KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK–ESAT
Kasteelpark Arenberg 10, 3001 Leuven-Heverlee

ANALYSIS AND DESIGN OF CRYPTOGRAPHIC HASH FUNCTIONS, MAC ALGORITHMS AND BLOCK CIPHERS

Promotoren:
Prof. Dr. ir. B. PRENEEL
Prof. Dr. ir. J. VANDEWALLE

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Bart VAN ROMPAY

Juni 2004

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK–ESAT
Kasteelpark Arenberg 10, 3001 Leuven-Heverlee

ANALYSIS AND DESIGN OF CRYPTOGRAPHIC HASH FUNCTIONS, MAC ALGORITHMS AND BLOCK CIPHERS

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Bart VAN ROMPAY

U.D.C. 681.3*D46

Juni 2004

D/2004/7515/65

ISBN 90-5682-527-5

Acknowledgements

It is a pleasure for me to thank all the people who have helped me to realise this Ph.D. thesis.

In the first place, I want to express my gratitude to Prof. Bart Preneel and Prof. Joos Vandewalle for being the promotors of this thesis. I appreciate their support and the opportunity they gave me to finish this work after a one-year round-the-world-travelling break. I also want to thank Bart Preneel for carefully reading and correcting earlier drafts of this thesis.

I am very grateful to Prof. Jean-Jacques Quisquater, Prof. Marc Van Barel, Prof. André Barbé, Prof. Lars Knudsen and Prof. Vincent Rijmen for agreeing to serve as jury members, and to Prof. Ludo Froyen for chairing the jury.

Special thanks go to Vincent Rijmen, for introducing me to cryptanalysis and providing me with assistance along the way. I have been very fortunate to work with some of the best cryptographers, including Vincent and my other co-authors Lars Ramkilde Knudsen, Bert den Boer and Alex Biryukov. Joan Daemen is also thanked for the interesting discussions.

The financial assistance provided by K. U. Leuven Research and Development, the Katholieke Universiteit Leuven, the Federal Office for Scientific, Technical and Cultural Affairs (Belgium), the Commission of the European Communities (Contract IST-1999-12324) and the Concerted Research Action (GOA) Mefisto-2000/06 of the Flemish Government are greatly appreciated.

In addition to my Ph.D. work I participated in several research projects, and I certainly learned a lot from the collaboration with people from universities as well as industry. Thanks go to the people I worked with in the scope of TIMESEC, FATIMA and NESSIE.

Thank you to all the current and past COSIC members for the nice working atmosphere. Péla deserves a big thank you for her support with the many and sometimes complex administrative matters; Elvira is thanked for doing the necessary background paperwork.

Last but of course not least, I want to thank my parents for their support and encouragement, and my fiancée Edith for her dedication and her belief which was a great motivation for me to complete this work.

Bart Van Rompay
June 2004

Abstract

This thesis is concerned with the analysis and design of cryptographic hash functions, MAC algorithms and block ciphers. Hash functions are versatile cryptographic building blocks, with applications such as the protection of the authenticity of information and digital signatures. The first part of this thesis gives an overview of existing hash functions and the different methods of designing these. Next, strategies for the cryptanalysis of hash functions are examined where we focus mainly on the popular algorithms based on the well-known MD4-design. The use of techniques similar to those introduced by H. Dobbertin in the mid-1990's (a combination of differential cryptanalysis and the solving of systems of non-linear equations), leads to the first known attack on the HAVAL algorithm. Besides that, a new method is developed for the cryptanalysis of the hash mode of PANAMA, a cryptographic module which can be used for both hashing and stream encryption.

The second part considers hash functions which are based on a secret key (these are also known as message authentication codes or MAC algorithms). We propose a new design, Two-Track-MAC, based on the two-trail construction that underlies the hash function RIPEMD-160. Our evaluation of this algorithm shows that it offers a high security level against all known strategies of attack. Another advantage is the efficiency, especially in applications where short messages are hashed or where the key is frequently changed. In those cases Two-Track-MAC performs better than other known constructions such as HMAC and MDx-MAC. We submitted our algorithm to the European NESSIE project, which had the goal of proposing a portfolio of secure cryptographic algorithms of the next generation. In February 2003 the NESSIE consortium announced that Two-Track-MAC is selected for the portfolio. Finally, the relation between hash functions and block ciphers is examined, and an attack is demonstrated on the block cipher ICE. This attack is a key-dependent variant on the technique of differential cryptanalysis.

Samenvatting

Dit proefschrift behandelt de analyse en het ontwerp van cryptografische hashfuncties, MAC-algoritmen en blokcijfers. Hashfuncties vormen veelzijdige bouwblokken in de cryptografie, die ondermeer gebruikt worden voor de bescherming van de authenticiteit van informatie en voor digitale handtekeningen. Het eerste deel van dit proefschrift geeft een overzicht van bestaande hashfuncties en de verschillende ontwerpmethoden. Vervolgens worden mogelijke strategieën voor de cryptanalyse van hashfuncties bestudeerd, voornamelijk gericht op de populaire algoritmen die gebaseerd zijn op het bekende MD4-ontwerp. Het gebruik van technieken gelijkaardig aan deze die halverwege de jaren negentig geïntroduceerd werden door H. Dobbertin (een combinatie van differentiële cryptanalyse en het oplossen van stelsels niet-lineaire vergelijkingen), leidt tot de eerste gekende aanval op het HAVAL-algoritme. Daarnaast wordt een nieuwe methode ontwikkeld voor de cryptanalyse van de hashmode van Panama, een cryptografische module die gebruikt kan worden voor zowel hashen als stroom-encryptie.

Het tweede deel behandelt hashfuncties die gebaseerd zijn op een geheime sleutel (MAC-algoritmen). We stellen een nieuw ontwerp voor, Two-Track-MAC, dat gebaseerd is op de tweelijnsconstructie van de hashfunctie RIPEMD-160. Onze evaluatie van dit algoritme toont aan dat het een hoge veiligheidsgraad bezit tegen alle gekende aanvalsstrategieën. Een ander voordeel is de efficiëntie, voornamelijk in toepassingen waar korte berichten gehasht worden of waar de geheime sleutel regelmatig veranderd moet worden. In deze gevallen presteert Two-Track-MAC beter dan andere bekende constructies zoals HMAC en MDx-MAC. We hebben ons algoritme ingediend als kandidaat voor het Europese NESSIE project, dat als doelstelling had om een portfolio samen te stellen met een nieuwe generatie van veilige cryptografische algoritmen. In februari 2003 maakte het NESSIE consortium bekend dat Two-Track-MAC geselecteerd is voor de portfolio. Tenslotte wordt het verband bestudeerd tussen hashfuncties en blokcijfers en een aanval voorgesteld voor het blokcijfer ICE. Deze aanval is een sleutelafhankelijke variant op de techniek van differentiële cryptanalyse.

Contents

List of Notations

List of Abbreviations

AES	Advanced Encryption Standard
ANSI	American National Standards Institute
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
FIPS	Federal Information Processing Standard
ICE	Information Concealment Engine
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
LFSR	Linear Feedback Shift Register
MAC	Message Authentication Code
MASH	Modular Arithmetic Secure Hash
MDx	Message Digest x
NESSIE	New European Schemes for Signatures, Integrity and Encryption
NIST	National Institute of Standards and Technology
RIPE	RACE Integrity Primitives Evaluation
RIPEMD	RIPE Message Digest
RSA	Rivest-Shamir-Adleman
SHA	Secure Hash Algorithm
TIMESEC	Digital Timestamping and the Evaluation of Security Primitives
VLIW	Very Long Instruction Word

List of Mathematical Symbols

f	compression function
g	output transformation
h	hash function or MAC algorithm
E	encryption algorithm
F	round function of a block cipher
X	input to a function
Y	output from a function
K	key for MAC or encryption algorithm
M	message
P	plaintext
C	ciphertext
X_i	input block
M_i	message block
H_i	chaining variable

IV	initial value
W_j	message word (32-bit or 64-bit)
f_r	Boolean function
n	output length (of a hash function or MAC algorithm)
b	block length (of a compression function or block cipher)
c	chaining variable length
k	key length
$\mathcal{D}$	domain of a function
$\mathcal{R}$	range of a function
$\mathcal{K}$	key space
$\mathcal{M}$	message space
$\|\mathcal{S}\|$	number of elements of the set $\mathcal{S}$
$\mathcal{O}$	order
$\in$	element of...
∞	infinity
exp	exponential function
max	maximum of...
mod	modulo (remainder of integer division)
$\lceil Z \rceil$	the smallest integer larger than or equal to Z
$\lfloor Z \rfloor$	the largest integer smaller than or equal to Z
$Z \dashv a$	truncation of Z to the a least significant bits
$Z^{\lll a}$	cyclic left-rotation by a bits of Z
$Z^{\ggg a}$	cyclic right-rotation by a bits of Z
$Z^{\hookrightarrow a}$	right-shift by a bits of Z
$\overline{Z}$	bitwise negation of Z
$Z_1 \\| Z_0$	concatenation of Z_1 and Z_0
$Z_1 Z_0$	bitwise AND of Z_1 and Z_0
$Z_1 \vee Z_0$	bitwise OR of Z_1 and Z_0
$Z_1 \oplus Z_0$	bitwise exclusive-OR of Z_1 and Z_0

Chapter 1

Introduction

1.1 Confidentiality and Authenticity

In modern society information has become a valuable commodity. It is often necessary to protect its **confidentiality**, which means that it should be infeasible for unauthorised people to learn the content. On the other hand it can be equally important to protect the **authenticity** of information. This has two aspects: it should be possible to check who the author is of a certain piece of information (*data origin authentication*) and that it has not been modified by anyone else (*data integrity*).

In former days this protection of information was achieved by a combination of physical security and trust: paper documents can be sealed in an envelope (which allows detection of disclosure) or in a locked safe (which should prevent disclosure). The protection of authenticity depends on the difficulty of forging documents and/or signatures. In the electronic age letters, contracts and other documents are replaced by sequences of binary digits but the demands for confidentiality and authenticity remain the same. The risks are often greater: unprotected data residing on open and untrusted networks (e.g., the Internet) can be easily accessed, copied or modified.

The same can be said for the protection of communications. In face-to-face communication between people it is relatively easy to create circumstances in which eavesdropping is infeasible. It also has the inherent aspect of authenticity because one can visually verify who the communication partner is. Nowadays many people use telecommunication networks as an efficient, cheap and reliable means of communication. It is however easy to tap messages transported over easily accessable channels, and in some cases the messages can even be modified along the way.

Cryptographic techniques have been used for many centuries to protect military and diplomatic secrets (a comprehensive account of this history is given by D. Kahn in [68]). Most of these techniques are *encryption schemes* that convert a message into a cryptogram by an invertible operation (encryption) depending on a small piece of secret information (the *key*). The cryptogram is unintelligible for an unauthorised person who intercepts it, but can be reconverted (decrypted) into the message by an authorised receiver who has been given knowledge of the key.

Throughout history cryptology (the study of cryptographic techniques) has concentrated mainly on the problem of confidentiality. It was in fact believed that by protecting the secrecy of information one would also automatically protect its authenticity. The reasoning is as follows: if decryption of a cryptogram results in a meaningful message it must have been constructed by someone who knows the secret key. However it is not always necessary to know the key (or break the encryption scheme) in order to falsify messages: the protection of integrity strongly depends on the encryption scheme and on the mode in which it is used. A famous example is the Vernam cipher or modulo 2 one-time pad [129]: this cipher offers unconditional secrecy but an attacker only needs to change a binary digit (bit) of the cryptogram in order to change the corresponding bit of the message.

Whereas in the past cryptology was more of an art practised by few, the publication of the Data Encryption Standard [54] and the invention of public-key cryptography [38] in the 1970's caused it to develop into a scientific research area. The introduction of new concepts and definitions resulted in a clear separation of the problems of confidentiality and authenticity and established the development of cryptographic schemes for the protection of authenticity as an important research topic. These schemes now have many commercial applications, for example to provide security for electronic commerce and wireless communication systems.

1.2 Cryptography and Cryptanalysis

Cryptology comprises two complementary fields of research: cryptography and cryptanalysis. A cryptographer is concerned with the development of new schemes or algorithms, providing security services such as confidentiality and authenticity. A cryptanalyst on the other hand is concerned with the development of attack methodologies that break a cryptographic algorithm, allowing, for example, unauthorised people access to secret information or the ability to forge documents. There is a strong relation between these two fields. A cryptographer who designs a new algorithm needs to evaluate the security of his design against all methods of attack available to the cryptanalyst. Furthermore, in order to attain a sufficiently high level of confidence from users, new cryptographic algorithms should be evaluated not only by their designer but also by independent cryptanalysts. They can only be recommended for use in critical applications after having been studied for a sufficient amount of time, by a reasonably large group of experienced cryptanalysts who cannot find any weaknesses.

1.2.1 The NESSIE project

The relation between the fields of cryptography and cryptanalysis can be illustrated by means of the NESSIE project [135]. NESSIE, which stands for "New European Schemes for Signatures, Integrity, and Encryption", was a project within the Information Society Technologies (IST) Programme of the European Commission. The main goal of the project was the establishment of a portfolio of strong cryptographic algorithms, obtained after an open call and evaluated using a transparent and open process. The portfolio would comprise several categories of algorithms useful for encryption, authentication and digital signatures.

Forty-two cryptographic algorithms (belonging to the different categories) were submitted in response to the call issued by NESSIE. Researchers inside and outside the project then tried to attack these algorithms during a period of more than two years. After a first evaluation phase a subset of twenty-four algorithms was selected for further study (allowing to focus the attention on the most promising candidates), and at the end of the project twelve of these were chosen for the portfolio (together with five existing standard algorithms). The submitters of the algorithms were allowed to propose minor changes to their designs at the end of the first phase of the evaluation, which gave them the opportunity to improve their designs and to address any minor weaknesses that had been found. The algorithms selected by NESSIE are not standards but the amount of evaluation they received and the fact that no weaknesses were found, means that they carry a reasonable amount of confidence. It is expected that at least several of them will be adopted by standardisation bodies in the near future.

1.3 This Thesis

The main focus of this thesis is on the analysis and design of one particular category of algorithms: cryptographic hash functions. These are algorithms that take inputs of arbitrary length (e.g., a digital document or message) and produce as output a short string of bits. Their most important use is for the protection of data authenticity, but they are a versatile building block, used also in conjunction with digital signature schemes and for many other applications such as password protection and pseudo-random string generation.

Cryptographic hash functions come in two types: those that depend on a secret key for their computation and those that do not. The first type are often called message authentication codes or MAC algorithms. From the cryptanalytic point of view we spend most of our attention on a particular class of unkeyed hash functions: custom-designed algorithms based on the ideas first introduced by R. Rivest for the MD4 hash function [114]. The motivation for this is the popularity of these hash functions which is due to their efficiency on common desktop computers based on 32-bit architectures. We also present a new design for a message authentication code derived from one of the hash functions based on MD4. Our algorithm, called Two-Track-MAC, was submitted to the NESSIE project and it has been selected for the NESSIE portfolio. Finally we study block ciphers (a category of cryptographic algorithms used for encryption), and we discuss the relation between block ciphers and hash functions.

It may be noted that cryptographic hash functions have received much less attention from the cryptologic community than encryption schemes. This is clear from the NESSIE project where seventeen block ciphers and six stream ciphers were submitted as candidates (both are categories of encryption schemes), but only one unkeyed and two keyed hash functions. Another example is the open competition used by the National Institute of Standards and Technology (NIST) in the United States to decide on the block cipher to be used as Advanced Encryption Standard [52]. This competition had fifteen candidates out of which the Rijndael block cipher [31] was finally chosen. On the other hand, for its hash function standard [51] NIST simply chose the SHA hash functions, designed by the NSA without disclosure of their design strategy or any supporting cryptanalytic results.

1.4 Outline and Main Contributions

The outline of this thesis is the following:

– Chapter 1 motivates our research on cryptography in general, and on cryptographic hash functions and MAC algorithms in particular.

– Chapter 2 explains the basic concepts of cryptographic hash functions and MAC algorithms, and discusses some of the most important applications. We give an overview of the use of hash functions in schemes for digital timestamping. This application was studied in the framework of the Belgian TIMESEC project [134] and the results have been published in [127].

– Chapter 3 gives an overview of the different design methods for unkeyed hash functions and lists the most important algorithms. Several of these are discussed in more detail in Chapters 4 and 5. Standardisation efforts in this field are also briefly discussed.

- Chapter 4 gives a detailed overview of the design ideas used for the popular hash functions of the MDx-class (algorithms based on MD4). It also discusses the cryptanalytic results published in the literature for these hash functions. The main contribution of this chapter is the new attack that has been developed for the HAVAL hash function. This is a joint work with A. Biryukov and has been published in [123]. Another contribution is the variant of the attack on MD4 which shows alternatives to the approach used by H. Dobbertin for his analysis [42] of this hash function. An overview related to this chapter was published in [126].

- Chapter 5 describes PANAMA, a cryptographic module that can be used for both hashing and stream encryption. Our contribution is the development of a new attack method for the hash mode of PANAMA. This is a joint work with V. Rijmen and has been published in [112].

- Chapter 6 discusses the design of MAC algorithms (hash functions that are based on a secret key) and lists the most important algorithms. The main contribution of this chapter is the proposal of a new design, Two-Track-MAC, a joint work with B. den Boer that was published in [37]. The algorithm was submitted to NESSIE [124] and has been selected for the portfolio of the project.

- Chapter 7 discusses the relation between hash functions and block ciphers. Our contribution is the cryptanalysis of the block cipher ICE, a joint work with L. R. Knudsen and V. Rijmen that was published in [125].

- Chapter 8 concludes, and suggests some topics for further research.

- The Appendices give detailed descriptions for some of the algorithms (Two-Track-MAC, MD4, HAVAL) discussed in this thesis, as well as supporting cryptanalytic results for PANAMA.

Chapter 2

Basic Concepts

2.1 Introduction

This chapter explains the basic concepts of cryptographic hash functions. We describe different types of hash functions, provide definitions and discuss the security provided by these algorithms. The use of hash functions in schemes that protect the authenticity of information is explained. Some other applications are described, in particular the use of hash functions for optimising digital signature schemes, and their use in schemes for digital timestamping. An overview of timestamping systems and the role of hash functions therein has been published in [127]. For a more detailed discussion on hash functions and information authentication, we refer to the treatment of B. Preneel in [98, 99].

2.2 Cryptographic Hash Functions and MACs

Hash functions are functions that compress an input of arbitrary length into a fixed number of output bits, the *hash result.* If such a function satisfies additional requirements it can be used for cryptographic applications, for example to protect the authenticity of messages sent over an insecure channel. The basic idea is that the hash result provides a unique imprint of a message, and that the protection of a short imprint is easier than the protection of the message itself (see Sect. 2.4.1). An illustration of the use of a hash function is shown in Fig. 2.1.

Related to hash functions are *message authentication codes* (MACs). These are also functions that compress an input of arbitrary length into a fixed number of output bits, but the computation depends on a secondary input of fixed length, the *key.* Therefore MACs are also referred to as *keyed hash functions.* In practical applications the key on which the computation of a MAC depends is

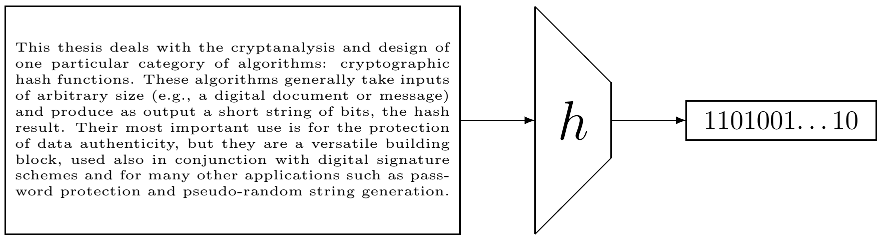
This thesis deals with the cryptanalysis and design of one particular category of algorithms: cryptographic hash functions. These algorithms generally take inputs of arbitrary size (e.g., a digital document or message) and produce as output a short string of bits, the hash result. Their most important use is for the protection of data authenticity, but they are a versatile building block, used also in conjunction with digital signature schemes and for many other applications such as password protection and pseudo-random string generation.
h
1101001. . . 10

Figure 2.1: Compression with a cryptographic hash function. The message input can have any length, but the number of output bits is fixed.

kept secret between two communicating parties, as explained in Sect. 2.4.3. In the remainder of this section we give informal definitions of (unkeyed) hash functions and message authentication codes, and of their cryptographic properties. We also refer to a more formalised theoretical treatment of these algorithms.

2.2.1 Cryptographic hash functions

For an (unkeyed) hash function, the requirement that the hash result serves as a unique imprint of a message input implies that it should be infeasible to find colliding pairs of messages (i.e., messages that hash to the same result). In some applications however it may be sufficient that for any given hash result it is infeasible to find a corresponding message, or that, given a message, it is infeasible to find another message hashing to the same result. Depending on these requirements Preneel [96] provides the following informal definitions for two different types of hash functions. A **one-way hash function** is a function h that satisfies the following conditions:

1. The input X can be of arbitrary length and the result $h(X)$ has a fixed length of n bits.

2. Given h and an input X, the computation of $h(X)$ must be 'easy'.

3. The function must be one-way in the sense that given a Y in the image of h, it is 'hard' to find a message X such that $h(X) = Y$ (*preimage-resistance*), and given X and $h(X)$ it is 'hard' to find a message $X' \neq X$ such that $h(X') = h(X)$ (*second preimage-resistance*).

A **collision-resistant hash function** is a function h that satisfies the following conditions:

1. The input X can be of arbitrary length and the result $h(X)$ has a fixed length of n bits.
2. Given h and an input X, the computation of $h(X)$ must be 'easy'.
3. The function must be one-way, i.e., preimage-resistant and second preimage-resistant.
4. The function must be collision-resistant: this means that it is 'hard' to find two distinct messages that hash to the same result (i.e., find X and X' ($X \neq X'$) such that $h(X) = h(X')$).

2.2.2 Message authentication codes

For a message authentication code, the computation (and therefore the output or *MAC result*) depends on a secondary input, the secret key. The main idea is that an adversary without knowledge of this key should be unable to 'forge' the MAC result for any new message, even when many previous messages and their corresponding MAC results are known. The following informal definition was given by Preneel [96]. A **message authentication code** or **MAC** is a function h that satisfies the following conditions:

1. The input X can be of arbitrary length and the result $h(K, X)$ has a fixed length of n bits. The function has as secondary input the key K, with a fixed length of k bits.
2. Given h, K and an input X, the computation of $h(K, X)$ must be 'easy'.
3. Given a message X (but with unknown K), it must be 'hard' to determine $h(K, X)$. Even when a large set of pairs $\{X_i, h(K, X_i)\}$ is known, it is 'hard' to determine the key K or to compute $h(K, X')$ for any new message $X' \neq X_i$ $(\forall i)$.

2.2.3 Formal definitions

Hash functions and message authentication codes can be formally defined in the following manner.

Definition 2.1 *A* **hash function** *is a function $h : \mathcal{D} \rightarrow \mathcal{R}$ where the domain $\mathcal{D} = \{0,1\}^*$, and the range $\mathcal{R} = \{0,1\}^n$ for some $n \geq 1$.*

Definition 2.2 *A* **MAC** *is a function $h : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{R}$ where the key space $\mathcal{K} = \{0,1\}^k$, the message space $\mathcal{M} = \{0,1\}^*$, and the range $\mathcal{R} = \{0,1\}^n$ for some $k, n \geq 1$.*

Formal definitions are also needed for the security properties of hash functions and MACs. Such definitions provide an upper bound for the probability of success of an algorithm executed by an adversary to find a preimage, second preimage or collision for a hash function, or to find a forgery for a MAC. The probability of success, or advantage, for an adversary A is denoted $\mathrm{Adv}(A)$. If one maximises this over all adversaries whose resources are bounded by R one obtains $\mathrm{Adv}(R)$, and this is a good quantitative measure of the strength of an algorithm (with respect to a certain property).

S. Goldwasser and M. Bellare provide a theoretical treatment of MACs based on this approach in [57] (see also the work of Bellare *et al.* in [12]). Here it is assumed that the actions of the adversary are divided into two phases. The first is a *learning* phase where the adversary is able to make a number of queries to an oracle. These queries consist of a message, and the oracle returns the corresponding MAC result. The key K for the MAC computations is chosen a priori and at random, and is unknown to the adversary. Next comes the *forgery* phase where the adversary outputs a pair consisting of a message M, different from all the messages contained in his queries to the oracle, and a result Y. If Y is the correct MAC result for M under the key K the adversary has been successful. The strength of the MAC algorithm against forgery attacks is defined as the maximum probability of success (advantage) for all adversaries whose resources are bounded by (t, q, μ). Here t is the running time of the adversary, q is the number of oracle queries he is allowed to make, and μ is the total length (in bits) of all oracle queries plus the length of the message in the output forgery.

A similar theoretical treatment of the different notions of hash function security is provided by P. Rogaway and T. Shrimpton in [117]. For a preimage or second preimage attack it is assumed that the adversary first receives a *challenge*. This is the point in the range for which a preimage must be found, or the point in the domain for which a second preimage must be found. Next, the adversary outputs a guess for the (second) preimage. The level of (second) preimage-resistance of a hash function is defined as the maximum probability of success (advantage) for all adversaries whose resources are bounded by t (the running time). However there are some complications to this treatment, and different types of preimage and second preimage-resistance can be defined. Firstly, there are different ways to determine the challenge. In particular, one can define the advantage of the adversary for a challenge chosen at random, or alternatively one can define the advantage for a specific value of the challenge and maximise over all possible values. Furthermore, for reasons outlined below, one usually considers the more generic concept of a *hash function family*, that is a finite set of hash functions with common domain and range. The advantage of the adversary then depends on the manner in which one chooses a particular hash function from the hash function family: the hash function can be a fixed function or a random element from the set of functions.

The theoretical treatment for collision attacks on hash functions is somewhat different. First of all, there is no challenge to the adversary. Secondly, because the adversary has no input to his attack and because there is no secret information involved in the computation of a hash function, a formal definition of collision-resistance only makes sense for a random element of a hash function family, and not for a fixed hash function. The reason is that, in theory, for a fixed hash function there always *exists* an adversarial algorithm that immediately outputs a pair of messages for some fixed collision (although in practice it may be very difficult to actually construct such a collision-finding algorithm). To summarise the results of [117], Rogaway and Shrimpton formally define seven different notions of hash function security: three types of preimage-resistance, three types of second preimage-resistance, and one type of collision-resistance. They also discuss relations between these different notions of security.

2.3 A Practical Approach to Security

The theoretical framework for the security of hash functions and MACs, discussed in the previous section, has its limitations. In practice, it is very difficult to prove an upper bound for the probability of success of an algorithm executed by an adversary to find a preimage, second preimage or collision for a hash function, or to produce a forgery for a MAC. While some results are known in the area of *provable security*, for most real-world applications the design of efficient algorithms can only be obtained by a practical approach. Such an approach is based on methods that evaluate the security of an algorithm by estimating the computing power needed by an adversary to break it. This leads to a definition of security that is based on heuristic arguments: the estimates for the required computing power are based on the results of known attacks.

When assessing the security of an algorithm, different types of attack need to be considered. The first type are generic attacks. These can be applied to any algorithm and their complexity depends only on generic parameters: the size of the output space (that is, the number of different hash results or MAC results that can be generated by the function), and for MACs also the size of the key space (that is, the number of possible key values). The second type are short-cut attacks. In this case the attacker analyses the internal structure of an algorithm and tries to find a weakness that allows him to develop an attack which is more efficient than the best generic attack. A cryptographic algorithm is usually considered broken if there exists a short-cut attack that is faster than the best generic attack.

Both generic and short-cut attacks may have different goals. For hash functions, the goal of the attacker may be to find a preimage or second preimage, or to generate a collision. It may be noted here that the property of collision-resistance implies second preimage-resistance, because an attacker who is capable of generating second preimages, would also be capable of generating collisions. For most practical hash functions the property of collision-resistance also implies preimage-resistance (in [117] Rogaway and Shrimpton show that this depends on how much the hash function compresses). It may also be noted that when a hash function is used for purposes other than which it was originally designed for, additional properties may be required. For example, when a hash function is used for the generation of pseudo-random strings (starting from a secret *seed*), one needs to consider attacks that analyse the pseudo-randomness of the generated output.

For message authentication codes the adversary will try to produce a forgery. In the case of an *existential forgery* the adversary determines the MAC result for at least one message, but has no control over the content of this message so it may be random or nonsensical. In the case of a *selective forgery* on the other hand, the adversary determines the MAC result for a particular message chosen a priori by him. In practice, it may be required that a forgery is *verifiable*, which means that the adversary knows that his forgery is correct with probability close to 1. An adversary may also perform a *key recovery* attack on a MAC algorithm. If successful, such an attack determines the value of the secret key, and this is more powerful than a forgery, since it allows for arbitrary selective forgeries.

The computation of a message authentication code involves secret data. As mentioned before, it is assumed that for his attack the adversary has knowledge of a set of pairs, consisting of messages and their corresponding MAC results. A number of different attack scenarios can be distinguished, based on the manner in which this information becomes available to the adversary. In a *known text attack* the adversary has access to a number of messages and their corresponding MAC results. In a *chosen text attack* the adversary is able to choose a set of messages and subsequently obtains a list of MAC results corresponding to these messages. Finally, in an *adaptive chosen text attack* the adversary can make the choice of a message depend on the outcome of previous queries. Note that an adaptive chosen text attack may not always be feasible in practice, but for a cryptographer who designs a MAC algorithm it is best to be conservative and require that the algorithm resists against the strongest possible attack, that is an existential forgery under an adaptive chosen text scenario.

2.3.1 Generic attacks on cryptographic hash functions

Below we discuss two generic attacks that can be applied to any hash function. The first attack can be used to find a preimage or second preimage, the second attack for generating collisions. Note that short-cut attacks on hash functions are examined in Chapters 3, 4 and 5.

Random (second) preimage attack

In this attack the adversary simply selects a random input and hopes that a given hash result occurs. If the hash function has a 'random' behaviour, his probability of success equals $1/|\mathcal{R}|$, where $|\mathcal{R}|$ denotes the number of elements in the range of the hash function (i.e., the size of the output space). The success probability can be increased by selecting additional inputs and checking the hash results. It is expected that a (second) preimage will be found after $r = \mathcal{O}(|\mathcal{R}|)$ operations. For example, for $r = 0.7 \cdot |\mathcal{R}|$, the success probability is about 50%, and for $r = |\mathcal{R}|$ it is about 63%.

In practice the attack can be carried out in parallel, by distributing the computational effort over a (possibly large) number of machines. Depending on the application in which the hash function is used, it may also be possible for a cryptanalyst to target a number of different hash values simultaneously, trying to find a preimage for one of them.

Birthday attack

This attack is based on the idea that for a group of 23 people the probability that at least two of them have the same birthday, is larger than 50% [47]. Because most people intuitively think that the group would need to be much larger, this is called the *birthday paradox*. It can be explained as follows. A year has 365 days. The probability that in a group of r people all of them have a different birthday, can be calculated as follows:[1]

$$q = \frac{365 \cdot 364 \cdot \ldots \cdot (365 - r + 1)}{365^r} = \prod_{i=0}^{r-1} (1 - \frac{i}{365}) \, .$$

Hence, the probability that at least two people in the group have the same birthday is $p = (1 - q)$. For $r = 23$ the probability $p \approx 0.507$. Note that the probability of a common birthday increases rapidly as the group becomes larger, e.g., for $r = 46$ we get a probability $p \approx 0.948$.

In a birthday attack on a hash function the adversary selects a set of r random inputs and hopes that at least two of these inputs have the same hash result (this means that a collision is found). The probability can be calculated in the same manner as above where the value 365 is replaced by $|\mathcal{R}|$ (it is assumed that the hash function has 'random' behaviour). If $r = \mathcal{O}(\sqrt{|\mathcal{R}|})$ and $|\mathcal{R}| \to \infty$, then the

[1]The existence of leap years is ignored and it is assumed that birthdays are randomly distributed over the year. As this is not the case the real probability is even higher.

probability of a collision can be approximated as follows:

$$p \approx 1 - \exp(-\frac{r^2}{2 \cdot |\mathcal{R}|})\,.$$

P. Flajolet and A. Odlyzko [55] have shown that the expected number of inputs needed for a collision is:

$$r_{col} = \sqrt{\frac{\pi \cdot |\mathcal{R}|}{2}}\,.$$

There is also a variant of the birthday attack where the adversary selects two distinct sets and tries to find a collision between their elements. Let the first set contain r_1 inputs, and the second set r_2 inputs. If $r = r_1 = r_2$, $r = \mathcal{O}(\sqrt{|\mathcal{R}|})$ and $|\mathcal{R}| \to \infty$, then the probability of a collision between the sets can be approximated by:

$$p \approx 1 - \exp(-\frac{r^2}{|\mathcal{R}|})\,.$$

For $r = r_1 = r_2 = \sqrt{|\mathcal{R}|}$ this probability is about $1 - \exp(-1)$ or 63%.

Based on this variant of the birthday attack, G. Yuval [130] proposed the following practical attack scenario.[2] The adversary generates r_1 variations on a genuine message and r_2 variations on a fraudulent message. The probability that there is a genuine message and a fraudulent message with the same hash result is about 63% when $r = r_1 = r_2 = \sqrt{|\mathcal{R}|}$. In order to find the collision, the adversary stores the hash results corresponding to one set of messages in a table. After sorting of the data in the table (which can be done in $\mathcal{O}(r \log r)$ time), he computes the hash results for the messages of the other set until a match is found in the table.

The main problem for a practical implementation of the attack by Yuval is the memory requirement: space is needed for the storage of about $\mathcal{O}(r)$ messages. J.-J. Quisquater and J.-P. Delescaille [107] demonstrated an alternative method which needs only very little memory. The main idea is to generate a random walk through the hash result space, and to use an efficient cycle-finding technique to find a collision. The storage of a number of *distinguished points* limits the number of steps that are executed after the repetition starts. In [122] P. van Oorschot and M. Wiener show that this method can be fully parallelised.

2.3.2 Generic attacks on message authentication codes

Below we discuss two generic attacks that can be applied to any message authentication code. Other attacks on MACs are examined in Chapter 6.

[2]This attack is directed towards a digital signature scheme, see Sect. 2.4.4.

Guessing of the MAC

In order to produce a selective forgery an adversary can choose a particular message and subsequently guess the corresponding MAC result. There are two ways of doing this: the adversary either guesses the MAC result directly, or he guesses the secret key and then computes the MAC result. If the output length (that is, the length of the MAC results) is equal to n bits, and the length of the secret key is equal to k bits, the probability of success is $p = \max(2^{-n}, 2^{-k})$. Note however that the attack is non-verifiable: the adversary does not know a priori whether his guess was correct. For each trial he submits a guess for the MAC result to the system which performs the verification. This implies that the attack can only be executed on-line. The number of possible trials for an attacker is strongly application-dependent: typically only a limited number of errors is allowed (the errors correspond to wrong guesses for the MAC result).

Exhaustive Key Search

This is a key recovery attack where the adversary tries one-by-one possible values for the secret key until the correct key value has been chosen. The expected number of trials is 2^{k-1} (a search through half of the key space), and the attack can be verified when $\lceil k/n \rceil$ known text-MAC pairs are available: for each guess of the secret key the adversary checks the correspondence between text and MAC result for each of the pairs; when $\lceil k/n \rceil$ pairs are available for checking it is expected that only the correct guess for the secret key satisfies all checks. In contrast to the previous attack, this attack is carried out off-line and it yields a complete break of the MAC algorithm. Note that the key search can also be parallelised, by distributing the computational effort over a number of machines.

2.4 Applications of Hash Functions and MACs

2.4.1 Message authentication based on a hash function

We consider the problem of the protection of the authenticity of information. This problem has two aspects: data integrity and data origin authentication. The following definitions were given by A. Menezes *et al.* [84]:

Definition 2.3 Data integrity *is the property whereby data has not been altered in an unauthorised manner since the time it was created, transmitted, or stored by an authorised source.*

Definition 2.4 Data origin authentication *is a type of authentication whereby a party is corroborated as the (original) source of specified data created at some (typically unspecified) time in the past.*

By definition, data origin authentication includes data integrity (information which has been modified effectively has a new source).

Let us now consider a situation where an originator wants to send a message to a recipient over an insecure channel. For this setting the problem of authenticity is also called *message authentication.* It means that it should be possible to determine the source of the message and to assure that the message is not modified in any way during the transmission. As mentioned in Sect. 2.2 the main idea of the use of a cryptographic hash function is to reduce the problem of protecting a (possibly long) message to the problem of protecting a short imprint of the message (the hash result).

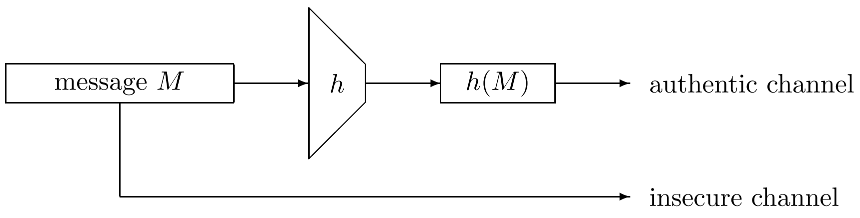

Figure 2.2: Message authentication using a hash function.

Fig. 2.2 shows the basic mechanism by which this can be achieved. It is assumed that an *authentic channel* is available for the transmission of short message imprints. Information sent over this authentic channel cannot be changed (eavesdropping may be possible so secrecy is not guaranteed). Moreover, the recipient knows from whom the information on this channel originates. An example may be a telephone link where the authenticity is offered by voice recognition. The originator of the message M uses a cryptographic hash function h to compress his message into a short bit string $h(M)$ (the hash result), and this value is transmitted to the recipient via the authentic channel (even for a telephone link this should be no problem, a hash result can typically be expressed by a few dozens of digits). The message itself is sent over the insecure channel. An active adversary may be able to modify the message M before it reaches the recipient. However, the recipient independently hashes the message he receives (using the same cryptographic hash function) and accepts the message only if the result agrees with the value $h(M)$. In order to cheat the system the adversary must create a modified message M', with the property that $h(M') = h(M)$. For this he must find a second preimage, which should be infeasible for a one-way or collision-resistant hash function. The system works by transferring the authenticity of the message to the authenticity of the hash result.

2.4.2 Authentication combined with encryption

We mentioned in Sect. 1.1 that the use of encryption is not sufficient to protect the integrity of data. In the case of additive stream ciphers (which can be seen as practical approximations of the Vernam cipher), an attacker can change any bit of the plaintext by modifying the corresponding bit of the ciphertext. For block ciphers the modification of a bit of the ciphertext affects a larger part of the plaintext in an unknown way. This depends on the error propagation characteristics of the mode in which the block cipher is used (see Chapter 7, Sect. 7.2.3), and makes it more difficult to modify ciphertexts with their decryption remaining meaningful.

However, the protection of integrity also depends on the inherent redundancy of the information. If the plaintext corresponding to a given ciphertext contains no redundancy (e.g., a random key), all decryptions of the (possibly modified) ciphertext are meaningful. Therefore some form of redundancy is always needed for the protection of integrity. One can use a cryptographic hash function to add sufficient redundancy to information before encrypting it. Fig. 2.3 shows a simple mechanism that allows for message authentication combined with encryption.

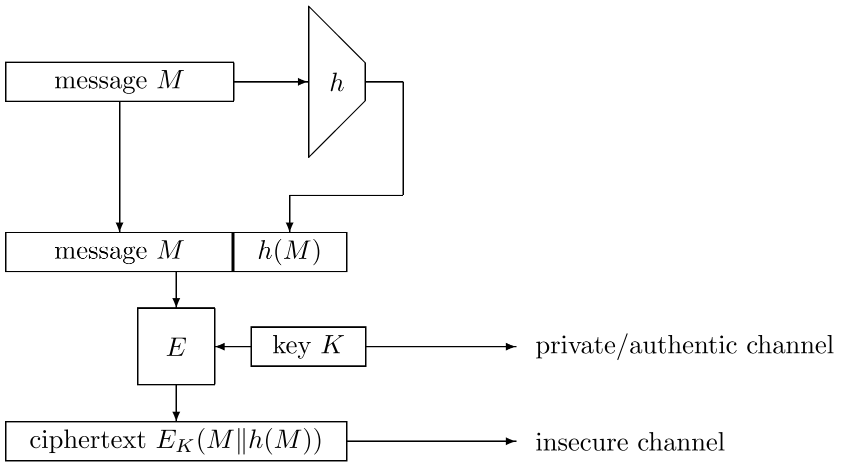

Figure 2.3: Message authentication combined with encryption.

The originator of a message M uses a cryptographic hash function h to compute the hash result $h(M)$. He appends this to the original message resulting in a string $M\|h(M)$. Next, he uses an encryption algorithm E based on a secret key K to encrypt this information, and he sends the ciphertext $E_K(M\|h(M))$ over the insecure channel. The recipient, who also knows the secret key, decrypts the ciphertext. He separates the recovered message from the recovered hash result and checks the correspondence using the hash function h (in other words, the recipient checks the redundancy of the decrypted ciphertext). For an adversary who does not know the secret key used in the encryption, it should be infeasible to change the ciphertext sent over the channel without disrupting the correspondence between the recovered message and the recovered hash result.

The mechanism of Fig. 2.3 works by transferring the secrecy and authenticity of the message to the secrecy and authenticity of the key that is used. A private and authentic channel is needed for the exchange of the secret key between the sender and the recipient. This channel may have a small capacity because the same key can be used for the encryption of many messages. A disadvantage of this mechanism is that the protection of authenticity depends on the protection of secrecy. If the encryption algorithm is weak, the authenticity is compromised as well because the adversary would be able to decrypt the ciphertext, change the information into $M' \| h(M')$ and re-encrypt it.

2.4.3 Message authentication based on a MAC

An alternative solution to the problem of message authentication is the use of a MAC algorithm. The originator of a message M uses a MAC algorithm h based on a secret key K to compress the message into a short bit string $h(K, M)$. He appends this to the original message resulting in a string $M \| h(K, M)$, which he sends over the insecure channel as illustrated in Fig. 2.4 below. An active adversary may be able to change the data on the insecure channel. The recipient of the data separates the recovered message from the recovered MAC result, and obtains two strings M' and Y'. He then uses the MAC algorithm based on the same secret key to check the correspondence, that is he verifies whether $Y' = h(K, M')$. If the equation is satisfied he accepts the data as authentic, that is, he assumes $M' = M$ (the data sent by the originator).

For this mechanism a separate channel is needed for distribution of the secret key between sender and receiver. This channel must offer both privacy and authenticity, but it may have a small capacity because the same key can be used for the authentication of many messages. For an adversary who does not know the secret key K it should be infeasible to change the message M into a modified version $M' \neq M$, because he is unable to compute the corresponding MAC result $h(K, M')$. The difference with the mechanism based on a hash function (Sect. 2.4.1) is that the message imprint can now be sent over the insecure channel, because the algorithm that is used for the compression of messages into message imprints, depends on secret data.

MAC algorithms can also be used for applications that require authentication combined with encryption. A similar mechanism can be used as the one described

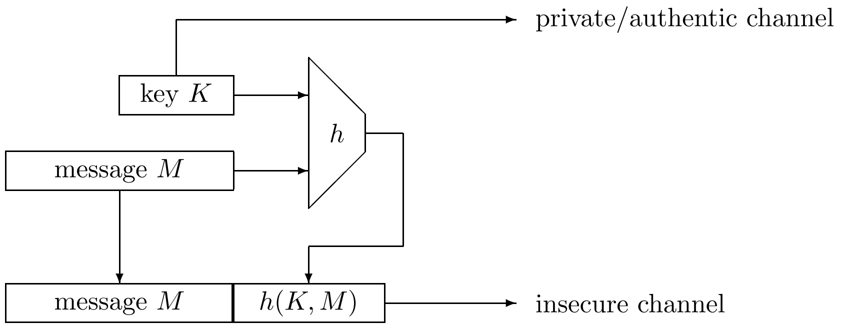

Figure 2.4: Message authentication using a MAC.

in Sect. 2.4.2 (Fig. 2.3), but where the hash function is replaced by a MAC algorithm based on a secret key. This has the advantage that the integrity is protected even when the security of the encryption is compromised. A drawback is that the application must handle two separate keys, one for the MAC algorithm and one for the encryption scheme. One should not use the same key for both purposes as weaknesses may arise due to the interaction. Moreover, the procedures for key management are normally different for authentication and encryption keys. A formal analysis of the security of authenticated encryption schemes, for three different composition methods, is given by Bellare and Namprempre in [13].

2.4.4 Optimisation of digital signature schemes

Digital signature schemes are a different type of cryptographic primitive. They are used for the protection of authenticity but, in addition, they also offer the service of *non-repudiation*. This implies that it is impossible for an originator who sends an authenticated message, to dispute at a later time having sent this message. Digital signatures are based on public-key cryptography. All users of a public-key cryptosystem have a public key, known to everyone else and linked by some mechanism to the correct identity. Every user also has a private key which should not be disclosed to anyone else (this may be assured when the private key is stored in tamper-resistant hardware).

Fig. 2.5 gives an example of how a digital signature scheme (e.g., one based on RSA [116]) may be used in combination with a cryptographic hash function. The user A has a key pair (S_A, P_A) where S_A denotes his private (secret) key, and P_A denotes his public key. He first compresses his message M with the hash function h. The hash result $h(M)$ is sent as input to the signature algorithm. This algorithm depends on the private key S_A and computes a value $sign_{S_A}(h(M))$ which we denote in shorthand as $s(M)$. The user then concatenates the signature $s(M)$ to his message and sends the information $M\|s(M)$ over the insecure channel.

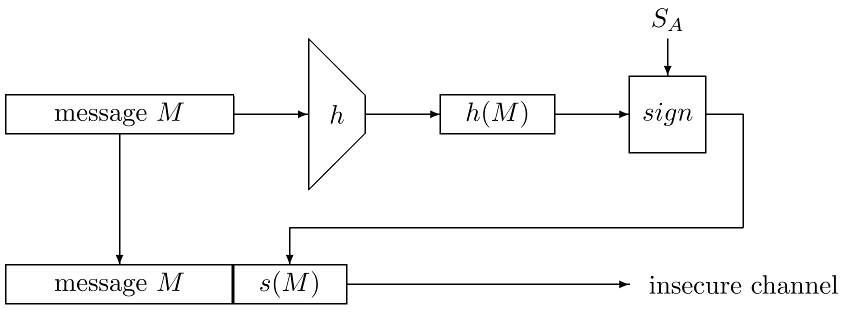

Figure 2.5: Digital signature scheme using a hash function.

Fig. 2.6 shows the procedure followed by the recipient of the signed message. He uses the signature $s(M)$ as input to the verification algorithm. This algorithm depends on the public key P_A, and computes the value $ver_{P_A}(s(M))$. The signature and verification algorithms are designed in such a way that $ver_P(sign_S(X)) = X$ when a correct key pair (S, P) is used.[3] Therefore the output of the verification algorithm should be $h(M)$ if no interference occured on the channel. The recipient also computes an independent hash of the message he receives. If the outcome of this is the same as the output of the verification algorithm, he accepts the message and signature as genuine.

The use of a cryptographic hash function in a digital signature scheme as shown above, has the advantage that the signature and verification algorithms have only short data strings to work with (the size is independent of the length of the message). This is important because public-key cryptography is much slower (several orders of magnitude) than conventional symmetric key algorithms or hash functions. Furthermore, the use of a hash function prevents attacks that exploit the algebraic structure of the message space in a signature scheme (see for example [46]).

For an adversary it should be impossible to modify the information on the channel with the signature of user A remaining valid. If he replaces the message by M', he cannot change the signature into $s(M')$ because he does not have the private key S_A. He can only try to find a modified message with the property $h(M') = h(M)$ (in this case the signature remains the same), but this involves finding a second preimage for the hash function.

[3]Note that other digital signature schemes use a slightly different method for verification.

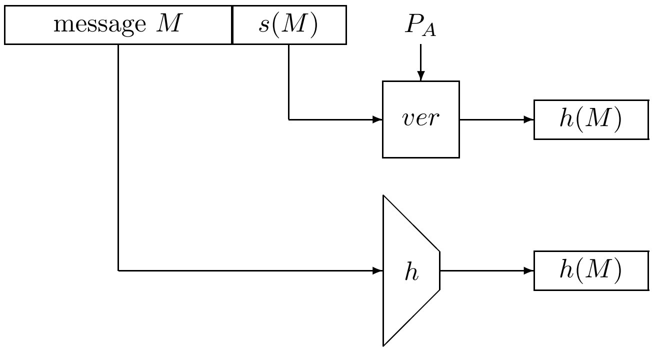

Figure 2.6: Verification of a digital signature.

The main difference between a digital signature scheme and the authentication mechanisms described in Sect. 2.4.1 and Sect. 2.4.3, is that it is possible for a third party to distinguish between the sender and recipient of an authenticated (signed) message. In the scheme of Sect. 2.4.1 the two parties usually have the same access to the authentic channel, and in the scheme of Sect. 2.4.3 the two parties share the same secret key. For a digital signature scheme however, each user has his own private key to authenticate his messages. This allows the service of non-repudiation, but it is important to notice that the hash function used must be collision-resistant (not only preimage and second preimage-resistant). Otherwise a dishonest user of the system might be able to construct a genuine message M and a fraudulent message M' with the same hash result, i.e., $h(M) = h(M')$. At a later time he would be able to dispute having generated a signature for M, and claim that he generated a signature for M' instead.

2.4.5 Applications as one-way function

One-way functions are similar to one-way hash functions, except that the length of the input is fixed rather than arbitrary. They can be constructed from a cryptographic hash function but also from other cryptographic algorithms such as block ciphers. Below we briefly describe some interesting applications of one-way functions.

Identification with passwords

For identification systems based on passwords a one-way function of each user password is stored in the password file (instead of the user password itself). In order to verify a user-entered password (for identification), the system applies the one-way function to this password and compares the result to the stored entry for the stated user-id. The use of the one-way function makes it infeasible to derive a valid password from an entry in the password file. Therefore this file must be only write-protected. Note that passwords usually have a fixed or maximum length, but sometimes passphrases of arbitrary length are used. A related application (also useful for identification) is *confirmation of knowledge.* Here a party proves that it has knowledge of a secret S without revealing the secret itself, by submitting a one-way function of S to another party (who also knows this secret).

Pseudo-random string generation

One-way functions can be used to generate pseudo-random bit sequences, e.g., by first selecting a random seed s and then applying the function to the sequence of values $s, s+1, s+2$, etc. For a function f the output sequence then corresponds to $f(s), f(s+1), f(s+2)$, etc. Note that if part of the output sequence is known and if the function can be inverted, an attacker would be able to compute the remainder of the output sequence which is a violation of the design intentions.

Key derivation and one-time passwords

A one-way function can be applied to compute a sequence of keys that are used for the protection of successive communication sessions. Starting from a master key K_0 the first session key is computed as $K_1 = f(K_0)$, the second session key is $K_2 = f(K_1)$, etc. A typical example is the procedure for key derivation used by payment systems (in point-of-sale terminals), where it is important that the disclosure of a currently active key does not compromise the security of previous transaction keys (this property is called forward security [14]). A related example is the generation of a password sequence to be used in a one-time password system. Here the sequence of passwords that has been generated should be used in reverse order. In that way a current password can be verified by applying the one-way function to it and checking whether the result matches the previous password. The one-way property prevents an adversary who knows the current password from computing any future passwords.

2.5 Digital Timestamping Schemes

For further evidence of the versatility of cryptographic hash functions we give an overview of the role they have in digital timestamping schemes. Digital timestamping is a service that provides *temporal authentication.* In particular, a digital timestamp provides proof that a certain piece of information existed prior to the date and time indicated on the timestamp. This has applications in the protection of intellectual property rights and for secure auditing procedures, e.g.:

- a researcher wants to establish his first-to-invent or first-to-file claims;
- a company wants to prove the integrity of its electronic business records, showing that these have not been tampered with or backdated.

Timestamping is also vital for a true non-repudiation service. For signed documents with a long lifetime, it may be necessary that the signature can be verified again at a later point in time. However, the lifetime of a digital signature is limited for various reasons:

- the private signing key may be compromised;
- the certificate that proves the link between the user and his public key, expires at a certain point in time;
- the cryptographic algorithm that is used in the signature scheme, may be broken.

This problem can be solved when a secure timestamping service is in place. A user who creates a signature over a document, requests a timestamp of the signed information, proving that the signature was generated before the time indicated in the timestamp (and, e.g., before a compromise of his private signing key at a later point in time).

In the following we give an overview of several techniques for digital timestamping. This overview is based on a study that was made for the Belgian TIMESEC project [134], and was published in [127]. Timestamping schemes were first described in detail by S. Haber and W. S. Stornetta [58], and further developed in [10, 59]. In our overview we focus on the different uses that are made of cryptographic hash functions.

2.5.1 Simple system based on a trusted third party

A basic solution for timestamping relies on the use of a trusted third party, the *Time Stamping Authority* (TSA). A client C who wants a timestamp for a document X first uses a hash function h_1 to compress his document. He then sends a request to the TSA including his identity ID_C and the hash result $h_1(X)$. The TSA appends to the request a serial number and the current time, and he digitally signs the result to produce the following timestamp which is returned to the client:

$$\text{timestamp} = sign_{TSA}(ID_C, h_1(X), n, t_n)\,.$$

Note that the channel between the client and the TSA should provide mutual authentication and message integrity, but secrecy is not required. The use of the hash function preserves the secrecy of the document (even the TSA does not know X). It also reduces the bandwidth and storage requirements.

If the client or another party wants to validate the timestamp he needs to verify the signature using the public key of the TSA. Furthermore it must be verified that the hash included in the timestamp corresponds to the document to which the timestamp is attached. Obviously, if documents are to be securely and uniquely represented by their hash value, a collision-resistant hash function must be used in the system. The choice of hash function should be imposed by the TSA.

2.5.2 Linking timestamps into a temporal chain

The limitation of the previous system is that the third party TSA must be completely trusted. There is no protection against a malicious TSA that issues backdated timestamps. Furthermore, if the private signing key of the TSA is compromised all issued timestamps become useless. In order to offer additional assurance the TSA can use a hash function h_2 (this can be the same function as h_1 or a different one) to link all timestamp requests in a one-way fashion.

Assume that the nth client sends a timestamp request for a hash value $H_n = h_1(X_n)$. The TSA concatenates the value H_n with a linking value L_{n-1} and computes a new linking value $L_n = h_2(L_{n-1}\|H_n)$. This is repeated for every timestamp request, as illustrated in Fig. 2.7 (note that the computation starts from an initial value L_0).

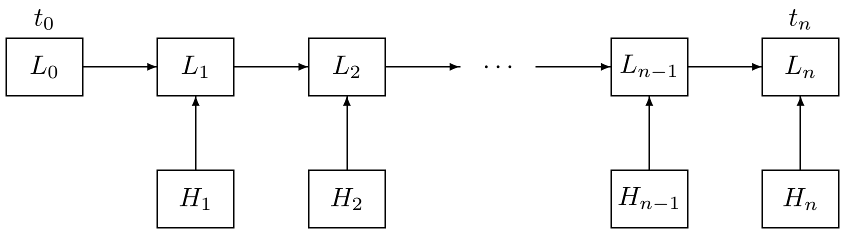

Figure 2.7: Linking timestamp requests into a temporal chain.

When a collision-resistant hash function h_2 is used, this procedure creates an unforgeable temporal chain. For example,

$$L_n = h_2(L_{n-1}\|H_n) = h_2(h_2(L_{n-2}\|H_{n-1})\|H_n)\,,$$

therefore the hash value H_n must have been processed after the value H_{n-1}. If the TSA keeps all values available on an online server, the temporal chain can be recomputed as part of the procedure for timestamp validation. In addition, the TSA should periodically publish some of the linking values in some widely available and unmodifiable media (e.g., a newspaper). Assume that, in the example of Fig. 2.7, the linking values L_0 and L_n are published at time t_0 and t_n respectively. For the ith timestamp request with $0 < i < n$ (corresponding to hash value H_i), it can be checked that this request has been processed between time t_0 and t_n. On the other hand it is not possible to insert a false (backdated) timestamp for a document hash H' in the chain: for a collision-resistant hash function h_2 no other computation path can be found from the value L_0 to the value L_n.

Note that the use of a linking scheme provides *relative* temporal authentication: for every pair of timestamps it can be determined which one comes first. This can be combined with the *absolute* temporal authentication provided by the digital signature of the TSA as described in Sect. 2.5.1. The advantage of the linking scheme is that it does not depend on the private key of the TSA. If a user has a timestamp and its position in the temporal chain is between two published values, this can no longer be changed (not even by the TSA).

2.5.3 Aggregation based on a Merkle tree

The system described above is not very practical for real-world applications because the TSA will have to store a huge number of timestamp requests. Furthermore, the computation path between two trusted (published) linking values can be very long. To solve this, the TSA collects all timestamp requests that it receives during a certain period of time. At the end of the period the TSA computes a Merkle tree [85] based on these requests.

Fig. 2.8 illustrates how a Merkle or binary authentication tree is constructed. The computation starts from the leaves H_1 to H_8. These are the hash values included in the timestamp requests received during the current *round* (period of time). A hash function h_3 is used to compute each parent node in the tree from its two children, e.g., $H_{1-2} = h_3(H_1 \| H_2)$. This is repeated until a single value, the root of the tree, is obtained (H_{1-8} in the example of Fig. 2.8). The procedure is called *aggregation.*[4]

For every round the TSA constructs a Merkle tree from the received timestamp requests, and the root values for successive rounds are linked using the procedure described in Sect. 2.5.2. This means that a new linking value is computed after every round. For example, let the root value of the Merkle tree for

[4]Note that if the number of leaves is not a power of 2, a number of dummy hash values need to be inserted into the tree such that the described procedure can be used.

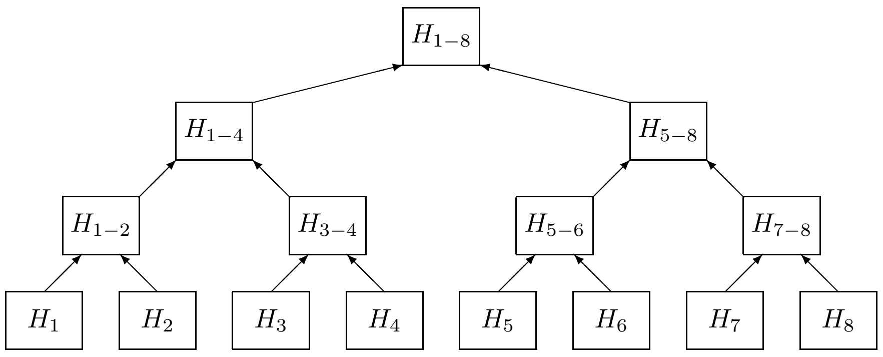

Figure 2.8: Aggregation based on a Merkle tree.

round n be $R_n = H_{1-8}$. Then the new linking value is $L_n = h_2(L_{n-1}\|R_n)$.

At the end of a round, the TSA also returns a timestamp for every request that it received. Such a timestamp includes the round number, the root value for the round, and the information that is needed to rebuild the branch of the tree to which the request belongs. For example, the hash value H_3 will result in a timestamp with the information $(n, R_n, H_4, H_{1-2}, H_{5-8})$.

For validation of a timestamp, the verifier first checks that the timestamp belongs to the specified round. In the case of the timestamp for the hash H_3 he computes $Y_1 = h_3(H_3\|H_4), Y_2 = h_3(H_{1-2}\|Y_1), Y_3 = h_3(Y_2\|H_{5-8})$, and he verifies that Y_3 corresponds to the root value R_n. The next step for the verifier is to check the linking between different round values R_i (which should be available online). The temporal chain should be checked until trusted values (published in reliable media) are encountered.

An adversary who wants to produce a false (backdated) timestamp for a document hash H', needs to construct a false tree branch from H' to the root value of a certain round. This is infeasible when a collision-resistant hash function h_3 is used.

The use of aggregation in this system allows a significant reduction of the length of the temporal chain (and of the verification time). If the average number of requests in a round is m, a reduction with a factor of m is obtained. The tradeoff is that every timestamp must include the information needed for rebuilding the corresponding tree branch (in order to check that the timestamp belongs to the specified round). For a binary tree with m leaves, the size of this information (and the time needed for verifying it) is $\mathcal{O}(\log m)$. Another disadvantage of aggregation is that all timestamps of the same round are simultaneous. For example in Fig. 2.8 there is no proof that H_3 was processed earlier than H_4. Therefore the duration of a round should not be too long (e.g., in a practical system this could be one second).

2.5.4 Other timestamping schemes

Further research on timestamping schemes is due to H. Lipmaa [80] and A. Buldas *et al.* [26, 27]. The use of graph theory, in particular directed acyclic graphs, allows the definition of timestamping schemes with a reduced length of the verification chain. Moreover, these schemes allow relative temporal authentication for timestamps of the same round. The drawback of these schemes is that they are more complex.

2.6 Conclusions

This introductory chapter presented practical definitions for one-way and collision-resistant hash functions (we also referred to more formal definitions). The security of these algorithms was discussed, and some generic techniques to find preimages or collisions. A survey was given of different applications of hash functions, the most important being message authentication and the optimisation of digital signature schemes. The description of the different uses of hash functions in systems for digital timestamping (published in [127]) illustrates the versatility of these cryptographic algorithms.

Chapter 3

Design of Cryptographic Hash Functions

3.1 Introduction

This chapter discusses the design of (unkeyed) cryptographic hash functions. First the required output length for these algorithms is considered. Next we describe the general model of an iterated hash function and discuss the security properties. The main part of the chapter is an overview of different approaches to the design of hash functions, and of the most important algorithms. Several of these algorithms are described and analysed in more detail in Chapters 4 and 5. Finally, a brief overview is given of standardisation efforts in this area. For a more detailed discussion on the different design strategies we refer to the treatment of B. Preneel in [98, 99].

3.2 Required Output Length

In Sect. 2.3 of the previous chapter we described the generic attacks that can be applied to any hash function. It was shown that the time complexity of the random preimage attack is $\mathcal{O}(|\mathcal{R}|)$ operations, and that the time complexity of the birthday attack is $\mathcal{O}(\sqrt{|\mathcal{R}|})$ operations. Here $|\mathcal{R}|$ denotes the number of points in the range of the function, and the 'operations' correspond to the computation of the hash result for a random input.

Assume now that the output space of a hash function consists of all n-bit strings, that is $\mathcal{R} = \{0,1\}^n$ (see Def. 2.1 in Chapter 2). Such an n-bit hash function is said to have *ideal security* if the following holds:

1. producing a preimage or second preimage requires about 2^n operations;
2. producing a collision requires about $2^{n/2}$ operations.

This corresponds to the complexities of the random preimage and birthday attacks respectively (note that $|\mathcal{R}| = 2^n$, and $\sqrt{|\mathcal{R}|} = 2^{n/2}$). In other words, for an ideal hash function these are the best known attacks.

The question remains which output length n is required to make these attacks infeasible for a practical design. In [122] P. van Oorschot and M. Wiener discuss the cost of implementing the birthday attack for the popular MD5 [115] hash function, which has an output length of $n = 128$ bits (so about 2^{64} operations are needed to produce a collision). The analysis, which dates from 1995, leads to an estimate that collisions for MD5 can be found in 21 days using a customised 10M$ machine. Considering 'Moore's law', which states that the computing power available for a given cost doubles every 18 months, it is clear that an output length of $n = 128$ bits is not sufficient for collision-resistance. This result is confirmed by other studies. In 1996 M. Blaze *et al.* [23] estimated that cryptanalytic attacks with a complexity of 2^{75} operations were just out of reach (of powerful adversaries such as an intelligence agency). This corresponds to about 2^{80} operations today. Therefore, a collision-resistant hash function should have an output length of at least $n = 160$ bits. For a one-way hash function (which only needs to be preimage and second preimage-resistant) the output length should be at least $n = 80$ bits. For long-term security larger output lengths should be chosen (e.g., 192 or 96 bits respectively). In the case of one-way hash functions, larger output lengths may also be required in applications where a random preimage attack can target many different hash values simultaneously.

3.3 Iterated Hash Functions

In Chapter 2 hash functions were defined as functions that take an input of arbitrary length, producing an output of a fixed length of n bits. As it is not easy to design a function with inputs of variable length, all known hash functions are based on a *compression function* with fixed size input. This function is used in an iterative manner: the input to the hash function is divided into blocks of a specific length and every block is processed by the compression function in a similar way. X. Lai and J. Massey [79] call the resulting algorithm an *iterated hash function.*

An example of such an iterative hash computation is shown in Fig. 3.1. Here, the algorithm computes the hash result for an input X that is divided into three blocks X_0, X_1, X_2. A compression function f is used which takes two inputs: a *chaining variable* H_i and a block X_i. The chaining variable input for the first iteration is equal to an initial value IV. For the next iterations the chaining variable

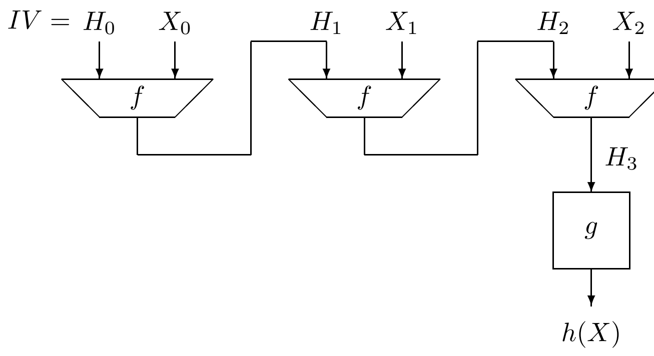

Figure 3.1: Example of an iterative hash computation.

input corresponds to the previous compression function output. The result H_3 from the last application of the compression function is sent to an output transformation g which computes the hash result $h(X) = h(X_0\|X_1\|X_2) = g(H_3)$.

