SØREN STEFFEN THOMSEN / 001856

CRYPTOGRAPHIC HASH FUNCTIONS
CRYPT0GRAPH1C HA5H FUNCT10N5
C2YP70624PH1C H45H FUNC710N5
C279706249H1C H45H F07C71075
027970624981 0 8458 507071075

Department of Mathematics
Technical University of Denmark
November 14, 2005

Hash, x. There is no definition for this word—nobody knows what hash is.

Ambrose Bierce, *The Devil's Dictionary*, 1906

Contents

1 Introduction **1**
1.1 Hash functions in short . . . 1
1.2 About this report . . . 1
1.3 Conventions . . . 3

2 Hash functions **5**
2.1 Properties . . . 5
2.2 The typical construction . . . 6
2.3 Applications of hash functions . . . 7
2.3.1 Digital signatures . . . 7
2.3.2 Commitment schemes . . . 8
2.3.3 Password protection schemes . . . 8
2.3.4 Data integrity . . . 8
2.3.5 Proof-of-work systems . . . 9
2.4 A brief history of hash functions . . . 9

3 Dobbertin's attack on MD4 **11**
3.1 Description of MD4 . . . 11
3.2 The attack . . . 12
3.2.1 Reaching the goal . . . 12
3.2.2 Inner almost-collision . . . 15
3.2.3 Right initial value . . . 17
3.3 The algorithm . . . 19
3.4 Summary . . . 20

4 Wang's attack on MD5 **21**
4.1 Description of MD5 . . . 21
4.2 Introduction to the attack . . . 22
4.3 The idea . . . 23
4.4 Details . . . 24
4.4.1 Origin of the conditions . . . 24
4.4.2 Message modification . . . 26
4.4.3 Propagating carries . . . 28

4.5 Detailed examination of the second iteration . . . 30
4.5.1 Conditions on the T-values . . . 30
4.5.2 Conditions on step variables . . . 34
4.6 Modifications by Klíma . . . 46
4.6.1 An overview . . . 47
4.6.2 Details . . . 47
4.7 Possible additional improvements . . . 47
4.8 An implementation . . . 49
4.9 Constructing meaningful collisions . . . 50

5 AES-based hash functions **52**
5.1 Block cipher-based hash functions in general . . . 52
5.1.1 Fast double-length schemes . . . 53
5.1.2 DES and MDC-2 . . . 54
5.1.3 Using AES . . . 54
5.1.4 Related-key attack on AES_{256} . . . 55
5.2 Extending AES to support 256-bit blocks . . . 55
5.3 Alternative constructions . . . 56
5.3.1 The Lucks scheme . . . 56
5.3.2 The Knudsen-Preneel scheme . . . 56
5.4 Summary . . . 63

6 General results on the Merkle-Damgård construction **65**
6.1 The motivation for using MD-strengthening . . . 65
6.2 Joux's multicollisions . . . 66
6.3 Kelsey/Schneier's 2nd preimage attack . . . 67
6.3.1 Expandable messages . . . 67
6.3.2 Finding a 2nd preimage . . . 68

7 Hash functions based on modular arithmetic **69**
7.1 A simple hash function . . . 69
7.2 The Chaum-van Heijst-Pfitzmann hash function . . . 70
7.3 MASH-1 and MASH-2 . . . 71

8 The SMASH hash function **73**
8.1 The design of SMASH . . . 73
8.1.1 The construction . . . 73
8.1.2 The compression function . . . 74
8.2 Analysis . . . 75
8.2.1 The forward prediction property . . . 76
8.2.2 Inverting the compression function . . . 76
8.2.3 Complexity of (2nd) preimage attacks . . . 76
8.3 An attack . . . 76
8.3.1 The idea . . . 76

8.3.2 Breaking a variant 77
8.3.3 Breaking SMASH 78
8.4 Possible improvements to SMASH 79
8.4.1 Using the secure compression function for every j steps . 80
8.4.2 Using different f functions in each step 80
8.4.3 Further dependency on the message 81
8.4.4 Using more than one fixed, bijective mapping 83

9 Future directions **87**
9.1 A brief discussion on strategy 87
9.2 Merkle-Damgård or not Merkle-Damgård? 88
9.3 A discussion on efficiency 89
9.4 The NIST Cryptographic Hash Workshop 90
9.5 Summary 91

A Conditions on step variables in the Wang MD5 attack **93**
A.1 First iteration 93
A.2 Second iteration 96

B An implementation of the Dobbertin MD4 attack **98**

C An implementation of the Wang MD5 attack **104**

D An implementation of the Wang MD4 attack **127**

E The individual steps of MD4 **136**

Chapter 1

Introduction

In this first chapter, cryptographic hash functions are very briefly introduced, and a short summary of each of the following chapters is given. Finally, the conventions used throughout the report are described.

1.1 Hash functions in short

Cryptographic hash functions (or simply *hash functions*) map strings of arbitrary length to strings of a fixed length n. It should be easy to compute the output string, which is called the *hash value*, the *hash result* or simply the *hash*. The hash value may also be thought of as the *fingerprint* of some message. There are 2^n different hash values for a given hash function, and the idea is that the probability that some message hashes to a given hash value is 2^{-n}. Hence, the hash should truly be a fingerprint of the message, and although it is clear that several messages hash to the same value, it should be computationally infeasible to find two such messages in practice.

To be specific, one often considers three different kinds of *attacks* on hash functions (for any attack it is assumed that the attacker can compute the hash of any message). These can be stated informally as follows:

The *collision attack*. Find two messages that hash to the same value

The *preimage attack*. Find a message that hashes to a given value

The *2*$^{\text{nd}}$ *preimage attack*. Given a message, find a different message which hashes to the same value

A more thorough introduction to hash functions is given in Chapter 2.

1.2 About this report

The area of cryptographic hash functions is attracting great attention in the cryptographic community at the moment. Most importantly this increased interest is due

to a fairly large number of interesting and disturbing collision attacks on hash functions still widely in use. Moreover, many hash functions still considered secure are based on hash functions that have been broken.

This document describes various results on hash functions. These include attacks, proofs of security, and suggestions to new hash functions based on existing material. First, in Chapter 2, an introduction to hash functions, including some applications and the history of hash functions, is given.

In Chapter 3 the collision attack [12] on MD4 by Hans Dobbertin is described. This attack excited greater interest in cryptanalysis of hash functions, and modifications of the technique used by Dobbertin were applied in attacks on SHA-0 [6, 2], MD5 [45], SHA-1 [2, 44] etc. An implementation of this attack can be found in Appendix B. An implementation using a different technique, developed by Xiaoyun Wang et al. [43], has been implemented for comparison (see Appendix D).

This brings us to Chapter 4, which contains the description, including the author's own modifications, improvements and implementation of a collision attack on MD5 developed by Xiaoyun Wang et al. [45] and modified by Vlastimil Klíma [25]. This attack makes use of a differential pattern discovered by Xiaoyun Wang. She also discovered good differential patterns of other hash functions like MD4, RIPEMD etc., and thus she has helped reduce the complexity of finding collisions for a number of hash functions. Currently, she is working to reduce the complexity of finding collisions for SHA-1 to a practically applicable level. Her latest attack [44] has complexity 2^{63}.

Some suggestions for using AES or AES components to form a secure hash function are presented in Chapter 5. These include a suggestion based on error-correcting codes. It turns out to be difficult to construct efficient and convenient AES-based hash functions, but in the light of recent attacks on dedicated hash functions, it might still be worth considering such alternatives.

In Chapter 6, some general results on the Merkle-Damgård construction are presented. First, some motivation for using the Merkle-Damgård construction is given, and after that some generic attacks on the construction are described.

Chapter 7 presents a number of hash functions that are based on modular arithmetic. Hash functions of this type have some advantages to more common constructions, although they are fairly inefficient.

An attack on a new hash function proposal, SMASH, is described in Chapter 8, and some proposals (in part developed by the author) to improvements of SMASH that hopefully render it secure, are presented.

The final chapter is a discussion on actions to be taken as a consequence of recent attacks, and possible strategies for selecting hash functions to be used in many years to come. This discussion includes some points from the Cryptograhic Hash Workshop hosted by NIST on October 31 to November 1, 2005.

The reader is assumed to possess some prior knowledge of cryptographic tools such as number theory, probability theory, modular arithmetic etc. Prior knowledge of specific hash function designs is not required. See `http://www.student.dtu.`

dk/~s001856/exam/ for links to this report, program code and other resources.

1.3 Conventions

In this document a *word* is a 32-bit entity. Arithmetic operations are performed modulo 2^{32} unless explicitly stated otherwise. Numbering generally starts from 0, so for instance bit number 0 of a word is the least significant bit, message block number 0 is the first message block etc. Numbers have a subscript h when they are written in hexadecimal form, and a subscript b when written in binary form, unless it is clear from the context which form is being used. Binary and hexadecimal numbers are always written in `this font`.

Some symbols and operators and their meanings in this context are given in Table 1.1.

Notation	Meaning
$\oplus$	The exclusive-or (XOR) operator.
$\wedge$	The logic AND operator.
$\vee$	The logic (inclusive) OR operator.
$\neg X$	The bitwise complement of X.
$X^{\lll s}$	X rotated left cyclically by s positions.
$a \bmod n$	The least non-negative integer r such that $a = kn + r$ for some integer k.
$a \operatorname{div} n$	The number of times that n divides a (i.e. the value of k above).
$m_0 \| m_1$	The concatenation of m_0 and m_1.
$\|m\|$	The bit length of m.
$\mathtt{0}^s$, $\mathtt{1}^s$	The concatenation of s `0`-bits, or s `1`-bits, respectively.
$a \leftarrow b$	a is assigned the value of b.
$a \doteq b$	*a must equal b*, meaning that it is a requirement that a equals b.
$X[s]$	Bit no. s of X.
$X[s-t]$	The set of bits s to t of X, $t > s$.
δX	The modular difference between X and X', i.e. $X' - X$.
ΔX	The XOR difference between X and X', i.e. $X \oplus X'$.
∇X	A list $\langle s, t, \ldots \rangle$ of the bit positions at which X and X' differ. Each position is preceded by $+$ (may be omitted) or $-$, where $+s$ indicates that $X[s] = 0$ and $X'[s] = 1$, and $-s$ indicates that $X[s] = 1$ and $X'[s] = 0$. If s is not present in the list, then $X[s] = X'[s]$.
$\nabla X[s]$	The signed difference $X'[s] - X[s]$.

Table 1.1: The meaning of symbols and operators used in this report.

Chapter 2

Hash functions

In this chapter some properties of hash functions are presented in more detail than the short description in Chapter 1. Some applications of hash functions are mentioned, and a short history of hash functions is presented in the final section.

2.1 Properties

As mentioned in the introduction, there exist three basic kinds of attacks on hash functions. One could also define some properties of hash functions depending on their resistance to these attacks. Hence, we might define the following semi-formal properties of a hash function $H : \{0,1\}^* \to \{0,1\}^n$:

Preimage resistance. H is said to be preimage resistant if given a value $h \in \{0,1\}^n$ it is infeasible to find a message m such that $H(m) = h$.

2nd preimage resistance. H is said to be 2nd preimage resistant if given a message m it is infeasible to find $m' \neq m$ such that $H(m') = H(m)$.

Collision resistance. H is said to be collision resistant if it is infeasible to find two different messages m and m' such that $H(m) = H(m')$.

Variants of the attacks exist as well. Using the brute-force method, the complexity of a collision attack is $2^{n/2}$ (this attack is also called the *birthday attack*), and the complexity of a (2nd) preimage attack is 2^n. Most attacks in practice are collision attacks, since these are usually easier to mount than preimage attacks. Note that a hash function which is collision resistant is also 2nd preimage resistant, since a 2nd preimage is also a collision. However, it is not implied in general that a collision resistant hash function is preimage resistant.

A hash function often makes use of a fixed *initial value*. Attacks that require alteration of this fixed value are called *free-start* or *pseudo-* attacks.

2.2 The typical construction

Usually a message is *padded* before being processed by the hash function. The padding makes it possible to split the message into *blocks* of equal size μ. Most hash functions use the so-called Merkle-Damgård construction, see Construction 1 and Figure 2.1.

Construction 1 (The Merkle-Damgård construction)**.** The padded message m is split into t blocks $m_0, \ldots, m_{t-1}$ of each μ bits. Let $h_0 = v$ be an initial n-bit value defined by the hash function, and let $f : \{0,1\}^n \times \{0,1\}^\mu \to \{0,1\}^n$ be a *compression function*. Define

$$h_{i+1} = f(h_i, m_i) \quad \text{for } 0 \leq i < t$$

The hash of m is then $H(m) = h_t$.

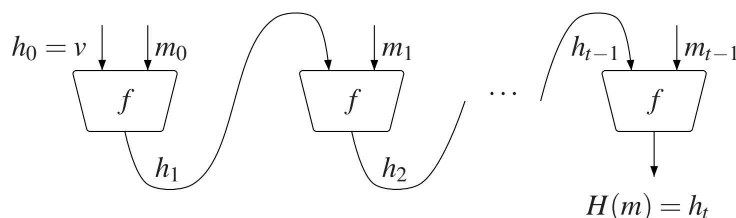


Figure 2.1: The Merkle-Damgård construction. $m = m_0 \| m_1 \| \cdots \| m_{t-1}$ is the message including padding, v is an initial value, and $H(m)$ is the final hash value.

The intermediate values $h_1, \ldots, h_{t-1}$ are called *chaining values* or *chaining variables*. Padding is usually performed as in Rule 1.

Rule 1 (Padding rule)**.** Let m be the message to be padded, $|m| = \tau$. Let μ be the block size of the hash function. Append a `1`-bit to m. Then append w `0`-bits, and finally (the *MD-strengthening*) append a u-bit representation of τ. u is a fixed number depending on the hash function, and w is the least non-negative integer such that the length in bits of the padded message is a multiple of μ.

Note that some hash functions (such as SHA-1) require that $\tau < 2^u$, while others (e.g. MD5) use the convention that the u least significant bits of τ are used in the padding.

When MD-strengthening is used in the Merkle-Damgård construction it is possible to prove (the proof is given in Section 6.1) that if the compression function is collision resistant, then so is the entire hash function. If we were to remove the MD-strengthening part of Rule 1 (and no other changes were made to the process), then we would not be able to make this proof. For instance, let H be the hash function, f the compression function and v the initial value, assume that $H(m_0 \| m_1) = H(m_1)$

(i.e. $m_0\|m_1$ and m_1 collide under H), and that $f(m_0,v) = v$. Then for the two messages, the input to the compression function when processing m_1 is identical, and so there is no collision for the compression function.

The compression function usually contains a number of steps, where in each step a part of the message block and the compression state are used to manipulate the compression state. At the end, when the whole message block has been included at least once in the computations, the initial state is often added to the final state to give the output of the compression function. Without this *feed forward* the compression function would be easily invertible, which would make free-start preimages easy to find. Invertible compression functions also facilitate true (2nd) preimage attacks of lower complexity than using the brute force method, as described in Section 8.2.3.

