value. For details see [Dob98a].

```
********************
CONTRACT

At the price of $176,495 Alf Blowfish
sells his house to Ann Bonidea ...
```

```
********************
CONTRACT

At the price of $276,495 Alf Blowfish
sells his house to Ann Bonidea ...
```

Figure 5.3: Two contracts producing the same MD4 hash value.

RIPEMD-0. RIPEMD-0 also consists of three rounds, but there are two parallel lines of computation, i.e. altogether 96 steps. Therefore, in contrast

[1] Note, that it is actually not a *computation of* 2^{20}*hash values*, like it is stated in many articles, but the attack on MD4 has a *complexity similar to computing* 2^{20}*hash values*.

to the MD4 attack, the inner almost collision part is skipped and the attack is applied only to two of the three rounds. However, as described in [Dob97] it is possible to launch the attack on the first two rounds as well as on the last two rounds.

Nevertheless, the attack is less efficient than the MD4 attack, simply because it is necessary to find a common solution of two systems of equations in parallel. Being solvable at all depends significantly on the very "parallel" construction of the two lines of computation, in which in each step the same parameters are used for both lines of computation. Details of this attack are explained in [Dob97].

5.2.3 Extensions and Further Analysis

Many details of the techniques can be improved by using some of the tools introduced in Chapter 4; this has already been mentioned at some places in the description of the attack. In this section we will focus on some important aspects and also describe extensions of Dobbertin's attack.

5.2.3.1 Exchanging $\lll$ and $+$

As described in Section 5.2.1, in the transformations of the equations, often approximations like

$$(R_i - R_{i-1})^{\ggg s_i} \quad \approx \quad R_i^{\ggg s_i} - R_{i-1}^{\ggg s_i} \tag{5.30}$$

or even

$$(R_i - R_{i-1})^{\ggg s_i} - (R'_i - R'_{i-1})^{\ggg s_i} \quad \approx \quad \Delta^{+} R_i^{\ggg s_i} - \Delta^{+} R_{i-1}^{\ggg s_i}$$

are used. Using the theorems from Section 4.1.3 it is possible to analyze these transformations theoretically:

For example, Corollary 4.13 (for $A = R_i$, $B = R_{i-1}$, $k = n - s_i$) states that approximation (5.30) holds with probability

$$\frac{1}{4}(1 + 2^{-s_i} + 2^{-(n-s_i)} + 2^{-n})$$

for random values for R_i and R_{i-1}. Thus, the approximation holds with a probability of a little more than $1/4$, the exact value depending on the amount of rotation. For example, in MD5 these values s_i are never smaller than 4 (and also not greater than 23, actually) and thus this probability is always smaller than $9/32$.

However, if one of the values is fixed the probabilities may change significantly. For example, if we fix $R_{19} = [10]$ the probability that approximation (5.30) can be applied is $1 - 2^{-10}$ for step 19, but only $2^{-27} + 2^{-32}$ for step 20 (cf. Corollary 4.13). Thus we can see that the applicability of this approximation depends very much on the context and can be analyzed using Corollary 4.13.

The second approximation has to be analyzed in two steps, if we want to apply this corollary:

$$\begin{array}{rlcl} & (R_i - R_{i-1})^{\ggg s_i} & - & (R'_i - R'_{i-1})^{\ggg s_i} \\ \approx & R_i^{\ggg s_i} - R_{i-1}^{\ggg s_i} & - & R_i'^{\ggg s_i} + R_{i-1}'^{\ggg s_i} \\ = & R_i^{\ggg s_i} - R_i'^{\ggg s_i} & - & (R_{i-1}^{\ggg s_i} - R_{i-1}'^{\ggg s_i}) \\ \approx & \Delta^{+} R_i^{\ggg s_i} & - & \Delta^{+} R_{i-1}^{\ggg s_i} \end{array}$$

The problem is, that the two approximations together with the corresponding probabilities are dependent. Thus it is not possible to get the complete probabilities directly.

Replacing Approximations by Transformations. Especially in cases where the four cases have similar probabilities (see above), we propose not to restrict oneself to certain assumptions, but to consider all the possible cases at once: this is possible by introducing two one-bit variables c_1, c_2 and replace for example $(R_i - R_{i-1})^{\ggg s_i}$ by

$$R_i^{\ggg s_i} - R_{i-1}^{\ggg s_i} + c_1 \cdot 2^{n-s_i} + c_2.$$

If it is important not to loose any information then one could even add the equations

$$c_1 = \begin{cases} 1, & \text{if } R_i < R_{i-1}, \\ 0, & \text{else}, \end{cases} \qquad c_2 = \begin{cases} 1, & \text{if } [R_i]_{s_i-1,\dots,0} < [R_{i-1}]_{s_i-1,\dots,0}, \\ 0, & \text{else.} \end{cases}$$

Then it is possible to consider all cases at once and this could be used to analyze the above 2-step transformation exactly.

5.2.3.2 Finding Inner Almost Collision Differences Theoretically

In Section 4.3.2 we proposed a method for computing modular differential patterns. This can be used to replace (and improve) the method used in Dobbertin's attack to decide which differences to require for the inner almost collisions. Example 4.41 provides a good impression of what is possible with this method. This example shows that with these simple methods it is easy to include one additional step (and maybe more) into the inner almost collision part in comparison to Dobbertin's original attack.

An extension of this method is proposed in Section 5.4.3.

5.2.3.3 Solving the Systems of Equations

The largest and most important extension and improvement of Dobbertin's techniques described here is an improvement for solving the systems of equations. Based on Dobbertin's algorithm for finding all solutions of one such equation by building a tree of solutions, we propose a new data structure, called a *solution graph*. With a solution graph it is possible not only to *find* all the solutions even of more than one such equation also more efficiently, but to *represent* the complete set of solutions efficiently, as well.

This is interesting as it also provides the possibility to improve on the procedure: For example, with these solution graphs it is possible to compute the whole sets of solutions for some different equations and then combine them to get the set of solutions of the whole system of equations, e.g. combining some steps. Or, for a randomization process it is possible to compute the set of values which are admissible for some equations and then choose the random values only from this set.

The description of these extension is deferred to Chapter 6 as it has some impact on other parts of cryptography, especially on T-functions, as well.

5.2.3.4 Application to Functions Using Recursive Message Expansion

Dobbertin's method, as described in Section 5.2.1, is very dependent on the message expansion by roundwise permutations. This dependance starts with the idea that introducing a difference in only one message word X_{i_0} affects only 4 steps; it is reflected in the simple structure of the attack (cf. Figure 5.2) and it is also important for dealing with the overlappings when actually solving the systems of equations.

In this section we will analyze some aspects of transferring this method to compression functions with *recursive message expansion*, especially to SHA-0 and SHA-1.

One main idea in the MD5 attack can be formulated as trying to achieve $\Delta^+ W_i = 0$ for as many consecutive steps as possible, because from this the number of required equations can be reduced considerably by using the concept of inner collisions (cf. Section 5.2.1).

Unfortunately, with a recursive message expansion this is not that easy. Here, most of the W_i depend on nearly all of the message words X_i and thus the differences are much harder to control. Additionally, we can more or less only control the $\oplus$-differences $\Delta^{\oplus} W_i$ (due to the $\mathbb{F}_2$-linear message expansion) and not the modular differences $\Delta^+ W_i$ which were used in the MD5 attack.

For message expansions like those of SHA-0 or SHA-1 (cf. (3.2) and (3.3)) it is easy to see that prescribing 16 consecutive $\Delta^{\oplus} W_i$ fixes all the $\Delta^{\oplus} W_i$, and even fixing only 16 *non*-consecutive $\Delta^{\oplus} W_i$ fixes “most of” the other ones.

Hence, just from these simple observations, it is clear that we can not achieve such a ”good” situation as in the MD5 attack. But what is the best we can achieve?

We analyzed this question with respect to the following criteria:

Analogously to our description of the MD5 attack, we call p_0 the first step i with $\Delta^{\oplus} W_i \neq 0$ and p_3 the last such step. Furthermore, let p_1 be a step such that p_0,…,p_1 constitutes an inner collision and p_2 denote the step such that $\Delta^{\oplus} W_i = 0$ for $i \in \{p_1+1, \dots, p_2-1\}$. Then our goal is to find a pattern coming from the message expansion with maximal values for p_0, $(s-p_3)$ and $p_2 - p_1$.

Additionally, we require lower bounds on some of the parts in order to avoid considering useless input patterns: To be able to look for two inner collisions independently, we require additionally that the part from p_1 to p_2 is not to small, to be precise at least 5 steps. Further, the number of steps between p_0 and p_1 (and between p_2 and p_3) has to be at least 5 in both cases as well. This constraint is necessary as the earliest possible step to erase a difference which was put in one of the registers in step p_0 (or p_2 resp.) is 5 steps later.

Finding such Patterns. To find such patterns, we considered the 2560×512 matrices over $\mathbb{F}_2$ describing the message expansions of SHA-0 and SHA-1 as linear mappings. In each such matrix a block of row $32i$ up to row $32(i+1)-1$ corresponds to the input word W_i and as the expansion is linear it also corresponds to $\Delta^{\oplus} W_i$ when considering differences instead. This means that the number of possible patterns with $\Delta^{\oplus} W_i = 0$ in $i \in \{0, \dots, p_0-1, p_1+1, \dots, p_2-1, p_3+1, \dots, s\}$ can be computed by looking at the rank of the submatrix consisting of the corresponding rows. If we denote this rank by r then the number of possible patterns is given by 2^{512-r}.

Actually, for SHA-0 it suffices to look at only one of the 32 bits, as the message expansion is done bitwise. Thus this expansion can be expressed by a 80×16 matrix which can be analyzed analogously, but much faster.

This is interesting in so far that from each 1-bit pattern which is admissible for SHA-0, we can construct an admissible SHA-0 pattern consisting of only 0 and -1. By Fact 3.3 this is then also admissible for SHA-1.

Table 5.2 shows the parameters of the best existing patterns for SHA-1 with respect to the total sum of steps in the parts $0, \dots, p_0-1$, $p_1+1, \dots, p_2-1$ and $p_3+1, \dots, s$. From this we can see that the regions for which we would have to solve the system of equations (e.g. to find such inner collisions) are

p_0	p_1	p_2	p_3	$p_0 + (79 - p_3)$ $+(p_2 - p_1 - 1)$	$p_1 - p_0 + 1$	$p_3 - p_2 + 1$	$512 - r$
9	31	37	71	22	23	35	1
10	32	38	72	22	23	35	1
5	37	48	72	22	33	25	1
6	38	49	73	22	33	25	1
7	39	50	74	22	33	25	1
7	47	60	76	22	41	17	1

Table 5.2: Degrees of freedom when requiring $\Delta^{+}W_i = 0$ for $i \in \{0, \ldots, p_0 - 1, p_1 + 1, \ldots, p_2 - 1, p_3 + 1, \ldots, s\}$.

much bigger than in the original MD5 attack.

We also made these tests for reduced versions (assuming smaller s) and it is interesting to note that the parameters of the best found patterns were the same for SHA-1 as for SHA-0, only the number of found patterns was bigger for SHA-1.

These results show, simply from looking at the input difference patterns which can be achieved, that it would be much harder to apply Dobbertin's attack to hash functions with a recursive message expansion.

Additionally, we would have to deal with other problems later on, as well. The patterns we were looking for are patterns for $\Delta^{\oplus}W_i$ and not for $\Delta^{+}W_i$ which are used in the equations later on. Moreover, the nonzero differences in the patterns we found consist mainly of $\Delta^{\oplus}W_i = -1$ which following Corollary 4.9 has the worst probability of fulfilling $\Delta^{\oplus}W_i = \Delta^{+}W_i$. Thus it will be quite infeasible to deduce useful modular differences from these patterns.

5.3 Method by $\mathbb{F}_2$-linear Approximations

In this section we present methods using $\mathbb{F}_2$-linear approximations to replace the non-linear operations especially in the step operation. The first attack extensively using this idea is Chabaud's and Joux's attack on SHA-0 [CJ98]. Later this method was improved by Biham und Chen who introduced the *neutral bits method* in [BC04a].

We present these methods here, describing the original method of Chabaud and Joux in Section 5.3.1, the neutral bits method in Section 5.3.2 before finally giving the results achieved by these methods in Section 5.3.3.

5.3.1 Original Method of Chabaud and Joux

In their attack on SHA-0 Chabaud and Joux (cf. [CJ98]) use an approach with $\oplus$-differences. But as it is nearly impossible to analyze the $\oplus$-difference behaviour directly in the original step operation, they substitute all nonlinear parts (i.e. the modular additions and the nonlinear, bitwise defined functions) by $\oplus$-additions (cf. Section 4.3.1). This results in an $\mathbb{F}_2$-linear function and thus the difference behaviour of this approximative function can be analyzed using linear algebra techniques.

Elementary Collisions. The crucial idea was to consider elementary collisions (or *local collisions* as they were called in [CJ98]) of this approximative function, that is, collisions appearing after as few steps as possible. Therefore they defined *perturbations* and corresponding *corrective patterns*.

Making a perturbation simply means to change one bit in the input difference pattern at some arbitrary step. The corresponding corrective pattern then consists of some bits in the steps following the perturbated step, which lead to a collision in the approximated function after as few steps as possible.

For example, in SHA-0 the step operation is given by

$$R_i = R_{i-1}^{\lll 5} + f_i(R_{i-2}, R_{i-3}^{\lll 30}, R_{i-4}^{\lll 30}) + R_{i-5}^{\lll 30} + W_i + K_i$$

which can be approximated by

$$R_i = R_{i-1}^{\lll 5} \oplus (R_{i-2} \oplus R_{i-3}^{\lll 30} \oplus R_{i-4}^{\lll 30}) \oplus R_{i-5}^{\lll 30} \oplus W_i \oplus K_i.$$

This means, if we decide to change $[W_i]_k$, then also $[R_i]_k$ is changed. Thus, in the next step, R_{i+1} would inherit this difference, but due the rotation, in this case $[R_{i+1}]_{k+5}$ would be changed. In order to get an inner collision in as few steps as possible we have to avoid this further nonzero difference. Therefore, in order to cancel this difference, we may change $[W_{i+1}]_{k+5}$. Thus the $(k+5)$-th input bit of step $i+1$, $[W_{i+1}]_{k+5}$, is the first correction bit of the corrective pattern.

By analogous considerations it can be determined which bits have to be changed in the next 4 steps to achieve zero differences (cf. Table 5.3). Then after 6 steps this leads to a collision in the approximated function, a so-called *elementary collision*. It is called elementary, as it can be shown by arguments about the dimensions of the corresponding subspaces, that all the collisions of the approximated function can be written as sums of these elementary collisions. Thus, it is quite easy with methods from linear algebra to find such input difference patterns which lead to output difference patterns corresponding to a collision of the approximated function. However, there are two drawbacks with this approach.

	R_i	$=$	$R_{i-1}^{\lll 5}$	$\oplus R_{i-2} \oplus$	$R_{i-3}^{\lll 30}$	$\oplus$	$R_{i-4}^{\lll 30}$	$\oplus$	$R_{i-5}^{\lll 30}$	$\oplus$	W_i	$\oplus K_i$
i	$[k]^{\oplus}$	$=$	0	$\oplus\ 0\ \oplus$	0	$\oplus$	0	$\oplus$	0	$\oplus$	$[k]^{\oplus}$	$\oplus\ 0$
$i+1$	0	$=$	$[k+5]^{\oplus}$	$\oplus\ 0\ \oplus$	0	$\oplus$	0	$\oplus$	0	$\oplus$	$[k+5]^{\oplus}$	$\oplus\ 0$
$i+2$	0	$=$	0	$\oplus\ [k]^{\oplus}\ \oplus$	0	$\oplus$	0	$\oplus$	0	$\oplus$	$[k]^{\oplus}$	$\oplus\ 0$
$i+3$	0	$=$	0	$\oplus\ 0\ \oplus$	$[k+30]^{\oplus}$	$\oplus$	0	$\oplus$	0	$\oplus$	$[k+30]^{\oplus}$	$\oplus\ 0$
$i+4$	0	$=$	0	$\oplus\ 0\ \oplus$	0	$\oplus$	$[k+30]^{\oplus}$	$\oplus$	0	$\oplus$	$[k+30]^{\oplus}$	$\oplus\ 0$
$i+5$	0	$=$	0	$\oplus\ 0\ \oplus$	0	$\oplus$	0	$\oplus$	$[k+30]^{\oplus}$	$\oplus$	$[k+30]^{\oplus}$	$\oplus\ 0$

Table 5.3: Elementary collision in linearized step operation of SHA-0.

First, the chosen input difference pattern, consisting of perturbations and corresponding corrective patterns, must be consistent with the message expansion. That is, one must be able to find messages (or rather message differences) which after the expansion result in the wanted input difference pattern. For SHA-0 (where this method was applied) this is quite simple, because the message expansion itself is linear. Thus this requirement results simply in some additional linear conditions. In order to apply this method on hash functions with a non-linear message expansion one would have to use an approximation again.

The other problem is that so far, the input difference pattern determined in this way only leads to output difference patterns corresponding to a collision if the *approximated* function is applied. The idea to find collisions for the original function actually, is to look for messages which have the same difference propagation in the original function as in the linearized function. That means, applying the computed input difference pattern to this message results in the same output difference pattern as in the case of the linearized function. Clearly, this cannot be true for every message, but it is possible to deduce conditions from the difference patterns which describe for which actual register values the difference propagation is the same.

Randomized Searches. Usually we can assume the additional conditions mentioned above to be (nearly) independent conditions, which are fulfilled for random messages with some probability. This probability can be deduced from a detailed look at the approximation. Thus the first, but quite naive way, of searching for collisions, would simply be to produce random messages, apply the chosen input difference pattern and hope that all the additional conditions are met such that the message results in a collision.

Chabaud and Joux used some refined randomized search to find actual collisions: they start by repeatedly choosing random values for X_0 and computing the first step until all the conditions for R_0 are fulfilled. Then they do the same with X_1, the second step and R_1 and so on up to X_{14}, the 15-th step and R_{14}. This can be done step by step, as the values $R_0, \ldots, R_{i-1}$ are not influenced by X_i for $i \leq 15$.

After having found this (first 15 words of a) message conforming to the first 15 steps, they only choose random values for X_{15}. This does not change the output difference pattern for the first 15 steps, but produces register values for the remaining steps which can to some extent be called random. From the analysis of Section 4.2.3, however, we know that especially in the first steps this randomness is quite limited.

In any case the probability for fulfilling the conditions for these *remaining steps* is mainly important for the overall complexity of this attack. Of course, one can construct at most 2^{32} different messages by choosing only X_{15}. Hence, after a certain number of (unsuccessful) tries for X_{15} one has to choose some new values for some of the other X_i, but that does not increase the overall complexity very much.

5.3.2 Neutral Bits Method

In [BC04a] Biham and Chen improved this approach, by looking for what they call *neutral bits*. Their idea is to increase the range of steps for which one tries to assure in advance (before the main part of the randomized search) that the randomly chosen messages conform to the difference pattern. Clearly, if we look at more than 15 steps, it is not possible anymore (as before) to change some message word arbitrarily without having to fear that the output difference pattern has changed in these steps. But this is, where the *neutral bits* come into play:

Definition 5.6 (Neutral Bits). *Suppose we start with a message conforming to the given difference pattern up to some step r. Then,*

- *a **bit** of the message is called **neutral**, if inverting it does not prevent the message from conforming to the difference pattern up to step r;*
- *a **pair of bits** is called **neutral**, if this is true for each of these bits and for inverting both of them simultaneously;*
- *a **set of bits** is called **neutral** if this holds for every subset of bits and it is called **2-neutral** if each pair of bits from this set is neutral.*

The maximum number of neutral bits for a given message and step r is denoted by $k(r)$.

Biham and Chen observed the following: if we have a 2-neutral set of bits, then after inverting any subset of these bits the message still conforms to the difference pattern up to step r with a probability of about $1/8$. This means, starting from one initial message which conforms to the difference pattern up to step r, we can produce about $2^{k(r)-3}$ messages which also conform up to step r.

The number of producable message can even be increased by not only using neutral bits but also *simultaneous-neutral* sets of bits.

Definition 5.7 (Simultaneous-Neutral). *A set of bits is called **simultaneous-neutral**, if the single bits of this set are not neutral, but inverting all the bits of the set simultaneously does not prevent the message from conforming to the differential pattern up to step r.*

Thus, each simultaneous-neutral set of bits can be viewed and used as a single neutral bit of a message, also increasing the number $k(r)$.

To apply this method successfuly, two things are required:

- deciding up to which step r the message has to conform to the given difference pattern;
- finding messages with large 2-neutral sets of bits for a given message efficiently.

For the first question we have to consider the probability $P(r)$ that a randomly chosen message conforms to the given difference pattern in the steps following step r. This probability can be approximated very well from the conditions on the register values and r should be chosen such that the number $2^{k(r)-3}$ of messages that can be generated is about $1/P(r)$. Then there is some non-negligible chance to find a collision by testing all the possible messages.

For actually finding large sets of neutral bits, Biham and Chen give a description how to reduce this problem to finding maximal cliques in a graph. Although this is an NP-hard problem, in the cases which are considered here it seems to work fine. Then to actually find messages which have large 2-neutral sets they suggest to perform some kind of local search. They start with one message and compute the corresponding set of 2-neutral bits. Then they test for some of the messages that can be produced by changing some certain subsets of these bits (according to another observation they made) which of these new messages have a larger 2-neutral set of bits and then take one of these messages as the new base message. By repeatedly doing this process they can maximize (locally) the size of the 2-neutral set of bits.

5.3.3 Results

In [CJ98] Chabaud and Joux describe a difference pattern for SHA-0 which is fulfilled with a probability of 2^{-61}. That means their attack has a complexity of about 2^{61}.

In [BC04a] Biham and Chen present collisions for an extended SHA-0 ($s = 82$) which were found using their neutral bit technique. Additionally, in [BC04b] applications of this method to reduced version of SHA-1 are presented which result in collisions for up to 36 steps and the conclusion that collisions for the *last* 53 steps should also be possible.

Joux et al. (cf. [JCJL04]) applied this technique to find an actual collision for the original (80 step) SHA-0, by combining four such differential patterns, constructed as described above, to produce a collision with two messages consisting of four blocks each. Further results on this can be found in [BCJ+05].

In [RO05] Rijmen and Oswald present some optimizations to the attack by Chabaud and Joux and apply them to reduced versions of SHA-1. Furthermore, they describe how a SHA-1 version reduced to 53 steps (excluding the steps in which ITE is used) could be attacked in theory with a complexity of about 2^{71}.