More generally, for an input X consisting of t blocks $X_0, X_1, \ldots X_{t-1}$ the iterative computation of the hash result can be described as follows:

$$\begin{aligned} H_0 &= IV \ , \\ H_{i+1} &= f(H_i, X_i) \quad \text{for } 0 \leq i < t \ , \\ h(X) &= g(H_t) \ . \end{aligned}$$

In Chapter 2 hash functions were formally defined by Def. 2.1. In the same way we can give a formal definition for iterated hash functions. For this we first need to define a compression function and an output transformation.

Definition 3.1 *A* **compression function** *is a function $f : \mathcal{D} \to \mathcal{R}$ where $\mathcal{D} = \{0,1\}^a \times \{0,1\}^b$ and $\mathcal{R} = \{0,1\}^c$ for some $a, b, c \geq 1$ with $a + b \geq c$.*

Definition 3.2 *An* **output transformation** *is a function $g : \mathcal{D} \to \mathcal{R}$ where $\mathcal{D} = \{0,1\}^a$ and $\mathcal{R} = \{0,1\}^n$ for some $a, n \geq 1$ with $a \geq n$.*

Definition 3.3 *Suppose that a compression function $f : (\{0,1\}^c \times \{0,1\}^b) \to \{0,1\}^c$ and an output transformation $g : \{0,1\}^c \to \{0,1\}^n$ are given. Then the* **iterated hash function** *is the hash function $h : (\{0,1\}^b)^* \to \{0,1\}^n$ defined by $h(X_0 \ldots X_{t-1}) = g(H_t)$ where $H_{i+1} = f(H_i, X_i)$ for $0 \leq i < t$. The input blocks $X_i \, (0 \leq i < t) \in \{0,1\}^b$ and the initial chaining value $H_0 = IV \in \{0,1\}^c$.*

From this definition it can be seen that the chaining values H_i have a length of c bits and the input blocks X_i have a length of b bits. If the total number of input bits is not a multiple of b, a *padding rule* must be specified. A typical example is to pad the input data on the right with a single 1-bit, followed by the smallest number of 0-bits which makes the total length of the padded data a multiple of b bits. This padding rule is unambiguous: it is possible to determine where the original input data ends and where the padding bits begin. In Sect. 3.3.1 we will see that for security reasons it is common practice that the padding bits also include a representation of the length of the unpadded input.

The output transformation is often omitted in an iterated hash construction, in other words the identity function is often chosen for g ($g(H_t) = H_t$). In this case the output length is equal to the length of the chaining variable, that means $n = c$. Those iterated hash functions which do employ an output transformation, use it to reduce the length of the hash result, that means $n < c$. This can be done either by simply selecting n out of c bits (e.g., the n leftmost bits) or by applying some folding technique.

3.3.1 Security of iterated hash functions

There are two elements in the definition of an iterated hash function, which have an important influence on the security of the scheme: the choice of the IV and the choice of the padding rule.

For the example of Fig. 3.1 assume that an attacker replaces the value IV by H_1 and that he deletes the first input block X_0, in other words he computes the hash for the input $X = X_1 \| X_2$ starting from an initial chaining variable H_1. Then it is easy to see that the same hash result is obtained, which means that producing second preimages or collisions is a trivial task. To prevent this, the initial value IV should be fixed. The exact value of IV is usually defined in the specification of an algorithm.

Another important security measure is the use of a padding rule which includes the length of the original input data into the padding bits. This prevents attacks based on fixed points, where the attacker tries to produce second preimages or collisions by inserting extra blocks into the input (see Sect. 3.3.2). We can now make the following statement regarding the relation between an iterated hash function h and the underlying compression function f with respect to the property of collision-resistance.

Theorem 3.4 (Merkle-Damgård) *If the IV is fixed and if the padding procedure includes the length of the input into the padding bits, then h is collision-resistant if f is collision-resistant.*

The proof for this theorem was given independently by R. Merkle [86] and I. Damgård [32]. It is based on the argument that a collision for h would imply a collision for f at some stage. The inclusion of the length into the padding bits is needed for this reasoning. The procedure of fixing the IV and adding a representation of the length is called *MD-strengthening.*

Related work by Lai and Massey [79] discusses the relation between an iterated hash function and the underlying compression function with respect to the property of second preimage-resistance.

Theorem 3.5 (Lai-Massey) *Assume that the IV is fixed and that the padding procedure includes the length of the input into the padding bits. Moreover, let the input X contain at least 2 blocks (without padding). Then finding a second preimage for h requires 2^n operations if and only if finding a second preimage for f, with an arbitrary chaining variable input, requires 2^n operations.*

The fact that the condition on f is necessary is based on the following argument: if it takes on average 2^s operations to find a second preimage for f (with $s < n$), then one can use a meet-in-the-middle attack to produce a second preimage for h in about $2 \cdot 2^{(n+s)/2}$ operations. This is discussed in more detail in Sect. 3.3.2.

The previous results imply that breaking an iterated hash function is at least as hard as breaking the underlying compression function. Therefore the security of a hash function can be examined by analysis of its compression function and the designer can focus his attention on this compression function. If an attack exists for the compression function this does not necessarily mean that the hash function can be broken. If preimages or collisions can be found for f, with the computation starting from a pre-specified chaining value H, an attack on h can be derived. For a collision-attack on f where the computation starts from an arbitrary chaining value H this is not the case. Another attack which cannot be extended is one that finds collisions for f with the two computations starting from different chaining values H and H'. Table 3.1 below summarises these different types of attacks. In Sect. 3.3.2 below we will show that preimages, second preimages and collisions can be extended from the compression function to the hash function with a correcting block attack, and that *pseudo-preimages* can be exploited in a meet-in-the-middle attack. On the other hand *random IV* or *pseudo-collisions* do not lead to an attack on the hash function. However if such an attack can be applied to the compression function, this is usually considered as an undesirable property and as a certificational weakness for the hash function.

3.3.2 Chaining attacks

Chaining attacks are based on the iterative nature of hash functions. They focus on the compression function rather than on the overall hash function. Below we describe different types of chaining attacks.

Table 3.1: Different types of attacks on a compression function.

Type of attack	Given	Find	Property
preimage	H, Y	X	$f(H, X) = Y$
second preimage	H, X	X'	$f(H, X) = f(H, X')$
collision	H	X, X'	$f(H, X) = f(H, X')$
pseudo-preimage	Y	H, X	$f(H, X) = Y$
random IV collision	-	H, X, X'	$f(H, X) = f(H, X')$
pseudo-collision	-	H, H', X	$f(H, X) = f(H', X)$

Fixed point attack

A fixed point for a compression function f is a pair H, X for which $f(H, X) = H$. This property allows an attacker to produce second preimages or collisions by inserting an arbitrary number of blocks with the value X at a point in the computation where the chaining variable has the required value H. The attack is only possible if the initial chaining value is not fixed (the attacker chooses $IV = H$), or if fixed points can be found for a significant fraction of all chaining values. Moreover, the attack only works when the padding procedure does not include the length of the input into the padding bits. A generalisation of this attack is the case where fixed points occur after more than one iteration of the compression function.

Correcting block attack

Assume that a cryptanalyst is searching a second preimage for a given input X consisting of t blocks. In other words, he needs to find an alternative input X' with the property $h(X') = h(X)$. In a correcting block attack the cryptanalyst chooses one of the input blocks X_i, and replaces it with an alternative block X'_i so that $f(H_i, X'_i) = f(H_i, X_i)$. If all other blocks of the alternative input X' are equal to the corresponding blocks of X, the same hash result will be obtained and a second preimage has been found.

If the size of the chaining variable is c bits, and the size of the input blocks is b bits with $b > c$, then the number of blocks X'_i satisfying the property $f(H_i, X'_i) = f(H_i, X_i)$ is approximately $2^b/2^c = 2^{b-c}$. The challenge is that such blocks are a small subset of all possible blocks, and for an ideal iterated hash function about 2^c operations are needed to find one. The attack is possible in the case when preimages can be found for the compression function with the computation starting from a pre-specified chaining value. That is, given the chaining values H_i, H_{i+1} the attacker must be able to (efficiently) find a block X'_i such that

$f(H_i, X_i') = H_{i+1}$.

In the case of a collision attack the cryptanalyst can choose both X and X'. The blocks $X_j = X_j'$ are chosen arbitrarily for $j \neq i$. A collision must then be found for the compression function starting from the pre-specified chaining value H_i, i.e., the attacker must find two distinct blocks X_i and X_i' so that $f(H_i, X_i) = f(H_i, X_i')$. This will give a collision in the hash result.

Meet-in-the-middle attack

This attack is a variation on the birthday attack (see Sect. 2.3.1), but instead of comparing the hash result, one compares intermediate chaining variables. When applicable, a meet-in-the-middle attack enables the cryptanalyst to construct second preimages, which is not possible for a simple birthday attack. Assume that a second preimage is searched, for a given input X. Both the initial chaining value ($H_0 = IV$) and the hash result $h(X)$ are fixed. The cryptanalyst identifies an attack point somewhere in the chain between two blocks of a candidate input X'. He then generates r_1 variations on the first part of X' and r_2 variations on the last part. Starting from the initial value IV and going backwards from the hash result $h(X') = h(X)$, the probability for a matching intermediate chaining variable is given by $p = 1 - \exp(-r_1 \cdot r_2/2^n)$.

For the attack to work, the cryptanalyst must be able to efficiently go backwards through the chain. This means that he must be able to invert the compression function in the following manner: given a value H_{i+1}, find a pair H_i, X_i such that $f(H_i, X_i) = H_{i+1}$ (this is called a pseudo-preimage). Lai and Massey [79] show that if it takes on average 2^s operations to invert f in this way, it is possible to find second preimages for h using a meet-in-the-middle attack with a complexity of about $2 \cdot 2^{(n+s)/2}$ operations. J.-J. Quisquater and J.-P. Delescaille [108] show that the use of cycle finding techniques allows meet-in-the-middle attacks with negligible storage, in the same way as for birthday attacks (see Sect. 2.3.1).

3.4 Hash Functions Based on Block Ciphers

From Sect. 3.3 it follows that a designer can focus his attention on the compression function. However, the design of a secure compression function is not an easy task. A possible approach is to base the compression function on an existing cryptographic primitive, such as a block cipher. This has the advantage that existing implementations (in software or hardware) can be reused. For applications which require both encryption and hashing, the size of the implementation can be minimised by using a block cipher for both purposes. Another argument is that some block ciphers — such as the DES [54] and AES [52] algorithms — have received a lot of public scrutiny, and thereby trust in their security properties. A disadvantage is that hash functions based on a block cipher are less efficient than the dedicated proposals discussed in Sect. 3.6. Also, the use of a block cipher for a purpose for which it was not designed may reveal weaknesses which are not relevant in the case of encryption.

In our discussion below we write the encryption operation as $Y = E_K(X)$. Here X denotes the plaintext, Y the ciphertext, and K the key. The size (in bits) of plaintext and ciphertext corresponds to the block length of the cipher and is denoted by b. The key length (in bits) is denoted by k. Typical values for these parameters are $b = 64$ and $k = 56$ for DES, or $b = 128$ and $k = 128$ for AES.[1] For more information on block ciphers we refer to Chapter 7.

3.4.1 Single block length constructions

Single block length hash functions are hash functions based on a block cipher, which produce a hash result with length equal to the block length of the cipher. In other words, the output length of the hash function is $n = b$. From the discussion in Sect. 3.2 it follows that for such constructions a block length of 64 bits is too short. A block length of 128 bits is sufficient for preimage-resistance but not for collision-resistance.

Two dual constructions are those attributed to Matyas-Meyer-Oseas and Davies-Meyer [83]. In each step of the iterated hash computation the current value of the chaining variable (H_i) and the input block to be processed (X_i) serve as key and plaintext of the encryption function (or vice versa). The plaintext input is then added to the output from the encryption function, which constitutes a feed-forward operation. The result from the feed-forward serves as the new chaining value for the next iteration. The computation of the compression function f depends on the encryption function E in the following manner for the two schemes:

- $H_{i+1} = f(H_i, X_i) = E_{z(H_i)}(X_i) \oplus X_i$ (Matyas-Meyer-Oseas);
- $H_{i+1} = f(H_i, X_i) = E_{X_i}(H_i) \oplus H_i$ (Davies-Meyer).

Note that these schemes do not use an output transformation. Therefore the length of the chaining variable is equal to the output length ($c = n = b$). If the key length of the block cipher is different from its block length, the Matyas-Meyer-Oseas scheme needs a function $z(\cdot)$ to transform the chaining value H_i to a key suitable for E (z is a mapping from $\{0,1\}^b$ to $\{0,1\}^k$). The feed-forward operation is based on modulo 2 addition (exclusive-OR), and has the purpose of making the compression function uninvertible. Without feed-forward it would be trivial to find a pair H_i, X_i so that $f(H_i, X_i) = H_{i+1}$ for a given H_{i+1} (simply

[1]Other key lengths, in particular $k = 192$ or 256 are possible for AES [52].

select the key and compute the inverse of the encryption operation). A graphical representation of the Matyas-Meyer-Oseas and Davies-Meyer schemes is shown in Fig. 3.2.

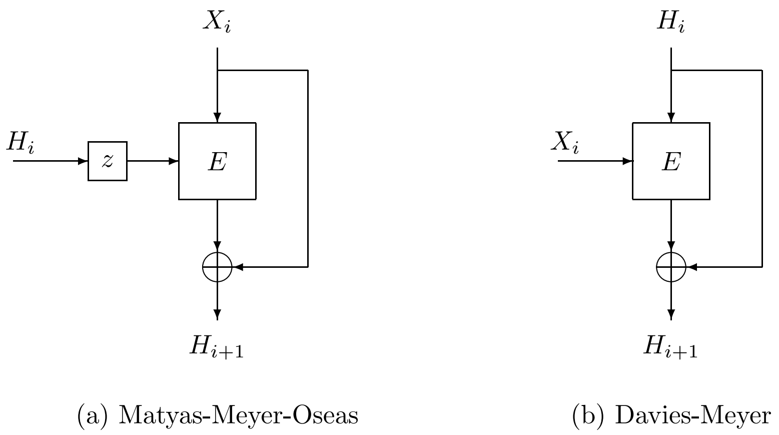

Figure 3.2: Two dual single block length hash functions.

The Matyas-Meyer-Oseas and Davies-Meyer schemes are not the only possible single block length constructions. In [102] Preneel *et al.* consider 64 possible schemes, and show that 12 of them are secure (including Matyas-Meyer-Oseas and Davies-Meyer). This means that finding a (second) preimage requires about 2^n operations, and finding a collision about $2^{n/2}$ operations. Formal proofs for the security of these schemes, under a black-box model for the block cipher, were given by J. Black *et al.* in [22]. An important (and secure) variant of Matyas-Meyer-Oseas, is the Miyaguchi-Preneel scheme [101, 88] which has the following compression function:

– $H_{i+1} = f(H_i, X_i) = E_{z(H_i)}(X_i) \oplus X_i \oplus H_i$ (Miyaguchi-Preneel).

Note that for these block cipher based hash functions a new key is used in each step of the iteration. Practical block ciphers have an internal key scheduling algorithm which converts the key into a sequence of round keys to be used in the internal operation of the cipher. When a cipher is used for encryption the key schedule only needs to be computed when a new encryption key is supplied (the same key can be used for many encryptions). In the hash modes described above however, the key schedule needs to be recomputed for each step of the iteration. This has a significant effect on the efficiency, especially for block ciphers with a slow key scheduling algorithm.[2]

[2]To give an idea, [92] gives the following performance figures (software implementation on

3.4.2 Double block length constructions

For most block ciphers the schemes discussed in Sect. 3.4.1 result in hash functions with an output length that is too short, especially with respect to collision-resistance. An alternative is the use of *double block length hash functions*, which produce a hash result with length equal to twice the block length of the cipher. In other words, the output length of the hash function is $n = 2 \cdot b$. This means, e.g., that DES will result in a 128-bit hash function, and AES in a 256-bit hash function.

The best known schemes in this class are MDC-2 and MDC-4, designed by B. Brachtl *et al.* [25, 87]. The compression function of MDC-2 uses two parallel computations of the Matyas-Meyer-Oseas (single block length) scheme. Let C^L and C^R denote the left and right $b/2$-bit halves of a b-bit value C. Then the compression function of MDC-2 can be described by

$$H_{i+1} \| \tilde{H}_{i+1} = f(H_i \| \tilde{H}_i, X_i)\,,$$

which depends on the following computations:

$$\begin{aligned} C_{i+1} &= E_{z(H_i)}(X_i) \oplus X_i\,, \\ \tilde{C}_{i+1} &= E_{\tilde{z}(\tilde{H}_i)}(X_i) \oplus X_i\,, \\ H_{i+1} &= C^L_{i+1} \,\|\, \tilde{C}^R_{i+1}\,, \\ \tilde{H}_{i+1} &= \tilde{C}^L_{i+1} \,\|\, C^R_{i+1}\,. \end{aligned}$$

Note that if the switch of the left and right halves would be omitted, the two chains would be independent and they could be attacked separately. There is no output transformation, so the length of the chaining variable is equal to the output length ($c = n = 2 \cdot b$).

The compression function of MDC-4 consists of two sequential executions of the MDC-2 compression function. For the second MDC-2 compression, the keys are derived from the outputs (chaining variables) of the first MDC-2 compression, and the plaintext inputs are the outputs (chaining variables) from the opposite sides of the previous MDC-4 compression.

The best known preimage attacks on MDC-2 and MDC-4 require respectively $2^{3n/4}$ and 2^n operations. For the best known collision attacks $2^{n/2}$ operations are needed for both MDC-2 and MDC-4. The preimage attack on MDC-2 shows that this construction does not have ideal security. Moreover, pseudo-preimages and pseudo-collisions can be found for the MDC-2 compression function with a

Pentium III/Linux) for DES: 472 cycles for encryption of a block and 883 cycles for the key schedule. In the case of AES with 128-bit key the figures are: 400 cycles for encryption of a block and 504 cycles for the key schedule.

complexity of about $2^{n/2}$ and $2^{n/4}$ respectively. Pseudo-collisions for the MDC-4 compression function can be found with a complexity of $2^{3n/8}$. Even better attacks apply when DES is used as block cipher ($b = 64, k = 56$) for MDC-2 or MDC-4. For more information we refer to the work of Knudsen and Preneel [75].

An important parameter describing the efficiency of these constructions is the *rate* of the block cipher based hash function. The rate is defined as the number of b-bit input blocks that can be processed with a single encryption. For MDC-2 and MDC-4 the rate is respectively 1/2 and 1/4. Note that the single block length hash functions described in Sect. 3.4.1 have a rate of 1 (except for Davies-Meyer where it is equal to the ratio k/n). Several double block length hash functions with rate 1 have been proposed but these have been broken [74].

3.4.3 Attacks based on properties of the underlying cipher

For block cipher based hash functions weaknesses in the cipher may lead to an attack on the hash function. Differential cryptanalysis [18] is a technique commonly applied to block ciphers, which studies the relation between input and output differences (see also Chapter 7). For the single block length hash functions of Sect. 3.4.1 it is easy to see that, if one finds a difference in the plaintext which yields the same difference in the ciphertext, a collision is obtained in the compression function output (chaining variable). The schemes which use the X_i blocks as plaintext input (Matyas-Meyer-Oseas and Miyaguchi-Preneel) are most vulnerable to such attacks because an attacker has complete control over these blocks. Preneel and Rijmen [96, 111] have studied differential attacks on block cipher based hash functions.

Obviously the best approach for a designer is to base his hash function on a block cipher which has received a substantial amount of analysis and which is believed to be secure. However, one must still be very careful because weaknesses which have no impact on the security of a block cipher when it is used for encryption, may have a serious effect on the security of a hash function based on this cipher. For DES several properties have been identified which may be of concern: fixed points, weak keys, key collisions and the complementation property. Whether these weaknesses can be exploited depends on the design of the hash function. In the case of DES the problems with weak keys and the complementation property can be solved by fixing bits 2 and 3 of the key input to "01" or "10", but this reduces the size of the key space and consequently the security level against preimage and collision attacks. For more information we refer to [89, 96].

3.5 Hash Functions Using Modular Arithmetic

A cryptographic hash function can also use modular arithmetic as the basis of its compression function. This allows the reuse of existing implementations of modular arithmetic (such as in public-key cryptosystems). An advantage of these schemes is that it is easy to scale the security level by choosing a modulus M of appropriate length. A significant disadvantage is that hash functions based on modular arithmetic are very slow, even when compared to the block cipher based constructions of Sect. 3.4. Also, many specific proposals were broken in the past.

Experience has led to the design of two variants of the MASH hash function [64]. The compression function of MASH-1 is based on a modular squaring operation, where the modulus M is chosen as a composite of sufficient bitlength m to make it infeasible to factor M. Typical values are $m = 1024, 1536, \ldots$ For an input block X_i and a chaining value H_i, the compression function computes a new chaining value H_{i+1} in the following manner. The block X_i is first expanded with redundancy bits to a block $\tilde{X}_i$ of double length (the four most significant bits of every byte in $\tilde{X}_i$ are set equal to "1111"). This block $\tilde{X}_i$ is added modulo 2 (exclusive-OR) to the chaining value H_i. Both $\tilde{X}_i$ and H_i have bitlength c, chosen as the largest multiple of 16 less than m. Next the modular square is computed and a feed-forward is applied with the chaining value H_i. This results in the following expression for the compression function:

$$H_{i+1} = ((((H_i \oplus \tilde{X}_i) \vee A)^2 \bmod M) \dashv c) \oplus H_i \,.$$

Here A is the constant $\texttt{f00}\ldots\texttt{00}_x$. The Boolean OR operation with A forces the four most significant bits of $H_i \oplus \tilde{X}_i$ to "1", prior to squaring. $\dashv c$ denotes truncation of the result of the squaring to the c least significant bits.

After the last iteration of the compression function there is an output transformation which reduces the length of the hash result to $n = c/2$ bits (or less). This transformation consists of a number of applications of the compression function (see [64] for details). The security of MASH-1 depends in part on the difficulty of extracting modular roots for a composite of unknown factorisation. It can be shown that the scheme would be totally insecure without the redundancy bits that are added to the blocks X_i. The hash function MASH-2 is a variant of MASH-1, where the only difference is that a modular exponentiation with exponent $e = 2^8 + 1$ is used (instead of modular squaring with $e = 2$). This provides an additional security margin. The best known preimage and collision attacks on MASH-1 and MASH-2 require respectively 2^n and $2^{n/2}$ operations.

3.6 Dedicated Hash Functions

Dedicated hash functions are algorithms which are designed for the explicit purpose of hashing. This means that they are not constrained to the reuse of existing components such as block ciphers or modular arithmetic, and that they can be designed with optimised performance in mind. The hash functions of this type which have received the most attention in practice are those based on the MD4 algorithm. MD4 is a hash function proposed by R. Rivest in 1990 [114]. It was designed specifically towards software implementation on 32-bit platforms. Due to security concerns Rivest designed a more conservative variant, called MD5 [115], shortly afterwards. Other important hash functions based on the same principles are HAVAL [131], and the RIPEMD [100] and SHA [51] families of hash functions. Chapter 4 is devoted to this group of algorithms and gives an extensive overview of design principles and cryptanalytic results.

Other notable hash functions include Tiger [1], an algorithm targeted towards 64-bit processors (it uses look-up tables with 8 input and 64 output bits), and WHIRLPOOL [9], an algorithm based on an internal block cipher ($b = k = 512$) used in the Miyaguchi-Preneel mode. Other dedicated algorithms have been proposed, many of which have been broken. A special case is PANAMA [30], a cryptographic module that can be used for both hashing and stream encryption. In Chapter 5 we describe PANAMA and show a cryptanalysis of its hashing mode.

3.7 Standardisation of Algorithms

Several organisations are active in the standardisation of cryptographic algorithms. The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) develop standards in the joint technical committee ISO/IEC JTC 1. The American National Standards Institute (ANSI) develops standards. The National Institute of Standards and Technology (NIST) develops Federal Information Processing Standards (FIPS), for use by U.S. federal government departments.

The most notable hash function standards are the following. ISO/IEC has developed standard 10118 for hash functions, with separate parts for different classes of hash functions. Part 2 of ISO/IEC 10118 [62] details hash functions based on an (unspecified) block cipher, more specifically the Matyas-Meyer-Oseas construction, MDC-2 and two more functions producing a hash value of double and triple block length respectively. Part 3 of ISO/IEC 10118 [63] specifies seven dedicated algorithms: two of the RIPEMD-family, four of the SHA-family, and WHIRLPOOL. Part 4 of ISO/IEC 10118 [64] includes the MASH-1 and MASH-2 hash functions based on modular arithmetic. ANSI has adopted hash functions in its public-key based banking standards: standard X9.30 [3] specifies the dedicated algorithm SHA-1 as a mandatory part of the Digital Signature Standard (DSS); standard X9.31 [4] specifies MDC-2 to be used in conjunction with an RSA-based digital signature scheme. The NIST standard FIPS 180-2 [51] includes several of the dedicated hash functions of the SHA-family (see Sect. 4.7 of the next chapter for details).

We also mention the efforts of the NESSIE project here (see Sect. 1.2.1). NESSIE published an open call for the submission of cryptographic algorithms (one-way and collision-resistant hash functions were among the requested categories). The Whirlpool hash function was submitted in response to this call. Further, NESSIE also evaluated the hash functions included in the NIST standard FIPS 180-2. All of these algorithms are recommended by NESSIE and are included as part of the NESSIE portfolio of cryptographic primitives.

3.8 Conclusions

We have given an overview of the different approaches that are used for the design of cryptographic hash functions. All known hash functions are based on the iteration of a compression function, and we have explained that a designer can focus his attention on building a secure compression function. For the design of a compression function one can either use existing cryptographic building blocks such as block ciphers, or start from scratch and build a function specific for the purpose of hashing. In the next chapter we discuss several of these dedicated hash functions in detail.

Chapter 4

Dedicated Hash Functions of the MDx-class

4.1 Introduction

In this chapter we give an extensive overview of a group of dedicated hash functions that are all based on similar design ideas. The first of these hash functions was the MD4 algorithm [114], proposed by R. Rivest in 1990. MD4 was a novel design, oriented towards software implementation on 32-bit architectures (as in most common desktop processors), and it achieved a remarkable performance: it is for example more than 10 times faster than the DES block cipher. In the meantime it has been shown that MD4 is not a secure hash function, but several other algorithms have been derived from MD4 (with improved strength); these are often called the MDx-class. Included in the MDx-class are the MD5 algorithm [115] (another design of Rivest), the HAVAL algorithm [131] (proposed by a group of researchers in Australia), the SHA algorithms [51] (U.S. hash function standards of the NIST), and the RIPEMD algorithms [44] (originally developed in the framework of the European RIPE project). These hash functions are the most popular in use today, due to their performance and because of the trust gained from cryptanalytic efforts. Figure 4.1 below gives a timeline showing the history of the MDx-class.

In the following we first give generic comments on the design and cryptanalysis of this type of hash functions. Next we describe the algorithms and discuss their security, and we conclude with a comparison of both security and performance of the different algorithms. An overview on the security of MDx-class hash functions has been published in [126], and the cryptanalysis of HAVAL presented in Sect. 4.5 below has been published in [123].

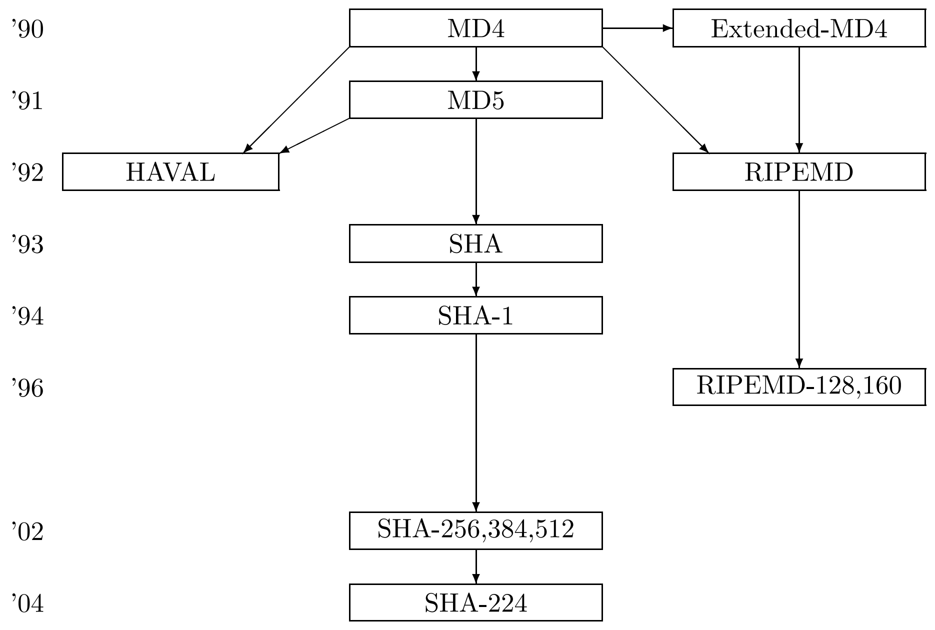

Figure 4.1: History of the MDx-class.

4.2 Design Principles

The hash functions of the MDx-class are iterated constructions, following the model described in Chapter 3 (Sect. 3.3). This means that hashing is based on the iteration of a compression function, taking a chaining variable and a message block as inputs and producing a new value for the chaining variable as output. An initial value is defined for the chaining variable, and the message to be hashed is first pre-processed by adding some padding bits and dividing it in blocks of equal length (b bits). We can make the following observations on the manner in which the iterated model is used for the MDx-class hash functions.

Message pre-processing. A padding scheme is used which appends a single 1-bit to the message, followed by a number d of 0-bits, and finally a number of bits containing a representation of the length of the original message. Here d is chosen as the smallest number (possibly zero) for which the length of the padded message is a multiple of the block length b.

Chaining variable and hash result. The lengths of the chaining variable and the hash result are equal ($c = n$ bits). Furthermore, the hash result is taken as the final value of the chaining variable, obtained after the last application of the compression function. This means that there is no output transformation.[1]

Collision-resistance of the compression function. The compression function is designed to be collision-resistant. From the iterated model, and from the fact that the padding bits contain a representation of the length of the message, this would imply that the hash function is also collision-resistant (according to the Merkle-Damgård theorem, Sect. 3.3.1).

Word-orientation. Hash functions from the MDx-class are word-oriented. This means that all data (chaining variable and message blocks) are divided into words of a specified length (w bits), and the compression function uses only operations on words of this length. MDx-class hash functions have a word length of $w = 32$ or $w = 64$ bits. Note that algorithms with word length w are well-suited for implementation on w-bit architectures.

Little-endian or big-endian conversion. The message to be hashed and the output from the algorithm are usually represented as strings of bytes. Therefore conversions must be made between strings of bytes and sequences of words (or vice-versa). For interoperable implementations on different processors, an unambiguous convention must be specified for these conversions. Consider a string of bytes b_i with increasing memory addresses i, and assume that we have to convert this to a sequence of 32-bit words. Each substring of four consecutive bytes (e.g., b_0, b_1, b_2, b_3) is converted to a 32-bit word W as follows. In a *little-endian* architecture the byte with the lowest memory address (b_0) is the least significant byte of the word: $W = 2^{24} \cdot b_3 + 2^{16} \cdot b_2 + 2^8 \cdot b_1 + b_0$. In a *big-endian* architecture the byte with the lowest memory address is the most significant byte: $W = 2^{24} \cdot b_0 + 2^{16} \cdot b_1 + 2^8 \cdot b_2 + b_3$. Note that algorithms based on the little or big-endian convention are best suited for implementation on little or big-endian architectures respectively (the correct conversions are made automatically).

Sequential structure. The compression function of MDx-class hash functions is of a sequential nature. This means that it consists of a large number of step operations that are executed sequentially (the result of a step must be known in order to proceed to the following step). Furthermore, the elementary step operations have a simple structure. For the parameters c (chaining length) and

[1]Some MDx-class hash functions have an optional output transformation for shorter hash results ($n < c$).

w (word length) the chaining variable consists of c/w words (of w bits each). The values of one or two of these words are updated in a step operation. Each step depends on one w-bit word of the message block that is being processed. Only simple operations on w-bit words are used.

Message expansion. For a block length of b bits, the message blocks that are processed by the compression function consist of b/w words (of w bits each). Assume now that the compression function has a structure consisting of s sequential step operations. As mentioned above every step depends on one message word, so s of these words are needed in total. Here $s > b/w$ for MDx-class hash functions. Therefore a procedure must be specified for expanding the b/w words of the message block input to a block of s words. Some hash functions use a very simple expansion where each of the b/w message words is used multiple times in a number of different steps ($\frac{s}{b/w}$ times to be precise), but other hash functions have a more complex procedure for expansion, based on a linear code.

4.3 Methods of Cryptanalysis

In this chapter we give an overview of attacks published in the literature for MDx-class hash functions. Most of these attacks try to find collisions for the hash function, that is two different messages which are hashed to the same result. Because of the iterated construction, and the Merkle-Damgård theorem, the cryptanalyst can focus his attention on the compression function and try to find two message blocks which are processed to the same output chaining variable. Here the input chaining variable must be equal to the initial value that is specified for the hash function. That is, the attacker tries to find two message blocks M_0 and M_0' such that $f(IV, M_0) = f(IV, M_0')$. This can be extended to a collision for the hash function in a trivial manner: one simply appends a number of common trailing blocks to M_0 and M_0' (these blocks should include the representation of the length). There are a number of concepts that are related to collisions (see also Sect. 3.3.1).

Random IV collisions. These are collisions for the compression function where the computation starts from an arbitrary value H for the input chaining variable: $f(H, M_0) = f(H, M_0')$. Such a collision can not be extended to the hash function itself, except if one is able to find a message block M_{-1} which connects the value H with the initial value IV for the chaining variable: $f(IV, M_{-1}) = H$. However, this would involve finding a preimage for the compression function.

Pseudo-collisions. These are collisions for the compression function, where a common message block M_0 is used, but two different values for the chaining variable input: $f(H, M_0) = f(H', M_0)$. In order to use this for a collision of the hash function one would need to find two message blocks M_{-1} and M'_{-1} which connect the IV with H and H' respectively. This again requires the finding of preimages.

The importance of random IV and pseudo-collisions. Although it is clear that these types of collisions for the compression function do not bring us closer to finding collisions for the hash function itself, their existence means that the Merkle-Damgård theorem, which states that a hash function is collision-resistant if its compression function is collision-resistant, cannot be applied. Therefore, this is regarded as an undesirable property of the compression function, and as a *certificational weakness* for the hash function.

Almost-collisions. Some attacks are able to find almost-collisions for a compression function. This means that two message blocks are found for which the difference between the outputs has a low Hamming weight (say u bits). For an n-bit hash function, one expects that the outputs from the compression function for two distinct message blocks are different in $n/2$ bits on the average. The case $u \ll n/2$ is considered significant, because it means that the compression function does not have *random behaviour*.

Inner (almost-)collisions. These are collisions or almost-collisions for the temporary values of the chaining variable (for two distinct message blocks), at some stage of the compression function (for example after s_1 step operations where $s_1 < s$). This may be helpful for an attacker who tries to generate a collision in the output of the compression function.

Cryptanalytic techniques

Different methods, and sometimes a combination of them, have been proposed for the analysis of hash functions. Differential cryptanalysis [18] is a probabilistic technique that studies the relation between input and output differences for a function. For hash functions collisions are obtained when the difference at the output of the compression function is equal to zero. Note that the input difference can be either in the message block or in the chaining variable input to the compression function. In the latter case pseudo-collisions are obtained and this is in fact similar to differential attacks applied on block ciphers, because there the input difference is in the plaintext and the output difference in the ciphertext. For more information we refer to Chapter 7 where we demonstrate a differential attack on a block cipher, and where we also discuss the relation between hash functions and block ciphers. We will see in this chapter that many collision attacks on MDx-class hash functions work with very small differences during the computation of the compression function. This makes it easier to assure that differences have no effect because they can be cancelled out in a further stage of the compression function. We will also see that many attacks require the solving of a system of non-linear equations where the unknowns are a number of chaining variable words at some stage of the compression function. Both of these ideas were introduced by H. Dobbertin [41]. Finally note that many attacks on hash functions exploit very specific properties, and internal details, of the compression function under analysis.

4.4 The MD4 and MD5 Algorithms

The MD4 and MD5 algorithms were both designed by Rivest. MD4 was first proposed in 1990. It was a novel design and its most attractive feature was that it achieves very good performance in software implementations on 32-bit architectures. However it was soon discovered that its security level was much lower than expected, as demonstrated by some attacks on reduced versions of the algorithm. This prompted the design of MD5 (1991), intended as an improved variant of MD4. The questions regarding the security of MD4 were confirmed when Dobbertin, in 1996, demonstrated a very practical collision-finding attack on the full MD4 algorithm. Furthermore, some serious weaknesses have been shown in MD5 as well.

In this section we give an extended security analysis of MD4, both because of the historical significance of this algorithm and because it allows us to develop our techniques for cryptanalysis which we apply on other hash functions of the MDx-class. We will also discuss more briefly the security weaknesses of MD5.

4.4.1 Description of the MD4 algorithm

The MD4 algorithm [114] computes hash results of 128 bits, for messages of arbitrary length. The algorithm has a word length of 32 bits, therefore the chaining variable is divided into four registers (A, B, C, D) of 32 bits each. The compression function works on message blocks of 512 bits, a block is divided into sixteen 32-bit words denoted by W_j for $j = 0, 1, \ldots, 15$.

Internally, the compression function consists of 48 sequential steps, where each step is used to update the value of one of the four registers. Another distinction that can be made is into rounds: round 1 consists of the first sixteen steps, round 2 of the middle sixteen steps and round 3 of the last sixteen steps. The step

Table 4.1: Word processing order for the three rounds of MD4.

Round 1	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7
	W_8	W_9	W_{10}	W_{11}	W_{12}	W_{13}	W_{14}	W_{15}
Round 2	W_0	W_4	W_8	W_{12}	W_1	W_5	W_9	W_{13}
	W_2	W_6	W_{10}	W_{14}	W_3	W_7	W_{11}	W_{15}
Round 3	W_0	W_8	W_4	W_{12}	W_2	W_{10}	W_6	W_{14}
	W_1	W_9	W_5	W_{13}	W_3	W_{11}	W_7	W_{15}

operation of MD4 is of the following form:

$$A \leftarrow (A + f_r(B, C, D) + W_j + U_r)^{\lll v_s} .$$

Here we consider a step that updates the value of the A register. The operation depends on the other three registers (B, C, D), and on the following:

– a message word W_j from the set $j = \{0, 1, \ldots, 15\}$;

– a Boolean function f_r that depends on the round;

– an additive constant U_r that depends on the round;

– a rotation constant v_s that depends on the step.

Note that the additions in the step operation are performed modulo 2^{32}.

The Boolean functions used in the three rounds of the compression function are the *selection*, *majority* and *exor* functions respectively. These functions operate at bit level and are implemented with the Boolean AND, OR, exclusive-OR and NOT operations. Each of the three rounds of the compression function uses every word W_j from the set $j = \{0, 1, \ldots, 15\}$ exactly once but the order in which the sixteen message words are applied in a round is different in every round as shown in Table 4.1.

A graphical representation of the step operation of MD4 is given in Fig. 4.2. Note that the registers change place after the step operation. Therefore four consecutive steps update the values of the registers A, D, C, B respectively. After four steps the complete chaining variable has been updated. A round of the compression function consists of four sequences of four steps. Hence, each register is updated four times in every round, and twelve times in the complete compression function (three rounds).

Note that the step operation of MD4 is reversible. For example, the previous value of the A register can be computed by

$$A_{prev} = A_{new}^{\ggg v_s} - f_r(B, C, D) - W_j - U_r .$$

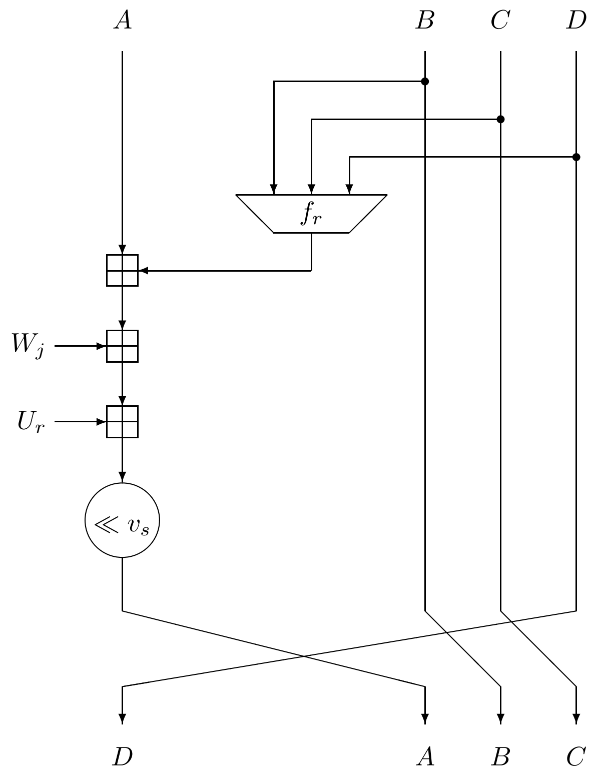

Figure 4.2: Step operation for MD4.

However after execution of all 48 steps, the compression function uses a feed-forward operation which adds the initial values of the registers (the values at the start of the compression function) to their final values (obtained after 48 steps). The result is the chaining variable output from the compression function. Due to the feed-forward at the end the compression function cannot be inverted. An outline of the compression function including the feed-forward is shown in Fig. 4.3. Inputs are a 4-word chaining variable (A, B, C, D) and a 16-word message block $\{W_j\}_{j=0\ldots15}$. Output is a new value for the chaining variable.

For a detailed description of MD4, and for an explanation of the notations that we use in our analysis below, we refer to Appendix B.

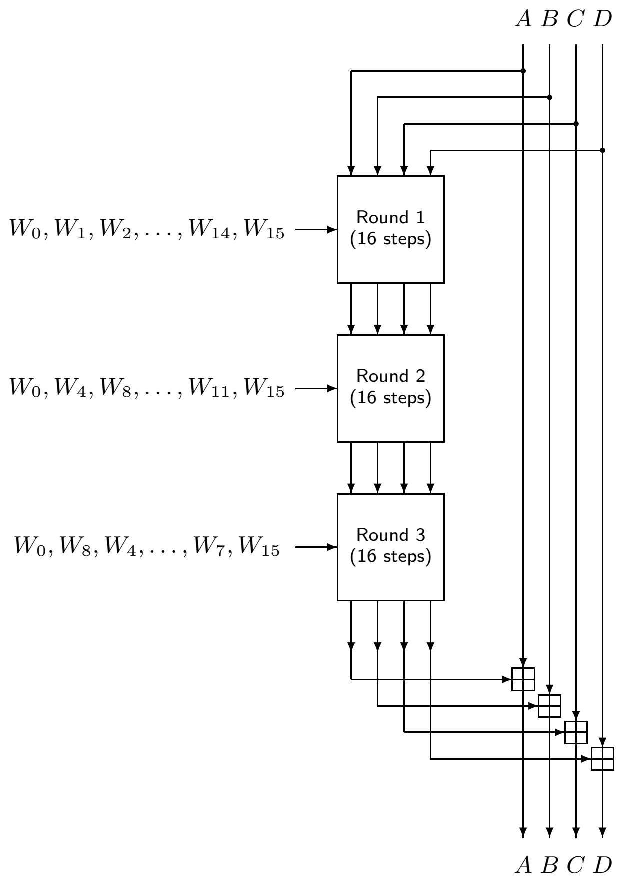
A B C D
Round 1 (16 steps)
Round 2 (16 steps)
Round 3 (16 steps)
$W_0, W_1, W_2, \ldots, W_{14}, W_{15}$
$W_0, W_4, W_8, \ldots, W_{11}, W_{15}$
$W_0, W_8, W_4, \ldots, W_7, W_{15}$
A B C D

Figure 4.3: Outline of the MD4 compression function.

4.4.2 Analysis of MD4 reduced to two rounds

We first consider a variant of MD4 where the compression function has been reduced to the first two rounds (step 1 up to step 32, see Appendix B), and we develop a technique for finding collisions for this reduced variant. The analysis is based on a modification of the first part of Dobbertin's attack [42] on the complete MD4 algorithm (see Sect. 4.4.4).

Outline of the attack

The goal of our attack is to find two distinct message blocks $\{W_j\}$ and $\{W'_j\}$ $(0 \leq j \leq 15)$ which are mapped by the compression function to the same output value, where the computation for the two message blocks starts from the same 128-bit input chaining value (A_0, B_0, C_0, D_0). We find such a collision for two message blocks with a small difference in only one of the words, more specifically:

$$\begin{aligned} W'_{12} &= W_{12} + 1\,, \\ W'_j &= W_j \ \ (j \neq 12)\,. \end{aligned}$$

The compression function reduced to two rounds updates each of the four registers of the chaining variable eight times (four times in a round). We denote these values by (A_i, B_i, C_i, D_i) with $1 \leq i \leq 8$. The output of the reduced compression function is then computed with a feed-forward as $(A_0 + A_8, B_0 + B_8, C_0 + C_8, D_0 + D_8)$.

It can be seen that the word W_{12}, respectively W'_{12} (which contains the only difference between the two message blocks), is applied two times, in step 13 and in step 20 (see Appendix B). Before step 13 all values of the registers are equal for the two message blocks; a collision will be obtained if the values of all registers are equal again after execution of step 20 (hereafter all message words that are used are the same for both message blocks so no new differences will occur in any computed register value). We can make our attack succeed if we control the differences in registers between step 13 and step 20 very carefully.

The first use of the word W_{12}, respectively W'_{12}, is in step 13 (round 1 of the compression function) where a new value is computed for the A register. This means that the first computed register value which is not equal for the two message blocks, is the value A_4, respectively A'_4. At this point we require the following correspondence between the registers for the two message blocks:

$$A_4 = A'_4 - 1 \quad B_3 = B'_3 \quad C_3 = C'_3 \quad D_3 = D'_3 \ .$$

The next use of W_{12}, respectively W'_{12}, occurs in step 20 (round 2 of the compression function) where a new value is computed for the B register. For a collision we need the following correspondence between register values at this point:

$$A_5 = A'_5 \quad B_5 = B'_5 \quad C_5 = C'_5 \quad D_5 = D'_5 \ .$$

Table 4.2 below shows the difference propagation used in the attack. We use the notations $\Delta A = A - A', \Delta B = B - B', \Delta C = C - C', \Delta D = D - D'$ where (A, B, C, D) are the values of the registers at this point for message block $\{W_j\}$, and similarly (A', B', C', D') for $\{W'_j\}$. Note that the differences are defined with respect to the modular addition operation. The difference values shown are the values *after* the corresponding step has been executed. We also list the message word applied in each step. Entries in bold face show which register has been updated in a particular step. Note that we write the value -1 as shorthand for $-1 \bmod 2^{32} = 2^{32} - 1$. This corresponds to a 32-bit quantity where all the bits are set equal to 1.

Table 4.2: Overview of the difference propagation through the registers for rounds 1 and 2 of MD4 (for an inner collision).

Step	ΔA	ΔB	ΔC	ΔD	word
13	$-\mathbf{1}$	0	0	0	$W_{12}(+1)$
14	-1	0	0	**0**	W_{13}
15	-1	0	**0**	0	W_{14}
16	-1	**1**	0	0	W_{15}
17	**0**	1	0	0	W_0
18	0	1	0	**0**	W_4
19	0	1	**0**	0	W_8
20	0	**0**	0	0	$W_{12}(+1)$

In step 13 which uses the word W_{12}, respectively W'_{12}, a difference in the A register is introduced: $A_4 - A'_4 = -1$. For steps 14 and 15 we require that the difference in the A register does not spread to the D and C registers: $D_4 - D'_4 = C_4 - C'_4 = 0$. We then let the difference in the A register spread to the B register in step 16: $B_4 - B'_4 = 1$. For step 17 we require that the differences in the A and B registers interact in such a way that the difference in the A register disappears: $A_5 - A'_5 = 0$. Now we have only a difference in the B register which should not spread to the D and C registers in steps 18 and 19: $D_5 - D'_5 = C_5 - C'_5 = 0$. Finally in step 20 where the words W_{12}, respectively W'_{12}, are used again we require that the difference in the B register disappears: $B_5 - B'_5 = 0$.

Constructing a system of difference equations

For each step in turn, we now look at the difference which is obtained after computing the new register value for the message blocks $\{W_j\}$ and $\{W'_j\}$.

Step 13. In this step we have a difference in the applied message word W_{12}, respectively W'_{12}. Using the definition of the step operation (see Appendix B), the equalities $A'_3 = A_3$, $B'_3 = B_3$, $C'_3 = C_3$, $D'_3 = D_3$, and $W'_{12} = W_{12} + 1$ we get the following equations

$$A_4 = (A_3 + f_1(B_3, C_3, D_3) + W_{12})^{\lll 3},$$

$$A'_4 \quad = \quad (A_3 + f_1(B_3, C_3, D_3) + W_{12} + 1)^{\lll 3}\,.$$

Note that the Boolean function f_1 is the selection function used in round 1 of the compression function. To simplify the analysis we choose $A_4 = -1$. It can be seen that this implies that $A_3 + f_1(B_3, C_3, D_3) + W_{12} = A_4^{\ggg 3} = -1$ and therefore that $A'_4 = (-1+1)^{\lll 3} = 0$. This agrees with the difference $A_4 - A'_4 = -1$.

Step 14. From the definition of the step operation we get

$$\begin{aligned} D_4 &= (D_3 + f_1(A_4, B_3, C_3) + W_{13})^{\lll 7}\,, \\ D'_4 &= (D_3 + f_1(A'_4, B_3, C_3) + W_{13})^{\lll 7}\,. \end{aligned}$$

We require that $D_4 - D'_4 = 0$ which leads to the condition

$$f_1(A_4, B_3, C_3) - f_1(A'_4, B_3, C_3) = 0\,. \tag{4.1}$$

Step 15. We require that $C_4 - C'_4 = 0$. It can be seen that this leads to the condition

$$f_1(D_4, A_4, B_3) - f_1(D_4, A'_4, B_3) = 0\,. \tag{4.2}$$

Step 16. In this step we need to obtain the right difference $B_4 - B'_4 = 1$. We have the following equations:

$$\begin{aligned} B_4 &= (B_3 + f_1(C_4, D_4, A_4) + W_{15})^{\lll 19}\,, \\ B'_4 &= (B_3 + f_1(C_4, D_4, A'_4) + W_{15})^{\lll 19}\,. \end{aligned}$$

If we now freely choose a value of $B_4 = Q_1$ and set $B'_4 = Q_1 - 1$ (to satisfy the difference $B_4 - B'_4 = 1$), this leads to the condition

$$f_1(C_4, D_4, A_4) - f_1(C_4, D_4, A'_4) = Q_1^{\ggg 19} - (Q_1 - 1)^{\ggg 19}\,. \tag{4.3}$$

Step 17. This is the first step of round 2 of the compression function so the majority function f_2 is used in the step operation (and an additive constant U_2, see Appendix B):

$$\begin{aligned} A_5 &= (A_4 + f_2(B_4, C_4, D_4) + W_0 + U_2)^{\lll 3}\,, \\ A'_5 &= (A'_4 + f_2(B'_4, C_4, D_4) + W_0 + U_2)^{\lll 3}\,. \end{aligned}$$

We require that $A_5 - A'_5 = 0$. By inserting the values $A_4 = -1$ and $A'_4 = 0$ we derive the condition

$$f_2(B_4, C_4, D_4) - f_2(B'_4, C_4, D_4) = 1\,. \tag{4.4}$$

Step 18. For this step we require $D_5 - D'_5 = 0$. This is satisfied when

$$f_2(A_5, B_4, C_4) - f_2(A_5, B'_4, C_4) = 0\,. \tag{4.5}$$

Step 19. We require $C_5 - C'_5 = 0$ which is satisfied when

$$f_2(D_5, A_5, B_4) - f_2(D_5, A_5, B'_4) = 0\,. \tag{4.6}$$

Step 20. In this step the message word W_{12}, respectively W'_{12} is applied again:

$$\begin{aligned} B_5 &= (B_4 + f_2(C_5, D_5, A_5) + W_{12} + U_2)^{\lll 13}\,, \\ B'_5 &= (B'_4 + f_2(C_5, D_5, A_5) + W_{12} + 1 + U_2)^{\lll 13}\,. \end{aligned}$$

We require that $B_5 - B'_5 = 0$ which is satisfied because $B'_4 = B_4 - 1$.

Solving the system of equations

To summarise our analysis, a collision for the two rounds of the reduced compression function is obtained by setting the register values

$$A_4 = -1 \quad A'_4 = 0 \quad B_4 = Q_1 \quad B'_4 = Q_1 - 1\ \ ,$$

and satisfying the system of equations (4.1) to (4.6). This is a system with six equations and six unknown variables B_3, C_3, C_4, D_4, A_5, D_5.

By using the definition of the selection function f_1 and inserting the values of A_4 and A'_4 it can be seen that equations (4.1) and (4.2) are equivalent to

$$B_3 = C_3\,, \quad D_4 = 0\,.$$

In a similar way, inserting also the value $D_4 = 0$, equation (4.3) can be rewritten as

$$\overline{C_4} = Q_1^{\ggg 19} - (Q_1 - 1)^{\ggg 19}.$$

From the definition of the majority function f_2 it can be seen that equations (4.5) and (4.6) are satisfied when

$$D_5 = A_5 = C_4\,.$$

In fact $D_5 = A_5 = C_4$ only needs to be satisfied in those bit positions where B_4 is different from B'_4 ($B_4 = B'_4 + 1$, so they usually differ in only one or a few bit positions, depending on the carry associated with the modular addition). At the other positions the bits of A_5 and D_5 can be freely chosen.

Hence, we solve the system of equations by setting the register values B_3, C_3, C_4, D_4, A_5, D_5 as specified above (we can freely choose a value Q_2 and assign it to $B_3 = C_3$). We have not yet considered equation (4.4), but because of the values of C_4, D_4 and $B_4 - B'_4$ it turns out that this equation is satisfied with a probability close to 1 (for a random choice of $B_4 = Q_1$).[2]

Performing the attack

Above we have shown that a collision is obtained by selecting a suitable set of register values B_3, C_3, A_4, B_4, C_4, D_4, A_5, D_5 for message block $\{W_j\}$ and a set B'_3, C'_3, A'_4, B'_4, C'_4, D'_4, A'_5, D'_5 for message block $\{W'_j\}$ (note that only A'_4 and B'_4 are different). Now we need to determine the message words themselves. It can be seen that some of these words are already fixed because of the register values that are fixed, for example from step 15

$$C_4 = (C_3 + f_1(D_4, A_4, B_3) + W_{14})^{\lll 11} ,$$

where the only unknown variable is W_{14}, we can compute the following value:

$$W_{14} = C_4^{\ggg 11} - C_3 - f_1(D_4, A_4, B_3) .$$

In the same way we have the following fixed message words

$$\begin{aligned} W_{15} &= B_4^{\ggg 19} - B_3 - f_1(C_4, D_4, A_4) , \\ W_0 &= A_5^{\ggg 3} - A_4 - f_2(B_4, C_4, D_4) - U_2 , \\ W_4 &= D_5^{\ggg 5} - D_4 - f_2(A_5, B_4, C_4) - U_2 . \end{aligned}$$

For any choice of the remaining twelve message words a collision can be generated (by defining the second message block with $W'_{12} = W_{12} + 1$ and $W'_j = W_j$ for $j \neq 12$). When all message words have been chosen we can compute backwards in round 1 starting from register values C_3, B_3, A_4, D_4. For example, choosing W_{13} and inverting step 14 gives us

$$D_3 = D_4^{\ggg 7} - f_1(A_4, B_3, C_3) - W_{13} .$$

In this way we finally obtain the register values A_0, B_0, C_0, D_0. This means that we have obtained a collision for the compression function starting from a random initial chaining variable.

It is easy to extend our attack so that we can find collisions for the compression function starting from a specified initial chaining variable (and consequently also for the hash function, see Sect. 3.3). For example we can choose eight message words W_1, W_2, W_3, W_5, W_6, W_7, W_8, W_9 and compute forwards from the specified (A_0, B_0, C_0, D_0) (remember that W_0 and W_4 are already fixed). In this

[2]It may be noted that equation (4.4) is not satisfied (and therefore the attack does not work) for the choice $B_4 = Q_1 = 0$ and $B'_4 = Q_1 - 1 = -1$.

way we obtain the register values C_2, B_2, A_3, D_3. We also know the register values C_3, B_3, A_4, D_4 so now we can compute the required values for the remaining four message words as follows:

$$\begin{aligned}
W_{10} &= C_3^{\ggg 11} - C_2 - f_1(D_3, A_3, B_2)\,, \\
W_{11} &= B_3^{\ggg 19} - B_2 - f_1(C_3, D_3, A_3)\,, \\
W_{12} &= A_4^{\ggg 3} - A_3 - f_1(B_3, C_3, D_3)\,, \\
W_{13} &= D_4^{\ggg 7} - D_3 - f_1(A_4, B_3, C_3)\,.
\end{aligned}$$

Complexity and flexibility of the attack

Our attack on the first two rounds of MD4 has negligible complexity. We construct a solution for the system of equations by selecting suitable values for some of the register values. We then compute four message words, choose eight other message words, and compute the remaining four message words. The message words are computed from simple equations derived from the step operations.

The flexibility of our attack can be measured by means of the total number of different collisions which can in theory be generated by the attack. When we have a solution for the system of equations we can freely choose eight message words of 32 bits, this leads to $2^{8 \cdot 32} = 2^{256}$ different collisions. Furthermore, many different solutions for the system of equations can be found: there are two 32-bit values Q_1 and Q_2 that can be freely chosen (64 bits combined), and most of the bits of A_5 and D_5 can also be chosen (on average more than 60 of the 64 bits can be chosen). Hence the number of possible solutions for the system of equations can be estimated by $2^{64+60} = 2^{124}$, and the total number of collisions by $2^{124} \cdot 2^{256} = 2^{380}$.

4.4.3 Analysis of the complete MD4 algorithm

Now we will extend our analysis to the complete MD4 algorithm which uses a compression function of three rounds. Applying the attack described in the previous section would give us an *inner collision* for rounds 1/2 of the compression function, but this does not help us in finding collisions after three rounds. The reason is that in round 3 the message word W_{12}, respectively W'_{12}, is applied once more and this introduces a difference in the register values which cannot be compensated.

The solution to this problem is that we generate an *inner almost-collision* for rounds 1/2 (this means that there is a small difference after round 2). As in the first attack we use message blocks that differ only in the word W_{12}, respectively W'_{12}. The small difference after round 2 must be chosen in such a way that there is a significant probability that it propagates to a difference in round 3 which is compensated when the word W_{12}, respectively W'_{12}, is used for the third time. So we combine the technique of the previous attack (constructing and solving a system of difference equations) with the technique of differential cryptanalysis. The attack which we describe in this section is a simplified variant of the attack described by Dobbertin in [42]. In particular, the procedure for solving the system of difference equations has been simplified (see also Sect. 4.4.4).

Outline of the attack

As in the previous attack we generate a collision for two message blocks $\{W_j\}$ and $\{W'_j\}$ $(0 \le j \le 15)$ with a small difference in only one message word:

$$\begin{aligned} W'_{12} &= W_{12} + 1\,, \\ W'_j &= W_j \;\; (j \neq 12)\,. \end{aligned}$$

Note that the compression function of the complete MD4 algorithm has three rounds, so it updates each of the four registers of the chaining variable twelve times (four times in a round). We denote these values by (A_i, B_i, C_i, D_i) with $1 \le i \le 12$. The output of the compression function is then computed with a feed-forward as $(A_0 + A_{12}, B_0 + B_{12}, C_0 + C_{12}, D_0 + D_{12})$.

The word W_{12}, respectively W'_{12}, is applied three times, in steps 13, 20 and 36. Before step 13 all values of the registers are equal for the two message blocks; a collision is obtained if the values of all registers are equal again after execution of step 36.

In phase I of the attack we generate an inner almost-collision after step 20 (round 2). This is similar to the inner collision generated by the attack described in Sect. 4.4.2, except that we look for the following correspondence between register values after step 20:

$$A_5 = A'_5 \quad B_5 = B'_5 + 1^{\lll 25} \quad C_5 = C'_5 - 1^{\lll 5} \quad D_5 = D'_5 \;.$$

In phase II of the attack we perform a differential cryptanalysis from step 20 to step 36. We require that the differences in the C and B registers do not spread to the A and D registers. Furthermore, the differences in the C and B registers after step 19 and step 20 have been chosen such that the difference in the C register disappears in step 35, and to compensate the difference in the B register in step 36 by means of the message word W_{12}, respectively W'_{12}. This results in a collision in the output of the compression function.

In order to generate an inner almost-collision (phase I of the attack) we need to fix a few of the words W_j in the message blocks. We can then randomly choose the remaining words in phase II and see if the differential attack works. The success probability of the differential attack is around 2^{-22}, so a collision can be found by randomly choosing the remaining words W_j and computing the difference after step 36 (which should be zero for all registers). This will succeed after, on average, 2^{22} trials. There is one more complication: matching the specified initial chaining variable (A_0, B_0, C_0, D_0). This can be done with a small adaptation of the differential attack.

Phase I: Finding an inner almost-collision for rounds 1/2

We first analyse the part of the compression function between step 13 and step 20. Table 4.3 below shows the difference propagation used in this phase of the attack. In step 13 which uses the word W_{12}, respectively W'_{12}, a difference in the A register is introduced: $A_4 - A'_4 = -1$. For step 14 we require that the difference in the A register does not spread to the D register: $D_4 - D'_4 = 0$. In step 15 we let the difference in the A register spread to an unspecified difference in the C register ($C_4 - C'_4 = \Delta_c$), and in step 16 the differences in the A and C registers spread to an unspecified difference in the B register ($B_4 - B'_4 = \Delta_b$). For step 17 we require that the difference in the A register is compensated by the differences in the B and C registers so that $A_5 - A'_5 = 0$. For step 18 we require that the differences in the B and C registers do not spread to the D register: $D_5 - D'_5 = 0$. Finally in steps 19 and 20 (note that the word W_{12}, respectively W'_{12}, is used again in step 20) we need to obtain the specified differences in the C and B registers: $C_5 - C'_5 = -1^{\lll 5}$ and $B_5 - B'_5 = 1^{\lll 25}$.

Table 4.3: Overview of the difference propagation through the registers for rounds 1 and 2 of MD4 (for an inner almost-collision).

Step	ΔA	ΔB	ΔC	ΔD	word
13	$\mathbf{-1}$	0	0	0	$W_{12}(+1)$
14	-1	0	0	$\mathbf{0}$	W_{13}
15	-1	0	$\boldsymbol{\Delta_c}$	0	W_{14}
16	-1	$\boldsymbol{\Delta_b}$	Δ_c	0	W_{15}
17	$\mathbf{0}$	Δ_b	Δ_c	0	W_0
18	0	Δ_b	Δ_c	$\mathbf{0}$	W_4
19	0	Δ_b	$\mathbf{-1^{\lll 5}}$	0	W_8
20	0	$\mathbf{1^{\lll 25}}$	$-1^{\lll 5}$	0	$W_{12}(+1)$

We construct a system of difference equations in a manner similar to the attack described in Sect. 4.4.2. We again choose the values $A_4 = -1$ and $A'_4 = 0$ (this satisfies step 13) and obtain the following system of equations[3] (Δ_c denotes

[3]Compared to the attack of Sect. 4.4.2, there is one additional equation for step 20.