2.3 Applications of hash functions

Hash functions are used in a number of cryptographic contexts, such as digital signatures, password protection schemes and data integrity, and keyed versions of hash functions (which are not covered here) are used as message authentication codes. Hash functions could also be used as pseudo-random number generators, but only in such cases where the randomness of the hash function has been sufficiently examined and deemed appropriate for this purpose [30, §9.2.6]. In fact, FIPS 186 [18] defines an approved pseudo-random number generator based on SHA-1 for use in generating secret parameters for the Digital Signature Algorithm, also defined in [18].

Hash functions are also used in non-cryptographic contexts for storing data in hash tables. The hash functions used for this purpose, however, cannot be compared with cryptographic hash functions except that both types are mappings from a large domain to a smaller range.

Some common applications of hash functions are now described in a little more detail, along with some reasoning as to why the hash function should be resistant against the three kinds of attacks mentioned.

2.3.1 Digital signatures

A digital signature is used for a number of purposes, including:

ensuring authenticity the recipient of a digitally signed message may be confident that the sender is the person he claims to be.

preventing repudiation the signer of a message cannot subsequently claim that he did not sign the message.

These properties only hold when the digital signature is secure.

For efficiency reasons, a message is usually hashed before it is signed. Hashing is usually much faster than signing and verifying, and hence time is saved if a short

hash is signed instead of the entire message. This also yields a shorter signature. A weakness of the hash function, however, could be exploited by an adversary to forge a digital signature. For instance, if the adversary is able to produce meaningful collisions of the hash function, he may get some person to sign a message and then subsequently claim that the signer in fact signed another message.

2.3.2 Commitment schemes

Two people who would like to agree on some value V in a fashion so that one person cannot make his choice based on the choice of the other person, may use hash functions for this purpose in the following way: Person A decides on a value V_A of V, appends a *nonce* N_A, which is some random value that is only used once, to V_A, and then computes the hash of $V_A \| N_A$. The nonce is appended because often the number of possible values of V is so small that an adversary can compute the hash of all possibilities of V. Appending a nonce of a reasonable length prevents this. Person A then sends this hash to B, who decides on his own value V_B without being able to see what A chose. He may then compute $H(V_B \| N_B)$, where N_B is B's own nonce, and send this hash to A. Now A and B can exchange nonces and choices of V, and verify that the other person in fact did choose the value of V that he claimed.
Of course, one may come up with a number of other contexts in which it is required that some party (or parties) commits to a certain piece of information prior to revealing what that information is.

2.3.3 Password protection schemes

For obvious reasons, passwords should never be stored in plain text. Instead, the hash of the password may be stored. When a password needs to be verified, the hash of the password is compared to the stored hash, and if they match there is a very good probability that the password is correct. Hash functions used for this purpose must be preimage resistant, since it should not be possible to compute the actual password from its hash.

2.3.4 Data integrity

When a message needs to be stored for a long time, and it is important that any alterations of the message can be detected, one may compute the hash of the message and store this value safely instead of the entire message. The integrity of the message can be checked at a later time by computing the hash again, and comparing this value with the stored hash. This way the problem of ensuring data integrity is reduced to the shorter hash instead of the entire message.

Hash functions used for this purpose should be at least 2nd preimage resistant, since otherwise an adversary may replace the original message with a 2nd preimage, and this forgery would not be detected using the technique described. In fact, the

hash function should even be collision resistant to prevent the originator of the message from playing the part of the adversary in the forgery just described.

2.3.5 Proof-of-work systems

An attempt to fight spam is the so-called proof-of-work system *Hashcash* [20]. In this scheme, the sender of an e-mail must prove that he has performed an amount of work in the process of sending the e-mail. To prove this, the sender must compose a certain header for the email, which is a string consisting of the sender's e-mail address, the current date and a random number such that the SHA-1 hash of the entire header has all zeroes as the first k bits. This requires 2^k evaluations of the SHA-1 compression function on the average (here, k is "currently" 19, but this could increase as computing power increases).
The recipient can quite easily check that the header has been composed in the correct manner. The theory behind this system is that spammers who submit thousands of e-mails will be severely troubled by having to construct the header.

2.4 A brief history of hash functions

The need for cryptographic hash functions first arose in contexts of digital authentication such as password protection. The term used for hash functions in the beginning was *one-way functions*, indicating that these functions should be preimage resistant. Whitfield Diffie and Martin E. Hellman were the first [11] to define such one-way functions. The ideas of 2nd preimage and collision resistance were developed in the following years, but it took a while for formal definitions to appear.

Diffie and Hellman also showed [11] how a secure cryptosystem could be used to create a hash function. Around 1980 the first concrete schemes, the Davies-Meyer and the Matyas-Meyer-Oseas schemes, appeared, but already at that time these schemes used in conjunction with DES, *the* encryption algorithm of the time, did not provide sufficient security.

It was not until 1988 that an applicable construction based on DES, the MDC-2 construction, was developed [32, 5]. About the same time, one of the first dedicated hash functions, MD2 [37], was developed by Ronald Rivest. MD2 was superseded by MD4 [36], which in turn was replaced (1991) by MD5 [38], the first hash function to be in widespread use. In 1993, the National Institute of Standards and Technology (NIST) approved [15] the SHA hash function, a hash function built upon the ideas of MD4 and later to become known as SHA-0. SHA-0 was withdrawn by the NSA shortly after its publication, and in 1995 it was superseded by SHA-1 [16], which uses the same compression function as SHA-0, but a slightly different message expansion. All the mentioned dedicated hash functions are now considered broken. In 2001, NIST published (in the draft version of [17]) a new set of hash functions including SHA-256, which is currently NIST's preferred hash

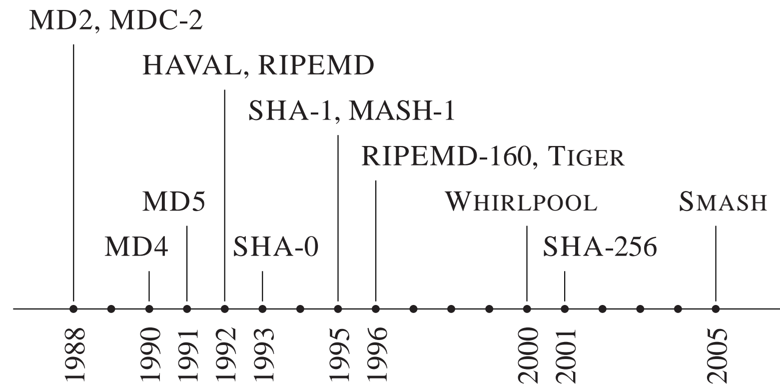


Figure 2.2: The "year of birth" of a number of hash functions

function.

The last couple of years have seen an increase in the amount of cryptanalytic work on hash functions, and especially the work of Xiaoyun Wang et al. [43, 44, 45] has increased the focus on hash functions in the cryptographic community. Since the attacks of Wang are all directed towards hash functions of the so-called MD4 family, and since these attacks seem fairly generic, the cryptographic community is currently facing the dilemma of whether to keep improving existing hash functions of this family, or to abandon the MD4 family and start considering alternatives constructions only. The WHIRLPOOL hash function, which is similar to AES and cannot be considered MD4-like, was developed in 2000 and has not yet been broken. A new proposal of the year 2005, SMASH, is based on completely different principles to those of the MD4 family. However, it was broken shortly after its publication.

Figure 2.2 shows the years of birth of all the hash functions mentioned, and a few others which have gained some attention. The only ones of these which are still considered secure and applicable are RIPEMD-160, SHA-256, TIGER, and WHIRLPOOL. The first two are MD4-like. TIGER is optimised for the 64-bit architecture.

Chapter 3

Dobbertin's attack on MD4

MD4 [36] is a hash function introduced in 1990 by Ronald Rivest, one of the creators of the RSA cryptosystem. It was superseded the following year by MD5. This attack was developed by Hans Dobbertin and published [12] in 1998.

3.1 Description of MD4

MD4 uses Construction 1, and Rule 1 for padding (with $u = 64$). The size μ of each message block is 512 bits, and the output of the compression function and the full hash function is 128 bits. The initial value of MD4 is

$$\begin{aligned} Q_{-3} &= \texttt{67452301}_h \\ Q_{-2} &= \texttt{10325476}_h \\ Q_{-1} &= \texttt{98badcfe}_h \\ Q_0 &= \texttt{efcdab89}_h \end{aligned}$$

The compression function consists of 48 steps, and it works as follows. Let three functions be defined:

$$\begin{aligned} f_0(X,Y,Z) = F(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\ f_1(X,Y,Z) = G(X,Y,Z) &= (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \\ f_2(X,Y,Z) = H(X,Y,Z) &= X \oplus Y \oplus Z \end{aligned}$$

These are also referred to as if, maj, and xor respectively.

Define three constants,

$$\begin{aligned} k_0 &= 0 \\ k_1 &= \texttt{5a827999}_h \\ k_2 &= \texttt{6ed9eba1}_h \end{aligned}$$

The message m is split into 16 words $m_0, \ldots, m_{15}$. Let W_t be the message word used in step t. Then $W_t = m_u$, where u is found in the table below for increasing t (read from left to right, then down).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15

Each step also contains a rotation value, S_t, which can be derived from the table below.

S_t		t div 16		
		0	1	2
t mod 4	0	3	3	3
	1	7	5	9
	2	11	9	11
	3	19	13	15

$r = t \operatorname{div} 16$ is the *round* number, i.e. there are three rounds of 16 steps each. Now the compression function can be stated as in Algorithm 3.1. In this context, the values Q_i are referred to as step variables.

Algorithm 3.1 The MD4 compression function

Input: $(Q_{-3}, Q_{-2}, Q_{-1}, Q_0)$ and $m_0, \dots, m_{15}$
  for $t = 0$ to 47 **do**
    $Q_{t+1} \leftarrow (f_r(Q_t, Q_{t-1}, Q_{t-2}) + Q_{t-3} + k_r + W_t)^{\lll S_t}$ {*Here,* $r = t \operatorname{div} 16$}
  end for
  return $(Q_{45} + Q_{-3}, Q_{46} + Q_{-2}, Q_{47} + Q_{-1}, Q_{48} + Q_0)$

When reading the attack described in the following, it may be helpful to consult Appendix E which is a list of the individual steps of MD4.

3.2 The attack

This chapter describes a collision attack on all 48 steps of MD4. The messages that collide are both 512 bits, i.e. one block excluding padding. The two messages m and m' are very similar. In fact, $m'_i = m_i$ for all $0 \le i < 16$ except $i = 12$, and $m'_{12} = m_{12} + 1$.

3.2.1 Reaching the goal

Working backwards from the end, it is clear that the difference between the two inputs being on m_{12}, if there is a collision after step 35, there is a collision for the entire message digest, since m_{12} is not used as input in steps later than 35. Hence, our goal is to achieve a collision after step 35. In the following, step variables Q_i are related to message m, and step variables Q'_i are related to message m'.

In step 35 the step operation is

$$Q_{36} \leftarrow (Q_{32} + H(Q_{35}, Q_{34}, Q_{33}) + m_{12} + k_2)^{\lll 15},$$

and hence, since $m'_{12} = m_{12} + 1$, if we require that $Q'_{32} \doteq Q_{32} - 1$ and all other step variables are equal for the two messages, then we get

$$\begin{aligned} Q'_{36} &= (Q'_{32} + H(Q'_{35}, Q'_{34}, Q'_{33}) + m'_{12} + k_2)^{\lll 15} \\ &= (Q_{32} - 1 + H(Q_{35}, Q_{34}, Q_{33}) + m_{12} + 1 + k_2)^{\lll 15} \\ &= Q_{36}. \end{aligned}$$

In other words, a collision is achieved if $\delta Q_{32} = -1$, and $\delta Q_i = 0$ for $i \in \{33, 34, 35\}$.

Moving backwards one step further, in step 34 the step operation is

$$Q_{35} \leftarrow (Q_{31} + H(Q_{34}, Q_{33}, Q_{32}) + m_4 + k_2)^{\lll 11}.$$

Since it is required that $\delta Q_{35} \doteq 0$, but $\delta Q_{32} \doteq -1$ and $\delta Q_{33} \doteq \delta Q_{34} \doteq 0$, the value of δQ_{31} must be non-zero. The simplest solution is to require $\delta Q_{31} \doteq 1$ and hope that the difference out of the H function (which is the xor function) is -1. This happens with a fairly good probability, which is seen from the following. What is needed is that

$$\begin{aligned} Q_{34} \oplus Q_{33} \oplus Q_{32} &= (Q_{34} \oplus Q_{33} \oplus Q'_{32}) + 1 \Leftrightarrow \\ Q_{34} \oplus Q_{33} \oplus (Q'_{32} + 1) &= (Q_{34} \oplus Q_{33} \oplus Q'_{32}) + 1. \end{aligned}$$

This can be simplified to

$$X \oplus (Y + 1) = (X \oplus Y) + 1.$$

It is clear that this equation holds if the last (least significant) bit of both X and Y is `0`. This happens with probability $\left(\frac{1}{2}\right)^2$. However, the equation also holds if the last two bits of Y are `01`, and the last two bits of X are `00`. This happens with probability $\left(\frac{1}{2}\right)^4$. This argument can be re-applied for values of the last bits of Y of `011`, `0111`, etc., which yields an accumulated probability of

$$\sum_{j=1}^{32} \left(\frac{1}{2}\right)^{2j} + \frac{1}{2^{64}} \approx \frac{1}{1 - \frac{1}{4}} - 1 + \frac{1}{2^{64}} \approx \frac{1}{3}.$$

This seems high enough that we can keep the requirement $\delta Q_{31} \doteq 1$.

The same calculations will show that if $\delta Q_{30} = \delta Q_{29} = 0$, then the probability of success in each of the steps 32 and 33 will be about $\frac{1}{3}$, and hence we require these differences.

For δQ_{28} circumstances change as the basic function used in step 31 is not xor, but rather the maj function. The step operation of step 31 is

$$Q_{32} \leftarrow (Q_{28} + G(Q_{31}, Q_{30}, Q_{29}) + m_{15} + k_1)^{\lll 13}.$$

Hence, we must find a value of δQ_{28} such that with a good probability, $\delta Q_{32} = -1$. The simplest solution is to require

$$G(Q_{31}, Q_{30}, Q_{29}) = G(Q'_{31}, Q'_{30}, Q'_{29}),$$

which is equivalent to

$$G(Q_{31}, Q_{30}, Q_{29}) = G(Q_{31} + 1, Q_{30}, Q_{29}). \tag{3.1}$$

If this is achieved, then with a very good probability $\delta Q_{28} \doteq -2^{19}$ will work, since this difference may turn into -1 by the rotation of 13. In fact, this happens unless a borrow bit propagates from bit 19 past bit 31, i.e. there is no `1`-bit among bits 19–31 of $G(Q_{31}, Q_{30}, Q_{29}) + m_{15} + k_1$ (see Section 4.4.3 for more on this). The probability of this happening is only 2^{-13}.