5.4 A Systematic Treatment of Wang's Method

The attacks by Wang et al. mainly use modular differences, similar as Dobbertin's method, which is not surprising as according to [WL04] her method was inspired by Dobbertin's collision for the compression function of MD5, [Dob96b]. Despite having various co-authors the original method described here is due to Wang.

Up to now in the literature only ad-hoc attacks using this method are presented without focussing on the actual method. In this section we present a systematic treatment of this method, supplemented with some own extensions. Most of the details given are based on the descriptions in [WY05, WLF+05] supplemented by information from [WL04] together with various own, independent ideas and analyzes based on [WLFY04].

5.4.1 The Method

Wang's method, as the two others described before, also splits into the two parts of first finding a differential pattern and then finding messages conforming to it.

5.4.1.1 Finding the Difference Pattern

Similar as in the Chabaud/Joux attack Wang starts by looking for a completely determined difference pattern, in contrast to Dobbertin's attack in which only the input pattern was chosen and the output pattern was only a little restrictive.

In Wang's method one crucial idea is to divide the search for an appropriate difference pattern again into two separate parts: finding a useful input difference pattern to have a "nice" differential behaviour in some part (e.g. in the later rounds), and then finding an appropriate output difference pattern for the remaining (earlier) steps.

This is again inspired by Dobbertin's method in which also first the input pattern is fixed before dealing with the resulting output pattern. Wang combines this with the idea of elementary collisions by allowing not only one input difference $\Delta^{+} X_i$ to be nonzero but several in order to achieve a collision after very few steps in some part, similar to the elementary collisions used in the attack by Chabaud and Joux (see Section 5.3). However, she finds such collisions *directly for the original step operation* and not for some approximation as Chabaud and Joux do.

For example, in the MD4-attack, the input pattern is chosen such that randomly chosen messages conform to the difference pattern *in the last* (i.e. third) *round* with a probability of $1/4$. This can be done by looking at the step operation and choosing the input differences such that they cancel after only a few steps. We describe it here using the method proposed in Section 4.3.2:

Example 5.8. The step operation of the last round of MD4 is described by the following equation (for step i):

$$R_i \;=\; (R_{i-4} + \; (R_{i-1} \oplus R_{i-2} \oplus R_{i-3}) + W_i + K_i)^{\lll s_i}.$$

Thus, if we induce a difference of [16] into $X_{12} = W_{35}$ which is used in step 35, we can see that in this step the value in the parantheses produces also a difference of [16], if we suppose that in the steps before there have been zero differences in the R_i (cf. Table 5.4). Then by the rotation by $s_{35} = 15$ bits, following Corollary 4.38 (or Corollary 4.14 in general) this modular difference is rotated to a difference of [31] with probability $1/2$. Hence, with this probability (depending on the actual values of the registers) we have $\Delta^{+} R_{35} = [31]$. The advantage of inducing this special modular difference is, that it implies $\Delta^{\oplus} R_{35} = [31]$ with probability 1 and as $[31] = [\overline{31}]$ we do not need to distinguish between the two possible signed differences in this case.

i	R_i	$= ($	R_{i-4}	$+$	f_i	$(R_{i-1},$	$R_{i-2},$	$R_{i-3})$	$+$	W_i	$)^{\lll s_i}$
35						0	0	0			
			0	$+$	0				$+$	$[16]$	
	$[31]$	$\leftarrow$	$(s_i = 15, Pr. = 1/2)$								
36						$[31]$					
						$\downarrow$ $(Pr.{=}1)$					
						$[31]^{\pm}$	0	0			
			0	$+$	$[31]$	(Pr.=1, f_i = XOR)			$+$	$[31, 28]$	
	$[31]$	$\leftarrow$	$(s_i = 3, Pr. = 1/2)$								
37						$[31]$					
						$\downarrow$ $(Pr.{=}1)$					
						$[31]^{\pm}$	$[31]^{\pm}$	0			
	0	$=$	0	$+$	0	(Pr.=1, f_i = XOR)			$+$	0	
38						0	$[31]^{\pm}$	$[31]^{\pm}$			
	0	$=$	0	$+$	0	(Pr.=1, f_i = XOR)			$+$	0	
39						0	0	$[31]^{\pm}$			
	0	$=$	$[31]$	$+$	$[31]$	(Pr.=1, f_i = XOR)			$+$	0	
40						0	0	0			
	0	$=$	$[31]$	$+$	0				$+$	$[31]$	

Table 5.4: Difference propagation in the last round of MD4.

Thus for the next step ($i = 36$) it follows that $f_{36}(R_{35}, R_{34}, R_{33}) = R_{35} \oplus R_{34} \oplus R_{33}$ results in a difference of again [31]. By choosing a difference of $[31, 28]$ for $X_2 = W_{36}$ we then get a difference of [28] in the parantheses (as [31]+[31]=0 modulo 2^{32}) which is again rotated to a difference of $\Delta^{+} R_{36} = [31]$ with probability $1/2$. Similar considerations can be made for the following steps to produce zero differences. The complete difference propagation up to the inner collision in step 41 is illustrated in Table 5.4.

By this consideration at least the input differences

$$
\begin{aligned}
&(\Delta^{+} X_{12}, \Delta^{+} X_2, \Delta^{+} X_{10}, \Delta^{+} X_6, \Delta^{+} X_{14}, \Delta^{+} X_1) \\
&\quad = (\Delta^{+} W_{35}, \ldots, \Delta^{+} W_{40}) = ([16], [31, 28], 0, 0, 0, [31])
\end{aligned}
$$

are determined. In order to have as few disturbances as possible from other steps (following Dobbertin's approach of using minimal input differences) the remaining $\Delta^+ X_i$ are chosen to be zero as well.

To determine the complete difference pattern, it remains to find an output pattern for the first two rounds which can be fulfilled given this input pattern. Not much is known about how Wang actually succeeded in finding these patterns. According to [WL04] she does this by hand and quite intuitively.

Basically, these difference patterns can be analyzed (and found to some extent) by our method proposed in Section 4.3.2. Applying this, we could go on similar as in Example 5.8, but the distinction is, that for this part there is no freedom in the choice of the differences for the W_i anymore, which was our main tool in the example above. There are only three items left, in which we sometimes have a little degree of freedom:

1. When looking at the difference propagation through the *bitwise function* f_i, depending on the actual input difference, there are many cases where different output differences are possible and we can choose between them by imposing certain conditions on the corresponding register bits (cf. Section 4.3.2).

2. When looking at the difference propagation through the *rotation*, in general up to four differences can be chosen (see Section 4.1.3). However, often one of the four cases is much more probable than the other ones such that it does not make much sense to consider the other cases at all.

3. Maybe the most important freedom left, is given in the assumptions to make about the actual *bitwise differences* $\Delta^{\pm}$ given some modular difference Δ^+. For example, as we have seen in Section 4.1.2 a modular difference of $[k]$ may (following Corollaries 4.5 and 4.6) result in $\oplus$-differences $[k+l, \ldots, k]$ with arbitrary values $l \in \{0, \ldots, 31-k\}$, depending on the actual register values, where small values for l are more probable than large values. Hence, by imposing conditions on these register values it is possible to influence the bitwise differences and thus the differences coming from the bitwise defined functions in the step operation, for example in order to "expand" or "move" certain modular differences.

In Section 5.4.3 we present an idea for a method using these three possibilities to build a tree of difference patterns from which hopefully the required patterns can be found.

Here, as an example we present and analyze some steps of the difference pattern which was actually used for the first two rounds in Wang's MD4 attack.

Example 5.9. The complete difference pattern spans over the steps 1 to 24 (for details see [WLF$^+$05]). Here, as an example, we concentrate only on analyzing steps 12 to 14, as illustrated in Table 5.5. After step 11 we assume having the following differences in the registers:

$$(\Delta^{+} R_8, \Delta^{+} R_9, \Delta^{+} R_{10}, \Delta^{+} R_{11}) = ([16], [\overline{25}, 21, \overline{19}], [\overline{29}], [31]).$$

There are some interesting constructions in these few steps: in step 12 the [16] in $\Delta^{+} R_8$ is cancelled by the prescribed $[\overline{16}]$ in $\Delta^{+} W_{12}$. Therefore it would be possible to achieve a zero difference in $\Delta^{+} R_{12}$ by imposing the right conditions on the the register bits influencing f_{12} following Table 4.5.
However, in this path other options are used: first the modular difference $\Delta^{+} R_9 = [\overline{25}, 21, \overline{19}]$ is expanded to the bitwise difference

$$\Delta^{\pm} R_9 = [\overline{25}, 22, \overline{21}, \overline{20}, 19].$$

This is necessary for being able to produce $\Delta^{+} f_{12} = [22, 19]$ by imposing the conditions shown on the right hand side of the table. If we would simply assume $\Delta^{\pm} R_9 = [\overline{25}, 21, \overline{19}]$, we would have input bit differences of $(0, 0, 0)$ and $(0, 0, \overline{1})$ in bit 22 and 19 respectively and by Table 4.5 we can see that it is impossible to cause an output difference of 1 (for achieving $\Delta^{+} f_{12} = [22, 19]$) from these input differences.
This difference, which is rotated to $\Delta^{+} R_{12} = [25, 22]$, is desired for two reasons: the difference of [22] is propagated (sometimes being rotated) from step to step til the very last step 24, in which it then cancels one "half" of the prescribed input difference $\Delta^{+} W_{24} = [31, \overline{28}]$. The difference of [25] is used right away in step 13 to reduce the weight of the output difference (cf. Table 5.5): in step 13 the quite big difference of $\Delta^{+} R_9 = [\overline{25}, 21, \overline{19}]$ would be propagated right through to $\Delta^{+} R_{13}$, but with $\Delta^{\pm} R_{12} = [25, 22]$ it is possible to cause $\Delta^{+} f_{13} = [25]$ which cancels at least one of the differing bits, resulting in $\Delta^{+} R_{13} = [28, \overline{26}]$.
Finally (for this example), in step 14 this modular difference is also expanded to the bitwise difference of $\Delta^{\pm} R_{13} = [29, \overline{28}, \overline{26}]$ in order to be able to produce $\Delta^{+} f_{14} = [29]$ which cancels $\Delta^{+} R_{10} = [\overline{29}]$.

5.4.1.2 Message Modifications

Having found such differential patterns, what remains is to find messages conforming to them. To do this, Wang applies what she calls *single-step and multi-step modifications.*[2] This means, she starts with some arbitrary

[2] In [WY05] she calls these techniques single-*message* and multi-*message* modifications while in [WLF$^+$05] she calls them single-*step* and multi-*step* modifications. As we regard it to be more appropriate, we use the latter terminology in this thesis.

i	R_i	$=($	R_{i-4}	$+$	f_i	$(R_{i-1},$	$R_{i-2},$	$R_{i-3})$	$+\ W_i\ )^{\lll s_i}$	Conditions
12						$[31]^{\pm}$	$[\overline{29}]^{\pm}$	$[\overline{25},22,\overline{21},\overline{20},19]^{\pm}$		$[R_{11}]_{19}$=$[R_{11}]_{22}$=$[R_{11}]_{29}$=0 $[R_{11}]_{20}$=$[R_{11}]_{21}$=$[R_{11}]_{25}$=1 $[R_9]_{31}$=$[R_{31}]_{21}$
			$[16]$	$+$	$[22,19]$	(Pr.=2^{-7}, f_i = ITE)			$+\ [\overline{16}]$	
	$[25,22]$	$\leftarrow$	$(s_i=3, Pr.\approx 1)$							
13						$[25,22]^{\pm}$	$[31]^{\pm}$	$[\overline{29}]^{\pm}$		$[R_{11}]_{25}$=1,$[R_{10}]_{25}$=0 $[R_{11}]_{22}$=$[R_{10}]_{22}$ $[R_{12}]_{29}$=1,$[R_{12}]_{31}$=0
			$[\overline{25},21,\overline{19}]$	$+$	$[25]$	(Pr.=2^{-5}, f_i = ITE)			$+\ 0$	
	$[28,\overline{26}]$	$\leftarrow$	$(s_i=7, Pr.\approx 1)$							
14						$[29,\overline{28},\overline{26}]^{\pm}$	$[25,22]^{\pm}$	$[31]^{\pm}$		$[R_{12}]_{29}$=1,$[R_{11}]_{29}$=0 $[R_{12}]_{26}$=$[R_{11}]_{26}$,$[R_{12}]_{28}$=$[R_{11}]_{28}$ $[R_{13}]_{31}$=1,$[R_{13}]_{22}$=$[R_{25}]_{31}$=0
	0	$=$	$[\overline{29}]$	$+$	$[29]$	(Pr.=2^{-7}, f_i = ITE)			$+\ 0$	

Table 5.5: Difference propagation in some steps of the first round of MD4.

message and determines up to which step i the message conforms to the fixed differential pattern. Then, depending on the step i she applies either a single- or a multi-step modification on this message to assure that the failing condition is fulfilled afterwards.

Single-Step Modifications. For the first round (step $0 \leq i \leq 15$) a *single-step modification* simply means to adjust the bits in the register R_i such that the conditions are fulfilled. Then the message word X_i is computed which is necessary to produce this register value from the transformed equation of the step operation. For the example of MD4 again, this means to compute

$$X_i = (R_i^{\ggg s_i}) - R_{i-4} - f_i(R_{i-1}, R_{i-2}, R_{i-3}) - K_i. \tag{5.31}$$

In other words, as long as we ignore the aspect of modifying a given base message, one could sum up (and simplify) the single-step modification part by choosing random values for $R_0, \dots, R_{15}$, adjusting them to fulfill all necessary conditions and compute the corresponding X_i from them by (5.31).

Multi-Step Modifications. For later rounds ($i \geq 16$) the necessary *multi-step modification* is a little bit more sophisticated. The general idea is, as before for the single-step modification, to look for a message bit which can be used to change the incorrect register bit. So, for example, to correct $[R_{16}]_k$, one could just invert $[X_0]_{k-3}$, as can be seen from the description of step 16:

$$R_{16} \ = \ (R_{12} + \ f_{16}(R_{15}, R_{14}, R_{13}) + X_0 + K_{15})^{\lll 3}.$$

But, as we know from Section 4.2.3, simply changing one bit in X_0 would cause a lot of changes in the register values following the first application of X_0. Probably not in the steps following *directly*, but up to step 16 there would be huge differences. Thus, many already fulfilled conditions would probably become false again.

Hence, the idea for a multi-step modification is to invert this bit *indirectly* and thereby cause as few changes as possible. For example, to change the $[X_0]_{k-3}$ as required above, one could simply change $[R_0]_k$, as

$$X_0 = (R_0^{\ggg 3}) - R_{-4} - f_0(R_{-1}, R_{-2}, R_{-3}) - K_0.$$

To avoid further changes in other registers, one also has to adjust the message blocks X_1, X_2, X_3, X_4 as they are used in the following steps which are also influenced by the change in R_0:

$$X_i = (R_i^{\ggg s_i}) - R_{i-4} - f_i(R_{i-1}, R_{i-2}, R_{i-3}) - K_i, \quad i = 1, 2, 3, 4.$$

Of course, this might also cause some conditions to fail now, but the probability that this happens is much smaller: the conditions include only register values and at least in $R_0, \dots, R_{15}$ only one bit was changed by this multi-step modification. Thus the first conditions which might be violated by this change (apart from conditions on $[R_0]_k$ itself) are conditions for steps greater than 15.

Different Kinds of Multi-Step Modifications. Another advantage of these multi-step modifications is that there are many possibilities to perform them. Hence, if one way causes some condition to fail, there are other ways one can try to correct one condition without loosing other conditions in return.

In the articles describing the attacks using Wang's method, [WLF+05, WY05], only two single examples of multi-step modifications are presented. Here we present suggestions for various other possibilities of applying such multi-step modifications. We use the example of MD5 here, but the ideas presented can be transferred to other functions with message expansion by roundwise permutations easily.

Changing some X_i. A first class of multi-step modifications (of which an example was presented above) uses the idea of indirectly changing one *message word* to achieve the desired effect and compensating this change by some additional little changes in the message to minimize the chance of violating other conditions. Starting from the step operation (presented here for some step in the second round, where the multi-step modifications are applied usually)

$$R_i = R_{i-1} + (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + X_{\sigma_1(i-16)} + K_i)^{\lll s_i} \tag{5.32}$$

we can see that to invert $[R_i]_k$ it suffices to invert $\left[X_{\sigma_1(i-16)}\right]_{k-s_i}$, where, of course, the numbers of the affected bits are meant to be read modulo 32. To achieve this, from the step operation for step $j = \sigma_1(i-16)$

$$X_j = (R_j^{\ggg s_j}) - R_{j-4} - f_j(R_{j-1}, R_{j-2}, R_{j-3}) - K_j \tag{5.33}$$

we can derive several possibilities:

1. As presented in the example above, it suffices to invert $[R_j]_{k-s_i+s_j}$.
2. Analogously, as long as $j \geq 4$, it is also possible to change $[R_{j-4}]_{k-s_i}$, instead.

3. As for the first round we have $f_j = \text{ITE}$, it is also possible to invert $[R_{j-2}]_{k-s_i}$ and $[R_{j-3}]_{k-s_i}$ simultaneously, having the effect of inverting $[f_j(R_{j-1}, R_{j-2}, R_{j-3})]_{k-s_i}$.

4. Depending on the actual values of $[R_{j-1}]_{k-s_i}$, $[R_{j-2}]_{k-s_i}$ and $[R_{j-3}]_{k-s_i}$, it might also suffice to invert other (combinations of) bits (cf. Table 4.4). For example, if $[R_{j-2}]_{k-s_i} \neq [R_{j-3}]_{k-s_i}$ then it suffices to invert $[R_{j-1}]_{k-s_i}$ to induce a change in $[X_j]_{k-s_i}$.

When doing this kind of changes, it is always important to remember adjusting the other X_i, which may be influenced by the changed registers, as well.

Direct Change of Other Registers. From (5.32) we can also find other options for inducing a change of $[R_i]_k$. Instead of changing $X_{\sigma_1(i-16)}$ we could also try to change one of the registers $R_{i-1}, \ldots, R_{i-4}$ appearing in (5.32) directly, e.g. invert $[R_{i-4}]_{k-s_i}$ or induce changes in f_i by similar means to those described above in items 3 and 4.

However, as we can only influence the register values of round 1 (i.e. for $i \leq 15$) directly (by adjusting some message words X_i), we may have to do this step repeatedly til we require to change some register value of the first round. For example, to invert $[R_{i-4}]_{k-s_i}$, we may require to change $[R_{i-8}]_{k-s_i-s_{i-4}}$ and therefore maybe consequently $[R_{i-12}]_{k-s_i-s_{i-4}-s_{i-8}}$. If $i - 12 \leq 15$, then we can directly cause this change by inverting some message bits and start this chain of changes.

Using Carry Bits. Sometimes it might not be possible to change certain bits, because there are already other conditions requiring specific values for them. In this case an additional idea in order to be able to still apply one of the methods described above is to use the effects of carry bits. This is more dependent on actual register values, but might be necessary in cases where many conditions have already been set.

For example, suppose we want to change $[X_j]_k$ (cf. (5.33)), but all the k-th bits of $R_j, \ldots, R_{j-4}$ are already fixed. Then this might still be possible by taking advantage of carry effects. For example, if $[R_j]_{k-1+s_j} = 0$ and $[R_{j-4}]_{k-1} = 1$ (and $k \neq 0$), inverting both of these bits simultaneously would imply to add $2^{k-1} + 2^{k-1} = 2^k$ to X_j, thereby causing a change in $[X_j]_k$. This idea can be used in combination with all the multi-step modifications suggested above.

5.4.2 Results

Wang et al. successfuly applied this technique to break various hash functions (cf. [WLFY04]). Two of those use compression functions consisting of only three rounds, namely MD4 and HAVAL-128, for a detailed description of the attack on MD4 see [WLF⁺05].

From looking at the method described above, it seems that functions with about three rounds can be broken by this method in general. Roughly spoken, this can be done solving the third round by choosing the input pattern, then solving the first round by single-step and, finally, the second round by multi-step modifications.

Functions with more than three rounds can only be broken if there are special weaknesses which can be exploited. For example, they also found collisions for RIPEMD-0 (also described in [WLF⁺05]), which consists of two parallel lines of computations of three rounds each, i.e. of six rounds altogether. The weakness here is, that the two lines of three rounds each are nearly identical in the design such that it suffices to find only one differential pattern for three rounds which can be applied simultaneously to both lines.

Nevertheless, it is much more complicated to find a message which complies to all conditions in this case, as there are many more conditions, and thus also more contradicting conditions, when applying the single- and multi-step modifications.

The most interesting collisions presented by Wang et al. are the collisions for MD5 for which a little bit more effort was required, as MD5 consists of four rounds (a detailed description can be found in [WY05]):

Wang's Attack on MD5. The general idea for the MD5 attack is to use multiblock messages, i.e. messages for which the compression function has to be invoked more than once (cf. Section 5.1.2). In the case of the MD5 attack the differential pattern for the first application of the compression function was chosen such that it leads to a difference vector of

$$\Delta^{+}(R_{60}^{(0)}, R_{61}^{(0)}, R_{62}^{(0)}, R_{63}^{(0)}) = ([31], [31, \overline{25}], [31, \overline{25}], [31, \overline{25}]).$$

The differential pattern for the second application of the compression function is very similar in wide parts. It starts with the differences

$$\Delta^{+}(R_{-4}^{(1)}, R_{-3}^{(1)}, R_{-2}^{(1)}, R_{-1}^{(1)}) = \Delta^{+}(R_{60}^{(0)}, R_{61}^{(0)}, R_{62}^{(0)}, R_{63}^{(0)})$$

and leads to the following ones:

$$\Delta^{+}(R_{60}^{(1)}, R_{61}^{(1)}, R_{62}^{(1)}, R_{63}^{(1)}) = ([31], [31, 25], [31, 25], [31, 25]).$$

Thus, in the final computation step (which adds again the initial register values $(R^{(1)}_{-4}, R^{(1)}_{-3}, R^{(1)}_{-2}, R^{(1)}_{-1})$ to the current ones $(R^{(1)}_{60}, R^{(1)}_{61}, R^{(1)}_{62}, R^{(1)}_{63})$) these differences cancel such that there is a collision after these two applications of the compression function.