$C_4 - C_4'$ and Δ_b denotes $B_4 - B_4'$):

$$\begin{aligned}
f_1(A_4, B_3, C_3) - f_1(A_4', B_3, C_3) &= 0 && (4.7)\\
f_1(D_4, A_4, B_3) - f_1(D_4, A_4', B_3) &= C_4^{\ggg 11} - C_4'^{\ggg 11} && (4.8)\\
f_1(C_4, D_4, A_4) - f_1(C_4', D_4, A_4') &= B_4^{\ggg 19} - B_4'^{\ggg 19} && (4.9)\\
f_2(B_4, C_4, D_4) - f_2(B_4', C_4', D_4) &= 1 && (4.10)\\
f_2(A_5, B_4, C_4) - f_2(A_5, B_4', C_4') &= 0 && (4.11)\\
f_2(D_5, A_5, B_4) - f_2(D_5, A_5, B_4') &= C_5^{\ggg 9} - C_5'^{\ggg 9} - \Delta_c && (4.12)\\
f_2(C_5, D_5, A_5) - f_2(C_5', D_5, A_5) &= B_5^{\ggg 13} - B_5'^{\ggg 13} - \Delta_b + 1 && (4.13)
\end{aligned}$$

We simplify the analysis by choosing $B_3 = C_3 = 0$. If we insert the values of A_4, A_4', B_3 and C_3 and use the definition of the selection function f_1 we see that equation (4.7) is satisfied and equation (4.8) is equivalent to:

$$D_4 = C_4^{\ggg 11} - C_4'^{\ggg 11} \,. \qquad (4.14)$$

This equation can be satisfied by assigning suitable values to $C_4 = Q_1$, $C_4' = Q_2$ and $D_4 = Q_3$. This is done independently from the other equations of the system, but a suitable choice for Q_1, Q_2, Q_3 must be made so that the functions f_1 and f_2 are easy to manipulate in equations (4.15) and (4.16) below. Now we add an additional equation to our system:

$$B_5^{\ggg 13} - B_5'^{\ggg 13} - \Delta_b + 1 = 0\,,$$

and rewrite the system of equations as follows (using the values of A_4, A_4', C_4, C_4', D_4 and the relations $C_5' = C_5 + 1^{\lll 5}$ and $B_5' = B_5 - 1^{\lll 25}$):

$$\begin{aligned}
&f_1(Q_1, Q_3, -1) - f_1(Q_2, Q_3, 0) = B_4^{\ggg 19} - B_4'^{\ggg 19} && (4.15)\\
&f_2(B_4, Q_1, Q_3) - f_2(B_4', Q_2, Q_3) = 1 && (4.16)\\
&f_2(A_5, B_4, Q_1) - f_2(A_5, B_4', Q_2) = 0 && (4.17)\\
&f_2(D_5, A_5, B_4) - f_2(D_5, A_5, B_4') = C_5^{\ggg 9} - (C_5 + 1^{\lll 5})^{\ggg 9} - \Delta_c && (4.18)\\
&f_2(C_5, D_5, A_5) - f_2((C_5 + 1^{\lll 5}), D_5, A_5) = 0 && (4.19)\\
&B_5^{\ggg 13} - (B_5 - 1^{\lll 25})^{\ggg 13} - \Delta_b + 1 = 0 && (4.20)
\end{aligned}$$

The resulting system has six unknown variables B_4, B_4', A_5, D_5, C_5 and B_5. We propose the following procedure for solving this system of equations:

1. Determine the unknowns B_4 and B_4' from equations (4.15) and (4.16). This can be done by trying random values for B_4, computing for each value of B_4 a corresponding value for B_4' from equation (4.15), and repeating this until equation (4.16) is satisfied.

2. Determine the unknown A_5 from equation (4.17). Simply try random values for A_5 until the equation is satisfied.

3. Determine the unknowns D_5 and C_5 from equations (4.18) and (4.19). This is done by trying random values for both D_5 and C_5 until the equations are satisfied. Note that Δ_c is known ($\Delta_c = Q_1 - Q_2$).

4. Determine the unknown B_5 from equation (4.20). This is done by trying random values for B_5 until the equation is satisfied. Note that Δ_b is known ($\Delta_b = B_4 - B'_4$, and B_4 and B'_4 have been determined).

The solution for this system of equations is more complicated compared to the solution for the inner collision in Sect. 4.4.2. However, the procedure that we propose is still very efficient. Because of the special form of the difference equations, and because the differences in the registers for the two message blocks are small, it turns out that they are satisfied with a high probability and a few trials are sufficient in each step of the procedure. There is however one more complication. The inner almost-collision that is found is only useful as a starting point for the next phase of the attack if the following condition is satisfied:

$$f_2(B_5, C_5, D_5) = f_2(B'_5, C'_5, D_5) \tag{4.21}$$

Because the differences $B_5 - B'_5$ and $C_5 - C'_5$ are small, there is a high probability for this condition to be true. Therefore, the procedure for solving the system of equations (4.15) to (4.20) is repeated a few times until equation (4.21) is satisfied. Such a solution is called an *admissable* inner almost-collision. The complexity for this phase of the attack is negligible compared to the complexity of the differential attack that we describe below.

We have now determined a suitable set of register values B_3, C_3, A_4, B_4, C_4, D_4, A_5, D_5, C_5, B_5 for message block $\{W_j\}$ and a set B'_3, C'_3, A'_4, B'_4, C'_4, D'_4, A'_5, D'_5, C'_5, B'_5 for message block $\{W'_j\}$ (note that only A'_4, B'_4, C'_4, B'_5 and C'_5 are different). It can be seen that this implies that six of the message words are now fixed, more specifically:

$$\begin{aligned}
W_{14} &= C_4^{\gg 11} - C_3 - f_1(D_4, A_4, B_3)\,, \\
W_{15} &= B_4^{\gg 19} - B_3 - f_1(C_4, D_4, A_4)\,, \\
W_0 &= A_5^{\gg 3} - A_4 - f_2(B_4, C_4, D_4) - U_2\,, \\
W_4 &= D_5^{\gg 5} - D_4 - f_2(A_5, B_4, C_4) - U_2\,, \\
W_8 &= C_5^{\gg 9} - C_4 - f_2(D_5, A_5, B_4) - U_2\,, \\
W_{12} &= B_5^{\gg 13} - B_4 - f_2(C_5, D_5, A_5) - U_2\,.
\end{aligned}$$

Phase II: Differential attack on rounds 2/3

In this phase of the attack we consider the part of the compression function between step 20 and step 36. We have an input difference in the B and C registers from the first phase of the attack ($B_5 - B'_5 = 1^{\lll 25}$ and $C_5 - C'_5 = -1^{\lll 5}$) and we require that all differences have disappeared after step 36. Table 4.4 below shows the difference propagation for this phase of the attack. For the B register we have the following differences: $B_5 - B'_5 = 1^{\lll 25}, B_6 - B'_6 = 1^{\lll 6}, B_7 - B'_7 = 1^{\lll 19}, B_8 - B'_8 = 1, B_9 - B'_9 = 0$. For the C register we have the following differences: $C_5 - C'_5 = -1^{\lll 5}, C_6 - C'_6 = -1^{\lll 14}, C_7 - C'_7 = -1^{\lll 23}, C_8 - C'_8 = -1, C_9 - C'_9 = 0$. Note that the different rotation amounts for these differences are due to the rotation that is performed when the B or C register is updated in a step operation. Also note that all differences for the A and D registers must be zero.

Table 4.4: Overview of the difference propagation through the registers for rounds 2 and 3 of MD4 (differential attack).

Step	ΔA	ΔB	ΔC	ΔD	prob.	word
21	**0**	$1^{\lll 25}$	$-1^{\lll 5}$	0	1	W_1
22	0	$1^{\lll 25}$	$-1^{\lll 5}$	**0**	1/9	W_5
23	0	$1^{\lll 25}$	$\mathbf{-1^{\lll 14}}$	0	1/3	W_9
24	0	$\mathbf{1^{\lll 6}}$	$-1^{\lll 14}$	0	1/3	W_{13}
25	**0**	$1^{\lll 6}$	$-1^{\lll 14}$	0	1/9	W_2
26	0	$1^{\lll 6}$	$-1^{\lll 14}$	**0**	1/9	W_6
27	0	$1^{\lll 6}$	$\mathbf{-1^{\lll 23}}$	0	1/3	W_{10}
28	0	$\mathbf{1^{\lll 19}}$	$-1^{\lll 23}$	0	1/3	W_{14}
29	**0**	$1^{\lll 19}$	$-1^{\lll 23}$	0	1/9	W_3
30	0	$1^{\lll 19}$	$-1^{\lll 23}$	**0**	1/9	W_7
31	0	$1^{\lll 19}$	$\mathbf{-1}$	0	1/3	W_{11}
32	0	**1**	-1	0	1/3	W_{15}
33	**0**	1	-1	0	1/3	W_0
34	0	1	-1	**0**	1/3	W_8
35	0	1	**0**	0	1/3	W_4
36	0	**0**	0	0	1	$W_{12}(+1)$

For each step we give the probability in Table 4.4. Note that because we start from an admissable inner almost-collision, the probability of step 21 is equal to 1. For inner almost-collisions that are not admissable the values of the registers are not suitable at the start of the differential attack.

Steps 21 to 32 are part of round 2 of the compression function and use the majority function f_2 in the step operation, steps 33 to 36 are part of round 3 and use the exor function f_3 in the step operation. The probabilities associated with these steps can be computed: it is required that differences are cancelled by the majority function, or that they compensate each other in the exor function. For more details we refer to the analysis of Dobbertin in [42]. It is interesting to note that we are not able to perform this differential attack on the last two rounds using a difference in only one of the four registers. This is due to the properties of the exor function f_3 which is applied in round 3 of the compression function.

By combining the probabilities for all steps we can estimate the global probability for the propagation from step 20 up to step 36 as $p_{36}^{20} \approx 2^{-30}$. The real probability is much higher however. The reason is that the probabilities for consecutive steps strongly depend on each other (because every step changes the value of only one out of four registers). Experiments show that the global probability $p_{36}^{20} \approx 2^{-22}$.

The differential attack can be performed as follows. In phase I of the attack we saw that message words W_0, W_4, W_8, W_{12}, W_{14} and W_{15} are already determined in order to obtain an admissable inner almost-collision. We can now randomly choose the remaining ten message words and compute forwards to step 36, starting from the known register values A_5, D_5, C_5, B_5 (or C_5', B_5' for the second message block). If the difference after step 36 is equal to zero for all registers then we have a collision and this happens on average after 2^{22} trials. There is however one more complication which we describe below.

Matching the initial value

When all message words W_j are determined we can also compute backwards in round 1 of the compression function, starting from the known register values C_3, B_3, A_4, D_4. In this way we obtain the register values A_0, B_0, C_0, D_0. This means that we have obtained a collision for the compression function starting from a random initial chaining variable.

The attack can be extended so that we find collisions for the compression function starting from a specified initial chaining variable (and consequently also for the hash function, see Sect. 3.3). First we observe that the straightforward approach used in Sect. 4.4.2 for matching the IV cannot be used. The reason is that there is no sequence of four message words, applied in consecutive steps of round 1, that we can still freely choose in phase II of the attack. A solution to this problem is described in [42]. The basic idea is to match the initial values for the B, C and D registers using the words W_9, W_{10} and W_{11}. Message word W_8 has already been determined before, however it is possible to match the initial value for the A register using words W_6 and W_7 and manipulating the properties of the selection function f_1. The differential attack now works by choosing random values for message words W_1, W_2, W_3, W_5, W_{13}, computing the remaining message words so that the initial chaining values are matched and checking if the difference after step 36 is zero for all registers. This succeeds after on average 2^{22} trials.

Complexity and flexibility of the attack

The attack on the complete MD4 hash function has a complexity of about 2^{22} computations of the compression function (this complexity is determined by the probability for the differential attack on the last two rounds). A programme that implements this attack takes only a few seconds on an Athlon 600 MHz processor. For each admissable inner almost-collision we can generate in theory about 2^{138} collisions (five message words of 32 bits can be freely chosen in the differential attack and the probability of success is 2^{-22}). The number of admissable inner almost-collisions is more difficult to estimate, but it is less than 2^{160} (we can randomly choose five words in the procedure for solving the system of difference equations). Therefore, the total number of collisions is less than $2^{160} \cdot 2^{138} = 2^{298}$.

Example collision for the MD4 algorithm

We give an example of two message blocks that are hashed by the compression function of MD4 to the same output value. For both message blocks the computation starts from the initial value specified for the algorithm (see Appendix B):

$$A_0 = \texttt{67452301}_x \;\; B_0 = \texttt{efcdab89}_x \;\; C_0 = \texttt{98badcfe}_x \;\; D_0 = \texttt{10325476}_x \,.$$

The first message block is:

$$\begin{array}{rlrl}
W_0 &= \texttt{238f9f19}_x & W_1 &= \texttt{db18463f}_x \\
W_2 &= \texttt{58fbbe89}_x & W_3 &= \texttt{e67bc739}_x \\
W_4 &= \texttt{2842db84}_x & W_5 &= \texttt{46113e5a}_x \\
W_6 &= \texttt{81b9080a}_x & W_7 &= \texttt{eb9d3337}_x \\
W_8 &= \texttt{0d8da57b}_x & W_9 &= \texttt{79f28417}_x \\
W_{10} &= \texttt{3cbb0fc0}_x & W_{11} &= \texttt{210c3d70}_x \\
W_{12} &= \texttt{1250d4e9}_x & W_{13} &= \texttt{ef44937a}_x \\
W_{14} &= \texttt{ffffbffb}_x & W_{15} &= \texttt{6eb32ef3}_x
\end{array}$$

and the second message block is determined by

$$\begin{array}{rcl}
W'_{12} &=& W_{12} + 1 \,, \\
W'_j &=& W_j \;\; (0 \leq j \leq 15, j \neq 12) \,.
\end{array}$$

For these two message blocks, the compression function computes the following common output value (note that this computation includes the feed-forward operation at the end):

$$A = \texttt{e044225a}_x \;\; B = \texttt{f07f7932}_x \;\; C = \texttt{ac7d9007}_x \;\; D = \texttt{91348c2f}_x\,.$$

The complete hash function includes an additional application of the compression function, starting from the output value given above. For both messages the same padding block is used as message input for this final application of the compression function, therefore a collision is obtained in the final hash result:

$$A = \texttt{c7294cf7}_x \;\; B = \texttt{15daf73c}_x \;\; C = \texttt{9dca10d0}_x \;\; D = \texttt{dd0d66c5}_x\,.$$

Note that the algorithm converts this set of words into a string of sixteen bytes, starting with the least significant byte of A and ending with the most significant byte of D (see Appendix B).

4.4.4 Other weaknesses in MD4

In this section we give a brief overview of other weaknesses that have been found in the MD4 hash function. First we note that it may be possible to develop a similar attack on MD4 where the difference in the message blocks is not in the word W_{12} but in some other word W_j. The word W_{12} has been chosen for the attack described in Sect. 4.4.3 because it gives a good trade-off between the complexities of the two phases of the attack. The number of steps between the applications of W_j in round 1 and round 2 should not be too high (otherwise we don't have enough free variables for the second phase of the attack). On the other hand the probability of the differential propagation characteristic (in rounds 2 and 3) must be high enough. In particular, the number of steps in the characteristic where the exor function f_3 is used must not be too high because of the fast diffusion of this function.

Dobbertin's collision attack on the complete MD4 algorithm

The attack we described in the previous section is a simplification of the attack described by Dobbertin in [42]. Dobbertin's collision attack finds more general solutions for the first phase of the attack (the generation of an inner almost-collision). In particular, it does not impose the condition that $D_4 - D'_4 = 0$. The resulting system of equations is more difficult to solve, and a technique called "continuous approximation" is introduced to achieve this. The resulting attack is more flexible: our variant of the attack produces only a subset of all possible inner almost-collisions. The time complexity is the same, it is determined by the probability for the differential analysis in the second phase of the attack. Finally, it is noted in [42] that the attack can be improved, it is even possible to generate collisions for meaningful messages (messages that contain only a limited number of random bytes).

Collisions for a 256-bit extension of MD4

Extended-MD4 [114] is a variant of the MD4 algorithm, intended for applications which require an output of 256 bits. It consists of the following procedure. Two copies of MD4 are executed in parallel over the message input. The first copy is the standard MD4, and the second copy is a slightly modified version: it uses a different initial value for the chaining variable, and different additive constants in rounds 2 and 3 of the compression function. Furthermore, after every 16-word message block has been processed, the values of the A registers in the two copies are exchanged. The final hash result is obtained by concatenating the results of the two copies.

According to [42] collisions can be generated for the compression function of this extended version. The two copies of MD4 can be attacked simultaneously, but the attack only works when the two copies start from the same (random or specified) initial value for the chaining variable. Therefore, it does not lead to collisions for the hash function itself.

Preimage attacks on reduced versions of MD4

Cryptanalysis of hash functions has dealt almost exclusively with collision-finding attacks so far. However, it has been shown that for MD4 with its compression function reduced to two rounds, it is also possible to find preimages. In [43] Dobbertin demonstrates such a preimage attack on the first two rounds of MD4. In a preimage attack we start from specified initial values (A_0, B_0, C_0, D_0) and specified output values $(A_0 + A_8, B_0 + B_8, C_0 + C_8, D_0 + D_8)$ for the registers.[4] This means that the register values (A_8, B_8, C_8, D_8) are also fixed. The goal is to determine a suitable message block $\{W_j\}$ (with $0 \leq j \leq 15$). The main idea is to separate the values of the B register from the other registers by exploiting the properties of the majority function which is used in the second round of the compression function (the B register is chosen because of the order in which the message words are applied, as explained below). It can be seen that if two of the inputs to the majority function are equal, then the value of the third input has no impact on the output of the function:

$$f_2(B_i, Q, Q) = f_2(Q, B_i, Q) = f_2(Q, Q, B_i) = Q\,. \tag{4.22}$$

Table 4.5 below shows the register values for the end of round 1 and for round 2 of the compression function. The shown values are the values *after* the corresponding step has been executed. We also list the message word applied in each step. Entries in bold face show which register has been updated in a particular step.

[4]Remember that the output values are computed with a feed-forward.

Table 4.5: Overview of the register values for a preimage attack on the first two rounds of MD4.

Step	A	B	C	D	word
13	$\mathbf{Q}$	B_3	C_3	D_3	W_{12}
14	Q	B_3	C_3	$\mathbf{Q}$	W_{13}
15	Q	B_3	$\mathbf{Q}$	Q	W_{14}
16	Q	$\mathbf{B_4}$	Q	Q	W_{15}
17	$\mathbf{Q}$	B_4	Q	Q	W_0
18	Q	B_4	Q	$\mathbf{Q}$	W_4
19	Q	B_4	$\mathbf{Q}$	Q	W_8
20	Q	$\mathbf{B_5}$	Q	Q	W_{12}
21	$\mathbf{Q}$	B_5	Q	Q	W_1
22	Q	B_5	Q	$\mathbf{Q}$	W_5
23	Q	B_5	$\mathbf{Q}$	Q	W_9
24	Q	$\mathbf{B_6}$	Q	Q	W_{13}
25	$\mathbf{Q}$	B_6	Q	Q	W_2
26	Q	B_6	Q	$\mathbf{Q}$	W_6
27	Q	B_6	$\mathbf{Q}$	Q	W_{10}
28	Q	$\mathbf{P}$	Q	Q	W_{14}
29	$\mathbf{A_8}$	P	Q	Q	W_3
30	A_8	P	Q	$\mathbf{D_8}$	W_7
31	A_8	P	$\mathbf{C_8}$	D_8	W_{11}
32	A_8	$\mathbf{B_8}$	C_8	D_8	W_{15}

We start the attack by choosing two random values P and Q, and we specify that $B_7 = P$ and $A_i = C_i = D_i = Q$ for $4 \leq i \leq 7$ (note that A_4, C_4, D_4 are the values for registers A, C and D at the end of round 1). It can be seen that message words $W_0, W_1, \ldots, W_{11}$ are now determined. They are computed from steps 17–19, 21–23, 25–27, and 29–31 in round 2. For example, step 17 leads to the following equation:[5]

$$Q = (Q + f_2(B_4, Q, Q) + W_0 + U_2)^{\lll 3},$$

from which the value of message word W_0 is computed as

$$W_0 = Q^{\ggg 3} - Q - f_2(B_4, Q, Q) - U_2 = Q^{\ggg 3} - 2 \cdot Q - U_2 .$$

Because of property (4.22) of the majority function, the values of B_4, B_5, B_6,

[5]Remember that U_2 denotes the additive constant used in round 2.

and consequently the values for message words W_{12}, W_{13}, W_{14} do not need to be known for these computations.

Message word W_{15} is also determined. Step 32 leads to the equation

$$B_8 = (P + f_2(C_8, D_8, A_8) + W_{15} + U_2)^{\lll 13} ,$$

from which the value of message word W_{15} is computed as

$$W_{15} = B_8^{\ggg 13} - P - f_2(C_8, D_8, A_8) - U_2 .$$

Using the values for message words $W_0, W_1, \ldots, W_{11}$ we can compute forwards in round 1 of the compression function, starting from the initial values (A_0, B_0, C_0, D_0). In this way we obtain the values for (A_3, B_3, C_3, D_3). Now we are able to determine the values that are needed for W_{12}, W_{13}, W_{14}. They are computed from steps 13–15 in round 1: the values (A_3, B_3, C_3, D_3) are known, and also the values $A_4 = C_4 = D_4 = Q$.

Finally we use the value of W_{15} to compute B_4 in step 16, and we use the values of W_{12}, W_{13}, W_{14} in steps 20, 24, 28 (round 2) to compute B_5, B_6, B_7. If the computed value of B_7 is equal to the value $B_7 = P$ which we chose at the start of the attack, we have obtained a valid preimage. This happens with a probability of 2^{-32}. Otherwise we repeat the procedure for new randomly chosen values P, Q and, on average, we succeed after 2^{31} trials. It can be seen that the attack reduces the problem of matching a 128-bit value to the problem of matching a 32-bit value (for register B only).

In [77] H. Kuwakado and H. Tanaka describe an alternative preimage attack on the first two rounds of MD4. This attack is based on the same principle, but it exploits the properties of the selection function instead (this function is used in round 1 of the compression function). The attack can also be applied on MD4 using only rounds 1 and 3 in the compression function.

It may be noted that both of the preimage attacks [43, 77] depend not only on the properties of the Boolean functions used by MD4, but also on the order in which the message words are applied in the different rounds of the compression function.

Alternative collision attacks on reduced versions of MD4

In Sect. 4.4.2 we have described a collision attack, with negligible complexity, on the first two rounds of MD4. Alternative collision attacks have been demonstrated on reduced versions of MD4. In [35] B. den Boer and A. Bosselaers describe a collision attack on the last two rounds of MD4. The attack exploits the fact that the same set of message words is used in the middle eight steps of rounds 2 and 3. A collision is constructed from two separate inner collisions, in round 2 (step 21 to step 28) and in round 3 (step 37 to step 44) of the compression function.

In [128] S. Vaudenay describes a collision attack on the first two rounds of MD4. This attack uses a similar strategy as the preimage attack of Dobbertin described above: it uses property (4.22) of the majority function to isolate the B register from the other registers. In this way the problem of finding a 128-bit collision is reduced to the problem of finding a 32-bit collision (for register B only). Moreover, if this attack is applied to the complete MD4 compression function (three rounds), one obtains almost-collisions (the average Hamming distance between the outputs from the compression function for a pair of message blocks is only $16 \ll 64$).

4.4.5 Description of the MD5 algorithm

The MD5 algorithm [115] was designed as a strengthened variant of MD4. The algorithm computes hash values of 128 bits so the chaining variable is divided into four registers (A, B, C, D) of 32 bits each. The design of MD5 is very similar to MD4 but with the following changes:

- The compression function now consists of 64 sequential steps, divided into four rounds (MD4 has only 48 steps and three rounds).
- The step operation is slightly different: each step now adds in the result of the previous step.
- The order in which the message words W_j are applied in round 2 and round 3 is different from MD4.
- The Boolean function in round 2 has been changed from the majority function to a selection function (different from the selection function used in round 1). A new Boolean function is introduced for round 4.
- Every step now uses a unique additive constant (so there are 64 different additive constants).
- The rotation constants are changed, different rounds never use the same value for a rotation constant.

The step operation of MD5 is of the following form:

$$A \leftarrow (A + f_r(B, C, D) + W_j + U_s)^{\lll v_s} + B\,.$$

Here we consider a step that updates the value of the A register. The operation depends on the other three registers (B, C, D), and on the following:

– a message word W_j from the set $j = \{0, 1, \ldots, 15\}$;

– a Boolean function f_r that depends on the round;

– an additive constant U_s that depends on the step;

– a rotation constant v_s that depends on the step.

A graphical representation of this step operation is given in Fig. 4.4. Four consecutive steps update the values of the registers A, D, C, B respectively, and this is repeated four times in a round. Hence, each register is updated sixteen times by the compression function (four rounds). It can be seen that (for example) when the A register is updated, the final operation is an addition of the B register which was updated in the previous step. The motivation for this final addition is that it assures that differences which are introduced in one register, propagate quickly to the other registers in the following steps.

The step operation of MD5 is reversible. For example, the previous value of the A register can be computed by

$$A_{prev} = (A_{new} - B)^{\gg v_s} - f_r(B, C, D) - W_j - U_s \, .$$

However after execution of all 64 steps, the compression function uses a feed-forward operation which adds the initial values of the registers (the values at the start of the compression function) to their final values (obtained after 64 steps). The result is the chaining variable output from the compression function. Due to the feed-forward at the end the compression function cannot be inverted.

For a detailed description of MD5 we refer to [115].

4.4.6 Weaknesses in MD5

We give a short overview of weaknesses found in the MD5 algorithm. Both of the attacks described below analyse the complete compression function of MD5, but they do not lead to an attack on the hash function itself. However, these attacks point to significant weaknesses in the design of the compression function and also imply that the theorem of Merkle-Damgård, which derives the security of a hash function from its underlying compression function (see Sect. 3.3.1), cannot be applied for MD5.

Pseudo-collisions for the compression function

In [36] B. den Boer and A. Bosselaers describe an attack that finds pseudo-collisions for the compression function of MD5. This means that two different initial chaining values are found, producing the same output value for a certain message block. More specifically, the attack searches an initial value (A, B, C, D),

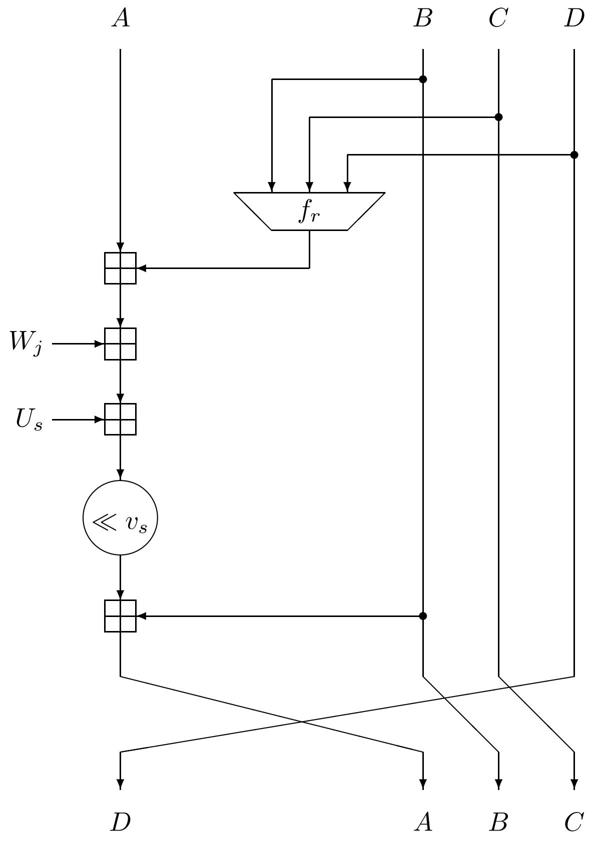

Figure 4.4: Step operation for MD5.

and a message block $\{W_j\}$ $(0 \leq j \leq 15)$, such that complementing the most significant bit of each of the four registers A, B, C, D has no influence on the output of the compression function. We refer to [36] for the details, but it is important to note that the attack works because the step operation of MD5 adds in the result of the previous step. Hence, a similar attack does not work for MD4. Although pseudo-collisions do not bring us closer to finding collisions for the hash function, this attack points to some undesirable characteristics in the design of the compression function of MD5.

Collisions for the compression function

In [39] Dobbertin demonstrates an attack that finds collisions for the compression function of MD5. This means that two different message blocks, $\{W_j\}$ and $\{W'_j\}$ $(0 \leq j \leq 15)$, are found that produce the same output value, for a certain initial chaining value (A, B, C, D). The attack applies similar techniques as those used in the cryptanalysis of MD4. Collisions are found for two message blocks with a small difference in only one of the words:

$$\begin{aligned} W'_{14} &= W_{14} + 1^{\lll 9}\,, \\ W'_j &= W_j \ \ (j \neq 14)\,. \end{aligned}$$

The general outline of the attack is as follows:

1. find an inner collision for rounds 1/2 (step 15 to step 26);
2. find an inner collision for rounds 3/4 (step 36 to step 51);
3. connect the two inner collisions.

The finding of inner collisions in the first two parts of the attack involves the construction and solution of a system of difference equations, as in the analysis of MD4. The analysis of MD5 is more complicated because a simultaneous solution must be found for the different parts of the attack, and because there is a great overlap between the message words that are involved in these different parts. Techniques related to differential cryptanalysis and continuous approximations are used. The complexity of the attack corresponds to about 2^{34} computations of the compression function.

At the end of the attack the message blocks $\{W_j\}$ and $\{W'_j\}$ are determined, and also the values of the registers at the end of step 14. Hence, it is possible to compute backwards in round 1 and obtain the register values (A_0, B_0, C_0, D_0). This means that a collision has been found for the compression function starting from a random initial chaining variable. In order to generate collisions for the hash function itself, it would be necessary to extend the attack in such a way that the prescribed initial value $(A, B, C, D) = IV$, as defined in the specification of MD5, is matched.

4.4.7 Conclusions for MD4 and MD5

We have given an extensive overview of the design and security of the MD4 and MD5 algorithms. MD4 has clearly been broken with respect to collision-resistance: a very practical attack exists which can even be adapted in order to give collisions for meaningful messages. Even with respect to the preimage-resistance of MD4 some weaknesses have been shown (on reduced versions of the algorithm), therefore it is advised that MD4 should no longer be used for applications requiring a secure one-way or collision-resistant hash function.

For MD5 only weaknesses in the compression function have been demonstrated so far. While these weaknesses cannot be exploited for an attack on the hash function in any normal applications, the security margin is small and the properties of the compression function are undesirable for a hash function which should be collision-resistant (the theoretical results for iterated hash functions, explained in Chapter 3, can no longer be used). Furthermore, the output length of 128 bits is now considered as insufficient for resistance against birthday attacks.

4.5 The HAVAL Algorithm

The HAVAL algorithm [131] was proposed by Y. Zheng *et al.* in 1992. Its has a structure that is quite similar to the MD4 and MD5 algorithms. In contrast to MD4 and MD5, HAVAL allows the computation of hashes of variable length. The specification of the algorithm allows for a trade-off between efficiency and security by means of a parameter, the *number of rounds*, which can be chosen equal to 3, 4 or 5.

In this section we give an extended security analysis of HAVAL. Our attack on the three-round version of HAVAL (for all possible output lengths) is the first known attack on a complete version of the HAVAL algorithm. This result has been published in [123].

4.5.1 Description of the HAVAL algorithm

The HAVAL algorithm computes hash values of variable length, more specifically 128, 160, 192, 224 or 256 bits. The algorithm has a word length of 32 bits and uses a chaining variable that is divided into eight registers (A, B, C, D, E, F, G, H) of 32 bits each. The compression function works on message blocks of 1024 bits each, a block is divided into thirty-two 32-bit words denoted by W_j for $j = 0, 1, \ldots, 31$.

Internally, the compression function consists of a number of rounds. This number can be chosen equal to 3, 4 or 5 (a larger number of rounds increases the security but decreases the efficiency of the algorithm). Each round itself consists of 32 sequential steps. Therefore, the total number of steps in the compression function is 96, 128 or 160. Each step updates the value of one of the eight registers. The step operation of HAVAL is of the following form:

$$A \leftarrow A^{\gg 11} + (f_r(B, C, D, E, F, G, H))^{\gg 7} + W_j + U_s\,.$$

Here we consider a step that updates the value of the A register. The operation depends on the other seven registers (B, C, D, E, F, G, H), and on the following:

Table 4.6: Word processing order for the first three rounds of HAVAL.

Round 1	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7
	W_8	W_9	W_{10}	W_{11}	W_{12}	W_{13}	W_{14}	W_{15}
	W_{16}	W_{17}	W_{18}	W_{19}	W_{20}	W_{21}	W_{22}	W_{23}
	W_{24}	W_{25}	W_{26}	W_{27}	W_{28}	W_{29}	W_{30}	W_{31}
Round 2	W_5	W_{14}	W_{26}	W_{18}	W_{11}	W_{28}	W_7	W_{16}
	W_0	W_{23}	W_{20}	W_{22}	W_1	W_{10}	W_4	W_8
	W_{30}	W_3	W_{21}	W_9	W_{17}	W_{24}	W_{29}	W_6
	W_{19}	W_{12}	W_{15}	W_{13}	W_2	W_{25}	W_{31}	W_{27}
Round 3	W_{19}	W_9	W_4	W_{20}	W_{28}	W_{17}	W_8	W_{22}
	W_{29}	W_{14}	W_{25}	W_{12}	W_{24}	W_{30}	W_{16}	W_{26}
	W_{31}	W_{15}	W_7	W_3	W_1	W_0	W_{18}	W_{27}
	W_{13}	W_6	W_{21}	W_{10}	W_{23}	W_{11}	W_5	W_2

– a message word W_j from the set $j = \{0, 1, \ldots, 31\}$;

– a Boolean function f_r that depends on the round;

– an additive constant U_s that depends on the step.

Each round of the compression function uses every word W_j from the set $j = \{0, 1, \ldots, 31\}$ exactly once but the order in which the thirty-two message words are applied in a round is different in different rounds. This is shown for the first three rounds in Table 4.6 below.

A graphical representation of the step operation of HAVAL is given in Fig. 4.5. Note that the registers change place after the step operation. Therefore eight consecutive steps update the values of the registers A, B, C, D, E, F, G, H respectively. After eight steps the complete chaining variable has been updated. A round of the compression function consists of four sequences of eight steps. Hence, each register is updated four times in every round.

Compared to MD4 the main difference in HAVAL is that the chaining variable has 256 bits instead of 128 bits. Therefore, the number of registers has been doubled (eight instead of four). The size of the message blocks has been doubled as well (thirty-two words instead of sixteen), and consequently also the number of steps in each round of the compression function.

Another important difference with MD4 is that the Boolean functions of HAVAL take seven words of input (instead of only three). A great deal of attention has been paid to the design of these Boolean functions, which satisfy the

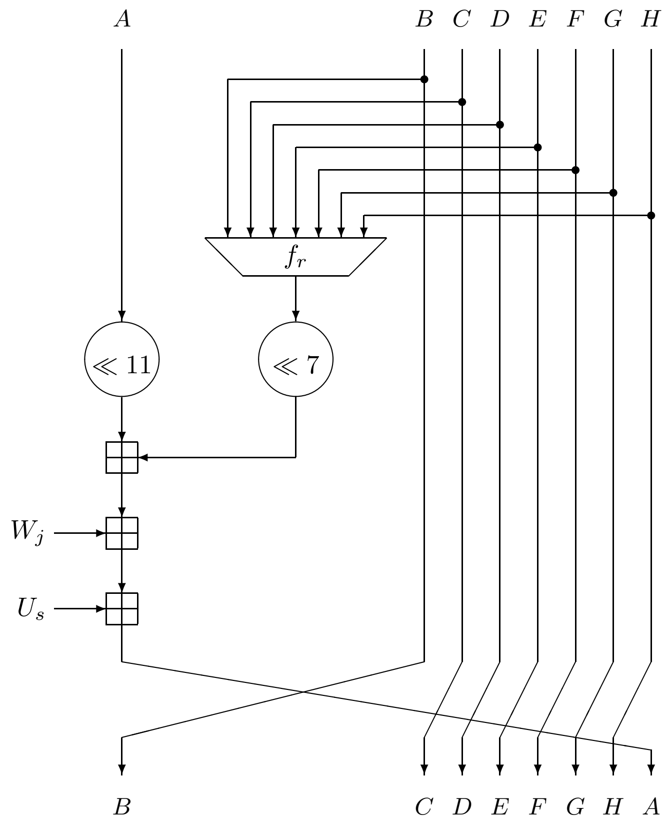

Figure 4.5: Step operation for HAVAL.

following properties:[6]

– they are 0-1 balanced;

– they are highly non-linear;

– they satisfy the strict avalanche criterion;

– the different functions are linearly inequivalent in structure;

– the different functions are mutually output-uncorrelated.

[6]See [109] for more information on the cryptographic properties of Boolean functions.

This is very different from the MD4 algorithm which uses simple Boolean functions (selection, majority, exor) on three inputs. These simple functions satisfy only the property of being 0-1 balanced. The exor function used in round 3 of the MD4 compression function is even linear.

Note that the step operation of HAVAL is reversible. For example, the previous value of the A register can be computed by

$$A_{prev} = (A_{new} - W_j - U_s - (f_r(B,C,D,E,F,G,H))^{\ggg 7})^{\lll 11} .$$

However after execution of all 96, 128 or 160 steps, the compression function uses a feed-forward operation which adds the initial values of the registers (the values at the start of the compression function) to their final values (obtained after 96, 128 or 160 steps). The result is the chaining variable output from the compression function. Because of the feed-forward at the end, the compression function cannot be inverted.

When the compression function has been used for the last time (on the last block of the padded message), the 256-bit output is sent to an output transformation and the result of this transformation is the hash result of the message. For 256-bit hash results the output transformation is simply the identity transformation. For other output lengths (128, 160, 192 or 224 bits) it applies a folding technique. For a detailed description of HAVAL, and for an explication of the notations used in our analysis below, see Appendix C.

4.5.2 Analysis of three-round HAVAL

In this section we describe a collision attack on the HAVAL algorithm with three rounds in the compression function (that is the minimum number allowed by the algorithm specification). This attack has been published in [123]. The overall strategy is similar to the strategy used for the attack on MD4 (see Sect. 4.4.3). First we generate an *inner almost-collision* for rounds 1/2 of the compression function (this means that there is a small difference after round 2). We use message blocks that differ only in the word W_{28}. The small difference after round 2 must be chosen in such a way that there is a significant probability that it propagates to a difference in round 3 which is compensated when the word W_{28} is used for the third time. We combine the technique of constructing and solving a system of difference equations with the technique of differential cryptanalysis.

Outline of the attack

The goal of our attack is to find two distinct message blocks $\{W_j\}$ and $\{W'_j\}$ $(0 \leq j \leq 31)$ which are mapped by the compression function to the same output value, where the computation for the two message blocks starts from the same 256-bit input chaining value $(A_0, \ldots, H_0)$. We find such a collision for two message blocks with a small difference in only one of the words:

$$\begin{aligned} W'_{28} &= W_{28} + 1\,, \\ W'_j &= W_j \ (j \neq 28)\,. \end{aligned}$$

The three-round compression function updates each of the eight registers of the chaining variable twelve times (four times in a round). These values are denoted by $(A_i, \ldots, H_i)$ with $1 \leq i \leq 12$. The output of the compression function is then computed with a feed-forward as $(A_0 + A_{12}, \ldots, H_0 + H_{12})$.

During the execution of the compression function some intermediate values for the registers will be different for the message blocks $\{W_j\}$ and $\{W'_j\}$. We define the difference after step j as

$$\Delta_j = (A - A', B - B', C - C', D - D', E - E', F - F', G - G', H - H')\,,$$

where $(A, \ldots, H)$ are the values of the registers at this point for message block $\{W_j\}$, and similarly $(A', \ldots, H')$ for $\{W'_j\}$. Note that this difference is defined with respect to the modular addition operation.

It can be seen that the word W_{28}, respectively W'_{28} (which contains the only difference between the two message blocks), is applied three times, in steps 29, 38 and 69. Before step 29 all values of the registers are equal for the two message blocks; a collision will be obtained if the values of all registers are equal again after execution of step 69 (hereafter all message words that are used are the same for both message blocks so no new differences will occur in any computed register value). In order to give our attack a chance of success we need to control the differences in registers between step 29 and step 69 very carefully.

In phase I of the attack we concentrate on the first two rounds of the compression function, more specifically the part between steps 29 and 38. The first use of the word W_{28}, respectively W'_{28}, is in step 29 (round 1 of the compression function) where a new value is computed for the E register. This means that the first computed register value which is not equal for the two message blocks, is the value E_4, respectively E'_4. At this point we have the following correspondence between the registers for the two message blocks:

$$\begin{array}{rrrr} A_4 = A'_4 & B_4 = B'_4 & C_4 = C'_4 & D_4 = D'_4 \\ E_4 = E'_4 + (W_{28} - W'_{28}) & F_3 = F'_3 & G_3 = G'_3 & H_3 = H'_3\ . \end{array}$$

So the difference after step 29 is:

$$\Delta_{29} = (0, 0, 0, 0, W_{28} - W'_{28}, 0, 0, 0) = (0, 0, 0, 0, -1, 0, 0, 0)\,.$$

The next use of W_{28}, respectively W'_{28}, occurs in step 38 (round 2 of the compression function) where a new value is computed for the F register. We will generate an inner almost-collision with the following correspondence between register values at this point:

$$\begin{array}{rrrr} A_5 = A'_5 & B_5 = B'_5 & C_5 = C'_5 & D_5 = D'_5 \\ E_5 = E'_5 + 1^{\lll 12} & F_5 = F'_5 & G_4 = G'_4 & H_4 = H'_4 \ . \end{array}$$

So we want only a small difference in register E after the execution of step 38. That is,

$$\Delta_{38} = (0, 0, 0, 0, 1^{\lll 12}, 0, 0, 0)\,.$$

In phase II of the attack we concentrate on the last two rounds of the compression function, more specifically the part between steps 38 and 69. As seen above we have only a small difference in the E register after step 38. We are now ready to perform a differential cryptanalysis on the following steps. The last occasion where the word W_{28}, respectively W'_{28}, is used is in step 69 (in round 3 of the compression function). For $39 \le j \le 68$ we require that $\Delta_j = (0, 0, 0, 0, E - E', 0, 0, 0)$. That is, we require that the difference in the E register after step 38 does not spread to any of the other registers. Furthermore, the difference in the E register after step 38 has been chosen in such a way that the use of W_{28}, respectively W'_{28}, in step 69 compensates the difference in the E register at that point. That means $\Delta_{69} = (0, 0, 0, 0, 0, 0, 0, 0)$. This will also result in a collision in the output of the compression function.

In order to generate an inner almost-collision (phase I of the attack) we need to fix a few of the words W_j in the message blocks. We can randomly choose the remaining words in phase II and see if the differential attack works. We found that the success probability of our differential attack is about 2^{-29}, so a collision can be found by randomly choosing the remaining words W_j and computing the difference after step 69 (which should be zero for all registers). This will succeed after, on average, 2^{29} trials. There is one more complication: matching the specified initial chaining variable $(A_0, \ldots, H_0)$. This can be done with a small modification of the differential attack.

Phase I: Finding an inner almost-collision for rounds 1/2

We first analyse the part of the compression function between step 29 and step 38. As noted above we require that $\Delta_{29} = (0, 0, 0, 0, -1, 0, 0, 0)$ and that $\Delta_{38} = (0, 0, 0, 0, 1^{\lll 12}, 0, 0, 0)$. Table 4.7 below shows the difference propagation used in this phase of the attack. We use the notations $\Delta A = A - A', \Delta B = B - B'$, etc. where (A, B, C, D, E, F, G, H) are the values of the registers at this point for message block $\{W_j\}$, and similarly $(A', B', C', D', E', F', G', H')$ for $\{W'_j\}$. The shown difference values are the values *after* the corresponding step has been executed. We also list the message word applied in each step. Entries in bold face show which register has been updated in a particular step.

In step 29 a difference in the E register is introduced: $E_4 - E'_4 = W_{28} - W'_{28} = -1$. We let this difference spread to the F register in step 30, more specifically $F_4 - F'_4 = 1$. From step 31 up to step 36 we require that the differences in the E and F registers do not spread to any of the other registers: $G_4 - G'_4 = H_4 - H'_4 = A_5 - A'_5 = B_5 - B'_5 = C_5 - C'_5 = D_5 - D'_5 = 0$. Then, in step 37, we need an interaction of the differences in the E and F registers, in such a way that the right difference $E_5 - E'_5 = 1^{\lll 12}$ is obtained. Finally, the difference in the F register has to disappear in step 38 where the word W_{28}, respectively W'_{28}, is used again: $F_5 - F'_5 = 0$.

Table 4.7: Overview of the difference propagation through the registers for rounds 1 and 2 of HAVAL (for an inner almost-collision).

Step	ΔA	ΔB	ΔC	ΔD	ΔE	ΔF	ΔG	ΔH	word
29	0	0	0	0	$-\mathbf{1}$	0	0	0	$W_{28}(+1)$
30	0	0	0	0	-1	**1**	0	0	W_{29}
31	0	0	0	0	-1	1	**0**	0	W_{30}
32	0	0	0	0	-1	1	0	**0**	W_{31}
33	**0**	0	0	0	-1	1	0	0	W_5
34	0	**0**	0	0	-1	1	0	0	W_{14}
35	0	0	**0**	0	-1	1	0	0	W_{26}
36	0	0	0	**0**	-1	1	0	0	W_{18}
37	0	0	0	0	$\mathbf{1^{\lll 12}}$	1	0	0	W_{11}
38	0	0	0	0	$1^{\lll 12}$	**0**	0	0	$W_{28}(+1)$

For each step in turn, we now look at the difference which is obtained after computing the new register value for the message blocks $\{W_j\}$ and $\{W'_j\}$. To simplify the analysis we first make the following specific choices:

$$E_4 = -1 \ , \ E'_4 = 0 \ , \ F_4 = 0 \ , \ F'_4 = -1 \ .$$

Note that these choices agree with the differences $E_4 - E'_4 = -1$ and $F_4 - F'_4 = 1$. The values 0 and -1 (modulo 2^{32}) correspond to 32-bit quantities where all the bits are set equal to 0 or 1 respectively.

Step 29. In this step we have a difference in the applied message word W_{28}, respectively W'_{28}. From the definition of the step operation (see Appendix C) and using $E'_3 = E_3, F'_3 = F_3, G'_3 = G_3, H'_3 = H_3, A'_4 = A_4, B'_4 = B_4, C'_4 = C_4, D'_4 = D_4$, it follows that

$$E_4 - E'_4 = W_{28} - W'_{28} = -1 \, .$$

Step 30. From the definition of the step operation it follows that

$$F_4 - F'_4 \quad = \quad (f_1(G_3, H_3, A_4, B_4, C_4, D_4, E_4))^{\ggg 7} -$$

$$(f_1(G_3, H_3, A_4, B_4, C_4, D_4, E'_4))^{\gg 7}\,.$$

If we now use the definition of the non-linear function f_1 (see Appendix C) and insert the values of E_4, E'_4, F_4, F'_4 we can rewrite this as

$$1 = (G_3 \oplus B_4C_4 \oplus H_3D_4 \oplus A_4C_4 \oplus A_4)^{\gg 7} - (B_4C_4 \oplus H_3D_4 \oplus A_4C_4 \oplus A_4)^{\gg 7}\,. \quad (4.23)$$

Step 31. We require that $G_4 - G'_4 = 0$. That means,

$$(f_1(H_3, A_4, B_4, C_4, D_4, E_4, F_4))^{\gg 7} - (f_1(H_3, A_4, B_4, C_4, D_4, E'_4, F'_4))^{\gg 7} = 0\,.$$

Using the definition of f_1 and inserting the values of E_4, E'_4, F_4, F'_4 we get

$$(A_4 \oplus C_4D_4 \oplus B_4D_4 \oplus B_4)^{\gg 7} = (H_3 \oplus C_4D_4 \oplus B_4D_4 \oplus B_4)^{\gg 7}\,.$$

This equation is satisfied when

$$A_4 = H_3\,. \quad (4.24)$$

Step 32. We require that $H_4 - H'_4 = 0$. That means,

$$(f_1(A_4, B_4, C_4, D_4, E_4, F_4, G_4))^{\gg 7} - (f_1(A_4, B_4, C_4, D_4, E'_4, F'_4, G_4))^{\gg 7} = 0\,.$$

In the same manner as above we can derive the following equation:

$$D_4 \oplus C_4 = B_4\,. \quad (4.25)$$

Step 33. We require that $A_5 - A'_5 = 0$. Note that this is the first step of round 2 of the compression function so the non-linear function f_2 is used (see Appendix C for the definition of the function f_2):

$$(f_2(B_4, C_4, D_4, E_4, F_4, G_4, H_4))^{\gg 7} - (f_2(B_4, C_4, D_4, E'_4, F'_4, G_4, H_4))^{\gg 7} = 0\,.$$

We obtain the equation

$$C_4H_4 \oplus C_4 = C_4G_4 \oplus H_4\,. \quad (4.26)$$

Step 34. We require that $B_5 - B'_5 = 0$. That means

$$(f_2(C_4, D_4, E_4, F_4, G_4, H_4, A_5))^{\gg 7} - (f_2(C_4, D_4, E'_4, F'_4, G_4, H_4, A_5))^{\gg 7} = 0\,,$$

which is satisfied when

$$D_4A_5 \oplus H_4 = 0\,. \quad (4.27)$$

Step 35. We require that $C_5 - C'_5 = 0$. That means

$$(f_2(D_4, E_4, F_4, G_4, H_4, A_5, B_5))^{\gg 7} - (f_2(D_4, E'_4, F'_4, G_4, H_4, A_5, B_5))^{\gg 7} = 0\,,$$

which is satisfied when

$$G_4B_5 \oplus H_4A_5 \oplus G_4 \oplus D_4 = 0\,. \tag{4.28}$$

Step 36. We require that $D_5 - D'_5 = 0$. That means

$$(f_2(E_4, F_4, G_4, H_4, A_5, B_5, C_5))^{\gg 7} - (f_2(E'_4, F'_4, G_4, H_4, A_5, B_5, C_5))^{\gg 7} = 0\,,$$

which is satisfied when

$$H_4C_5 \oplus A_5B_5 \oplus H_4 \oplus G_4 = -1\,. \tag{4.29}$$

Step 37. In this step we need to obtain the right difference $E_5 - E'_5 = 1^{\ll 12}$. From the definition of the step operation it follows that

$$\begin{aligned} E_5 - E'_5 \quad = \quad & E_4^{\gg 11} - E_4^{'\gg 11} + (f_2(F_4, G_4, H_4, A_5, B_5, C_5, D_5))^{\gg 7} - \\ & (f_2(F'_4, G_4, H_4, A_5, B_5, C_5, D_5))^{\gg 7}\,. \end{aligned}$$

Using the definition of f_2 and inserting the values of E_4, E'_4, F_4, F'_4 we get

$$\begin{aligned} 1^{\ll 12} \quad = \quad & -1 + (G_4A_5D_5 \oplus G_4B_5C_5 \oplus G_4A_5 \oplus A_5C_5 \oplus G_4H_4 \oplus B_5D_5 \oplus \\ & B_5C_5)^{\gg 7} - (G_4A_5D_5 \oplus G_4B_5C_5 \oplus G_4A_5 \oplus A_5C_5 \oplus G_4H_4 \oplus \\ & B_5D_5 \oplus B_5C_5 \oplus G_4 \oplus -1)^{\gg 7}\,. \end{aligned} \tag{4.30}$$

Step 38. Finally, in this step we require that the difference in the F register disappears: $F_5 - F'_5 = 0$. From the definition of the step operation we see that

$$\begin{aligned} F_5 - F'_5 \quad = \quad & F_4^{\gg 11} - F_4^{'\gg 11} + W_{28} - W'_{28} + \\ & (f_2(G_4, H_4, A_5, B_5, C_5, D_5, E_5))^{\gg 7} - \\ & (f_2(G_4, H_4, A_5, B_5, C_5, D_5, E'_5))^{\gg 7}\,. \end{aligned}$$

Because $F_4^{\gg 11} - F_4^{'\gg 11} = 1$ and $W_{28} - W'_{28} = -1$, the requirement $F_5 - F'_5 = 0$ leads to the equation

$$(f_2(G_4, H_4, A_5, B_5, C_5, D_5, E_5))^{\gg 7} - (f_2(G_4, H_4, A_5, B_5, C_5, D_5, E'_5))^{\gg 7} = 0\,,$$

which is satisfied when

$$B_5H_4 \oplus C_5 = 0\,. \tag{4.31}$$

Solution for the system of equations

The equations (4.23) to (4.31) which we derived above, need to be satisfied in order to obtain an inner almost-collision. Therefore, we need a solution for an underdetermined system of nine equations in twelve variables. It can be seen that the following set of register values constitutes such a solution:

$$\begin{array}{llllll} G_3 = 1^{\lll 7} & H_3 = 0 & A_4 = 0 & B_4 = 0 & C_4 = 0 & D_4 = 0 \\ G_4 = 0 & H_4 = 0 & A_5 = -1 & B_5 = -1 & C_5 = 0 & D_5 = 1^{\lll 18}\ . \end{array}$$

Note that $G_3 = 1^{\lll 7}$ is a solution to $G_3^{\ggg 7} = 1$, and $D_5 = 1^{\lll 18}$ is a solution to $-1 + D_5^{\ggg 7} - (D_5 \oplus -1)^{\ggg 7} = 1^{\lll 12}$. These two equations are derived from (4.23) and (4.30) respectively by inserting the values given for the other variables.

As previously seen we also have $E_4 = -1$ and $F_4 = 0$. Fixing these 14 register values, in order to generate an inner almost-collision, also determines the values of some words of the message block $\{W_j\}$, in particular:[7]

$$\begin{array}{rcl} W_{30} & = & G_4 - G_3^{\ggg 11} - (f(H_3, A_4, B_4, C_4, D_4, E_4, F_4))^{\ggg 7}\ , \\ W_{31} & = & H_4 - H_3^{\ggg 11} - (f(A_4, B_4, C_4, D_4, E_4, F_4, G_4))^{\ggg 7}\ , \\ W_5 & = & A_5 - A_4^{\ggg 11} - (g(B_4, C_4, D_4, E_4, F_4, G_4, H_4))^{\ggg 7} - U_0\ , \\ W_{14} & = & B_5 - B_4^{\ggg 11} - (g(C_4, D_4, E_4, F_4, G_4, H_4, A_5))^{\ggg 7} - U_1\ , \\ W_{26} & = & C_5 - C_4^{\ggg 11} - (g(D_4, E_4, F_4, G_4, H_4, A_5, B_5))^{\ggg 7} - U_2\ , \\ W_{18} & = & D_5 - D_4^{\ggg 11} - (g(E_4, F_4, G_4, H_4, A_5, B_5, C_5))^{\ggg 7} - U_3\ . \end{array}$$

Note that we get the same values $W_j' = W_j$ when we use the alternative register values $G_3', H_3', A_4', B_4', C_4', D_4', E_4', F_4', G_4', H_4', A_5', B_5', C_5', D_5'$ in the computations (only E_4' and F_4' are different). Six words of the message blocks $\{W_j\}$ and $\{W_j'\}$ are now determined. We still have a free choice for the remaining 26 words of these message blocks in phase II of the attack, as described below.

Other solutions for the system of equations

As an alternative for the solution given above, different solutions for the system of equations (4.23) to (4.31) can be found. In general, for an arbitrary choice of two 32-bit values Q_1 and Q_2, the following set of register values is a solution for the system of equations (and leads to an inner almost-collision):

[7] U_0, U_1, U_2, U_3 denote the additive constants used in steps 33–36 of round 2.

$$\begin{array}{ll}
G_3 = (1 + Q_1^{\ggg 7})^{\lll 7} \oplus Q_1 & G_4 = (Q_2^{\ggg 7} - 1^{\lll 12} - 1)^{\lll 7} \oplus Q_2 \oplus -1 \\
H_3 = Q_1 & H_4 = 0 \\
A_4 = Q_1 & A_5 = (Q_2^{\ggg 7} - 1^{\lll 12} - 1)^{\lll 7} \oplus Q_2 \\
B_4 = 0 & B_5 = -1 \\
C_4 = 0 & C_5 = 0 \\
D_4 = 0 & D_5 = Q_2 \ .
\end{array}$$

Note that for $Q_1 = 0$ and $Q_2 = 1^{\lll 18}$ this reduces to the solution given earlier. For any choice of Q_1 and Q_2 a specific set of register values is obtained, and hence also a specific set of message words W_{30}, W_{31}, W_5, W_{14}, W_{26}, and W_{18}. However, in those cases where bit 12 of Q_2 is equal to 1 (starting the count from the least significant bit which is called bit 0), the differential attack in phase II does not work. Solutions with bit 12 of Q_2 equal to 0 (leading to a successful differential attack), are called *admissable* inner almost-collisions. 2^{63} different admissable inner almost-collisions can be generated, but only one of them is needed for phase II of the attack. The complexity of finding an admissable inner almost-collision is negligible compared to the complexity of the differential attack which we describe below.

Phase II: Differential attack on rounds 2/3

In this phase of the attack we consider the part of the compression function between steps 38 and 69. We have an input difference $\Delta_{38} = (0, 0, 0, 0, 1^{\lll 12}, 0, 0, 0)$ (from phase I of the attack) and we require that $\Delta_{69} = (0, 0, 0, 0, 0, 0, 0, 0)$. Table 4.8 below shows the difference propagation for this phase of the attack. For the E register we have the following differences: $E_5 - E'_5 = 1^{\lll 12}, E_6 - E'_6 = 1^{\lll 1}, E_7 - E'_7 = 1^{\lll 22}, E_8 - E'_8 = 1^{\lll 11}, E_9 - E'_9 = 0$. For the other registers all differences must be zero.

There are two cases for the computation of the probability of a difference propagation through a step. The content of the E register is updated in steps 45, 53, 61 and 69. In step 45 for example we compute

$$\begin{aligned}
E_6 &= E_5^{\ggg 11} + (f_2(F_5, G_5, H_5, A_6, B_6, C_6, D_6))^{\ggg 7} + W_1 + U_{12} \, , \\
E'_6 &= E_5^{'\ggg 11} + (f_2(F_5, G_5, H_5, A_6, B_6, C_6, D_6))^{\ggg 7} + W_1 + U_{12} \, .
\end{aligned}$$

Hence, we see that the difference

$$E_6 - E'_6 = E_5^{\ggg 11} - E_5^{'\ggg 11} \, ,$$

and we require $E_5 - E'_5 = 1^{\lll 12}$ and $E_6 - E'_6 = 1^{\lll 1}$ (the difference gets rotated by 11 bit positions to the right). This happens with a probability which is close to one. In the other steps we require that the difference in the E register does

Table 4.8: Overview of the difference propagation through the registers for rounds 2 and 3 of HAVAL (differential attack).

Step	ΔA	ΔB	ΔC	ΔD	ΔE	ΔF	ΔG	ΔH	word
39	0	0	0	0	$1^{\lll 12}$	0	**0**	0	W_7
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
44	0	0	0	**0**	$1^{\lll 12}$	0	0	0	W_{22}
45	0	0	0	0	$\mathbf{1^{\lll 1}}$	0	0	0	W_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
52	0	0	0	**0**	$1^{\lll 1}$	0	0	0	W_9
53	0	0	0	0	$\mathbf{1^{\lll 22}}$	0	0	0	W_{17}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
60	0	0	0	**0**	$1^{\lll 22}$	0	0	0	W_{13}
61	0	0	0	0	$\mathbf{1^{\lll 11}}$	0	0	0	W_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
68	0	0	0	**0**	$1^{\lll 11}$	0	0	0	W_{20}
69	0	0	0	0	**0**	0	0	0	$W_{28}(+1)$

not spread to a different register. In step 46 for example we compute

$$F_6 = F_5^{\ggg 11} + (f_2(G_5, H_5, A_6, B_6, C_6, D_6, E_6))^{\ggg 7} + W_1 + U_{13}\,,$$
$$F_6' = F_5^{\ggg 11} + (f_2(G_5, H_5, A_6, B_6, C_6, D_6, E_6'))^{\ggg 7} + W_1 + U_{13}\,.$$

Here the difference $F_6 - F_6' =$

$$(f_2(G_5, H_5, A_6, B_6, C_6, D_6, E_6))^{\ggg 7} - (f_2(G_5, H_5, A_6, B_6, C_6, D_6, E_6'))^{\ggg 7}\,,$$

and we require that $F_6 - F_6' = 0$ which is equivalent to

$$f_2(G_5, H_5, A_6, B_6, C_6, D_6, E_6) = f_2(G_5, H_5, A_6, B_6, C_6, D_6, E_6')\,.$$

Using the definition of f_2 we can derive the following condition:

$$E_6 B_6 H_5 \oplus E_6 C_6 = E_6' B_6 H_5 \oplus E_6' C_6\,,$$

which is satisfied when $B_6 H_5 \oplus C_6 = 0$ at those bit positions where E_6 is different from E_6'. Because $E_6 = E_6' + 1^{\lll 1}$ and the carry associated with the modular addition operation, this happens with a probability of about 1/3 ($\frac{1}{2^2} + \frac{1}{4^2} + \frac{1}{8^2} + \cdots \approx \frac{1}{3}$).

By combining the probabilities for all steps we can estimate the global probability for the propagation from step 38 up to step 69 as $p_{69}^{38} \approx (1/3)^{27} \approx 2^{-42.8}$. The real probability is much higher however. This is partly because of the values of the registers at the start of the differential attack.[8] Furthermore, the probabilities for consecutive steps strongly depend on each other (because every step changes the value of only one out of eight registers). If we consider a sequence of eight steps, experiments show that the probability is about 2^{-9} which is better than $(1/3)^7 \approx 2^{-11.1}$. For the complete propagation from step 38 up to step 69 we found the estimation

$$p^{38}_{69} \approx 2^{-29}\,.$$

The differential attack can be performed as follows. In phase I of the attack message words $W_{30}, W_{31}, W_5, W_{14}, W_{26}$, and W_{18} were determined in order to get the right input difference Δ_{38}. We can now randomly choose the remaining 26 words and compute forwards to step 69, starting from the known register values E_4, F_4, G_4, H_4, A_5, B_5, C_5, D_5 (or E'_4, F'_4 for the second message block). If the difference after step 69 is equal to zero for all registers then we have a collision and this happens on average after 2^{29} trials. There is however one more complication which we describe below.

Matching the initial value

When all message words W_j are determined we can also compute backwards in round 1 of the compression function, starting from the known register values G_3, H_3, A_4, B_4, C_4, D_4, E_4, F_4. This is done by inverting the step operations. For example, inverting step 30 gives us

$$F_3 = (F_4 - (f_1(G_3, H_3, A_4, B_4, C_4, D_4, E_4))^{\gg 7} - W_{29})^{\ll 11}\,.$$

In that way we finally obtain the register values $(A_0, \dots, H_0)$. This means that we have obtained a collision for the compression function starting from a random initial chaining variable.

We can easily extend our attack so that we find collisions for the compression function starting from a specified initial chaining variable (and consequently also for the hash function, see Sect. 3.3). First note that there is one sequence of eight message words, which are applied in consecutive steps in round 1 of the compression function, and none of which have been determined in phase I of the attack (for obtaining an inner almost-collision). This sequence of message words is the sequence of $W_6, W_7, \dots, W_{13}$ (which are used in steps 7 up to 14). These words will be used to match the initial chaining variable.

In our differential attack we randomly choose values for 18 message words (as before but excluding the eight words needed to match the initial value). We also know the fixed values for the words $W_{30}, W_{31}, W_5, W_{14}, W_{26}, W_{18}$ (determined in

[8]Related to this, the reason that not all inner almost-collisions lead to a successful differential attack is that in some cases the values of the registers are not suitable at the start of the differential attack.

phase I of the attack). Now we compute backwards in round 1 of the compression function down to the (inverted) step 15 where W_{14} is applied. In this manner we derive the register values (G_1, H_1, A_2, B_2, C_2, D_2, E_2, F_2). Next we compute forwards starting from the specified $(A_0, \ldots, H_0)$ and up to step 6 where W_5 is applied. This gives us the register values (G_0, H_0, A_1, B_1, C_1, D_1, E_1, F_1) and now we can compute the required values for the message words $W_6, W_7, \ldots, W_{13}$. For example,

$$W_6 = G_1 - G_0^{\ggg 11} - (f_1(H_0, A_1, B_1, C_1, D_1, E_1, F_1))^{\ggg 7} .$$

After we have matched the specified initial values for all registers (and thereby determined the values for all 32 message words W_j) we check the differential attack between steps 39 and 69 as before and repeat the procedure until a collision has been found. On average we succeed after 2^{29} trials, where a trial can be abandoned as soon as the difference propagation in a register is not correct.

Complexity and flexibility of the attack

Our attack on three-round HAVAL has a complexity of about 2^{29} computations of the compression function (this complexity is determined by the probability for the differential attack on the last two rounds. A programme that implements the attack runs on average in less than one hour on an Athlon 600 MHz processor. The number of collisions which can be generated, at least in theory, starting from a given admissable inner almost-collision, is equal to 2^{547}, since we can freely choose 18 words (that is a maximum of 2^{576} trials), and the success probability is about 2^{-29}. Because there are 2^{63} different admissable inner almost-collisions to start from, the total number of collisions which can be generated by our attack is equal to $2^{547+63} = 2^{610}$.