The probability that (3.1) holds should be evaluated. If bit 0 of Q_{31} is `0`, then adding 1 does not produce a carry, and in this case (3.1) holds if bit 0 of Q_{30} and Q_{29} are equal, because then those two determine the majority independently of the third. Similarly, if the last two bits of Q_{31} are `01`, adding 1 causes both these bits to flip, and again the equation holds if Q_{30} and Q_{29} agree on the last two bits. This argument again repeats, and hence the overall probability reduces to about $\frac{1}{3}$ (same computation as for δQ_{31}). Hence, it makes sense to require $\delta Q_{28} \doteq -2^{19}$.

The requirement on δQ_{27} is chosen in the exact same way to be 2^{23} with a probability of success in step 30 of, again, about $\frac{1}{3}$.

In step 29 two of the input variables to the maj function, Q_{28} and Q_{27}, differ. However, the difference occurs on different bit positions, 4 positions apart, and hence it would not be unreasonable to say that these two differences are independent – at least for the most significant terms in the calculation of the probability of success. Thus, we deduce that the probability of success in step 29 when $\delta Q_{26} \doteq 0$ is $\left(\frac{1}{3}\right)^2 = \frac{1}{9}$, which is probably the best we can hope for, so we keep this requirement.

We can keep working our way back to step 20 following the same principles. The required differences and the probability of success for each step are shown in Table 3.1. Here, $P_{20} = 1$ because in the following we require that the values of the step variables after step 19 are *admissible*, i.e. that

$$G(Q_{20}, Q_{19}, Q_{18}) = G(Q'_{20}, Q'_{19}, Q_{18}). \tag{3.2}$$

Note that we always introduce a requirement on δQ_{i-3} in step i, because Q_{i-3} is the "oldest" step variable being used in that step.

The probability of success through all 16 steps can be computed as $\left(\frac{1}{3}\right)^{19} \approx 2^{-30}$, but in practice the probability is quite a lot better (about 2^{-22} according to [12]).

To see why we cannot go back any further, first observe that the input word used in step 19 is m_{12}. The computation taking place is

$$Q_{20} \leftarrow (Q_{16} + G(Q_{19}, Q_{18}, Q_{17}) + m_{12} + k_1)^{\lll 13} \quad \text{and} \tag{3.3}$$

$$Q'_{20} \leftarrow (Q'_{16} + G(Q_{19} + 2^5, Q_{18}, Q_{17}) + m_{12} + 1 + k_1)^{\lll 13}, \tag{3.4}$$

and we require that $\delta Q_{20} \doteq -2^{25}$. With a good probability this can be translated into

$$Q'_{20}{}^{\lll 19} - Q_{20}{}^{\lll 19} = -2^{12}. \tag{3.5}$$

Step (i)	δQ_{i-3}	P_i	Function	Input word
19	?			$\boxed{m_{12}}$
20	0	1	G	m_1
21	0	$\frac{1}{9}$	G	m_5
22	2^5	$\frac{1}{3}$	G	m_9
23	-2^{25}	$\frac{1}{3}$	G	m_{13}
24	0	$\frac{1}{9}$	G	m_2
25	0	$\frac{1}{9}$	G	m_6
26	2^{14}	$\frac{1}{3}$	G	m_{10}
27	-2^6	$\frac{1}{3}$	G	m_{14}
28	0	$\frac{1}{9}$	G	m_3
29	0	$\frac{1}{9}$	G	m_7
30	2^{23}	$\frac{1}{3}$	G	m_{11}
31	-2^{19}	$\frac{1}{3}$	G	m_{15}
32	0	$\frac{1}{3}$	H	m_0
33	0	$\frac{1}{3}$	H	m_8
34	1	$\frac{1}{3}$	H	m_4
35	-1	1	H	$\boxed{m_{12}}$

Table 3.1: Required differences on the "oldest" step variables used in each step and the probabilities of success (P_i) in steps 20–35.

We have the possibility to manipulate δQ_{16} only. Hence, to satisfy (3.5) we must require $\delta Q_{16} \doteq -2^{12}$ and see what the possibility is that $G(Q_{19}+2^5, Q_{18}, Q_{17}) = G(Q_{19}, Q_{18}, Q_{17}) - 1$ so that the other terms in (3.3) and (3.4) even out. This is clearly impossible, and so there must be some other, "complex" difference (i.e. one where more than one bit is set) δQ_{16}, and therefore we shall try to achieve the required differences on subsequent step variables by what we could call "qualified trial and error".

3.2.2 Inner almost-collision

We now know what difference $(\delta Q_{17}, \delta Q_{18}, \delta Q_{19}, \delta Q_{20})$ to aim for after step 19, where m_{12} is used as input word. We call this difference an *inner almost-collision*. m_{12} is also used as input word in step 12, and so we shall try to find initial values of the registers before step 12 that cause the right difference after step 19.

Since $m_i = m_i'$ for $i < 12$, $\delta Q_i = 0$ for $i < 13$. In step 12, the operations taking

place are

$$\begin{aligned} Q_{13} &\leftarrow (Q_9 + F(Q_{12}, Q_{11}, Q_{10}) + m_{12})^{\lll 3} \quad \text{and} \\ Q'_{13} &\leftarrow (Q_9 + F(Q_{12}, Q_{11}, Q_{10}) + m_{12} + 1)^{\lll 3}. \end{aligned}$$

Hence, for Q_{13} and Q'_{13} we require that

$$Q'_{13}{}^{\lll 29} - Q_{13}{}^{\lll 29} \doteq 1.$$

By continuing this way the following requirements are identified:

$$\begin{aligned} Q'_{13}{}^{\lll 29} - Q_{13}{}^{\lll 29} &\doteq 1 && (3.6) \\ Q'_{14}{}^{\lll 25} - Q_{14}{}^{\lll 25} &\doteq F(Q'_{13}, Q_{12}, Q_{11}) - F(Q_{13}, Q_{12}, Q_{11}) && (3.7) \\ Q'_{15}{}^{\lll 21} - Q_{15}{}^{\lll 21} &\doteq F(Q'_{14}, Q'_{13}, Q_{12}) - F(Q_{14}, Q_{13}, Q_{12}) && (3.8) \\ Q'_{16}{}^{\lll 13} - Q_{16}{}^{\lll 13} &\doteq F(Q'_{15}, Q'_{14}, Q'_{13}) - F(Q_{15}, Q_{14}, Q_{13}) && (3.9) \\ Q'_{13} - Q_{13} &\doteq G(Q_{16}, Q_{15}, Q_{14}) - G(Q'_{16}, Q'_{15}, Q'_{14}) && (3.10) \\ Q'_{14} - Q_{14} &\doteq G(Q_{17}, Q_{16}, Q_{15}) - G(Q_{17}, Q'_{16}, Q'_{15}) && (3.11) \\ Q'_{19}{}^{\lll 23} - Q_{19}{}^{\lll 23} &\doteq Q'_{15} + G(Q_{18}, Q_{17}, Q'_{16}) - && (3.12) \\ & \quad Q_{15} - G(Q_{18}, Q_{17}, Q_{16}) && \\ Q'_{20}{}^{\lll 19} - Q_{20}{}^{\lll 19} &\doteq Q'_{16} + G(Q'_{19}, Q_{18}, Q_{17}) + 1 - && (3.13) \\ & \quad Q_{16} - G(Q_{19}, Q_{18}, Q_{17}) && \end{aligned}$$

Here, of course,

$$\begin{aligned} Q'_{20} - Q_{20} &\doteq -2^{25} \quad \text{and} && (3.14) \\ Q'_{19} - Q_{19} &\doteq 2^5 && (3.15) \end{aligned}$$

as required in the previous section.

Since equations (3.6)-(3.13) have 14 unknowns, we have 6 degrees of freedom. From (3.6)-(3.8) we see that good choices of Q_{13}, Q'_{13} and Q_{12} are

$$\begin{aligned} Q_{13} &= -1 \\ Q'_{13} &= 0 \\ Q_{12} &= 0, \end{aligned}$$

since then (3.6) is satisfied, and F being the if function, (3.7) and (3.8) can be simplified quite a lot to

$$\begin{aligned} Q_{11} &\doteq Q'_{14}{}^{\lll 25} - Q_{14}{}^{\lll 25} \quad \text{and} && (3.16) \\ Q_{14} &\doteq Q_{15}{}^{\lll 21} - Q'_{15}{}^{\lll 21}. && (3.17) \end{aligned}$$

The requirements (3.11), (3.12) and (3.13) respectively can be rewritten as follows:

$$
\begin{aligned}
Q'_{14} &\doteq Q_{14} - G(Q_{17}, Q'_{16}, Q'_{15}) + G(Q_{17}, Q_{16}, Q_{15}) && (3.18)\\
Q'_{15} &\doteq Q_{15} - G(Q_{18}, Q_{17}, Q'_{16}) + G(Q_{18}, Q_{17}, Q_{16}) + && (3.19)\\
&\quad {Q'_{19}}^{\lll 23} - {Q_{19}}^{\lll 23} &&\\
Q'_{16} &\doteq Q_{16} - G(Q'_{19}, Q_{18}, Q_{17}) + G(Q_{19}, Q_{18}, Q_{17}) + && (3.20)\\
&\quad {Q'_{20}}^{\lll 19} - {Q_{20}}^{\lll 19} - 1 &&
\end{aligned}
$$

The only two requirements that we have not touched yet are (3.10) and (3.9). These can be rewritten into

$$
\begin{aligned}
G(Q_{16}, Q_{15}, Q_{14}) - G(Q'_{16}, Q'_{15}, Q'_{14}) &\doteq 1 && (3.21)\\
F(Q'_{15}, Q'_{14}, 0) - F(Q_{15}, Q_{14}, -1) - {Q'_{16}}^{\lll 13} + {Q_{16}}^{\lll 13} &\doteq 0 && (3.22)
\end{aligned}
$$

For future reference let ζ denote the left-hand side of (3.22).

Now, we can choose Q_i, $i \in \{15, \ldots, 20\}$, compute Q_{14}, Q'_{14}, Q'_{15}, and Q'_{16} from (3.17)-(3.20), and then check that (3.21) and (3.22) hold. If they do, and if the solution is admissible (3.2), we have found the desired inner almost-collision.

Assuming that we have done this, if we choose m_{13} arbitrarily and compute Q_{11} from (3.16), the other input words used in steps 12–19, and the values Q_9 and Q_{10} can be computed from the step operations:

$$
\begin{aligned}
m_{12} &\leftarrow {Q_{20}}^{\lll 19} - Q_{16} - G(Q_{19}, Q_{18}, Q_{17}) - k_1 && (3.23)\\
m_{14} &\leftarrow {Q_{15}}^{\lll 21} - Q_{11} - Q_{14} && (3.24)\\
m_{15} &\leftarrow {Q_{16}}^{\lll 13} - F(Q_{15}, Q_{14}, -1) && (3.25)\\
m_{0} &\leftarrow {Q_{17}}^{\lll 29} + 1 - G(Q_{16}, Q_{15}, Q_{14}) - k_1 && (3.26)\\
m_{4} &\leftarrow {Q_{18}}^{\lll 27} - Q_{14} - G(Q_{17}, Q_{16}, Q_{15}) - k_1 && (3.27)\\
m_{8} &\leftarrow {Q_{19}}^{\lll 23} - Q_{15} - G(Q_{18}, Q_{17}, Q_{16}) - k_1 && (3.28)\\
Q_{10} &\leftarrow {Q_{14}}^{\lll 25} - m_{13} && (3.29)\\
Q_{9} &\leftarrow -1 - Q_{10} - m_{12} && (3.30)
\end{aligned}
$$

3.2.3 Right initial value

All that is left is to find values of the un-assigned input words so that the step variables into step 12 are right. This can be done deterministically by looking at the assignments that take place in each step. Don't forget that m_0, m_4, and m_8 are fixed at this point. We can choose m_1, m_2, m_3 and m_5 randomly and then compute Q_i, $0 < i \leq 6$.

The value of Q_9 that we are aiming for is computed in step 8 as

$$
Q_9 \leftarrow (Q_5 + F(Q_8, Q_7, Q_6) + m_8)^{\lll 3}. \qquad (3.31)
$$

To make things simple, we fix Q_8 at -1, so

$$
F(Q_8, Q_7, Q_6) = Q_7,
$$

and then we fix Q_7:

$$Q_7 = Q_9^{\lll 29} - Q_5 - m_8$$

– which causes (3.31) to be satisfied.

Now, we must make sure that Q_7 and Q_8 are assigned these values. This is done by setting m_6 and m_7 correctly. The operations performed in steps 6 and 7 are

$$\begin{aligned} Q_7 &\leftarrow (Q_3 + F(Q_6, Q_5, Q_4) + m_6)^{\lll 11} \quad \text{and} \\ Q_8 &\leftarrow (Q_4 + F(Q_7, Q_6, Q_5) + m_7)^{\lll 19}. \end{aligned}$$

Hence Q_7 has the right value if

$$m_6 \leftarrow (Q_9^{\lll 29} - Q_5 - m_8)^{\lll 21} - Q_3 - F(Q_6, Q_5, Q_4), \tag{3.32}$$

and Q_8 gets the correct value if

$$m_7 \leftarrow -1 - Q_4 - F(Q_7, Q_6, Q_5). \tag{3.33}$$

To make sure that Q_{10} is correct, we observe that in step 9,

$$Q_{10} \leftarrow (Q_6 + F(Q_9, Q_8, Q_7) + m_9)^{\lll 7}.$$

This means that we must fix m_9 as follows:

$$m_9 \leftarrow Q_{10}^{\lll 25} - Q_6 - F(Q_9, -1, Q_7). \tag{3.34}$$

To obtain the correct Q_{11}, we look at step 10:

$$Q_{11} \leftarrow (Q_7 + F(Q_{10}, Q_9, Q_8) + m_{10})^{\lll 11}.$$

Hence, m_{10} must be assigned the value

$$m_{10} \leftarrow Q_{11}^{\lll 21} - Q_7 - F(Q_{10}, Q_9, -1). \tag{3.35}$$

Finally, we make sure that Q_{12} is correct. In step 11,

$$Q_{12} \leftarrow (-1 + F(Q_{11}, Q_{10}, Q_9) + m_{11})^{\lll 19},$$

and therefore for m_{11} we get

$$m_{11} \leftarrow Q_{12}^{\lll 13} + 1 - F(Q_{11}, Q_{10}, Q_9). \tag{3.36}$$

We are now ready to state the actual algorithm that finds collisions for the MD4 hash function.

3.3 The algorithm

Dobbertin's collision attack can be stated as in Algorithm 3.2. Here, $c(\zeta)$ is the number of times that 16 divides ζ, where ζ is the 32-bit word defined in Section 3.2.2 (hence, $c(\zeta) = 8 \Rightarrow \zeta = 0$).