The special weakness exploited in this attack (cf. also [dBB94] on this) is, that it is possible to induce a difference of [31] in some register by choosing special input differences and then this output difference is propagated from step to step with probability 1 in the third round and with probability $1/2$ per step in a large part of the fourth round. This pattern is analyzed in Table 5.6.

Hence, it is possible to find an input difference pattern which leads to an output difference pattern in round 3 and 4 which is fulfilled with high probability. Thus, it is possible to attack even this four round hash function with the method described above.

5.4.3 Extension of Wang's Method

An important part of Wang's method which has not been well described yet in the literature, is the search for a useful difference pattern. The only thing known about this is that Wang found the used patterns "by hand".

Even more all actual collisions for MD5 published so far, i.e. in [WLFY04, WY05, LWdW05, Kli05], use the same differential pattern, which does not provide very practical input differences. For example, this difference pattern does not allow to produce collisions with ASCII text: it uses only input differences of [31] and [15] which means that, considered *bytewise*, only the most significant bits may differ. But in ASCII mainly the least significant 7 bits are used and changing the most significant bit usually produces unprintable or at least "unusable" characters. Hence, with respect to the practical relevance, one important aspect is to look for other difference patterns which are better suited.

Thus, in this section, we propose ideas for (automatically) searching for similar patterns.

Building a Tree of Difference Patterns. The general idea for this algorithm is to build a tree of possible differential patterns by applying the method from Section 4.3.2 and following the different options left after fixing the input differences (cf. also Section 5.4.1.1, page 105). These choices include the different assumptions on a signed bitwise difference coming from a given modular difference, different results when rotating differences and different propagations of differences through the bitwise functions. Of course,

i	R_i	$= R_{i-1} +$	$($	$R_{i-4} +$	$f_i\,(R_{i-1},\ R_{i-2},\ R_{i-3})$	$+\ W_i\ )^{\lll s_i}$	Cond.
34					$0 \quad 0 \quad 0$		–
				$0\ +$	0	$+[\overline{15}]$	
	$[31] \leftarrow$	0	$+[31]$	$(s_i = 16, Pr. = 1/2)$			
35					$[31]^{\oplus} \quad 0 \quad 0$		–
	$[31] \leftarrow$	$[31]\ +$	$($	$0\ +$	$[31]$ (Pr.=1, f_i = XOR)	$+[31]\ )$	
36					$[31]^{\oplus} \quad [31]^{\oplus} \quad 0$		–
	$[31] \leftarrow$	$[31]\ +$	$($	$0\ +$	0 (Pr.=1, f_i = XOR)	$+\ 0\ )$	
37					$[31]^{\oplus} \quad [31]^{\oplus} \quad [31]^{\oplus}$		–
	$[31] \leftarrow$	$[31]\ +$	$($	$0\ +$	$[31]$ (Pr.=1, f_i = XOR)	$+[31]\ )$	
38					$[31]^{\oplus} \quad [31]^{\oplus} \quad [31]^{\oplus}$		–
⋮ 47	$[31] \leftarrow$	$[31]\ +$	$($	$[31]\ +$	$[31]$ (Pr.=1, f_i = XOR)	$+\ 0\ )$	
48					$[31]^{\oplus} \quad [31]^{\oplus} \quad [31]^{\oplus}$		$[R_{i-1}]_{31}$
⋮ 49	$[31] \leftarrow$	$[31]\ +$	$($	$[31]\ +$	$[31]$ (Pr.$=\frac{1}{2}$,f_i = ONX$_{xzy}$)	$+\ 0\ )$	$=[R_{i-3}]_{31}$
50					$[31]^{\oplus} \quad [31]^{\oplus} \quad [31]^{\oplus}$		$[R_{49}]_{31}$
	$[31] \leftarrow$	$[31]\ +$	$($	$[31]\ +$	0 (Pr.$=\frac{1}{2}$,f_i = ONX$_{xzy}$)	$+[31]\ )$	$\neq[R_{47}]_{31}$
51					$[31]^{\oplus} \quad [31]^{\oplus} \quad [31]^{\oplus}$		$[R_{i-1}]_{31}$
⋮ 59	$[31] \leftarrow$	$[31]\ +$	$($	$[31]\ +$	$[31]$ (Pr.$=\frac{1}{2}$,f_i = ONX$_{xzy}$)	$+\ 0\ )$	$=[R_{i-3}]_{31}$
60					$[31]^{\oplus} \quad [31]^{\oplus} \quad [31]^{\oplus}$		$[R_{59}]_{31}$
	$[31] \leftarrow$	$[31]\ +$	$($	$[31]\ +$	0 (Pr.$=\frac{1}{2}$,f_i = ONX$_{xzy}$)	$+[31]\ )$	$\neq[R_{57}]_{31}$
61					$[31]^{\oplus} \quad [31]^{\oplus} \quad [31]^{\oplus}$		$[R_{60}]_{31}$
				$[31]\ +$	$[31]$ (Pr.$=\frac{1}{2}$,f_i = ONX$_{xzy}$)	$+[\overline{15}]$	$=[R_{58}]_{31}$
	$[31,\overline{25}] \leftarrow$	$[31]$	$+[\overline{25}]$	$(s_i = 10, Pr. \approx 1)$			
62					$[31,\overline{25}]^{\pm} \quad [31]^{\oplus} \quad [31]^{\oplus}$		$[R_{61}]_{31}$ $=[R_{59}]_{31}$
	$[31,\overline{25}] \leftarrow$	$[31,\overline{25}]\ +$	$($	$[31]\ +$	$[31]$ (Pr.$=\frac{1}{4}$,f_i = ONX$_{xzy}$)	$+\ 0\ )$	$[R_{59}]_{25}$=0
63					$[31,\overline{25}]^{\pm} \quad [31,\overline{25}]^{\pm} \quad [31]^{\oplus}$		$[R_{62}]_{31}$ $=[R_{60}]_{31}$
	$[31,\overline{25}] \leftarrow$	$[31,\overline{25}]\ +$	$($	$[31]\ +$	$[31]$ (Pr.$=\frac{1}{4}$,f_i = ONX$_{xzy}$)	$+\ 0\ )$	$[R_{60}]_{25}$=1

Table 5.6: Difference propagation in last two rounds of Wang's MD5-attack.

following all the options at every place would result in a very wide tree soon and thus some way of pruning the tree is necessary.

In detail: again, as already mentioned for the method described in Section 4.3.2, this tree can be built going forward as well as going backward through the step operation. We will describe it here going forward, i.e. we suppose we have some description of the step operation, with which R_i can be computed from the values $R_{i-1}, \ldots, R_{i-r}$ together with the input message word W_i.

At each vertex of the tree we store various information:

1. The step, which the vertex corresponds to, for example given by the index i of the message word W_i used in that step. This is always one step after (or before, if going backwards) the step which the preceeding vertex corresponds to.

2. The r difference values which we assume for the registers before (if going forwards) or after (if going backwards) the current step, i.e. $(\Delta R_{i-1}, \ldots, \Delta R_{i-r})$ or $(\Delta R_i, \ldots, \Delta R_{i-r+1})$ respectively, where Δ means either $\Delta^{\pm}$ or Δ^{+} (see below).

3. An estimation for the probability to reach this specific difference state from the situation given in the root of the tree when randomly choosing the initial register values and also the input message words. In order to reduce the computational effort we assume some independency of the conditions here, which cause these probabilities.

The question what kind of differences to store, (i.e. *signed bitwise*, $\Delta^{\pm}$, or *modular*, Δ^{+}, differences) actually depends on the use of the register in the considered step operation: as long as no bitwise operations are applied on a specific register, it is better to only store the modular difference, as for the bitwise difference, one would have to make assumptions usually decreasing the corresponding probability. But as soon as such an assumption has to be made, the assumed bitwise difference has to be stored for as many steps as it is required bitwisely. After that it can be transformed back to the corresponding modular difference, potentially allowing some merging of vertices differing only in the assumptions on bitwise differences, but referring to the same modular difference.

For example, when constructing a tree for MD5 going forward, based on the step operation

$$R_i = R_{i-1} + (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i},$$

we store

$$(\Delta^{\pm} R_{i-1}, \Delta^{\pm} R_{i-2}, \Delta^{\pm} R_{i-3}, \Delta^{+} R_{i-4}).$$

Then following the method from Section 4.3.2 we compute the possible values for $\Delta^{+}R_i$ (and the corresponding probabilities) by first checking what is possible in the bitwise functions and what happens during the rotation by s_i. Then, in the succeeding vertices we store the difference values corresponding to R_i, R_{i-1}, R_{i-2} and R_{i-3}. But following our choice of differences to store, we would first have to make assumptions on the signed bitwise difference $\Delta^{\pm}R_i$ and on the other hand we could transform $\Delta^{\pm}R_{i-3}$ back to $\Delta^{+}R_{i-3}$.

Cost Functions for Pruning. As we know from Section 5.4.1.1 there are many options to choose from when building such trees. Following all the possibilities would result in very large trees, but most of the vertices would have very low probabilities. Hence, an important part of the construction algorithm is a rule which decides, whether to look for successors of a certain vertex or to stop and change to another vertex for examination.

Our suggestion is to use a cost function, including some or all of the following values:

1. The *probability* stored for that vertex. Including this is obvious and we suggest to actually use a value of $-\log_2(p)$ for a probability of p as this alludes to the number of bitwise conditions to be fulfilled.

2. The *weights* of the stored *differences* should also be included. Reasons for this are, that it is much easier to keep control of what is happening for small differences and that, in the end, we want to come back to zero differences.
   However, the question remains which weight to apply, w_N or w_H? Both of them have their special meanings: w_N describes the number of differences which have to be cancelled later, especially concerning the modular differences considered here. The Hamming weight w_H of bitwise differences is also important, as a large w_H implies low probabilities for the difference propagation through the bitwise function in later steps. Hence, maybe both of them should be included in the cost function.

3. The *distance* of the vertex from the root of the tree (small distances causing large costs) to enforce also looking at deeper parts of the tree instead of staying at the first layers where usually the probabilities are bigger.

However, we suggest to test different cost functions with various weights for these parameters in actual implementations to find a cost function which is appropriate for the considered situation.

This way, it is possible to build a tree of difference patterns by simply starting with only a root and then expanding the leaf with the least cost function value till the tree is “big enough” (see below).

It is possible to extend this method to include the choice of different $\Delta^{+} W_i$ when building the tree, but this would lead to much bigger trees, and therefore we only consider the case of fixed values for $\Delta^{+} W_i$ here.

Finding a Colliding Pattern. For the actual attack it is important to find special patterns, e.g. constituting an inner collision, i.e. starting with zero differences, say in step p_0, and also ending with zero differences, say in step p_1. Here, we present three different ideas for reaching this goal:

1. The first — quite trivial, but also not very promising — approach is to simply build the tree starting from a root with only zero differences (corresponding to step p_0) until another vertex is found which also includes only zero differences. The problem with this approach is, that when using fixed $\Delta^{+} W_i$, both, the step $p_0 + 1$ in which the first nonzero differences appear and also the step p_1 in which the inner collision has to be achieved, are fixed and thus the zero difference vertex has to appear in the tree at a fixed distance $p_1 - p_0$ from the root, decreasing the probability of success significantly.

2. The second idea is to build not only one, but two trees: one tree going forward starting from a root corresponding to step p_0 and one tree going backward starting from a root corresponding to step p_1. Both trees are constructed as described above but with the additional restriction that there is another step $\tilde{p}$ such that vertices storing $(\Delta R_{\tilde{p}}, \ldots, \Delta R_{\tilde{p}-r+1})$, i.e. corresponding to step $\tilde{p} + 1$ or $\tilde{p}$ respectively, are not expanded any further. Thus, after some time, we have two trees with many leaves in which achievable difference tuples $(\Delta R_{\tilde{p}}, \ldots, \Delta R_{\tilde{p}-r+1})$ are stored. The idea then is to find some way of connecting these trees for finding a complete differential pattern, using one of the following two possibilities:

   (a) The first idea is to extend the trees by new vertices until there are two vertices (one in each tree) which have the same differences stored. Then the two paths from the roots to these leaves can be combined to form the searched differential pattern.

   (b) If it is not possible to directly find vertices with the same differences, one should look for vertices with *similar* differences. Here, similar means, that some of them are equal and some have small modular differences differing only by few positions. Then it might be possible (applying appropriate heuristics, cf. Example 5.9) to go some steps back in one of the trees to correct the "bad" difference, moving it by the required amount. This might result in some vertices to be constructed which would usually not have been considered due to the pruning rule, but as we now have some specific goal in mind, it might be helpful to consider these vertices anyway.

3. The last, and maybe most promising idea is to construct again two trees, starting from both sides as before, but this time not up to a common step $\tilde{p}$, but up to two different steps such that there is a gap of about one to three steps in between. The crucial idea is to look for pairs of vertices which agree on the differences which are included in both of them. Then the other differences from both vertices can be used as inputs for another computational step for checking some equations of the form (5.18) as in Dobbertin's method, but this time simply to find out if it is possible at all to connect the two difference paths by choosing specific values for the corresponding R_i.

Example 5.10. Assume we have a tree of difference paths going forward, including a leaf storing

$$(\Delta^{\pm}R_9, \Delta^{\pm}R_8, \Delta^{\pm}R_7, \Delta^{+}R_6) = ([25, \overline{14}], 0, [5], [30]),$$

and a tree going backwards having a leaf with

$$(\Delta^{+}R_{11}, \Delta^{\pm}R_{10}, \Delta^{\pm}R_9, \Delta^{\pm}R_8) = ([17], [\overline{14}], [25, \overline{14}], 0).$$

Then there is a connection of the two corresponding difference paths if and only if there is a common solution (for the variables $R_6, \ldots, R_{11}$) of the two equations

$$\begin{aligned}
&\mathrm{ITE}(R_9, R_8, R_7) - \mathrm{ITE}(R_9 \oplus [25, 14], R_8, R_7 \oplus [5]) \\
&\quad = \;(R_{10} - R_9)^{\ggg 17} - (R_{10} - R_9 + [25])^{\ggg 17} - [30], \\
&\mathrm{ITE}(R_{10}, R_9, R_8) - \mathrm{ITE}(R_{10} \oplus [14], R_9 \oplus [25, 14], R_8) \\
&\quad = \;(R_{11} - R_{10})^{\ggg 22} - (R_{11} - R_{10} - [17, 14])^{\ggg 22} - [5].
\end{aligned}$$

These two equations correspond to (5.18) for $i = 10, 11$, inserting all the differences given above and assuming $\Delta^{+}W_{10} = \Delta^{+}W_{11} = 0$.

Such systems of equations can be solved by applying the methods of solution graphs, which are described in Chapter 6. Here, the generalized solution graphs, described in Section 6.5.2 are of special interest, as they allow to make use of the fact, that we are not interested in specific solutions for the R_i, but only in their existence.

5.5 Practical Relevance

Usually, when discussing the practical impact of a new attack on a hash function, an objection raised frequently is that finding a collision *only* means finding two arbitrary colliding "bitstrings". Most of the methods described here do not directly provide the possibility to find collisions of meaningful messages, as it can be done, for example, with the generic birthday attack (using Yuval's approach, cf. Section 2.1.1) and also with Dobbertin's attack on MD4 (cf. Section 5.2.2).

For the Wang attack, in [LWdW05] (see also [LdW05]) it is shown, how to construct two X.509 certificates sharing the same MD5 hash value. This is an important step towards really practically relevant collisions. The drawback with these collisions is that the colliding certificates differ only in the public key and not in the other information included in the certificate, like e.g. the name of the owner of the certificate. In [Kam04] and [Mik04] the question of constructing colliding executables by applying the Wang attack is addressed which also poses an important practical threat.

5.5.1 Meaningful Collisions: The Poisoned Message Attack

In this section we describe the *poisoned message attack*, a way of producing meaningful collisions in the form of two "advanced language" documents (e.g. postscript documents) which share the same hash value but display completely different contents. To accomplish this, we require only an attack which allows to produce a random collision for an arbitrary prescribed IV, as e.g. Wang's method for MD5 (cf. Section 5.4.2).

This part is based on a joint work with Stefan Lucks and has also been published in [LD05].

Poisoned Message Attack. The main idea of the *poisoned message attack* is to exploit "if-then-else" constructions which are available in many advanced document languages. For example, in the postscript language the command

$$\texttt{(}S_1\texttt{)(}S_2\texttt{)eq\{}T_1\texttt{\}\{}T_2\texttt{\}ifelse}$$

executes the string T_1 if the strings S_1 and S_2 are equal, and T_2 otherwise.[3]

Together with the general weakness of all iterated hash functions, that starting from a collision $h(X) = h(X')$ all extensions XS, $X'S$ by an arbitrary common string S also form collisions, we can form meaningful, colliding postscript documents using the following simple procedure:

Suppose we have a postscript document $M_1M_2M_3$ and we want to replace M_2 by some M_2'. Therefore we start by choosing $X_1 :=$ "M_1P_1`(`" where P_1 is a padding string (e.g. consisting of spaces or some arbitrary comment line) such that the length of X_1 is a multiple of 64, say $64k$. Then computing the hash value of any message of the form X_1S will start by processing the same k blocks coming from X_1, resulting each time in the same chaining value IV_0, which we can compute in advance.

After computing this chaining value we can apply the Wang attack (using this initial value IV_0) which gives us two messages X_2 and X_2' and appending these to X_1 yields a collision $h(X_1X_2) = h(X_1X_2')$.

By defining

$$X_3 := \text{“}\texttt{)(}X_2\texttt{)eq\{}M_2\texttt{\}\{}M_2'\texttt{\}ifelse}M_3\text{”}$$

we can now produce two documents $D = X_1X_2X_3$ and $D' = X_1X_2'X_3$ constituting a collision

$$h(D) = h(X_1X_2X_3) = h(X_1X_2'X_3) = h(D').$$

But as

$$\begin{array}{rrcl} & D & = & M_1P_1\texttt{(}X_2\texttt{)(}X_2\texttt{)eq\{}M_2\texttt{\}\{}M_2'\texttt{\}ifelse}M_3, \\ \text{and} & D' & = & M_1P_1\texttt{(}X_2'\texttt{)(}X_2\texttt{)eq\{}M_2\texttt{\}\{}M_2'\texttt{\}ifelse}M_3, \end{array}$$

D displays the same contents as $M_1M_2M_3$, whereas D' displays the contents of $M_1M_2'M_3$.

This shows, that the poisoned message attack can be used to produce collisions with arbitrary contents. Of course, one could easily detect the forgery when looking directly at the source code. However, usually one trusts what is displayed by programs interpreting such advanced document languages, and thus such a forgery has a good probability of being successful.

The poisoned message attack works for all document languages in which a similar if-then-else construction is available and for any hash function for which collisions can be found for an arbitrary prescribed IV.

[3] Here we use the `typewriter` characters to denote those symbols which are meant to be printed directly, whereas the mathtype symbols S_i and T_i are placeholders for other strings. We also omit concatenation symbols $\|$ for the sake of readability.

Implementation. By using our analysis of Wang's method presented in Section 5.4 we have been able to implement her attack and to apply the poisoned message attack succesfully on MD5. For examplary colliding documents, e.g. two versions of this thesis sharing the same MD5 hash value, one with the original Englisch title and one with the corresponding German title, see [PMA].

Rowe's Rule: the odds are five to six that
the light at the end of the tunnel
is the headlight of an oncoming train.
(Paul Dickson)

Chapter 6

Solution Graphs

In his attacks on the hash functions MD4, MD5 and RIPEMD-0 (as described in Section 5.2), Dobbertin used, as one key ingredient, a special algorithm for finding solutions of the equations. The main idea for this algorithm is to examine the equations bitwise, starting from the least significant bits, to build a *tree of solutions*.

This approach can be successful only because the operations included in these equations are limited to bitwise functions and modular additions. The crucial property of these two operations is, that each of the output bits does not depend on more significant input bits, allowing a "solving from right to left", i.e. from least significant to most significant bits. Due to Klimov and Shamir, functions having this property are nowadays called "T-functions", as introduced and analyzed by them in a series of papers ([KS02, KS03, KS04, Kli04, KS05]).

In this chapter we propose a data structure, called a *solution graph*, which was inspired by Dobbertin's algorithm mentioned above. With this data structure it is possible to find and represent the sets of solutions of (systems of) equations which can be described completely by T-functions. The efficiency of the applied algorithms depends significantly on some property of T-functions, which we call the *narrowness*.

Therefore, we define the subclass of w-*narrow* T-functions. In a w-narrow T-function the dependance of the k-th output bit on the first k input bits is even more restricted: the k-th output bit must be computable from only the k-th input bits and a bitstring of length w computed from the first $k-1$ input bits.

Additionally we present a few algorithms which can be used for analyzing and solving such systems of equations described by T-functions. These algorithms include enumerating all solutions, computing the number of solutions, choosing random solutions and also combining two or more solution graphs, e.g. to compute the intersection of two sets of solutions or to compute the concatenation of two T-functions.

In Section 6.1, we start by describing the original algorithm used by Dobbertin in his attacks. In Section 6.2 we give the necessary details on T-functions and define the notion of narrowness. Sections 6.3 and 6.4 are devoted to the actual definition of solution graphs, and to presenting some theorems and algorithms for them.

In Section 6.5 we propose an extension of solution graphs which is important for practical applications as it allows to apply the algorithms also in contexts which are a little more general than systems of equations describable by "pure" T-functions. Finally, in Section 6.6 we present two examples of applications of these algorithms for which we have developed implementations.

6.1 Dobbertin's Original Algorithm

Let $\mathcal{S}$ be a system of equations in which only bitwise functions and modular additions (or subtractions) are used. Furthermore, let $\mathcal{S}_k$ denote the system of equations in which only the k least significant bits of each equation are considered. As (e.g. by Lemma 4.1) those k bits only depend on the k least significant bits of all the inputs, we will consider the solutions of $\mathcal{S}_k$ to have only k bits per variable as well.

Then, again by Lemma 4.1, the following theorem easily follows:

Theorem 6.1. *Every solution of $\mathcal{S}_k$ is an extension of a solution of $\mathcal{S}_{k-1}$.*

This theorem directly leads to the following algorithm for enumerating all the solutions of $\mathcal{S}$.

Algorithm 6.2.

1. Find all solutions (having only 1 bit per variable) of $\mathcal{S}_1$.

2. For every found solution of some $\mathcal{S}_k, k \in \{1, \dots, n-1\}$, recursively check which extensions of this solution by 1 bit per variable are solutions of $\mathcal{S}_{k+1}$.