Example collision for three-round HAVAL

We give an example of two message blocks that are hashed by the compression function of three-round HAVAL to the same output value. For both message blocks the computation starts from the initial value defined in the algorithm specification (see Appendix C):

$$\begin{array}{llll} A_0 = \texttt{ec4e6c89}_x & B_0 = \texttt{082efa98}_x & C_0 = \texttt{299f31d0}_x & D_0 = \texttt{a4093822}_x \\ E_0 = \texttt{03707344}_x & F_0 = \texttt{13198a2e}_x & G_0 = \texttt{85a308d3}_x & H_0 = \texttt{243f6a88}_x \ . \end{array}$$

The first message block is:

$$\begin{array}{ll}
W_0 = \texttt{94c0875e}_x & W_1 = \texttt{dd25f63e}_x \\
W_2 = \texttt{f5d09361}_x & W_3 = \texttt{b51db8b2}_x \\
W_4 = \texttt{b00c36e4}_x & W_5 = \texttt{bad7de19}_x \\
W_6 = \texttt{32a68bb5}_x & W_7 = \texttt{c5aff25d}_x \\
W_8 = \texttt{ad0dea24}_x & W_9 = \texttt{a7e1ee7c}_x \\
W_{10} = \texttt{617b92dd}_x & W_{11} = \texttt{f9da283d}_x \\
W_{12} = \texttt{b2844d83}_x & W_{13} = \texttt{b8d498eb}_x \\
W_{14} = \texttt{c72fec88}_x & W_{15} = \texttt{8f467c05}_x \\
W_{16} = \texttt{507ea2c1}_x & W_{17} = \texttt{c2d94121}_x \\
W_{18} = \texttt{cb1af394}_x & W_{19} = \texttt{036daf20}_x \\
W_{20} = \texttt{bba7fb8c}_x & W_{21} = \texttt{6daee6aa}_x \\
W_{22} = \texttt{04fc029f}_x & W_{23} = \texttt{d37c05f4}_x \\
W_{24} = \texttt{993aea13}_x & W_{25} = \texttt{3ccfab88}_x \\
W_{26} = \texttt{41ab9931}_x & W_{27} = \texttt{3c7cae0c}_x \\
W_{28} = \texttt{f704bafc}_x & W_{29} = \texttt{b60635de}_x \\
W_{30} = \texttt{f0000000}_x & W_{31} = \texttt{00000000}_x
\end{array}$$

and the second message block is determined by

$$\begin{array}{rcl}
W'_{28} & = & W_{28} + 1\,, \\
W'_j & = & W_j \;\; (0 \le j \le 31, j \ne 28)\,.
\end{array}$$

For these two message blocks, the compression function computes the following common output value (note that this computation includes the feed-forward operation at the end):

$$\begin{array}{llll}
A = \texttt{1f46758c}_x & B = \texttt{7618c292}_x & C = \texttt{e5220b62}_x & D = \texttt{77ea845b}_x \\
E = \texttt{ef9fd8de}_x & F = \texttt{41ec28af}_x & G = \texttt{5205cb85}_x & H = \texttt{260412c4}_x \;\; .
\end{array}$$

The complete hash function includes an additional application of the compression function, starting from the output value given above. For both messages the same padding block is used as message input for this final application of the compression function, therefore a collision is obtained in the final hash result:

$$\begin{array}{llll}
A = \texttt{7d476278}_x & B = \texttt{f603a907}_x & C = \texttt{6d985fef}_x & D = \texttt{4b5e66b7}_x \\
E = \texttt{b6541db5}_x & F = \texttt{16ccd71d}_x & G = \texttt{e8f9cf7c}_x & H = \texttt{141e38e2}_x \;\; .
\end{array}$$

Note that the algorithm converts this set of words into a string of 32 bytes, starting with the least significant byte of H and ending with the most significant byte of A (see Appendix C).

4.5.3 Other weaknesses in HAVAL

It may be noted that collisions are easily found for reduced two-round versions of HAVAL. Instead of an inner almost-collision, it is also possible to generate an inner collision for rounds 1 and 2 of the compression function. For example, an inner collision is obtained when we choose the register values

$$\begin{array}{llllll} G_3 = 1^{\lll 7} & H_3 = 0 & A_4 = 0 & B_4 = 0 & C_4 = 0 & D_4 = 0 \\ G_4 = 0 & H_4 = 0 & A_5 = -1 & B_5 = -1 & C_5 = Q_1 & D_5 = 0 \end{array}$$

in phase I of the attack described in the previous section (the value Q_1 can be freely chosen). The complexity of a collision attack on two rounds of HAVAL is negligible (no differential cryptanalysis is needed).

An alternative attack on reduced two-round versions of HAVAL

P. Kasselman and W. Penzhorn [69] describe a collision attack on rounds 2 and 3 of the compression function, where the difference between the two messages is in word W_{19}. In this case there are only eight steps between the two applications of the chosen message word.[9] The corresponding attack works as follows (the compression function consists of rounds 2 and 3, that is steps 33 up to 96):

1. In step 57 message word W_{19}, respectively W'_{19}, introduces a difference in register A.
2. For steps 58 to 64 it is required that the difference does not spread to any of the other registers.
3. In step 65 message word W_{19}, respectively W'_{19}, is applied again and it compensates the difference in register A.

Similar to our attack, a system of equations can be constructed and a solution is easily found. Furthermore, a special choice must be made for the difference $W_{19} - W'_{19}$. It can be shown that the following four values for this difference lead to a successful attack:

$$\texttt{55555555}_x\,,\ \texttt{55555556}_x\,,\ \texttt{aaaaaaaa}_x\,,\ \texttt{aaaaaaab}_x\,.$$

Note that a similar attack can be tried on rounds 1 and 2 of the compression function, by using a difference in message word W_{26} (this word is applied in steps 27 and 35). However, it turns out that such an attack does not work because the system of equations which is constructed, contains two conflicting equations and therefore has no solutions.

[9] For our attack there are nine steps between the two applications of message word W_{28}.

4.5.4 Conclusions

An extensive overview has been given of the design and security of the HAVAL algorithm. We have shown a practical attack for generating collisions in three-round HAVAL. This version of the algorithm should no longer be used in applications requiring a secure collision-resistant hash function.

The strategy for our attack is quite similar to the strategy used for the attack on MD4 (see Sect. 4.4.3). Surprisingly, our result shows that the use of highly non-linear functions, which is the main focus of the design of HAVAL, does not result in a hash function which is significantly stronger compared to MD4 (note that the compression function of MD4 also has three rounds but only 16 steps in each round). For MD4, the linear exor function which is used in round 3 of its compression function increases the complexity of the differential attack because this function makes it necessary to work with a difference in two of the registers. For HAVAL on the other hand, we are able to use a differential propagation characteristic with a difference in only one of the registers. Because of this the probability of the characteristic is only slightly lower than the probability of the characteristic used for MD4 (2^{-29} compared to 2^{-22}) even though the characteristic extends over many more steps (31 steps compared to 16).

We conclude that the structure of HAVAL is certainly not stronger than the structure of MD4 (for the same number of rounds). Note also that the security level of HAVAL should be much higher than the security level of MD4 because the hash results which are generated, are up to 256 bits long, compared to 128 bits for MD4. It remains an open problem to extend our techniques in order to find weaknesses in the four-round or five-round versions of HAVAL.

4.6 The RIPEMD Family

The RIPEMD hash function was designed in 1992 in the framework of the European RIPE project [113]. The design of RIPEMD is based on MD4; its compression function consists essentially of two parallel versions of the MD4 compression function. In 1996 Dobbertin found a collision attack on versions of RIPEMD reduced to two rounds out of three. This prompted a redesign resulting in the hash functions RIPEMD-128 and RIPEMD-160 (proposed by Dobbertin *et al.*). The RIPEMD-160 algorithm has the advantage of longer hash values (160 bits instead of 128). In this section we explain the design of this family of hash functions, and discuss the weaknesses which have been found in reduced versions of the original RIPEMD.

4.6.1 Description of the RIPEMD algorithm

The RIPEMD algorithm [113] computes hash values of 128 bits, for messages of arbitrary length. The algorithm has a word length of 32 bits, therefore the chaining variable is divided into four registers (A, B, C, D) of 32 bits each. The compression function works on message blocks of 512 bits, a block is divided into sixteen 32-bit words denoted by W_j for $j = 0, 1, \ldots, 15$.

Internally, the compression function consists of two trails that are executed in parallel. Each of these trails is a slightly modified version of the compression function of MD4 (without the feed-forward operation), consisting of 48 sequential steps divided into three rounds (see Sect. 4.4.1 for a description of MD4). The two trails of RIPEMD differ from MD4 as follows:

– The order in which the message words are applied in round 2 and round 3 is different.

– The rotation constants are changed.

– One trail uses the same additive constants as MD4 but the other trail uses different additive constants. This is the only difference between the two trails of RIPEMD.

In Fig. 4.6 below we give an outline of the compression function of RIPEMD. The chaining variable (A, B, C, D) serves as starting value for both the left and right trails. These two trails are executed in parallel and update the registers through 48 sequential steps (three rounds). At the end the two trails are combined with each other and with the chaining variable input (a feed-forward operation intended to make the compression function uninvertible). The result is the chaining variable output from the compression function. More specifically, if we denote the chaining variable input with (A_0, B_0, C_0, D_0), the output from the left trail with (A_L, B_L, C_L, D_L) and the output from the right trail with (A_R, B_R, C_R, D_R), then the chaining variable output is computed as:

$$(A, B, C, D) = (B_0 + C_L + D_R, C_0 + D_L + A_R, D_0 + A_L + B_R, A_0 + B_L + C_R) .$$

For a detailed description of RIPEMD we refer to [113].

4.6.2 Analysis of reduced versions of RIPEMD

We give a short overview of attacks found for reduced versions of RIPEMD. First we note that the structure of RIPEMD is very similar to the structure of Extended-MD4 (see Sect. 4.4.4). Both use two parallel instances of MD4. The main difference is that RIPEMD combines the two trails after processing each message block, whereas Extended-MD4 only exchanges the values of the A

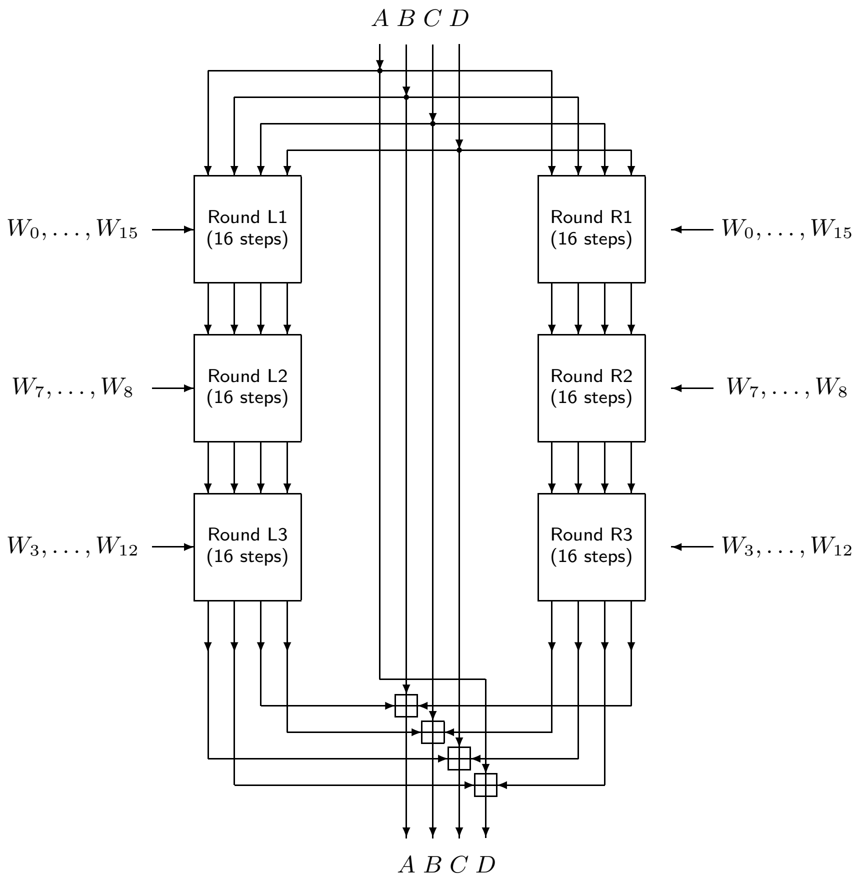

Figure 4.6: Outline of the RIPEMD compression function.

registers between the two trails. Another difference is that Extended-MD4 uses the same additive constant in round 1 of the two trails, but RIPEMD does not. Both algorithms share an important weakness: the message words are applied in the same order for both trails. This weakness is exploited in the second attack described below.

Collisions for the separate trails of RIPEMD

In [34] C. Debaert and H. Gilbert analyse the two trails of RIPEMD separately. RIPEMD^L and RIPEMD^R are defined as hash functions with a compression function consisting of only the left or right trail respectively, followed by a feed-forward. Note that the trails of RIPEMD are slightly modified versions of the 48 steps (three rounds) of the MD4 compression function, with some changes in the rotation (and additive) constants and in the order of the message words. It is shown that these changes introduce some additional constraints for the attack, but collisions can still be found with a time complexity comparable to the attack on MD4 (see Sect. 4.4.3). These observations provide arguments that the collision attack on MD4 is not due to a particularly weak selection of parameters (constants, order of message words). The selection made for the separate trails of RIPEMD does not result in a stronger algorithm.

Collisions for RIPEMD reduced to two rounds

In [41] Dobbertin describes a collision attack on versions of RIPEMD with two parallel trails in the compression function but only two rounds (32 steps) in each trail. A collision is found for two message blocks with a difference Δ in only one of the words:

$$\begin{aligned} W'_{13} &= W_{13} + \Delta\,, \\ W'_j &= W_j \;\; (j \neq 13)\,. \end{aligned}$$

The general outline of the attack is as follows:

1. find a simultaneous inner collision for the left and right trail (analyse step 14 up to step 19 in both trails);
2. find a backwards collision, that is a common starting value (A, B, C, D) for the two trails (analyse step 1 up to step 13 in both trails);
3. match the right initial value (meet-in-the-middle attack).

We refer to [41] for the details of this attack. Specific solutions are given for the first part of the attack, and a probabilistic algorithm which solves the second part. The second part of the attack involves the construction and solution of a system of difference equations. This is done with techniques similar to those used for the cryptanalysis of MD4 in [42]. Note that in this case the differences are defined by a comparison of the register values in the left and right trails, whereas for MD4 the differences are defined by a comparison of the register values for the two different message blocks. The first two parts of the attack allow us to find collisions for the compression function (reduced to two rounds). The third part extends the attack and finds collisions for the hash function. This is done by means of a meet-in-the-middle attack which matches the correct initial chaining variable $(A, B, C, D) = IV$, as defined in the specification of RIPEMD. The basic idea of the meet-in-the-middle attack is to search for a suitable block of message words $\{V_j\}$ $(0 \leq j \leq 15)$, such that the messages $M = \{V_j\}||\{W_j\}$ and $M' = \{V_j\}||\{W'_j\}$ produce the same hash result. The complexity of the collision attack on the reduced RIPEMD is determined by the second part and corresponds to about 2^{31} computations of the (two-round) compression function.

4.6.3 Description of RIPEMD-128 and RIPEMD-160

The RIPEMD-128 and RIPEMD-160 algorithms are strengthened variants of RIPEMD [100]. The cryptanalytic results obtained on the original RIPEMD algorithm reduced to two rounds, have been taken into account for their design. The main lesson that has been learned from RIPEMD is that the two trails of the compression function have to be more different from each other in order to prevent a cryptanalyst from attacking them both simultaneously. Therefore RIPEMD-128 and RIPEMD-160 have the following differences between the two trails of their compression function:

- as in the original RIPEMD the additive constants used in the two trails are different;
- the order in which the message words are applied is different for the two trails;
- the two trails do not apply the same Boolean function in the same round.

Furthermore, the number of rounds in the compression function has been increased (compared to the original RIPEMD): RIPEMD-128 uses four rounds (64 sequential steps) and RIPEMD-160 uses five rounds (80 sequential steps) in both trails of the compression function. It can also be noted that the Boolean functions that are used, are not the same as in RIPEMD. In particular, the majority function is no longer used.

RIPEMD-128 computes hash results of 128 bits so its chaining variable is divided into four registers (A, B, C, D) of 32 bits each. The step operation is of the same form as the step operation of RIPEMD and MD4 (see Sect. 4.4.1 for a description). The feed-forward operation and the combination of the two trails at the end of the compression function are also the same as in RIPEMD (see Sect. 4.6.1).

RIPEMD-160 on the other hand computes hash results of 160 bits, in order to increase the resistance against birthday attacks. Therefore the chaining variable is divided into five registers (A, B, C, D, E) of 32 bits each. The step operation of RIPEMD-160 is of the following form:

$$\begin{aligned} A &\leftarrow (A + f_r(B, C, D) + W_j + U_r)^{\lll v_s} + E\,, \\ C &\leftarrow C^{\lll 10}\,. \end{aligned}$$

So every step computes a new value for two of the five registers. In this case we consider a step that updates the value of the A register and also rotates the value of the C register by ten bit positions to the left. The operation that updates the A register depends on the other four registers B, C, D, E, and on the following:

– a message word W_j from the set $j = \{0, 1, \ldots, 15\}$;

– a Boolean function f_r that depends on the round and on the trail;

– an additive constant U_r that depends on the round and on the trail;

– a rotation constant v_s that depends on the step.

A graphical representation of this step operation is given in Fig. 4.7 below. Note that five consecutive steps update the values of the registers A, E, D, C, B respectively, and also rotate the values of the registers C, B, A, E, D respectively by ten bit positions to the left. After five steps the complete chaining variable has been updated. Hence, in both trails of the compression function the five registers are updated sixteen times.

The step operation of RIPEMD-160 is reversible. For example, the previous values of the A and C registers can be computed by

$$\begin{aligned} C_{prev} &= C_{new}^{\gg 10}\,, \\ A_{prev} &= (A_{new} - E)^{\gg v_s} - f_r(B, C_{prev}, D) - W_j - U_r\,. \end{aligned}$$

At the end of the compression function the two trails are combined and a feed-forward operation is used to make the compression function non-invertible. If we denote the chaining variable input with $(A_0,\ B_0,\ C_0,\ D_0,\ E_0)$, the output from the left trail with $(A_L, B_L, C_L, D_L, E_L)$ and the output from the right trail with $(A_R, B_R, C_R, D_R, E_R)$, then the chaining variable output of the compression function of RIPEMD-160 is computed as:

$$\begin{aligned} A &= B_0 + C_L + D_R\,, \\ B &= C_0 + D_L + E_R\,, \\ C &= D_0 + E_L + A_R\,, \\ D &= E_0 + A_L + B_R\,, \\ E &= A_0 + B_L + C_R\,. \end{aligned}$$

For a detailed description of RIPEMD-128 and RIPEMD-160 we refer to [100]. No weaknesses have been shown in these algorithms so far.

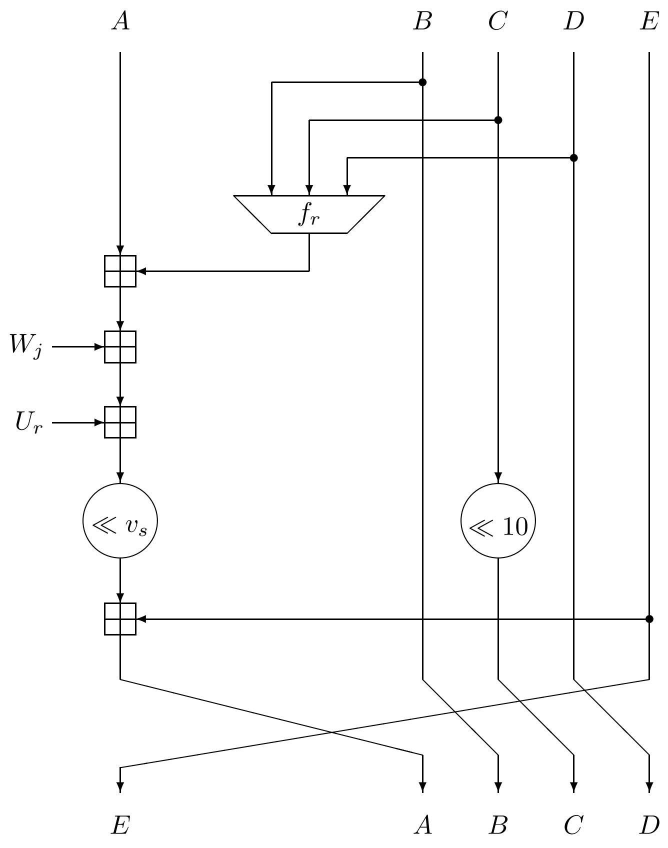

Figure 4.7: Step operation for RIPEMD-160.

4.6.4 Extension to RIPEMD-256 and RIPEMD-320

In [44] the designers of RIPEMD-128 and RIPEMD-160 also indicate how these hash functions can be extended in order to give a hash result of 256 or 320 bits respectively. The compression functions of RIPEMD-128 and RIPEMD-160 contain two parallel lines, therefore the length of the hash results can be doubled (to 256 or 320 bits respectively) by omitting the combination of the two lines at the end of the compression function (both lines now use a simple feed-forward at the end of the compression function). Note that the hash functions now require an initial value for the chaining variable of 256 bits (eight words) or 320 bits (ten words) respectively. Furthermore, some interaction between the two lines is introduced: at the end of round 1 the value of register A in the left line is swapped with the value of register A in the right line, after round 2 the same is done with the values of register B in the two lines, etc. More information can be found in [132].

It must be noted that the security level of the extensions RIPEMD-256 and RIPEMD-320 is only guaranteed to be the same as the security level of RIPEMD-128 and RIPEMD-160 respectively. The reason is that the internal structure of the extended algorithms is not essentially stronger, so if, for example, a collision-finding attack is found for RIPEMD-128, it is possible that a similar attack of comparable complexity exists for RIPEMD-256. On the other hand it must be noted that no shortcut attacks are known for RIPEMD-128 or RIPEMD-160, and the complexity of a generic birthday attack still depends on the length of the hash result ($2^{n/2}$ operations, where n is the length in bits of the hash result).

4.6.5 Conclusions

We have given an overview of the design of the algorithms from the RIPEMD family. Weaknesses have been shown in reduced versions of the original RIPEMD hash function, but these have been taken into account for the design of the strengthened variants RIPEMD-128 and RIPEMD-160. Note that an output length of 128 bits is now considered as insufficient to resist against birthday attacks, and an output length of 160 bits does not offer long term security. The 256-bit and 320-bit extended versions can be used, but these do not guarantee a higher security level.

4.7 The SHA Family

The SHA algorithm was designed by NSA and published by NIST as federal standard FIPS 180 in 1993 [49]. SHA is another hash function inspired by MD4. The design of SHA introduces a special procedure for expanding the 16-word message block input to the compression function to a block of 80 words. The design principles of SHA were not made public, however in 1994 NIST announced that a technical flaw had been found in SHA which made the algorithm less secure than originally thought. No further details were made public, but a small modification was made to the algorithm resulting in the hash function SHA-1, and the corresponding standard FIPS 180-1 [50]. In 1998 F. Chabaud and A. Joux found a theoretical collision attack for the original version of SHA (this algorithm is often called SHA-0 now). Their analysis supports the change that was made for the SHA-1 hash function.

In 2002 NIST updated its hash function standard to FIPS 180-2 [51]. This new standard specifies, besides SHA-1, three new hash functions SHA-256, SHA-384 and SHA-512. They have larger output lengths in order to offer a security level against birthday attacks, which is similar to the security level of the AES block cipher with 128-bit, 192-bit and 256-bit keys respectively (AES is specified in FIPS 197 [52]). In 2004 a change notice was added to the standard, specifying another new hash function SHA-224. It has a security level (against birthday attacks) which is similar to the security level of the Triple-DES block cipher with 112-bit key (Triple-DES is specified in FIPS 46-3 [54]).

In this section we explain the design of this family of hash functions and we discuss the collision attack on SHA-0, and a weakness found in SHA-1. We also give some remarks on the security of SHA-224, SHA-256, SHA-384 and SHA-512.

4.7.1 Description of the SHA-1 algorithm

The SHA-1 algorithm [51] computes hash results of 160 bits, for messages of any length shorter than 2^{64} bits. The algorithm has a word length of 32 bits, therefore the chaining variable is divided into five registers (A, B, C, D, E) of 32 bits each. The compression function works on message blocks of 512 bits, a block is divided into sixteen 32-bit words denoted by W_j for $j = 0, 1, \ldots, 15$.

Internally, the compression function is divided into 80 sequential steps. Another distinction that can be made is into rounds: there are four rounds, each consisting of a sequence of 20 steps. The step operation of SHA-1 is of the following form:

$$\begin{aligned} E &\leftarrow E + f_r(B, C, D) + A^{\lll 5} + W_j + U_r\,, \\ B &\leftarrow B^{\lll 30}\,. \end{aligned}$$

So every step computes a new value for two of the five registers. In this case we consider a step that updates the value of the E register and also rotates the value of the B register by 30 bit positions to the left. The operation that updates the E register depends on the other four registers A, B, C, D, and on the following:

– a message word W_j from the set $j = \{0, 1, \ldots, 79\}$;

– a Boolean function f_r that depends on the round;

– an additive constant U_r that depends on the round.

The Boolean functions that are used in the different rounds of the compression function are the selection, majority and exor functions. The exor function is used in both round 2 and round 4. The first sixteen words W_j (for $j = 0, 1, \ldots, 15$) are equal to the message block input of the compression function. The remaining sixty-four words W_j (for $j = 16, 17, \ldots, 79$) are computed by the following procedure for message expansion:

$$W_j = (W_{j-3} \oplus W_{j-8} \oplus W_{j-14} \oplus W_{j-16})^{\lll 1}\,.$$

The only difference between SHA-1 and SHA-0 (the original version of SHA) is in this equation. For SHA-0 the rotation by one bit position to the left was not included.

A graphical representation of the step operation of SHA-1 is given in Fig. 4.8 below. Note that five consecutive steps update the values of the registers E, D, C, B, A respectively, and also rotate the values of the registers B, A, E, D, C respectively by 30 bit positions to the left. After five steps the complete chaining variable has been updated. A round of the compression function consists of four sequences of five steps. Hence, each register is updated four times in every round, and sixteen times in the complete compression function (four rounds).

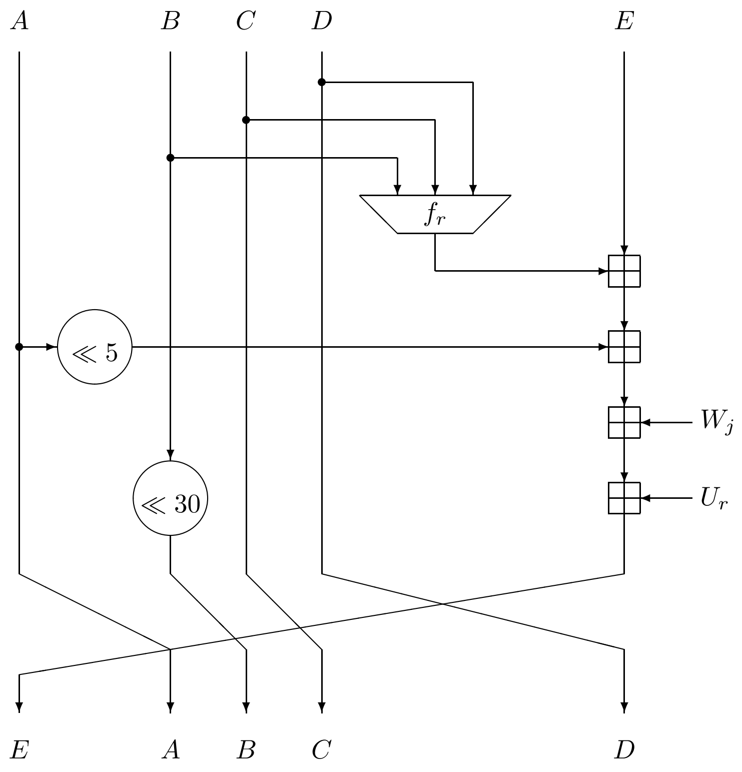

Figure 4.8: Step operation for SHA-1.

The step operation of SHA-1 is reversible. For example, the previous values of the E and B registers can be computed by

$$\begin{aligned} B_{prev} &= B_{new}^{\gg 30} , \\ E_{prev} &= E_{new} - A^{\ll 5} - f_r(B_{prev}, C, D) - W_j - U_r . \end{aligned}$$

However after execution of all 80 steps, the compression function uses a feed-forward operation which adds the initial values of the registers (the values at the start of the compression function) to their final values (obtained after 80 steps). The result is the chaining variable output from the compression function. Due to the feed-forward at the end the compression function cannot be inverted. For a detailed description of SHA-1 we refer to [51].

4.7.2 Security of SHA-1

The hash function SHA-1 is difficult to attack with techniques such as those that were introduced by Dobbertin for other hash functions of the MDx-class. Due to the procedure for message expansion it is not possible to apply a difference in only a few of the message words W_j $(j = 0, 1, \ldots, 79)$.[10]

Preneel [97] has remarked that the message expansion of the original version SHA-0 (which did not include the bit rotation) operates at bit level as a systematic linear (80,16,23) code, or a code with length 80, size 16 and minimum distance 23. This means that at least 23 words in the expanded message block have a difference. For the message expansion of SHA-1 the analysis is more complicated because the code no longer operates at bit level: it is a systematic linear (2560,512) code (the minimum distance is unknown). Below we explain the collision attack that has been demonstrated for SHA-0, and a weakness of SHA-1.

Collisions for the original version SHA-0

The original SHA algorithm, now commonly known as SHA-0, has been analysed by Chabaud and Joux [29], resulting in a theoretical collision-finding attack, which has a complexity of about 2^{61} computations of the compression function. Note that SHA-0 is similar to SHA-1; the only difference is in the procedure which expands the 16-word message block input to a block of 80 words. This procedure is defined for SHA-0 by:

$$W_j = W_{j-3} \oplus W_{j-8} \oplus W_{j-14} \oplus W_{j-16} .$$

In contrast to SHA-1 there is no rotation by one bit position to the left.

[10] On the other hand, in our attack on 3-pass HAVAL, for example, there are only three (of the 96) step operations where the applied message word has a difference (this happens once in every round of the compression function, when the word W_{28} is applied).

The general strategy of the attack is to track the propagation of local perturbations and to look for differential masks that can be added to the 80-word block with non-trivial probability of keeping the output of the compression function unchanged (a differential mask can be added to the 80-word block if it corresponds to a difference pattern in the 16-word message block before expansion). In [29] a number of simplified variants of SHA-0 are analysed first:

– A first variant, called SHI1, replaces the additions and the Boolean functions f_r by the exclusive-OR operation. Single bit errors or perturbations are introduced to the input of SHI1 and these perturbations are traced through the compression function. The perturbations are made to disappear by introducing five other bit errors or corrections. This allows a trivial attack on SHI1 via differential masking.

– A second variant, called SHI2, replaces the additions by the exclusive-OR operation, but it keeps the Boolean functions f_r of SHA-0. However we can still view the Boolean functions as acting like the exclusive-OR operation with some probability, and this leads to a probabilistic perturbation attack on SHI2. The probability for this attack is about 2^{-24}.

– A third variant, called SHI3, uses additions as in SHA-0, but it uses the exclusive-OR operation instead of the Boolean functions f_r. In this case the additions cause the perturbations to spread out due to carry propagation. However one is still able to devise a perturbation attack on SHI3 with a probability of 2^{-44}.

– Finally SHA-0 itself is analysed by taking into account the analyses of SHI2 and SHI3, and it is shown that a perturbation attack can be mounted with a probability of 2^{-61}. This corresponds to an attack with a time complexity of about 2^{61} computations of the compression function.

It is important to note that, although SHA-1 and SHA-0 are very similar, the described perturbation attack cannot be applied to SHA-1. The rotation by one bit position to the left which is added in the expansion procedure of SHA-1 means that the linear code of the expansion no longer operates at bit level: a modification of a single bit influences bits at other positions in the words as well. This makes the attack strategy of [29] ineffective and provides strong evidence that the transition from SHA-0 to SHA-1 raised the level of security.

Slide attack on SHA-1

In [118] M.-J. Saarinen describes a slide attack on SHA-1. Slide attacks [19, 20] have been proposed earlier for the cryptanalysis of block ciphers. They exploit weaknesses in ciphers which use identical or periodic round functions, and the most interesting property of these attacks is that they are in many cases independent of the number of rounds, and independent of the exact properties of the round function.

The first observation that is made for the SHA-1 hash function is that the procedure for message expansion can be slid. Consider two message blocks $\{W_j\}$ and $\{W'_j\}$ $(0 \le j \le 15)$ that are chosen as follows:

$$\begin{aligned} W'_j &= W_{j+1} \ (0 \le j \le 14)\,, \\ W'_{15} &= (W_0 \oplus W_2 \oplus W_8 \oplus W_{13})^{\lll 1}\,. \end{aligned}$$

After message expansion the following is true:

$$W'_j = W_{j+1} \ (0 \le j \le 78)\,.$$

The second observation is that for twenty steps in each round of the compression function, the Boolean function f_r and the additive constant U_r are unchanged. Hence any two consecutive steps of the compression function are similar, except for three transitions between different rounds: this happens after steps 20, 40 and 60.

Suppose now that the hashing computation for $\{W_j\}$ and $\{W'_j\}$ starts from initial chaining variables (A, B, C, D, E) and (A', B', C', D', E') respectively, which are related as follows:

$$B' = A \ , \ C' = B^{>>30} \ , \ D' = C \ , \ E' = D \ .$$

Then the purpose of the attack is to find two message blocks and corresponding initial chaining variables for which the same relation (between the registers) still holds at the end of the compression function. Such a pair of message blocks and corresponding initial chaining variables is called a *slid pair*. The general strategy for the attack is to choose suitable values for the chaining variables at the end of step 20 and step 40, and perform a meet-in-the-middle match. This procedure is repeated until the transition for step 60 is also dealt with, which happens with a probability of 2^{-32}. The attack has a time and space complexity of the order 2^{32}. Although this slide attack does not help in finding collisions or preimages for the hash function, the analysis demonstrates an unexpected property of the compression function of SHA-1, which does not have the expected random behaviour.

4.7.3 Description of the SHA-256 algorithm

The SHA-256 algorithm [51] computes hash values of 256 bits, for messages of any length shorter than 2^{64} bits. The algorithm has a word length of 32 bits, therefore the chaining variable is divided into eight registers (A, B, C, D, E, F, G, H) of 32 bits each. The compression function works on message blocks of 512 bits, a block is divided into sixteen 32-bit words denoted by W_j for $j = 0, 1, \ldots, 15$.

Internally, the compression function is divided into 64 sequential steps.[11] The step operation of SHA-256 is of the following form:

$$\begin{aligned} D &\leftarrow D + H + f_1(E,F,G) + \Sigma_1(E) + W_j + U_s \,, \\ H &\leftarrow H + f_1(E,F,G) + \Sigma_1(E) + f_2(A,B,C) + \Sigma_2(A) + W_j + U_s \,. \end{aligned}$$

So every step computes a new value for two of the eight registers. In this case we consider a step that updates the values of the D and H registers. The operation that updates the D and H registers depends on the other six registers A, B, C, E, F, G and on the following:

- a message word W_j from the set $j = \{0, 1, \ldots, 63\}$;
- two Boolean functions f_1 and f_2;
- two functions Σ_1 and Σ_2;
- an additive constant U_s that depends on the step.

The Boolean functions f_1 and f_2 are the selection and majority functions respectively. The functions Σ_1 and Σ_2 compute an output word from an input word Z as follows:

$$\begin{aligned} \Sigma_1(Z) &= Z^{\ggg 6} \oplus Z^{\ggg 11} \oplus Z^{\ggg 25} \,, \\ \Sigma_2(Z) &= Z^{\ggg 2} \oplus Z^{\ggg 13} \oplus Z^{\ggg 22} \,. \end{aligned}$$

The first sixteen words W_j (for $j = 0, 1, \ldots, 15$) are equal to the message block input of the compression function. The remaining forty-eight words W_j (for $j = 16, 17, \ldots, 63$) are computed as follows:

$$W_j = \sigma_1(W_{j-2}) + W_{j-7} + \sigma_2(W_{j-15}) + W_{j-16} \,,$$

where the functions σ_1 and σ_2 compute an output word from an input word Z as follows:

$$\begin{aligned} \sigma_1(Z) &= Z^{\ggg 17} \oplus Z^{\ggg 19} \oplus Z^{\hookrightarrow 10} \,, \\ \sigma_2(Z) &= Z^{\ggg 7} \oplus Z^{\ggg 18} \oplus Z^{\hookrightarrow 3} \,. \end{aligned}$$

A graphical representation of the step operation of SHA-256 is given in Fig. 4.9 below. Note that four consecutive steps update the values of the registers D and H, C and G, B and F, A and E respectively. After four steps the complete

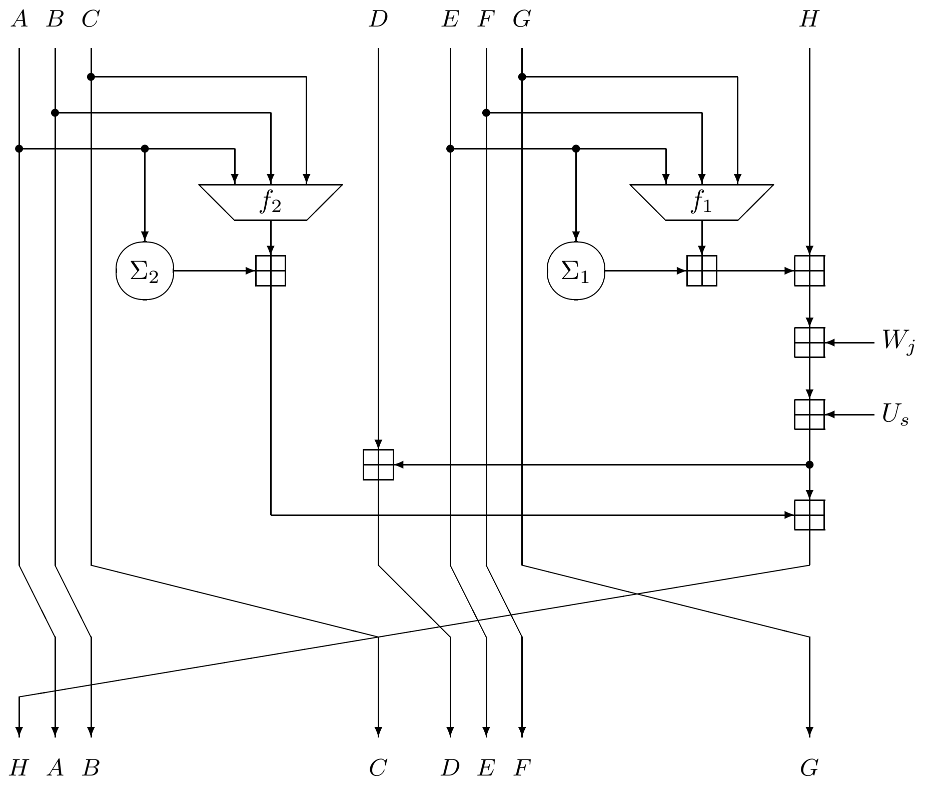

Figure 4.9: Step operation for SHA-256.

chaining variable has been updated. Hence, each register is updated sixteen times by the compression function.

The step operation of SHA-256 is reversible. For example, the previous values of the D and H registers can be computed by

$$\begin{aligned} H_{prev} &= H_{new} - f_1(E,F,G) - \Sigma_1(E) - f_2(A,B,C) - \Sigma_2(A) - W_j - U_s\,, \\ D_{prev} &= D_{new} - H_{prev} - f_1(E,F,G) - \Sigma_1(E) - W_j - U_s\,. \end{aligned}$$

However after execution of all 64 steps, the compression function uses a feed-forward operation which adds the initial values of the registers (the values at the start of the compression function) to their final values (obtained after 64 steps).

[11]Note that for SHA-256 there is no clear distinction into rounds.

The result is the chaining variable output from the compression function. Due to the feed-forward at the end the compression function cannot be inverted.

For a detailed description of SHA-256 we refer to [51].

4.7.4 Security of SHA-256

The design of SHA-256 has some similarities to SHA-1, but there are important differences in the structure. As for SHA-1 the design principles for this algorithm have not been made public. It must be noted that the number of steps in the compression function of SHA-256 is lower than for SHA-1 (64 steps compared to 80). On the other hand, two registers are updated in every step of the compression function where for SHA-1 only one register is updated in a step. SHA-256 uses the same Boolean functions f_1 and f_2 in all steps of the compression function, but a unique additive constant in every step. The Σ_i functions (in the step operations) and the σ_i functions (in the message expansion) achieve a faster diffusion than the bit rotations which are used in SHA-1. For the message expansion there is also faster diffusion because modular additions are used instead of exor operations. H. Gilbert and H. Handschuh [56] have remarked that the Σ_i and σ_i functions are one-to-one mappings (so no internal collisions can be achieved through them). Note that the slide attack on SHA-1 does not extend to SHA-256 because every step of the compression function uses a unique additive constant. The collision attack on SHA-0 is not applicable because of the rotations and shifts in the σ_i functions of the message expansion.

In [56] Gilbert and Handschuh point to a weakness in simplified variants of SHA-256. These variants are obtained by replacing the chaining values at the input of the compression function, and the additive constants that are used, by symmetric values (the 32-bit words have equal left and right 16-bit halves). Furthermore, all modular additions are replaced by exor operations, and the shifts in the σ_i functions by rotations. For such a simplified variant of SHA-256 it can be shown that a message block input to the compression function, which consists of symmetric 32-bit words, always leads to an output consisting of symmetric words. Therefore the complexity of a birthday attack is reduced to 2^{64} operations (instead of 2^{128}), because a collision on the left half of each output word implies a collision on the whole output word. The observation can be generalised for simplified variants with symmetric words consisting of four, eight, ... equal parts (resulting in even lower complexities for a birthday attack). It should be noted that the same weakness exists for similarly simplified variants of SHA-1 and other hash functions of the MDx-class.

4.7.5 SHA-224, SHA-384 and SHA-512

The SHA-224 algorithm [51] computes hash results of 224 bits, for messages of any length shorter than 2^{64} bits. SHA-224 is defined in the exact same manner as SHA-256, with two exceptions: it uses different initial values for the registers of the chaining variable, and the output from the algorithm (obtained after the final application of its compression function) is truncated to its leftmost 224 bits.

The SHA-384 and SHA-512 algorithms [51] compute hash results of 384 bits and 512 bits respectively, for messages of any length shorter than 2^{128} bits. These functions have a word length of 64 bits. The chaining variable is 512 bits long, and divided into eight registers (A, B, C, D, E, F, G, H) of 64 bits each. The compression function of SHA-384 and SHA-512 works on message blocks of 1024 bits, a block is divided into sixteen 64-bit words denoted by W_j for $j = 0, 1, \ldots, 15$. The structure of SHA-512 is similar to the structure of SHA-256, except that it uses 80 sequential steps in the compression function (instead of 64 steps). It can be seen that this implies that each of the registers is updated twenty times by the compression function. The main difference between SHA-512 and SHA-256 is the word length (64 compared to 32 bits). Therefore, SHA-512 uses different initial values for the registers of the chaining variable, and different additive constants. Furthermore, the rotation and shift amounts in the functions Σ_1, Σ_2, σ_1, σ_2 are different. SHA-384 is defined in the same manner as SHA-512 except that it starts from different initial values for the registers of the chaining variable, and the output from the algorithm (obtained after the final application of its compression function) is truncated to its leftmost 384 bits.

For a detailed description of SHA-224, SHA-384 and SHA-512 we refer to [51]. With respect to the security of these algorithms similar remarks can be made as for SHA-256. Note that SHA-384 and SHA-512 have more steps in the compression function (80 steps compared to 64).

4.7.6 Conclusions

We have given an overview of the design of the algorithms from the SHA family. A theoretical attack has been described on the original SHA algorithm, which supports the move to SHA-1. The slide attack which was recently shown for SHA-1 does not impact the security of this algorithm as a one-way or collision-resistant hash function, but it points to an unexpected property of the compression function. Note that the output length of SHA-1 (160 bits) does not offer long term security against birthday attacks. The SHA-224, SHA-256, SHA-384 and SHA-512 algorithms offer larger output lengths. They have been proposed and standardised only recently by NIST but unfortunately NIST did not use an open standardisation process for its hash function standard, in contrast to the standardisation procedure used for the block cipher standard AES. Early cryptanalytic results suggest that these algorithms have a high security margin against known attack strategies.

4.8 Comparison of MDx-class Hash Functions

In this section we summarise the observations made on the different hash functions from the MDx-class. Table 4.9 compares the structure of the different algorithms, and Table 4.10 summarises the most important attacks which have been found. We also compare the software performance of most of the algorithms in Table 4.11. These performance figures are due to the NESSIE project [92], the implementations are on a Pentium 3 processor running Linux. Other performance comparisons can be found in [90, 24]; they lead to comparable results. We estimate that the performance of five-round HAVAL is close to the performance of SHA-1 and RIPEMD-160. It may be noted that SHA-384 and SHA-512 perform much better on 64-bit platforms. The early proposals of the MDx-class are more efficient than the more recent proposals, but according to Table 4.10 they are also less secure.

Table 4.9: Structure of MDx-class hash functions (lengths are given in bits).

Algorithm	output length	block length	word length	number of steps
MD4	128	512	32	3×16
MD5	128	512	32	4×16
HAVAL	128–256	1024	32	3, 4, or 5×32
RIPEMD	128	512	32	$3 \times 16 \,\|\|\, 3 \times 16$
RIPEMD-128	128	512	32	$4 \times 16 \,\|\|\, 4 \times 16$
RIPEMD-160	160	512	32	$5 \times 16 \,\|\|\, 5 \times 16$
SHA	160	512	32	4×20
SHA-1	160	512	32	4×20
SHA-224/256	224/256	512	32	1×64
SHA-384/512	384/512	1024	64	1×80

4.9 Conclusions

In this chapter we have given an extensive overview of the hash functions of the MDx-class. This class of hash functions is a good example of the interaction between the fields of cryptography and cryptanalysis: important weaknesses have been shown in the early proposals, and these have been taken into account for the design of the newer algorithms (unfortunately at the cost of decreased efficiency).

Table 4.10: Known attacks for MDx-class hash functions.

Algorithm	Weakness
MD4	collisions (also preimages for two-round reduced versions)
MD5	pseudo-collisions and collisions for compression function
HAVAL	collisions for three-round version
RIPEMD	collisions for two-round reduced versions
SHA	theoretical collision attack
SHA-1	slide attack

Table 4.11: Software performance of MDx-class hash functions (PIII, Linux).

Algorithm	cycles/byte	relative performance
MD4	4.7	1
MD5	7.2	0.65
RIPEMD-160	18	0.26
SHA	15	0.31
SHA-1	15	0.31
SHA-224/256	39	0.12
SHA-384/512	83	0.06

We have given an extensive analysis of MD4, and we have presented the first attack on a complete version of HAVAL. This result has been published in [123].

Based on our present knowledge RIPEMD-160 and SHA-1 seem the most interesting hash functions if an output length of 160 bits is sufficient. For larger output lengths the recent hash functions of the SHA family can be used, although they should receive more extensive public cryptanalysis.

Chapter 5

The PANAMA Cryptographic Module

5.1 Introduction

PANAMA is a cryptographic module proposed in 1998 by J. Daemen and C. Clapp [30]. It can serve both as a stream cipher, and as a cryptographic hash function with hash results of 256 bits. PANAMA achieves high performance (for large amounts of data) because of its inherent parallelism. In this chapter we analyse the security of PANAMA when used as a hash function, and demonstrate an attack that finds collisions much faster than a generic birthday attack. Our attack has been published in [112].

5.2 Description of PANAMA

The PANAMA stream/hash module [30] is based on a finite state machine with two types of internal memory: the *state* and the *buffer*. The module is used in an iterative manner, where each step of the iteration updates both the state and the buffer. There are two different modes for the iteration function, called *push* and *pull* mode. The push mode takes an input and has no output, the pull mode has no input but generates an output. A graphical representation of these modes is given in Fig. 5.1 and Fig. 5.2 respectively. Below we only describe how to use PANAMA as a hash function, we refer to [30] for a full specification of the cryptographic module.

The state of PANAMA consists of 544 bits, divided into seventeen 32-bit words $a_0, a_1, \ldots, a_{16}$. We will also use the notation a^s and a^t to refer to the set of words

a_1 to a_8, and a_9 to a_{16} respectively. In this way the state can be denoted by a 3-tuple (a_0, a^s, a^t), where a_0 represents the first 32-bit word, and a^s and a^t each represent a value of 256 bits (eight words). The buffer of Panama is a linear feedback shift register (LFSR), containing 32 stages of 256 bits (eight words) each. An 8-word stage is denoted by b^j and its words by b^j_i $(0 \leq j \leq 31$ and $0 \leq i \leq 7)$.

In order to use Panama as a hash function, the message input M is converted into a string $\tilde{M}$ with a length that is a multiple of 256 bits. This is done by appending a single 1-bit followed by the smallest number of 0-bits (possibly none) resulting in a length that is a multiple of 256 bits. The string $\tilde{M}$ is then divided into message blocks of 256 bits each. We denote this by $\tilde{M} = m^0 \parallel m^1 \parallel \ldots \parallel m^n$.

Before the hashing starts all the internal memory bits (of the state and the buffer) are set to zero. Next, for each message block m^k $(0 \leq k \leq n)$, the following steps are executed (*push mode* with m^k as input). First, the state is updated by applying the non-linear transformation ρ (which is composed of three specific transformations $\theta \circ \pi \circ \gamma$, see Sect. 5.2.1 below):

$$(a_0, a^s, a^t) \leftarrow \rho(a_0, a^s, a^t). \tag{5.1}$$

Secondly, the message block m^k and b^{16}, the contents of the buffer stage 16, are exored into the state. The least significant bit of the word a_0 is flipped. These three operations are denoted here[1] by σ:

$$(a_0, a^s, a^t) \leftarrow \sigma(a_0, a^s, a^t) = (a_0 \oplus 1, a^s \oplus m^k, a^t \oplus b^{16}). \tag{5.2}$$

Thirdly, the message block m^k is fed into the buffer and the LFSR is stepped once. The buffer updating function λ is slightly more complex than in an ordinary LFSR, $b \leftarrow \lambda(b)$ is defined by:

$$\begin{array}{lcl} b^j & \leftarrow & b^{j-1} \text{ if } j \notin \{0, 25\}\,, \\ b^0 & \leftarrow & b^{31} \oplus m^k\,, \\ b^{25}_i & \leftarrow & b^{24}_i \oplus b^{31}_{i+2 \bmod 8} \text{ for } 0 \leq i \leq 7\,. \end{array} \tag{5.3}$$

When all the message blocks have been processed, 33 extra iterations are performed (*pull mode*), but now the message input to the buffer is replaced by the state (part a^s), and the message input of σ is replaced by b^4 (the contents of the buffer stage 4). The outputs from the first 32 of these extra iterations are discarded, and the 256-bit hash result is defined as the output from the last iteration, this corresponds to part a^t of the final state value.

[1] Our notation differs from the one in [30], where σ is included in ρ.

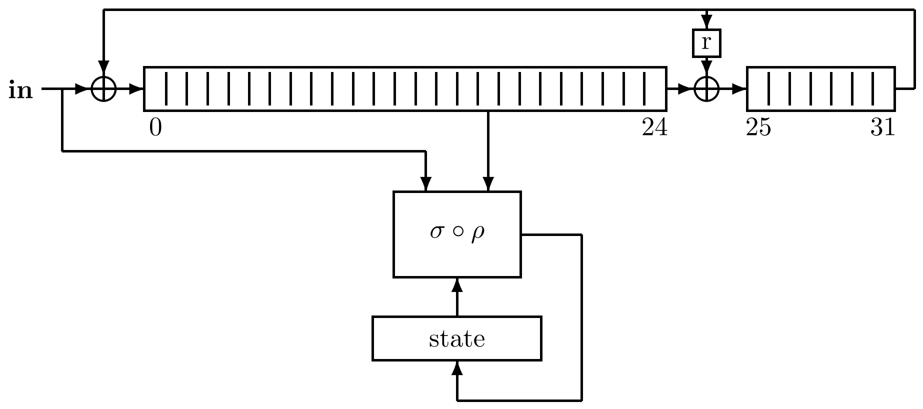

Figure 5.1: The Panama module in *push* mode.

.

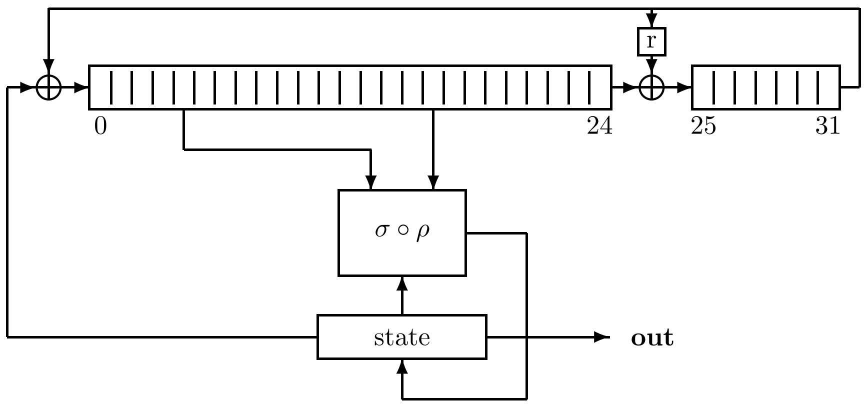

Figure 5.2: The Panama module in *pull* mode.

.

5.2.1 Components of PANAMA

We describe here the different components of the non-linear state updating transformation $\rho = \theta \circ \pi \circ \gamma$ (see equation (5.1)). We also make some preliminary observations which will be useful for our analysis in Sect. 5.3. The transformations operate on the seventeen words of the PANAMA state, where each word consists of 32 bits. Below we denote the input words for a transformation with a_i, and the output words with c_i, where $0 \leq i \leq 16$. Note that the '+' operations used in the indices should be taken modulo 17.

The transformation θ

The linear transformation θ is defined as follows:

$$c = \theta(a) \Leftrightarrow c_i = a_i \oplus a_{i+1} \oplus a_{i+4}\,. \tag{5.4}$$

We have computed the inverse transformation θ^{-1}:

$$a_i = c_{i+1} \oplus c_{i+2} \oplus c_{i+5} \oplus c_{i+9} \oplus c_{i+10} \oplus c_{i+11} \oplus c_{i+12} \oplus c_{i+14} \oplus c_{i+16}\,. \tag{5.5}$$

The transformation π

The transformation π which combines bit rotations with a permutation of the word positions is defined below. Here $[j]$ $(0 \leq j \leq 31)$ denotes the bit position in a word, the multiplication $7i$ in the index is modulo 17, and the addition of $s(i) = i \cdot (i+1)/2$ is modulo 32 and corresponds to a bit rotation.

$$c = \pi(a) \Leftrightarrow c_i[j] = a_{7i}[j + s(i)]\,. \tag{5.6}$$

The transformation γ

The transformation γ is the only non-linear component. We will analyse this transformation at bit level. Because γ does not mix bits with different positions in the words, it can be considered as a parallel application of 32 transformations γ_b.

We study the differential properties of γ. Let a and $a \oplus d$ denote two 17-bit vectors that are input to γ_b. Note that the inputs a and $a \oplus d$ contain one bit from each of the seventeen state words, for a certain bit position $[j]$. Let c and $c \oplus e$ denote the corresponding outputs: $c = \gamma_b(a), c \oplus e = \gamma_b(a \oplus d)$. From the definition of γ, we have the following:

$$\begin{aligned} c_i &= a_i \oplus (a_{i+1} \vee (1 \oplus a_{i+2}))\,, & (5.7)\\ c_i \oplus e_i &= a_i \oplus d_i \oplus ((a_{i+1} \oplus d_{i+1}) \vee (1 \oplus a_{i+2} \oplus d_{i+2}))\,. & (5.8) \end{aligned}$$

Using De Morgan's law, (5.7) can be transformed into

$$c_i = a_i \oplus a_{i+1}a_{i+2} \oplus a_{i+2} \oplus 1\,. \tag{5.9}$$

Doing the same with (5.8) and combining the result with (5.9), we get

$$e_i \oplus d_i \oplus d_{i+2} \oplus d_{i+1}d_{i+2} = a_{i+1}d_{i+2} \oplus a_{i+2}d_{i+1}\,. \tag{5.10}$$

Table 5.1 below lists the solutions to this equation, in a format that is useful for our analysis in Sect. 5.3. It shows the relation between the absolute value of the input (a), the input difference (d) and the output difference (e). Note that an x or y in the column of a means that this bit can take two values, e.g., $(x, 1 \oplus x)$ means 'both $(0, 1)$ and $(1, 0)$ are possible'.

Table 5.1: The transformation γ_b: relation between input difference (d), output difference (e) and absolute value of the input (a).

$d_i \oplus e_i$	d_{i+1}	d_{i+2}	(a_{i+1}, a_{i+2})
0	0	0	(x, y)
0	0	1	$(1, x)$
0	1	0	$(x, 0)$
0	1	1	(x, x)
1	0	0	-
1	0	1	$(0, x)$
1	1	0	$(x, 1)$
1	1	1	$(x, 1 \oplus x)$

5.2.2 Observations on the design

The design of the Panama hash function differs significantly from the design of the hash functions of the MDx-class, which we discussed in Chapter 4. Panama can also be described in the iterated hash model of Chapter 3 (Sect. 3.3), but it differs from the MDx-class hash functions in the following ways:

– **Length of the chaining variable:** The MDx-class hash functions have a chaining variable of the same length as the hash result (128 to 512 bits, depending on the algorithm). For Panama the chaining variable corresponds to the internal state and buffer, with a total length of 544+8192=8736 bits. The hash result is only 256 bits long.

– **Nature of the iteration function:** The MDx-class hash functions are based on the iteration of a compression function, consisting in itself of a sequence of a large number of (simple) step operations. In PANAMA the iteration function, used to update the state and buffer, has a parallel (rather than a sequential) structure: for each of the transformations γ, π, θ and σ the seventeen state words a_0 to a_{16} can in principle be updated in parallel.

– **Presence of an output transformation:** For MDx-class hash functions, the hash result is the final value of the chaining variable. PANAMA first processes all message blocks using the iteration function in push mode, and then applies 33 extra iterations in pull mode, which form an output transformation mapping the value of the chaining variable (state and buffer) to the hash result (part of the final state).

These differences are the consequence of a different design strategy. For the MDx-class hash functions, the compression function is designed to be collision-resistant in itself, and the iteration mechanism ensures that the resulting hash function is collision-resistant (according to the Merkle-Damgård theorem, see Sect. 3.3.1). In order to find collisions a cryptanalyst targets the compression function, and hence a single message block (see for example the attacks on MD4 and HAVAL in Chapter 4).

For the PANAMA hash function, the diffusion and non-linearity realised by successive applications of the iteration function are expected to prevent cryptanalysis. In Sect. 5.3 below we will see that in a collision attack on PANAMA the attacker needs to target a message stream consisting of many blocks. Note also that the state updating transformation of PANAMA is invertible. Therefore a meet-in-the-middle attack can be used to find preimages, but this is not faster than a brute-force preimage search because the state (544 bits) is more than twice as long as the hash result (256 bits).

Performance aspects

The design of PANAMA is oriented towards software implementation on 32-bit architectures. According to [30] the per-byte workload of PANAMA is similar to that of MD4, the fastest member of the MDx-class. Furthermore, it is shown that the performance of PANAMA can substantially benefit from processors capable of a high degree of instruction-level parallelism. On the other hand, the performance of PANAMA decreases significantly when hashing short messages. This is due to the overhead of the output transformation (33 pull iterations), which is equivalent to the hashing of about 1 Kbyte.

5.3 Analysis of PANAMA in Hashing Mode

In this section we describe a method to produce collisions for the PANAMA hash function. A collision occurs when two different messages are hashed to the same result. The output length of the PANAMA hash function is 256 bits, which means that a generic birthday attack would need about 2^{128} hash computations to find a colliding pair of messages. We will show that with our method a collision can be found with significantly less operations and a small amount of memory. This result has been published in [112].

We will use the following notation: a message stream is denoted by $m^0, m^1, \ldots$ m^n, where m^0 is the first message block and m^n the last. Each block m^k corresponds to a set of eight 32-bit words. Furthermore, let X be a set of eight 32-bit words X_i, then $Y = r(X) \Leftrightarrow Y_i = X_{i+2 \bmod 8}$ $(0 \leq i \leq 7)$.

5.3.1 Message format for collisions

As seen in Sect. 5.2, the hash result of PANAMA is taken from the state, part a^t. This means that the buffer content is not present in the output. Hence, there are two types of collisions for PANAMA: collisions in the state only, and collisions in both the buffer and the state. The 33 pull iterations after the processing of the last message block have as function to make it difficult to produce collisions of the first kind. We will try to find collisions of the second kind. The linear updating function of the buffer imposes strict conditions on the format of colliding messages.

Collisions for the buffer

Because of the linear feedback in the buffer, a difference pattern in a single message block gives rise to an infinite difference propagation in the buffer. Only difference patterns that meet a particular condition cause a finite difference propagation, that is a collision, in the buffer.

It can be seen from definition (5.3) of the buffer updating function λ that the simplest collision for the buffer is obtained from the following difference pattern in the message stream:

$$dX, 0, 0, 0, 0, 0, 0, r(dX), 24 \text{ zero stages}, dX\,. \tag{5.11}$$

Furthermore, all difference patterns for buffer collisions are composed by adding shifted versions of pattern (5.11).

Collisions for the state

For a difference pattern of type (5.11), there will be five occasions where the input difference of the state is non-zero: the first two non-zero blocks are each injected twice into the state: once when they are the current message block and once when they pass through the buffer stage 16 (this follows from the definition of the transformation σ, see Sect. 5.2). The last non-zero block cancels out all differences in the buffer, but is injected once into the state.

We are going to use a strategy of immediate compensation in the state: every time a difference is introduced into the state, we will try to let it die out as quickly as possible. A collision can then be seen as consisting of five *sub-collisions*, where a sub-collision is defined as a collision in the state only. This is also observed by the designers of PANAMA in [30]. The five sub-collisions are related because they are based on the same set of input difference words dX_i $(0 \leq i \leq 7)$. This is shown in Table 5.2 below.

Table 5.2: Differences introduced into the state by the transformation σ.

sub-collision	a_0	a^s	a^t
1	-	dX	-
2	-	$r(dX)$	-
3	-	-	dX
4	-	-	$r(dX)$
5	-	dX	-

Since the state updating function ρ is invertible, a difference dX which is introduced can only disappear under the influence of another difference dY. An intuitive choice for the format of the colliding messages is as follows:

$$dX, dY, 0, 0, 0, 0, 0, r(dX), r(dY), 23 \text{ zero stages}, dX, dY \ . \tag{5.12}$$

Since this format is the addition of two shifted copies of difference pattern (5.11), it will produce a collision in the buffer. The strategy of immediate compensation demands that the differences introduced into the state by the 'dX' values, are compensated by the 'dY' values. The difference propagation in the state (a_0, a^s, a^t) should be as follows:

$$\begin{array}{ccccccc}
(0,0,0) & \xrightarrow{\sigma} & (0,dX,0) & \xrightarrow{\rho} & (0,dY,0) & \xrightarrow{\sigma} & (0,0,0) \\
(0,0,0) & \xrightarrow{\sigma} & (0,r(dX),0) & \xrightarrow{\rho} & (0,r(dY),0) & \xrightarrow{\sigma} & (0,0,0) \\
(0,0,0) & \xrightarrow{\sigma} & (0,0,dX) & \xrightarrow{\rho} & (0,0,dY) & \xrightarrow{\sigma} & (0,0,0) \\
(0,0,0) & \xrightarrow{\sigma} & (0,0,r(dX)) & \xrightarrow{\rho} & (0,0,r(dY)) & \xrightarrow{\sigma} & (0,0,0)
\end{array} \tag{5.13}$$

Note that the first propagation path applies to the first and the fifth sub-collision. The difference propagation through σ is satisfied automatically, but a solution must be found for the difference propagation through ρ. It turns out that due to the interaction between the buffer updating function and the state updating function, there are no solutions dX, dY for (5.13). A proof is given in Appendix D.

Therefore we will use a difference pattern of the following form:

$$dX, 0, dY, 0, 0, 0, 0, r(dX), 0, r(dY), 22 \text{ zero stages}, dX, 0, dY\ . \tag{5.14}$$

Table 5.3 below gives a schematic overview of the difference values in the buffer and the state during the attack. The values shown for the buffer and the state are the values *after* the message block m^k has been processed. The non-zero differences in the state during subcollision l are denoted with z_l. Because of the strategy of immediate compensation, the differences in the state are zero most of the time.

5.3.2 Overview of the procedure

We will use differences where the bits of a 32-bit difference word are all set or all unset.[2] This allows to denote a difference pattern in the state with seventeen bits, one for each word. Furthermore, it means that the bit rotation in π has no effect on the difference values. We make this choice mainly because it is easier to think about differences of this format.

A collision will be found by combining the results of the searches for the five sub-collisions. The method we present here to find a sub-collision works for any value of the state at the start of the procedure. Note that in every sub-collision, the message input of σ is 'fresh': it can be chosen freely. Since the message words are added to the state words a_1 to a_8, it is easy to control part a^s of the state. The state word a_0 is exored with the constant 00000001_x so we have no direct control over its value. The buffer input of σ is added to the state words a_9 to a_{16}. This input is influenced by the values of message blocks which have been injected into the state earlier on.

It seems difficult to make use of the fact that the buffer can be controlled: changing the buffer would require a recomputation of the current state. Our method assumes that only the message input can be controlled. Because the message input of σ influences only eight of the seventeen state words, we have not enough degrees of freedom if we vary the current message block only. Therefore, we will also have to select values for the common message block before the message blocks with difference dX or $r(dX)$.

[2]Note that these differences are defined with respect to the exclusive-OR operation, that is $x = y \oplus \Delta$ with $\Delta = \texttt{00000000}_x$ or $\texttt{ffffffff}_x$.

