Analysis and Design of Cryptographic Hash Functions

Pieter Retief Kasselman

Voorgelê ter vervulling van 'n deel van die vereistes vir die graad Magister in Ingenieurswese in die Fakulteit Ingenieurswese, Universiteit van Pretoria.

18 November 1999

Summary

Analysis and Design of Cryptographic Hash Functions

by

P. R. Kasselman and W. T. Penzhorn

Department of Electrical, Electronic and Computer Engineering

University of Pretoria

MEng Electronic Engineering

Indexing Terms: Hash Functions, Cryptanalysis, Cryptography, Message Integrity Code, Message Authentication Code, Differential Cryptanalysis, Boolean Functions, MD4, MD5, HAVAL.

Cryptographic hash functions are one of the primitive building blocks commonly used in information security. They form an important building block for authentication protocols, encryption algorithms, digital signatures and integrity checking algorithms. Two important properties of hash functions used in cryptographic applications are collision resistance and one-wayness. In this dissertation the focus is on collision resistance.

The dissertation provides a detailed overview of existing cryptographic hash functions, including definitions of fundamental properties, generic threats, and popular designs for cryptographic hash functions. Special attention is given to dedicated cryptographic hash functions related to the MD4 hash function.

Between 1990 and 1994 a number of practical cryptographic hash functions were designed and implemented, following the design principles of MD4. These cryptographic hash functions include MD4, MD5, SHA, SHA-1, HAVAL, RIPEMD-128 and RIPEMD-160. These functions were designed to exhibit the properties of collision resistance and one-wayness.

In this dissertation the attacks by Dobbertin on MD4 and MD5 are reconstructed. A novel approach is introduced that allows the execution of the attack on MD4 to be optimised. This new approach allows a reduction in computation time for a collision by a factor 64.

Based on these attacks a generalised attack is formulated. The generalised attack provides a new framework for the analysis of the collision resistant property of any cryptographic hash function.

This newly derived framework for the analysis of cryptographic hash functions is then applied to reduced versions of SHA and HAVAL. The results obtained in this investigation are the first cryptanalytical result to be published on the HAVAL hash function. The investigation shows that a collision can be found for a reduced version of HAVAL in less than a minute on a 200 MHz Pentium Pro personal computer. This result suggests that three and even four round HAVAL should not be used for security applications where message integrity and non-repudiation is required.

Based on the findings of these cryptanalytic attacks, a new set of design criteria for dedicated cryptographic hash functions is formulated. The design criteria aim to alleviate the common weaknesses identified in dedicated hash functions such as MD4, MD5, SHA, SHA-1 and HAVAL. Thereby the generalised attack developed in this dissertation can be thwarted.

Samevatting

Analise en Ontwerp van Kriptografiese Hutsfunksies

deur

P. R. Kasselman and W. T. Penzhorn
Department Elektriese, Elektroniese en Rekenaar Ingenieurswese
Universiteit van Pretoria
M Ing Electroniese Ingenieurswese

Indekseringterme: Hutsfunksies, Kripto-analisie, Kriptografie, Boodskap Integriteit Kode, Boodskap Stawing Kode, Differensiële Kriptoanalise, Boolese Funksies, MD4, MD5, HAVAL.

Kriptografiese hutsfunksies is een van die primitiewe boublokke wat algemeen gebruik word in informasiesekerheid. Dit vorm 'n belangrike boublok vir stawingsprotokolle, enkripsiealgoritmes, digitale handtekeninge en integriteitmeganismes. Twee belangrike eienskappe van hutsfunksies is weerstand teen botsings en die eenrigtingeienskap. In hierdie verhandeling val die fokus op die botsingweerstandseienskap.

Die verhandeling bevat 'n volledige oorsig van bestaande kriptografiese hutsfunksies, insluitend definisies van fundamentele eienskappe, generies bedreigings en populêre ontwerpe vir kriptografiese hutsfunksies. Spesiale aandag word gegee aan toegewyde kriptografiese hutsfunksies wat verwant is aan die MD4 hutsfunksie.

Tussen 1990 en 1994 is 'n aantal kriptografies hutsfunksies ontwerp en geïmplimenteer. Hierdie ontwerpe is gegrond op die ontwerpsbeginsels van MD4. Die kriptografiese hutsfunksies sluit in MD4, MD5, SHA, SHA-1, HAVAL, RIPEMD-128 en RIPEMD-160. Hierdie funksies is almal ontwerp om die eenrigting en botsingsweerstand eienskappe te vertoon.

In hierdie verhandeling word die aanvalle van Dobbertin op MD4 en MD5 gerekonstrueer. 'n Unieke benadering word voorgestel wat die aanval op MD4 optimeer. Die nuwe benadering verminder die berekeningkompleksiteit om 'n botsing te verkry met 'n faktor 64.

'n Veralgemeende aanval word geformuleer op grond van hierdie aanvalle. Die veralgemeende aanval voorsien 'n nuwe raamwerk vir die analise van die botsingsweerstand eienskap van enige toegewydte kriptografiese hutsfunksie.

Hierdie nuwe raamwerk vir die analiese van kriptografiese hutsfunksies word dan toegepas

op afgeskaalde weergawes van SHA en HAVAL. Die resultate van hierdie studie is die eerste kripto-analitiese resultate wat vir HAVAL gepubliseer is. Die studie toon dat 'n botsing vir die laaste twee rondtes van drierondte HAVAL verkry kan word in minder as 'n minuut op 'n 200 MHz Pentium Pro persoonlike rekenaar. Hierdie resultaat dui aan dat drie en selfs vier rondte HAVAL nie gebruik moet word vir sekuriteitstoepassings waar boodskapintegriteit vereis word nie.

Op grond van die kriptoanalitiese resultate word 'n nuwe stel ontwerpskriteria vir toegewyde kriptografiese hutsfunksies geformuleer. Die ontwerpskriteria is daarop gemik om die gedeelde swakhede geïdentifiseer in toegewyde hutsfunksies soos MD4, MD5, SHA, SHA-1 en HAVAL te vermy. Hierdeur kan die veralgemeende aanval wat in die verhandeling ontwikkel is gefnuik word.

Acknowledgements

I would like to make use of this opportunity to thank the following individuals and organisations.

My study leader, Prof. W.T. Penzhorn for his contributions, suggestions and encouragement throughout this project.

Prof. G.J. Kühn for the many stimulating conversations on the topic of cryptology.

Dr Bart Preneel and Antoon Bosselaers from the COSIC research group at the Katholieke Universiteit Leuven, Belgium for their comments on my work and suggesting the analysis of the HAVAL hash function.

My wife, Marelize, for her continued support and companionship throughout this project.

Ciphertec cc for the opportunity to perform research on the topic of cryptographic hash functions.

The management of Nedcor Bank Ltd for the time afforded to me in order to complete this dissertation.

CONTENTS

CHAPTER 1: INTRODUCTION

1.1 INFORMATION SECURITY

The exchange of information is an important social, economic and political activity. The importance of the exchange of information regarding all aspects of life has increased dramatically over the last 50 years. This period has seen the emergence of massive communication systems which have the sole purpose of facilitating the exchange of information. The proliferation of telephone and computer networks (both local and wide area networks) are manifestations of this phenomenon. As both the amount of information being exchanged and the significance attached to that information have increased, the need for ensuring the privacy and authenticity of the information have increased. This is especially true when considering communication systems where no human interaction is required. A computer network is a typical example of such a communication system. The need for privacy and authenticity is especially acute if a breech in either secrecy or authenticity may result in losses (financial or otherwise) for the participating parties in a communication system.

Organisations and people that use communication systems often express their needs for information security and trust in terms of three distinct requirements, frequently referred to as the *CIA-Model* of security:

- *Confidentiality* – ensuring the non-disclosure of sensitive information
- *Integrity* – ensuring that data and information is only changed and modified in a specified and authorised manner
- *Availability* – ensuring that systems work promptly and correctly, and that service is not denied to authorised users.

Over the past three decades numerous techniques have been invented and developed to ensure both privacy, authenticity and availability of information in communication systems. These techniques are often constructed from a number of basic cryptographic primitives. One of the primitives most widely used is the cryptographic hash function.

1.2 HASH FUNCTIONS AND SECURITY

Informally, a hash function is defined as a function that compresses an input string of arbitrary length to an output string of fixed length (see Figure 1.1).

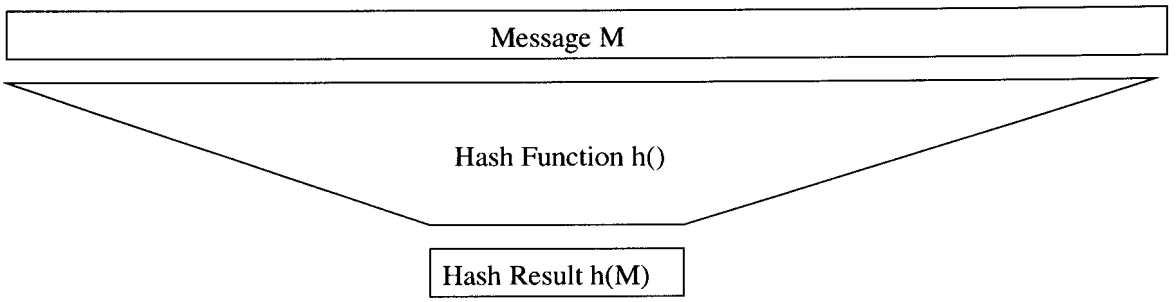

Figure 1.1: Hash Function

More formally the following definition applies to a hash function [1]:

Definition 1.1 (Hash Function) *A hash function is defined as an easily computable function, $h()$, that maps every binary sequence of length μ or greater to a binary sequence of length m, where μ and m are specified parameters.*

A wide range of terminology exists for hash functions used in security applications [2],[3]. These terms include message integrity codes, message authentication codes, manipulation detection codes, cryptographic hash functions or simply hash functions. In this document hash functions used for security applications are referred to as cryptographic hash functions or hash functions for conciseness.

1.2.1 Applications of Hash Functions

Hash functions have their origin in the field of computer science and were originally used for data storage and retrieval [4]. For data storage applications a hash function is used to compute an abbreviated representation of a filename. This abbreviation is then used to index and store the file. When the file is retrieved, the hash value for the given filename is used to retrieve the data. Using a hash function reduces the storage requirements for a data retrieval system, since only the hash value has to be stored, instead of the entire filename.

A number of other uses have been found for hash functions. Amongst others, hash functions are used in compiler symbol tables, graph theory problems, transposition tables in computer games, spell checkers, tests for set equality and security applications [2], [3]. It is the application of hash functions in security solutions that constitutes the topic for this dissertation. A short list of security applications that rely on the use of cryptographic hash functions is shown below:

- Authentication protocols [5].
- Digital signatures [6].
- Electronic commerce [5].
- Encryption schemes [7] [8] [9].

Because of the wide ranging applications of hash functions in security solutions they are considered important cryptographic primitives. The above list of applications can roughly be split into two categories:

- Authentication services.
- Encryption services.

Authentication Applications

The list of security applications primarily represents authentication and non-repudiation services. When using a cryptographic hash function in an authentication scheme the authenticity of the message is transferred to the hash value. It is then necessary to provide protection and authentication for the hash value only, instead of the entire message. The hash value serves as an authentication tag for the message, and can be appended to a message. A number of cryptographic protocols and electronic commerce implementations rely on cryptographic hash functions to provide these services. These protocols include S/MIME, SSL, TLS, WTLS, SET as well as the EMV specifications.

Encryption Applications

The non-linear and one-way property of cryptographic hash functions are exploited when used in encryption schemes. Cryptographic hash functions can be used as non-linear elements in block ciphers based on the Feistel structure. Four block ciphers, LION, BEAR, LIONESS and AARDVARK that are based on the existence of secure cryptographic hash functions were recently proposed in [7], [8] and [9]. It is also possible to exploit the non-linear and one-way properties of a cryptographic hash functions in the construction of stream ciphers.

1.2.2 Properties of Hash Functions

Earlier in this section a hash function is defined as a function that compresses an input string of arbitrary length to an output string of fixed length. The inherent weakness of all hash functions is contained within this definition. An intuitive explanation of the inherent weakness of hash functions is presented in this section.

Consider the projection of an M dimensional space onto an N dimensional space with $M > N$, f is the mapping (hash) function (see Figure 1.2).

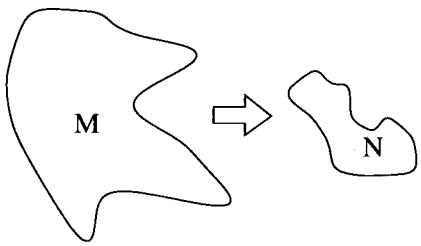

Figure 1.2: Hashing: A Spatial View $f : M \mapsto N$

From Figure 1.2 it is clear that the projection of all possible representations in M onto N is not unique if $M > N$. The lack of uniqueness of the mapping function $f()$ implies that more than one vector in M is mapped to the same vector in N. This is known as a collision. A graphical representation of the above statement is given for $M = 3$ and $N = 2$ in Figure 1.3.

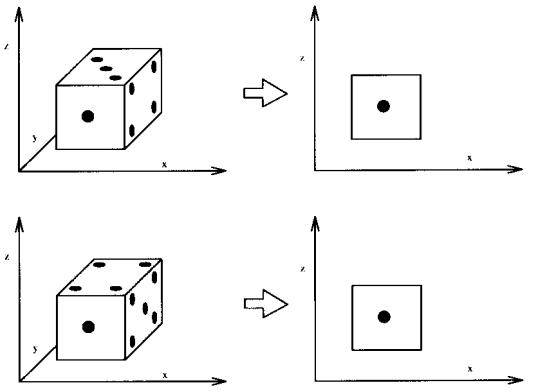

Figure 1.3: Graphical example of a collision

A more formal representation of the above is given below:

$$f() : m_1 \mapsto n_1$$
$$f() : m_2 \mapsto n_1$$

with:

$$\begin{aligned} n_1 &\in N \\ m_1, m_2 &\in M \\ m_1 &\neq m_2. \end{aligned}$$

From the above it follows intuitively that the number of collisions increases as the ratio $\frac{M}{N}$ increases.

This argument leads to the conclusion that collisions exist for all hash functions as previously defined. The existence of collisions for all possible hash functions is an inherent weakness. This weakness is, by definition, also present in cryptographic hash functions. Consequently it is required that cryptographic hash functions exhibit the properties of one-wayness and collision resistance. These properties make it computationally intractable to find collisions.

The properties of one-wayness and collision resistance allow hash functions to be used in order to provide the services of integrity, digital signatures and non-repudiation.

One-way Property

From a cryptographic point of view one-way hash functions are of particular interest. The one-way property is defined in [1] as:

Definition 1.2 (One-Way Hash Functions) *A one-way hash function is defined as a hash function such that, for virtually every binary string y of length m it is infeasible to find a binary string x of length μ or greater such that $y = h(x)$.*

Collision Resistant Property

A further desirable property of cryptographic hash functions is that of collision resistance. The concept of a collision was introduced earlier in this section. It is appropriate to introduce a more formal definition of a collision before defining the concept of collision resistance [59].

Definition 1.3 (Collisions) *A collision is obtained for a given hash function $h()$ if two distinct messages, M and $\tilde{M}$, are found, such that for a specific initial value (denoted by IV):*

$$h(IV, M) = h(IV, \tilde{M}).$$

Consequently a collision resistant hash function is defined as [59]:

Definition 1.4 (Collision Resistant Hash Function) *A collision resistant hash function is defined as a hash function for which it is computationally intractable to find collisions.*

Generally when referring to cryptographic hash functions it is expected that they exhibit the properties of one-wayness and collision resistance.

The One-way Property, Collisions and Information Theory

It can be shown that the presence of collisions is a pre-requisite for one-wayness by applying the principles of information theory and source coding to the hashing problem.In this model

the message to be hashed represents the source, the hash result corresponds to the encoded symbols or messages, and the hash algorithm represents the source encoding algorithm. Let $H(X)$ be the entropy of the message source (X is a random variable). Let each symbol x_i represent a message for $i = 1, 2, 3, \ldots, M$. It is then known that $H(X) \leq \log_2 M$. It is also known that for each symbol to be encoded uniquely with N bits, N is chosen such that $N = \log_2 M$. This implies that M should not exceed 2^N if no collisions are required. However, remember that for cryptographic hash functions:

1. The requirement of one-wayness has to be fullfilled.
2. The input alphabet should have an arbitrary size.

Hash functions used in cryptographic applications have to be one-way. If the hash result corresponds to one, and only one input, the property of one-wayness is violated.

The requirement that the input alphabet should have an arbitrary size implies that $M \geq 2^N$ or $M < 2^N$. If $M < 2^N$ the source encoding algorithm may become inefficient. From a cryptographic point of view this is a minor problem. If $M \geq 2^N$ the probability of an encoding error, P_e, becomes non-zero. If $P_e \neq 0$, collisions exist. If P_e approach 1, the function becomes one-way. In order for P_e to approach 1, $N \leq H(X) - \epsilon$, for any $\epsilon > 0$. By setting N to a fixed size and choosing $M \ggg 2^N$, the above condition is satisfied. Thus the source coding algorithm becomes one-way but produces collisions. Thus collisions exist, not only as a result of the requirement of encoding messages of arbitrary length but also as a result of the requirement for one-wayness. As long as it remains difficult to obtain messages that have the same hash result, the function is called collision resistant.

In order to demonstrate the need for cryptographically secure hash functions consider the following example:

Example

Consider a typical electronic transaction. Two parties agree to the sale of a specified item for R10000,00. An (electronic) contract is drafted. The seller computes the hash value of the contract and applies his digital signature to the hash value. The buyer does the same and the sale is agreed upon. However the parties involved did not utilise a collision resistant hash

function. Consequently the seller was able to draft an alternative contract which has the same hash result as the original, except that the agreed upon price is changed from R10000,00 to R20000,00. The buyer now finds that he is committed to purchase the item in question at twice the agreed upon price.

In the above example the participating parties relied upon the hash function to provide assurance of the data integrity. In effect the message integrity was transferred to the integrity of the hash function. It is shown that the use of a weak hash function compromises the security objective of data integrity. Similar examples pertaining to authentication protocols and encryption schemes may be listed where the failure of the hash function undermines the security objective. For this reason efficient and strong cryptographic hash functions are required.

1.2.3 Hash Functions Today

Hash functions are widely used in cryptographic applications. As demonstrated in the previous section the properties of one-wayness and collision resistance are of particular importance in security applications. During the last decade numerous proposals were made to construct dedicated hash functions that are both one-way and collision resistant. These proposals include MD4, MD5, SHA, SHA-1, HAVAL, RIPEMD-128 and RIPEMD-160. MD4 was published in 1990 by Rivest [10]. By the end of 1991 it was demonstrated that neither the first two rounds (Merkle) nor the last two rounds of MD4 (Bosselaers and den Boer [11]) are collision resistant. The lessons learned from these attacks led to the design of MD5 [12] and SHA [13]. In 1996 Dobbertin showed that MD4 is not a collision resistant hash function by demonstrating a technique which allowed the construction of collisions for all three rounds of MD4 [14]. Within six months Dobbertin demonstrated that collisions may be found for the compress function of MD5 [12]. Although details of this attack have not been published, it is believed to be based on similar techniques as described in [14]. Dobbertin has also shown that these attacks are applicable to RIPEMD-128. The speed with which these attacks could be adapted to different hash functions derived from the same basic construction is a cause for concern since it may be indicative of a fundamental flaw in the design of the basic construction. This concern has led to the design of RIPEMD-160 to replace RIPEMD-128 [15]. In 1998 Dobbertin showed that the first two rounds of MD4 are not one-way. The attacks formulated by Dobbertin utilises techniques borrowed from a wide range of disciplines ranging from genetic algorithms to Boolean algebra. These hash functions are all based on

the same design principles and criteria. The weaknesses found in these hash functions may be indicative of a common design weakness.

As shown above a number of the popular dedicated hash function constructions were found to be cryptographically inadequate. In particular it was found that the requirement for collision resistance is hard to satisfy. One of the reasons for this is the threat model used when considering the property of collision resistance. In this threat model the cryptanalyst not only has full knowledge of the algorithm used (Kerckhoff's principle [59]) but also has control over all aspects of the input to the hash function. The attacker is often a legitimate participant in the system and is trusted to a certain extent. Given the above threat model it should remain computationally difficult to construct collisions or find a specified hash results.

1.3 PROBLEM STATEMENT

Cryptographic hash functions are important cryptographic primitives and are widely used in security applications where message integrity is required. The design of cryptographic hash functions have proved to be a difficult task. A recent spate of attacks showed that a number of commonly used hash functions exhibit cryptographic weaknesses. The absence of secure cryptographic hash functions will make dependable message integrity, non-repudiation and message authenticity impractical. It is therefore important to understand the basis of the attacks, determine if they share common elements and establish design criteria to foil these attacks.

1.4 HYPOTHESIS

It is the hypothesis that the recent spate of attacks formulated by Dobbertin has a common underlying structure and that these attacks exploit certain architectural properties of the MD4 family of hash functions.

1.5 SCOPE

In this dissertation only dedicated, iterated cryptographic hash functions are studied. In particular the MD4 family of hash functions are considered. Although a general review of

generic attacks are included in this dissertation only the attacks formulated by Dobbertin are considered in depth.

1.6 DISSERTATION OBJECTIVES AND METHODOLOGY

This dissertation has the following objectives:

1. Lay a foundation for the analysis and design of cryptographic hash functions.
2. Reconstruct the attacks on MD4 and MD5 as formulated by Dobbertin.
3. Generalise the analysis of MD4 and MD5 to create a framework for the analysis of iterated dedicated hash functions.
4. Apply the generalised analysis framework to practical cryptographic hash functions.
5. Formulate design criteria to prevent the successful application of the generalised analysis framework.

In order to lay a foundation for the analysis and design of cryptographic hash functions we present an in-depth study of the current state of cryptographic hash functions. Included in this study are the definitions (Chapter 1), taxonomy (Chapter 2), generic threats (Chapter 3), common requirements (Chapter 4) and general designs of cryptographic hash functions (Chapter 5). Once a general foundation is laid for the understanding of cryptographic hash functions the focus is shifted to practical dedicated cryptographic hash functions.

As part of the focus on dedicated cryptographic hash functions the attacks on MD4 and MD5 are reconstructed (Chapters 6 and 7). The C-programs used to reconstruct these attacks are attached as Appendix B, C, D and E. This is one of the main objectives of the dissertation. A novel approach is derived that allows the attack on MD4 to be optimised to obtain a reduction in computation time for a collision by a factor 64.

Based on the reconstruction of these attacks a generalised attack is formulated (Chapter 8). The generalised attack provides a framework for the analysis of the collision resistant property of any cryptographic hash function.

The newly derived framework for analysing a cryptographic hash functions is applied to reduced versions of SHA and HAVAL (Chapters 9 and 10). Extensive simulations were performed using the C programming language. A sample of the resulting source code is included as Appendices F and G. To the best of our knowledge this is the first cryptanalytical result that has been published on the HAVAL hash function. The result shows that a collision can be established for a reduced version of HAVAL in less than a minute on a 200 MHz Pentium Pro. This result suggests that three and even four round HAVAL should not be used for security applications where message integrity and non-repudiation is required.

The dissertation is concluded by presenting design criteria for dedicated cryptographic hash functions (Chapter 11). The design criteria is based on the common weaknesses identified in the analysis of MD4, MD5, SHA, SHA-1 and HAVAL. It is the intention that the application of these design criteria will defeat the generalised attack on iterated cryptographic hash functions presented earlier in the dissertation.

1.7 RESULTS

The following are the main results of this dissertation:

1. Successful reconstruction of the attacks on MD4 and MD5 as formulated by Dobbertin.
2. Speedup the attack on MD4.
3. Create a generalised framework for the analysis of iterated dedicated hash functions based on the MD4 family.
4. Apply the generalised analysis framework to reduced versions of HAVAL and SHA.
5. Formulate design criteria to prevent the successful application of the generalised analysis framework.

CHAPTER 2: TAXONOMY OF CRYPTOGRAPHIC HASH FUNCTIONS

2.1 INTRODUCTION

In Chapter 1 the relevant definitions and properties related to hash functions were defined. In this chapter a taxonomy of practical cryptographic hash functions is presented, along with the common approaches to the design and analysis of cryptographic hash functions.

The taxonomy is based on the terminology that exists in the banking community and is taken from [3]. Cryptographic hash functions are divided into the following categories:

1. Message Authentication Codes (MAC).
2. Manipulation Detection Codes (MDC).
   (a) One Way Hash Function (OWHF).
   (b) Collision Resistant Hash Function (CRHF).

A graphical summary of the above taxonomy is shown in Figure 2.1.

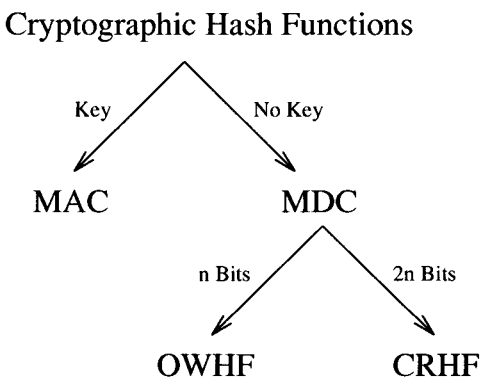

Figure 2.1: A Taxonomy of Cryptographic Hash Functions

Informal definitions for the categories of hash functions are suggested in [16] and refined in [3]. The distinction made between the different cryptographic hash functions is based quantitatively on the following definitions.

2.1.1 MAC

A MAC is a hash function for which a secret key is required. This adds to the security of the hash scheme, since the attacker's abilities decrease as his knowledge decreases. However the requirement for a secret key does not protect the users against an attack by an insider. The addition of a secret key leads to the additional problem of key management. It does however have the advantage that a secure channel[1] is no longer required for the hash value, since the secret key protects the hash value. It is however necessary to provide a secure channel for the key used in the MAC. More formally:

Definition 2.1 *A* **MAC** *is a function h() satisfying the following conditions:*

1. *The description of h() must be publicly known and the only secret information lies in the key, K, (extension of Kerkhoff's principle).*
2. *The argument X can be of arbitrary length and the result h(K,X) has a fixed length of n bits ($n \leq 32 \ldots 64$).*
3. *Given h(), K and X, the computation of h(K,X) must be easy.*
4. *Given h() and X, it is hard to determine h(K,X) with a probability of success significantly higher than 2^{-n}. Even where a large set of pairs $\{X_i, h(X_i, K)\}$ is known, where X_i have been selected by the opponent, it is "hard" to determine the key K or to compute h(K,X') for any $X_i \neq X'$.*

2.1.2 MDC

A MDC is a hash function that is computed without knowledge of a secret key. These functions are known publicly. For these hash functions, no key management is required, but an authentic channel needs to be provided for the hash value.

Two variants of MDCs are identified in [16] and [3]. The following definitions are used to distinguish between one way hash functions (OWHF) and collision resistant hash functions (CRHF).

[1] An authentic or secure channel could be provided through encryption of the hash value, a separate channel or a courier.

One Way Hash Function (OWHF)

Definition 2.2 *A* **One Way Hash Function** *is a function h() satisfying the following conditions:*

1. *The description of h() must be publicly known and should not require any secret information for its operation (extension of Kerkhoff's principle).*
2. *The argument X can be of arbitrary length and the result h(X) has a fixed length of n bits ($n \leq 64$).*
3. *Given h() and X, the computation of h(X) must be easy.*
4. *The hash function must be one way in the sense that:*
   (a) *given a Y in the image of h(), it is "hard" to find a message X such that h(X) = Y.*
   (b) *given X and H(X) it is "hard" to find a message $X' \neq X$ such that $h(X) = h(X')$.*

Collision Resistant Hash Function (CRHF)

Definition 2.3 *A* **Collision Resistant Hash Function** *is a function h() satisfying the following conditions:*

1. *The description of h() must be publicly known and should not require any secret information for its operation (extension of Kerkhoff's principle).*
2. *The argument X can be of arbitrary length and the result h(X) has a fixed length of n bits ($n \leq 128$).*
3. *Given h() and X, the computation of h(X) must be easy.*
4. *The hash function must be one way in the sense that:*
   (a) *given a Y in the image of h(), it is "hard" to find a message X such that h(X) = Y.*
   (b) *given X and H(X) it is "hard" to find a message $X' \neq X$ such that $h(X) = h(X')$.*

5. *The hash function must be collision resistant: This means that it is hard to find two distinct messages that hash to the same result.*

The nature of the differences between OWHF and CRHF is discussed in [17]. The underlying difference between OWHF and CRHF is related to the the type of attack the respective hash functions are required to withstand. For cryptographic purposes a CRHF is of greater value than an OWHF.

Implicit to the above definitions are the requirements for one-wayness, computational intractability, collision resistance and simplicity. These requirements are related to both the functional and security properties of cryptographic hash functions.

A new hash function should therefore be designed to adhere to the above definitions and implied requirements. The definitions and requirements can be made more formal by specifying quantitative criteria for the terms *hard* and *easy*.

2.2 APPROACHES TO THE DESIGN AND ANALYSIS OF CRYPTOGRAPHIC HASH FUNCTIONS

Two approaches could be considered for the analysis, design and classification of cryptographic hash functions. Since hash functions are used extensively in authentication applications and protocols [17], hash functions could be classified along the same lines as authentication codes. In [18] the following classification is given for authentication schemes:

1. Computationally secure.
2. Provably secure.
3. Unconditionally secure.

The above classification is not satisfactory when dealing with hash functions. As explained in Chapter 1 collisions exist for all hash functions. This property of hash functions leaves only the computationally secure classification as a viable option. The above classifications does not contribute a great deal to design criteria for cryptographic hash functions.

In [3] Preneel suggests that the same classification scheme be used as that proposed by Rueppel for stream ciphers. Accordingly one of three approaches are available:

1. Information theoretic approach.
2. Complexity theoretic approach.
3. System based or practical approach.

The information theoretic approach and complexity theoretic approach yields interesting constructions of variable security. In general the constructions based on these two approaches are impractical. This leaves the system based or practical approach. In the system based approach, practical schemes with fixed parameters and dimensions are studied.

There has been numerous proposals for the design of cryptographic hash functions based on the system based approach to hash functions. Many of these designs are based on existing cryptographic primitives such as block and stream ciphers. Other proposals utilise modular arithmetic and the hardness of number theoretical problems as a basis for design. However the hash functions which have found the widest acceptance in industry are dedicated hash functions. The following definition of a *d*edicated cryptographic hash function is presented:

Definition 2.4 (Dedicated Cryptographic Hash Functions) *A dedicated cryptographic hash function is a hash function which has been designed to meet the requirements set for cryptographic applications and is to be used explicitly for hashing purposes.*

One family of dedicated hash functions, known as the MD4-family of hash functions, has found widespread acceptance in industry. Members of this family of dedicated hash functions are used by the Secure Electronic Transaction (SET) protocol specified by Mastercard and Visa, Secure Socket Layer (SSL) protocol commonly used for securing Internet commerce as well as the Secure MIME (S/MIME) protocol used to secure electronic mail to name a few of the more popular protocols.

In this dissertation the practical approach is used to analyse dedicated cryptographic hash functions and establish suitable design criteria for dedicated cryptographic hash functions.

CHAPTER 3: THREATS AGAINST HASH FUNCTIONS

3.1 INTRODUCTION

Before proceeding to establish requirements for hash functions, it is appropriate to consider the threats against hash functions. In this chapter both the attackers as well as the attacks they are capable of are considered.

Attackers are classified with regard to their capabilities and their position regarding the system under attack. As the wealth and resources of an opponent increases, the difficulty of designing a secure hash function increases. For this reason it is important to be aware of the capabilities of various classes of attackers. When designing a hash function it should be decided which class of attacker is to be denied a successful attack.

In addition to the attackers and their capabilities, the attacks they are capable of are considered. A taxonomy of these attacks are presented in this chapter. For the attacks described in this chapter, the computational power and storage capabilities required for the execution of these attacks are emphasised. These requirements are stated as a function of the number of bits, n, contained in the hash length. In this report the attacks specific to MDCs, MACs and hash algorithms based on block ciphers are considered.

3.2 TAXONOMY OF ATTACKERS

A distinction is made between the capabilities of attackers and their position with regard to the hash function they seek to attack.

3.2.1 Capabilities

The capability of an attacker is measured in terms of the resources available to him. In [19] a taxonomy of attackers on tamper resistant devices is presented. This classification is based on the resources available to the attackers. This taxonomy can be extended to security mechanisms in general, including hash algorithms. Attackers are categorised as follows:

Class I (clever outsiders): They are often very intelligent but may have insufficient knowledge of the system. They may have access to only moderately sophisticated equipment.

They often try to take advantage of an existing weakness in the system, rather than try to create one.

Class II (knowledgeable insiders): They have substantial specialised technical education and experience. They have varying degrees of understanding of parts of the system, but potential access to most of it. They often have access to highly sophisticated tools and instruments for analysis.

Class III (funded organisation): They are able to assemble teams of specialists with related and complementary skills backed by great funding resources. The are capable of in-depth analysis of the system, designing sophisticated attacks, and using the most advanced analysis tools. They may use Class II adversaries as part of the attack.

The threat from Class I and Class II attackers can be dealt with by placing a hash algorithm in the public domain and allowing experts in the field to analyse and review the algorithm before widespread implementation. This approach will also ensure that the threat from Class III attackers are minimised. When designing a hash function it is advised that the hash function should be able to withstand attacks from a Class III opponent. This is difficult since it is not always known what a Class III opponent's capabilities are.

3.2.2 Position

In addition to the taxonomy of attackers based on their capabilities, a taxonomy of attackers is presented with regard to their position concerning the system they seek to attack. Regarding cryptographic hash functions the following attackers are identified:

Legitimate Participants: These are participants who rightfully share in a communication process. They are allowed to generate, sign and transmit messages. In the case of MACs they have access to the shared secret key. These attackers can generate two messages that yield the same hash value and substitute the one message for another when convenient.

Active Eavesdroppers: These attackers are not allowed to generate, sign and transmit messages. They are hostile eavesdroppers who seek to intercept and modify messages without detection. This imply that they would attempt to construct a false message

that has a specific hash value and replace a valid message when intercepted. They are not expected to have access to shared secret keys for MACs.

These attackers can belong to Class I, Class II or Class III attackers, depending on their capabilities. The taxonomy of attackers is summarised in Figure 3.1.

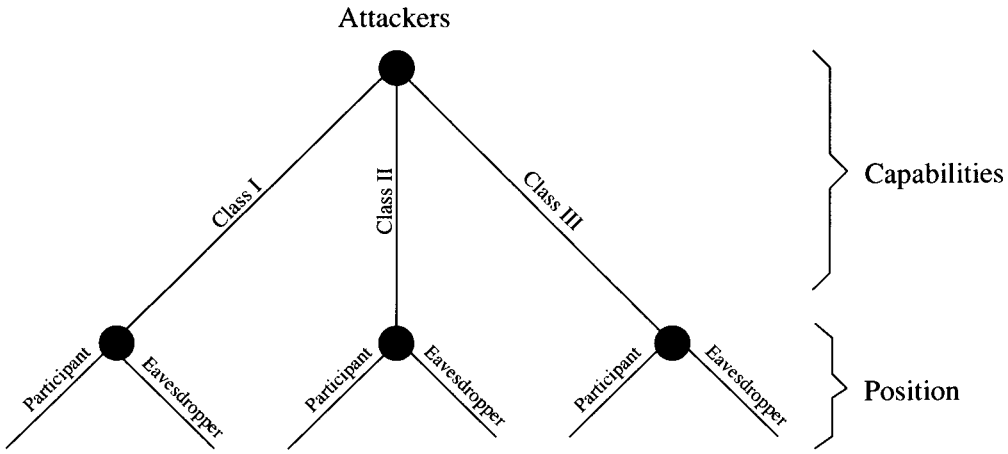

Figure 3.1: Taxonomy of Attackers

3.3 TERMINOLOGY

In this chapter the following attacks are considered:

1. Attacks on MDCs.
2. Attacks on MACs.
3. Attacks on underlying block ciphers.
4. High level attacks.
5. Attacks dependent on the algorithm.

Before proceeding with a description of the generic attacks on hash Functions, it is useful to consider the terminology used in describing the generic attacks. The terminology introduced in this section serves as an indication of what an attacker could hope to achieve when attacking a hash function.

3.3.1 MDC Terminology

When attacking a MDC, an attacker could hope to construct a:

Pre-image: Establishing a pre-image is equivalent to finding a message that results in a specified hash value.

Second pre-image: A second pre-image requires the attacker to find two messages that results in a specific hash value.

Pseudo-pre-image: A pseudo-pre-image requires that two messages, X and X', with two different initial values, IV and IV' should be found such that the $h(IV, X)$ and $h(IV', X')$ result in the same specified hash value.

Collision: A collision is established if an attacker can find two messages X and X' such that $h(IV, X)$ and $h(IV, X')$ result in the same unspecified hash value.

Collision for different IV's: A collision for different IV's is established if two messages and two IV's can be found such that $h(IV, X)$ and $h(IV', X')$ hash to the same hash value.

Pseudo-collision: A pseudo collision is established if an attacker can find two messages X and X' such that $h(IV, X)$ and $h(IV', X')$ yield the same hash value for two specified IV's.

Constructing collisions, collisions for different IV and pseudo-collisions are easier than constructing a second pre-image or pseudo pre-image. Attacks specific to MDCs are considered in Section 3.4.

3.3.2 MAC Terminology

A MAC makes use of a secret key to compute a hash value. Thus for a MAC the collision is dependent on both public knowledge (the message) and secret knowledge (the key). The attacker is therefore faced with two problems. The first deals with the key, the second deals with the construction of collisions. When dealing with a MAC, an attacker could hope to achieve one of the following objectives [3]:

Key recovery: The attacker can determine the secret key K.

Universal forgery: The attacker constructs an alternative algorithm that mimics the MAC algorithm.

Selective forgery: For a message chosen by the attacker the correct MAC can be determined.

Existential forgery: An attacker can determine a correct MAC for at least one plaintext. The resulting plaintext may be random or non-sensical.

Once the secret key is known to an attacker, he can determine the MAC for any message. With the MAC algorithm and the secret key known, an attacker can proceed to construct a collision. Techniques describing key retrieval and the construction of forgeries are considered in Section 3.5. The objectives for generating a collision for a MAC when the secret key is known are similar to those for a MDC.

3.4 ATTACKS ON MDCS

In this section a number of generic attacks on MDCs are considered. These attacks can be classified as belonging to one of two categories. They are:

1. Attacks independent of the algorithm.
2. Attacks dependent on the chaining.

These attacks are generic and could be used against any hash function. In this section, these attacks are summarised and evaluated according to the computational power required to execute them successfully.

3.4.1 Attacks Independent of the Algorithm

For MDCs two attacks are considered to be independent of the algorithm. This implies that these attacks can be carried out against the ideal cryptographic hash function described in Chapter 1. These attacks are known as the random attack and the birthday attack.

Random Attack

In this attack it is assumed that the attacker is given a message X and requires a message X' such that $X' \neq X$ and $h(IV, X') = h(IV, X)$ (i.e. the attacker has to find a second pre-image). This can be accomplished by randomly selecting X' from all possible admissible messages. The probability of success is 2^{-n} with n the length in bits of the hash value. If an attacker performs T trials, the probability of finding a valid value for X' so that $X' \neq X$ and $h(X') = h(X)$ becomes $T \cdot 2^{-n}$. Thus, the larger n the larger number of trials T are required. According to [1] approximately $0.7 \cdot 2^{-n}$ trials are required to find a collision using this technique. Thus for a n-bit hash value the expected workload to find a second pre-image is in the order of $O(2^n)$.

Birthday Attack

This attack is based on the the birthday paradox from probability theory. According to this paradox, it can be shown that the probability that two individuals in a group of 23 people share a birthday, is approximately 52%. The number of people in the group is much smaller than expected. A related problem states that for two groups of 17 people, the probability that two people have a common birthday, is larger than 50%. This property can be exploited to attack hash functions as explained below.

For two sets of messages r_1 and r_2 it is shown in [3] that:

$$P_r(h(X) = h(X')) = 1 - e^{-\frac{r_1 \cdot r_2}{n}}$$

for:

$$r_1 = r_2 = O(\sqrt{n})$$

the probability

$$\begin{aligned} P_r(h(X) = h(X')) &= 1 - e^{-1} \\ &\approx 63\%. \end{aligned}$$

An example of an algorithm that makes use of this result is presented as algorithm 3.1.

Algorithm 3.1 *Birthday attack for hash functions*

1. *For a n bit hash value let $r_1 = r_2 \approx O(2^{\frac{n}{2}})$*
2. *Generate r_1 variations on the valid message X.*
3. *Generate r_2 variations on the forged message X'.*
4. *Compare the hash values for the r_1 variations of X with the r_2 variations of X'. When a message X and X' is found for which $h(IV, X) = h(IV, X')$, a collision is established.*

The attacker can now generate a message that contains X and then later replace X with X' and claim that he originally generated X', since the hash values for both messages are the same.

In [20] and [21] alternative algorithms for efficient collision search is proposed. These techniques are based on Pollard's ρ method for finding cycles in periodic functions in a finite domain. These techniques were used in the analysis of DES.

The significance of this attack is that the number of operations required to find a collision is $O(2^{\frac{n}{2}})$ instead of $O(2^n)$ for the random attack. A similar order of magnitude is required in storage capabilities. Thus a birthday attack requires less operations than a random attack. The only way to defend against birthday attacks is by increasing the number of bits n in order to make it computationally infeasible to launch a birthday attack.

3.4.2 Attacks Dependant on the Chaining

If a message is longer than the maximum block length of the hash algorithm, the message is segmented. The segments are then processed iteratively (Chapter 5 Section 5.3). This is known as the Damgård-Merkle scheme [22], [23]. A number of attacks have been derived which are only applicable if an iterated structure is used. The attacks are summarised below.

Meet in The Middle Attack

This is a variation of the birthday attack. This attack allows the attacker to construct two messages, X and X', for which $h(X) = h(X')$. The messages X and X' should be at least twice as long as the elementary block length of the hash function. The following algorithm describes the meet in the middle attack.

Algorithm 3.2 *Meet in the middle attack for hash functions*
Consider Figure 3.2.

1. *For a n bit hash value let $r_1 = r_2 \approx O(2^{\frac{n}{2}})$*
2. *Generate r_1 variations on X_1'.*
3. *Generate r_2 variations on X_2'.*
4. *Work forward from the IV and compute r_1 variations of the intermediate values $IM1$ with $IM1 = h(IV, X_1')$ and save this in a buffer of intermediate values IM_1.*
5. *Work backward from $h(X)$ and compute r_2 variations on $h(X) = h(IM2, X_2')$ and save the intermediate values in a buffer of intermediate values IM_2.*
6. *Use a search algorithm and search for two intermediate values that are equal in $IM1$ and $IM2$ respectively. If two equal values are found, a collision is established.*

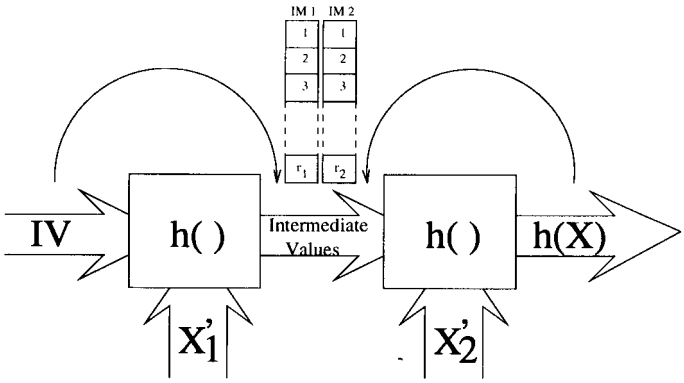

Figure 3.2: Meet in the middle attack

As in the case of the birthday attack the number of operations required to establish a collision are in the order of $O(2^{\frac{n}{2}})$. The advantage of this attack is that it allows an attacker to hit a

specific hash value. This attack is only possible if the message is longer than an elementary message block.

It is possible to defend against these attacks by increasing the number of bits in a hash value to such an extent that the meet in the middle attack is computationally infeasible. Another defence against this attack is to constrain the message lengths to less than the elementary block length.

When imposing constraints on the solutions obtained with the meet in the middle attack, the attack is called the constrained meet in the middle attack.

Generalised Meet in The Middle Attack

To avoid the meet in the middle attack, two-fold iterated schemes were suggested in [6]. These schemes include computing two hash values for a given message using two different IV's ($h(IV, X)$ and $h(IV', X)$) or computing the hash value on the concatenation of the message to itself ($h(IV, X||X)$). These schemes can be extended to so-called p-fold schemes where p hash values are computed for the same message using p initial values, or by concatenating the message p times to itself and then computing a hash value ($h(IV, X||X||\ldots X)$). A graphical representation of these p-fold schemes are shown in Figure 3.3.

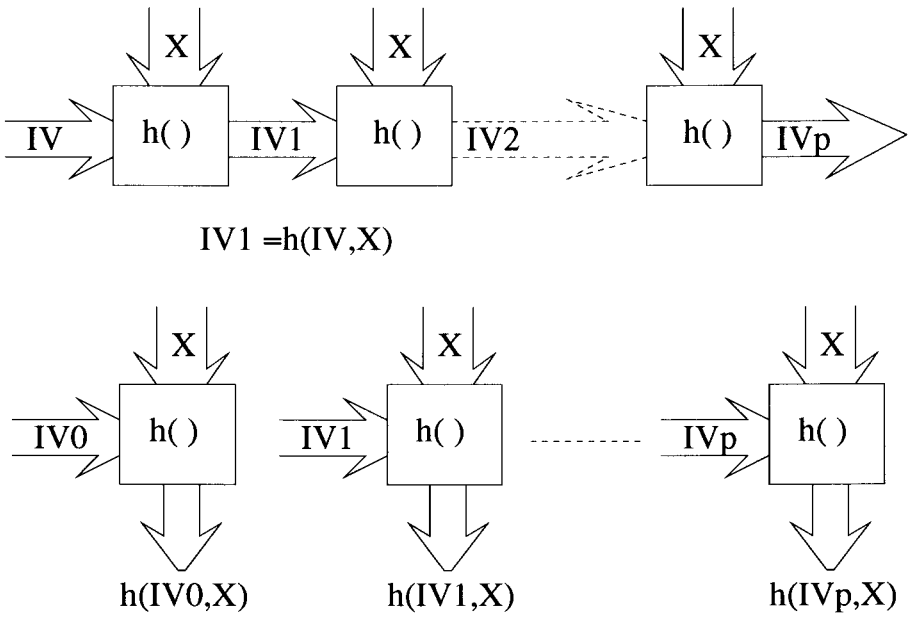

Figure 3.3: Two P-Fold Hashing Schemes

It has been shown in [24] and [25] that the meet in the middle attack can be extended to break these schemes. The extension of the meet in the middle attack to attack p-fold schemes is called the generalised meet in the middle attack. For this attack only $O(10^p \cdot 2^{\frac{n}{2}})$ operations are required instead of $O(2^{\frac{pn}{2}})$ [24], [25].

This attack can be foiled by choosing the number of bits, n, large enough in order to make the attack computationally infeasible.

Correcting Block Attack

Several variants of the correcting block attack exists. The first variant assumes that an attacker has a message X for which a forgery, X', has to be constructed. All the blocks in X' are then changed so that they differ from X. One message block in X', X'_i is then constructed so that $h(X) = h(X')$. The block X'_i is then designated as the correcting block. The correcting block is usually inserted as the last block in the message, but may be inserted at the beginning of a message or in the middle of a message. For this variant of the correcting block attack, the construction of the correcting block X'_i may be accomplished with the random attack. Since a specific hash value has to be generated, the birthday attack cannot be used. If two correcting blocks are allowed, it is possible to use the meet in the middle attack to generate two blocks, X'_i and X'_{i+1}, that together cancel the effect that message blocks X'_0 to X'_{i-1} have on the chaining variable. Another alternative in constructing the correcting block requires the attacker to have knowledge of the algorithm. By manipulating the algorithm a correcting block X'_i can be constructed. Note that the construction of a message block by manipulating the algorithm depends on the algorithm. If the construction of X'_i is independent of the algorithm, the amount of work required is $O(2^{\frac{n}{2}})$.

Another variant of the correcting block attack is described next. An attacker generates two messages X and X' and then generates two correcting blocks Y and Y'. The correcting blocks are then concatenated to X and X'. The correcting blocks Y and Y' should be chosen such that $h(IV, X||Y) = h(IV, X'||Y')$. If the final hash value is specified, the correcting blocks Y and Y' can be constructed using the random attack. If the final hash value is not specified, the birthday attack can be used to generate a collision. If more than one block is allowed as a correcting block, the meet in the middle attack can be used. Note that with the meet in the middle attack the attacker can generate a specific hash value. It is also possible to construct the correcting blocks Y and Y' by manipulating the algorithm. The attack then

depends on an analytical weakness in the algorithm.

It is possible to defend against block correcting attacks by adding redundancy to the message before hashing. Redundancy includes padding rules and attaching the number of blocks or bits in a message as the last block. If the attacker is the originator of the message these measures are not sufficient, since the attacker can control the number of blocks or the length of the message under consideration. Choosing the value of n large enough, makes the block correcting attack computationally infeasible. If the hash function itself can be manipulated to produce a correcting block, the attack becomes dependent on the algorithm used. The algorithm should be replaced if that is the case.

Fixed Point Attack

A fixed point may be defined as a hash value for which $h(X_i, H_{i-1}) = H_{i-1}$. This property allows an attacker to insert an arbitrary number of blocks corresponding to X_i after the first occurrence of X_i (see Figure 3.4).

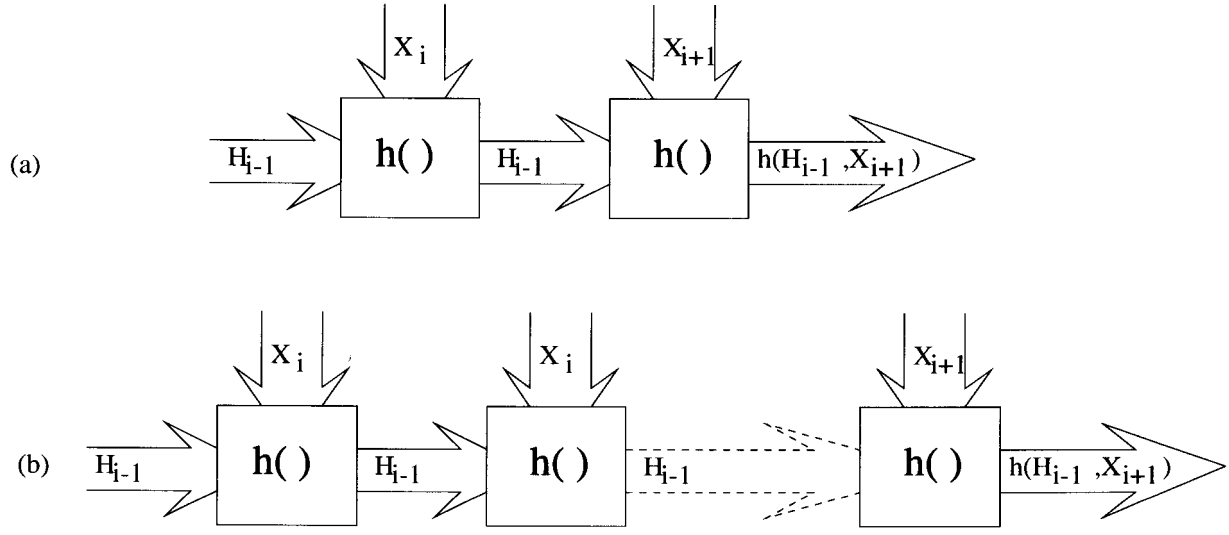

Figure 3.4: The fixed point attack

A collision can be established if the chaining variables can be set equal to H_{i-1}. This can be accomplished by using the random attack to find a suitable value for X_{i-1} or the meet in the middle attack which would allow the attacker to specify X_{i-1} and X_{i-2} so that H_{i-1} can be established. It might also be possible to manipulate the hash algorithm to find a suitable

value for X_{i-1} that will produce H_{i-1}. It is possible to foil this attack by adding redundancy to the message. The redundancy should contain the number blocks in the message. The fixed point can be found by the random attack, or by the meet in the middle attack. This implies that the work factor is approximately $O(2^{\frac{n}{2}})$. By choosing n sufficiently large, it becomes computationally infeasible to perform the random attack or the meet in the middle attack. The value of an attack in which one block is repeated a number of times and still yields the same hash code is debatable.

Differential Attacks

Differential cryptanalysis is based on the study of the relationship between input and output differences in iterated cryptographic algorithms. Since hash functions are usually based on iterated algorithms, differential cryptanalysis is applicable to cryptographic hash functions. The differential attack against hash functions is a probabilistic attack. An attacker searches for input differences that will result in specific output differences. If an attacker intends to create a collision the output difference should be zero.

The differences can be found with a probabilistic search (random attack). For a random attack, IV, $h(IV, X)$ and IV' are specified. The attacker has to find X' so that $h(IV, X) = h(IV', X')$. Another technique that an attacker could use is the birthday attack. For the birthday attack it is assumed that IV and IV' are specified. It now remains to find values for X and X' such that $h(IV, X) = h(IV', X')$ with the birthday attack as described in Section 3.4.1. It is also possible to use the meet in the middle attack to establish a collision. This requires that the attacker should be able to choose two message blocks in both the original and the forged message. This places the magnitude of the work factor at $O(2^{\frac{n}{2}})$. It is also possible to execute the differential attack by manipulating the hash algorithm. The differential attack then becomes dependent on the algorithm. When block ciphers are used, the differences should be chosen in such a way as to exploit the chaining.

Differential attacks can be launched against the chaining of an iterated hash Function, or against the hash algorithm itself. In Chapters 6, 7, 9 and 10 it is shown how differential analysis is employed against the compress function of dedicated hash functions such as MD4, MD5, SHA and HAVAL.

3.5 ATTACKS ON MACS

All of the attacks against MDCs are applicable to MACs if the key for the MAC is known to the attacker. In addition to the attacks on MDCs, the following attacks are applicable to MACs.

3.5.1 Key Collisions

A key collision occurs when, for two distinct keys, K_1 and K_2, $h(X, K_1) = h(X, K_2)$. Due to the presence of a key, this attack is applicable to MACs. MACs based on block ciphers are especially vulnerable to this attack, since this phenomenon has been observed in block ciphers [20], [21]. It was shown in [20] that this attack can be implemented against the Data Encryption Standard (DES) using the meet in the middle attack. In [21] a refined technique based on the theory of distinguished points are proposed. Both attacks resulted in the discovery of key collisions for DES. This attack allows an attacker to construct a message that yields the same hash value for different keys used. For DES, 21 collisions were found in [21] and 48 collisions are described in [20]. When the meet in the middle or birthday attacks are used, approximately $O(2^{\frac{n}{2}})$ operations are required (n is the number of bits in the hash value). For these attacks a large storage space and an efficient sorting algorithm is required. In [21] a technique is suggested that reduces the storage requirements and eliminates the necessity of an efficient sorting algorithm.

Another block cipher with a large number of known key collisions is LOKI. It is known that 15 key collisions exist for every key in LOKI [26]. For this reason it is advised that LOKI is not used as a round function for the construction of a MDC or MAC [27]. For a MDC the birthday attack or the meet in the middle attack can be used. For a MAC a key collision can be established.

3.5.2 Exhaustive Key Search

Exhaustive key search is intended to recover the key for a MAC. For a given MAC both the hash value, $h(K, X)$, and the message, X, are known. The key is recovered by exhaustively trying all possible keys, K_i, until a key, K, is found that results in $h(K, X)$. If key collisions exist for the hash function, $h()$, several messages X_i and their corresponding hash values,

$h(K, X_i)$, are required to confirm that K is a valid key. Therefore this attack is effectively a known plaintext-MAC attack. The effort required to find a k-bit binary key is on the order of $O(2^k)$. In [3] it is stated that for a k-bit key and a n-bit hash value the number of plaintext-MAC pairs, M, required to determine the key uniquely is slightly larger than $\frac{k}{n}$, provided that no key collisions occur.

3.5.3 Chosen Text Attacks

This attack allows the generation of a forgery and, in general, does not allow key recovery. The attack is described [28] and [29]. The attack requires that the MAC is based on an iterative structure (see Chapter 5 Section 5.3). The basic idea is that for two messages, X_1 and X_2, with:

$$\begin{aligned} X_1 &= a_1||b. \\ X_2 &= a_2||b. \\ a_1 &\neq a_2. \end{aligned}$$

The MAC results, $h(K, X_1) = h(K, X_2)$, are likely to occur given $O(2^{\frac{n}{2}})$ chosen messages, with n the hash length. This is reminiscent of the birthday attack. Given $h(K, X_1) = h(K, X_2)$ it is expected that $h(K, a_1) = h(K, a_2)$. This implies that for an arbitrary string c, the MAC values $h(K, a_1||c)$ and $h(K, a_2||c)$ are equal. Thus, a forgery can be obtained by requesting $h(K, a_1||c)$, and in effect obtaining the MAC for $h(K, a_2||c)$.

This attack imposes requirements on the storage space available to an attacker and also assumes that the secret key, K, is not changed before $O(2^{\frac{n}{2}})$ chosen messages can be requested. Note that a forgery is obtained without retrieving the secret key K.

3.5.4 Known and Chosen Text Attack

This attack is presented by Preneel and van Oorschot in [29]. It is viewed as an extension of the chosen text attack described above. The attack is outlined as a proof in [29]. Note that this attack, as is the case for the chosen text attack, generally enables forgery attacks, not key retrieval.

The attack states that r known plaintexts are required with $r = \sqrt{2} \cdot 2^{\frac{n}{2}}$. For the previous

attack it was assumed that the MAC was given by a permutation $g()$ of the hash result, $h()$ (e.g. the unity mapping). For this attack it is assumed that a random mapping is used with the resulting MAC having m bits instead of n bits with $m < n$. When considering this attack, it is necessary to differentiate between a collision before the random mapping is applied, (internal collision) and a collision after the random mapping is applied (external collision). With $g()$ being a random mapping function, 2^{n-m} external collision are expected. Additional computations are now required to generate a verifiable forgery. An internal collision can be identified by attaching a known string y to each collision pair and checking whether the corresponding MACs are equal. This requires $2 \cdot (1 + 2^{n-m})$ chosen text-MAC requests. For internal collisions the resulting MACs are always equal. Retain only the text-MAC pairs resulting in internal collisions. At this stage 2^{n-2m} external collisions and a single internal collision should be available. If more than one external collision remains, these external collisions have to be eliminated by choosing a different value for y and proceed as before, until a single external collision remains. It is believed to be highly probable that the remaining external collision is the result of an internal collision. It is expected that the number of chosen and known texts required to find an internal collision are:

$$\frac{2 \cdot 2^{n-m} \cdot 2^m}{2^m - 1} + 2\left\lceil \frac{n}{m} \right\rceil \approx 2 \cdot 2^{n-m} + 2\left\lceil \frac{n}{m} \right\rceil .$$

Finding a single internal collision that allows the generation of a MAC forgery requires:

$$\begin{aligned} \text{Known text-MAC pairs} &= \sqrt{2} \cdot 2^{\frac{n}{2}} \\ \text{Chosen text-MAC pairs} &= 2 \cdot 2^{n-m} + 2\left\lceil \frac{n}{m} \right\rceil . \end{aligned}$$

This attack is extended to cover the case where all the known texts have a common sequence of s trailing blocks. It is stated that if that is the case, fewer known and chosen texts are required. In particular it is shown that if:

$$r^2 \cdot s = O(2^n)$$

the probability that the set of known messages, r, contains two messages that collide under the hash function, $h()$, is approximately:

$$1 - e^{\left(-\frac{r^2 \cdot (s+1)}{2^{n+1}}\right)}.$$

Given this probability, an internal collision for $h()$ can be found if the known text-MAC pairs have s identical trailing blocks. If $g()$ is a random mapping, this attack requires:

$$\text{Known text-MAC pairs} = \sqrt{\frac{2}{s+1}} \cdot 2^{\frac{n}{2}}$$

$$\text{Chosen text-MAC pairs} = 2 \cdot \frac{2^{n-m}}{s+1} + 2 \left\lceil \frac{n - \log_2(s+1)}{m} \right\rceil .$$

In [29] both these attacks are applied to MAA [30] and CBC-MAC (see Chapter 5 Section 5.4). The refined version of this attack with s-blocks is shown to be effective against MAA. The refined attack cannot be applied to CBC-MAC with maximal feedback, since the round function is bijective. The unrefined attack described initially can however be employed against CBC-MAC.

In [31] these attacks are extended to and tailored for forgery and key recovery for MAA and the envelope MAC constructions (see Chapter 5 Section A.5).

It is noted that a large number of chosen texts and known texts along with their MAC values are required. If it is assumed that key collisions does not occur for the chosen MAC, the secret key, K, may not change while collecting the known and chosen texts. It is therefore suggested that the chosen text attacks, and known and chosen text attacks, described in this and the previous section can both be foiled by effective key management. By changing the secret key at a regular interval, it becomes impossible to collect enough chosen and known texts to execute these forgery attacks.

3.6 ATTACKS ON UNDERLYING BLOCK CIPHERS

It has been suggested that block ciphers could be adapted for use as building blocks for cryptographic hash functions [23], [32], [3]. The motivations for these suggestions are presented in Chapter 5. It should be noted that when using block ciphers in a hash function configuration, additional attacks based on the underlying block cipher are possible. The attacks discussed in Sections 3.4 and 3.5 are applicable to hash functions derived from block ciphers. Specifically the construction of fixed points is considered easy if the underlying block cipher is either DES or LOKI [3].

3.6.1 Complementation Property

Symmetry under complementation was one of the first properties discovered for DES [33]. Let $E(K, X)$ denote the encryption of X with the key K. Then for a message, X, and a key, K, the complementation property is stated as follows:

$$\forall K, X : C = E(K, X) \Leftrightarrow \overline{C} = E(\overline{K}, \overline{X})$$

with C the ciphertext and $\overline{C}$ the complement of C. This property is known to exist for DES and LOKI91 [34]. When used in a hash function construction as a MAC, this reduces the effort required for exhaustive search by a factor of two. In addition, this property allows the construction of trivial collisions. It is known that LOKI89 has a large set of keys for which this property holds [34].

3.6.2 Weak Keys

A weak key is a key for which the following property holds:

$$\begin{aligned} E(K, X) &= C \\ E(K, C) &= X, \quad \forall X. \end{aligned}$$

Thus the encryption, E, and decryption, D, operations are equivalent for a given key K. Thus for certain keys, some block ciphers are involutions. This property holds for certain keys of DES, LOKI89, LOKI91 [34] [35] and IDEA [36].

For a semi-weak key the following property holds:

$$\begin{aligned} E(K_1, X) &= C \\ E(K_2, C) &= X, \quad \forall X. \\ K_1 &\neq K_2. \end{aligned}$$

These properties of block ciphers can be exploited in certain hash functions based on blocks ciphers to yield fixed points (see Section 3.4.2).

3.7 HIGH LEVEL ATTACKS

In addition to the above attacks, the following attacks are considered feasible. These attacks are not so much an attack on the hash algorithm, than an attack on the interaction of the hash algorithm with the environment in which it is used.

3.7.1 Differential Fault Analysis

In [37] an attack on public key systems implemented in hardware is described by Boneh, DeMillo and Lipton from the Math and Cryptography Research Group, Bellcore. Based on the attack presented in [37], Biham and Shamir describes an attack that retrieves the key for a hardware implementation of DES [38]. This attack is termed differential fault analysis (DFA). Both of these attacks are specifically applicable to algorithms implemented in hardware, and for this reason are considered as high level attacks.

The attacks described in [38] exploit the effect of a transient error in a hardware device. The resulting erroneous output is then analysed to determine the secret key. In [38] it is claimed that less than 200 ciphertexts are required to find the last sub-key in a DES implementation. The remaining eight key bits can be found by exhaustive search.

In [39] Quisquater claims that this technique is applicable to MACs implemented in hardware. According to [37] attacks based on differential fault analysis can be countered by verifying results before output and protecting the registers used to store values using error correcting codes. Thus differential fault analysis imposes conditions on the implementation of a hash algorithm, rather than on the design of a hash algorithm.

3.7.2 Differential Power Analysis

Differential power analysis was first proposed by Kocher []. This attack allows an attacker to derive the secret key used by an algorithm. This is accomplished by observing the fluctuations in power consumption of the device, while performing cryptographic operations. It is a non-destructive attack. This type of attack is especially efficient against smart card implementations. It has been demonstrated that a secret key can be obtained using this approach. This attack can be used against MACs implemented in hardware.

3.7.3 Attack on the Interaction with the Signature Scheme

As remarked in Chapter 1 hash functions are often used in digital signature schemes. It has been shown in [40], that even if the hash function is a collision resistant hash function, the signature scheme can be attacked successfully. The success of this attack is due to the underlying multiplicative structure of both the hash function and the signature scheme [3].

3.7.4 Attacks on the Protocol

These attacks are concerned with attacks such as replay of messages and the construction of valid messages from previously intercepted messages. It is thus in effect an attack on the protocol. These attacks can be thwarted by the use of a nonce or a timestamp. It is thus necessary to protect a system in which messages and hash values are vulnerable to interception with a suitable protocol.

Equivalent Systems

The calculation of the equivalent shift register length for a given stream Cipher, allows an attacker to construct a linear feedback shift register that mimics the operation of the stream cipher. This calculation is based on the Massey-Berlekamp algorithm. Similarly an attacker could attempt to construct an equivalent system for a cryptographic hash function. This attack is applicable to MACs in particular [3]. The attacker attempts to find a system that produces the same MAC for a given message, without knowledge of the secret key [3]. No standard technique is known to exist for mimicking hash functions.

3.7.5 Attacks Dependant on the Algorithm

These attacks exploit a weakness in the algorithm. These attacks are usually discovered only after the publication of a hash algorithm. These attacks are only a threat if the work factor for finding a collision is substantially less than $O(2^{\frac{n}{2}})$. This dissertation investigates attacks of this nature.

3.8 ATTACKERS AND ATTACKS

This section relates the attacks the various classes of attackers are capable of.

The distinction between classes of attackers are based on their capabilities, the information at their disposal and their position with regard to the system under consideration. The difference with regard to Class I and Class II attackers lie, to a large degree, within the information at their disposal. The advantage that Class II attackers have over Class I attackers can be eradicated by publishing the hash algorithm and its design criteria.

All three classes of attackers are capable of executing any of the attacks presented in this chapter. The probability of success however, increases as the attacker's capabilities increases. The feasibility of the attacks presented in this chapter are measured in terms of effort required to establish a collision for a hash function. In general Class III attackers have the largest resources in terms of computing power, followed by Class II and then Class I attackers.

A further point of interest is the position of the class of attacker with respect to the system under consideration. A legitimate participant can construct two messages that result in the same hash value. One message can be signed and transmitted and at a later stage the messages can be swapped. An active eavesdropper is not allowed to construct and sign messages. Active eavesdroppers are expected to intercept and modify messages. The eavesdropper is restricted to finding messages that hash to specific values. As seen in Section 3.3 the computational effort required to construct two messages that result in the same hash value is considerably less than that required to hit a specific hash value. For example, even though a Class II attacker might not be capable to construct a message that results in a specific hash value, it may be possible for the Class II attacker to construct two messages that yield the same hash value.

Thus although all three classes of attackers are capable of all possible attacks, the probability of success differs substantially as the attackers' knowledge and capabilities differs. In addition it appears that legitimate participants require less effort to construct messages that result in collisions, than active eavesdroppers.

3.9 FEASIBILITY

All of the above attacks pose requirements in terms of processing power and storage space. When designing a hash function the parameters that contribute the security of the function, such as the hash length, should be chosen in such a manner as to render any of the above attacks infeasible. In order to make a sufficiently informed choice for these parameters the capabilities of an opponent has to be known or estimated. Estimating the computational power of an opponent is a complicated process. The following aspects should be considered when estimating computational capabilities.

1. Processor speed.
2. Volatile memory access time.
3. Amount of volatile memory available.
4. Storage space access time.
5. Storage space available.
6. System bandwidth.

In addition to these aspects, additional factors, such as the opponent's ability to construct dedicated hardware or hardware subsystems and the interconnection of these systems to realise the above attacks, should be taken into account. The feasibility of an attack also depends on the class of attacker dealt with (see Section 3.2).

The generic attacks described in this chapter are not considered feasible for hash lengths of 128-bits or more. It is already considered that 128 bits will not provide sufficient protection within the next few years, due to the increase in available computational power [41]. The development of new fields in computing such as quantum computing may also change the estimate of computationally secure hash lengths [42].

3.10 CONCLUSION

A taxonomy of possible attackers were presented. The taxonomy is based on the attackers' capabilities and their position with regard to the system they seek to attack.

A review of the general attacks against MDCs were presented in Section 3.4. These attacks are discussed in [17]. It was shown that these attacks require a maximum of $O(2^n)$ operations and a minimum of $O(2^{\frac{n}{2}})$. Due to the definition of hash functions, these attacks cannot be avoided. When designing a hash function, the relevant parameters should be chosen to minimise the effect of these attacks. The feasibility of a specific attack is measured by the workload associated with each of these attacks. The workload is expressed in terms of the number of bits, n, contained in the hash values. This implies that an attack can be made infeasible by choosing n sufficiently large.

In addition to the general attacks on MDCs, the general attacks on MACs were presented in Section 3.5. These attacks are aimed at either retrieving the secret key, or obtaining a forgery for the MAC. The probability of success for these techniques are proportional to the size of the secret key and the number of bits, n, in the resulting hash function.

When constructing cryptographic hash functions based on block ciphers, additional attacks are possible. These additional attacks were presented in Section 3.6. These attacks exploit certain properties of block ciphers and allows the establishment of collisions. As before, a legitimate participant has a larger probability of success than an active eavesdropper.

Several high level attacks were presented. These attacks concentrate on the environment in which a hash function is used. Hash functions and the systems within which they are used, should be implemented in a secure manner to avoid these attacks.

The relationship between attackers and the type of attacks they are capable of were investigated in Section 3.8. It is shown that an attacker operating as a legitimate participant has a larger probability of success than an attacker operating as an active eavesdropper. Therefore legitimate participants poses a more significant threat than active eavesdroppers. Likewise, it is more likely that a Class III attacker will execute an attack successfully than a Class II or Class I attacker.

When designing a cryptographic hash function, both the attackers and the attacks they are capable of should be considered. Cryptographic hash functions should be designed to withstand the attacks described in this chapter.

CHAPTER 4: REQUIREMENTS FOR CRYPTOGRAPHIC HASH FUNCTIONS

4.1 INTRODUCTION

This chapter contains a description of the requirements for cryptographic hash functions. These requirements are based on:

1. The definition of a cryptographic hash function.
2. Known attacks on cryptographic hash functions.

The requirements are divided into two classes, namely functional requirements and security requirements [43]. These requirements are often contradictory. The contradictions are not restricted to the security requirements and the functional requirements, but are sometimes found in the functional or security requirements themselves. The conflict between security and functional requirements are treated in a separate section in this chapter.

4.2 FUNCTIONAL REQUIREMENTS

The functional requirements deals with the practical implementation of a hash function. The requirements presented in this section are intended as goals to be met by practical hash algorithms.

4.2.1 Message Reduction

According to the definition of a hash function a message of arbitrary length is compressed to a string of fixed length. Thus a fundamental functional requirement for a cryptographic hash function is the reduction of a message of arbitrary length to a string of fixed length.

Concerning the input length of the message, several solutions are possible. The designer could specify different hash algorithms for different message lengths. This is an impractical solution. Alternatively a scalable hash algorithm can be used. The third possibility is the use

of padding to extend the message to a specific length, or a multiple of a specific length. This is the most often used technique.

If the message is longer than the elementary block length of the hash algorithm, the message is padded to be a multiple of the elementary block length. The padded message is then segmented and processed iteratively by the hash algorithm. This process is known as chaining and is commonly used in hash functions (see Chapter 5 Section 5.3).

4.2.2 Repeatability

Repeatability is important since it should be possible to produce the same hash value for the same message.

In Chapter 5 an ideal construction is presented for a cryptographic hash function. In this construction the hash value is generated independent from the message. The construction requires the use of a database to produce the same hash value for the same message. The construction is impractical due to the storage requirements, accessibility of such a database and the difficulty of constructing binary symmetric sources. Attaining repeatability through the use of a database is therefore infeasible.

Repeatability can be achieved in a practical hash function by making the hash value dependent on the message. Thus possession of the message implies possession of the hash value.

4.2.3 Data Type Independence

The hash function should not be dependent on the type of data to be processed. In other words, a specific file type should be processed in the same manner by the given hash function as any other file (e.g. a binary executable should be processed in the same manner as an ASCII file). A hash function should therefore not be designed to process a specific data type, especially if the hash function is intended for widespread use.

4.2.4 Fast Calculation

The fast calculation of hash values for messages is an important requirement. When using a digital signature scheme the hash value instead of the message is signed. For this reason it should be faster to compute and sign the hash value than to sign the entire message. The reduction in time and bandwidth requirements are important motivations for using hash functions.

The speed of a hash function can be increased by efficient implementation or efficient design. A fast hash function can be designed by either optimising the algorithm for a specific computer architecture or by simplifying the design to reduce the number of operations required by the hash algorithm.

4.2.5 One Pass per Message

This requirement specifies that the message should be processed only once. This effectively rules out all of the so-called p-fold schemes. Thus a message is loaded into memory only once. When computing online this has the advantage that no permanent storage is required. When computing the hash value for a file on a computer disk drive, the time required to access the file and load it into memory more than once is eliminated. This requirement augments the requirement for fast calculation.