3. Output the found solutions of $\mathcal{S}_n (= \mathcal{S})$.

Example 6.3. An actual toy example application of this algorithm – finding the solutions x of the equation $\mathcal{S}$ given by $(x \vee \mathtt{0010}_2) + \mathtt{0110}_2 = \mathtt{0001}_2$ with $n = 4$ – is illustrated in Figure 6.1: We start at the root of the tree and check whether 0 or 1 are possible values for $[x]_0$, i.e. if they are solutions of $\mathcal{S}_1$ which is given by $([x]_0 \vee 0) + 0 = 1$. Obviously, 0 is not a solution of this equation and thus we need not consider any more values for x starting with 0. But 1 is a solution of $\mathcal{S}_1$, thus we have to check whether extensions (i.e. $\mathtt{01}_2$ or $\mathtt{11}_2$) are solutions of $\mathcal{S}_2$: $(x \vee \mathtt{10}_2) + \mathtt{10}_2 = \mathtt{01}_2$. Doing this recursively finally leads to the "tree of solutions", illustrated on the left hand side of Figure 6.1.

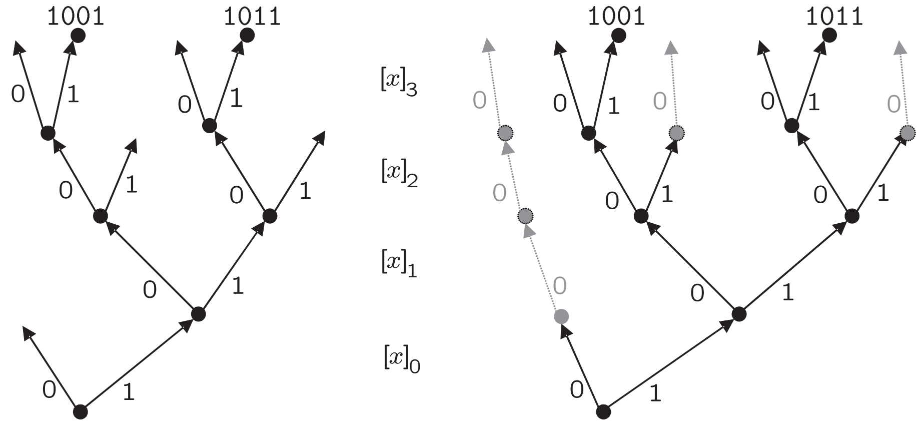

Figure 6.1: "Tree of solutions" for the equation $(x \vee \mathtt{0010}_2) + \mathtt{0110}_2 = \mathtt{0001}_2$ with $n = 4$.

If this method is implemented directly as described in Algorithm 6.2, it has a worst case complexity which is about twice as large as that of an exhaustive search, because the full solution tree of depth n has $2^{n+1}-1$ vertices. An example of such a "worst case solution tree" is given in Figure 6.2.

To actually achieve a worst case complexity similar to that of an exhaustive search a little modification is necessary to the algorithm: the checking should be done for complete paths (as indicated by the *grey* arrows in the tree on the right hand side in Figure 6.1), which can also be done in one machine operation, and not bit by bit. This means, we would start by checking $\mathtt{0000}_2$ and recognize that this fails already in the least significant bit. In the next step we would check $\mathtt{0001}_2$ and see that the three least significant bits are okay. This means in the following step we would only change the fourth bit and test $\mathtt{1001}_2$ which would give us the first solution. All in all we would need only 7 checks for this example as indicated by the grey arrows.

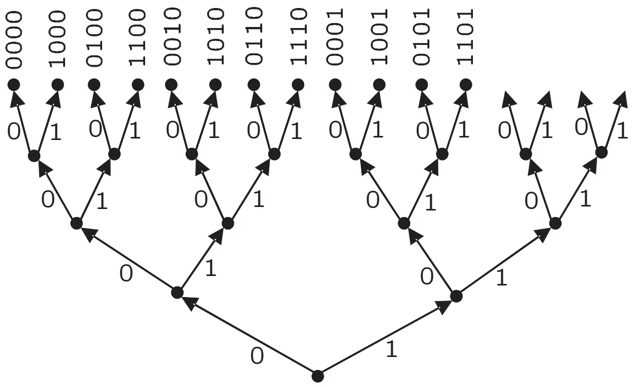

Figure 6.2: "Tree of solutions" for the equation $(\mathtt{0100}_2 \oplus (x + \mathtt{0101}_2)) - (\mathtt{0100}_2 \oplus x) = \mathtt{1101}_2$ with $n = 4$.

The worst case complexity of this modified algorithm (which is what was actually implemented in Dobbertin's attacks) is clearly 2^n as this is the number of leaves of a full solution tree. However, it is also quite clear, that in the average case, or rather in the case of fewer solutions, this algorithm is much more efficient.

Remark 6.4. In Dobbertin's attack (cf. Section 5.2) this algorithm has been applied mainly to solve equations of type (5.18). However, as these equations also include rotations (and not only bitwise functions and modular additions), this was not possible directly.
Therefore, Dobbertin first applied the approximation

$$(R_i - R_{i-1})^{\ggg s_i} - (R'_i - R'_{i-1})^{\ggg s_i} \quad \approx \quad (\Delta^{+} R_i)^{\ggg s_i} - (\Delta^{+} R_{i-1})^{\ggg s_i}$$

(see also Section 5.2.3.1) to the right hand side of the equation to get

$$\begin{aligned} & f_i(R_{i-1}, R_{i-2}, R_{i-3}) - f_i(R'_{i-1}, R'_{i-2}, R'_{i-3}) \\ & \qquad = (\Delta^{+} R_i)^{\ggg s_i} - (\Delta^{+} R_{i-1})^{\ggg s_i} - \Delta^{+} R_{i-4} - \Delta^{+} W_i \end{aligned} \tag{6.1}$$

After fixing all but one of the variables, as it is usually done in Dobbertin's method, this equation is easily solvable by Algorithm 6.2.

6.2 T-Functions

The crucial property of bitwise functions and modular addition which allows us to prove Theorem 6.1 is, that no output bit depends on more significant input bits. As this is exactly what constitutes a T-function, let us first recall the definition from [Kli04]:

Definition 6.5 (T-Function). *A function $f : \{0,1\}^{m\times n} \to \{0,1\}^{l\times n}$ is called a **T-function** if the k-th column of the output $[f(x)]_{k-1}$ depends only on the first k columns of the input $[x]_{k-1}, \ldots, [x]_0$:*

$$\begin{pmatrix} [x]_0 \\ [x]_1 \\ [x]_2 \\ \vdots \\ [x]_{n-1} \end{pmatrix}^T \mapsto \begin{pmatrix} f_0([x]_0) \\ f_1([x]_0, [x]_1) \\ f_2([x]_0, [x]_1, [x]_2) \\ \vdots \\ f_{n-1}([x]_0, [x]_1, \ldots, [x]_{n-1}) \end{pmatrix}^T \tag{6.2}$$

There are many examples of T-functions. All bitwise defined functions, e.g. those from Definition 3.4, are T-functions, because the k-th output bit depends only on the k-th input bits. But also other common functions, like addition or multiplication of integers (modulo 2^n) are T-functions, as can be easily seen from the schoolbook methods. For example, when executing an addition, to compute the k-th bit of the sum, the only necessary information (besides the k-th bits of the addends) is the carrybit coming from computing the $(k-1)$-th bit.

This is also a good example of some other more special property that many T-functions have: one needs much less information than "allowed" by the definition of a T-function: in order to compute the k-th output column $[f(x)]_{k-1}$ only the k-th input column $[x]_{k-1}$ is required and additionally only very little information about the first $k-1$ columns $[x]_{k-2}, \ldots, [x]_0$, for example some value $\alpha_k([x]_{k-2}, \ldots, [x]_0) \in \{0,1\}^w$ of w bits width. This leads to our definition of a *w-narrow T-function*:

Definition 6.6 (w-narrow).
*A T-function f is called **w-narrow** if there are mappings*

$$\alpha_1 : \{0,1\}^m \to \{0,1\}^w, \qquad \alpha_k : \{0,1\}^{m+w} \to \{0,1\}^w, k = 2, \ldots, n-1 \tag{6.3}$$

and auxiliary variables

$$a_1 := \alpha_1([x]_0), \qquad a_k := \alpha_k([x]_{k-1}, a_{k-1}), k = 2, \ldots, n-1 \tag{6.4}$$

such that f can be written as

$$\begin{pmatrix} [x]_0 \\ [x]_1 \\ [x]_2 \\ [x]_3 \\ \vdots \\ [x]_{n-1} \end{pmatrix}^T \mapsto \begin{pmatrix} f_0([x]_0) \\ f_1([x]_1, a_1) \\ f_2([x]_2, a_2) \\ f_3([x]_3, a_3) \\ \vdots \\ f_{n-1}([x]_{n-1}, a_{n-1}) \end{pmatrix}^T \tag{6.5}$$

The smallest w such that some f is w-narrow is called the ***narrowness*** *of f.*

Let us take a look at some examples of w-narrow T-functions.

Example 6.7.

1. The identity function and all bitwise defined functions are 0-narrow.

2. As described above, addition of two integers modulo 2^n is a 1-narrow T-function, as only the carrybit needs to be remembered in each step.

3. A left shift by s bits is an s-narrow T-function.

4. Each T-function $f : \{0,1\}^{m\times n} \to \{0,1\}^{l\times n}$ is $(m(n-1))$-narrow.

Remark 6.8. From Definition 6.6 it directly follows that Theorem 6.1 still holds and Algorithm 6.2 still works if we require more generally that $\mathcal{S}$ is built only of T-functions.

Furthermore, from Definition 6.6 one can directly derive the following lemma about the composition of narrow functions:

Lemma 6.9. *Let $f, g_1, \ldots, g_r$ be T-functions which are w_f-, w_{g_1}-,…,w_{g_r}-narrow respectively. Then the function h defined by*

$$h(x) := f(g_1(x), \ldots, g_r(x))$$

is $(w_f + w_{g_1} + \ldots + w_{g_r})$-narrow.

Note that this lemma (as the notion of w-narrow itself) gives only an upper bound on the narrowness of a function: for example, the addition of 4 integers can be composed of three (1-narrow) 2-integer-additions. Thus by Lemma 6.9 it is 3-narrow. But it is also 2-narrow, because the carry value to remember can never become greater than 3 (which can be represented in $\{0,1\}^2$) when adding 4 bits and a maximum (earlier) carry of 3.

6.3 Solution Graphs for Narrow T-functions

In this section we will describe a data structure which allows to represent the set of solutions of a system of equations of T-functions.

In general, the trees built in Dobbertin's algorithm and thus the computational cost to generate them, may become quite large, in the worst case up to the cost of an exhaustive search. But this can be improved a lot in many cases, or, to be more precise, in the case of T-functions which are w-narrow for some small w, as we will show in the sequel.

Let us first note, that it suffices to consider only the problem of solving one equation

$$f(x) = 0, \tag{6.6}$$

where $f : \{0,1\}^{m\times n} \to \{0,1\}^n$ is some T-function:
If we had an equation described by two T-functions $g(x) = h(x)$ we could simply define $\hat{g}(x) := g(x) \oplus h(x)$ and consider the equation $\hat{g}(x) = 0$ instead. If we had a system of several such equations $\hat{g}_1(x) = 0, \ldots, \hat{g}_r(x) = 0$ (or a function $\hat{g} : \{0,1\}^{m\times n} \to \{0,1\}^{l\times n}$ with component functions $\hat{g}_1, \ldots, \hat{g}_r$) we could simply define $f(x) := \bigvee_{i=1}^{r} \hat{g}_i(x)$ and consider only the equation $f(x) = 0$.
As both operations, $\oplus$ and $\vee$, are 0-narrow, due to Lemma 6.9, the narrowness of f is at most the sum of the narrownesses of the involved functions.

If f in (6.6) is a w-narrow T-function for some "small" w, a solution graph, as given in the following definition, can be efficiently constructed and allows many algorithms which are useful for cryptanalyzing such functions.

Definition 6.10 (Solution Graph). *A directed graph $\mathcal{G}$ is called a **solution graph** for an equation $f(x) = 0$ where $f : \{0,1\}^{m\times n} \to \{0,1\}^n$, if the following properties hold:*

1. *The vertices of $\mathcal{G}$ can be arranged in $n+1$ layers such that each edge goes from a vertex in layer l to some vertex in layer $l+1$ for some $l \in \{0, \ldots, n-1\}$.*
2. *There is only one vertex in layer 0, called the **root**.*
3. *There is only one vertex in layer n, called the **sink**.*
4. *The edges are labelled with values from $\{0,1\}^m$ such that the labels for all edges starting in one vertex are pairwise distinct.*
5. *There is a 1-to-1 correspondence between paths from the root to the sink in $\mathcal{G}$ and solutions of the equation $f(x) = 0$:*
   For each solution x there exists a path from the root to the sink such that the k-th edge on this path is labelled with $[x]_{k-1}$ and vice versa.

*The maximum number of vertices in one layer of a solution graph $\mathcal{G}$ is called the **width** of $\mathcal{G}$.*

In the following we will describe how to efficiently construct a solution graph which represents the complete set of solutions of (6.6). Therefore let f be w-narrow with some auxiliary functions $\alpha_1, \ldots, \alpha_{n-1}$ as in Definition 6.6. To identify the vertices during the construction we label them with two variables (l, a) each, where $l \in \{0, \ldots, n\}$ is the number of the layer and $a \in \{0,1\}^w$ corresponds to a possible output of one of the auxiliary functions α_i. This labelling is only required for the construction and can be deleted afterwards.

Then the solution graph can be constructed by the following algorithm:

Algorithm 6.11 (Construction of a Solution Graph).

```
1. Start with one vertex labelled with $(0, *)$.

2. For each value for $[x]_0$, for which it holds that $f_0([x]_0) = 0$:
     Add an edge
                 $(0, *) \longrightarrow (1, \alpha_1([x]_0))$
     and label this edge with the value of $[x]_0$.

3. For each layer $l$, $l \in \{1, \ldots, n-2\}$, and each vertex $(l, a_l)$ in layer $l$:
     For each value for $[x]_l$ for which $f_l([x]_l, a_l) = 0$:
       Add some edge
                 $(l, a_l) \longrightarrow (l+1, \alpha_{l+1}([x]_l, a_l))$
       and label this edge with the value of $[x]_l$.

4. For each vertex $(n-1, a)$ in layer $n-1$ and each value for $[x]_{n-1}$ for which
   $f_{n-1}([x]_{n-1}, a) = 0$:
     Add an edge
                 $(n-1, a) \longrightarrow (n, *)$
     and label it with the value of $[x]_{n-1}$.
```

Toy examples of the results of this construction can be found in Figure 6.3. Compared with the trees in Figure 6.1 and 6.2, resulting from Dobbertin's algorithm, this shows that these solution graphs are much more efficient.

From the description of Algorithm 6.11 the following properties can be easily deduced:

Theorem 6.12. *Let $f : \{0,1\}^{m \times n} \rightarrow \{0,1\}^n$ be a w-narrow T-function and $\mathcal{G}$ the graph for $f(x) = 0$ constructed by Algorithm 6.11. Then $\mathcal{G}$*

- *is a solution graph for $f(x) = 0$,*
- *has width at most 2^w, i.e. $\mathcal{G}$ has $v \leq (n-1)2^w + 2$ vertices and $e \leq (v-1)2^m$ edges.*

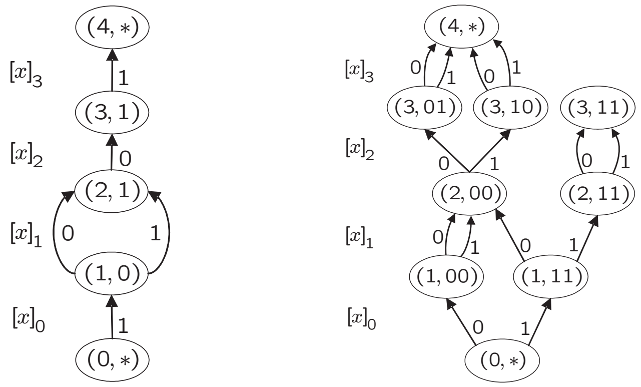

Figure 6.3: Solution graphs for the equations $((x \vee \mathtt{0010}_2) + \mathtt{0110}_2) \oplus \mathtt{0001}_2 = 0$ (on the left) and $((\mathtt{0100}_2 \oplus (x + \mathtt{0101}_2)) - (\mathtt{0100}_2) \oplus\ x) \oplus \mathtt{1101}_2 = 0$ (on the right) with $n = 4$.

Proof. From the description of Algorithm 6.11 it is obvious that Properties 1-3 from Definition 6.10 are fulfilled for $\mathcal{G}$. Furthermore for some fixed vertex a_l in layer l the algorithm adds an edge labelled with $[x]_l$ starting in this vertex only if $f_l([x]_l, a_l) = 0$. As each vertex-label pair is only considered once in the algorithm, it follows that in $\mathcal{G}$ all edges starting in one vertex are pairwise distinct (Property 4).

To see the 1-to-1 correspondence between paths in $\mathcal{G}$ and solutions of the equation (Property 5), first consider a solution x, i.e. $f(x) = 0$. Then with the auxiliary functions $\alpha_1, \ldots, \alpha_{n-1}$ from Definition 6.6 we can compute $a_1, \ldots, a_{n-1}$ from (6.4) such that

$$f_0([x]_0) = 0, \qquad f_i([x]_i, a_i) = 0,\ i = 1, \ldots, n-1.$$

Hence, Algorithm 6.11 produces a path

$$(0, *) \xrightarrow{[x]_0} (1, a_1) \xrightarrow{[x]_1} \ldots\ldots\ldots \xrightarrow{[x]_{n-2}} (n-1, a_{n-1}) \xrightarrow{[x]_{n-1}} (n, *).$$

Vice versa, let us now start with a path

$$(0, *) \xrightarrow{[y]_0} (1, b_1) \xrightarrow{[y]_1} \ldots\ldots\ldots \xrightarrow{[y]_{n-2}} (n-1, b_{n-1}) \xrightarrow{[y]_{n-1}} (n, *)$$

in $\mathcal{G}$. Then, from the existence of an edge $(l, b_l) \xrightarrow{[y]_l} (l+1, b_{l+1})$ and the description of Algorithm 6.11 we can deduce that

$$f_l([y]_l, b_l) = 0, \qquad \alpha_{l+1}([y]_l, b_l) = b_{l+1}.$$

Together with similar properties for the first and the last edges of the path this means, that $f(y) = 0$. The upper bound 2^w on the width of $\mathcal{G}$ and thus the bounds on the number of vertices and edges follow directly from the unique labelling of the vertices by (l, a) with $a \in \{0, 1\}^w$. □

This theorem gives an upper bound on the size of the constructed solution graph, which depends significantly on the narrowness of the examined function f. This shows that, as long as f is w-narrow for some small w, such a solution graph can be constructed efficiently.

6.4 Algorithms for Solution Graphs

The design of solution graphs, as presented here, is very similar to the design of binary decision diagrams (BDDs). For a detailed overview of the subject of BDDs, see for example [Weg00]. Recently, in [Kra04] Krause independently introduced *special ordered binary decision diagrams* (SOBDDs), which are probably the BDD variant, which comes closest to our proposal of solution graphs.

Due to these similarities, it is not surprising, that many ideas of algorithms for BDDs can be adapted to construct efficient algorithms for solution graphs. The complexity of these algorithms naturally depends mainly on the size of the involved solution graphs. Thus, we will first describe how to reduce this size.

6.4.1 Reducing the Size

We describe this using the example of the solution graph on the right hand side of Figure 6.3: there are no edges starting in $(3, 11)$ and thus there is no path from the root to the sink which crosses this vertex. This means, due to Definition 6.10, that this vertex is of no use for representing any solution, and therefore it can be deleted. After this deletion the same applies for $(2, 11)$ and thus this vertex can also be deleted.

For further reduction of the size let us define what we mean by *equivalent* vertices:

Definition 6.13 (Equivalent Vertices). *Two vertices a and b in a solution graph are called **equivalent**, if for each edge $a \to c$ (for some arbitrary vertex c) labelled with x there is an edge $b \to c$ labelled with x and vice versa.*

For the reduction of the size, it is important to notice the following lemma:

Lemma 6.14. *If a and b are equivalent, then there are the same paths (according to the labelling of their edges) from a to the sink as from b to the sink.*

For example, let us now consider the vertices $(3, 01)$ and $(3, 10)$. From each of these two vertices there are two edges, labelled with 0 and 1 respectively, which point to $(4, *)$ and thus these two vertices are equivalent. According to Lemma 6.14 this means that a path from the root to one of those two vertices can be extended to a path to the sink by the same subpaths, independently of whether it goes through $(3, 01)$ or $(3, 10)$. Due to the defining property of a solution graph, this means, that we can merge these two equivalent vertices into one, reducing the size once more. The resulting solution graph is presented in Figure 6.4. In this figure the labels of the vertices are omitted as they are only required for the construction algorithm.

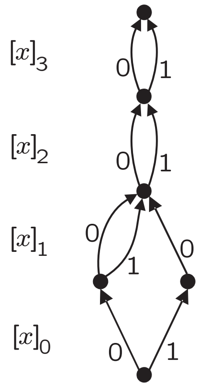

Figure 6.4: Solution graph for the equation $((\mathtt{0100}_2 \oplus (x + \mathtt{0101}_2)) - (\mathtt{0100}_2 \oplus x)) \oplus \mathtt{1101}_2 = 0$ (cf. Figure 6.3) after reducing its size.

Of course, merging two equivalent vertices, and also the deletion of vertices as described above, may again cause two vertices to become equivalent, which have not been equivalent before. But this concerns only vertices in the layer below the layer in which two vertices were merged. Thus for the reduction algorithm it is important to work from top (layer $n-1$) to bottom (layer 1):

Algorithm 6.15 (Reduction of the Size).

1. Delete each vertex (together with corresponding edges) for which there is no path from the root to this vertex or no path from this vertex to the sink.

2. For each layer l starting from $n-1$ down to 1 merge all pairs of vertices in layer l which are equivalent.

To avoid having to check all possible pairs of vertices in one layer for equivalence separately in order to find the equivalent vertices (which would result in a quadratic complexity), in Algorithm 6.15 one should first sort the vertices of the active layer according to their set of outgoing edges. Then equivalent vertices can be found in linear time.