Table 5.3: Schematic overview of the difference propagation through the buffer stages and the state.

k	m^k	Buffer								State		
		0	$\cdots$	16	$\cdots$	24	25	$\cdots$	31	a_0	a^s	a^t
0	0	0		0		0	0		0	0	0	0
1	dX	dX		0		0	0		0	0	dX	0
2	0	0		0		0	0		0	z_1	z_1	z_1
3	dY	dY		0		0	0		0	0	0	0
4	0	0		0		0	0		0	0	0	0
5	0	0		0		0	0		0	0	0	0
6	0	0		0		0	0		0	0	0	0
7	0	0		0		0	0		0	0	0	0
8	$r(dX)$	$r(dX)$		0		0	0		0	0	$r(dX)$	0
9	0	0		0		0	0		0	z_2	z_2	z_2
10	$r(dY)$	$r(dY)$		0		0	0		0	0	0	0
11	0	0		0		0	0		0	0	0	0
12	0	0		0		0	0		0	0	0	0
13	0	0		0		0	0		0	0	0	0
14	0	0		0		0	0		0	0	0	0
15	0	0		0		0	0		0	0	0	0
16	0	0		0		0	0		0	0	0	0
17	0	0		dX		0	0		0	0	0	dX
18	0	0		0		0	0		0	z_3	z_3	z_3
19	0	0		dY		0	0		0	0	0	0
20	0	0		0		0	0		0	0	0	0
21	0	0		0		0	0		0	0	0	0
22	0	0		0		0	0		0	0	0	0
23	0	0		0		0	0		0	0	0	0
24	0	0		$r(dX)$		0	0		0	0	0	$r(dX)$
25	0	0		0		dX	0		0	z_4	z_4	z_4
26	0	0		$r(dY)$		0	dX		0	0	0	0
27	0	0		0		dY	0		0	0	0	0
28	0	0		0		0	dY		0	0	0	0
29	0	0		0		0	0		0	0	0	0
30	0	0		0		0	0		0	0	0	0
31	0	0		0		0	0		0	0	0	0
32	0	0		0		$r(dX)$	0		dX	0	0	0
33	dX	0		0		0	0		0	0	dX	0
34	0	0		0		$r(dY)$	0		dY	z_5	z_5	z_5
35	dY	0		0		0	0		0	0	0	0

For every sub-collision we use the following notation. The state will be denoted by the capitals A to N, according to the timeline (5.15). The corresponding difference values are dA to dN.[3]

$$\begin{array}{l} K \xrightarrow{\sigma 0} L \xrightarrow{\gamma} M \xrightarrow{\pi} N \xrightarrow{\theta} A \xrightarrow{\sigma 1} B \xrightarrow{\gamma} C \xrightarrow{\pi} D \xrightarrow{\theta} E \\ E \xrightarrow{\sigma 2} F \xrightarrow{\gamma} G \xrightarrow{\pi} H \xrightarrow{\theta} I \xrightarrow{\sigma 3} J \end{array} \tag{5.15}$$

We consider in total four σ operations, σk denoting the σ operation with message block Pk ($k = 0, 1, 2, 3$). Here $P0$ to $P3$ correspond to m^0 to m^3 for the first sub-collision, $P0$ to $P3$ correspond to m^7 to m^{10} for the second sub-collision, etc. (see Table 5.3). Message blocks $P0$ and $P2$ are equal for both messages of the collision. For sub-collisions 1, 2 and 5 the block $P1$ has the 'dX difference', and the block $P3$ the 'dY' difference. For the other two sub-collisions these differences are imposed by the contents of the buffer stage 16 in $\sigma 1$ and $\sigma 3$. The states K to A are equal in both messages, the common block $P0$ is used to bring the state to a value that allows a sub-collision to happen. It can be seen that the exact values of the $P3$ blocks (for the two messages of the collision) have no importance, we only require that for every sub-collision the difference in the state I is canceled by the difference in $P3$ (or by the difference in the buffer stage 16 for $\sigma 3$).

5.3.3 The chosen difference format

We give here a difference pattern that is a solution of (5.14). We describe the propagation of the difference from state B to state I for every sub-collision. As noted above the pattern is described by seventeen bits, one bit for each word of the state ($0 \leq i \leq 16$).

The difference propagation through the linear transformations π and θ does not depend on the exact values of the state (for the two messages of the collision), only on the difference. The transformation $\sigma 2$ doesn't change the difference at all. The difference propagation through the non-linear transformation γ does depend on the exact values of the state and this imposes a set of conditions on the value of the state at 'time' B and F. These conditions, which can be derived using the results from Table 5.1, must be satisfied for the sub-collisions to occur. Note again that we can work with single bits because γ can be seen as 32 parallel transformations γ_b.

Table 5.4 below shows the required difference propagation for sub-collision 1 (and for sub-collision 5). In this table state B has the difference dX in part a^s ($dB_6 = dB_7 = 1$), as imposed by the message block $P1$. State I has the difference dY in part a^s ($dI_2 = dI_3 = dI_5 = dI_8 = 1$), which will be canceled by

[3]Furthermore, the state words $a_0 \ldots a_{16}$ for state A to N are denoted $A_0 \ldots A_{16}$ to $N_0 \ldots N_{16}$. The corresponding difference words are $dA_0 \ldots dA_{16}$ to $dN_0 \ldots dN_{16}$.

the difference in the message block $P3$. Parts a_0 ($i = 0$), a^s ($i = 1, \dots 8$) and a^t ($i = 9, \dots 16$) of the state are clearly separated in the table. Table 5.1 has been used to derive the following conditions on B and F for the difference propagation of Table 5.4.[4] Note that these conditions are at bit level, they must be satisfied for every bit position $[j]$ $(0 \leq j < 32)$.

$$B_5 = 1\,,\ B_6 \oplus B_7 = 1\,,\ B_8 = 0\,, \tag{5.16}$$

$$\begin{aligned} &F_0 \oplus F_1 = 1\,,\ F_2 = 1\,,\ F_3 = 0\,,\ F_5 = 0\,,\ F_6 = 0\,,\ F_7 = F_8\,,\ F_8 \oplus F_9 = 1\,,\\ &F_{10} = 0\,,\ F_{11} = 0\,,\ F_{12} = F_{13} = F_{14}\,,\ F_{15} = 1\,,\ F_{16} = 1. \end{aligned} \tag{5.17}$$

Table 5.4: The required difference propagation for sub-collision 1 (and 5).

i	dB	dC	dD	$dE = dF$	dG	dH	dI
0	0	0	0	1	0	0	0
1	0	0	1	1	1	0	0
2	0	0	0	0	1	1	1
3	0	0	0	0	0	1	1
4	0	0	0	1	1	0	0
5	0	1	0	0	1	1	1
6	1	1	0	0	0	1	0
7	1	1	0	1	0	0	0
8	0	0	1	1	1	1	1
9	0	0	0	1	1	1	0
10	0	0	0	0	1	1	0
11	0	0	0	0	0	1	0
12	0	0	0	1	1	1	0
13	0	0	1	1	0	0	0
14	0	0	0	1	1	0	0
15	0	0	0	0	0	0	0
16	0	0	0	0	1	1	0

For the other sub-collisions we refer to the similar Tables D.1, D.2 and D.3, and the corresponding conditions (D.16) to (D.21) in Appendix D. All these sub-collisions are related because in every case state B contains the difference dX or $r(dX)$ in part a^s (imposed by the message) or part a^t (imposed by the

[4]For example, $dB_4 \oplus dC_4 = 0, dB_5 = 0, dB_6 = 1$ implies $B_5 = 1$. We use Table 5.1 where B is the input a, dB is the input difference d and dC is the output difference e.

buffer stage 16), while state I must contain the difference dY respectively $r(dY)$ in the corresponding part of the state.

Other solutions for the difference format are possible. We have chosen one which results in the minimum number of conditions on B and F. It is not clear how to make an optimal choice for an easy solution of these conditions in the next step of the attack.

5.3.4 Producing collisions

In order to generate each of the sub-collisions we need to solve the conditions for the state values at 'time' B and F as given in Sect. 5.3.3 and Appendix D. Values for the state words a_1 to a_8 (part a^s) can be met by a suitable choice of the message block used at the corresponding time: $P1$ for B and $P2$ for F. To meet the values for the other state words (a_0 and a_9 to a_{16}) we need to compute backwards to the previous message block: $P0$ for B and $P1$ for F. In this way the problem of finding a sub-collision can be solved in three steps:

1. Solve a system of equations in the unknowns $L_1, \ldots L_8$; this determines the message block $P0$.
2. Solve a system of equations in the unknowns $B_1, \ldots B_8$; this determines the message block $P1$.
3. Solve a system of equations in the unknowns $F_1, \ldots F_8$; this determines the message block $P2$.

The difficulty in solving these systems of equations is that when we compute backwards to the previous message blocks, we obtain increasingly non-linear equations because of the transformation γ. In our approach we only go back through one application of γ. To explain how we solve the problem, we will first describe the solution of the first sub-collision in some detail. After that we will summarise the procedure for the other sub-collisions and discuss the complexity of the attack.

Producing sub-collision 1 (and 5)

From (5.17) it can be seen that there are seven conditions which include the unknowns $F_1, \ldots F_8$. Let $[j]$ denote the bit position in a word, then we have the following equations for $0 \leq j < 32$:

$$\begin{aligned} &F_0[j] \oplus F_1[j] = 1\,,\ F_2[j] = 1\,,\ F_3[j] = 0\,,\ F_5[j] = 0\,,\\ &F_6[j] = 0\,,\ F_7[j] = F_8[j]\,,\ F_8[j] \oplus F_9[j] = 1\,. \end{aligned} \tag{5.18}$$

These conditions are easy to satisfy because $F_1, \ldots F_8$ are obtained by bitwise addition of $E_1, \ldots E_8$ and message block $P2$ which can be freely chosen.

In (5.17) we have 6 remaining conditions on $F_0, F_9, \ldots F_{16}$, which can be transformed into equations in the unknowns $B_1, \ldots B_8$. These have to be added to the 3 conditions we already have in these unknowns from (5.16),

$$B_5[j] = 1\,,\; B_6[j] \oplus B_7[j] = 1\,,\; B_8[j] = 0\,, \tag{5.19}$$

resulting in an overdetermined system of nine equations (and 32 bit positions). Generally such a system of equations has no solution, unless one of the equations is dependent on the other eight. If there is no hidden structure in these equations, we can assume that we will need 2^{32} trials before we can solve them. Experimental evidence confirms this assumption.

To compute backwards from state F to state B we apply the transformations σ, θ, π and γ, which lead to the following equations. Here $mb_i[j] = b^{16}_{i-9}[j]$ for $9 \leq i \leq 16$, $mb_0[j] = 0$ for $0 \leq j \leq 30$ and $mb_0[31] = 1$.

$$\begin{array}{l} F_i[j] = E_i[j] \oplus mb_i[j] \quad (i = 0, 9 \ldots 16)\,, \\ E_i[j] = D_i[j] \oplus D_{i+1}[j] \oplus D_{i+4}[j]\,, \\ D_i[j] = C_{7i}[j + s(i)]\,, \\ C_i[j] = B_i[j] \oplus B_{i+1}[j]B_{i+2}[j] \oplus B_{i+2}[j] \oplus 1\,. \end{array} \tag{5.20}$$

Using these equations, for example the condition $F_{10}[j] = 0$ can be transformed into the following equation in the unknowns B_2, B_3 and B_4:

$$\begin{array}{ll} & B_2[j+23] \oplus B_3[j+23]B_4[j+23] \oplus B_4[j+23] \\ \oplus & B_9[j+2] \oplus B_{10}[j+2]B_{11}[j+2] \oplus B_{11}[j+2] \\ \oplus & B_{13}[j+9] \oplus B_{14}[j+9]B_{15}[j+9] \oplus B_{15}[j+9] \\ \oplus & mb_{10}[j] \oplus 1 = 0\,. \end{array} \tag{5.21}$$

It is because of the different bit rotations in this equation (and others), that we need to solve the equations bit per bit.

In order to simplify the system of nine equations in $B_1, \ldots B_8$, we impose an extra condition $B_0[j] = 0$, which can be transformed into the equation:

$$\begin{array}{ll} & L_0[j] \oplus L_1[j]L_2[j] \oplus L_2[j] \\ \oplus & L_7[j+1] \oplus L_8[j+1]L_9[j+1] \oplus L_9[j+1] \\ \oplus & L_{11}[j+10] \oplus L_{12}[j+10]L_{13}[j+10] \oplus L_{13}[j+10] \\ \oplus & mb_0[j] \oplus 1 = 0\,, \end{array} \tag{5.22}$$

in the unknowns L_1, L_2, L_7 and L_8. Starting from an arbitrary but specified state K and buffer, the values $L_0, L_9, \ldots L_{16}$ are fixed, and condition (5.22) can easily be satisfied by picking random values for $L_1, \ldots L_6, L_8$ and computing the required value of L_7. This determines message block $P0$, and allows us to compute the values of the states M, N, A and $B_0, B_9, \ldots B_{16}$.

Returning to the system of equations in $B_1, \ldots B_8$ we have already seen that there is a requirement on 32 bits (because the system is overdetermined). It turns out that there is another complication because by replacing $B_6[j]$ by $B_7[j] \oplus 1$ (see (5.19)), we end up with an equation in two bit positions of the same state word B_7, of the form:

$$B_7[j+27] \oplus B_7[j+1] = \ldots,$$

which translates to two requirements, one for bitwise addition of all values obtained with even j, and one for bitwise addition of all values obtained with odd j. This can be seen as a requirement on two bits. Hence the probability that a solution for this system of equations can be found is $1/2^{32+2}$. We can try random values of the initial state K until this happens and at that time message blocks $P0$ and $P1$ are determined. We can then compute the values of the states C, D, E and $F_0, F_9, \ldots F_{16}$, and choose a suitable value of message block $P2$ to satisfy (5.18) (the conditions on $F_1, \ldots F_8$).

The complexity to find sub-collision 1 (or 5) with this procedure is 2^{34} trials. Each trial requires one computation of the state updating transformation $\theta \circ \pi \circ \gamma$ (to bring the state from L to A). Furthermore, equation (5.22) is used to find a suitable set $L_1, \ldots L_8$; $B_0, B_9, \ldots B_{16}$ are computed (part of the σ transformation), and a few equations are used to determine if a suitable set $B_1, \ldots B_8$ can be found.

Producing the other sub-collisions

Sub-collision 2. In this case there are six conditions in the unknowns $F_1, \ldots F_8$, which can easily be met by choosing the message block $P2$. The seven remaining conditions on $F_0, F_9, \ldots F_{16}$, are added to the three existing conditions in the unknowns $B_1, \ldots B_8$, resulting in an overdetermined system of ten equations, which leads to a requirement on $2 \cdot 32 = 64$ bits. The system is simplified by specifying $B_0[j] = B_9[j] = 0$. This leads to two extra conditions in the unknowns $L_1, \ldots L_8$, which are easily satisfied with a suitable choice of message block $P0$. It turns out that there are two more requirements that need to be satisfied in order for a solution to exist for the system of equations in the unknowns $B_1, \ldots B_8$: one requirement on two bits (similar to the case for sub-collision 1), and another requirement on sixteen bits. This last requirement comes from a non-linear equation in the unknown B_8 of the following form:

$$B_7[j+27]B_8[j+27] \oplus B_8[j+27] = \ldots,$$

where B_7 has already been solved. It can be seen that for $B_7[j+27] = 0$ we can compute $B_8[j+27]$, but for $B_7[j+27] = 1$ all unknowns drop from the equation. On average this happens for sixteen bits and we end up with a requirement that has to be satisfied in order to get a solvable system of equations. Hence the probability that a solution can be found is $1/2^{32+32+2+16}$, and the complexity for finding this sub-collision is 2^{82} trials.

Sub-collision 3. There are seven conditions in the unknowns $F_1, \ldots F_8$, which can easily be met by choosing the message block $P2$. The seven remaining conditions on $F_0, F_9, \ldots F_{16}$, lead to seven equations in the unknowns $B_1, \ldots B_8$. We simplify the system by specifying $B_0[j] = 0$. Added to the three existing conditions in $B_9, \ldots B_{16}$, this leads to a system of four equations in the unknowns $L_1, \ldots L_8$, which are easy to solve with a suitable choice of the message block $P0$. The system of equations in $B_1, \ldots B_8$ can be solved with two requirements on 16 and 32 bits respectively. The 16-bit requirement is due to the non-linearity (similar to the case for sub-collision 2), and the 32-bit requirement comes from the fact that one equation specifies the same unknown as a set of three other equations. The complexity for finding this sub-collision is $2^{16+32} = 2^{48}$ trials.

Sub-collision 4. This case is similar to the previous one, except there are only four conditions in the unknowns $F_1, \ldots F_8$, and the system of four equations in the unknowns $L_1, \ldots L_8$ imposes another requirement of 32 bits, which raises the complexity to $2^{16+32+32} = 2^{80}$ trials.

Complexity of the attack

As seen above the complexities for finding the five sub-collisions correspond to 2^{34}, 2^{82}, 2^{48}, 2^{80} and 2^{34} trials respectively. There is no problem in connecting the sub-collisions, as we only need an arbitrary initial state for each, which can be obtained by choosing random message blocks between the sub-collisions. So the total complexity for our collision-finding attack is determined by sub-collision 2 (which is the most difficult to find). This complexity is 2^{82} trials which corresponds to about 2^{82} computations of the transformation $\sigma \circ \rho$. An outline of the attack has been given in Table 5.3 above. Note that an arbitrary number of common message blocks (with zero difference) can be hashed before block m^0 (this defines a random value for state K in sub-collision 1 in our attack). Furthermore, an arbitrary number of common message blocks can be hashed after block m^{35} (this defines the final hash result).

We tested the attack for a reduced version of PANAMA where all the words have a length of eight bits instead of 32 bits (this version has 64-bit hash results), and this confirmed the given complexities (for the 8-bit version the complexity of the attack is about 2^{22} trials). Sub-collisions 1, 3 (and 5) were also tested for a 16-bit version which again confirmed the complexity.

The complexity of our attack is too high to actually find collisions for Panama, but it is much faster than a generic birthday attack which would require about 2^{128} hash computations. We believe that improvements to our attack are possible. First of all better methods to solve the systems of non-linear equations can be looked for. Furthermore, our attack still has a lot of freedom. We can compute further backwards which has the advantage that we get more unknowns, but the difficulty that we get more complex non-linear equations.

5.3.5 Improving PANAMA

In order to improve the security of the Panama hash function, the design could be altered in such a manner that the message influences a smaller part of the state (e.g., exor only four message words into the state at every application of σ). Instead, more buffer words could be used as input of σ. The consequence for the attacker would be that he has a smaller degree of freedom, and therefore needs to go further backwards in the attack to obtain enough unknowns, which gives him more complex equations to solve. Preferrably, the different buffer stages that would be used as input of σ, should be selected in such a way that the different sub-collisions can no longer be treated independently. Of course, changing the design of Panama in this manner would decrease the performance of the algorithm. Furthermore, there still is a fundamental problem that we have the freedom to choose message blocks, and are able to go backwards in the attack by choosing message blocks before the sub-collisions.

5.4 Conclusions

In this chapter we have discussed the design of the Panama cryptographic module, and we have presented a method for producing collisions for the Panama hash function. This attack has been published in [112]. Our result has no impact on the security of the Panama stream cipher. In the stream cipher mode of Panama two push operations are used to load a 256-bit secret key and a 256-bit diversification parameter. Next 32 pull operations are performed where the outputs are discarded, and finally an arbitrary number of pull operations to generate the keystream (256 bits at every step). No independent cryptanalysis of this stream cipher mode has been published so far.

Chapter 6

Design of Message Authentication Codes

6.1 Introduction

In this chapter we discuss the design of message authentication codes. First the required output length and key length for these algorithms is considered. All known constructions are based on the iteration of a compression function with fixed size input, in the same way as for cryptographic hash functions (see Chapter 3). We explain that for an iterated construction one needs to consider forgery attacks based on internal collisions. The most common approach of designing MAC algorithms is to base the compression function on an existing cryptographic primitive, either a block cipher or an unkeyed hash function. The main part of the chapter and a contribution of ours is the proposal of a new design, called Two-Track-MAC. We published this design in [37], and submitted it to the NESSIE project [124], which selected it as part of its portfolio of cryptographic algorithms.

6.2 Output and Key Length

For a MAC algorithm the output length (that is, the length of the MAC results) may be shorter than for a cryptographic hash function, because the security requirements are different. The relevance of attacks where the adversary guesses the MAC result (see Sect. 2.3.2) depends strongly on the application: one has to consider the total number of trials a system is subject to during its lifetime, and the consequences of a successful forgery. Output lengths of $n = 32, 64, \ldots$

bits may be sufficient in practice. The key length must be chosen appropriately to protect against exhaustive key search attacks (Sect. 2.3.2). Considering the analysis of M. Blaze *et al.* [23] (see Sect. 3.2), a length of at least $k = 80$ bits is recommended.

6.3 The Iterated Model

Practical MAC constructions are based on the same iterated model as the one defined for cryptographic hash functions, see Sect. 3.3. The difference is that the secret key may be used in the initial value, in the compression function, and/or in the output transformation. For a message input X that consists of t blocks $X_0, X_1, \ldots, X_{t-1}$ (after padding), the iterative computation for a key K can be described as follows:

$$\begin{aligned} H_0 &= IV_K \ , \\ H_{i+1} &= f_K(H_i, X_i) \quad \text{for } 0 \le i < t \ , \\ h(K, X) &= g_K(H_t) \ . \end{aligned}$$

A padding rule must be defined to make the length of the message input a multiple of the block length. Contrary to hash functions, an output transformation is frequently applied. The output length n is less than or equal to the length c of the chaining variable. A general security result, due to M. Bellare *et al.* [11], shows that iteration of a finite length pseudo-random function provides a pseudo-random function with arbitrary length inputs.

6.3.1 Forgeries based on internal collisions

For iterated MAC constructions one needs to consider attacks based on internal collisions. A collision for a MAC consists of a pair of message inputs X, X' with the same MAC result $h(K, X) = h(K, X')$. Note that $h(K, X) = g_K(H_t)$ and $h(K, X') = g_K(H'_t)$. *Internal collisions* are those that occur before the output transformation, which implies that $H_t = H'_t$. For *external collisions* $H_t \neq H'_t$ but $g_K(H_t) = g_K(H'_t)$.

In [106] B. Preneel and P. van Oorschot show that an internal collision can be used to obtain a verifiable MAC forgery with a chosen text attack that requires only a single requested MAC result. The attack is based on the following observation: given an internal collision pair X, X' and an arbitrary message block X_a we have $h(K, X \| X_a) = h(K, X' \| X_a)$. Therefore requesting the MAC result for the single chosen text $X \| X_a$ permits forgery, as the MAC result for $X' \| X_a$ is the same (so it can be 'computed' without knowledge of K). The following theorem indicates the difficulty of finding an internal collision for use in this attack:

Theorem 6.1 (Preneel-van Oorschot) *Let h be an iterated MAC algorithm with c-bit chaining variable and n-bit output. An internal collision for h can be found using u known text-MAC pairs and v chosen texts. The expected values for u and v are as follows: $u = \sqrt{2}\cdot 2^{c/2}$ and $v = 0$ if the output transformation g_K is a permutation. If g_K is not a permutation, v is approximately $2\cdot(2^{c-n}+\lfloor\frac{c-n}{n-1}\rfloor+1)$.*

The required number u of known text-MAC pairs follows from a simple birthday attack argument. One can only check if a collision occured by comparing the MAC results. If the output transformation g_K is a permutation (e.g., the identity transformation) all collisions are necessarily internal collisions, so no further work is needed. In the case that g_K is not a permutation ($c > n$), one needs to check for every collision whether it is an internal or an external one. This can be done by appending the same string to both message inputs of a collision pair and checking whether the corresponding MAC results are equal (this requires two additional chosen-text queries). For more details we refer to [106]. There it is also shown that the attack can be further optimised if the set of text-MAC pairs has a common sequence of s trailing blocks. Note that the attack cannot be precluded by including the length of the message input into the padding bits. It can be precluded by making the algorithm non-deterministic. This means that every MAC computation should be different, e.g., by including a sequence number or a random number in the computation.

The attack described above can be applied to any iterated MAC algorithm, and the complexity depends on the length c of the chaining variable and the length n of the output. Although the attack improves upon the complexity of the generic attacks described in Sect. 2.3.2, it must be noted that it is more difficult in practice than the birthday attack on unkeyed hash functions. The reason is that the attacker needs an on-line interaction with the key owner who must produce the MAC results for a huge number of messages (note also that this does not allow for parallelisation). For values of $c \geq 128$ bits the attack becomes totally infeasible.

6.4 MACs Based on Block Ciphers

The compression function of an iterated MAC algorithm can be based on an existing cryptographic primitive. The most common way of basing it on a block cipher is by using the cipher in CBC-mode (cipher block chaining, see Chapter 7, Sect. 7.2.3). This means that the MAC key is used as cipher key in each step of the iteration, and the message block to be processed in the current step serves as plaintext input to the cipher, after being added (modulo 2) to the ciphertext output from the previous step. If we write the encryption operation as $Y = E_K(X)$ (where X denotes the plaintext, Y the ciphertext, and K the key), then the compression function of the CBC-MAC construction can be expressed as follows (see also Fig. 6.1):

$$\text{CBC-MAC: } H_{i+1} = f_K(H_i, X_i) = E_K(H_i \oplus X_i)\,.$$

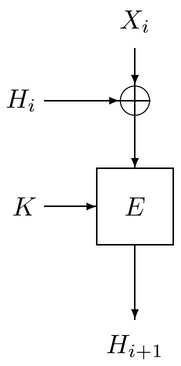

Figure 6.1: Compression function for CBC-MAC.

Note that the initial value for the computation is usually chosen as zero, so $H_0 = IV = 0$. For this construction the length of the chaining variable is equal to the block length of the cipher that is used, that is $c = b$. Contrary to hash functions based on block ciphers (Sect. 3.4), the key schedule of the cipher needs to be computed only once (for every authentication key K).

CBC-MAC is often defined with an additional output transformation g which consists of truncating the final chaining value H_t to the n leftmost bits ($n \leq c$). For the choice $n = c$ (g is the identity transformation), the complexity of a forgery attack based on internal collisions (Sect. 6.3.1) corresponds to a single chosen message and about $2^{c/2}$ known messages. If one chooses $n = c/2$ an additional $2^{c/2}$ chosen messages are needed. Hence, truncating the output of the CBC-MAC algorithm makes internal collision based attacks more difficult. There is a trade-off however because the probability of guessing the MAC result will increase.

A security proof for CBC-MAC has been given by Bellare *et al.* [12]. This proof provides a lower bound on the complexity of breaking the algorithm under the assumption that the underlying block cipher is a pseudo-random permutation. This lower bound is close to the upper bound provided by Theorem 6.1 (the complexity of an attack based on internal collisions). However, it is important to note that the security proof is only valid when the CBC-MAC algorithm is used to authenticate messages of a fixed length. If the messages can have a variable length, existential forgeries can be found with a so-called *exor-forgery* attack. The easiest variant of this attack requires only a single known text-MAC pair. We consider the CBC-MAC algorithm with g chosen as the identity transformation, and we assume that the attacker knows a pair X, Y where X is a message input of a single block and Y is the corresponding MAC result: $Y = h(K, X) = f_K(IV, X)$. Then it follows that $h(K, X \,\|\, (X \oplus Y \oplus IV)) = Y$.[1] This implies that the attacker can construct a new message $X \,\|\, (X \oplus Y \oplus IV)$ with the same MAC result, which is a forgery. If the algorithm uses a padding rule which appends the length of the input, a simple extension of the exor-forgery attack can be used. Truncation of the MAC result to the n leftmost bits ($n < c$) is also insufficient: Knudsen [72] has shown that in this case a variant of the attack is possible, requiring $2^{(c-n)/2}$ chosen texts. The use of a strong output transformation g precludes the exor-forgery attack, an example is an output transformation which computes the MAC result Y from the final chaining value H_t by encryption under a key K', that is $Y = E_{K'}(H_t)$. The secondary key K' is usually derived from the key K and the resulting algorithm is known as EMAC (Extended-MAC). A security proof for this construction is given by E. Petrank and C. Rackoff [95], and contrary to the plain CBC-MAC it can be used for messages of variable length. Note that if one chooses the keys K and K' independently for EMAC, this does not raise the level of security because a *divide-and-conquer* approach allows an attacker to perform separate key recoveries on these two keys (see Sect. 6.5 for more information on divide-and-conquer attacks). With respect to performance, and comparing to the plain CBC-MAC, the EMAC construction needs to perform an additional encryption (the output transformation), and the key schedule needs to be computed twice (for the keys K and K').

An alternative construction is OMAC (One-key MAC), proposed by T. Iwata and K. Kurosawa [67]. OMAC is also based on the CBC-mode, has a security proof (for variable-length messages) and better performance than EMAC: it does not require an additional encryption as output transformation, neither a second computation of the key schedule. Compared to the plain CBC-MAC this construction requires that some key-derived material is added to the input $H_{t-1} \oplus X_{t-1}$ of the last iteration step. Another provably secure and efficient alternative is PMAC, proposed by J. Black and P. Rogaway [21]. PMAC has the advantage of being parallelisable: all block cipher invocations (for different message blocks) can be computed at the same time. This is in contrast to the CBC-based constructions EMAC and OMAC, which are inherently sequential.

[1] This is because $H_1 = f_K(IV, X) = E_K(IV \oplus X) = Y$ and $H_2 = f_K(Y, X \oplus Y \oplus IV) = E_K(Y \oplus (X \oplus Y \oplus IV)) = E_K(X \oplus IV) = Y$.

6.5 MACs Based on Hash Functions

As an alternative to the use of a block cipher, a cryptographic hash function can be used as the basis of a MAC algorithm. Below we first review some of the early proposals for hash function based MACs, and their weaknesses. Next we discuss the most popular constructions, HMAC and MDx-MAC.

In the past it has been suggested to simply use an unkeyed hash function h_u and include the secret key in the input. For example one could prepend K to the message X, resulting in the construction $h_1(K, X) = h_u(K\|X)$. Suppose now that an attacker knows the MAC result Y corresponding to a message X, that is $Y = h_u(K\|X)$. Then he is able to compute the MAC result for a message $X\|X_a$, with arbitrary block X_a, because $h_u(K\|X\|X_a) = f_u(Y, X_a)$. Here f_u is the underlying compression function, Y serves as chaining value for the further computation and no secret key is involved. Note that the use of MD-strengthening (inclusion of the length in the padding bits) does not prevent such an *extension attack*. An alternative construction appends the key to the message, that is $h_2(K, X) = h_u(X\|K)$. This can be seen as an iterated MAC algorithm where only the output transformation depends on the secret key. An attacker might use a *birthday attack* to find collisions for the unkeyed hash function h_u, that is he determines two messages X and X' such that $h_u(X) = h_u(X')$. This can be done off-line and is independent of the key. If the attacker subsequently obtains the MAC result corresponding to X (one chosen-text query), he also knows the MAC result for X' which is the same because $h_u(X\|K) = h_u(X'\|K)$.

The envelope method addresses the weaknesses discussed above. For this construction a key K_1 is prepended and another key K_2 is appended to the message input X, that is $h_3(K_1\|K_2, X) = h_u(K_1\|X\|K_2)$. Usually, the key K_1 is extended with padding bits to the length of one block, and the second key K_2 is derived from K_1. Bellare *et al.* [11] provide a security proof for this construction based on the assumption that the compression function of the hash function is pseudo-random. This is an interesting result but it must be noted that the compression function of a hash function is usually evaluated with respect to preimage and collision-resistance, and not with respect to pseudo-randomness. If K_1 and K_2 are chosen independently, and assuming both have a length of k bits, there exists a *divide-and-conquer* attack which recovers both keys with a time complexity of 2^k operations, rather than the expected complexity of 2^{2k}. This attack was demonstrated by Preneel and van Oorschot [104]. The main idea is to first generate internal collisions, using the technique described in Sect. 6.3.1, and then perform an exhaustive search on the key K_1. All guesses for K_1 which do not result in an internal collision are eliminated, and $\lceil k/c \rceil$ different internal collisions are sufficient to determine K_1 uniquely. Finally, the key K_2 can be found with another (separate) exhaustive search. Although this attack is impractical (it requires a large number of known text-MAC pairs to find internal collisions), it shows that the strength of the construction comes from its individual keys rather than from their combined length. An improved key recovery attack on a specific instance of the envelope method, based on the hash function MD5, was demonstrated by Preneel and van Oorschot in [105]. Some properties of the padding procedure of MD5 are exploited together with the fact that the key K_2 may be divided over the last two input blocks (note that the attack works also for other MDx-class hash functions used in a similar envelope construction). This attack is not practical but it represents a certificational weakness for the envelope method.

6.5.1 The HMAC construction

HMAC, proposed by Bellare *et al.* [15], uses a nested construction and two keys K_1 and K_2 that are first extended (with zero bits) to the length of a block. For an underlying hash function h_u it computes the MAC result in the following manner:

$$h(K_1 \| K_2, X) = h_u(K_2 \| h_u(K_1 \| X))\,.$$

The keys K_1 and K_2 are usually dependent on each other (a specific method is given in [15]); if they are independent a divide-and-conquer attack applies, just as for the envelope method. The output length of HMAC is equal to the output length of the unkeyed hash function that is used (n_u bits). Optionally, the HMAC outputs may be truncated to the n leftmost bits ($n \leq n_u$). With respect to performance it may be noted that for a message input X that consists of t blocks (after padding as defined for the hash function), $t+3$ computations of the compression function are needed (assuming that the length of $h_u(K_1 \| X)$, that is the output length of the hash function, is smaller than the length of one block). However, if one precomputes the values $f_u(IV, K_1)$ and $f_u(IV, K_2)$ this can be reduced to $t+1$ computations of the compression function. Note that the extra computation represents the output transformation.

There is a security proof for HMAC which relies on weaker assumptions than those needed for the security proof of the envelope method. In particular it is shown in [15] that the HMAC construction is secure if the following holds: the hash function h_u is collision-resistant when the initial value is secret; the compression function f_u that underlies the hash function h_u is a secure MAC algorithm for messages of one block (with the secret key in the H_i input, and the message in the X_i input); the values $f_u(IV, K_1)$ and $f_u(IV, K_2)$ cannot be distinguished from random (f_u is a 'weak' pseudo-random function, 'weak' because the values K_1 and K_2 are secret).

6.5.2 The MDx-MAC construction

MDx-MAC, proposed by Preneel and van Oorschot [104], is a construction that can be based on the hash function MD5 or similar hash functions such as RIPEMD-128, RIPEMD-160 or SHA-1 (see Chapter 4). The underlying hash function (MDx) is converted into a MAC by some small modifications. The secret key K is first expanded to three 128-bit subkeys K_0, K_1, K_2 (for RIPEMD-160 or SHA-1 the subkey K_0 is 160 bits long). MDx-MAC is then obtained from MDx with the following modifications:

1. The initial value IV of MDx is replaced by the subkey K_0.

2. The subkey K_1 is split into four 32-bit values. These values are added to the additive constants used in the different rounds of each iteration of the MDx compression function (see [104] for details).

3. After the block that contains the padding bits (as defined for MDx) an additional complete block is appended. The block is derived from the subkey K_2 (see [104] for details). The processing of this block represents the output transformation.

4. The MAC result may optionally be truncated to the n leftmost bits ($n \leq 128$ for MD5/RIPEMD-128, $n \leq 160$ for RIPEMD-160/SHA-1).

With respect to performance it may be noted that for a message input of t blocks (after padding as defined for the hash function), $t + 1$ computations of the compression function are needed. The derivation of the subkeys from the key K requires six computations of the MDx compression function (see [104] for details). In order to use MDx-MAC the code of an existing implementation of the MDx hash function must be modified. In contrast, both the envelope method and HMAC are black-box constructions which can use an existing hash function implementation with no modifications.[2]

Similar to the envelope method, the security of MDx-MAC can be proven under the assumption that the compression function is pseudo-random. The use of secret key material in every iteration of the compression function is meant to make this assumption more plausible. Note that MDx-MAC is not vulnerable to the certificational attack of [105] on the envelope method, because the subkey K_2 is expanded to a complete final block.

[2]For HMAC some modifications to the code of the hash function (but not to the compression function) are needed for precomputation of the values $f_u(IV, K_1)$ and $f_u(IV, K_2)$.

6.6 Other MAC Constructions

Contrary to hash functions, very few dedicated MAC algorithms are known. This is partly because the constructions based on block ciphers or hash functions seem adequate, and partly because some algorithms are proprietary and not made public. One example of a known dedicated MAC is the former ISO standard MAA (Message Authenticator Algorithm), proposed by D. Davies [33]. Significant weaknesses were shown for this algorithm by Preneel *et al.* [103]. Below we describe two special constructions, and in Sect. 6.8 we will discuss our own design Two-Track-MAC, which is derived from the RIPEMD-160 hash function.

6.6.1 MACs based on universal hashing

A family of hash functions $H = \{h : \mathcal{D} \to \mathcal{R}\}$ is a finite set of functions with common domain $\mathcal{D}$ and (finite) range $\mathcal{R}$. We may also denote this by $H : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $H_K : \mathcal{D} \to \mathcal{R}$ is a function in the family for each $K \in \mathcal{K}$ (the key space). One chooses a random function h from the family by choosing $K \in \mathcal{K}$ uniformly and letting $h = H_K$.

A universal hash function family is a family of hash functions with some combinatoric property. For example, a hash function family $H = \{h : \mathcal{D} \to \mathcal{R}\}$ is *ϵ-almost-universal* if for any distinct $X, X' \in \mathcal{D}$, the probability that $h(X) = h(X')$ is no more than ϵ, when $h \in H$ is chosen at random. This notion was introduced by J. Carter and M. Wegman [28] and it can be used for message authentication, for example by hashing a message with a function drawn from a universal hash function family and encrypting the output of the hash function with a block cipher (the encrypted hash output serves as MAC result). The combinatoric property of the universal hash function family is often not difficult to prove, and it can be shown that the security of the resulting MAC scheme depends on the security of the cipher that is used. The UMAC algorithm [76] is an example of a MAC based on universal hashing. It has very good performance, although the efficiency decreases significantly in the case of short messages and the algorithm is complex to specify or implement.

6.6.2 MAC based on PANAMA

The Panama stream/hash module has been described and analysed in Chapter 5. In [30] its authors suggest that the Panama hash function can also be used as a MAC algorithm by simply including a secret key in the message input. If the hash function is secure, the resulting MAC scheme should also be secure independent of the positions of the key bits in the message input. For example, we may consider a scheme that computes a MAC result for a message input X and secret key K by appending the key to the message and computing the 256-bit PANAMA hash: $h(K, X) = h_u(X \| K)$.

It is easy to show that our collision attack on the hash mode of PANAMA (see Sect. 5.3) leads to a forgery attack of comparable complexity on this MAC scheme. One first generates a collision for two messages X and X': $h_u(X) = h_u(X')$. Note that this is an internal collision: all contents of state and buffer before the output transformation are equal for the two messages. This remains so when a common (unknown) value K is appended to the messages. Therefore, if we request the MAC result for message X, we also know the MAC result for a 'new' message X': $h(K, X') = h(K, X)$. The attack can be performed off-line and independent of the key. A single chosen text query is needed.

6.7 Standardisation of Algorithms

Several organisations have taken initiatives for standardisation of message authentication codes. ISO/IEC has developed standard 9797 for MAC algorithms, with two separate parts. Part 1 of ISO/IEC 9797 [65] describes MACs based on an (unspecified) block cipher, more specifically the CBC-MAC construction and a number of variants including EMAC. Part 2 of ISO/IEC 9797 [66] details MACs based on an (unspecified) hash function, more specifically the HMAC and MDx-MAC constructions (and a variant of MDx-MAC to be used for short input strings). ANSI has adopted the CBC-MAC construction based on the DES block cipher in its retail banking standard X9.19 [2]. This includes an optional output transformation, which implies that the last message block is encrypted under triple-DES. The resulting algorithm is widely known as the retail-MAC. ANSI has also specified the HMAC construction (for an unspecified hash function) in standard X9.71 [5]. NIST has developed FIPS 113 [48] for DES-based CBC-MAC and FIPS 198 [53] for HMAC based on the SHA-1 hash function. Recently, NIST announced that it intends to propose the OMAC construction based on the AES block cipher as a possible future standard [121].

We also mention the efforts of the NESSIE project here (see Sect. 1.2.1). NESSIE requested the submission of message authentication codes in its call for cryptographic primitives. Two algorithms were submitted: UMAC and our proposal Two-Track-MAC (see the description in the next section). Further, NESSIE evaluated the standard constructions EMAC and HMAC. All of these algorithms are recommended by NESSIE and are included as part of the NESSIE portfolio of cryptographic primitives.

6.8 A New MAC Based on RIPEMD-160

In Sect. 6.5 we have discussed several constructions of MACs based on an existing cryptographic hash function. Now we present our design of a new message authentication code based on the two-trail construction that underlies the hash function RIPEMD-160 (see Sect. 4.6). Our algorithm, called Two-Track-MAC (TTMAC in short), has been submitted as a candidate algorithm for the NESSIE project [124], and in February 2003 it was announced that TTMAC is selected as part of the NESSIE portfolio of recommended cryptographic primitives. The design has also been published in [37]. Below we describe TTMAC, discuss its security and analyse the strength against known attack strategies. We also look at performance considerations and give motivations for the use of our design instead of other constructions based on a hash function (HMAC and MDx-MAC).

6.8.1 Description of Two-Track-MAC

In our description below we first consider the special case where the padded message input consists of a single block of 512 bits, next we consider arbitrary length messages and we conclude with some more details and observations on the design. Note that we only give a high level description here. The internal details of the two trails used in the compression function can be derived from the source code given in Appendix A. There we also provide test vectors for TTMAC.

MAC computation for single-block messages

The unkeyed hash function RIPEMD-160 (see Sect. 4.6 and [100]) uses two trails in its compression function. If we separate those two trails then each trail can be seen as a transformation of a 160-bit input I, controlled by a message block M consisting of sixteen words of 32 bits. The 160 bits of the input I (and of the output) consist of five words of 32 bits. Call the output of the different trails $\mathcal{L}(I, M)$ and $\mathcal{R}(I, M)$ (left respectively right trail output for an input I and a message block M), then our proposal for a MAC on a relatively short message M (with a length of 512 bits after padding) and a key K of 160 bits is (in short notation):

$$\mathrm{TTMAC}(K, M) = \mathcal{R}(K, M) - \mathcal{L}(K, M).$$

In more detail, suppose that we have a key consisting of five 32-bit words: $K = (A_K, B_K, C_K, D_K, E_K)$. Then this key serves as initial value for both the left and right trails (these are identical to the trails used in the compression function of RIPEMD-160). The message block M consists of sixteen 32-bit words W_j $(0 \leq j < 16)$. Now let $(A_L, B_L, C_L, D_L, E_L)$ denote the output from the left trail, and let $(A_R, B_R, C_R, D_R, E_R)$ denote the output from the right trail (these outputs are obtained after five rounds of 16 step operations each). Then the MAC result is defined by the five-word value:

$$(A, B, C, D, E) = (A_R - A_L, B_R - B_L, C_R - C_L, D_R - D_L, E_R - E_L)\,,$$

where all subtractions are performed modulo 2^{32}. Figure 6.2 below gives an outline of this computation. It differs from a RIPEMD-160 unkeyed hash computation in two points (for a single-block message):

1. Instead of using the initial value specified for RIPEMD-160, the two trails are initialised with the secret key $(A_K, B_K, C_K, D_K, E_K)$.
2. At the end we use a different procedure for combining the results of the separate trails.

MAC computation for longer messages

If the padded message is longer than one block, i.e., $M = M_0 \| M_1 \| \cdots \| M_{t-1}$ where each block M_i has a length of 512 bits, we define two new operations $\mathcal{L}^*$ and $\mathcal{R}^*$. The operation $\mathcal{L}^*$ is based on the operation $\mathcal{L}$ (left trail), which had a straightforward inverse operation on the first (160 bits long) argument. This new operation $\mathcal{L}^*$ has a simple feed-forward with the first argument:

$$\mathcal{L}^*(I, M_i) = \mathcal{L}(I, M_i) - I\,,$$

this is five times a subtraction modulo 2^{32}. Similarly the operation $\mathcal{R}^*$ is defined in shorthand as

$$\mathcal{R}^*(I, M_i) = \mathcal{R}(I, M_i) - I\,.$$

As before, the secret key $(A_K, B_K, C_K, D_K, E_K)$ serves as initial value for the two trails. We compute two 160-bit values (5 words each) as follows:

$$\begin{aligned}(A_L, B_L, C_L, D_L, E_L) &= \mathcal{L}^*((A_K, B_K, C_K, D_K, E_K), M_0)\,,\\ (A_R, B_R, C_R, D_R, E_R) &= \mathcal{R}^*((A_K, B_K, C_K, D_K, E_K), M_0)\,.\end{aligned}$$

Now we introduce a transformation $\mathcal{X}$ which takes $(A_L, B_L, C_L, D_L, E_L)$ and $(A_R, B_R, C_R, D_R, E_R)$ as inputs, and which produces two 5-word values, denoted $(A_1, B_1, C_1, D_1, E_1)$ and $(A_2, B_2, C_2, D_2, E_2)$, as outputs, where:

$$\begin{aligned}A_1 &= (B_L + E_L) - D_R\,,\\ B_1 &= C_L - E_R\,,\\ C_1 &= D_L - A_R\,,\\ D_1 &= E_L - B_R\,,\end{aligned}$$

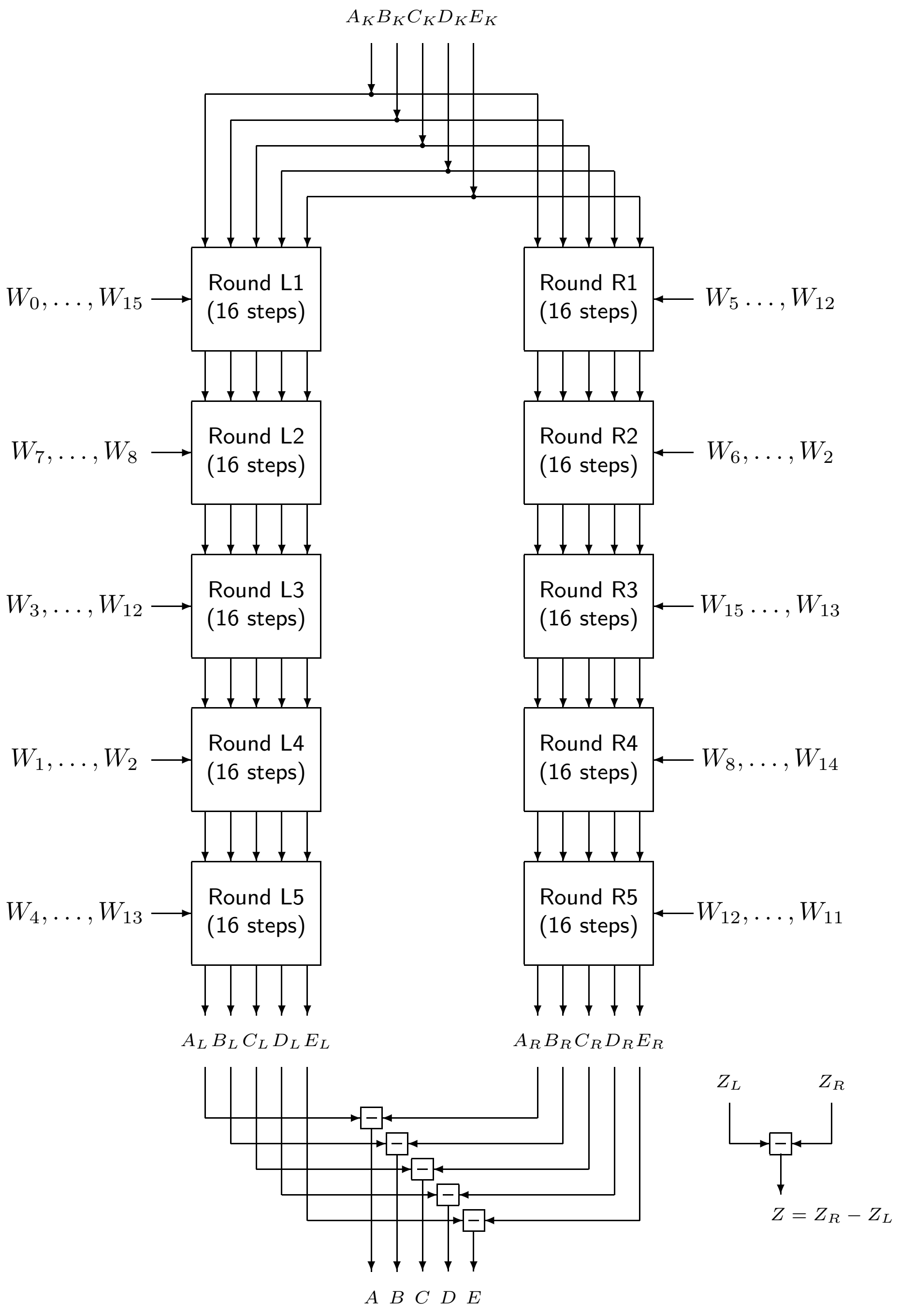

Figure 6.2: Outline of TTMAC for a message of a single block.

$$\begin{aligned} E_1 &= A_L - C_R\,; \\ \\ A_2 &= D_L - E_R\,, \\ B_2 &= (E_L + C_L) - A_R\,, \\ C_2 &= A_L - B_R\,, \\ D_2 &= B_L - C_R\,, \\ E_2 &= C_L - D_R\,. \end{aligned}$$

Here all subtractions and additions are modulo 2^{32}. The obtained 5-word values $(A_1, B_1, C_1, D_1, E_1)$ and $(A_2, B_2, C_2, D_2, E_2)$ are the starting values for the left, respectively, right trail to incorporate the next 512-bit message block M_1. If there are more message blocks M_i the iteration is the same. This leads to the following algorithm:

$$\begin{aligned} (A_L, B_L, C_L, D_L, E_L) &= \mathcal{L}^*((A_K, B_K, C_K, D_K, E_K), M_0)\,, \\ (A_R, B_R, C_R, D_R, E_R) &= \mathcal{R}^*((A_K, B_K, C_K, D_K, E_K), M_0)\,; \end{aligned}$$

$$\begin{gathered} ((A_1, B_1, C_1, D_1, E_1), (A_2, B_2, C_2, D_2, E_2)) = \\ \mathcal{X}((A_L, B_L, C_L, D_L, E_L), (A_R, B_R, C_R, D_R, E_R))\,. \end{gathered}$$

From then on iteratively, for $i = 1, ..., t-2$:

$$\begin{aligned} (A_L, B_L, C_L, D_L, E_L) &= \mathcal{L}^*((A_1, B_1, C_1, D_1, E_1), M_i), \\ (A_R, B_R, C_R, D_R, E_R) &= \mathcal{R}^*((A_2, B_2, C_2, D_2, E_2), M_i); \end{aligned}$$

$$\begin{gathered} ((A_1, B_1, C_1, D_1, E_1), (A_2, B_2, C_2, D_2, E_2)) = \\ \mathcal{X}((A_L, B_L, C_L, D_L, E_L), (A_R, B_R, C_R, D_R, E_R))\,. \end{gathered}$$

For the last message block M_{t-1} however, the role of the left and right trails is interchanged. Moreover, the two trails are combined to produce the 5-word (160-bit) MAC result TTMAC$(K, M) = (A, B, C, D, E)$:

$$\begin{aligned} (A_L, B_L, C_L, D_L, E_L) &= \mathcal{L}^*((A_2, B_2, C_2, D_2, E_2), M_{t-1}), \\ (A_R, B_R, C_R, D_R, E_R) &= \mathcal{R}^*((A_1, B_1, C_1, D_1, E_1), M_{t-1}); \end{aligned}$$

$$(A, B, C, D, E) = (A_R - A_L, B_R - B_L, C_R - C_L, D_R - D_L, E_R - E_L)\,.$$

Note again that the subtractions are modulo 2^{32}. Figure 6.3 below gives an outline of the procedure (here Z_* refers to the set $(A_*, B_*, C_*, D_*, E_*)$).

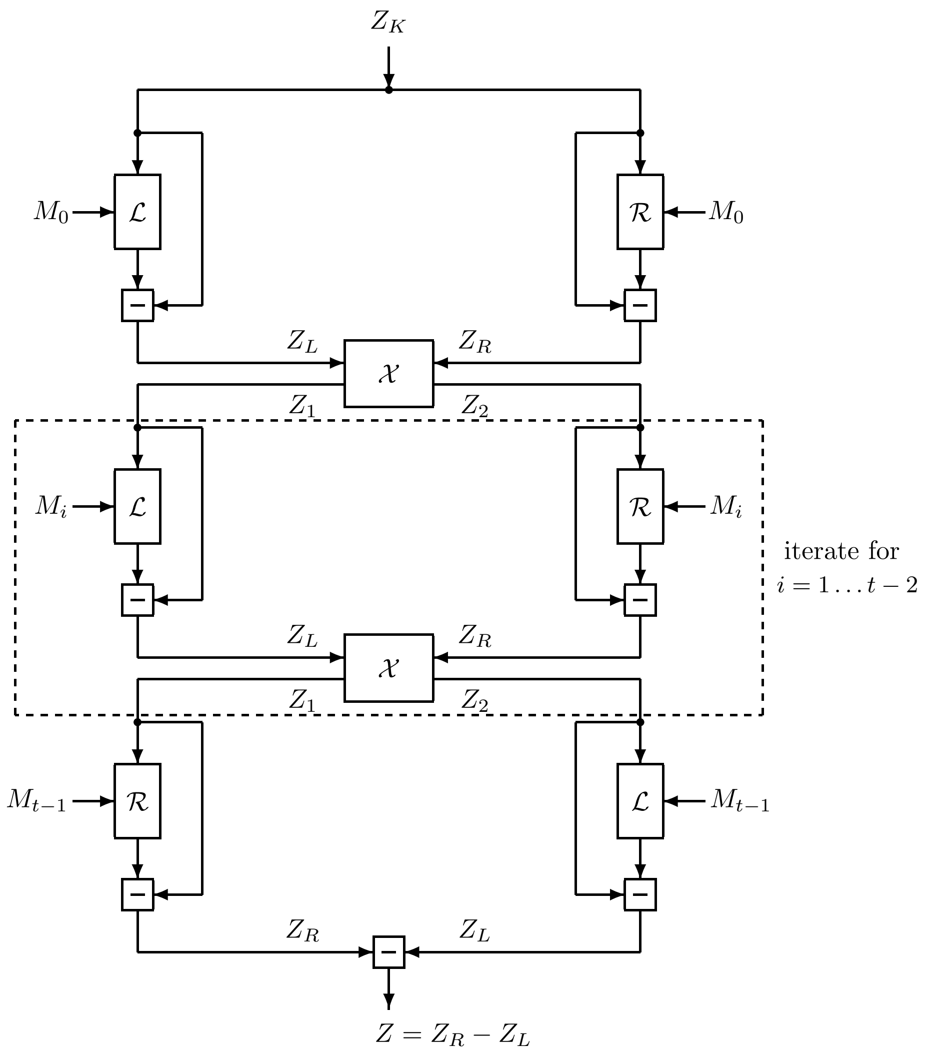

Figure 6.3: High level view of TTMAC for a message of arbitrary length.

Message pre-processing

The same pre-processing rules as in the RIPEMD-160 hash function are used to format the message input to the algorithm. First the message is padded to a length which is a multiple of 512 bits. Assume that the message is r bits long.

Then first a single 1-bit is appended, followed by a number s of 0-bits, where $0 \leq s < 512$ and $r + 1 + s \equiv 448 \bmod 512$. Finally, a 64-bit representation of the original length r (mod 2^{64}) is appended (in the form of two 32-bit words with the least significant word first). The message is then divided into blocks of 512 bits each, and each block is converted to a sequence of sixteen 32-bit words using the little-endian convention (see Chapter 4). Likewise, the 160-bit secret key is converted to a sequence of five 32-bit words using the little-endian convention.

Output transformation

An optional output transformation can be used to reduce the length of the MAC result (to 32, 64, 96 or 128 bits). This transformation computes the necessary number of output words, in such a manner that all of the normal output words are used. Let the normal 5-word result be (A, B, C, D, E), then the final (shortened) MAC result is computed as follows (note that the additions are modulo 2^{32}).

– For a 32-bit MAC result we compute the word:

$$A_T = A + B + C + D + E\,.$$

– For a MAC result of 64, 96 or 128 bits we compute respectively the first two, the first three or all four of the following words:

$$\begin{aligned} A_T &= A + B + D\,, \\ B_T &= B + C + E\,, \\ C_T &= C + D + A\,, \\ D_T &= D + E + B\,. \end{aligned}$$

Finally the sequence of output words is converted into a string of 32, 64, 96, 128 or 160 bits, by means of the little-endian convention. For example, a 160-bit MAC result corresponds to the string derived from the concatenated 5-word value $A\|B\|C\|D\|E$, starting with the least significant byte of A and ending with the most significant byte of E.

Observations on the design

The main difference between TTMAC and RIPEMD-160, the hash function on which it is based, is that for TTMAC we omit the combination of the two trails at the end of the compression function. Hence the size of the chaining variable is doubled (320 bits instead of 160). This is in fact similar to the design of RIPEMD-320 (the extended variant of RIPEMD-160, see Sect. 4.6.4), but a different mechanism is used for interaction between the two trails. To clarify this, we give a high level description of TTMAC in the iterated MAC model:

- The 320-bit chaining variable consists of two 160-bit values (Z_1, Z_2).
- The 160-bit key is expanded to 320 bits and used as initial value for the chaining variable. That is, in the first application of the compression function the chaining variable input is: $(Z_1, Z_2) = (Z_K, Z_K)$.
- Each application of the compression function uses a 512-bit message block M_i to compute a new value for the chaining variable (Z_1, Z_2). The outline of the compression function is as follows:

$$\begin{aligned} Z_L &= \mathcal{L}^*(Z_1, M_i)\,, \\ Z_R &= \mathcal{R}^*(Z_2, M_i)\,, \\ (Z_1, Z_2) &= \mathcal{X}(Z_L, Z_R)\,. \end{aligned}$$

- For the last message block M_{t-1} a modified version of the compression function is used (note that the transformation $\mathcal{X}$ is not applied):

$$\begin{aligned} Z_L &= \mathcal{L}^*(Z_2, M_{t-1})\,, \\ Z_R &= \mathcal{R}^*(Z_1, M_{t-1})\,, \\ (Z_1, Z_2) &= (Z_R, Z_L)\,. \end{aligned}$$

- An output transformation computes the 160-bit MAC result Z from the final 320-bit chaining variable (Z_1, Z_2) as follows:[3]

$$Z = Z_1 - Z_2\,.$$

Note that for a message that consists of a single 512-bit block (after padding), the modified version of the compression function is applied on the chaining variable input $(Z_1, Z_2) = (Z_K, Z_K)$. In this case, the operations $\mathcal{L}^*$ and $\mathcal{R}^*$ can be replaced by $\mathcal{L}$ and $\mathcal{R}$ respectively. The feed-forward (subtraction of the input Z_K) is not needed because the feed-forward in the two trails cancels out in the output transformation.

6.8.2 Design rationale

The idea for the security is simple: now we have a chaining variable (Z_1, Z_2) of 320 bits. This is twice as long as for other MAC constructions (e.g., MDx-MAC and HMAC) based on the RIPEMD-160 hash function (or on the SHA-1 hash function). The most effective generic attack on MAC algorithms is the forgery attack based on internal collisions. The complexity to find such internal

[3]An optional second output transformation can be used to reduce the length of Z.

collisions with a birthday attack depends on the length of the chaining variable, and is completely unrealistic for TTMAC.

Another attack is possible if the MAC result of a message contains all the information (or lacks only a small amount of information) on the internal chaining variable for a longer message, containing the first message as a prefix. In our case we have a chaining variable of 320 bits, so we can use 160 bits of information (the difference $Z_1 - Z_2$) as the MAC output without compromising the chaining variable (an attacker still needs to guess 160 bits which is not easier than guessing the secret key itself). Furthermore, we use the idea of interchanging the left and right trails for the last message block as a free extra defence against extension attacks. It may be noted that other MAC constructions need to apply the compression function with some secret key material at the end of the computation (as output transformation) in order to prevent these attacks. In our case, the secret key is only used as initial value for the two trails.

So now the worry for the cryptographer are the two trails of RIPEMD-160 itself. A single trail has one important weakness: it is a bijective operation, where the attacker can choose the bijection, which is parameterised by the 512-bit value M_i. But as long as two trails are used, parameterised by the same 512-bit value M_i, and only the difference between the trails will come out in the open, there is no danger that an attacker can invert the operation. Moreover, we use feed-forward to counter a straightforward inverting operation on the compression function. This makes the transformation of the 320-bit chaining variable, parameterised by a 512-bit message block, a one-way operation. Note that we do not use feed-forward for single-block messages, because there the feed-forward from the left trail would cancel out the feed-forward from the right trail, in other words we do not need feed-forward there. In order to prevent attacks which target a single trail of the compression function, the transformation $\mathcal{X}$ mixes the outputs from the functions $\mathcal{L}^*$ and $\mathcal{R}^*$. Note that $\mathcal{X}$ can be inverted: one can compute backwards from the output (Z_1, Z_2) to the input (Z_L, Z_R). This ensures that there exist no sets (Z_L, Z_R),(Z'_L, Z'_R) that are mapped by $\mathcal{X}$ to a common value (Z_1, Z_2).

6.8.3 Security of Two-Track-MAC

The security of TTMAC can be proven under the assumption that the compression function is pseudo-random. The same proof can be used as the one that applies for the envelope method [11] (or for MDx-MAC). The compression function of TTMAC is very similar to the one of RIPEMD-160. Although RIPEMD-160 has been designed towards preimage and collision-resistance rather than towards pseudo-randomness, its compression function appears to be strong due to the use of different operations (Boolean functions, bit rotations, additions mod 2^{32}), and because of the large number of steps in the two parallel trails.

Below we analyse the resistance of TTMAC against known attacks. These include generic attacks (Sect. 2.3.2), attacks based on internal collisions (Sect. 6.3.1) and specific attacks which have been used on other hash function based constructions (Sect. 6.5). We show that these attacks are either not applicable or have unrealistic complexities, and we make some comparisons to other MAC constructions.

Guessing of the MAC

The success probability p of an attack where the adversary simply guesses the MAC result depends on the length n of the output ($p = 2^{-n}$). TTMAC supports values of $n = 32, 64, 96, 128$ or 160 bits. Note that the length of the secret key is $k = 160$ bits, so guessing the key and computing the output is not more effective than guessing the output directly. The choice of a suitable output length depends strongly on the application.

Exhaustive key search

Given that TTMAC uses a 160-bit secret key, an adversary would need on average 2^{159} trials before succesfully guessing the correct key value, and $160/n$ known text-MAC pairs for verification of the attack.

Attacks based on internal collisions

The complexity to find internal collisions with a birthday attack depends on the length of the chaining variable. TTMAC has a chaining variable of 320 bits, therefore about 2^{160} known text-MAC pairs are needed to have an internal collision. Furthermore, due to the output transformation about 2^{320-n} chosen texts are needed to distinguish this internal collision from all the external collisions. Note that other MAC schemes, e.g., MDx-MAC and HMAC, based on RIPEMD-160 or on SHA-1, have a chaining variable of 160 bits (and produce outputs of up to 160 bits). Therefore these schemes can be attacked with a complexity of about 2^{80} known text-MAC pairs and just a few chosen texts (if the output length is 160 bits).

Other attacks

For TTMAC the (expanded) secret key is used as initial value for the chaining variable. This implies that there is no danger of a key-independent birthday attack. An extension attack would seem more natural because the key is used only at the start of the computation, but this is addressed by the output transformation and by the interchanging of the left and right trail in the last iteration (see Sect. 6.8.2). Divide-and-conquer key recovery attacks do not need to be considered for TTMAC (key material is used in only one place). This can be compared to HMAC where the possibility of a divide-and-conquer attack implies that the strength of the algorithm depends on the individual keys, and not on their combined length.

6.8.4 Performance considerations

The performance of TTMAC is closely related to the performance of the RIPEMD-160 hash function (see Table 4.11 in Chapter 4) on which it is based. The compression function of TTMAC is very similar to the one of RIPEMD-160, and a detailed analysis shows that TTMAC uses only a few percent more operations on a message than RIPEMD-160 would do to get an unkeyed hash (experiments indicate that 97% of the speed of RIPEMD-160 can be achieved). Moreover, this is already the case for the shortest possible message of 512 bits (after padding). The reason is that for a message input of t blocks both RIPEMD-160 and TTMAC perform t computations of the underlying compression function.

This may be compared to other constructions which can be based on RIPEMD-160. As noted in Sect. 6.5 both HMAC and MDx-MAC need $t+1$ computations of the underlying compression function. The extra invocation is due to the output transformation, and this is relatively costly for short messages (e.g., for a single-block message less than 50% of the speed of unkeyed hashing is achieved). Note that if one uses SHA-1 instead of RIPEMD-160 as the underlying hash function for HMAC or MDx-MAC, this has little impact on the performance of these constructions, see the comparison in Table 4.11.

A different consideration with respect to performance is the amount of work needed when the authentication key is changed. For TTMAC the key is used only to define the initial value of the chaining variable: $(Z_1, Z_2) = (Z_K, Z_K)$. Changing the key does not slowdown the speed of the computation of TTMAC. For the HMAC and MDx-MAC constructions on the other hand, a key-change requires respectively two or six extra computations of the underlying compression function (see Sect. 6.5). Some applications require, for example, that the key is changed for every new MAC authentication. The use of TTMAC in such applications would offer a significant improvement in efficiency, especially if the messages to be authenticated have a short length.