Note that in practice it is a good idea to introduce counters that ensure that a bad set of step variables chosen in the beginning does not cause the program to deadlock. For instance, one may introduce a criteria that the program tries to flip bits of $Q_{15}, \ldots, Q_{20}$ at most 100000 times before selecting completely new values of these step variables. Also, one may limit the number of times that the program chooses $m_1, \ldots, m_5$ before starting all over.

The algorithm has been implemented in the C programming language. This implementation (which can be found in Appendix B) was able to find collisions in 1.97 seconds on average on a standard PC (based on a sample of 1000 collisions).

Algorithm 3.2 Dobbertin's MD4 collision attack

Ensure: m and m', where $m'_i = m_i$ for all i except 12, and $m'_{12} = m_{12} + 1$, form a collision of MD4
  repeat
    repeat
      Choose $Q_{15}, \ldots, Q_{20}$ at random, and from (3.14), (3.15), (3.20), (3.19), (3.17) and (3.18) compute $Q'_{20}, Q'_{19}, Q'_{16}, Q'_{15}, Q_{14}$ and Q'_{14} (in that order).
    until (3.21) holds
    Save $Q_{15}, \ldots, Q_{20}$ as basic values.
    while $c(\zeta) < 8$ **do**
      repeat
        Change one random bit in each of the basic values, and compute $Q'_{20}, Q'_{19}, Q'_{16}, Q'_{15}, Q_{14}$ and Q'_{14} again.
      until (3.21) holds and $c(\zeta)$ does not decrease
      Save these $Q_{15}, \ldots, Q_{20}$ as new basic values.
    end while $\{$*Now (3.21) and (3.22) are satisfied, i.e. we have an inner-almost collision*$\}$
  until $G(Q_{20}, Q_{19}, Q_{18}) = G(Q'_{20}, Q'_{19}, Q_{18})$ $\{$*Now the inner-almost collision is admissible*$\}$
  Choose m_{13} at random, compute Q_{11} from (3.16), and m_i ($i \in \{0, 4, 8, 12, 14, 15\}$), Q_{10} and Q_9 from (3.23), ..., (3.30).
  repeat
    Choose m_1, m_2, m_3, m_5 randomly, compute $Q_1, \ldots, Q_6$ from the step operations, and compute $m_6, m_7, m_9, m_{10}, m_{11}$ from (3.32), ..., (3.36).
  until m and m' form a collision

3.4 Summary

Dobbertin's attack on MD4 has been described. The attack finds collisions of MD4 in about two seconds on a standard PC. An implementation can be found in Appendix B.

A faster technique for finding collisions of MD4 has been developed [43] since the attack of Dobbertin. This attack has complexity 2^8 according to the authors and thus can be carried out in a fraction of a second. It makes use of a different differential pattern than the one found by Dobbertin, and like the attack mentioned in the following chapter, it also makes use of message modification. An implementation can be found in Appendix D, but it comes with no further explanation (although the same techniques are used in the MD5 attack explained in the following chapter). This implementation has complexity 2^{10} and finds MD4 collisions in around 2 ms on a standard PC, which makes it about 1000 times faster than the Dobbertin implementation.

Chapter 4

Wang's attack on MD5

In this chapter a collision attack on MD5 [38] is presented. The attack was developed by Wang et al. [45], and it has complexity at most 2^{37} according to the authors. A modification by Vlastimil Klíma [25] improves the complexity slightly.

First, the MD5 hash function is described. Then, the general idea of the attack is explained, and after that we shall go deeper into the details, and also describe issues related to an implementation of the attack.

4.1 Description of MD5

MD5 is a hash function that takes as input a message of arbitrary length and returns a hash value of 128 bits. Each message block is 512 bits, padded as in Rule 1 with $u = 64$ (the least significant u bits of $|m|$ are used in the MD-strengthening). The overall construction is that of Construction 1.

The initial value is

$$\begin{aligned} Q_{-3} &= \texttt{67452301}_h \\ Q_{-2} &= \texttt{10325476}_h \\ Q_{-1} &= \texttt{98badcfe}_h \\ Q_0 &= \texttt{efcdab89}_h \end{aligned}$$

The compression function of MD5 takes as input the 128-bit value (Q_{-3}, Q_{-2}, Q_{-1}, Q_0) and a 512-bit message block which is split into sixteen 32-bit words, $m_0, \ldots, m_{15}$.

In each step t ($0 \leq t < 64$) of the compression function a constant, k_t, is used in the step operation. This constant can be computed as follows:

$$k_t \leftarrow \lfloor \text{abs}(\sin(t+1)) \times 2^{32} \rfloor.$$

Of course, in a software implementation one would usually precompute these values and place them in an array.

The order in which the message words are processed can also be described mathematically. Let W_t be the message word used in step t. Then $W_t = m_i$ where

$$i = \begin{cases} t & \text{for } t < 16 \\ (5t+1) \bmod 16 & \text{for } 16 \le t < 32 \\ (3t+5) \bmod 16 & \text{for } 32 \le t < 48 \\ 7t \bmod 16 & \text{for } 48 \le t < 64 \end{cases}$$

We often call steps 0–15 *round 1*, steps 16–31 *round 2* etc.

Each step also contains a rotation value, S_t, which can be derived from the table below.

S_t		t div 16			
		0	1	2	3
t mod 4	0	7	5	4	6
	1	12	9	11	10
	2	17	14	16	15
	3	22	20	23	21

In each step a bitwise function f_t of three variables is used. The definition of these is

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

The compression function performs the operations of Algorithm 4.1.

Algorithm 4.1 The MD5 compression function

Input: $(Q_{-3}, Q_{-2}, Q_{-1}, Q_0)$ and $m_0, \ldots, m_{15}$
  for $t = 0$ to 63 **do**
    $Q_{t+1} \leftarrow Q_t + (f_t(Q_t, Q_{t-1}, Q_{t-2}) + Q_{t-3} + k_t + W_t)^{\lll S_t}$
  end for
  return $(Q_{61} + Q_{-3}, Q_{62} + Q_{-2}, Q_{63} + Q_{-1}, Q_{64} + Q_0)$

4.2 Introduction to the attack

The attack makes use of differential cryptanalysis (see [3]), using not the ordinary XOR differential, but rather a combination of a kind of signed XOR differential and a modular differential. What is meant by a signed XOR differential is that one keeps track of not only which bits differ, but also whether the bit difference is positive ($\texttt{0} \rightarrow \texttt{1}$) or negative ($\texttt{1} \rightarrow \texttt{0}$).

Wang somehow found a differential yielding with high probability a collision on two two-block messages. For each step a particular characteristic must hold, and this gives rise to a number of conditions on the intermediate register values.

The number of bit conditions is much higher than 64, and since a collision can be expected after about 2^{64} executions of the hash function using exhaustive search, this doesn't seem very promising. However, a large number of the conditions can be ensured to hold by performing modifications of the message. These will be described in the following.

As mentioned, collisions found by this attack consist of two-block messages. In the following we use the term *iteration* for the processing of a block, e.g. *the first iteration* means the processing of the first message block. Similarly, *the first part of the attack* means finding a usable first message block. When we refer to *the two messages* we mean the colliding messages.

Let $M = m_0\|m_1\|\cdots\|m_{15}$ and $\hat{M} = \hat{m}_0\|\hat{m}_1\|\cdots\|\hat{m}_{15}$ be the two blocks of the "first" message, and let $M' = m'_0\|m'_1\|\cdots\|m'_{15}$ and $\hat{M}' = \hat{m}'_0\|\hat{m}'_1\|\cdots\|\hat{m}'_{15}$ be the two blocks of the colliding message. Then

$$\delta m_i = \begin{cases} 2^{31} & \text{for } i \in \{4, 14\} \\ 2^{15} & \text{for } i = 11 \\ 0 & \text{for all other } i \end{cases} \tag{4.1}$$

and

$$\delta \hat{m}_i = \begin{cases} 2^{31} & \text{for } i \in \{4, 14\} \\ -2^{15} & \text{for } i = 11 \\ 0 & \text{for all other } i \end{cases} \tag{4.2}$$

The modular difference on the chaining values after processing the first block is

$$\delta Q_i = \begin{cases} 2^{31} & \text{for } i = -3 \\ 2^{25} + 2^{31} & \text{for } i \in \{0, -1, -2\} \end{cases} \tag{4.3}$$

Both blocks produce intermediate collisions from step 23 to step 34, i.e. in both iterations, $Q_i = Q'_i$ for $24 \leq i \leq 35$.

4.3 The idea

The general idea of the attack will now be described.

For the first part of the attack do the following:

- Choose a random message block M.
- For each step, do the step operation, and ensure that the conditions on the step variable are satisfied by performing modification of the message word used. If some conditions cannot be satisfied this way, start over.
- Check that all required differences hold. If so, we have found a usable near-collision. Otherwise, start over.

For the second part of the attack, do the following:

- Initialise the registers using the chaining values from the first iteration.
- Choose a random message block $\hat{M}$.
- For each step, do the step operation, and ensure that the conditions on the step variable are satisfied by performing modification of the message word used. If some conditions cannot be satisfied this way, start over.
- Check that all required differences hold. If so, we have found a collision. Otherwise, start over.

After having successfully performed the steps above, we have found a collision between two two-block messages, namely $(M, \hat{M})$ and $(M', \hat{M}')$, where (4.1) and (4.2) hold.

4.4 Details

In this section (hopefully) all the details, possible pitfalls and some tricks of the attack shall be explained. It shall generally be assumed that the reader is either familiar with [45] or has the paper close at hand (note that in this context numbering starts from 0, whereas in [45] numbering starts from 1).

For convenience we define

$$T_t = f_t(Q_t, Q_{t-1}, Q_{t-2}) + Q_{t-3} + k_t + W_t$$

and

$$R_t = T_t^{\lll S_t},$$

and hence $Q_{t+1} = Q_t + T_t^{\lll S_t} = Q_t + R_t$.

4.4.1 Origin of the conditions

As mentioned, a very specific characteristic must hold after each step for the full attack to work. The characteristics can, of course, be found in [45], but they are more carefully explained and slightly corrected in [21] for the first iteration, and Section 4.5 of this document gives a fairly detailed description of the conditions of the second iteration. All the conditions that are considered "correct" in this document can be found in Appendix A.

For the characteristic to hold after each step, several conditions on the step variables must be satisfied. We now consider the first iteration. For instance, in step 11 the operation performed is

$$Q_{12} \leftarrow Q_{11} + (F(Q_{11}, Q_{10}, Q_9) + Q_8 + k_{11} + m_{11})^{\lll 22},$$

and we have the following (required) differences on the values involved:

$$
\begin{aligned}
\delta Q_{12} &\doteq -2^7 - 2^{13} + 2^{31} \\
\delta Q_{11} &\doteq 2^{30} + 2^{31} \\
\nabla Q_{11} &\doteq \langle 30, 31 \rangle \\
\nabla Q_{10} &\doteq \langle -12, 13, 31 \rangle \\
\nabla Q_{9} &\doteq \langle -0, 1, 6, 7, -8, -31 \rangle \\
\delta Q_{8} &\doteq 2^0 - 2^{15} - 2^{17} - 2^{23} \\
\delta m_{11} &= 2^{15}
\end{aligned}
$$

Since $\delta Q_{12} - \delta Q_{11} = -2^7 - 2^{13} - 2^{30}$ we require

$$\delta T_{11} = \delta(F(Q_{11}, Q_{10}, Q_9) + Q_8 + k_{11} + m_{11}) \doteq -2^8 - 2^{17} - 2^{23},$$

and since we have $\delta(Q_8 + m_{11}) = 2^0 - 2^{17} - 2^{23}$, we require that

$$\delta f_{11} \doteq -2^0 - 2^8.$$

This is achieved by applying certain conditions on bits 0 and 8 of Q_{11}, Q_{10} and Q_9. Note that F is the `if` function, meaning that when Q_{11} is a `1` then the bit of Q_{10} at that bit position is chosen as output, and otherwise the bit of Q_9 at that bit position is chosen. Since we have $\nabla Q_9 \doteq \langle -0, 1, 6, 7, -8, -31 \rangle$, if we make sure that $Q_{11}[0] = Q_{11}[8] = 0$, then we get at least $\delta f_{11} = -2^0 - 2^8$. Since there are also differences on other bits involved in f_{11}, we must make sure that these differences *disappear* in the function. This is done by ensuring that $Q_{11}[12] = Q_{11}[13] = 0$, $Q_{11}[1] = Q_{11}[6] = Q_{11}[7] = 1$, $Q_{10}[30] = Q_9[30]$, and since the differences on bit 31 of Q_{10} and Q_9 have opposite signs, we know that the difference on $Q_{11}[31]$ disappears.

In order for the signed XOR differences to be possible we must further require that e.g. $Q_{11}[30] = 0$ and $Q_{10}[12] = 1$. Note that as discovered in [21] we do not strictly require that there is a negative signed XOR difference on $Q_9[31]$, and a positive one on $Q_{10}[31]$ and $Q_{11}[31]$, it could be vice versa (see e.g. [21, Table 11]).

Now some of the conditions on the involved variables have been explained. Other steps cause other conditions. Note that we optimistically expect that

$$\delta T_{11} = -2^8 - 2^{17} - 2^{23} \Rightarrow \delta R_{11} = -2^7 - 2^{13} - 2^{30}.$$

This is indeed the case for most values of T_{11}, but not all. This is a fairly important issue, which will be covered in greater detail in Section 4.4.3. For now, we always assume that

$$T' - T = \delta T \Rightarrow (T')^{\lll S} - T^{\lll S} = (\delta T)^{\lll S}.$$

4.4.2 Message modification

As mentioned, most of the conditions on step variables can be ensured to hold by performing message modifications. These message modifications were only described superficially in [45], and a little more in depth in [43]. The description given here is more detailed, and the additional ideas are the author's own.

Conditions in the *first round* of the compression function can be ensured by choosing Q_{t+1} randomly, but satisfying the conditions, and then applying the following single-message modification:

$$m_t \leftarrow (Q_{t+1} - Q_t)^{\ggg S_t} - F(Q_t, Q_{t-1}, Q_{t-2}) - Q_{t-3} - k_t. \tag{4.4}$$

Hence, we do not need to choose m_t randomly, then compute Q_{t+1}, then correct Q_{t+1} to satisfy the conditions, and finally recompute m_t (as suggested in [45]).

This method only works in the first round, because once m_t is defined we cannot simply redefine it. Therefore, message modifications in the second round must be performed in some other way.