Similar to what can be proven for ordered BDDs, for solution graphs reduced by Algorithm 6.15 it can be shown that they have minimal size:

Theorem 6.16. *Let $\mathcal{G}$ be a solution graph for some function f and let $\tilde{\mathcal{G}}$ be the output of Algorithm 6.15 applied to $\mathcal{G}$. Then there is no solution graph for f which has less vertices than $\tilde{\mathcal{G}}$.*

Proof. For $(x_{l-1},\ldots,x_0) \in \{0,1\}^l$, let

$$\mathcal{E}_{x_{l-1}\ldots x_0} := \left\{(x_{n-1},\ldots,x_l) \in \{0,1\}^{n-l} \mid f(x_{n-1}\ldots x_l x_{l-1}\ldots x_0) = 0\right\}$$

be the set of all extensions of $x_{l-1}\ldots x_0$ which lead to a solution of $f(x)=0$. If $\mathcal{E}_{x_{l-1}\ldots x_0}$ is not empty, then in any solution graph $\mathcal{G}'$ for $f(x)=0$, there is a path starting in the root which is labelled with $x_0,\ldots,x_{l-1}$ and ends in some vertex $a_{x_{l-1}\ldots x_0}$ in layer l. Let $\mathcal{G}'_{x_{l-1}\ldots x_0}$ denote the subgraph of $\mathcal{G}'$ consisting of the vertex $a_{x_{l-1}\ldots x_0}$ (as root) and all paths from $a_{x_{l-1}\ldots x_0}$ to the sink in $\mathcal{G}'$. Then, as $\mathcal{G}'$ represents the set of solutions of $f(x)=0$, $\mathcal{G}'_{x_{l-1}\ldots x_0}$ represents the set $\mathcal{E}_{x_{l-1}\ldots x_0}$.
Hence, if $\mathcal{E}_{x_{l-1}\ldots x_0} \neq \mathcal{E}_{x'_{l-1}\ldots x'_0}$, then also $a_{x_{l-1}\ldots x_0}$ and $a_{x'_{l-1}\ldots x'_0}$ must be different (as otherwise $\mathcal{G}'_{x_{l-1}\ldots x_0} = \mathcal{G}'_{x'_{l-1}\ldots x'_0}$). Thus, the number of vertices $v_l(\mathcal{G}')$ in layer l of the arbitrary solution graph $\mathcal{G}'$ for $f(x)=0$ must be greater or equal to the number of different sets $\mathcal{E}_{x_{l-1}\ldots x_0}$, i.e.

$$v_l(\mathcal{G}') \quad \geq \quad \sharp\left\{\mathcal{E}_{x_{l-1}\ldots x_0} \mid (x_{l-1},\ldots,x_0) \in \{0,1\}^n\right\}.$$

In the following we will show that for $\tilde{G}$ these values are equal, i.e.

$$v_l(\tilde{\mathcal{G}}) = \sharp\left\{\mathcal{E}_{x_{l-1}\ldots x_0} \mid (x_{l-1},\ldots,x_0) \in \{0,1\}^n\right\}$$

and thus there is no solution graph for $f(x)=0$ with less vertices than $\tilde{G}$:
In each solution graph there is only one vertex in layer n, the sink, and thus the equation holds for layer n of $\tilde{\mathcal{G}}$. Now suppose that it holds for layers $n,\ldots,l+1$ and assume that it does not hold for layer l, i.e. there are more vertices in layer l of $\tilde{G}$ than sets $\mathcal{E}_{x_{l-1}\ldots x_0}$.
Then there must be two distinct vertices $a_{x_{l-1}\ldots x_0}$ and $a_{x'_{l-1}\ldots x'_0}$ in layer l such

that $\mathcal{E}_{x_{l-1}\dots x_0} = \mathcal{E}_{x'_{l-1}\dots x'_0}$. Consider an arbitrary edge starting in $a_{x_{l-1}\dots x_0}$ labelled with x_l. This edge leads to a vertex $a_{x_l\dots x_0}$ corresponding to $\mathcal{E}_{x_l\dots x_0}$ and by definition it holds that this set is equal to $\mathcal{E}_{x_l x'_{l-1}\dots x'_0}$. As the claim is fulfilled for layer $l+1$ this means that also $a_{x_l\dots x_0} = a_{x_l x'_{l-1}\dots x'_0}$ and thus an edge from $a_{x'_{l-1}\dots x'_0}$ to $a_{x_l x'_{l-1}\dots x'_0}$ labelled with x_l must exist. Hence, $a_{x_{l-1}\dots x_0}$ and $a_{x'_{l-1}\dots x'_0}$ are equivalent and would have been merged by Step 2 of Algorithm 6.15. □

With the help of the following theorem it is possible to compute the narrowness of f, i.e. the smallest value w such that f is w-narrow. Theorem 6.12 gives a bound on the width of a solution graph based on a bound for the narrowness of the considered function; the following theorem provides the reverse direction:

Theorem 6.17. *Let $f : \{0,1\}^{m\times n} \to \{0,1\}^n$ be a T-function and define $\tilde{f} : \{0,1\}^{(m+1)\times n} \to \{0,1\}^n$ by $\tilde{f}(x,y) := f(x) \oplus y$.*
If $\mathcal{G}$ is a minimal solution graph of width W for the equation $\tilde{f}(x,y) = 0$, then f is a $\lceil \log_2 W \rceil$-narrow T-function.

Proof. As $\mathcal{G}$ has width W it is possible to label the vertices of each layer l of $\mathcal{G}$ with unique values $a_l \in \{0,1\}^{\lceil \log_2 W \rceil}$. Then we can define the following auxiliary functions corresponding to $\mathcal{G}$:

$$\alpha_i(x, a_{i-1}) := \begin{cases} a_i, \text{ if an edge } a_{i-1} \xrightarrow{(x,.)} a_i \text{ exists in } \mathcal{G} \\ 0, \text{ else} \end{cases} \tag{6.7}$$

$$g_{i-1}(x, a_{i-1}) := \begin{cases} y, \text{ if an edge } a_{i-1} \xrightarrow{(x,y)} . \text{ exists in } \mathcal{G} \\ 0, \text{ else} \end{cases} \tag{6.8}$$

Two things remain to be shown:

1. (6.7) and (6.8) are well-defined, i.e. if two edges $a_{i-1} \xrightarrow{(x,y)} a_i$ and $a_{i-1} \xrightarrow{(x,y')} a'_i$ exist, then $a_i = a'_i$ and $y = y'$.
2. The $\lceil \log_2 W \rceil$-narrow T-function $g : \{0,1\}^{m\times n} \to \{0,1\}^n$ defined by

   $$[g(x)]_i := g_i([x]_i, b_i) \text{ where } b_1 := \alpha_1([x]_0, \mathrm{root}_{\mathcal{G}}), b_i := \alpha_i([x]_{i-1}, b_{i-1})$$

   if equal to f.

ad 1): As $\mathcal{G}$ is minimal, there exist paths in $\mathcal{G}$

- from the root to a_{i-1} (labelled $(x_0, y_0), \dots, (x_{i-2}, y_{i-2})$),

- from a_i to the sink (labelled $(x_i, y_i), \ldots, (x_{n-1}, y_{n-1})$),
- from a'_i to the sink (labelled $(x'_i, y'_i), \ldots, (x'_{n-1}, y'_{n-1})$).

Then, by the definition of $\widetilde{f}$ and $\mathcal{G}$ and the existence of the two edges it follows that

$$\begin{array}{rcl} f(x_{n-1} \ldots x_i x x_{i-2} \ldots x_0) & = & y_{n-1} \ldots y_i y y_{i-2} \ldots y_0 \\ & & \Rightarrow [f(x_{n-1} \ldots x_i x x_{i-2} \ldots x_0)]_{i-1} = y, \\ f(x'_{n-1} \ldots x'_i x x_{i-2} \ldots x_0) & = & y'_{n-1} \ldots y'_i y' y_{i-2} i \ldots y_0 \\ & & \Rightarrow \left[f(x'_{n-1} \ldots x'_i x x_{i-2} \ldots x_0)\right]_{i-1} = y'. \end{array}$$

As f is a T-function this means that $y = y'$ and as different edges starting in the same vertex have different labels this also means $a_i = a'_i$.
ad 2): Let $x \in B^{m+n}$ and $y = f(x)$, i.e. $\widetilde{f}(x, y) = 0$. Then we can find the following path in $\mathcal{G}$:

$$\mathrm{root}_{\mathcal{G}} \xrightarrow{([x]_0,[y]_0)} a_1 \xrightarrow{([x]_1,[y]_1)} \ldots\ldots\ldots \xrightarrow{([x]_{n-2},[y]_{n-2})} a_{n-1} \xrightarrow{([x]_{n-1},[y]_{n-1})} \mathrm{sink}_{\mathcal{G}}.$$

From the definition of the b_i and the definition of the α_i it follows that

$$a_1 = b_1 \quad \Longrightarrow \quad a_2 = b_2 \quad \Longrightarrow \quad \ldots \quad \Longrightarrow \quad a_{n-1} = b_{n-1}$$

and thus

$$[g(x)]_i = g_i([x]_i, b_i) = [y]_i = [f(x)]_i \quad \Longrightarrow \quad f = g.$$

□

In the following we always suppose that we have solution graphs of minimal size (from Algorithm 6.15 and Lemma 6.16) as inputs.

6.4.2 Computing Solutions

Similar to what can be done by Dobbertin's algorithm (see Algorithm 6.2), a solution graph can also be used to enumerate all the solutions:

Algorithm 6.18 (Enumerate Solutions).
Compute all possible paths from the root to the sink by a depth-first search and output the corresponding labelling of the edges.

Of course, the complexity of this algorithm is directly related to the number of solutions. If there are many solutions, it is similar to the complexity of an exhaustive search (as for Algorithm 6.2), simply because all of them need to be written. But if there are only a few, it is very fast, usually much faster than Algorithm 6.2.

However, often we are only interested in the number of solutions of an equation which can be computed much more efficiently, namely, with a complexity linear in the size of the solution graph. The following algorithm achieves this by labeling every vertex with the number of possible paths from that vertex to the sink. Then the number computed for the root gives the number of solutions:

Algorithm 6.19 (Number of Solutions).

1. Label the sink with 1.

2. For each layer l from $n-1$ down to 0:
   Label each vertex A in l with the sum of the labels of all vertices B (in layer $l+1$) for which an edge $A \to B$ exists.

3. Output the label of the root.

An application of this algorithm is illustrated in Figure 6.5.

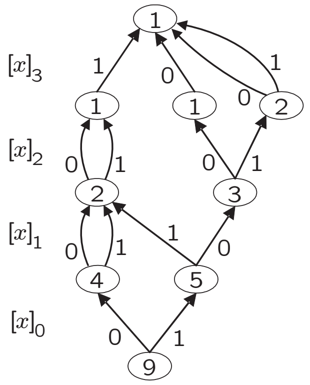

Figure 6.5: A solution graph after application of Algorithm 6.19.

After having labelled all vertices by Algorithm 6.19 it is even possible to choose solutions from the represented set uniformly at random:

Algorithm 6.20 (Random Solution).
Prerequisite: The vertices have to be labelled as in Algorithm 6.19.

1. Start at the root.
2. Repeat
   - From the active vertex A (labelled with n_A) randomly choose one outgoing edge such that the probability for choosing $A \to B$ is ${}^{n_B}/_{n_A}$ where n_B is the label of B.
   - Remember the label of $A \to B$
   - Make B the active vertex.

   until the sink is activated.
3. Output the solutions corresponding to the remembered labels of the edges on the chosen path.

6.4.3 Combining Solution Graphs

So far, we only considered the situation in which the whole system of equations is reduced to one equation $f(x) = 0$, as described at the beginning of Section 6.3, and then a solution graph is constructed from this equation. Sometimes it is more convenient to consider several (systems of) equations separately and then combine their sets of solutions in some way. Therefore, let us now consider two equations

$$\begin{aligned} g(x_1, \ldots, x_r, y_1, \ldots, y_s) &= 0, & (6.9) \\ h(x_1, \ldots, x_r, z_1, \ldots, z_t) &= 0, & (6.10) \end{aligned}$$

which include some common variables $x_1, \ldots, x_r$ as well as some distinct variables $y_1, \ldots, y_s$ and $z_1, \ldots, z_t$ respectively. Let $\mathcal{G}_g$ and $\mathcal{G}_h$ be the solution graphs for (6.9) and (6.10) respectively.
Then the set of solutions of the form $(x_1, \ldots, x_r, y_1, \ldots, y_s, z_1, \ldots, z_t)$ which fulfill both equations simultaneously can be computed by the following algorithm.

Algorithm 6.21 (Intersection).
Let the vertices in $\mathcal{G}_g$ be labelled with $(l, a_g)_g$ where l is the layer and a_g is some identifier which is unique per layer, and those of $\mathcal{G}_h$ analogously with some $(l, a_h)_h$. Then construct a graph whose vertices will be labelled with (l, a_g, a_h) by the following rules:

1. Start with the root $(0, *_g, *_h)$.
2. For each layer $l \in \{0, \ldots, n-1\}$ and each vertex (l, a_g, a_h) in layer l:

- Consider each pair of edges

$$((l, a_g)_g \to (l+1, b_g)_g, (l, a_h)_h \to (l+1, b_h)_h)$$

labelled with

$$\begin{aligned} (X_g, Y_g) &= ([x_1]_l, \ldots, [x_r]_l, [y_1]_l, \ldots, [y_s]_l) \\ \text{and } (X_h, Z_h) &= ([x_1]_l, \ldots, [x_r]_l, [z_1]_l, \ldots, [z_t]_l) \text{ respectively.} \end{aligned}$$

- If $X_g = X_h$, add an edge

$$(l, a_g, a_h) \to (l+1, b_g, b_h)$$

and label it with (X_g, Y_g, Z_h).

The idea of this algorithm is to traverse the two input graphs $\mathcal{G}_g$ and $\mathcal{G}_h$ in parallel and to simulate computing both functions in parallel in the output graph by storing all necessary information in the labels of the output graph. For an illustration of this algorithm, see Figure 6.6. Also notice that this algorithm can be easily generalized for more than two input graphs.

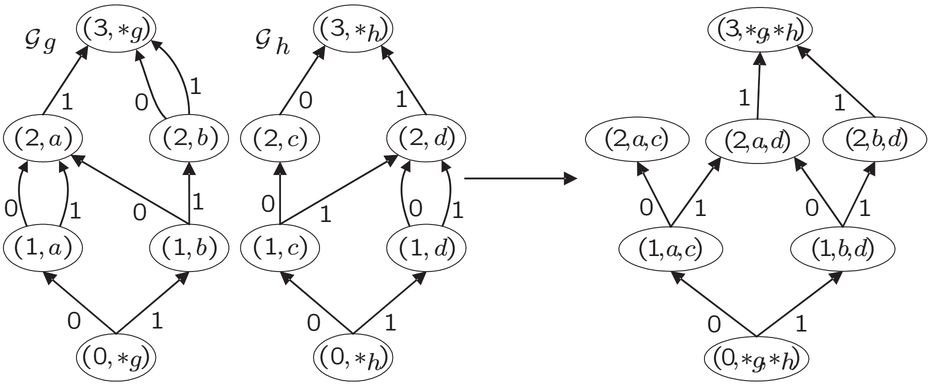

Figure 6.6: Intersection of two solution graphs by Algorithm 6.21.

Apart from just computing mere intersections of sets of solutions, Algorithm 6.21 can also be used to solve equations given by the concatenation of two T-functions:

$$f(g(x)) = y \tag{6.11}$$

To solve this problem, just introduce some auxiliary variable z and apply Algorithm 6.21 to the two solution graphs which can be constructed for the equations $f(z) = y$ and $g(x) = z$ respectively.

Combining this idea (applied to the situation $f = g$) with some square-and-multiply technique, allows for some efficient construction of a solution graph for an equation of the form $f^i(x) = y$ with some (small) fixed value i.

6.5 Extensions of this Method

As described in Remark 6.4, to actually apply his algorithm to the equations appearing in his attack, Dobbertin had to fix some values and to transform the equations by some approximations first, before the algorithm could be applied. This problem still holds for the solution graphs proposed here, as e.g. the rotations appearing in the original equations are not T-functions.

However, in this section we present some ideas to integrate also other (non-T-)functions, as, for example, right bit shifts or bit rotations, which are similar, but not T-functions according to Definition 6.5. The most important idea is the introduction of *generalized solution graphs*, which can (comparing them again with BDDs) be viewed as a non-deterministic variant of solution graphs.

6.5.1 Including Right Shifts

Let us first consider a system of equations which includes only T-functions and some right shift expressions $x^{\gg r}$. This can be transformed by substituting every appearance of $x^{\gg r}$ by an auxiliary variable z_r and adding an extra equation

$$z_r^{\ll r} = x \wedge (\underbrace{11\dots1}_{n-r}\underbrace{0\dots0}_{r}) \tag{6.12}$$

which defines the relationship between x and z_r. Then the resulting system is completely described by T-functions and can be solved with a solution graph.

Here, similarly as when solving (6.11) some problem occurs: we have to add an extra (auxiliary) variable z, which potentially increases the size of the needed solution graph. This is even worse as the solution graph stores all possible values of z corresponding to solutions for the other variables, even if we are not interested in them at all. This can be dealt with by softening Definition 6.10 to *generalized solutions graphs.*

6.5.2 Generalized Solution Graphs

Definition 6.22 (Generalized Solution Graph). *A directed graph $\mathcal{G}$ is called a **generalized solution graph** for an equation $f(x) = 0$ where $f : \{0,1\}^{m\times n} \to \{0,1\}^n$, if all properties from Definition 6.10 hold with the exception that the labels of edges starting in one vertex are* not *required to be pairwise distinct.*

Then we can use similar algorithms as those described above, e.g. for reducing the size or combining two graphs. But usually these algorithms are a little bit more sophisticated: for example, for minimizing the size, it does not suffice to consider equivalent vertices as defined in Definition 6.13. In a generalized solution graph it is also possible that the sets of *incoming* edges are equal and, clearly, two such vertices with equal sets of incoming edges (which we will also call equivalent in the case of general solution graphs) can also be merged. But this also means that merging two equivalent vertices in layer l may not only cause vertices in layer $l-1$ to become equivalent, but also vertices in layer $l+1$. Thus, in the generalized version of Algorithm 6.15 we have to go back and forth in the layers to ensure that in the end there are no equivalent vertices left.

Remark 6.23. This definition of a generalized solution graph allows to "remove" variables without losing the information about their existence. This means, instead of representing the set $\{(x,y) \mid f(x,y)=0\}$ with a solution graph $\mathcal{G}$, we can represent the set $\{x \mid \exists y : f(x,y)=0\}$ with a solution graph $\mathcal{G}'$ which is constructed from $\mathcal{G}$ by simply deleting the parts of the labels which correspond to y. Of course, this does not decrease the size of the generalized solution graph directly, but (hopefully) it allows further reductions of the size.

6.5.3 Including Bit Rotations

Let us now take a look at another commonly used function which is not a T-function, a bit rotation by r bits:

$$f(x) := x^{\lll r}. \tag{6.13}$$

If we would fix the r most significant bits of x, for example to some value c, then this function can be described by a bit shift of r positions and a bitwise defined function

$$f(x) := (x^{\ll r}) \vee c \tag{6.14}$$

which is an r-narrow T-function. Thus, by iterating over all 2^r possible values for c an equation involving (6.13) can also be solved by solution graphs.

If we use generalized solution graphs, it is actually possible to combine all 2^r such solution graphs to one graph, in which again the complete set of solutions is represented: this can be done by simply merging all the roots and all the sinks of the 2^r solution graphs as they are clearly equivalent in the generalized sense.

6.6 Examples of Applications

In this section we present two examples of systems of equations which were solved by using an actual implementation of the techniques presented in this chapter. They both originate from testing the applicability of Dobbertin's method (see Section 5.2) to SHA-1.

The first system is an example of a system of equations to solve when looking for inner collisions, comparable to the system labelled (i) in Section 5.2.1.2. It includes 14 equations and essentially 22 variables R_1, …, R_{13}, $\Delta^{+} R_3$, …, $\Delta^{+} R_{11}$:

$$
\begin{array}{rcll}
0 & = & \Delta^{+} R_3 & +1 \\
0 & = & \Delta^{+} R_4 - (R_3'^{\lll 5} - R_3^{\lll 5}) & +1 \\
\mathrm{ITE}(R_3', R_2^{\lll 30}, R_1^{\lll 30}) - \mathrm{ITE}(R_3, R_2^{\lll 30}, R_1^{\lll 30}) & = & \Delta^{+} R_5 - (R_4'^{\lll 5} - R_4^{\lll 5}) & +1 \\
\mathrm{ITE}(R_4', R_3'^{\lll 30}, R_2^{\lll 30}) - \mathrm{ITE}(R_4, R_3^{\lll 30}, R_2^{\lll 30}) & = & \Delta^{+} R_6 - (R_5'^{\lll 5} - R_5^{\lll 5}) & \\
\mathrm{ITE}(R_5', R_4'^{\lll 30}, R_3'^{\lll 30}) - \mathrm{ITE}(R_5, R_4^{\lll 30}, R_3^{\lll 30}) & = & \Delta^{+} R_7 - (R_6'^{\lll 5} - R_6^{\lll 5}) & +1 \\
\mathrm{ITE}(R_6', R_5'^{\lll 30}, R_4'^{\lll 30}) - \mathrm{ITE}(R_6, R_5^{\lll 30}, R_4^{\lll 30}) & = & \Delta^{+} R_8 - (R_7'^{\lll 5} - R_7^{\lll 5}) - (R_3'^{\lll 30} - R_3^{\lll 30}) & +1 \\
\mathrm{ITE}(R_7', R_6'^{\lll 30}, R_5'^{\lll 30}) - \mathrm{ITE}(R_7, R_6^{\lll 30}, R_5^{\lll 30}) & = & \Delta^{+} R_9 - (R_8'^{\lll 5} - R_8^{\lll 5}) - (R_4'^{\lll 30} - R_4^{\lll 30}) & +1 \\
\mathrm{ITE}(R_8', R_7'^{\lll 30}, R_6'^{\lll 30}) - \mathrm{ITE}(R_8, R_7^{\lll 30}, R_6^{\lll 30}) & = & \Delta^{+} R_{10} - (R_9'^{\lll 5} - R_9^{\lll 5}) - (R_5'^{\lll 30} - R_5^{\lll 30}) & \\
\mathrm{ITE}(R_9', R_8'^{\lll 30}, R_7'^{\lll 30}) - \mathrm{ITE}(R_9, R_8^{\lll 30}, R_7^{\lll 30}) & = & \Delta^{+} R_{11} - (R_{10}'^{\lll 5} - R_{10}^{\lll 5}) - (R_6'^{\lll 30} - R_6^{\lll 30}) & \\
\mathrm{ITE}(R_{10}', R_9'^{\lll 30}, R_8'^{\lll 30}) - \mathrm{ITE}(R_{10}, R_{19}^{\lll 30}, R_8^{\lll 30}) & = & - (R_{11}'^{\lll 5} - R_{11}^{\lll 5}) - (R_7'^{\lll 30} - R_7^{\lll 30}) & +1 \\
\mathrm{ITE}(R_{11}', R_{10}'^{\lll 30}, R_9'^{\lll 30}) - \mathrm{ITE}(R_{11}, R_{10}^{\lll 30}, R_9^{\lll 30}) & = & - (R_8'^{\lll 30} - R_8^{\lll 30}) & \\
\mathrm{ITE}(R_{12}, R_{11}'^{\lll 30}, R_{10}'^{\lll 30}) - \mathrm{ITE}(R_{12}, R_{11}^{\lll 30}, R_{10}^{\lll 30}) & = & - (R_9'^{\lll 30} - R_9^{\lll 30}) & \\
\mathrm{ITE}(R_{13}, R_{12}^{\lll 30}, R_{11}'^{\lll 30}) - \mathrm{ITE}(R_{13}, R_{12}^{\lll 30}, R_{11}^{\lll 30}) & = & - (R_{10}'^{\lll 30} - R_{10}^{\lll 30}) & +1 \\
0 & = & - (R_{11}'^{\lll 30} - R_{11}^{\lll 30}) & +1
\end{array}
$$

It was not possible to solve this system in full generality, but for the application it sufficed to find some fixed values for $\Delta^{+} R_3, \ldots, \Delta^{+} R_{11}$ such that there are many solutions for the R_i and then to construct a generalized solution graph for the solutions for $R_1, \ldots, R_{13}$.