4.2.6 Minimum of Secret Information

This requirement is the main reason why MDCs are preferred over MACs. A MAC requires that a secret key is shared between two or more users. This introduces the problem of key management. It also serves as an argument against the use of MACs in an environment where the issue of key management poses a problem.

4.2.7 Modular Design

The hash function should be designed to be modular. This allows the hash algorithm to be replaced within a system if a weakness or deficiency in the algorithm is detected after implementation in a system. Implicit in this requirement is the need for a well defined

interface between the hash function and the system it is used in. The interface components of importance are:

1. Input block size.
2. Output hash value.
3. Key size (MACs only).

4.2.8 Ease of Implementation

If the hash algorithm is intended for widespread use, ease of implementation is important. This requirement ensures the proliferation of the algorithm, since many unskilled users can implement and use the algorithm independently. It is also important to supply sufficient test data for an independent user to verify the operation of the algorithm.

4.2.9 Machine Independence

An algorithm optimised for a specific architecture is advantageous in terms of speed and ease of implementation for the architecture concerned [10], [44], [45]. It is, however, possible that a penalty is paid when transferring the algorithm to another architecture. The penalty may be paid in both a loss of speed and increased complexity of implementation.

4.2.10 Distribution and Obtainability

It is important that a hash function's algorithm, test data, documentation and implementation should be easily obtainable if intended for use in the public domain.

4.3 SECURITY REQUIREMENTS

The security requirements for cryptographic hash functions deals with those properties of a hash function that influences the security of the hash function.

4.3.1 Confusion and Diffusion

The concepts of confusion and diffusion in cryptography were first introduced by Shannon [46].

Confusion is described as: *"The method of confusion is to make the relation between the simple statistics of the ciphertext and the simple description of the key a very complex and involved one"*.

Diffusion is described as: *"In the method of diffusion the statistical structure of the plaintext which leads to its redundancy is dissipated into long range statistics."* These concepts are interpreted in [1] as follows:

Confusion: The ciphertext statistics should depend on the plaintext statistics in a manner too complicated to be exploited by the cryptanalyst.

Diffusion: Each digit of the plaintext and each digit of the secret key should influence many digits of the ciphertext.

The concepts of confusion and diffusion can be applied to hash functions. For MDCs a secret key is not required. Thus the security of the MDC depends solely on the concept of diffusion. In other words, there should be no apparent relationship between the input to the MDC and the resulting output. For MACs a secret key is required, thus in addition to the requirement of diffusion the concept of confusion is considered relevant. Thus, for a MAC, a person in possession of the MAC algorithm, the message, X, and the MAC result, it should be difficult to determine the key K, used to determine the MAC result.

4.3.2 Message and Hash Value Independence

For the ideal hash function construction presented in Chapter 5 a binary symmetric source is used to generate the hash function. Thus there is no dependence between the hash value and the message. In terms of hash functions this is the most secure a hash function could be, since it is impossible to manipulate a message to yield a specific hash value. The ideal hash function construction presented in Chapter 5 is impractical due to the requirement for repeatability. In practical hash functions the hash value depends on the message. For a secure

hash function it is required that there is no apparent or predictable relationship between the hash value and the message. This requirement is related to the concept of diffusion.

4.3.3 Computational Feasibility

When designing a cryptographic hash function the computational feasibility of the known generic attacks should be considered. It should be computationally infeasible to employ any of the generic attacks described in Chapter 3 to construct a collision. It should not be possible to construct a collision in less than $O(2^{\frac{n}{2}})$ operations (n being the hash length). Several degrees of computational infeasibility is defined.

Collision Resistance: This requirement states that it should be computationally infeasible to construct two arbitrary messages, X and X', such that $h(X) = h(X')$. Hash functions that meet this criterion are called collision resistant hash functions (CRHF).

Finding a Specific Hash Value: This requirement states that, given a message, X, and the corresponding hash value $h(X)$ it is computationally infeasible to find a second message, X', such that $h(X) = h(X')$. A hash function that satisfies this criterion is said to be a one way hash function (OWHF). The condition for a OWHF is weaker than the condition for a CRHF.

Sensical Collisions: A sensical collision is a collision for which X and X' can be constructed such that $h(X) = h(X')$, and X and X' are meaningful in the context used. This is the weakest condition imposed on cryptographic hash functions.

It is recommended that a cryptographic hash function is designed to meet the requirements set for a collision resistant hash function.

4.3.4 Interaction with other Algorithms

Supplementary to the requirements of computational infeasibility of finding collisions for a hash function is the requirement for the computational infeasibility of finding a combination of the hash function and the signature scheme that results in a forged signature.

4.3.5 MDC Hash Space

Central to the computational feasibility of finding a collision for a hash function lies the hash space. The generic attacks described in Chapter 3 are evaluated in terms of the order of the number of operations required to establish a collision. None of the generic techniques requires less than $O(2^{\frac{n}{2}})$ operations. The hash length n should be chosen large enough to render the generic attacks harmless, given limited time and resources. Current estimates recommend that $n \geq 128$ bits, and preferably $n = 160$ bits. The length of the hash value should be updated every 3-5 years to accommodate the increase in computational power and advances in computing technology.

4.3.6 MAC Key and Hash Space

The matter of key space is applicable to MACs. A MAC key should be chosen long enough in order to prohibit exhaustive key search. For a k-bit key approximately $O(2^k)$ operations are required to recover a key through exhaustive search. It is proposed that the length of the key should be at least 64 bits.

When considering the hash length of a MAC two factors should be kept in mind. First is the use of a secret key. The secret key adds to the security of the MAC, and consequently the requirements imposed on the hash length is reduced to half of that specified for a MDC with a similar level of security (see Chapter 2). The reduction in the hash space reduces the effort required by an opponent who is in possession of the secret key, K, to find a collision or establish a forgery. Thus for a MAC the security is derived from the difficulty of retrieving the key rather than the difficulty of finding a collision. Thus a MDC with hash length n would afford the same level of security as a MAC with a key of length $\frac{n}{2}$. The second fact which should influence the choice of a MAC length is the results obtained from the attack in [29]. In [29] it is shown that the workload for generating a collision doubles if the length of the MAC value, m is less than the length of the chaining variable, n. The mapping from n to m should appear random. It is proposed that the chaining variables should be at least 128-bits with the final MAC value chosen as 64 bits.

The importance of key management should be kept in mind when designing a system that makes use of a MAC.

4.3.7 Message Dependence

Due to the functional requirement for repeatability, the hash value depends on the message. If the hash value depends on the message, it is important that the hash value depends on every bit in the message. If this is not the case an attacker could easily manipulate the bits not used in the computation of the hash value to produce another message that yields the same hash value.

4.3.8 One-Wayness

It should be computationally infeasible to reconstruct a message from its hash value. This is both a functional and a security requirement. From the point of view of security requirements, the construction of a message from its hash value allows an attacker to construct the message even if the message is encrypted. The requirement for one-wayness can be wavered if both the hash value and the message is encrypted. This solution incurs a time penalty, due to the additional encryption required.

4.3.9 Error Extension

A cryptographic hash algorithm should exhibit maximum error extension. This implies that if one bit is changed in the message, approximately half the bits in the hash value should change. This ensures that an attacker can not expect a collision close to a specific message with a high probability. This implies that the attacker has to search the entire hash space. This is similar to the requirements set for a block cipher.

4.3.10 Distribution of Preimages

The output values and preimages of a hash function should be distributed smoothly. This condition is required to prevent an attacker from searching for those preimages which are known to occur more frequently, thus reducing the effort required for finding a collision. When a hash function based on the Damgård-Merkle scheme is used, the preimages and output values of every block should be distributed smoothly to minimise the possibility of a successful attack on the chaining (see Chapter 3 Section 3.4.2).

4.3.11 Decomposable Algorithms

In [43] it is proposed that a hash algorithm should not be a decomposable algorithm. This would prevent an attacker from analysing individual building blocks to construct a collision. This requirement is intended to prevent attacks dependent on the algorithm.

4.3.12 Conditions on Chaining

If the Damgård-Merkle scheme is used every bit in the chaining variables should be used in the next iteration of the hash algorithm. This ensures that the applicability of the meet in the middle attack is minimised. An additional condition on the chaining variables is imposed by the meet in the middle attack. The number of operations required for the meet in the middle attack are $O(2^{\frac{n}{2}})$, with n the number of bits in the chaining variable. Consequently the chaining variable should have at least the same length as the hash value. For ease of implementation the chaining variable is often chosen the same as the hash value. The number of bits, n, should be at least 128 bits.

4.3.13 Redundancy

Redundancy could be added to the message to prevent certain variants of block correcting and fixed point attacks. Redundancy is especially useful for detecting the addition of message blocks. Redundancy appended to messages include:

1. Time-stamps.
2. Message length.
3. Intermediate value or chaining variable.

Time-Stamps

The use of a time-stamp has the advantage that it prevents replay attacks. The addition of time-stamps introduces several problems and potential weaknesses which could result in a successful high level attack. The use of time-stamps require a synchronised clock. Synchronised clocks are expensive to implement over distributed networks. The use of time-stamps

require a timing window to allow for transition times, especially when used in distributed systems. As the timing window increases, the possibility of a successful replay attack increases. If the timing window becomes too small, synchronisation becomes a problem. An additional risk is introduced when utilising a small timing window. A small timing window implies that the grain of the time-stamp becomes small. A time-stamp with a fine grain increases the probability that an attacker could find two messages with a given time stamp that results in a collision. Once such a pair is generated, the attacker waits for the time specified by the time-stamp before sending the messages.

Message Length

The addition of the message length to the message has the advantage that the message length can not be increased or decreased. When using the Damgård-Merkle construction, padding is required to extend the length of the message to a multiple of the elementary block length. The padding scheme should be designed to contain sufficient redundancy to prevent the addition of blocks.

Intermediate Value

The addition of the message length to the message before hashing makes it easy to detect if a segment of the message was added or deleted. It does not allow the user to detect if a segment of a message was replaced with a different segment. Substitution of message segments can be detected by adding an intermediate result to the message. When using the Damgård-Merkle construction, a chaining variable could be appended to the message before hashing.

4.4 FUNCTIONAL VS. SECURITY REQUIREMENTS

Several of the above requirements for cryptographical hash functions are contradictory. In this section a short review is given of the contradictions. Where applicable resolution strategies for these conflicting requirements are suggested.

4.4.1 Repeatability and Security

This contradiction stems from the definition of an ideal cryptographic hash function. For practical purposes the repeatability in a hash function can not be achieved through the use of a database as is the case with the ideal cryptographic hash function. Instead, repeatability is obtained by deriving the hash value from the message. This implies that the hash value depends on the message and not on a binary symmetric source. It is therefore possible to manipulate the message to yield a specific hash value. This is not possible when using a binary symmetric source to generate the random numbers. Thus by making the hash value dependent on the message, repeatability is obtained at the cost of reduced security. This problem can be overcome by using the message as a seed to a good pseudo random number generator. Therefore it is sufficient to state that there should be no apparent relationship between the message and the hash function.

4.4.2 Chaining and Security

In Section 4.2.1 the need for chaining is introduced as a functional requirement. Unfortunately the use of chaining structures make the hash function vulnerable to a variety of attacks, designed specifically to exploit the chaining mechanism (Chapter 3 Section 3.4.2). The attacks on the chaining require at least $O(2^{\frac{n}{2}})$ operations. The alternatives to chaining are considered impractical. If possible the message length should be limited to a single block to avoid chaining and the consequent attacks. If chaining can not be avoided it is advised that the number of bits, n, be chosen such that the most powerful attack on chaining is computationally infeasible. Suitable values for n exceeds 128 bits.

4.4.3 Speed and Security

The requirement for speed is rooted in limited computing resources and CPU power. Hash operations are specifically intended to speed-up digital signature schemes. When designing a hash algorithm for fast execution the designer is often faced with a trade-off between speed of execution and security [3]. The designer should be careful not to increase the security and thereby sacrificing an unacceptable amount of processing time. Likewise the designer should not increase the speed of an algorithm at an excessive cost to the security of the hash function. The designer should decide what level of security and what hashing times are

tolerable. Note that a slow algorithm is not necessarily more secure than a fast algorithm (a badly designed algorithm can be both slow and insecure).

4.4.4 Speed and Machine Independence

A contradiction between the two functional requirements specified in Section 4.2.4 and Section 4.2.9 exist. In order to design a fast hash function without paying severe penalties in terms of security, hash algorithms are optimised with a specific computer architecture in mind [10], [44] [45] and [47]. This is acceptable as long as the majority of the intended users share this architecture. When the specified architecture is not common to the majority of users, penalties are paid in terms of speed and a lack of ease of implementation. If a hash function is designed with a specific architecture in mind, the design should be optimised to make use of the general architecture of the processors or systems concerned (e.g. 32-bit Intel architecture). The design should not require specific instructions available only to certain processors if diverse processors architectures are used (e.g. multiply and accumulate instruction in a DSP is not available in the 80x86 family).

Note that when implementing an algorithm the implementation should be optimised for the architecture used. However, when defining an algorithm it should not be overly optimised for a specific architecture.

4.4.5 Decomposability and Ease of Implementation

In Section 4.3.11 it is stated that a hash function should not be decomposable. This implies that the hash function does not have individual building blocks. This presents a problem in terms of the functional requirement for ease of implementation. If a hash algorithm is not composed from individual building blocks, it becomes difficult to implement the algorithm step-wise and test the functionality of each block. It is proposed that a hash function is constructed from individual building blocks, but that analysis of individual blocks and the interaction of these blocks does not allow the hash function to be manipulated in a deterministic way. This approach allows a user to implement and test each block and its interaction with other building blocks before constructing the entire hash algorithm.

4.4.6 Security and Bandwidth

In Sections 4.3.5, 4.3.6 and 4.3.3 security requirements are set which influences the length of a message and the consequent hash value. These security requirements are formulated to foil the known generic attacks presented in Chapter 3. Most of these attacks can be made computationally infeasible by increasing the number of bits n. The disadvantage of this defence mechanism is that as the computational power of computers increase the number of bits n has to be increased. When the hash values are transmitted over a channel, an increase in n results in an increase in bandwidth required. Thus, unless bandwidth availability grows at the rate of computational power, poorer throughput and reduced performance of communication systems that make use of cryptographically secure hash functions will be the result.

Another defence technique is to add redundancy to the message and then compute the hash value. The addition of redundancy once again implies reduced performance.

If bandwidth is constrained, a designer could decrease the security to improve the throughput of a system. If security is deemed to be of greater concern than bandwidth requirements, the number of bits, n, should be chosen to be secure in terms of computational feasibility.

4.5 CONCLUSION

This chapter contains a description of both functional and security requirements for cryptographic hash functions. The functional requirements deals with the successful implementation and proliferation of cryptographic hash functions. The security requirements stated in this chapter are intended to render the known generic attacks on hash functions harmless.

Many of the requirements mentioned in this chapter are conflicting. Some of these contradictions are discussed in Section 4.4. It is the intention of this discussion to make a designer aware of these contradictions and to suggest strategies to find optimal trade-offs for these contradictions.

Due to the conflicting nature of the requirements for cryptographic hash functions, a designer should consider these requirements as a guideline. The designer should be influenced by the application for which the cryptographic hash function is to be designed and should accordingly make appropriate trade-offs between these requirements.

CHAPTER 5: GENERAL DEDICATED HASH FUNCTION CONSTRUCTIONS

5.1 INTRODUCTION

This chapter introduces the notion of an ideal cryptographic hash function construction. This is an impractical construction and consequently the notion of the iterated cryptographic hash function is introduced. This construction was independently introduced by Dåmgard and Merkle and is commonly used in the construction of dedicated cryptographic hash functions. Dedicated hash function constructions based on this construction include the MD4 family of hash functions.

5.2 IDEAL CRYPTOGRAPHIC HASH FUNCTION

In [48] a construction is proposed for a super or ideal cryptographic hash function. The proposed construction is not dependent on a secret key, but can easily be extended by adding a key dependent element to the construction. In accordance to the definition of a hash function in Chapter 1, an input of variable length results in a hash value of a fixed length of n bits.

The construction consists of a database and a binary symmetric source. The message, X, is submitted to the hash function. The database is searched for the submitted message. If the message is found, the hash value associated with the message, $h(X)$, is presented as an output. If the message is not found in the database, the binary symmetric source generates a binary string of n bits. This binary string is presented as the hash value $h(X)$. The message, X, and the newly generated hash function, $h(X)$, is stored in the database for future use. Thus, the ideal cryptographic hash function is secure in the sense of cryptographic hash functions. A representation of this construction is shown in Figure 5.1.

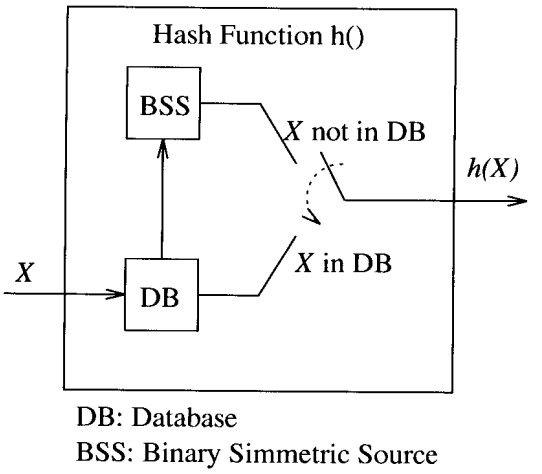

Figure 5.1: Ideal Cryptographic Hash Function

The proposed construction is impractical for the following reasons.

1. An infinite number of messages exists, and consequently infinite storage space is required for the database.

2. Due to the use of binary symmetric source all users should have access to the same database or hash function. This is impractical over large, distributed networks (such as the Internet).

3. A binary symmetric source is difficult to construct.

The construction for an ideal cryptographic hash function is reminiscent of the one time pad or Vernam cipher. Both the one time pad and the ideal cryptographic hash function are considered impractical. The goal of stream cipher design is to simulate certain properties of a one time pad while avoiding those properties which make it impractical. Likewise, when designing cryptographic hash functions, the goal should be to simulate certain properties of the ideal cryptographic hash function while circumventing the properties which are considered impractical.

The important design requirement of apparent independence between the message and the hash value was deduced from this construction (Chapter 4). Various building blocks that facilitate the construction of a hash function that satisfies this requirement, is presented in Section 5.5.

5.3 ITERATED HASH FUNCTIONS

The Damgård-Merkle scheme forms the basis of the majority of known hash functions. This scheme was independently proposed in [22] and [23]. It is an iterated scheme and hash functions constructed according to this scheme are referred to as iterated hash functions [49]. The following three components are identified in the Damgård-Merkle scheme:

1. The segmentation and padding rule.
2. The compress function.
3. The chaining rule.

5.3.1 The Segmentation and Padding Rule

By definition the hash function should hash a message of arbitrary length to a fixed length. The segmentation and padding feature of the Damgård-Merkle scheme allows the hashing of messages of arbitrary length, which is one of the functional requirements.

The segmentation rule is used to divide a message of an arbitrary length into blocks of fixed lengths. No special segmentation rules are known to exist. When segmentation is required the message is simply processed in a serial manner, dividing the message into blocks of a given length. The fixed block length is referred to as the elementary block length. If the message is not a multiple of the elementary block length, padding is required (see Figure 5.2).

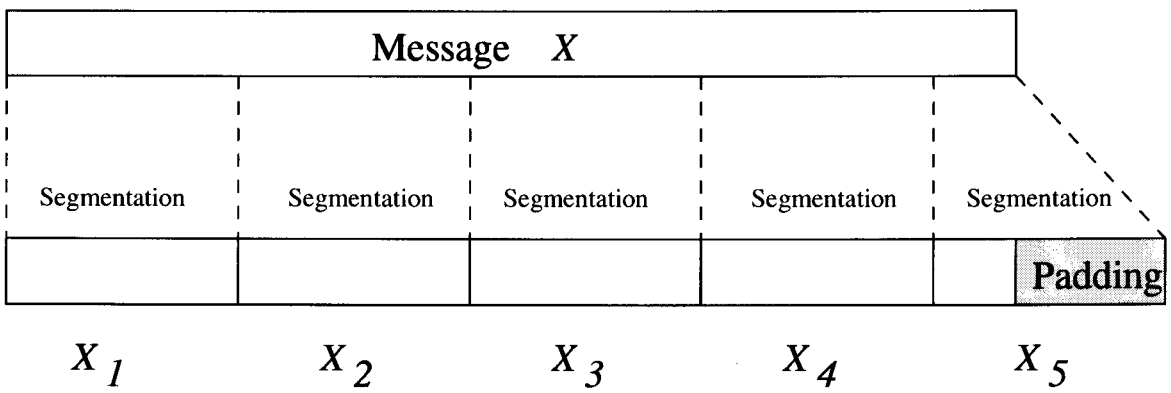

Figure 5.2: Segmentation and Padding

If the message is a multiple of the elementary hash length, padding is not required, but dependent on the applications, it is sometimes applied.

The padding rule is used to expand a message so that the message length is an exact multiple of the elementary block length on which the hash function operates. The padding rule can be used to add additional information to a message (redundancy). The redundancy provides additional security for the hash function against attacks (see Chapter 4). A number of padding rules have been proposed. A summary of these rules follows:

1. Pad the message with 0's until the padded message is a multiple of the block length. This padding rule is ambiguous, since it is not known how many of the trailing zeros are part of the message. This rule requires that either the length of the message be known, or that the message length is included in the message.

2. Pad the message with a single 1 followed, if necessary, by 0's until the padded message is a multiple of the required block length. If the message is a multiple of the required block length before padding commences, a block is added to the message. The additional block contains a single 1 followed by 0's.

3. Let z be a number of zeros and let r be the number of bits required for the binary representation of z. Pad the message with z zeros, except for the last r bits. Let the last r bits contain the binary representation of z. If less than r-bits remains in the last block, additional blocks are added until r can be appended to the zero padded message.

4. Let r be the number of bits required for the binary representation of the message length. Let b be the remaining number of bits in the last message block. Let z be the difference in the number of bits between b and r. Pad the message with z zeros, except for the last r bits. Let the last r bits contain the binary representation of the message length. If less than r-bits remains in the last block, additional blocks are added until r can be appended to the zero padded message.

The choice of padding rule depends on the application. However, padding rule number 4 offers the most security. This rule prevents an attacker from deleting or adding message blocks. If the attacker is an active eavesdropper, the threat posed by attacks such as the block correcting attack and fixed point attacks are minimised. It does however not necessarily prevent a legitimate participant from constructing messages of equal length using the fixed point and block correcting attacks.

In [23] Damgård presents a proof that if the round function, $f()$, is a collision resistant function (CRF), the construction described results in a collision resistant hash function. This proof holds only if the message length is appended to the message before hashing [3]. A variant of padding rule 4 is used for dedicated hash functions such as MD4, MD5, SHA and SHA-1.

For these reasons, padding rule number 4 or a variant thereof, is suggested for use in cryptographic hash functions.

5.3.2 The Compress Function

The second building block is the compress function or round function. The compress function, $f()$, reduces an input block X_i of m bits to a block of n bits. The compression function is the heart of the hash function since this is where the reduction of the message length occurs.

Damgård proved that generating a collision for a hash function based on the iterated scheme requires that either a collision has to be generated for the round function, $f()$, or a problem has to be solved of comparable difficulty. The proof is given in the framework of computational complexity theory. This effectively implies that the conditions and requirements imposed on cryptographic hash functions are transferred to the round function $f()$. In [3] it is stated that $f()$ should be a bijective function. This statement is based on the results obtained in [30].

The definitions for MACs, OWHFs and CRHFs can be modified and applied to the round functions used for these functions as follows:

Round Function for MAC

Definition 5.1 *The round function for a MAC is a function f() satisfying the following conditions:*

1. *The description of f() must be publicly known and the only secret information lies in the key, K, (extension of Kerkhoff's principle).*

2. *The key K should be used in every application of f() to every block of X_i.*

3. *The argument X_i is a segment of the message X. X_i has a fixed length of m bits and the result $f(K, X_i)$ has a fixed length of n bits.*

4. *Given f(), K and X_i, the computation of f(K,X) must be easy.*

5. *Given f() and X_i, it is hard to determine $f(K, X_i)$ with a probability of success significantly higher than 2^{-n}. Even where a large set of pairs $\{X_i, f(X_i, K)\}$ is known, where X_i have been selected by the opponent, it is "hard" to determine the key, K, or to compute $f(K, X_i')$ for any $X_i \neq X_i'$.*

Note the explicit requirement that the secret key should be used in each application of $f()$. This requirement is stated to discourage the use of the initial value, IV, as a secret key, K, in a MAC. If the IV is used as the key K, the key is used only in the first iteration of $f()$. This allows an attacker to add message blocks and update the hash value without knowledge of K. In certain hash functions, such as MD4, knowledge of the hash value and the message allows an attacker to determine K if K is used as the IV. For this reason it is advised that $f()$ is dependent on K.

Round Function for OWHF

Definition 5.2 *The round function for a OWHF is a function f() satisfying the following conditions:*

1. *The description of f() must be publicly known and should not require any secret information for its operation (extension of Kerkhoff's principle).*

2. *The argument X_i is a segment of the message X. X_i has a fixed length of m bits and the result $f(X_i)$ has a fixed length of n bits.*

3. *Given f() and X_i, the computation of $f(X_i)$ must be easy.*

4. *The hash function must be one way in the sense that:*

   (a) *given a Y in the image of f(), it is "hard" to find a message X_i such that $f(X_i) = Y$.*

(b) given X_i and $f(X_i)$ it is "hard" to find a message block $X_i' \neq X_i$ such that $f(X_i) = f(X_i')$.

Round Function for CRHF

Definition 5.3 *The round function for a OWHF is a function f() satisfying the following conditions:*

1. *The description of f() must be publicly known and should not require any secret information for its operation (extension of Kerkhoff's principle).*
2. *The argument X_i is a segment of the message X. X_i has a fixed length of m bits and the result $f(X_i)$ has a fixed length of n bits.*
3. *Given f() and X_i, the computation of $f(X_i)$ must be easy.*
4. *The hash function must be one way in the sense that:*
   (a) *given a Y in the image of f(), it is "hard" to find a message X_i such that $f(X_i) = Y$.*
   (b) *given X_i and $f(X_i)$ it is "hard" to find a message $X_i' \neq X_i$ such that $f(X_i) = f(X_i')$.*
5. *The round function f() must be collision resistant: This means that it is hard to find two distinct messages that result in the same image for the round function f().*

Thus, the conditions imposed on a round function used in a MAC, OWHF or CRHF, are similar to those imposed on the respective cryptographic hash functions. It is also interesting to note that the conditions imposed on the round functions are similar to those defined in [3] for one way functions (OWF) and collision resistant functions (CRF). A number of frequently used building blocks used in the construction of round functions for secure hash functions are identified in Sections 5.4 and 5.5.

A large number of attacks on hash functions based on the Damgård-Merkle Scheme, focuses on the compress function [17]. Although attacks on the compress function are usually specific to the hash algorithm, the general attacks described in Chapter 3 are also applicable. The parameters of the compress function should be chosen to render these attacks harmless.

5.3.3 The Chaining Rule

The third building block is the chaining rule. Chaining is used when the message length exceeds the maximum allowable input length to the compress function.

When processing the message in blocks, the previous result of the compress function has to be taken into account. This is accomplished by feeding the result from the previous compress operation back and combine it in some way with the new block that has to be processed.

The chaining rule determines which part of the chaining variable should be fed back. This approach is often used when a block cipher is used as a round function, since many block ciphers has a key length that is shorter than the block length. It is advised that the full result is fed back and used in the next iteration of the compress function.

Note that the introduction of chaining, allows attacks dependent on the chaining. These attacks include meet in the middle attacks, correcting block attacks, fixed point attacks and differential attacks (see Chapter 3 Section 3.4). The length of the chaining variables, measured in bits, should therefore be chosen to render these attacks computationally infeasible.

5.3.4 Construction

The interaction of the building blocks identified in the Damgård-Merkle scheme are described as follows. For a hash function $h()$ with a compress function $f()$ an initial value IV and a suitably padded message X, the interaction of the various building blocks are described by:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(X_i, H_{i-1}) \quad i \in \{1, 2, 3 \ldots j\} \\ h(X) &= H_j. \end{aligned}$$

A graphical representation of the interaction of the three building blocks are shown in Figure 5.3.

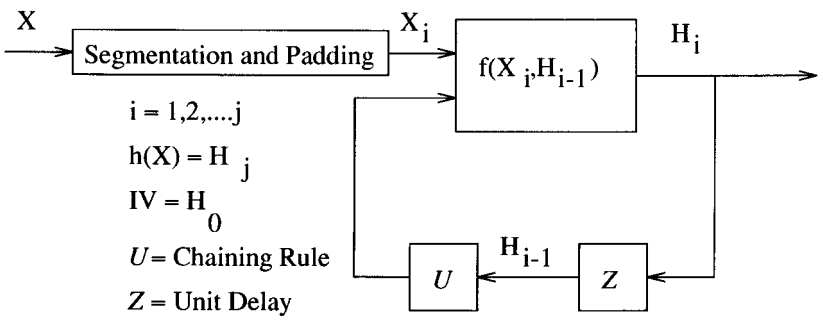

Figure 5.3: General Hash Function Construction (MDCs)

The construction shown in Figure 5.3 is specific to MDCs. For a MAC the interaction of the compress function, $f()$, the initial value, IV, the secret key, K, and a suitably padded message, X, is described as follows:

$$\begin{aligned} H_0 &= IV \\ H_i &= f(K, X_i, H_{i-1}) \quad i \in \{1, 2, 3 \ldots j\} \\ h(X) &= H_j. \end{aligned}$$

A graphical representation of the interaction of the various components of an iterated hash function used as a MAC, is shown in Figure 5.4.

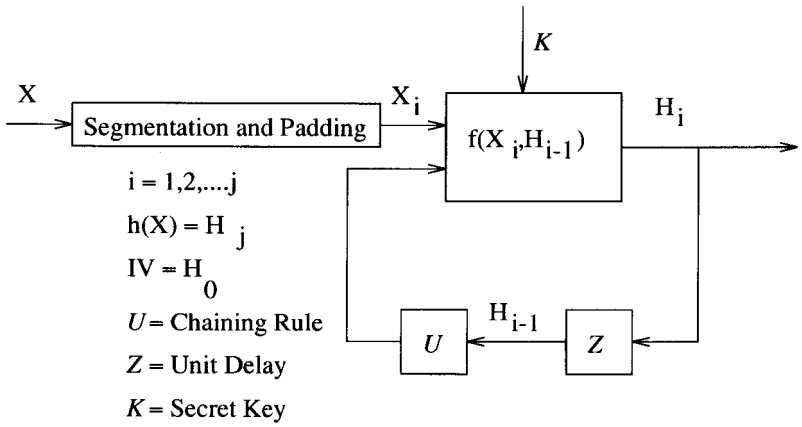

Figure 5.4: General Hash Function Construction (MACs)

A number of the better known dedicated hash functions based on the Damgård-Merkle scheme include BCA, MD4, MD5, SHA, SHA-1, Haval, RIPEMD, N-Hash, Snefru and Tiger.

5.4 ROUND FUNCTION CONSTRUCTIONS

The majority of hash functions designed in recent years make use of an iterative structure (Section 5.3). Hash functions based on iterative structures require secure round functions. A number of conditions are imposed on the round functions (Section 5.3). Currently there are three round function constructions in popular use. They are:

1. MD4-Family Construction.
2. Block Ciphers.
3. Stream Ciphers.

In this chapter the MD4-family construction is considered in detail. The use of block and stream ciphers in round function constructions are considered in Appendix A.

5.4.1 MD4-Family Construction

The round function construction used for MD4 is described in [10]. This construction has been widely adopted in the design of other hash functions such as MD5 [45], SHA-1 [13], Tiger [47] and RIPEMD-160 [15]. The round functions of these dedicated hash functions are similar in design and construction.

Consider the iterated hash function as represented in Figures 5.3 and 5.4. Note that at most three inputs are supplied to the round function. These inputs consist of the current message block, X_i, the previous hash result, H_{i-1}, and a secret key, K. Note that the secret key is only applicable when the construction is used as a MAC. The generalised MD4-family construction does not allow for the inclusion of a secret key. Adaptations of this construction that does make allowance for a secret key is presented in Appendix A.

The round function used in the MD4-family of constructions is itself an iterated construction. The round function take as input the previous hash result H_{i-1} and the current message block, X_i (see Figure 5.5).

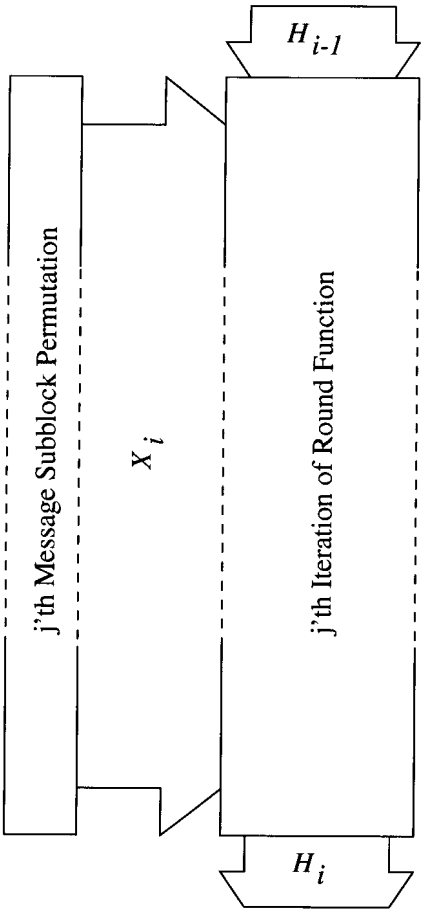

Figure 5.5: Generalised MD4 Round Function Construction

The message block X_i is segmented into k sub-blocks. The previous hash result is set equal to the initial chaining variable, C, for the round function. The set of message sub-blocks are permutated and applied to the j'th iteration of the round functions. The chaining variable, C, is then updated and applied to the next iteration of the round function. This process is repeated three or four times. The permutation of the sub-block and the method used for updating the chaining variable for each round is different for each iteration in the round function. Each iteration of the round function is constructed from the elementary building blocks described in Section 5.5.

5.5 ROUND FUNCTION BUILDING BLOCKS

This section contains descriptions of the building blocks frequently used in the construction of round functions for cryptographic hash functions. These building blocks facilitate the

fulfillment of the requirements of diffusion and confusion as defined by Shannon [46] (see Chapter 4 Section 4.3.1). For this reason the building blocks identified in this section are also commonly used in the construction of other cryptographic functions.

5.5.1 Bit Permutations

The use of bit permutations as building blocks in cryptographic primitives are considered in [48]. A bit permutation of a vector modifies the order of the components of the vector, without changing the values of the individual bits. In [48] it is recommended that the new bit positions should be calculated from the old bit positions using a simple expression. As an example, consider the simple cyclic rotation over a vector of length l. The bit in position i is rotated over d-positions and the new bit position is given by $i+d \bmod l$. These permutations are popular and are used in a number of dedicated hash functions, including MD4, MD5, SHA and SHA-1.

If the bit permutation is implemented in dedicated hardware, the bit permutations can be achieved through "hardwiring" the permutation into silicon. The price paid for this approach is the large amount silicon required to accomplish such a task [48].

If the bit permutation is to be implemented in software on general purpose processors, a permutation where individual bits are to be moved around should be avoided due to the reduction in performance. Lookup tables can be used to speed-up the bit permutation process, but as remarked in [48] the size of the lookup tables grows exponentially, rendering this technique infeasible. Instead it is advised that bit permutations be implemented in a block-wise manner. Bit permutations which satisfy this requirement include vector rotations, as described earlier. Note that the choice of a specific vector length, say 32 bits, is likely to favour certain architectures, while putting others at a disadvantage. Thus, portability of the algorithm is decreased.

As mentioned previously, vector rotations form popular building blocks for a number of well known dedicated hash functions. The degree of security obtained from the use of these vector rotations is considered inadequate in certain dedicated hash functions. The use of vector rotations in MD4 in particular is shown to add little in terms of additional security [14], [17]. One of the reasons the vector rotations in MD4 are ineffective is that the rotation factor, d, is a constant for each step. With the rotation constants known, the effect of the rotations

can be calculated and countered. In the RC5 and RC6 encryption algorithm data dependent vector rotations are introduced [27], [50]. It is therefore conceivable that security obtained from vector rotations can be increased by making the rotation factor, d, data dependent when designing dedicated hash functions.

Thus, when choosing a bit permutation the method of implementation, the portability of the algorithm, the reduction in speed for a complicated bit permutation technique and the required cryptographic strength should be kept in mind.

5.5.2 Bitwise Boolean Operations

Bitwise Boolean operations are widely used in the MD-family of hash functions (MD4, MD5 and SHA-1). A bitwise Boolean operation treats all individual components of the binary vector in the same manner. Commonly used bitwise operators are complementation, bitwise AND, OR and XOR.

Bitwise binary operators are easily described in both hardware and software. For a processor architecture with word length k, the bitwise Boolean operation on a vector of length n can be split into $\lceil \frac{n}{k} \rceil$ operations, if $n > k$. For these reasons bitwise Boolean operations are portable, not only between different processor architectures, but also between hardware and software platforms.

A number of desirable properties of Boolean functions are proposed in chapter 3 of [51]. These properties should be used as design criteria when constructing bitwise Boolean operations for cryptographic hash functions.

5.5.3 Substitution Boxes

A substitution box, or S-box, is defined as an $n \times m$ mapping of a n bit vector to an m bit vector, m and n need not be equal [51].

S-boxes are traditionally used as building blocks in block ciphers and stream ciphers. S-boxes have not been used extensively in the design of cryptographic hash functions. An example of a dedicated hash function that makes use of S-boxes is Tiger [47].

For ease of implementation and description it is proposed to keep S-boxes small, specify a way to generate the S-boxes at run time, or limit the number of S-boxes used [48]. When using hardware implementations the S-boxes should be kept small due to the silicon area required. In software S-boxes are usually contained in arrays. This limits portability due to specific data word lengths used by specific processors.

In chapter 4 of [51] it is remarked that if $m = 1$ the mapping is a Boolean function. Thus a Boolean function is a special instance of a S-box. The MD family of hash functions makes use of bitwise Boolean functions rather than S-boxes. This choice is due to the memory and performance penalties associated with the use of large S-boxes. It is believed that the use of cryptographically strong S-boxes instead of bitwise Boolean operators will result in stronger cryptographic hash functions [47]. An extensive treatment of the issue of S-box design and analysis is given in [51].

5.5.4 Modular Arithmetic Operations

Modular arithmetic has been identified as a building block from which hash functions can be constructed [48]. A number of hash functions have been based on modular arithmetic [2]. In [2] three arguments are presented in favour of using modular arithmetic:

1. Hardness of number theoretic problems.
2. Availability of modular arithmetic implementations.
3. Scalability.

The schemes based on modular arithmetic are classified according to the size of the modulus used. Schemes with a small modulus (32 bits) have been proposed in [52]. These schemes are believed vulnerable to divide and conquer attacks [3]. Schemes with a large modulus (512 bit or more) are evaluated in [3]. It has been shown in [40] that these schemes are insecure when used with the RSA signature scheme. The use of modular arithmetic for the construction of cryptographically strong hash functions is considered limited [2].

5.6 CONCLUSION

In this chapter, a generic construction for building MACs and MDCs that satisfies the requirements presented in Chapter 4, was introduced. In particular the construction of the iterated hash scheme used by the MD4 family of functions were considered. The design of appropriate round functions is considered in Section 5.4 and 5.5. Commonly used building blocks for cryptographic primitives are discussed in Section 5.5.

CHAPTER 6: ANALYSIS OF THE MD4 HASH ALGORITHM

6.1 INTRODUCTION

In this chapter the MD4 algorithm is considered. The MD4 algorithm is described followed by the reconstruction of the analysis of MD4 as presented by Dobbertin [14]. In addition the attack presented by Dobbertin is extended in a novel way that allows the computational requirements to be reduced by a factor 64.

6.2 INTRODUCTION TO MD4

MD4 is a dedicated hash function proposed by R. Rivest [10], [44]. MD4 is an acronym for "Message Digest 4". MD4 is an unkeyed dedicated cryptographic hash function (MDC). MD4 is based on the iterative construction proposed discussed in Chapter 5. The MD4 algorithm was designed to meet the following criteria.

1. Security.
2. Speed.
3. Simplicity and compactness.
4. Favors little endian architectures.

The most prominent design criterion is security. This implies that it should be computationally infeasible to find two messages, M_1 and M_2, that hashes to the same value. In other words, MD4 is intended to be a collision resistant hash function. The remaining three criteria are concerned with high speed implementation in software.

A complete definition of MD4, including the padding rule is given in [10] and [44]. Since this chapter is specifically concerned with the cryptanalysis of MD4 it is useful to consider the operation of MD4 before concentrating on the analysis.

6.3 NOTATION

Before proceeding to describe the operation of MD4 it is appropriate to define the notation used in this chapter.

$$\begin{aligned} X[j] &= \text{32-bit word}, \qquad j \in \{0, 1, 2 \ldots 15\} \\ \tilde{X}[j] &= \text{Alternative 32-bit word}, \quad j \in \{0, 1, 2, \ldots 15\} \\ (AA, BB, CC, DD) &= \text{Hash variables} \\ (A_i, B_i, C_i, D_i) &= \text{Chaining variables after step } i, \quad i \in \{0, 1, 2, \ldots 47\} \end{aligned}$$

6.4 THE MD4 ALGORITHM

MD4 is an iterated hash function. Each iteration requires the application of the compress function. For MD4 the compress function is defined by the sequential application of three distinct rounds. The elementary size of a message block is 512 bits. If the message is not a multiple of 512 bits, a padding rule is used. Before the message block is processed it is divided into 16 blocks of 32 bits each. Four 32-bit chaining variables are used to produce a 128-bit hash value. The following steps are identified in the MD4 algorithm.

Algorithm 6.1 *MD4 hash algorithm*

1. *Pad message.*
2. *Append the length of the message.*
3. *Hash and chaining variable initialisation.*
4. *Round 1.*
5. *Round 2.*
6. *Round 3.*
7. *Update hash variables.*
8. *Has the entire message been processed ?*
   (a) *No: Repeat from step 4.*
   (b) *Yes: Continue.*
9. *Output hash value.*

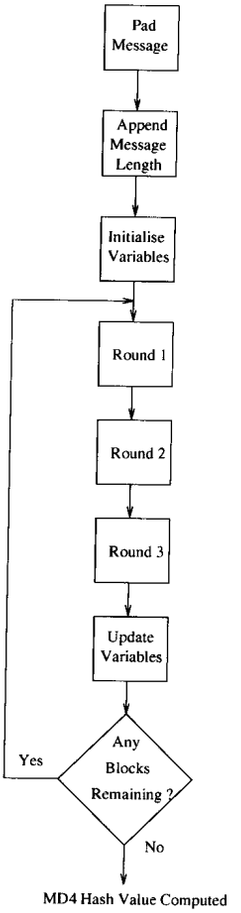

Figure 6.1: MD4 Block Diagram

A block diagram of the steps in the MD4 algorithm is shown in Figure 6.1. A description of each of the steps identified in the MD4 algorithm is presented next.

6.4.1 Message Padding

The first two steps ensure that the message length is a multiple of 512 bits. This allows the message to be processed in blocks of 512 bits at a time. The padding rule is described in [10] and [44].

6.4.2 Initial Values

Step 3 initiates the four chaining variables as follows:

$$
\begin{aligned}
A_0 &= \texttt{0x67452301} \\
B_0 &= \texttt{0xEFCDAB89} \\
C_0 &= \texttt{0x98BADCFE} \\
D_0 &= \texttt{0x10325476}.
\end{aligned}
$$

The hash variables contains the hash value for each iteration and is initialised as shown below:

$$
\begin{aligned}
AA &= A_0 \\
BB &= B_0 \\
CC &= C_0 \\
DD &= D_0.
\end{aligned}
$$

6.4.3 Iterative Rounds

Steps 4-8 performs the iterative computation of the hash value. The hash value is computed by applying three distinct rounds to each 512 bit block of the message. The hash function derives its strength from these three rounds. The hash variables are updated once all three rounds have been completed. If all of the 512 bit blocks have been processed the updated hash variables contains the final hash value.

The elementary operation within each round is described by:

$$
\begin{aligned}
R(S,T,U,V,X,K,W,r,j) &= (S + f_r(T,U,V) + X[j] + K_r)^{\lll W_{r_j}} \\
S &= R(S,T,U,V,X,K,W,r,j)
\end{aligned}
$$

with:

$$\begin{aligned} S, T, U, V &= \text{32 bit words} \\ X[j] &= j\text{'th 32 bit word of the message,} \quad j \in \{0, 1, 2 \ldots 15\} \\ r &= \text{Round } r \quad r \in \{1, 2, 3\} \\ K_r &= \text{The constant for the r'th round} \\ f_r &= \text{The boolean function in the r'th round} \\ x^{\lll W_{r_j}} &= \text{Circular rotate } x \text{ left by} W_{r_j} \text{ bits} \\ x + y &= \text{Modulo } 2^{32} \text{ addition of } x \text{ and } y. \end{aligned}$$

Each round differs from the other with regard to K_r and f_r. The index j in $X[j]$ is used to permutate the 512 bit input in 32-bit blocks for each round of the hash function. In each round W takes on one of four values.

Three boolean functions are defined for MD4.

$$\begin{aligned} F(X, Y, Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\ G(X, Y, Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\ H(X, Y, Z) &= X \oplus Y \oplus Z \end{aligned}$$

with:

$$\begin{aligned} \wedge &= \text{Bitwise AND} & \neg &= \text{Bitwise NOT} \\ \vee &= \text{Bitwise OR} & \oplus &= \text{Bitwise XOR.} \end{aligned}$$

The function $G(X, Y, Z)$ is a majority function. Thus for each bit position in X, Y, and Z the binary value that occurs more than once is selected. The function $F(X, Y, Z)$ is essentially a selection function. A graphical representation of the selection function $F(X, Y, Z)$ is shown in Figure 6.2.

The constants for K_r are defined in [10] as:

$$\begin{aligned} K_1 &= \texttt{0x00000000} \\ K_2 &= \texttt{0x5A827999} \\ K_3 &= \texttt{0x6ED9EBA1} \end{aligned}$$

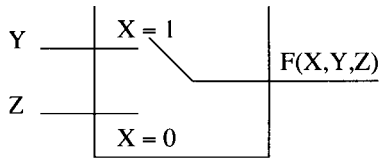

Figure 6.2: Selection Function $F(X, Y, Z)$

Three distinct rounds are identified for MD4. These rounds constitutes the compress function for MD4. It is therefore considered appropriate to give a detailed description of the rounds of MD4, since all the known attacks on MD4 focuses on the compress function. The equations describing the round functions are presented with this fact in mind.

Round 1

For the first round K_1 = `0x00000000` and is omitted from the equations. The boolean function $f_1 = F(X, Y, Z)$ for the first round. The four possible rotation constants for the first round of MD4 are defined as:

$$fs1 = 3 \qquad fs3 = 11$$
$$fs2 = 7 \qquad fs4 = 19$$

The complete set of equations for the first round are shown below:

$$A_3 = (A_0 + F(B_0, C_0, D_0) + X[0])^{\lll fs1} \quad (6.1)$$
$$D_3 = (D_0 + F(A_3, B_0, C_0) + X[1])^{\lll fs2} \quad (6.2)$$
$$C_3 = (C_0 + F(D_3, A_3, B_0) + X[2])^{\lll fs3} \quad (6.3)$$
$$B_3 = (B_0 + F(C_3, D_3, A_3) + X[3])^{\lll fs4} \quad (6.4)$$
$$A_7 = (A_3 + F(B_3, C_3, D_3) + X[4])^{\lll fs1} \quad (6.5)$$
$$D_7 = (D_3 + F(A_7, B_3, C_3) + X[5])^{\lll fs2} \quad (6.6)$$
$$C_7 = (C_3 + F(D_7, A_7, B_3) + X[6])^{\lll fs3} \quad (6.7)$$
$$B_7 = (B_3 + F(C_7, D_7, A_7) + X[7])^{\lll fs4} \quad (6.8)$$
$$A_{11} = (A_7 + F(B_7, C_7, D_7) + X[8])^{\lll fs1} \quad (6.9)$$
$$D_{11} = (D_7 + F(A_{11}, B_7, C_7) + X[9])^{\lll fs2} \quad (6.10)$$
$$C_{11} = (C_7 + F(D_{11}, A_{11}, B_7) + X[10])^{\lll fs3} \quad (6.11)$$
$$B_{11} = (B_7 + F(C_{11}, D_{11}, A_{11}) + X[11])^{\lll fs4} \quad (6.12)$$
$$A_{15} = (A_{11} + F(B_{11}, C_{11}, D_{11}) + X[12])^{\lll fs1} \quad (6.13)$$
$$D_{15} = (D_{11} + F(A_{15}, B_{11}, C_{11}) + X[13])^{\lll fs2} \quad (6.14)$$
$$C_{15} = (C_{11} + F(D_{15}, A_{15}, B_{11}) + X[14])^{\lll fs3} \quad (6.15)$$
$$B_{15} = (B_{11} + F(C_{15}, D_{15}, A_{15}) + X[15])^{\lll fs4} \quad (6.16)$$

Round 2

For the second round K_2 takes on the value as defined previously. The boolean function $f_2 = G(X, Y, Z)$ for the second round. The four possible rotation constants for the first round of MD4 are defined as:

$$gs1 = 3 \qquad gs3 = 9$$
$$gs2 = 5 \qquad gs4 = 13$$

The complete set of equations describing the second round are shown below:

$$A_{19} = (A_{15} + g(B_{15}, C_{15}, D_{15}) + X[0] + K_2)^{\lll gs1} \quad (6.17)$$
$$D_{19} = (D_{15} + g(A_{19}, B_{15}, C_{15}) + X[4] + K_2)^{\lll gs2} \quad (6.18)$$
$$C_{19} = (C_{15} + g(D_{19}, A_{19}, B_{15}) + X[8] + K_2)^{\lll gs3} \quad (6.19)$$
$$B_{19} = (B_{15} + g(C_{19}, D_{19}, A_{19}) + X[12] + K_2)^{\lll gs4} \quad (6.20)$$
$$A_{23} = (A_{19} + g(B_{19}, C_{19}, D_{19}) + X[1] + K_2)^{\lll gs1} \quad (6.21)$$
$$D_{23} = (D_{19} + g(A_{23}, B_{19}, C_{19}) + X[5] + K_2)^{\lll gs2} \quad (6.22)$$
$$C_{23} = (C_{19} + g(D_{23}, A_{23}, B_{19}) + X[9] + K_2)^{\lll gs3} \quad (6.23)$$
$$B_{23} = (B_{19} + g(C_{23}, D_{23}, A_{23}) + X[13] + K_2)^{\lll gs4} \quad (6.24)$$
$$A_{27} = (A_{23} + g(B_{23}, C_{23}, D_{23}) + X[2] + K_2)^{\lll gs1} \quad (6.25)$$
$$D_{27} = (D_{23} + g(A_{27}, B_{23}, C_{23}) + X[6] + K_2)^{\lll gs2} \quad (6.26)$$
$$C_{27} = (C_{23} + g(D_{27}, A_{27}, B_{23}) + X[10] + K_2)^{\lll gs3} \quad (6.27)$$
$$B_{27} = (B_{23} + g(C_{27}, D_{27}, A_{27}) + X[14] + K_2)^{\lll gs4} \quad (6.28)$$
$$A_{31} = (A_{27} + g(B_{27}, C_{27}, D_{27}) + X[3] + K_2)^{\lll gs1} \quad (6.29)$$
$$D_{31} = (D_{27} + g(A_{31}, B_{27}, C_{27}) + X[7] + K_2)^{\lll gs2} \quad (6.30)$$
$$C_{31} = (C_{27} + g(D_{31}, A_{31}, B_{27}) + X[11] + K_2)^{\lll gs3} \quad (6.31)$$
$$B_{31} = (B_{27} + g(C_{31}, D_{31}, A_{31}) + X[15] + K_2)^{\lll gs4} \quad (6.32)$$

Round 3

For the third round K_3 takes on the value as previously defined. The boolean function $f_3 = H(X, Y, Z)$ for the third round. The four possible rotation constants for the first round of MD4 are defined as:

$$hs1 = 3 \qquad hs3 = 11$$
$$hs2 = 9 \qquad hs4 = 15$$

The complete set of equations describing the third round are shown below:

$$A_{35} = (A_{31} + h(B_{31}, C_{31}, D_{31}) + X[0] + K_3)^{\lll hs1} \quad (6.33)$$
$$D_{35} = (D_{31} + h(A_{35}, B_{31}, C_{31}) + X[8] + K_3)^{\lll hs2} \quad (6.34)$$
$$C_{35} = (C_{31} + h(D_{35}, A_{35}, B_{31}) + X[4] + K_3)^{\lll hs3} \quad (6.35)$$
$$B_{35} = (B_{31} + h(C_{35}, D_{35}, A_{35}) + X[12] + K_3)^{\lll hs4} \quad (6.36)$$
$$A_{39} = (A_{35} + h(B_{35}, C_{35}, D_{35}) + X[2] + K_3)^{\lll hs1} \quad (6.37)$$
$$D_{39} = (D_{35} + h(A_{39}, B_{35}, C_{35}) + X[10] + K_3)^{\lll hs2} \quad (6.38)$$
$$C_{39} = (C_{35} + h(D_{39}, A_{39}, B_{35}) + X[6] + K_3)^{\lll hs3} \quad (6.39)$$
$$B_{39} = (B_{35} + h(C_{39}, D_{39}, A_{39}) + X[14] + K_3)^{\lll hs4} \quad (6.40)$$
$$A_{43} = (A_{39} + h(B_{39}, C_{39}, D_{39}) + X[1] + K_3)^{\lll hs1} \quad (6.41)$$
$$D_{43} = (D_{39} + h(A_{43}, B_{39}, C_{39}) + X[9] + K_3)^{\lll hs2} \quad (6.42)$$
$$C_{43} = (C_{39} + h(D_{43}, A_{43}, B_{39}) + X[5] + K_3)^{\lll hs3} \quad (6.43)$$
$$B_{43} = (B_{39} + h(C_{43}, D_{43}, A_{43}) + X[13] + K_3)^{\lll hs4} \quad (6.44)$$
$$A_{47} = (A_{43} + h(B_{43}, C_{43}, D_{43}) + X[3] + K_3)^{\lll hs1} \quad (6.45)$$
$$D_{47} = (D_{43} + h(A_{47}, B_{43}, C_{43}) + X[11] + K_3)^{\lll hs2} \quad (6.46)$$
$$C_{47} = (C_{43} + h(D_{47}, A_{47}, B_{43}) + X[7] + K_3)^{\lll hs3} \quad (6.47)$$
$$B_{47} = (B_{43} + h(C_{47}, D_{47}, A_{47}) + X[15] + K_3)^{\lll hs4} \quad (6.48)$$

A graphical representation of the three rounds that constitutes the compress function of MD4 is shown in Figure 6.3 (derived from [11]).

Update Variables

After completion of all three rounds, the hash variables are updated as shown below:

$$AA = A_0 = A_{47} + AA$$
$$BB = B_0 = B_{47} + BB$$
$$CC = C_0 = C_{47} + CC$$
$$DD = D_0 = D_{47} + DD$$

If all of the 512 bit message blocks have been processed, the final hash value is given by AA,

BB, CC, and DD. If there are unprocessed message blocks remaining, A_0, B_0, C_0, and D_0 contains the new initial values for the next iteration of MD4.

6.5 CRYPTANALYSIS OF MD4

The MD4 hash function has been extensively analysed since its introduction in 1990 [10],[44]. In 1991 an attack on the last two rounds of MD4 was presented by Bosselaers and den Boer [11]. An unpublished attack on the first two rounds of MD4 is credited to Merkle. In 1994 Vaudenay published an attack on the first two rounds of MD4 [53]. In 1995 Dobbertin presented a technique to cryptanalyse the MD4 hash function [14]. The result included in this chapter builds on the results obtained by Dobbertin by presenting an algorithm which requires $O(2^6)$ times less iterations for finding a collision.

This contains selected results obtained from the cryptanalysis of MD4. The remainder of this chapter is organised as follows. First the notation used for the cryptanalysis of MD4 is introduced. A review of the attack in [14] is then presented. This is followed by a description of an alternative algorithm which can be used to speed up the attack proposed in [14]. The results obtained from the use of the alternative algorithm are then considered. The source code that implements this attack is included for reference purposes as Appendix C.

6.6 NOTATION

Before proceeding with a description of the cryptanalysis of MD4 the following notation is introduced.

$$
\begin{aligned}
X[j] &= \text{32-bit word} \\
&\quad j \in [1, 16] \\
M_1 &= \text{Message 1} \\
M_2 &= \text{Message 2} \\
Z_i &= \text{Chaining variable for } M_1 \text{ after step } i \\
\tilde{Z}_i &= \text{Chaining variable for } M_2 \text{ after step } i \\
&\quad Z \in \{A, B, C, D\} \\
&\quad i \in [0, 47]
\end{aligned}
$$

The relationship between M_1 and M_2 is given by:

$$\tilde{X}[12] = X[12] + 1 \tag{6.49}$$

In [14] the following notation is introduced.

$$\begin{aligned}
\text{COMPRESS}_z^y &= \text{Value of chaining variables after steps} \\
&\quad \text{y to z of the compress function is performed.} \\
\Delta_i &= (A_i - \tilde{A}_i, B_i - \tilde{B}_i, C_i - \tilde{C}_i, D_i - \tilde{D}_i) \\
&\quad i \in [0, 47] \\
Z^{\lll X} &= \text{Left circular rotation of Z by X bit positions.} \\
-Z^{\lll X} &= -(Z^{\lll X}).
\end{aligned}$$

The following relationship exists between the notation used in [14] and the notation used in this appendix.

$$\begin{aligned}
B &= B_{11} \\
C &= C_{11} \\
U &= A_{15} \\
\tilde{U} &= \tilde{A}_{15}
\end{aligned}
\qquad
\begin{aligned}
V &= D_{15} \\
\tilde{V} &= \tilde{D}_{15} \\
W &= C_{15} \\
\tilde{W} &= \tilde{C}_{15} \\
Z &= B_{15} \\
\tilde{Z} &= \tilde{B}_{15}
\end{aligned}
\qquad
\begin{aligned}
A_* &= A_{19} \\
B_* &= B_{19} \\
\tilde{B}_* &= \tilde{B}_{19} \\
C_* &= C_{19} \\
\tilde{C}_* &= \tilde{C}_{19} \\
D_* &= D_{19}.
\end{aligned}$$

6.7 DOBBERTIN'S ATTACK: A REVIEW

The cryptanalysis of MD4 is described in [14]. The attack could be viewed as a divide and conquer attack. The attack is divided into two parts. The first part is concerned with the establishment of a so-called inner almost-collision. The second part of the attack is based on a differential attack and the matching of initial values. The differential attack can only succeed if the criteria set for the establishment of the inner almost-collisions has been met.

An inner almost-collision is obtained if a set of chaining variables are found for which the

difference between $\text{COMPRESS}_{19}^{12}$ performed on M_1 and M_2 is:

$$\Delta_{19} = (0, 1^{\lll 25}, -1^{\lll 5}, 0)$$

In order for the differential attack to be successful the above condition has to be met. Thus obtaining an inner almost-collision is central to the success of the attack described in [14]. The above condition implies that the following relationship should hold between the chaining variables obtained for message M_1 and M_2 after step 19.

$$\begin{aligned} \tilde{A}_{19} &= A_{19} & \tilde{C}_{19} &= C_{19} + 1^{\lll 5} \\ \tilde{B}_{19} &= B_{19} - 1^{\lll 25} & \tilde{D}_{19} &= D_{19}. \end{aligned}$$

Using these relationships and conditions the following set of non-linear equations were derived in [14].

$$1 = \tilde{A}_{15}^{\lll 29} - A_{15}^{\lll 29} \tag{6.50}$$

$$F(\tilde{A}_{15}, B_{11}, C_{11}) - F(A_{15}, B_{11}, C_{11}) = \tilde{D}_{15}^{\lll 25} - D_{15}^{\lll 25} \tag{6.51}$$

$$F(\tilde{D}_{15}, \tilde{A}_{15}, B_{11}) - F(D_{15}, A_{15}, B_{11}) = \tilde{C}_{15}^{\lll 21} - C_{15}^{\lll 21} \tag{6.52}$$

$$F(\tilde{C}_{15}, \tilde{D}_{15}, \tilde{A}_{15}) - F(C_{15}, D_{15}, A_{15}) = \tilde{B}_{15}^{\lll 13} - B_{15}^{\lll 13} \tag{6.53}$$

$$G(\tilde{B}_{15}, \tilde{C}_{15}, \tilde{D}_{15}) - G(B_{15}, C_{15}, D_{15}) = A_{15} - \tilde{A}_{15} \tag{6.54}$$

$$G(\tilde{A}_{19}, \tilde{B}_{15}, \tilde{C}_{15}) - G(A_{19}, B_{15}, C_{15}) = D_{15} - \tilde{D}_{15} \tag{6.55}$$

$$G(\tilde{D}_{19}, \tilde{A}_{19}, \tilde{B}_{15}) - G(D_{19}, A_{19}, B_{15}) = C_{15} - \tilde{C}_{15} + \tilde{C}_{19}^{\lll 23} - C_{19}^{\lll 23} \tag{6.56}$$

$$G(\tilde{C}_{19}, \tilde{D}_{19}, \tilde{A}_{19}) - G(C_{19}, D_{19}, A_{19}) = B_{15} - \tilde{B}_{15} - 1 + \tilde{B}_{19}^{\lll 19} - B_{19}^{\lll 19} \tag{6.57}$$

Once a solution to equations (6.50) to (6.57) have been obtained, the messages M_1 and M_2

that leads to the inner almost-collision can be found by solving equations (6.58) to (6.66):

$$
\begin{aligned}
X[13] &= \text{Arbitrary} && (6.58)\\
X[14] &= C_{15}^{\lll 21} - C_{11} - F(D_{15}, A_{15}, B_{11}) && (6.59)\\
X[15] &= B_{15}^{\lll 13} - B_{11} - F(C_{15}, D_{15}, A_{15}) && (6.60)\\
X[0] &= A_{19}^{\lll 29} - A_{15} - F(B_{15}, C_{15}, D_{15}) - K_1 && (6.61)\\
X[4] &= D_{19}^{\lll 27} - D_{15} - F(A_{19}, B_{15}, C_{15}) - K_1 && (6.62)\\
X[8] &= C_{19}^{\lll 23} - C_{15} - F(D_{19}, A_{19}, B_{15}) - K_1 && (6.63)\\
X[12] &= B_{19}^{\lll 19} - B_{15} - F(C_{19}, D_{19}, A_{19}) - K_1 && (6.64)\\
D_{11} &= D_{15}^{\lll 25} - F(A_{15}, B_{11}, C_{11}) - X[13] && (6.65)\\
A_{11} &= A_{15}^{\lll 29} - F(B_{11}, C_{11}, D_{11}) - X[12]. && (6.66)
\end{aligned}
$$

By setting:

$$A_{15} = \texttt{0xFFFFFFFF} \qquad \tilde{A}_{15} = \texttt{0x00000000} \qquad \tilde{B}_{11} = \texttt{0x00000000}\,.$$

Equations (6.50) to (6.57) are reduced to:

$$
\begin{aligned}
\tilde{B}_{15} &= B_{15} - G(\tilde{C}_{19}, \tilde{D}_{19}, \tilde{A}_{19}) + G(C_{19}, D_{19}, A_{19}) + \\
&\quad \tilde{B}_{19}^{\lll 19} - B_{19}^{\lll 19} - 1 && (6.67)\\
\tilde{C}_{15} &= C_{15} - G(\tilde{D}_{19}, \tilde{A}_{19}, \tilde{B}_{15}) + G(D_{19}, A_{19}, B_{15}) + \\
&\quad \tilde{C}_{19}^{\lll 23} - C_{19}^{\lll 23} && (6.68)\\
D_{15} &= \tilde{C}_{15}^{\lll 21} - C_{15}^{\lll 21} && (6.69)\\
\tilde{D}_{15} &= D_{15} - G(\tilde{A}_{19}, \tilde{B}_{15}, \tilde{C}_{15}) + G(A_{19}, B_{15}, C_{15}) && (6.70)\\
C_{11} &= \tilde{D}_{15}^{\lll 25} - D_{15}^{\lll 25}. && (6.71)
\end{aligned}
$$

The solutions obtained for the above equations should also satisfy the following two conditions.

$$
\begin{aligned}
G(\tilde{B}_{15}, \tilde{C}_{15}, \tilde{D}_{15}) - G(B_{15}, C_{15}, D_{15}) &= 1 && (6.72)\\
F(\tilde{C}_{15}, \tilde{D}_{15}, \tilde{A}_{15}) - F(C_{15}, D_{15}, A_{15}) &= \tilde{B}_{15}^{\lll 13} - B_{15}^{\lll 13}. && (6.73)
\end{aligned}
$$

In his paper Dobbertin suggests an algorithm to solve the set of non-linear equations described by (6.67) to (6.73). This algorithm is replicated below.

Algorithm 6.2 *Dobbertin's Algorithm for Producing Almost-Inner Collisions*

1. *Choose A_{19}, B_{19}, C_{19}, D_{19}, B_{15} and C_{15} randomly. Compute $\tilde{B}_{19}$, $\tilde{C}_{19}$, $\tilde{B}_{15}$, $\tilde{C}_{15}$, D_{15} and $\tilde{D}_{15}$ as described in (6.67) to (6.71). Test if (6.72) is satisfied. If the test is passed goto 2.*

2. *Take A_{19}, B_{19}, C_{19}, D_{19}, B_{15} and C_{15} found in 1 as "basic values". Change one random bit in each of these variables, compute $\tilde{B}_{19}$, $\tilde{C}_{19}$, $\tilde{B}_{15}$, $\tilde{C}_{15}$, D_{15} and $\tilde{D}_{15}$ and test if (6.72) is satisfied. Then test if the left 4 bits of (6.73) are equal to 0. If this test is passed take the corresponding values A_{19}, B_{19}, C_{19}, D_{19}, B_{15} and C_{15} as the new "basic values". The next is doing the same as before, but now testing if the 8 left bits of (6.73) instead of 4 bits are zero. Continue with the left 12,16 ... left bits until (6.73) is fulfilled.*

3. *Now (6.72) and (6.73) are satisfied and we obtain an inner almost-collision by setting $B_{11} = 0$ and defining A_{11}, C_{11}, D_{11} and $X[i]$ according to equations (6.58) to (6.66).*

For the inner almost-collision to be admissible, it is required that the following equation holds:

$$G(\tilde{B}_{19}, \tilde{C}_{19}, \tilde{D}_{19}) = G(B_{19}, C_{19}, D_{19}) \tag{6.74}$$

If equation (6.74) does not hold the differential attack is unlikely to succeed. Algorithm 6.2 has to be repeated until equation 6.74 is satisfied.

In the next section an alternative algorithm for solving the above set of non-linear Boolean equations is presented.

6.8 ALTERNATIVE ALGORITHM FOR ESTABLISHING INNER ALMOST-COLLISIONS

In this section we proceed to describe an alternative algorithm that leads to the solution of equations (6.50) to (6.57) and the establishment of inner almost-collisions.

For equations (6.50) to (6.57), make the following settings.

$$\begin{aligned} B_{11} &= \texttt{0x00000000} & A_{15} &= \texttt{0xFFFFFFFF} \\ C_{11} &= \texttt{0x00000000} & \tilde{A}_{15} &= \texttt{0x00000000} \end{aligned}$$

The choice of $\tilde{A}_{15}$ and A_{15} immediately satisfies equation (6.50). The choices for B_{11} and C_{11} implies that D_{15} and $\tilde{D}_{15}$ are equal. A collision can now be established by setting:

$$\begin{aligned} D_{15} &= \texttt{0xFFFDFFFE} \\ \tilde{D}_{15} &= D_{15} \\ C_{15} &= \texttt{0xEDFFCFFF} \\ \tilde{C}_{15} &= \texttt{0xFDFFDFFF} \\ \tilde{B}_{15} &= B_{15} + \tilde{B}_{19}^{\lll 19} - B_{19}^{\lll 19} - 1 \end{aligned}$$

The values for $\tilde{D}_{15}$, D_{15}, $\tilde{C}_{15}$ and C_{15} are chosen to satisfy equations (6.52) and (6.53) and to facilitate the easy manipulation of the functions $F(X, Y, Z)$ and $G(X, Y, Z)$. The choice of the relationship between $\tilde{B}_{15}$ and B_{15} ensures that it is easy to find a solution to equation (6.57). The following set of equations now needs to be solved.

$$F(\tilde{C}_{15}, \tilde{D}_{15}, \tilde{A}_{15}) - F(C_{15}, D_{15}, A_{15}) = \tilde{B}_{15}^{\lll 13} - B_{15}^{\lll 13} \quad (6.75)$$

$$G(\tilde{B}_{15}, \tilde{C}_{15}, \tilde{D}_{15}) - G(B_{15}, C_{15}, D_{15}) = \texttt{0xFFFFFFFF} \quad (6.76)$$

$$G(\tilde{A}_{19}, \tilde{B}_{15}, \tilde{C}_{15}) - G(A_{19}, B_{15}, C_{15}) = 0 \quad (6.77)$$

$$G(\tilde{D}_{19}, \tilde{A}_{19}, \tilde{B}_{15}) - G(D_{19}, A_{19}, B_{15}) = C_{15} - \tilde{C}_{15} + \tilde{C}_{19}^{\lll 23} - C_{19}^{\lll 23} \quad (6.78)$$

$$G(\tilde{C}_{19}, \tilde{D}_{19}, \tilde{A}_{19}) - G(C_{19}, D_{19}, A_{19}) = 0 \quad (6.79)$$

Thus equations (6.50) to (6.57) can be reduced to equations (6.75) to (6.79). Note that equations (6.75) to (6.79) each contain a single unknown variable. It is now possible to define an algorithm that has a high probability to yield an admissible inner almost-collision. The suggested algorithm for finding an inner almost-collision is defined below:

Algorithm 6.3 *Alternative Algorithm for Producing inner almost-Collisions*

1. *Choose a random value for $\tilde{B}_{15}$ and compute B_{15}. Repeat this step until equation (6.75) and (6.76) are satisfied.*
2. *Choose a random value for A_{19}. Repeat this step until equation (6.77) is satisfied.*
3. *Choose a random value for D_{19}, $\tilde{B}_{19}$, $\tilde{C}_{19}$ and compute B_{19} and C_{19}. Repeat this step until equation (6.78) and equation (6.79) are satisfied.*
4. *Verify if equation (6.74) holds.*
   (a) *If equation (6.74) holds an admissible inner collision was found. Proceed to construct M_1 and M_2 as described by equations (6.58) to (6.66).*
   (b) *If equation (6.74) does not hold, repeat this algorithm from step 1.*

Once an admissible inner almost-collision is found, the differential attack described in [14] may be used to find a collision for all three rounds of MD4.

6.9 RESULTS

When comparing the performance of Algorithm 6.2 with that of Algorithm 6.3, two observations are made.

6.9.1 Number of Collisions

It is noted that when Algorithm 6.3 is used to find an admissible inner almost-collision, only a subset of all possible admissible inner almost-collisions is produced. This is due to the selection of A_{15}, $\tilde{A}_{15}$, B_{15}, $\tilde{B}_{15}$, C_{15} and $\tilde{C}_{15}$. When constructing an admissible inner almost-collision the attacker is free to choose 5 variables at random. This leaves the attacker with 2^{160} options. Each random choice does not however guarantee an admissible inner almost-collision. On average the attacker has to make 2^8 random choices for the 5 variables before an inner almost-collision is found. This reduces the number of inner almost-collisions to an estimated 2^{152}. According to [14] approximately one in every nine inner almost-collisions are admissible. Thus approximately 2^{149} admissible inner almost-collisions can be found with Algorithm 6.3. It is pointed out in [14] that it is possible to construct approximately 2^{106} message pairs that yield a collision for each admissible inner almost-collision under the MD4

hash function. Thus with the use of Algorithm 6.3 it is possible, to generate approximately 2^{255} message pairs that hash to the same value using MD4. When using Algorithm 6.2 the number of message pairs that result in a collision are estimated at 2^{281}.

6.9.2 Speedup Factor

Algorithm 6.3 has an advantage over Algorithm 6.2 when the number of operations required to find an admissible inner almost-collision is considered. A practical measurement of Algorithm 6.2 shows that approximately $O(2^{14})$ trials are required to find an admissible inner almost-collision. A similar measurement shows that Algorithm 6.3 requires on average $O(2^{8})$ trials to find an admissible inner almost-collision. This represents a speedup factor of approximately 64.

6.9.3 Example

An example of two messages that were constructed using Algorithm 6.3 for finding admissible inner almost-collisions is shown below. The common hash value is included for reference.

$$
\begin{aligned}
X[0] &= \texttt{0xD6E3C2EA} & X[8] &= \texttt{0x25B0C32D} \\
X[1] &= \texttt{0x31759BA4} & X[9] &= \texttt{0xD1E9E09B} \\
X[2] &= \texttt{0x09028A49} & X[10] &= \texttt{0xEC08A64A} \\
X[3] &= \texttt{0x00DC9F7B} & X[11] &= \texttt{0x32CC035A} \\
X[4] &= \texttt{0x9688334C} & X[12] &= \texttt{0x669080A4} \\
X[5] &= \texttt{0x6A848F6B} & X[13] &= \texttt{0x31C4794B} \\
X[6] &= \texttt{0xB5E292DD} & X[14] &= \texttt{0xFFFFBFFB} \\
X[7] &= \texttt{0x4DCC5516} & X[15] &= \texttt{0xA281EB3F}
\end{aligned}
$$

$$\tilde{X}[12] = \texttt{0x669080A5}$$

Common Digest:

```
0x94a568a0 0x84678967 0xea8da2b9 0x055dd5ab
```

6.10 CONCLUSION

This chapter contains a concise description of the operation of MD4. Attention has been paid in particular to the three rounds that constitutes the compress function for MD4. This is due to the importance of these rounds in the cryptanalysis of MD4 as presented in this Chapter. An implementation of the MD4 algorithm is attached as Appendix B.