6.8.5 Generalised two-trail MAC construction

Our new MAC construction does not necessarily depend on the hash function RIPEMD-160. More generally, the construction needs two operations $\mathcal{L}$ and $\mathcal{R} : (\mathcal{S}_1 \times \mathcal{S}_2) \to \mathcal{S}_1$. The set $\mathcal{S}_1$ should be large enough to make collisions improbable. The size of the set $\mathcal{S}_2$ should be chosen large if messages are expected to be long. The operations $\mathcal{L}$ and $\mathcal{R}$ are allowed to be invertible if the second argument is fixed, but the operations $\mathcal{L}^*$ and $\mathcal{R}^*$ (including feed-forward from the first input) should be infeasible to invert. The operations $\mathcal{L}$ and $\mathcal{R}$ might be bijective in the first argument, but they should behave unpredictably on changes in the second argument, if the first argument is unknown (but perhaps fixed). It would be even better if the change in the output of the functions $\mathcal{L}$ and $\mathcal{R}$ is unpredictable with known first argument, i.e., the only way to know the effect of a change is to compute the new function value. Based on the experience that a first version of RIPEMD was partially broken (see Sect. 4.6.2), it is recommended that $\mathcal{L}$ and $\mathcal{R}$ should be as different as possible. In the case that $\mathcal{S}_1$ contains all 160-bit strings, one can use the same transformation $(Z_1, Z_2) = \mathcal{X}(Z_L, Z_R)$ as we use for TTMAC based on RIPEMD-160. Of course one also needs to define a padding rule because the message length needs to be a multiple of some fixed quantity. With the transformation $\mathcal{X}$ and a padding rule one can define the procedure for the computation of the MAC result for a message of any length.

6.9 Conclusions

In this chapter we have discussed the design of message authentication codes, algorithms that can be seen as keyed variants of cryptographic hash functions. Most MAC algorithms are based on a block cipher or hash function. Our contribution is the proposal of a new design, called Two-Track-MAC (published in [37]). This algorithm is based on the hash function RIPEMD-160; it offers a high security level against all known strategies of attack, and it is efficient, especially in the case of short messages or frequent key changes. Two-Track-MAC was submitted as a candidate for the European NESSIE project [124], and it has been selected for the NESSIE portfolio of recommended cryptographic primitives.

Chapter 7

Block Ciphers

7.1 Introduction

In Chapter 1 we have explained the distinction between the concepts of confidentiality and authenticity. The main focus of this thesis has been on hash functions and message authentication codes, two types of cryptographic algorithms that are used for protecting data authenticity. Block cipher algorithms on the other hand are designed for the purpose of encrypting data, in order to protect its confidentiality. However block ciphers are also a fundamental cryptographic building block. Indeed we have seen in Chapters 3 and 6 that many practical hash functions and MAC algorithms are derived from an underlying block cipher. In this chapter we provide a definition of block ciphers and a short overview of different designs. We then discuss the security of block ciphers, and develop a variant on the well-known technique of differential cryptanalysis for a specific algorithm called ICE. Finally we give another example of the close relationship between block ciphers and hash functions, by means of the SHACAL ciphers which are derived from hash functions of the SHA family. Our cryptanalysis of the ICE algorithm has been published in [125].

7.2 Definition and Basic Concepts

The following formal definition for a block cipher is similar to the definition given by S. Goldwasser and M. Bellare [57].

Definition 7.1 *A* **block cipher** *is a function $E : \mathcal{D} \rightarrow \mathcal{R}$ where the domain $\mathcal{D} = \{0,1\}^k \times \{0,1\}^b$ and the range $\mathcal{R} = \{0,1\}^b$ for some $k, b \geq 1$. A block cipher function E takes two inputs, a key $K \in \{0,1\}^k$ and a plaintext $P \in \{0,1\}^b$, and* *it returns a ciphertext $C \in \{0,1\}^b$ with $C = E(K,P)$. For each $K \in \{0,1\}^k$ let $E_K : \{0,1\}^b \to \{0,1\}^b$ be the function defined by $E_K(\cdot) = E(K,\cdot)$. Then for any block cipher, and any key K, the function E_K is a permutation on $\{0,1\}^b$.*

The function E_K (encryption with key K) is a permutation (a one-to-one mapping), so it has an inverse function (decryption with the same key K) which we denote by $D_K = E_K^{-1}$. The property $D_K(E_K(P)) = P$ means that by decrypting a ciphertext one recovers the corresponding plaintext.

7.2.1 Security requirements

In applications where a block cipher is used for encryption, a random key K is chosen and kept secret between a pair of users. The users encrypt their data before sending it to each other over an insecure channel. For an adversary who observes a ciphertext C sent over the channel, and who does not know the value of K that is being used, it should be computationally infeasible to obtain the plaintext P corresponding to C. More generally, assume that the adversary has knowledge of a number of plaintext-ciphertext pairs (P_i, C_i). Then it should be infeasible for him to gain any knowledge on the value of K from observing these pairs. Moreover, he should be unable to find the plaintext corresponding to a new ciphertext C', where $C' \neq C_i$ for any i. This is the classical view on block cipher security. More formal notions of security have been proposed by regarding a block cipher as a set of pseudo-random permutations; a user chooses a permutation out of the set by selecting the value of the key. One can now require that for every key the operation of the block cipher be indistinguishable from a random permutation. For more information on this approach to block cipher security we refer to the work of Goldwasser and Bellare [57].

7.2.2 Distinction from stream ciphers

Stream ciphers are a different type of secret-key encryption algorithms. Whereas block ciphers are memoryless and time-invariant transformations, stream ciphers do have memory and use a transformation which varies with time. Most stream ciphers process the plaintext input one bit at a time. Block ciphers process plaintexts in blocks of a fixed length of b bits, typical lengths are $b = 64$ or $b = 128$ bits (see Sect. 7.3). An example of a stream cipher is the encryption mode of the Panama cryptographic module (Chapter 5).

7.2.3 Modes of operation

Many applications of block ciphers require the encryption of plaintext strings of variable length. The most straightforward approach to realise this with a b-bit block cipher, is to divide the plaintext in separate blocks of b bits and encrypt these blocks independently. The ciphertext consists of the concatenation of the encrypted blocks. This procedure is called the *Electronic Code Book* (ECB) mode of operation. Note that a well-defined padding rule must be used to pre-process the plaintext in such a way that the input length is guaranteed to be a multiple of b bits. A disadvantage of the ECB mode is that repetitions of blocks in the plaintext lead to equal ciphertext blocks. This problem is solved in the *Cipher Block Chaining* (CBC) mode, where the value of a ciphertext block depends not only on the corresponding plaintext block but also on the previous ciphertext block. A block cipher can also be used in a mode which emulates the operation of a stream cipher, examples of this are the Cipher Feedback (CFB), Output Feedback (OFB) and Counter (CTR) modes of operation. Note that the five encryption modes mentioned here are included in a proposed standard of NIST [120].

7.3 Design of Block Ciphers

In practice, block ciphers are designed in a manner similar to the one used for cryptographic hash functions: designers learn from mistakes made in the past and try to prevent known methods of attack. There are no block ciphers which are both practical and provably secure, although in some cases bounds can be given for the complexity of specific known attacks. Confidence in a new design is only obtained when it does not get broken after a substantial amount of expert cryptanalysis. Public research in this area was initiated after the adoption of the *Data Encryption Standard* (DES) by the US federal government (NIST) in the 1970's [54]. Another important milestone has been the adoption of the new *Advanced Encryption Standard* (AES) based on the Rijndael algorithm in 2001 [52].

7.3.1 Choice of the parameters

There are two important parameters in the design of a block cipher: the length of the key (k bits) and the length of the blocks (b bits). The key length should be chosen large enough to make *exhaustive key search* infeasible (this is similar to exhaustive key search attacks on MAC algorithms, see Sect. 2.3.2). The DES algorithm has a key length of $k = 56$ bits which no longer guarantees immunity.[1] The AES algorithm on the other hand supports key lengths of $k = 128, 192$, or 256 bits which offers a very high level of security. In choosing the block length one has to take into account the so-called *matching ciphertext attack*. Because

[1]This was demonstrated in practice by the solution of DES challenges proposed by the RSA company, using dedicated hardware [45] or distributed computing over the Internet [133].

of the birthday paradox only about $2^{b/2}$ ciphertext blocks are needed to obtain a matching pair, and this leaks information on the plaintext (we refer to the work of L. Knudsen [73] for more details). Therefore it is recommended that a single key is used to encrypt at most $2^{b/2}$ blocks. Most designs published in the literature have a block length of $b = 64$ bits (e.g., DES) or $b = 128$ bits (e.g., AES).

7.3.2 Block cipher constructions

Most known block ciphers have an internal iterated structure which consists of a concatenation of identical *round transformations*. The idea is that the round transformation is easy to describe and implement, and that by repeating it a sufficient number of times a strong encryption function is obtained. A round transformation can be described by $X_i = \mathsf{round}(K_i, X_{i-1})$, where X_{i-1} is the input to the ith round, X_i the output and K_i a round key. An encryption function of r rounds then consists of initialising with the plaintext block $X_0 = P$, computing the values X_i for $i = 1, 2 \ldots, r$ and finally setting the ciphertext block $C = X_r$. The round keys K_i $(1 \leq i \leq r)$ are derived from the encryption key K by means of a separate *key scheduling algorithm*. Below we review some well-known design principles. For a detailed treatment on block ciphers we refer to the work of V. Rijmen [110].

Components of a cipher

Typical designs use a round transformation consisting of several distinct components, each with their own functionality. The purpose of non-linear substitution boxes (or *S-boxes*) is to achieve *confusion*, that is a complex mixing of bits which are close to each other. This is usually implemented by means of table look-ups. Bit permutations can be used to achieve *diffusion*, that is a re-arranging of bits so that bits which are close to each other in one round, are not close to each other in the next round. The concepts of confusion and diffusion are well-known in cryptology, having been introduced by C. Shannon in [119]. A component is also needed to mix the secret round key values K_i with the intermediate X_{i-1} values.

Feistel ciphers

Feistel ciphers are a special type of block ciphers, where the intermediate values X_{i-1} are divided into two halves (L_{i-1}, R_{i-1}) and where the round transformation is based on a round function F which depends on the round key K_i and operates on one half of the input. The output of F is added to the other half of the input, and finally the two halves are swapped to obtain the output of the round transformation. This means that the round transformation $(L_i, R_i) = \mathsf{round}(K_i, (L_{i-1}, R_{i-1}))$ has the following structure:

$$\begin{aligned} R_i &= L_{i-1} \oplus F(K_i, R_{i-1})\,, \\ L_i &= R_{i-1}\,. \end{aligned}$$

In the last round the swapping of the two halves is omitted. The advantage of the Feistel construction is that the decryption operation is the same as the encryption operation, except that the round keys need to be used in reverse order. This minimises the implementation cost when both encryption and decryption are needed. Note also that the round function F which is the heart of the cipher, does not need to be a permutation (that is, invertible). This may facilitate the design of such a function.

The best known example of a Feistel cipher is the DES algorithm [54]. DES has a block length of $b = 64$ bits and works on 32-bit halves. Its Feistel structure consists of sixteen rounds and the round function F depends on a 48-bit round key. The sixteen round keys are derived from the 56-bit encryption (or decryption) key by means of the key schedule. The ICE algorithm [78] which is analysed in Sect. 7.5 is another Feistel cipher with a block length of $b = 64$ bits. The MISTY1 [82] and Camellia [6] algorithms which have been selected for the NESSIE portfolio are based on a variant of the Feistel structure[2] with $b = 64$ and $b = 128$ bits respectively.

Ciphers based on a uniform transformation structure

Some block ciphers use a uniform transformation structure, in the sense that every round operates on the complete intermediate value X_{i-1} in a similar way. The advantage may be that less rounds are needed to obtain a secure algorithm. This type of block ciphers are also known as substitution-permutation networks, because they separate the role of confusion and diffusion by decomposing the rounds into a layer of S-boxes (substitution) followed by a layer for diffusion (permutation). Note however that the diffusion layer is not necessarily based on a simple bit permutation, more general linear transformations can also be used.

The best known block cipher based on a uniform transformation structure is the AES [52] (derived from the more general Rijndael algorithm [31]). AES uses a diffusion layer derived from a *Maximum-Distance-Separable* (MDS) code, which implies a high diffusion rate. It has a block length of $b = 128$ bits, uses a key of $k = 128, 192$ or 256 bits and operates over $10, 12$ or 14 rounds respectively (the round keys are 128 bits long). Note that for this type of cipher all components

[2]The overall structure is similar, but extra functions are used after some of the rounds.

must be invertible. Moreover, decryption is generally different from encryption (requiring the inverse of the components), except if the design is based on components which are involutions. This is for example the case for the block ciphers Khazad and Anubis [8, 7] (two candidates of NESSIE with a block length of $b = 64$ and $b = 128$ bits respectively).

7.4 Security and Cryptanalytic Techniques

As discussed in Sect. 7.2.1 the usual goal of an adversary attacking a block cipher is to recover the secret key. However, a stronger requirement for the security of a block cipher is that there should exist no attacks which distinguish the algorithm from a random permutation. Because the computation performed by a block cipher involves secret data (the key), different attack scenarios can be distinguished based on the information that is available to the cryptanalyst:

– *Ciphertext-only attack.* The adversary has access to a number of ciphertexts. Possibly he also has some knowledge about the nature of the plaintexts (e.g., characters encoded in ASCII).

– *Known-plaintext attack.* The adversary has access to a number of plaintexts and the corresponding ciphertexts.

– *Chosen-plaintext attack.* The adversary is able to choose a set of plaintexts and subsequently obtains a list of ciphertexts corresponding to these plaintexts.

Even more powerful attacks may be considered, for example using adaptively chosen plaintexts, chosen ciphertexts or a combination of chosen plaintexts and chosen ciphertexts. A designer should be conservative and require that his algorithm resists the strongest possible attacks, even though these may not be realistic in practice.

The security of a block cipher is bounded by the length of the key, because the adversary can always try one-by-one all possible key values and check the correspondence between plaintext and ciphertext of one or more known pairs. In general, an algorithm is considered to be broken if a short-cut attack has been found which is faster than an exhaustive search of the key space. For practical implementations of an attack the data and memory complexity should also be considered. These properties correspond to the required number of plaintext-ciphertext pairs and the amount of storage that is needed for an attack.

Since the introduction of DES much research has been done in the area of block cipher cryptanalysis, and a number of techniques have been developed. The first and best-known of these are differential and linear cryptanalysis. Differential cryptanalysis [18], introduced by E. Biham and A. Shamir, is a chosen-plaintext attack where the propagation of differences through a block cipher is studied. Linear cryptanalysis [81], introduced by M. Matsui, is a known-plaintext attack based on linear approximations of a cipher. Both differential and linear attacks have been developed, which can (theoretically) recover DES keys in time less than exhaustive search.[3] Therefore, Triple-DES is often used as a more secure alternative to DES. Triple-DES, which is specified in [54], is based on three sequential operations of DES. Note that there are several variants of Triple-DES, and that the length of Triple-DES keys can be 112 or 168 bits. Many variations on differential and linear analysis have been proposed later, as well as a number of new techniques. Recent block ciphers, such as AES, are designed with resistance to these known attacks in mind. For a good overview of the most important cryptanalytic techniques we refer to the NESSIE security report [91]. Below we discuss differential cryptanalysis in more detail, a variant of this technique will be developed for our attack on the ICE algorithm in Sect. 7.5.

7.4.1 Differential cryptanalysis

The basic idea in a differential attack is that two chosen plaintexts (P_1, P_2) with a certain difference $\Delta P = P_1 \oplus P_2$ can encipher to two ciphertexts (C_1, C_2) such that the difference $\Delta C = C_1 \oplus C_2$ has a specific value with non-negligible probability. Here the differences are defined with respect to bitwise addition (exclusive-OR). More generally, other definitions are possible, e.g., $\Delta Z = Z_1 - Z_2$ where '$-$' denotes subtraction mod 2^w (and w is the word length in bits). The attacker can choose this, usually the differences will be defined by means of the inverse of the operation that is used for addition of the round keys.

For a differential attack the cryptanalyst needs to find and use a suitable *characteristic*. According to [18] "a t-round characteristic describes a possible evolution of the difference in the various rounds of an iterated cryptosystem and estimates the probability that a random pair with the specified plaintext difference would have the specified differences in the various rounds when it is encrypted under a random key." The probability that a given input difference to a round results in a particular output difference at the end of this round can be computed. For most block ciphers this depends on the properties of the non-linear S-boxes that are used. To simplify the theoretical analysis, one then makes the approximation that the different rounds are independent and this allows the computation of the probability for the t-round characteristic as the product of the probabilities for every round. Note that under certain circumstances several shorter characteristics can be concatenated in order to obtain the overall charac-

[3]However these attacks remain less practical than exhaustive key search due to their data complexity (number of required plaintext-ciphertext pairs).

teristic of t rounds. Sometimes a short characteristic can be concatenated several times to itself, this is called an *iterative characteristic.*

In order to develop a key-recovery attack one does not need a characteristic which extends over the complete block cipher. Usually one or more rounds at the end of the cipher are not included (this means that no assumption is made on the difference propagation in these rounds). The attacker then tries to find encrypted plaintext pairs which follow the characteristic. These are called *right pairs.* Encrypted plaintext pairs which do not follow the characteristic are *wrong pairs.* The process of distinguishing right pairs from wrong pairs is called *filtering*, and usually a small number of right pairs is sufficient for the next step of the attack. Here the attacker guesses the value of some bits of the round key used in the final round, and he checks whether these guesses are consistent with all of the data that is available for the right pairs. We will demonstrate this procedure in the next section where we develop differential attacks on the block cipher ICE. However these attacks are somewhat different from standard differential attacks. In particular we will see that the characteristics which are used, are not valid for all possible keys.

7.5 A Key-Dependent Differential Attack on ICE

The ICE algorithm [78], proposed by M. Kwan in 1997 as an alternative to DES, introduced the concept of a keyed permutation to improve the resistance against differential and linear cryptanalysis. However in this section we demonstrate that a key-dependent differential attack can be applied to ICE. This attack has been published in [125].

7.5.1 Description of the ICE algorithm

ICE, which stands for *Information Concealment Engine*, is a 64-bit block cipher with a Feistel structure similar to DES. The standard ICE algorithm takes a 64-bit key and uses sixteen rounds, each depending on a round key. There is also a fast variant, called Thin-ICE, which uses eight rounds with a 64-bit key, and there are open-ended variants ICE-m which use $16 \times m$ rounds and $(64 \times m)$-bit keys. The structure of Thin-ICE is illustrated in Fig. 7.1. Here both the plaintext (P_L, P_R) and the ciphertext (C_L, C_R) are divided into 32-bit halves. The values K_i denote the round keys. The other variants of ICE have a similar structure but with more rounds. Below we give an outline of the round function F, which is the heart of the Feistel structure, and we describe the components of this function. We also discuss the key scheduling algorithm which is used to derive the round keys from the encryption (or decryption) key K. For a complete description of ICE we refer to [78]. Note that in this section bits are numbered from right to left, starting at bit zero. The rightmost (least significant) bit of an n-bit value V is denoted $V[0]$, while the leftmost bit is denoted $V[n-1]$.

Outline of the round function

The ICE round function F maps 32-bit inputs to 32-bit outputs, using a 60-bit round key. This round key is split into a 20-bit and a 40-bit subkey. First the 32-bit input to the round function is expanded to a 40-bit value. The 20-bit subkey performs a keyed permutation and the 40-bit subkey is exored to the resulting value. Finally four 10 to 8-bit S-boxes and a bit permutation are used to obtain the 32-bit result of the round function.

The expansion function

The 32-bit input I to the round function F is expanded to four 10-bit values $E0, E1, E2, E3$ (eight of the bits at the input are duplicated).

The keyed permutation and key addition

The 20-bit subkey, which we denote KP, performs a keyed permutation on the expanded 40-bit text, swapping bits between $E0$ and $E2$, and between $E1$ and $E3$. If the subkey bit $KP[10+j]$ ($j < 10$) is set (equal to one), bits $E0[j]$ and $E2[j]$ are swapped. If the subkey bit $KP[j]$ ($j < 10$) is set, bits $E1[j]$ and $E3[j]$ are swapped. The 40-bit result from this keyed permutation is then exored with the 40-bit subkey (denoted KA).

The S-boxes

The round function of ICE uses four S-boxes ($S0, S1, S2, S3$) with 10-bit inputs and 8-bit outputs to map the 40-bit value obtained after the key addition to a 32-bit value. These S-boxes are based on Galois Field exponentiation. Each S-box takes a 10-bit input X, from which $X[9]$ and $X[0]$ are concatenated to form the row selector R. Bits $X[8]...X[1]$ are concatenated to form the column selector C. For each row there is an exor offset value O_R, and a Galois Field prime (irreducible polynomial) P_R. Note that the values of O_R and P_R are different for the different S-boxes. The 8-bit output of an S-box, corresponding to its input X, is computed as $(C \oplus O_R)^7 \bmod P_R$, under Galois Field arithmetic. Usually this will be implemented by means of a look-up table.

The permutation function

The final operation in the round function F is a bit permutation where the four 8-bit S-box outputs are combined into the 32-bit output of F.

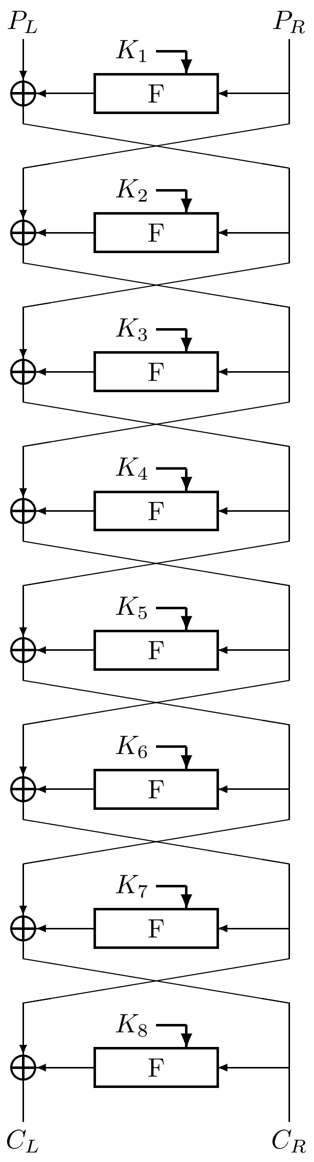

Figure 7.1: The Feistel structure of Thin-ICE (eight rounds).

The key scheduling algorithm

The ICE key schedule maps a 64-bit user key K into sixteen 60-bit round keys K_i (divided into a 20-bit subkey KP_i and a 40-bit subkey KA_i). Important for our attack is that each round key bit depends on only one bit of the user key. Hence if an attacker is able to determine the value of one round key, he also knows 60 bits of the user key. Note that the Thin-ICE and ICE-m key schedules are similar, except that the number of derived round keys is different (and for ICE-m the length of the user key).

7.5.2 Strategy for a differential attack on ICE

In [78] the designer of ICE considered the strength of the algorithm against differential attacks. The analysis considers only symmetric input differences, having equal left and right 16-bit halves of the 32-bit input to the F function. This was claimed to be the best strategy since they are the only differences that are not affected by the keyed permutation. As a consequence the attacker has to target at least two S-boxes at a time and the probabilities for the difference propagation are too low to be used in a realistic attack.

The approach used in our attack is to use (asymmetric) differences with a low Hamming weight (as low as possible). Whether these differences are affected by the keyed permutation depends on the values of only a few key bits. The advantage of this approach is that the attacker has to target only one S-box in the round function. In this way he finds characteristics with a probability that is high enough to (theoretically) recover ICE keys for the algorithm reduced to fifteen rounds, in time less than the expected cost for exhaustive search. Applied to Thin-ICE (the fast eight-round variant of the algorithm) the complexity is low enough to make the attack practical. The complication of our approach is that the attacks are key-dependent.

7.5.3 Differential characteristics for ICE

As explained above we will focus on low Hamming weighted differences that address only one S-box in the round function. It turns out that it is not possible to build a two-round iterative characteristic like the one used for the cryptanalysis of DES in [18], with a difference that addresses only one S-box (using only the middle six bits out of the ten input bits to that S-box so that it is not affected by the expansion in the round function). We can however build three-round iterative characteristics of the form specified in Fig. 7.2.

For this three-round characteristic we require the transitions ($\alpha \rightarrow \beta$) in the second round function, and ($\beta \rightarrow \alpha$) in the third round function. These transitions occur with probabilities p_1 and p_2 respectively. Because we restrict

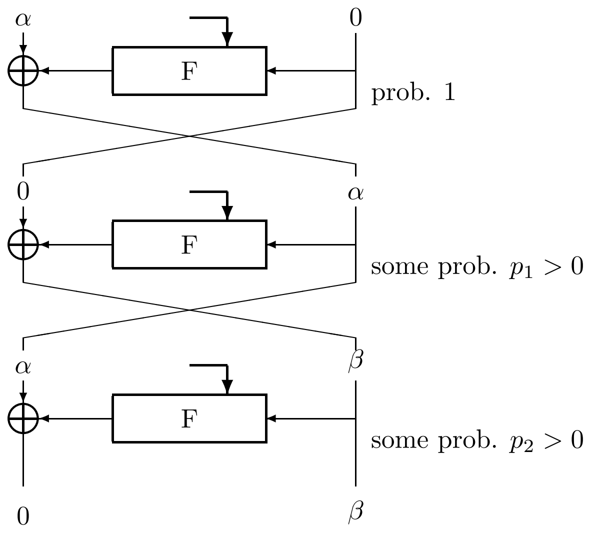

Figure 7.2: Three-round iterative characteristic.

the differences α and β to one S-box, they can have a Hamming weight of no more than four each, since the 8-bit output of an S-box delivers, after the permutation and the key dependent permutation in the next round, up to four bits to an S-box in the next application of the round function.

The characteristic will be valid if α and β are not affected by the keyed permutation in the corresponding rounds. This happens if the permutation key bits used in the bit positions that are set (equal to one) in α and β, are equal to zero (so the difference is not permuted from the left 20-bit half of the expanded text to the right or vice versa).

There are also characteristics which are valid only if certain permutation key bits are equal to one (corresponding to the bits set in α or β or both). In general we call these *conditional characteristics.* They have a certain probability with respect to a subset of the key space. Note that the concept of conditional characteristics was first used by I. Ben-Aroya and E. Biham in their cryptanalysis of the Lucifer algorithm [16]. Their usage is advisable when they improve the probability over the best probability of a non-conditional characteristic by a factor that is higher than the inverse of the *key fraction* (the ratio between the size of the subset and the size of the key space), especially if several such characteristics can efficiently share the same structure of chosen plaintexts.

If we consider only differences with Hamming weight one (this maximises the key fraction) there is a total of 105 conditional characteristics with

$$2^{-13} \geq p_1 \cdot p_2 \geq 2^{-18}.$$

Table 7.1 lists some of the differences of Hamming weight one, which can be used to construct a three-round iterative characteristic with probability $p_1 \cdot p_2 \geq 2^{-15}$, together with the corresponding probabilities. The differences α and β are denoted by the bit position in the 32-bit value that is set at one. We also list the required value for the permutation key bits corresponding to α and β in the second and third round of the characteristic. By interchanging the values of α and β we get twice the number of characteristics (except for the fourth entry in the table which has $\alpha = \beta$).

Table 7.1: Characteristics with Hamming weight one and $p_1 \cdot p_2 \geq 2^{-15}$.

α	β	p_1 (-$\log_2$)	p_2 (-$\log_2$)	round 2	round 3
18	28	6	7	0	0
26	29	6	7.4	1	1
31	26	8	6	1	0
7	7	7	7	0	0
3	10	7.4	7.4	0	0
27	3	7.4	7.4	1	1
4	22	7.4	7.4	1	0
18	30	6	9	1	0
15	23	7	8	0	0
22	7	7	8	1	1

7.5.4 Differential attacks on reduced versions

An attack on six rounds

We use the best three-round characteristic, with $\alpha = 28$ and $\beta = 18$, followed by a trivial round with probability one. The probability for this four-round characteristic is $p_1 \cdot p_2 = 2^{-7} \cdot 2^{-6} = 2^{-13}$. The characteristic is valid if the bits set in the differences α and β are not permuted in the respective rounds (rounds 2 and 3). This can be translated to the following conditions for the permutation subkeys KP_2 and KP_3 (used in rounds 2 and 3 respectively):

$$KP_2[14] = 0 \text{ and } KP_3[2] = 0\,.$$

Examination of the key scheduling algorithm shows the corresponding condition for the 64-bit user key K:

$$K[20] = 1 \text{ and } K[12] = 0\,.$$

Fig. 7.3 shows the six-round algorithm and the four-round characteristic. The expected input difference to the round function in round 5 is β, the expected output difference equals the difference in the right half of the ciphertext (ΔC_R). This allows us to check if an arbitrary encrypted pair (with difference α in the left half of the plaintext, and no difference in the right half) is a right pair for the characteristic. The difference $\beta = 18$ delivers an input difference to S-box $S1$ or $S3$, depending on the value of the corresponding permutation subkey bit. So the output differences from S-boxes $S0$ and $S2$ have to be zero, as well as the output difference from either $S1$ or $S3$. This is equivalent to checking the values of $24 - 1 = 23$ bits. So a wrong pair has a probability 2^{-23} of surviving this filtering process. The probability of generating a right pair is much higher (2^{-13}), so when a pair survives the filtering, with a high probability it is a right pair.

For such a right pair we know the inputs and the difference at the output ($\Delta C_L \oplus \beta$) of the last round, and for all possible values of the round key K_6 we can check whether they correspond. This is done separately for each S-box. Repeat this for about four right pairs (we need to generate about $4 \cdot 2^{13} = 2^{15}$ pairs of plaintexts); the correct round key K_6 will be suggested each time and can be distinguished from other suggested round key values.

The signal-to-noise-ratio (the ratio of the number of times the correct key is suggested and the number of times an arbitrary key is suggested) for this attack can be calculated with the method described in [18]. It depends on the number of plaintext pairs m, the probability of the characteristic p, the number k of simultaneous key bits that we count on, the average count a per analysed pair, and the fraction b of the analysed pairs among all the pairs.

$$S/N = \frac{m \cdot p}{m \cdot a \cdot b / 2^k}\,.$$

In this case we have $m = 2^{15}$ and $p = 2^{-13}$. When concentrating on one S-box we are counting on $k = 20$ key bits (ten bits used for the keyed permutation with KP_6 and ten bits used for the exor with KA_6). The average count a equals 2^{12}, since we count on 2^{20} keys and check an 8-bit value (difference at the output of the S-box). The fraction b (filtering) equals 2^{-23}. Hence the signal-to-noise-ratio is:

$$S/N = \frac{2^{15} \cdot 2^{-13}}{2^{15} \cdot 2^{12} \cdot 2^{-23} / 2^{20}} = \frac{2^2}{2^{-16}} = 2^{18},$$

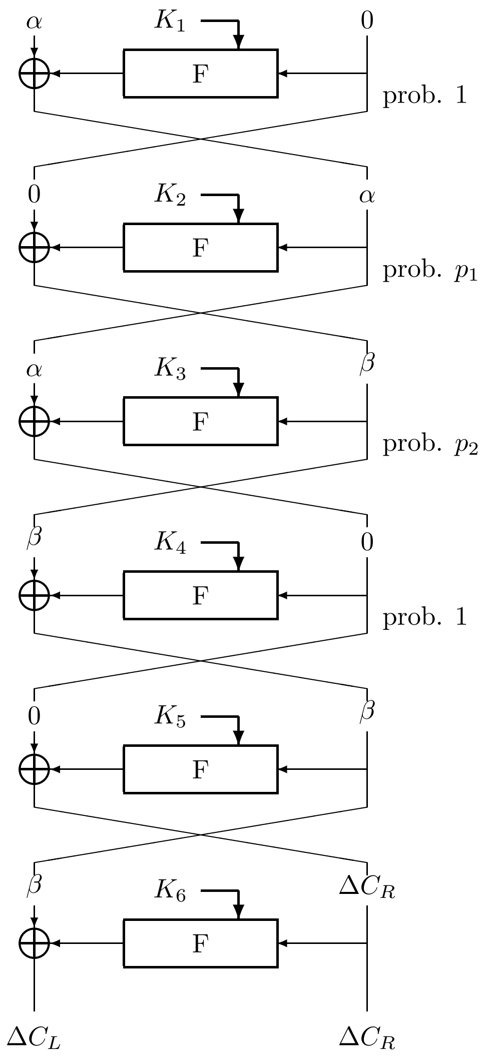

Figure 7.3: The characteristic for an attack on six rounds.

and similarly for the other three S-boxes. However $S0$ and $S2$, as well as $S1$ and $S3$, use the same permutation subkey bits, which we have to determine only once. For the second S-box which uses these permutation subkey bits we can count on just the ten exor subkey bits. In this way we determine all sixty bits of the round key K_6, which correspond to sixty bits of the user key K. The remaining four bits of the user key can easily be found by exhaustive search.

An attack on eight rounds (Thin-ICE)

We can extend the previous attack in a straightforward manner, using a six-round characteristic with a probability $p_1 \cdot p_2 \cdot p_2 \cdot p_1 = 2^{-7} \cdot 2^{-6} \cdot 2^{-6} \cdot 2^{-7} = 2^{-26}$. The attack can be improved however by inserting a round before the first round of the characteristic without reducing the probability, like in the attack on DES [18]. The assumed evolution of differences (during the encryption of a right pair) is shown in Fig. 7.4. In the first round the difference α at the input of the round function is an input difference to S-box $S0$ or $S2$, depending on the value of the corresponding permutation subkey bit. We guess this bit and repeat the attack if we have guessed wrong. We compensate the difference at the output of the round function of round 1 by using a structure of 2^9 plaintexts:

$$P_i = P \oplus (v_i, 0), \bar{P}_i = P \oplus (v_i, 0) \oplus (0, \alpha) \qquad \text{for } 0 \leq i < 2^8,$$

with v_i denoting all the possibilities for the eight bits that are exored with the output bits from $S0$ or $S2$; (l, r) denotes the left and right 32-bit halves of a 64-bit text.

The probability for the characteristic is $p_1 \cdot p_2 \cdot p_2 = 2^{-7} \cdot 2^{-6} \cdot 2^{-6} = 2^{-19}$. The characteristic is valid under the following conditions for the permutation subkeys used in rounds 3, 4 and 6:

$$KP_3[14] = 0,\ KP_4[2] = 0 \text{ and } KP_6[2] = 0\,.$$

The corresponding condition for the 64-bit user key is:

$$K[3] = 0,\ K[59] = 1 \text{ and } K[25] = 1\,.$$

In the defined structure there are 2^{16} pairs, of which 2^8 satisfy the first round. These can be isolated in 2^8 time as follows. Since the expected output differences from S-boxes $S1$ and $S3$ in round 7 are zero, we sort the texts according to the values of the corresponding bits in the right half of the ciphertext and find the matching values. We filter these further by S-box $S0$ or $S2$ (like in the six-round attack), and can expect $2^8 \cdot 2^{-19} = 2^{-11}$ right pairs in a structure. By using 2^{13} structures four right pairs are expected. In total however there are $2^{16} \cdot 2^{13} = 2^{29}$ pairs. After filtering for 23 bits we expect there will remain $2^6 = 64$ wrong

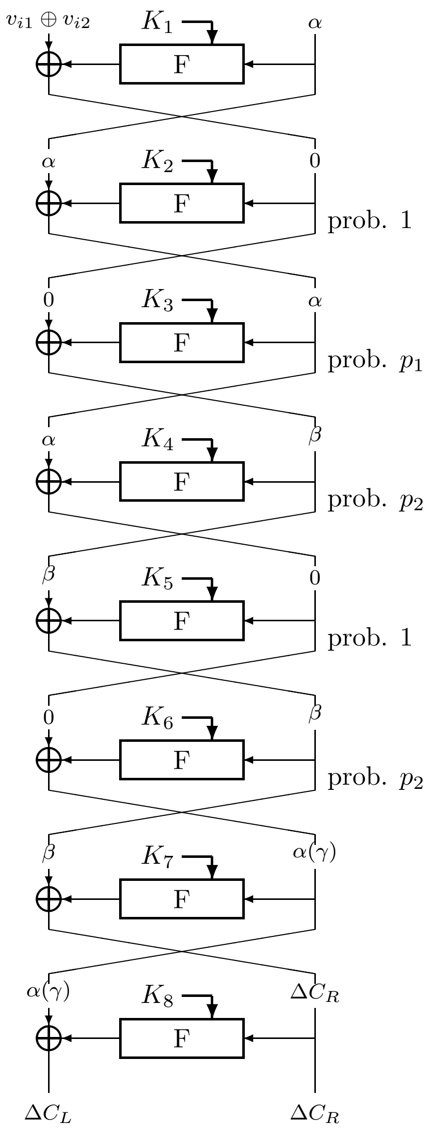

Figure 7.4: The characteristic for an attack on eight rounds.

pairs. For this mixture of right and wrong pairs we try all possible values for K_8 concentrating on one S-box at a time.

In the calculation of the signal-to-noise-ratio for this attack there is an extra factor 2^{-8}, imposed by the first round structure ($m = 2^{13} \cdot 2^{16}$, but only $2^{13} \cdot 2^{8}$ pairs satisfy the first round):

$$S/N = \frac{2^{13} \cdot 2^{8} \cdot 2^{-19}}{2^{13} \cdot 2^{16} \cdot 2^{12} \cdot 2^{-23}/2^{20}} = \frac{2^{2}}{2^{-2}} = 2^{4}.$$

We can easily extend this attack to make it valid for twice as many keys. Just guess the value of $KP_6[2]$ (bit 2 of the permutation subkey in round 6). If it equals 1 instead of 0, the round function in that round delivers an output exor different from $\alpha = 28$. With probability 2^{-6} this output exor will be $\gamma = 30$ and we can perform the attack in a similar way. The condition for the 64-bit user key therefore reduces to:

$$K[3] = 0 \text{ and } K[59] = 1\,.$$

Alternatively we can do an eight-round attack using the characteristic with $\alpha = 31$ and $\beta = 26$. According to Table 7.1 the probability of this characteristic is $p_1 \cdot p_2 \cdot p_2 = 2^{-8} \cdot 2^{-6} \cdot 2^{-6} = 2^{-20}$. The conditions for the permutation subkey bits translate to the following condition for the user key:

$$K[48] = 1,\ K[18] = 1 \text{ and } K[48] = 1\,.$$

So the permutation subkey bit in round 6 doesn't impose an extra condition on the user key, and we don't have to guess this bit when using the characteristic with $\alpha = 31$ and $\beta = 26$.

Practical aspects of the analysis

The attacks on the six-round version and the eight-round version (Thin-ICE) have been implemented and, on the average, work as predicted. However, using low Hamming weighted differences causes some complications. The input difference to the last round is caused by the output difference from the previous round. That output difference is caused by just one S-box and has a Hamming weight of no more than eight, with an average of four.

Each S-box in the last round receives two bits from these eight bits. Because the keyed permutation swaps bits between the 'partner' S-boxes $S0 - S2$ and $S1 - S3$, an S-box will finally receive between zero and four from these bits at its input, depending on the value of the permutation subkey. Only these bits can cause an input difference. If S-box $S0$ or $S1$ gets k bits, then respectively $S2$ or $S3$ will get $4-k$ bits. If a particular S-box gets k bits with a possible difference, the probability to get input difference zero is approximately 2^{-k}. In Table 7.2

Table 7.2: Probabilities to get a zero input difference to an S-box.

probability	fraction of subkeys
1	$1/2^4$
2^{-1}	$4/2^4$
2^{-2}	$6/2^4$
2^{-3}	$4/2^4$
2^{-4}	$1/2^4$

we list the possible values for that probability, and the fraction of subkeys for which it holds.

If the input difference to an S-box is zero, all of the guesses for the permutation subkey that cause a zero difference will be counted, as will all possibilities for the exor subkey. Therefore the attack is less efficient and we have to look for some more right pairs for the characteristic (in practice between four and eight), hence use more plaintexts.

For a fraction 2^{-4} of the keys the input difference to the S-box will always be zero, so we can determine only some of the permutation subkey bits and none of the exor subkey bits. But then the partner S-box has a probability for zero input difference of only 2^{-4}. We determine the permutation subkey bits via this S-box, and the ten exor subkey bits that we cannot determine can be looked for exhaustively after the differential attack (together with the four bits of the user key that are not used in the 60-bit round key of the last round).

It is possible to exploit the occasions of zero input differences to improve our attack. If the input difference to an S-box in the last round is zero, the output difference is zero as well. In that case we know the corresponding difference at the input to the round function in the second to last round and we can check if its value corresponds to the value that is required for the characteristic. In this way we can do some extra filtering, which is important for the eight-round attack where we expect to get 64 wrong pairs. It will increase the signal-to-noise-ratio and reduce the number of required plaintexts.

7.5.5 Extending the attack

The three-round iterative characteristic can be extended in a straightforward manner to attack the ICE algorithm with an arbitrary number of rounds. But if the number of rounds exceeds 9, the signal-to-noise-ratio will drop below one, making the attack impossible (the last remark of previous section allows only a slight improvement by extra filtering). There are however several ways to improve the signal-to-noise-ratio.

Counting on more key bits

When a pair survives the filtering (and is assumed to be a right pair, following the characteristic), we know the inputs and the difference at the output of the last round and check whether they correspond. In the basic attack we concentrate on one S-box and count on twenty subkey bits (ten used for the permutation and ten for exoring). Instead we can consider two partner S-boxes ($S0$ and $S2$, or $S1$ and $S3$) at the same time. They share ten permutation subkey bits and both use ten exor subkey bits. This allows us to count on thirty key bits, and results in an improvement of the signal-to-noise-ratio by a factor of 2^8 because we check the values of eight more bits at the output of the second S-box (in the calculation of S/N we have $2^k = 2^{30}$ and $a = 2^{30}/2^{16} = 2^{14}$). In theory further improvements (by a factor of 2^{16} or 2^{24}) are possible by considering respectively three or four S-boxes (fifty or sixty key bits).

Checking differences in the first round

When a pair is assumed to follow the characteristic we can also check subkey bits in the first round of the algorithm. In this first round we use a special structure (cf. the attack on eight rounds) and guess the value of the permutation subkey bit corresponding with the difference of Hamming weight one. Hence we can count on the ten exor subkey bits of the S-box where the difference of Hamming weight one is located. Moreover, due to the key schedule some of these subkey bits in the first round represent the same user key bits as some of the subkey bits in the last round of the algorithm. This allows us to improve the signal-to-noise-ratio by a factor of 2^8 by counting on just a few more key bits. Note also that some of the key bits that we count on are already known, because of the condition on the user key for the characteristic to be valid.

Filtering in the last round

The most important improvement can be made by adapting the characteristic. In the previous attacks we used the characteristics with the highest probabilities. The resulting attacks are called 2R-attacks (cf. Biham and Shamir [18]), because they don't make assumptions for the last two rounds of the algorithm. Instead we can perform 1R-attacks, using a characteristic up to the last to one round. An example of the last rounds for such a characteristic is shown in Fig. 7.5.

Although the probability of such a characteristic is generally lower than for a 2R-attack, it is useful because it allows much more filtering and an overall improvement of the signal-to-noise-ratio. In the last round we can check if the

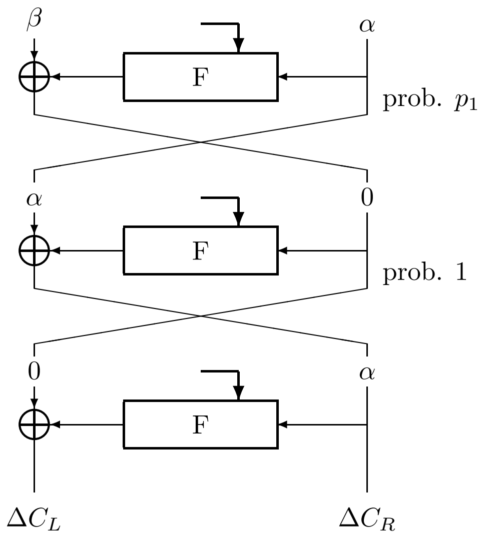

Figure 7.5: Characteristic (last rounds) for a 1R-attack.

difference at the output of the round function (ΔC_L in Fig. 7.5) is possible, like we did in the last to one round in the previous attacks. This corresponds to checking the values of 23 bits. But we can also check the difference in the right half of the ciphertext ($\Delta C_R = \alpha$ in Fig. 7.5), thus filter for 32 more bits. This results in an improvement of the signal-to-noise-ratio by a factor of $2^{32} \cdot p_c$, where p_c represents the factor by which the probability of the characteristic is reduced when we perform a 1R-attack instead of a 2R-attack.

Results for an arbitrary number of rounds

Table 7.3 lists for each number of rounds: the probability of the characteristic, the required number of chosen plaintexts (assuming four right pairs for the characteristic are sufficient, and that we need both guesses for the permutation subkey bit in the first round structure), the number of subkey bits counted on (excluding key bits in the first round because of the overlap), the signal-to-noise-ratio and the fraction of keys that can be found with the attack. Note that the number of plaintexts determines the data and time complexity, the number of counters determines the memory that is needed for an attack.

When the number of rounds exceeds nine we list two different attacks: a 2R-attack (where the signal-to-noise-ratio is improved by counting on more key bits and checking the difference in the first round), and a 1R-attack (which has

Table 7.3: Differential analysis for an arbitrary number of rounds. († $\log_2$)

rounds	probability †	plaintexts †	counters †	S/N †	key frac. †
4	0	4	20	23	all
5	-6	10	20	17	all
6	-13	16	20	18	-2
7	-13	17	20	10	-2
8	-19	23	20	4	-2
9	-26	29	20	5	-4
10	-26	30	20	5	-4
10	-32	36	20	23	-5
11	-32	36	20	-1	-4
11	-39	42	20	24	-6
12	-39	43	20	16	-6
13	-39	43	30	0	-6
13	-45	49	20	10	-7
14	-45	49	50	2	-6
14	-52	55	20	11	-8
15	-52	56	20	3	-8
16	-52	56	60	3	-8
16	-58	62	30	5	-9

a lower probability and requires more plaintexts). When the number of rounds is a multiple of three, we have listed only the 1R-attack because it has the same probability as the 2R-attack ($p_c = 1$). In the other cases we have $p_c = p_1 = 2^{-7}$ or $p_c = p_2 = 2^{-6}$. Note that the key fraction is lower for a 1R-attack, because the characteristic imposes more conditions on the user key (except when the number of rounds is a multiple of three).

The table shows that the differential analysis works for up to fifteen rounds of ICE: for this attack eight key bits are fixed and an exhaustive search would have 2^{56} possibilities, our attack requires at most 2^{56} plaintexts. For the complete sixteen-round algorithm the number of plaintexts needed is still smaller than the total number of available plaintexts, but an exhaustive search in the covered fraction of the key space would be faster.

7.5.6 Key dependency

We described the previous attacks using the best conditional characteristic. In Table 7.3 we have listed for how many keys this works. For Thin-ICE (eight

Table 7.4: Thin-ICE (eight rounds): the number of characteristics and plaintexts versus the fraction of keys that can be found.

characteristics	plaintexts ($\log_2$)	key fraction
1	23	25%
3	24	63%
5	25	81%
6	26	88%
8	27	95%

rounds) the attack works for a fraction 2^{-2} of the keys. For other keys however we can use a different characteristic with a lower probability. We can use several characteristics with the same set of plaintexts, if we use a special structure for these plaintexts. For two characteristics this is a quartet structure (like the one used for the analysis of DES in [18]), for three an octet structure and so on. The number of plaintexts we need depends on the characteristic with the lowest probability. If we want to be able to determine as many possible keys with as few possible plaintexts we use the characteristics with the highest probabilities. Table 7.4 shows the evolution of these numbers for the Thin-ICE algorithm.

7.5.7 Conclusions

The ICE algorithm was proposed as a possible alternative to DES, the most important feature of the design being the use of keyed permutations. In this section we have shown that differential attacks can be applied to ICE and the main conclusion is that the keyed permutations do not prevent differential cryptanalysis. Although the analysis is more complex and becomes key dependent, in our opinion the intention of the design was not reached.

This is demonstrated by the fact that there is a practical attack on the fast variant Thin-ICE. In its basic form it finds the secret key in 25% of the cases using 2^{23} chosen plaintexts, and in 95% of the cases using 2^{27} plaintexts. The optimal characteristic for our attack on Thin-ICE has a probability of 2^{-19} which is much higher than the probability of the optimal characteristic based on a symmetric input difference, which was shown to be 2^{-56} by the designer of ICE [78].

7.6 Block Ciphers Based on Hash Functions

In Chapter 3 we have seen that block ciphers can be used to construct a cryptographic hash function. The inverse is also true: some block ciphers have been proposed where the design was based on an existing hash function. In this section we illustrate this by means of the SHACAL-1 and SHACAL-2 ciphers, which are based on hash functions of the SHA family.

7.6.1 The SHACAL ciphers

In [60] H. Handschuh and D. Naccache proposed to use the compression function of the cryptographic hash function SHA-1 in encryption mode. As noted in Sect. 4.7.1 the eighty elementary step operations of SHA-1 are reversible. Therefore, if one uses a secret key instead of the message and a plaintext instead of the input chaining variable, and if one omits the feed-forward operation at the end of the compression function, an invertible function is obtained. This function can be used as a block cipher that encrypts 160-bit plaintexts into 160-bit ciphertexts by means of eighty step operations, dependent on a 512-bit key. This key is first expanded to eighty 32-bit words (2560 bits) by means of the key scheduling algorithm, which is equivalent to the procedure for message expansion in SHA-1. Alternatively, one can also regard SHA-1 as being based on an underlying block cipher, that is used in Davies-Meyer mode.[4]

After the announcement of a number of new hash functions for the NIST hash function standard, Handschuh and Naccache also proposed a new block cipher that uses the compression function of the hash function SHA-256 in encryption mode [60]. This cipher has 256-bit blocks, 512-bit keys and uses sixty-four step operations. The two proposed block ciphers, based on SHA-1 and SHA-256, are called SHACAL-1 and SHACAL-2 respectively. They were submitted as candidates for the NESSIE project.

Known security results

The block cipher SHACAL-1 has been evaluated for its resistance against advanced types of differential attacks. More particularly, in [70] *boomerang attacks* were applied, and in [17] *rectangle attacks* were applied to the algorithm. The best of these attacks works (theoretically) with a data complexity of 2^{152} chosen plaintexts and a time complexity of 2^{509} operations, for a variant of SHACAL-1 reduced to 49 steps. From the analysis it can be seen that the probability of differential characteristics decreases rapidly when the number of steps increases, due to the fact that the Hamming weight of the difference words grows larger.

[4]See Sect. 3.4.1, a small difference is that the Davies-Meyer mode of SHA-1 uses modular addition instead of exclusive-OR for the feed-forward operation.

For example, the best 28-step characteristic which was found has a probability of 2^{-107}, and the best 30-step characteristic has a probability of 2^{-138}. Overall the security margin of SHACAL-1 against such attacks appears to be very large. It may be noted that if a differential attack on the complete SHACAL-1 cipher would be possible, then this might also affect the security of the SHA-1 hash function. In particular, it might lead to pseudo-collisions for SHA-1. For SHACAL-2 an *impossible differential attack* was applied in [61]. This attack works for a variant of SHACAL-2 reduced to 30 steps, in less time than needed for an exhaustive search for a 512-bit key.

The slide attack on SHA-1, which we discussed in Sect. 4.7.2, can also be applied to the block cipher SHACAL-1. In particular it is shown in [118] that an attacker who has access to two SHACAL-1 encryption oracles whose keys are *slid* (in the same way that the procedure for message expansion can be slid for SHA-1), is able to distinguish the cipher from a random permutation with a complexity of about 2^{96} chosen plaintexts. This does not lead to a practical attack but it presents an undesirable property. SHACAL-2 is not vulnerable to such a slide attack, because each of the step operations in this algorithm uses a unique additive constant. Note that SHACAL-2 has been selected as one of the block ciphers in the NESSIE portfolio.

7.7 Conclusions

This chapter has given a discussion on block ciphers. These algorithms are not only used for encryption purposes but they are also useful for the construction of other cryptographic primitives, including hash functions and message authentication codes. We have presented some well-known design principles and discussed the security of block ciphers. The technique of differential cryptanalysis has been explained in some detail, and a key-dependent variant of this technique was developed for an attack on the block cipher ICE. This attack has been published in [125]. Finally we also mentioned the design of block ciphers based on hash functions, with the example of two ciphers based on the hash functions SHA-1 and SHA-256.

Chapter 8

Conclusions and Open Problems

8.1 Cryptanalysis of Hash Functions

One of the main topics of this thesis has been the cryptanalysis of hash functions. These algorithms are versatile building blocks used in many cryptographic applications, especially towards the protection of the authenticity of information. The analysis of cryptographic hash functions is a young research area. The MD4 hash function [114] which is a precursor of the most popular hash functions in use today, dates from 1990; in 1995 an innovating approach for analysing this type of hash functions was introduced by H. Dobbertin [41]. This may be compared to the situation for block ciphers, where the academic research started in the 1970's (after the adoption of the DES standard [54] by the US government). The methods of differential [18] and linear cryptanalysis [81] were first presented in the early 1990's, and are now a well-known tool for evaluating the security of block ciphers.

In Chapter 4 we have given an extensive overview of the attacks published in the literature for hash functions based on the design ideas of MD4, and we hope that this will contribute to a better understanding of this field. It may be noted that the cryptanalytic methods of Dobbertin [41, 42, 39] seem to be more generally applicable than the earliest attacks on MD4 [35] and MD5 [36], and new designs of hash functions should take this into account. An important contribution of our research is the attack which we developed for the HAVAL algorithm. HAVAL [131] was designed in 1992 when the techniques of Dobbertin were still unknown. The main focus of the design was on the complex non-linear Boolean functions which are used in the compression function. In [40] Dobbertin noted:

> "Various promising new characteristics are involved in the design of HAVAL. These probably improve the cryptographic strength, but could also be pitfalls introducing unexpected weaknesses. It should be investigated whether there is a suitable modification of the MD4 attack, which could be applied to the 3-round version of HAVAL. As long as such analysis has not been worked out, there is no sufficient base to assess the strength of HAVAL."

Our work [123] has shown that the version of HAVAL with three rounds in the compression function can be broken, with an attack that follows a strategy similar to the one for Dobbertin's attack on MD4 [42]. It is the first published attack on a complete version of HAVAL. Our result makes it clear that for a secure design it is not sufficient to use strong building blocks (e.g., the non-linear Boolean functions of HAVAL), but the effect of these building blocks on the overall security must be determined.

Although the experience with the cryptanalysis of MDx-class hash functions has led to the design of a number of algorithms (particularly those of the RIPEMD [100] and SHA [51] families) which appear to have a large security margin against all known strategies of attack, there are limitations to this approach for designing hash functions. In particular, the more recent (and more secure) algorithms are considerably less efficient than MD4 and MD5 [115] (see Table 4.11). This is contrary to the evolution which can be noticed in the design of block ciphers: the recent US standard AES [52], and the NESSIE block ciphers MISTY1 [82] and Camellia [6], are considerably faster than DES, and especially Triple-DES, as shown in [92]. According to [92] the MD4 hash function is more than ten times faster than DES, but the SHA-256 hash function [51] is *slower* than AES.[1] Therefore, there certainly is an interest in fresh ideas for the design of new hash functions. An example of a new type of design is the PANAMA cryptographic module [30], which can be used for both hashing and stream encryption. The inherent parallelism of PANAMA allows very fast software implementations on VLIW processors (at least for the hashing of long message streams). However, in Chapter 5 and in [112] we have demonstrated a theoretical collision attack on the hash mode of PANAMA, which is much faster than a generic birthday attack.

[1]SHA-256 has been designed to offer a security level similar to AES (about 2^{128} operations for the best attack on both algorithms). Note however that SHA-256 is still faster than AES used in a hashing mode [92].

8.2 Design of a New MAC Selected by NESSIE

In Chapter 6 we have presented a new design for a message authentication code, or *keyed* hash function. Our algorithm, called Two-Track-MAC or TTMAC [37], is closely based on the design of the unkeyed hash function RIPEMD-160 [100]. The compression function of TTMAC uses the two trails of RIPEMD-160, and we have demonstrated that thanks to this two trail construction our algorithm is faster than other MACs based on RIPEMD-160 (or SHA-1), especially when processing short messages. Furthermore, TTMAC has extremely good key agility, which makes it well suited to applications where the secret key needs to be changed frequently (e.g., after every message). We have also shown that TTMAC has a high security margin against all known attacks on MAC algorithms. TTMAC has been submitted to the European NESSIE project [124], and in February 2003 the NESSIE consortium announced [94] that TTMAC is part of its portfolio of recommended cryptographic primitives. The selection of TTMAC is motivated in [93] as follows:

> "TTMAC (also known as Two-Track-MAC) has the highest security level of the MAC primitives considered by NESSIE. The design of TTMAC is based on the hash function RIPEMD-160 (with small modifications). The security can be proven on the assumption that the underlying compression function is pseudo-random. TTMAC has specific performance advantages: it is especially efficient in the case of short messages, and has optimal key-agility."

Three other MAC algorithms are included in the NESSIE portfolio: UMAC [76], EMAC [95] and HMAC [15]. The NESSIE consortium [93] remarks:

> "No security weaknesses were found for any of these primitives. NESSIE makes a broad recommendation in this area because every primitive has its own specific advantages."

8.3 Study of Block Ciphers

In Chapter 7 we have studied block ciphers. The main use of these algorithms is for encrypting data (the protection of confidentiality), but they can also be used for the construction of hash functions and message authentication codes. On the other hand there also exist block cipher designs which are based on a hash function (see for example the SHACAL [60] block ciphers). We have presented [125] a key-dependent variant on the technique of differential cryptanalysis, applied for an attack on the block cipher ICE [78].

8.4 Further Research

During the research for this thesis a number of problems were encountered, which are still unsolved. Additional cryptanalysis of MDx-class hash functions is possible, especially for the most recent algorithms of the SHA family which have been adopted as standards by NIST without disclosure of their design strategy or any supporting evaluation. Due to the use of linear codes in these hash functions, a cryptanalysis of them would probably be based for a large part on coding theory. It is an open problem whether our collision attack on the three-round version of HAVAL can be extended to versions with more rounds in the compression function. Attacks which find pseudo-collisions or almost-collisions would also be interesting. The MD5 algorithm is still very popular today despite the fact that collisions can be found for its compression function [39] (but not yet for the hash function itself). Trying to find real collisions for MD5 is therefore an interesting research problem. Alternatively, it would be very useful to implement a brute-force birthday attack on MD5. MD5 hash results are only 128 bits long, so such an attack, requiring about 2^{64} operations, should be feasible with today's equipment (using dedicated hardware, distributed computing, or both), but this has not been demonstrated yet.

Another cryptanalytic challenge is trying to find an improvement for our collision attack on PANAMA. The current attack is theoretical (much faster than a birthday attack but too complex to be practical), but we anticipate that by using methods such as relinearisation [71] for solving the systems of non-linear equations, collisions for PANAMA might actually be found. As a generic comment on hash function security, we note that more research is needed to study other properties of hash functions, for example the resistance against preimage attacks, or against attacks that analyse the pseudo-randomness of the output (when part of the input is secret). Additional evaluation of our own design Two-Track-MAC is welcome.

From the designing point of view there exists an interest for new types of hash functions, which should offer a high security level combined with good performance. It would also be desirable to have more provable security properties for hash functions (similar to block ciphers, where designers of new algorithms try to prove the resistance against, for example, differential and linear cryptanalysis). For hash functions that are based on a block cipher, it is still an open problem which requirements for the block cipher are sufficient in order to produce a secure hash function.

Bibliography

[1] R. Anderson and E. Biham, "Tiger: a fast new hash function." in *Fast Software Encryption* (D. Gollmann, ed.), no. 1039 in Lecture Notes in Computer Science, pp. 89–97, Springer-Verlag, 1996.

[2] ANSI X9.19, "Financial institution retail message authentication." American Bankers Association, 1996.

[3] ANSI X9.30.2, "Public key cryptography using irreversible algorithms for the financial services industry – Part 2: The Secure Hash Algorithm (SHA-1)." American Bankers Association, 1997.

[4] ANSI X9.31, "Public key cryptography using reversible algorithms for the financial services industry (rDSA)." American Bankers Association, 1998.

[5] ANSI X9.71, "Keyed hash message authentication code (MAC)." American Bankers Association, 2000.

[6] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-bit block cipher suitable for multiple platforms." Primitive submitted to NESSIE by NTT Corp., Sept. 2000. Available at `http://www.cryptonessie.org/`.

[7] P. S. L. M. Barreto and V. Rijmen, "The Anubis block cipher." Primitive submitted to NESSIE, Sept. 2000. Available at `http://www.cryptonessie.org/`.

[8] P. S. L. M. Barreto and V. Rijmen, "The Khazad legacy-level block cipher." Primitive submitted to NESSIE, Sept. 2000. Available at `http://www.cryptonessie.org/`.

[9] P. S. L. M. Barreto and V. Rijmen, "The Whirlpool hashing function." Primitive submitted to NESSIE, Sept. 2000. Available at `http://www.cryptonessie.org/`.

[10] D. Bayer, S. Haber, and W. S. Stornetta, "Improving the efficiency and reliability of digital time-stamping." in *Methods in Communication, Security, and Computer Science – Sequences'91*, pp. 329–334, Springer-Verlag, 1992.

[11] M. Bellare, R. Canetti, and H. Krawczyk, "Pseudorandom functions revisited: The cascade construction and its concrete security." in *Proceedings of the 37th Annual Symposium on the Foundations of Computer Science*, pp. 514–523, IEEE, 1996.

[12] M. Bellare, J. Kilian, and P. Rogaway, "The security of cipher block chaining." in *Advances in Cryptology – Crypto'94* (Y. Desmedt, ed.), no. 839 in Lecture Notes in Computer Science, pp. 341–358, Springer-Verlag, 1994.

[13] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm." in *Advances in Cryptology – Asiacrypt 2000* (T. Okamoto, ed.), no. 1976 in Lecture Notes in Computer Science, pp. 531–545, Springer-Verlag, 2000.

[14] M. Bellare and B. Yee, "Forward-security in private-key cryptography." in *Topics in Cryptology – CT-RSA'03* (M. Joye, ed.), no. 2612 in Lecture Notes in Computer Science, pp. 1–18, Springer-Verlag, 2003.

[15] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication." in *Advances in Cryptology – Crypto'96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 1–15, Springer-Verlag, 1996.

[16] I. Ben-Aroya and E. Biham, "Differential cryptanalysis of Lucifer." in *Advances in Cryptology – Crypto'93* (D. Stinson, ed.), no. 773 in Lecture Notes in Computer Science, pp. 187–199, Springer-Verlag, 1994.

[17] E. Biham, N. Keller, and O. Dunkelman, "Rectangle attacks on SHACAL-1." in *Fast Software Encryption* (T. Johansson, ed.), no. 2887 in Lecture Notes in Computer Science, pp. 22–35, Springer-Verlag, 2003.

[18] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. Springer, 1993.

[19] A. Biryukov and D. Wagner, "Slide attacks." in *Fast Software Encryption* (L. R. Knudsen, ed.), no. 1636 in Lecture Notes in Computer Science, pp. 245–259, Springer-Verlag, 1999.

[20] A. Biryukov and D. Wagner, "Advanced slide attacks." in *Advances in Cryptology – Eurocrypt 2000* (B. Preneel, ed.), no. 1807 in Lecture Notes in Computer Science, pp. 589–606, Springer-Verlag, 2000.

[21] J. Black and P. Rogaway, "A block-cipher mode of operation for parallelizable message authentication." in *Advances in Cryptology – Eurocrypt 2002* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 384–397, Springer-Verlag, 2002.

[22] J. Black, P. Rogaway, and T. Shrimpton, "Black-box analysis of the block-cipher-based hash-function constructions from PGV." in *Advances in Cryptology – Crypto 2002* (M. Yung, ed.), no. 2442 in Lecture Notes in Computer Science, pp. 320–335, Springer-Verlag, 2002.

[23] M. Blaze, W. Diffie, R. L. Rivest, B. Schneier, T. Shimomura, and E. Thompson, "Minimal key lengths for symmetric ciphers to provide adequate commercial security." Jan. 1996. Available at `http://www.schneier.com/paper-keylength.html`.

[24] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Fast hashing on the Pentium." in *Advances in Cryptology – Crypto'96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 298–312, Springer-Verlag, 1996.

[25] B. O. Brachtl, D. Coppersmith, M. M. Hyden, S. M. Matyas, C. H. Meyer, J. Oseas, and S. P. M. Schilling, "*Data Authentication using Modification Detection Codes Based on a Public One Way Encryption Function.*" U. S. Patent Number 4,908,861, Mar. 1990.

[26] A. Buldas, P. Laud, H. Lipmaa, and J. Villemson, "Time-stamping with binary linking schemes." in *Advances in Cryptology – Crypto'98* (H. Krawczyk, ed.), no. 1462 in Lecture Notes in Computer Science, pp. 486–501, Springer-Verlag, 1998.

[27] A. Buldas, H. Lipmaa, and B. Schoenmakers, "Optimally efficient accountable time-stamping." in *Public Key Cryptography* (H. Imai and Y. Zheng, eds.), no. 1751 in Lecture Notes in Computer Science, pp. 293–305, Springer-Verlag, 2000.

[28] J. L. Carter and M. N. Wegman, "Universal classes of hash functions." *Journal of Computer and System Sciences*, vol. 18, pp. 143–154, 1979.