The choice for good values for some of the $\Delta^{+} R_i$ could be done by either theoretical means (see e.g. Section 4.3.2) or by constructing solution graphs for single equations of the system and counting solutions with fixed values for some $\Delta^{+} R_i$.

For example, from the solution graph for the last equation it is possible (as described in Section 6.5.2) to remove the R_{11} such that we get a solution graph which represents all values for $\Delta^{+} R_{11}$ for which an R_{11} exists such that

$$0 = -(R_{11}'^{\lll 30} - R_{11}^{\lll 30}) + 1.$$

This solution graph shows that only $\Delta^{+} R_{11} \in \{1, 4, 5\}$ is possible. Then by inserting each of these values in the original solution graph (by Algorithm 6.21) and counting the possible solutions for R_{11} (by Algorithm 6.19) it can be seen that $\Delta^+ R_{11} = 4$ is the best choice. However, in this situation this could also be seen very easily by applying the tools from Section 4.1.3, especially Corollary 4.14.

Having fixed $\Delta^+ R_{11} = 4$ also the last but one equation includes only one of the $\Delta^+ R_i$, namely $\Delta^+ R_{10}$ (implicitly in R'_{10}). Then possible solutions for $\Delta^+ R_{10}$ can be derived similarly as before for $\Delta^+ R_{11}$ and doing this repeatedly gave us some good choices for $\Delta^+ R_{11}, \Delta^+ R_{10}, \Delta^+ R_9, \Delta^+ R_8, \Delta^+ R_7$ and (using the first two equations) for $\Delta^+ R_3$ and $\Delta^+ R_4$.
Finding values $\Delta^+ R_5$ and $\Delta^+ R_6$ such that the whole system still remains solvable was quite hard and could be done by repeatedly applying some of the techniques described in this section, e.g. by combining generalized solution graphs for different of the equations and removing those variables R_i from the graphs which were no longer of any explicit use. This way we found four possible values for $\Delta^+ R_5$ and $\Delta^+ R_6$.

After fixing all the $\Delta^+ R_i$ variables in a second step we were then able to construct the generalized solution graph for the complete system of equations with the remaining variables $R_1, \dots, R_{13}$. It contains about 700 vertices, more than 80000 edges and represents about 2^{205} solutions.

The second example system of equations appeared when trying to find a "connection" for the two inner collisions parts, i.e. solve the system which was called (iii) in the description in Section 5.2.1.2. After some reduction steps it can be written as follows:

$$\begin{array}{rcl}
C_1 & = & R_9 + \mathrm{ITE}(R_{12}^{\lll 2}, R_{11}, R_{10}) \\
C_2 & = & (C_3 - R_{10} - R_{11}) \oplus (C_4 + R_9^{\lll 2}) \\
C_5 & = & (C_6 - R_{11}) \oplus (C_7 + R_{10}^{\lll 2} - (R_9^{\lll 7})) \\
C_8 & = & (C_9 - R_{12}) \oplus (C_{10} + R_9^{\lll 2}) \\
 & & \qquad \oplus (C_{11} + R_{11}^{\lll 2} - (R_{10}^{\lll 7}) - \mathrm{ITE}(R_9^{\lll 2}, C_{12}, C_{13}))
\end{array}$$

In these equations the C_i are constants which come from some transformations of the original (quite large) system of equations together with some random choices of values. For this system we are interested in finding at least one solution for $R_9, R_{10}, R_{11}, R_{12}$.

As the first three equations are quite simple and (after eliminating the rotations) also quite narrow, the idea for solving this system was the following: first compute a generalized solution graph for the first three equations which represents all possible solutions for R_9, R_{10}, R_{11} for which at least one corresponding value for R_{12} exists. For this set of solutions we observed about 2^{11} to 2^{15} solutions. Then we could enumerate all these solutions from this graph and for each such solution we just had to compute the value for R_{12}

corresponding to the last equation

$$\begin{aligned} R_{12} &= C_9 - (C_8 \oplus (C_{10} + R_9^{\lll 2}) \\ &\qquad \oplus (C_{11} + R_{11}^{\lll 2} - (R_{10}^{\lll 7}) - \mathrm{ITE}(R_9^{\lll 2}, C_{12}, C_{13}))) \end{aligned}$$

and check whether it also fulfilled the first equation. If we consider the first equation with random but fixed values for R_9, R_{10}, R_{11} we see that either there is no solution or there are many solutions for R_{12}, as only every second bit of R_{12} (on average) has an effect on the result of $\mathrm{ITE}(R_{12}^{\lll 2}, R_{11}, R_{10})$. However, since the values for R_9, R_{10}, R_{11} were chosen from the solution graph of the first three equations there is at least one solution and thus the probabiliy that the value for R_{12} from the last equation also fulfills the first, is quite good.

This way we succeded in solving this system of equations efficiently.

Always try to stop talking
before people stop listening.
(anonymous)

Chapter 7

Conclusion

In this final chapter we summarize the current status of the hash functions of the MD4-family and present some perspectives. It is very important to distinguish between the question what functions can be relied on for practical applications at the moment and in the near future, and the question where to put efforts in research from a long term view.

We start with a look at the short or medium term view with a summary of the status of the hash functions of the MD4-family.

7.1 Status of Current Hash Functions

When considering the security of hash functions with respect to practical applications, it is important to distinguish between two aspects, collision resistance and preimage resistance.

Collision Resistance. Today for nearly all functions of the MD4-family, as presented in detail in Chapter 5, attacks on the collision resistance are known, which require an effort lying below that of the generic birthday attack, the so-called *birthday bound*. The only exceptions to this are the hash functions of the new RIPEMD-generation (i.e. RIPEMD-{128,160,256,320}) and those of the new SHA-generation (i.e. SHA-{224,256,384,512}).

However, we must have in mind that this birthday bound is not the deciding criterion for the practical relevance of an attack. Rather, the question is, if it is possible to actually find practically relevant collisions at all. Concerning this, note that for hash functions with a small output length of 128 bits, like MD5 or RIPEMD-128, even the generic birthday attack (which is able to produce *meaningful* collisions due to Yuval's idea, cf. Section 2.1.1) will become feasible soon such that they cannot be considered secure anymore, even if, as in the case of RIPEMD-128, no attacks below the birthday bound are known.

On the other hand, e.g. for SHA-1 there is an attack in about 2^{69} steps (cf. [WYY05]), i.e. below the birthday bound of 2^{80} steps, but today this can still be considered infeasible. Similarly, imagine as an example, that in the future there might be attacks on the larger sized SHA-functions, e.g. perhaps an attack on SHA-512 in about 2^{200} steps. Despite violating the original design goal of admitting no attacks below the birthday bound, of course, such an attack would not mean that the functions would become insecure for practical use, as an effort of 2^{200} steps is far from being feasible even in the distant future.

More interesting is the question of the relevance of found attacks for practical use. At least, from what we described in Section 5.5, we can say that the way from finding *any* collision at all to finding *meaningful* collisions seems to be not as long as one may expect.

Preimage Resistance. Against preimage resistance not many attacks on hash functions of the MD4-family are known. To be precise, Dobbertin's attack from [Dob98b], inverting two out of three rounds of MD4 is the only such attack worth mentioning.

Obviously, a reason for this can be found in the fact, that preimage resistance implies collision resistance, i.e. if one would have found an attack on the preimage resistance, one would also be able to produce collisions. Thus one reason for the absence of preimage attacks may be that preimage resistance is simply that much stronger than collision resistance that it is not possible at all to break it at the moment for the functions considered here.

Alternatively, the reason might be that after a first attack on a hash function has been found (which is usually an attack on the collision resistance for the reasons given above), the appeal to disprove another property is much smaller than before. Thus, maybe just not many people have looked at breaking preimage resistance of functions of the MD4-family.

Recommendations. The immanent problem of today's hash functions is that there are no proofs of security. This means, that the confidence in their security is usually only based on the absence of succesfull attacks and the effort put in finding such attacks.

Hence, for achieving some practical security it is important to rely on the cryptological community to analyze proposed functions thoroughly such that in some kind of evolutionary process weak functions are identified and sorted out or repaired, while the strong proposals survive.

Therefore at the moment, for areas in which a very high level of security is required only the functions SHA-{224,256,384,512} can be recommended. They form the current state of the evolutionary process being based on a very well studied hash function, SHA-1, and the results of the cryptanalysis of this function.

For the short term view, we can also recommend RIPEMD-160, which has a similar background of being designed using well-analyzed functions as a basis, improving them by the results of known attacks. The drawback here is the output length of 160 bits which at least in a long term view will likely be too small to resist even the generic birthday attack. The larger sized variants, RIPEMD-256 and RIPEMD-320, despite having the potential of providing a higher security level, have not been in the focus of cryptanalysts so far, and thus cannot be recommended without further study.

If one is not looking for the utmost degree of security or is only interested in security for a short period of time, e.g. for reasons of efficiency, it might also be a reasonable choice to apply SHA-1, as at the moment the known attacks are still very inefficient and practically infeasible.

7.2 Perspectives

In order to be able to provide secure hash functions in the future, a big effort has to be made in cryptological research. From the current situation the main question evolving is whether to stick to the currently used basic design of the MD4-family or to try out new ideas.

Having in mind the bunch of new very efficient attacks developped in recent times one may argue that a simple mixture of modular addition with bitwise functions is too weak to serve as a basic principle in such functions and that new design ideas are necessary starting from the scratch.

However, why should we simply set aside nearly 15 years of cryptological research on the MD4-family together with the experience gained in this area of research? This thesis shows that nowadays we understand the relation of modular addition and bitwise operations as used in the considered hash functions quite well. Hence, we should use this knowledge and experience to improve the existing functions and design stronger functions using these principles. In general, this is possible, as even the already broken functions could be repaired using the same design principles by simply adding more steps, and thus increasing the complexity of the applied attacks. This shows that the concept is not broken, it is simply a question of efficiency in this case.

But, of course, we have to be open-minded for new approaches which can be incorporated in the existing design ideas, e.g. for reaching a higher level of security without loosing efficiency in return.

One important idea which fits well to the already existing design ideas might be the concept of T-functions. According to Klimov and Shamir (cf. e.g. [KS05]) these functions provide a "semi-wild" approach, combining a wild mixture of basic operations with the possibility to still be able to prove theoretical statements on the constructions, similar to what we would like to have in the designs of hash functions.

Hence, to conclude this thesis, we can say that for the future mainly two directions of research are necessary: first, using the knowledge we have gathered over the years in an evolutionary process to find functions we can really trust in, and second, examining independent, new ideas to include them and thereby improve these functions.

He who breaks a thing to find out what it is,
has left the path of wisdom.
(Gandalf in J. R. R. Tolkien's The Lord of the Rings)

Appendix A

Specifications

In this appendix we present the exact specifications of the compression functions used in the hash functions of the MD4-family.

For each function, first we describe the main **parameters**, i.e. the output length n of the hash function, the output length m of the compression function, the number r of registers, the number s of steps applied in one application of the compression function, the word size w, the number t of words which one message block is split into and finally the length l of the input message block.

Then we describe the **message expansion**. We either give the different permutations σ_k, if using a message expansion by roundwise permutations (cf. Section 3.2.1), or the formula for the recursion for the W_i, if using a recursive message expansion (cf. Section 3.2.2).

In order to describe the **step operations** we give the formula(s) defining the step operation, i.e. how to compute R_i from $R_{i-1}, \ldots, R_{i-r}, W_i$ together with all the tables describing the included step-dependent parts.
As for some functions there are also little changes in other registers than the one computed by the given formula we also give the tuple $(R_i, \ldots)$ describing the actual contents of the r registers after step i, which we call the **state**.
In the case of two parallel lines of computations we use R_i^L for one (the "left") line and R_i^R for the other ("right") line of computations.

To complete the description of the hash function, we need to give the **initial values** $R_{-r}, \ldots, R_{-1}$ and describe how the **output** of the compression functions and (if different) the output of the hash function are computed.

MD4 ([Riv91, Riv92a])

Parameters:

$n = 128$	$m = 128$	$l = 512$
$r = 4$	$s = 48$	$w = 32$

Message Expansion:

$\sigma_k(i)$		i															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
	2	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15

Initial Value:

$$R_{-4} = \texttt{0x67452301}, \quad R_{-3} = \texttt{0x10325476}$$
$$R_{-2} = \texttt{0x98badcfe}, \quad R_{-1} = \texttt{0xefcdab89}$$

Step Operation:

$$R_i = (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i}$$

i	f_i	K_i
$0, .., 15$	ITE	`0x00000000`
$16, .., 31$	MAJ	`0x5a827999`
$32, .., 47$	XOR	`0x6ed9eba1`

j	s_{4j}	s_{4j+1}	s_{4j+2}	s_{4j+3}
$0, .., 3$	3	7	11	19
$4, .., 7$	3	5	9	13
$8, .., 11$	3	9	11	15

State:

$$(R_i, R_{i-1}, R_{i-2}, R_{i-3})$$

Output:

$$(R_{44} + R_{-4}, R_{47} + R_{-1}, R_{46} + R_{-2}, R_{45} + R_{-3})$$

MD5 ([Riv92b])

Parameters:

$n = 128$	$m = 128$	$l = 512$
$r = 4$	$s = 64$	$w = 32$

Message Expansion:

$\sigma_k(i)$		i															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	1	6	11	0	5	10	15	4	9	14	3	8	13	2	7	12
	2	5	8	11	14	1	4	7	10	13	0	3	6	9	12	15	2
	3	0	7	14	5	12	3	10	1	8	15	6	13	4	11	2	9

Initial Value:

$$R_{-4} = \texttt{0x67452301}, \quad R_{-3} = \texttt{0x10325476}$$
$$R_{-2} = \texttt{0x98badcfe}, \quad R_{-1} = \texttt{0xefcdab89}$$

Step Operation:

$$R_i = R_{i-1} + (R_{i-4} + f_i(R_{i-1}, R_{i-2}, R_{i-3}) + W_i + K_i)^{\lll s_i}$$

i	f_i
$0, .., 15$	ITE
$16, .., 31$	ITE_{zxy}
$32, .., 47$	XOR
$48, .., 63$	ONX_{xzy}

j	s_{4j}	s_{4j+1}	s_{4j+2}	s_{4j+3}
$0, .., 3$	7	12	17	22
$4, .., 7$	5	9	14	20
$8, .., 11$	4	11	16	23
$12, .., 15$	6	10	15	21

K_i : (in hexadecimal notation from left to right)

0xd76aa478	0xe8c7b756	0x242070db	0xc1bdceee	0xf57c0faf	0x4787c62a	0xa8304613	0xfd469501
0x698098d8	0x8b44f7af	0xffff5bb1	0x895cd7be	0x6b901122	0xfd987193	0xa679438e	0x49b40821
0xf61e2562	0xc040b340	0x265e5a51	0xe9b6c7aa	0xd62f105d	0x02441453	0xd8a1e681	0xe7d3fbc8
0x21e1cde6	0xc33707d6	0xf4d50d87	0x455a14ed	0xa9e3e905	0xfcefa3f8	0x676f02d9	0x8d2a4c8a
0xfffa3942	0x8771f681	0x6d9d6122	0xfde5380c	0xa4beea44	0x4bdecfa9	0xf6bb4b60	0xbebfbc70
0x289b7ec6	0xeaa127fa	0xd4ef3085	0x04881d05	0xd9d4d039	0xe6db99e5	0x1fa27cf8	0xc4ac5665
0xf4292244	0x432aff97	0xab9423a7	0xfc93a039	0x655b59c3	0x8f0ccc92	0xffeff47d	0x85845dd1
0x6fa87e4f	0xfe2ce6e0	0xa3014314	0x4e0811a1	0xf7537e82	0xbd3af235	0x2ad7d2bb	0xeb86d391

State:

$$(R_i, R_{i-1}, R_{i-2}, R_{i-3})$$

Output:

$$(R_{60} + R_{-4}, R_{63} + R_{-1}, R_{62} + R_{-2}, R_{61} + R_{-3})$$

Extended MD4 ([Riv91])

Parameters:

$$\begin{array}{rrr} n = \;\; 256 & m = 256 & l = 512 \\ r = 2 \times 4 & s = \;\; 48 & w = \;\; 32 \end{array}$$

Message Expansion:

$\sigma_k(i)$		i															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
	2	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15

Initial Value:

$$\begin{array}{ll} R^L_{-4} = \texttt{0x67452301}, & R^L_{-3} = \texttt{0x10325476} \\ R^L_{-2} = \texttt{0x98badcfe}, & R^L_{-1} = \texttt{0xefcdab89} \end{array}$$

$$\begin{array}{ll} R^R_{-4} = \texttt{0x33221100}, & R^R_{-3} = \texttt{0xffeeccdd} \\ R^R_{-2} = \texttt{0xbbaa9988}, & R^R_{-1} = \texttt{0x77665544} \end{array}$$

Step Operation:

$$R^L_i = (R^L_{i-4} + f_i(R^L_{i-1}, R^L_{i-2}, R^L_{i-3}) + W_i + K^L_i)^{\lll s_i}$$

$$R^R_i = (R^R_{i-4} + f_i(R^R_{i-1}, R^R_{i-2}, R^R_{i-3}) + W_i + K^R_i)^{\lll s_i}$$

i	f_i	K^L_i	K^R_i
$0, .., 15$	ITE	`0x00000000`	`0x00000000`
$16, .., 31$	MAJ	`0x5a827999`	`0x50a28be6`
$32, .., 47$	XOR	`0x6ed9eba1`	`0x5c4dd124`

j	s_{4j}	s_{4j+1}	s_{4j+2}	s_{4j+3}
$0, .., 3$	3	7	11	19
$4, .., 7$	3	5	9	13
$8, .., 11$	3	9	11	15

State:

$$(R^L_i, R^L_{i-1}, R^L_{i-2}, R^L_{i-3}, R^R_i, R^R_{i-1}, R^R_{i-2}, R^R_{i-3})$$

Output:

$$\begin{aligned} (R^R_{44} + R^R_{-4},\ R^L_{47} + R^L_{-1},\ R^L_{46} + R^L_{-2},\ R^L_{45} + R^L_{-3}, \\ R^L_{44} + R^L_{-4},\ R^R_{47} + R^R_{-1},\ R^R_{46} + R^R_{-2},\ R^R_{45} + R^R_{-3}) \end{aligned}$$

RIPEMD-0 ([RIP95])

Parameters:

$$\begin{array}{rlrlrl} n = & 128 & m = & 128 & l = & 512 \\ r = & 2 \times 4 & s = & 48 & w = & 32 \end{array}$$

Message Expansion:

$\sigma_k(i)$	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	7	4	13	1	10	6	15	3	12	0	9	5	14	2	11	8
2	3	10	2	4	9	15	8	1	14	7	0	6	11	13	5	12

Initial Value:

$$R^L_{-4} = \texttt{0x67452301}, \quad R^L_{-3} = \texttt{0x10325476}$$
$$R^L_{-2} = \texttt{0x98badcfe}, \quad R^L_{-1} = \texttt{0xefcdab89}$$

$$R^R_{-4} = \texttt{0x67452301}, \quad R^R_{-3} = \texttt{0x10325476}$$
$$R^R_{-2} = \texttt{0x98badcfe}, \quad R^R_{-1} = \texttt{0xefcdab89}$$

Step Operation:

$$R^L_i = (R^L_{i-4} + f_i(R^L_{i-1}, R^L_{i-2}, R^L_{i-3}) + W_i + K^L_i)^{\lll s_i}$$
$$R^R_i = (R^R_{i-4} + f_i(R^R_{i-1}, R^R_{i-2}, R^R_{i-3}) + W_i + K^R_i)^{\lll s_i}$$

i	f_i	K^L_i	K^R_i
$0, .., 15$	ITE	0x00000000	0x50a28be6
$16, .., 31$	MAJ	0x5a827999	0x00000000
$32, .., 47$	XOR	0x6ed9eba1	0x5c4dd124

s_{16k+i}	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
1	7	6	8	13	11	9	7	15	7	12	15	9	7	11	13	12
2	11	13	14	7	14	9	13	15	6	8	13	6	12	5	7	5

State:

$$(R^L_i, R^L_{i-1}, R^L_{i-2}, R^L_{i-3}, R^R_i, R^R_{i-1}, R^R_{i-2}, R^R_{i-3})$$

Output:

$$(R^L_{-1} + R^L_{46} + R^R_{45},\ R^L_{-2} + R^L_{45} + R^R_{44},\ R^L_{-3} + R^L_{44} + R^R_{47},\ R^L_{-4} + R^L_{47} + R^R_{46})$$

RIPEMD-128 ([DBP96, Bos])

Parameters:

$$n = 128 \qquad m = 128 \qquad l = 512$$
$$r = 2 \times 4 \qquad s = 64 \qquad w = 32$$

Message Expansion:

$\sigma_k^L(i)$	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8
2	3	10	14	4	9	15	8	1	2	7	0	6	13	11	5	12
3	1	9	11	10	0	8	12	4	13	3	7	15	14	5	6	2

$\sigma_k^R(i)$	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0	5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12
1	6	11	3	7	0	13	5	10	14	15	8	12	4	9	1	2
2	15	5	1	3	7	14	6	9	11	8	12	2	10	0	4	13
3	8	6	4	1	3	11	15	0	5	12	2	13	9	7	10	14

Initial Value:

$R^L_{-4} = \texttt{0x67452301},\ R^L_{-3} = \texttt{0x10325476},\ R^L_{-2} = \texttt{0x98badcfe},\ R^L_{-1} = \texttt{0xefcdab89}$

$R^R_{-4} = \texttt{0x67452301},\ R^R_{-3} = \texttt{0x10325476},\ R^R_{-2} = \texttt{0x98badcfe},\ R^R_{-1} = \texttt{0xefcdab89}$

Step Operation:

$$\begin{aligned} R_i^L &= (R_{i-4}^L + f_i^L(R_{i-1}^L, R_{i-2}^L, R_{i-3}^L) + W_i + K_i^L)^{\lll s_i^L} \\ R_i^R &= (R_{i-4}^R + f_i^R(R_{i-1}^R, R_{i-2}^R, R_{i-3}^R) + W_i + K_i^R)^{\lll s_i^R} \end{aligned}$$

i	f_i^L	f_i^R	K_i^L	K_i^R
$0, .., 15$	XOR	ITE$_{zxy}$	0x00000000	0x50a28be6
$16, .., 31$	ITE	ONX	0x5a827999	0x5c4dd124
$32, .., 47$	ONX	ITE	0x6ed9eba1	0x6d703ef3
$48, .., 63$	ITE$_{zxy}$	XOR	0x8f1bbcdc	0x00000000

s^L_{16k+i} :

	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
1	7	6	8	13	11	9	7	15	7	12	15	9	11	7	13	12
2	11	13	6	7	14	9	13	15	14	8	13	6	5	12	7	5
3	11	12	14	15	14	15	9	8	9	14	5	6	8	6	5	12

s^R_{16k+i} :

	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0	8	9	9	11	13	15	15	5	7	7	8	11	14	14	12	6
1	9	13	15	7	12	8	9	11	7	7	12	7	6	15	13	11
2	9	7	15	11	8	6	6	14	12	13	5	14	13	13	7	5
3	15	5	8	11	14	14	6	14	6	9	12	9	12	5	15	8

State:

$$(R_i^L, R_{i-1}^L, R_{i-2}^L, R_{i-3}^L, R_i^R, R_{i-1}^R, R_{i-2}^R, R_{i-3}^R)$$

Output:

$$(R_{-1}^L + R_{62}^L + R_{61}^R,\ R_{-2}^L + R_{61}^L + R_{60}^R,\ R_{-3}^L + R_{60}^L + R_{63}^R,\ R_{-4}^L + R_{63}^L + R_{62}^R)$$

RIPEMD-160 ([DBP96, Bos])

Parameters:

$$n = 160 \qquad m = 160 \qquad l = 512$$
$$r = 2 \times 5 \qquad s = 80 \qquad w = 32$$

Message Expansion:

$\sigma^L_k(i)$	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1		7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8
2		3	10	14	4	9	15	8	1	2	7	0	6	13	11	5	12
3		1	9	11	10	0	8	12	4	13	3	7	15	14	5	6	2
4		4	0	5	9	7	12	2	10	14	1	3	8	11	6	15	13

$\sigma^R_k(i)$	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0		5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12
1		6	11	3	7	0	13	5	10	14	15	8	12	4	9	1	2
2		15	5	1	3	7	14	6	9	11	8	12	2	10	0	4	13
3		8	6	4	1	3	11	15	0	5	12	2	13	9	7	10	14
4		12	15	10	4	1	5	8	7	6	2	13	14	0	3	9	11

Initial Value:

$R^L_{-5} =$ `0xc059d148`, $R^L_{-4} =$ `0x7c30f4b8`, $R^L_{-3} =$ `0x1d840c95`, $R^L_{-2} =$ `0x98badcfe`, $R^L_{-1} =$ `0xefcdab89`
$R^R_{-5} =$ `0xc059d148`, $R^R_{-4} =$ `0x7c30f4b8`, $R^R_{-3} =$ `0x1d840c95`, $R^R_{-2} =$ `0x98badcfe`, $R^R_{-1} =$ `0xefcdab89`

Step Operation:

$$\begin{aligned} R^L_i &= (R^L_{i-5}{}^{\lll 10} + f^L_i(R^L_{i-1}, R^L_{i-2}, R^L_{i-3}{}^{\lll 10}) + W_i + K^L_i)^{\lll s_i} + R^L_{i-4}{}^{\lll 10} \\ R^R_i &= (R^R_{i-5}{}^{\lll 10} + f^R_i(R^R_{i-1}, R^R_{i-2}, R^R_{i-3}{}^{\lll 10}) + W_i + K^R_i)^{\lll s_i} + R^R_{i-4}{}^{\lll 10} \end{aligned}$$

i	f^L_i	f^R_i	K^L_i	K^R_i
$0,..,15$	XOR	ONX$_{yzx}$	0x00000000	0x50a28be6
$16,..,31$	ITE	ITE$_{zxy}$	0x5a827999	0x5c4dd124
$32,..,47$	ONX	ONX	0x6ed9eba1	0x6d703ef3
$48,..,63$	ITE$_{zxy}$	ITE	0x8f1bbcdc	0x7a6d76e9
$64,..,79$	ONX$_{yzx}$	XOR	0xa953fd4e	0x00000000

s^L_{16k+i} :

	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0	11	14	15	12	5	8	7	9	11	13	14	15	6	7	9	8
1	7	6	8	13	11	9	7	15	7	12	15	9	11	7	13	12
2	11	13	6	7	14	9	13	15	14	8	13	6	5	12	7	5
3	11	12	14	15	14	15	9	8	9	14	5	6	8	6	5	12
4	9	15	5	11	6	8	13	12	5	12	13	14	11	8	5	6

s^R_{16k+i} :

	i															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k 0	8	9	9	11	13	15	15	5	7	7	8	11	14	14	12	6
1	9	13	15	7	12	8	9	11	7	7	12	7	6	15	13	11
2	9	7	15	11	8	6	6	14	12	13	5	14	13	13	7	5
3	15	5	8	11	14	14	6	14	6	9	12	9	12	5	15	8
4	8	5	12	9	12	5	14	6	8	13	6	5	15	13	11	11

State:

$$(R^L_i, R^L_{i-1}, R^{L\lll 10}_{i-2}, R^{L\lll 10}_{i-3}, R^{L\lll 10}_{i-4}, R^R_i, R^R_{i-1}, R^{R\lll 10}_{i-2}, R^{R\lll 10}_{i-3}, R^{R\lll 10}_{i-4})$$

Output:

$$\begin{aligned}(R^L_{-1} + R^L_{78} + R^{R\lll 10}_{77}, R^L_{-2} + R^{L\lll 10}_{77} + R^{R\lll 10}_{76}, R^{L\lll 10}_{-3} + R^{L\lll 10}_{76} + R^{R\lll 10}_{75}, \\ R^{L\lll 10}_{-4} + R^{L\lll 10}_{75} + R^R_{79},\ R^{L\lll 10}_{-5} + R^L_{79} + R^{R\lll 10}_{78})\end{aligned}$$

RIPEMD-256 ([DBP96, Bos])

The specification of RIPEMD-256 is identical to that of RIPEMD-128, with the following exceptions:

Parameters:

$$\begin{array}{rrr} n = \ \ 256 & m = 256 & l = 512 \\ r = 2 \times 4 & s = \ \ 64 & w = \ \ 32 \end{array}$$

Initial Value:

$R^L_{-4} = \texttt{0x67452301},\ R^L_{-3} = \texttt{0x10325476},\ R^L_{-2} = \texttt{0x98badcfe},\ R^L_{-1} = \texttt{0xefcdab89}$
$R^R_{-4} = \texttt{0x76543210},\ R^R_{-3} = \texttt{0x01234567},\ R^R_{-2} = \texttt{0x89abcdef},\ R^R_{-1} = \texttt{0xfedcba98}$

Step Operation:
Like in RIPEMD-128, with the following addition:
After step 15 (and 31, 47, 63 respectively) exchange the contents of R^L_{12} with R^R_{12} (and of R^L_{31} with R^R_{31},R^L_{46} with R^R_{46},R^L_{61} with R^R_{61} respectively).

Output:

$$(R^L_{-4} + R^L_{60},\ R^L_{-1} + R^L_{63},\ R^L_{-2} + R^L_{62},\ R^L_{-3} + R^L_{61}$$
$$R^R_{-4} + R^R_{60},\ R^R_{-1} + R^R_{63},\ R^R_{-2} + R^R_{62},\ R^R_{-3} + R^R_{61})$$

RIPEMD-320 ([DBP96, Bos])

The specification of RIPEMD-320 is identical to that of RIPEMD-160, with the following exceptions:

Parameters:

$$\begin{array}{rrr} n = \ \ 320 & m = 320 & l = 512 \\ r = 2 \times 5 & s = \ \ 80 & w = \ \ 32 \end{array}$$

Initial Value:

$R^L_{-5} = \texttt{0xc059d148},\ R^L_{-4} = \texttt{0x7c30f4b8},\ R^L_{-3} = \texttt{0x1d840c95},\ R^L_{-2} = \texttt{0x98badcfe},\ R^L_{-1} = \texttt{0xefcdab89}$
$R^R_{-5} = \texttt{0x841d950c},\ R^R_{-4} = \texttt{0x83cf0b47},\ R^R_{-3} = \texttt{0x59c048d1},\ R^R_{-2} = \texttt{0x89abcdef},\ R^R_{-1} = \texttt{0xfedcba98}$

Step Operation:
Like in RIPEMD-160, with the following addition:
After step 15 (and 31, 47, 63, 79 respectively) exchange the contents of R^L_{14} with R^R_{14} (and of R^L_{27} with R^R_{27}, R^L_{46} with R^R_{46}, R^L_{59} with R^R_{59}, R^L_{77} with R^R_{77} respectively).

Output:

$$({R^L_{-5}}^{\lll 10} + {R^L_{75}}^{\lll 10},\ R^L_{-1} + R^L_{79},\ R^L_{-2} + R^L_{78},\ {R^L_{-3}}^{\lll 10} + {R^L_{77}}^{\lll 10},\ {R^L_{-4}}^{\lll 10} + {R^L_{76}}^{\lll 10}$$
$${R^R_{-5}}^{\lll 10} + {R^R_{75}}^{\lll 10},\ R^R_{-1} + R^R_{79},\ R^R_{-2} + R^R_{78},\ {R^R_{-3}}^{\lll 10} + {R^R_{77}}^{\lll 10}),\ {R^R_{-4}}^{\lll 10} + {R^R_{76}}^{\lll 10})$$

SHA-0([FIP]), SHA-1([FIP02])

Parameters:

$$\begin{array}{rrr} n = 160 & m = 160 & l = 512 \\ r = 5 & s = 80 & w = 32 \end{array}$$

Message Expansion:

$$\begin{array}{lrcl} \text{SHA-0:} & W_i & = & W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}, \quad i \in \{16, \ldots, 79\} \\ \text{SHA-1:} & W_i & = & (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})^{\lll 1}, \quad i \in \{16, \ldots, 79\} \end{array}$$

Initial Value:

$$\begin{array}{rcl} R_{-5} & = & \texttt{0x0f4b87c3} \\ R_{-4} & = & \texttt{0x40c951d8} \\ R_{-3} & = & \texttt{0x62eb73fa} \\ R_{-2} & = & \texttt{0xefcdab89} \\ R_{-1} & = & \texttt{0x67452301} \end{array}$$

Step Operation:

$$R_i := (R_{i-1}^{\lll 5}) + f_i(R_{i-2}, R_{i-3}^{\lll 30}, R_{i-4}^{\lll 30}) + R_{i-5}^{\lll 30} + W_i + K_i$$

i	f_i	K_i
$0, .., 19$	ITE	`0x5a827999`
$20, .., 39$	XOR	`0x6ed9eba1`
$40, .., 59$	MAJ	`0x8f1bbcdc`
$60, .., 79$	XOR	`0xca62c1d6`

State:

$$(R_i, R_{i-1}, R_{i-2}^{\lll 30}, R_{i-3}^{\lll 30}, R_{i-4}^{\lll 30})$$

Output:

$$(R_{-1} + R_{79}, R_{-2} + R_{78}, R_{-3}^{\lll 30} + R_{77}^{\lll 30}, R_{-4}^{\lll 30} + R_{76}^{\lll 30}, R_{-5}^{\lll 30} + R_{75}^{\lll 30})$$

SHA-256([FIP02])

Parameters:

$$\begin{array}{rrr} n = 256 & m = 256 & l = 512 \\ r = 8 & s = 64 & w = 32 \end{array}$$

Message Expansion:

$$\begin{array}{rcl} W_i & = & \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} \\ \text{where } \sigma_0(x) & := & (x^{\ggg 7}) \oplus (x^{\ggg 18}) \oplus (x^{\gg 3}) \\ \sigma_1(x) & := & (x^{\ggg 17}) \oplus (x^{\ggg 19}) \oplus (x^{\gg 10}) \end{array}$$

Initial Value:

$$\begin{array}{ll} R_{-4} = \texttt{0xa54ff53a} & T_{-4} = \texttt{0x5be0cd19} \\ R_{-3} = \texttt{0x3c6ef372} & T_{-3} = \texttt{0x1f83d9ab} \\ R_{-2} = \texttt{0xbb67ae85} & T_{-2} = \texttt{0x9b05688c} \\ R_{-1} = \texttt{0x6a09e667} & T_{-1} = \texttt{0x510e527f} \end{array}$$

Step Operation:

$$\begin{array}{rcl} T_i & = & T + R_{i-4} \\ R_i & = & T + \Sigma_0(R_{i-1}) + \mathrm{MAJ}(R_{i-1}, R_{i-2}, R_{i-3}) \\ \text{where} \quad T & = & T_{i-4} + \Sigma_1(T_{i-1}) + \mathrm{ITE}(T_{i-1}, T_{i-2}, T_{i-3}) + K_i + W_i \\ \Sigma_0(x) & = & (x^{\ggg 2}) \oplus (x^{\ggg 13}) \oplus (x^{\ggg 22}) \\ \Sigma_1(x) & = & (x^{\ggg 6}) \oplus (x^{\ggg 11}) \oplus (x^{\ggg 25}) \end{array}$$

K_i : (in hexadecimal notation from left to right)

0x428a2f98	0x71374491	0xb5c0fbcf	0xe9b5dba5	0x3956c25b	0x59f111f1	0x923f82a4	0xab1c5ed5
0xd807aa98	0x12835b01	0x243185be	0x550c7dc3	0x72be5d74	0x80deb1fe	0x9bdc06a7	0xc19bf174
0xe49b69c1	0xefbe4786	0x0fc19dc6	0x240ca1cc	0x2de92c6f	0x4a7484aa	0x5cb0a9dc	0x76f988da
0x983e5152	0xa831c66d	0xb00327c8	0xbf597fc7	0xc6e00bf3	0xd5a79147	0x06ca6351	0x14292967
0x27b70a85	0x2e1b2138	0x4d2c6dfc	0x53380d13	0x650a7354	0x766a0abb	0x81c2c92e	0x92722c85
0xa2bfe8a1	0xa81a664b	0xc24b8b70	0xc76c51a3	0xd192e819	0xd6990624	0xf40e3585	0x106aa070
0x19a4c116	0x1e376c08	0x2748774c	0x34b0bcb5	0x391c0cb3	0x4ed8aa4a	0x5b9cca4f	0x682e6ff3
0x748f82ee	0x78a5636f	0x84c87814	0x8cc70208	0x90befffa	0xa4506ceb	0xbef9a3f7	0xc67178f2

State:

$$(R_i, R_{i-1}, R_{i-2}, R_{i-3}, T_i, T_{i-1}, T_{i-2}, T_{i-3})$$

Output:

$$\begin{array}{l} (R_{-1} + R_{63},\ R_{-2} + R_{62},\ R_{-3} + R_{61},\ R_{-4} + R_{60}, \\ \qquad T_{-1} + T_{63},\ T_{-2} + T_{62},\ T_{-3} + T_{61},\ T_{-4} + T_{60}) \end{array}$$

SHA-512([FIP02])

Parameters:

$$n = 512 \qquad m = 512 \qquad l = 1024$$
$$r = 8 \qquad s = 80 \qquad w = 64$$

Message Expansion:

$$\begin{aligned} W_i &= \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} \\ \text{where } \sigma_0(x) &:= (x^{\ggg 1}) \oplus (x^{\ggg 8}) \oplus (x^{\gg 7}) \\ \sigma_1(x) &:= (x^{\ggg 19}) \oplus (x^{\ggg 61}) \oplus (x^{\gg 6}) \end{aligned}$$

Initial Value:

$R_{-4} =$ `0xa54ff53a5f1d36f1` $T_{-4} =$ `0x5be0cd19137e2179`
$R_{-3} =$ `0x3c6ef372fe94f82b` $T_{-3} =$ `0x1f83d9abfb41bd6b`
$R_{-2} =$ `0xbb67ae8584caa73b` $T_{-2} =$ `0x9b05688c2b3e6c1f`
$R_{-1} =$ `0x6a09e667f3bcc908` $T_{-1} =$ `0x510e527fade682d1`

Step Operation:

$$\begin{aligned} T_i &= T + R_{i-4} \\ R_i &= T + \Sigma_0(R_{i-1}) + \mathrm{MAJ}(R_{i-1}, R_{i-2}, R_{i-3}) \\ \text{where} \quad T &= T_{i-4} + \Sigma_1(T_{i-1}) + \mathrm{ITE}(T_{i-1}, T_{i-2}, T_{i-3}) + K_i + W_i \\ \Sigma_0(x) &= (x^{\ggg 28}) \oplus (x^{\ggg 34}) \oplus (x^{\ggg 39}) \\ \Sigma_1(x) &= (x^{\ggg 14}) \oplus (x^{\ggg 18}) \oplus (x^{\ggg 41}) \end{aligned}$$

K_i : (in hexadecimal notation from left to right)

0x428a2f98d728ae22	0x7137449123ef65cd	0xb5c0fbcfec4d3b2f	0xe9b5dba58189dbbc
0x3956c25bf348b538	0x59f111f1b605d019	0x923f82a4af194f9b	0xab1c5ed5da6d8118
0xd807aa98a3030242	0x12835b0145706fbe	0x243185be4ee4b28c	0x550c7dc3d5ffb4e2
0x72be5d74f27b896f	0x80deb1fe3b1696b1	0x9bdc06a725c71235	0xc19bf174cf692694
0xe49b69c19ef14ad2	0xefbe4786384f25e3	0x0fc19dc68b8cd5b5	0x240ca1cc77ac9c65
0x2de92c6f592b0275	0x4a7484aa6ea6e483	0x5cb0a9dcbd41fbd4	0x76f988da831153b5
0x983e5152ee66dfab	0xa831c66d2db43210	0xb00327c898fb213f	0xbf597fc7beef0ee4
0xc6e00bf33da88fc2	0xd5a79147930aa725	0x06ca6351e003826f	0x142929670a0e6e70
0x27b70a8546d22ffc	0x2e1b21385c26c926	0x4d2c6dfc5ac42aed	0x53380d139d95b3df
0x650a73548baf63de	0x766a0abb3c77b2a8	0x81c2c92e47edaee6	0x92722c851482353b
0xa2bfe8a14cf10364	0xa81a664bbc423001	0xc24b8b70d0f89791	0xc76c51a30654be30
0xd192e819d6ef5218	0xd69906245565a910	0xf40e35855771202a	0x106aa07032bbd1b8
0x19a4c116b8d2d0c8	0x1e376c085141ab53	0x2748774cdf8eeb99	0x34b0bcb5e19b48a8
0x391c0cb3c5c95a63	0x4ed8aa4ae3418acb	0x5b9cca4f7763e373	0x682e6ff3d6b2b8a3
0x748f82ee5defb2fc	0x78a5636f43172f60	0x84c87814a1f0ab72	0x8cc702081a6439ec
0x90befffa23631e28	0xa4506cebde82bde9	0xbef9a3f7b2c67915	0xc67178f2e372532b
0xca273eceea26619c	0xd186b8c721c0c207	0xeada7dd6cde0eb1e	0xf57d4f7fee6ed178
0x06f067aa72176fba	0x0a637dc5a2c898a6	0x113f9804bef90dae	0x1b710b35131c471b
0x28db77f523047d84	0x32caab7b40c72493	0x3c9ebe0a15c9bebc	0x431d67c49c100d4c
0x4cc5d4becb3e42b6	0x597f299cfc657e2a	0x5fcb6fab3ad6faec	0x6c44198c4a475817

State:

$$(R_i, R_{i-1}, R_{i-2}, R_{i-3}, T_i, T_{i-1}, T_{i-2}, T_{i-3})$$

Output:

$$\begin{aligned} &(R_{-1} + R_{79},\ R_{-2} + R_{78},\ R_{-3} + R_{77},\ R_{-4} + R_{76}, \\ &\quad T_{-1} + T_{79},\ T_{-2} + T_{78},\ T_{-3} + T_{77},\ T_{-4} + T_{76}) \end{aligned}$$

SHA-224([FIP02])

The definition of SHA-224 is identical to that of SHA-256, with the following exceptions:

Parameters:

$$\begin{array}{rrr} n = 224 & m = 256 & l = 512 \\ r = 8 & s = 64 & w = 32 \end{array}$$

Initial Value:

$$\begin{array}{ll} R_{-4} = \texttt{0xf70e5939} & T_{-4} = \texttt{0xbefa4fa4} \\ R_{-3} = \texttt{0x3070dd17} & T_{-3} = \texttt{0x64f98fa7} \\ R_{-2} = \texttt{0x367cd507} & T_{-2} = \texttt{0x68581511} \\ R_{-1} = \texttt{0xc1059ed8} & T_{-1} = \texttt{0xffc00b31} \end{array}$$

Output:
The output of the compression function is identical to that of SHA-256, the output of the hash function is truncated to

$$(R_{-1} + R_{63},\ R_{-2} + R_{62},\ R_{-3} + R_{61},\ R_{-4} + R_{60},\\ T_{-1} + T_{63},\ T_{-2} + T_{62},\ T_{-3} + T_{61}).$$

SHA-384([FIP02])

The definition of SHA-384 is identical to that of SHA-512, with the following exceptions:

Parameters:

$$\begin{array}{rrr} n = 384 & m = 512 & l = 1024 \\ r = 8 & s = 80 & w = 64 \end{array}$$

Initial Value:

$$\begin{array}{ll} R_{-4} = \texttt{0x152fecd8f70e5939} & T_{-4} = \texttt{0x47b5481dbefa4fa4} \\ R_{-3} = \texttt{0x9159015a3070dd17} & T_{-3} = \texttt{0xdb0c2e0d64f98fa7} \\ R_{-2} = \texttt{0x629a292a367cd507} & T_{-2} = \texttt{0x8eb44a8768581511} \\ R_{-1} = \texttt{0xcbbb9d5dc1059ed8} & T_{-1} = \texttt{0x67332667ffc00b31} \end{array}$$

Output:
The output of the compression function is identical to that of SHA-512, the output of the hash function is truncated to

$$(R_{-1} + R_{79},\ R_{-2} + R_{78},\ R_{-3} + R_{77},\ R_{-4} + R_{76}, T_{-1} + T_{79},\ T_{-2} + T_{78}).$$

Bibliography

[BC04a] E. Biham and R. Chen. *Near-Collisions of SHA-0.* In *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *LNCS*, pages 290–305. Springer-Verlag, 2004.