In the second round we compute Q_{t+1} and check if the conditions hold, starting with the lowest bit number. If some condition does not hold, we may be able to correct Q_{t+1} as follows: Let $W_t = m_u$, i.e. W_t was first used in step u. The step operation of step u is

$$Q_{u+1} = Q_u + (F(Q_u, Q_{u-1}, Q_{u-2}) + Q_{u-3} + k_u + m_u)^{\lll S_u}. \tag{4.5}$$

If we need to correct $Q_{t+1}[n]$, then we may change $m_u[n - S_t]$. However, this causes Q_{u+1} to change, so this is only possible if there is no condition on $Q_{u+1}[n - S_t + S_u]$. Assuming there is no condition on that bit, we prevent the change of m_u from causing changes in other bits of Q_{u+1} by checking if $Q_{u+1}[n - S_t + S_u]$ is a `0` or a `1`, and then we add respectively subtract $2^{(n-S_t) \bmod 32}$ from m_u, causing a change on the desired bit of m_u.

If there is a condition on $Q_{u+1}[n - S_t + S_u]$ we have to try something else. Looking again at the step operation (4.5) of step u, we may instead (provided there are no conditions preventing this) flip $Q_{u-3}[n - S_t]$ by adding or subtracting $2^{(n-S_t) \bmod 32}$ (if $Q_{u-3}[n - S_t]$ is a `0` or a `1` respectively), and then respectively subtract or add $2^{(n-S_t) \bmod 32}$ to m_u, causing these two changes to cancel out. Now we have changed the desired bit of m_u without changing $m_u + Q_{u-3}$.

Of course, in both the mentioned cases we must subsequently perform single-message modification (according to (4.4)) on all the message words that take part in the same step operation as the step variable that we changed, i.e. in the first case we perform modifications on m_{u+1}, m_{u+2}, m_{u+3} and m_{u+4}, and in the second case we perform modifications on m_{u-1}, m_{u-2}, m_{u-3} and m_{u-4}. If one of these subscripts to m is less than 0 or greater than 15, then the whole operation is impossible.

Even in cases where none of the methods mentioned so far can be used, we may have a chance of correcting $Q_{t+1}[n]$. In (4.5) we have a few other variables to try. However, all the remaining variables occur inside the F function, so a change only

has the desired effect if we are able to control bit $n-S_t$ of Q_u, Q_{u-1} and Q_{u-2}, or these bits just happen to have the right conditions on them for our purposes. If, for instance, we are able to correct $Q_{u-1}[n-S_t]$, then this only makes a difference if $Q_u[n-S_t] = 1$ (since F is the if function in the first round). It may even be possible to change $Q_u[n-S_t]$, but since Q_u occurs twice in (4.5), this is fairly complicated, as it will affect $Q_{u+1}[n-S_t]$. The best thing would be to avoid changes in more than one bit of one step variable.
Now it would probably be in its place with an example.

Example. There are three conditions on Q_{18} in the first iteration (on bits 17, 29, and 31). Assume that we just computed Q_{18} and found that $Q_{18}[17]$ was correct, but $Q_{18}[29]$ was not. The step operation of step 17 is

$$Q_{18} = Q_{17} + (G(Q_{17}, Q_{16}, Q_{15}) + Q_{14} + t_{17} + m_6)^{\lll 9},$$

so we have to change $m_6[20]$. The step operation in step 6 is

$$Q_7 = Q_6 + (F(Q_6, Q_5, Q_4) + Q_3 + k_6 + m_6)^{\lll 17}$$

Just changing $m_6[20]$ would affect $Q_7[5]$, but there are already conditions on all bits of Q_7, so we must try something else. Let's try changing Q_3 as well. There is no condition on $Q_3[20]$, so we flip this bit, but since $Q_4[20] \doteq Q_3[20]$, we need to flip $Q_4[20]$ as well. Q_4 takes part in the F function of step 6, so this only works if $Q_6[20] = 1$ (and hence $Q_4[20]$ does not affect this step at all), which holds if the conditions on Q_6 are satisfied. Now, since the change on Q_4 has no effect in this step, we can add or subtract 2^{20} from m_6 depending on whether flipping $Q_3[20]$ was equivalent to respectively subtracting or adding 2^{20} to Q_3 – this way we make sure that $Q_3 + m_6$ does not change by this operation, but $m_6[20]$ has changed as desired. Of course, we must also perform single-message modification on m_2, m_3, m_4 and m_5, i.e. we compute

$$\begin{aligned}
m_2 &\leftarrow (Q_3 - Q_2)^{\ggg 17} - F(Q_2, Q_1, Q_0) - Q_{-1} - k_2 \\
m_3 &\leftarrow (Q_4 - Q_3)^{\ggg 22} - F(Q_3, Q_2, Q_1) - Q_0 - k_3 \\
m_4 &\leftarrow (Q_5 - Q_4)^{\ggg 7} - F(Q_4, Q_3, Q_2) - Q_1 - k_4 \\
m_5 &\leftarrow (Q_6 - Q_5)^{\ggg 12} - F(Q_5, Q_4, Q_3) - Q_2 - k_5
\end{aligned}$$

We then re-compute Q_{18}, where bit 29 is now correct. Since this change could propagate to bits of higher order, we have to correct low-order bits before high-order bits.

Note that we would not be able to perform this particular multi-message modification on $Q_{18}[17]$, because we would have to change $m_6[8]$, $Q_3[8]$, and $Q_4[8]$, and since $Q_6[8] = 0$, this would cause the outcome of step 6 to change, which is not allowed since all bits of Q_7 are fixed. However, $Q_{18}[31]$ can be changed in the same way as $Q_{18}[29]$.

4.4.3 Propagating carries

The issues of this section were not addressed by Wang et al. [43, 45]. In [21] the processing of the first block of the message is explained in detail with respect to the added conditions caused by these issues.

In the Wang MD5 attack we often expect a difference δT_t on T_t to "survive" a rotation, meaning that

$$T_t' - T_t = \delta T_t \Rightarrow (T_t')^{\lll S_t} - T_t^{\lll S_t} = (\delta T_t)^{\lll S_t}, \tag{4.6}$$

or equivalently

$$(\delta T_t)^{\lll S_t} = \delta R_t.$$

However, this does not always hold. For instance, if a carry caused by δT_t propagates from the low-order bits past bit $(31 - S_t)$, or if a carry from the high-order bits propagates past bit 31 (we call these two values *limits*, which means we can talk about both in the same way), then the propagation is in a sense "split in two" by the rotation, and then (4.6) does not hold. Hence, since we expect the equation to hold, we must make sure that no carries propagate too far.
Note that Daum [10] also describes these issues, but here we use a different approach in that we try to define exact conditions that must hold for (4.6) to hold, instead of determining the probability that (4.6) holds. The following analysis is therefore the author's own.

There are a number of ways of ensuring that (4.6) holds. If, for instance, $\delta T_t = 2^8 - 2^{10} + 2^{29}$, and $S_t = 17$, then we can prevent the carry caused by the term 2^8 from propagating past bit 9, the carry caused by the term -2^{10} from propagating past bit 14, and the carry caused by the term 2^{29} from propagating past bit 31, and (4.6) will be satisfied. This can be done by ensuring that

1. $\texttt{0} \in T_t[8-9]$,
2. $\texttt{1} \in T_t[10-14]$, *and*
3. $\texttt{0} \in T_t[29-31]$.

For random words, this occurs with a total probability of

$$P = (1-2^{-2})(1-2^{-5})(1-2^{-3}) \approx 0.64.$$

The conditions imposed are sufficient, but not necessary for (4.6) to hold. In fact, a test will show that if (1) *does not* hold, then (2) is a redundant condition, and if (1) holds then (2) *must* hold also. Why so?

When we talk about carries propagating or not propagating past some bit, we are really not being precise enough; it depends if we are looking at the arithmetic operation as an addition or a subtraction. If we look at it as an addition, then a carry caused by a power of 2 which is negative *must* propagate past the limit. We could also look at this as a subtraction by a positive power of 2, and in this case the carry

must not propagate past the limit. However, it is an advantage to group terms into those with exponent less than or equal to $31 - S_t$ and those with exponent greater than $31 - S_t$, and then always consider an addition by these terms. When we do this, we must make sure that if the term with the greatest exponent within each of the two "buckets" is positive, then a carry does not propagate past the limit, and if the term is negative then a carry *does* propagate past the limit.

In the example above, we need a carry to propagate past bit 14 when we consider addition, and that is why either a carry must propagate from bits 8 and 9 into bit 10, from where it will keep propagating past bit 14 (which is the case when (1) does not hold), *or* a carry must be produced in bits 10–14, which happens whenever (2) holds. The probability that these two conditions as well as (3) hold is $P = (1-(1-2^{-2}) \times 2^{-5}) \times (1-2^{-3}) \approx 0.85$, which is quite a lot better than $P = 0.64$ as we had before.

Conversely, assume instead that $\delta T_t = -2^8 + 2^{10} - 2^{29}$ (notice that all signs have changed). Now, no carry must propagate past bit 14. For this to hold there must be a `0` among bits 10–14 *or* bits 8 and 9 must both be `0`. First note that $2^{10} - 2^8 = 2^9 + 2^8$. Hence, the first condition ensures that any carry out of bit 9 is "caught". The second condition ensures that there *is* no carry out of bit 9. Obviously, for the high-order bits, we now require that there is a `1` among bits 29–31.

In the cases where a bucket of terms consists of all negative terms, for instance two negative terms, the rules are slightly different. For instance, in the running example, if all signs were negative (i.e. $\delta T_t = -2^8 - 2^{10} - 2^{29}$) then we would need a carry to propagate past bit 14. The terms $-2^8 - 2^{10}$ could instead be expressed as $2^8 + 2^9 + 2^{11} + 2^{12} + \ldots$, so in order for a carry to propagate past bit 14 we need either a `1` among bits 11–14, or a `1` in bit 10 *and* a `1` among bits 8–9.

To summarize, a technique for stating conditions could be expressed as follows:

- Place terms of δT_t in two buckets, one with powers of two with exponents greater than $31 - S_t$ (the limit for this bucket is 31), and one with the remaining terms (the limit for this bucket is $31 - S_t$).
- For each bucket, check the sign of the term with the greatest exponent. If it is positive, a carry must not propagate past the limit, and if it is negative a carry *must* propagate past the limit.
- Express the sum of the terms of each bucket as a sum of positive terms only. Note that this is always possible since we reduce modulo 2^{32} – this is actually equivalent to looking at the bit representation of δT_t: 2^s is a term if and only if $T_t[s] = 1$.
- Check which conditions must hold on T_t for a carry to propagate or not propagate past the limit.

It is the author's hope that it is clear from the examples how to perform the last step.

4.5 Detailed examination of the second iteration

In this section we perform a fairly detailed examination of all the conditions that must be fulfilled in the second iteration for the attack to work. The examination will be done in much the same way (although perhaps not quite as thoroughly) as the examination of [21], which only concerned the first part of the attack. It is definitely with this fine work in mind that this section is written. Note that although it may be instructive to follow this section thoroughly, it is not imperative for the understanding of the attack.

We start by looking at the conditions on the T-values. In some cases the exact conditions on the T-values become quite complicated, and then we shall define more strict conditions, that are sufficient but not always necessary – however, we aim at defining conditions that are not much less probable than the "optimal" ones.

4.5.1 Conditions on the T-values

Step 0 $Q_1 \leftarrow Q_0 + (F(Q_0, Q_{-1}, Q_{-2}) + Q_{-3} + \hat{m}_0 + k_0)^{\lll 7}$
We have $\delta Q_0 = 2^{25} + 2^{31}$. We need $Q_1 = 2^{25} + 2^{31}$.
Hence, $\delta T_0 \doteq 0$.
Conditions on T_0: None.

Step 1 $Q_2 \leftarrow Q_1 + (F(Q_1, Q_0, Q_{-1}) + Q_{-2} + \hat{m}_1 + k_1)^{\lll 12}$
We have $\delta Q_1 = 2^{25} + 2^{31}$. We need $\delta Q_2 = 2^5 + 2^{25} + 2^{31}$.
Hence, $\delta T_1 \doteq 2^{25}$.
Conditions on T_1: $\delta T_1 = 2^{25}$ must not propagate past bit 31, and hence we require $\mathtt{0} \in T[25-31]$ $(P = 1 - 2^{-7})$.

Step 2 $Q_3 \leftarrow Q_2 + (F(Q_2, Q_1, Q_0) + Q_{-1} + \hat{m}_2 + k_2)^{\lll 17}$
We have $\delta Q_2 = 2^5 + 2^{25} + 2^{31}$. We need $\delta Q_3 = 2^5 + 2^{11} + 2^{16} + 2^{25} + 2^{31}$.
Hence, $\delta T_2 \doteq 2^{26} + 2^{31}$.
Conditions on T_2: 2^{26} must not propagate past bit 30, and $(2^{31})^{\lll 17} \doteq +2^{16}$, so we must require $\mathtt{0} \in T[26-30]$ $(P = 1 - 2^{-5})$ and $T[31] = \mathtt{0}$ $(P = 2^{-1})$.

Step 3 $Q_4 \leftarrow Q_3 + (F(Q_3, Q_2, Q_1) + Q_0 + \hat{m}_3 + k_3)^{\lll 22}$
We have $\delta Q_3 = 2^5 + 2^{11} + 2^{16} + 2^{25} + 2^{31}$. We need $\delta Q_4 = -2^1 + 2^5 + 2^{25} + 2^{31}$.
Hence, $\delta T_3 \doteq -2^{11} - 2^{21} - 2^{26}$.
Conditions on T_3: $-2^{11} - 2^{21} - 2^{26}$ must propagate past bit 31, so we must require $\mathtt{1} \in T[27-31]$ $(P = 1 - 2^{-5})$, or $T[26] = \mathtt{1} \wedge \mathtt{1} \in T[22-25]$, or a more complicated but less probable combination.

Step 4 $Q_5 \leftarrow Q_4 + (F(Q_4, Q_3, Q_2) + Q_1 + \hat{m}_4 + k_4)^{\lll 7}$
We have $\delta Q_4 = -2^1 + 2^5 + 2^{25} + 2^{31}$. We need $\delta Q_5 = 1 + 2^6 + 2^8 + 2^9 + 2^{31}$.
Hence, $\delta T_4 \doteq 2^1 + 2^2 - 2^{18} + 2^{25} + 2^{26} - 2^{30} + 2^{31}$.

Conditions on T_4: $2^1+2^2-2^{18}$ must propagate past bit 24, $2^{25}+2^{26}-2^{30}$ must propagate past bit 30, and $(2^{31})^{\lll 7} \doteq +2^6$. Hence, we must require $\texttt{0} \in T[2-18]$ ($P=1-2^{-17}$), or some other combination, but this one is very likely to occur. Also, we must require $T[30]=\texttt{1}$ ($P=2^{-1}$, or some other combination which is much less likely), and $T[31]=\texttt{0}$ ($P=2^{-1}$) to make sure that adding 2^{31} causes a positive change.