The description of MD4 is followed by a description of the attack by Dobbertin on MD4. It is shown that a speedup of the attack on MD4 is possible. The speedup factor is estimated to be a factor of 64. The improvement in speed is attained at the cost of a reduction in the number of possible messages which result in a collision. These results were also presented at the Comsig 97 conference [56].

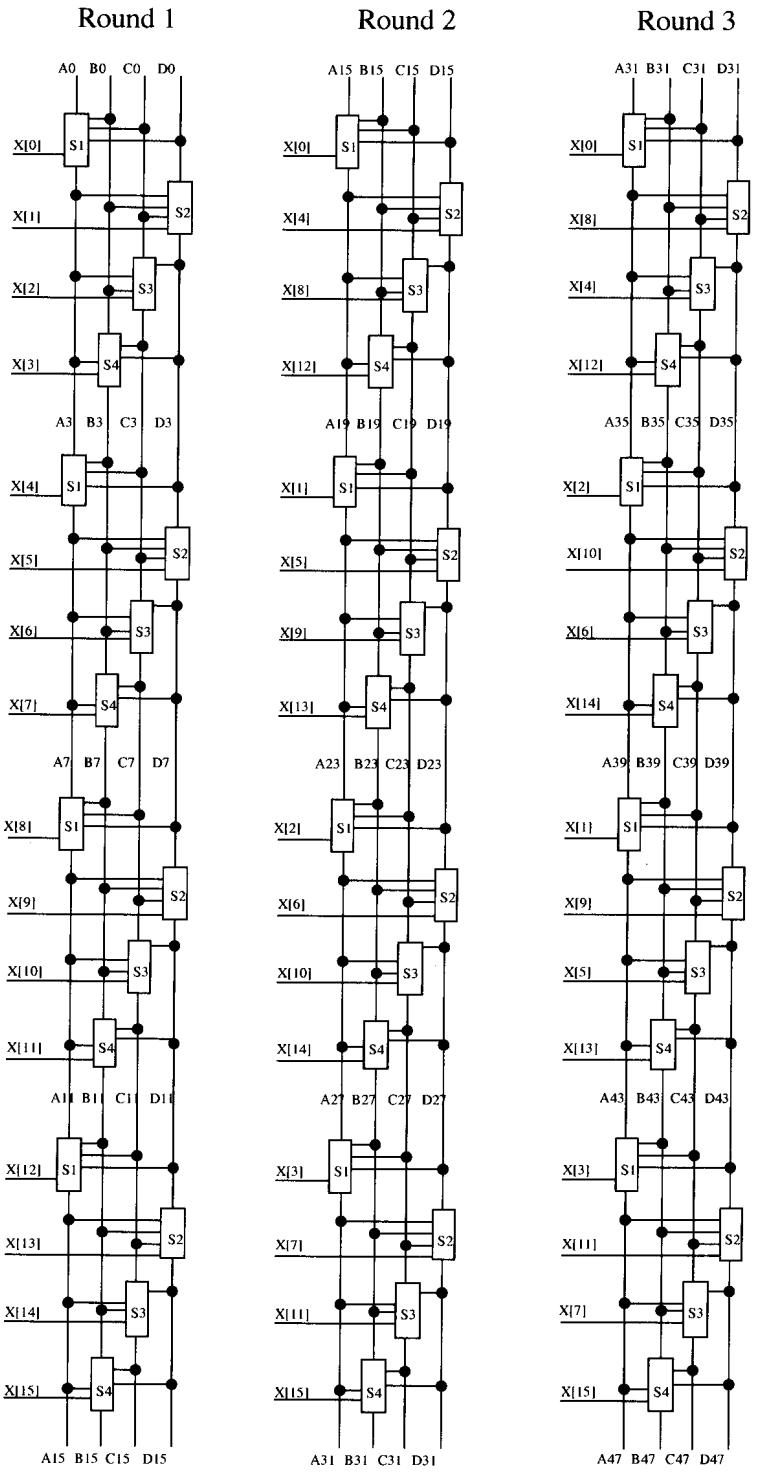

Figure 6.3: Diagrammatic Representation of the Compress Function of MD4

CHAPTER 7: ANALYSIS OF THE MD5 HASH ALGORITHM

7.1 INTRODUCTION

In this chapter we begin with a concise description of the MD5 hash algorithm. We then proceed by reconstructing the attack on MD5 as formulated by Dobbertin. The reconstruction is based on the source code used by Dobbertin to implement the attack. This is the first time a detailed description of the attack on MD5 is published.

7.2 INTRODUCTION TO MD5

MD5 is a dedicated hash function proposed by R. Rivest [45]. MD5 is the successor to MD4. MD5 is an extension of MD4. In extending MD4 to become MD5, a more conservative approach was taken. MD5 has the following features not encountered in MD4:

1. A fourth round.
2. A unique additive constant for each round.
3. The result of each step is added in the following step.
4. The permutations on the message words for rounds 2 and 3 were changed.
5. The rotation factors for each round were optimised for maximum avalanche effect.
6. The rotation factors for each round is unique.
7. The elementary function used in the second round was changed from $(X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$ to $(X \wedge Z) \vee (Y \wedge (\neg Z))$. The intention of this change is to reduce the symmetry in $g()$.

The most prominent design criterion is security. In [45] it is conjectured that it is computationally infeasible to find two messages, M_1 and M_2, that hashes to the same value, or to find a message that results in a specified hash value. In other words, MD5 is intended to be both collision resistant and pre-image resistant.

A complete definition of MD5, including the padding rule is given in [45]. In view of the analysis of MD5 it is useful to consider the operation of the algorithm.

7.3 NOTATION

Before proceeding to describe the operation of MD5 it is convenient to introduce the following notation:

$$\begin{aligned} X_j &= \text{32-bit word}, \qquad j \in \{0, 1, 2 \ldots 15\} \\ \tilde{X}_j &= \text{Alternative 32-bit word}, \quad j \in \{0, 1, 2, \ldots 15\} \\ (AA, BB, CC, DD) &= \text{Hash variables} \\ (A_i, B_i, C_i, D_i) &= \text{Chaining variables after step } i, \quad i \in \{0, 1, 2, \ldots 47\} \end{aligned}$$

7.4 THE MD5 ALGORITHM

Algorithm 7.1 *MD5 hash algorithm*

1. *Pad message.*
2. *Append the length of the message.*
3. *Hash and chaining variable initialisation.*
4. *Round 1.*
5. *Round 2.*
6. *Round 3.*
7. *Round 4.*
8. *Update hash variables.*
9. *Has the entire message been processed ?*
   (a) *No: Repeat from step 4.*
   (b) *Yes: Continue.*
10. *Output hash value.*

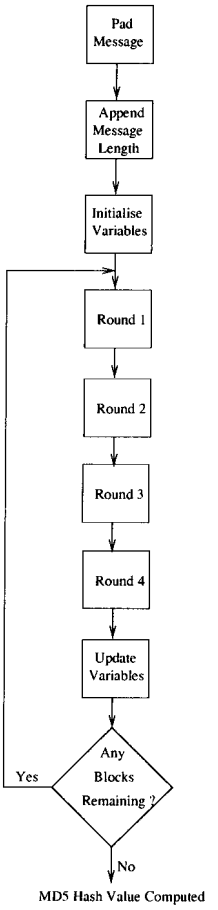

Figure 7.1: MD5 Block Diagram

MD5 is an iterated hash function based on the Dåmgard-Merkle construction [22], [23]. Each iteration requires the application of the compress function. The MD5 compress function is defined by the sequential application of four distinct rounds. The elementary size of a message block is 512 bits. If the message is not a multiple of 512 bits, a padding rule is used. Before the message block is processed, it is divided into 16 blocks of 32 bits each. Four 32-bit chaining variables are used, producing a 128-bit hash value. Algorithm 7.1 presents the main steps in MD5 along with a block diagrammatic representation of the structure of MD5 (see Figure 7.1). A description of each of the steps identified in the MD4 algorithm is presented next.

7.4.1 Message Padding

The first two steps ensure that the message length is a multiple of 512 bits. This allows the message to be processed in blocks of 512 bits at a time. The padding rule is described in [45].

7.4.2 Initial Values

The four chaining variables are initialised as:

$$\begin{aligned} A_0 &= \texttt{0x67452301} \\ B_0 &= \texttt{0xEFCDAB89} \\ C_0 &= \texttt{0x98BADCFE} \\ D_0 &= \texttt{0x10325476}. \end{aligned}$$

The hash variables contains the hash value for each iteration and is initialised as shown below:

$$\begin{aligned} AA &= A_0 \\ BB &= B_0 \\ CC &= C_0 \\ DD &= D_0. \end{aligned}$$

Note that this is identical to the initial values used for MD4 [10], [44].

7.4.3 Iterative Rounds

Steps 4-9 perform the iterative computation of the hash value. The hash value is obtained from the application of four distinct rounds to each 512 bit block of the message. The hash function derives its strength from these four rounds. The hash variables are updated once all four rounds have been completed. If all of the 512 bit blocks have been processed the updated hash variables contain the final hash value. The elementary operation within each round is described by:

$$\begin{aligned} R(S,T,U,V,X,K,W,r,j,i) &= S + (f_r(T,U,V) + X_j + T_i) \lll W_{r_j} \\ S &= R(S,T,U,V,X,K,W,r,j) \end{aligned}$$

with:

$$\begin{aligned} S,T,U,V &= \text{32 bit words} \\ X_j &= j\text{'th 32 bit word of the message,} \quad j \in \{0,1,2\ldots15\} \\ r &= \text{Round } r \quad r \in \{1,2,3,4\} \\ i &= \text{Step } (r-1)\cdot 16 + j \\ T_i &= \text{The i'th constant} \\ f_r &= \text{The boolean function in the r'th round} \\ x \lll W_{r_j} &= \text{Circular rotate } x \text{ left by} W_{r_j} \text{ bits} \\ x + y &= \text{Modulo } 2^{32} \text{ addition of } x \text{ and } y. \end{aligned}$$

Each round differs from the other with regard f_r. The index j in X_j is used to permutate the 512 bit input in 32-bit blocks for each round of the hash function. In each round W takes on one of four values. For each step the additive constant T_i is unique.

Four Boolean functions are defined for MD4.

$$\begin{aligned} f(X,Y,Z) &= (X \wedge Y) \vee ((\neg X) \wedge Z) \\ g(X,Y,Z) &= (X \wedge Z) \vee (Y \wedge (\neg Z)) \\ h(X,Y,Z) &= X \oplus Y \oplus Z \\ i(X,Y,Z) &= Y \oplus (X \vee (\neg Z)). \end{aligned}$$

with:

$\wedge$ = Bitwise AND $\quad$ $\neg$ = Bitwise NOT

$\vee$ = Bitwise OR $\quad$ $\oplus$ = Bitwise XOR.

The constants for T_i are defined in [45] and may be found in Appendix D which contains the source code for an implementation of MD5.

A graphical representation of a single step in the MD5 round function is shown in Figure 7.2

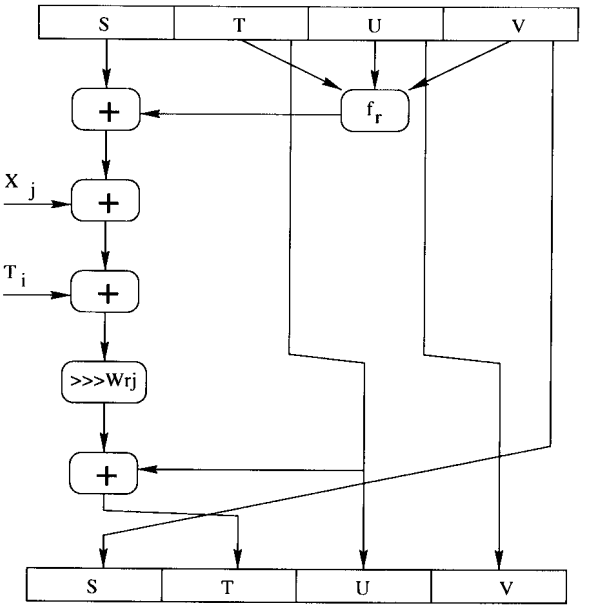

Figure 7.2: A Single Step in the MD5 Round Function

Four distinct rounds constitute the compress function for MD5. The following equations describe these rounds.

Round 1

The rotation constants for the first round are defined as:

$fs1 = 7$ $\quad$ $fs3 = 17$

$fs2 = 12$ $\quad$ $fs4 = 22$

The set of equations describing the first round are shown below:

$$A_3 = B_0 + (A_0 + f(B_0, C_0, D_0) + X_0 + T_0)^{\lll fs1} \quad (7.1)$$
$$D_3 = A_3 + (D_0 + f(A_3, B_0, C_0) + X_1 + T_1)^{\lll fs2} \quad (7.2)$$
$$C_3 = D_3 + (C_0 + f(D_3, A_3, B_0) + X_2 + T_2)^{\lll fs3} \quad (7.3)$$
$$B_3 = C_3 + (B_0 + f(C_3, D_3, A_3) + X_3 + T_3)^{\lll fs4} \quad (7.4)$$
$$A_7 = B_3 + (A_3 + f(B_3, C_3, D_3) + X_4 + T_4)^{\lll fs1} \quad (7.5)$$
$$D_7 = A_7 + (D_3 + f(A_7, B_3, C_3) + X_5 + T_5)^{\lll fs2} \quad (7.6)$$
$$C_7 = D_7 + (C_3 + f(D_7, A_7, B_3) + X_6 + T_6)^{\lll fs3} \quad (7.7)$$
$$B_7 = C_7 + (B_3 + f(C_7, D_7, A_7) + X_7 + T_7)^{\lll fs4} \quad (7.8)$$
$$A_{11} = B_7 + (A_7 + f(B_7, C_7, D_7) + X_8 + T_8)^{\lll fs1} \quad (7.9)$$
$$D_{11} = A_{11} + (D_7 + f(A_{11}, B_7, C_7) + X_9 + T_9)^{\lll fs2} \quad (7.10)$$
$$C_{11} = D_{11} + (C_7 + f(D_{11}, A_{11}, B_7) + X_{10} + T_{10})^{\lll fs3} \quad (7.11)$$
$$B_{11} = C_{11} + (B_7 + f(C_{11}, D_{11}, A_{11}) + X_{11} + T_{11})^{\lll fs4} \quad (7.12)$$
$$A_{15} = B_{11} + (A_{11} + f(B_{11}, C_{11}, D_{11}) + X_{12} + T_{12})^{\lll fs1} \quad (7.13)$$
$$D_{15} = A_{15} + (D_{11} + f(A_{15}, B_{11}, C_{11}) + X_{13} + T_{13})^{\lll fs2} \quad (7.14)$$
$$C_{15} = D_{15} + (C_{11} + f(D_{15}, A_{15}, B_{11}) + X_{14} + T_{14})^{\lll fs3} \quad (7.15)$$
$$B_{15} = C_{15} + (B_{11} + f(C_{15}, D_{15}, A_{15}) + X_{15} + T_{15})^{\lll fs4} \quad (7.16)$$

Round 2

The rotation constants used in round two are defined as:

$$gs1 = 5 \qquad gs3 = 14$$
$$gs2 = 9 \qquad gs4 = 20$$

The second round in the compress function is described by:

$$A_{19} = B_{15} + (A_{15} + g(B_{15}, C_{15}, D_{15}) + X_1 + T_{16})^{\lll gs1} \quad (7.17)$$

$$D_{19} = A_{19} + (D_{15} + g(A_{19}, B_{15}, C_{15}) + X_6 + T_{17})^{\lll gs2} \quad (7.18)$$

$$C_{19} = D_{19} + (C_{15} + g(D_{19}, A_{19}, B_{15}) + X_{11} + T_{18})^{\lll gs3} \quad (7.19)$$

$$B_{19} = C_{19} + (B_{15} + g(C_{19}, D_{19}, A_{19}) + X_0 + T_{19})^{\lll gs4} \quad (7.20)$$

$$A_{23} = B_{19} + (A_{19} + g(B_{19}, C_{19}, D_{19}) + X_5 + T_{20})^{\lll gs1} \quad (7.21)$$

$$D_{23} = A_{23} + (D_{19} + g(A_{23}, B_{19}, C_{19}) + X_{10} + T_{21})^{\lll gs2} \quad (7.22)$$

$$C_{23} = D_{23} + (C_{19} + g(D_{23}, A_{23}, B_{19}) + X_{15} + T_{22})^{\lll gs3} \quad (7.23)$$

$$B_{23} = C_{23} + (B_{19} + g(C_{23}, D_{23}, A_{23}) + X_4 + T_{23})^{\lll gs4} \quad (7.24)$$

$$A_{27} = B_{23} + (A_{23} + g(B_{23}, C_{23}, D_{23}) + X_9 + T_{24})^{\lll gs1} \quad (7.25)$$

$$D_{27} = A_{27} + (D_{23} + g(A_{27}, B_{23}, C_{23}) + X_{14} + T_{25})^{\lll gs2} \quad (7.26)$$

$$C_{27} = D_{27} + (C_{23} + g(D_{27}, A_{27}, B_{23}) + X_3 + T_{26})^{\lll gs3} \quad (7.27)$$

$$B_{27} = C_{27} + (B_{23} + g(C_{27}, D_{27}, A_{27}) + X_8 + T_{27})^{\lll gs4} \quad (7.28)$$

$$A_{31} = B_{27} + (A_{27} + g(B_{27}, C_{27}, D_{27}) + X_{13} + T_{28})^{\lll gs1} \quad (7.29)$$

$$D_{31} = A_{31} + (D_{27} + g(A_{31}, B_{27}, C_{27}) + X_2 + T_{29})^{\lll gs2} \quad (7.30)$$

$$C_{31} = D_{31} + (C_{27} + g(D_{31}, A_{31}, B_{27}) + X_7 + T_{30})^{\lll gs3} \quad (7.31)$$

$$B_{31} = C_{31} + (B_{27} + g(C_{31}, D_{31}, A_{31}) + X_{12} + T_{31})^{\lll gs4} \quad (7.32)$$

Round 3

The rotation constants used in the third round are defined as:

$$hs1 = 4 \qquad hs3 = 16$$

$$hs2 = 11 \qquad hs4 = 23$$

The third round of the MD5 compress function is defined by:

$$A_{35} = B_{31} + (A_{31} + h(B_{31}, C_{31}, D_{31}) + X_5 + T_{32})^{\lll hs1} \quad (7.33)$$
$$D_{35} = A_{35} + (D_{31} + h(A_{35}, B_{31}, C_{31}) + X_8 + T_{33})^{\lll hs2} \quad (7.34)$$
$$C_{35} = D_{35} + (C_{31} + h(D_{35}, A_{35}, B_{31}) + X_{11} + T_{34})^{\lll hs3} \quad (7.35)$$
$$B_{35} = C_{35} + (B_{31} + h(C_{35}, D_{35}, A_{35}) + X_{14} + T_{35})^{\lll hs4} \quad (7.36)$$
$$A_{39} = B_{35} + (A_{35} + h(B_{35}, C_{35}, D_{35}) + X_1 + T_{36})^{\lll hs1} \quad (7.37)$$
$$D_{39} = A_{39} + (D_{35} + h(A_{39}, B_{35}, C_{35}) + X_4 + T_{37})^{\lll hs2} \quad (7.38)$$
$$C_{39} = D_{39} + (C_{35} + h(D_{39}, A_{39}, B_{35}) + X_7 + T_{38})^{\lll hs3} \quad (7.39)$$
$$B_{39} = C_{39} + (B_{35} + h(C_{39}, D_{39}, A_{39}) + X_{10} + T_{39})^{\lll hs4} \quad (7.40)$$
$$A_{43} = B_{39} + (A_{39} + h(B_{39}, C_{39}, D_{39}) + X_{13} + T_{40})^{\lll hs1} \quad (7.41)$$
$$D_{43} = A_{43} + (D_{39} + h(A_{43}, B_{39}, C_{39}) + X_0 + T_{41})^{\lll hs2} \quad (7.42)$$
$$C_{43} = D_{43} + (C_{39} + h(D_{43}, A_{43}, B_{39}) + X_3 + T_{42})^{\lll hs3} \quad (7.43)$$
$$B_{43} = C_{43} + (B_{39} + h(C_{43}, D_{43}, A_{43}) + X_6 + T_{43})^{\lll hs4} \quad (7.44)$$
$$A_{47} = B_{43} + (A_{43} + h(B_{43}, C_{43}, D_{43}) + X_9 + T_{44})^{\lll hs1} \quad (7.45)$$
$$D_{47} = A_{47} + (D_{43} + h(A_{47}, B_{43}, C_{43}) + X_{12} + T_{45})^{\lll hs2} \quad (7.46)$$
$$C_{47} = D_{47} + (C_{43} + h(D_{47}, A_{47}, B_{43}) + X_{15} + T_{46})^{\lll hs3} \quad (7.47)$$
$$B_{47} = C_{47} + (B_{43} + h(C_{47}, D_{47}, A_{47}) + X_2 + T_{47})^{\lll hs4} \quad (7.48)$$

Round 4

The fourth round employs the following rotation constants:

$$is1 = 6 \qquad is3 = 15$$
$$ihs2 = 10 \qquad is4 = 21$$

The final round in the MD5 compress function is obtained from:

$$A_{51} = B_{47} + (A_{47} + i(B_{47}, C_{47}, D_{47}) + X_0 + T_{48})^{\lll is1} \quad (7.49)$$
$$D_{51} = A_{51} + (D_{47} + i(A_{51}, B_{47}, C_{47}) + X_7 + T_{49})^{\lll is2} \quad (7.50)$$
$$C_{51} = D_{51} + (C_{47} + i(D_{51}, A_{51}, B_{47}) + X_{14} + T_{50})^{\lll is3} \quad (7.51)$$
$$B_{51} = C_{51} + (B_{47} + i(C_{51}, D_{51}, A_{51}) + X_5 + T_{51})^{\lll is4} \quad (7.52)$$
$$A_{55} = B_{51} + (A_{51} + i(B_{51}, C_{51}, D_{51}) + X_{12} + T_{52})^{\lll is1} \quad (7.53)$$
$$D_{55} = A_{55} + (D_{51} + i(A_{55}, B_{51}, C_{51}) + X_3 + T_{53})^{\lll is2} \quad (7.54)$$
$$C_{55} = D_{55} + (C_{51} + i(D_{55}, A_{55}, B_{51}) + X_{10} + T_{54})^{\lll is3} \quad (7.55)$$
$$B_{55} = C_{55} + (B_{51} + i(C_{55}, D_{55}, A_{55}) + X_1 + T_{55})^{\lll is4} \quad (7.56)$$
$$A_{59} = B_{55} + (A_{55} + i(B_{55}, C_{55}, D_{55}) + X_8 + T_{56})^{\lll is1} \quad (7.57)$$
$$D_{59} = A_{59} + (D_{55} + i(A_{59}, B_{55}, C_{55}) + X_{15} + T_{57})^{\lll is2} \quad (7.58)$$
$$C_{59} = D_{59} + (C_{55} + i(D_{59}, A_{59}, B_{55}) + X_6 + T_{58})^{\lll is3} \quad (7.59)$$
$$B_{59} = C_{59} + (B_{55} + i(C_{59}, D_{59}, A_{59}) + X_{13} + T_{59})^{\lll is4} \quad (7.60)$$
$$A_{63} = B_{59} + (A_{59} + i(B_{59}, C_{59}, D_{59}) + X_4 + T_{60})^{\lll is1} \quad (7.61)$$
$$D_{63} = A_{63} + (D_{59} + i(A_{63}, B_{59}, C_{59}) + X_{11} + T_{61})^{\lll is2} \quad (7.62)$$
$$C_{63} = D_{63} + (C_{59} + i(D_{63}, A_{63}, B_{59}) + X_2 + T_{62})^{\lll is3} \quad (7.63)$$
$$B_{63} = C_{63} + (B_{59} + i(C_{63}, D_{63}, A_{63}) + X_9 + T_{63})^{\lll is4} \quad (7.64)$$

Update Variables

After completion of all three rounds, the hash variables are updated as shown below:

$$AA = A_0 = A_{47} + AA$$
$$BB = B_0 = B_{47} + BB$$
$$CC = C_0 = C_{47} + CC$$
$$DD = D_0 = D_{47} + DD$$

Once the last 512 bit message block have been processed, the final hash value is given by AA, BB, CC, and DD. If there are unprocessed message blocks remaining, A_0, B_0, C_0, and D_0 contains the new initial values for the next iteration of MD5.

7.5 ANALYSIS OF MD5

MD5 is a dedicated hash function as described in Section 7.4. As noted in Section 7.4 MD5 is an extension of MD4. MD5 was designed to be a more secure hash function and is therefore more conservative in design than MD4. In 1996 Dobbertin presented an attack on MD4, showing that it is possible to find collisions for MD4 in less than a second on a personal computer. The cryptanalysis of MD4 is presented in [14]. Additional work relating to the cryptanalysis of MD4 is presented in [17]. Dobbertin applied similar cryptanalytical techniques to RIPEMD and found that the first two rounds and last two rounds of RIPEMD are not collision resistant [54]. At the rump session of EUROCRYPT'96 it was announced that it is possible to find collisions for the compress function of MD5 [54]. An outline of this attack was published in [55]. In [12] it is stated that using these techniques, the attack requires approximately 10 hours on a personal computer with a Pentium processor. From these publications it appears as if the techniques employed in the cryptanalysis of MD5 are similar to those used on MD4 and RIPEMD.

The attack on MD5 as described in this chapter is based mostly on the notes of Antoon Bosselaers and the C source code developed by Dobbertin.

7.5.1 Notation

Before proceeding it is useful to introduce the following notation. Let

Z = Message word, chaining variable or a collection of chaining variables.

$\tilde{Z}$ = Alternative value for Z.

In addition, the following operator is defined:

$$Z^{\ggg Y} = \text{Circular rotation of } Z \text{ to the right by } Y \text{ bits.}$$

and

$$-Z^{\ggg Y} = -(Z^{\ggg Y}) \text{ bits.}$$

Thus, rotation has precedence over negation.

7.5.2 Outline of the Attack

The attack on MD5 is based on the assumption that all the message words are identical except for one message word X_i. The difference between X_i and $\tilde{X}_i$ is given by:

$$\tilde{X}_i = X_i + \Delta \tag{7.65}$$

where Δ is a 32 bit word with a small Hamming weight. For the attack on MD5 as described by Dobbertin in [12] and [55], the following choices are made:

$$\Delta = \texttt{0x00000200} \tag{7.66}$$

$$X_i = X_{14}. \tag{7.67}$$

It is stated in [55] that it may be possible to utilise other message words as well as other values for Δ. The message word X_{14} is used in equations (7.15) (7.26), (7.36) and (7.51) (once in each round). Using the notation defined in Section 7.5.1 the following definitions are presented.

$COMPRESS_x^y$ = Value of chaining variables after equations y to x of the compress function was applied to the message.

$\widetilde{COMPRESS}_x^y$ = Value of chaining variables after equations y to x of the compress function was applied to the modified message.

The attack is now reduced to find two messages so that:

$$COMPRESS_{7.26}^{7.15} - \widetilde{COMPRESS}_{7.26}^{7.15} = 0 \tag{7.68}$$

and

$$COMPRESS_{7.51}^{7.36} - \widetilde{COMPRESS}_{7.51}^{7.36} = 0 \tag{7.69}$$

If these conditions are met, so-called inner collisions are established. The attacker then has to find suitable message words which would link equation (7.26) to equation (7.36). From the above it appears that the attack on MD5 is a divide and conquer attack. Three phases are identified in this attack.

1. Find a message such that an inner collision is established for steps (7.15) to (7.26).
2. Find a message such that an inner collision is established for steps (7.36) to (7.51).
3. Find suitable message words to link the results obtained for the first two phases.

The inner collisions may be found by deriving two sets of difference equations (one for each step). These equations then has to be solved simultaneously due to the large overlap in words. At the same time a link between these equations has to be sought.

Each of these steps are discussed in a separate section in this chapter. Reference implementations are attached as appendices. An efficient technique exists to determine if solutions to certain Boolean expressions of a particular form exists. This technique is common to the first two phases of the attack and is therefore discussed in a separate section. Use is also made of the continuous approximation techniques used in the analysis of MD4 and RIPEMD.

7.5.3 Phase I: Inner Collisons for First Two Rounds

The first phase deals with the solution of a set of difference equations with the specific aim of finding an inner collision, such that equation (7.68) is satisfied. The difference equations are derived from equations (7.15) to (7.26) and are written as:

$$(C_{15} - D_{15})^{\ggg fs3} - (\tilde{C}_{15} - D_{15})^{\ggg fs3} = X_{14} - \tilde{X}_{14}. \tag{7.70}$$

$$(B_{15} - C_{15})^{\ggg fs4} - (\tilde{B}_{15} - \tilde{C}_{15})^{\ggg fs4} = f(C_{15}, D_{15}, A_{15}) - f(\tilde{C}_{15}, D_{15}, A_{15}) \tag{7.71}$$

$$(A_{19} - B_{15})^{\ggg gs1} - (\tilde{A}_{19} - \tilde{B}_{15})^{\ggg gs1} = g(B_{15}, C_{15}, D_{15}) - g(\tilde{B}_{15}, \tilde{C}_{15}, D_{15}) \tag{7.72}$$

$$(D_{19} - A_{19})^{\ggg gs2} - (\tilde{D}_{19} - \tilde{A}_{19})^{\ggg gs2} = g(A_{19}, B_{15}, C_{15}) - g(\tilde{A}_{19}, \tilde{B}_{15}, \tilde{C}_{15}) \tag{7.73}$$

$$(C_{19} - D_{19})^{\ggg gs3} - (\tilde{C}_{19} - \tilde{D}_{19})^{\ggg gs3} = C_{15} - \tilde{C}_{15} + g(D_{19}, A_{19}, B_{15}) - g(\tilde{D}_{19}, \tilde{A}_{19}, \tilde{B}_{15}) \tag{7.74}$$

$$(B_{19} - C_{19})^{\ggg gs4} - (\tilde{B}_{19} - \tilde{C}_{19})^{\ggg gs4} = B_{15} - \tilde{B}_{15} + g(C_{19}, D_{19}, A_{19}) - g(\tilde{C}_{19}, \tilde{D}_{19}, \tilde{A}_{19}) \tag{7.75}$$

$$(A_{23} - B_{19})^{\ggg gs1} - (\tilde{A}_{23} - \tilde{B}_{19})^{\ggg gs1} = A_{19} - \tilde{A}_{19} + g(B_{19}, C_{19}, D_{19}) - g(\tilde{B}_{19}, \tilde{C}_{19}, \tilde{D}_{19}) \quad (7.76)$$

$$(D_{23} - A_{23})^{\ggg gs2} - (\tilde{D}_{23} - \tilde{A}_{23})^{\ggg gs2} = D_{19} - \tilde{D}_{19} + g(A_{23}, B_{19}, C_{19}) - g(\tilde{A}_{23}, \tilde{B}_{19}, \tilde{C}_{19}) \quad (7.77)$$

$$(C_{23} - D_{23})^{\ggg gs3} - (C_{23} - \tilde{D}_{23})^{\ggg gs3} = C_{19} - \tilde{C}_{19} + g(D_{23}, A_{23}, B_{19}) - g(\tilde{D}_{23}, \tilde{A}_{23}, \tilde{B}_{19}) \quad (7.78)$$

$$0 = B_{19} - \tilde{B}_{19} + g(C_{23}, D_{23}, A_{23}) - g(C_{23}, \tilde{D}_{23}, \tilde{A}_{23}) \quad (7.79)$$

$$0 = A_{23} - \tilde{A}_{23} + g(B_{23}, C_{23}, D_{23}) - g(B_{23}, C_{23}, \tilde{D}_{23}) \quad (7.80)$$

$$0 = D_{23} - \tilde{D}_{23} + X_{14} - \tilde{X}_{14}. \quad (7.81)$$

Equations (7.70) to (7.81) may be simplified by making appropriate choices for certain chaining variables. From equation (7.81) the following condition is imposed on D_{23} and $\tilde{D}_{23}$:

$$\tilde{D}_{23} - D_{23} = X_{14} - \tilde{X}_{14} \quad (7.82)$$

from equation (7.66) equation (7.81) may be written as:

$$\tilde{D}_{23} - D_{23} = -1^{\lll 9}. \quad (7.83)$$

By setting $A_{23} = \tilde{A}_{23}$ and $B_{19} = \tilde{B}_{19}$ the conditions imposed by equations (7.80) and (7.79) are satisfied if:

$$g(B_{23}, C_{23}, \tilde{D}_{23}) = g(B_{23}, C_{23}, D_{23}) \quad (7.84)$$

and:

$$g(C_{23}, \tilde{D}_{23}, A_{23}) = g(C_{23}, D_{23}, A_{23}) \quad (7.85)$$

Equations (7.84) and (7.85) are satisfied relatively easily due to the low complexity and consequent ease with which the Boolean function $g()$ can be manipulated. If equations (7.84) and (7.85) holds, a high probability exists that:

$$g(D_{23}, A_{23}, B_{19}) = g(\tilde{D}_{23}, A_{23}, B_{19}).$$

If this is the case, equation (7.78) may be simplified to:

$$(C_{23} - D_{23})^{\ggg gs3} - (C_{23} - \tilde{D}_{23})^{\ggg gs3} = C_{19} - \tilde{C}_{19} \tag{7.86}$$

on the assumption that:

$$(C_{23} - D_{23})^{\ggg gs3} - (C_{23} - \tilde{D}_{23})^{\ggg gs3} = (\tilde{D}_{23} - D_{23})^{\ggg gs3}$$

The following condition is imposed on the relationship between C_{19} and $\tilde{C}_{19}$:

$$(\tilde{D}_{23} - D_{23})^{\ggg gs3} = C_{19} - \tilde{C}_{19}. \tag{7.87}$$

Thus:

$$C_{19} - \tilde{C}_{19} = -1^{\ggg 5}. \tag{7.88}$$

Given the above assumptions, the set of difference equations from (7.70) to (7.81) may be reduced to:

$$\begin{aligned}
(C_{15} - D_{15})^{\ggg fs3} - (\tilde{C}_{15} - D_{15})^{\ggg fs3} &= X_{14} - \tilde{X}_{14}. && (7.89)\\
(B_{15} - C_{15})^{\ggg fs4} - (\tilde{B}_{15} - \tilde{C}_{15})^{\ggg fs4} &= f(C_{15}, D_{15}, A_{15}) - f(\tilde{C}_{15}, D_{15}, A_{15}) && (7.90)\\
(A_{19} - B_{15})^{\ggg gs1} - (\tilde{A}_{19} - \tilde{B}_{15})^{\ggg gs1} &= g(B_{15}, C_{15}, D_{15}) - g(\tilde{B}_{15}, \tilde{C}_{15}, D_{15}) && (7.91)\\
(D_{19} - A_{19})^{\ggg gs2} - (\tilde{D}_{19} - \tilde{A}_{19})^{\ggg gs2} &= g(A_{19}, B_{15}, C_{15}) - g(\tilde{A}_{19}, \tilde{B}_{15}, \tilde{C}_{15}) && (7.92)\\
(C_{19} - D_{19})^{\ggg gs3} - (\tilde{C}_{19} - \tilde{D}_{19})^{\ggg gs3} &= C_{15} - \tilde{C}_{15} + g(D_{19}, A_{19}, B_{15}) - \\
& \quad g(\tilde{D}_{19}, \tilde{A}_{19}, \tilde{B}_{15}) && (7.93)\\
(B_{19} - C_{19})^{\ggg gs4} - (\tilde{B}_{19} - \tilde{C}_{19})^{\ggg gs4} &= B_{15} - \tilde{B}_{15} + g(C_{19}, D_{19}, A_{19}) - \\
& \quad g(\tilde{C}_{19}, \tilde{D}_{19}, \tilde{A}_{19}) && (7.94)\\
0 &= A_{19} - \tilde{A}_{19} + g(B_{19}, C_{19}, D_{19}) - \\
& \quad g(B_{19}, \tilde{C}_{19}, \tilde{D}_{19}) && (7.95)\\
(D_{23} - A_{23})^{\ggg gs2} - (\tilde{D}_{23} - A_{23})^{\ggg gs2} &= D_{19} - \tilde{D}_{19} + g(A_{23}, B_{19}, C_{19}) - \\
& \quad g(A_{23}, B_{19}, \tilde{C}_{19}). && (7.96)
\end{aligned}$$

Consider equation (7.89). By re-writing equation (7.89) as:

$$\tilde{C}_{15} = ((C_{15} - D_{15})^{\ggg fs3} + \tilde{X}_{14} - X_{14})^{\lll fs3} + D_{15}. \tag{7.97}$$

Valid solutions to C_{15}, $\tilde{C}_{15}$ and D_{15} may be found by setting:

$$C_{15} = D_{15} - 1 \tag{7.98}$$

substituting equations (7.98) and (7.66) in (7.97) the following expression for C_{15} is obtained.

$$\begin{aligned} \tilde{C}_{15} &= ((-1)^{\ggg fs3} + 1^{\lll 9})^{\lll fs3} + D_{15} && (7.99)\\ &= (-1 + 1^{\lll 9})^{\lll fs3} + D_{15} && (7.100)\\ &= (-1 + 1^{\lll 9})^{\lll 17} + D_{15}. && (7.101) \end{aligned}$$

With equations (7.98) and (7.101) in hand the difference between C_{15} and $\tilde{C}_{15}$ is obtained as:

$$\begin{aligned} \tilde{C}_{15} - C_{15} &= ((-1 + 1^{\lll 9})^{\lll 17} + D_{15}) - (D_{15} - 1) && (7.102)\\ \tilde{C}_{15} - C_{15} &= ((-1 + 1^{\lll 9})^{\lll 17} + 1 && (7.103)\\ &= \texttt{0x03FE0001}. && (7.104) \end{aligned}$$

By setting D_{15} to:

$$D_{15} = 1^{\lll 16} - 1^{\lll 25} - 1 \tag{7.105}$$

C_{15} and $\tilde{C}_{15}$ are easily computed as:

$$\begin{aligned} C_{15} &= \texttt{0xFE00FFFF} && (7.106)\\ \tilde{C}_{15} &= \texttt{0x01FF0000}. && (7.107) \end{aligned}$$

With equation (7.89) satisfied, further simplifications may be achieved by setting:

$$\begin{aligned} D_{19} &= \texttt{0x00000000} && (7.108)\\ \tilde{D}_{19} &= \texttt{0xFFFFFFFF}. && (7.109) \end{aligned}$$

These choices allow the simplification of equation (7.95) to:

$$\tilde{A}_{19} - A_{19} = C_{19} - B_{19}. \tag{7.110}$$

This simplification is a direct result of the manipulation of the bitwise Boolean function $g()$. With these simplifications the condition imposed by expression (7.68) may be written as:

$$COMPRESS_{7.22}^{7.15} - \widetilde{COMPRESS}_{7.22}^{7.15} = \{\varepsilon_1^1, \varepsilon_2^1, \varepsilon_3^1, \varepsilon_4^1\} \tag{7.111}$$

with:

$$\varepsilon_1^1 = \texttt{0x00000000} \tag{7.112}$$
$$\varepsilon_2^1 = \texttt{0x00000000} \tag{7.113}$$
$$\varepsilon_3^1 = \texttt{0x08000000} \tag{7.114}$$
$$\varepsilon_4^1 = \texttt{0xF8000000}. \tag{7.115}$$

Given the above choices, equations (7.89) – (7.96) may be re-written as:

$$0 = X_{14} - \tilde{X}_{14} - (C_{15} - D_{15})^{\ggg fs3} + (\tilde{C}_{15} - D_{15})^{\ggg fs3}. \tag{7.116}$$
$$0 = C_{15} - \tilde{C}_{15} + f(C_{15}, D_{15}, A_{15})^{\lll fs4} - f(\tilde{C}_{15}, D_{15}, A_{15})^{\lll fs4} - B_{15} + \tilde{B}_{15} \tag{7.117}$$
$$0 = B_{15} - \tilde{B}_{15} + g(B_{15}, C_{15}, D_{15})^{\lll gs3} - g(\tilde{B}_{15}, \tilde{C}_{15}, D_{15})^{\lll gs3} - A_{19} + \tilde{A}_{19} \tag{7.118}$$
$$0 = (D_{19} - A_{19})^{\ggg gs2} - (\tilde{D}_{19} - \tilde{A}_{19})^{\ggg gs2} - g(A_{19}, B_{15}, C_{15}) + g(\tilde{A}_{19}, \tilde{B}_{15}, \tilde{C}_{15}) \tag{7.119}$$
$$0 = (C_{19} - D_{19})^{\ggg gs3} - (\tilde{C}_{19} - \tilde{D}_{19})^{\ggg gs3} - C_{15} + \tilde{C}_{15} - g(D_{19}, A_{19}, B_{15}) + g(\tilde{D}_{19}, \tilde{A}_{19}, \tilde{B}_{15}) \tag{7.120}$$
$$0 = (A_{19} - \tilde{A}_{19})^{\ggg gs4} - (A_{19} - \tilde{A}_{19} - 1^{\lll 5})^{\ggg gs4} - B_{15} + \tilde{B}_{15} - g(C_{19}, D_{19}, A_{19}) + g(\tilde{C}_{19}, \tilde{D}_{19}, \tilde{A}_{19}) \tag{7.121}$$
$$0 = A_{19} - \tilde{A}_{19} + g(B_{19}, C_{19}, D_{19}) - g(B_{19}, \tilde{C}_{19}, \tilde{D}_{19}) \tag{7.122}$$
$$0 = D_{19} - \tilde{D}_{19} + g(A_{23}, B_{19}, C_{19}) - g(A_{23}, B_{19}, \tilde{C}_{19}) - (D_{23} - A_{23})^{\ggg gs2} + (\tilde{D}_{23} - A_{23})^{\ggg gs2}. \tag{7.123}$$

A solution to equations (7.89) to (7.96) will probably result in an inner collision for the first two rounds of MD5, given the assumptions and choices described earlier in this section. Algorithm 7.2 presents the procedure used by Dobbertin to find a solution to the set of difference equations as defined by equations (7.89) to (7.96).

Algorithm 7.2 *Construction of Inner Collision: Phase I*

1. *Make initial choices for* D_{15}, C_{15}, $\tilde{C}_{15}$, D_{19} *and* $\tilde{D}_{19}$ *as defined by equations (7.105) – (7.109). Set a counter* $n = 0$.

2. *Choose a random value for* A_{15}.

3. *Determine* $B_{15} - \tilde{B}_{15}$ *from equation (7.117).*

4. *Choose a random value for* B_{15} *and calculate* $\tilde{B}_{15}$ *from the result obtained in step 3. Proceed to step 6.*

5. *If* $n > 0$ *use the values for* $A_{15_{Basic}}$, $B_{15_{Basic}}$ *and* $C_{15_{Basic}}$ *as determined in step 14. Choose values with a small Hamming distance from these basic values for* A_{15}, B_{15} *and* C_{15} *and proceed to step 6.*

6. *Determine* $A_{19} - \tilde{A}_{19}$ *from equation (7.118).*

7. *Choose* C_{19} *at random and calculate* $\tilde{C}_{19}$ *according to equation (7.88).*

8. *From equation (7.121) determine all possible solutions for* A_{19}.

9. *Calculate all possible values for* $\tilde{A}_{19}$ *from the result obtained in step 6.*

10. *Determine, for each valid value of* A_{19} *and* $\tilde{A}_{19}$, *a possible solution to (7.119) by determining all possible valid values for* B_{15} *and* $\tilde{B}_{15}$ *(note that in future computations, these values for* B_{15} *and* $\tilde{B}_{15}$ *should be used, instead of the values computed in step 4).*

11. *Confirm whether equation (7.118) holds for the newly computed results obtained for* B_{15}, $\tilde{B}_{15}$, A_{19} *and* $\tilde{A}_{19}$.

12. *If equation (7.118) holds, confirm if equation (7.117) holds. If equation (7.118) does not hold, proceed to step 2.*

13. *If equation (7.117) holds, determine to which extent equation (7.120) holds.*

14. (a) *If the left* $4 \cdot n$ *bits of equation (7.120) are equal to zero, set* $n = n + 1$. *Preserve the current values for* A_{15}, B_{15} *and* C_{19} *as* $A_{15_{Basic}}$, $B_{15_{Basic}}$ *and* $C_{19_{Basic}}$.

    (b) *If* $0 < n < 8$ *and the left* $4 \cdot n$ *bits of equation (7.120) are not equal to zero, return to step 5.*

    (c) *If the left* $4 \cdot n$ *bits of equation (7.120) are not equal to zero, and* $n = 0$ *return to step 1.*

15. *If $n = 8$, determine if the assumptions made when reducing the set of difference equations from equations (7.70) – (7.81) to equations (7.116) – (7.123) holds. Confirm the validity of equations (7.78) – (7.81). Specifically confirm whether equation (7.123) holds. If all of these conditions are satisfied, there exists a high probability that an inner collision was found. If any of these conditions are not satisfied , return to step 1.*

Algorithm 7.2 may be modified to restart after a number of iterations to prevent dead ends, while searching for a solution. An algorithm which allows the construction of all possible solutions to an expression of a certain form is described in Section 7.5.6. Algorithm 7.2 produces an inner almost collision for the first two rounds of MD5 in less than one hour on a 120 MHz Pentium PC. An implementation of Algorithm 7.2 is attached as Appendix E.

7.5.4 Phase II: Inner Collisions for Last Two Rounds

The second phase is similar to the first phase insofar as it involves the solution of a set of difference equations, with the specific aim of finding an inner collision such that equation (7.69) is satisfied. The difference equations are derived from equations (7.36) to (7.51) and are written as:

$$(B_{35} - C_{35})^{\ggg hs4} - (\tilde{B}_{35} - C_{35})^{\ggg hs4} = X_{14} - \tilde{X}_{14} \quad (7.124)$$

$$(A_{39} - B_{35})^{\ggg hs1} - (\tilde{A}_{39} - \tilde{B}_{35})^{\ggg hs1} = h(B_{35}, C_{35}, D_{35}) - h(\tilde{B}_{35}, C_{35}, D_{35}) \quad (7.125)$$

$$(D_{39} - A_{39})^{\ggg hs2} - (\tilde{D}_{39} - \tilde{A}_{39})^{\ggg hs2} = h(A_{39}, B_{35}, C_{35}) - h(\tilde{A}_{39}, \tilde{B}_{35}, C_{35}) \quad (7.126)$$

$$(C_{39} - D_{39})^{\ggg hs3} - (\tilde{C}_{39} - \tilde{D}_{39})^{\ggg hs3} = h(D_{39}, A_{39}, B_{35}) - h(\tilde{D}_{39}, \tilde{A}_{39}, \tilde{B}_{35}) \quad (7.127)$$

$$(B_{39} - C_{39})^{\ggg hs4} - (\tilde{B}_{39} - \tilde{C}_{39})^{\ggg hs4} = B_{35} - \tilde{B}_{35} + h(C_{39}, D_{39}, A_{39}) - h(\tilde{C}_{39}, \tilde{D}_{39}, \tilde{A}_{39}) \quad (7.128)$$

$$(A_{43} - B_{39})^{\ggg hs1} - (\tilde{A}_{43} - \tilde{B}_{39})^{\ggg hs1} = A_{39} - \tilde{A}_{39} + h(B_{39}, C_{39}, D_{39}) - h(\tilde{B}_{39}, \tilde{C}_{39}, \tilde{D}_{39}) \quad (7.129)$$

$$(D_{43} - A_{43})^{\ggg hs2} - (\tilde{D}_{43} - \tilde{A}_{43})^{\ggg hs2} = D_{39} - \tilde{D}_{39} + h(A_{43}, B_{39}, C_{39}) - h(\tilde{A}_{43}, \tilde{B}_{39}, \tilde{C}_{39}) \quad (7.130)$$

$$(C_{43} - D_{43})^{\ggg hs3} - (\tilde{C}_{43} - \tilde{D}_{43})^{\ggg hs3} = C_{39} - \tilde{C}_{39} + h(D_{43}, A_{43}, B_{39}) - h(\tilde{D}_{43}, \tilde{A}_{43}, \tilde{B}_{39}) \quad (7.131)$$

$$
\begin{aligned}
(B_{43} - C_{43})^{\ggg hs4} - (\tilde{B}_{43} - \tilde{C}_{43})^{\ggg hs4} &= B_{39} - \tilde{B}_{39} + \\
&\quad h(C_{43}, D_{43}, A_{43}) - h(\tilde{C}_{43}, \tilde{D}_{43}, \tilde{A}_{43}) \quad (7.132) \\
(A_{47} - B_{43})^{\ggg hs1} - (\tilde{A}_{47} - \tilde{B}_{43})^{\ggg hs1} &= A_{43} - \tilde{A}_{43} + \\
&\quad h(B_{43}, C_{43}, D_{43}) - h(\tilde{B}_{43}, \tilde{C}_{43}, \tilde{D}_{43}) \quad (7.133) \\
(D_{47} - A_{47})^{\ggg hs2} - (\tilde{D}_{47} - \tilde{A}_{47})^{\ggg hs2} &= D_{43} - \tilde{D}_{43} + \\
&\quad h(A_{47}, B_{43}, C_{43}) - h(\tilde{A}_{47}, \tilde{B}_{43}, \tilde{C}_{43}) \quad (7.134) \\
(C_{47} - D_{47})^{\ggg hs3} - (\tilde{C}_{47} - \tilde{D}_{47})^{\ggg hs3} &= C_{43} - \tilde{C}_{43} + \\
&\quad h(D_{47}, A_{47}, B_{43}) - h(\tilde{D}_{47}, \tilde{A}_{47}, \tilde{B}_{43}) \quad (7.135) \\
(B_{47} - C_{47})^{\ggg hs4} - (B_{47} - \tilde{C}_{47})^{\ggg hs4} &= B_{43} - \tilde{B}_{43} + \\
&\quad h(C_{47}, D_{47}, A_{47}) - h(\tilde{C}_{47}, \tilde{D}_{47}, \tilde{A}_{47}) \quad (7.136) \\
0 &= A_{47} - \tilde{A}_{47} + \\
&\quad i(B_{47}, C_{47}, D_{47}) - i(B_{47}, \tilde{C}_{47}, \tilde{D}_{47}) \quad (7.137) \\
0 &= D_{47} - \tilde{D}_{47} + \\
&\quad i(A_{51}, B_{47}, C_{47}) - i(A_{51}, B_{47}, \tilde{C}_{47}) \quad (7.138) \\
0 &= C_{47} - \tilde{C}_{47} + X_{14} - \tilde{X}_{14}. \quad (7.139)
\end{aligned}
$$

Two stages are distinguished in this phase of the attack. The first deals with the differential properties of the set of equations and the second deals with the solution of the set of difference equations.

Differential Analysis

Before proceeding to find solutions to equations (7.124) to (7.139) the following observations are made. Dobbertin defines the differences for the chaining variables B_{39}, A_{43}, C_{43} and D_{43} as:

$$
\begin{aligned}
\tilde{A}_{43} &= A_{43} - \varepsilon_1^2 \quad (7.140) \\
\tilde{B}_{39} &= B_{39} - \varepsilon_2^2 \quad (7.141) \\
\tilde{C}_{43} &= C_{43} - \varepsilon_3^2 \quad (7.142) \\
\tilde{D}_{43} &= D_{43} - \varepsilon_4^2 \quad (7.143) \\
& \quad (7.144)
\end{aligned}
$$

where:

$$\varepsilon_1^2 = \texttt{0x40004000} \tag{7.145}$$
$$\varepsilon_2^2 = \texttt{0x80004000} \tag{7.146}$$
$$\varepsilon_3^2 = \texttt{0xFFFBFE00} \tag{7.147}$$
$$\varepsilon_4^2 = \texttt{0x40000200}. \tag{7.148}$$

The values chosen for ε_1^2, ε_2^2, ε_3^2, and ε_4^2 may be obtained from a differential attack. The differential attack is applied to equations (7.132) to (7.139). By starting at equation (7.139) and working back to equation (7.132), the following differential relationships are observed:

$$C_{47} - \tilde{C}_{47} = \tilde{X}_{14} - X_{14} \tag{7.149}$$
$$D_{47} - \tilde{D}_{47} = i(A_{51}, B_{47}, \tilde{C}_{47}) - i(A_{51}, B_{47}, C_{47}) \tag{7.150}$$
$$A_{47} - \tilde{A}_{47} = i(B_{47}, \tilde{C}_{47}, \tilde{D}_{47}) - i(B_{47}, C_{47}, D_{47}) \tag{7.151}$$
$$B_{43} - \tilde{B}_{43} = (B_{47} - C_{47})^{\ggg hs4} - (B_{47} - \tilde{C}_{47})^{\ggg hs4} - h(C_{47}, D_{47}, A_{47}) + h(\tilde{C}_{47}, \tilde{D}_{47}, \tilde{A}_{47}) \tag{7.152}$$
$$C_{43} - \tilde{C}_{43} = (C_{47} - D_{47})^{\ggg hs3} - (\tilde{C}_{47} - \tilde{D}_{47})^{\ggg hs3} - h(D_{47}, A_{47}, B_{43}) + h(\tilde{D}_{47}, \tilde{A}_{47}, \tilde{B}_{43}) \tag{7.153}$$
$$D_{43} - \tilde{D}_{43} = (D_{47} - A_{47})^{\ggg hs2} - (\tilde{D}_{47} - \tilde{A}_{47})^{\ggg hs2} - h(A_{47}, B_{43}, C_{43}) + h(\tilde{A}_{47}, \tilde{B}_{43}, \tilde{C}_{43}) \tag{7.154}$$
$$A_{43} - \tilde{A}_{43} = (A_{47} - B_{43})^{\ggg hs1} - (\tilde{A}_{47} - \tilde{B}_{43})^{\ggg hs1} - h(B_{43}, C_{43}, D_{43}) + h(\tilde{B}_{43}, \tilde{C}_{43}, \tilde{D}_{43}) \tag{7.155}$$
$$B_{39} - \tilde{B}_{39} = (B_{43} - C_{43})^{\ggg hs4} - (\tilde{B}_{43} - \tilde{C}_{43})^{\ggg hs4} - h(C_{43}, D_{43}, A_{43}) + h(\tilde{C}_{43}, \tilde{D}_{43}, \tilde{A}_{43}) \tag{7.156}$$

From equation (7.149) it is observed that:

$$C_{47} - \tilde{C}_{47} = \Delta = \texttt{0x00000200} \tag{7.157}$$

By setting A_{51}, B_{47} and $\tilde{C}_{47}$ to zero and observing the difference obtained from equation (7.149), equation (7.150) yields:

$$D_{47} - \tilde{D}_{47} = \texttt{0x00000200}. \tag{7.158}$$

For randomly chosen values for A_{51}, B_{47} and $\tilde{C}_{47}$ the relationship defined in equation (7.158) holds with a probability of 19%. Consider equation (7.151). By setting B_{47}, $\tilde{C}_{47}$ and $\tilde{D}_{47}$ to zero and by observing the differential values defined by equations (7.157) and (7.158) the following difference is obtained from equation (7.151).

$$A_{47} - \tilde{A}_{47} = \texttt{0x00000000}. \tag{7.159}$$

For randomly chosen values of B_{47}, $\tilde{C}_{47}$ and $\tilde{D}_{47}$ the relationship defined by equation (7.159) holds with a 19% probability. By setting $\tilde{A}_{47}$ to one and B_{47}, $\tilde{C}_{47}$ and $\tilde{D}_{47}$ to zero, we obtain the following differential from equation (7.152).[1]

$$B_{43} - \tilde{B}_{43} = \texttt{0xFFFC0000}. \tag{7.160}$$

For randomly chosen values of $\tilde{A}_{47}$, B_{47}, $\tilde{C}_{47}$ and $\tilde{D}_{47}$ the relationship in equation (7.160) holds with a probability of 33%. By setting $\tilde{C}_{47}$ and $\tilde{D}_{47}$ to zero and by setting $\tilde{A}_{47}$ and $\tilde{B}_{43}$ to -1, the following relationship is observed from equation (7.153).

$$C_{43} - \tilde{C}_{43} = \texttt{0xFFFBFE00}. \tag{7.161}$$

The relationship in equation (7.161) holds with a probability of 11% if the values for $\tilde{C}_{47}$, $\tilde{D}_{47}$, $\tilde{A}_{47}$ and $\tilde{B}_{43}$ are chosen at random and the previously determined differential values are observed. From equation (7.154) the following differential relationship is observed by setting $\tilde{D}_{47}$ equal to zero, $\tilde{A}_{47}$ and $\tilde{B}_{47}$ to -1 and $\tilde{C}_{43}$ to $-\texttt{0xFFFBFE00}$.

$$D_{43} - \tilde{D}_{43} = \texttt{0x40000200}. \tag{7.162}$$

This relationship holds with a probability of 8% for randomly chosen values for $\tilde{D}_{47}$, $\tilde{A}_{47}$, $\tilde{B}_{47}$ and $\tilde{C}_{43}$. If the following settings are made:

$$\begin{aligned}
\tilde{A}_{47} &= \texttt{0x00000000} \\
\tilde{B}_{43} &= \texttt{0x00000000} \\
\tilde{C}_{43} &= -\texttt{0xFFFBFE00} \\
\tilde{D}_{43} &= -(\texttt{0x40000200} + \texttt{0x40000200})
\end{aligned}$$

[1] This relationship only holds when $(-C_{47})^{\gg hs4} = -(C_{47})^{\gg hs4}$

Equation (7.155) yields the following differential.

$$A_{43} - \tilde{A}_{43} = 0\text{x}40004000. \tag{7.163}$$

Equation (7.163) holds with a probability of 5% if $\tilde{A}_{47}$, $\tilde{B}_{43}$, $\tilde{C}_{43}$ and $\tilde{D}_{43}$ are chosen at random and the previously computed differentials are used. Consider equation (7.156). By setting:

$$\begin{aligned}
\tilde{B}_{43} &= 0\text{x}00000000 \\
\tilde{C}_{43} &= -0\text{x}\text{FFFBFE00} \\
\tilde{D}_{43} &= -(0\text{x}40000200 + 0\text{x}40000200)) \\
\tilde{A}_{43} &= 0\text{x}40000200 - 0\text{x}40004000
\end{aligned}$$

equation (7.156) yields the following differential:

$$B_{39} - \tilde{B}_{39} = 0\text{x}80084000. \tag{7.164}$$

This equation holds with a probability of 1.8% for randomly chosen values for $\tilde{B}_{43}$, $\tilde{C}_{43}$, $\tilde{D}_{43}$ and $\tilde{A}_{43}$.

Let $\Pr(\Delta_{i,j,k,l})$ denote the probability that a differential associated with a specific step holds. The differential attack is summarised in Table 7.1.

$\{i,j,k,l\}$	$A_i - \tilde{A}_i$	$B_j - \tilde{B}_j$	$C_k - \tilde{C}_k$	$D_l - \tilde{D}_l$	$\Pr(\Delta_{i,j,k,l})$
$\{51,47,51,51\}$	0	0	0	0	—
$\{51,47,47,51\}$	0	0	**0x00000200**	0	1.0
$\{51,47,47,47\}$	0	0	0x00000200	**0x00000200**	0.19
$\{47,47,47,47\}$	**0x00000000**	0	0x00000200	0x00000200	0.19
$\{47,43,47,47\}$	0x00000000	**0xFFFC0000**	0x00000200	0x00000200	0.33
$\{47,43,43,47\}$	0x00000000	0xFFFC0000	**0xFFFBFE00**	0x00000200	0.11
$\{47,43,43,43\}$	0x00000000	0xFFFC0000	0xFFFBFE00	**0x40000200**	0.08
$\{43,43,43,43\}$	**0x40004000**	0xFFFC0000	0xFFFBFE00	0x40000200	0.05
$\{43,39,43,43\}$	0x40004000	**0x80084000**	0xFFFBFE00	0x40000200	0.018

Table 7.1: Differential Attack: MD5

Let $\Pr(\Delta)$ denote the probability that the differential pattern in Table 7.1 holds. Assuming statistical independence between successive steps the probability that the differential attack holds is approximately:

$$\Pr(\Delta) = 1^{-7}.$$

However, a practical implementation has shown that the assumption of statistical independence is not valid. There exists a high probability that, given that one differential is satisfied, that the following differential will also be satisfied. A practical implementation of the differential attack has shown that the probability that the differential attack described in Table 7.1 holds is approximately:

$$\Pr(\Delta) = \frac{1}{26000}.$$

It appears that there exists a number of differentials which may be used instead of those shown in Table 7.1.

If the following constraints are imposed on B_{39}, A_{43}, C_{43} and D_{43} the probability that the differential attack holds is increased. Choose A_{43} at random except for bits 10, 15 and 31 which should be set to one and bit 19 which should be set to zero. C_{43} may be chosen at random except for bits 15 and 31 which should be set to one and bits 10 and 19 which should be set to zero[2]. Let D_{43} be specified by:

$$D_{43} = (A_{43} + (\varepsilon_4^2 - \varepsilon_1^2)^{\lll 31}) \oplus \gamma_1 \tag{7.165}$$

where γ_1 is a 32 bit binary vector with a low Hamming weight. In a similar fashion let B_{39} be defined as:

$$B_{39} = (\varepsilon_2^2 - 1)^{\lll 31}) \oplus \gamma_2 \tag{7.166}$$

where γ_2 denotes a 32 bit binary vector of low Hamming weight ($\gamma_1 \neq \gamma_2$.

The condition imposed by expression (7.69) for determining an internal collision may be re-written as:

$$0 = COMPRESS_{7.51}^{7.44} - \widetilde{COMPRESS}_{7.51}^{7.44} \tag{7.167}$$

[2]The bits in the 32 bit words are numbered from the LSB to the MSB

If the constraints described above are imposed on B_{39}, A_{43}, C_{43} and D_{43} the probability that expression 7.167 holds is approximately:

$$\Pr(\Delta) = \frac{1}{1500}.$$

Establishing an Inner Collision

Thus, the differential attack described here allows the problem of finding a solution to expression (7.69) to be reduced to finding a solution to the following expression:

$$COMPRESS_{7.44}^{7.36} - \widetilde{COMPRESS}_{7.44}^{7.36} = \{\varepsilon_1^2, \varepsilon_2^2, \varepsilon_3^2, \varepsilon_4^2\}. \tag{7.168}$$

Thus the set of difference equations which has to be solved, may be simplified to:

$$\begin{aligned}
(B_{35} - C_{35})^{\ggg hs4} - (\tilde{B}_{35} - C_{35})^{\ggg hs4} &= X_{14} - \tilde{X}_{14} && (7.169)\\
(A_{39} - B_{35})^{\ggg hs1} - (\tilde{A}_{39} - \tilde{B}_{35})^{\ggg hs1} &= h(B_{35}, C_{35}, D_{35}) - h(\tilde{B}_{35}, C_{35}, D_{35}) && (7.170)\\
(D_{39} - A_{39})^{\ggg hs2} - (\tilde{D}_{39} - \tilde{A}_{39})^{\ggg hs2} &= h(A_{39}, B_{35}, C_{35}) - h(\tilde{A}_{39}, \tilde{B}_{35}, C_{35}) && (7.171)\\
(C_{39} - D_{39})^{\ggg hs3} - (\tilde{C}_{39} - \tilde{D}_{39})^{\ggg hs3} &= h(D_{39}, A_{39}, B_{35}) - h(\tilde{D}_{39}, \tilde{A}_{39}, \tilde{B}_{35}) && (7.172)\\
(B_{39} - C_{39})^{\ggg hs4} - (\tilde{B}_{39} - \tilde{C}_{39})^{\ggg hs4} &= B_{35} - \tilde{B}_{35} + && \\
& \quad h(C_{39}, D_{39}, A_{39}) - h(\tilde{C}_{39}, \tilde{D}_{39}, \tilde{A}_{39}) && (7.173)\\
(A_{43} - B_{39})^{\ggg hs1} - (\tilde{A}_{43} - \tilde{B}_{39})^{\ggg hs1} &= A_{39} - \tilde{A}_{39} + && \\
& \quad h(B_{39}, C_{39}, D_{39}) - h(\tilde{B}_{39}, \tilde{C}_{39}, \tilde{D}_{39}) && (7.174)\\
(D_{43} - A_{43})^{\ggg hs2} - (\tilde{D}_{43} - \tilde{A}_{43})^{\ggg hs2} &= D_{39} - \tilde{D}_{39} + && \\
& \quad h(A_{43}, B_{39}, C_{39}) - h(\tilde{A}_{43}, \tilde{B}_{39}, \tilde{C}_{39}) && (7.175)\\
(C_{43} - D_{43})^{\ggg hs3} - (\tilde{C}_{43} - \tilde{D}_{43})^{\ggg hs3} &= C_{39} - \tilde{C}_{39} + && \\
& \quad h(D_{43}, A_{43}, B_{39}) - h(\tilde{D}_{43}, \tilde{A}_{43}, \tilde{B}_{39}). && (7.176)
\end{aligned}$$

Additional simplifications to equations (7.124) – (7.139) can be achieved by making the following observations:

$$\begin{aligned}
(B_{35} - C_{35})^{\ggg hs4} - (\tilde{B}_{35} - C_{35})^{\ggg hs4} &= X_{14} - \tilde{X}_{14} && (7.177)\\
(B_{35} - C_{35})^{\ggg 23} - (\tilde{B}_{35} - C_{35})^{\ggg 23} &= -1^{\lll 9} && (7.178)\\
B_{35} &= ((\tilde{B}_{35} - C_{35})^{\ggg 23} - 1^{\lll 9})^{\lll 23} + C_{35}. && (7.179)
\end{aligned}$$

Equation (7.179) may, with high probability, be written as:

$$B_{35} = ((\tilde{B}_{35} - C_{35})^{\lll 9} - 1^{\lll 9})^{\lll 23} + C_{35}. \quad (7.180)$$

$$B_{35} = \tilde{B}_{35} - C_{35} - 1 + C_{35}. \quad (7.181)$$

$$B_{35} = \tilde{B}_{35} - 1. \quad (7.182)$$

By setting

$$B_{35} = -1. \quad (7.183)$$

and

$$\tilde{B}_{35} = 0. \quad (7.184)$$

equation (7.179) is satisfied. These choices inherently simplify equation (7.125) as follows.

$$(A_{39} - B_{35})^{\ggg hs1} - (\tilde{A}_{39} - \tilde{B}_{35})^{\ggg hs1} = h(-1, C_{35}, D_{35}) - h(0, C_{35}, D_{35}) \quad (7.185)$$

$$(A_{39} - B_{35})^{\ggg hs1} - (\tilde{A}_{39} - \tilde{B}_{35})^{\ggg hs1} = \overline{C_{35} \oplus D_{35}} - C_{35} \oplus D_{35} \quad (7.186)$$

with the use of the following identity:

$$X \oplus Y + \overline{X \oplus Y} = 1 \quad (7.187)$$

$$\overline{X \oplus Y} = 1 - X \oplus Y \quad (7.188)$$

equation (7.186) reduces to:

$$1 - (A_{39} - B_{35})^{\ggg hs1} + (\tilde{A}_{39} - \tilde{B}_{35})^{\ggg hs1} = 2 \cdot (C_{35} \oplus D_{35}) \quad (7.189)$$

$$C_{35} \oplus D_{35} = (1 - (A_{39} - B_{35})^{\ggg hs1} - (\tilde{A}_{39} - \tilde{B}_{35})^{\ggg hs1})^{\ggg 1} \quad (7.190)$$

thus D_{35} may be obtained from:[3]

$$D_{35} = (1 - (A_{39} - B_{35})^{\ggg hs1} - (\tilde{A}_{39} - \tilde{B}_{35})^{\ggg hs1})^{\ggg 1} \oplus C_{35}. \quad (7.191)$$

[3]Note that equation 7.191 only holds if the right hand side of equation (7.189) is even.

Now consider equation (7.172). Remember that:

$$h(A, B, C) = A \oplus B \oplus C.$$

Thus equation (7.172) may be written as:

$$(C_{39} - D_{39})^{\ggg hs3} - (\tilde{C}_{39} - \tilde{D}_{39})^{\ggg hs3} = D_{39} \oplus A_{39} \oplus B_{35} - \tilde{D}_{39} \oplus \tilde{A}_{39} \oplus \tilde{B}_{35} \quad (7.192)$$

Let:

$$X = D_{39} \oplus A_{39} \oplus B_{35} \quad (7.193)$$

$$\tilde{X} = \tilde{D}_{39} \oplus \tilde{A}_{39} \oplus \tilde{B}_{35} \quad (7.194)$$

From equations (7.183) and (7.184) equations (7.193) and (7.194) may be simplified to:

$$X = \overline{D_{39} \oplus A_{39}} \quad (7.195)$$

$$\tilde{X} = \tilde{D}_{39} \oplus \tilde{A}_{39} \quad (7.196)$$

Equation 7.192 may now be written as:

$$X - \tilde{X} = (C_{39} - D_{39})^{\ggg hs3} - (\tilde{C}_{39} - \tilde{D}_{39})^{\ggg hs3}. \quad (7.197)$$

Equation 7.197 may be written, with a probability of approximately 25% as:

$$X - \tilde{X} = ((C_{39} - \tilde{C}_{39}) - (D_{39} - \tilde{D}_{39}))^{\ggg hs3}. \quad (7.198)$$

Equation (7.173) may now be written as:

$$(B_{39} - C_{39})^{\ggg hs4} - (\tilde{B}_{39} - \tilde{C}_{39})^{\ggg hs4} = B_{35} - \tilde{B}_{35} + h(C_{39}, \overline{X}, 0) - h(\tilde{C}_{39}, \tilde{X}, 0) \quad (7.199)$$

Equation (7.174) may now be written as:

$$(A_{43} - B_{39})^{\ggg hs1} - (\tilde{A}_{43} - \tilde{B}_{39})^{\ggg hs1} = (\overline{X} \oplus D_{39} - \tilde{X} \oplus \tilde{D}_{39}) + h(B_{39}, C_{39}, D_{39}) - h(\tilde{B}_{39}, \tilde{C}_{39}, \tilde{D}_{39}) \quad (7.200)$$

Thus:

$$A_{39} = \overline{X} \oplus D_{39} \tag{7.201}$$

$$\tilde{A}_{39} = \tilde{X} \oplus \tilde{D}_{39} \tag{7.202}$$

Dobbertin proposes that Algorithm 7.3 is used to determine a solution to expression (7.168).

Algorithm 7.3 *Construction of Inner Collision: Phase II*

1. *Choose B_{35} and $\tilde{B}_{35}$ as specified by equations (7.183) and (7.184)*
2. *Find values for $A_{43}, B_{39}, C_{43}, D_{43}$ which will satisfy the differential attack summarised in Table 7.1.*
3. *Determine $C_{39} - \tilde{C}_{39}$ from equation (7.176).*
4. *Choose a random value for C_{39} and determine $\tilde{C}_{39}$ from the result obtained in step 3. Now determine $D_{39} - \tilde{D}_{39}$ from equation (7.176).*
5. *With $D_{39} - \tilde{D}_{39}$ in hand determine all possible solutions to C_{39} and $\tilde{C}_{39}$ using an iterative search procedure.*
6. *Calculate $X - \tilde{X}$ given that the assumptions in (7.198) holds.*
7. *From equation (7.199) determine all valid solutions to X and $\tilde{X}$.*
8. *Determine solutions for D_{39} and $\tilde{D}_{39}$ from equation (7.200) using an iterative approach.*
9. *A_{39} and $\tilde{A}_{39}$ may be calculated from equations (7.201) and (7.202).*
10. *Determine if the assumption in equation (7.198) holds.*
    (a) *If the assumption in expression (7.198) does not hold return to step 1.*
    (b) *If the assumption in (7.198) holds, determine all possible solutions to C_{35} using equation (7.171).*
11. *If solutions for C_{35} exist determine D_{35} from (7.191).*
12. *If a valid solution D_{35} is found an inner collision for the second round was found.*

Dobbertin's implementation of this attack is attached as Appendix E.2.

7.5.5 Phase III: Establishing a Connection

The third phase requires that the solutions to the sets of equations obtained from the previous two phases are connected. When commencing with the third phase of the attack, the chaining variables C_{15}, B_{15}, A_{19}, D_{19}, C_{19} B_{19} and A_{23} are known from phase one of the attack. The chaining variables C_{35}, D_{35}, A_{39}, B_{39}, C_{39}, D_{39}, A_{43} and D_{43} are known from phase two of the attack. A number of message words and chaining variables may now be computed. Message words X_1, X_6, X_{11}, X_0 and X_5 may be computed from equations (7.17), (7.18), (7.19), (7.20) and (7.21). Likewise message words X_4, X_7, X_{10} and X_{13} may be obtained from equations (7.38), (7.39), (7.40) and (7.41). In addition chaining variables D_{23} and D_{43} are be obtained from equations (7.22) and (7.41) respectively.