[BC04b] E. Biham and R. Chen. *Near-Collisions of SHA-0 and SHA-1.* Presented at SAC 2004, Waterloo, Canada, August 2004. (`http://www.cs.technion.ac.il/~biham/`).

[BCJ+05] E. Biham, R. Chen, A. Joux, P. Carribault, W. Jalby, and C. Lemuet. *Collisions of SHA-0 and Reduced SHA-1.* In *Advances in Cryptology – EUROCRYPT 2005*, LNCS. Springer-Verlag, 2005.

[Bos] A. Bosselaers. *The RIPEMD-160 page.* (`http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html`).

[CJ98] F. Chabaud and A. Joux. *Differential Collisions in SHA-0.* In *Advances in Cryptology – CRYPTO'98*, volume 1462 of *LNCS*, pages 56–71. Springer-Verlag, 1998.

[Dam90] I. Damgård. *A Design Principle for Hash Functions.* In *Advances in Cryptology – Crypto '89*, volume 435 of *LNCS*. Springer-Verlag, 1990.

[Dau05] M. Daum. *Narrow T-functions.* In *FSE 2005*, volume 3557 of *LNCS*, pages 50–67. Springer-Verlag, 2005.

[dBB92] B. den Boer and A. Bosselaers. *An attack on the last two rounds of MD4.* In *Advances in Cryptology – Crypto '91*, volume 576 of *LNCS*, pages 194–203. Springer-Verlag, 1992.

[dBB94] B. den Boer and A. Bosselaers. *Collisions for the compression function of MD5.* In *Advances in Cryptology – Eurocrypt '93*, volume 773 of *LNCS*, pages 293–304. Springer-Verlag, 1994.

[DBP96] H. Dobbertin, A. Bosselaers, and B. Preneel. *RIPEMD-160: A strengthened version of RIPEMD.* In *Fast Software Encryption – Cambridge Workshop*, volume 1039 of *LNCS*, pages 71–82. Springer-Verlag, 1996.

[DD06] M. Daum and H. Dobbertin. *Hashes and Message Digests.* In Hossein Bidgoli, editor, *The Handbook of Information Security*, chapter 109. John Wiley & Sons, to appear in 2006.

[Dob95] H. Dobbertin. *Alf swindles Ann.* *CryptoBytes*, 1(3):5, 1995.

[Dob96a] H. Dobbertin. *Cryptanalysis of MD4.* In *Fast Software Encryption – Cambridge Workshop*, volume 1039 of *LNCS*, pages 53–69. Springer-Verlag, 1996.

[Dob96b] H. Dobbertin. *Cryptanalysis of MD5.* Presented at rump session of EuroCrypt '96, 1996.

[Dob96c] H. Dobbertin. *The status of MD5 after a recent attack.* *CryptoBytes*, 2(2):1–6, 1996.

[Dob97] H. Dobbertin. *RIPEMD with two-round compress function is not collision-free.* *Journal of Cryptology*, 10:51–68, 1997.

[Dob98a] H. Dobbertin. *Cryptanalysis of MD4.* *Journal of Cryptology*, 11:253–274, 1998.

[Dob98b] H. Dobbertin. *The First Two Rounds of MD4 are Not One-Way.* In *Fast Software Encryption '98*, volume 1372 of *LNCS*, pages 284–292. Springer-Verlag, 1998.

[FIP] *Federal Information Processing Standard 180, Secure Hash Standard.*

[FIP02] *Federal Information Processing Standard 180-2, Secure Hash Standard*, August 2002. (`http://www.csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`).

[JCJL04] A. Joux, P. Carribault, W. Jalby, and C. Lemuet. *Collisions in SHA-0.* Presented at the rump session of CRYPTO 2004, August 2004.

[Jou04] A. Joux. *Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions.* In *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *LNCS*, pages 306 – 316. Springer-Verlag, 2004.

[Kam04] D. Kaminsky. *MD5 to be considered harmful someday.* preprint, December 2004. (`http://www.doxpara.com/md5someday.pdf`).

[Kli04] A. Klimov. *Applications of T-functions in Cryptography.* PhD thesis, Weizmann Institute of Science, 2004. (`http://www.wisdom.weizmann.ac.il/~ask/`).

[Kli05] V. Klima. *Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications.* Cryptology ePrint Archive, Report 2005/102, 2005. `http://eprint.iacr.org/`.

[Knu98] D. E. Knuth. *The art of computer programming.* Addison Wesley Longman, 1998.

[KP00] P. Kasselman and W. Penzhorn. *Cryptanalysis of Reduced Version of HAVAL. IEEE Electronic Letters*, 36(1):30–31, 2000.

[Kra04] M. Krause. *OBDD-based Cryptanalysis of Oblivious Keystream Generators.* Technical report, University of Mannheim, 2004. To appear in TOCS.

[KS02] A. Klimov and A. Shamir. *A New Class of Invertible Mappings.* In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *LNCS*. Springer-Verlag, 2002.

[KS03] A. Klimov and A. Shamir. *Cryptographic Applications of T-functions.* In *Selected Areas in Cryptography (SAC)*, volume 3006 of *LNCS*. Springer-Verlag, 2003.

[KS04] A. Klimov and A. Shamir. *New Cryptographic Primitives Based on Multiword T-functions.* In *FSE 2004*, volume 3017 of *LNCS*. Springer-Verlag, 2004.

[KS05] A. Klimov and A. Shamir. *New Applications of T-functions in Block Ciphers and Hash Functions.* In *FSE 2005*, volume 3557 of *LNCS*. Springer, 2005.

[LD05] S. Lucks and M. Daum. *The Story of Alice and her Boss.* Presented at the rump session of Eurocrypt '05, May 2005. (see also [PMA]).

[LdW05] A. Lenstra and B. de Weger. *On the possibility of constructing meaningful hash collisions for public keys.* In *ACISP 2005*, volume 3574 of *LNCS*, pages 267–279. Springer-Verlag, 2005. (for full version see `http://www.win.tue.nl/~bdeweger/CollidingCertificates/ddl-full.pdf`).

[Lin02] N. Linial. *Finite Metric Spaces - Combinatorics, Geometry and Algorithms.* In *Proceedings of the International Congress of Mathematicians III*, pages 573–586, 2002.

[Luc04] S. Lucks. *Design Principles for Iterated Hash Functions.* Cryptology ePrint Archive, Report 2004/253, 2004. `http://eprint.iacr.org/`.

[LWdW05] A. Lenstra, X. Wang, and B. de Weger. *Colliding X.509 Certificates.* Cryptology ePrint Archive, Report 2005/067, 2005. `http://eprint.iacr.org/`.

[Mer90] R. Merkle. *One Way Hash Functions and DES.* In *Advances in Cryptology – CRYPTO '89*, volume 435 of *LNCS*. Springer-Verlag, 1990.

[Mik04] O. Mikle. *Practical Attacks on Digital Signatures Using MD5 Message Digest.* Cryptology ePrint Archive, Report 2004/356, 2004. `http://eprint.iacr.org/`.

[MO90] F. Morain and J. Olivos. *Speeding Up the Computations on an Elliptic Curve Using Addition-Subtraction Chains.* *Theoretical Informatics and Applications*, 24(6):531–544, 1990.

[MvOV96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[PMA] *The Poisoned Message Attack Website.* (`http://www.cits.rub.de/MD5Collisions/`).

[RBPV03] B. van Rompay, A. Biryukov, B. Preneel, and J. Vandewalle. *Cryptanalysis of 3-pass HAVAL.* In *Advances in Cryptology – Asiacrypt'2003*, volume 2894 of *LNCS*, pages 228–245. Springer-Verlag, 2003.

[Rei60] G. W. Reitwiesner. *Binary Arithmetic.* *Advances in Computers*, 1:231–308, 1960.

[RIP95] RIPE Consortium. *Ripe Integrity Primitives – Final report of RACE Integrity Primitives Evaluation (R1040)*, volume 1007 of *LNCS*. Springer-Verlag, 1995.

[Riv91] R. Rivest. *The MD4 message digest algorithm.* In *Advances in Cryptology – Crypto '90*, volume 537 of *LNCS*, pages 303–311. Springer-Verlag, 1991.

[Riv92a] R. Rivest. *The MD4 message-digest algorithm, Request for Comments (RFC) 1320.* Internet Activities Board, Internet Privacy Task Force, 1992.

[Riv92b] R. Rivest. *The MD5 message-digest algorithm, Request for Comments (RFC) 1321.* Internet Activities Board, Internet Privacy Task Force, 1992.

[RO05] V. Rijmen and E. Oswald. *Update on SHA-1.* In *CT-RSA*, volume 3376 of *LNCS*, pages 58–71. Springer-Verlag, 2005.

[Rue91] R. A. Rueppel. *Stream Ciphers.* In G. J. Simmons, editor, *Contemporary Cryptology.* IEEE Press, 1991.

[Vau95] S. Vaudenay. *On the need for multipermutations: cryptanalysis of MD4 and SAFER.* In *Fast Software Encryption – Leuven Workshop*, volume 1008 of *LNCS*, pages 286–297. Springer-Verlag, 1995.

[Weg00] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications.* SIAM Monographs on Discrete Mathematics and Applications, 2000.

[WL04] X. Wang and X. Lai. Private Communication, November 2004.

[WLF$^+$05] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. *Cryptanalysis for Hash Functions MD4 and RIPEMD.* In *Advances in Cryptology – EUROCRYPT 2005*, LNCS. Springer-Verlag, 2005.

[WLFY04] X. Wang, X. Lai, D. Feng, and H. Yu. *Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD.* Presented at the rump session of CRYPTO 2004, August 2004. (`http://eprint.iacr.org/2004/199`).

[WY05] X. Wang and H. Yu. *How to Break MD5 and Other Hash Functions.* In *Advances in Cryptology – EUROCRYPT 2005*, LNCS. Springer-Verlag, 2005.

[WYY05] X. Wang, Y. L. Yin, and H. Yu. *Collision Search Attacks on SHA1.* preprint, February 2005. (`http://www.infosec.sdu.edu.cn/paper/sha-attack-note.pdf`).

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

[ZPS93] Y. Zheng, J. Pieprzyk, and J. Seberry. *HAVAL - a one-way hashing algorithm with variable length and output.* In *Advances in Cryptology – Auscrypt '92*, Lecture Notes in Computer Science, pages 83–104. Springer-Verlag, 1993.

List of Symbols

$\{0,1\}^*$	set of bit strings of arbitrary length
$\mathbb{1}()$	characteristic function with $\mathbb{1}(x) = 1 \iff x$ is true ($\mathbb{1}(x) = 0$ otherwise)
$X \| Y$	concatenation of the two bitstrings X and Y
$[X]_i$	i-th bit of X
$\lfloor x \rfloor_k$	number $\tilde{x} \in \mathbb{Z}$ with $0 \leq \tilde{x} \leq 2^k - 1$ and $\tilde{x} = x \bmod 2^k$
$\mathbb{F}_q$	finite field with q elements
$\mathbb{F}_2^n$	n-dimensional vector space over $\mathbb{F}_2$
$\mathbb{Z}_{2^n}$	ring of integers modulo 2^n
$\oplus$	addition in $\mathbb{F}_2^n$
$+$	addition in $\mathbb{Z}$ or in $\mathbb{Z}_{2^n}$
$+_{\mathbb{Z}}$	addition in $\mathbb{Z}$
$+_n$	addition in $\mathbb{Z}_{2^n}$
$\Delta^{\oplus}(x, x')$	$\oplus$-difference of x and x' ($\Delta^{\oplus}(x, x') = x \oplus x'$)
$\Delta^{\oplus} x$	short version of $\Delta^{\oplus}(x, x')$
$\Delta^{+}(x, x')$	modular difference of x and x' ($\Delta^{+}(x, x') = x - x'$)
$\Delta^{+} x$	short version of $\Delta^{+}(x, x')$
$\Delta^{\pm}(x, x')$	signed bitwise difference of x and x' ($\left[\Delta^{\pm}(x, x')\right]_i = x_i - x_i'$)
$\Delta^{\pm} x$	short version of $\Delta^{\pm}(x, x')$
$[i_1, \dots, i_r]$	$= (x_{n-1}, \dots, x_0)$, if $x_j = 1$ for all $j \in \{i_1, \dots, i_r\}$ ($x_j = 0$ else) ($\overline{i_k}$ means that $x_{i_k} = -1$ instead of $x_{i_k} = 1$)
ITE	$\mathrm{ITE}(x, y, z) = x \oplus y \oplus z$ (cf. Definition 3.4)
MAJ	$\mathrm{MAJ}(x, y, z) = xy \oplus xz \oplus yz$ (cf. Definition 3.4)
ONX	$\mathrm{ONX}(x, y, z) = xy \oplus \overline{x}z$ (cf. Definition 3.4)
XOR	$\mathrm{XOR}(x, y, z) = (x \vee \overline{y}) \oplus z$ (cf. Definition 3.4)

f_{zxy}	bitwise applied Boolean function with swapped parameters, e.g. $f_{zxy}(x, y, z) = f(z, x, y)$

$d_H(x, y)$	Hamming-distance of x and y (cf. Example 4.17)
$d_M(x, y)$	modular distance of x and y (cf. Example 4.18)
$d_N^{\oplus}(x, y)$	NAF-distance with respect to $\oplus$ (cf. Definition 4.31)
$d_N^{+}(x, y)$	NAF-distance with respect to $+$ (cf. Definition 4.31)
$w_H(x)$	Hamming-weight of x (cf. Example 4.17)
$w_M(x)$	modular weight of x (cf. Example 4.18)
$w_N(x)$	NAF-weight of x (cf. Definition 4.27)

σ_k	permutation used in k-th round
f_i	bitwise Boolean function applied in step i
IV	initial value or chaining value used to initialize the registers of the compression function
K_i	constant used in the step operation in step i
R_i	content of register changed in step i after step i
s_i	amount of bit rotation applied in step i
W_i	input word used in step i, dependent on message
$X^{(i)}$	i-th block of the input message
X_i	i-th word of one input message block

$x^{\ll r}$	shift of x by r bits to the left
$x^{\lll r}$	rotation of x by r bits to the left
$x^{\gg r}$	shift of x by r bits to the right
$x^{\ggg r}$	rotation of x by r bits to the right

Index

applications, 2
approximating functions, 64
approximation, 93
  $\mathbb{F}_2$-linear, 97
asymmetric cryptography, 5
attack
  Dobbertin's, 28, 68, 118, 124
    extensions, 93
  generic, 16
  Wang's, 28, 71, 118
  Yuval's, 16
attack strategy, 74
attacks
  historical overview, 73
avalanche effect, 8, 39, 49–81
  quantifying, 57
avalanche factor, **58**, 60, 89
  backward computation, 61
  step dependance, 62

balanced, 32
BDD, 130
binary decision diagrams, 130
birthday bound, 143
birthday paradox, 16
  generalized, 16
bit rotation, 31, 40, 45–49, 65, 139
  medium-sized, 48
bit shift, 31
bitwise Boolean operations, 31
bitwise defined, 31
bitwise function, 67, 121, 122, 145
broken
  academically, 15, 17
  practically, 15

cancel differences, 71
chaining value, 19
collision, **14**, 19
  elementary, 98, 103
  meaningful, 92, 144
  multiblock, 76
  of the compression function, **20**, 79
  of the hash function, 79, 86
  practically relevant, 118
  pseudo-, **20**, 79
collision resistant, 3, **14**, 20, 143
compression function, **18**, 24
  general structure, 23
connecting inner collisions, 90
connection, 83, 141
connection part, 85
connections between $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$, 39
constraints, 77, 79–84
continuous approximation, 88, 91
corrective pattern, 98
cost function, 115

design philosophies, 5
difference pattern, 9, 61, **74**, 78, 84, 98, 103, 105, 111
  finding, 102
  modular, 65
  search for, 112
difference patterns
  tree of, 105, 112

difference propagation, 9, 40, 62, 67, 70
- $\oplus$-, 63, 64
- bitwise function, 105
- controlling, 62
- for modular differences, 65
- rotation, 105

differences, 62
- $\oplus$-, 41–45, 63, 66, 68, 75
- input, 67, **75**
- modular, 41–49, 65, 66, 68, 75, 77, 102, 114
- output, 67, **75**
- signed bitwise, 8, 39, 41–45, 67, 112, 114
  - propagation, 69

diffusion, 29
digital fingerprint, 1
digital signature scheme, 2
distortion, **50**
- maximal, 51

Dobbertin's method, 9, 77, 103, 117, 124, 140

efficiency, 5, 7, 146
empirical results, 59
equivalent vertices, **130**
evolutionary process, 86, 89, 146
expansion factor, 57

$\mathbb{F}_2^n$, 40
Floyd's Algorithm, 17
framework, 23, 39

generalized solution graph, **138**
generic attacks, 16
generic birthday attack, 118, 143, 145
group operations, 40

Hamming distance, **50**
Hamming weight, **50**
hash function
- cryptographic, 1, **13**
  - ideal, 14
- iterated, 19
- non-cryptographic, 2

hash value, **13**
HAVAL, 27, 28, 73, 77, 111

identification of $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$, 40
incompatible operations, 8
inner almost collision, **84**, 87, 94
inner collision, **81**, 83, 104, 116, 140, 141
- finding, 86

input difference pattern, **75**
ITE, 32, 63, 64
iterated compression, 17
iterated hash function, 19

length of the chaining value, 25
lines of computation, 27
Lipschitz constant, 57

MAJ, 32, 63, 64
MD-design principle, 17, 76
MD-family, 27
MD-strengthening, 19, 76
MD4, 6, 28, 34, 59, 60, 73, 77, 91, 103, 105, 108, 111, 118, 121, 144
- Extended, 6, 28, 73, 77, 92

MD4-family, 6, 23, 26, 143
MD5, 6, 28, 34, 59, 60, 68, 73, 76, 77, 91, 109, 111, 118, 121, 144
meaningful messages, 118
Merkle-Damgård theorem, 20, 76
message expansion, 23, 27, 28
- by roundwise permutation, 77
- MD5, 28
- recursive, 95
- RIPEMD-160, 29
- SHA-0, 30, 96
- SHA-1, 30, 96

SHA-224, 30
SHA-256, 30
message modifications, 106
metric, 49
maximum, 58
sum, 58
modular addition, 31, 40, 42, 63, 121, 122, 145
modular subtraction, 42
multi-step modification, 106–111
different kinds, 109
multiblock collisions, 76
multiblock messages, 111
multicollision, 21

NAF, **52**
NAF-distance, 39, **55**, 59
$\oplus$-, **55**
average, 55
modular, **55**
properties, 56
NAF-weight, **53**, 62, 71, 76
average, 55
narrowness, 10, 121, **126**, 130, 133
near-collision, **76**
neutral bit, **100**
neutral bits method, 100
non-adjacent form, **52**
non-linearity, 32, 63
normed distortion, **51**
notation, 7, 25, 32, 39
number of input words, 25
number of steps, 25

one-way, 3, 15
ONX, 32, 63, 64
output difference pattern, **75**
output length
of the compression function, 25
of the hash function, 24
overlappings, 85

parameters, 24, 25
perspectives, 143, 145
perturbations, 98
practical impact, 118
practical relevance, 143
preimage resistance, 143, 144
preimage resistant, **14**
proofs of security, 144
propagation
of bitwise differences, 9
of modular differences, 9
pruning, 114, 115
pseudo-collision, **20**, 79
resistance, 76
resistant, 20
pseudo-collisions, 76
public key, 5

quasi-norm, **49**

random function, 61
randomized search, 99
recursive message expansion, 27, 29
register, 24
RIPEMD, 28
-0, 6, 25, 28, 73, 77, 91, 92, 111, 121
-128, 6, 35, 143, 144
-160, 6, 35, 59, 61, 143, 145
-256, 6, 36, 143, 145
-320, 6, 36, 143, 145
RIPEMD-family, 27
root, 127
roundwise permutation, 27, 28

second preimage resistant, **14**
secret key, 5
SHA
-0, 6, 25, 30, 36, 73, 76, 95, 97–99, 102
-1, 6, 30, 36, 59, 60, 73, 95, 102, 140, 144, 145

-224, 7, 25, 30, 36, 143, 145
-256, 6, 30, 36, 143, 145
-384, 6, 25, 30, 37, 143, 145
-512, 6, 25, 30, 37, 143, 145
SHA-family, 27
simultaneous-neutral, **101**
single-step modification, 106, 108, 111
sink, 127
software implementations, 25
solution graph, 10, 95, 117, 121, **127**
algorithms, 130
combining, 136
construction, 128
enumerate solutions, 134
generalized, 117, **138**, 141
intersection, 136
minimal size, 132
number of solutions, 135
reduction of the size, 131
width, **127**
step operation, 23, 31, 33, 49
reversed, 60
subfamilies, 27
symmetric cryptography, 4
system of equations, 77, 82, 87, 95, 122, 126, 140, 142

T-function, 10, 121, 125, **125**, 146
tree of solutions, 121, 123

w-narrow, 121, **125**, 127, 130, 133
Wang's method, 9, 102, 112, 118
weight, **49**

X.509 certificates, 118
XOR, 32

$\mathbb{Z}_{2^n}$, 40