Step 5 $\quad Q_6 \leftarrow Q_5+(F(Q_5,Q_4,Q_3)+Q_2+\hat{m}_5+k_5)^{\lll 12}$
We have $\delta Q_5 = 1+2^6+2^8+2^9+2^{31}$. We need $\delta Q_6 = -2^{16}-2^{20}+2^{31}$.
Hence, $\delta T_5 \doteq -2^4-2^8-2^{20}-2^{26}-2^{28}-2^{29}$.
Conditions on T_5: -2^4-2^8 must propagate past bit 19, and $-2^{20}-2^{26}-2^{28}-2^{29}$ must propagate past bit 31. Hence, we require that $\texttt{1} \in T[9-19]$ ($P=1-2^{-11}$), and that $\texttt{1} \in T[30-31]$ or $T[29]=T[28]=T[27]=\texttt{1}$ ($P=1-(2^{-2})(1-2^{-3}) = 25/32$).

Step 6 $\quad Q_7 \leftarrow Q_6+(F(Q_6,Q_5,Q_4)+Q_3+\hat{m}_6+k_6)^{\lll 17}$
We have $\delta Q_6 = -2^{16}-2^{20}+2^{31}$. We need $\delta Q_7 = -2^6-2^{27}+2^{31}$.
Hence, $\delta T_6 \doteq 2^3-2^{10}-2^{21}+2^{31}$.
Conditions on T_6: 2^3-2^{10} must propagate past bit 14, -2^{21} must propagate past bit 30, and $(2^{31})^{\lll 17} \doteq +2^{16}$. This means that we require $\texttt{1} \in T[10-14]$ ($P=1-2^{-5}$), $\texttt{1} \in T[21-30]$ ($P=1-2^{-10}$), and that $T[31]=\texttt{0}$ ($P=2^{-1}$) so that adding 2^{31} causes a positive change.

Step 7 $\quad Q_8 \leftarrow Q_7+(F(Q_7,Q_6,Q_5)+Q_4+\hat{m}_7+k_7)^{\lll 22}$
We have $\delta Q_7 = -2^6-2^{27}+2^{31}$. We need $\delta Q_8 = 2^{15}-2^{17}-2^{23}+2^{31}$.
Hence, $\delta T_7 \doteq -2^1+2^5+2^{16}+2^{25}-2^{27}$.
Conditions on T_7: -2^1+2^5 must not propagate past bit 9, and $2^{16}+2^{25}-2^{27}$ must propagate past bit 31. Hence, we require that $\texttt{0} \in T[4-9]$ ($P=1-2^{-6}$), and that $\texttt{1} \in T[27-31]$ ($P=1-2^{-5}$).

Step 8 $\quad Q_9 \leftarrow Q_8+(F(Q_8,Q_7,Q_6)+Q_5+\hat{m}_8+k_8)^{\lll 7}$
We have $\delta Q_8 = 2^{15}-2^{17}-2^{23}+2^{31}$. We need $\delta Q_9 = 1+2^6+2^{31}$.
In this step, things are a little more complicated. Since we have the term 2^0 from δQ_5 which is not removed by δf_8, but we don't need 2^7 in δQ_9, we use $(-2^{31}+1)^{\lll 7} \stackrel{?}{\doteq} -2^6+2^7$ to create 2^6 in δQ_9. Also, we use $(2^8+2^9)^{\lll 7}$ to cancel out $2^{15}-2^{17}$. Hence, $\delta T_8 \doteq 1+2^8+2^9+2^{16}+2^{25}-2^{31}$.
Conditions on T_8: $1+2^8+2^9+2^{16}$ must not propagate past bit 24, 2^{25} must not propagate past bit 30, and $(2^{31})^{\lll 7} \doteq -2^6$. Hence, we require $\texttt{0} \in T[16-24]$ ($P=1-2^{-9}$), $\texttt{0} \in T[25-30]$ ($P=1-2^{-6}$), and $T[31]=\texttt{1}$ ($P=2^{-1}$).

Step 9 $\quad Q_{10} \leftarrow Q_9+(F(Q_9,Q_8,Q_7)+Q_6+\hat{m}_9+k_9)^{\lll 12}$
We have $\delta Q_9 = 1+2^6+2^{31}$. We need $\delta Q_{10} = 2^{12}+2^{31}$.
Hence, $\delta T_9 \doteq 2^0-2^{20}-2^{26}$.

Conditions on T_9: 2^0 must not propagate past bit 19, and $-2^{20}-2^{26}$ must propagate past bit 31. Hence, we require $0 \in T[0-19]$ ($P = 1-2^{-20}$) and $1 \in T[27-31]$ ($P = 1-2^{-5}$).

Step 10 $Q_{11} \leftarrow Q_{10} + (F(Q_{10}, Q_9, Q_8) + Q_7 + \hat{m}_{10} + k_{10})^{\lll 17}$
We have $\delta Q_{10} = 2^{12} + 2^{31}$. We need $\delta Q_{11} = 2^{31}$.
Hence, $\delta T_{10} \doteq -2^{27}$.
Conditions on T_{10}: -2^{27} must propagate past bit 31, so we require $1 \in T[27-31]$ ($P = 1-2^{-5}$).

Step 11 $Q_{12} \leftarrow Q_{11} + (F(Q_{11}, Q_{10}, Q_9) + Q_8 + \hat{m}_{11} + k_{11})^{\lll 22}$
We have $\delta Q_{11} = 2^{31}$. We need $\delta Q_{12} = -2^7 - 2^{13} + 2^{31}$.
Hence, $\delta T_{11} \doteq -2^{17} - 2^{23}$.
Conditions on T_{11}: $-2^{17} - 2^{23}$ must propagate past bit 31, so we require $1 \in T[24-31]$ ($P = 1-2^{-8}$).

Step 12 $Q_{13} \leftarrow Q_{12} + (F(Q_{12}, Q_{11}, Q_{10}) + Q_9 + \hat{m}_{12} + k_{12})^{\lll 7}$
We have $\delta Q_{12} = -2^7 - 2^{13} + 2^{31}$. We need $\delta Q_{13} = 2^{24} + 2^{31}$.
Hence, $\delta T_{12} \doteq 1 + 2^6 + 2^{17}$.
Conditions on T_{12}: $1 + 2^6 + 2^{17}$ must not propagate past bit 24. Hence we require $0 \in T[17-24]$ ($P = 1-2^{-8}$).

Step 13 $Q_{14} \leftarrow Q_{13} + (F(Q_{13}, Q_{12}, Q_{11}) + Q_{10} + \hat{m}_{13} + k_{13})^{\lll 12}$
We have $\delta Q_{13} = 2^{24} + 2^{31}$. We need $\delta Q_{14} = 2^{31}$.
Hence, $\delta T_{13} \doteq -2^{12}$.
Conditions on T_{13}: -2^{12} must propagate past bit 19, so we require $1 \in T[12-19]$ ($P = 1-2^{-8}$).

Step 14 $Q_{15} \leftarrow Q_{14} + (F(Q_{14}, Q_{13}, Q_{12}) + Q_{11} + \hat{m}_{14} + k_{14})^{\lll 17}$
We have $\delta Q_{14} = 2^{31}$. We need $\delta Q_{15} = 2^3 + 2^{15} + 2^{31}$.
Hence, $\delta T_{14} \doteq 2^{18} + 2^{30}$.
Conditions on T_{14}: $2^{18} + 2^{30}$ must not propagate past bit 31. Hence we require $0 \in T[30-31]$ ($P = 1-2^{-2}$).

Step 15 $Q_{16} \leftarrow Q_{15} + (F(Q_{15}, Q_{14}, Q_{13}) + Q_{12} + \hat{m}_{15} + k_{15})^{\lll 22}$
We have $\delta Q_{15} = 2^3 + 2^{15} + 2^{31}$. We need $\delta Q_{16} = -2^{29} + 2^{31}$.
Hence, $\delta T_{15} \doteq -2^7 - 2^{13} - 2^{25}$.
Conditions on T_{15}: -2^7 must propagate past bit 9, and $-2^{13} - 2^{25}$ must propagate past bit 31. Hence, we require $1 \in T[7-9]$ ($P = 1-2^{-3}$) and $1 \in T[26-31]$ ($P = 1-2^{-6}$).

Step 16 $Q_{17} \leftarrow Q_{16} + (G(Q_{16}, Q_{15}, Q_{14}) + Q_{13} + \hat{m}_1 + k_{16})^{\lll 5}$
We have $\delta Q_{16} = -2^{29} + 2^{31}$. We need $\delta Q_{17} = 2^{31}$.
Hence, $\delta T_{16} \doteq 2^{24}$.
Conditions on T_{16}: 2^{24} must not propagate past bit 26. Hence, we require $0 \in T[24-26]$ ($P = 1 - 2^{-3}$).

Step 17 $Q_{18} \leftarrow Q_{17} + (G(Q_{17}, Q_{16}, Q_{15}) + Q_{14} + \hat{m}_6 + k_{17})^{\lll 9}$
We have $\delta Q_{17} = 2^{31}$. We need $\delta Q_{18} = 2^{31}$.
Hence, $\delta T_{17} \doteq 0$.
Conditions on T_{17}: None.

Step 18 $Q_{19} \leftarrow Q_{18} + (G(Q_{18}, Q_{17}, Q_{16}) + Q_{15} + \hat{m}_{11} + k_{18})^{\lll 14}$
We have $\delta Q_{18} = 2^{31}$. We need $\delta Q_{19} = 2^{17} + 2^{31}$.
Hence, $\delta T_{18} \doteq 2^{3}$.
Conditions on T_{18}: 2^{3} must not propagate past bit 17, so we require $0 \in T[3-17]$ ($P = 1 - 2^{-15}$).

Step 19 $Q_{20} \leftarrow Q_{19} + (G(Q_{19}, Q_{18}, Q_{17}) + Q_{16} + \hat{m}_0 + k_{19})^{\lll 20}$
We have $\delta Q_{19} = 2^{17} + 2^{31}$. We need $\delta Q_{20} = 2^{31}$.
Hence, $\delta T_{19} \doteq -2^{29}$.
Conditions on T_{19}: -2^{29} must propagate past bit 31, so we require $1 \in T[29-31]$ ($P = 1 - 2^{-3}$).

Steps 20 and 21 In both these steps we have $\delta Q_t = 2^{31}$ and we need $\delta Q_{t+1} = 2^{31}$ so there are no conditions on T_t. Hence, we leave out further details of these two steps.

Step 22 $Q_{23} \leftarrow Q_{22} + (G(Q_{22}, Q_{21}, Q_{20}) + Q_{19} + \hat{m}_{15} + k_{22})^{\lll 14}$
We have $\delta Q_{22} = 2^{31}$. We need $\delta Q_{23} = 0$.
Hence, $\delta T_{22} \doteq \pm 2^{17}$. Since $\delta Q_{19} = +2^{17}$ we actually get $\delta T_{22} = +2^{17}$.
Conditions on T_{22}: 2^{17} must not propagate past bit 17, so we require $T[17] = 0$ ($P = 2^{-1}$).

Steps 23–33 In all these steps we have $\delta Q_t = 0$ and we need $\delta Q_{t+1} = 0$, so again, there are no requirements on T_t since $\delta T_t \doteq 0$.

Step 34 $Q_{35} \leftarrow Q_{34} + (H(Q_{34}, Q_{33}, Q_{32}) + Q_{31} + \hat{m}_{11} + k_{34})^{\lll 16}$
We have $\delta Q_{34} = 0$. We need $\delta Q_{35} = 2^{31}$.
Hence, $\delta T_{34} \doteq \pm 2^{15}$. Since $\delta \hat{m}_{11} = -2^{15}$ we actually get $\delta T_{34} = -2^{15}$.
Conditions on T_{34}: -2^{15} must propagate past bit 15, so we require $T[15] = 1$ ($P = 2^{-1}$).

Steps 35–60 In all these steps we have $\delta Q_t = 2^{31}$ and we need $\delta Q_{t+1} = 2^{31}$, so there are no requirements on T_t.

Step 61 $Q_{62} \leftarrow Q_{61} + (I(Q_{61}, Q_{60}, Q_{59}) + Q_{58} + \hat{m}_{11} + k_{61})^{\lll 10}$
We have $\delta Q_{61} = 2^{31}$ and we need $\delta Q_{62} = -2^{25} + 2^{31}$.
Hence, $\delta T_{50} \doteq -2^{15}$.
Conditions on T_{61}: -2^{15} must propagate past bit 21, so we require $\mathtt{1} \in T[15-21]$ $(P = 1 - 2^{-7})$.

Steps 62 and 63 In the last two steps we have $\delta Q_t = -2^{25} + 2^{31}$ and we need $\delta Q_{t+1} = -2^{25} + 2^{31}$, so there are no requirements on T_t.

One way (and probably the best way) to check that the conditions on T_t hold, is, given Q_t and Q_{t+1}, to compute

$$T_t = (Q_{t+1} - Q_t)^{\ggg S_{t+1}},$$

and then check this value. In fact, since (in the first round) we select Q_{t+1} before we have a value of m_t, we can compute T_t as described and then compute m_t based on the value of T_t, i.e. by

$$m_t = T_t - F(Q_t, Q_{t-1}, Q_{t-2}) - Q_{t-3} - k_t.$$

In subsequent rounds, or whenever we don't use the T-value to update a message word, we simply discard the value after having confirmed that the conditions on it hold.

4.5.2 Conditions on step variables

Here we examine the conditions on the step variables when processing the second message block. We have the same requirements on ∇Q_t as Wang et al. in [45], except for bit 31 where we often have a relative condition in contrast with the absolute conditions in [45]. These will be explained.

First note that for the characteristics to hold at the beginning of the second iteration, we must require $Q_{-2}[25] \doteq \mathtt{0}$, $Q_{-1}[25] \doteq \mathtt{1}$, $Q_{-1}[26] \doteq \mathtt{0}$ and $Q_0[25] \doteq \mathtt{0}$.

All the conditions derived in this section can be looked up in Appendix A.2. Note that to save space, instead of e.g. $i \in \{1,2,3,4,7,8,9,26\}$ we may write $i \in \{1..4, 7..9, 26\}$.

Step 0 $Q_1 \leftarrow Q_0 + (F(Q_0, Q_{-1}, Q_{-2}) + Q_{-3} + \hat{m}_0 + k_0)^{\lll 7}$
We have $\delta Q_{-3} = 2^{31}$. We need $\delta T_0 = 0$.
Hence, $\delta f_0 \doteq 2^{31}$.

Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_0	$\langle 25, \pm 31\rangle$
Q_{-1}	$\langle -25, 26, \pm 31\rangle$
Q_{-2}	$\langle 25, \pm 31\rangle$

Since Q_0, Q_{-1}, Q_{-2} all differ on bit 31, we require that $\nabla Q_{-1}[31] \doteq \nabla Q_{-2}[31]$ implying that $Q_{-1}[31] \doteq Q_{-2}[31]$ (note that Wang et al. state that $\nabla Q_{-1}[31] \doteq \nabla Q_{-2}[31] \doteq \texttt{+1}$, but they only require $Q_{-1}[31] \doteq Q_{-2}[31]$, not that they are both $\texttt{0}$). To avoid further differences we require $Q_0[26] \doteq \texttt{0}$. Also, since $\nabla Q_1[25] \doteq \texttt{+1}$, we require $Q_1[25] \doteq \texttt{0}$.