A connection is obtained if solutions to X_2, X_3, X_8 and X_{12} is found such that equations (7.203)–(7.203) holds.

$$C_{27} = D_{27} + (C_{23} + g(D_{27}, A_{27}, B_{23}) + X_3 + T_{26})^{\lll gs3} \quad (7.203)$$

$$B_{27} = C_{27} + (B_{23} + g(C_{27}, D_{27}, A_{27}) + X_8 + T_{27})^{\lll gs4} \quad (7.204)$$

$$A_{31} = B_{27} + (A_{27} + g(B_{27}, C_{27}, D_{27}) + X_{13} + T_{28})^{\lll gs1} \quad (7.205)$$

$$D_{31} = A_{31} + (D_{27} + g(A_{31}, B_{27}, C_{27}) + X_2 + T_{29})^{\lll gs2} \quad (7.206)$$

$$C_{31} = D_{31} + (C_{27} + g(D_{31}, A_{31}, B_{27}) + X_7 + T_{30})^{\lll gs3} \quad (7.207)$$

$$B_{31} = C_{31} + (B_{27} + g(C_{31}, D_{31}, A_{31}) + X_{12} + T_{31})^{\lll gs4} \quad (7.208)$$

$$A_{35} = B_{31} + (A_{31} + h(B_{31}, C_{31}, D_{31}) + X_5 + T_{32})^{\lll hs1} \quad (7.209)$$

$$D_{35} = A_{35} + (D_{31} + h(A_{35}, B_{31}, C_{31}) + X_8 + T_{33})^{\lll hs2} \quad (7.210)$$

$$C_{35} = D_{35} + (C_{31} + h(D_{35}, A_{35}, B_{31}) + X_{11} + T_{34})^{\lll hs3} \quad (7.211)$$

Equations (7.203)–(7.203) are bounded by the chaining variables A_{27}, B_{23}, C_{23} and D_{23} as obtained from phase one of the attack, as well as by chaining variables A_{35}, B_{31}, C_{35} and D_{35}, obtained from phase two of the attack. Thus the only chaining variables which may be manipulated without affecting the previous phases of the attack, are A_{31}, B_{27}, C_{27} and D_{31}. An additional constraint is imposed by the fact that X_5, X_7, X_{11} and X_{13} are determined by phase one and two of the attack. This leaves only four message words namely X_2, X_3, X_8 and X_{12}, which may be used to establish a connection. The fact that expression (7.68) is simplified to expression (7.111) gives an attacker additional degrees of freedom and allows, to a limited extent, the manipulation of chaining variables B_{23} C_{23} and D_{27}. Note that D_{23}, which is associated with the first stage of the attack, depends on X_{10} which is obtained from

the second phase of the attack. Thus the chaining variables in the two previous phases are interdependent on each other. This requires that the results obtained from the previous phases has to be manipulated simultaneously when attempting to establish a connection between the two phases.

Three stages are identified in the third phase of the attack. The first stage is concerned with finding suitable values for the chaining variables C_{23}, B_{23}, A_{27} and D_{27}. The second stage requires that a connection be made between phase one and two. During the third stage it is required that the existence of an inner collision obtained for the second phase is verified.

Stage 1

Remember that the attack on the first phase is simplified in Section 7.5.3 by requiring that condition (7.111) instead of condition (7.68) has to be met. From Section 7.5.3 it is known that if condition (7.111) is met, condition (7.68) holds with a high probability. The following procedure is proposed by Dobbertin to find suitable values for C_{23}, B_{23}, A_{27} and D_{27}.

Algorithm 7.4 *Stage 1: Determine if inner collision for Phase I*

1. *Determine D_{23} from equation (7.22).*
2. *Choose X_{15} at random.*
3. *Determine C_{23} and B_{23} from equations (7.23) and (7.24).*
4. *Set A_{23} equal to B_{23}.*
5. *Calculate X_9 from equation (7.25)*
6. *Set D_{27} = `0xFFFFFFFF`.*
7. *Calculate X_{14} from (7.26).*
8. *Determine if expression (7.68) holds.*
   (a) *If condition (7.68) does not hold, restart from step 1.*
   (b) *If condition (7.68) does hold, proceed to stage two of phase three.*

The choice for D_{27} ensures that the following relationship holds:

$$B_{27} = g(B_{27}, C_{27}, D_{27}). \tag{7.212}$$

Stage 2

The second stage is directly concerned with establishing a connection between the two phases. The following simplification to equation (7.209) is of particular importance:

$$(A_{35} - B_{31})^{\ggg hs1} - A_{31} - X_5 - T_{32} = h(B_{31}, C_{31}, D_{31}) \tag{7.213}$$

$$(A_{35} - B_{31})^{\ggg hs1} - A_{31} - X_5 - T_{32} = B_{31} \oplus C_{31} \oplus D_{31} \tag{7.214}$$

$$D_{31} = ((A_{35} - B_{31})^{\ggg hs1} - A_{31} - X_5 - T_{32}) \oplus B_{31} \oplus C_{31}. \tag{7.215}$$

Dobbertin proposes the following algorithm to establish a connection between the two phases.

Algorithm 7.5 *Second Stage: Establish Connection*

1. *Set* $B_{31} = B_{35} - X_{14}$.
2. *Determine* C_{31} *from equation (7.211).*
3. *Choose a random value for* $B_{27_{Basic}}$. *Choose* B_{27} *with a Hamming distance of 1 from* $B_{27_{Basic}}$.
4. *Calculate* A_{31} *from equation (7.205).*
5. *Calculate* D_{31} *from equation (7.215).*
6. *Determine* C_{27} *from equation (7.207).*
7. *Calculate* X_8^1 *from equation (7.204) and* X_8^2 *from equation (7.210).*
   (a) *If equations (7.204) and (7.210) yields the same result for* X_8 *a connection was found.*

(b) *If equations (7.204) and (7.210) yields different results for X_8 continue from step 8.*

8. *Determine the Hamming distance between the two values calculated for X_8.*

   (a) *If the current Hamming distance is less than the previous Hamming distance, save the current Hamming distance and set $B_{27_{Basic}} = B_{27}$. Return to step 4.*

   (b) *If the current Hamming distance is not less than the previous Hamming distance, proceed from step 9.*

9. *Set: $C_{23} = C_{23} - (X_8^1 - X_8^2)$, $B_{23} = B_{23} - (X_8^1 - X_8^2)$ and recompute D_{27}. Confirm if equation (7.212) holds.*

   (a) *If equation (7.212) does not hold, restart from step 1.*

   (b) *If equation (7.212) does hold, proceed to step 10.*

10. *Determine if the changes made in step 9 still allows an inner collision for the first phase.*

    (a) *If not, restart from Algorithm 7.4.*

    (b) *If an internal collision is found, restart from step 4.*

Step 8 in Algorithm 7.5 is reminiscent of the continuous approximation used in phase one of the attack on MD5.

Upon successful completion of Algorithm 7.5 X_2, X_3, X_8 and X_{12} may be computed from equations (7.206), (7.203), (7.204) and (7.208) respectively. Once the first two stages have been completed, only the third stage of the third phase remains.

Stage 3

The third stage requires that the existence of an inner collision for the first two stages be confirmed. If an inner collision is not found, the attacker should restart from stage 1 of the third phase. If the inner collision for the second phase exists, a collision for the round function of MD5 was found. The message words and initial value for which the collision holds may now be calculated.

Example

The following collision was constructed for the round function of MD5 The initial value for which a collision was found is given by:

$$A0 = \texttt{0xF7987AA4}$$
$$B0 = \texttt{0x6EF00D2B}$$
$$C0 = \texttt{0xCAFBC0A2}$$
$$D0 = \texttt{0x7678589B}$$

The message is defined by:

$$X_0 = \texttt{0xAA1DDA5E} \qquad X_8 = \texttt{0x4C1E82F6}$$
$$X_1 = \texttt{0xD97ABFF5} \qquad X_9 = \texttt{0xAF5B46C9}$$
$$X_2 = \texttt{0xA5CA745B} \qquad X_{10} = \texttt{0x236BB992}$$
$$X_3 = \texttt{0xC9DDCECB} \qquad X_{11} = \texttt{0x6B7A669B}$$
$$X_4 = \texttt{0x1006363E} \qquad X_{12} = \texttt{0x40CC7121}$$
$$X_5 = \texttt{0x7218209D} \qquad X_{13} = \texttt{0xD93E0972}$$
$$X_6 = \texttt{0xE01C135D} \qquad X_{14} = \texttt{0x95FACCCD}$$
$$X_7 = \texttt{0x9DA64D0E} \qquad X_{15} = \texttt{0x72409780}$$

with

$$\tilde{X}_{14} = X_{14} + 1^{\lll 9}.$$

The common hash value obtained for these two messages is:

$$\text{Common Digest} = \texttt{0x88963A3F 0x4B40C08A 0xF529663D 0x0B410DAD}.$$

Note that due to the overlap in message words (specifically X_{10}) and the consequent interdependence of the chaining variables in phases one and two of the attack, considerable effort is required to find solutions to both phase one and two which will allow a successful connection

to be made. Given a solution to the second phase the probability that the solution to the first phase will still result in an inner collision for the first phase is estimated at 25%.

The techniques used to construct a connection described in this section, manipulates the results obtained from phases one and two of the attack simultaneously. However the potential effort required to solve the sets of equations simultaneously is avoided by finding a solution to expression (7.111) instead of (7.68). This allows the limited manipulation of the chaining variables B_{23}, C_{23} and A_{27}. Thus, the chaining variables which are most influenced by the interdependence between phase one and two, may be manipulated to a limited extent. An implementation of the third phase of the attack is attached as Appendix E.3.

7.5.6 Determining if Solutions Exist

In Sections 7.5.3 and 7.5.4 it is assumed that an algorithm exists for determining if solutions to certain Boolean expressions exist. This section gives a description of such an algorithm as used by Dobbertin in the analysis of MD5. Dobbertin's attack requires the solution of sets of difference equations. From the previous sections it is observed that these equations are of the form:

$$T = f(a_1, b_1, x_1) - f(a_2, b_2, x_2) \tag{7.216}$$

or:

$$0 = f(a_1, b_1, x_1) - f(a_2, b_2, x_2) - T \tag{7.217}$$

with $f()$ a bitwise Boolean function which operates on binary words of length l. The variables x_1 and x_2 are related by:

$$x_2 = x_1 + \delta x$$

thus equation (7.217) may be written as:

$$0 = f(a_1, b_1, x_1) - f(a_2, b_2, x_1 + \delta x) - T \tag{7.218}$$

where a_1, b_1, a_2, b_2, T and δx are constants, x_1 is the unknown variable.

It is important to note that due to the nature of bitwise calculations, the i'th bit of the solution corresponds to the i'th bit of x_1. To be more specific, if a bit in position i is changed in x_1, this can at most cause bit changes ranging from bit position i up to the MSB (allowing for carry) when equation (7.218) is evaluated.

In [55] it is stated that all solutions of x_1 has the structure of a binary tree and can be computed using a bitwise recursive process. This process is presented as Algorithm 7.6.

Algorithm 7.6 *Recursive Search Procedure*

1. *Initialise a_1, b_1, a_2, b_2, T and x_1. Set δx to represent the desired difference between x_1 and x_2.*
2. *Set the current depth of the binary tree, i, to 0.*
3. *Determine the current depth of the binary tree.*
4. *If the current depth of the tree, i, equals l (the length of a binary word), record x_1 as a valid solution. Decrement the depth counter i by one. Return to the instruction (step) following the most recent entry or re-entry into Algorithm 7.6. If control is returned to the instruction following the original entry into Algorithm 7.6, Algorithm 7.6 is terminated.*
5. *Calculate the right hand side of equation (7.218). Determine if the i'th bit of the result equals zero.*
6. *If the i'th bit of the right hand side of equation (7.218) equals zero, increment the tree-depth counter i and re-enter Algorithm 7.6 at step 3.*
7. *Toggle the i'th bit in x_1. Re-compute the right hand side of equation (7.218). Determine if the i'th bit of the result equals zero.*
8. *If the i'th bit of the right hand side of equation (7.218) equals zero, increase the tree-depth counter i by one. Re-enter Algorithm 7.6 at step 3.*
9. *Toggle the i'th bit in x_1 and return the number of valid solutions found for x_1. Decrement the tree-depth counter i by one. Return to the instruction (step) following the most recent entry, or re-entry into Algorithm 7.6. If control is returned to the instruction following the original entry into Algorithm 7.6, Algorithm 7.6 is terminated.*

Algorithm 7.6 may be modified to search only for a limited number of solutions. In order to illustrate the relationship between Algorithm 7.6 and a binary tree, the following example is presented.

Example

Let l equal 4. Thus all the variables in equation (7.218) are 4 bit words. For these 4 bit words, let the LSB be numbered 0 and the MSB 3. Let $f()$ be a bitwise Boolean function defined by:

$$f(a, b, x) = (a \wedge x) \vee (\neg x \wedge b)$$

Let the subscript 2 denote the base 2. The following is an example of the application of Algorithm 7.6 to the four bit problem. Set:

$$\begin{aligned} T &= 0000_2 & a_2 &= a_1 \\ a_1 &= 0110_2 & b_2 &= b_1 \\ b_1 &= 1100_2 & \delta x &= 0101_2. \end{aligned}$$

Choose a random value for x_1, say 1001_2. A numerical example of the use of Algorithm 7.6 to find solutions to the given expressions is presented in Table 7.2.

Eight distinct headings are included in Table 7.2. The *Operation nr.* indicates the number of the operation. The *Step nr.* indicates which step in Algorithm 7.6 is associated with the given operation number. The column denoted i contains the tree-depth counter. The column marked $i = 4?$ indicates a test condition. The fifth column is marked x_1 and contains the present value of x_1. The column marked *Equation (7.218)* contains the value to which equation (7.218) evaluates. The *Resume From* column contains an operation number. Associated with the operation number is a step number. If the *Resume From* column contains an operation number, the next step which should be executed is the step following the step number associated with operation number in the *Resume From* column. The *Node* column contains the current node in the binary tree depicted in Figure 7.3. The variables which are updated in each operation are marked in gray.

Operation nr.	Step nr.	i	$i = 4$?	x_1	Equation (7.218)	Resume From Operation nr.	Node
1	2	0	No	1001_2	—	—	r
2	3	0	No	1001_2	—	—	r
3	5	0	No	1001_2	1110_2	—	r
4	6	1	No	1001_2	1110_2	—	B
5	3	1	No	1001_2	1110_2	—	B
6	5	1	No	1001_2	1110_2	—	B
7	7	1	No	1011_2	1010_2	—	B
8	9	0	No	1001_2	—	4	r
9	7	0	No	1000_2	0000_2	—	r
10	8	1	No	1000_2	0000_2	—	A
11	3	1	No	1000_2	0000_2	—	A
12	5	1	No	1000_2	0000_2	—	A
13	6	2	No	1000_2	0000_2	—	C
14	3	2	No	1000_2	0000_2	—	C
15	5	2	No	1000_2	0000_2	—	C
16	6	3	No	1000_2	0000_2	—	G
17	3	3	No	1000_2	0000_2	—	G
18	5	3	No	1000_2	0000_2	—	G
19	6	4	No	1000_2	0000_2	—	O
20	3	4	Yes	1000_2	0000_2	—	O
21	4	3	No	1000_2	0000_2	19	G
22	7	3	No	0000_2	0000_2	—	G
23	8	4	No	0000_2	0000_2	—	P
24	3	4	Yes	0000_2	0000_2	—	P
25	4	3	No	0000_2	0000_2	23	G
26	9	2	No	1000_2	—	16	C
27	7	2	No	1100_2	1000_2	—	C
28	8	3	No	1100_2	1000_2	—	H
29	3	3	No	1100_2	1000_2	—	H
30	5	3	No	1100_2	1000_2	—	H
31	7	3	No	0100_2	1000_2	—	H

Operation nr.	Step nr.	i	$i = 4$?	x_1	Equation (7.218)	Resume From Operation nr.	Node
32	9	2	No	1100_2	—	28	C
33	9	1	No	1000_2	—	13	A
34	7	1	No	1010_2	0000_2	—	A
35	8	2	No	1010_2	0000_2	—	D
36	3	2	No	1010_2	0000_2	—	D
37	5	2	No	1010_2	0000_2	—	D
38	6	3	No	1010_2	0000_2	—	I
39	3	3	No	1010_2	0000_2	—	I
40	5	3	No	1010_2	0000_2	—	I
41	6	4	No	1010_2	0000_2	—	T
42	3	4	Yes	1010_2	0000_2	—	T
43	4	3	No	1010_2	0000_2	41	I
44	7	3	No	0010_2	0000_2	—	I
45	8	4	No	0010_2	0000_2	—	S
46	3	4	Yes	0010_2	0000_2	—	S
47	4	3	No	0010_2	0000_2	45	I
48	9	2	No	1010_2	—	38	D
49	7	2	No	1110_2	1000_2	—	D
50	8	3	No	1110_2	1000_2	—	J
51	3	3	No	1110_2	1000_2	—	J
52	5	3	No	1110_2	1000_2	—	J
53	7	3	No	0110_2	1000_2	—	J
54	9	2	No	1110_2	—	50	D
55	9	1	No	1010_2	—	35	A
56	9	0	No	1000_2	—	10	r
57	9	0	No	1001_2	—	Exit	r

Table 7.2: Numerical Example: Recursive Search Procedure

An implementation of Algorithm 7.6 with reference to the example given above, is listed in Appendix E.

All possible values for x_1 may be represented as a binary tree (see Figure 7.3).

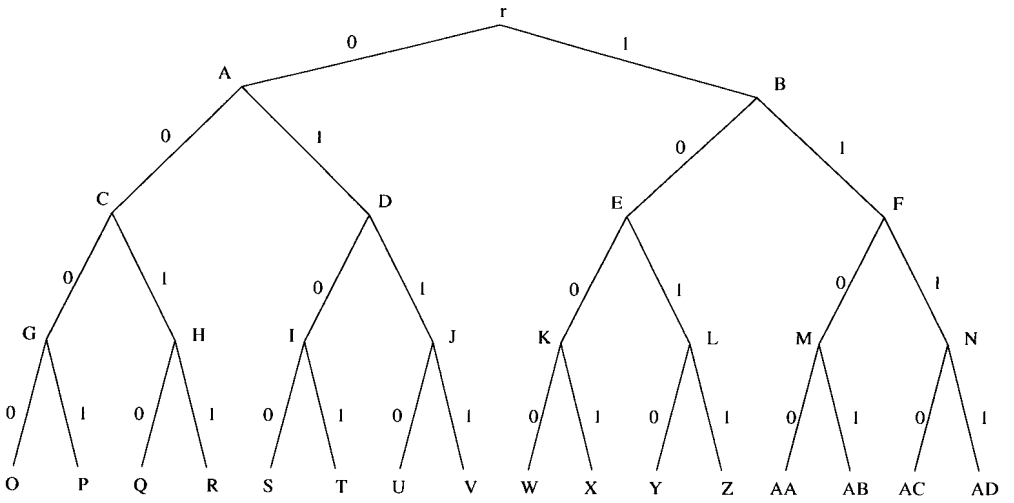

Figure 7.3: Binary Tree Representation of Four Bit Variable

Since all possible values of x_1 are included in this tree, all possible solutions to equation (7.218) are also included in this tree. The relationship between the binary tree in Figure 7.3 and Algorithm 7.6 may be observed from the above example by tracing the node positions at each step of the example. It is noted that if the children of a node (for instance node B) does not yield solutions, the entire branch may be pruned from the search space. Thus Algorithm 7.6 in effect searches through a binary tree. The pruned binary tree which contains only the solutions to the above example is shown in Figure 7.4.

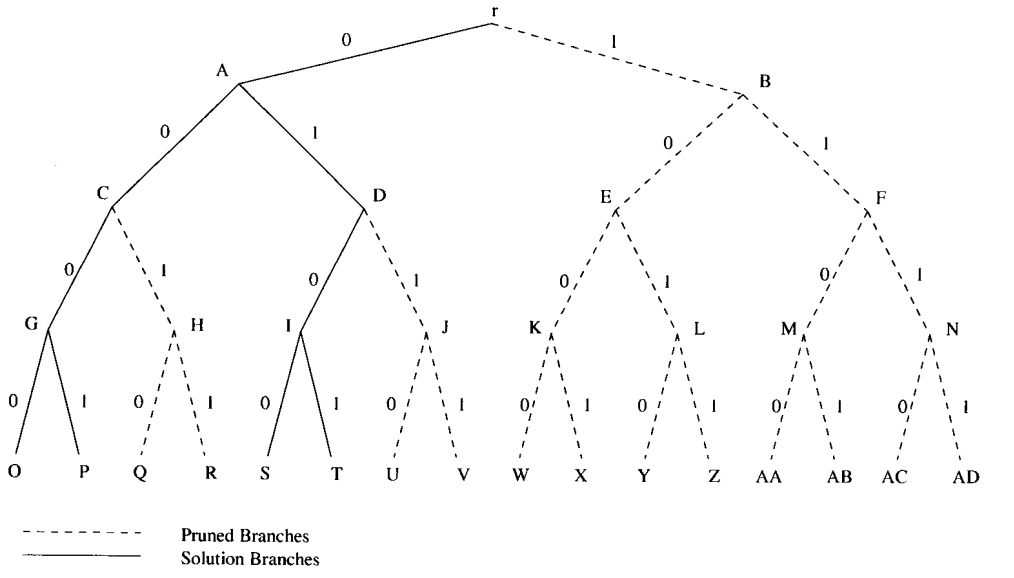

Figure 7.4: Pruned Binary Tree with Solutions

It should be remembered that Algorithm 7.6 only yields results if the Boolean functions operate bitwise on a multiple bit word. Any rotations or other sources of diffusion in the Boolean function, renders the above approach ineffective. The power of this search algorithm lies in the ability to reject entire branches at a time (e.g. in operation nr. 3, all the children nodes which are attached to node A are immediately rejected as possible solutions). The ability to reject entire subsets of possible solutions has the result that the existence of solutions, as well as the solutions themselves, can be determined quickly and efficiently.

7.5.7 Conclusion

This chapter contains a description of the cryptanalysis of MD5 as developed by Dobbertin. The attack on MD5 is similar to the attack on MD4 in a number of aspects. Both attacks require that sets of difference equations should be solved. Furthermore both attacks rely, to some extent, upon the fact that certain differential patterns are more likely to propagate than others. Furthermore both attacks depend on the fact that the Boolean expressions used in the compress functions may be manipulated. In both attacks, the fact that the rotation operations may be separated from the Boolean equations by the use of counter rotations allows the attacker to establish and solve equations of a certain form.

The attack on MD5 is however considerably more complex than the attack on MD4. The additional complexity of this attack is reflected in the amount of time required to find a collision for the compress function of MD5 when it is compared to the time required to find a collision for the compress function of MD4. The additional complexity appears to be due mainly to the use of an additional round. A further contribution to the complexity is the use of the XOR function in the third round. In the attack on MD4, Dobbertin avoids dealing with the XOR function (except briefly during the differential attack on MD4 [14]). In the attack on MD5 this is not the case. The contribution of the constants to the security of MD5 is considered to be low. The constants are only brought into play during the third phase of the attack, when a connection between the results obtained for phase one and two are established. Certain choices and conditions are imposed which appears to speed up the process of finding an inner collision for the second phase of the attack. It is unclear why these choices are made.

Of particular interest is the algorithm used by Dobbertin to determine if solutions to Boolean equations of a specific form exist. This algorithm is presented in Section 7.5.6. It appears

that this algorithm is particularly suited to evaluate Boolean equations which has no diffusion properties over a certain number of bits. It is therefore particularly useful when dealing with bitwise operations as used in MD4 and MD5. If other cryptographic primitives, which utilises bitwise operations are analysed, this algorithm may be a useful tool. It is not known if this algorithm is described in the literature.

7.6 ACKNOWLEDGMENTS

I would like to make use of this opportunity to thank Antoon Bosselaers and the personnel at the COSIC group at the Katholieke Universiteit Leuven for their assistance and support during this investigation into the cryptanalysis of MD4 and MD5. The algorithms and techniques presented in this chapter were derived from the programs written by Dobbertin and the notes of Bosselaers.

CHAPTER 8: GENERALISED ANALYSIS OF THE MD4 FAMILY OF DEDICATED HASH FUNCTIONS

8.1 INTRODUCTION

The vast majority of hash functions (MDCs and MACs) are based on the iterated model known as the Dåmgard-Merkle scheme [22], [23]. This generalised construction is discussed in Chapter 5 Section 5.3. It was proved by Dåmgard that the security of the overall construction relies on the security of the compress function. Dedicated hash functions, including the MDx family of hash functions, are primarily based on the iterative model presented by Dåmgard and Merkle.

In 1996 Dobbertin presented an attack against RIPEMD-128 and MD4 [54], [14]. These attacks showed that it is possible to construct collisions for MD4 and RIPEMD-128. In addition to the above attacks Dobbertin presented an attack on the compress function of MD5. This attack demonstrated that it is computationally feasible to establish collisions for the compress function of MD5 [12], [55].

The similarity in the structure of the hash functions (MD4, MD5 and RIPEMD-128) suggests a common factor in the attacks. If a common factor could be found in all of the attacks, it may be indicative of a weakness in the design of an entire hash function family. Once this weakness is identified it may be possible to derive design criteria for dedicated hash functions. Hash functions designed according to these design criteria would be immune to these attacks.

For these reasons the attacks against MD4 and MD5 were reconstructed and studied (Chapter 6 and 7). Many interesting techniques and properties were discovered in the course of this analysis. Specifically a technique was discovered to speed-up the construction of collisions against MD4. This work resulted in a publication [56].

In this chapter the attacks on MD4 and MD5 are generalised to provide a framework for the analysis of any iterated cryptographic hash function.

8.2 GENERALISED ATTACKS

The attacks described by Dobbertin applies directly to the compress function of the hash functions. It is the aim of these attacks to find two messages M and $\tilde{M}$ with length equal to a single block such that:

$$f(IV, M) = f(IV, \tilde{M})$$

with $f()$ the compress function and IV the initial value used. The compress functions of the dedicated hash function under consideration are constructed by applying a number of steps iteratively. Each step may be expressed as an equation containing Boolean mappings, rotation operators, additive constants and addition $\bmod 2^{32}$ operators.

Let $f_i^j()$ represents the application of the compress function from step i to step j. Before proceeding it is appropriate to state the following definitions.

Definition 1 (Inner-Collision) *An inner-collision is defined if, between steps i and j of the compress function, $f_i^j(C, M) = f_i^j(C, \tilde{M})$. C_i is the internal chaining variable and is given by $C_i = f_0^i(IV, M) = f_0^i(IV, \tilde{M})$.*

Definition 2 (Almost Inner-Collision) *An almost inner-collision is defined if, between steps i and j, of the compress function $f_i^j(C, M) = f_i^j(C, \tilde{M}) + \Delta_i$ with Δ_i a specified difference. As before C_i is the internal chaining variable.*

The attacks by Dobbertin exploit the fact that

$$f(IV, M) = f(IV, \tilde{M})$$

holds if

$$\begin{array}{ccc} f_0^i(IV, M) & = & f_0^i(IV, \tilde{M}) \\ f_{i+1}^j(C_i, M) & = & f_{i+1}^j(C_i, \tilde{M}) \\ f_{j+1}^k(C_j, M) & = & f_{j+1}^k(C_j, \tilde{M}) \\ \vdots & \vdots & \vdots \\ f_{k+1}^l(C_k, M) & = & f_{k+1}^l(C_k, \tilde{M}) \end{array}$$

for a compression function defined by l steps. Thus a number of consecutive inner-collisions may be used to construct a collision. It is noted that an inner-collision may be constructed from an almost inner-collision if differences and chaining variables are found such that:

$$\begin{aligned} f_i^j(C_i, M) &= f_i^j(C_i, \tilde{M}) + \Delta_j \\ f_j^k(C_j + \Delta_j, M) &= f_j^k(C_j, \tilde{M}) \end{aligned}$$

The attacks defined by Dobbertin are focused on finding inner-collisions and almost inner-collisions for sections of the compress function which would result in the construction of collisions for the entire compress function. These attacks share two elements. The first element deals with the derivation of sets of difference equations. The second element requires a solution to these sets of difference equations.

8.2.1 Difference Equations

The chaining variables in MD4 and MD5 are 32 bit words. For these hash functions a difference equation is defined as the difference $\bmod\ 2^{32}$ between two expressions. This may be written as:

$$\Delta E = \left[E_1(M_i) - E_2(\tilde{M}_i)\right] \bmod\ 2^{32}. \tag{8.1}$$

The first expression, $E_1(M_i)$, describes a given step. The variables in this expression are associated with the first message. The second expression, $E_2(\tilde{M}_i)$, describes the same step as the first. The variables used in the second expression are associated with the second message. Expression 8.1 is referred to as a difference equation. A set of difference equations is obtained if difference equations can be derived for a number of consecutive steps in the hash function.

Obtaining a set of difference equations for all the steps of a hash function is possible but not recommended since the dedicated hash functions described in the literature has upwards of 48 steps (MD4), and writing difference equations for all the steps results in a large number of interrelated non-linear equations which are difficult to solve.

For this reason the difference between two messages is restricted to a single message word. In MD4, MD5 message words are re-used in consecutive rounds. The order in which the

message words are processed is changed for each round. In the analysis of MD4 and MD5 the message words in which the differences occur are selected according to the number of steps separating the occurrence of the given word in consecutive rounds.

There exist no exact rule for selecting the message word which will be altered. Instead a general guideline is that the number of steps between the occurrence of a given word should not be too small (less than four steps for MD4 and MD5) since this makes it difficult to derive a set of solvable difference equations. At the same time the number of steps between the occurrence of two steps should not be too large since this increases the number of variables which has to be solved.

In hash functions with an even number of rounds, a trade-off between the number of steps separating the message word in each pair of rounds has to be found. The difference equations should be setup to yield inner-collisions for each pair of rounds.

If the number of rounds are uneven, difference equations should be established for any combination of even rounds. One of the sets of difference equations should be setup to result in an almost inner-collision. This almost inner-collision should be specified in such a manner that a difference pattern propagates and interacts with the message word in the unmatched round. This interaction should result in a collision or inner-collision. As an example consider MD4 where it was shown that

$$f(IV, M) = f(IV, \tilde{M})$$

holds if:

$$\begin{aligned} f_{12}^{19}(C_{12}, M) &= f_{12}^{19}(C_{12}, \tilde{M}) + \Delta_{19} \\ f_{20}^{35}(C_{19} + \Delta_{19}, M) &= f_{20}^{35}(C_{19}, \tilde{M}) \end{aligned}$$

Thus by limiting the difference between two messages to a single word, the number of equations in the set of difference equations are reduced to a manageable size. However it should be noted that there does not necessarily exist solutions to such a set of difference equations.

8.2.2 Solution of Difference Equations

Once a set of difference equations is obtained, the problem remains to solve these equations. Dobbertin proposed one technique which resembles the operation of a genetic algorithm [14]. An investigation into the use of genetic algorithms to solve expressions similar to those encountered in the MD4 family are presented in [57]. The techniques used by Dobbertin to solve these sets of difference equations are summarised in [17] and [58]. The techniques required to solve these sets of equations are fairly specific to each set of equations. However, some common elements may be obtained from these techniques.

In both the analysis of MD4 and MD5 the difference between the messages are restricted to a single word. The Hamming distance between the two message words are kept small, since it is generally easier to manipulate the effect of a small number of bits on the chaining variables. The position at which the difference in the message word is introduced depends on the diffusion mechanisms of the hash function. One of the diffusion mechanisms in MD4 and MD5 is rotation. The bit positions in which differences are introduced are chosen specifically to counter the effect of rotation.

Another technique employed in both the attacks on MD4 and MD5 is the choice of specific values to simplify the sets of difference equations. These choices effectively allows the manipulation of the Boolean mappings. In the attack on MD5 this approach is used to establish certain differential patterns which simplifies the sets of difference equations.

The utilisation of a technique called iterative approximation is encountered in the attacks on MD4 and MD5. This technique evaluates the result obtained from a random initial setting. These initial settings are referred to as the basic values for the set of equations. The basic values are then changed by small increments and the corresponding result is observed. If the result is a closer approximation to the desired result, the changed basic values becomes the new basic values. This process is repeated until the desired result is obtained.

Another approach used to find solutions to sets of difference equations is used in the analysis of MD5. This technique requires that a large number of possible solutions are collected and then tried one after the other. This process is combined with the iterative approximation technique. A large number of basic values and possible solutions are generated. These possible solutions are then tried one after the other. The best solution is retained and serves as the basis for the next iteration. The techniques used to produce a large number of possible

solutions are described in [57].

8.3 APPLICATION OF GENERALISED ATTACKS

The generalisation of the attack as described in this chapter was applied to a limited number of rounds of the HAVAL and SHA hash functions. Both are part of the MDx-family of hash functions. Using the generalised technique it can be shown that collisions can be establihed for limited rounds of HAVAL and single rounds of SHA. The details of these attacks are included in Chapters 9 and 10. To the best of my knowledge this is the first publicly published cryptanalytical results for SHA and HAVAL.

8.4 CONCLUSION

This chapter generalises the approach used by Dobbertin in the cryptanalysis of MD4 and MD5 as discussed in Chapters 6 and 7. The attacks on MD4 and MD5 focus on reducing the problem of finding a collision to solving a limited number of interrelated equations. This may be viewed as a divide and conquer approach. The complexity of finding collisions are reduced through two mechanisms. The first mechanism reduces the number of equations to be solved and the second simplifies these equations to facilitate the establishment of a solution. Detailed descriptions of these mechanisms may be found in [14], [17] and [57]. The application of this generalised technique to two other hash functions in the MD4 family, SHA-1 and HAVAL, is presented in the next two chapters.

CHAPTER 9: ANALYSIS OF THE SHA AND SHA-1 HASH ALGORITHMS

9.1 INTRODUCTION

In this chapter the SHA and SHA-1 hash functions are analysed. First the SHA and SHA-1 hash functions are described along with the relevant notation used in this chapter. This is followed by describing the algebraic structure of the message expansion algorithm used by SHA. We then proceed to exploit this algebraic structure of the message expansion algorithm by applying the generalised analysis framework presented in Chapter 8. We show that it is possible to construct collisions for each of the individual rounds of the SHA hash function. The source code that implements the attack is attached in Appendix F. The same techniques are then applied to SHA-1.

9.2 INTRODUCTION TO SHA

SHA is an acronym for Secure Hash Algorithm. SHA and SHA-1 are dedicated hash functions based on the iterative Damgård-Merkle construction [22] [23]. Both of the round functions utilised by these algorithms take a 512 bit input (or a multiple of 512) and produce a 160 bit hash value. SHA was first published as Federal Information Processing Standard 180 (FIPS 180). The secure hash algorithm is based on principles similar to those used in the design of MD4 [10]. SHA-1 is a technical revision of SHA and was published as FIPS 180-1 [13]. It is believed that this revision makes SHA-1 more secure than SHA [13] [50] [59]. SHA and SHA-1 differ from MD4 with regard to the number of rounds used, the size of the hash result and the definition of a single step. A further difference between SHA/SHA-1 and MD4 is the use of a message expansion algorithm instead of re-using the message blocks in different orders in each round. SHA-1 was designed to be both pre-image resistant and collision resistant [13].

9.3 NOTATION

The following notation is used in this chapter.

$\wedge$ = Bitwise AND $\quad$ $\neg$ = Bitwise NOT

$\vee$ = Bitwise OR $\quad$ $\oplus$ = Bitwise XOR.

Let $X^{\lll Z}$ denote the left rotation of X by Z bit positions.

9.4 SHA

Five steps may be identified in the definition of SHA.

1. Message padding.
2. Initialise chaining variables.
3. Perform message expansion.
4. Apply compress function.
5. Update variables.

Step 1 ensures that the message is padded to a multiple of 512 bits. Steps 3 to 5 are repeated for each 512 bit block until the entire padded message has been processed.

9.4.1 Message Padding

The purpose of padding is to produce a $n \cdot 512$ bit message. The message is padded by appending a 1 to the message, followed by m 0's followed by a 64 bit integer. The 64 bit integer is the binary representation of the length of the original message l. If l is less than 2^{32} the first 32 bits of the final 64 bits are zero.

9.4.2 Initialise Chaining Variables

The five chaining variables are initialised to the following hexadecimal values.

$$\begin{aligned} H_0 &= \texttt{0x67452301} \\ H_1 &= \texttt{0xEFCDAB89} \\ H_2 &= \texttt{0x98BADCFE} \\ H_3 &= \texttt{0x10325476} \\ H_4 &= \texttt{0xC3D2E1F0}. \end{aligned}$$

9.4.3 Message Expansion

The message is processed in 512 bit blocks. The padded message M is the concatenation of n blocks of 512 bits. Let $|$ denote concatenation, then:

$$M = M_1|M_2|M_3|\ldots|M_n.$$

Each message block M_i is divided into 16 words W_0, W_1, W_2, ... ,W_{15}. Each word has a length of 32 bits. The execution of the compress function involves 80 steps. The first 16 steps are performed on W_0, W_1, W_2, ... ,W_{15}. The remaining 64 steps are performed on message words W_{16}, W_{17}, W_{18}, ... ,W_{79} which are obtained from the following message expansion algorithm:

$$W_t = W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16} \tag{9.1}$$

with $t = 16, 17, 18, \ldots 79$.

9.4.4 Compress Function

The compress function of SHA is defined by the following procedure:

1. Set $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$ and $E = H_4$.
2. For t = 0 to 79 do

(a) TEMP = $A^{\lll 5} + f_t(B, C, D) + E + W_t + K_t$.

(b) $E = D, D = C, C = B^{\lll 30}, B = A, A = TEMP$

with the constant K_t defined in hexadecimal as:

$$\begin{aligned} K_t &= \texttt{0x5A827999} \quad (0 \leq t \leq 19) \\ K_t &= \texttt{0x6ED9EBA1} \quad (20 \leq t \leq 39) \\ K_t &= \texttt{0x8F1BBCDC} \quad (40 \leq t \leq 59) \\ K_t &= \texttt{0xCA62C1D6} \quad (60 \leq t \leq 79) \end{aligned}$$

and the round function f_t defined as:

$$\begin{aligned} f_t &= (B \wedge C) \vee (\neg B \wedge D) && (0 \leq t \leq 19) \\ f_t &= B \oplus C \oplus D && (20 \leq t \leq 39) \\ f_t &= (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) && (40 \leq t \leq 59) \\ f_t &= B \oplus C \oplus D && (60 \leq t \leq 79) \end{aligned} .$$

Figure 9.1 shows a graphical representation of a single step of the SHA and SHA-1 round function.

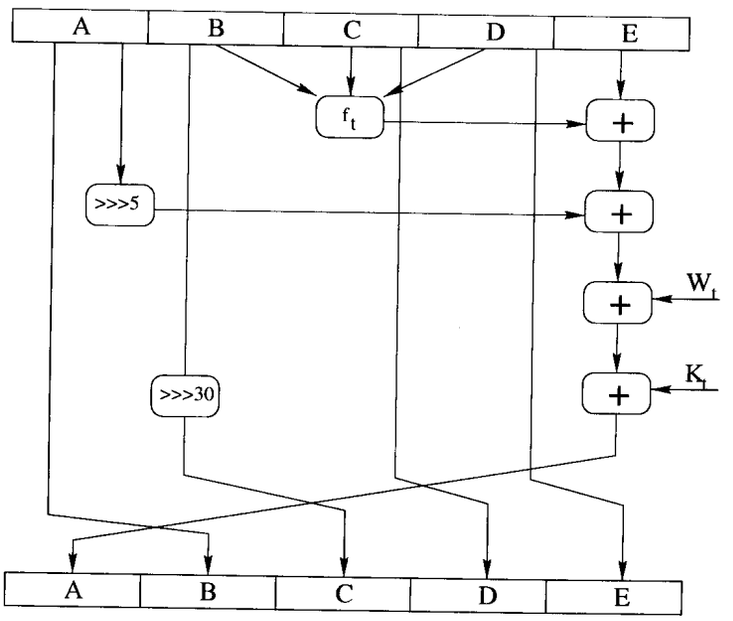

Figure 9.1: Single Step in Round Function: SHA and SHA-1

9.4.5 Update Chaining Variables

Upon completion of the compress function the chaining variables are updated as shown below:

$$\begin{aligned} H_0 &= H_0 + A \\ H_1 &= H_1 + B \\ H_2 &= H_2 + C \\ H_3 &= H_3 + D \\ H_4 &= H_4 + E \end{aligned}$$

Once the n'th 512 bit block has been processed the resulting values of the chaining variables serve as the hash result of the message.

9.5 SHA-1

SHA-1 is identical to SHA in all respects except for the message expansion algorithm.

9.5.1 Message Expansion

The message expansion algorithm used in SHA-1 is defined as the message expansion algorithm defined by expression 9.1 with the addition of a left rotation by one bit position. The message expansion algorithm as used in SHA-1 is defined by expression 9.2.

$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})^{\lll 1}. \tag{9.2}$$

This modification "corrects a technical flaw that made the standard less secure than had been thought" [50].

9.6 ANALYSIS OF SHA AND SHA-1

In this chapter the basic elements encountered in the attacks on MD4 and MD5 as formulated by Dobbertin are applied to SHA and SHA-1. A direct application of the attacks formulated

by Dobbertin requires the establishment of a set of difference equations, followed by a solution of these equations (see Chapter 8). The use of message expansion algorithms in SHA and SHA-1 instead of the permutation of message words in consecutive rounds prevents an attacker from deriving sets of difference equations according to the principles laid down in Chapter 8. Thus when dealing with message expansion algorithms the approach described in Chapter 8 should be modified. This chapter describes these modifications. Specific attention is given to the properties of the message expansion algorithms. It is shown that the modified approach may be used to construct a collision for the first round (first twenty steps) of SHA. Based on this attack, an attack on the first two rounds of SHA is proposed. Certain elements (but not all) of the proposed attack are confirmed. The extent to which these attacks are applicable to SHA-1 is considered.

9.7 SHA

The secure hash algorithm (SHA) is described in Section 9.4. In this section certain properties of the message expansion algorithm is considered. It is shown that these properties may be exploited to construct sets of difference equations which are readily solved. A solution to these difference equations results in the construction of collisions for the first round of SHA. An attack which exploits the properties of the message expansion algorithm is then proposed for the first two rounds of SHA.

9.7.1 Message Expansion Algorithm

The message expansion algorithm used in SHA is presented in Chapter 9. The algorithm expands a 16 word input to 64 words. The expanded message is concatenated to the original message to form an 80 word message block. Remember that the message expansion algorithm is given by:

$$W_t = W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}$$

where all message words, W_t, are 32 bit variables. It is observed that this expansion algorithm is in effect 32 identical linear feedback shift registers which operate in parallel. This observation may be represented graphically as shown in Figure 9.2.

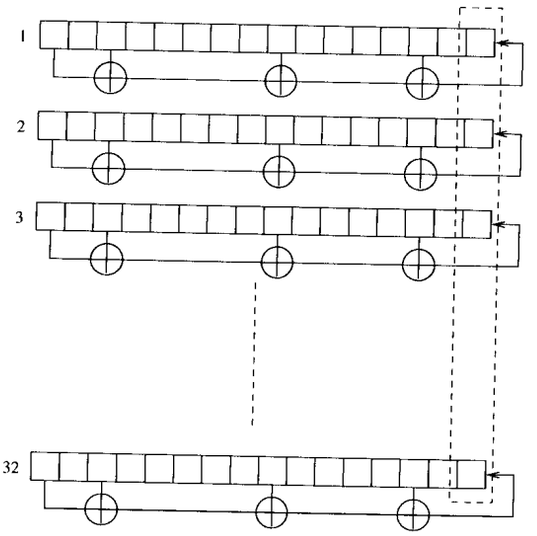

Figure 9.2: Message Expansion Algorithm: SHA

The connection polynomial, $f(x)$, is given by:

$$f(x) = x^{16} + x^{13} + x^8 + x^2 + 1. \tag{9.3}$$

It may be shown that $f(x)$ is primitive. It is known that there exists

$$\frac{\phi(p^m - 1)}{m}$$

monic primitive polynomials of degree m in $Z_p[x]$ ($\phi(n)$ is the Euler function) [59]. For the case in hand this corresponds to 2048 possible monic primitive polynomials in $Z_2[x]$. The reason for the specific choice of $f(x)$ has not been disclosed, but at least two arguments in favour of using $f(x)$ have been found. The first argument deals with performance. The polynomial $f(x)$ has a low weight (5), and consequently requires fewer instructions to expand the message. The second argument deals with the positioning of the non-zero coefficients in the polynomial. The non-zero coefficients should be spread as evenly as possible over the polynomial. This requirement ensures that each word is used in the expansion within a small number of steps. This makes it more difficult for an attacker to hide or minimise the effect of a specific word. There are 52 primitive polynomials of degree 16 with a weight of 5. It is found that the maximum number of consecutive coefficients which are equal to zero varies between 5 and 10. There are only 16 primitive polynomials with a maximum number

of consecutive zero coefficients equal to zero. They are:

$$f_1(x) = x^{16} + x^{10} + x^5 + x^3 + 1 \quad (9.4)$$
$$f_2(x) = x^{16} + x^{10} + x^7 + x + 1 \quad (9.5)$$
$$f_3(x) = x^{16} + x^{10} + x^7 + x^3 + 1 \quad (9.6)$$
$$f_4(x) = x^{16} + x^{10} + x^7 + x^4 + 1 \quad (9.7)$$
$$f_5(x) = x^{16} + x^{10} + x^7 + x^6 + 1 \quad (9.8)$$
$$f_6(x) = x^{16} + x^{10} + x^9 + x^6 + 1 \quad (9.9)$$
$$f_7(x) = x^{16} + x^{12} + x^6 + x + 1 \quad (9.10)$$
$$f_8(x) = x^{16} + x^{12} + x^7 + x + 1 \quad (9.11)$$
$$f_9(x) = x^{16} + x^{12} + x^9 + x^6 + 1 \quad (9.12)$$
$$f_{10}(x) = x^{16} + x^{13} + x^8 + x^2 + 1 \quad (9.13)$$
$$f_{11}(x) = x^{16} + x^{13} + x^9 + x^6 + 1 \quad (9.14)$$
$$f_{12}(x) = x^{16} + x^{13} + x^{11} + x^6 + 1 \quad (9.15)$$
$$f_{13}(x) = x^{16} + x^{14} + x^8 + x^3 + 1 \quad (9.16)$$
$$f_{14}(x) = x^{16} + x^{15} + x^9 + x^4 + 1 \quad (9.17)$$
$$f_{15}(x) = x^{16} + x^{15} + x^9 + x^6 + 1 \quad (9.18)$$
$$f_{16}(x) = x^{16} + x^{15} + x^{10} + x^4 + 1 \quad (9.19)$$
$$(9.20)$$

Eight of these polynomials has two consecutive non-zero coefficients ($f_2(x)$, $f_5(x)$, $f_6(x)$, $f_7(x)$, $f_8(x)$, $f_{14}(x)$, $f_{15}(x)$ and $f_{16}(x)$). The primitive polynomial used in the message expansion algorithm of SHA (and SHA-1) is found among the remaining eight polynomials ($f_{10}(x)$).

It is observed that the message block of 512 bits is divided into 16 32-bit words and that each round requires 20 steps. This differs from hash functions such as MD4 and MD5 where the number of message words in a message block are equal to the number of steps in each round. Before presenting a motivation for this design choice consider Proposition 1.

Proposition 1 *If the message expansion algorithm defined for SHA is used, it is possible to construct two distinct messages which, after expansion, are identical in 15 consecutive expanded words.*

Proof. Consider a single linear feedback shift register with a connection polynomial $f(x)$ of degree n which is monic and primitive in $Z_2[x]$. It is known that all non-zero sequences generated by such a linear feedback shift register have $n-1$ consecutive zeros. In addition it is known that any possible output sequence of such a linear feedback shift register is a cyclic shift of every other possible output sequence of the same linear feedback shift register [60]. Thus there exists a non-zero sequence, $\boldsymbol{a}$, generated by the linear feedback shift register with feedback polynomial $f(x)$ such that $\boldsymbol{a}$ has $n-1$ consecutive zeros starting at a specified position in the sequence $\boldsymbol{a}$.

Let $f(x) = c_0 + c_1x + c_2x^2 \ldots + c_nx^n$ with coefficients from $GF(2)$. Then the companion matrix of the polynomial $f(x)$ is the $n \times n$ matrix, $\boldsymbol{C}$, which has 1's in the diagonal above the main diagonal and n'th row entries c_0, c_1, c_2 $\ldots$ c_{n-1} [61]. Then the t'th state of the linear feedback shift register is given by:

$$a(t) = \boldsymbol{C}^t a(0)$$

with $a(t)$ a $1 \times n$ vector representing the state of the linear feedback shift register. The elements of the vector $a(0)$ represents the initial state of the shift register. Let the sequence $\boldsymbol{a}$ be the concatenation (denoted by $|$) of the n'th element of the vector $a(t)$ for all values of $t > 0$. Thus:

$$\boldsymbol{a} = a_n(1)|a_n(2)|a_n(3)|\ldots|a_n(t).$$

Consider two sequences $\boldsymbol{a^1}$ and $\boldsymbol{a^2}$ defined as:

$$\begin{aligned} \boldsymbol{a^1} &= a_n^1(1)|a_n^1(2)|a_n^1(3)|\ldots|a_n^1(t) \\ \boldsymbol{a^2} &= a_n^2(1)|a_n^2(2)|a_n^2(3)|\ldots|a_n^2(t). \end{aligned}$$

The states, $a^1(t)$ and $a^2(t)$, of the linear feedback shift registers which generates the sequences $\boldsymbol{a^1}$ and $\boldsymbol{a^2}$ are given by:

$$\begin{aligned} a^1(t) &= \boldsymbol{C}^t a^1(0) \\ a^2(t) &= \boldsymbol{C}^t a^2(0). \end{aligned}$$

The summation over $GF(2)$ of the states of the linear feedback shift registers (and implicitly

the summation of the sequences $\boldsymbol{a^1}$ and $\boldsymbol{a^2}$) is given by:

$$\begin{aligned} a^1(t) \oplus a^2(t) &= \boldsymbol{C}^t a^1(0) \oplus \boldsymbol{C}^t a^2(0) \\ &= \boldsymbol{C}^t (a^1(0) \oplus a^2(0)). \end{aligned}$$

If

$$a^1(0) \oplus a^2(0) = a(0) \tag{9.21}$$

then

$$\begin{aligned} a^1(t) \oplus a^2(t) &= \boldsymbol{C}^t (a(0)) \\ &= a(t) \end{aligned}$$

which implies that:

$$\boldsymbol{a} = \boldsymbol{a^1} \oplus \boldsymbol{a^2}.$$

The message expansion algorithm used by SHA consists of 32 identical linear feedback shift registers applied in parallel. Let the initial conditions $a^1(0)$ and $a^2(0)$ for all 32 linear feedback shift registers represent two distinct messages. All elements in $\boldsymbol{a^1}$ and $\boldsymbol{a^2}$ are taken as the result of the message expansion algorithm. Find values for $a^1(0)$ and $a^2(0)$ such that expression (9.21) is satisfied for all 32 registers. Then, upon expansion, the message words will be identical in $n - 1$ consecutive bit positions. The primitive polynomial used by the linear feedback shift register in the message expansion algorithm in SHA has degree 16. It is therefore possible to construct two messages which upon expansion will be identical in 15 consecutive words. Thus the proposition is shown to be true. ■

With Proposition 1 in hand the use of 20 steps in a round rather than 16 becomes obvious. If the number of steps in each round is set to 16 instead of 20 and the number of rounds are retained it would be possible to construct two distinct messages which, in one of the four rounds, would differ in only one position, thus effectively stripping a round from the hash function. By defining 20 steps in each round this attack is prevented. However if the number of steps are limited to 16, and the number of rounds is extended from four to five, the total number of steps required by the hash function remain at 80.

Thus it appears that the message expansion algorithm was chosen with specific aims, and that the number of steps in each round was chosen to complement the message expansion

algorithm. One property, and a potential weakness, of the message expansion algorithm is the fact that if two messages differ only in a single message word, and if that difference is limited to a single bit position, the words obtained from the message expansion algorithm will, at most, differ in the same bit position. Even if the differences occur in more than one message word, as long as these differences occur in the same bit position, the expanded words would at most differ in the same bit positions. This property is considered a possible salient point and the extent to which this may be exploited is considered in the next section.

9.7.2 Difference Equations

In this section specific attention is given to the exploitation of the message expansion algorithm in order to obtain sets of differential equations.

It is possible to obtain a set of difference equations over all 80 steps of the compress function of SHA. As remarked in Chapter 8 this is impractical due to the large number of interrelated equations which have to be solved. An attempt should therefore be made to reduce the number of difference equations. It is observed that by direct application of Proposition 1 the number of difference equations may be reduced from 80 to 65. This reduction is obtainable by requiring that the last 15 words resulting from the message expansion algorithm should be identical. Although this reduces the effort required to find a collision, the effort required to solve this set of equations is unknown.

Before proceeding with the analysis of SHA and the construction of difference equations it is convenient to state the following definition.

Definition 3 *A difference pattern is the sequence generated by $M_i - \tilde{M}_i$ for all i. M_i and $\tilde{M}_i$ represents the first message and second message words at a specific step i of the dedicated hash function.*

The difference operator may be either the difference $\bmod 2$ or the difference $\bmod 2^{32}$. In order to illustrate the derivation of a set of difference equations consider only the first round. It is readily observed that for six consecutive steps, the following difference pattern (mod

2^{32}) allows the derivation of a set of difference equations which are easily solved:

$$\begin{aligned} M_i - \tilde{M}_i &= 1 \\ M_{i+1} - \tilde{M}_{i+1} &= -1 \\ M_{i+2} - \tilde{M}_{i+2} &= -1 \\ M_{i+3} - \tilde{M}_{i+3} &= 0 \\ M_{i+4} - \tilde{M}_{i+4} &= 0 \\ M_{i+5} - \tilde{M}_{i+5} &= -1 \end{aligned}$$

The difference $\bmod 2^{32}$ is chosen to be either 1,0 or -1 since these differences may be written in terms of the rotation invariant integers 0 and -1. By limiting all differences to the LSB a difference pattern $\bmod 2$ may be obtained. The difference pattern $\bmod 2$ is given by:

$$\begin{aligned} M_i \oplus \tilde{M}_i &= 1 \\ M_{i+1} \oplus \tilde{M}_{i+1} &= 1 \\ M_{i+2} \oplus \tilde{M}_{i+2} &= 1 \\ M_{i+3} \oplus \tilde{M}_{i+3} &= 0 \\ M_{i+4} \oplus \tilde{M}_{i+4} &= 0 \\ M_{i+5} \oplus \tilde{M}_{i+5} &= 1 \end{aligned}$$

The set of difference equations corresponding to this difference pattern is given by:

$$A_{i+1} - \tilde{A}_{i+1} = M_i - \tilde{M}_i \tag{9.22}$$

$$0 = A_{i+1}^{\lll 5} - \tilde{A}_{i+1}^{\lll 5} + M_{i+1} - \tilde{M}_{i+1} \tag{9.23}$$

$$0 = M_{i+2} - \tilde{M}_{i+2} + f(B_{i+2}, C_{i+2}, D_{i+2}) - f(\tilde{B}_{i+2}, C_{i+2}, D_{i+2}) \tag{9.24}$$

$$0 = f(B_{i+3}, C_{i+3}, D_{i+3}) - f(B_{i+3}, \tilde{C}_{i+3}, D_{i+3}) \tag{9.25}$$

$$0 = f(B_{i+4}, C_{i+4}, D_{i+4}) - f(B_{i+4}, C_{i+4}, \tilde{D}_{i+4}) \tag{9.26}$$

$$0 = M_{i+5} - \tilde{M}_{i+5} + E_{i+5} - \tilde{E}_{i+5} \tag{9.27}$$

The Boolean mapping $f()$ is the mapping defined for the first round of SHA. The variables are updated as described in Section 9.4. This set of expressions may be solved by setting the chaining variables C_2, D_2, B_3 and B_4 to appropriate values.

There exists many difference patterns for the first round which yields sets of solvable difference equations. It now remains to determine if the difference pattern leading to this set of equations can be found in the first round. It is now appropriate to state Corollary 2.

Corollary 2 *Each sequence,* $\boldsymbol{a}$*, generated by a linear feedback shift register with feedback polynomial* $f(x)$ *represents a difference pattern which may be generated by two distinct sequences,* $\boldsymbol{a^1}$ *and* $\boldsymbol{a^2}$*.*

Proof. It is shown in the proof of Proposition 1 that:

$$\boldsymbol{a^1} \oplus \boldsymbol{a^2} = \boldsymbol{a}$$

if

$$a^1(0) \oplus a^2(0) = a(0). \tag{9.28}$$

This implies that if the k'th element of the sequence $a_k = 0$, $a_k^1 = a_k^2$. Conversely it implies that if $a_k = 1$, $a_k^1 \neq a_k^2$. Thus the sequence $\boldsymbol{a}$ represents the difference between the two sequences $\boldsymbol{a^1}$ and $\boldsymbol{a^2}$ if condition (9.28) holds. ■

As before, all differences are limited to the LSB of the message words. This causes all differences in the expanded message to be generated by a single linear feedback shift register. An attacker may now search for the desired difference pattern $\bmod 2$. Once the difference pattern $\bmod 2$ is found it is expanded to the difference pattern $\bmod 2^{32}$ through the application of Corollary 2. The attacker is free to choose values for $a^1(0)$ and $a^2(0)$ as long as expression (9.28) holds. A difference pattern $\bmod 2^{32}$ may be established by remarking that for $a_k^1 \oplus a_k^2 = 1$, either $a_k^1 = 1$ and $a_k^2 = 0$ or $a_k^1 = 0$ and $a_k^2 = 1$. This implies that if $a_k^1 \oplus a_k^2 = 1$, either $a_k^1 - a_k^2 = -1$ or $a_k^1 - a_k^2 = 1$. According to Corollary 2 the attacker may choose whether $a_k^1 - a_k^1 = 1$ or $a_k^1 - a_k^1 = -1$ for $k < n$.

Thus, the attacker searches for an initial value which, in combination with the expanded message word, would allow the establishment of the set of difference equations defined by equations (9.22) to (9.27) as the only set of difference equations for the first round. If such a pattern is found, it is expanded to the desired difference pattern $\bmod 2^{32}$ by applying Corollary 2. It is found that there exists only one difference pattern which allows equations (9.22) to (9.27) to be the only set of difference equations in the first round. The difference

pattern mod2 is given by:

$$M_{11} \oplus \tilde{M}_{11} = 1 \tag{9.29}$$
$$M_{12} \oplus \tilde{M}_{12} = 1 \tag{9.30}$$
$$M_{13} \oplus \tilde{M}_{13} = 1 \tag{9.31}$$
$$M_{14} \oplus \tilde{M}_{14} = 0 \tag{9.32}$$
$$M_{15} \oplus \tilde{M}_{15} = 0 \tag{9.33}$$
$$M_{16} \oplus \tilde{M}_{16} = 1. \tag{9.34}$$

It is observed that M_{11} to M_{15} forms part of the initial value and according to Corollary 2 any combination of the LSB's of M_{11} to M_{15} may be chosen in an attempt to obtain the desired difference pattern mod 2^{32} for message words M_{11} to M_{16}. There exists a combination which allows the attacker to find the required difference mod 2^{32}.

The difference pattern defined by equation (9.29) to (9.33) results in the set of difference equations shown below:

$$A_{12} - \tilde{A}_{12} = M_{11} - \tilde{M}_{11} \tag{9.35}$$
$$0 = A_{12}^{\lll 5} - \tilde{A}_{12}^{\lll 5} + M_{12} - \tilde{M}_{12} \tag{9.36}$$
$$0 = M_{13} - \tilde{M}_{13} + f(B_{13}, C_{13}, D_{13}) - f(\tilde{B}_{13}, C_{13}, D_{13}) \tag{9.37}$$
$$0 = f(B_{14}, C_{14}, D_{14}) - f(B_{14}, \tilde{C}_{14}, D_{14}) \tag{9.38}$$
$$0 = f(B_{15}, C_{15}, D_{15}) - f(B_{15}, C_{15}, \tilde{D}_{15}) \tag{9.39}$$
$$0 = M_{16} - \tilde{M}_{16} + E_{16} - \tilde{E}_{16}. \tag{9.40}$$

Updating of chaining variables are performed as specified in Section 9.4. This set of equations implies that:

$$A_{12} - \tilde{A}_{12} = 1.$$

Due to the effect of rotation, the values for A_{12} and $\tilde{A}_{12}$ are chosen to be rotation invariant.

Thus in hexadecimal notation:

$$A_{12} = \texttt{0x00000000} \tag{9.41}$$

$$\tilde{A}_{12} = \texttt{0xFFFFFFFF} \tag{9.42}$$

$$\tag{9.43}$$

These choices for A_{12} and $\tilde{A}_{12}$ ensures that equation (9.36) is satisfied. Equation (9.37) is satisfied if:

$$f(B_{13}, C_{13}, D_{13}) - f(\tilde{B}_{13}, C_{13}, D_{13}) = -1.$$

Due to the choices for A_{12} and $\tilde{A}_{12}$ this condition is satisfied if:

$$D_{13} - C_{13} = -1$$

In order to solve equation (9.38),

$$f(B_{14}, C_{14}, D_{14}) - f(B_{14}, \tilde{C}_{14}, D_{14}) = 0.$$

By setting $B_{14} = 0$, equation (9.38) is satisfied. Likewise equation (9.39) is satisfied if $B_{15} = -1$. Equation (9.40) is automatically satisfied due to the choices initially made for A_{12} and $\tilde{A}_{12}$.

It is noted that this is not the only technique which results in a solution for the set of difference equations given by (9.35) to (9.40). Other more elaborate techniques exist, but are slower and do not guarantee that a solution is obtained.

Given appropriate choices for the chaining variables in question, it is possible to reconstruct the message. For the initial values specified for SHA, the following messages result in a collision for the first round of the compress function of SHA.

$$M_0 = \texttt{0x20760CF1} \qquad M_8 = \texttt{0xF4AD6572}$$
$$M_1 = \texttt{0x0C1F1475} \qquad M_9 = \texttt{0x5F059EA3}$$
$$M_2 = \texttt{0x56139C91} \qquad M_{10} = \texttt{0x050A6650}$$
$$M_3 = \texttt{0xA904D458} \qquad M_{11} = \texttt{0x5279A115}$$
$$M_4 = \texttt{0x07F3FF32} \qquad M_{12} = \texttt{0xBB4E5B88}$$
$$M_5 = \texttt{0x69B971AD} \qquad M_{13} = \texttt{0x724D80BA}$$
$$M_6 = \texttt{0x13E8DD88} \qquad M_{14} = \texttt{0x438ECCB0}$$
$$M_7 = \texttt{0x40CA61AC} \qquad M_{15} = \texttt{0xA1EDDF3D}$$

The alternative message is identical to the above message except for:

$$\tilde{M}_{11} = \texttt{0x5279A114}$$
$$\tilde{M}_{12} = \texttt{0xBB4E5B89}$$
$$\tilde{M}_{13} = \texttt{0x724D80BB}.$$

The common message digest for the first round of SHA is given by:

Common Digest = `0x5C2E4C26 0x494479D5 0x828CE366 0xC45C9D77 0xE437389D`.

An implementation of this attack on the first round of SHA is attached as Appendix F. Attacks similar to that described above may be readily applied to rounds two, three and four if these rounds are considered individually.

9.7.3 Extended Attack

The attack presented above may be extended to other difference patterns. This allows the formulation of the following attack on the first round of SHA.

First a number of relatively short difference patterns which result in inner collisions are obtained. It should then be determined if any of these short difference patterns occur in the first round of SHA. If none of these patterns are found in the first round, a search should

be conducted for concatenations of these difference patterns. If a concatenation of these difference patterns are found, it is known that the resulting set of difference equations may be solved and a high probability exist that a message may be reconstructed which would result in a collision for the first round. Note that the construction of a message which results in a collision is not assured. This is due to the fact that even though the difference pattern results in a set of solvable difference equations, the specific choices made to solve these equations may contradict the bounds incurred by the message expansion algorithm.

9.7.4 Proposed Attack

It may be possible to extend the attack described in the previous section to the second and possibly third and fourth rounds. As before, short difference patterns which results in inner collisions should be obtained for all the rounds in question. The output of the message expansion algorithm should then be searched for the concatenation of a number of these patterns. If a difference pattern is found which is composed from the concatenation of the shorter difference pattern, it may be possible to find a solution to the set of difference equations. If a solution is obtained, it should be verified if a message may be reconstructed which would result in a collision. As before the specific choices made to solve these equations may contradict the bounds incurred by the message expansion algorithm.

An application of this proposed attack showed that it is possible to find difference patterns which are solvable. Unfortunately it was found that the choices made for a number of the chaining variables which allow solutions to be found for the short difference equations leaves a limited number of degrees of freedom. This limits the attackers ability to reconstruct a message which results in a collision for more than one round. If more sophisticated techniques are found to solve the sets of difference equations, fewer explicit choices would have to be made and it may become possible to find solutions to the sets of difference equations which allows the construction of messages which result in collisions for two or more rounds.

9.8 SHA-1

As remarked in Chapter 9, the only difference between SHA and SHA-1 lies in the message expansion algorithm. In this section the message expansion algorithm used in SHA-1 is considered and a number of its characteristics are discussed.

9.8.1 Message Expansion Algorithm

The message expansion algorithm used in SHA-1 is defined by:

$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})^{\lll 1}. \tag{9.44}$$

It is noted that the only difference between the message expansion algorithms used in SHA and SHA-1 is the addition of a rotation by one bit position. A graphic representation of the message expansion algorithm used in SHA-1 is shown in Figure 9.3

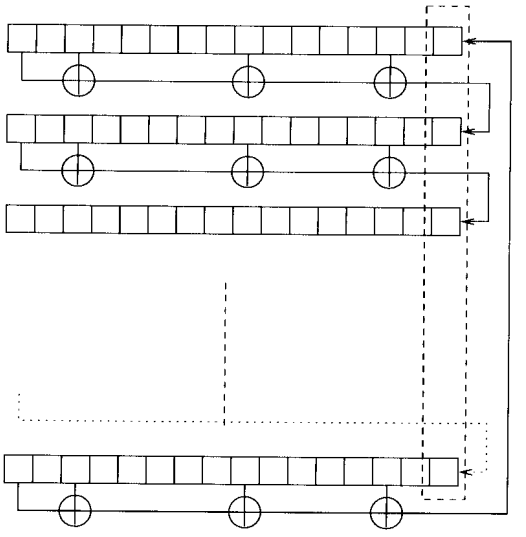

Figure 9.3: Message Expansion Algorithm: SHA-1

The rotation operator introduces diffusion in the message expansion algorithm. A difference introduced in a single bit position is no longer limited to the same bit position in the expanded message (as is the case for SHA), but is spread over a number of bit positions. As the original message is expanded a larger number of bit positions are affected by changing a single bit in the message word. Thus the addition of diffusion to the message expansion algorithm is believed to increase the security offered by SHA-1.

The use of the rotation operator makes it difficult to specify a difference pattern which is easily related to an initial message. Thus the analysis of SHA presented in Section 9.7 is not directly applicable to SHA-1. It is believed that the addition of the rotation operator to the message expansion algorithm makes it considerably more difficult to obtain and exploit difference equations. At present no analysis of the message expansion algorithm used by SHA-1 has been published in the open literature.

9.9 CONCLUSION

The analysis presented in this chapter leads to the conclusion that SHA-1 is more secure than the original SHA. The additional security of SHA-1 is derived solely from the modified message expansion algorithm. It was shown that it is possible to exploit the characteristics of the message algorithm defined for SHA by constructing a collision for the first round of SHA. In addition an attack on more than one round of SHA is proposed. It may be argued that the ease with which the message expansion algorithm used in SHA is manipulated may have served as one of the reasons for the modification to the message expansion algorithm used in SHA-1.

Additional factors which complicates the analysis of SHA and SHA-1 have been found. Specifically the method used for updating the chaining variables and the use of a rotation over 30 bits for chaining variables C_i, D_i and E_i contributes to the difficulty of solving sets of difference equations. The method used for updating the chaining variables ensures that a difference introduced in a message propagates to each of the chaining variables. This is not the case for MD4 and MD5 where a difference may be manipulated to appear only in certain selected chaining variables. An additional difficulty is the use of rotation over chaining variables C_i, D_i and E_i. A single rotation is introduced when C_i is updated. This rotated value is re-used in chaining variables D_i and E_i. If a set of difference equations is obtained, the use of the rotation limits the number of choices which could be made for the chaining variables involved. This in turn reduces the number of solutions which are easily obtained for the difference equations. It is these factors which, at present, prevent the proposed attack on the first two rounds of SHA to be successful. If improved solution techniques becomes available, it may become possible to execute the proposed attack and be able to construct messages which result in collisions for the first two rounds of SHA.

CHAPTER 10: ANALYSIS OF THE HAVAL HASH ALGORITHM

10.1 INTRODUCTION

In this chapter the HAVAL hash function is analysed within the generalised framework presented in Chapter 8. First we describe the HAVAL hash function and the relevant notation needed in this chapter. We then show how the generalised attack formulated in Chapter 8 can be applied to the last two rounds of three round HAVAL to establish a collision. This is the first published cryptanalytical result for the HAVAL hash function. The source code that implements the attack is attached in Appendix G.

10.2 INTRODUCTION TO HAVAL

HAVAL is an iterated hash function based on the Dåmgard-Merkle scheme. The HAVAL construction is closely related to the MD4 family of hash functions. HAVAL was designed by Zheng, Pieprzyk and Seberry in 1994 [62]. The HAVAL hash function specification includes 15 variations [62]. These variations are based on the number of iterations (or rounds) used by the round function (3, 4 or 5) as well as the number bits used as output (128, 160, 192, 224 or 256). The round function of HAVAL takes message blocks in multiples of 1024 bits and produces an output of 256 bits which can then be reduced to 128, 160, 192, 224 or 256 bits, depending on the security requirements. In this dissertation we focus on three round HAVAL for all possible output lengths. This is the first time any cryptographic analysis of HAVAL has been made public. The analysis presented in this chapter may also be applied to 4 and 5 round HAVAL.

10.3 NOTATION

Before proceeding with a description of HAVAL and the cryptanalysis of HAVAL it is appropriate to introduce the notation to be used in this chapter. The following operators are

used in this chapter:

$$\begin{aligned} \vee &= \text{Bitwise OR.} \\ \oplus &= \text{Bitwise Exclusive OR (XOR).} \\ \overline{X} &= \text{Bitwise Complement of } X \\ X^{\ggg Y} &= \text{Bitwise rotation to the right of } X \text{ by } Y \text{ positions.} \\ \wedge &= \text{Bitwise AND} \end{aligned}$$

In this chapter the bitwise AND between two variables are often indicated by x_1x_2 rather than $x_1 \wedge x_2$ for brevity. The notation of $ordi()$ for $i = 1, 2, 3, 4, 5$ indicates the word processing order for round i of the round function. The constants used by the HAVAL hash function are indicated by $K_{j,i}$ with $j = 2, 3, 4, 5$ and $i \in \{32, 33, 34, \ldots, 160\}$. There are a total of 128 additive constants used in the round functions and a further 8 constants that define the default initial values for HAVAL. The constants can be found in [62]. The constants are defined as the first 4352 bits of π. The 136 constants are not explicitly defined in this Chapter since they play no role in the analyses presented in this Chapter.

10.4 HAVAL

In this section a short description of the HAVAL hash function is presented. For a more complete description refer to [62].

Eight steps may be identified in the definition of HAVAL.

Algorithm 10.1 *HAVAL hash algorithm*

1. *Message padding.*
2. *Initialise chaining variables.*
3. *Order words for each round in the round function.*
4. *Apply compress function.*
5. *Update variables.*
6. *Has the entire message been processed ?*
   (a) *No: Repeat from step 3.*
   (b) *Yes: Continue.*
7. *Select appropriate output bits.*
8. *Output hash value.*

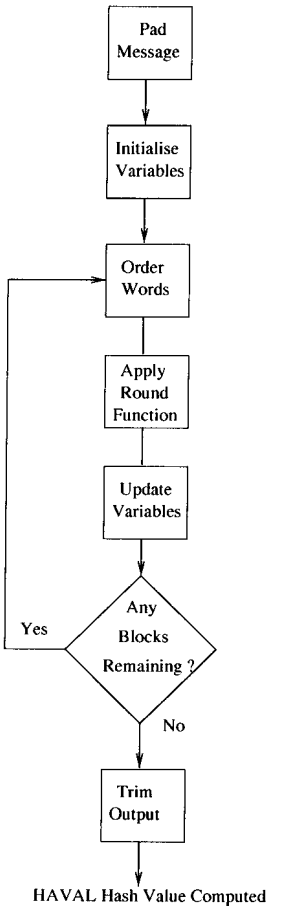

Figure 10.1: HAVAL Block Diagram

Step 1 ensures that the message is padded to a multiple of 1024 bits. Steps 3 to 5 are repeated for each 1024-bit block until the entire padded message has been processed. Step seven is used to construct the appropriate hash length (128, 160, 192, 224 or 256). Step seven is performed once the final message block is processed.

10.4.1 Message Padding

The round function of HAVAL requires a 1024-bit message block as input. Consequently the message to be hashed has to be a multiple of 1024-bits. This is accomplished by using a padding algorithm. Message padding is applied even if the unpadded message is a multiple of 1024 bits in length.

The message is padded by appending a 1 to the message, followed by m 0's ($m \leq 0$) until the length of padded message equals $944 \bmod 1024$. Once the message is padded to this length a three bit VERSION field, followed by a 3-bit PASS field and a 10-bit FPTFIELD is appended to the padded message. Once these fields are appended a 64-bit MSGLEN field is appended to form a message with a length that is a multiple of 1024. The fields mentioned above has the following meaning:

VERSION: This is a 3-bit field representing the version of HAVAL in use. The current version is 1.

PASS: This is a 3-bit field defining the number of rounds or iterations used by the round function of HAVAL. Valid values are 3, 4 or 5.

FPTFIELD: This is 10-bit field that specifies the length of the hash result. Valid values are 128, 160, 192, 224 or 256.

MSGLEN: This is a 64-bit field representing the length of the original message. The 64-bit integer is the binary representation of the length of the original message l. If l is less than 2^{32} the first 32 bits of the final 64 bits are zero.