[29] F. Chabaud and A. Joux, "Differential collisions in SHA-0." in *Advances in Cryptology – Crypto'98* (H. Krawczyk, ed.), no. 1462 in Lecture Notes in Computer Science, pp. 56–71, Springer-Verlag, 1998.

[30] J. Daemen and C. Clapp, "Fast hashing and stream encryption with Panama." in *Fast Software Encryption* (S. Vaudenay, ed.), no. 1372 in Lecture Notes in Computer Science, pp. 60–74, Springer-Verlag, 1998.

[31] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer, 2002.

[32] I. B. Damgård, "A design principle for hash functions." in *Advances in Cryptology – Crypto'89* (G. Brassard, ed.), no. 435 in Lecture Notes in Computer Science, pp. 416–427, Springer-Verlag, 1990.

[33] D. Davies, "A message authenticator algorithm suitable for a mainframe computer." in *Advances in Cryptology – Proceedings of Crypto'84* (G. R. Blakley and D. Chaum, eds.), no. 196 in Lecture Notes in Computer Science, pp. 393–400, Springer-Verlag, 1985.

[34] C. Debaert and H. Gilbert, "The RIPEMD^L and RIPEMD^R improved variants of MD4 are not collision free." in *Fast Software Encryption* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 52–65, Springer-Verlag, 2002.

[35] B. den Boer and A. Bosselaers, "An attack on the last two rounds of MD4." in *Advances in Cryptology – Crypto'91* (J. Feigenbaum, ed.), no. 576 in Lecture Notes in Computer Science, pp. 194–203, Springer-Verlag, 1992.

[36] B. den Boer and A. Bosselaers, "Collisions for the compression function of MD5." in *Advances in Cryptology – Eurocrypt'93* (T. Helleseth, ed.), no. 765 in Lecture Notes in Computer Science, pp. 293–304, Springer-Verlag, 1994.

[37] B. den Boer, B. Van Rompay, B. Preneel, and J. Vandewalle, "New (Two-Track-)MAC based on the two trails of RIPEMD." in *Selected Areas in Cryptography* (S. Vaudenay and A. M. Youssef, eds.), no. 2259 in Lecture Notes in Computer Science, pp. 314–324, Springer-Verlag, 2001.

[38] W. Diffie and M. E. Hellman, "New directions in cryptography." *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, 1976.

[39] H. Dobbertin, "The status of MD5 after a recent attack." *CryptoBytes*, vol. 2, no. 2, pp. 1,3–6, 1996.

[40] H. Dobbertin, "The status of MD5 after a recent attack." Unpublished manuscript, earlier version of [39], 1996.

[41] H. Dobbertin, "RIPEMD with two-round compress function is not collision-free." *Journal of Cryptology*, vol. 10, no. 1, pp. 51–70, 1997.

[42] H. Dobbertin, "Cryptanalysis of MD4." *Journal of Cryptology*, vol. 11, no. 4, pp. 253–271, 1998.

[43] H. Dobbertin, "The first two rounds of MD4 are not one-way." in *Fast Software Encryption* (S. Vaudenay, ed.), no. 1372 in Lecture Notes in Computer Science, pp. 284–292, Springer-Verlag, 1998.

[44] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160, a strengthened version of RIPEMD." in *Fast Software Encryption* (D. Gollmann, ed.), no. 1039 in Lecture Notes in Computer Science, pp. 71–82, Springer-Verlag, 1996.

[45] Electronic Frontier Foundation, *Cracking DES: Secrets of Encryption Research, Wiretap Politics, and Chip Design*. O'Reilly & Associates, 1998.

[46] J. H. Evertse and E. V. Heijst, "Which new RSA-signatures can be computed from certain given RSA-signatures?." *Journal of Cryptology*, vol. 5, no. 1, pp. 41–52, 1992.

[47] W. Feller, *An Introduction to Probability Theory and its Applications*. John Wiley & Sons, New York, 1968. 3rd edition.

[48] FIPS 113, "Computer Data Authentication." National Bureau of Standards, May 1985.

[49] FIPS 180, "Secure Hash Standard (SHS)." National Institute of Standards and Technology, May 1993. Replaced by [51].

[50] FIPS 180-1, "Secure Hash Standard (SHS)." National Institute of Standards and Technology, Apr. 1995. Replaced by [51].

[51] FIPS 180-2, "Secure Hash Standard (SHS)." National Institute of Standards and Technology, Aug. 2002. Change notice added in Feb. 2004.

[52] FIPS 197, "Advanced Encryption Standard." National Institute of Standards and Technology, Nov. 2001.

[53] FIPS 198, "The Keyed-Hash Message Authentication Code (HMAC)." National Institute of Standards and Technology, Mar. 2002.

[54] FIPS 46-3, "Data Encryption Standard." National Institute of Standards and Technology, Oct. 1999. (Specifies the use of Triple-DES).

[55] P. Flajolet and A. Odlyzko, "Random mapping statistics." in *Advances in Cryptology – Eurocrypt'89* (J.-J. Quisquater and J. Vandewalle, eds.), no. 434 in Lecture Notes in Computer Science, pp. 329–354, Springer-Verlag, 1990.

[56] H. Gilbert and H. Handschuh, "Security analysis of SHA-256 and sisters." in *Selected Areas in Cryptography* (M. Matsui and R. Zuccherato, eds.), no. 3006 in Lecture Notes in Computer Science, pp. 175–193, Springer-Verlag, 2004.

[57] S. Goldwasser and M. Bellare, "Lecture notes on cryptography." Aug. 2001. Available at `http://www-cse.ucsd.edu/users/mihir/papers/gb.html`.

[58] S. Haber and W. S. Stornetta, "How to time-stamp a digital document." *Journal of Cryptology*, vol. 3, no. 2, pp. 99–111, 1991.

[59] S. Haber and W. S. Stornetta, "Secure names for bit-strings." in *Conference on Computer and Communications Security – CCS'97*, pp. 28–35, ACM Press, 1997.

[60] H. Handschuh and D. Naccache, "SHACAL." Primitive submitted to NESSIE, Sept. 2000. Available at `http://www.cryptonessie.org/`.

[61] S. Hong, J. Kim, G. Kim, J. Sung, C. Lee, and S. Lee, "Impossible differential attack on 30-round SHACAL-2." in *Progress in Cryptology – Indocrypt 2003* (T. Johansson and S. Maitra, eds.), no. 2904 in Lecture Notes in Computer Science, pp. 97–106, Springer-Verlag, 2003.

[62] ISO/IEC 10118-2, "Information Technology - Security Techniques - Hash-functions - Part 2: Hash-functions using an n-bit block cipher." 2000.

[63] ISO/IEC 10118-3, "Information Technology - Security Techniques - Hash-functions - Part 3: Dedicated hash-functions." 2004.

[64] ISO/IEC 10118-4, "Information Technology - Security Techniques - Hash-functions - Part 4: Hash-functions using modular arithmetic." 1998.

[65] ISO/IEC 9797-1, "Information Technology - Security Techniques - Message Authentication Codes (MACs) - Part 1: Mechanisms using a block cipher." 1999.

[66] ISO/IEC 9797-2, "Information Technology - Security Techniques - Message Authentication Codes (MACs) - Part 2: Mechanisms using a dedicated hash-function." 2002.

[67] T. Iwata and K. Kurosawa, "OMAC: One-key CBC MAC." in *Fast Software Encryption* (T. Johansson, ed.), no. 2887 in Lecture Notes in Computer Science, pp. 129–153, Springer-Verlag, 2003.

[68] D. Kahn, *The Codebreakers : The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996. Revised edition.

[69] P. Kasselman and W. Penzhorn, "Cryptanalysis of reduced version of HAVAL." *Electronics Letters*, vol. 36, no. 1, pp. 30–31, 2000.

[70] J. Kim, D. Moon, W. Lee, S. Hong, S. Lee, and S. Jung, "Amplified boomerang attack against reduced-round SHACAL." in *Advances in Cryptology – Asiacrypt 2002* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 243–253, Springer-Verlag, 2002.

[71] A. Kipnis and A. Shamir, "Cryptanalysis of the HFE public key cryptosystem by relinearization." in *Advances in Cryptology – Crypto'99* (M. Wiener, ed.), no. 1666 in Lecture Notes in Computer Science, pp. 19–30, Springer-Verlag, 1999.

[72] L. R. Knudsen, "Chosen-text attack on CBC-MAC." *Electronics Letters*, vol. 33, no. 1, pp. 48–49, 1997.

[73] L. R. Knudsen, "Contemporary block ciphers." in *Lectures on Data Security. Modern Cryptology in Theory and Practice* (I. B. Damgård, ed.), no. 1561 in Lecture Notes in Computer Science, pp. 105–126, Springer-Verlag, 1999.

[74] L. R. Knudsen, X. Lai, and B. Preneel, "Attacks on fast double block length hash functions." *Journal of Cryptology*, vol. 11, no. 1, pp. 59–72, 1998.

[75] L. R. Knudsen and B. Preneel, "Fast and secure hashing based on codes." in *Advances in Cryptology – Crypto'97* (B. Kaliski, ed.), no. 1294 in Lecture Notes in Computer Science, pp. 485–498, Springer-Verlag, 1997.

[76] T. Krovetz, J. Black, S. Halevi, A. Hevia, H. Krawczyk, and P. Rogaway, "UMAC." Primitive submitted to NESSIE, Sept. 2000. Available at `http://www.cryptonessie.org/`.

[77] H. Kuwakado and H. Tanaka, "New algorithm for finding preimages in a reduced version of the MD4 compression function." *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E83-A, no. 1, pp. 97–100, 2000.

[78] M. Kwan, "The design of the ICE encryption algorithm." in *Fast Software Encryption* (E. Biham, ed.), no. 1267 in Lecture Notes in Computer Science, pp. 69–82, Springer-Verlag, 1997.

[79] X. Lai and J. L. Massey, “Hash functions based on block ciphers.” in *Advances in Cryptology – Eurocrypt’92* (R. A. Rueppel, ed.), no. 658 in Lecture Notes in Computer Science, pp. 55–70, Springer-Verlag, 1993.

[80] H. Lipmaa, *Secure and Efficient Time-stamping Systems*. Doctoral dissertation, University of Tartu, 1999.

[81] M. Matsui, “Linear cryptanalysis method for DES cipher.” in *Advances in Cryptology – Eurocrypt’93* (T. Helleseth, ed.), no. 765 in Lecture Notes in Computer Science, pp. 386–397, Springer-Verlag, 1994.

[82] M. Matsui, “Specification of MISTY1 – a 64-bit block cipher.” Primitive submitted to NESSIE by E. Takeda, Sept. 2000. Available at `http://www.cryptonessie.org/`.

[83] S. M. Matyas, C. H. Meyer, and J. Oseas, “Generating strong one-way functions with cryptographic algorithm.” *IBM Techn. Disclosure Bull.*, vol. 27, no. 10A, pp. 5658–5659, 1985.

[84] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.

[85] R. C. Merkle, “A certified digital signature.” in *Advances in Cryptology – Crypto’89* (G. Brassard, ed.), no. 435 in Lecture Notes in Computer Science, pp. 218–238, Springer-Verlag, 1990.

[86] R. C. Merkle, “One-way hash functions and DES.” in *Advances in Cryptology – Crypto’89* (G. Brassard, ed.), no. 435 in Lecture Notes in Computer Science, pp. 428–446, Springer-Verlag, 1990.

[87] C. H. Meyer and M. Schilling, “Secure program load with manipulation detection code.” in *Proceedings Securicom*, pp. 111–130, 1988.

[88] S. Miyaguchi, M. Iwata, and K. Ohta, “New 128-bit hash function.” in *Proceedings 4th International Joint Workshop on Computer Communications*, pp. 279–288, 1989.

[89] S. Miyaguchi, K. Ohta, and M. Iwata, “Confirmation that some hash functions are not collision free.” in *Advances in Cryptology – Eurocrypt’90* (I. B. Damgård, ed.), no. 473 in Lecture Notes in Computer Science, pp. 326–343, Springer-Verlag, 1991.

[90] J. Nakajima and M. Matsui, “Performance analysis and parallel implementation of dedicated hash functions.” in *Advances in Cryptology – Eurocrypt 2002* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 165–180, Springer-Verlag, 2002.

[91] NESSIE consortium, "NESSIE Security Report." Deliverable report D20, NESSIE, 2002. Available at `http://www.cryptonessie.org/`.

[92] NESSIE consortium, "Performance of Optimized Implementations of the NESSIE Primitives." Deliverable report D21, NESSIE, 2002. Available at `http://www.cryptonessie.org/`.

[93] NESSIE consortium, "NESSIE Portfolio of recommended cryptographic primitives." Public report, NESSIE, Feb. 2003. Available at `http://www.cryptonessie.org/`.

[94] NESSIE consortium, "NESSIE Project Announces Final Selection of Crypto Algorithms." Press release, NESSIE, Feb. 2003. Available at `http://www.cryptonessie.org/`.

[95] E. Petrank and C. Rackoff, "CBC MAC for real-time data sources." *Journal of Cryptology*, vol. 13, no. 3, pp. 315–338, 2000.

[96] B. Preneel, *Analysis and Design of Cryptographic Hash Functions*. Doctoral dissertation, K. U. Leuven, Jan. 1993.

[97] B. Preneel, "Custom designed hash functions." Unpublished manuscript, 1995.

[98] B. Preneel, "Cryptographic primitives for information authentication — state of the art." in *State of the Art in Applied Cryptography* (B. Preneel and V. Rijmen, eds.), no. 1528 in Lecture Notes in Computer Science, pp. 50–105, Springer-Verlag, 1998.

[99] B. Preneel, "The state of cryptographic hash functions." in *Lectures on Data Security. Modern Cryptology in Theory and Practice* (I. B. Damgård, ed.), no. 1561 in Lecture Notes in Computer Science, pp. 158–182, Springer-Verlag, 1999.

[100] B. Preneel, A. Bosselaers, and H. Dobbertin, "The cryptographic hash function RIPEMD-160." *CryptoBytes*, vol. 3, no. 2, pp. 9–14, 1997.

[101] B. Preneel, R. Govaerts, and J. Vandewalle, "Cryptographically secure hash functions: an overview." ESAT Internal Report, K. U. Leuven, 1989.

[102] B. Preneel, R. Govaerts, and J. Vandewalle, "Hash functions based on block ciphers: A synthetic approach." in *Advances in Cryptology – Crypto'93* (D. Stinson, ed.), no. 773 in Lecture Notes in Computer Science, pp. 368–378, Springer-Verlag, 1994.

[103] B. Preneel, V. Rijmen, and P. C. van Oorschot, "A security analysis of the Message Authenticator Algorithm (MAA)." *European Transactions on Telecommunications*, vol. 8, no. 5, pp. 455–470, 1997.

[104] B. Preneel and P. C. van Oorschot, "MDx-MAC and building fast MACs from hash functions." in *Advances in Cryptology – Crypto'95* (D. Coppersmith, ed.), no. 963 in Lecture Notes in Computer Science, pp. 1–14, Springer-Verlag, 1995.

[105] B. Preneel and P. C. van Oorschot, "On the security of two MAC algorithms." in *Advances in Cryptology – Eurocrypt'96* (U. Maurer, ed.), no. 1070 in Lecture Notes in Computer Science, pp. 19–32, Springer-Verlag, 1996.

[106] B. Preneel and P. C. van Oorschot, "On the security of iterated message authentication codes." *IEEE Transactions on Information Theory*, vol. IT-45, no. 1, pp. 188–199, 1999.

[107] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? Application to DES." in *Advances in Cryptology – Eurocrypt'89* (J.-J. Quisquater and J. Vandewalle, eds.), no. 434 in Lecture Notes in Computer Science, pp. 429–434, Springer-Verlag, 1990.

[108] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search. New results and applications to DES." in *Advances in Cryptology – Crypto'89* (G. Brassard, ed.), no. 435 in Lecture Notes in Computer Science, pp. 408–413, Springer-Verlag, 1990.

[109] M. Quisquater, *Applications of Character Theory and the Möbius Inversion Principle to the Study of Cryptographic Properties of Boolean Functions*. Doctoral dissertation, K. U. Leuven, May 2004.

[110] V. Rijmen, *Cryptanalysis and Design of Iterated Block Ciphers*. Doctoral dissertation, K. U. Leuven, Oct. 1997.

[111] V. Rijmen and B. Preneel, "Improved characteristics for differential cryptanalysis of hash functions based on block ciphers." in *Fast Software Encryption* (B. Preneel, ed.), no. 1008 in Lecture Notes in Computer Science, pp. 242–248, Springer-Verlag, 1995.

[112] V. Rijmen, B. Van Rompay, B. Preneel, and J. Vandewalle, "Producing collisions for PANAMA." in *Fast Software Encryption* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 37–51, Springer-Verlag, 2002.

[113] RIPE, *Integrity Primitives for Secure Information Systems. Final Report of RACE Integrity Primitives Evaluation (RIPE-RACE 1040)* (A. Bosselaers and B. Preneel, eds.). No. 1007 in Lecture Notes in Computer Science, Springer-Verlag, 1995.

[114] R. L. Rivest, "The MD4 message digest algorithm." in *Advances in Cryptology – Crypto'90* (A. Menezes and S. A. Vanstone, eds.), no. 537 in Lecture Notes in Computer Science, pp. 303–311, Springer-Verlag, 1991.

[115] R. L. Rivest, "The MD5 message-digest algorithm." IETF Request for Comments, RFC 1321, Apr. 1992.

[116] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[117] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: definitions, implications, and seperations for preimage resistance, second-preimage resistance, and collision resistance." in *Fast Software Encryption* (B. Roy and W. Meier, eds.), no. 3017 in Lecture Notes in Computer Science, Springer-Verlag, 2004 (to appear).

[118] M.-J. O. Saarinen, "Cryptanalysis of block ciphers based on SHA-1 and MD5." in *Fast Software Encryption* (T. Johansson, ed.), no. 2887 in Lecture Notes in Computer Science, pp. 36–44, Springer-Verlag, 2003.

[119] C. E. Shannon, "Communication theory of secrecy systems." *Bell System Technical Journal*, vol. 28, pp. 656–715, Oct. 1949.

[120] "Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques." National Institute of Standards and Technology. To be published at `http://csrc.nist.gov/publications/fips/`.

[121] "Special Publication 800-38B: Recommendation for Block Cipher Modes of Operation: the OMAC Authentication Mode." National Institute of Standards and Technology. To be published at `http://csrc.nist.gov/publications/fips/`.

[122] P. C. van Oorschot and M. J. Wiener, "Parallel collision search with cryptanalytic applications." *Journal of Cryptology*, vol. 12, no. 1, pp. 1–28, 1999.

[123] B. Van Rompay, A. Biryukov, B. Preneel, and J. Vandewalle, "Cryptanalysis of 3-pass HAVAL." in *Advances in Cryptology – Asiacrypt 2003* (C. S. Laih, ed.), no. 2894 in Lecture Notes in Computer Science, pp. 228–245, Springer-Verlag, 2003.

[124] B. Van Rompay and B. den Boer, "Two-Track-MAC." Primitive submitted to NESSIE, Sept. 2000. Available at `http://www.cryptonessie.org/`.

[125] B. Van Rompay, L. R. Knudsen, and V. Rijmen, "Differential cryptanalysis of the ICE encryption algorithm." in *Fast Software Encryption* (S. Vaudenay, ed.), no. 1372 in Lecture Notes in Computer Science, pp. 270–283, Springer-Verlag, 1998.

[126] B. Van Rompay, B. Preneel, and J. Vandewalle, "On the security of dedicated hash functions." in *Proceedings 19th Symposium on Information Theory in the Benelux* (M. van der Schaar-Mitrea and P. H. N. de With, eds.), pp. 103–110, 1998.

[127] B. Van Rompay, B. Preneel, and J. Vandewalle, "The digital timestamping problem." in *Proceedings 20th Symposium on Information Theory in the Benelux* (A. Barbé, E. C. van der Meulen, and P. Vanroose, eds.), pp. 71–78, 1999.

[128] S. Vaudenay, "On the need for multipermutations: Cryptanalysis of MD4 and SAFER." in *Fast Software Encryption* (B. Preneel, ed.), no. 1008 in Lecture Notes in Computer Science, pp. 286–297, Springer-Verlag, 1995.

[129] G. S. Vernam, "Cipher printing telegraph system for secret wire and radio telegraph communications." *Journal American Institute of Electrical Engineers*, vol. XLV, pp. 109–115, 1926.

[130] G. Yuval, "How to swindle Rabin." *Cryptologia*, vol. 3, pp. 187–189, 1979.

[131] Y. Zheng, J. Pieprzyk, and J. Seberry, "HAVAL – a one-way hashing algorithm with variable length of output." in *Advances in Cryptology – Auscrypt'92* (J. Seberry and Y. Zheng, eds.), no. 718 in Lecture Notes in Computer Science, pp. 83–104, Springer-Verlag, 1993.

[132] "The hash function RIPEMD-160.". `http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html`.

[133] "Project DES.". `http://www.distributed.net/des/`.

[134] "Digital Time-Stamping and the Evaluation of Security Primitives." 1996–1999. `http://www.dice.ucl.ac.be/crypto/TIMESEC/TIMESEC.html`.

[135] "New European Schemes for Signatures, Integrity, and Encryption." 2000–2003. `http://www.cryptonessie.org/`.

Appendix A

Specification of Two-Track-MAC

A high level description of the Two-Track-MAC (TTMAC) algorithm has been given in Chapter 6 (Sect. 6.8). In this appendix we give the source code for a reference implementation of Two-Track-MAC, and we provide test vectors.

Source Code

```
/* --------------------------------------------------------------
 *
 *  C Implementation TTMAC Message Authentication
 *
 *  Files:   ttmac.h, ttmac.c, test.c
 *
 *  Author:  Bart Van Rompay, ESAT/SCD-COSIC
 *  Note:    based on RIPEMD-160 code of
 *           Antoon Bosselaers (ESAT/SCD-COSIC)
 *  Date:    June 2004
 *  Version: 1.0
 *
 *  Copyright (c) Katholieke Universiteit Leuven
 *  2004, All Rights Reserved
 *
 *  Conditions for use of the TTMAC Software
 *
 *  The TTMAC software is freely available for use under the
 *  terms and conditions described hereunder, which shall be
 *  deemed to be accepted by any user of the software and
 *  applicable on any use of the software:
 *
 *  1. K.U.Leuven Department of Electrical Engineering
 *     (ESAT/SCD-COSIC) shall for all purposes be considered
 *     the owner of the TTMAC software and of all copyright,
 *     trade secret, patent or other intellectual property
 *     rights therein.
 *  2. The TTMAC software is provided on an "as is" basis
 *     without warranty of any sort, express or implied.
 *     K.U.Leuven makes no representation that the use of the
 *     software will not infringe any patent or proprietary
 *     right of third parties. User will indemnify K.U.Leuven
 *     and hold K.U.Leuven harmless from any claims or
 *     liabilities which may arise as a result of its use of
 *     the software. In no circumstances K.U.Leuven R&D will
 *     be held liable for any deficiency, fault or other
 *     mishappening with regard to the use or performance of
 *     the software.
 *  3. User agrees to give due credit to K.U.Leuven in
 *     scientific publications or communications in relation
 *     with the use of the TTMAC software.
 *
 * ------------------------------------------------------- */


/*** file ttmac.h ***/

/** typedef 8 and 32 bit types, resp.  **/
/* adapt these, if necessary,
   for your operating system and compiler */

typedef  unsigned char  byte;
typedef  unsigned long  dword;

/** define constants **/
```

```
/* length of left and right internal variable */
#define STATEsize 160

/* length of message block */
#define BLOCKsize 512

/* MAC output length : 32, 64, 96, 128 or 160 bits */
#define MACsize   160

/** define structure used to store the expanded key
    and the state (note: the expanded key is the
    initial state)  **/

  struct MDstruct {

  /* left  160-bit state variable */
  dword         L[STATEsize/32];

  /* right 160-bit state variable */
  dword         R[STATEsize/32];

  /* buffer for incomplete message block */
  byte          buf[BLOCKsize/8-1];

  /* number of bytes buffered */
  unsigned int  bufbytes;

  /* number of bytes processed */
  unsigned long nbytes;

};

/** macro definitions **/

/* collect four bytes into one word (little-endian): */
#define BYTES_TO_DWORD(strptr)               \
            (((dword) *((strptr)+3) << 24) | \
             ((dword) *((strptr)+2) << 16) | \
             ((dword) *((strptr)+1) <<  8) | \
             ((dword) *(strptr)))
```

```
/* extract a byte from a word */
#define DWORD_TO_BYTES(x, i) (x[(i)>>2] >> (8*((i)&3)))

/* ROL(x, n) cyclically rotates x over n bits to the left */
/* x must be of an unsigned 32 bits type and 0 <= n < 32. */
#define ROL(x, n)        (((x) << (n)) | ((x) >> (32-(n))))

/* the five basic functions F(), G(), H(), I() and J() */
#define F(x, y, z)        ((x) ^ (y) ^ (z))
#define G(x, y, z)        (((x) & (y)) | (~(x) & (z)))
#define H(x, y, z)        (((x) | ~(y)) ^ (z))
#define I(x, y, z)        (((x) & (z)) | ((y) & ~(z)))
#define J(x, y, z)        ((x) ^ ((y) | ~(z)))

/* the ten basic operations FF() through JJJ() */
#define FF(a, b, c, d, e, x, s)        {\
      (a) += F((b), (c), (d)) + (x);\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define GG(a, b, c, d, e, x, s)        {\
      (a) += G((b), (c), (d)) + (x) + 0x5a827999UL;\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define HH(a, b, c, d, e, x, s)        {\
      (a) += H((b), (c), (d)) + (x) + 0x6ed9eba1UL;\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define II(a, b, c, d, e, x, s)        {\
      (a) += I((b), (c), (d)) + (x) + 0x8f1bbcdcUL;\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define JJ(a, b, c, d, e, x, s)        {\
      (a) += J((b), (c), (d)) + (x) + 0xa953fd4eUL;\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define FFF(a, b, c, d, e, x, s)        {\
      (a) += F((b), (c), (d)) + (x);\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define GGG(a, b, c, d, e, x, s)        {\
      (a) += G((b), (c), (d)) + (x) + 0x7a6d76e9UL;\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define HHH(a, b, c, d, e, x, s)        {\
      (a) += H((b), (c), (d)) + (x) + 0x6d703ef3UL;\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define III(a, b, c, d, e, x, s)        {\
      (a) += I((b), (c), (d)) + (x) + 0x5c4dd124UL;\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }
#define JJJ(a, b, c, d, e, x, s)        {\
      (a) += J((b), (c), (d)) + (x) + 0x50a28be6UL;\
      (a) = ROL((a), (s)) + (e);\
      (c) = ROL((c), 10);\
   }

/** function prototypes **/

void MDkeysetup(
     const unsigned char * const key,
     struct MDstruct * const structpointer);

void MDadd(
     const unsigned char * const plaintext,
     unsigned long numberofbits,
     struct MDstruct * const structpointer);

void MDfinalize(
     const struct MDstruct * const structpointer,
     unsigned char * const result);

void mix(struct MDstruct * const sp);
void ltrail(dword *A, dword *M);
void rtrail(dword *B, dword *M);

/*** end of file ttmac.h ***/


/*** file ttmac.c ***/

/**  header files **/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ttmac.h"

/** key expansion and state initialisation
   (note: the expanded key is the initial state) **/

void MDkeysetup(
     const unsigned char * const key,
     struct MDstruct * const structpointer)
{
  unsigned int i;

  for (i=0; i<(STATEsize/32); i++)
    structpointer->L[i] = structpointer->R[i] \
                        = BYTES_TO_DWORD(key+i*4);

  structpointer->nbytes = 0;
  structpointer->bufbytes = 0;

  return;
}

/**- modify the state according to the data at plaintext
   - numberofbits tells the function how many bits of
     plaintext to process
   - if numberofbits is not a multiple of BLOCKsize, the last
     incomplete block is stored in the buffer of structpointer
   - if structpointer already contains such a buffer when the
     function is called the data therein is prepended to
     plaintext **/
```

```
void MDadd(
     const unsigned char * const plaintext,
     unsigned long numberofbits,
     struct MDstruct * const structpointer)
{
  unsigned int  i;
  unsigned int  bytesinbuf = structpointer->bufbytes;
  unsigned long numberofbytes = numberofbits/8;
  unsigned long bytesleft;
  byte          *message;
  dword         M[BLOCKsize/32];  /* message words */

  /* case 1: (buffer +) plaintext is shorter than one block
             store the plaintext in the buffer and return */

  if ( (numberofbytes + bytesinbuf) < (BLOCKsize/8) ) {
    message = (byte *) plaintext;
    for (i=0; i<numberofbytes; i++)
      structpointer->buf[bytesinbuf + i] = *message++;
    structpointer->bufbytes += numberofbytes;
    return;
  }

  /* case 2a: buffer + plaintext is at least one block
              put bytes from buffer at start of message */
  /* note: this is inefficient so try to avoid this case */

  if (bytesinbuf) {
    message = (byte*)malloc(numberofbytes + bytesinbuf);

    for (i=(numberofbytes + bytesinbuf); i>bytesinbuf; i--)
      *(message + i - 1) = *(plaintext + i - 1 - bytesinbuf);
    for (i=bytesinbuf; i>0; i--)
      *(message + i - 1) = structpointer->buf[i - 1];
  }

  /* case 2b: no buffer, plaintext is at least one block */

  else message = (byte *) plaintext;
  /* process the plaintext in blocks */

  for (bytesleft=(numberofbytes + bytesinbuf);
       bytesleft>(BLOCKsize/8 - 1);
       bytesleft-=(BLOCKsize/8)) {

    for (i=0; i<(BLOCKsize/32); i++) {
      M[i] = BYTES_TO_DWORD(message);
      message += 4;
    }

    ltrail(structpointer->L, M); rtrail(structpointer->R, M);
    mix(structpointer);
    structpointer->nbytes += BLOCKsize/8;
  }

  /* store the last incomplete block in the buffer */

  for (i=0; i<bytesleft; i++)
    structpointer->buf[i] = *message++;

  structpointer->bufbytes = bytesleft;

  if (bytesinbuf) { free(message); }

  return;
}

/**- get the MAC value from the state
   - first process the data left in the buffer of
     structpointer, add padding (including length info)
   - truncate the result (depends on the value of MACsize)
   - store the MAC value at the address of the parameter
     result **/

void MDfinalize(
     const struct MDstruct * const structpointer,
     unsigned char * const result)
{
  unsigned int        i;
  struct MDstruct     state = *structpointer;
```

```
unsigned int        bytesinbuf = state.bufbytes;
unsigned long       bytelength = \
                    (state.nbytes) + bytesinbuf;
dword               M[BLOCKsize/32];  /* message words */
dword               E[STATEsize/32];  /* complete output */
dword               F[MACsize/32];    /* truncated output */

memset(M, 0, BLOCKsize/32*sizeof(dword));

/* put bytes from buffer into M
   byte i goes into word M[i div 4] at pos. 8*(i mod 4) */

for (i=0; i<bytesinbuf; i++)
  M[i>>2] ^= (dword) ((state.buf[i]) << (8 * (i&3)));

/* append the bit 1 */

M[bytesinbuf>>2] ^= (dword) (1 << (8 * (bytesinbuf&3) + 7));

if (bytesinbuf > 55) {
  /* length goes to next block */

  ltrail(state.L, M); rtrail(state.R, M);
  mix(&state);

  memset(M, 0, BLOCKsize/32*sizeof(dword));
}

/* append length in bits */

M[14] = bytelength << 3;
M[15] = bytelength >> 29;

/* process the last block
   (note: reversal of left and right trail) */

rtrail(state.L, M); ltrail(state.R, M);

/* combine trails to produce MAC (modular subtraction) */

for (i=0; i<(STATEsize/32); i++)
```

```
    E[i] = (state.L[i]) - (state.R[i]);

  /* optional truncation of the output (MAC result) */

#if (MACsize == 160)
  for (i=0; i<(MACsize/8); i++) \
  result[i] = DWORD_TO_BYTES(E, i);
#else
#if (MACsize == 32)
  F[0] = E[0] + E[1] + E[2] + E[3] + E[4];
#else
  F[0] = E[0] + E[1] + E[3];
  F[1] = E[1] + E[2] + E[4];
#if (MACsize > 64)
  F[2] = E[2] + E[3] + E[0];
#if (MACsize > 96)
  F[3] = E[3] + E[4] + E[1];
#endif
#endif
#endif
  for (i=0; i<(MACsize/8); i++) \
  result[i] = DWORD_TO_BYTES(F, i);
#endif

  return;
}

void mix(struct MDstruct * const sp)
{
  dword a0 = sp->L[0], a1 = sp->L[1], a2 = sp->L[2],
        a3 = sp->L[3], a4 = sp->L[4];

  dword b4 = sp->R[4];

  sp->L[0] = (a1 + a4) - sp->R[3];
  sp->L[1] = a2 - sp->R[4];
  sp->L[2] = a3 - sp->R[0];
  sp->L[3] = a4 - sp->R[1];
  sp->L[4] = a0 - sp->R[2];

  sp->R[4] = a2 - sp->R[3];
  sp->R[3] = a1 - sp->R[2];
  sp->R[2] = a0 - sp->R[1];
  sp->R[1] = (a4 + a2) - sp->R[0];
  sp->R[0] = a3 - b4;

  return;
}

void ltrail(dword *A, dword *M)
{
  dword a0 = A[0], a1 = A[1], a2 = A[2], a3 = A[3], a4 = A[4];

  /* round 1 */
  FF(a0, a1, a2, a3, a4, M[ 0], 11);
  FF(a4, a0, a1, a2, a3, M[ 1], 14);
  FF(a3, a4, a0, a1, a2, M[ 2], 15);
  FF(a2, a3, a4, a0, a1, M[ 3], 12);
  FF(a1, a2, a3, a4, a0, M[ 4],  5);
  FF(a0, a1, a2, a3, a4, M[ 5],  8);
  FF(a4, a0, a1, a2, a3, M[ 6],  7);
  FF(a3, a4, a0, a1, a2, M[ 7],  9);
  FF(a2, a3, a4, a0, a1, M[ 8], 11);
  FF(a1, a2, a3, a4, a0, M[ 9], 13);
  FF(a0, a1, a2, a3, a4, M[10], 14);
  FF(a4, a0, a1, a2, a3, M[11], 15);
  FF(a3, a4, a0, a1, a2, M[12],  6);
  FF(a2, a3, a4, a0, a1, M[13],  7);
  FF(a1, a2, a3, a4, a0, M[14],  9);
  FF(a0, a1, a2, a3, a4, M[15],  8);

  /* round 2 */
  GG(a4, a0, a1, a2, a3, M[ 7],  7);
  GG(a3, a4, a0, a1, a2, M[ 4],  6);
  GG(a2, a3, a4, a0, a1, M[13],  8);
  GG(a1, a2, a3, a4, a0, M[ 1], 13);
  GG(a0, a1, a2, a3, a4, M[10], 11);
  GG(a4, a0, a1, a2, a3, M[ 6],  9);
  GG(a3, a4, a0, a1, a2, M[15],  7);
  GG(a2, a3, a4, a0, a1, M[ 3], 15);
  GG(a1, a2, a3, a4, a0, M[12],  7);
  GG(a0, a1, a2, a3, a4, M[ 0], 12);
  GG(a4, a0, a1, a2, a3, M[ 9], 15);
  GG(a3, a4, a0, a1, a2, M[ 5],  9);
  GG(a2, a3, a4, a0, a1, M[ 2], 11);
  GG(a1, a2, a3, a4, a0, M[14],  7);
  GG(a0, a1, a2, a3, a4, M[11], 13);
  GG(a4, a0, a1, a2, a3, M[ 8], 12);

  /* round 3 */
  HH(a3, a4, a0, a1, a2, M[ 3], 11);
  HH(a2, a3, a4, a0, a1, M[10], 13);
  HH(a1, a2, a3, a4, a0, M[14],  6);
  HH(a0, a1, a2, a3, a4, M[ 4],  7);
  HH(a4, a0, a1, a2, a3, M[ 9], 14);
  HH(a3, a4, a0, a1, a2, M[15],  9);
  HH(a2, a3, a4, a0, a1, M[ 8], 13);
  HH(a1, a2, a3, a4, a0, M[ 1], 15);
  HH(a0, a1, a2, a3, a4, M[ 2], 14);
  HH(a4, a0, a1, a2, a3, M[ 7],  8);
  HH(a3, a4, a0, a1, a2, M[ 0], 13);
  HH(a2, a3, a4, a0, a1, M[ 6],  6);
  HH(a1, a2, a3, a4, a0, M[13],  5);
  HH(a0, a1, a2, a3, a4, M[11], 12);
  HH(a4, a0, a1, a2, a3, M[ 5],  7);
  HH(a3, a4, a0, a1, a2, M[12],  5);

  /* round 4 */
  II(a2, a3, a4, a0, a1, M[ 1], 11);
  II(a1, a2, a3, a4, a0, M[ 9], 12);
  II(a0, a1, a2, a3, a4, M[11], 14);
  II(a4, a0, a1, a2, a3, M[10], 15);
  II(a3, a4, a0, a1, a2, M[ 0], 14);
  II(a2, a3, a4, a0, a1, M[ 8], 15);
  II(a1, a2, a3, a4, a0, M[12],  9);
  II(a0, a1, a2, a3, a4, M[ 4],  8);
  II(a4, a0, a1, a2, a3, M[13],  9);
  II(a3, a4, a0, a1, a2, M[ 3], 14);
  II(a2, a3, a4, a0, a1, M[ 7],  5);
  II(a1, a2, a3, a4, a0, M[15],  6);
  II(a0, a1, a2, a3, a4, M[14],  8);
  II(a4, a0, a1, a2, a3, M[ 5],  6);
  II(a3, a4, a0, a1, a2, M[ 6],  5);
```

```
  II(a2, a3, a4, a0, a1, M[ 2], 12);

  /* round 5 */
  JJ(a1, a2, a3, a4, a0, M[ 4],  9);
  JJ(a0, a1, a2, a3, a4, M[ 0], 15);
  JJ(a4, a0, a1, a2, a3, M[ 5],  5);
  JJ(a3, a4, a0, a1, a2, M[ 9], 11);
  JJ(a2, a3, a4, a0, a1, M[ 7],  6);
  JJ(a1, a2, a3, a4, a0, M[12],  8);
  JJ(a0, a1, a2, a3, a4, M[ 2], 13);
  JJ(a4, a0, a1, a2, a3, M[10], 12);
  JJ(a3, a4, a0, a1, a2, M[14],  5);
  JJ(a2, a3, a4, a0, a1, M[ 1], 12);
  JJ(a1, a2, a3, a4, a0, M[ 3], 13);
  JJ(a0, a1, a2, a3, a4, M[ 8], 14);
  JJ(a4, a0, a1, a2, a3, M[11], 11);
  JJ(a3, a4, a0, a1, a2, M[ 6],  8);
  JJ(a2, a3, a4, a0, a1, M[15],  5);
  JJ(a1, a2, a3, a4, a0, M[13],  6);

  A[0] = a0 - A[0];
  A[1] = a1 - A[1];
  A[2] = a2 - A[2];
  A[3] = a3 - A[3];
  A[4] = a4 - A[4];

  return;
}

void rtrail(dword *B, dword *M)
{
  dword b0 = B[0], b1 = B[1], b2 = B[2], b3 = B[3], b4 = B[4];

  /* parallel round 1 */
  JJJ(b0, b1, b2, b3, b4, M[ 5],  8);
  JJJ(b4, b0, b1, b2, b3, M[14],  9);
  JJJ(b3, b4, b0, b1, b2, M[ 7],  9);
  JJJ(b2, b3, b4, b0, b1, M[ 0], 11);
  JJJ(b1, b2, b3, b4, b0, M[ 9], 13);
  JJJ(b0, b1, b2, b3, b4, M[ 2], 15);
  JJJ(b4, b0, b1, b2, b3, M[11], 15);
  JJJ(b3, b4, b0, b1, b2, M[ 4],  5);
  JJJ(b2, b3, b4, b0, b1, M[13],  7);
  JJJ(b1, b2, b3, b4, b0, M[ 6],  7);
  JJJ(b0, b1, b2, b3, b4, M[15],  8);
  JJJ(b4, b0, b1, b2, b3, M[ 8], 11);
  JJJ(b3, b4, b0, b1, b2, M[ 1], 14);
  JJJ(b2, b3, b4, b0, b1, M[10], 14);
  JJJ(b1, b2, b3, b4, b0, M[ 3], 12);
  JJJ(b0, b1, b2, b3, b4, M[12],  6);

  /* parallel round 2 */
  III(b4, b0, b1, b2, b3, M[ 6],  9);
  III(b3, b4, b0, b1, b2, M[11], 13);
  III(b2, b3, b4, b0, b1, M[ 3], 15);
  III(b1, b2, b3, b4, b0, M[ 7],  7);
  III(b0, b1, b2, b3, b4, M[ 0], 12);
  III(b4, b0, b1, b2, b3, M[13],  8);
  III(b3, b4, b0, b1, b2, M[ 5],  9);
  III(b2, b3, b4, b0, b1, M[10], 11);
  III(b1, b2, b3, b4, b0, M[14],  7);
  III(b0, b1, b2, b3, b4, M[15],  7);
  III(b4, b0, b1, b2, b3, M[ 8], 12);
  III(b3, b4, b0, b1, b2, M[12],  7);
  III(b2, b3, b4, b0, b1, M[ 4],  6);
  III(b1, b2, b3, b4, b0, M[ 9], 15);
  III(b0, b1, b2, b3, b4, M[ 1], 13);
  III(b4, b0, b1, b2, b3, M[ 2], 11);

  /* parallel round 3 */
  HHH(b3, b4, b0, b1, b2, M[15],  9);
  HHH(b2, b3, b4, b0, b1, M[ 5],  7);
  HHH(b1, b2, b3, b4, b0, M[ 1], 15);
  HHH(b0, b1, b2, b3, b4, M[ 3], 11);
  HHH(b4, b0, b1, b2, b3, M[ 7],  8);
  HHH(b3, b4, b0, b1, b2, M[14],  6);
  HHH(b2, b3, b4, b0, b1, M[ 6],  6);
  HHH(b1, b2, b3, b4, b0, M[ 9], 14);
  HHH(b0, b1, b2, b3, b4, M[11], 12);
  HHH(b4, b0, b1, b2, b3, M[ 8], 13);
  HHH(b3, b4, b0, b1, b2, M[12],  5);
  HHH(b2, b3, b4, b0, b1, M[ 2], 14);
  HHH(b1, b2, b3, b4, b0, M[10], 13);
  HHH(b0, b1, b2, b3, b4, M[ 0], 13);
  HHH(b4, b0, b1, b2, b3, M[ 4],  7);
  HHH(b3, b4, b0, b1, b2, M[13],  5);

  /* parallel round 4 */
  GGG(b2, b3, b4, b0, b1, M[ 8], 15);
  GGG(b1, b2, b3, b4, b0, M[ 6],  5);
  GGG(b0, b1, b2, b3, b4, M[ 4],  8);
  GGG(b4, b0, b1, b2, b3, M[ 1], 11);
  GGG(b3, b4, b0, b1, b2, M[ 3], 14);
  GGG(b2, b3, b4, b0, b1, M[11], 14);
  GGG(b1, b2, b3, b4, b0, M[15],  6);
  GGG(b0, b1, b2, b3, b4, M[ 0], 14);
  GGG(b4, b0, b1, b2, b3, M[ 5],  6);
  GGG(b3, b4, b0, b1, b2, M[12],  9);
  GGG(b2, b3, b4, b0, b1, M[ 2], 12);
  GGG(b1, b2, b3, b4, b0, M[13],  9);
  GGG(b0, b1, b2, b3, b4, M[ 9], 12);
  GGG(b4, b0, b1, b2, b3, M[ 7],  5);
  GGG(b3, b4, b0, b1, b2, M[10], 15);
  GGG(b2, b3, b4, b0, b1, M[14],  8);

  /* parallel round 5 */
  FFF(b1, b2, b3, b4, b0, M[12] ,  8);
  FFF(b0, b1, b2, b3, b4, M[15] ,  5);
  FFF(b4, b0, b1, b2, b3, M[10] , 12);
  FFF(b3, b4, b0, b1, b2, M[ 4] ,  9);
  FFF(b2, b3, b4, b0, b1, M[ 1] , 12);
  FFF(b1, b2, b3, b4, b0, M[ 5] ,  5);
  FFF(b0, b1, b2, b3, b4, M[ 8] , 14);
  FFF(b4, b0, b1, b2, b3, M[ 7] ,  6);
  FFF(b3, b4, b0, b1, b2, M[ 6] ,  8);
  FFF(b2, b3, b4, b0, b1, M[ 2] , 13);
  FFF(b1, b2, b3, b4, b0, M[13] ,  6);
  FFF(b0, b1, b2, b3, b4, M[14] ,  5);
  FFF(b4, b0, b1, b2, b3, M[ 0] , 15);
  FFF(b3, b4, b0, b1, b2, M[ 3] , 13);
  FFF(b2, b3, b4, b0, b1, M[ 9] , 11);
  FFF(b1, b2, b3, b4, b0, M[11] , 11);
```

```
  B[0] = b0 - B[0];
  B[1] = b1 - B[1];
  B[2] = b2 - B[2];
  B[3] = b3 - B[3];
  B[4] = b4 - B[4];

  return;
}

/*** end of file ttmac.c ***/


/*** file test.c ***/

/**  header files **/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ttmac.h"

/** read MAC key from file fname
    key should be given in hexadecimal format **/

void readkey(char *fname, byte **key, unsigned int *keysize)
{
  FILE         *file;
  unsigned int  i, temp;

  if ( (file = fopen(fname, "rb")) == NULL ) {
    fprintf(stderr, \
            "readkey: cannot open file \"%s\".\n", fname);
    exit(1);
  }

  fseek(file, 0L, SEEK_END);
  if ((*key = malloc((ftell(file)+1)/2)) == NULL) {
    fprintf(stderr, "readkey: allocation error");
    exit(1);
  }

  fclose(file);
  if ( (file = fopen(fname, "r")) == NULL ) {
    fprintf(stderr, \
            "readkey: cannot open file \"%s\".\n", fname);
    exit(1);
  }

  for (i=0;;i++) {
    if (fscanf(file, "%02x", &temp) == EOF)
      break;
    (*key)[i] = (byte)temp;
  }

  *keysize = 8*i;

  fclose(file);
}

/** return TTMAC(key, message) **/

void MACstring(byte *key, char *message)

{
  unsigned int        i;
  unsigned long       bitlength;
  struct MDstruct     state;
  byte                mac[MACsize/8];

  bitlength = (unsigned long) (8*strlen(message));

  MDkeysetup(key, &state);

  MDadd((byte *) message, bitlength, &state);

  MDfinalize(&state, mac);

  printf("* message: \"%s\"\n* MAC: ", message);
  for (i=0; i<(MACsize/8); i++)
    printf("%02x", mac[i]);
  printf("\n\n");
}
```

```
/** main test programme **/

main(int argc, char *argv[])
{
  byte                *key;
  unsigned int        i, keysize;

  if (argc == 1) {
    printf("For each command line argument in turn:\n");
    printf("  -sstring  -- print key, string and MAC\n");
    return(0);
   }

  readkey("Keyfile.hex", &key, &keysize);

  if (keysize > STATEsize)
    printf("\n%d-bit key shortened to 160 bits !", keysize);
  else if (keysize < STATEsize) {
    printf("\n%d-bit key is too short, \
           supply a key of 160 bits !\n", keysize);
    return(1);
  }

  printf("\n* key: ");
  for (i=0; i<(STATEsize/8); i++)
    printf("%02x", key[i]);
  printf("\n");

  for (i = 1; i < argc; i++) {
    if (argv[i][0] == '-' && argv[i][1] == 's')
      MACstring(key, argv[i] + 2);
  }

  return(0);

}

/*** end of file test.c ***/
```

Test Vectors

```
* key: 00112233445566778899aabbccddeeff01234567

TTMAC test suite results (ASCII):

* message: "" (empty string)
* MAC: 2dec8ed4a0fd712ed9fbf2ab466ec2df21215e4a

* message: "a"
* MAC: 5893e3e6e306704dd77ad6e6ed432cde321a7756

* message: "abc"
* MAC: 70bfd1029797a5c16da5b557a1f0b2779b78497e

* message: "message digest"
* MAC: 8289f4f19ffe4f2af737de4bd71c829d93a972fa

* message: "abcdefghijklmnopqrstuvwxyz"
* MAC: 2186ca09c5533198b7371f245273504ca92bae60

* message: "abcdbcdecdefdefgefghfghighij
            hijkijkljklmklmnlmnomnopnopq"
* MAC: 8a7bf77aef62a2578497a27c0d6518a429e7c14d

* message: "A...Za...z0...9"
* MAC: 54bac392a886806d169556fcbb6789b54fb364fb

* message: 8 times "1234567890"
* MAC: 0ced2c9f8f0d9d03981ab5c8184bac43dd54c484

* message: 1 million times "a"
* MAC: 27b3aedb5df8b629f0142194daa3846e1895f3d2
```

Appendix B

Description of MD4

In this appendix we give a description of MD4 and explain the notations which are used in our analysis in Chapter 4 (Sect. 4.4). Not all of the details are fully described: for a complete specification see [114]. MD4 is defined as the iteration of a compression function which we specify below. Each application of this compression function uses a four-word chaining variable and a sixteen-word message block as input, and produces four words of output (a new value for the chaining variable, to be used as input for the next application of the compression function). All words have a length of 32 bits (four bytes). The initial value for the chaining variable (to be used as input for the first application of the compression function) is specified as follows (hexadecimal notation):

$$IV = \texttt{67452301}_x\ \texttt{efcdab89}_x\ \texttt{98badcfe}_x\ \texttt{10325476}_x\ .$$

Note that there is a padding rule which appends bytes to the message so that its length becomes a multiple of 64 bytes (16 words × 4 bytes/word). The added bytes include a representation of the length of the original message. The little-endian convention is used to transform the message (sequence of bytes) into a sequence of words.

The compression function uses three Boolean functions, each of which takes three words of input and produces one word of output:

$$\begin{aligned}
f_1(Z_2, Z_1, Z_0) &= (Z_2 Z_1) \vee (\overline{Z_2} Z_0)\,,\\
f_2(Z_2, Z_1, Z_0) &= (Z_2 Z_1) \vee (Z_1 Z_0) \vee (Z_2 Z_0)\,,\\
f_3(Z_2, Z_1, Z_0) &= Z_2 \oplus Z_1 \oplus Z_0\,.
\end{aligned}$$

These functions are the selection, majority, and exor functions respectively, and operate at bit-level: they can be performed independently at each of the 32 bit positions in the words.

Let the notations $\mathcal{F}_1(Z_3, Z_2, Z_1, Z_0, W, s)$, $\mathcal{F}_2(Z_3, Z_2, Z_1, Z_0, W, s)$ and $\mathcal{F}_3(Z_3, Z_2, Z_1, Z_0, W, s)$ be equivalent to respectively:

$$\begin{aligned}(Z_3 + f_1(Z_2, Z_1, Z_0) + W + U_1)^{\lll s}\,,\\ (Z_3 + f_2(Z_2, Z_1, Z_0) + W + U_2)^{\lll s}\,,\\ (Z_3 + f_3(Z_2, Z_1, Z_0) + W + U_3)^{\lll s}\,.\end{aligned}$$

Here the '+' operation denotes addition modulo 2^{32}. The additive constants U_1, U_2 and U_3 have the following values (note that U_1 can be ignored, and U_2 and U_3 correspond to $\sqrt{2}$ and $\sqrt{3}$ respectively):

$$U_1 = \texttt{00000000}_x\,,\ U_2 = \texttt{5a827999}_x\,,\ U_3 = \texttt{6ed9eba1}_x\,.$$

Suppose now that the input chaining variable (A_0, B_0, C_0, D_0) is given, and a message block $\{W_j\}$ $(0 \le j < 16)$. Then the compression function applies the following 48 steps (three rounds of 16 steps each):

ROUND 1 STEP

$$A_1 = \mathcal{F}_1(A_0, B_0, C_0, D_0, W_0, 3) \tag{B.1}$$
$$D_1 = \mathcal{F}_1(D_0, A_1, B_0, C_0, W_1, 7) \tag{B.2}$$
$$C_1 = \mathcal{F}_1(C_0, D_1, A_1, B_0, W_2, 11) \tag{B.3}$$
$$B_1 = \mathcal{F}_1(B_0, C_1, D_1, A_1, W_3, 19) \tag{B.4}$$
$$A_2 = \mathcal{F}_1(A_1, B_1, C_1, D_1, W_4, 3) \tag{B.5}$$
$$D_2 = \mathcal{F}_1(D_1, A_2, B_1, C_1, W_5, 7) \tag{B.6}$$
$$C_2 = \mathcal{F}_1(C_1, D_2, A_2, B_1, W_6, 11) \tag{B.7}$$
$$B_2 = \mathcal{F}_1(B_1, C_2, D_2, A_2, W_7, 19) \tag{B.8}$$
$$A_3 = \mathcal{F}_1(A_2, B_2, C_2, D_2, W_8, 3) \tag{B.9}$$
$$D_3 = \mathcal{F}_1(D_2, A_3, B_2, C_2, W_9, 7) \tag{B.10}$$
$$C_3 = \mathcal{F}_1(C_2, D_3, A_3, B_2, W_{10}, 11) \tag{B.11}$$
$$B_3 = \mathcal{F}_1(B_2, C_3, D_3, A_3, W_{11}, 19) \tag{B.12}$$
$$A_4 = \mathcal{F}_1(A_3, B_3, C_3, D_3, W_{12}, 3) \tag{B.13}$$
$$D_4 = \mathcal{F}_1(D_3, A_4, B_3, C_3, W_{13}, 7) \tag{B.14}$$
$$C_4 = \mathcal{F}_1(C_3, D_4, A_4, B_3, W_{14}, 11) \tag{B.15}$$
$$B_4 = \mathcal{F}_1(B_3, C_4, D_4, A_4, W_{15}, 19) \tag{B.16}$$

ROUND 2 STEP

$$A_5 = \mathcal{F}_2(A_4, B_4, C_4, D_4, W_0, 3) \tag{B.17}$$

$$D_5 = \mathcal{F}_2(D_4, A_5, B_4, C_4, W_4, 5) \tag{B.18}$$
$$C_5 = \mathcal{F}_2(C_4, D_5, A_5, B_4, W_8, 9) \tag{B.19}$$
$$B_5 = \mathcal{F}_2(B_4, C_5, D_5, A_5, W_{12}, 13) \tag{B.20}$$
$$A_6 = \mathcal{F}_2(A_5, B_5, C_5, D_5, W_1, 3) \tag{B.21}$$
$$D_6 = \mathcal{F}_2(D_5, A_6, B_5, C_5, W_5, 5) \tag{B.22}$$
$$C_6 = \mathcal{F}_2(C_5, D_6, A_6, B_5, W_9, 9) \tag{B.23}$$
$$B_6 = \mathcal{F}_2(B_5, C_6, D_6, A_6, W_{13}, 13) \tag{B.24}$$
$$A_7 = \mathcal{F}_2(A_6, B_6, C_6, D_6, W_2, 3) \tag{B.25}$$
$$D_7 = \mathcal{F}_2(D_6, A_7, B_6, C_6, W_6, 5) \tag{B.26}$$
$$C_7 = \mathcal{F}_2(C_6, D_7, A_7, B_6, W_{10}, 9) \tag{B.27}$$
$$B_7 = \mathcal{F}_2(B_6, C_7, D_7, A_7, W_{14}, 13) \tag{B.28}$$
$$A_8 = \mathcal{F}_2(A_7, B_7, C_7, D_7, W_3, 3) \tag{B.29}$$
$$D_8 = \mathcal{F}_2(D_7, A_8, B_7, C_7, W_7, 5) \tag{B.30}$$
$$C_8 = \mathcal{F}_2(C_7, D_8, A_8, B_7, W_{11}, 9) \tag{B.31}$$
$$B_8 = \mathcal{F}_2(B_7, C_8, D_8, A_8, W_{15}, 13) \tag{B.32}$$

ROUND 3 STEP

$$A_9 = \mathcal{F}_3(A_8, B_8, C_8, D_8, W_0, 3) \tag{B.33}$$
$$D_9 = \mathcal{F}_3(D_8, A_9, B_8, C_8, W_8, 9) \tag{B.34}$$
$$C_9 = \mathcal{F}_3(C_8, D_9, A_9, B_8, W_4, 11) \tag{B.35}$$
$$B_9 = \mathcal{F}_3(B_8, C_9, D_9, A_9, W_{12}, 15) \tag{B.36}$$
$$A_{10} = \mathcal{F}_3(A_9, B_9, C_9, D_9, W_2, 3) \tag{B.37}$$
$$D_{10} = \mathcal{F}_3(D_9, A_{10}, B_9, C_9, W_{10}, 9) \tag{B.38}$$
$$C_{10} = \mathcal{F}_3(C_9, D_{10}, A_{10}, B_9, W_6, 11) \tag{B.39}$$
$$B_{10} = \mathcal{F}_3(B_9, C_{10}, D_{10}, A_{10}, W_{14}, 15) \tag{B.40}$$
$$A_{11} = \mathcal{F}_3(A_{10}, B_{10}, C_{10}, D_{10}, W_1, 3) \tag{B.41}$$
$$D_{11} = \mathcal{F}_3(D_{10}, A_{11}, B_{10}, C_{10}, W_9, 9) \tag{B.42}$$
$$C_{11} = \mathcal{F}_3(C_{10}, D_{11}, A_{11}, B_{10}, W_5, 11) \tag{B.43}$$
$$B_{11} = \mathcal{F}_3(B_{10}, C_{11}, D_{11}, A_{11}, W_{13}, 15) \tag{B.44}$$
$$A_{12} = \mathcal{F}_3(A_{11}, B_{11}, C_{11}, D_{11}, W_3, 3) \tag{B.45}$$
$$D_{12} = \mathcal{F}_3(D_{11}, A_{12}, B_{11}, C_{11}, W_{11}, 9) \tag{B.46}$$
$$C_{12} = \mathcal{F}_3(C_{11}, D_{12}, A_{12}, B_{11}, W_7, 11) \tag{B.47}$$
$$B_{12} = \mathcal{F}_3(B_{11}, C_{12}, D_{12}, A_{12}, W_{15}, 15) \tag{B.48}$$

Finally, the four-word output of the compression function is computed with a feed-forward of the input chaining variable:

$$A = A_0 + A_{12}\,,\; B = B_0 + B_{12}\,,\; C = C_0 + C_{12}\,,\; D = D_0 + D_{12}\,.$$

The obtained words (A, B, C, D) serve as input chaining variable for the next application of the compression function. If this was the final use of the compression function (the last sixteen words of the padded message have been processed), the concatenated 128-bit (sixteen-byte) value $A\|B\|C\|D$ serves as hash result of the message, where the little-endian convention is used to transform the sequence of words into a sequence of bytes (starting with the least significant byte of A and ending with the most significant byte of D).

Appendix C

Description of HAVAL

In this appendix we give a description of three-round HAVAL and explain the notations which are used in our analysis in Chapter 4 (Sect. 4.5). Not all of the details are fully described: for a complete specification see [131]. HAVAL is defined as the iteration of a compression function which we specify below. Each application of this compression function uses an eight-word chaining variable and a 32-word message block as input, and produces eight words of output (a new value for the chaining variable, to be used as input for the next application of the compression function). All words have a length of 32 bits (four bytes). The initial value for the chaining variable (to be used as input for the first application of the compression function) is specified as follows (hexadecimal notation):

$$\begin{aligned} IV \;=\; & \texttt{ec4e6c89}_x\ \texttt{082efa98}_x\ \texttt{299f31d0}_x\ \texttt{a4093822}_x \\ & \texttt{03707344}_x\ \texttt{13198a2e}_x\ \texttt{85a308d3}_x\ \texttt{243f6a88}_x\ . \end{aligned}$$

Note that there is a padding rule which appends bytes to the message so that its length becomes a multiple of 128 bytes (32 words × 4 bytes/word). The added bytes include a representation of the length of the original message. The little-endian convention is used to transform the message (sequence of bytes) into a sequence of words.

The compression function uses three Boolean functions, each of which takes seven words of input and produces one word of output:

$$\begin{aligned} f_1(Z_6,Z_5,Z_4,Z_3,Z_2,Z_1,Z_0) &= (Z_2Z_3)\oplus(Z_6Z_0)\oplus(Z_5Z_1)\oplus(Z_4Z_2)\oplus Z_4\,, \\ f_2(Z_6,Z_5,Z_4,Z_3,Z_2,Z_1,Z_0) &= (Z_3Z_5Z_0)\oplus(Z_5Z_1Z_2)\oplus(Z_3Z_5)\oplus(Z_3Z_1) \\ &\quad \oplus(Z_5Z_4)\oplus(Z_0Z_2)\oplus(Z_1Z_2)\oplus(Z_6Z_5)\oplus Z_6\,, \\ f_3(Z_6,Z_5,Z_4,Z_3,Z_2,Z_1,Z_0) &= (Z_5Z_4Z_3)\oplus(Z_5Z_2)\oplus(Z_4Z_1)\oplus(Z_3Z_6) \\ &\quad \oplus(Z_0Z_3)\oplus Z_0\,. \end{aligned}$$

Note that the functions f_1, f_2 and f_3 operate at bit-level: they can be performed independently at each of the 32 bit positions in the words.

Let the notations $\mathcal{F}_1(Z_7, Z_6, Z_5, Z_4, Z_3, Z_2, Z_1, Z_0, W)$, $\mathcal{F}_2(Z_7, Z_6, Z_5, Z_4, Z_3, Z_2, Z_1, Z_0, W)$ and $\mathcal{F}_3(Z_7, Z_6, Z_5, Z_4, Z_3, Z_2, Z_1, Z_0, W)$ be equivalent to respectively:

$$Z_7^{\gg 11} + (f_1(Z_6, Z_5, Z_4, Z_3, Z_2, Z_1, Z_0))^{\gg 7} + W\,,$$
$$Z_7^{\gg 11} + (f_2(Z_6, Z_5, Z_4, Z_3, Z_2, Z_1, Z_0))^{\gg 7} + W\,,$$
$$Z_7^{\gg 11} + (f_3(Z_6, Z_5, Z_4, Z_3, Z_2, Z_1, Z_0))^{\gg 7} + W\,.$$

Here the '+' operation denotes addition modulo 2^{32}.