Step 1 $\quad Q_2 \leftarrow Q_1 + (F(Q_1, Q_0, Q_{-1}) + Q_{-2} + \hat{m}_1 + k_1)^{\lll 12}$
We have $\delta Q_{-2} = 2^{25} + 2^{31}$. We need $\delta T_1 = 2^{25}$.
Hence, $\delta f_1 \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_1	$\langle 25, \pm 31\rangle$
Q_0	$\langle 25, \pm 31\rangle$
Q_{-1}	$\langle -25, 26, \pm 31\rangle$

Hence, we require that $Q_1[26] \doteq \texttt{1}$ and that $Q_0[31] \doteq Q_{-1}[31]$. Also, since $\nabla Q_2 \doteq \langle 5, 25, \pm 31\rangle$, we require $Q_2[5] \doteq Q_2[25] \doteq \texttt{0}$.

Step 2 $\quad Q_3 \leftarrow Q_2 + (F(Q_2, Q_1, Q_0) + Q_{-1} + \hat{m}_2 + k_2)^{\lll 17}$
We have $\delta Q_{-1} = 2^{25} + 2^{31}$. We need $\delta T_2 = 2^{26} + 2^{31}$.
Hence, $\delta f_2 \doteq 2^{25}$ (using $2^{25} + 2^{25} = 2^{26}$).
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_2	$\langle 5, 25, \pm 31\rangle$
Q_1	$\langle 25, \pm 31\rangle$
Q_0	$\langle 25, \pm 31\rangle$

Since we require that the differences on bit 31 disappear, we must require $Q_1[31] \doteq \neg Q_0[31]$. The differences on bit 25 are just what we need, but we need $Q_1[5] \doteq Q_0[5]$. Also, since

$$\nabla Q_3 \doteq \langle -5, -6, 7, -11, 12, -16, \ldots, -20, 21, -25, \ldots, -29, 30, \pm 31\rangle,$$

we require $Q_3[5] = Q_3[6] = Q_3[11] = Q_3[16] = \ldots = Q_3[20] = Q_3[25] = \ldots = Q_3[29] = \texttt{1}$ and $Q_3[7] = Q_3[12] = Q_3[21] = Q_3[30] = \texttt{0}$.

Step 3 $Q_4 \leftarrow Q_3 + (F(Q_3, Q_2, Q_1) + Q_0 + \hat{m}_3 + k_3)^{\lll 22}$
We have $\delta Q_0 = 2^{25} + 2^{31}$. We need $\delta T_3 = -2^{11} - 2^{21} - 2^{26}$.
Hence, $\delta f_3 \doteq -2^{11} - 2^{21} - 2^{25} - 2^{26} + 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_3	$\langle -5, -6, 7, -11, 12, -16, \ldots, -20, 21, -25, \ldots, -29, 30, \pm 31 \rangle$
Q_2	$\langle 5, 25, \pm 31 \rangle$
Q_1	$\langle 25, \pm 31 \rangle$

Since we get $+2^{25}$ from this, but we needed -2^{25}, we use instead -2^{26}, which we already require, so this becomes -2^{27}. I.e. we require instead $\delta f_3 \doteq -2^{11} - 2^{21} + 2^{25} - 2^{27} + 2^{31}$. Hence, we must require $Q_2[11] \doteq \mathtt{1}$ and $Q_1[11] \doteq \mathtt{0}$, $Q_2[21] \doteq \mathtt{0}$ and $Q_1[21] \doteq \mathtt{1}$, $Q_2[27] \doteq \mathtt{1}$ and $Q_1[27] \doteq \mathtt{0}$, and $Q_2[31] \doteq Q_1[31]$. Also, to avoid further differences we must require $Q_2[i] \doteq Q_1[i]$ for $i \in \{5..7, 12, 16..20, 26, 28..30\}$ (note that we already required $Q_1[5] \doteq Q_0[5]$), and $Q_1[5] \doteq \mathtt{0}$ (implying altogether that $Q_2[5] \doteq Q_1[5] \doteq Q_0[5] \doteq \mathtt{0}$). Finally, from

$$\nabla Q_4 \doteq \langle 1, 2, 3, -4, 5, -25, 26, \pm 31 \rangle$$

we get (trivial) conditions on Q_4.

Step 4 $Q_5 \leftarrow Q_4 + (F(Q_4, Q_3, Q_2) + Q_1 + \hat{m}_4 + k_4)^{\lll 7}$
We have $\delta Q_1 = 2^{25} + 2^{31}$ and $\delta \hat{m}_4 = 2^{31}$. We need $\delta T_4 = 2^1 + 2^2 - 2^{18} + 2^{25} + 2^{26} - 2^{30} + 2^{31}$.
Hence, $\delta f_4 \doteq 2^1 + 2^2 - 2^{18} + 2^{26} - 2^{30} + 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_4	$\langle 1, 2, 3, -4, 5, -25, 26, \pm 31 \rangle$
Q_3	$\langle -5, -6, 7, -11, 12, -16, \ldots, -20, 21, -25, \ldots, -29, 30, \pm 31 \rangle$
Q_2	$\langle 5, 25, \pm 31 \rangle$

To create the required δf_4 we need $Q_3[1] \doteq \mathtt{1}$ and $Q_2[1] \doteq \mathtt{0}$, $Q_3[2] \doteq \mathtt{1}$ and $Q_2[2] \doteq \mathtt{0}$, $Q_4[18] \doteq \mathtt{1}$ and since $2^{26} - 2^{30} = -2^{26} - 2^{27} - 2^{28} - 2^{29}$ (and we can produce the latter but not the former), we also require $Q_2[26] \doteq \mathtt{1}$ (already required, henceforth written *a.r.*), and $Q_4[i] = \mathtt{1}$ for $i \in \{27, 28, 29\}$. For the last term, 2^{31}, to appear we also require $Q_3[31] \doteq Q_2[31]$. To avoid further differences we require $Q_3[3] \doteq Q_2[3]$, $Q_3[4] \doteq Q_2[4]$, and $Q_4[i] \doteq \mathtt{0}$ for $i \in \{6, 7, 11, 12, 16, 17, 19..21, 30\}$. Finally, we get the trivial conditions on Q_5 from

$$\nabla Q_5 \doteq \langle 0, -6, 7, 8, -9, -10, -11, 12, \pm 31 \rangle.$$

Step 5 $Q_6 \leftarrow Q_5 + (F(Q_5, Q_4, Q_3) + Q_2 + \hat{m}_5 + k_5)^{\lll 12}$
We have $\delta Q_2 = 2^5 + 2^{25} + 2^{31}$. We need $\delta T_5 = -2^4 - 2^8 - 2^{20} - 2^{26} - 2^{28} - 2^{29}$.
Hence, $\delta f_5 \doteq -2^4 - 2^5 - 2^8 - 2^{20} - 2^{25} - 2^{26} - 2^{28} - 2^{29} + 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_5	$\langle 0, -6, 7, 8, -9, -10, -11, 12, \pm 31 \rangle$
Q_4	$\langle 1, 2, 3, -4, 5, -25, 26, \pm 31 \rangle$
Q_3	$\langle -5, -6, 7, -11, 12, -16, \ldots, -20, 21, -25, \ldots, -29, 30, \pm 31 \rangle$

To create the desired δf_4 we must require $Q_5[4] \doteq 1$, $Q_5[5] \doteq 0$, $Q_3[8] \doteq 1$ and $Q_4[8] \doteq 0$, $Q_5[20] \doteq 0$, $Q_5[26] \doteq 0$, $Q_5[28] \doteq 0$, $Q_5[29] \doteq 0$ and $Q_4[31] \doteq Q_3[31]$. To avoid further differences we must require $Q_4[i] \doteq Q_3[i]$ for $i \in \{0, 9, 10\}$, $Q_4[6] \doteq 0$, $Q_4[7] \doteq 0$, $Q_4[11] \doteq 0$, $Q_4[12] \doteq 0$ (the latter four a.r.), $Q_5[i] \doteq 0$ for $i \in \{1, 2, 3\}$, and $Q_5[i] \doteq 1$ for $i \in \{16..19, 21, 27, 30\}$. Finally, we get the trivial conditions from

$$\nabla Q_6 \doteq \langle 16, -17, 20, -21, \pm 31 \rangle.$$

Step 6 $Q_7 \leftarrow Q_6 + (F(Q_6, Q_5, Q_4) + Q_3 + \hat{m}_6 + k_6)^{\lll 17}$
We have $\delta Q_3 = 2^5 + 2^{11} + 2^{16} + 2^{25} + 2^{31}$. We need $\delta T_6 = 2^3 - 2^{10} - 2^{21} + 2^{31}$.
Hence, $\delta f_6 \doteq 2^3 - 2^5 - 2^{10} - 2^{11} - 2^{16} - 2^{21} - 2^{25}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_6	$\langle 16, -17, 20, -21, \pm 31 \rangle$
Q_5	$\langle 0, -6, 7, 8, -9, -10, -11, 12, \pm 31 \rangle$
Q_4	$\langle 1, 2, 3, -4, 5, -25, 26, \pm 31 \rangle$

First, we require $Q_6[3] \doteq 0$. Since we can create $+2^5$ but not -2^5, we create $2^5 - 2^6 = -2^5$. Hence, we require $Q_6[5] \doteq 0$ and $Q_6[6] \doteq 1$. Furthermore, we require $Q_6[10] \doteq 1$ and $Q_6[11] \doteq 1$. Since we already required $Q_4[16] \doteq 0$ and $Q_5[16] \doteq 1$, we must produce -2^{16} as $2^{16} - 2^{17}$, hence $Q_4[17] \doteq 0$ and $Q_5[17] \doteq 1$ (both a.r.). Continuing, we require $Q_5[21] \doteq 1$ and $Q_4[21] \doteq 0$ (both a.r.), and $Q_6[25] \doteq 0$. To avoid further differences we require $Q_5[20] \doteq Q_4[20]$ (a.r.), $Q_5[31] \doteq \neg Q_4[31]$, $Q_6[i] \doteq 0$ for $i \in \{0, 7..9, 12\}$, and $Q_6[i] \doteq 1$ for $i \in \{1, 2, 4, 26\}$. Finally, from

$$\nabla Q_7 \doteq \langle 6, 7, 8, -9, 27, -28, \pm 31 \rangle$$

we get trivial conditions.

Step 7 $Q_8 \leftarrow Q_7 + (F(Q_7, Q_6, Q_5) + Q_4 + \hat{m}_7 + k_7)^{\lll 22}$
We have $\delta Q_4 = -2^1 + 2^5 + 2^{25} + 2^{31}$. We need $\delta T_7 = -2^1 + 2^5 + 2^{16} + 2^{25} - 2^{27}$.
Hence, $\delta f_7 \doteq 2^{16} - 2^{27} + 2^{31}$.

Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_7	$\langle 6,7,8,-9,27,-28,\pm 31\rangle$
Q_6	$\langle 16,-17,20,-21,\pm 31\rangle$
Q_5	$\langle 0,-6,7,8,-9,-10,-11,12,\pm 31\rangle$

We first require $Q_7[16] \doteq \texttt{1}$, $Q_5[27] \doteq \texttt{1}$ (a.r.) and $Q_6[27] \doteq \texttt{0}$, and $Q_6[31] \doteq Q_5[31]$. To avoid further differences we require $Q_6[6] \doteq \texttt{1}$, $Q_6[7] \doteq \texttt{0}$, $Q_6[8] \doteq \texttt{0}$, $Q_6[9] \doteq \texttt{0}$ (all four a.r.), $Q_6[28] \doteq Q_5[28]$ (implying $Q_6[28] \doteq \texttt{0}$), $Q_7[i] \doteq \texttt{0}$ for $i \in \{17,20,21\}$, and $Q_7[i] \doteq \texttt{1}$ for $i \in \{0,10..12\}$. Finally, since

$$\nabla Q_8 \doteq \langle -15,16,-17,23,24,25,-26,\pm 31\rangle,$$

we get some trivial conditions on Q_8.

Step 8 $\quad Q_9 \leftarrow Q_8 + (F(Q_8,Q_7,Q_6)+Q_5+\hat{m}_8+k_8)^{\lll 7}$
We have $\delta Q_5 = 2^0+2^6+2^8+2^9+2^{31}$. We need $\delta T_8 = 2^0+2^8+2^9+2^{16}+2^{25}-2^{31}$.
Hence, $\delta f_8 \doteq -2^6+2^{16}+2^{25}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_8	$\langle -15,16,-17,23,24,25,-26,\pm 31\rangle$
Q_7	$\langle 6,7,8,-9,27,-28,\pm 31\rangle$
Q_6	$\langle 16,-17,20,-21,\pm 31\rangle$

We cannot create -2^6 directly, but we can create $2^6+2^7+2^8-2^9 = -2^6$, by requiring that $Q_8[i] \doteq \texttt{1}$ for $i \in \{6,7,8,9\}$. Furthermore, we require $Q_7[16] \doteq \texttt{1}$ (a.r.), $Q_6[25] \doteq \texttt{0}$ (a.r.) and $Q_7[25] \doteq \texttt{1}$, and $Q_7[31] \doteq \neg Q_6[31]$. To avoid further differences we also require $Q_7[i] \doteq Q_6[i]$ for $i \in \{15,23,24,26\}$ (implying $Q_7[26] \doteq \texttt{1}$), $Q_7[17] \doteq \texttt{0}$ (a.r.), $Q_8[i] \doteq \texttt{0}$ for $i \in \{27,28\}$ and $Q_8[i] \doteq \texttt{1}$ for $i \in \{20,21\}$. Finally, we get the trivial conditions from

$$\nabla Q_9 \doteq \langle -0,1,-6,-7,-8,9,\pm 31\rangle.$$

Step 9 $\quad Q_{10} \leftarrow Q_9 + (F(Q_9,Q_8,Q_7)+Q_6+\hat{m}_9+k_9)^{\lll 12}$
We have $\delta Q_6 = -2^{16}-2^{20}+2^{31}$. We need $\delta T_9 = 2^0-2^{20}-2^{26}$.
Hence, $\delta f_9 \doteq 2^0+2^{16}-2^{26}+2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_9	$\langle -0,1,-6,-7,-8,9,\pm 31\rangle$
Q_8	$\langle -15,16,-17,23,24,25,-26,\pm 31\rangle$
Q_7	$\langle 6,7,8,-9,27,-28,\pm 31\rangle$

We require $Q_8[0] \doteq 0$ and $Q_7[0] \doteq 1$ (the latter a.r.), $Q_9[16] \doteq 1$, $Q_9[26] \doteq 1$ and $Q_8[31] \doteq Q_7[31]$. To avoid further differences, we require $Q_8[1] \doteq Q_7[1]$, $Q_8[i] \doteq 1$ for $i \in \{6..9\}$ (all four a.r.), $Q_9[i] \doteq 0$ for $i \in \{15, 17, 23..25\}$, and $Q_9[i] \doteq 1$ for $i \in \{27, 28\}$. Finally, we get one trivial condition from

$$\nabla Q_{10} \doteq \langle 12, \pm 31 \rangle.$$

Step 10 $\quad Q_{11} \leftarrow Q_{10} + (F(Q_{10}, Q_9, Q_8) + Q_7 + \hat{m}_{10} + k_{10})^{\lll 17}$
We have $\delta Q_7 = -2^6 - 2^{27} + 2^{31}$. We need $\delta T_{10} = -2^{27}$.
Hence, $\delta f_{10} \doteq 2^6 + 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{10}	$\langle 12, \pm 31 \rangle$
Q_9	$\langle -0, 1, -6, -7, -8, 9, \pm 31 \rangle$
Q_8	$\langle -15, 16, -17, 23, 24, 25, -26, \pm 31 \rangle$

We cannot create 2^6 directly, but we can create $-2^6 - 2^7 - 2^8 + 2^9$ by requiring $Q_{10}[i] \doteq 1$ for $i \in \{6..9\}$, and $Q_9[31] \doteq Q_8[31]$. To avoid further differences we require that $Q_9[12] \doteq Q_8[12]$, $Q_{10}[i] \doteq 0$ for $i \in \{0, 1\}$, and $Q_{10}[i] \doteq 1$ for $i \in \{15..17, 23..26\}$. There are no conditions on Q_{11} yet, as $\nabla Q_{11} \doteq \langle \pm 31 \rangle$.