10.4.2 Initialise Chaining Variables

The eight chaining variables are initialised to the following hexadecimal values.

$$A_0 = \texttt{0xEC4E6C89} \qquad E_0 = \texttt{0x03707344}$$
$$B_0 = \texttt{0x082EFA98} \qquad F_0 = \texttt{0x13198A2E}$$
$$C_0 = \texttt{0x299F31D0} \qquad G_0 = \texttt{0x85A308D3}$$
$$D_0 = \texttt{0xA4093822} \qquad H_0 = \texttt{0x243F6A88.}$$

10.4.3 Word Processing Order

The 1024-bit message block is divided into 32 words of 32 bits each. These words are processed in a different order for each round or iteration of the round function. The word

processing order for HAVAL is shown in Table 10.1.

ord1()	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ord2()	5	14	26	18	11	28	7	16	0	23	20	22	1	10	4	8
	30	3	21	9	17	24	29	6	19	12	15	13	2	25	31	27
ord3()	19	9	4	20	28	17	8	22	29	14	25	12	24	30	16	26
	31	15	7	3	1	0	18	27	13	6	21	10	23	11	5	2
ord4()	24	4	0	14	2	7	28	23	26	6	30	20	18	25	19	3
	22	11	31	21	8	27	12	9	1	29	5	15	17	10	16	13
ord5()	27	3	21	26	17	11	20	29	19	0	12	7	13	8	31	10
	5	9	14	30	18	6	28	24	2	23	16	22	4	1	25	15

Table 10.1: HAVAL Word Processing Order

10.4.4 Compress Function

The compress or round function is discussed in this section. Each round in the compress function utilise a different boolean function and a different word order. Each boolean function takes seven bits as input and produces a single bit as output. The boolean functions are expanded to form boolean mappings by operating bitwise on 32-bit words. The boolean mappings used in HAVAL are defined by equations (10.1) to (10.5).

$$F1(B_i, C_i, D_i, E_i, F_i, G_i, H_i) = G_iD_i \oplus F_iC_i \oplus E_iB_i \oplus H_iG_i \oplus H_i. \quad (10.1)$$

$$F2(B_i, C_i, D_i, E_i, F_i, G_i, H_i) = G_iF_iE_i \oplus C_iD_iF_i \oplus G_iF_i \oplus G_iD_i \oplus F_iB_i \oplus C_iE_i \oplus C_iD_i \oplus H_iF_i \oplus H_i. \quad (10.2)$$

$$F3(B_i, C_i, D_i, E_i, F_i, G_i, H_i) = G_iF_iE_i \oplus G_iD_i \oplus F_iC_i \oplus E_iB_i \oplus E_iH_i \oplus H_i \quad (10.3)$$

$$F4(B_i, C_i, D_i, E_i, F_i, G_i, H_i) = G_iF_iE_i \oplus F_iD_iC_i \oplus E_iD_iB_i \oplus G_iD_i \oplus F_iB_i \oplus E_iD_i \oplus E_iC_i \oplus E_iB_i \oplus D_iC_i \oplus D_iB_i \oplus H_iD_i \oplus H_i. \quad (10.4)$$

$$F5(B_i, C_i, D_i, E_i, F_i, G_i, H_i) = G_iD_i \oplus F_iC_i \oplus E_iB_i \oplus H_iG_iF_iE_i \oplus H_iC_i \oplus H_i. \quad (10.5)$$

Round 1

The first round of the HAVAL hash function is repeating the following steps for $0 \leqslant i \leqslant 31$.

$$
\begin{aligned}
1)P &= \begin{cases} F1 \circ \phi_{3,1}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=3} \\ F1 \circ \phi_{4,1}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=4} \\ F1 \circ \phi_{5,1}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=5} \end{cases} \\
2)R &= P^{\ggg 7} + A_i^{\ggg 11} + W_{ord1(i)}. \\
3)A_{i+1} &= B_i, \quad B_{i+1} = C_i, \quad C_{i+1} = D_i, \quad D_{i+1} = E_i. \\
E_{i+1} &= F_i, \quad F_{i+1} = G_i, \quad G_{i+1} = H_i, \quad H_{i+1} = R.
\end{aligned}
$$

Note that the permutation defined by $\phi_{j,1}$, $j \in \{3, 4, 5\}$ is performed before the function $F1$ is executed. The permutations are defined in Table 10.2.

Permutations	B	C	D	E	F	G	H
	↓	↓	↓	↓	↓	↓	↓
$\phi_{3,1}$	G	H	E	C	B	F	D
$\phi_{3,2}$	D	D	G	H	C	E	B
$\phi_{3,3}$	B	G	F	E	D	C	H
$\phi_{4,1}$	F	B	G	D	C	E	H
$\phi_{4,2}$	E	C	F	H	G	B	D
$\phi_{4,3}$	G	D	E	B	H	F	C
$\phi_{4,4}$	B	D	H	C	F	G	C
$\phi_{5,1}$	E	D	G	H	C	F	B
$\phi_{5,2}$	B	F	G	H	E	D	C
$\phi_{5,3}$	F	B	H	D	E	G	C
$\phi_{5,4}$	G	C	E	F	H	D	B
$\phi_{5,5}$	F	C	H	B	D	E	G

Table 10.2: Permutations (Note "↓" indicates "replace by")

Round 2

The second round of the HAVAL hash function is repeating the following steps for $32 \leqslant i \leqslant 63$.

$$
\begin{aligned}
1)P &= \begin{cases} F2 \circ \phi_{3,2}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=3} \\ F2 \circ \phi_{4,2}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=4} \\ F2 \circ \phi_{5,2}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=5} \end{cases} \\
2)R &= P^{\ggg 7} + A_i^{\ggg 11} + W_{ord2(i)} + K_{2,i}. \\
3)A_{i+1} &= B, \quad B_{i+1} = C_i, \quad C_{i+1} = D_i, \quad D_{i+1} = E_i. \\
E_{i+1} &= F_i, \quad F_{i+1} = G_i, \quad G_{i+1} = H_i, \quad H_{i+1} = R.
\end{aligned}
$$

Note that the permutation defined by $\phi_{j,2}$, $j \in \{3, 4, 5\}$ is performed before the function $F2$ is executed. The permutations are defined in Table 10.2.

Round 3

The third round of the HAVAL hash function is repeating the following steps for $64 \leqslant i \leqslant 95$.

$$
\begin{aligned}
1)P &= \begin{cases} F3 \circ \phi_{3,3}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=3} \\ F3 \circ \phi_{4,3}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=4} \\ F3 \circ \phi_{5,3}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=5} \end{cases} \\
2)R &= P^{\ggg 7} + A^{\ggg 11} + W_{ord3(i)} + K_{3,i}. \\
3)A &= B, \quad B = C, \quad C = D, \quad D = E. \\
E &= F, \quad F = G, \quad G = H, \quad H = R.
\end{aligned}
$$

Note that the permutation defined by $\phi_{j,3}$, $j \in \{3, 4, 5\}$ is performed before the function $F3$ is executed. The permutations are defined in Table 10.2.

Round 4

The fourth round of the HAVAL hash function is repeating the following steps for $96 \leqslant i \leqslant 127$.

$$\begin{aligned} 1)P &= \begin{cases} F4 \circ \phi_{4,4}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=4} \\ F4 \circ \phi_{5,4}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=5} \end{cases} \\ 2)R &= P^{\ggg 7} + A^{\ggg 11} + W_{ord4(i)} + K_{4,i}. \\ 3)A &= B, \quad B = C, \quad C = D, \quad D = E. \\ E &= F, \quad F = G, \quad G = H, \quad H = R. \end{aligned}$$

Note that the permutation defined by $\phi_{j,4}$, $j \in \{4, 5\}$ is performed before the function $F4$ is executed. The permutations are defined in Table 10.2.

Round 5

The 01 round of the HAVAL hash function is repeating the following steps for $128 \leqslant i \leqslant 159$.

$$\begin{aligned} 1)P &= \begin{cases} F5 \circ \phi_{5,5}(B_i, C_i, D_i, E_i, F_i, G_i, H_i) & \text{if PASS=5} \end{cases} \\ 2)R &= P^{\ggg 7} + A^{\ggg 11} + W_{ord5(i)} + K_{5,i}. \\ 3)A &= B, \quad B = C, \quad C = D, \quad D = E. \\ E &= F, \quad F = G, \quad G = H, \quad H = R. \end{aligned}$$

Note that the permutation defined by $\phi_{5,5}$ is performed before the function $F5$ is executed. The permutations are defined in Table 10.2.

A single step in a round of the round function is graphically represented as shown in Figure 10.2.

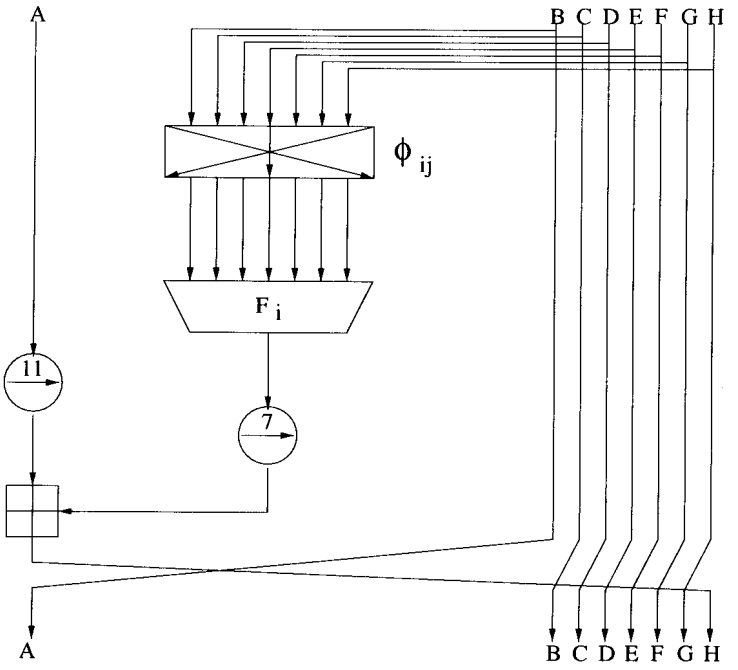

Figure 10.2: Single Step in HAVAL Hash Function

10.4.5 Tailoring the output

HAVAL can be used to produce a hash length of 128, 160, 192, 224 or 256 bits, depending on the security requirement. In this dissertation only the case where the output is 256-bits (maximum security) is considered. For a detailed description of the procedure used to select an output of less than 256 bits refer to [62]. Note that a collision for the 256-bit output implies a collision for all subsets of the output.

10.5 ANALYSIS OF HAVAL

In this section the principles derived in Chapter 8 are applied to HAVAL. In order to demonstrate the applicability of the attack it is shown that the last two rounds of three round HAVAL is not collision resistant. The last two rounds of three round HAVAL are described by steps 32 to 95. The equations describing the second round of three round HAVAL is described by

equation (10.6) for $i \in \{32, 33, 34, \ldots 63\}$.

$$\begin{aligned} H_{i+1} &= (F2 \circ \phi_{3,2}(B_i, C_i, D_i, E_i, F_i, G_i, H_i))^{\ggg 7} + A_i^{\ggg 11} + W_{ord2(i)} + K_{2,i}. \\ A_{i+1} &= B_i, \quad B_{i+1} = C_i, \quad C_{i+1} = D_i, \quad D_{i+1} = E_i \\ E_{i+1} &= F_i, \quad F_{i+1} = G_i, \quad G_{i+1} = H_i. \end{aligned} \tag{10.6}$$

The equations describing the last round of three round HAVAL is described by equation (10.7) for $i \in \{64, 65, 66, \ldots 95\}$.

$$\begin{aligned} H_{i+1} &= (F3 \circ \phi_{3,3}(B_i, C_i, D_i, E_i, F_i, G_i, H_i))^{\ggg 7} + A_i^{\ggg 11} + W_{ord3(i)} + K_{3,i}. \\ A_{i+1} &= B_i, \quad B_{i+1} = C_i, \quad C_{i+1} = D_i, \quad D_{i+1} = E_i \\ E_{i+1} &= F_i, \quad F_{i+1} = G_i, \quad G_{i+1} = H_i. \end{aligned} \tag{10.7}$$

10.5.1 Difference Equations

In order to establish a collision for the last two rounds of three round HAVAL an inner collision has to be established as described in Chapter 8. It is possible to derive a set of difference equations that allows a collision for the full 256-bit output of the last two round of three round HAVAL. One approach which results in an inner collision for the last two rounds of HAVAL is described below.

Consider steps 56 to 64 of the HAVAL as represented by equations (10.8) to (10.16).

$$\begin{aligned}
H_{57} &= (F2 \circ \phi_{3,2}(B_{56}, C_{56}, D_{56}, E_{56}, F_{56}, G_{56}, H_{56}))^{\ggg 7} + A_{56}^{\ggg 11} + W_{ord2(56)} + K_{2,56}. \\
A_{57} &= B_{56}, \quad B_{57} = C_{56}, \quad C_{57} = D_{56}, \quad D_{57} = E_{56} \\
E_{57} &= F_{56}, \quad F_{57} = G_{56}, \quad G_{57} = H_{56}. \qquad (10.8) \\
H_{58} &= (F2 \circ \phi_{3,2}(B_{57}, C_{57}, D_{57}, E_{57}, F_{57}, G_{57}, H_{57}))^{\ggg 7} + A_{57}^{\ggg 11} + W_{ord2(57)} + K_{2,57}. \\
A_{58} &= B_{57}, \quad B_{58} = C_{57}, \quad C_{58} = D_{57}, \quad D_{58} = E_{57} \\
E_{58} &= F_{57}, \quad F_{58} = G_{57}, \quad G_{58} = H_{57}. \qquad (10.9) \\
H_{59} &= (F2 \circ \phi_{3,2}(B_{58}, C_{58}, D_{58}, E_{58}, F_{58}, G_{58}, H_{58}))^{\ggg 7} + A_{58}^{\ggg 11} + W_{ord2(58)} + K_{2,58}. \\
A_{59} &= B_{58}, \quad B_{59} = C_{58}, \quad C_{59} = D_{58}, \quad D_{59} = E_{58} \\
E_{59} &= F_{58}, \quad F_{59} = G_{58}, \quad G_{59} = H_{58}. \qquad (10.10) \\
H_{60} &= (F2 \circ \phi_{3,2}(B_{59}, C_{59}, D_{59}, E_{59}, F_{59}, G_{59}, H_{59}))^{\ggg 7} + A_{59}^{\ggg 11} + W_{ord2(59)} + K_{2,59}. \\
A_{60} &= B_{59}, \quad B_{60} = C_{59}, \quad C_{60} = D_{59}, \quad D_{60} = E_{59} \\
E_{60} &= F_{59}, \quad F_{60} = G_{59}, \quad G_{60} = H_{59}. \qquad (10.11) \\
H_{61} &= (F2 \circ \phi_{3,2}(B_{60}, C_{60}, D_{60}, E_{60}, F_{60}, G_{60}, H_{60}))^{\ggg 7} + A_{60}^{\ggg 11} + W_{ord2(60)} + K_{2,60}. \\
A_{61} &= B_{60}, \quad B_{61} = C_{60}, \quad C_{61} = D_{60}, \quad D_{61} = E_{60} \\
E_{61} &= F_{60}, \quad F_{61} = G_{60}, \quad G_{61} = H_{60}. \qquad (10.12) \\
H_{62} &= (F2 \circ \phi_{3,2}(B_{61}, C_{61}, D_{61}, E_{61}, F_{61}, G_{61}, H_{61}))^{\ggg 7} + A_{61}^{\ggg 11} + W_{ord2(61)} + K_{2,61}. \\
A_{62} &= B_{61}, \quad B_{62} = C_{61}, \quad C_{62} = D_{61}, \quad D_{62} = E_{61} \\
E_{62} &= F_{61}, \quad F_{62} = G_{61}, \quad G_{62} = H_{61}. \qquad (10.13) \\
H_{63} &= (F2 \circ \phi_{3,2}(B_{62}, C_{62}, D_{62}, E_{62}, F_{62}, G_{62}, H_{62}))^{\ggg 7} + A_{62}^{\ggg 11} + W_{ord2(62)} + K_{2,62}. \\
A_{63} &= B_{62}, \quad B_{63} = C_{62}, \quad C_{63} = D_{62}, \quad D_{63} = E_{62} \\
E_{63} &= F_{62}, \quad F_{63} = G_{62}, \quad G_{63} = H_{62}. \qquad (10.14) \\
H_{64} &= (F2 \circ \phi_{3,2}(B_{63}, C_{63}, D_{63}, E_{63}, F_{63}, G_{63}, H_{63}))^{\ggg 7} + A_{63}^{\ggg 11} + W_{ord2(63)} + K_{2,63}. \\
A_{64} &= B_{63}, \quad B_{64} = C_{63}, \quad C_{64} = D_{63}, \quad D_{64} = E_{63} \\
E_{64} &= F_{63}, \quad F_{64} = G_{63}, \quad G_{64} = H_{63}. \qquad (10.15) \\
H_{65} &= (F3 \circ \phi_{3,3}(B_{64}, C_{64}, D_{64}, E_{64}, F_{64}, G_{64}, H_{64}))^{\ggg 7} + A_{64}^{\ggg 11} + W_{ord3(64)} + K_{3,64}. \\
A_{65} &= B_{64}, \quad B_{65} = C_{64}, \quad C_{65} = D_{64}, \quad D_{65} = E_{64} \\
E_{65} &= F_{64}, \quad F_{65} = G_{64}, \quad G_{65} = H_{64}. \qquad (10.16)
\end{aligned}$$

Note that $ord2(56)$ and $ord3(64)$ both select message word 19 (W_{19}). Consider two messages, M and $\tilde{M}$ that differ only in message word 19. It is then possible to derive a set of

difference equations using the principles stated in Chapter 8. In the notation of Chapter 8 we show that a message can be constructed such that:

$$f_{56}^{64}(T, M) = f_{56}^{64}(T, \tilde{M}).$$

where T represents the chaining variables $A_{56}, B_{56}, C_{56}, D_{56}, E_{56}, F_{56}, G_{56}$ and H_{56}. Once the inner collision is established it is shown that:

$$f_{32}^{95}(IV, M) = f_{32}^{95}(IV, \tilde{M}).$$

for an arbitrarily chosen IV.

The required inner collision can be established by solving the following set of difference equations

$$\begin{aligned}
H_{57} - \tilde{H}_{57} &= (F2 \circ \phi_{3,2}(B_{56}, C_{56}, D_{56}, E_{56}, F_{56}, G_{56}, H_{56}))^{\ggg 7} + A_{56}^{\ggg 11} + W_{ord2(56)} + K_{2,56} - \\
&\quad ((F2 \circ \phi_{3,2}(\tilde{B}_{56}, \tilde{C}_{56}, \tilde{D}_{56}, \tilde{E}_{56}, \tilde{F}_{56}, \tilde{G}_{56}, \tilde{H}_{56}))^{\ggg 7} + \tilde{A}_{56}^{\ggg 11} + \\
&\quad \tilde{W}_{ord2(56)} + K_{2,56}) \qquad (10.17) \\
H_{58} - \tilde{H}_{58} &= (F2 \circ \phi_{3,2}(B_{57}, C_{57}, D_{57}, E_{57}, F_{57}, G_{57}, H_{57}))^{\ggg 7} + A_{57}^{\ggg 11} + W_{ord2(57)} + K_{2,57} - \\
&\quad ((F2 \circ \phi_{3,2}(\tilde{B}_{57}, \tilde{C}_{57}, \tilde{D}_{57}, \tilde{E}_{57}, \tilde{F}_{57}, \tilde{G}_{57}, \tilde{H}_{57}))^{\ggg 7} + \tilde{A}_{57}^{\ggg 11} + \\
&\quad \tilde{W}_{ord2(57)} + K_{2,57}) \qquad (10.18) \\
H_{59} - \tilde{H}_{59} &= (F2 \circ \phi_{3,2}(B_{58}, C_{58}, D_{58}, E_{58}, F_{58}, G_{58}, H_{58}))^{\ggg 7} + A_{58}^{\ggg 11} + W_{ord2(58)} + K_{2,58} - \\
&\quad ((F2 \circ \phi_{3,2}(\tilde{B}_{58}, \tilde{C}_{58}, \tilde{D}_{58}, \tilde{E}_{58}, \tilde{F}_{58}, \tilde{G}_{58}, \tilde{H}_{58}))^{\ggg 7} + \tilde{A}_{58}^{\ggg 11} + \\
&\quad \tilde{W}_{ord2(58)} + K_{2,58}) \qquad (10.19) \\
H_{60} - \tilde{H}_{60} &= (F2 \circ \phi_{3,2}(B_{59}, C_{59}, D_{59}, E_{59}, F_{59}, G_{59}, H_{59}))^{\ggg 7} + A_{59}^{\ggg 11} + W_{ord2(59)} + K_{2,59} - \\
&\quad ((F2 \circ \phi_{3,2}(\tilde{B}_{59}, \tilde{C}_{59}, \tilde{D}_{59}, \tilde{E}_{59}, \tilde{F}_{59}, \tilde{G}_{59}, \tilde{H}_{59}))^{\ggg 7} + \tilde{A}_{59}^{\ggg 11} + \\
&\quad \tilde{W}_{ord2(59)} + K_{2,59}) \qquad (10.20)
\end{aligned}$$

$$
\begin{aligned}
H_{61} - \tilde{H}_{61} &= (F2 \circ \phi_{3,2}(B_{60}, C_{60}, D_{60}, E_{60}, F_{60}, G_{60}, H_{60}))^{\ggg 7} + A_{60}^{\ggg 11} + W_{ord2(60)} + K_{2,60} - \\
&\quad ((F2 \circ \phi_{3,2}(\tilde{B}_{60}, \tilde{C}_{60}, \tilde{D}_{60}, \tilde{E}_{60}, \tilde{F}_{60}, \tilde{G}_{60}, \tilde{H}_{60}))^{\ggg 7} + \tilde{A}_{60}^{\ggg 11} + \\
&\quad \tilde{W}_{ord2(60)} + K_{2,60}) \qquad (10.21) \\
H_{62} - \tilde{H}_{62} &= (F2 \circ \phi_{3,2}(B_{61}, C_{61}, D_{61}, E_{61}, F_{61}, G_{61}, H_{61}))^{\ggg 7} + A_{61}^{\ggg 11} + W_{ord2(61)} + K_{2,61} - \\
&\quad ((F2 \circ \phi_{3,2}(\tilde{B}_{61}, \tilde{C}_{61}, \tilde{D}_{61}, \tilde{E}_{61}, \tilde{F}_{61}, \tilde{G}_{61}, \tilde{H}_{61}))^{\ggg 7} + \tilde{A}_{61}^{\ggg 11} + \\
&\quad \tilde{W}_{ord2(61)} + K_{2,61}) \qquad (10.22) \\
H_{63} - \tilde{H}_{63} &= (F2 \circ \phi_{3,2}(B_{62}, C_{62}, D_{62}, E_{62}, F_{62}, G_{62}, H_{62}))^{\ggg 7} + A_{62}^{\ggg 11} + W_{ord2(62)} + K_{2,62} - \\
&\quad ((F2 \circ \phi_{3,2}(\tilde{B}_{62}, \tilde{C}_{62}, \tilde{D}_{62}, \tilde{E}_{62}, \tilde{F}_{62}, \tilde{G}_{62}, \tilde{H}_{62}))^{\ggg 7} + \tilde{A}_{62}^{\ggg 11} + \\
&\quad \tilde{W}_{ord2(62)} + K_{2,62}) \qquad (10.23) \\
H_{64} - \tilde{H}_{64} &= (F2 \circ \phi_{3,2}(B_{63}, C_{63}, D_{63}, E_{63}, F_{63}, G_{63}, H_{63}))^{\ggg 7} + A_{63}^{\ggg 11} + W_{ord2(63)} + K_{2,63} - \\
&\quad ((F2 \circ \phi_{3,2}(\tilde{B}_{63}, \tilde{C}_{63}, \tilde{D}_{63}, \tilde{E}_{63}, \tilde{F}_{63}, \tilde{G}_{63}, \tilde{H}_{63}))^{\ggg 7} + \tilde{A}_{63}^{\ggg 11} + \\
&\quad \tilde{W}_{ord2(63)} + K_{2,63}) \qquad (10.24) \\
H_{65} - \tilde{H}_{65} &= (F3 \circ \phi_{3,3}(B_{64}, C_{64}, D_{64}, E_{64}, F_{64}, G_{64}, H_{64}))^{\ggg 7} + A_{64}^{\ggg 11} + W_{ord3(64)} + K_{3,64} - \\
&\quad ((F3 \circ \phi_{3,3}(\tilde{B}_{64}, \tilde{C}_{64}, \tilde{D}_{64}, \tilde{E}_{64}, \tilde{F}_{64}, \tilde{G}_{64}, \tilde{H}_{64}))^{\ggg 7} + \tilde{A}_{64}^{\ggg 11} + \\
&\quad \tilde{W}_{ord3(64)} + K_{3,64}) \qquad (10.25)
\end{aligned}
$$

In order to establish an inner collision the following constraints are imposed:

$$
\begin{aligned}
H_{57} &\neq \tilde{H}_{57} \quad B_{56} = \tilde{B}_{56} \quad C_{56} = \tilde{C}_{56} \quad D_{56} = \tilde{D}_{56} \\
E_{56} &= \tilde{E}_{56} \quad F_{56} = \tilde{F}_{56} \quad G_{56} = \tilde{G}_{56} \quad H_{56} = \tilde{H}_{56} \\
W_{ord3(56)} &\neq \tilde{W}_{ord3(56)} \quad A_{56} = \tilde{A}_{56} \\
H_{65} &= \tilde{H}_{65} \quad B_{64} = \tilde{B}_{64} \quad C_{64} = \tilde{C}_{64} \quad D_{64} = \tilde{D}_{64} \\
E_{64} &= \tilde{E}_{64} \quad F_{64} = \tilde{F}_{64} \quad G_{64} = \tilde{G}_{64} \quad H_{64} = \tilde{H}_{64} \\
W_{ord3(64)} &\neq \tilde{W}_{ord3(64)}.
\end{aligned}
$$

The reader is reminded that $W_{ord3(64)} = W_{ord3(56)} = W_{19}$ and $\tilde{W}_{ord3(64)} = \tilde{W}_{ord3(56)} = \tilde{W}_{19}$.

Given these constraints the difference equations defined by (10.17) to (10.25) reduces to:

$$H_{57} - \tilde{H}_{57} = W_{19} - \tilde{W}_{19} \tag{10.26}$$

$$0 = (F2 \circ \phi_{3,2}(B_{57}, C_{57}, D_{57}, E_{57}, F_{57}, G_{57}, H_{57}))^{\ggg 7} - (F2 \circ \phi_{3,2}(B_{57}, C_{57}, D_{57}, E_{57}, F_{57}, G_{57}, \tilde{H}_{57}))^{\ggg 7} \tag{10.27}$$

$$0 = (F2 \circ \phi_{3,2}(B_{58}, C_{58}, D_{58}, E_{58}, F_{58}, G_{58}, H_{58}))^{\ggg 7} - (F2 \circ \phi_{3,2}(B_{58}, C_{58}, D_{58}, E_{58}, F_{58}, \tilde{G}_{58}, H_{58}))^{\ggg 7} \tag{10.28}$$

$$0 = (F2 \circ \phi_{3,2}(B_{59}, C_{59}, D_{59}, E_{59}, F_{59}, G_{59}, H_{59}))^{\ggg 7} - (F2 \circ \phi_{3,2}(B_{59}, C_{59}, D_{59}, E_{59}, \tilde{F}_{59}, G_{59}, H_{59}))^{\ggg 7} \tag{10.29}$$

$$0 = (F2 \circ \phi_{3,2}(B_{60}, C_{60}, D_{60}, E_{60}, F_{60}, G_{60}, H_{60}))^{\ggg 7} - (F2 \circ \phi_{3,2}(B_{60}, C_{60}, D_{60}, \tilde{E}_{60}, F_{60}, G_{60}, H_{60}))^{\ggg 7} \tag{10.30}$$

$$0 = (F2 \circ \phi_{3,2}(B_{61}, C_{61}, D_{61}, E_{61}, F_{61}, G_{61}, H_{61}))^{\ggg 7} - (F2 \circ \phi_{3,2}(B_{61}, C_{61}, \tilde{D}_{61}, E_{61}, F_{61}, G_{61}, H_{61}))^{\ggg 7}. \tag{10.31}$$

$$0 = (F2 \circ \phi_{3,2}(B_{62}, C_{62}, D_{62}, E_{62}, F_{62}, G_{62}, H_{62}))^{\ggg 7} - (F2 \circ \phi_{3,2}(B_{62}, \tilde{C}_{62}, D_{62}, E_{62}, F_{62}, G_{62}, H_{62}))^{\ggg 7}. \tag{10.32}$$

$$0 = (F2 \circ \phi_{3,2}(B_{63}, C_{63}, D_{63}, E_{63}, F_{63}, G_{63}, H_{63}))^{\ggg 7} - (F2 \circ \phi_{3,2}(\tilde{B}_{63}, C_{63}, D_{63}, E_{63}, F_{63}, G_{63}, H_{63}))^{\ggg 7}. \tag{10.33}$$

$$0 = A_{64}^{\ggg 11} + W_{19} - (\tilde{A}_{64}^{\ggg 11} + \tilde{W}_{19}) \tag{10.34}$$

From equations (10.26) to (10.34) it is observed that the chaining variables listed in Table 10.3 are affected when trying to solve the set of difference equations.

A_{56}	B_{56}	C_{56}	D_{56}	E_{56}	F_{56}	G_{56}	H_{56}	
A_{57}	B_{57}	C_{57}	D_{57}	E_{57}	F_{57}	G_{57}	H_{57}	$\tilde{H}_{57}$
A_{58}	B_{58}	C_{58}	D_{58}	E_{58}	F_{58}	G_{58}	$\tilde{G}_{58}$	H_{58}
A_{59}	B_{59}	C_{59}	D_{59}	E_{59}	F_{59}	$\tilde{F}_{59}$	G_{59}	H_{59}
A_{60}	B_{60}	C_{60}	D_{60}	E_{60}	$\tilde{E}_{60}$	F_{60}	G_{60}	H_{60}
A_{61}	B_{61}	C_{61}	D_{61}	$\tilde{D}_{61}$	E_{61}	F_{61}	G_{61}	H_{61}
A_{62}	B_{62}	C_{62}	$\tilde{C}_{62}$	D_{62}	E_{62}	F_{62}	G_{62}	H_{62}
A_{63}	B_{63}	$\tilde{B}_{63}$	C_{63}	D_{63}	E_{63}	F_{63}	G_{63}	H_{63}
A_{64}	$\tilde{B}_{64}$	B_{64}	C_{64}	D_{64}	E_{64}	F_{64}	G_{64}	H_{64}

Table 10.3: Affected Chaining Variables

Table 10.3 will be used to indicate which of the chaining variables are affected in each step of the solution of the set of difference equations. Once a chaining variable is chosen to have certain value it is marked in gray.

10.5.2 Solution to Differential Equations

In this section one technique that allows a solution to the set of differential equations described by equations (10.26) to (10.34) is described. In order to solve the set of differential equations the following properties of Boolean algebra are used [63].

For two boolean variables, denoted by x_1 and x_2, where $x_1 \neq x_2$, the following expressions hold:

$$x_1 \oplus x_1 = 0. \tag{10.35}$$

$$x_1 \wedge (x_1 \vee x_2) = x_1. \tag{10.36}$$

$$\begin{aligned} x_1 \wedge \overline{(x_1 \vee x_2)} &= x_1 \wedge \overline{x_1} \wedge \overline{x_2} \\ &= 0. \end{aligned} \tag{10.37}$$

These expressions aid in the solution of the set of differential equations defined by the next 8 steps.

Step 1

An inner collision can be established by finding a solution to equations (10.26) to (10.34). In order to find a solution to this set of difference equations it is useful to remember that $A_{64} = B_{63} = C_{62} = D_{61} = E_{60} = F_{59} = G_{58} = H_{57}$ and $\tilde{A}_{64} = \tilde{B}_{63} = \tilde{C}_{62} = \tilde{D}_{61} = \tilde{E}_{60} = \tilde{F}_{59} = \tilde{G}_{58} = \tilde{H}_{57}$. By taking the above relationships into account the following condition has to be satisfied in order to solve the set of equations.

$$H_{57} - \tilde{H}_{57} = W_{19} - \tilde{W}_{19} \tag{10.38}$$

$$H_{57}^{\ggg 11} - \tilde{A}_{57}^{\ggg 11} = \tilde{W}_{19} - W_{19} \tag{10.39}$$

This relationship can be simplified and is re-stated as:

$$\tilde{A}_{57}^{\ggg 11} - H_{57}^{\ggg 11} = H_{57} - \tilde{H}_{57}. \tag{10.40}$$

Let the difference between $H_{57} - \tilde{H}_{57}$ and $W_{19} - \tilde{W}_{19}$ be denoted by:

$$\Delta H_{57} = H_{57} - \tilde{H}_{57}. \tag{10.41}$$

$$\Delta W_{19} = W_{19} - \tilde{W}_{19}. \tag{10.42}$$

$$\Delta W_{19} = \Delta H_{57}. \tag{10.43}$$

In order for equation (10.40) to be satisfied, ΔH_{57} can be chosen as `0xAAAAAAAAA`, `0xAAAAAAAAB`, `0x55555555` or `0x55555556`. The probability that equations (10.38) and (10.39) holds for each of the possible values of ΔH_{57} are given in Table 10.4.

ΔH_{57}	Pr
0xAAAAAAAAA	0.112
0xAAAAAAAAB	0.448
0x555555555	0.448
0x555555556	0.110

Table 10.4: Probability that a given ΔH satisfies (10.38) and (10.39) for random values of H_{57}

Here H_{57} can be selected at random. Note that any choice for H_{57} implies a selection for $\tilde{H}_{57}$ through the chosen relationship of ΔH_{57}. If equation (10.38) and (10.39) holds for the chosen H_{57}, continue with Step 2. The affected chaining variables for Step 1 are shown in Table 10.5.

A_{56}	B_{56}	C_{56}	D_{56}	E_{56}	F_{56}	G_{56}	H_{56}	
A_{57}	B_{57}	C_{57}	D_{57}	E_{57}	F_{57}	G_{57}	H_{57}	$\tilde{H}_{57}$
A_{58}	B_{58}	C_{58}	D_{58}	E_{58}	F_{58}	G_{58}	$\tilde{G}_{58}$	H_{58}
A_{59}	B_{59}	C_{59}	D_{59}	E_{59}	F_{59}	$\tilde{F}_{59}$	G_{59}	H_{59}
A_{60}	B_{60}	C_{60}	D_{60}	E_{60}	$\tilde{E}_{60}$	F_{60}	G_{60}	H_{60}
A_{61}	B_{61}	C_{61}	D_{61}	$\tilde{D}_{61}$	E_{61}	F_{61}	G_{61}	H_{61}
A_{62}	B_{62}	C_{62}	$\tilde{C}_{62}$	D_{62}	E_{62}	F_{62}	G_{62}	H_{62}
A_{63}	B_{63}	$\tilde{B}_{63}$	C_{63}	D_{63}	E_{63}	F_{63}	G_{63}	H_{63}
A_{64}	$\tilde{A}_{64}$	B_{64}	C_{64}	D_{64}	E_{64}	F_{64}	G_{64}	H_{64}

Table 10.5: Affected Chaining Variables: Step 1

Step 2

Once suitable values for H_{57} and $\tilde{H}_{57}$ have been found, a solution has to be found to equation (10.33). Equation (10.33) can be solved by applying the permutation $\phi_{3,2}$ and considering the resulting function $F2$. Equation (10.33) can then be written as:

$$\begin{aligned} 0 \;=\; & (E_{63}C_{63}H_{63} \oplus F_{63}G_{63}C_{63} \oplus E_{63}C_{63} \oplus E_{63}G_{63} \oplus C_{63}D_{63} \oplus F_{63}H_{63} \oplus F_{63}G_{63} \oplus \\ & B_{63}C_{63} \oplus B_{63}) - (E_{63}C_{63}H_{63} \oplus F_{63}G_{63}C_{63} \oplus E_{63}C_{63} \oplus E_{63}G_{63} \oplus C_{63}D_{63} \oplus \\ & F_{63}H_{63} \oplus F_{63}G_{63} \oplus \tilde{B}_{63}C_{63} \oplus \tilde{B}_{63}). \end{aligned} \tag{10.44}$$

If only the terms that differ from each other are considered equation (10.44) reduces to:

$$0 \;=\; (B_{63}C_{63} \oplus B_{63}) - (\tilde{B}_{63}C_{63} \oplus \tilde{B}_{63}) \tag{10.45}$$

$$(B_{63}C_{63} \oplus B_{63}) \;=\; (\tilde{B}_{63}C_{63} \oplus \tilde{B}_{63}) \tag{10.46}$$

By setting C_{63} to:

$$C_{63} = B_{63} \vee \tilde{B}_{63}. \tag{10.47}$$

Consequently equation (10.46) reduces to:

$$(B_{63} \oplus B_{63}) \;=\; (\tilde{B}_{63} \oplus \tilde{B}_{63}) \tag{10.48}$$

$$0 \;=\; 0. \tag{10.49}$$

Thus our choice for C_{63} satisfies equation (10.46). The chaining variable table after the completion of step 2 is shown below.

A_{56}	B_{56}	C_{56}	D_{56}	E_{56}	F_{56}	G_{56}	H_{56}	
A_{57}	B_{57}	C_{57}	D_{57}	E_{57}	F_{57}	G_{57}	H_{57}	$\tilde{H}_{57}$
A_{58}	B_{58}	C_{58}	D_{58}	E_{58}	F_{58}	G_{58}	$\tilde{G}_{58}$	H_{58}
A_{59}	B_{59}	C_{59}	D_{59}	E_{59}	F_{59}	$\tilde{F}_{59}$	G_{59}	H_{59}
A_{60}	B_{60}	C_{60}	D_{60}	E_{60}	$\tilde{E}_{60}$	F_{60}	G_{60}	H_{60}
A_{61}	B_{61}	C_{61}	D_{61}	$\tilde{D}_{61}$	E_{61}	F_{61}	G_{61}	H_{61}
A_{62}	B_{62}	C_{62}	$\tilde{C}_{62}$	D_{62}	E_{62}	F_{62}	G_{62}	H_{62}
A_{63}	B_{63}	$\tilde{B}_{63}$	C_{63}	D_{63}	E_{63}	F_{63}	G_{63}	H_{63}
A_{64}	$\tilde{A}_{64}$	B_{64}	C_{64}	D_{64}	E_{64}	F_{64}	G_{64}	H_{64}

Table 10.6: Affected Chaining Variables: Step 2

Step 3

The next step is to solve equation (10.32). As before equation (10.32) can be solved by applying the permutation $\phi_{3,2}$ and considering the resulting function $F2$. Equation (10.32) can then be written as:

$$\begin{aligned} 0 &= (E_{62}C_{62}H_{62} \oplus F_{62}G_{62}C_{62} \oplus E_{62}C_{62} \oplus E_{62}G_{62} \oplus C_{62}D_{62} \oplus F_{62}H_{62} \oplus F_{62}G_{62} \oplus \\ &\quad B_{62}C_{62} \oplus B_{62}) - (E_{62}\tilde{C}_{62}H_{62} \oplus F_{62}G_{62}\tilde{C}_{62} \oplus E_{62}\tilde{C}_{62} \oplus E_{62}G_{62} \oplus \tilde{C}_{62}D_{62} \oplus \\ &\quad F_{62}H_{62} \oplus F_{62}G_{62} \oplus B_{62}\tilde{C}_{62} \oplus B_{62}). \end{aligned} \tag{10.50}$$

If only the terms that differ from each other are considered equation (10.50) reduces to:

$$\begin{aligned} 0 &= (E_{62}C_{62}H_{62} \oplus F_{62}G_{62}C_{62} \oplus E_{62}C_{62} \oplus C_{62}D_{62} \oplus B_{62}C_{62}) - \\ &\quad (E_{62}\tilde{C}_{62}H_{62} \oplus F_{62}G_{62}\tilde{C}_{62} \oplus E_{62}\tilde{C}_{62} \oplus \tilde{C}_{62}D_{62} \oplus B_{62}\tilde{C}_{62}). \end{aligned} \tag{10.51}$$

Equation (10.51) holds if the following expressions hold true:

$$E_{62}C_{62}H_{62} = E_{62}\tilde{C}_{62}H_{62}. \tag{10.52}$$

$$E_{62}C_{62} = E_{62}\tilde{C}_{62}. \tag{10.53}$$

$$F_{62}G_{62}C_{62} \oplus C_{62}D_{62} = \oplus F_{62}G_{62}\tilde{C}_{62} \oplus \tilde{C}_{62}D_{62}. \tag{10.54}$$

$$B_{62}C_{62} = B_{62}\tilde{C}_{62}. \tag{10.55}$$

Chaining variable B_{62} and E_{62} is chosen such that:

$$B_{62} = \overline{C_{62} \vee \tilde{C}_{62}}. \tag{10.56}$$

$$E_{62} = \overline{C_{62} \vee \tilde{C}_{62}}. \tag{10.57}$$

This choice of B_{62} reduces equation (10.55) to:

$$B_{62}C_{62} = B_{62}\tilde{C}_{62} \tag{10.58}$$

$$\overline{C}_{62}C_{62}\tilde{C}_{62} = \overline{\tilde{C}}_{62}\tilde{C}_{62}C_{62} \tag{10.59}$$

$$0 = 0. \tag{10.60}$$

Likewise equation (10.53) reduces to:

$$E_{62}C_{62} = E_{62}\tilde{C}_{62} \tag{10.61}$$

$$\overline{C}_{62}C_{62}\tilde{C}_{62} = \overline{\tilde{C}}_{62}\tilde{C}_{62}C_{62} \tag{10.62}$$

$$0 = 0. \tag{10.63}$$

Similarly equation (10.53) reduces to:

$$E_{62}C_{62}H_{62} = E_{62}\tilde{C}_{62}H_{62}. \quad (10.64)$$

$$\overline{C}_{62}C_{62}\tilde{C}_{62}H_{62} = \overline{\tilde{C}}_{62}\tilde{C}_{62}C_{62}H_{62}. \quad (10.65)$$

$$0 \cdot H_{62} = 0 \cdot H_{62}. \quad (10.66)$$

$$0 = 0. \quad (10.67)$$

Note that no explicit choice is made for H_{62}. Chaining variable D_{62} is already determined through the choice for C_{63}. By setting:

$$G_{62} = C_{62} \vee \tilde{C}_{62}. \quad (10.68)$$

$$F_{62} = C_{62} \vee \tilde{C}_{62}. \quad (10.69)$$

It is assured that equation (10.54) holds.

The chaining variable table after the completion of step 3 is shown in Table 10.7.

A_{56}	B_{56}	C_{56}	D_{56}	E_{56}	F_{56}	G_{56}	H_{56}	
A_{57}	B_{57}	C_{57}	D_{57}	E_{57}	F_{57}	G_{57}	H_{57}	$\tilde{H}_{57}$
A_{58}	B_{58}	C_{58}	D_{58}	E_{58}	F_{58}	G_{58}	$\tilde{G}_{58}$	H_{58}
A_{59}	B_{59}	C_{59}	D_{59}	E_{59}	F_{59}	$\tilde{F}_{59}$	G_{59}	H_{59}
A_{60}	B_{60}	C_{60}	D_{60}	E_{60}	$\tilde{E}_{60}$	F_{60}	G_{60}	H_{60}
A_{61}	B_{61}	C_{61}	D_{61}	$\tilde{D}_{61}$	E_{61}	F_{61}	G_{61}	H_{61}
A_{62}	B_{62}	C_{62}	$\tilde{C}_{62}$	D_{62}	E_{62}	F_{62}	G_{62}	H_{62}
A_{63}	B_{63}	$\tilde{B}_{63}$	C_{63}	D_{63}	E_{63}	F_{63}	G_{63}	H_{63}
A_{64}	$\tilde{A}_{64}$	B_{64}	C_{64}	D_{64}	E_{64}	F_{64}	G_{64}	H_{64}

Table 10.7: Affected Chaining Variables: Step 3

Step 4

In this step equation (10.31) is solved. Equation (10.31) can be solved by applying the permutation $\phi_{3,2}$ and considering the resulting function $F2$. Equation (10.31) then reduces

to:

$$\begin{aligned} 0 &= (E_{61}C_{61}H_{61} \oplus F_{61}G_{61}C_{61} \oplus E_{61}C_{61} \oplus E_{61}G_{61} \oplus C_{61}D_{61} \oplus F_{61}H_{61} \oplus \\ &\quad F_{61}G_{61} \oplus B_{61}C_{61} \oplus B_{61}) - (E_{61}C_{61}H_{61} \oplus F_{61}G_{61}C_{61} \oplus E_{61}C_{61} \oplus \\ &\quad E_{61}G_{61} \oplus C_{61}\tilde{D}_{61} \oplus F_{61}H_{61} \oplus F_{61}G_{61} \oplus B_{61}C_{61} \oplus B_{61}). \end{aligned} \tag{10.70}$$

Consider only the terms that differ from each other. Then equation (10.70) reduces to:

$$0 = (C_{61}D_{61}) - (C_{61}\tilde{D}_{61}). \tag{10.71}$$

The choice for B_{62} in Step 3 also ensures that equation (10.70) holds. Consequently the chaining variable table remains unchanged (see Table 10.7) after the completion of Step 4.

Step 5

In this step equation (10.30) is solved. Equation (10.30) can be solved by applying the permutation $\phi_{3,2}$ and considering the resulting function expression. Equation (10.30) can then be written as:

$$\begin{aligned} 0 &= (E_{60}C_{60}H_{60} \oplus F_{60}G_{60}C_{60} \oplus E_{60}C_{60} \oplus E_{60}G_{60} \oplus C_{60}D_{60} \oplus F_{60}H_{60} \oplus \\ &\quad F_{60}G_{60} \oplus B_{60}C_{60} \oplus B_{60}) - (\tilde{E}_{60}C_{60}H_{60} \oplus F_{60}G_{60}C_{60} \oplus \tilde{E}_{60}C_{60} \oplus \\ &\quad \tilde{E}_{60}G_{60} \oplus C_{60}D_{60} \oplus F_{60}H_{60} \oplus F_{60}G_{60} \oplus B_{60}C_{60} \oplus B_{60}). \end{aligned} \tag{10.72}$$

If only the terms that differ from each other are considered equation (10.72) reduces to:

$$0 = (E_{60}C_{60}H_{60} \oplus E_{60}C_{60} \oplus E_{60}G_{60}) - (\tilde{E}_{60}C_{60}H_{60} \oplus \tilde{E}_{60}C_{60} \oplus \tilde{E}_{60}G_{60}) \tag{10.73}$$

Equation (10.73) holds if the following expressions hold true:

$$E_{60}C_{60}H_{60} = \tilde{E}_{60}C_{60}H_{60}. \tag{10.74}$$

$$E_{60}C_{60} = \tilde{E}_{60}C_{60}. \tag{10.75}$$

$$E_{60}G_{60} = \tilde{E}_{60}G_{60}. \tag{10.76}$$

$$\tag{10.77}$$

The value for G_{60} is already determined by the choice made for E_{62} in Step 3. The value chosen in Step 3 for E_{62} satisfies equation (10.76). Similarly the previous choice for F_{62} implies that the value for H_{60} is fixed. Likewise it is found that this particular choice for F_{62} assures that equation (10.74) holds if C_{60} is chosen such that:

$$C_{60} = \overline{E_{60} \vee \tilde{E}_{60}}. \tag{10.78}$$

This choice for C_{60} also assures that equation (10.75) holds. The chaining variable table can now be updated to reflect the additional choices made. Table 10.8 is shown below.

A_{56}	B_{56}	C_{56}	D_{56}	E_{56}	F_{56}	G_{56}	H_{56}	
A_{57}	B_{57}	C_{57}	D_{57}	E_{57}	F_{57}	G_{57}	H_{57}	$\tilde{H}_{57}$
A_{58}	B_{58}	C_{58}	D_{58}	E_{58}	F_{58}	G_{58}	$\tilde{G}_{58}$	H_{58}
A_{59}	B_{59}	C_{59}	D_{59}	E_{59}	F_{59}	$\tilde{F}_{59}$	G_{59}	H_{59}
A_{60}	B_{60}	C_{60}	D_{60}	E_{60}	$\tilde{E}_{60}$	F_{60}	G_{60}	H_{60}
A_{61}	B_{61}	C_{61}	D_{61}	$\tilde{D}_{61}$	E_{61}	F_{61}	G_{61}	H_{61}
A_{62}	B_{62}	C_{62}	$\tilde{C}_{62}$	D_{62}	E_{62}	F_{62}	G_{62}	H_{62}
A_{63}	B_{63}	$\tilde{B}_{63}$	C_{63}	D_{63}	E_{63}	F_{63}	G_{63}	H_{63}
A_{64}	$\tilde{A}_{64}$	B_{64}	C_{64}	D_{64}	E_{64}	F_{64}	G_{64}	H_{64}

Table 10.8: Affected Chaining Variables: Step 5

Step 6

In this step equation (10.29) is solved. Equation (10.29) can be solved by applying the permutation $\phi_{3,2}$ and considering the resulting function expression. Equation (10.29) can then be written as:

$$\begin{aligned} 0 \;=\; & (E_{59}C_{59}H_{59} \oplus F_{59}G_{59}C_{59} \oplus E_{59}C_{59} \oplus E_{59}G_{59} \oplus C_{59}D_{59} \oplus F_{59}H_{59} \oplus \\ & F_{59}G_{59} \oplus B_{59}C_{59} \oplus B_{59}) - (E_{59}C_{59}H_{59} \oplus \tilde{F}_{59}G_{59}C_{59} \oplus E_{59}C_{59} \oplus \\ & E_{59}G_{59} \oplus C_{59}D_{59} \oplus \tilde{F}_{59}H_{59} \oplus \tilde{F}_{59}G_{59} \oplus B_{59}C_{59} \oplus B_{59}). \end{aligned} \tag{10.79}$$

If only the terms that differ from each other are considered equation (10.79) reduces to:

$$0 \;=\; (F_{59}G_{59}C_{59} \oplus F_{59}H_{59} \oplus F_{59}G_{59}) - (\tilde{F}_{59}G_{59}C_{59} \oplus \tilde{F}_{59}H_{59} \oplus \tilde{F}_{59}G_{59}) \tag{10.80}$$

Equation (10.80) holds if the following expressions hold true:

$$F_{59}G_{59}C_{59} \oplus F_{59}G_{59} = \tilde{F}_{59}G_{59}C_{59} \oplus \tilde{F}_{59}G_{59}. \quad (10.81)$$

$$F_{59}H_{59} = \tilde{F}_{59}H_{59}. \quad (10.82)$$

$$(10.83)$$

The values for G_{59} and H_{59} are already determined by the choices made for D_{62} and E_{62} in Step 3. The values chosen in Step 3 for D_{62} and E_{60} also satisfies equations (10.81) and (10.82) C_{59} is chosen such that:

$$C_{59} = F_{59} \vee \tilde{F}_{59}. \quad (10.84)$$

The updated chaining variable table is presented as Table 10.9.

A_{56}	B_{56}	C_{56}	D_{56}	E_{56}	F_{56}	G_{56}	H_{56}	
A_{57}	B_{57}	C_{57}	D_{57}	E_{57}	F_{57}	G_{57}	H_{57}	$\tilde{H}_{57}$
A_{58}	B_{58}	C_{58}	D_{58}	E_{58}	F_{58}	G_{58}	$\tilde{G}_{58}$	H_{58}
A_{59}	B_{59}	C_{59}	D_{59}	E_{59}	F_{59}	$\tilde{F}_{59}$	G_{59}	H_{59}
A_{60}	B_{60}	C_{60}	D_{60}	E_{60}	$\tilde{E}_{60}$	F_{60}	G_{60}	H_{60}
A_{61}	B_{61}	C_{61}	D_{61}	$\tilde{D}_{61}$	E_{61}	F_{61}	G_{61}	H_{61}
A_{62}	B_{62}	C_{62}	$\tilde{C}_{62}$	D_{62}	E_{62}	F_{62}	G_{62}	H_{62}
A_{63}	B_{63}	$\tilde{B}_{63}$	C_{63}	D_{63}	E_{63}	F_{63}	G_{63}	H_{63}
A_{64}	$\tilde{A}_{64}$	B_{64}	C_{64}	D_{64}	E_{64}	F_{64}	G_{64}	H_{64}

Table 10.9: Affected Chaining Variables: Step 6

Step 7

In this step equation (10.28) is solved. As before consider the expression obtained by applying the permutation $\phi_{3,2}$ to equation (10.28) and considering the resulting function expression. Equation (10.28) can then be written as:

$$\begin{aligned} 0 = {} & (E_{58}C_{58}H_{58} \oplus F_{58}G_{58}C_{58} \oplus E_{58}C_{58} \oplus E_{58}G_{58} \oplus C_{58}D_{58} \oplus F_{58}H_{58} \oplus - \\ & F_{58}G_{58} \oplus B_{58}C_{58} \oplus B_{58})(E_{58}C_{58}H_{58} \oplus F_{58}\tilde{G}_{58}C_{58} \oplus E_{58}C_{58} \oplus \\ & E_{58}\tilde{G}_{58} \oplus C_{58}D_{58} \oplus F_{58}H_{58} \oplus F_{58}\tilde{G}_{58} \oplus B_{58}C_{58} \oplus B_{58}). \end{aligned} \quad (10.85)$$

consider only the terms that differ from each other. Then equation (10.85) reduces to:

$$\begin{aligned} 0 &= (F_{58}G_{58}C_{58} \oplus E_{58}G_{58} \oplus F_{58}G_{58}) - \\ &\quad (F_{58}\tilde{G}_{58}C_{58} \oplus E_{58}\tilde{G}_{58} \oplus F_{58}\tilde{G}_{58}). \end{aligned} \tag{10.86}$$

Equation (10.86) holds if the following expressions hold true:

$$F_{58}G_{58}C_{58} = F_{58}\tilde{G}_{58}C_{58}. \tag{10.87}$$

$$E_{58}G_{58} = E_{58}\tilde{G}_{58}. \tag{10.88}$$

$$F_{58}G_{58} = F_{58}\tilde{G}_{58}. \tag{10.89}$$

$$\tag{10.90}$$

The value for F_{58} is determined by the choice for B_{62} in Step 5. This choice for F_{58} also ensures that equations (10.87) and (10.89) holds. The value for E_{58} is determined by the value chosen for C_{60} in step 5. This choice also assures that equation (10.88) holds. The chaining variable table remains unchanged after the completion of step 7.

Step 8

In this step equation (10.27) is solved. As before consider the expression obtained by applying the permutation $\phi_{3,2}$ to equation (10.27) and considering the resulting function expression. Equation (10.27) can then be written as:

$$\begin{aligned} 0 &= (E_{57}C_{57}H_{57} \oplus F_{57}G_{57}C_{57} \oplus E_{57}C_{57} \oplus E_{57}G_{57} \oplus C_{57}D_{57} \oplus F_{57}H_{57} \oplus \\ &\quad F_{57}G_{57} \oplus B_{57}C_{57} \oplus B_{57}) - (E_{57}C_{57}\tilde{H}_{57} \oplus F_{57}G_{57}C_{57} \oplus E_{57}C_{57} \oplus \\ &\quad E_{57}G_{57} \oplus C_{57}D_{57} \oplus F_{57}\tilde{H}_{57} \oplus F_{57}G_{57} \oplus B_{57}C_{57} \oplus B_{57}). \end{aligned} \tag{10.91}$$

consider only the terms that differ from each other. Then equation (10.91) reduces to:

$$\begin{aligned} 0 &= (E_{57}C_{57}H_{57} \oplus F_{57}H_{57}) - \\ &\quad (E_{57}C_{57}\tilde{H}_{57} \oplus F_{57}\tilde{H}_{57}). \end{aligned} \tag{10.92}$$

Equation (10.92) holds if the following expressions hold.

$$E_{57}C_{57}H_{57} = E_{57}C_{57}\tilde{H}_{57}. \tag{10.93}$$

$$F_{57}H_{57} = F_{57}\tilde{H}_{57}. \tag{10.94}$$

The value for F_{57} is determined by the choice for C_{60} in step 5. This choice also satisfies equation (10.94). Similarly the value of E_{57} is determined by the choice for C_{59} in step 6. Given this value for E_{57} equation (10.93) holds if C_{57} is chosen such that:

$$C_{57} = \overline{H_{57} \vee \tilde{H}_{57}}. \tag{10.95}$$

The updated chaining variable table is shown in Table 10.10.

A_{56}	B_{56}	C_{56}	D_{56}	E_{56}	F_{56}	G_{56}	H_{56}	
A_{57}	B_{57}	C_{57}	D_{57}	E_{57}	F_{57}	G_{57}	H_{57}	$\tilde{H}_{57}$
A_{58}	B_{58}	C_{58}	D_{58}	E_{58}	F_{58}	G_{58}	G_{58}	H_{58}
A_{59}	B_{59}	C_{59}	D_{59}	E_{59}	F_{59}	$\tilde{F}_{59}$	G_{59}	H_{59}
A_{60}	B_{60}	C_{60}	D_{60}	E_{60}	$\tilde{E}_{60}$	F_{60}	G_{60}	H_{60}
A_{61}	B_{61}	C_{61}	D_{61}	$\tilde{D}_{61}$	E_{61}	F_{61}	G_{61}	H_{61}
A_{62}	B_{62}	C_{62}	$\tilde{C}_{62}$	D_{62}	E_{62}	F_{62}	G_{62}	H_{62}
A_{63}	B_{63}	$\tilde{B}_{63}$	C_{63}	D_{63}	E_{63}	F_{63}	G_{63}	H_{63}
A_{64}	$\tilde{A}_{64}$	B_{64}	C_{64}	D_{64}	E_{64}	F_{64}	G_{64}	H_{64}

Table 10.10: Affected Chaining Variables: Step 8

After the completion of step 8 the set of difference equations defined by equations (10.26) to (10.34) are solved. The chaining variables not marked in gray can take on randomly selected values.

Message Construction

Once the set of difference equations is solved it remains to construct the two messages that will result in a collision for the last two rounds of three round HAVAL. In general a message word for the second round can be derived using the following equation:

$$W_{ord2(i)} = H_{i+1} - (F2 \circ \phi_{3,2}(B_i, C_i, D_i, E_i, F_i, G_i, H_i))^{\gg 7} - A_i^{\gg 11} - K_{2,i}. \tag{10.96}$$

By applying equation (10.96) for $32 \leqslant i \leqslant 63$ two messages, M and $\tilde{M}$ can be derived. In order to meet a specific initial value ($i = 32$) appropriate selections should be made for H_{32}, H_{33}, H_{34}, H_{35}, H_{36}, H_{37}, H_{38} and H_{39}. An implementation of this attack is included in Appendix G.

10.5.3 Collision Example

A collision for the last two rounds of HAVAL was constructed using the techniques described in this chapter.

For the initial values of:

$$\begin{aligned} A_{32} &= \texttt{0xEC4E6C89} \\ B_{32} &= \texttt{0x082EFA98} \\ C_{32} &= \texttt{0x299F31D0} \\ D_{32} &= \texttt{0xA4093822} \\ E_{32} &= \texttt{0x03707344} \\ F_{32} &= \texttt{0x13198A2E} \\ G_{32} &= \texttt{0x85A308D3} \\ H_{32} &= \texttt{0x243F6A88}. \end{aligned}$$

The derived message is defined by:

$$
\begin{aligned}
W_0 &= \texttt{0x3A379ED0} & W_{16} &= \texttt{0x79F23F4E}\\
W_1 &= \texttt{0x1EB81543} & W_{17} &= \texttt{0x9C1596E8}\\
W_2 &= \texttt{0x279C0073} & W_{18} &= \texttt{0xB62B4D8B}\\
W_3 &= \texttt{0xC9295C45} & W_{19} &= \texttt{0xDEC04668}\\
W_4 &= \texttt{0x6988BCBA} & W_{20} &= \texttt{0x4BA12694}\\
W_5 &= \texttt{0xEE1E55A2} & W_{21} &= \texttt{0x9D8DED5C}\\
W_6 &= \texttt{0xDE458436} & W_{22} &= \texttt{0x456CFCB4}\\
W_7 &= \texttt{0xB1C55B3C} & W_{23} &= \texttt{0x7253D2B9}\\
W_8 &= \texttt{0xBDA229EB} & W_{24} &= \texttt{0x61ED5DB4}\\
W_9 &= \texttt{0xE27926BE} & W_{25} &= \texttt{0xE4C2E748}\\
W_{10} &= \texttt{0x8BACAC22} & W_{26} &= \texttt{0xFD80A2AD}\\
W_{11} &= \texttt{0xBDF710B4} & W_{27} &= \texttt{0xC033F56E}\\
W_{12} &= \texttt{0x6516723B} & W_{28} &= \texttt{0x3010FDA9}\\
W_{13} &= \texttt{0x26991773} & W_{29} &= \texttt{0x344A7F71}\\
W_{14} &= \texttt{0x9EA6FD1F} & W_{30} &= \texttt{0x0DB561C7}\\
W_{15} &= \texttt{0x0BB27961} & W_{31} &= \texttt{0xC7A1E175}
\end{aligned}
$$

The alternative message is identical to the first with the exception that $\tilde{W}_{19}$ is chosen such that:

$$\tilde{W}_{19} = \texttt{0x34159BBD}$$

The resulting collision (including the feed forward step) for the last two rounds of three round HAVAL is:

$$
\begin{aligned}
f_{32}^{95}(IV, W) &= \texttt{0x4DF09D997588F9C7BE20863B2EED2AAC}\\
&\quad\ \texttt{D5B1E116D927279E250D19CB00850706}.
\end{aligned}
$$

10.6 CONCLUSION

In this chapter it was shown that the generalised technique described in Chapter 8 can be applied to the HAVAL hash function. In particular it is shown that the last two rounds of three

round HAVAL is not collision resistant. It is demonstrated that a collision can be established for all 256 output bits produced. This attack is considerably more efficient than the birthday attack which would require 2^{128} evaluations of the last two rounds of HAVAL. This is the first cryptanalytical results obtained for the HAVAL hash function. It is believed that it is possible to extend the attack to all three rounds of HAVAL. The attack can be executed on a 200 MHz Pentium Pro in less than a minute. Source code that implements this attack is attached as Appendix G.

CHAPTER 11: DESIGN CRITERIA FOR DEDICATED HASH FUNCTIONS

11.1 INTRODUCTION

In this chapter guidelines and design criteria for the design of dedicated hash functions based on the MD4 family of hash functions are presented. The discussion of the design criteria in this chapter is based on the experience gained from the analysis of dedicated hash functions included in the MD4 family of hash functions. These hash functions include MD4, MD5, HAVAL, SHA and SHA-1. These hash functions share a common ancestry (MD4), and consequently they share a number of features. These hash functions also differ in a number of respects, such as the Boolean mappings and the message word re-use mechanisms.

Each of the components encountered in the MD4 family of hash functions is discussed with regard to their contribution to the security of the hash functions in which they occur. In this chapter the emphasis is on the security requirements expected from the building blocks with occasional reference to the functional requirements.

11.2 BASIC STRUCTURE

The compress function construction used for MD4 is described in [10]. This construction has been widely adopted in the design of other hash functions such as MD5 [45], SHA, SHA-1 [13] and Tiger [47]

The MD4 family of hash functions takes two parameters as inputs namely the previous hash result and the message block. If the first message block is processed, the previous hash result is replaced by the initial value. The generalised MD4 family construction does not allow for the inclusion of a secret key. A survey of a number of adaptations of this construction that does make allowance for a secret key is presented in Chapter 5.

The compress function used in the MD4 family of constructions, is an iterated construction. The compress function takes as input the previous hash result H_{i-1} and the current message block, M_i.

The compress function consists of a number of rounds. These rounds are constructed from a

number of steps. Each step is constructed from a number of elementary operations, including Boolean mappings, rotations and additions $\mathrm{mod} 2^{32}$.

The message block M_i is segmented into k sub-blocks. The initial chaining variable, C, is set equal to the previous hash result. The set of message sub-blocks is divided into a number of l-bit message words. These message words are re-used in consecutive rounds of the compress function according to a specified rule. The chaining variable, C, is updated in each step of each round of the compress function. The output of the compress function is obtained by adding the initial value of the chaining variable C (i.e. the previous output of the compress function or the initial value) to the final value of the chaining variable. The final hash value for the message is the output obtained from the final application of the compress function.

11.3 BUILDING BLOCKS

A number of basic building blocks are used in the construction of the compress functions of the MD4 family of hash functions. These include message expansion algorithms, message block permutations, rotations, addition $\mathrm{mod}\ 2^{32}$ and Boolean mappings or S-boxes. In this section the contribution of each of these basic building blocks to the security of the dedicated hash functions is considered.

11.3.1 Boolean Mappings

The Boolean mappings used in MD4, MD5, SHA, SHA-1 and HAVAL are constructed with the same technique and exhibit similar properties. The Boolean mappings used by these functions take a number of 32 bit input words and produce a single 32 bit output word.

The Boolean mappings utilised by these hash functions are constructed from Boolean functions. A number of design criteria for Boolean functions are established in Chapter 3 of [51]. These criteria deal with the zero-one balance, high non-linearity values and the propagation properties of the Boolean functions. In the definition of MD4 [10] and MD5 [45] it is stated that if the inputs to the Boolean function are independent and unbiased, then the output of the Boolean function will be independent and unbiased. The functions defined for MD4 and MD5 are used in both SHA and SHA-1. It should be noted that a number of the

Boolean functions defined for use with MD4 and MD5 do not satisfy the criteria set forth in [51]. HAVAL represents a family of hash functions derived from MD4 and is defined in [62]. The Boolean functions used in HAVAL are derived from bent functions and were designed to have zero-one balance, a high non-linearity and satisfy the strict avalanche criterion (SAC). In addition these functions are linearly inequivalent in structure and are mutually output-uncorrelated.

Having obtained a Boolean function which satisfies the desired properties, a Boolean mapping is constructed by applying the Boolean function to a number of bits in parallel. Boolean mappings constructed in this manner inherits the properties of the Boolean function. The number of bits are chosen to reflect a specific computer architecture. Currently 32 bit machines are in common use and consequently the Boolean mappings are defined over 32 bit variables. Many general purpose processors contain logical bitwise operators in their instruction sets. Thus the use of bitwise logical functions as Boolean mappings are advantageous, if the overall performance of the hash function is considered.

It is maintained that the design criteria applied to construct the Boolean mappings used by the MD4 family are necessary, but not sufficient. The practice of extending the Boolean function to a Boolean mapping by applying the Boolean function in parallel to a number of bits, has proved to be a salient point in the cryptanalysis of MD4, MD5 and HAVAL (Chapters 6 7 10). If a single input bit used by the Boolean mapping is changed, at most a single output bit of the Boolean mapping is changed. In addition, the compliance to the SAC by the Boolean function used to construct the Boolean mapping, implies that a single input bit may be changed without any changes occurring in the output of the Boolean mapping. For these reasons it is proposed that the Boolean mapping should be constructed to satisfy the bit independence criterion (BIC) as described in [51].

The only dedicated hash function which uses randomly generated Boolean mappings is Tiger [47]. This hash function was designed with 64-bit architectures in mind. For this reason it utilises four 8×64 bit S-boxes. The following design criteria for the Boolean mappings are quoted verbatim from [47].

1. All the entries of all the S boxes should be distinct. Moreover, no two entries should have more than three equal bytes.

2. Each byte-column is a permutation of all the 256 possible byte values.

3. The columns of all the S boxes should be as different as possible, and have some long cycles.

4. No two differences of S box entries ($S_i(t_1) \oplus S_i(t_2)$ and $S_j(t_3) \oplus S_j(t_4)$) should have more than four equal bytes.

5. The speed of the generation should not be too slow, in order to enable applications to generate the S boxes on the fly.

6. This algorithm, and the structure of the S boxes of Tiger, were chosen in a way which reduces linear and differential properties, and similarities of these properties in the four S boxes (between other things, by reducing the number of S boxes, and making them independent, unlike some original idea we had, which was intended to reduce the total memory size of the S boxes, and by choosing the large S boxes, which reduce linear and differential properties).

7. The randomizing parameter is easy to remember.

Items 1-4 as well as item 6 address the security properties of the S-boxes while items 5 and 7 deal with the functional requirements of the S-boxes. It is claimed by the designers of Tiger that the use of randomly generated Boolean mappings increase the security of the hash function. No quantitative arguments are presented to support this statement.

The randomly generated Boolean mapping used in Tiger is more difficult to manipulate than the Boolean mappings constructed from Boolean functions which are applied in parallel. This is due to the design criteria which have to be met by the randomly generated Boolean mappings. The design criteria applied in the generation of the Boolean mappings used by Tiger has the effect that it is impossible to construct a collision for a given Boolean mapping. Furthermore, a change in a single input bit is likely to cause a difference in more than one output bit.

Thus, it appears that the use of well chosen, randomly generated Boolean Mappings, avoid the problems and potential weaknesses observed in the bitwise Boolean mappings. However, the use of randomly generated Boolean mappings may result in an increase of resource requirements, such as storage capability and processor time. It is recommended that random mappings are utilised for optimal security.

11.3.2 Rotation

The rotation operator is used by all members of the MD4 family of hash functions. Rotations may be described in a number of ways. In [48] a rotation is described as a bit permutation. The rotation of x by n bits may also be viewed as a linear feedback shift register of length d with a feedback polynomial

$$f(x) = x^d + 1$$

which is clocked n times.

The rotation operation provides diffusion of bits throughout the hash function. MD4 and MD5 contain a single rotation in each step. HAVAL, SHA and SHA-1 contain two rotations applied to different chaining variables in each step. The use of rotations may complicate the solution of the sets of difference equations obtained. In particular in SHA and SHA-1 the use of two rotations and the permanent effect of the second rotation impose certain limits on an attacker, since an attacker has to choose values for chaining variables, which are invariant to the rotations (see Chapter 9). It is shown in [14] and [58] that it is possible to counter these rotations and find solutions to the sets of difference equations containing these rotations. A similar result is shown in Chapter 10 for HAVAL. However, the absence of rotations would have reduced the workload for finding inner collisions considerably.

Thus, rotations complicate the task of an attacker (depending on the use of the rotation) and therefore contribute to the security of the hash function. The rotation amounts should be chosen to obtain optimal diffusion in the compress function of the hash function.

None of the dedicated hash functions based on MD4 utilises data dependent rotations. It is unknown if data dependent rotations add or subtract to the security of dedicated hash functions. It may be possible for an attacker to exploit the data dependence. From a security point of view, it is recommended that caution be exercised if data dependent rotations are used.

11.3.3 Message Word Reuse

The iterative hash scheme employed in the MD4 family of hash functions requires that a message, M, is segmented into a number of message blocks, M_i. Each message block is

processed by the compress function $f()$. The compress functions of the MD4 family of hash functions divide the message block, M_i, into a number of message words. Each bit of each of these message words is re-used at least twice by each of the compress functions used by the MD4 family of hash functions. Two techniques of message re-use are employed by the MD4 family of hash functions. The first technique applies a permutation which changes the order in which the message words are accessed in consecutive rounds. The second technique applies a message expansion algorithm which derives new message words from the original message words. Both techniques are considered in this section.

Message Permutation

Each of the hash functions derived from MD4 takes a message block of fixed length as input. The message block is then divided into message words of 32 bits each. The message block is used in every round of the hash function. The order in which the message words are processed in each of these rounds are determined by the message permutation. MD4, MD5 and HAVAL employs message permutations. By exploiting the order in which message words are accessed in consecutive rounds, an attacker may obtain a set of difference equations. If this set of difference equations are solvable it may be possible to find collisions for the hash function. If message permutations are to be used care should be taken to choose the permutations in such a manner as to prevent solvable difference equations to be derived. This may prove difficult, if it is remembered that it is possible to obtain a solvable difference equation set representing 12 consecutive steps in MD5. Consequently it is advised that message permutations are avoided in the construction of dedicated hash functions.

Message Expansion

SHA and SHA-1 uses a message expansion algorithm instead of a fixed message permutation. The properties of the message expansion algorithm used by SHA is described in Chapter 9. From the analysis of SHA and SHA-1 it appears that the use of a message expansion algorithm adds to the effort required to derive a set of difference equations. The danger exist that an attacker may manipulate the message expansion algorithm to find one or more possible sets of solvable difference equations. An example of how the message expansion algorithm used by SHA may be manipulated is presented in Chapter 9.

It is noted that the dedicated hash functions based on the MD4 construction may be used as block ciphers. Specifically the initial value may be taken as the plaintext and the message block taken as the key. The hash result then represents the ciphertext. Thus, if we use SHA in this manner, we obtain a block cipher with a 160 bit block length (the initial value) and a 512 bit key (the message word). [1] The analogy may be extended to the message expansion algorithm. In block cipher terminology the message expansion algorithm is the equivalent of the key schedule. It is known that weak key schedules weaken the associated block cipher [64], [65]. In particular weak key scheduling algorithms allow related key attacks [65]. These attacks exploit a chosen difference between two unknown keys and allow the recovery of the keys given a number of known or chosen plaintexts. A weak message expansion algorithm may weaken the associated hash function by allowing the construction of two or more messages, which upon expansion, exhibit a specified difference pattern. This may allow the derivation of a set (or sets) of solvable difference Equations, which result in collisions. Thus both strong key scheduling algorithms and strong message expansion algorithms attempt to limit the extent to which an attacker may exploit specified differences between two distinct keys or messages. It is therefore suggested that the message expansion algorithm should meet the same requirements as those set for key scheduling algorithms.