Suppose that the input chaining variable $(A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0)$ is given, and a message block $\{W_j\}$ $(0 \leq j < 32)$. Then the compression function of 3-round HAVAL applies the following 96 steps (three rounds of 32 steps each):

ROUND 1 STEP

$$A_1 = \mathcal{F}_1(A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0, W_0) \tag{C.1}$$
$$B_1 = \mathcal{F}_1(B_0, C_0, D_0, E_0, F_0, G_0, H_0, A_1, W_1) \tag{C.2}$$
$$C_1 = \mathcal{F}_1(C_0, D_0, E_0, F_0, G_0, H_0, A_1, B_1, W_2) \tag{C.3}$$
$$D_1 = \mathcal{F}_1(D_0, E_0, F_0, G_0, H_0, A_1, B_1, C_1, W_3) \tag{C.4}$$
$$E_1 = \mathcal{F}_1(E_0, F_0, G_0, H_0, A_1, B_1, C_1, D_1, W_4) \tag{C.5}$$
$$F_1 = \mathcal{F}_1(F_0, G_0, H_0, A_1, B_1, C_1, D_1, E_1, W_5) \tag{C.6}$$
$$G_1 = \mathcal{F}_1(G_0, H_0, A_1, B_1, C_1, D_1, E_1, F_1, W_6) \tag{C.7}$$
$$H_1 = \mathcal{F}_1(H_0, A_1, B_1, C_1, D_1, E_1, F_1, G_1, W_7) \tag{C.8}$$
$$A_2 = \mathcal{F}_1(A_1, B_1, C_1, D_1, E_1, F_1, G_1, H_1, W_8) \tag{C.9}$$
$$B_2 = \mathcal{F}_1(B_1, C_1, D_1, E_1, F_1, G_1, H_1, A_2, W_9) \tag{C.10}$$
$$C_2 = \mathcal{F}_1(C_1, D_1, E_1, F_1, G_1, H_1, A_2, B_2, W_{10}) \tag{C.11}$$
$$D_2 = \mathcal{F}_1(D_1, E_1, F_1, G_1, H_1, A_2, B_2, C_2, W_{11}) \tag{C.12}$$
$$E_2 = \mathcal{F}_1(E_1, F_1, G_1, H_1, A_2, B_2, C_2, D_2, W_{12}) \tag{C.13}$$
$$F_2 = \mathcal{F}_1(F_1, G_1, H_1, A_2, B_2, C_2, D_2, E_2, W_{13}) \tag{C.14}$$
$$G_2 = \mathcal{F}_1(G_1, H_1, A_2, B_2, C_2, D_2, E_2, F_2, W_{14}) \tag{C.15}$$
$$H_2 = \mathcal{F}_1(H_1, A_2, B_2, C_2, D_2, E_2, F_2, G_2, W_{15}) \tag{C.16}$$
$$A_3 = \mathcal{F}_1(A_2, B_2, C_2, D_2, E_2, F_2, G_2, H_2, W_{16}) \tag{C.17}$$
$$B_3 = \mathcal{F}_1(B_2, C_2, D_2, E_2, F_2, G_2, H_2, A_3, W_{17}) \tag{C.18}$$
$$C_3 = \mathcal{F}_1(C_2, D_2, E_2, F_2, G_2, H_2, A_3, B_3, W_{18}) \tag{C.19}$$
$$D_3 = \mathcal{F}_1(D_2, E_2, F_2, G_2, H_2, A_3, B_3, C_3, W_{19}) \tag{C.20}$$

$$E_3 = \mathcal{F}_1(E_2, F_2, G_2, H_2, A_3, B_3, C_3, D_3, W_{20}) \quad \text{(C.21)}$$
$$F_3 = \mathcal{F}_1(F_2, G_2, H_2, A_3, B_3, C_3, D_3, E_3, W_{21}) \quad \text{(C.22)}$$
$$G_3 = \mathcal{F}_1(G_2, H_2, A_3, B_3, C_3, D_3, E_3, F_3, W_{22}) \quad \text{(C.23)}$$
$$H_3 = \mathcal{F}_1(H_2, A_3, B_3, C_3, D_3, E_3, F_3, G_3, W_{23}) \quad \text{(C.24)}$$
$$A_4 = \mathcal{F}_1(A_3, B_3, C_3, D_3, E_3, F_3, G_3, H_3, W_{24}) \quad \text{(C.25)}$$
$$B_4 = \mathcal{F}_1(B_3, C_3, D_3, E_3, F_3, G_3, H_3, A_4, W_{25}) \quad \text{(C.26)}$$
$$C_4 = \mathcal{F}_1(C_3, D_3, E_3, F_3, G_3, H_3, A_4, B_4, W_{26}) \quad \text{(C.27)}$$
$$D_4 = \mathcal{F}_1(D_3, E_3, F_3, G_3, H_3, A_4, B_4, C_4, W_{27}) \quad \text{(C.28)}$$
$$E_4 = \mathcal{F}_1(E_3, F_3, G_3, H_3, A_4, B_4, C_4, D_4, W_{28}) \quad \text{(C.29)}$$
$$F_4 = \mathcal{F}_1(F_3, G_3, H_3, A_4, B_4, C_4, D_4, E_4, W_{29}) \quad \text{(C.30)}$$
$$G_4 = \mathcal{F}_1(G_3, H_3, A_4, B_4, C_4, D_4, E_4, F_4, W_{30}) \quad \text{(C.31)}$$
$$H_4 = \mathcal{F}_1(H_3, A_4, B_4, C_4, D_4, E_4, F_4, G_4, W_{31}) \quad \text{(C.32)}$$

ROUND 2 STEP

$$A_5 = \mathcal{F}_2(A_4, B_4, C_4, D_4, E_4, F_4, G_4, H_4, W_5 + U_0) \quad \text{(C.33)}$$
$$B_5 = \mathcal{F}_2(B_4, C_4, D_4, E_4, F_4, G_4, H_4, A_5, W_{14} + U_1) \quad \text{(C.34)}$$
$$C_5 = \mathcal{F}_2(C_4, D_4, E_4, F_4, G_4, H_4, A_5, B_5, W_{26} + U_2) \quad \text{(C.35)}$$
$$D_5 = \mathcal{F}_2(D_4, E_4, F_4, G_4, H_4, A_5, B_5, C_5, W_{18} + U_3) \quad \text{(C.36)}$$
$$E_5 = \mathcal{F}_2(E_4, F_4, G_4, H_4, A_5, B_5, C_5, D_5, W_{11} + U_4) \quad \text{(C.37)}$$
$$F_5 = \mathcal{F}_2(F_4, G_4, H_4, A_5, B_5, C_5, D_5, E_5, W_{28} + U_5) \quad \text{(C.38)}$$
$$G_5 = \mathcal{F}_2(G_4, H_4, A_5, B_5, C_5, D_5, E_5, F_5, W_7 + U_6) \quad \text{(C.39)}$$
$$H_5 = \mathcal{F}_2(H_4, A_5, B_5, C_5, D_5, E_5, F_5, G_5, W_{16} + U_7) \quad \text{(C.40)}$$
$$A_6 = \mathcal{F}_2(A_5, B_5, C_5, D_5, E_5, F_5, G_5, H_5, W_0 + U_8) \quad \text{(C.41)}$$
$$B_6 = \mathcal{F}_2(B_5, C_5, D_5, E_5, F_5, G_5, H_5, A_6, W_{23} + U_9) \quad \text{(C.42)}$$
$$C_6 = \mathcal{F}_2(C_5, D_5, E_5, F_5, G_5, H_5, A_6, B_6, W_{20} + U_{10}) \quad \text{(C.43)}$$
$$D_6 = \mathcal{F}_2(D_5, E_5, F_5, G_5, H_5, A_6, B_6, C_6, W_{22} + U_{11}) \quad \text{(C.44)}$$
$$E_6 = \mathcal{F}_2(E_5, F_5, G_5, H_5, A_6, B_6, C_6, D_6, W_1 + U_{12}) \quad \text{(C.45)}$$
$$F_6 = \mathcal{F}_2(F_5, G_5, H_5, A_6, B_6, C_6, D_6, E_6, W_{10} + U_{13}) \quad \text{(C.46)}$$
$$G_6 = \mathcal{F}_2(G_5, H_5, A_6, B_6, C_6, D_6, E_6, F_6, W_4 + U_{14}) \quad \text{(C.47)}$$
$$H_6 = \mathcal{F}_2(H_5, A_6, B_6, C_6, D_6, E_6, F_6, G_6, W_8 + U_{15}) \quad \text{(C.48)}$$
$$A_7 = \mathcal{F}_2(A_6, B_6, C_6, D_6, E_6, F_6, G_6, H_6, W_{30} + U_{16}) \quad \text{(C.49)}$$
$$B_7 = \mathcal{F}_2(B_6, C_6, D_6, E_6, F_6, G_6, H_6, A_7, W_3 + U_{17}) \quad \text{(C.50)}$$
$$C_7 = \mathcal{F}_2(C_6, D_6, E_6, F_6, G_6, H_6, A_7, B_7, W_{21} + U_{18}) \quad \text{(C.51)}$$
$$D_7 = \mathcal{F}_2(D_6, E_6, F_6, G_6, H_6, A_7, B_7, C_7, W_9 + U_{19}) \quad \text{(C.52)}$$

$$E_7 = \mathcal{F}_2(E_6, F_6, G_6, H_6, A_7, B_7, C_7, D_7, W_{17} + U_{20}) \tag{C.53}$$
$$F_7 = \mathcal{F}_2(F_6, G_6, H_6, A_7, B_7, C_7, D_7, E_7, W_{24} + U_{21}) \tag{C.54}$$
$$G_7 = \mathcal{F}_2(G_6, H_6, A_7, B_7, C_7, D_7, E_7, F_7, W_{29} + U_{22}) \tag{C.55}$$
$$H_7 = \mathcal{F}_2(H_6, A_7, B_7, C_7, D_7, E_7, F_7, G_7, W_6 + U_{23}) \tag{C.56}$$
$$A_8 = \mathcal{F}_2(A_7, B_7, C_7, D_7, E_7, F_7, G_7, H_7, W_{19} + U_{24}) \tag{C.57}$$
$$B_8 = \mathcal{F}_2(B_7, C_7, D_7, E_7, F_7, G_7, H_7, A_8, W_{12} + U_{25}) \tag{C.58}$$
$$C_8 = \mathcal{F}_2(C_7, D_7, E_7, F_7, G_7, H_7, A_8, B_8, W_{15} + U_{26}) \tag{C.59}$$
$$D_8 = \mathcal{F}_2(D_7, E_7, F_7, G_7, H_7, A_8, B_8, C_8, W_{13} + U_{27}) \tag{C.60}$$
$$E_8 = \mathcal{F}_2(E_7, F_7, G_7, H_7, A_8, B_8, C_8, D_8, W_2 + U_{28}) \tag{C.61}$$
$$F_8 = \mathcal{F}_2(F_7, G_7, H_7, A_8, B_8, C_8, D_8, E_8, W_{25} + U_{29}) \tag{C.62}$$
$$G_8 = \mathcal{F}_2(G_7, H_7, A_8, B_8, C_8, D_8, E_8, F_8, W_{31} + U_{30}) \tag{C.63}$$
$$H_8 = \mathcal{F}_2(H_7, A_8, B_8, C_8, D_8, E_8, F_8, G_8, W_{27} + U_{31}) \tag{C.64}$$

ROUND 3 STEP

$$A_9 = \mathcal{F}_3(A_8, B_8, C_8, D_8, E_8, F_8, G_8, H_8, W_{19} + U_{32}) \tag{C.65}$$
$$B_9 = \mathcal{F}_3(B_8, C_8, D_8, E_8, F_8, G_8, H_8, A_9, W_9 + U_{33}) \tag{C.66}$$
$$C_9 = \mathcal{F}_3(C_8, D_8, E_8, F_8, G_8, H_8, A_9, B_9, W_4 + U_{34}) \tag{C.67}$$
$$D_9 = \mathcal{F}_3(D_8, E_8, F_8, G_8, H_8, A_9, B_9, C_9, W_{20} + U_{35}) \tag{C.68}$$
$$E_9 = \mathcal{F}_3(E_8, F_8, G_8, H_8, A_9, B_9, C_9, D_9, W_{28} + U_{36}) \tag{C.69}$$
$$F_9 = \mathcal{F}_3(F_8, G_8, H_8, A_9, B_9, C_9, D_9, E_9, W_{17} + U_{37}) \tag{C.70}$$
$$G_9 = \mathcal{F}_3(G_8, H_8, A_9, B_9, C_9, D_9, E_9, F_9, W_8 + U_{38}) \tag{C.71}$$
$$H_9 = \mathcal{F}_3(H_8, A_9, B_9, C_9, D_9, E_9, F_9, G_9, W_{22} + U_{39}) \tag{C.72}$$
$$A_{10} = \mathcal{F}_3(A_9, B_9, C_9, D_9, E_9, F_9, G_9, H_9, W_{29} + U_{40}) \tag{C.73}$$
$$B_{10} = \mathcal{F}_3(B_9, C_9, D_9, E_9, F_9, G_9, H_9, A_{10}, W_{14} + U_{41}) \tag{C.74}$$
$$C_{10} = \mathcal{F}_3(C_9, D_9, E_9, F_9, G_9, H_9, A_{10}, B_{10}, W_{25} + U_{42}) \tag{C.75}$$
$$D_{10} = \mathcal{F}_3(D_9, E_9, F_9, G_9, H_9, A_{10}, B_{10}, C_{10}, W_{12} + U_{43}) \tag{C.76}$$
$$E_{10} = \mathcal{F}_3(E_9, F_9, G_9, H_9, A_{10}, B_{10}, C_{10}, D_{10}, W_{24} + U_{44}) \tag{C.77}$$
$$F_{10} = \mathcal{F}_3(F_9, G_9, H_9, A_{10}, B_{10}, C_{10}, D_{10}, E_{10}, W_{30} + U_{45}) \tag{C.78}$$
$$G_{10} = \mathcal{F}_3(G_9, H_9, A_{10}, B_{10}, C_{10}, D_{10}, E_{10}, F_{10}, W_{16} + U_{46}) \tag{C.79}$$
$$H_{10} = \mathcal{F}_3(H_9, A_{10}, B_{10}, C_{10}, D_{10}, E_{10}, F_{10}, G_{10}, W_{26} + U_{47}) \tag{C.80}$$
$$A_{11} = \mathcal{F}_3(A_{10}, B_{10}, C_{10}, D_{10}, E_{10}, F_{10}, G_{10}, H_{10}, W_{31} + U_{48}) \tag{C.81}$$
$$B_{11} = \mathcal{F}_3(B_{10}, C_{10}, D_{10}, E_{10}, F_{10}, G_{10}, H_{10}, A_{11}, W_{15} + U_{49}) \tag{C.82}$$
$$C_{11} = \mathcal{F}_3(C_{10}, D_{10}, E_{10}, F_{10}, G_{10}, H_{10}, A_{11}, B_{11}, W_7 + U_{50}) \tag{C.83}$$
$$D_{11} = \mathcal{F}_3(D_{10}, E_{10}, F_{10}, G_{10}, H_{10}, A_{11}, B_{11}, C_{11}, W_3 + U_{51}) \tag{C.84}$$

$$\begin{aligned}
E_{11} &= \mathcal{F}_3(E_{10}, F_{10}, G_{10}, H_{10}, A_{11}, B_{11}, C_{11}, D_{11}, W_1 + U_{52}) && \text{(C.85)}\\
F_{11} &= \mathcal{F}_3(F_{10}, G_{10}, H_{10}, A_{11}, B_{11}, C_{11}, D_{11}, E_{11}, W_0 + U_{53}) && \text{(C.86)}\\
G_{11} &= \mathcal{F}_3(G_{10}, H_{10}, A_{11}, B_{11}, C_{11}, D_{11}, E_{11}, F_{11}, W_{18} + U_{54}) && \text{(C.87)}\\
H_{11} &= \mathcal{F}_3(H_{10}, A_{11}, B_{11}, C_{11}, D_{11}, E_{11}, F_{11}, G_{11}, W_{27} + U_{55}) && \text{(C.88)}\\
A_{12} &= \mathcal{F}_3(A_{11}, B_{11}, C_{11}, D_{11}, E_{11}, F_{11}, G_{11}, H_{11}, W_{13} + U_{56}) && \text{(C.89)}\\
B_{12} &= \mathcal{F}_3(B_{11}, C_{11}, D_{11}, E_{11}, F_{11}, G_{11}, H_{11}, A_{12}, W_6 + U_{57}) && \text{(C.90)}\\
C_{12} &= \mathcal{F}_3(C_{11}, D_{11}, E_{11}, F_{11}, G_{11}, H_{11}, A_{12}, B_{12}, W_{21} + U_{58}) && \text{(C.91)}\\
D_{12} &= \mathcal{F}_3(D_{11}, E_{11}, F_{11}, G_{11}, H_{11}, A_{12}, B_{12}, C_{12}, W_{10} + U_{59}) && \text{(C.92)}\\
E_{12} &= \mathcal{F}_3(E_{11}, F_{11}, G_{11}, H_{11}, A_{12}, B_{12}, C_{12}, D_{12}, W_{23} + U_{60}) && \text{(C.93)}\\
F_{12} &= \mathcal{F}_3(F_{11}, G_{11}, H_{11}, A_{12}, B_{12}, C_{12}, D_{12}, E_{12}, W_{11} + U_{61}) && \text{(C.94)}\\
G_{12} &= \mathcal{F}_3(G_{11}, H_{11}, A_{12}, B_{12}, C_{12}, D_{12}, E_{12}, F_{12}, W_5 + U_{62}) && \text{(C.95)}\\
H_{12} &= \mathcal{F}_3(H_{11}, A_{12}, B_{12}, C_{12}, D_{12}, E_{12}, F_{12}, G_{12}, W_2 + U_{63}) && \text{(C.96)}
\end{aligned}$$

The additive constants U_i which are used in the last two rounds are 32-bit values derived from the fractional part of π (see [131]). Finally, the eight-word output of the compression function is computed with a feed-forward of the input chaining variable:

$$\begin{array}{llll}
A = A_0 + A_{12} & B = B_0 + B_{12} & C = C_0 + C_{12} & D = D_0 + D_{12}\\
E = E_0 + E_{12} & F = F_0 + F_{12} & G = G_0 + G_{12} & H = H_0 + H_{12}\,.
\end{array}$$

The obtained words (A, B, C, D, E, F, G, H) serve as input chaining variable for the next application of the compression function. If this was the final use of the compression function (the last 32 words of the padded message have been processed), the concatenated 256-bit (32-byte) value $H\|G\|F\|E\|D\|C\|B\|A$ serves as hash result of the message, where the little-endian convention is used to transform the sequence of words into a sequence of bytes (starting with the least significant byte of H and ending with the most significant byte of A). The specification in [131] gives an optional output transformation which allows to reduce the length of this hash result to 128, 160, 192 or 224 bits.

Appendix D

Supporting Cryptanalytic Results for PANAMA

In this appendix we provide some supporting results for the cryptanalysis of PANAMA, given in Chapter 5 (Sect. 5.3).

Collisions in two steps are not possible

We show why there are no solutions for the difference pattern (5.12). In this case the 'timeline' (5.15) reduces to:

$$A \xrightarrow{\sigma 0} B \xrightarrow{\gamma} C \xrightarrow{\pi} D \xrightarrow{\theta} E \xrightarrow{\sigma 1} F \qquad \text{(D.1)}$$

Here $\sigma 0$ introduces the 'dX difference', and $\sigma 1$ the 'dY difference' which must cancel the difference in state E for every sub-collision.

We start by considering sub-collision 1. We know that for any value of $P0$ and dX,

$$dB_i = 0, \text{ for } i = 0, 9, 10, 11, \ldots, 16. \qquad \text{(D.2)}$$

Since γ does only mix nearby words it follows that

$$dC_i = 0, \text{ for } i = 9, 10, 11, \ldots 15. \qquad \text{(D.3)}$$

Applying the definition of π gives the following:

$$dD_i = 0, \text{ for } i = 2, 4, 7, 9, 11, 14, 16. \qquad \text{(D.4)}$$

With (5.5), this can be translated to the following conditions on dE_i:

$$\begin{aligned}
dE_3 + dE_4 + dE_7 + dE_{11} + dE_{12} + dE_{13} + dE_{14} + dE_{16} + dE_1 &= 0\\
dE_5 + dE_6 + dE_9 + dE_{13} + dE_{14} + dE_{15} + dE_{16} + dE_1 + dE_3 &= 0\\
dE_8 + dE_9 + dE_{12} + dE_{16} + dE_0 + dE_1 + dE_2 + dE_4 + dE_6 &= 0\\
dE_{10} + dE_{11} + dE_{14} + dE_1 + dE_2 + dE_3 + dE_4 + dE_6 + dE_8 &= 0\\
dE_{12} + dE_{13} + dE_{16} + dE_3 + dE_4 + dE_5 + dE_6 + dE_8 + dE_{10} &= 0\\
dE_{15} + dE_{16} + dE_2 + dE_6 + dE_7 + dE_8 + dE_9 + dE_{11} + dE_{13} &= 0\\
dE_0 + dE_1 + dE_4 + dE_8 + dE_9 + dE_{10} + dE_{11} + dE_{13} + dE_{15} &= 0
\end{aligned} \tag{D.5}$$

Fulfilling (5.12) also requires that $dF_i = 0, \forall i$. Without specifying $P1$ or dY_i, we have conditions on dE:

$$dE_i = 0, \text{ for } i = 0, 9, 10, 11, \ldots, 16. \tag{D.6}$$

Furthermore,

$$dE_i = dY_{i-1}, \text{ for } i = 1, 2, \ldots, 8. \tag{D.7}$$

Combining (D.5), (D.6) and (D.7), gives us the following conditions on dY:

$$\begin{aligned}
dY_2 + dY_3 + dY_6 + dY_0 &= 0\\
dY_4 + dY_5 + dY_0 + dY_2 &= 0\\
dY_7 + dY_0 + dY_1 + dY_3 + dY_5 &= 0\\
dY_0 + dY_1 + dY_2 + dY_3 + dY_5 + dY_7 &= 0\\
dY_2 + dY_3 + dY_4 + dY_5 + dY_7 &= 0\\
dY_1 + dY_5 + dY_6 + dY_7 &= 0\\
dY_0 + dY_3 + dY_7 &= 0
\end{aligned} \tag{D.8}$$

This can be transformed into:

$$dY_2 = 0, dY_6 = dY_7, dY_5 = dY_1, dY_3 + dY_6 + dY_0 = 0, dY_4 + dY_1 + dY_0 = 0. \tag{D.9}$$

We can do the same excercise for sub-collision 3, where the same differences are injected into the state, but now via the buffer tap. We get the following:

$$dB_i = 0, \quad \text{for} \quad i = 0, 1, 2, \ldots, 8; \tag{D.10}$$

$$dC_i = 0, \quad \text{for} \quad i = 0, 1, 2, \ldots, 6; \tag{D.11}$$

$$dD_i = 0, \quad \text{for} \quad i = 0, 3, 5, 8, 10, 13, 15. \tag{D.12}$$

$$\begin{aligned}
dE_1 + dE_2 + dE_5 + dE_9 + dE_{10} + dE_{11} + dE_{12} + dE_{14} + dE_{16} &= 0\\
dE_4 + dE_5 + dE_8 + dE_{12} + dE_{13} + dE_{14} + dE_{15} + dE_0 + dE_2 &= 0\\
dE_6 + dE_7 + dE_{10} + dE_{14} + dE_{15} + dE_{16} + dE_0 + dE_2 + dE_4 &= 0\\
dE_9 + dE_{10} + dE_{13} + dE_0 + dE_1 + dE_2 + dE_3 + dE_5 + dE_7 &= 0\\
dE_{11} + dE_{12} + dE_{15} + dE_2 + dE_3 + dE_4 + dE_5 + dE_7 + dE_9 &= 0\\
dE_{14} + dE_{15} + dE_1 + dE_5 + dE_6 + dE_7 + dE_8 + dE_{10} + dE_{12} &= 0\\
dE_{16} + dE_0 + dE_3 + dE_7 + dE_8 + dE_9 + dE_{10} + dE_{12} + dE_{14} &= 0
\end{aligned} \tag{D.13}$$

Now we have that $dE_i = 0$ for $i = 0, \ldots, 8$, and $dE_i = dY_{i-9}$ for $i = 9, \ldots, 16$.

$$\begin{aligned}
dY_0 + dY_1 + dY_2 + dY_3 + dY_5 + dY_7 &= 0\\
dY_3 + dY_4 + dY_5 + dY_6 &= 0\\
dY_1 + dY_5 + dY_6 + dY_7 &= 0\\
dY_0 + dY_1 + dY_4 &= 0\\
dY_2 + dY_3 + dY_6 + dY_0 &= 0\\
dY_5 + dY_6 + dY_1 + dY_3 &= 0\\
dY_7 + dY_0 + dY_1 + dY_3 + dY_5 &= 0
\end{aligned} \qquad \text{(D.14)}$$

Combining these results with the results from sub-collision 1 (D.9), we get:

$$dY_0 = dY_1 = dY_2 = dY_4 = dY_5 = 0, dY_3 = dY_6 = dY_7. \qquad \text{(D.15)}$$

Because sub-collisions 2 and 4 will result in similar conditions on $r(dY)$, the only valid solution is $dY = 0$. It is a bit surprising that combining the results of sub-collisions 1 and 3 alone does not suffice to reach this conclusion; we get seven equations for each sub-collision, but they are not independent, e.g., two of the seven equations from sub-collision 1 are redundant.

Difference propagation for sub-collisions 2,3,4

We give here Tables D.1, D.2 and D.3 with the required difference propagation for sub-collisions 2, 3 and 4, as well as the corresponding conditions on the absolute value of the state at 'time' B and F. Sub-collision 1 (and 5) was described in Sect. 5.3.3 (Table 5.4), where we also discussed the relations between Tables 5.4, D.1, D.2 and D.3.

Sub-collision 2

The difference propagation in sub-collision 2, as specified in Table D.1, leads to the following conditions for B:

$$B_3 = 1, B_4 + B_5 = 1, B_6 = 0, \qquad \text{(D.16)}$$

and the following conditions for F:

$$\begin{aligned}
&F_0 = 0, F_1 = 0, F_2 + F_3 = 1, F_3 = F_4, F_5 = 1, F_6 = 1, F_7 + F_8 = 1,\\
&F_9 = 1, F_{10} = 0, F_{12} = 0, F_{13} = 1, F_{14} = F_{15}, F_{15} + F_{16} = 1.
\end{aligned} \qquad \text{(D.17)}$$

Table D.1: The required difference propagation for sub-collision 2.

i	dB	dC	dD	$dE = dF$	dG	dH	dI
0	0	0	0	0	1	1	0
1	0	0	0	0	1	0	1
2	0	0	0	1	1	0	0
3	0	1	1	1	0	1	1
4	1	1	0	1	1	1	0
5	1	1	0	0	0	1	0
6	0	0	0	0	1	1	1
7	0	0	0	1	0	1	0
8	0	0	1	1	1	0	1
9	0	0	0	0	1	0	0
10	0	0	0	0	0	1	0
11	0	0	0	1	1	1	0
12	0	0	0	0	0	1	0
13	0	0	0	0	0	1	0
14	0	0	0	1	0	0	0
15	0	0	1	1	1	0	0
16	0	0	0	1	1	0	0

Sub-collision 3

The difference propagation in sub-collision 3, as specified in Table D.2, leads to the following conditions for B:

$$B_{13} = 0, B_{14} + B_{15} = 1, B_{16} = 0, \tag{D.18}$$

and the following conditions for F:

$$F_0 = 1, F_1 + F_2 = 1, F_2 + F_3 = 1, F_4 = 0, F_5 + F_6 = 1, F_6 + F_7 = 1, F_7 = F_8,$$
$$F_8 = F_9, F_9 + F_{10} = 1, F_{11} = 0, F_{12} = 1, F_{13} = F_{14} = F_{15}, F_{16} = 0. \tag{D.19}$$

Sub-collision 4

The difference propagation in sub-collision 4, as specified in Table D.3, leads to the following conditions for B:

$$B_{11} = 0, B_{12} = B_{13}, B_{14} = 0, \tag{D.20}$$

Table D.2: The required difference propagation for sub-collision 3.

i	dB	dC	dD	$dE = dF$	dG	dH	dI
0	0	0	0	0	1	1	0
1	0	0	0	1	0	1	0
2	0	0	1	1	1	1	0
3	0	0	0	1	0	1	0
4	0	0	0	0	1	0	0
5	0	0	0	1	0	0	0
6	0	0	0	1	1	0	0
7	0	0	1	1	1	1	0
8	0	0	0	1	0	0	0
9	0	0	1	1	1	0	0
10	0	0	0	1	1	1	1
11	0	0	0	0	0	1	1
12	0	1	0	0	0	0	0
13	0	1	0	1	1	1	1
14	1	1	1	1	1	1	0
15	1	1	0	1	1	0	0
16	0	0	0	0	0	1	1

and the following conditions for F:

$$\begin{aligned}&F_0 = 1, F_4 = 0, F_6 = 1, F_7 = 0, F_8 + F_9 = 1, F_8 = F_{10},\\ &F_{11} = 0, F_{12} + F_{13} = 1, F_{12} = F_{14}, F_{12} = F_{15}, F_{12} = F_{16}.\end{aligned} \tag{D.21}$$

Table D.3: The required difference propagation for sub-collision 4.

i	dB	dC	dD	$dE = dF$	dG	dH	dI
0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0
2	0	0	0	0	0	1	0
3	0	0	0	0	1	1	0
4	0	0	0	0	1	1	0
5	0	0	0	1	1	0	0
6	0	0	0	0	1	0	0
7	0	0	0	0	1	0	0
8	0	0	0	1	0	1	0
9	0	0	1	1	1	0	1
10	0	1	0	1	0	0	0
11	0	0	0	0	1	1	1
12	1	1	0	1	0	1	0
13	1	1	0	1	1	1	0
14	0	0	1	1	1	1	1
15	0	0	0	1	0	1	0
16	0	0	1	1	1	0	1

Nederlandse Samenvatting

Analyse en Ontwerp van Cryptografische Hashfuncties, MAC-algoritmen en Blokcijfers

1. Inleiding

In onze moderne maatschappij vormt informatie een belangrijk handelsgoed. Het is vaak nodig om de *confidentialiteit* ervan te beschermen, wat betekent dat het voor niet-geauthoriseerde personen onmogelijk moet zijn om de inhoud van de informatie te achterhalen. Langs de andere kant kan het even belangrijk zijn om de *authenticiteit* van gegevens te beschermen. Dit wil zeggen dat het mogelijk moet zijn om de oorsprong (auteur) ervan na te gaan, en om te controleren dat de gegevens door niemand anders gewijzigd zijn. Er wordt tegenwoordig steeds meer overgegaan van papieren naar elektronische documenten, en dit kan het risico op schending van de confidentialiteit of authenticiteit van gegevens vergroten: het is in vele gevallen heel gemakkelijk om informatie in elektronische vorm te doorzoeken, kopiëren of zelfs wijzigen. Dit is bijvoorbeeld het geval bij informatie die zonder bijkomende beveiliging op het Internet geplaatst wordt. Hetzelfde geldt voor de bescherming van de communicatie tussen personen: elektronische berichten die via telecommunicatielijnen worden uitgewisseld, kunnen onderweg afgeluisterd en in sommige gevallen zelfs gewijzigd worden.

Cryptografische technieken worden al vele eeuwen gebruikt om militaire en diplomatieke geheimen te beschermen. Historisch gezien heeft de nadruk echter steeds gelegen op de bescherming van confidentialiteit: een *encryptieschema* zet een bericht om in een cryptogram, door middel van een transformatie (vercijfering) die afhankelijk is van een geheime sleutel. Het bekomen cryptogram is onleesbaar voor wie deze geheime sleutel niet kent. Een geauthoriseerde persoon met kennis van de sleutel daarentegen, kan na ontvangst van het cryptogram de inverse transformatie (ontcijfering) toepassen en het bericht weer verkrijgen. Er werd lang van uitgegaan dat encryptie volstaat om ook de oorsprong van informatie te controleren: als het ontcijferen van een cryptogram resulteert in een zinvol bericht, dan moet het cryptogram wel opgesteld zijn door iemand met kennis van de sleutel. Dat men hier echter niet zomaar van kan uitgaan blijkt duidelijk uit het voorbeeld van het Vernam schema [129]. Dit encryptieschema garandeert perfecte geheimhouding, maar wanneer een tegenstrever een binair symbool (bit) van het cryptogram verandert, verandert ook de overeenkomende bit van het bericht dat bekomen wordt na ontcijfering.

In de jaren zeventig hebben de publicatie van de Amerikaanse encryptiestandaard, de DES [54], en de uitvinding van publieke sleutel cryptografie [38] ervoor gezorgd dat cryptologie (de studie van cryptografische technieken) is uitgegroeid tot een wetenschappelijk onderzoeksdomein. Dit domein kan onderverdeeld worden in twee subdomeinen: *cryptografie* en *cryptanalyse*. Een cryptograaf ontwikkelt nieuwe schema's (algoritmen), bijvoorbeeld om niet-geauthoriseerde toegang tot gegevens te voorkomen, of om te beletten dat documenten vervalst kunnen worden. Een cryptanalyst aan de andere kant probeert methoden te ontwikkelen om deze algoritmen te breken en zodoende de doelstellingen van de cryptograaf te omzeilen. Er bestaat een sterke band tussen de twee subdomeinen: een nieuw algoritme kan alleen veilig verondersteld worden na voldoende bestudeerd te zijn door ervaren cryptanalysten die geen zwakheden kunnen vinden.

De ontwikkeling van de cryptologie heeft meegebracht dat er meer aandacht besteed wordt aan andere concepten dan louter encryptie. De bescherming van de authenticiteit van informatie is hier een voorbeeld van, en er zijn ook nieuwe toepassingen zoals digitale handtekeningen en elektronisch geld. In dit proefschrift richten we ons voornamelijk op de studie van *cryptografische hashfuncties*. Dit zijn functies die berichten van willekeurige lengte afbeelden op een hashresultaat met een bepaalde lengte, al dan niet onder de invloed van een geheime sleutel. Hun voornaamste gebruik is voor de authentisering van gegevens. Hoofdstuk 2 van het proefschrift geeft een overzicht van de basisconcepten, met definities voor hashfuncties en de cryptografische eigenschappen waar ze aan moeten voldoen. Er wordt ook uitgelegd hoe deze functies gebruikt worden om de authenticiteit van gegevens te beschermen, en enkele andere toepassingen worden beschreven. Er wordt dieper ingegaan op het gebruik van hashfuncties in systemen voor digitale tijdsstempels. Hoofdstuk 3 beschrijft verschillende methoden voor het ontwerp van sleutel-onafhankelijke hashfuncties. Hoofdstuk 4 is gewijd aan de studie van een belangrijke klasse van hashfuncties, namelijk de ontwerpen die afgeleid zijn van het bekende MD4 algoritme. De verschillende algoritmen worden beschreven, een overzicht wordt gegeven van de aanvallen die gepubliceerd zijn in de literatuur, en nieuwe aanvallen worden ontwikkeld, in het bijzonder de eerste gekende aanval op het HAVAL algoritme. Hoofdstuk 5 bestudeert PANAMA, een cryptografische module die zowel voor encryptie als voor hashen kan dienen. Een aanval wordt voorgesteld op de PANAMA module gebruikt als hashfunctie. Hoofdstuk 6 richt zich op hashfuncties waarvoor de berekening afhankelijk is van een geheime sleutel. Dergelijke hashfuncties worden ook MAC-algoritmen genoemd ('message authentication codes'). Verschillende methoden voor het ontwerp van MAC-algoritmen worden beschreven, en een nieuw eigen ontwerp wordt voorgesteld: het algoritme Two-Track-MAC. Hoofdstuk 7 beschouwt blokcijfers, een type van cryptografische algoritmen die voor encryptie gebruikt worden. Een sleutelafhankelijke variant op de techniek van differentiële cryptanalyse wordt ontwikkeld, toegepast op het blokcijfer ICE. Tevens wordt de relatie tussen blokcijfers en hashfuncties besproken. Hoofdstuk 8 sluit het proefschrift af, en formuleert nieuwe uitdagingen voor toekomstig onderzoek.

2. Basisconcepten

Cryptografische hashfuncties zijn functies die berichten van willekeurige lengte afbeelden op een hashresultaat, bestaande uit een vast aantal bits. Om bruikbaar te zijn voor cryptografische doeleinden, moet een hashfunctie echter aan een aantal bijkomende voorwaarden voldoen. De volgende informele definities voor twee types van hashfuncties werden gegeven door B. Preneel [96]. Een **éénwegs-hashfunctie** is een functie h die aan de volgende voorwaarden voldoet:

1. De ingang X mag een willekeurige lengte hebben en het resultaat $h(X)$ heeft een vaste lengte van n bits.
2. Als h en een ingang X gegeven zijn, dan moet de berekening van $h(X)$ 'gemakkelijk' zijn.
3. De functie moet aan de éénwegseigenschap voldoen: als een waarde Y gegeven is, die in het beeld van de functie h ligt, moet het 'moeilijk' zijn om een bericht X te vinden waarvoor geldt dat $h(X) = Y$ (*invers beeld*); als een waarde X gegeven is (en bijgevolg $h(X)$ gekend is) moet het 'moeilijk' zijn om een bericht $X' \neq X$ te vinden waarvoor geldt dat $h(X') = h(X)$ (*tweede invers beeld*).

Een **botsing-bestendige hashfunctie** is een éénwegs-hashfunctie h die voldoet aan de volgende bijkomende voorwaarde:

4. De functie moet botsing-bestendig zijn: het moet 'moeilijk' zijn om twee verschillende berichten X en X' te vinden die door h op hetzelfde hashresultaat $h(X) = h(X')$ worden afgebeeld.

MAC-algoritmen ('message authentication codes') zijn een speciaal type van hashfuncties waarvoor de berekening afhankelijk is van een geheime sleutel. Het moet voor een tegenstrever onhaalbaar zijn om het MAC-resultaat van een bericht te 'vervalsen'. Dit betekent dat hij het juiste MAC-resultaat niet kan berekenen zonder kennis van de sleutel. De volgende informele definitie werd gegeven door Preneel [96]. Een **MAC-algoritme** is een functie h die voldoet aan de volgende voorwaarden:

1. De ingang X mag een willekeurige lengte hebben en het resultaat $h(K,X)$ heeft een vaste lengte van n bits. De functie heeft als bijkomende ingang de sleutel K, met een vaste lengte van k bits.
2. Als h, K en een ingang X gegeven zijn, moet de berekening van $h(K,X)$ 'gemakkelijk' zijn.
3. Als een bericht X gegeven is (maar met onbekende K), moet het 'moeilijk' zijn om $h(K,X)$ te bepalen. Zelfs als een grote verzameling met paren $\{X_i, h(K,X_i)\}$ gekend is, moet het 'moeilijk' zijn om de sleutel K te bepalen of om $h(K,X')$ te berekenen voor eender welk nieuw bericht $X' \neq X_i$ $(\forall i)$.

Men kan een (sleutelonafhankelijke) hashfunctie of een MAC-algoritme gebruiken om berichten te authentiseren. De kern van het idee is dat het gemakkelijker is om de authenticiteit van een kort hash- of MAC-resultaat te beschermen dan de authenticiteit van het bericht zelf. In het geval van een sleutelonafhankelijke hashfunctie wordt het hashresultaat naar de bestemmeling verzonden via een veilig kanaal waarop de authenticiteit gewaarborgd is (bv. een telefoonlijn zodat de bestemmeling de stem van de zender kan herkennen). De bestemmeling kan vervolgens de authenticiteit van het bericht, dat via een onveilig kanaal gecommuniceerd is, controleren door er zelf de hashfunctie op toe te passen. In het geval van een MAC-algoritme kan het MAC-resultaat samen met het bericht verzonden worden over het onveilige kanaal. Voor het communiceren van de gebruikte geheime sleutel heeft men dan wel een veilig kanaal nodig dat zowel authenticiteit als geheimhouding waarborgt.

Er zijn ook heel wat andere toepassingen voor hashfuncties. Eén van de belangrijkste is het gebruik van een sleutelonafhankelijke hashfunctie in combinatie met een digitaal handtekeningsschema. Zo een schema is gebaseerd op publieke sleutel cryptografie, en de performantie kan beduidend verbeterd worden wanneer men eerst de hashfunctie toepast op het bericht en vervolgens de handtekening berekent over het korte hashresultaat (in plaats van over het bericht zelf). Voor deze toepassing is het belangrijk dat de gebruikte hashfunctie niet alleen aan de éénwegseigenschap voldoet, maar ook botsing-bestendig is. Anders zouden gebruikers in staat kunnen zijn om te ontkennen dat ze bepaalde handtekeningen geplaatst hebben. Er wordt in dit hoofdstuk ook een beschrijving gegeven van systemen voor digitale tijdsstempels, en van de rol die hashfuncties daarin vervullen. Door digitale tijdsstempels kan men bewijzen dat bepaalde gegevens bestonden vóór een zeker tijdstip, wat onder meer nuttig is voor de bescherming van intellectuele eigendomsrechten, voor veilige audits en voor het verlengen van de levensduur van digitale handtekeningen. De beschrijving is gebaseerd op een studie die gemaakt werd voor het Belgische TIMESEC-project [134], en is gepubliceerd in [127].

3. Ontwerp van Cryptografische Hashfuncties

Een eerste vraag die beantwoord moet worden bij het ontwerp van een (sleutelonafhankelijke) hashfunctie, is welke lengte (in bits) men kiest voor de hashresultaten. Voor een lengte van n bits zijn er 2^n verschillende elementen in het beeld van de functie en dit aantal bepaalt de moeilijkheidsgraad van enkele algemeen toepasbare aanvallen. Het is voor een tegenstrever altijd mogelijk om de éénwegseigenschap van een hashfunctie te breken in ongeveer 2^n stappen, en om botsende berichten te vinden in ongeveer $2^{n/2}$ stappen (elke stap vereist de berekening van een hashresultaat). De moeilijkheidsgraad van $2^{n/2}$ stappen voor het vinden van botsingen kan verklaard worden door de zogenaamde *verjaardagsparadox.* Met de beschikbare rekenkracht van huidige computers wordt aangenomen dat een lengte van $n = 80$ bits nodig is voor een éénwegs-hashfunctie. Als de hashfunctie ook botsing-bestendig moet zijn is een lengte van $n = 160$ bits nodig. Omwille van de wet van Moore (de rekenkracht die beschikbaar is voor een bepaalde kostprijs verdubbelt elke achttien maanden) zijn grotere bitlengtes nodig voor veiligheid op lange termijn.

Zoals eerder uitgelegd, kunnen de berichten die door een hashfunctie verwerkt worden, een willekeurige lengte hebben. Alle bekende hashfuncties zijn echter gebaseerd op het itereren van een *compressiefunctie* die een ingang heeft van vaste lengte. Het bericht dat gehasht wordt, wordt dan eerst verdeeld in blokken van die lengte, en elk blok wordt op gelijkaardige wijze verwerkt door de compressiefunctie. De berekening maakt ook gebruik van een *kettingvariabele.* Deze variabele wordt in het begin van de berekening gelijk gesteld aan een initiële waarde (deze waarde maakt deel uit van de specificatie van de hashfunctie). Elke iteratie van de compressiefunctie berekent een nieuwe waarde voor de kettingvariabele, en het hashresultaat wordt gelijkgesteld aan de laatst bekomen waarde voor de kettingvariabele (dit is na verwerking van het laatste blok van het bericht). In sommige gevallen wordt nog een uitgangstransformatie toegepast. Het bericht van willekeurige lengte dat gehasht wordt, moet verdeeld kunnen worden in blokken van vaste lengte. Het is dan ook nodig om een schema te definiëren dat een variabel aantal bits toevoegt aan het bericht, en op deze wijze ervoor zorgt dat de lengte van het gewijzigde bericht steeds een veelvoud is van de lengte van de blokken. Indien dit schema ook bits toevoegt die een representatie vormen van de oorspronkelijke lengte van het bericht, kan men bewijzen dat de hashfunctie veilig is als de gebruikte compressiefunctie veilig is (Merkle-Damgård [86, 32] en

Lai-Massey [79]). Dit betekent dat een cryptograaf zich kan concentreren op het ontwerpen van een veilige compressiefunctie.

Verschillende methoden zijn voorgesteld voor het ontwerp van compressiefuncties. Men kan de compressiefunctie baseren op een bestaand cryptografisch schema zoals een blokcijfer (dit is een type van algoritme dat gebruikt wordt voor encryptie). Het voordeel hiervan is dat bestaande software- of hardware-implementaties herbruikt kunnen worden. Een ander argument is dat sommige blokcijfers, zoals het DES- [54] en het AES-algoritme [52], grondig geëvalueerd zijn, en dat er daardoor vertrouwd wordt op hun veiligheid. Er zijn verschillende constructies om een hashfunctie te verkrijgen uit een blokcijfer. Er kan een onderscheid gemaakt worden tussen constructies waarvoor de lengte van het hashresultaat gelijk is aan de bloklengte van het gebruikte blokcijfer (bv. Matyas-Meyer-Oseas en Davies-Meyer [83]), en constructies waarvoor de lengte van het hashresultaat dubbel zo groot is (bv. MDC-2 en MDC-4 [25, 87]). Een ander type van hashfuncties zijn deze waarvan de compressiefunctie gebaseerd is op modulair rekenen (zoals deze gebruikt in systemen van publieke sleutel cryptografie). Het veiligheidsniveau van dergelijke hashfuncties kan gemakkelijk aangepast worden door de keuze van de gebruikte modulus; het nadeel is dat deze constructies veel trager zijn dan andere types van hashfuncties. Een voorbeeld is de hashfunctie MASH-1 [64], waarvan de compressiefunctie gebaseerd is op een modulaire kwadrateringsoperatie. Er zijn ook algoritmen met een compressiefunctie die expliciet ontworpen is om te hashen. De populairste algoritmen van dit type zijn degene die gebaseerd zijn op het bekende MD4-algoritme [114], en die ontworpen zijn met het oog op een goede software-performantie. Tot slot van dit hoofdstuk wordt een overzicht gegeven van de standardisatieprocedures voor hashfuncties. De belangrijkste organisaties die hashfuncties standardiseren zijn ISO/IEC, ISO, ANSI en NIST.

4. Hashfuncties van de MDx-klasse

De MDx-klasse bestaat uit hashfuncties waarvan het ontwerp geïnspireerd is door het bekende MD4-algoritme. MD4 werd in 1990 door R. Rivest voorgesteld en is gericht op software-implementatie voor 32-bit architecturen. Ondertussen is aangetoond dat MD4 geen veilige hashfunctie is, maar een aantal andere algoritmen zijn voorgesteld waarvan het ontwerp gebaseerd is op gelijkaardige ideeën. Hiertoe behoren MD5 [115] (een ander ontwerp van Rivest), HAVAL [131] (voorgesteld door Australische onderzoekers), de SHA-algoritmen [51] (FIPS-standaarden van de Amerikaanse overheid), en de RIPEMD-algoritmen [44] (oorspronkelijk ontwikkeld in het kader van het Europese RIPE project). Deze hashfuncties zijn de populairste voor praktisch gebruik, omwille van hun software-performantie (op 32-bit of 64-bit platformen naargelang het algoritme) en omwille van het vertrouwen dat gegroeid is door de gekende cryptanalytische resultaten.

Zoals eerder uitgelegd berekent de compressiefunctie van een hashfunctie een nieuwe waarde voor de kettingvariabele, gebruik makend van het huidige ingangsblok. Typisch voor hashfuncties van de MDx-klasse is dat de compressiefunctie een sequentiële structuur heeft, bestaande uit een groot aantal eenvoudige stappen die elk een deel van de kettingvariabele aanpassen. Alle bewerkingen gebeuren met woorden van een vaste lengte (32 of 64 bits naargelang het algoritme), en zowel de kettingvariabele als het ingangsblok worden opgedeeld in zulke woorden. Voor de meeste algoritmen kan de compressiefunctie ook onderverdeeld worden in een aantal ronden. Elk woord van het ingangsblok wordt dan éénmaal gebruikt in elke ronde van de compressiefunctie.

In dit hoofdstuk wordt een beschrijving gegeven van de verschillende hashfuncties van de MDx-klasse, en van de aanvallen die er in de literatuur voor gepubliceerd zijn. De cryptanalytische methoden van H. Dobbertin [41, 42, 39] blijken meer algemeen toepasbaar te zijn dan vroegere aanvallen. Ze zijn gebaseerd op een combinatie van verschillende technieken: differentiële cryptanalyse waarbij de verschillen zo klein mogelijk gehouden worden, en methoden voor het oplossen van stelsels van niet-lineaire vergelijkingen. Men zoekt eerst een botsing voor de compressiefunctie, voor twee berichten die elk uit één blok bestaan. Vervolgens probeert men dit uit te breiden naar een botsing voor de hashfunctie zelf. Het probleem van het vinden van een botsing voor de compressiefunctie wordt onderverdeeld in verschillende kleinere problemen, zoals het vinden van een interne botsing of een interne bijna-botsing. Een belangrijke bijdrage van ons onderzoek is de aanval die we ontwikkeld hebben op het HAVAL-algoritme. Het ontwerp van de compressiefunctie van HAVAL is voornamelijk gebaseerd op het gebruik van complexe niet-lineaire Booleaanse functies. We hebben echter aangetoond dat de versie van HAVAL met drie ronden in de compressiefunctie gebroken kan worden met een aanval die een strategie volgt die vergelijkbaar is met de strategie van de aanval van Dobbertin op MD4 [42]. Deze aanval vergt ongeveer 2^{29} berekeningen van de compressiefunctie. Het is de eerste gepubliceerde aanval op een volledige versie van HAVAL. Het resultaat, dat gepubliceerd werd in [123], toont aan dat het voor een veilig ontwerp niet volstaat om sterke bouwblokken te gebruiken (bv. de niet-lineaire Booleaanse functies van HAVAL), maar dat men ook het effect van deze bouwblokken op de globale veiligheid moet nagaan.

Aan te bevelen hashfuncties van de MDx-klasse lijken deze van de RIPEMD- en SHA-families te zijn. Het MD4-algoritme is duidelijk gebroken op het gebied van botsing-bestendigheid, en voor MD5 zijn belangrijke zwakheden aangetoond (botsingen kunnen gevonden worden voor de compressiefunctie, maar tot nu toe kunnen ze niet uitgebreid worden naar de hashfunctie zelf). Bovendien is de lengte van het hashresultaat voor deze algoritmen slechts 128 bits, wat sowieso te weinig is voor een echte botsing-bestendige hashfunctie. HAVAL biedt het voordeel van een variabele lengte (tot 256 bits) voor de hashresultaten, en heeft verschillende veiligheidsniveaus op basis van het aantal gebruikte ronden in de compressiefunctie (drie tot vijf ronden zijn mogelijk). Ons werk toont echter aan dat botsingen gemakkelijk gevonden kunnen worden voor de versie met drie ronden. De RIPEMD-algoritmen hebben een compressiefunctie die bestaat uit twee parallelle lijnen (elk bestaande uit een aantal ronden), en voor de SHA-algoritmen wordt een speciale procedure gebruikt voor expansie van het ingangsblok. De laatste twee benaderingen voor het ontwerp lijken de veiligheid gevoelig te verbeteren. Als een lengte van 160 bits voor het hashresultaat voldoende is, kan RIPEMD-160 of SHA-1 gebruikt worden. Voor langere hashresultaten komen de meest recente SHA-algoritmen (SHA-224, SHA-256, SHA-384 en SHA-512) in aanmerking. Er moet echter opgemerkt worden dat deze hashfuncties tot nu toe nog maar een beperkte publieke evaluatie gekregen hebben.

5. De PANAMA Cryptografische Module

De ervaring die is opgedaan met de cryptanalyse van hashfuncties van de MDx-klasse, heeft geleid tot het ontwerp van een aantal algoritmen (in het bijzonder deze van de RIPEMD- en SHA-families) die een hoge veiligheidsmarge lijken te hebben tegen alle gekende aanvalsstrategieën. Deze algoritmen zijn echter beduidend minder performant dan MD4 en MD5. Er bestaat dan ook zeker interesse voor nieuwe ideeën voor het ontwerp van hashfuncties. Een voorbeeld van een nieuw type van ontwerp is de PANAMA cryptografische module [30], die gebruikt kan worden voor zowel hashen als voor stroom-encryptie. Opmerkelijk aan het ontwerp van PANAMA is de parallelle (in plaats van sequentiële) structuur van de iteratiefunctie, en het grote interne geheugen dat vereist is (de kettingvariabele bestaat uit een staat en een buffer, met een gecombineerde lengte van $544 + 8192 = 8736$ bits). Er wordt een uitgangstransformatie gebruikt en de lengte van het hashresultaat is 256 bits. De veiligheid van PANAMA is niet gebaseerd op het itereren van een botsing-bestendige compressiefunctie, maar er wordt verwacht dat de diffusie en niet-lineariteit gerealiseerd door opeenvolgende toepassingen van de iteratiefunctie cryptanalyse onmogelijk maken. Het inherente parallellisme van PANAMA laat heel performante software-implementaties op VLIW-processoren toe (tenminste in het geval dat er lange berichten gehasht worden).

De analyse die we in dit hoofdstuk maken van PANAMA toont echter aan dat er een theoretische aanval bestaat die veel efficiënter is voor het vinden van botsende berichten dan een algemeen toepasbare aanval gebaseerd op de verjaardagsparadox: 2^{82} operaties volstaan voor het vinden van botsingen (voor ideale veiligheid zouden 2^{128} operaties vereist moeten zijn). De strategie van deze aanval is sterk verschillend van de strategie van de aanvallen op hashfuncties van de MDx-klasse. We zoeken botsingen in zowel de staat als de buffer en hiervoor werken we met verschillen tussen twee berichten bestaande uit een groot aantal blokken. Het zoeken naar een botsing wordt onderverdeeld in vijf kleinere problemen die onafhankelijk van elkaar kunnen opgelost worden: elke keer dat er een verschil ontstaat in de staat laten we dit verschil zo snel mogelijk uitdoven. Hiervoor moeten we stelsels van niet-lineaire vergelijkingen oplossen. Ons resultaat werd gepubliceerd in [112]. Merk op dat dit geen impact heeft op de veiligheid van de encryptiemode van PANAMA (wanneer de cryptografische module gebruikt wordt als stroomcijfer).

6. Ontwerp van MAC-algoritmen

In dit hoofdstuk bespreken we het ontwerp van MAC-algoritmen (hashfuncties waarvoor de berekening afhankelijk is van een geheime sleutel). Net zoals bij sleutel-onafhankelijke hashfuncties is het ontwerp van een MAC-algoritme gebaseerd op de iteratie van een compressiefunctie met ingang van vaste lengte. Voor MAC-algoritmen is het doorgaans noodzakelijk voor de veiligheid om een uitgangstransformatie te gebruiken. Zowel de initiële waarde voor de kettingvariabele, de compressiefunctie als de uitgangstransformatie kunnen afhankelijk zijn van de geheime sleutel. De vereiste lengte van het MAC-resultaat wordt in grote mate bepaald door de applicatie waarin het algoritme gebruikt wordt; in de praktijk zijn lengtes van 32 of 64 bits vaak voldoende. De lengte van de geheime sleutel moet groot genoeg zijn om aanvallen waar exhaustief naar de sleutel gezocht wordt onhaalbaar te maken. Hiervoor is een sleutellengte van tenminste 80 bits vereist (meer voor veiligheid op lange termijn). Voor MAC-algoritmen moet men ook rekening houden met aanvallen gebaseerd op interne botsingen (dit zijn botsingen die optreden vóór de uitgangstransformatie). Preneel en van Oorschot [104, 105] hebben aangetoond dat zo een interne botsing gebruikt kan worden voor een verifieerbare MAC-vervalsing met een aanval op basis van slechts één enkel opgevraagd MAC-resultaat.

Men kan de compressiefunctie van een MAC-algoritme baseren op een bestaand cryptografisch schema. Een blokcijfer kan hiervoor gebruikt worden in CBC-mode ('Cipher Block Chaining'). De veiligheid van deze constructie kan bewezen worden onder de veronderstelling dat het gebruikte blokcijfer een pseudowillekeurige permutatie is [12]. Dit bewijs geldt enkel wanneer berichten van vaste lengte verwerkt worden, maar er zijn ook varianten op de CBC-MAC constructie (bv. EMAC [95] en OMAC [67]) die veilig zijn voor berichten van willekeurige lengte. In plaats van een blokcijfer kan men ook een hashfunctie gebruiken als basis voor een MAC-algoritme. Hiervoor moet men de functie afhankelijk maken van een secundaire ingang, de geheime sleutel. Twee aanbevolen constructies zijn

HMAC [15] en MDx-MAC [104]. Ook voor deze constructies kan de veiligheid bewezen worden op basis van een aantal veronderstellingen.

We stellen in dit hoofdstuk ook een eigen nieuw ontwerp voor: het algoritme Two-Track-MAC (gepubliceerd in [37]). Het ontwerp van Two-Track-MAC (of TTMAC) is gebaseerd op een compressiefunctie met dezelfde tweelijnsconstructie als in de compressiefunctie van de sleutelonafhankelijke hashfunctie RIPEMD-160 [100]. De twee parallelle lijnen worden echter niet gecombineerd op het einde van de compressiefunctie. Dit betekent dat de kettingvariabele een lengte heeft van 320 bits (in plaats van 160 bits). We zorgen er wel voor dat er een interactie optreedt tussen de twee lijnen, en na de laatste berekening van de compressiefunctie (dus na de verwerking van het laatste blok van het bericht) is er een uitgangstransformatie die het MAC-resultaat berekent door het verschil te berekenen tussen de twee lijnen. Opmerkelijk in het ontwerp van TTMAC is dat de geheime sleutel enkel gebruikt wordt als initiële waarde voor de kettingvariabele, en niet in de compressiefunctie of in de uitgangstransformatie. Dit is in tegenstelling tot andere MAC-constructies op basis van een hashfunctie (zoals HMAC en MDx-MAC); het wordt mogelijk gemaakt door de tweelijnsconstructie.

De veiligheid van TTMAC kan bewezen worden onder de veronderstelling dat de compressiefunctie pseudo-willekeurig is. Deze compressiefunctie is sterk gelijkaardig aan de compressiefunctie van RIPEMD-160; alhoewel RIPEMD-160 eerder ontworpen is met oog op de éénwegseigenschap en botsing-bestendigheid, lijkt de compressiefunctie ervan sterk te zijn door het gebruik van verschillende operaties (Booleaanse functies, bit-rotaties en modulaire optellingen). Onze evaluatie toont aan dat TTMAC een hoog veiligheidsniveau bezit tegen alle gekende aanvalsstrategieën. Het veiligheidsniveau is hoger dan voor HMAC en MDx-MAC (gebaseerd op RIPEMD-160 of SHA-1). De performantie van TTMAC benadert deze van RIPEMD-160. Een belangrijk voordeel is dat TTMAC zeer efficiënt is in het geval dat korte berichten (één of enkele blokken van 512 bits) verwerkt worden. Dit is omwille van de eenvoudige uitgangstransformatie, vergeleken met de HMAC en MDx-MAC constructies waar in de uitgangstransformatie de compressiefunctie van de onderliggende hashfunctie moet berekend worden. Een ander voordeel van ons algoritme is dat de geheime sleutel enkel als initiële waarde gebruikt wordt. TTMAC is dan ook heel geschikt voor toepassingen waar de sleutel regelmatig veranderd wordt (bv. na elke MAC-authenticatie). Voor HMAC en MDx-MAC daarentegen vereist elke sleutelverandering respectievelijk twee of zes extra berekeningen van de compressiefunctie.

We hebben TTMAC ingediend als kandidaat [124] voor het Europese NESSIE-project ('New European Schemes for Signatures, Integrity, and Encryption', [135]). Dit project had als voornaamste doelstelling om een portfolio van sterke cryptografische algoritmen voor te stellen. Deze portfolio bestaat uit verschillende categorieën van algoritmen, die bruikbaar zijn voor encryptie, authentisering of digitale handtekeningen. In totaal werden 42 cryptografische algoritmen ingediend als kandidaat voor NESSIE. Deze algoritmen werden gedurende meer dan twee jaar geëvalueerd, en uiteindelijk (februari 2003) werden twaalf algoritmen uitgekozen voor de verschillende categorieën van de portfolio (samen met vijf bestaande standaarden). TTMAC werd gekozen als MAC-algoritme voor de portfolio (samen met UMAC [76] en de standaarden EMAC [95] en HMAC [15]). Volgens NESSIE [93] heeft elk van deze algoritmen zijn eigen specifieke voordelen. Merk op dat de door NESSIE gekozen algoritmen geen standaarden zijn, maar dat er een behoorlijke graad van vertrouwen is in hun veiligheid omwille van de evaluatie die is uitgevoerd en het feit dat er geen zwakheden werden gevonden. Er wordt verwacht dat tenminste verschillende ervan in de nabije toekomst zullen aangenomen worden door standardisatieorganisaties.

7. Blokcijfers

Blokcijfers zijn een type van cryptografische algoritmen die hoofdzakelijk gebruikt worden voor encryptie. In ECB-mode ('Electronic Code Book') wordt het bericht opgedeeld in blokken van een vaste lengte, en wordt vervolgens elk van deze blokken vercijferd, gebruik makend van de geheime sleutel. Er zijn ook andere modes (bv. CBC of 'Cipher Block Chaining') voor het gebruik van een blokcijfer, die ervoor zorgen dat de verschillende vercijferingen niet onafhankelijk van elkaar gebeuren. Het moet voor een tegenstrever die de cijfertekst kent, onhaalbaar zijn om het oorspronkelijke bericht (de klaartekst) te weten te komen. Als de tegenstrever kennis heeft van een (groot) aantal klaarteksten en hun overeenkomende cijferteksten, moet het voor hem ook onhaalbaar zijn om de gebruikte geheime sleutel te weten te komen.

We geven in dit hoofdstuk een korte beschrijving van verschillende methoden voor het ontwerp van blokcijfers. De meest gekozen bloklengtes zijn 64 of 128 bits, en de sleutellengte moet groot genoeg zijn om aanvallen waar exhaustief naar de sleutel gezocht wordt onhaalbaar te maken. De veiligheid van een blokcijfer is gebaseerd op de bekende concepten van *confusie* en *diffusie*, geïntroduceerd door C. Shannon [119]. Praktische ontwerpen maken gebruik van een ronde-transformatie die een voldoende aantal keren herhaald wordt om een sterke encryptiefunctie te bekomen. De meeste algoritmen zijn gebaseerd op de zogenaamde Feistel-constructie of op een uniforme transformatiestructuur (ook substitutie-permutatie netwerk of SPN genoemd). Het DES-algoritme [54] is een voorbeeld van een Feistel-cijfer met een bloklengte van 64 bits, en het AES-algoritme [52] is een uniforme transformatiestructuur met een bloklengte van 128 bits. De eigenschap van confusie wordt in de meeste blokcijfers gerealiseerd door het gebruik van zogenaamde substitutiedozen of S-boxen.

Eén van de best bekende technieken voor het analyseren van blokcijfers is differentiële cryptanalyse, een probabilistische techniek die de propagatie van verschillen doorheen een algoritme bestudeert. In dit hoofdstuk ontwikkelen we een sleutelafhankelijke variant op deze techniek, toegepast op het blokcijfer ICE. ICE [78] is een Feistel-cijfer met een bloklengte van 64 bits, en bestaande uit zestien ronden (er is ook een snelle variant met acht ronden). Het meest opvallende kenmerk van dit algoritme zijn de sleutelafhankelijke permutaties die gebruikt worden om onder meer differentiële cryptanalyse te bemoeilijken. Volgens de ontwerper van het algoritme zou de beste strategie voor een differentiële aanval zijn om gebruik te maken van symmetrische 32-bit verschillen aan de ingang van de F-functie die de kern vormt van de ronde-transformatie. De waarschijnlijkheden in de differentiële cryptanalyse worden dan te laag om bruikbaar te zijn voor een aanval. We hebben echter ontdekt dat we een sleutelafhankelijke differentiële aanval kunnen uitvoeren, gebruik makend van kleine verschillen met een Hamming gewicht gelijk aan één. De analyse maakt gebruik van *voorwaardelijke differentiële karakteristieken* die geldig zijn voor slechts een deel van alle mogelijke sleutelwaarden. Op deze manier kunnen we de versie van ICE met acht ronden breken: de aanval werkt met een complexiteit van 2^{23} klaarteksten (en encryptie-operaties) voor 25% van de sleutels, en met een complexiteit van 2^{27} klaarteksten voor 95% van de sleutels. In theorie kunnen gereduceerde versies van ICE met hoogstens vijftien ronden gebroken worden met een complexiteit die lager is dan de complexiteit voor een exhaustieve zoektocht naar de sleutel.

Tenslotte hebben we het verband besproken tussen blokcijfers en hashfuncties. We hebben reeds eerder uitgelegd dat een blokcijfer gebruikt kan worden voor het ontwerp van een hashfunctie (en ook voor een MAC-algoritme). Langs de andere kant kan men een hashfunctie gebruiken om een blokcijfer te ontwerpen. De SHACAL-1 en SHACAL-2 blokcijfers [60], die gebaseerd zijn op de hashfuncties SHA-1 en SHA-256 (respectievelijk), zijn hier een voorbeeld van.

8. Toekomstig Onderzoek

Tijdens het onderzoek voor dit proefschrift zijn een aantal vragen gerezen die nog onbeantwoord zijn. Verdere cryptanalyse van hashfuncties van de MDx-klasse is mogelijk, vooral voor de meest recente algoritmen van de SHA-familie die door NIST gekozen zijn als standaarden zonder dat hun ontwerpstrategie werd bekendgemaakt, en zonder ondersteunende publieke evaluatie van hun veiligheid. Omwille van het gebruik van lineaire codes in deze hashfuncties, zou een cryptanalyse ervan waarschijnlijk voor een groot deel gebaseerd moeten worden op codeertheorie. Het is een open probleem of onze aanval op HAVAL uitgebreid kan worden naar versies met meer ronden in de compressiefunctie. Het MD5-algoritme is nog steeds populair ondanks het feit dat botsingen gevonden kunnen worden voor de compressiefunctie. Het is dus een interessante uitdaging om botsingen te zoeken voor de MD5-hashfunctie zelf. Anderzijds zou het nuttig zijn om een aanval gebaseerd op de verjaardagsparadox te implementeren voor MD5, wat haalbaar zou moeten zijn omdat de lengte van de hashresultaten slechts 128 bits is.

Nog een andere cryptanalytische uitdaging is om onze aanval op Panama te verbeteren. De huidige aanval is theoretisch (veel sneller dan een aanval gebaseerd op de verjaardagsparadox, maar te complex om praktisch te zijn). We houden er rekening mee dat door het gebruik van methoden zoals relinearisatie [71] voor het oplossen van de stelsels met niet-lineaire vergelijkingen, botsingen voor Panama in de praktijk gevonden zouden kunnen worden. Een algemene opmerking over de veiligheid van hashfuncties is dat meer onderzoek vereist is om andere eigenschappen te bestuderen. We denken in de eerste plaats aan de éénwegseigenschap, en ook bv. aan aanvallen die de pseudo-willekeurigheid van het resultaat analyseren (in het geval dat een deel van de ingang geheim is). We nodigen ook uit tot verdere evaluatie van ons eigen ontwerp Two-Track-MAC.

Op het gebied van het ontwerpen van nieuwe algoritmen, bestaat er zeker interesse voor nieuwe types van hashfuncties die een hoog veiligheidsniveau zouden moeten combineren met een goede performantie. Het zou ook nuttig zijn om meer bewijsbare eigenschappen te hebben voor hashfuncties (vergelijkbaar met blokcijfers waar ontwerpers van nieuwe algoritmen de weerstand tegen bijvoorbeeld differentiële cryptanalyse trachten te bewijzen). Voor hashfuncties die gebaseerd zijn op een blokcijfer is de vraag nog onbeantwoord welke voorwaarden aan het blokcijfer moeten gesteld worden opdat de hashfunctie veilig zou zijn.

Bart Van Rompay was born on April 25, 1973 in Leuven, Belgium. He received the degree of Electrical Engineering (Telecommunications) from the Katholieke Universiteit Leuven, Belgium in June 1996. His Master's thesis dealt with the cryptanalysis of the Blowfish block cipher. In October 1996 he started working in the research group COSIC (Computer Security and Industrial Cryptography) at the Department of Electrical Engineering (ESAT) of the K. U. Leuven.