In [66] a number of design criteria for key scheduling algorithms are proposed. In particular it is advised that the key schedule provides some guarantee of key/ciphertext Strict Avalanche Criterion and Bit Independence Criterion. In addition each bit should be used by round $\frac{R}{2}$ of a R-round cipher. In [67] it is stated that the key schedule should have a high diffusion, and should behave irregularly with regard to the components of the round function of the block cipher.

In [67] a distinction is made between two kinds of key schedules namely pseudo-random key generation and key evolution. In the pseudo random approach the cipher key is used to seed a pseudo-random noise generator. The output of the pseudo-random noise generator serves as the round key. The relationship between the cipher key and the round keys generated in this way are complex. However, these schemes are slow and keys cannot be generated online (during encryption). Thus, pseudo-random round key generation schemes incurs a performance penalty. The key evolution strategy uses the cipher key as key to the first round and derives each round key from the previous round key by means of a transformation ψ.

[1] It should be noted that it is not recommended to use hash functions in this mode. Dedicated hash functions are not designed to be used as secret key encryption algorithms, and are likely to exhibit characteristics which make them susceptible to linear and differential cryptanalysis.

The process of key evolution is described by:

$$\begin{aligned} k^0 &= k \\ k^i &= \psi(k^{i-1}). \end{aligned}$$

The transform ψ may be described by bit permutations, rotations or elements from coding theory and abstract algebra. The advantage of this scheme is that it is fast and the round keys may be derived online. A disadvantage is that the underlying structure of the transform ψ may be exploited by an attacker.

From the preliminary study of SHA and SHA-1 it is observed that the key evolution strategy is used by the message expansion algorithms. As noted above it was shown that the underlying structure of the message expansion algorithm may be exploited to enable the construction of collisions for a limited number of rounds (see Chapter 9). In both [66] and [67] the importance of diffusion of key bits in the key scheduling algorithm is stressed. It is noted that the diffusion in the message expansion algorithm used by SHA is poor. Although the diffusion properties are improved by the addition of a rotation operator in SHA-1, it is unknown if it provides sufficient protection.

11.3.4 Addition mod 2^{32}

Addition mod 2^{32} is used by all the dedicated hash functions based on MD4. The addition mod 2^{32} contributes to the avalanche effect in the dedicated hash function. The contribution to the avalanche effect is ascribed to the propagation of the carry bit in addition operations.

11.3.5 Additive Constants

Additive constants are used by all members of the MD4 family of hash functions. MD4, SHA, SHA-1 and HAVAL use different constants in each round, while MD5 uses a different additive constant for each step. The use of an additive constant contributes little to the collision resistant property of the hash function. The attacks by Dobbertin, which results in collisions for hash functions require the derivation and solution of sets of difference equations. The constants are easily cancelled from these difference equations, and therefore does not contribute to the difficulty of solving the set of difference equations. The use of different

additive constants in each round does add to the pre-image resistance of a hash function. However the use of a different additive constant for each step requires additional storage capabilities and is considered unjustified if compared to the increase in security obtained (especially in terms of collision resistance).

11.3.6 Composition

A single step in each of the dedicated hash functions belonging to the MD4 family is composed by combining addition $\mod 2^{32}$, rotation operations and Boolean mappings or S-boxes. These steps are repeated a number of times (at least 48 times). A number of these steps represent a single round of the hash function. Each step compresses n 32-bit variables to a single 32 bit variable. In [57] the use of genetic algorithms to find solutions to expressions of the form encountered in dedicated hash functions is investigated. From the results in [57] it is observed that a single step in a dedicated hash function is neither collision resistant nor pre-image resistant. It is more difficult to find collisions for a number of these steps used iteratively. However it was shown that for MD4 [14] and MD5 [12] it is possible to find collisions for these iterated structures.

The Damgård-Merkle constructions [22] [23] shows that a collision resistant hash function may be constructed from a collision resistant function. The general design of the MD4 family resembles the iterative structure of the Damgård-Merkle construction. The security offered by the Damgård-Merkle construction would be attainable if the compress function of the dedicated hash functions are collision resistant. It is observed that the compress functions of these dedicated hash functions are themselves constructed by iteratively applying a number of similar steps. However, the ability to construct collisions for single steps and even a number of consecutive steps, (cryptanalysis of MD4 and MD5) implies that the level of security offered by the Damgård-Merkle construction can not be attained. It should however be noted that this does not imply that collisions are easily constructed for these hash functions.

11.4 CONCLUSION

In this chapter the basic building blocks encountered in dedicated hash functions are considered. Observations regarding the contribution of these building blocks to the security of dedicated hash functions derived from MD4 are made. The security offered by Boolean

mappings are considered. It is believed that randomly generated Boolean mappings offer more security than the bitwise Boolean mappings currently employed by the majority of the members of the MD4 family. The functional advantages of using bitwise Boolean mappings rather than randomly generated Boolean mappings outweighed the additional security obtained from randomly generated Boolean mappings in the design of the MD4 family (with the exception of Tiger). Rotations complicate the task of the cryptanalyst considerably and is considered a valuable building block. Specific attention is given to the message expansion algorithm. It is observed that the message expansion algorithm is similar to key schedule algorithms. Based on this observation and the results obtained from the analysis of the message expansion algorithms used by SHA and SHA-1, it is proposed that message expansion algorithms should exhibit properties similar to that required for key scheduling algorithms. Specific attention needs to be given to the diffusion properties of message expansion algorithms. Additive constants contribute little to the collision resistance property of the hash functions considered. Addition $\bmod 2^{32}$ adds to the diffusion process in the hash function due to the properties of the carry bit. The building blocks used in the MD4 family are combined into a single step, which is used iteratively. It is known that the individual steps are not collision resistant and consequently it is not known if the compress function constructed from these individual steps is collision resistant. If the step functions could be replaced by collision resistant one way functions, the resulting compress function would also be collision resistant and one-way according to the Damgård-Merkle construction. It is not known if collision resistant one way functions exist.

CHAPTER 12: CONCLUSION

12.1 DISCUSSION

Cryptographic hash functions are important primitive building blocks in information security. These functions form the corner stone of numerous authentication protocols, encryption algorithms and digital signatures. These cryptographic primitives are vital for creating a secure electronic commerce environment. Electronic commerce protocols such as SET and EMV rely on the existence of cryptographic hash functions. The two properties that make hash functions indispensable in cryptographic applications, are collision resistance and one-wayness. Throughout this dissertation we paid specific attention to the property of collision resistance. The property of collision resistance is vital for the non-repudiation service obtained through digital signatures.

However, designing hash functions that exhibit this property has proven to be extremely difficult. In the period from 1990 to 1994 a number of practical cryptographic hash functions were designed and implemented. These cryptographic hash functions include MD4, MD5, SHA, SHA-1, HAVAL, RIPEMD-128 and RIPEMD160. It was thought that these algorithms exhibited the properties of collision resistance and one-wayness. However in 1996 Dobbertin demonstrated that MD4 is not a collision resistant hash function. Within months the result was extended to RIPEMD-128 and MD5. One of the objectives of this dissertation was to generalise these attacks, apply it to other hash functions, and then derive design criteria that will defeat the generalised attack.

In pursuit of this objective a general introduction to cryptographic hash functions is presented in Chapters 1, 2, 3, 4. Chapter 1 introduced the relevant definitions and concepts surrounding cryptographic hash functions. Once the relevant concepts and definitions were in place, a number of generic attacks against cryptographic hash functions were considered in Chapter 3. Based on the definitions in Chapter 1 and the generic attacks presented in Chapter 3, a number of high level functional and security requirements were formulated. Given these requirements, a number cryptographic hash function designs were reviewed, including the MD4-family of functions.

12.2 RESULTS

After introducing the relevant concepts and definitions regarding cryptographic hash functions, a detailed description and re-construction of the attack on MD4 as formulated by Dobbertin is presented in Chapter 6. Using a novel approach, an alternative solution is presented which illustrates that a speed-up factor of 64 of the attack on MD4 as formulated by Dobbertin, can be achieved. In Chapter 7 the attack on MD5 as formulated by Dobbertin is considered. The attack is reconstructed from the source code used by Dobbertin to construct the collisions for MD5. Of particular interest are the techniques used to solve the non-linear Boolean equations. In Chapter 8 the attacks on MD4 and MD5 are generalised. The generalised attack presents a framework for the analyses of all iterated hash functions. In Chapters 9 and 10 the generalised attack is applied to SHA and HAVAL. It is shown that the generalised attack can be applied to reduced versions of SHA and HAVAL. The new results obtained for the HAVAL hash function indicates that three round HAVAL should not be used for cryptographic applications. To the best of our knowledge this is the first cryptanalytical result that has been published regarding the HAVAL cryptographic hash function.

In Chapter 11 we conclude this dissertation by presenting design criteria for dedicated cryptographic hash functions. The design criteria are based on the lessons learned from the analysis of MD4, MD5, SHA, SHA-1 and HAVAL. It is the intention that the application of these design criteria will defeat the generalised attack presented in Chapter 8.

12.3 SUMMARY AND FUTURE WORK

In this dissertation the attacks on MD4 and MD5 were generalised and applied to the SHA and HAVAL hash functions. Design criteria were proposed to defeat this generalised attack. It remains to determine the full extent to which the generalised attack presented in Chapter 8 can be applied to a number of dedicated cryptographic hash functions. It may prove interesting to apply this technique to a number of the Advanced Encryption Standard (AES) candidates to determine the difficulty of obtaining collisions for these cryptographic primitives. Another topic of interest lies in the design of message/key expansion algorithms. The use of strong diffusion structures such as those proposed by Massey in such a design may prove to be a challenging and interesting topic.

BIBLIOGRAPHY

[1] J. L. Massey, "Cryptography: Fundamentals and aplications," 1995. Copies of Transparencies, Advanced Technology Seminars.

[2] W. T. Penzhorn, "Hash functions and authentication," Tech. Rep. WP2, Ciphertec cc, 24 January 1995.

[3] B. Preneel, *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, 1993.

[4] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and Systems Sciences*, vol. 18, pp. 143–154, 1979.

[5] *Secure Electronic Transaction (SET) Specification Book 3: Formal Protocol Definition*, 24 June, Revised August 1 1996.

[6] D. W. Davies and W. L. Price, "The application of digital signatures based on public-key cryptosystems," in *Proc. Fifth Intl. Computer Communications Conference*, pp. 525–530, October 1980.

[7] R. Anderson and E. Biham, "Two practical and provably secure block ciphers: BEAR and LION," in *Fast Software Encryption, Third International Workshop* (D. Gollman, ed.), Lecture Notes in Computer Science No. 1039, Springer-Verlag, pp. 113–120, 1996.

[8] P. Morin, "Provably secure and efficient block ciphers," *Third Annual Workshop on Selected Areas in Cryptography*, pp. 30–37, 1996.

[9] C. H. Lim, "Message encryption and authentication using one-way hash functions," *Third Annual Workshop on Selected Areas in Cryptography*, pp. 38–48, 1996.

[10] R. L. Rivest, "The MD4 message digest algorithm," in *Advances in Cryptology - CRYPTO ' 90*, Lecture Notes in Computer Science vol 537, Springer-Verlag, pp. 303 – 311, 1991.

[11] B. den Boer and A. Bosselaers, "An attack on the last two rounds of MD4," in *Advances in Cryptology - CRYPTO ' 91*, Lecture Notes in Computer Science No. 576, Springer-Verlag, pp. 194–203, 1992.

[12] H. Dobbertin, "Cryptanalysis of MD5 compress," *Rump Session EUROCRYPT ' 96*, 1996.

[13] National Institute of Standards and Technology (NIST), *FIPS Publication 180-1: Secure Hash Standard (SHS)*, April 17, 1995.

[14] H. Dobbertin, "Cryptanalysis of MD4," in *Fast Software Encryption, Third International Workshop* (D. Gollman, ed.), Lecture Notes in Computer Science No. 1039, Springer-Verlag, pp. 53–69, 1996.

[15] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: a strengthened version of ripemd," in *Fast Software Encryption, Third International Workshop* (D. Gollman, ed.), Lecture Notes in Computer Science No. 1039, Springer-Verlag, pp. 71–82, 1996.

[16] R. C. Merkle, "One way hash functions and DES," *Crypto '89*, pp. 428–446, 1989.

[17] P. R. Kasselman, "Analysis of dedicated hash functions," Tech. Rep. Ciph-96-10, Ciphertec cc, November 1996.

[18] G. J. Simmons, "A survey of information authentication," in *Contemporary Cryptology, The Science of Information Integrity* (G. J. Simmons, ed.), pp. 379–419, New York: IEEE Press, 1991.

[19] D. G. Abraham, G. M. Dolan, G. P. Double and J. V. Stevens, "Transaction security system," *IBM Systems Journal*, vol. 30, no. 2, pp. 206–209, 1991.

[20] 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.), Lecture Notes in Computer Science No. 435, Springer-Verlag, pp. 408–415, 1990.

[21] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? Applications to DES," in *Advances in Cryptology - EUROCRYPT ' 89* (J. Quisquater and J. Vandewalle,

eds.), Lecture Notes in Computer Science No. 434, Springer-Verlag, pp. 428–433, 1990.

[22] I. Damgård, "A design principle for hash functions," in *Advances in Cryptology - CRYPTO ' 89* (G. Brassard, ed.), Lecture Notes in Computer Science No. 435, Springer-Verlag, pp. 416–427, 1990.

[23] R. C. Merkle, "One way hash functions and DES," in *Advances in Cryptology - CRYPTO ' 89* (G. Brassard, ed.), Lecture Notes in Computer Science No. 435, Springer-Verlag, pp. 428–446, 1990.

[24] D. Coppersmith, "Another birthday attack," in *Advances in Cryptology - CRYPTO ' 85* (H. C. Williams, ed.), Lecture Notes in Computer Science No. 218, Springer-Verlag, pp. 14–17, 1986.

[25] M. Girault, R. Cohen, and M. Campana, "A generalised birthday attack," in *Advances in Cryptology - EUROCRYPT ' 88* (C. G. Günther, ed.), Lecture Notes in Computer Science No. 330, Springer-Verlag, pp. 129–156, 1988.

[26] L. Knudsen, "Cryptanalysis of LOKI," in *Cryptography and Coding III*, vol. 45, pp. 223–236, The Institute of Mathematics and its Applications Conference Series, Clarendon Press, Oxford, 1993.

[27] M. J. B. Robshaw, "Block ciphers," Tech. Rep. TR 601, RSA Laborities, 2 August 1995.

[28] B. Kaliski and M. Robshaw, "Message authentication with MD5," *CryptoBytes*, vol. 1, pp. 5–8, Spring 1995.

[29] B. Preneel and P. C. van Oorschot, "MDx-MAC and building fast MACs from hash functions," in *Advances in Cryptology - CRYPTO ' 94* (D. Coppersmith, ed.), Lecture Notes in Computer Science No. 963, Springer-Verlag, pp. 1–14, 1995.

[30] D. W. Davies, "A message authenticator algorithm," in *Advances in Cryptology - CRYPTO ' 84* (G. R. Blakley and D. C. Chaum, eds.), Lecture Notes in Computer Science No. 196, Springer-Verlag, pp. 393–400, 1985.

[31] B. Preneel and P. C. van Oorschot, "On the security of two MAC algorithms," in *Advances in Cryptology - EUROCRYPT ' 96* (U. Maurer, ed.), Lecture Notes in Computer Science No. 1070, Springer-Verlag, pp. 19–32, 1996.

[32] R. S. Winternitz, "Producing a one-way hash function from DES," in *Advances in Cryptology - CRYPTO ' 83* (D. Chaum, ed.), (New York), pp. 203–207, Plenum Press, 1984.

[33] M. E. Hellman, R. Merkle, R. Schroeppel, L. Washington, W. Diffie, S. Pohlig, and P. Schweitzer, "Results of an initial attempt to cryptanalyze the NBS Data Encryption Standard," Tech. Rep. SEL 76–042, Stanford University, 1976.

[34] M. Kwan and J. Pieprzyk, "A general purpose technique for locating key scheduling weakness in DES-like cryptosystems," in *Advances in Cryptology–ASIACRYPT '91* (H. Imai, R. Rivest, and T. Matsumoto, eds.), Lecture Notes in Computer Science No. 739, Springer-Verlag, pp. 237–246, 1993.

[35] L. Brown, M. Kwan, J. Pieprzyk, and J. Seberry, "Improving resistance to differential cryptanalysis and the redesign of LOKI," in *Advances in Cryptology–ASIACRYPT '91* (H. Imai, R. Rivest, and T. Matsumoto, eds.), Lecture Notes in Computer Science No. 739, Springer-Verlag, pp. 36–50, 1993.

[36] J. Daemen, R. Govaerts, and J. Vandewalle, "Weak keys for IDEA," in *Advances in Cryptology - CRYPTO ' 93* (D. R. Stinson, ed.), Lecture Notes in Computer Science No. 773, Springer-Verlag, pp. 224–231, 1994.

[37] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking computations (Extended abstract)." Internet, 1996.

[38] A. Shamir and E. Biham, "Research anouncement: A new cryptanalytic attack on DES." Internet: http://jya.com/dfa.htm, 1996.

[39] J.-J. Quisquater, "Short cut for exhaustive key search using fault analysis: Applications to DES, MAC, Keyed hash function, Identification protocols, ..." Internet, 1996.

[40] D. Coppersmith, "Analysis of ISO/CCITT Document X.509 Annex D." Internal Memo, IBM T.J. Watson Center, June 11, 1989.

[41] T. Beth, F. Bausspieß, and F. Damm, "Workshop on cryptographic hash functions," Tech. Rep. 92/11, E.I.S.S., 1992.

[42] G. Brassard, "The impending demise of RSA?," *CryptoBytes*, vol. 1, pp. 1–4, Spring 1995.

[43] F. Bausspieß and F. Damm, "Requirements for cryptographic hash functions," Tech. Rep. 92/2, E.I.S.S., 1992.

[44] R. L. Rivest, "The MD4 message digest algorithm." Internet Request for Comments (RFC), 1990. RFC 1320.

[45] R. L. Rivest, "The MD5 message-digest algorithm." Internet Request for Comments (RFC), April 1992. RFC 1321.

[46] C. E. Shannon, "Communication theory of secrecy systems," in *Claude Elwood Shannon - Collected Papers* (N. J. A. Sloane and A. D. Wyner, eds.), pp. 84–143, IEEE Press, 1983.

[47] R. Anderson and E. Biham, "Tiger: A fast new hash function." Internet, 1996.

[48] J. Daemen, *Cipher and Hash Function Design.* PhD thesis, Katholieke Universiteit Leuven, 1993.

[49] X. Lai and J. L. Massey, "Hash functions based on block ciphers," in *Advances in Cryptology - EUROCRYPT ' 92*, Lecture Notes in Computer Science No. 658, Springer-Verlag, pp. 55–70, 1992.

[50] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* New York: John Wiley & Sons, 1993.

[51] G. J. Kühn, "S-box design and analysis," Tech. Rep. Ciph-96-13, Ciphertec cc, Desember 1996.

[52] R. R. Jueneman, "A high speed manipulation detection code," in *Advances in Cryptology - CRYPTO ' 86* (A. Odlyzko, ed.), Lecture Notes in Computer Science No. 263, Springer-Verlag, pp. 327–346, 1987.

[53] S. Vaudenay, "On the need of multipermuations: Cryptanalysis of MD4 and SAFER," in *Proceedings of the 1994 Leuven Workshop on Cryptographic Algorithms*, Lecture Notes in Computer Science vol 1008, Springer-Verlag, pp. 286–297, 1995.

[54] H. Dobbertin, "RIPEMD with two round compress is not collision-free," *Journal of Cryptology*, vol. 10, no. 1, pp. 51–70, 1997.

[55] H. Dobbertin, "The status of MD5 after a recent attack," *CryptoBytes*, vol. 2, pp. 1–6, Summer 1996.

[56] P. R. Kasselman, "A fast attack on the MD4 hash function," in *Comsig 97* (M. Ings, ed.), no. 97TH8312 in IEEE Catalog, pp. 147–150, South African Section IEEE, 1997.

[57] P. R. Kasselman, "Analysis and design of hash functions: Part II," Tech. Rep. Ciph/97/12/1(II), Ciphertec cc, December 1997.

[58] P. R. Kasselman, "Analysis and design of hash functions: Part I," Tech. Rep. Ciph/97/12/1(I), Ciphertec cc, December 1997.

[59] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. New York: CRC Press, 1997.

[60] A. Meijer, "Galois field counters and linear feedback shift registers," 1997. Class notes.

[61] W. W. Peterson and E. J. Weldon, *Error Correcting Codes*. Massachsetts: MIT Press, 2 ed., 1972.

[62] 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.), (Berlin), pp. 83–104, 1993.

[63] J. F. Wakerly, *Digital Design Principles and Practices*. London: Prentice-Hall International Editions, 1990.

[64] E. K. Grossman and B. Tuckerman, "Analysis of a feistel-like cipher weakened by having no rotating key," Tech. Rep. RC 6375, IBM T.J. Watson Research, Jan 1977.

[65] E. Biham, "New types of cryptanalytic attacks using related keys," in *Advances in Cryptology — EUROCRYPT '93* (T. Helleseth, ed.), Lecture Notes in Computer Science Vol 735, (Berlin), Springer-Verlag, pp. 398–409, 1994.

[66] C. M. Adams, "Simple and effective key scheduling for symmetric ciphers (extended abstract)," *Workshop on Selected Areas in Cryptography*, pp. 129–133, 1994.

[67] V. Rijmen, *Cryptanalysis and Design of Iterated Block Ciphers*. PhD thesis, Katholieke Universiteit Leuven, 1997.

[68] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and Systems Sciences*, vol. 22, pp. 265–279, 1981.

[69] A. R. Meijer, "Universal hash functions in authentication," Tech. Rep. KT437122, Ciphertec cc, 28 February 1996.

[70] S. M. Matyas, C. H. Meyer, and J. Oseas, "Generating strong one-way functions with cryptographic algorithms," *IBM Tech. Disclosure Bull.*, vol. 27, no. 10A, pp. 5658–5659, 1985.

[71] B. Preneel, R. Govaerts, and J. Vandewalle, "Hash functions based on block ciphers: a synthetic approach," in *Advances in Cryptology - CRYPTO ' 93* (D. R. Stinson, ed.), Lecture Notes in Computer Science No. 773, Springer-Verlag, pp. 368–378, 1994.

[72] X. Lai, R. Rueppel, and J. Woollven, "A fast cryptographic checksum algorithm based on stream ciphers," in *Advances in Cryptology — AUSCRYPT '92* (J. Seberry and Y. Zheng, eds.), Lecture Notes in Computer Science No. 718, (Berlin), Springer-Verlag, pp. 339–348, 1993.

[73] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology - CRYPTO ' 96* (N. Koblitz, ed.), Lecture Notes in Computer Science No. 1109, Springer-Verlag, pp. 1–15, 1995.

[74] M. Bellare, R. Canetti, and H. Krawczyk, "The HMAC construction," *CryptoBytes*, vol. 2, pp. 12–15, Spring 1996.

[75] G. Tsudik, "Message authentication with one-way hash functions," *ACM SIGCOMM, Computer Communication Review*, vol. 22, pp. 29–38, Oct. 1992.

[76] J. M. Galvin, K. McCloghrie, and J. R. Davin, "Secure management of snmp networks," *Integrated Network Management II*, pp. 703–714, 1991.

[77] J. Linn, "The Kerberos version 5 GSS-API mechanism." RFC 1964, 1996.

[78] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication." Internet: http://www.research.ibm.com.security/ or http://www-cse.ucsd.edu/users/mihir/papers/papers.html, 1996.

[79] M. Bellare, R. Guérin, and P. Rogaway, "XOR MACs: New methods for message authentication using finite pseudorandom functions," in *Advances in Cryptology - CRYPTO ' 95* (D. Coppersmith, ed.), Lecture Notes in Computer Science No. 963, Springer-Verlag, pp. 15–28, 1995.

[80] M. Bellare, R. Guérin, and P. Rogaway, "XOR MACs: New methods for message authentication using finite pseudorandom functions." Internet: http://www-cse.ucsd.edu/users/mihir/papers/papers.html, 1995.

[81] W. T. Penzhorn, "Study into international standards," Tech. Rep. Ciph-96-12, Ciphertec cc, 6 December 1996.

[82] M. Bellare, J. Kilian, and P. Rogaway, "The security of cipher block chaining," in *Advances in Cryptology - CRYPTO ' 94* (Y. G. Desmedt, ed.), Lecture Notes in Computer Science No. 839, Springer-Verlag, pp. 341–358, 1994.

APPENDIX A: ADDITIONAL HASH FUNCTION CONSTRUCTIONS

A.1 INTRODUCTION

This Appendix describes a number of additional hash function constructions. Specific attention is given to tree constructions, the cascading of hash functions and the use of block and stream ciphers to construct round functions that can be used as part of the Dåmgard-Merkle construction. A number of generic techniques that allows the construction of MACs based on MACs are also considered. A short review of current international standards is also included.

A.2 TREE CONSTRUCTIONS

This scheme is specifically intended for high speed hashing. The first construction along these lines was presented by Carter and Wegman [68]. In [69] this scheme is referred to as concatenation hashing. It was re-discovered independently by Preneel [3] and Damgård [22]. This hash scheme can be generalised and requires the following:

1. A round function $f()$ that takes a fixed input of m bits and reduces it to n bits.
2. Padding rule.
3. A scalable number of parallel processors.

The difference between the construction by Preneel and the construction by Carter and Wegman is found in the round function used. Carter and Wegman propose the use of a universal hash function (a complexity theoretic construction). Preneel specifies that any secure round function, $f()$, could be used. It is advised that the round function chosen for this scheme should adhere to the conditions imposed on round functions used in the Damgård-Merkle scheme (see Section 5.3).

For the tree hashing scheme to work it is required that the message length, r should be a multiple of the block length m. In addition it is required that the number of blocks in the original message should be a multiple of two. These requirements imply a form of padding. The same padding rules as described in Section 5.3 can be applied to this construction.

A.2.1 Construction

A description of the construction of a tree structured hash function is presented below.

For a message of length $r = 2^q$ with $q \in \mathbb{Z}, q > 0$:

$$\begin{aligned} H_i^1 &= f(X_{2i-1}, X_{2i}) \qquad i \in \{1, 2, 3 \ldots 2^{q-1}\} \\ H_i^j &= f(H_{2i-1}^{j-1}, H_{2i}^{j-1}) \qquad i \in \{1, 2, 3 \ldots 2^{q-1}\} \\ & \qquad\qquad\qquad\qquad\quad j \in \{2, 3 \ldots r-1\} \\ h(X) &= f(H_1^{r-1}, H_2^{r-1}). \end{aligned}$$

A graphical representation of this scheme is presented in Figure A.1.

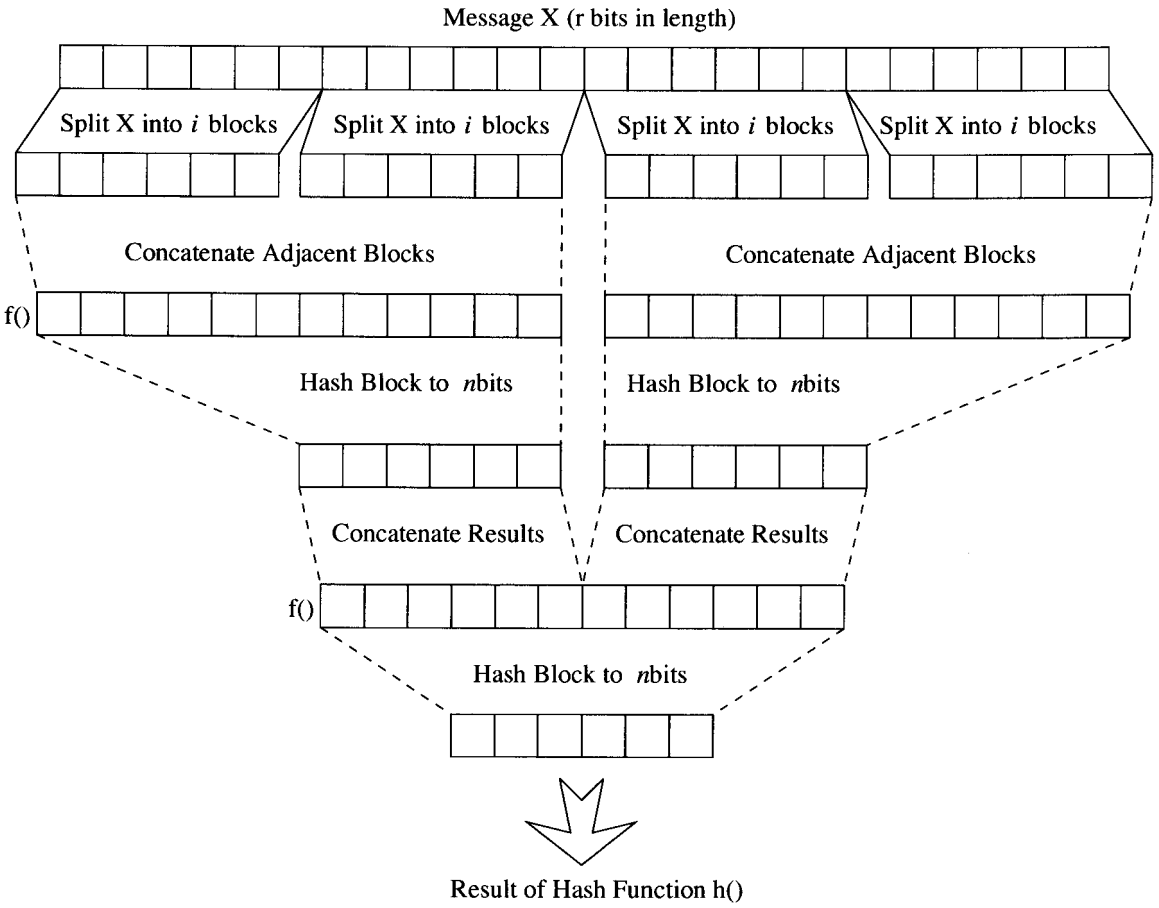

Figure A.1: General Tree Construction for Hash Function

Note that the scheme can be adapted for use as a MAC by making the round function key dependent.

A.2.2 Practicality

This scheme is faster than the Damgård-Merkle scheme. It is stated in [3] that the hash function, $h()$ can be performed for an r bit input with $\frac{k}{2n}$ processors with:

$$O\left(\frac{n}{r-n} \cdot \log_2\left(\frac{k}{n}\right)\right)$$

evaluations of $f()$. A further advantage of this scheme is the avoidance of chaining and consequently all attacks dependent on the chaining.

The tree hashing scheme has the disadvantage that no known analysis has been performed on this structure. Consequently little is known of its security. Another disadvantage of this scheme is the cost involved. This hash function depends on the use of several processors which can operate in parallel. Since it is required that a hash function should be able to hash messages of arbitrary length, an arbitrary number of processors are required. As the message length $r \rightarrow \infty$ the number of processors, $c \rightarrow \infty$. It can be assumed that the cost of implementing such a hash function would escalate accordingly. An implementation of this scheme is therefore impractical due to the costs involved.

This scheme can be made practical by introducing chaining. This solution should be seen as a hybrid between the Damgård-Merkle scheme and the tree construction scheme. The round function $f()$ in the Damgård-Merkle scheme is effectively replaced by the tree hashing scheme. The hybrid scheme has the disadvantages of reducing the performance and re-introducing attacks dependent on the chaining.

Thus a trade-off between speed, cost and security has to be made when using the tree construction. According to [3] a speedup factor of c is achieved if c processors are used. It is possible to use the tree construction not only in the construction of a hash function, $h()$, but also in the construction of a round function $f()$. It is not known if any practical hash functions are based on this scheme.

A.3 CASCADING OF HASH FUNCTIONS

The following observations are made in [3] regarding the cascading of hash functions. Let $A||B$ denote concatenation of message B to A. If $h_1()$ and $h_2()$ are hash functions that

produces hash values of n_1 and n_2 bits respectively, and $g()$ is a one way function that yields a n_3-bit result, then:

1. $h(X_1, X_2) = g(h_1(X_1)||h_1(X_2))$ is a CRHF if $h_1()$ is a CRHF and $g()$ is a CRF.
2. $h(X_1) = g(h_1(X_1)||h_2(X_1))$ is a CRHF if either $h_1()$ or $h_2()$ is a CRHF and $g()$ is a CRF.
3. $h(X_1) = h_1(X_1)||h_2(X_1)$ is a CRHF if either $h_1()$ or $h_2()$ is a CRHF.

The first construction is equivalent to the tree construction discussed in Section A.2 and results in a hash length of n_3. The second construction describes a CRHF that is at least as strong as the strongest of $h_1()$ or $h_2()$, provided $g()$ is a CRF. The resultant hash length for the second construction is n_3. The third construction omits the use of a CRF, consequently the hash length equals $n_1 + n_2 > n_3$.

In terms of hash speed, the first construction is more efficient since two message blocks are hashed at a time. In terms of security the last two constructions are more secure since two hash functions are used. If one of the hash functions is insecure, the entire construction does not become insecure. These constructions can be extended to more than two hash functions.

A.4 ROUND FUNCTION CONSTRUCTIONS

A.4.1 Block Ciphers

Block ciphers are often used in an iterated construction as a round function. The popularity of block ciphers used as round functions are due to the correlation between the requirements set for hash functions and block ciphers. The use of block ciphers has the advantage that the cost and effort of designing and analysing a new round function is drastically reduced, provided that a trusted block cipher is used. An additional advantage when using a block cipher as a round function is that block ciphers are designed to accommodate a secret key. This is an advantage when constructing a MAC. Three disadvantages should be noted when using a block cipher as a round function. The first disadvantage deals with the functional requirement of speed. Hash functions that contain block ciphers as building blocks are slower than dedicated hash functions. The second disadvantage is the introduction of additional

attacks, based on certain properties of blocks ciphers (see Chapter 3 Section 3.6). A third disadvantage is that a number of hash functions have a block size of 64 bits. It is generally believed that the hash length should equal 128 bits or more, in order to provide protection against birthday attacks (see Chapter 3 Section 3.4).

A distinction is made between round functions for which the length of the chaining variable is equal to the block length, and round functions for which the length of the chaining variable is equal to twice the block length of the block cipher.

Before proceeding consider the following notation. Let $E(K, X_i)$ denote the encryption of message block X_i with key K.

Hash Length Equal to Block Length

Figure A.2 depicts a generic configuration for a block cipher used as a round function in a hash function.

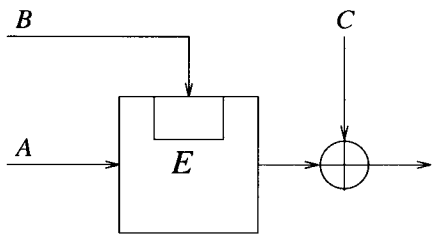

Figure A.2: Generic Round Function from a Block Cipher

Note that three inputs, A, B, and C are available. Each input can take one of four possible inputs, X_i, H_{i-1}, $X_i \oplus H_{i-1}$ or a constant. Thus there are $4^3 = 64$ possible configurations of the construction presented in Figure A.2. A general expression for the above construction is given by:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(A, B) \oplus C. \end{aligned}$$

These structures were analysed in [3]. The result of this analysis is that only four of these constructions are considered secure against all attacks. They are defined by the following

expressions [3], [50]:

$$\begin{aligned} H_i &= E(u(H_{i-1}), X_i) \oplus X_i \\ H_i &= E(u(H_{i-1}), (X_i \oplus H_{i-1})) \oplus X_i \oplus H_{i-1} \\ H_i &= E(u(H_{i-1}), X_i) \oplus H_{i-1} \oplus X_i \\ H_i &= E(u(H_{i-1}), (X_i \oplus H_{i-1})) \oplus X_i \end{aligned}$$

The function $u()$ is a mapping from the ciphertext space to the key space. A visual representation of these constructions are presented in Figure A.3.

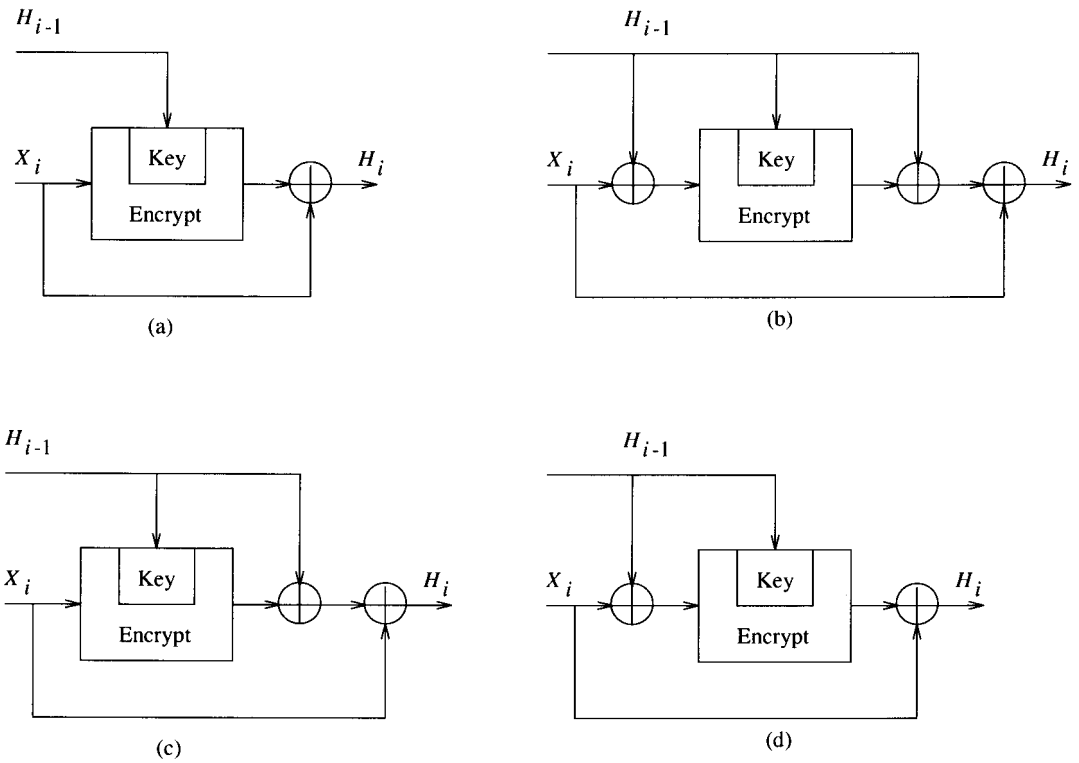

Figure A.3: The Four Secure Round Function Constructions Based on Block Ciphers

Figure A.3(a) is known as the Matyas hash scheme [70]. Figure A.3(c) is known as the Preneel-Miyaguchi hash scheme. The dual of the scheme in Figure A.3(a) is known as the Davies-Meyer scheme and is depicted in Figure A.4.

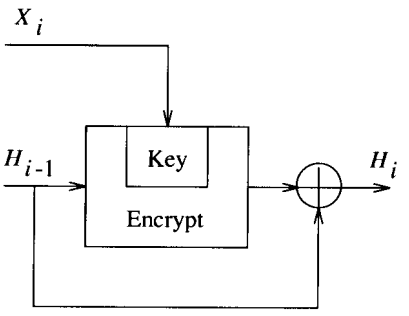

Figure A.4: Davies-Meyer Hash Scheme

The analytical expression for the Davies-Meyer scheme is given by:

$$H_i = E(h(X_i), H_{i-1}) \oplus H_{i-1}.$$

These schemes have been analysed in [3] with regard to:

- Direct attacks.
- Permutation attacks.
- Backward attacks.
- Fixed point attacks.

Hash Length Equal to Twice the Block Length

Two constructions based on a n bit cipher resulting in a $2 \cdot n$ bit hash result are proposed in [49]. These constructions are based upon the availability of a block cipher with a n-bit block size and a $2 \cdot n$ bit key. One such block cipher is IDEA. These constructions are considered variants of the Davies-Meyer scheme mentioned earlier. The first is denoted the tandem Davies-Meyer, and is shown in Figure A.5(a).

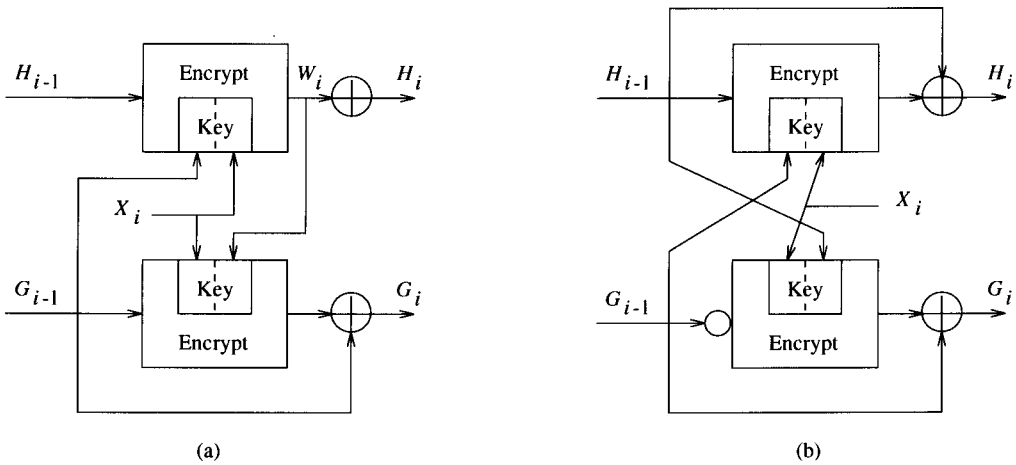

Figure A.5: (a) Tandem Davies-Meyer Scheme (b) Abreast Davies-Meyer Scheme

It is described analytically as follows:

$$\begin{aligned} G_0 &= IV_1 \\ H_0 &= IV_2 \\ W_i &= E(G_{i-1}||X_i, H_{i-1}) \\ G_i &= G_{i-1} \oplus E(X_i||W_{i-1}, G_{i-1}) \\ H_i &= W_i \oplus H_{i-1}. \end{aligned}$$

The Davies-Meyer abreast scheme is also defined in [49] and shown in Figure A.5(b). Analytically the construction is expressed as:

$$\begin{aligned} G_0 &= IV_1 \\ H_0 &= IV_2 \\ G_i &= G_{i-1} \oplus E(X_i||H_{i-1}, (G_{i-1})) \\ H_i &= H_{i-1} \oplus E(G_{i-1}||X_i, (H_{i-1})) \end{aligned}$$

Two additional schemes, which employ block ciphers to construct round functions with a hash length equal to twice the block length, are MDC2 and MDC4. MDC2 is defined as

follows:

$$\begin{aligned} H_0 &= IV_1 \\ H_1 &= IV_2 \\ T1_i &= E(H_{i-1}, X_i) \\ &\quad LT1_i||RT1_i \\ T2_i &= E(G_{i-1}, X_i) \\ &\quad LT2_i||RT2_i \\ H_i &= LT1_i||RT2_i \\ G_i &= LT2_i||RT1_i \end{aligned}$$

with:

$$\begin{aligned} LTx_i &= \text{Lefthand } \frac{n}{2} \text{ bits of } n \text{ bit block.} \\ RTx_i &= \text{Righthand } \frac{n}{2} \text{ bits of } n \text{ bit block.} \\ x &= 1 \text{ or } 2. \end{aligned}$$

MDC4 is defined as the application of two consecutive rounds of MDC2. Thus, the result is that MDC2 is approximately twice as fast as MDC4. It is believed that MDC4 is more secure than MDC2.

Block diagrammatic forms of MDC2 and MDC4 are presented in Figure A.6(a) and Figure A.6(b) respectively.

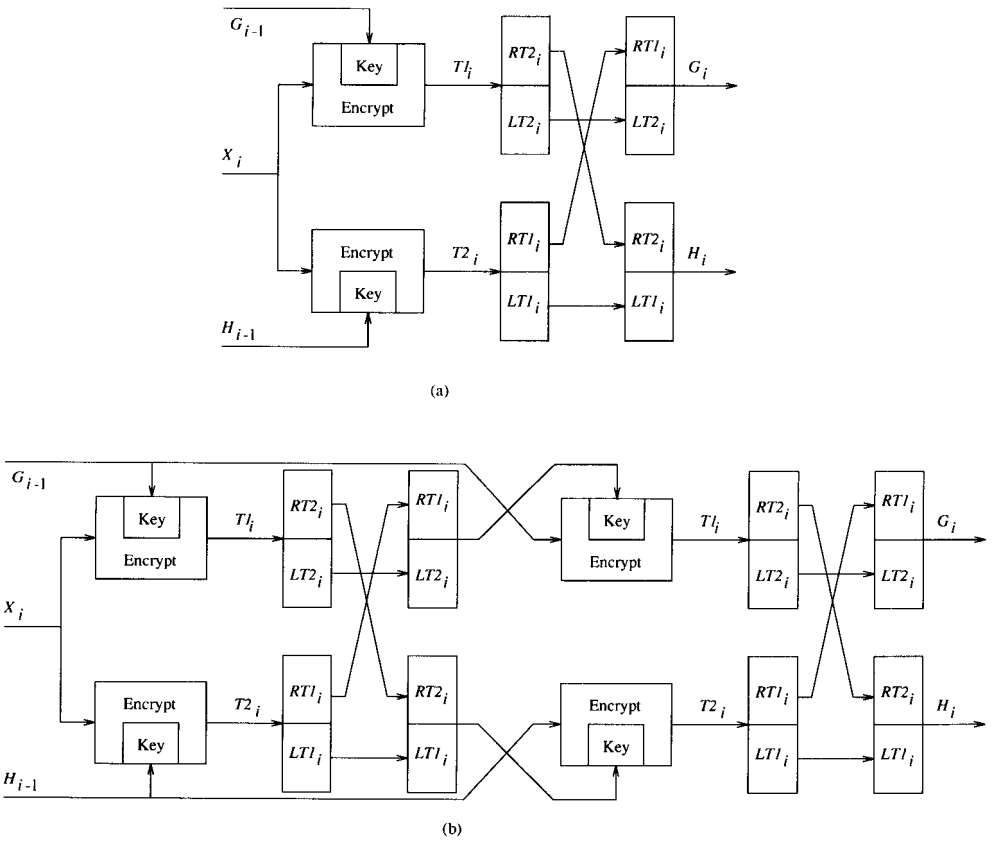

Figure A.6: (a) MDC2 (b) MDC4

Little research has been performed with regard to the security of MDC2 and MDC4.

MACs Based on Block Ciphers

As shown in Section 5.3 MACs can be based on iterative schemes, providing that the resulting hash value is key dependent. Block ciphers are suitable for constructing MAC round functions since block ciphers are designed to accommodate secret keys. Two constructions based on blocks ciphers are available.

1. Cipher block chaining (CBC).
2. Cipher feedback chaining (CFB).

The block cipher used in CBC mode for a MAC round function construction is described as follows:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(K, X_i \oplus H_{i-1}) \qquad \in \{1, 2, 3, \ldots, j\} \\ H(X) &= H_j. \end{aligned}$$

A graphical representation of this construction for a MAC round function is shown in Figure A.7(a). The block cipher construction used in CFB mode is described as follows:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(K, H_{i-1}) \oplus X_i \qquad \in \{1, 2, 3, \ldots, j\} \\ H(X) &= H_j. \end{aligned}$$

Refer to Figure A.7(b) for a graphical representation of this construction. Note that in the case of CFB, the final result has to be encrypted once again in order to remove the linear dependence of the MAC on the last plaintext block. A third construction was proposed in [71]. It is represented in Figure A.7(c) and is described below:

$$\begin{aligned} H_0 &= IV \\ H_i &= E(K, X_i \oplus H_{i-1}) \oplus X_i \qquad \in \{1, 2, 3, \ldots, j\} \\ H(X) &= H_j. \end{aligned}$$

It is believed that this construction is harder to invert [2] than the previously mentioned constructions. In [3] it is advised that, should encryption of the MAC be required, a different key should be used for encryption purposes.

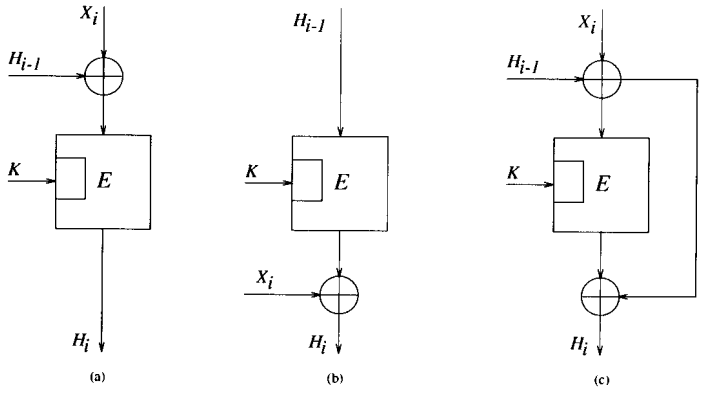

Figure A.7: Block Cipher Based MAC Round Functions

Suitable Block Ciphers

A number of block ciphers have been proposed. However, not all block ciphers are suitable for use as round functions in cryptographic hash functions. It should first be noted that the security of the hash function constructed from a block cipher is based on the assumption that the underlying block cipher is secure. Thus block ciphers which are considered insecure should be avoided. In addition certain properties of block ciphers allow the resulting hash function to be susceptible to specific attacks. These properties include:

1. Key collisions.
2. Complementation property.
3. Weak keys.

The manner in which these properties are exploited is considered in Chapter 3, Section 3.5 and 3.6.

A.4.2 Stream Ciphers

It is conceivable that the round function of a hash function can be based on a stream cipher. This type of construction is hinted at in [69]. It may be possible to adapt the construction presented in [72] to construct a round function for an iterated hash function. Little is known of the security of hash function based on stream ciphers.

A.5 MAC CONSTRUCTIONS BASED ON MDCS

Traditionally MACs are based on block ciphers (see Section A.4.1). Recently various techniques for constructing hash functions from MDCs were proposed [29], [73], [74]. The preference for the use of MDC based MACs over block cipher based MACs is based on the following factors:

1. Speed of execution.
2. Export restrictions.

Speed of execution is an important functional requirement (see Chapter 4 Section 4.2.4). The matter of export restrictions is a political one. Several countries, most notably the USA, restrict the export of certain cryptographic functions. A large number of block ciphers are covered by these restrictions. Thus, MACs based on block ciphers may not be exported to other countries. It has been proposed that MACs are used in electronic transactions on the Internet [5]. The Internet spans across the globe, and participants from different countries may wish to engage in electronic banking transactions. Thus, MACs based on block ciphers cannot be used for secure Internet Transactions, due to export restrictions. For this reason MACs based on MDCs are preferred over MACs based on block ciphers, since MDCs are not restricted by export controls. Because of these reasons MDC based MACs were adopted in Kerberos, IPSec and SET [5] [29].

It should be remembered that MDCs were not designed to accommodate a key. Thus, when constructing MACs from MDCs, care should be taken in the manner in which the key for the MAC is introduced. The key should be introduced in such a manner that the resulting hash value does not reveal any information of the secret key. This requirement is based on the principles of confusion and diffusion as introduced by Shannon [46].

A number of MAC constructions based on MDCs were proposed. These include:

1. Affix construction.
2. IPSec recommendations
3. NMAC construction.
4. HMAC construction.
5. MDx-MAC construction.
6. XOR-MAC construction.

A description of each of these constructions are presented in this section.

A.5.1 Affix Construction

Three affix constructions are identified. They are:

1. Secret prefix method.
2. Secret suffix method.
3. Envelope method.

These constructions and the security offered by these constructions are considered next.

Secret Prefix Method

The secret prefix construction was proposed independently in [75] and [76]. This construction requires that the secret key, K_1, be prepended to the message X. Thus the prefix method can be described as

$$\text{MAC}(X) = h(K_1||X)$$

with:

$$\begin{aligned} h() &= \text{Iterated MDC} \\ || &= \text{Concatenate.} \end{aligned}$$

A graphical representation of this construction is shown in Figure A.8(a). This construction is considered insecure due to the message extension or padding attacks [75] [29]. A variant of the prefix construction with MD5 is used in Kerberos V. This construction is denoted as MD2.5 [77]. Concern is expressed over the security of this construction in [29].

Secret Suffix Method

This construction is described in [75]. The secret suffix construction appends the secret key K_2 to the message X before hashing. The construction is described as follows:

$$\text{MAC}(X) = h(X||K_2)$$

with:

$$\begin{aligned} h() &= \text{Iterated MDC} \\ || &= \text{Concatenate.} \end{aligned}$$

A graphical representation of this construction is shown in Figure A.8(b). A number of attacks on this construction are described in [29]. If off-line attacks are allowed, an internal collision can be found in approximately $O(2^{\frac{n}{2}})$ off-line operations. A second attack is possible if a second pre-image attack on the underlying MDC is possible. A third attack is considered possible if t text-MAC pairs are known. The number of known text-MAC pairs reduces the computational effort to construct a second pre-image from $O(2^n)$ to $O(2^{\frac{n}{t}})$.

Envelope Method

The envelope construction is described in [75]. This construction prepends the secret key, K_1, and appends the secret key, K_2, to the message, X, before hashing (see Figure A.8(b)). The construction is described as follows:

$$\text{MAC}(X) = h(K_1||X||K_2)$$

with:

$$\begin{aligned} h() &= \text{Iterated MDC} \\ || &= \text{Concatenate.} \end{aligned}$$

In [75] it is claimed that the effective key length for the envelope construction is equal to the length of K_1 (k_1 bits) added to the length of K_2 (k_2 bits). Thus according to [75] $O(2^{k_1+k_2})$ operations are required to establish a collision. This is shown to be incorrect in [29]. In [29] it is demonstrated that the effective key length is less than or equal $k_i + 1$, for whichever is the larger of k_1 or k_2. This implies that the number of operations required to establish a collision are less than or equal to $O(2^{k_i+1})$ for whichever is the larger, k_1 or k_2. Thus the security gained by selecting $K_1 \neq K_2$ is less than expected. A divide and conquer attack on the envelope method is described that establishes an internal collision and then searches exhaustively for the two respective keys.

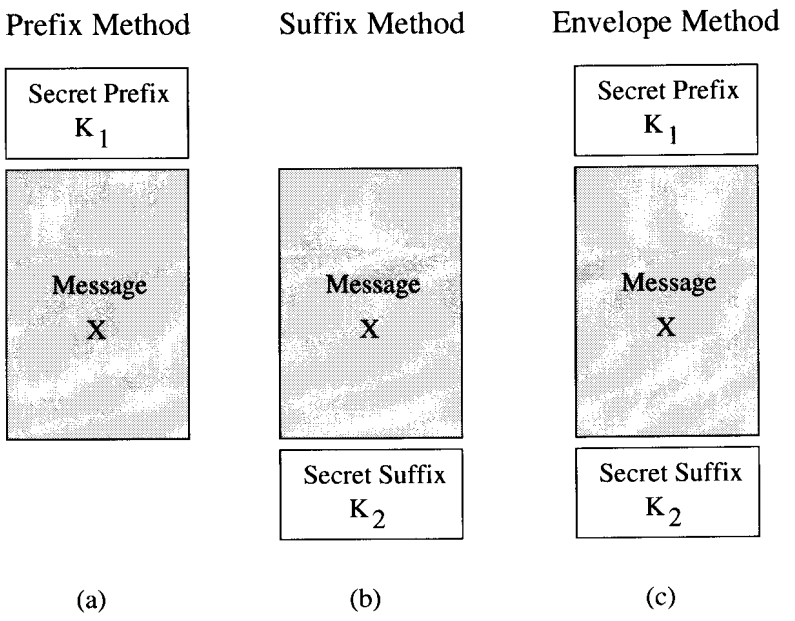

Figure A.8: Affix Constructions

The attacks on the affix constructions are easier than attacks on an ideal MAC. However, provided that the hash function, $h()$, is collision resistant, the attacks on the affix constructions remain computationally infeasible. Key recovery attacks on these constructions are presented in [31].

A.5.2 IPSec recommendations

In [28] three MAC constructions based on MD5 are presented. These proposals were submitted to the IPSec working group. The constructions could be viewed as variations of the affix methods described in Section A.5.1. The three proposals are summarised below:

1.

$$\text{MAC}(X) = \text{MD5}(K_1||\text{MD5}(K_2||X))$$

with:

$$\begin{aligned} \text{MD5}() &= \text{MD5 hash function} \\ || &= \text{Concatenate} \\ K_1 &= \text{128 bit key} \\ K_2 &= \text{128 bit key} \\ X &= \text{Message.} \end{aligned}$$

2.

$$\text{MAC}(X) = \text{MD5}(K_1||p||X||K_1)$$

with:

$$\begin{aligned} \text{MD5}() &= \text{MD5 hash function} \\ || &= \text{Concatenate} \\ K_1 &= \text{128 bit key} \\ p &= \text{384 padding bits} \\ X &= \text{Message.} \end{aligned}$$

3.

$$\text{MAC}(X) = \text{MD5}(K_1||\text{MD5}(K_1||X))$$

with:

$$\begin{aligned} \text{MD5}() &= \text{MD5 hash function} \\ || &= \text{Concatenate} \\ K_1 &= \text{128 bit key} \\ X &= \text{Message.} \end{aligned}$$

The second proposal is effectively the envelope method with K_1 padded to form a 512 bit block. In addition it differs from the envelope method since $K_1 \neq K_2$. Thus, K_1 specifies an initial value for the MD5 hash algorithm applied to the message block. Reservation has been expressed on the use of the initial value as a secret key [3]. The key is however also appended to the message for hashing. This should increase the security of the resultant

MAC. This technique is considered susceptible to a divide and conquer attack as described in Section A.5.1.

In [28] it is stated that the chosen message attack requires 2^{64} chosen texts. In [29] it shows that this can be reduced to $2^{56.5}$ known text-MAC pairs if it is assumed that the number of trailing blocks, s, are 2^{16}.

A.5.3 NMAC Construction

NMAC is an acronym for nested message authentication code. It is defined in [73] and [78]. It is a generic construction of the following form:

$$\text{NMAC}(X) = h_{K_2}(h_{K_2}(X))$$

with:

$$\begin{aligned} h_{K_i}() &= \text{Keyed hash function or a MAC} \\ K_1 &= \text{Key 1} \\ K_2 &= \text{Key 2} \\ X &= \text{Message.} \end{aligned}$$

The NMAC construction does not propose a technique for constructing a keyed hash function. The security of this construction is based on the conditions imposed on the compress function of the keyed hash function and the iterated hash function itself.

A.5.4 HMAC Construction

The HMAC construction is a variant of the NMAC construction for which the IV is fixed. This construction requires no changes to the MDC used for constructing a MAC. The construction involves a single key, K, of length k bits. The use of a single key is advantageous with regard to key management and its associated problems. The HMAC construction is defined in [73], [74], [78] as:

$$\text{HMAC}(X) = h(\overline{K} \oplus \texttt{opad} || h(\overline{K} \oplus \texttt{ipad} || X))$$

with:

$h()$ = Hash function

$\overline{K}$ = The key, K, padded with $0's$ to form an elementary block

`opad` = The byte `0X36` repeated to form a elementary block

`ipad` = The byte `0X5C` repeated to form a elementary block

$\oplus$ = Bitwise XOR

$||$ = Concatenation

X = Message.

This procedure is summarised in Figure A.9

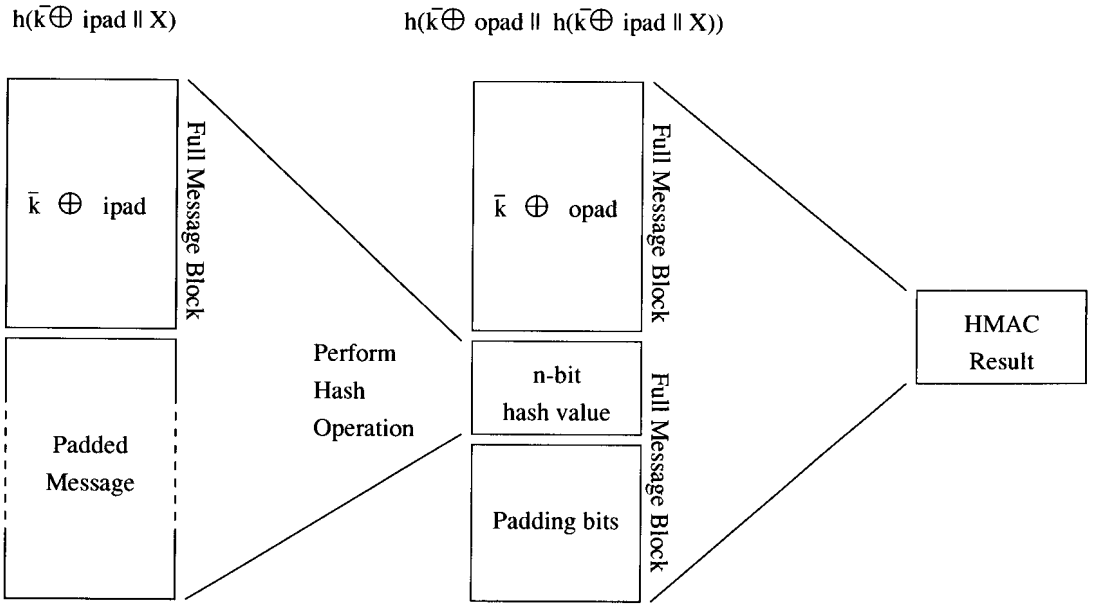

Figure A.9: HMAC Construction

The security of the HMAC construction is based on the security of NMAC. The relation between these two constructions are found in the construction of the two derived keys K_1 =`opad` $\oplus\overline{K}$ and K_2 =`ipad` $\oplus\overline{K}$. Thus HMAC is a specific instance of NMAC. It is stated in [73] that attacks against HMAC may exist, but that these attacks are not necessarily applicable to NMAC. Note that effectively using the same key, K, in both applications of $h()$ does not weaken the construction significantly due to the existence of the divide and conquer attack mentioned in Section A.5.1.

The HMAC construction has become the mandatory construction for use in authentication transforms for Internet security protocols. The HMAC construction is also specified in the SET specification [5]. At present all of the known generic attacks against HMAC are considered infeasible [73], [74], [78].

A.5.5 MDx-MAC Construction

This construction is suggested in [29]. The following design goals were set for this construction:

1. The secret key should be involved at the beginning, at the end, and in every iteration of the hash function.
2. The deviation of the original hash function should be minimal.
3. The performance should be close to that of the hash function.
4. Additional memory requirements should be kept minimal.
5. The approach should be generic, i.e. should apply to any hash function based on the same principles as MD4.

This construction can be used with MD5, RIPEMD or SHA. MD4 is omitted due to the attack described in [17]. In this section MDx refers to one of the three hash functions mentioned above. Let $\overline{MDx}$ refer to an implementation of MDx with both padding and appending length omitted. The resulting construction utilises three 128 bit (16 byte) constants, T_0, T_1 and T_2. These constants are used to construct three additional 768 bit constants U_0, U_1 and U_2. If the key is shorter than 128 bits, the key is expanded to be of a 128 bits length. Once this is accomplished, three sub-keys, K_0, K_1 and K_2, are derived as follows:

$$K_i = \overline{MDx}(K||U_i||K) \qquad i \in \{0, 1, 2\}$$

The constants U_i are required to ensure that the hash is computed over two iterations of the hash function, thus increasing the difficulty of retrieving K from any of the K_i, even if two of the sub-keys are known (see Figure A.10). The mapping from K to K_i is not bijective, but the reduction in entropy is believed to negligible [29].

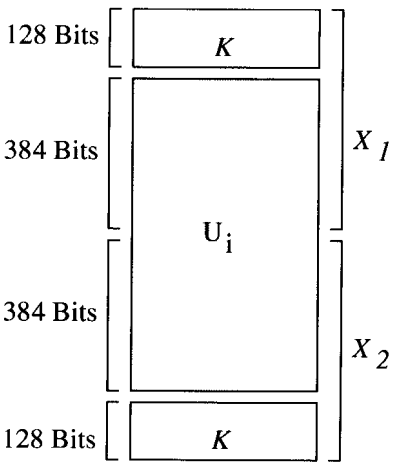

Figure A.10: MDx Key Expansion

Once this step is completed the leftmost 128 bits of the sub-key K_1 is split into four 32-bit blocks denoted as $K_1[i]$, with $0 \leq i \leq 3$ (see Figure A.11).

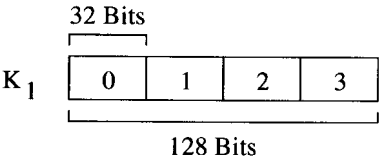

Figure A.11: MDx K_1 Sub-key Partitioning

The resulting MAC is now calculated as follows:

1. The initial value, IV, of MDx is replaced by K_0.
2. $K_1[i \bmod 4]$ is added modulo 2^{32} to the constants used in round i of each iteration of MDx.
3. Following the last block after normal processing of MDx (i.e. including the padding and addition of message length), append an additional 512 bit block. The additional block is derived from the constants T_0, T_1, T_2 and the sub-key K_2 as shown below:

$$K_2||K_2 \oplus T_0||K_2 \oplus T_1||K_2 \oplus T_2.$$

4. The MAC result is the leftmost m bits of the resulting hash value. It is advised that $m = \frac{n}{2}$.

A summary of the above procedure is shown in Figure A.12.

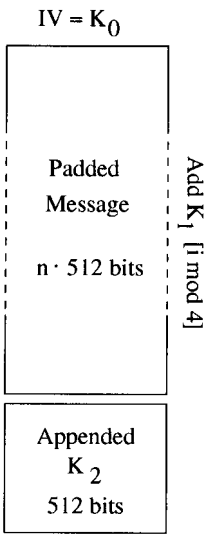

Figure A.12: MDx MAC construction

Let s represent the number of common trailing blocks in two messages. In [29] it is stated that if the MAC length $m = \frac{n}{2}$, a forgery attack requires $O(\frac{2^m}{s+1})$ chosen text-MAC pairs and $O(\frac{2^m}{\sqrt{s+1}})$ known texts. Thus MDx-MAC is more secure than the envelope method described in Section A.5.1. It is also stated in [29] that the divide and conquer attack described in Section A.5.1 is not applicable to MDx-MAC.

A.5.6 XOR-MAC Constructions

This construction is described in [79] and [80] and resembles the multiple message hash scheme described in [68] and [69]. This scheme does not show how to construct a MAC from a MDC, but does make use of keyed MDCs in the construction of the MAC. A generic description of this scheme is presented below:

$$\text{XOR-MAC}(X) = F_k(X[1]) \oplus F_k(X[2]) \oplus F_k(X[3]) \oplus \ldots \oplus F_k(X[m])$$

with:

$$\begin{aligned} F_k() &= \text{Keyed pseudorandom function} \\ \oplus &= \text{Bitwise XOR} \\ X &= \text{Message} \\ m &= \text{Number of elementary blocks in message } X \\ X[i] &= \text{Elementary message block i} \quad i \in \{1, 2, 3, \ldots m\}. \end{aligned}$$

Thus the message is divided into elementary block lengths and then processed by a pseudorandom function. The pseudorandom function, PRF, should be keyed. It is suggested in [79] and [80] that the PRF could be either a block cipher, or a keyed hash function. The result of the PRF for each message block is then XOR'ed with the previous result. Once the last XOR is performed, the MAC is calculated. Figure A.13 presents a visual interpretation of this scheme.

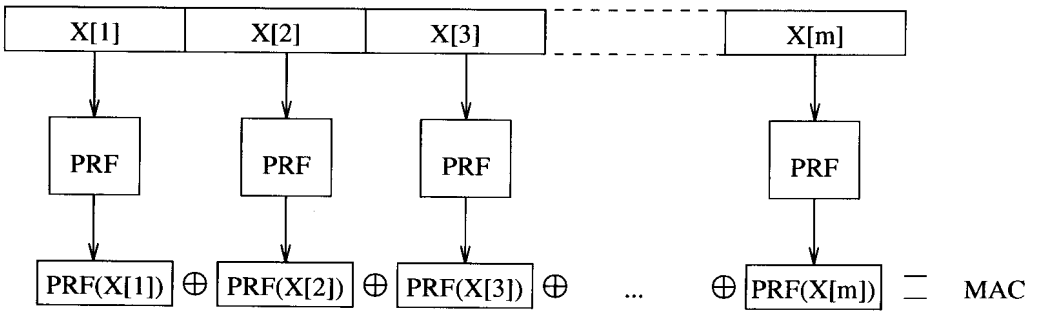

Figure A.13: XOR-MAC construction

Concrete proposals for schemes making use of this construction are presented in [79] and [80], followed by analysis of the security of these proposals. This scheme is highly parallelliseable and has the additional advantages of out of order verification. If only a single block in a message is changed, the output can be updated without recomputing the entire MAC.

A.6 INTERNATIONAL STANDARDS

A number of hash function constructions are under consideration for standardisation. A summary of these can be found in chapter 3 and chapter 5 of [81]. The multipart standard, ISO/IEC 10118, contains the following proposals.

ISO/IEC 10118-1: This part of the standard provides general definitions and background to the remainder of the standard. The iterative hash function construction as defined in Section 5.3 is contained in this part of the proposed standard.

ISO/IEC 10118-2: This part of the standard specifies hash functions constructed from block ciphers. Two methods are specified. The first method is the general construction specified in Section A.4.1 with the block length equal to the hash length (see Figure A.2). The second is equal to MDC2 (see Figure A.6(a)).

ISO/IEC 10118-3: This part of the standard describes the following three dedicated hash functions, SHS (secure hash standard), RIPEMD-128 and RIPEMD-160.

ISO/IEC 10118-4: This part of the standard specifies two hash algorithms based on modular arithmetic, namely MASH-1 and MASH-2.

In addition to the above standard, ISO/IEC 9797 specifies a method for using a key and a n-bit block cipher to construct a MAC. The process is summarised as follows:

1. Pad data to form a n-bit block.
2. Encipher data in CBC mode.
3. The final ciphertext block is the resulting MAC.

The construction specified in ISO/IEC 9797 corresponds to the construction shown in Figure A.7(a).

A.7 CONCLUSION

When selecting a hash function construction the construction should be evaluated according to the requirements set in Chapter 4. A trade-off between cost, security and speed has to be made. The generic attacks described in Chapter 3 should be infeasible.

A final matter of interest is the matter of injectivity, surjectivity and bijectivity of the round function used in an iterated scheme. In [30] it is stated that an injective function should never be used in an iterated hash function. In [82] the question is raised whether a bijective round

function allows stronger security claims. This question is answered in part in [29], where it is shown that the known and chosen text attack is not applicable to MACs with bijective round functions. The influence of the bijectivity of the round functions of MDCs and MACs on the security of the entire construction remains unresolved.

APPENDIX B: SOURCE CODE: IMPLEMENTATION OF MD4

This appendix contains an implementation of the MD4 hash algorithm as described in [10] and [44].

```
/* This header file includes the functions used to implement the MD4 algorithm
 * as described in Crypto 91 by R.Rivest
 *
 * Author: P.R. Kasselman
 * Date: August 20, 1996
 * Filename: md4.h
 * Copyright: Ciphertec cc */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define A0 0x67452301
#define B0 0xefcdab89
#define C0 0x98badcfe
#define D0 0x10325476
#define ROOT2 0x5a827999
#define ROOT3 0x6ed9eba1

#define FS1 3
#define FS2 7
#define FS3 11
#define FS4 19

#define GS1 3
#define GS2 5
#define GS3 9
#define GS4 13

#define HS1 3
#define HS2 9
#define HS3 11
#define HS4 15

int PadBit(int argc, char filename[], unsigned int *PadLen);
void Init(unsigned int *A, unsigned int *B, unsigned int *C, unsigned int *D);
void SaveParms(unsigned int A, unsigned int B, unsigned int C,
       unsigned int D, unsigned int *AA, unsigned int *BB,
       unsigned int *CC, unsigned int *DD);
void ReadArray(int argc, char filename[], unsigned int M[], int n);
unsigned int Rotate(unsigned int X, unsigned int s);
unsigned int FunctionF(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int FunctionG(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int FunctionH(unsigned int X, unsigned int Y, unsigned int Z);
void Round1(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[]);
void Round2(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[]);
```

```
void Round3(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[]);
void Update(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int AA, unsigned int BB, unsigned int CC,
    unsigned int DD);
void PrintSignature(unsigned int A, unsigned int B, unsigned int C,
    unsigned int D);
void PrintReverse(unsigned int X);
void RestoreFile(char filename[], unsigned int FileLen);

int md4(int argc, char filename[]);

unsigned int RotateRight(unsigned int X, unsigned int s);

/* This routine performs the bit padding as requied for the MD4 algorithm */
int PadBit(int argc, char filename[], unsigned int *PadLen)
{

   unsigned int i,j;
   unsigned int FileLen, FileBits, PadBits, PadBytes, TempInt;
   unsigned char temp;
   FILE *fp;

   FileLen = 0;

   if(argc != 2)
     {
printf("No file specified\n");
exit(1);
     }

   fp = fopen(filename, "r+b");

   if(!fp)
     {
printf("Error opening file\n");
fclose(fp);
exit(1);
     }

   /* Determine the size of the file */

   while(!feof(fp))
     {
fread(&temp, sizeof(unsigned char), 1, fp);
if(!feof(fp))
       {
  FileLen++;
       }
     }

   FileBits = FileLen*8*sizeof(unsigned char);
```

```
    /* Compute the number of bits needed for padding */

    PadBits = abs( (448 - FileBits) % 512 );
    PadBytes = PadBits/8;

    /* Pad bit "1" */
    temp = 0x80;
    fwrite(&temp, sizeof(unsigned char), 1, fp);

    /* Pad zero bits */
    temp = 0;
    for(i=0; i<PadBytes-1; i++)
      {
fwrite(&temp, sizeof(unsigned char), 1, fp);
      }

    /* Append the size of the file
     * (For this implimentation no file larger than 2^32 is expected)*/

    TempInt = FileBits;
    fwrite(&TempInt, sizeof(unsigned int), 1, fp);
    TempInt = 0x00;
    fwrite(&TempInt, sizeof(unsigned int), 1, fp);

    *PadLen = (int)(ceil((double)(FileLen+1) / 64)*64);

    fclose(fp);

    return(FileLen);
}

/* Initialise the Buffers to be processed */

void Init(unsigned int *A, unsigned int *B, unsigned int *C, unsigned int *D)
{
    *A = A0;
    *B = B0;
    *C = C0;
    *D = D0;
}

/* This function updates the holding variables AA, BB, CC, DD */

void SaveParms(unsigned int A, unsigned int B, unsigned int C,
        unsigned int D, unsigned int *AA, unsigned int *BB,
        unsigned int *CC, unsigned int *DD)
{
    *AA = A;
    *BB = B;
    *CC = C;
    *DD = D;
}

/* This function reads the modified data file and updates the M array */
```

```
void ReadArray(int argc, char filename[], unsigned int M[], int n)
{
   unsigned int i;
   unsigned int TempInt;
   FILE *fp;

   if(argc != 2)
     {
printf("No file specified\n");
exit(1);
     }

   fp = fopen(filename, "r+b");

   if(!fp)
     {
printf("Error opening file\n");
fclose(fp);
exit(1);
     }

   /* Read the file */

   for(i=0; i<n; i++)
     {
fread(&TempInt, sizeof(unsigned int), 1, fp);
M[i] = TempInt;
     }

   fclose(fp);
}

void Round1(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[])
{
   *A = Rotate((*A + FunctionF(*B,*C,*D) + X[0]), 3);
   *D = Rotate((*D + FunctionF(*A,*B,*C) + X[1]), 7);
   *C = Rotate((*C + FunctionF(*D,*A,*B) + X[2]), 11);
   *B = Rotate((*B + FunctionF(*C,*D,*A) + X[3]), 19);

   *A = Rotate((*A + FunctionF(*B,*C,*D) + X[4]), 3);
   *D = Rotate((*D + FunctionF(*A,*B,*C) + X[5]), 7);
   *C = Rotate((*C + FunctionF(*D,*A,*B) + X[6]), 11);
   *B = Rotate((*B + FunctionF(*C,*D,*A) + X[7]), 19);

   *A = Rotate((*A + FunctionF(*B,*C,*D) + X[8]), 3);
   *D = Rotate((*D + FunctionF(*A,*B,*C) + X[9]), 7);
   *C = Rotate((*C + FunctionF(*D,*A,*B) + X[10]), 11);
   *B = Rotate((*B + FunctionF(*C,*D,*A) + X[11]), 19);

   *A = Rotate((*A + FunctionF(*B,*C,*D) + X[12]), 3);
   *D = Rotate((*D + FunctionF(*A,*B,*C) + X[13]), 7);
   *C = Rotate((*C + FunctionF(*D,*A,*B) + X[14]), 11);
   *B = Rotate((*B + FunctionF(*C,*D,*A) + X[15]), 19);
```

```
}

void Round2(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[])
{
   *A = Rotate((*A + FunctionG(*B,*C,*D) + X[0] + ROOT2), 3);
   *D = Rotate((*D + FunctionG(*A,*B,*C) + X[4] + ROOT2), 5);
   *C = Rotate((*C + FunctionG(*D,*A,*B) + X[8] + ROOT2), 9);
   *B = Rotate((*B + FunctionG(*C,*D,*A) + X[12] + ROOT2), 13);

   *A = Rotate((*A + FunctionG(*B,*C,*D) + X[1] + ROOT2), 3);
   *D = Rotate((*D + FunctionG(*A,*B,*C) + X[5] + ROOT2), 5);
   *C = Rotate((*C + FunctionG(*D,*A,*B) + X[9] + ROOT2), 9);
   *B = Rotate((*B + FunctionG(*C,*D,*A) + X[13] + ROOT2), 13);

   *A = Rotate((*A + FunctionG(*B,*C,*D) + X[2] + ROOT2), 3);
   *D = Rotate((*D + FunctionG(*A,*B,*C) + X[6] + ROOT2), 5);
   *C = Rotate((*C + FunctionG(*D,*A,*B) + X[10] + ROOT2), 9);
   *B = Rotate((*B + FunctionG(*C,*D,*A) + X[14] + ROOT2), 13);

   *A = Rotate((*A + FunctionG(*B,*C,*D) + X[3] + ROOT2), 3);
   *D = Rotate((*D + FunctionG(*A,*B,*C) + X[7] + ROOT2), 5);
   *C = Rotate((*C + FunctionG(*D,*A,*B) + X[11] + ROOT2), 9);
   *B = Rotate((*B + FunctionG(*C,*D,*A) + X[15] + ROOT2), 13);

}

void Round3(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int X[])
{
   *A = Rotate((*A + FunctionH(*B,*C,*D) + X[0] + ROOT3), 3);
   *D = Rotate((*D + FunctionH(*A,*B,*C) + X[8] + ROOT3), 9);
   *C = Rotate((*C + FunctionH(*D,*A,*B) + X[4] + ROOT3), 11);
   *B = Rotate((*B + FunctionH(*C,*D,*A) + X[12] + ROOT3), 15);

   *A = Rotate((*A + FunctionH(*B,*C,*D) + X[2] + ROOT3), 3);
   *D = Rotate((*D + FunctionH(*A,*B,*C) + X[10] + ROOT3), 9);
   *C = Rotate((*C + FunctionH(*D,*A,*B) + X[6] + ROOT3), 11);
   *B = Rotate((*B + FunctionH(*C,*D,*A) + X[14] + ROOT3), 15);

   *A = Rotate((*A + FunctionH(*B,*C,*D) + X[1] + ROOT3), 3);
   *D = Rotate((*D + FunctionH(*A,*B,*C) + X[9] + ROOT3), 9);
   *C = Rotate((*C + FunctionH(*D,*A,*B) + X[5] + ROOT3), 11);
   *B = Rotate((*B + FunctionH(*C,*D,*A) + X[13] + ROOT3), 15);

   *A = Rotate((*A + FunctionH(*B,*C,*D) + X[3] + ROOT3), 3);
   *D = Rotate((*D + FunctionH(*A,*B,*C) + X[11] + ROOT3), 9);
   *C = Rotate((*C + FunctionH(*D,*A,*B) + X[7] + ROOT3), 11);
   *B = Rotate((*B + FunctionH(*C,*D,*A) + X[15] + ROOT3), 15);

}
```

```
unsigned int FunctionF(unsigned int X, unsigned int Y, unsigned int Z)
{
    unsigned int W;

    W = ((X&Y) | ((~X)&Z));

    return(W);
}

unsigned int FunctionG(unsigned int X, unsigned int Y, unsigned int Z)
{
    unsigned int W;

    W = ((X&Y) | (X&Z) | (Y&Z));

    return(W);
}

unsigned int FunctionH(unsigned int X, unsigned int Y, unsigned int Z)
{
    unsigned int W;

    W = (X^Y^Z);

    return(W);
}

unsigned int Rotate(unsigned int X, unsigned int s)
{
    unsigned int temp;

    temp = X;
    X = (X << s) | (temp >> (32-s));
    return(X);
}

void Update(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int AA, unsigned int BB, unsigned int CC,
     unsigned int DD)
{
    *A = *A + AA;
    *B = *B + BB;
    *C = *C + CC;
    *D = *D + DD;
}


void PrintSignature(unsigned int A, unsigned int B, unsigned int C,
     unsigned int D)
{
   printf("Signature: ");
   PrintReverse(A);
   PrintReverse(B);
```

```
    PrintReverse(C);
    PrintReverse(D);
    printf("\n");
}

void PrintReverse(unsigned int X)
{
    int i;

    for(i=0; i<4; i++)
      {
printf("%.2x", X & 0x000000ff);
X = X >> 8;
      }

}

void RestoreFile(char filename[], unsigned int FileLen)
{
    unsigned char TempChar;
    unsigned char *array;
    unsigned int i;
    FILE *fp;

    fp = fopen(filename, "r+b");

    if(!fp)
      {
printf("Error opening file\n");
fclose(fp);
exit(1);
      }

    /* Allocate dynamic memmory */

    array = (unsigned char *)calloc(FileLen, sizeof(unsigned char));

    /* Read the file */

    for(i=0; i<FileLen; i++)
      {
fread(&TempChar, sizeof(unsigned char), 1, fp);
array[i] = TempChar;
      }

    fclose(fp);

    fp = fopen(filename, "wb");

    if(!fp)
      {
printf("Error opening file\n");
fclose(fp);
exit(1);
      }
```

```
    /* Write the file */

    for(i=0; i<FileLen; i++)
      {
TempChar = array[i];
fwrite(&TempChar, sizeof(unsigned char), 1, fp);
      }

    fclose(fp);

    /* Liberate some memory */

    free(array);

}

/* This function uses the above routines to construct a MD4 signature */

/* NB set argc = 2 */

int md4(int argc, char filename[])
{
    unsigned int i,j;
    unsigned int FileLen, PadLen;
    unsigned int A, B, C, D, AA, BB, CC, DD;
    unsigned int *M, *X;

    FileLen = 0;

    FileLen = PadBit(argc, filename, &PadLen);

    M = (unsigned int *)calloc(PadLen/4, sizeof(unsigned int));
    X = (unsigned int *)calloc(16, sizeof(unsigned int));

    ReadArray(argc, filename, M, PadLen/4);

    RestoreFile(filename, FileLen);

    Init(&A, &B, &C, &D);

    for(i=0; i<PadLen/64; i++)
      {

SaveParms(A, B, C, D, &AA, &BB, &CC, &DD);

for(j=0; j<16; j++)
  {
     X[j] = M[i*16+j];
  }

Round1(&A, &B, &C, &D, X);
Round2(&A, &B, &C, &D, X);
Round3(&A, &B, &C, &D, X);
```

```
Update(&A, &B, &C, &D, AA, BB, CC, DD);
     }

   PrintSignature(A,B,C,D);

   /* Liberate some memory */

   free(M);
   free(X);

   return(0);
}

unsigned int RotateRight(unsigned int X, unsigned int s)
{
   unsigned int temp;

   temp = X;
   X = (X >> s) | (temp << (32-s));
   return(X);
}
```

APPENDIX C: SOURCE CODE: ATTACK ON ALL THREE ROUNDS OF MD4

This is an implementation of the attack on MD4 as described by Dobbertin in [14]. Algorithm 6.3 is used for finding admissible inner almost-collisions in this implementation. This attack yields two messages that hash to the same value in less than one minute.

```
/* This a working version of the full attack on MD4. The alternative algorithm
 * for establishing inner almost-collisions is used in this program.
 *
 * Author: P.R. Kasselman
 * Filename: md4ga5.c
 * Date: 11 October 1996
 * Copyright: Ciphertec (1996) */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "md4.h"

#define A 0
#define B 1
#define C 2
#define D 3

int main()
{
   unsigned int i,j,k,l, Iteration;
   unsigned int Bi, Ci, U, Ut, V, Vt, W, Wt, Z, Zt;
   unsigned int As, Ds, Bs, Bst, Cs, Cst;
   unsigned int Condition, NewZ, NewW, Final, NextPhase;
   unsigned int DeltaW, DeltaV;
   unsigned int TempInt;
   unsigned int ABCD0[47][4], ABCD1[47][4], Delta19[4];
   unsigned int X1[16], X2[16];
   unsigned int Flag23, Flag27, DispFlag23, DispFlag27;
   unsigned int LastCondition, Collision, CollisionFlag;
   FILE *fp;

   time_t TheTime;

   TheTime = time(NULL);
   srand(TheTime);

   Iteration = 0;

   U = -1;
   Ut = 0;

   V = 0xfffdfffe;
   Vt = V;

   Wt = 0xfdffdfff;
   W = Wt + 0xeffff000;
```

```
    Bi = 0;
    Ci = 0;

    NextPhase = 1;

    while(NextPhase != 0)
      {
Bst = rand();
Bs = Bst + Rotate(1,25);

Cs = rand();
Cst = Cs + Rotate(1,5);

NewW = FunctionF(Vt,Ut,Bi) - FunctionF(V,U,Bi);
DeltaW = Rotate(Wt,21) - Rotate(W,21);

NewZ = 1;
Condition = 1;

while(Condition != -1)
  {
     while(NewZ !=0)
       {
  Zt = rand() | 0x00000000;
  Z = Zt + 0x00001001;

  NewZ = FunctionF(Wt,Vt,Ut) - FunctionF(W,V,U) -
    Rotate(Zt,13) + Rotate(Z,13);
     }

     NewZ = 1;

     Condition = FunctionG(Zt,Wt,Vt) - FunctionG(Z,W,V);
  }

DeltaV = 1;

while(DeltaV != 0)
  {
     As = rand();
     DeltaV = FunctionG(As,Zt,Wt) - FunctionG(As,Z,W);
  }

TempInt = 1;
Final = 1;

while(Final != 0)
  {
     while(TempInt != 0)
       {
  Cs = rand();
  Cst = Cs + Rotate(1,5);

  Bst = rand();
  Bs = Bst + Rotate(1,25);
```

```
   Ds = rand();
   TempInt = FunctionG(Ds,As,Zt) - FunctionG(Ds,As,Z) - W + Wt
     - Rotate(Cst,23) + Rotate(Cs,23);

        }

      TempInt = 1;

      Final = FunctionG(Cst,Ds,As) - FunctionG(Cs,Ds,As) - Z + Zt
        - Rotate(Bst,19) + Rotate(Bs,19) + 1;


   }


NextPhase = FunctionG(Bst,Cst,Ds) - FunctionG(Bs,Cs,Ds);
      }


    if(NextPhase == 0)
      {
printf("An admissable inner collision was found\n");
      }

    ABCD0[11][C] = Ci;
    ABCD1[11][C] = Ci;

    ABCD0[11][B] = Bi;
    ABCD1[11][B] = Bi;

    ABCD0[15][A] = U;
    ABCD1[15][A] = Ut;

    ABCD0[15][D] = V;
    ABCD1[15][D] = Vt;

    ABCD0[15][C] = W;
    ABCD1[15][C] = Wt;

    ABCD0[15][B] = Z;
    ABCD1[15][B] = Zt;

    ABCD0[19][A] = As;
    ABCD1[19][A] = As;

    ABCD0[19][D] = Ds;
    ABCD1[19][D] = Ds;

    ABCD0[19][C] = Cs;
    ABCD1[19][C] = Cst;

    ABCD0[19][B] = Bs;
    ABCD1[19][B] = Bst;
```

```
/* Find the message values that corrosponds to the computed values */

X1[13] = rand();
X2[13] = X1[13];

X1[14] = RotateRight(ABCD0[15][C],11) - ABCD0[11][C] -
  FunctionF(ABCD0[15][D],ABCD0[15][A],ABCD0[11][B]);

X2[14] = X1[14];

X1[15] = RotateRight(ABCD0[15][B],19) - ABCD0[11][B] -
  FunctionF(ABCD0[15][C],ABCD0[15][D],ABCD0[15][A]);

X2[15] = X1[15];

X1[0] = RotateRight(ABCD0[19][A],3) - ABCD0[15][A] -
  FunctionG(ABCD0[15][B],ABCD0[15][C],ABCD0[15][D]) - ROOT2;

X2[0] = X1[0];

X1[4] = RotateRight(ABCD0[19][D],5) - ABCD0[15][D] -
  FunctionG(ABCD0[19][A],ABCD0[15][B],ABCD0[15][C]) - ROOT2;

X2[4] = RotateRight(ABCD1[19][D],5) - ABCD1[15][D] -
  FunctionG(ABCD1[19][A],ABCD1[15][B],ABCD1[15][C]) - ROOT2;


X1[8] = RotateRight(ABCD0[19][C],9) - ABCD0[15][C] -
  FunctionG(ABCD0[19][D],ABCD0[19][A],ABCD0[15][B]) - ROOT2;

X2[8] = RotateRight(ABCD1[19][C],9) - ABCD1[15][C] -
  FunctionG(ABCD1[19][D],ABCD1[19][A],ABCD1[15][B]) - ROOT2;

X1[12] = RotateRight(ABCD0[19][B],13) - ABCD0[15][B] -
  FunctionG(ABCD0[19][C],ABCD0[19][D],ABCD0[19][A]) - ROOT2;

X2[12] = RotateRight(ABCD1[19][B],13) - ABCD1[15][B] -
  FunctionG(ABCD1[19][C],ABCD1[19][D],ABCD1[19][A]) - ROOT2;

ABCD0[11][D] = RotateRight(ABCD0[15][D],7) -
  FunctionF(ABCD0[15][A],ABCD0[11][B],ABCD0[11][C]) - X1[13];

ABCD1[11][D] = ABCD0[11][D];

ABCD0[11][A] = RotateRight(ABCD0[15][A],3) -
  FunctionF(ABCD0[11][B],ABCD0[11][C],ABCD0[11][D]) - X1[12];

ABCD1[11][A] = ABCD0[11][A];

/* Find the message values that will satisfy the initial values */

CollisionFlag = 1;
Flag23 = 0;
DispFlag23 = 0;
```

```
    Flag27 = 0;
    DispFlag27 = 0;

    while(CollisionFlag != 0)
      {
Iteration++;

if(Flag23 == 0)
  {
     X1[1] = rand();
     X2[1] = X1[1];

     X1[5] = rand();
     X2[5] = X1[5];
  } else if (DispFlag23 == 0) {
     printf("X1 and X5 are fixed\n");
     DispFlag23 = 1;
  }


if(Flag27 == 0)
  {
     X1[2] = rand();
     X2[2] = X1[2];
  } else if (DispFlag27 == 0) {
     printf("X2 is fixed\n");
     DispFlag27 = 1;
  }

X1[3] = rand();
X2[3] = X1[3];

ABCD0[0][A] = A0;
ABCD0[0][B] = B0;
ABCD0[0][C] = C0;
ABCD0[0][D] = D0;

ABCD0[3][A] = Rotate((ABCD0[0][A] +
      FunctionF(ABCD0[0][B],ABCD0[0][C],ABCD0[0][D]) +
      X1[0]), 3);

ABCD0[3][D] = Rotate((ABCD0[0][D] +
      FunctionF(ABCD0[3][A],ABCD0[0][B],ABCD0[0][C]) +
      X1[1]), 7);

ABCD0[3][C] = Rotate((ABCD0[0][C] +
      FunctionF(ABCD0[3][D],ABCD0[3][A],ABCD0[0][B])
      + X1[2]), 11);

ABCD0[3][B] = Rotate((ABCD0[0][B] +
      FunctionF(ABCD0[3][C],ABCD0[3][D],ABCD0[3][A])
      + X1[3]), 19);

ABCD0[7][A] = Rotate((ABCD0[3][A] +
      FunctionF(ABCD0[3][B],ABCD0[3][C],ABCD0[3][D]) +
```

```
        X1[4]), 3);

ABCD0[7][D] = Rotate((ABCD0[3][D] +
        FunctionF(ABCD0[7][A],ABCD0[3][B],ABCD0[3][C]) +
        X1[5]), 7);

ABCD0[7][B] = -1;

ABCD0[7][C] = RotateRight(ABCD0[11][A],3) - ABCD0[7][A] - X1[8];

X1[6] = ( RotateRight(ABCD0[7][C], 11) - ABCD0[3][C] -
 FunctionF(ABCD0[7][D],ABCD0[7][A],ABCD0[3][B]));

X2[6] = X1[6];

X1[7] = ( RotateRight(ABCD0[7][B], 19) - ABCD0[3][B] -
 FunctionF(ABCD0[7][C],ABCD0[7][D],ABCD0[7][A]) );

X2[7] = X1[7];

TempInt = Rotate((ABCD0[7][A] +
  FunctionF(ABCD0[7][B],ABCD0[7][C],ABCD0[7][D]) +
  X1[8]), 3);

X1[9] = ( RotateRight(ABCD0[11][D], 7) - ABCD0[7][D] -
 FunctionF(ABCD0[11][A],ABCD0[7][B],ABCD0[7][C]));

X2[9] = X1[9];


X1[10] = (RotateRight(ABCD0[11][C], 11) - ABCD0[7][C] -
  FunctionF(ABCD0[11][D],ABCD0[11][A],ABCD0[7][B]));

X2[10] = X1[10];

X1[11] = (RotateRight(ABCD0[11][B], 11) - ABCD0[7][B] -
  FunctionF(ABCD0[11][C],ABCD0[11][D],ABCD0[11][A]));

X2[11] = X1[11];

ABCD0[23][A] = Rotate((ABCD0[19][A] +
       FunctionG(ABCD0[19][B],ABCD0[19][C],ABCD0[19][D])
        + X1[1] + ROOT2), GS1);

ABCD1[23][A] = Rotate((ABCD1[19][A] +
       FunctionG(ABCD1[19][B],ABCD1[19][C],ABCD1[19][D])
        + X2[1] + ROOT2), GS1);

Collision = ABCD0[23][A] - ABCD1[23][A];

if(Collision == 0)
  {

     ABCD0[23][D] = Rotate((ABCD0[19][D] +
      FunctionG(ABCD0[23][A],ABCD0[19][B],ABCD0[19][C])
```

```
        + X1[5] + ROOT2), GS2);

      ABCD1[23][D] = Rotate((ABCD1[19][D] +
       FunctionG(ABCD1[23][A],ABCD1[19][B],ABCD1[19][C])
        + X2[5] + ROOT2), GS2);

      Collision = ABCD0[23][D] - ABCD1[23][D];

  }

if(Collision == 0)
  {
     ABCD0[23][C] = Rotate((ABCD0[19][C] +
      FunctionG(ABCD0[23][D],ABCD0[23][A],ABCD0[19][B])
      + X1[9] + ROOT2), GS3);

     ABCD1[23][C] = Rotate((ABCD1[19][C] +
      FunctionG(ABCD1[23][D],ABCD1[23][A],ABCD1[19][B])
      + X2[9] + ROOT2), GS3);

     Collision = ABCD0[23][C] - ABCD1[23][C];
  }

if(Collision == -1*Rotate(1,14))
  {
     ABCD0[23][B] = Rotate((ABCD0[19][B] +
    FunctionG(ABCD0[23][C],ABCD0[23][D],ABCD0[23][A])
      + X1[13] + ROOT2), GS4);

     ABCD1[23][B] = Rotate((ABCD1[19][B] +
      FunctionG(ABCD1[23][C],ABCD1[23][D],ABCD1[23][A])
      + X2[13] + ROOT2), GS4);


     Collision = ABCD0[23][B] - ABCD1[23][B];

  }

/* Is next iteration satisfied */

if(Collision == Rotate(1,6))
  {
     Flag23 = 1;

     ABCD0[27][A] = Rotate((ABCD0[23][A] +
    FunctionG(ABCD0[23][B],ABCD0[23][C],ABCD0[23][D])
      + X1[2] + ROOT2), GS1);

     ABCD1[27][A] = Rotate((ABCD1[23][A] +
    FunctionG(ABCD1[23][B],ABCD1[23][C],ABCD1[23][D])
      + X2[2] + ROOT2), GS1);

     Collision = ABCD0[27][A] - ABCD1[27][A];
  }
```

```
if(Collision == 0)
  {

      ABCD0[27][D] = Rotate((ABCD0[23][D] +
       FunctionG(ABCD0[27][A],ABCD0[23][B],ABCD0[23][C])
        + X1[6] + ROOT2), GS2);

      ABCD1[27][D] = Rotate((ABCD1[23][D] +
       FunctionG(ABCD1[27][A],ABCD1[23][B],ABCD1[23][C])
        + X2[6] + ROOT2), GS2);

      Collision = ABCD0[27][D] - ABCD1[27][D];

  }

if(Collision == 0)
  {
      ABCD0[27][C] = Rotate((ABCD0[23][C] +
       FunctionG(ABCD0[27][D],ABCD0[27][A],ABCD0[23][B])
       + X1[10] + ROOT2), GS3);

      ABCD1[27][C] = Rotate((ABCD1[23][C] +
       FunctionG(ABCD1[27][D],ABCD1[27][A],ABCD1[23][B])
       + X2[10] + ROOT2), GS3);

      Collision = ABCD0[27][C] - ABCD1[27][C];
  }

if(Collision == -1*Rotate(1,23))
  {
      ABCD0[27][B] = Rotate((ABCD0[23][B] +
     FunctionG(ABCD0[27][C],ABCD0[27][D],ABCD0[27][A])
       + X1[14] + ROOT2), GS4);

      ABCD1[27][B] = Rotate((ABCD1[23][B] +
       FunctionG(ABCD1[27][C],ABCD1[27][D],ABCD1[27][A])
       + X2[14] + ROOT2), GS4);


      Collision = ABCD0[27][B] - ABCD1[27][B];

  }

/* Calculate steps 28-31 */

if(Collision == Rotate(1,19))
  {

      Flag23 = 1;

      ABCD0[31][A] = Rotate((ABCD0[27][A] +
    FunctionG(ABCD0[27][B],ABCD0[27][C],ABCD0[27][D])
       + X1[3] + ROOT2), GS1);

      ABCD1[31][A] = Rotate((ABCD1[27][A] +
```

```
      FunctionG(ABCD1[27][B],ABCD1[27][C],ABCD1[27][D])
         + X2[3] + ROOT2), GS1);

       Collision = ABCD0[31][A] - ABCD1[31][A];
   }

 if(Collision == 0)
   {

      ABCD0[31][D] = Rotate((ABCD0[27][D] +
       FunctionG(ABCD0[31][A],ABCD0[27][B],ABCD0[27][C])
        + X1[7] + ROOT2), GS2);

      ABCD1[31][D] = Rotate((ABCD1[27][D] +
       FunctionG(ABCD1[31][A],ABCD1[27][B],ABCD1[27][C])
        + X2[7] + ROOT2), GS2);

      Collision = ABCD0[31][D] - ABCD1[31][D];

   }

 if(Collision == 0)
   {
      ABCD0[31][C] = Rotate((ABCD0[27][C] +
       FunctionG(ABCD0[31][D],ABCD0[31][A],ABCD0[27][B])
       + X1[11] + ROOT2), GS3);

      ABCD1[31][C] = Rotate((ABCD1[27][C] +
       FunctionG(ABCD1[31][D],ABCD1[31][A],ABCD1[27][B])
       + X2[11] + ROOT2), GS3);

      Collision = ABCD0[31][C] - ABCD1[31][C];
   }

 if(Collision == -1)
   {
      ABCD0[31][B] = Rotate((ABCD0[27][B] +
     FunctionG(ABCD0[31][C],ABCD0[31][D],ABCD0[31][A])
       + X1[15] + ROOT2), GS4);

      ABCD1[31][B] = Rotate((ABCD1[27][B] +
       FunctionG(ABCD1[31][C],ABCD1[31][D],ABCD1[31][A])
       + X2[15] + ROOT2), GS4);


      Collision = ABCD0[31][B] - ABCD1[31][B];

   }

/* Calculate steps 32-35 */

 if(Collision == 1)
   {

      ABCD0[35][A] = Rotate((ABCD0[31][A] +
```

```
      FunctionH(ABCD0[31][B],ABCD0[31][C],ABCD0[31][D])
         + X1[0] + ROOT3), HS1);

       ABCD1[35][A] = Rotate((ABCD1[31][A] +
      FunctionH(ABCD1[31][B],ABCD1[31][C],ABCD1[31][D])
         + X2[0] + ROOT3), HS1);

       Collision = ABCD0[35][A] - ABCD1[35][A];

   }

if(Collision == 0)
   {

       ABCD0[35][D] = Rotate((ABCD0[31][D] +
        FunctionH(ABCD0[35][A],ABCD0[31][B],ABCD0[31][C])
        + X1[8] + ROOT3), HS2);

       ABCD1[35][D] = Rotate((ABCD1[31][D] +
        FunctionH(ABCD1[35][A],ABCD1[31][B],ABCD1[31][C])
        + X2[8] + ROOT3), HS2);

       Collision = ABCD0[35][D] - ABCD1[35][D];

   }

if(Collision == 0)
   {
       ABCD0[35][C] = Rotate((ABCD0[31][C] +
        FunctionH(ABCD0[35][D],ABCD0[35][A],ABCD0[31][B])
        + X1[4] + ROOT3), HS3);

       ABCD1[35][C] = Rotate((ABCD1[31][C] +
        FunctionH(ABCD1[35][D],ABCD1[35][A],ABCD1[31][B])
        + X2[4] + ROOT3), HS3);

       Collision = ABCD0[35][C] - ABCD1[35][C];
   }

if(Collision == 0)
   {
       ABCD0[35][B] = Rotate((ABCD0[31][B] +
      FunctionH(ABCD0[35][C],ABCD0[35][D],ABCD0[35][A])
        + X1[12] + ROOT3), HS4);

       ABCD1[35][B] = Rotate((ABCD1[31][B] +
        FunctionH(ABCD1[35][C],ABCD1[35][D],ABCD1[35][A])
        + X2[12] + ROOT3), HS4);


       Collision = ABCD0[35][B] - ABCD1[35][B];

       if(Collision == 0)
         {
   CollisionFlag = 0;
```

```
        }

    }


      }

    printf("Iterations: %u\n", Iteration);

    printf("\tX1\t\t X2\n");
    for(i=0; i<16; i++)
      {
printf("(%u)\t%.8X\t %.8X\n", i, X1[i], X2[i]);
      }

    /* Write results to file */

    fp = fopen("X1.dat", "wb");

    if(!fp)
      {
printf("Error opening file\n");
fclose(fp);
exit(1);
      }

    /* Write the first message to file */

    for(i=0; i<16; i++)
      {
TempInt = X1[i];
fwrite(&TempInt, sizeof(unsigned int), 1, fp);
      }

    fclose(fp);

    fp = fopen("X2.dat", "wb");

    if(!fp)
      {
printf("Error opening file\n");
fclose(fp);
exit(1);
      }

    /* Write the second message to file */

    for(i=0; i<16; i++)
      {
TempInt = X2[i];
fwrite(&TempInt, sizeof(unsigned int), 1, fp);
      }

    fclose(fp);
```

```
    /* Test to see if a collision has occured */

    md4(2,"X1.dat");
    md4(2,"X2.dat");


    return(0);
}
```

APPENDIX D: IMPLEMENTATION: MD5

This Appendix contains the C source code of an implementation of MD5.

```
#define A0 0x67452301
#define B0 0xefcdab89
#define C0 0x98badcfe
#define D0 0x10325476

#define FS1 7
#define FS2 12
#define FS3 17
#define FS4 22

#define GS1 5
#define GS2 9
#define GS3 14
#define GS4 20

#define HS1 4
#define HS2 11
#define HS3 16
#define HS4 23

#define IS1 6
#define IS2 10
#define IS3 15
#define IS4 21

int T[64] = {0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf,
     0x4787c62a, 0xa8304613, 0xfd469501, 0x698098d8, 0x8b44f7af, 0xffff5bb1,
     0x895cd7be, 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821, 0xf61e2562,
     0xc040b340, 0x265e5a51, 0xe9b6c7aa, 0xd62f105d, 0x02441453, 0xd8a1e681,
     0xe7d3fbc8, 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed, 0xa9e3e905,
     0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a, 0xfffa3942, 0x8771f681, 0x6d9d6122,
     0xfde5380c, 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70, 0x289b7ec6,
     0xeaa127fa, 0xd4ef3085, 0x04881d05, 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8,
     0xc4ac5665, 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3,
     0x8f0ccc92, 0xffeff47d, 0x85845dd1, 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314,
     0x4e0811a1, 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391};

char *PadBit(char *Message, unsigned int *Length);
void char_2_int_array(char *Message, unsigned int *MessageInt, unsigned in-
t Length);
unsigned int *AppendLength(unsigned int *MessageInt,
   unsigned int *AppendLength, unsigned int OrgLen);
unsigned int MD5_F(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int MD5_G(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int MD5_H(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int MD5_I(unsigned int X, unsigned int Y, unsigned int Z);
void InitMD5Buf(unsigned int *A, unsigned int *B,
unsigned int *C, unsigned int *D);
void SaveMD5Parms(unsigned int A, unsigned int B, unsigned int C,
  unsigned int D, unsigned int *AA, unsigned int *BB,
  unsigned int *CC, unsigned int *DD);
void Round1(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X);
```

```
void Round2(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X);
void Round3(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X);
void Round4(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X);
void Update(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int AA, unsigned int BB, unsigned int CC,
    unsigned int DD);
unsigned int RotateLeft(unsigned int X, unsigned int s);
void PrintSignature(unsigned int *A);
void PrintReverse(unsigned int X);
unsigned int Reverse(unsigned int X);
void MD5(char *Message, unsigned int Length, unsigned int *Hash);

void MD5(char *Message, unsigned int Length, unsigned int *Hash)
{
   int i,j;
   int PaddedSize, AppendSize;
   unsigned int *MessageInt, *X;
   unsigned int A, B, C, D, AA, BB, CC, DD;

   PaddedSize = Length;

   X =(unsigned int *)calloc(16, sizeof(unsigned int));
   if(X == NULL)
     {
printf("Error Allocating Memory\n");
exit(1);
     }

   /* Determine the number of Padding bytes required */
   Message = PadBit(Message, &PaddedSize);

   MessageInt =(unsigned int *)calloc(PaddedSize/(sizeof(unsigned int)), size-
of(unsigned int));

   char_2_int_array(Message, MessageInt, PaddedSize);
   AppendSize = PaddedSize;

   for(i=0; i<PaddedSize/(sizeof(unsigned int)); i++)
     {
MessageInt[i] = Reverse(MessageInt[i]);
     }

   MessageInt = AppendLength(MessageInt, &AppendSize, Length);

   InitMD5Buf(&A,&B,&C,&D);

   for(i=0; i<(AppendSize*8)/512; i++)
     {
```

```
SaveMD5Parms(A, B, C, D, &AA, &BB, &CC, &DD);

for(j=0; j<16; j++)
  {
     X[j] = MessageInt[i*16+j];
  }

Round1(&A, &B, &C, &D, X);
Round2(&A, &B, &C, &D, X);
Round3(&A, &B, &C, &D, X);
Round4(&A, &B, &C, &D, X);
Update(&A, &B, &C, &D, AA, BB, CC, DD);
     }

   Hash[0] = A;
   Hash[1] = B;
   Hash[2] = C;
   Hash[3] = D;

   free(X);
   free(MessageInt);
}

/* This routine performs the bit padding as requied for the MD4 algorithm */
char *PadBit(char *Message, unsigned int *Length)
{
   char *TempPtr;
   unsigned int i,j;
   unsigned int FileLen, FileBits, PadBits, PadBytes;
   unsigned char temp;

   FileBits = *Length*8*sizeof(unsigned char);

   /* Compute the number of bits needed for padding */
   PadBits = abs( (448 - FileBits) % 512 );
   if(PadBits == 0)
     {
PadBits = 512*(FileBits/512) + 512;
     }

   PadBytes = PadBits/8;

   TempPtr = (char *)realloc(Message, (*Length+PadBytes));
   if(TempPtr == NULL)
     {
printf("Error Reallocating Memory\n");
exit(1);
     }

   /* Pad bit "1" */

   TempPtr[*Length] = 0x80;
   /* Pad zero bits */
   for(i=1; i<PadBytes; i++)
     {
```

```
TempPtr[*Length+i] = 0x00;
     }

   Message = (char *)realloc(TempPtr, (*Length+PadBytes));
   if(Message == NULL)
     {
printf("Error Reallocating Memory\n");
exit(1);
     }

   *Length = (PadBytes+(*Length));

   return(Message);
}

unsigned int *AppendLength(unsigned int *MessageInt,
   unsigned int *AppendLength, unsigned int OrgLen)
{
   int i;
   unsigned int *TempPtr;

   *AppendLength = *AppendLength+2*sizeof(unsigned int);

   TempPtr = (unsigned int *)realloc(MessageInt, *AppendLength);
   if(TempPtr == NULL)
     {
printf("Error Reallocating Memory\n");
exit(1);
     }

   TempPtr[*AppendLength/sizeof(unsigned int)-2] = OrgLen*8;
   TempPtr[*AppendLength/sizeof(unsigned int)-1] = 0;

   MessageInt = (unsigned int *)realloc(TempPtr, *AppendLength);
   if(MessageInt == NULL)
     {
printf("Error Reallocating Memory\n");
exit(1);
     }

   return(MessageInt);
}

void char_2_int_array(char *Message, unsigned int *MessageInt,
      unsigned int Length)
{
   int i, j;

   for(i=0; i<(Length/sizeof(unsigned int)); i++)
     {
for(j=0; j<sizeof(unsigned int); j++)
  {
     MessageInt[i] = (MessageInt[i] << 8*sizeof(char)) |
       (Message[i*sizeof(unsigned int) + j] & 0x000000ff);
  }
```

```
    }

}

unsigned int MD5_F(unsigned int X, unsigned int Y, unsigned int Z)
{
   return((X&Y) | (~X&Z));
}

unsigned int MD5_G(unsigned int X, unsigned int Y, unsigned int Z)
{
   return((X & Z) | (Y & (~Z)));
}

unsigned int MD5_H(unsigned int X, unsigned int Y, unsigned int Z)
{
   return(X ^ Y ^ Z);
}

unsigned int MD5_I(unsigned int X, unsigned int Y, unsigned int Z)
{
   return(Y ^ (X | (~Z)));
}

void InitMD5Buf(unsigned int *A, unsigned int *B,
unsigned int *C, unsigned int *D)
{
   *A = A0;
   *B = B0;
   *C = C0;
   *D = D0;
}

void SaveMD5Parms(unsigned int A, unsigned int B, unsigned int C,
       unsigned int D, unsigned int *AA, unsigned int *BB,
       unsigned int *CC, unsigned int *DD)
{
   *AA = A;
   *BB = B;
   *CC = C;
   *DD = D;
}

void Round1(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
    unsigned int *X)
{
   *A = *B + RotateLeft((*A + MD5_F(*B,*C,*D) + X[0] + T[0]), FS1);
   *D = *A + RotateLeft((*D + MD5_F(*A,*B,*C) + X[1] + T[1]), FS2);
   *C = *D + RotateLeft((*C + MD5_F(*D,*A,*B) + X[2] + T[2]), FS3);
   *B = *C + RotateLeft((*B + MD5_F(*C,*D,*A) + X[3] + T[3]), FS4);

   *A = *B + RotateLeft((*A + MD5_F(*B,*C,*D) + X[4] + T[4]), FS1);
   *D = *A + RotateLeft((*D + MD5_F(*A,*B,*C) + X[5] + T[5]), FS2);
   *C = *D + RotateLeft((*C + MD5_F(*D,*A,*B) + X[6] + T[6]), FS3);
```

```
    *B = *C + RotateLeft((*B + MD5_F(*C,*D,*A) + X[7] + T[7]), FS4);

    *A = *B + RotateLeft((*A + MD5_F(*B,*C,*D) + X[8] + T[8]), FS1);
    *D = *A + RotateLeft((*D + MD5_F(*A,*B,*C) + X[9] + T[9]), FS2);
    *C = *D + RotateLeft((*C + MD5_F(*D,*A,*B) + X[10] + T[10]), FS3);
    *B = *C + RotateLeft((*B + MD5_F(*C,*D,*A) + X[11] + T[11]), FS4);

    *A = *B + RotateLeft((*A + MD5_F(*B,*C,*D) + X[12] + T[12]), FS1);
    *D = *A + RotateLeft((*D + MD5_F(*A,*B,*C) + X[13] + T[13]), FS2);
    *C = *D + RotateLeft((*C + MD5_F(*D,*A,*B) + X[14] + T[14]), FS3);
    *B = *C + RotateLeft((*B + MD5_F(*C,*D,*A) + X[15] + T[15]), FS4);
}

void Round2(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int *X)
{
    *A = *B + RotateLeft((*A + MD5_G(*B,*C,*D) + X[1] + T[16]), GS1);
    *D = *A + RotateLeft((*D + MD5_G(*A,*B,*C) + X[6] + T[17]), GS2);
    *C = *D + RotateLeft((*C + MD5_G(*D,*A,*B) + X[11] + T[18]), GS3);
    *B = *C + RotateLeft((*B + MD5_G(*C,*D,*A) + X[0] + T[19]), GS4);

    *A = *B + RotateLeft((*A + MD5_G(*B,*C,*D) + X[5] + T[20]), GS1);
    *D = *A + RotateLeft((*D + MD5_G(*A,*B,*C) + X[10] + T[21]), GS2);
    *C = *D + RotateLeft((*C + MD5_G(*D,*A,*B) + X[15] + T[22]), GS3);
    *B = *C + RotateLeft((*B + MD5_G(*C,*D,*A) + X[4] + T[23]), GS4);

    *A = *B + RotateLeft((*A + MD5_G(*B,*C,*D) + X[9] + T[24]), GS1);
    *D = *A + RotateLeft((*D + MD5_G(*A,*B,*C) + X[14] + T[25]), GS2);
    *C = *D + RotateLeft((*C + MD5_G(*D,*A,*B) + X[3] + T[26]), GS3);
    *B = *C + RotateLeft((*B + MD5_G(*C,*D,*A) + X[8] + T[27]), GS4);

    *A = *B + RotateLeft((*A + MD5_G(*B,*C,*D) + X[13] + T[28]), GS1);
    *D = *A + RotateLeft((*D + MD5_G(*A,*B,*C) + X[2] + T[29]), GS2);
    *C = *D + RotateLeft((*C + MD5_G(*D,*A,*B) + X[7] + T[30]), GS3);
    *B = *C + RotateLeft((*B + MD5_G(*C,*D,*A) + X[12] + T[31]), GS4);
}

void Round3(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int *X)
{
    *A = *B + RotateLeft((*A + MD5_H(*B,*C,*D) + X[5] + T[32]), HS1);
    *D = *A + RotateLeft((*D + MD5_H(*A,*B,*C) + X[8] + T[33]), HS2);
    *C = *D + RotateLeft((*C + MD5_H(*D,*A,*B) + X[11] + T[34]), HS3);
    *B = *C + RotateLeft((*B + MD5_H(*C,*D,*A) + X[14] + T[35]), HS4);

    *A = *B + RotateLeft((*A + MD5_H(*B,*C,*D) + X[1] + T[36]), HS1);
    *D = *A + RotateLeft((*D + MD5_H(*A,*B,*C) + X[4] + T[37]), HS2);
    *C = *D + RotateLeft((*C + MD5_H(*D,*A,*B) + X[7] + T[38]), HS3);
    *B = *C + RotateLeft((*B + MD5_H(*C,*D,*A) + X[10] + T[39]), HS4);

    *A = *B + RotateLeft((*A + MD5_H(*B,*C,*D) + X[13] + T[40]), HS1);
    *D = *A + RotateLeft((*D + MD5_H(*A,*B,*C) + X[0] + T[41]), HS2);
    *C = *D + RotateLeft((*C + MD5_H(*D,*A,*B) + X[3] + T[42]), HS3);
```

```
    *B = *C + RotateLeft((*B + MD5_H(*C,*D,*A) + X[6] + T[43]), HS4);

    *A = *B + RotateLeft((*A + MD5_H(*B,*C,*D) + X[9] + T[44]), HS1);
    *D = *A + RotateLeft((*D + MD5_H(*A,*B,*C) + X[12] + T[45]), HS2);
    *C = *D + RotateLeft((*C + MD5_H(*D,*A,*B) + X[15] + T[46]), HS3);
    *B = *C + RotateLeft((*B + MD5_H(*C,*D,*A) + X[2] + T[47]), HS4);
}

void Round4(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int *X)
{
    *A = *B + RotateLeft((*A + MD5_I(*B,*C,*D) + X[0] + T[48]), IS1);
    *D = *A + RotateLeft((*D + MD5_I(*A,*B,*C) + X[7] + T[49]), IS2);
    *C = *D + RotateLeft((*C + MD5_I(*D,*A,*B) + X[14] + T[50]), IS3);
    *B = *C + RotateLeft((*B + MD5_I(*C,*D,*A) + X[5] + T[51]), IS4);

    *A = *B + RotateLeft((*A + MD5_I(*B,*C,*D) + X[12] + T[52]), IS1);
    *D = *A + RotateLeft((*D + MD5_I(*A,*B,*C) + X[3] + T[53]), IS2);
    *C = *D + RotateLeft((*C + MD5_I(*D,*A,*B) + X[10] + T[54]), IS3);
    *B = *C + RotateLeft((*B + MD5_I(*C,*D,*A) + X[1] + T[55]), IS4);

    *A = *B + RotateLeft((*A + MD5_I(*B,*C,*D) + X[8] + T[56]), IS1);
    *D = *A + RotateLeft((*D + MD5_I(*A,*B,*C) + X[15] + T[57]), IS2);
    *C = *D + RotateLeft((*C + MD5_I(*D,*A,*B) + X[6] + T[58]), IS3);
    *B = *C + RotateLeft((*B + MD5_I(*C,*D,*A) + X[13] + T[59]), IS4);

    *A = *B + RotateLeft((*A + MD5_I(*B,*C,*D) + X[4] + T[60]), IS1);
    *D = *A + RotateLeft((*D + MD5_I(*A,*B,*C) + X[11] + T[61]), IS2);
    *C = *D + RotateLeft((*C + MD5_I(*D,*A,*B) + X[2] + T[62]), IS3);
    *B = *C + RotateLeft((*B + MD5_I(*C,*D,*A) + X[9] + T[63]), IS4);

    /*printf("%8.8X %8.8X %8.8X %8.8X\n", *A, *B, *C, *D);*/
}

void Update(unsigned int *A, unsigned int *B, unsigned int *C, unsigned in-
t *D,
     unsigned int AA, unsigned int BB, unsigned int CC,
     unsigned int DD)
{
    *A = *A + AA;
    *B = *B + BB;
    *C = *C + CC;
    *D = *D + DD;
}

unsigned int RotateLeft(unsigned int X, unsigned int s)
{
    unsigned int temp;

    temp = X;
    X = (X << s) | (temp >> (32-s));
    return(X);
}
```

```
void PrintSignature(unsigned int *A)
{
   int i;

   for(i=0; i<4; i++)
     {
PrintReverse(A[i]);
     }
}

void PrintReverse(unsigned int X)
{
   int i;

   for(i=0; i<4; i++)
     {
printf("%.2x", X & 0x000000ff);
X = X >> 8;
     }

}

unsigned int Reverse(unsigned int X)
{
   int i;
   unsigned int Y;

   Y=0;
   for(i=0; i<4; i++)
     {
Y = (Y<<8) | X & 0x000000ff;
X = X >> 8;
     }
   return(Y);
}

#undef A0
#undef B0
#undef C0
#undef D0

#undef FS1
#undef FS2
#undef FS3
#undef FS4

#undef GS1
#undef GS2
#undef GS3
#undef GS4

#undef HS1
#undef HS2
#undef HS3
#undef HS4
```

```
#undef IS1
#undef IS2
#undef IS3
#undef IS4
```

APPENDIX E: SOURCE CODE: ANALYSIS OF MD5

E.1 SOURCE CODE: FIRST PHASE OF THE ATTACK ON MD5

This section of the appendix contains the C source code used in the first phase of the attack on MD5.

```
/* This program is a reconstruction of Dobbertin's attack on MD5.
 *
 * Author: P.R. Kasselman
 * Date: 16/06/1997
 * Filename: md5an04.c */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

unsigned int Rotate(unsigned int X, unsigned int s);
unsigned int RotateRight(unsigned int X, unsigned int s);
unsigned int F(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int G(unsigned int X, unsigned int Y, unsigned int Z);
unsigned int GG(unsigned int x, unsigned int c1, unsigned int d1,
unsigned int c2, unsigned int d2, unsigned int a);
unsigned int GGG(unsigned int x, unsigned int a1, unsigned int c1,
unsigned int a2, unsigned int c2, unsigned int b);
unsigned int NG(unsigned int x, unsigned int d1, unsigned int a1,
unsigned int d2, unsigned int a2, unsigned int c);
unsigned int search1(unsigned int x, unsigned int c1, unsigned int d1,
  unsigned int c2, unsigned int d2, unsigned int a,
  unsigned int Target, unsigned int *data);
unsigned int search2(unsigned int x, unsigned int a1, unsigned int c1,
  unsigned int a2, unsigned int c2, unsigned int b,
  unsigned int Target, unsigned int *data);
unsigned int search3(unsigned int x, unsigned int d1, unsigned int a1,
  unsigned int d2, unsigned int a2, unsigned int c,
  unsigned int Target, unsigned int *data);

#define FS1 7
#define FS2 12
#define FS3 17
#define FS4 22

#define GS1 5
#define GS2 9
#define GS3 14
#define GS4 20

#define EPS1 0x08000000
#define EPS2 0xfffffe00

#define MAX 0x00100000

int main()
{
 unsigned int i,j,k,l;
 unsigned int x,konst;
```

```
unsigned int data1[MAX], data2[MAX], data3[MAX];
unsigned int b2,c2,a,b1,c1;
unsigned int A15,B15,C15,D15,A19,B19,C19,D19,A23,B23,C23,D23;
unsigned int Bt15,Ct15,At19,Ct19,Dt19,Dt23;
unsigned int DeltaX14;
unsigned int DeltaA19, DeltaC19, DeltaD19, DeltaC15, DeltaB15;
unsigned int BasicA15, BasicB15, BasicC19;
unsigned int Stop, Phase, Test, Funny, MaxPhase, Y, alt;
unsigned int SearchCount1, SearchCount2, SearchCount3, SearchCount4;
unsigned int Diff1, Diff2, Diff3;
unsigned int Count1, Count2, Count3;
unsigned int Target1, Target2, Target3;
unsigned int M[8];
time_t TheTime;

TheTime = time(NULL);
srandom(TheTime);

/* Specify the difference for X14 */
DeltaX14 = Rotate(1,9);

M[0] = 0x0000000f;
M[1] = 0x000000ff;
M[2] = 0x00000fff;
M[3] = 0x0000ffff;
M[4] = 0x000fffff;
M[5] = 0x00ffffff;
M[6] = 0x0fffffff;
M[7] = 0xffffffff;

SearchCount1 = 0;
SearchCount2 = 0;
SearchCount3 = 0;
SearchCount4 = 0;

/* Make initial choices for D19, Dt19, D15, C15 and Ct15 that is guaranteed
   to satisfy the conditions imposed. */

D19 = 0x00000000;
Dt19 = -1-D19;
DeltaD19 = Dt19-D19;

printf("Delta D19: %8.8X\n", DeltaD19);

C19 =  random() ^ Rotate(random(),1);
Ct19 = C19+EPS1;
DeltaC19 = Ct19-C19;

printf("Delta C19: %8.8X\n", DeltaC19);

D15 = Rotate(1,16)-Rotate(1,25);
C15 = D15 - 1;
Ct15 = Rotate(Rotate(C15-D15,15)+DeltaX14,17)+D15;
DeltaC15 = Ct15-C15;
```

```
 Phase = 0;

 printf("Delta C15: %8.8X\n", DeltaC15);

 Stop = 0;
 Count3 = 0;
 while(Stop == 0)
  {
if(Phase == 0)
 {
   MaxPhase = 0;
SearchCount1 = 0;
SearchCount2 = 0;
  SearchCount3 = 0;
  SearchCount4 = 0;

  /* Choose A15 randomly */
  A15 = random() ^ Rotate(random(),1);

  /* Determine Delta B15 based from (2) */


    DeltaB15 = Rotate(F(Ct15,D15,A15)-F(C15,D15,A15),22)
  + Rotate(Ct15-C15,0);

  /*printf("Delta B15: %8.8X\n", DeltaB15);*/

  /* Choose B15 randomly */
  B15 = random() ^ Rotate(random(),1);

  /* Compute Bt15 based on the random choice of B15 */
  Bt15 = DeltaB15 + B15;

  /* Determine Delta A19 from (3) */
  DeltaA19 = Rotate(G(Bt15,Ct15,D15)-G(B15,C15,D15),5)
  + Rotate(Bt15-B15,0);

  /*printf("Delta A19: %8.8X\n", DeltaA19);*/

  /* Choose C19 randomly */
  C19 = random() ^ Rotate(random(),1);;
Ct19 = C19+EPS1;
 }

if(Phase > 0)
 {
  /* Choose A15 to be close to the previous "good" value */
  Diff1 = random()&random()&random()&random();
  A15 = BasicA15 ^ Diff1;

  /* Determine Delta B15 based from (2) */
  DeltaB15 = Rotate(F(Ct15,D15,A15)-F(C15,D15,A15),22)
  + Rotate(Ct15-C15,0);

  /*printf("Delta B15: %8.8X\n", DeltaB15);*/
```

```
  /* Choose B15 to be close to the previous "good" value */
  Diff2 = random()&random()&random()&random();
  B15 = BasicB15 ^ Diff2;

  /* Compute Bt15 based on the random choice of B15 */
  Bt15 = DeltaB15 + B15;

  /* Determine Delta A19 from (3) */
  DeltaA19 = Rotate(G(Bt15,Ct15,D15)-G(B15,C15,D15),5)
  + Rotate(Bt15-B15,0);

  /*printf("Delta A19: %8.8X\n", DeltaA19);*/

  /* Choose C19 randomly */
  Diff3 = random()&random()&random()&random();
  C19 = BasicC19 ^ Diff3;
Ct19 = C19+EPS1;
 }

   Funny = 0;

/* Set a target */
Target1 =  Rotate(-DeltaA19,12) - Rotate(-DeltaA19-EPS1,12) + DeltaB15;

/* For the given target determine if any solutions exists, and if so,
 * how many solutions exists. */

x = random();

/* Find valid  values for A19 and implicitly At19 */
Count1 = 0;
Count1 = search1(x,C19,D19,Ct19,Dt19,DeltaA19,Target1,data1);

/* The array data1 now contains all valid values of A19 */
for(i=0; i<Count1; i++)
 {
A19  = data1[i];
At19 = data1[i] + DeltaA19;

Target2 = Rotate(D19-A19, 23) - Rotate(Dt19-At19, 23);

  x = random();

/* Find valid  values for B15 and implicitly Bt15 */
Count2 = 0;
Count2 = search2(x,A19,C15,At19,Ct15,DeltaB15,Target2,data2);
for(j=0; j<Count2; j++)
 {
B15  = data2[j];
Bt15 = data2[j] + DeltaB15;

/* Confirm if equation (3) holds */
Test = G(B15,C15,D15)-G(Bt15,Ct15,D15)
- Rotate(A19 - B15, 27)
```

```
+ Rotate(At19 - Bt15, 27);
if(Test == 0)
 {
/* Confirm if equation (2) holds */
Test = F(C15,D15,A15) - F(Ct15,D15,A15)
- Rotate(B15-C15,10)
+ Rotate(Bt15-Ct15,10);

if(Test == 0)
   {
    /* Now determine if equation (5) holds */
  Test = G(D19,A19,B15)-G(Dt19,At19,Bt15)
- Rotate(C19 - D19, 18)
+ Rotate(Ct19 - Dt19, 18)
+ C15 - Ct15;

if(((Test&M[0])==0)&&(Phase<2))
 {
   Phase = 1;
   Funny = 1;
   /*printf("Hit %d\n", Phase);*/
 }

if(((Test&M[1])==0)&&(Phase<3))
 {
   Phase = 2;
   Funny = 2;
   /* printf("Hit %d\n", Phase);*/
 }
if(((Test&M[2])==0)&&(Phase<4))
 {
   Phase = 3;
   Funny = 3;
   /*printf("Hit %d\n", Phase);*/
 }
if(((Test&M[3])==0)&&(Phase<5))
 {
   Phase = 4;
   Funny = 4;
   /*printf("Hit %d\n", Phase);*/
   SearchCount1++;

   if(SearchCount1 > 10000)
    {
     SearchCount1 = 0;
Phase = 0;
printf("Resetting Process\n");
    }

 }

if(((Test&M[4])==0)&&(Phase<6))
 {
   Phase = 5;
   Funny = 5;
```

```
    SearchCount2++;
    /*printf("Hit %d\n", Phase);*/

    if(SearchCount2 > 10000)
     {
      SearchCount2 = 0;
Phase = 0;
printf("Resetting Process\n");
     }

 }

if(((Test&M[5])==0)&&(Phase<7))
 {
    Phase = 6;
    Funny = 6;
    SearchCount3++;
        /*printf("Hit %d\n", Phase);*/
    if(SearchCount3 > 10000)
     {
      SearchCount3 = 0;
Phase = 0;
printf("Resetting Process\n");
     }
 }

if(((Test&M[6])==0)&&(Phase<8))
 {
    Phase = 7;
    Funny = 7;
    SearchCount4++;

    /*printf("Hit %d\n", Phase);*/
    if(SearchCount4 > 16000)
     {
      SearchCount4 = 0;
Phase = 0;
printf("Resetting Process\n");
     }
 }

if(Test == 0)
 {
  printf("Checking Results\n");
  Test = F(Ct15,D15,A15)
  - F(C15,D15,A15)
  - Rotate(Bt15-Ct15,10)
  + Rotate(B15-C15,10);

  if(Test!=0)
   {
    printf("Error 1\n");
    exit(1);
   }
```

```
  Test = G(B15,C15,D15)
  - G(Bt15,Ct15,D15)
  - Rotate(A19-B15,27)
  + Rotate(At19-Bt15,27);

  if(Test!=0)
   {
    printf("Error 2\n");
    exit(1);
   }
Test = G(A19,B15,C15)
  - G(At19,Bt15,Ct15)
  - Rotate(D19-A19,23)
  + Rotate(Dt19-At19,23);

  if(Test!=0)
   {
    printf("Error 3\n");
    exit(1);
   }

Test = G(D19,A19,B15)
  - G(Dt19,At19,Bt15)
  - Rotate(C19-D19,18)
  + Rotate(Ct19-Dt19,18)
  + C15-Ct15;

  if(Test!=0)
   {
    printf("Error 4\n");
    exit(1);
   }

  /* Now that equations (1)-(7) are
   * satisfied, determine if (8) is
   * satisfied and if the error
   * propagates as expected */

  Target3 =  Rotate(-DeltaA19,12)
        - Rotate(-DeltaA19-EPS1,12)
        + DeltaB15;

  x = random();

  /* Find valid  values for C19 and
   * implicitly Ct19 */

  Count3 = 0;
  Count3 = search3(x,D19,A19,Dt19,At19,DeltaC19,Target3,data3);

  for(k=0; k<Count3; k++)
   {
    C19 = data3[k];
    Ct19 = data3[k] + EPS1;
    B19 = C19 - DeltaA19;
```

```
/* Confirm if (6) holds */
Test = G(C19,D19,A19)
     - G(Ct19,Dt19,At19)
     + B15 - Bt15
     - Rotate(B19-C19,12)
     + Rotate(B19-Ct19,12);
if(Test != 0)
 {
  printf("Error 4\n");
  exit(1);
 }

/* Confirm if (7) holds */
Test = G(B19,C19,D19)
     - G(B19,Ct19,Dt19)
     + A19 - At19;
if(Test != 0)
 {
  printf("Error 5\n");
  exit(1);
 }

/* Determine if propagation of
 * differences are as expected */

 alt = 0;
 if(alt==0)
  {
   for(l=0; l<10; l++)
    {
     A23 = random()
      ^ Rotate(random(),1);
     D23 = random()
      ^ Rotate(random(),1);
     Dt23 = D23 + EPS2;
     Y = random()
              ^ Rotate(random(),1);

     Test = Rotate(Ct19
      + G(Dt23,A23,B19)
       + Y,14)
      + Dt23;

     Test = Test - (Rotate(C19
      + G(D23,A23,B19)
      + Y,14)
      + D23);

     if(Test != 0)
      {
       Phase = 0;
      }

     if(Test == 0)
```

```
            {
    Test = Rotate(Dt23-A23,23)
       - Rotate(D23-A23,23);
    Test = Test
       - G(A23,B19,Ct19)
       + G(A23,B19,C19);
    Test = Test - (Dt19-D19);

    if(Test == 0)
     {
        Stop++;
        printf("A15  = 0x%8.8X;\n", A15);
        printf("D15  = 0x%8.8X;\n", D15);
        printf("C15  = 0x%8.8X;\n", C15);
        /*printf("Ct15 = 0x%8.8X;\n", Ct15);*/
        printf("B15  = 0x%8.8X;\n", B15);
        /*printf("Bt15 = 0x%8.8X;\n", Bt15);*/
        printf("A19  = 0x%8.8X;\n", A19);
        /*printf("At19 = 0x%8.8X;\n", At19);*/
        printf("D19  = 0x%8.8X;\n", D19);
        /*printf("Dt19 = 0x%8.8X;\n", Dt19);*/
        printf("C19  = 0x%8.8X;\n", C19);
        /*printf("Ct19 = 0x%8.8X;\n", Ct19);*/
        printf("B19  = 0x%8.8X;\n", B19);
        printf("A23  = 0x%8.8X;\n", A23);
        /*printf("B23  = 0x%8.8X;\n", B23);
        printf("C23  = 0x%8.8X;\n", C23);
        printf("D23  = 0x%8.8X;\n", D23);
        printf("Dt23 = 0x%8.8X;\n", Dt23);*/

        exit(0);
             }

           }
         }
         if(Stop == 1)
          {
           printf("Found a Solution\n");
          }
       }
    }

 }

/* Preserve the basic values */
if(Funny == Phase)
 {
  BasicA15 = A15;
     BasicB15 = B15;
     BasicC19 = C19;
 }
 }
 }
 }
 }
```

```
if(Phase > MaxPhase)
 {
  MaxPhase = Phase;
printf("Best Current Phase: %d\n", MaxPhase);
printf("%d %d %d %d\n", SearchCount1, SearchCount2,
SearchCount3,SearchCount4);
 }
  }

 printf("Target: %8.8X\n", Target1);
 printf("Target: %8.8X\n", Target2);
 printf("Number of solutions (7): %d\n", Count1);
 printf("Number of solutions (4a): %d\n", Count2);
 printf("Number of solutions (4b): %d\n", Count3);
 printf("Number of Valid Solutions (3) %d\n", Stop);
 return(0);
}

/* Hierdie deel kan so klein bietjie meer elegant gedoen word,
 * maar a.g.v. tydbeperkings gaan ek nie nou daaraan karring nie */

unsigned int search1(unsigned int x, unsigned int c1, unsigned int d1,
  unsigned int c2, unsigned int d2, unsigned int a,
  unsigned int Target, unsigned int *data)
{
 int i;
 static int index, count;

 i = index;

 if(i==0)
  {
count = 0;
  }


 /* This is where you check the depth of the tree */
 if(i==32)
  {
   data[count] = x;
count++;
return(count);
  }

 index++;

 if(count >= MAX)
  {
   index--;
   return(count);
  }

 if( (((GG(x,c1,d1,c2,d2,a) - Target) >> i) & 0x00000001) == 0)
  {
```

```
search1(x,c1,d1,c2,d2,a,Target,data);
  }

  x = x^(1<<i);

 if( (((GG(x,c1,d1,c2,d2,a)-Target) >> i) & 0x00000001) == 0)
  {
search1(x,c1,d1,c2,d2,a,Target,data);
  }

 x = x^(1<<i);

 index--;
 return(count);
}

/* Soek oplossings vir vergelyking 4 */
unsigned int search2(unsigned int x, unsigned int a1, unsigned int c1,
  unsigned int a2, unsigned int c2, unsigned int b,
  unsigned int Target, unsigned int *data)
{
 int i;
 static int index, count;

 i = index;

 if(i==0)
  {
count = 0;
  }


 /* This is where you check the depth of the tree */
 if(i==32)
  {
   data[count] = x;
count++;
return(count);
  }

 index++;

 if(count >= MAX)
  {
   index--;
   return(count);
  }

 if( (((GGG(x,a1,c1,a2,c2,b)-Target) >> i) & 0x00000001) == 0)
  {
search2(x,a1,c1,a2,c2,b,Target,data);
  }

  x = x^(1<<i);
```

```
 if( (((GGG(x,a1,c1,a2,c2,b)-Target) >> i) & 0x00000001) == 0)
  {
search2(x,a1,c1,a2,c2,b,Target,data);
  }

 x = x^(1<<i);

 index--;
 return(count);
}

/* Soek oplossings vir vergelyking (6) en (7) gekombineer */
unsigned int search3(unsigned int x, unsigned int d1, unsigned int a1,
  unsigned int d2, unsigned int a2, unsigned int c,
  unsigned int Target, unsigned int *data)
{
 int i;
 static int index, count;

 i = index;

 if(i==0)
  {
count = 0;
  }


 /* This is where you check the depth of the tree */
 if(i==32)
  {
   data[count] = x;
count++;
return(count);
  }

 index++;

 if(count >= MAX)
  {
   index--;
   return(count);
  }

 if( (((NG(x,d1,a1,d2,a2,c) - Target) >> i) & 0x00000001) == 0)
  {
search3(x,d1,a1,d2,a2,c,Target,data);
  }

  x = x^(1<<i);

 if( (((NG(x,d1,a1,d2,a2,c)-Target) >> i) & 0x00000001) == 0)
  {
search3(x,d1,a1,d2,a2,c,Target,data);
  }
```

```
 x = x^(1<<i);

 index--;
 return(count);
}


unsigned int Rotate(unsigned int X, unsigned int s)
{
   unsigned int temp;

   temp = X;
   X = (X << s) | (temp >> (32-s));
   return(X);
}

unsigned int RotateRight(unsigned int X, unsigned int s)
{
   unsigned int temp;

   temp = X;
   X = (X >> s) | (temp << (32-s));
   return(X);
}

unsigned int F(unsigned int X, unsigned int Y, unsigned int Z)
{
   return((X&Y) | ((~X)&Z));
}

unsigned int G(unsigned int X, unsigned int Y, unsigned int Z)
{
   return((X&Z) | (Y&(~Z)));
}

unsigned int GG(unsigned int x, unsigned int c1, unsigned int d1,
unsigned int c2, unsigned int d2, unsigned int a)
{
 /* a is the difference DeltaA19. x is the basic value */
 return(G(c1,d1,x)-G(c2,d2,x+a));
}

unsigned int GGG(unsigned int x, unsigned int a1, unsigned int c1,
unsigned int a2, unsigned int c2, unsigned int b)
{
 /* b is the difference DeltaB15. x is the basic value */
 return(G(a1,x,c1)-G(a2,x+b,c2));
}

unsigned int NG(unsigned int x, unsigned int d1, unsigned int a1,
unsigned int d2, unsigned int a2, unsigned int c)
{
 /* a is the difference DeltaA19. x is the basic value */
 return(G(x,d1,a1)-G(x+c,d2,a2));
}
```

E.2 SOURCE CODE: SECOND PHASE OF THE ATTACK ON MD5

This section of the appendix contains the C source code used in the second phase of the attack on MD5.

```
/* This is the program written by Hans Dobbertin to implement the attack on
 * the last two rounds of MD5. */

#include <stdio.h>
#include <stdlib.h>

#define UL unsigned long
#define MAX 10000
#define shift(x,i) (UL)(((x)<<(i))^((x)>>(32-(i))))
#define f(x,y,z) ((x)^(y)^(z))
#define F(x) (UL)(f(x+a,b2,c2)-f(x,b1,c1))
#define H(x) (UL)(((b1)^(x+a))+((b2)^(x+a))+((c1)^(x))+((c2)^(x)))

UL a,b1,c1,b2,c2,konst;
int *max,max_test;
UL **feld;
int suche(),suchh(),sucheb();

main(int ac,char *av[])
{
   int rr,kk,nn,jj,ii,inn,out,maxi2;
   int count1,count2,count3,count4,count5,count6;
   UL A0,A1,A2,B0,B1,B2,C0,C1,C2,D0,D1,D2,K,K0,U1,U2,V1,V2,W1,W2,Z1,Z2,anfang;
   UL A2_basic,B2_basic,C2_basic,D2_basic,C,D;
   UL X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,XX,dX;
   UL Y7,Y8,Y9,Y10,Y11,Y12,X,dC10,Y,Z,U,UU,delta;
   UL test0,test1,test2,test3,test4,test5,test6,test7,test8,test9;
   UL eps1,eps2,eps3,eps4,tem,test,test10,test11;
   int phase,erfolg,sh1,sh2,sh3,diff1,diff2,diff3,tem1,tem2;
   UL dA1,dB1,dD1,dC1,AA1,BB1,CC1,DD1,AA2,BB2,CC2,DD2,dXnull;
   UL *feld0,*feld1,*feld2,*feld3,*feld4,*feld5,*feld6;
   int Count1,Count2,Count3,versuch,flag,ind,maxi,ERFOLG;
   int max0=MAX,max1=MAX,max2=MAX,max3=MAX,max4=MAX,max5=MAX,max6=MAX;
   float quotient;

   if(ac!=2)
     {
      fprintf(stderr,"Usage: %s seed\n",av[0]);
      exit(1);
     }


   srand(atoi(av[1]));

   if((feld0=(UL*)malloc(MAX*sizeof(UL)))==NULL ||
      (feld1=(UL*)malloc(MAX*sizeof(UL)))==NULL ||
      (feld2=(UL*)malloc(MAX*sizeof(UL)))==NULL ||
```

```
      (feld3=(UL*)malloc(MAX*sizeof(UL)))==NULL ||
      (feld4=(UL*)malloc(MAX*sizeof(UL)))==NULL ||
      (feld5=(UL*)malloc(MAX*sizeof(UL)))==NULL ||
      (feld6=(UL*)malloc(MAX*sizeof(UL)))==NULL)
     {
      fprintf(stderr,"Nicht genuegend speicher\n");
      exit(1);
     }


   out=0x100;
   inn=0x1000;
   maxi=0x10;
   maxi2=1000;
   ERFOLG=0;
   versuch=0;

   delta=shift(1,9);
   BB1=0;
   B1=-1;
   dB1=BB1-B1;

   test10 =  0x80044000;
   test11 =  0x40040000;
   eps1   = -0x40004000;
   eps2   = -0x80084000;
   eps3   = -0xFFFBFE00;
   eps4   = -0x40000200;

LABEL_A:

   max_test=out;
   erfolg=0;

LABEL_NEU:
   A2=(rand()^shift(rand(),1));
   C2=(rand()^shift(rand(),1));
   C2=C2&0xbffbbdff;
   A2=A2&0xbffbbdff;
   D2=(A2+shift(eps1-eps4,31))^(rand()&rand()&rand()&rand());
   C2=C2^0x40004000;
   D2=D2^0x00000200;
   A2=A2^0x40004200;


   test=f(C2+eps3,D2+eps4,0)-f(C2,D2,0)-test11;
   if(test!=0)
     {
goto LABEL_NEU;
     }

   test=f(C2+eps3,D2+eps4,A2+eps1)-f(C2,D2,A2)-test10;
   if(test!=0)
     {
goto LABEL_NEU;
```

```
      }

    B2=shift(-eps2-1,31)^(rand()&rand()&rand());

    UU=(A2+eps1)^(B2+eps2);
    U =A2^B2;
    dC1=shift(C2-D2+eps3-eps4,16)-shift(C2-D2,16);
    dC1=dC1-((D2+eps4)^UU)+(D2^U);
    C1=(rand()^shift(rand(),1));
    CC1 = C1+dC1;
    dD1 = -(CC1^UU)+(C1^U)+shift(D2-A2+eps4-eps1,21)-shift(D2-A2,21);

    dXnull=(shift(A2-B2+eps1-eps2,28)-shift(A2-B2,28)+1+eps2)&1;
    test=dXnull^((shift(B2+eps2-CC1,9)-shift(B2-C1,9))&1)^(dC1&1);

    if(test==0)
      {
konst=shift(D2-A2+eps4-eps1,21)-shift(D2-A2,21)-dD1;
b2=UU;
c2=0;
b1=U;
c1=0;
a=dC1;
feld=&feld1;
max=&max1;

if(count1=suche())
  {
     for(nn=0; nn<maxi; nn++)
       {
  ind=(unsigned)rand()%count1;
  C1=feld1[ind];
  CC1=C1+dC1;
  if(dXnull==((shift(dC1-dD1,16))&1))
    {
       dX = shift(dC1-dD1,16);
       /*dX = shift(dC1-dD1,16)-shift(1,16);*/
    }
  if(dXnull!=((shift(dC1-dD1,16))&1))
    {
       dX = shift(dC1-dD1,16)+1;
    }

  konst=shift(B2+eps2-CC1,9)-shift(B2-C1,9)-dB1;
  b2=CC1^BB1;
  c2=0;
  b1=C1^B1;
  c1=0;
  a = dX;
  feld=&feld2;
  max=&max2;
  max_test=inn;
  if(count2=suche())
    {
       if(erfolg==0)
```

```
 {
    fprintf(stderr,"** %8d  %8d  %8d",count1,nn,count2);
    fprintf(stderr,"** %8.8X  %8.8X  %8d\n",dC1,dD1,ERFOLG);
    erfolg=1;
 }
       for(kk=0; kk<count2; kk++)
 {
    ind=(unsigned)rand()%count2;
    X=feld2[ind];
    XX=X+dX;
    konst=shift(A2-B2+eps1-eps2,28)-shift(A2-B2,28)-2;
    b1=(B2+eps2)^CC1;
    b2=XX^BB1;
    c1=(-B2-1)^C1;
    c2=(-X-1)^B1;
    a=dD1;
    feld=&feld3;
    max=&max3;
    max_test=inn;
    if(count3=sucheh())
      {
 ERFOLG=ERFOLG+1;
 fprintf(stderr,"LABEL_B %d\n",ERFOLG);
 goto LABEL_B;
      }
 }
       max_test=out;
    }
       }
  }
     }

   goto LABEL_A;

LABEL_B:

   fprintf(stdout,"%d\n",ERFOLG);
   fprintf(stdout,"A2 = 0x%8.8X;\n",A2);
   fprintf(stdout,"B2 = 0x%8.8X;\n",B2);
   fprintf(stdout,"C2 = 0x%8.8X;\n",C2);
   fprintf(stdout,"D2 = 0x%8.8X;\n",D2);
   fprintf(stdout,"C1 = 0x%8.8X;\n",C1);
   fprintf(stdout,"dC1= 0x%8.8X;\n",dC1);
   fprintf(stdout,"dD1= 0x%8.8X;\n\n",dD1);

   erfolg=0;
   flag=0;
   max_test=0x200000;
   konst=shift(D2-A2+eps4-eps1,21)-shift(D2-A2,21)-dD1;
   b2=UU;
   c2=0;
   b1=U;
   c1=0;
   a =dC1;
   feld=&feld1;
```

```
   max=&max1;
   if(count1=suche())
     {
for(nn=0; nn<maxi2; nn++)
  {
     ind=(unsigned)rand()%count1;
     C1=feld1[ind];
     CC1=C1+dC1;
     if(dXnull==((shift(dC1-dD1,16))&1))
       {
  dX = shift(dC1-dD1,16);
  /*dX = shift(dC1-dD1,16)-shift(1,16);*/
       }

     if(dXnull!=((shift(dC1-dD1,16))&1))
       {
  dX = shift(dC1-dD1,16)+1;
       }

     konst=shift(B2+eps2-CC1,9)-shift(B2-C1,9)-dB1;
     b2=CC1^BB1;
     c2=0;
     b1=C1^B1;
     c1=0;
     a=dX;
     feld=&feld2;
     max=&max2;
     if(count2=suche())
       {
  /*fprintf(stderr,"II. ** %8d %8d %8d %8.8X\n",count1,nn,count2,dD1);*/
  for(kk=0; kk<count2; kk++)
    {
       ind=(unsigned)rand()%count2;
       X=feld2[ind];
       XX=X+dX;
       konst=shift(A2-B2+eps1-eps2,28)-shift(A2-B2,28)-2;
       b1=(B2+eps2)^CC1;
       b2=XX^BB1;
       c1=(-B2-1)^C1;
       c2=(-X-1)^B1;
       a=dD1;
       feld=&feld3;
       max=&max3;
       if(count3=sucheh())
 {
    for(rr=0; rr<count3; rr++)
      {
 ind=(unsigned)rand()%count3;
 D1=feld3[ind];
 DD1=D1+dD1;
 test=dX-(shift(CC1-DD1,16)-shift(C1-D1,16));
 if(test==0)
   {
      A1=D1^X^B1;
      AA1=DD1^XX^BB1;
```

```
      dA1=AA1-A1;
      test=(shift(AA1-BB1,28)-shift(A1-B1,28)-1)&1;
      if(test==0)
{
   flag=1;
   konst=shift(DD1-AA1,21)-shift(D1-A1,21);
   b2=AA1;
   c2=BB1;
   b1=A1;
   c1=B1;
   a=0;
   feld=&feld4;
   max=&max4;
   C=rand();
   test = f(AA1,BB1,C)-f(A1,B1,C);
   test = test-shift(DD1-AA1,21)+shift(D1-A1,21);
   if((test&0xffff)==0)
     {
fprintf(stderr,"def = %8.8x %d\n",test,ERFOLG);
     }

   if(count4=sucheb())
     {
test = f(D2+eps4,A2+eps1,B2+eps2)-f(D2,A2,B2);
test = test-shift(C2-D2+eps3-eps4,16)+shift(C2-D2,16)+dC1;
if(test!=0)
  {
     fprintf(stderr,"ERROR1\n");
     exit(1);
  }

test = f(A2+eps1,B2+eps2,CC1)-f(A2,B2,C1);
test = test-shift(D2-A2+eps4-eps1,21)+shift(D2-A2,21)+dD1;
if(test!=0)
  {
     fprintf(stderr,"ERROR2\n");
     exit(1);
  }

test = f(B2+eps2,CC1,DD1)-f(B2,C1,D1);
test = test-shift(A2-B2+eps1-eps2,28)+shift(A2-B2,28)+dA1;
if(test!=0)
  {
     fprintf(stderr,"ERROR3\n");
     exit(1);
  }

test = f(CC1,DD1,AA1)-f(C1,D1,A1);
test = test-shift(B2-CC1+eps2,9)+shift(B2-C1,9)+dB1;
if(test!=0)
  {
     fprintf(stderr,"ERROR4\n");
     exit(1);
  }
```

```
test = f(DD1,AA1,BB1)-f(D1,A1,B1);
test = test-shift(CC1-DD1,16)+shift(C1-D1,16);
if(test!=0)
  {
     fprintf(stderr,"ERROR5\n");
     exit(1);
  }


C=feld4[0];
test = f(AA1,BB1,C)-f(A1,B1,C);
test = test-shift(DD1-AA1,21)+shift(D1-A1,21);
if(test!=0)
  {
     fprintf(stderr,"ERROR6\n");
     exit(1);
  }

D=shift(shift(AA1-BB1,28)-shift(A1-B1,28)-1,31)^C;
test = f(BB1,C,D)-f(B1,C,D);
test = test-shift(AA1-BB1,28)+shift(A1-B1,28);
if(test!=0)
  {
     fprintf(stderr,"ERROR7\n");
  }

fprintf(stderr,"A2 = Ox%8.8X;\n",A2);
fprintf(stderr,"B2 = Ox%8.8X;\n",B2);
fprintf(stderr,"C2 = Ox%8.8X;\n",C2);
fprintf(stderr,"D2 = Ox%8.8X;\n\n",D2);
fprintf(stderr,"eps1= Ox%8.8X;\n",eps1);
fprintf(stderr,"eps2= Ox%8.8X;\n",eps2);
fprintf(stderr,"eps3= Ox%8.8X;\n",eps3);
fprintf(stderr,"eps4= Ox%8.8x;\n",eps4);
fprintf(stderr,"dC1 = Ox%8.8X;\n",dC1);
fprintf(stderr,"dd1 = Ox%8.8X;\n",dD1);
fprintf(stderr," **** gef. ****\n");
fprintf(stderr,"AA1 = Ox%8.8x;\n",AA1);
fprintf(stderr,"A1  = Ox%8.8x;\n",A1);
fprintf(stderr,"BB1 = Ox%8.8X;\n",BB1);
fprintf(stderr,"B1  = Ox%8.8x;\n",B1);
fprintf(stderr,"CC1 = Ox%8.8X;\n",CC1);
fprintf(stderr,"C1  = Ox%8.8X;\n",C1);
fprintf(stderr,"DD1 = Ox%8.8X;\n",DD1);
fprintf(stderr,"D1  = Ox%8.8x;\n",D1);
fprintf(stderr,"C   = Ox%8.8X;\n",C);
fprintf(stderr,"D   = Ox%8.8X;\n",D);

fprintf(stdout,"A2  =   Ox%8.8X;\n",A2);
fprintf(stdout,"B2  =   Ox%8.8X;\n",B2);
fprintf(stdout,"C2  =   Ox%8.8X;\n",C2);
fprintf(stdout,"D2  =   Ox%8.8X;\n",D2);
fprintf(stdout,"eps1= Ox%8.8x;\n",eps1);
fprintf(stdout,"eps2= Ox%8.8X;\n",eps2);
fprintf(stdout,"eps3= Ox%8.8X;\n",eps3);
```

```
fprintf(stdout,"eps4= Ox%8.8X;\n",eps4);
fprintf(stdout,"dC1 = Ox%8.8X;\n",dC1);
fprintf(stdout,"dD1 = Ox%8.8X;\n",dD1);
fprintf(stdout,"**** gef. ****\n");
fprintf(stdout,"AA1 = Ox%8.8X;\n",AA1);
fprintf(stdout,"A1  = Ox%8.8X;\n",A1);
fprintf(stdout,"BB1 = Ox%8.8X;\n",BB1);
fprintf(stdout,"B1  = Ox%8.8X;\n",B1);
fprintf(stdout,"CC1 = Ox%8.8X;\n",CC1);
fprintf(stdout,"C1  = Ox%8.8X;\n",C1);
fprintf(stdout,"DD1 = Ox%8.8X;\n",DD1);
fprintf(stdout,"D1  = Ox%8.8x;\n",D1);
fprintf(stdout,"C   = Ox%8.8x;\n",C);
fprintf(stderr,"D   = Ox%8.8X;\n",D);
exit(1);
      }
}
    }
       }
     }
  }
     }
        }
   }
      }


    if(flag==0)
      {
fprintf(stderr,"nichts gefunden\n");
      }
    if(flag==1)
      {
fprintf(stderr,"nichts gefunden +\n");
      }

    goto LABEL_A;
} /* End Main */


int suche()
{
    int i;
    static int index,count;
    static UL x;

    i=index;

    if(i==0)
      {
count=0;
      }

    if(i==32)
      {
if(count>=*max)
```

```
  {
     (*max)*=2;
     if((*feld=(UL*)realloc(*feld,(*max)*sizeof(UL)))==NULL)
       {
  printf("Nicht genuegend Speicher 1\n");
  exit(1);
       }

  }

(*feld)[count]=x;
count++;
return count;
     }

   index++;

/*   if(count >= MAX)
     {
index--;
return(count);
     }*/


   if((((F(x)-konst)>>i)&1)==0)
     {
suche();
     }

   if(count >= max_test)
     {
index--;
return count;
     }


   x^=1ul<<i;
   if((((F(x)-konst)>>i)&1)==0)
     {
suche();
     }

   x^=1ul<<i;
   index--;
   return count;
}

int sucheh()
{
   int i;
   static int index,count;
   static UL x;

   i=index;
```

```
   if(i==0)
     {
count=0;
     }

   if(i==32)
     {
if(count>=*max)
  {
     (*max)*=2;
     if((*feld=(UL*)realloc(*feld,(*max)*sizeof(UL)))==NULL)
       {
  printf("Nicht genuegend Speicher 2\n");
  exit(1);
       }
      }
(*feld)[count]=x;
count++;
return count;
     }

   index++;

   /*if(count >= MAX)
     {
index--;
return(count);
     }*/

   if((((H(x)-konst)>>i)&1)==0)
     {
sucheh();
     }

   if(count >= max_test)
     {
index--;
return count;
     }


   x=1ul<<i;

   if((((H(x)-konst)>>i)&1) == 0)
     {
sucheh();
     }

   x^=1ul<<i;
   index--;
   return count;
}

int sucheb()
{
```

```
   int i;
   static int index,count;
   static UL x;

   i=index;

   if(i==0)
     {
count=0;
     }

   if(i==32)
     {
if(count>=*max)
  {
     (*max)*=2;
     if((*feld=(UL*)realloc(*feld,(*max)*sizeof(UL)))==NULL)
       {
  printf("Nicht genuegend Speicher 3\n");
  exit(1);
       }

  }

(*feld)[count]=x;
count++;
return count;
     }

   index++;

   /*if(count >= MAX)
     {
index--;
return(count);
     }*/

   if((((F(x)-konst)>>i)&1)==0)
     {
sucheb();
     }

   if(count >= max_test)
     {
index--;
return count;
     }


   x^=1ul<<i;
   if((((F(x)-konst)>>i)&1)==0)
     {
sucheb();
     }
```

```
    x^=1ul<<i;
    index--;
    return count;
}
```

E.3 THIRD PHASE OF THE ATTACK ON MD5

This section of the appendix contains the C source code used in the third phase of the attack on MD5.

```
/* This program is a modification of the original program for connecting the
 * first two stages of the MD5 attack. The modification ensures that the
 * connection established is a valid connection.
 *
 * Original Author: Hans Dobbertin?
 * Modifications  : P.R. Kasselman
 * Date of Modifications: 30 September 1996
 * Filename: md5an13.c */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "libtiming.h"

#define ulong unsigned long
#define shift(x,i) (ulong)((((ulong)x)<<(i))^(((ulong)x)>>(32-(i))))
#define f(x,y,z) ((x)&(y) | (~(x))&(z))
#define g(x,y,z) ((x)&(z) | (~(z))&(y))
#define h(x,y,z) ((x)^(y)^(z))
#define i(x,y,z) ((y)^((x)|(~(z))))

static FILE *f;

/* random number generator from Numerical Recipes in C p. 282 */
#define IM1 2147483563
#define IM2 2147483399
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+(IM1-1)/NTAB)

double ran2(long *idum)
{
    int j;
    long k;
    static long idum2 = 123456789;
    static long iy = 0;
    static long iv[NTAB];
    double temp;
```

```
   /* Initialization */

   if (*idum <= 0)
     {
/* make sure *idum != 0 */
if (*idum == 0)
  {
     *idum = 1;
  }

printf("Random generator initialized with %ld\n", *idum);
fprintf(f, "Random generator initialized with %ld\n", *idum);
idum2 = *idum;
for (j=NTAB+7; j>=0; j--)
  {
     /* idum = (IAllidum) % IM1*/
     k = *idum/IQ1;
     *idum = IA1*(*idum-k*IQ1)-k*IR1;
     if (*idum < 0)
       {
  *idum += IM1;
       }
     if (j < NTAB)
       {
  iv[j] = *idum;
       }
  }

iy = iv[0];
     }

   /* idum = (IA1*idum) % IM1 */

   k = *idum/IQ1;

   *idum = IA1*(*idum-k*IQ1)-k*IR1;

   if (*idum < 0)
     {
*idum += IM1;
     }

   /* idum2 = (IA2*idum) % IM2 */
   k = idum2/IQ2;
   idum2 = IA2*(idum2-k*IQ2)-k*IR2;

   if (idum2 < 0)
     {
idum2 += IM2;
     }

   j = iy/NDIV;
   iy = iv[j] - idum2;
   iv[j] = *idum;
```

```
   if (iy < 1)
     {
iy += IM1-1;
     }

   return (double)iy/IM1;
}

static long randstate;
void srand32(long seed)
{
   if (seed>0)
     {
randstate = -seed;
     } else {
randstate = seed;
     }
}

unsigned long rand32(void)
{
   return (unsigned long)(4294967296.0*ran2(&randstate));
}

main(int ac,char *av[])
{
   ulong A,B,B_basic,C,D;
   ulong A0,B0,C0,D0;
   ulong C31,B31;
   ulong A39,B35,C39,D39;
   ulong A43,B39,D43;
   ulong A35,C35,D35;
   ulong A19,B15,C15,D19;
   ulong A23,B19,C19,D23;
   ulong C27,B27,A31,D31;
   ulong        BB_12,CC_12,C2_34,BB_34;
   ulong A15,D15;
   ulong A27,B23,C23,D27;
   ulong A1,B1,C1,D1;
   ulong At0,Bt0,Ct0,Dt0,At1,Bt1,Ct1,Dt1,At2,Bt2,Ct2,Dt2;
   ulong test0,test1;
   ulong X0,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12,X13,X14,X15,X0_12;
   ulong X8A,X8B;
   ulong konst1,konst2,konst3,konst4,konst5;
   int         gew=32,gew0,k,Versuch,vvv;
   unsigned int iterations34, count1, sum34_vvv;
   unsigned int ConnectionFlag;
   double       t1, t2, time_tot;

   ulong K0=0xd76aa478, K1=0xe8c7b756, K2=0x242070db, K3=0xc1bdceee;
   ulong K4=0xf57c0faf, K5=0x4787c62a, K6=0xa8304613, K7=0xfd469501;
   ulong K8=0x698098d8, K9=0x8b44f7af, K10=0xffff5bb1, K11=0x895cd7be;
   ulong K12=0x6b901122, K13=0xfd987193, K14=0xa679438e, K15=0x49b40821;
   ulong K16=0xf61e2562, K17=0xc040b340, K18=0x265e5a51, K19=0xe9b6c7aa;
```

```
    ulong K20=0xd62f105d, K21=0x02441453, K22=0xd8a1e681, K23=0xe7d3fbc8;
    ulong K24=0x21e1cde6, K25=0xc33707d6, K26=0xf4d50d87, K27=0x455a14ed;
    ulong K28=0xa9e3e905, K29=0xfcefa3f8, K30=0x676f02d9, K31=0x8d2a4c8a;
    ulong K32=0xfffa3942, K33=0x8771f681, K34=0x6d9d6122, K35=0xfde5380c;
    ulong K36=0xa4beea44, K37=0x4bdecfa9, K38=0xf6bb4b60, K39=0xbebfbc70;
    ulong K40=0x289b7ec6, K41=0xeaa127fa, K42=0xd4ef3085, K43=0x04881d05;
    ulong K44=0xd9d4d039, K45=0xe6db99e5, K46=0x1fa27cf8, K47=0xc4ac5665;
    ulong K48=0xf4292244, K49=0x432aff97, K50=0xab9423a7, K51=0xfc93a039;
    ulong K52=0x655b59c3, K53=0x8f0ccc92, K54=0xffeff47d, K55=0x85845dd1;

    int s0 = 7,s1 =12,s2 =17,s3 =22;
    int s4 = 7,s5 =12,s6 =17,s7 =22;
    int s8 = 7,s9 =12,s10=17,s11=22;
    int s12= 7,s13=12,s14=17,s15=22;
    int s16= 5,s17= 9,s18=14,s19=20;
    int s20= 5,s21= 9,s22=14,s23=20;
    int s24= 5,s25= 9,s26=14,s27=20;
    int s28= 5,s29= 9,s30=14,s31=20;
    int s32= 4,s33=11,s34=16,s35=23;
    int s36= 4,s37=11,s38=16,s39=23;
    int s40= 4,s41=11,s42=16,s43=23;
    int s44= 4,s45=11,s46=16,s47=23;
    int s48= 6,s49=10,s50=15,s51=21;
    int s52= 6,s53=10,s54=15,s55=21;

    if(ac!=2)
      {
fprintf(stderr,"Usage: %s seed\n",av[0]);
exit(1);
      }


    srand32(atol(av[1]));
    f = fopen("md5_dob.dat", "w");

    /* Daten    Runde 12 */
    A15 = 0XFA08A191;
    D15 = 0xFE010000;
    A19 = 0x3079FC64;
    B15 = 0x7C091F7C;
    C15 = 0xFE00FFFF;
    D19 = 0x00000000;
    B19 = 0x007DE57C;
    C19 = 0xA83AE412;
    A23 = 0x001003E0;
    A23 = 0x20200380;
    A23 = 0x20001380;

    A15  = 0x45A9B039;
    D15  = 0xFE010000;
    C15  = 0xFE00FFFF;
    B15  = 0x681702E8;
    A19  = 0xA27F00A0;
    D19  = 0x00000000;
    C19  = 0x5E4D2843;
```

```
   B19  = 0xCC7E11F4;
   A23  = 0x8A14C72D;

   /* Daten Runnde 34 */
   A43 = 0x7DC2498C;
   B39 = 0x40040E86;

   /* These chaining variables are not used in this program and may therefore
    * be ommitted */
   /*   C2_34 = 0x12F37166;
   D43 = 0xBDC18273; */
   A39 = 0x579DA695;
   B35 = 0xFFFFFFFF;
   C39 = 0xC783FA3D;
   D39 = 0x8DE34058;
   C35 = 0x00028800;
   D35 = 0x3420f162;

   /* Determine message words from the first part of the attack */
   X1 = shift(A19-B15,32-s16)-K16-A15-g(B15,C15,D15);
   X6 = shift(D19-A19,32-s17)-K17-D15-g(A19,B15,C15);
   X11 = shift(C19-D19,32-s18)-K18-C15-g(D19,A19,B15);
   X0 = shift(B19-C19,32-s19)-K19-B15-g(C19,D19,A19);
   X5 = shift(A23-B19,32-s20)-K20-A19-g(B19,C19,D19);

   /* Determine message words from the second part of the attack */
   X4 = shift(D39-A39,32-s37)-K37-D35-h(A39,B35,C35);
   X7 = shift(C39-D39,32-s38)-K38-C35-h(D39,A39,B35);
   X10= shift(B39-C39,32-s39)-K39-B35-h(C39,D39,A39);
   X13= shift(A43-B39,32-s40)-K40-A39-h(B39,C39,D39);

   /* This is a constant to be used later on (when confirming a collision) */
   BB_12 = shift(B15-C15,32-s15)-K15-f(C15,D15,A15);
   /* This is a constant to be used later on for confirmation of collision-
s */
   CC_12 = shift(C15-D15,32-s14)-K14;

   /* Calculate chaining variables to be used when establishing a connection */
   D23 = shift(D19+g(A23,B19,C19)+X10+K21,s21)+A23;
   A35 = shift(A39-B35,32-s36)-K36-X1-h(B35,C35,D35);

   /* This constant is used at a later stage (X14 is not included in
      this expression) */
   BB_34 = shift(B35-C35,32-s35)-K35-h(C35,D35,A35);

   D43 = shift(D39+h(A43,B39,C39)+X0 +K41,s41)+A43;

   /* Data of inner collisions 1-2 and 3-4 should be chosen such that X0
    is the same in both inner collisions.  This was not the case in the first
    version of the attack, and the reason why K19 had to be adapted. */

   sum34_vvv = 0;
   iterations34 = 50;
   time_tot = 0;
```

```
   /* The purpose of this loop is to find the average time required
    * to find a collision */

/*   for (count1=0; count1<iterations34; count1++) */
     {
/* Number of inner collisions 1-2/connections to find a collision */
vvv = 0;

printf("\nSearch %u started\n", count1+1);
fflush(stdout);
fprintf(f, "State random generator = %ld\n", randstate);

while(1)
  {
     /* Look for a collision */
     do
       {
  /* Look for inner collision in round 1-2 */
  X15 = rand32();

  /* Determine the last four chaining variables that will
   * result in a internal collision for the first two rounds */

  C23 = shift(C19+g(D23,A23,B19)+X15+K22,s22)+D23;
  B23 = shift(B19+g(C23,D23,A23)+X4 + K23,s23)+C23;

  /*A27 = 0xffffffff;*/
  A27 = B23;
  X9 = shift(A27-B23,32-s24)-g(B23,C23,D23)-A23-K24;
  D27 = 0xffffffff;
  X14 = shift(D27-A27,32-s25)-g(A27,B23,C23)-D23-K25;

  /* Check whether inner collision 1-2 */
  A = A15;
  B = BB_12-X15;
  C = CC_12-X14-f(D15,A15,B);
  D = D15;
  C = shift(C+f(D,A,B)+X14+((ulong)1<<9)+K14,s14)+D;
  B = shift(B+f(C,D,A)+X15+K15,s15)+C;
  A = shift(A+g(B,C,D)+X1 +K16,s16)+B;
  D = shift(D+g(A,B,C)+X6 +K17,s17)+A;
  C = shift(C+g(D,A,B)+X11+K18,s18)+D;
  B = shift(B+g(C,D,A)+X0 +K19,s19)+C;
  A = shift(A+g(B,C,D)+X5 +K20,s20)+B;
  D = shift(D+g(A,B,C)+X10+K21,s21)+A;
  C = shift(C+g(D,A,B)+X15+K22,s22)+D;
  B = shift(B+g(C,D,A)+X4 +K23,s23)+C;
  A = shift(A+g(B,C,D)+X9 +K24,s24)+B;
  D = shift(D+g(A,B,C)+X14+((ulong)1<<9)+K25,s25)+A;

  /*if( A==A27 && B==B23 && C==C23 && D==D27 )
    {
       printf("%8.8X %8.8X %8.8X %8.8X\n", A-A27, B-B23, C-C23, D-D27);
    }*/
```

```
  /* Proceed if an internal collision for 1-2 was found */
  } while( A!=A27 || B!=B23 || C!=C23 || D!=D27 );

     /* begin connection */

     D35 = D35;
     A35 = A35;
     B31 = BB_34-X14;
     C31 = shift(C35-D35,32-s34)-h(D35,A35,B35)-X11-K34;

     Versuch=0;

     /* These are constants for the while loop that follows.
      * Pre-computation saves some time */

     B_basic = rand32();
     konst1 = A27+X13+K28;
     konst2 = shift(A35-B31,32-s32)-X5-K32;
     konst3 = B31^C31;
     konst4 = -X7-K30;

     ConnectionFlag = 0;
     while(ConnectionFlag == 0)
       {
/* Looking for a connection */

  B27 = B_basic^((ulong)1 << (rand32()&0x1f));
  A31 = shift(B27+konst1,s28)+B27;
  D31 = (konst2-A31)^konst3;
  C27 = shift(C31-D31,32-s30)-g(D31,A31,B27)+konst4;

  X8A = shift(B27-C27,32-s27)-B23-g(C,D27,A27)-K27;
  X8B = shift(D35-A35,32-s33)-D31-h(A35,B31,C31)-K33;
  test0 = X8B-X8A;

  if (test0==0)
    {
       /* you are really lucky */
       X12 = shift(B31-C31,32-s31)-B27-g(C31,D31,A31)-K31;
       X2 = shift(D31-A31,32-s29)-D27-g(A31,B27,C27)-K29;
       X3 = shift(C27-D27,32-s26)-C23-g(D27,A27,B23)-K26;
       X8 = shift(B27-C27,32-s27)-B23-g(C27,D27,A27)-K27;

       /*printf("Lucky You\n")*/
       /* Immediatle exit from the while loop */
       break;
    }

  /* Compute the Hamming Distance */
  /* Perform an iterative aproach */
  for (gew0=0, test1=test0; test1; test1>>=1)
    gew0 += test1&1;
  Versuch++;
  if (Versuch>30)     /*originally 150*/
    {
```

```
        /* Effectively restart the process */
        Versuch = 0;
        gew = 32;
        B_basic = rand32();

        /* Restart from the beginning of the loop */
        continue;
    }

  /* Continuous approximation techniques */
  if (gew0-2<gew) /* originally gew0-1 */
    {
        gew = gew0;
        if (gew>11) /* originally 7 */
 {
    /* If X8A and X8B are closer to each other than
     * before retain the value of B as the new
     * basic value */

    B_basic =B27;

    /* Restart from the beginning of the loop */
    continue;
 }

/* Verbesserung */
/* Check whether B23 is changed by same amount as CO */
        /*if( (C23 & A23) != ((C23-test0) & A23))
 {
    break;
 }*/

        /* Change C23 and consequently recalculate B23 */
        C23 = C23-test0;
        B23 = shift(B19+g(C23,D23,A23)+X4 + K23,s23)+C23;

        /* Recompute D27 */
        D27 = shift(D23+g(A27,B23,C23)+X14+K25,s25)+A27;

        /* A, B, C, D not changed */
        /* Check whether A still only depends on B and A27 */
if( g(B27,C27,D27) != B27)
 {
    break;
 }

        /* Sanity Check */
        X8A = shift(B27-C27,32-s27)-B23-g(C27,D27,A27)-K27;
        X8B = shift(D35-A35,32-s33)-D31-h(A35,B31,C31)-K33;

        if(test0 == 0)
 {
    /* Reset the condition and determine if a collision
     * for both round1 1-2 and 3-4 holds */
```

```
    test0 = 1;

    X15 = shift(C23-D23,32-s22)-g(D23,A23,B19)-C19-K22;
    X9 = shift(A27-B23,32-s24)-g(B23,C23,D23)-A23-K24;

    /* Good Fiddling */
    X12 = shift(B31-C31,32-s31)-B27-g(C31,D31,A31)-K31;
    X2 = shift(D31-A31,32-s29)-D27-g(A31,B27,C27)-K29;
    X3 = shift(C27-D27,32-s26)-C23-g(D27,A27,B23)-K26;
    X8 = shift(B27-C27,32-s27)-B23-g(C27,D27,A27)-K27;

    /*printf("Good Fiddle\n");*/

    /* check whether inner collision 1-2 holds */
    A = A15;
    B = BB_12-X15;
    C = CC_12-X14-f(D15,A15,B);
    D = D15;
    C = shift(C+f(D,A,B)+X14+((ulong)1<<9)+K14,s14)+D;
    B = shift(B+f(C,D,A)+X15+K15,s15)+C;
    A = shift(A+g(B,C,D)+X1 +K16,s16)+B;
    D = shift(D+g(A,B,C)+X6 +K17,s17)+A;
    C = shift(C+g(D,A,B)+X11+K18,s18)+D;
    B = shift(B+g(C,D,A)+X0 +K19,s19)+C;
    A = shift(A+g(B,C,D)+X5 +K20,s20)+B;
    D = shift(D+g(A,B,C)+X10+K21,s21)+A;
    C = shift(C+g(D,A,B)+X15+K22,s22)+D;
    B = shift(B+g(C,D,A)+X4 +K23,s23)+C;
    A = shift(A+g(B,C,D)+X9 +K24,s24)+B;
    D = shift(D+g(A,B,C)+X14+((ulong)1<<9)+K25,s25)+A;

    if( A!=A27 || B!=B23 || C!=C23 || D!=D27 )
      {
/* If an inner collision does not hold
 * restart by finding a new inner collision
 * for rounds1-2 */
break;
      }


    /* Check Propagation */
    C = shift(C+g(D,A,B)+X3+K26,s26)+D;
    B = shift(B+g(C,D,A)+X8 +K27,s27)+C;
    A = shift(A+g(B,C,D)+X13 +K28,s28)+B;
    D = shift(D+g(A,B,C)+X2+K29,s29)+A;
    C = shift(C+g(D,A,B)+X7+K30,s30)+D;
    B = shift(B+g(C,D,A)+X12 +K31,s31)+C;
    A = shift(A+h(B,C,D)+X5 +K32,s32)+B;
    D = shift(D+h(A,B,C)+X8+K33,s33)+A;

    /* Verify the existence of a connection */
    if( A!=A35 || B!=B31 || C!=C31 || D!=D35 )
      {
break;
      }
```

```
    test0 = 0;
    break;
}
    }
       }

     if(test0!=0)
       {
/* The Verbesserung failed, so start with a new inner
 * collision*/
continue;
       }

    /* connection found */
    vvv++;
    /* end connect */

    if (!(vvv%20))
      {
printf("Connection and inner collision 1-2 %d found\r", vvv);
fflush(stdout);
      }

    /* Check whether inner collision 3-4 */

    /* Daten Runde 34 */
    C = C39;
    B = B39;
    A = A43;
    D = D43;
    C = shift(C+h(D,A,B)+X3 +K42,s42)+D;
    B = shift(B+h(C,D,A)+X6 +K43,s43)+C;
    A = shift(A+h(B,C,D)+X9 +K44,s44)+B;
    D = shift(D+h(A,B,C)+X12+K45,s45)+A;
    C = shift(C+h(D,A,B)+X15+K46,s46)+D;
    B = shift(B+h(C,D,A)+X2 +K47,s47)+C;
    A = shift(A+i(B,C,D)+X0 +K48,s48)+B;
    D = shift(D+i(A,B,C)+X7 +K49,s49)+A;
    C = shift(C+i(D,A,B)+X14+K50,s50)+D;
    A0 = A;
    B0 = B;
    C0 = C;
    D0 = D;

    B = B31;
    A = A35;
    D = D35;
    C = C35;
    B = shift(B+h(C,D,A)+X14+((ulong)1<<9)+K35,s35)+C;
    A = shift(A+h(B,C,D)+X1 +K36,s36)+B;
    D = shift(D+h(A,B,C)+X4 +K37,s37)+A;
    C = shift(C+h(D,A,B)+X7 +K38,s38)+D;
    B = shift(B+h(C,D,A)+X10+K39,s39)+C;
    A = shift(A+h(B,C,D)+X13+K40,s40)+B;
```

```
   D = shift(D+h(A,B,C)+X0 +K41,s41)+A;
   C = shift(C+h(D,A,B)+X3 +K42,s42)+D;
   B = shift(B+h(C,D,A)+X6 +K43,s43)+C;
   A = shift(A+h(B,C,D)+X9 +K44,s44)+B;
   D = shift(D+h(A,B,C)+X12+K45,s45)+A;
   C = shift(C+h(D,A,B)+X15+K46,s46)+D;
   B = shift(B+h(C,D,A)+X2 +K47,s47)+C;
   A = shift(A+i(B,C,D)+X0 +K48,s48)+B;
   D = shift(D+i(A,B,C)+X7 +K49,s49)+A;
   C = shift(C+i(D,A,B)+X14+((ulong)1<<9)+K50,s50)+D;
   A ^= A0;
   B ^= B0;
   C ^= C0;
   D ^= D0;

   if(B==0)
     {
fprintf(f,"%8.8X %8.8X %8.8X %8.8X %5d\n",A,B,C,D,vvv);
fflush(f);
printf("%5d: %8.8X %8.8X %8.8X %8.8X\n", vvv, A, B, C, D);
fflush(stdout);
     }

   if( A==0 && B==0 && C==0 && D==0)
     {
/* Step backwards through round 1 to find IV */
A = A15;
B = B15;

C = C15;
D = D15;
B = shift(B-C,32-s15)-K15-X15-f(C,D,A);
C = shift(C-D,32-s14)-K14-X14-f(D,A,B);
D = shift(D-A,32-s13)-K13-X13-f(A,B,C);
A = shift(A-B,32-s12)-K12-X12-f(B,C,D);
B = shift(B-C,32-s11)-K11-X11-f(C,D,A);
C = shift(C-D,32-s10)-K10-X10-f(D,A,B);
D = shift(D-A,32-s9)-K9-X9-f(A,B,C);
A = shift(A-B,32-s8)-K8-X8-f(B,C,D);
B = shift(B-C,32-s7)-K7-X7-f(C,D,A);
C = shift(C-D,32-s6)-K6-X6-f(D,A,B);
D = shift(D-A,32-s5)-K5-X5-f(A,B,C);
A = shift(A-B,32-s4)-K4-X4-f(B,C,D);
B = shift(B-C,32-s3)-K3-X3-f(C,D,A);
C = shift(C-D,32-s2)-K2-X2-f(D,A,B);
D = shift(D-A,32-s1)-K1-X1-f(A,B,C);
A = shift(A-B,32-s0)-K0-X0-f(B,C,D);

fprintf(f,"**** %d Versuche ****\n", vvv);
fprintf(f,"**** Collision %u for MDS compress ****\n", count1+1);
fprintf(f,"**** IV = (%8.8X,%8.8X,%8.8X,%8.8X) ****\n",A,B,C,D);
fprintf(f,"***1 X14' = X14+2^9 ****\n");

fprintf(f,"X0 = 0x%8.8x;\n",X0);
fprintf(f,"X1 = 0x%8.8X;\n",X1);
```

```
  fprintf(f,"X2 = 0x%8.8X;\n",X2);
  fprintf(f,"X3 = 0x%8.8X;\n",X3);
  fprintf(f,"X4 = 0x%8.8X;\n",X4);
  fprintf(f,"X5 = 0x%8.8X;\n",X5);
  fprintf(f,"X6 = 0x%8.8X;\n",X6);
  fprintf(f,"X7 = 0x%8.8x;\n",X7);
  fprintf(f,"X8 = 0x%8.8X;\n",X8);
  fprintf(f,"X9 = 0x%8.8X;\n",X9);
  fprintf(f,"X10= 0x%8.8X;\n",X10);
  fprintf(f,"X11= 0x%8.8X;\n",X11);
  fprintf(f,"X12= 0x%8.8X;\n",X12);
  fprintf(f,"X13= 0x%8.8X;\n",X13);
  fprintf(f,"X14= 0x%8.8X;\n",X14);
  fprintf(f,"X15=       0x%8.8X;\n",X15);
  fflush(f);

  /* Break from the while(1) loop if a collision for 3-4 was
   * found */
  break;
       }

  }


/* solution found */

sum34_vvv += vvv;

fprintf(f, "\ncollision %u found after %d trials\n", count1+1, vvv);
fflush(f);
printf("\nCollision %u found after %d trials\n", count1+1, vvv);
fflush(stdout);
     }

   fprintf(f, "Mean number ic-12 for a ic-34: %u\n", sum34_vvv/iterations34);
   fprintf(f, "Mean time for a collision: %f seconds\n", time_tot/iterations34);
   fclose(f);

   PrintUserTime();

   return 0;

 }
```

APPENDIX F: SOURCE CODE: COLLISIONS FOR FIRST ROUND OF SHA

This Appendix contains an implementation of the attack on the first round of SHA. The implementation is written in ANSI-C.

```
/* This program is used to investigate the effect of the message expansion
 * algorithm used in SHA (not SHA-1). The results obtained from the analysis
 * in sha01 and sha02 is verified. In particular it is verified if a
 * collision can be obtained for the first round of SHA.
 *
 * The difference pattern has a defining length of 6.
 * The pattern is:  11 12 13 14 15 16
 *                   1  1  1  0  0  1
 *
 * This program extends the results in sha06.c. Specifically a message is
 * constructed which results in a collision after one round of SHA.
 *
 * Date: 14/11/97
 * Author: P.R. Kasselman
 * Filename: sha07.c */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

unsigned int Rotate(unsigned int X, unsigned int s);
unsigned int RotateRight(unsigned int X, unsigned int s);
void PrintBin(unsigned int j);
unsigned int DefLen(unsigned int j);
unsigned int SHA_F1(unsigned int B, unsigned int C, unsigned int D);
void UpdateChain(unsigned int Temp, unsigned int *A,
 unsigned int *B, unsigned int *C,
 unsigned int *D, unsigned int *E, int i);
void ReverseUpdateChain(unsigned int Temp, unsigned int *A,
unsigned int *B, unsigned int *C,
unsigned int *D, unsigned int *E, int i);

#define K1 0x5A827999

int main()
{
   unsigned int i,j;
   unsigned int Temp1, Temp2, TempInt, Stop, Iteration;
   unsigned int M1[80], M2[80];
   unsigned int A[80], B[80], C[80], D[80], E[80];
   unsigned int At[80], Bt[80], Ct[80], Dt[80], Et[80];
   time_t TheTime;

   /* Seed Random number generator */
   TheTime = time(NULL);
   srandom(TheTime);

   /* Initialise Chaining varaibles */
   Temp1 = 0;
   Temp2 = 0;
```

```
   A[0] = 0x67452301;
   B[0] = 0xEFCDAB89;
   C[0] = 0x98BADCFE;
   D[0] = 0x10325476;
   E[0] = 0xC3D2E1F0;

   At[0] = 0x67452301;
   Bt[0] = 0xEFCDAB89;
   Ct[0] = 0x98BADCFE;
   Dt[0] = 0x10325476;
   Et[0] = 0xC3D2E1F0;

   for(i=0; i<4; i++)
     {
UpdateChain(Temp1, A, B, C, D, E, i);
UpdateChain(Temp2, At, Bt, Ct, Dt, Et, i);
     }

   /* Determine if collision holds */

   Stop = 1;
   Iteration = 0;

   while(Stop != 0)
     {
/* Specify required differences in messages */

for(i=11; i<14; i++)
  {
     M1[i] = (random()^Rotate(random(),1)) & 0xfffffffe;
     M2[i] = M1[i];
  }

M1[11] = M1[11] | 0x00000001;
M2[12] = M2[12] | 0x00000001;
M2[13] = M2[13] | 0x00000001;

M1[i] = (random()^Rotate(random(),1)) & 0xfffffffe;
M2[i] = M1[i];

printf("%8.8X %8.8X\n", M1[11], M2[11]);
printf("%8.8X %8.8X\n", M1[12], M2[12]);
printf("%8.8X %8.8X\n", M1[13], M2[13]);

/* Initialise the chaining variables */

printf("\nConfirmation of Collision\n");

/* Step 1 */

Temp1 = 0x00000000;
Temp2 = 0xffffffff;

A[11] = random() ^ Rotate(random(),1);
```

```
B[11] = A[11] + RotateRight(1,30);
C[11] = -1-Rotate(0,5)-SHA_F1(Temp1,Rotate(A[11],30),Rotate(B[11],30))-K1-
M1[13];
D[11] = 0-Rotate(Temp1,5)-SHA_F1(A[11],Rotate(B[11],30),C[11])-K1- M1[12];
E[11] = Temp1-Rotate(A[11],5)-SHA_F1(B[11],C[11],D[11])-K1-M1[11];

At[11] = A[11];
Bt[11] = B[11];
Ct[11] = -1-Rotate(0,5)-SHA_F1(Temp2,Rotate(At[11],30),Rotate(Bt[11],30))-
K1-M2[13];
Dt[11] = 0 - Rotate(Temp2,5) - SHA_F1(At[11],Rotate(Bt[11],30),Ct[11]) - K1 -
M2[12];
Et[11] = Temp2-Rotate(At[11],5)-SHA_F1(Bt[11],Ct[11],Dt[11])-K1-M2[11];

printf("Temp1 - Temp2  - (M1[11] - M2[11]) = %8.8X\n",
       Temp1-Temp2 - (M1[11] - M2[11]));

UpdateChain(Temp1, A, B, C, D, E, 11);
UpdateChain(Temp2, At, Bt, Ct, Dt, Et, 11);

/* Step 2 */

printf("A[12] - At[12] + (M1[12] - M2[12]) = %8.8X\n",
       A[12]-At[12]+(M1[12]-M2[12]));

Temp1 = 0x00000000;
Temp2 = 0x00000000;

UpdateChain(Temp1, A, B, C, D, E, 12);
UpdateChain(Temp2, At, Bt, Ct, Dt, Et, 12);

/* Step 3 */

TempInt = SHA_F1(B[13],C[13],D[13]) - SHA_F1(Bt[13],Ct[13],Dt[13]);

printf("F1(B[13],C[13],D[13])-F1(Bt[13],Ct[13],Dt[13])+(M1[13]-M2[13]) = %8.8X\n",
       TempInt + M1[13] - M2[13]);

Temp1 = 0xffffffff;
Temp2 = 0xffffffff;

UpdateChain(Temp1, A, B, C, D, E, 13);
UpdateChain(Temp2, At, Bt, Ct, Dt, Et, 13);

/* Step 4 */

TempInt = SHA_F1(B[14],C[14],D[14]) - SHA_F1(Bt[14],Ct[14],Dt[14]);

printf("F1(B[14],C[14],D[14])-F1(Bt[14],Ct[14],Dt[14])+(M1[14]-M2[14]) = %8.8X\n",
       TempInt + M1[14] - M2[14]);

Temp1 = random() ^ Rotate(random(),1);
Temp2 = Temp1;

UpdateChain(Temp1, A, B, C, D, E, 14);
```

```
UpdateChain(Temp2, At, Bt, Ct, Dt, Et, 14);

/* Step 5 */

TempInt = SHA_F1(B[15],C[15],D[15]) - SHA_F1(Bt[15],Ct[15],Dt[15]);

printf("F1(B[15],C[15],D[15])-F1(Bt[15],Ct[15],Dt[15])+(M1[15]-M2[15]) = %8.8X\n",
       TempInt + M1[15] - M2[15]);

Temp1 = random() ^ Rotate(random(),1);
Temp2 = Temp1;

UpdateChain(Temp1, A, B, C, D, E, 15);
UpdateChain(Temp2, At, Bt, Ct, Dt, Et, 15);

/* Step 6 */

TempInt = E[16] - Et[16];

printf("E[16]-Et[16]+(M1[16]-M2[16]) = %8.8X\n",
       TempInt + M1[16] - M2[16]);

/* Work forward */

for(i=16; i<20; i++)
  {
     Temp1 = random() ^ Rotate(random(),1);
     Temp2 = Temp1;

     UpdateChain(Temp1, A, B, C, D, E, i);
     UpdateChain(Temp2, At, Bt, Ct, Dt, Et, i);
  }

/* Work back to meet the initial values */

for(i=10; i>=5; i--)
  {
     Temp1 = random()^Rotate(random(),1);
     Temp2 = Temp1;

     ReverseUpdateChain(Temp1, A, B, C, D, E, i+1);
     ReverseUpdateChain(Temp2, At, Bt, Ct, Dt, Et, i+1);
  }

for(i=4; i>=1; i--)
  {
     ReverseUpdateChain(E[i], A, B, C, D, E, i+1);
     ReverseUpdateChain(Et[i], At, Bt, Ct, Dt, Et, i+1);
  }

/* Reconstruct message */
for(i=0; i<16; i++)
  {
     M1[i] = A[i+1]-Rotate(A[i],5)-SHA_F1(B[i],C[i],D[i])-E[i]-K1;
     M2[i] = At[i+1]-Rotate(At[i],5)-SHA_F1(Bt[i],Ct[i],Dt[i])-Et[i]-K1;
```

```
  }

/* Expand Message */
for(i=16; i<21; i++)
  {
     M1[i] = (M1[i-3] ^ M1[i-8] ^ M1[i-14] ^ M1[i-16]);
     M2[i] = (M2[i-3] ^ M2[i-8] ^ M2[i-14] ^ M2[i-16]);
  }

/* Determine the common hash value */
for(i=0; i<20; i++)
  {
     Temp1 = Rotate(A[i],5)+SHA_F1(B[i],C[i],D[i])+E[i]+M1[i]+K1;
     UpdateChain(Temp1, A, B, C, D, E, i);

     Temp2 = Rotate(At[i],5)+SHA_F1(Bt[i],Ct[i],Dt[i])+Et[i]+M2[i]+K1;
     UpdateChain(Temp2, At, Bt, Ct, Dt, Et, i);
  }

i--;
Stop = (A[i]^At[i])+(B[i]^Bt[i])+(C[i]^Ct[i])+(D[i]^Dt[i])+(E[i]^Et[i]);
Iteration++;
     }

   printf("\nChaining Varaibles\n");
   printf("\tA\tB\tC\tD\tE\n");
   for(i=0; i<20; i++)
     {
printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X\n",i,A[i],B[i],C[i],D[i],E[i]);
     }

   printf("\tAt\tBt\tCt\tDt\tEt\n");
   for(i=0; i<20; i++)
     {
printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X\n",i,At[i],Bt[i],Ct[i],Dt[i],Et[i]);
     }

   printf("\nDifference Between Messages\n");
   for(i=0; i<20; i++)
     {
printf("M1[%d] - M2[%d] = %8.8X\n", i, i, M1[i] - M2[i]);
     }

   printf("\nCollision Message\n");
   for(i=0; i<20; i++)
     {
printf("M1[%d] = %8.8X \t M2[%d] = %8.8X\n", i, M1[i], i, M2[i]);
     }

   i = 19;
   printf("\nCommon Hash Value for First Round\n");
   printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X\n",i,A[i],B[i],C[i],D[i],E[i]);
   printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X\n",i,At[i],Bt[i],Ct[i],Dt[i],Et[i]);

   printf("\nDifference in hash values\n");
```

```
   printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X\n",
  i,A[i]^At[i],B[i]^Bt[i],C[i]^Ct[i],D[i]^Dt[i],E[i]^Et[i]);

   printf("Number of Iterations : %d\n", Iteration);
   return(0);
}

   unsigned int SHA_F1(unsigned int B, unsigned int C, unsigned int D)
     {
   return((B&C) | (~B&D));
}

unsigned int Rotate(unsigned int X, unsigned int s)
{
   unsigned int temp;

   temp = X;
   X = (X << s) | (temp >> (32-s));
   return(X);
}

unsigned int DefLen(unsigned int j)
{
   unsigned int i, TempInt;

   for(i=0; i<32; i++)
     {
if(((j >> (31-i)) & 0x00000001) == 1)
  {
     TempInt = i;
     break;
  }
     }

   for(i=31; i>=0; i--)
     {
if(((j >> (31-i)) & 0x00000001) == 1)
  {
     TempInt = i-TempInt;
     break;
  }
     }

   return(TempInt);
}

void PrintBin(unsigned int j)
{
   unsigned int i;

   for(i=0; i<32; i++)
     {
printf("%d", (j >> (31-i)) & 0x00000001);
     }
```

```
}

void UpdateChain(unsigned int Temp, unsigned int *A,
 unsigned int *B, unsigned int *C,
 unsigned int *D, unsigned int *E, int i)
{
   E[i+1] = D[i];
   D[i+1] = C[i];
   C[i+1] = Rotate(B[i],30);
   B[i+1] = A[i];
   A[i+1] = Temp;
}

void ReverseUpdateChain(unsigned int Temp, unsigned int *A,
unsigned int *B, unsigned int *C,
unsigned int *D, unsigned int *E, int i)
{
   A[i-1] = B[i];
   B[i-1] = RotateRight(C[i],30);
   C[i-1] = D[i];
   D[i-1] = E[i];
   E[i-1] = Temp;
}

unsigned int RotateRight(unsigned int X, unsigned int s)
{
   unsigned int temp;

   temp = X;
   X = (X >> s) | (temp << (32-s));
   return(X);
}
```

APPENDIX G: SOURCE CODE: IMPLEMENTATION OF HAVAL ATTACK

This appendix contains an implementation of the attack on HAVAL as described in Chapter 10.

```
/*
 * The analysis for the last two rounds of HAVAL is concidered.
 *
 * This program verifies the existence of collisions for the last two
 * rounds of three round HAVAL.
 *
 * Author: P.R. Kasselman
 * Date: 24/04/1999
 * Filename: haval29.c */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

unsigned int Rotate(unsigned int X, unsigned int s);
unsigned int RotateRight(unsigned int X, unsigned int s);

unsigned int FF2(unsigned int B, unsigned int C, unsigned int D,
       unsigned int E, unsigned int F, unsigned int G,
       unsigned int H);

unsigned int FF3(unsigned int B, unsigned int C, unsigned int D,
       unsigned int E, unsigned int F, unsigned int G,
       unsigned int H);

unsigned int InverseStep(unsigned int H2, unsigned int A1, unsigned int B1,
 unsigned int C1, unsigned int D1, unsigned int E1,
 unsigned int F1, unsigned int G1, unsigned int H1,
 unsigned int K);

/*#define MAX 0x00100000*/
#define MAX 100000
#define MAX_LIMIT 1000

#define RFACT    0
#define RFACT7   7
#define RFACT11 11

#define AA    0
#define BB    1
#define CC    2
#define DD    3
#define EE    4
#define FF    5
#define GG    6
#define HH    7

int main()
{
   char TempChar;
```

```
    unsigned int i, j;
    unsigned int Test, Count1;
    unsigned int DeltaW19, W19, Wt19;
    unsigned int DeltaH56;
    unsigned int Ht56, H56;
    unsigned int Bt62, B62, Ct61, C61, Dt60, D60;
    unsigned int Et59, E59, Ft58, F58, Gt57, G57;
    unsigned int C62, D62, E62, F62, G62, H62;
    unsigned int B61, D61, E61, F61, G61, H61;
    unsigned int B60, C60, E60, F60, G60, H60;
    unsigned int B59, C59, D59, F59, G59, H59;
    unsigned int B58, C58, D58, E58, G58, H58;
    unsigned int B57, C57, D57, E57, F57, H57;
    unsigned int B56, C56, D56, E56, F56, G56;
    unsigned int DataH56[MAX];
    unsigned int Chain[8][96];
    unsigned int W[32], Wt[32];
    unsigned int K1[32] = {0x452821E6L, 0x38D01377L, 0xBE5466CFL, 0x34E90C6CL,
       0xC0AC29B7L, 0xC97C50DDL, 0x3F84D5B5L, 0xB5470917L, 0x9216D5D9L,
       0x8979FB1BL, 0xD1310BA6L, 0x98DFB5ACL, 0x2FFD72DBL, 0xD01ADFB7L,
       0xB8E1AFEDL, 0x6A267E96L, 0xBA7C9045L, 0xF12C7F99L, 0x24A19947L,
       0xB3916CF7L, 0x0801F2E2L, 0x858EFC16L, 0x636920D8L, 0x71574E69L,
       0xA458FEA3L, 0xF4933D7EL, 0x0D95748FL, 0x728EB658L, 0x718BCD58L,
       0x82154AEEL, 0x7B54A41DL, 0xC25A59B5L};
    unsigned int K2[32] = {0x9C30D539L, 0x2AF26013L, 0xC5D1B023L, 0x286085F0L,
       0xCA417918L, 0xB8DB38EFL, 0x8E79DCB0L, 0x603A180EL, 0x6C9E0E8BL,
       0xB01E8A3EL, 0xD71577C1L, 0xBD314B27L, 0x78AF2FDAL, 0x55605C60L,
       0xE65525F3L, 0xAA55AB94L, 0x57489862L, 0x63E81440L, 0x55CA396AL,
       0x2AAB10B6L, 0xB4CC5C34L, 0x1141E8CEL, 0xA15486AFL, 0x7C72E993L,
       0xB3EE1411L, 0x636FBC2AL, 0x2BA9C55DL, 0x741831F6L, 0xCE5C3E16L,
       0x9B87931EL, 0xAFD6BA33L, 0x6C24CF5CL};
    unsigned int Ord2[32] = {5, 14, 26, 18, 11, 28, 7, 16, 0, 23, 20, 22, 1, 10,
       4, 8, 30, 3, 21, 9, 17, 24, 29, 6, 19, 12, 15, 13, 2, 25, 31, 27};
    unsigned int Ord3[32] = {19, 9, 4, 20, 28, 17, 8, 22, 29, 14, 25, 12, 24,
       30, 16, 26, 31, 15, 7, 3, 1, 0, 18, 27, 13, 6, 21, 10, 23, 11, 5, 2};
    unsigned int A1, B1, C1, D1, E1, F1, G1, H1, TempInt;
    time_t TheTime;
    FILE *fp1, *fp2;

    TheTime = time(NULL);
    srandom(TheTime);

    DeltaW19  = 0xaaaaaaab;
    DeltaH56 = DeltaW19;

    W19 = random()^Rotate(random(),1);
    Wt19 = W19 - DeltaW19;

    Count1 = 0;

    for(i=0; i<MAX; i++)
      {
Test = 1;
while(Test != 0)
  {
```

```
      Count1++;
      H56 = random()^Rotate(random(),1);
      Ht56 = H56 - DeltaH56;

      Test = (RotateRight(H56,RFACT11) - RotateRight(Ht56,RFACT11)) +
        (DeltaW19);

  }
DataH56[i] = H56;
      }

   printf("Equation (9) = %8.8X\n", (RotateRight(H56,RFACT11) + W19) - (Ro-
tateRight(Ht56,RFACT11) + Wt19));
   printf("\nAverage number of Iterations: %lf\n\n", ((double)(Count1))/MAX);

   /* Determine if (8) holds */
   B62 = H56;
   Bt62 = H56 - DeltaH56;
   C62 = H56|Ht56;
   D62 = ~(H56|Ht56);
   E62 = H56|Ht56;
   F62 = H56|Ht56;
   G62 = ~(H56|Ht56);
   H62 = random() ^ Rotate(random(),1);

   Test = (RotateRight(FF2(B62,C62,D62,E62,F62,G62,H62),RFACT7) -
   RotateRight(FF2(Bt62,C62,D62,E62,F62,G62,H62),RFACT7));

   printf("Equation (8) = %8.8X\n", Test);


   /* Determine if (7) Holds */

   B61 = ~(H56|Ht56);
   C61 = B62;
   Ct61 = Bt62;
   D61 = C62;
   E61 = D62;
   F61 = E62;
   G61 = F62;
   H61 = G62;

   Test = (RotateRight(FF2(B61,C61,D61,E61,F61,G61,H61),RFACT7) -
   RotateRight(FF2(B61,Ct61,D61,E61,F61,G61,H61),RFACT7));

   printf("Equation (7) = %8.8X\n", Test);

   /* Determine if (6) Holds */

   B60 = ~(H56|Ht56);
   C60 = B61;
   D60 = C61;
   Dt60 = Ct61;
   E60 = D61;
   F60 = E61;
```

```
G60 = F61;
H60 = G61;

Test = (RotateRight(FF2(B60,C60,D60,E60,F60,G60,H60),RFACT7) -
RotateRight(FF2(B60,C60,Dt60,E60,F60,G60,H60),RFACT7));

printf("Equation (6) = %8.8X\n", Test);

/* Determine if (5) Holds */

B59 = (H56|Ht56);
C59 = B60;
D59 = C60;
E59 = D60;
Et59 = Dt60;
F59 = E60;
G59 = F60;
H59 = G60;

Test = (RotateRight(FF2(B59,C59,D59,E59,F59,G59,H59),RFACT7) -
RotateRight(FF2(B59,C59,D59,Et59,F59,G59,H59),RFACT7));

printf("Equation (5) = %8.8X\n", Test);

/* Determine if (4) Holds */

B58 = ~(H56|Ht56);
C58 = B59;
D58 = C59;
E58 = D59;
F58 = E59;
Ft58 = Et59;
G58 = F59;
H58 = G59;

Test = (RotateRight(FF2(B58,C58,D58,E58,F58,G58,H58),RFACT7) -
RotateRight(FF2(B58,C58,D58,E58,Ft58,G58,H58),RFACT7));

printf("Equation (4) = %8.8X\n", Test);

/* Determine if (3) Holds */

B57 = ~(H56|Ht56);
C57 = B58;
D57 = C58;
E57 = D58;
F57 = E58;
G57 = F58;
Gt57 = Ft58;
H57 = G58;

Test = (RotateRight(FF2(B57,C57,D57,E57,F57,G57,H57),RFACT7) -
RotateRight(FF2(B57,C57,D57,E57,F57,Gt57,H57),RFACT7));

printf("Equation (3) = %8.8X\n", Test);
```

```
   /* Determine if (2) Holds */

   B56 = ~(H56|Ht56);
   C56 = B57;
   D56 = C57;
   E56 = D57;
   F56 = E57;
   G56 = F57;
   H56 = G57;
   Ht56 = Gt57;

   Test = (RotateRight(FF2(B56,C56,D56,E56,F56,G56,H56),RFACT7) -
   RotateRight(FF2(B56,C56,D56,E56,F56,G56,Ht56),RFACT7));

   printf("Equation (2) = %8.8X\n", Test);

   printf("B56: %8.8X\n", B56);
   printf("C56: %8.8X\n", C56);
   printf("D56: %8.8X\n", D56);
   printf("E56: %8.8X\n", E56);
   printf("F56: %8.8X\n", F56);
   printf("G56: %8.8X\n", G56);
   printf("H56: %8.8X\n", H56);
   printf("Ht56: %8.8X\n", Ht56);
   printf("H57: %8.8X\n", H57);
   printf("H58: %8.8X\n", H58);
   printf("H59: %8.8X\n", H59);
   printf("H60: %8.8X\n", H60);
   printf("H61: %8.8X\n", H61);
   printf("H62: %8.8X\n", H62);

   /* Derive a message that results in a collision for the last two round-
s of
    * three round HAVAL */

   /* Start forward search */
   Chain[AA][56] = random()^Rotate(random(),1);
   Chain[BB][56] = B56;
   Chain[CC][56] = C56;
   Chain[DD][56] = D56;
   Chain[EE][56] = E56;
   Chain[FF][56] = F56;
   Chain[GG][56] = G56;
   Chain[HH][56] = H56;
   Chain[HH][57] = H57;

   W[12] = InverseStep(Chain[HH][57], Chain[AA][56], Chain[BB][56],
       Chain[CC][56], Chain[DD][56], Chain[EE][56],
       Chain[FF][56], Chain[GG][56], Chain[HH][56], K1[25]);

   Chain[AA][57] = Chain[BB][56];
   Chain[BB][57] = Chain[CC][56];
   Chain[CC][57] = Chain[DD][56];
   Chain[DD][57] = Chain[EE][56];
```

```
Chain[EE][57] = Chain[FF][56];
Chain[FF][57] = Chain[GG][56];
Chain[GG][57] = Chain[HH][56];

Chain[HH][58] = H58;

W[15] = InverseStep(Chain[HH][58], Chain[AA][57], Chain[BB][57],
    Chain[CC][57], Chain[DD][57], Chain[EE][57],
    Chain[FF][57], Chain[GG][57], Chain[HH][57], K1[26]);

Chain[AA][58] = Chain[BB][57];
Chain[BB][58] = Chain[CC][57];
Chain[CC][58] = Chain[DD][57];
Chain[DD][58] = Chain[EE][57];
Chain[EE][58] = Chain[FF][57];
Chain[FF][58] = Chain[GG][57];
Chain[GG][58] = Chain[HH][57];

Chain[HH][59] = H59;

W[13] = InverseStep(Chain[HH][59], Chain[AA][58], Chain[BB][58],
    Chain[CC][58], Chain[DD][58], Chain[EE][58],
    Chain[FF][58], Chain[GG][58], Chain[HH][58], K1[27]);

Chain[AA][59] = Chain[BB][58];
Chain[BB][59] = Chain[CC][58];
Chain[CC][59] = Chain[DD][58];
Chain[DD][59] = Chain[EE][58];
Chain[EE][59] = Chain[FF][58];
Chain[FF][59] = Chain[GG][58];
Chain[GG][59] = Chain[HH][58];

Chain[HH][60] = H60;

W[2] = InverseStep(Chain[HH][60], Chain[AA][59], Chain[BB][59],
   Chain[CC][59], Chain[DD][59], Chain[EE][59],
   Chain[FF][59], Chain[GG][59], Chain[HH][59], K1[28]);

Chain[AA][60] = Chain[BB][59];
Chain[BB][60] = Chain[CC][59];
Chain[CC][60] = Chain[DD][59];
Chain[DD][60] = Chain[EE][59];
Chain[EE][60] = Chain[FF][59];
Chain[FF][60] = Chain[GG][59];
Chain[GG][60] = Chain[HH][59];

Chain[HH][61] = H61;

W[25] = InverseStep(Chain[HH][61], Chain[AA][60], Chain[BB][60],
    Chain[CC][60], Chain[DD][60], Chain[EE][60],
    Chain[FF][60], Chain[GG][60], Chain[HH][60], K1[29]);

Chain[AA][61] = Chain[BB][60];
Chain[BB][61] = Chain[CC][60];
Chain[CC][61] = Chain[DD][60];
```

```
   Chain[DD][61] = Chain[EE][60];
   Chain[EE][61] = Chain[FF][60];
   Chain[FF][61] = Chain[GG][60];
   Chain[GG][61] = Chain[HH][60];

   Chain[HH][62] = H62;

   W[31] = InverseStep(Chain[HH][62], Chain[AA][61], Chain[BB][61],
       Chain[CC][61], Chain[DD][61], Chain[EE][61],
       Chain[FF][61], Chain[GG][61], Chain[HH][61], K1[30]);

   Chain[AA][62] = Chain[BB][61];
   Chain[BB][62] = Chain[CC][61];
   Chain[CC][62] = Chain[DD][61];
   Chain[DD][62] = Chain[EE][61];
   Chain[EE][62] = Chain[FF][61];
   Chain[FF][62] = Chain[GG][61];
   Chain[GG][62] = Chain[HH][61];

   Chain[HH][63] = random()^Rotate(random(),1);

   W[27] = InverseStep(Chain[HH][63], Chain[AA][62], Chain[BB][62],
       Chain[CC][62], Chain[DD][62], Chain[EE][62],
       Chain[FF][62], Chain[GG][62], Chain[HH][62], K1[31]);

   /* Find W[19]  and Wt[19] */
   Chain[AA][55] = random()^Rotate(random(),1);
   Chain[BB][55] = Chain[AA][56];
   Chain[CC][55] = Chain[BB][56];
   Chain[DD][55] = Chain[CC][56];
   Chain[EE][55] = Chain[DD][56];
   Chain[FF][55] = Chain[EE][56];
   Chain[GG][55] = Chain[FF][56];
   Chain[HH][55] = Chain[GG][56];

   W[19] = InverseStep(Chain[HH][56], Chain[AA][55], Chain[BB][55],
       Chain[CC][55], Chain[DD][55], Chain[EE][55],
       Chain[FF][55], Chain[GG][55], Chain[HH][55], K1[24]);

   Wt[19] = InverseStep(Ht56, Chain[AA][55], Chain[BB][55],
Chain[CC][55], Chain[DD][55], Chain[EE][55],
Chain[FF][55], Chain[GG][55], Chain[HH][55], K1[24]);

   for(i=0; i<32; i++) {
      if(i!=19) {
 Wt[i] = W[i];
      }
   }

   printf("Wt[19]: %8.8X\n", Wt[19]);

   printf("Equation (9) = %8.8X\n", (RotateRight(H56,RFACT11) + W[19]) - (Ro-
tateRight(Ht56,RFACT11) + Wt[19]));

   /* Test if message words derived allows a collision to be established */
```

```
   A1 = Chain[AA][55];
   B1 = Chain[BB][55];
   C1 = Chain[CC][55];
   D1 = Chain[DD][55];
   E1 = Chain[EE][55];
   F1 = Chain[FF][55];
   G1 = Chain[GG][55];
   H1 = Chain[HH][55];

   for(i=0; i<8; i++) {
      printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", 55+i, A1, B1,

      TempInt = RotateRight(FF2(B1,C1,D1,E1,F1,G1,H1), RFACT7) +
RotateRight(A1, RFACT11) + W[Ord2[i+24]] + K1[i+24];

      A1 = B1;
      B1 = C1;
      C1 = D1;
      D1 = E1;
      E1 = F1;
      F1 = G1;
      G1 = H1;
      H1 = TempInt;
   }

   for(i=0; i<1; i++) {
      printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", 62+i, A1, B1,

      TempInt = RotateRight(FF3(B1,C1,D1,E1,F1,G1,H1), RFACT7) +
RotateRight(A1, RFACT11) + W[Ord3[i]] + K2[i];

      A1 = B1;
      B1 = C1;
      C1 = D1;
      D1 = E1;
      E1 = F1;
      F1 = G1;
      G1 = H1;
      H1 = TempInt;
   }

   printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n\n", 62+i, A1, B1,

   /* Try for Wt19 */
   A1 = Chain[AA][55];
   B1 = Chain[BB][55];
   C1 = Chain[CC][55];
   D1 = Chain[DD][55];
   E1 = Chain[EE][55];
   F1 = Chain[FF][55];
   G1 = Chain[GG][55];
   H1 = Chain[HH][55];

   for(i=0; i<8; i++) {
```

```
      printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", 55+i, A1, B1,

      TempInt = RotateRight(FF2(B1,C1,D1,E1,F1,G1,H1), RFACT7) +
RotateRight(A1, RFACT11) + Wt[Ord2[i+24]] + K1[i+24];

      A1 = B1;
      B1 = C1;
      C1 = D1;
      D1 = E1;
      E1 = F1;
      F1 = G1;
      G1 = H1;
      H1 = TempInt;
   }

   for(i=0; i<1; i++) {
      printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", 62+i, A1, B1,

      TempInt = RotateRight(FF3(B1,C1,D1,E1,F1,G1,H1), RFACT7) +
RotateRight(A1, RFACT11) + Wt[Ord3[i]] + K2[i];

      A1 = B1;
      B1 = C1;
      C1 = D1;
      D1 = E1;
      E1 = F1;
      F1 = G1;
      G1 = H1;
      H1 = TempInt;
   }

   printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n\n", 62+i, A1, B1,

   /* Target IVs */

   Chain[HH][31] = 0x243F6A88L;
   Chain[GG][31] = 0x85A308D3L;
   Chain[FF][31] = 0x13198A2EL;
   Chain[EE][31] = 0x03707344L;
   Chain[DD][31] = 0xA4093822L;
   Chain[CC][31] = 0x299F31D0L;
   Chain[BB][31] = 0x082EFA98L;
   Chain[AA][31] = 0xEC4E6C89L;

   for(i=0; i<8; i++) {
      Chain[AA][32+i] = Chain[BB][31+i];
      Chain[BB][32+i] = Chain[CC][31+i];
      Chain[CC][32+i] = Chain[DD][31+i];
      Chain[DD][32+i] = Chain[EE][31+i];
      Chain[EE][32+i] = Chain[FF][31+i];
      Chain[FF][32+i] = Chain[GG][31+i];
      Chain[GG][32+i] = Chain[HH][31+i];

   }
```

```
   /* Derive all words except last 8 */

   for(i=0; i<16; i++) {
      Chain[AA][54-i] = random()^Rotate(random(),1);
      Chain[BB][54-i] = Chain[AA][55-i];
      Chain[CC][54-i] = Chain[BB][55-i];
      Chain[DD][54-i] = Chain[CC][55-i];
      Chain[EE][54-i] = Chain[DD][55-i];
      Chain[FF][54-i] = Chain[EE][55-i];
      Chain[GG][54-i] = Chain[FF][55-i];
      Chain[HH][54-i] = Chain[GG][55-i];

      W[Ord2[23-i]] = InverseStep(Chain[HH][55-i], Chain[AA][54-i],
  Chain[BB][54-i], Chain[CC][54-i],
  Chain[DD][54-i], Chain[EE][54-i],
  Chain[FF][54-i], Chain[GG][54-i],
  Chain[HH][54-i], K1[23-i]);
   }

   for(i=0; i<8; i++) {
      Chain[BB][38-i] = Chain[AA][39-i];
      Chain[CC][38-i] = Chain[BB][39-i];
      Chain[DD][38-i] = Chain[CC][39-i];
      Chain[EE][38-i] = Chain[DD][39-i];
      Chain[FF][38-i] = Chain[EE][39-i];
      Chain[GG][38-i] = Chain[FF][39-i];
      Chain[HH][38-i] = Chain[GG][39-i];

      W[Ord2[7-i]] = InverseStep(Chain[HH][39-i], Chain[AA][38-i],
  Chain[BB][38-i], Chain[CC][38-i],
  Chain[DD][38-i], Chain[EE][38-i],
  Chain[FF][38-i], Chain[GG][38-i],
  Chain[HH][38-i], K1[7-i]);
   }

   for(i=0; i<32; i++) {
      if(i!=19) {
 Wt[i] = W[i];
      }
      printf("%i) %8.8X %8.8X\n", i, W[i], Wt[i]);
   }

   /* With IV target reached and message word derived, proceed to verify
    * collision for last two rounds of three round HAVAL */

   A1 = Chain[AA][31];
   B1 = Chain[BB][31];
   C1 = Chain[CC][31];
   D1 = Chain[DD][31];
   E1 = Chain[EE][31];
   F1 = Chain[FF][31];
   G1 = Chain[GG][31];
   H1 = Chain[HH][31];

   for(i=0; i<32; i++) {
```

```
      printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", i+1, A1, B1,

      TempInt = RotateRight(FF2(B1,C1,D1,E1,F1,G1,H1), RFACT7) +
RotateRight(A1, RFACT11) + W[Ord2[i]] + K1[i];

      A1 = B1;
      B1 = C1;
      C1 = D1;
      D1 = E1;
      E1 = F1;
      F1 = G1;
      G1 = H1;
      H1 = TempInt;
   }

   for(i=0; i<32; i++) {
      printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", i+1, A1, B1,

      TempInt = RotateRight(FF3(B1,C1,D1,E1,F1,G1,H1), RFACT7) +
RotateRight(A1, RFACT11) + W[Ord3[i]] + K2[i];

      A1 = B1;
      B1 = C1;
      C1 = D1;
      D1 = E1;
      E1 = F1;
      F1 = G1;
      G1 = H1;
      H1 = TempInt;
   }

   printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n\n", i+1, A1, B1, C

   /* With IV target reached and message word derived, proceed to verify
    * collision for last two rounds of three round HAVAL */

   A1 = Chain[AA][31];
   B1 = Chain[BB][31];
   C1 = Chain[CC][31];
   D1 = Chain[DD][31];
   E1 = Chain[EE][31];
   F1 = Chain[FF][31];
   G1 = Chain[GG][31];
   H1 = Chain[HH][31];

   for(i=0; i<32; i++) {
      printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", i+1, A1, B1,

      TempInt = RotateRight(FF2(B1,C1,D1,E1,F1,G1,H1), RFACT7) +
RotateRight(A1, RFACT11) + Wt[Ord2[i]] + K1[i];

      A1 = B1;
      B1 = C1;
      C1 = D1;
      D1 = E1;
```

```
      E1 = F1;
      F1 = G1;
      G1 = H1;
      H1 = TempInt;
   }

   for(i=0; i<32; i++) {
      printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", i+1, A1, B1,

      TempInt = RotateRight(FF3(B1,C1,D1,E1,F1,G1,H1), RFACT7) +
RotateRight(A1, RFACT11) + Wt[Ord3[i]] + K2[i];

      A1 = B1;
      B1 = C1;
      C1 = D1;
      D1 = E1;
      E1 = F1;
      F1 = G1;
      G1 = H1;
      H1 = TempInt;
   }

   printf("%d) %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X %8.8X\n", i+1, A1, B1, C1,

   /* Write message words to file */

   fp1 = fopen("data1.dat", "w");
   fp2 = fopen("data2.dat", "w");

   if(fp1 == NULL || fp2 == NULL)
     {
printf("Error Opening File\n");
exit(1);
     }


   for(i=0; i<32; i++) {

      for(j=0; j<4; j++) {
 TempChar = (char) (W[i] >> ((j)*8) & 0x000000ff);
 fwrite(&TempChar, sizeof(char), 1, fp1);

 TempChar = (char) (Wt[i] >> ((j)*8) & 0x000000ff);
 fwrite(&TempChar, sizeof(char), 1, fp2);
      }
   }

   fclose(fp1);
   fclose(fp2);

   return(0);
}

unsigned int InverseStep(unsigned int H2, unsigned int A1, unsigned int B1,
 unsigned int C1, unsigned int D1, unsigned int E1,
```

```
 unsigned int F1, unsigned int G1, unsigned int H1,
 unsigned int K)
{
   unsigned int W;

   W = H2 - RotateRight(FF2(B1, C1, D1, E1, F1, G1, H1), RFACT7) -
     RotateRight(A1, RFACT11) - K;

   return(W);
}

unsigned int Rotate(unsigned int X, unsigned int s)
{
   unsigned int temp;

   temp = X;
   X = (X << s) | (temp >> (32-s));
   return(X);
}

unsigned int RotateRight(unsigned int X, unsigned int s)
{
   unsigned int temp;

   temp = X;
   X = (X >> s) | (temp << (32-s));
   return(X);
}

unsigned int FF2(unsigned int B, unsigned int C, unsigned int D,
       unsigned int E, unsigned int F, unsigned int G,
       unsigned int H)
{
   unsigned int x0, x1, x2, x3, x4, x5, x6;

   /* Permuteer die insette tot die funksies soos gespesifiseer */
   x0 = B;
   x1 = E;
   x2 = C;
   x3 = H;
   x4 = G;
   x5 = F;
   x6 = D;

   return((x1&x2&x3) ^ (x2&x4&x5) ^ (x1&x2) ^ (x1&x4) ^ (x2&x6) ^ (x3&x5) ^
  (x4&x5) ^ (x0&x2) ^ (x0));
}

unsigned int FF3(unsigned int B, unsigned int C, unsigned int D,
       unsigned int E, unsigned int F, unsigned int G,
       unsigned int H)
{
   unsigned int x0, x1, x2, x3, x4, x5, x6;

   /* Permuteer die insette tot die funksies soos gespesifiseer */
```