Step 11 $\quad Q_{12} \leftarrow Q_{11} + (F(Q_{11}, Q_{10}, Q_9) + Q_8 + \hat{m}_{11} + k_{11})^{\lll 22}$
We have $\delta Q_8 = 2^{15} - 2^{17} - 2^{23} + 2^{31}$ and $\delta \hat{m}_{11} = -2^{15}$. We need $\delta T_{11} = -2^{17} - 2^{23}$.
Hence, $\delta f_{11} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{11}	$\langle \pm 31 \rangle$
Q_{10}	$\langle 12, \pm 31 \rangle$
Q_9	$\langle -0, 1, -6, -7, -8, 9, \pm 31 \rangle$

To create δf_{11} we simply require $Q_{10}[31] \doteq Q_9[31]$. To avoid further differences we also require $Q_{11}[12] \doteq 0$ and $Q_{11}[i] \doteq 1$ for $i \in \{0, 1, 6..9\}$. Furthermore, trivial conditions are added by

$$\nabla Q_{12} \doteq \langle -7, 13, 14, 15, 16, 17, 18, -19, \pm 31 \rangle.$$

Step 12 $\quad Q_{13} \leftarrow Q_{12} + (F(Q_{12}, Q_{11}, Q_{10}) + Q_9 + \hat{m}_{12} + k_{12})^{\lll 7}$
We have $\delta Q_9 = 2^0 + 2^6 + 2^{31}$. We need $\delta T_{12} = 2^0 + 2^6 + 2^{17}$.
Hence, $\delta f_{12} \doteq 2^{17} + 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f func-

tion.

Q_t	∇Q_t
Q_{12}	$\langle -7,13,14,15,16,17,18,-19,\pm 31\rangle$
Q_{11}	$\langle \pm 31\rangle$
Q_{10}	$\langle 12,\pm 31\rangle$

Since we previously required $Q_{10}[17] \doteq 1$, we cannot produce $+2^{17}$ directly, but we can produce $-2^{17}+2^{18}$ by requiring $Q_{11}[17] \doteq 0$, $Q_{10}[18] \doteq 0$ and $Q_{11}[18] \doteq 1$. Furthermore we require $Q_{11}[31] \doteq Q_{10}[31]$, and to avoid further differences $Q_{11}[i] \doteq Q_{10}[i]$ for $i \in \{7,13..16,19\}$ (implying $Q_{11}[15] \doteq Q_{11}[16] \doteq 1$, condition on $Q_{11}[7]$ a.r.), and $Q_{12}[12] \doteq 1$. From

$$\nabla Q_{13} \doteq \langle -24,-25,-26,-27,-28,-29,30,\pm 31\rangle$$

we get some additional conditions.

Step 13 $\quad Q_{14} \leftarrow Q_{13} + (F(Q_{13},Q_{12},Q_{11}) + Q_{10} + \hat{m}_{13} + k_{13})^{\lll 12}$
We have $\delta Q_{10} = 2^{12}+2^{31}$. We need $\delta T_{13} = -2^{12}$.
Hence, $\delta f_{13} \doteq -2^{13}+2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{13}	$\langle -24,-25,-26,-27,-28,-29,30,\pm 31\rangle$
Q_{12}	$\langle -7,13,14,15,16,17,18,-19,\pm 31\rangle$
Q_{11}	$\langle \pm 31\rangle$

We cannot produce -2^{13} directly, but we can produce $2^{13}+2^{14}+\ldots+2^{18}-2^{19}$ by requiring $Q_{13}[i] \doteq 1$ for $i \in \{13..19\}$. We also require $Q_{12}[31] \doteq Q_{11}[31]$, and to avoid further differences we require $Q_{12}[i] \doteq Q_{11}[i]$ for $i \in \{24..30\}$, and $Q_{13}[7] \doteq 0$. There are no trivial conditions added by $\nabla Q_{14} \doteq \langle \pm 31\rangle$.

Step 14 $\quad Q_{15} \leftarrow Q_{14} + (F(Q_{14},Q_{13},Q_{12}) + Q_{11} + \hat{m}_{14} + k_{14})^{\lll 17}$
We have $\delta Q_{11} = 2^{31}$ and $\delta \hat{m}_{14} = 2^{31}$. We need $\delta T_{14} = 2^{18}+2^{30}$.
Hence, $\delta T_{14} \doteq 2^{18}+2^{30}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{14}	$\langle \pm 31\rangle$
Q_{13}	$\langle -24,-25,-26,-27,-28,-29,30,\pm 31\rangle$
Q_{12}	$\langle -7,13,14,15,16,17,18,-19,\pm 31\rangle$

We first require $Q_{14}[18] \doteq 0$ and $Q_{14}[30] \doteq 1$. Then, to avoid further differences we require $Q_{13}[31] \doteq \neg Q_{12}[31]$, $Q_{14}[i] \doteq 0$ for $i \in \{24..29\}$ and $Q_{14}[i] \doteq 1$ for $i \in \{7,13..17,19\}$. Finally we have a few additional, trivial conditions from $\nabla Q_{15} \doteq \langle 3,15,\pm 31\rangle$.

Step 15 $Q_{16} \leftarrow Q_{15} + (F(Q_{15}, Q_{14}, Q_{13}) + Q_{12} + \hat{m}_{15} + k_{15})^{\lll 22}$
We have $\delta Q_{12} = -2^7 - 2^{13} + 2^{31}$. We need $\delta T_{15} = -2^7 - 2^{13} - 2^{25}$.
Hence, $\delta f_{15} \doteq -2^{25} + 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{15}	$\langle 3, 15, \pm 31 \rangle$
Q_{14}	$\langle \pm 31 \rangle$
Q_{13}	$\langle -24, -25, -26, -27, -28, -29, 30, \pm 31 \rangle$

We first require $Q_{15}[25] \doteq 0$ and $Q_{14}[31] \doteq Q_{13}[31]$. To avoid further differences we also require $Q_{14}[i] \doteq Q_{13}[i]$ for $i \in \{3, 15\}$ (the latter a.r.), and $Q_{15}[i] \doteq 1$ for $i \in \{24, 26..30\}$. Finally, one trivial condition is added by $\nabla Q_{16} \doteq \langle -29, \pm 31 \rangle$.

Step 16 $Q_{17} \leftarrow Q_{16} + (G(Q_{16}, Q_{15}, Q_{14}) + Q_{13} + \hat{m}_1 + k_{16})^{\lll 5}$
We have $\delta Q_{13} = 2^{24} + 2^{31}$. We need $\delta T_{16} = 2^{24}$.
Hence, $\delta f_{16} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function – which is now the G function.

Q_t	∇Q_t
Q_{16}	$\langle -29, \pm 31 \rangle$
Q_{15}	$\langle 3, 15, \pm 31 \rangle$
Q_{14}	$\langle \pm 31 \rangle$

Bearing in mind that $G(Q_{16}, Q_{15}, Q_{14})$ is equivalent to $F(Q_{14}, Q_{16}, Q_{15})$, we first require that $Q_{16}[31] \doteq Q_{15}[31]$, and to avoid further differences $Q_{14}[29] \doteq 0$ (a.r.) and $Q_{14}[i] \doteq 1$ for $i \in \{3, 15\}$ (implying $Q_{13}[3] \doteq 1$ since we already required $Q_{14}[3] \doteq Q_{13}[3]$, condition on $Q_{14}[15]$ a.r.). We get no additional, trivial conditions from $\nabla Q_{17} \doteq \langle \pm 31 \rangle$.

Step 17 $Q_{18} \leftarrow Q_{17} + (G(Q_{17}, Q_{16}, Q_{15}) + Q_{14} + \hat{m}_6 + k_{17})^{\lll 9}$
We have $\delta Q_{14} = 2^{31}$. We need $\delta T_{17} = 0$.
Hence, $\delta f_{17} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{17}	$\langle \pm 31 \rangle$
Q_{16}	$\langle -29, \pm 31 \rangle$
Q_{15}	$\langle 3, 15, \pm 31 \rangle$

To produce 2^{31} we require $Q_{17}[31] \doteq Q_{16}[31]$, and to avoid further differences we require $Q_{17}[i] \doteq Q_{16}[i]$ for $i \in \{3, 15\}$, and $Q_{15}[29] \doteq 1$ (a.r.). We get no additional, trivial conditions from $\nabla Q_{18} \doteq \langle \pm 31 \rangle$.

Step 18 $\quad Q_{19} \leftarrow Q_{18} + (G(Q_{18}, Q_{17}, Q_{16}) + Q_{15} + \hat{m}_{11} + k_{18})^{\lll 14}$
We have $\delta Q_{15} = 2^3 + 2^{15} + 2^{31}$ and $\delta \hat{m}_{11} = -2^{15}$. We need $\delta T_{18} = 2^3$.
Hence, $\delta f_{18} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{18}	$\langle \pm 31 \rangle$
Q_{17}	$\langle \pm 31 \rangle$
Q_{16}	$\langle -29, \pm 31 \rangle$

We require $Q_{18}[i] \doteq Q_{17}[i]$ for $i \in \{29, 31\}$. We get one additional condition from $\nabla Q_{19} \doteq \langle 17, \pm 31 \rangle$.

Step 19 $\quad Q_{20} \leftarrow Q_{19} + (G(Q_{19}, Q_{18}, Q_{17}) + Q_{16} + \hat{m}_0 + k_{19})^{\lll 20}$
We have $\delta Q_{16} = -2^{29} + 2^{31}$. We need $\delta T_{19} = -2^{29}$.
Hence, $\delta f_{19} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{19}	$\langle 17, \pm 31 \rangle$
Q_{18}	$\langle \pm 31 \rangle$
Q_{17}	$\langle \pm 31 \rangle$

We require $Q_{19}[31] \doteq Q_{18}[31]$ and $Q_{17}[17] \doteq \texttt{0}$. We get no additional conditions from $\nabla Q_{20} \doteq \langle \pm 31 \rangle$.

Step 20 $\quad Q_{21} \leftarrow Q_{20} + (G(Q_{20}, Q_{19}, Q_{18}) + Q_{17} + \hat{m}_5 + k_{20})^{\lll 5}$
We have $\delta Q_{17} = 2^{31}$. We need $\delta T_{20} = 0$.
Hence, $\delta f_{20} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{20}	$\langle \pm 31 \rangle$
Q_{19}	$\langle 17, \pm 31 \rangle$
Q_{18}	$\langle \pm 31 \rangle$

We require $Q_{20}[31] \doteq Q_{19}[31]$ and $Q_{18}[17] \doteq \texttt{1}$. We get no additional conditions from $\nabla Q_{21} \doteq \langle \pm 31 \rangle$.

Step 21 $\quad Q_{22} \leftarrow Q_{21} + (G(Q_{21}, Q_{20}, Q_{19}) + Q_{18} + \hat{m}_{10} + k_{21})^{\lll 9}$
We have $\delta Q_{18} = 2^{31}$. We need $\delta T_{21} = 0$.
Hence, $\delta f_{21} \doteq 2^{31}$.

Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{21}	$\langle \pm 31 \rangle$
Q_{20}	$\langle \pm 31 \rangle$
Q_{19}	$\langle 17, \pm 31 \rangle$

We require $Q_{21}[i] \doteq Q_{20}[i]$ for $i \in \{17, 31\}$. We get no additional conditions from $\nabla Q_{22} \doteq \langle \pm 31 \rangle$.

Step 22 $\quad Q_{23} \leftarrow Q_{22} + (G(Q_{22}, Q_{21}, Q_{20}) + Q_{19} + \hat{m}_{15} + k_{22})^{\lll 14}$
We have $\delta Q_{19} = 2^{17} + 2^{31}$. We need $\delta T_{22} = 2^{17}$.
Hence, $\delta f_{22} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{22}	$\langle \pm 31 \rangle$
Q_{21}	$\langle \pm 31 \rangle$
Q_{20}	$\langle \pm 31 \rangle$

We require $Q_{22}[31] \doteq Q_{21}[31]$. Note that $\delta Q_{23} \doteq 0$.

Step 23 $\quad Q_{24} \leftarrow Q_{23} + (G(Q_{23}, Q_{22}, Q_{21}) + Q_{20} + \hat{m}_4 + k_{23})^{\lll 20}$
We have $\delta Q_{20} = 2^{31}$ and $\delta \hat{m}_4 = 2^{31}$. We need $\delta T_{23} = 0$.
Hence, $\delta f_{23} \doteq 0$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{23}	$\langle \rangle$
Q_{22}	$\langle \pm 31 \rangle$
Q_{21}	$\langle \pm 31 \rangle$

We require $Q_{23}[31] \doteq \mathtt{0}$. Note that $\delta Q_{24} \doteq 0$.

Step 24 $\quad Q_{25} \leftarrow Q_{24} + (G(Q_{24}, Q_{23}, Q_{22}) + Q_{21} + \hat{m}_9 + k_{24})^{\lll 5}$
We have $\delta Q_{21} = 2^{31}$. We need $\delta T_{24} = 0$.
Hence, $\delta f_{24} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{24}	$\langle \rangle$
Q_{23}	$\langle \rangle$
Q_{22}	$\langle \pm 31 \rangle$

We require $Q_{24}[31] \doteq \neg Q_{23}[31]$, implying $Q_{24}[31] \doteq \mathtt{1}$. Note that $\delta Q_{25} \doteq 0$.

Step 25 $Q_{26} \leftarrow Q_{25} + (G(Q_{25}, Q_{24}, Q_{23}) + Q_{22} + \hat{m}_{14} + k_{25})^{\lll 9}$
We have $\delta Q_{22} = 2^{31}$ and $\delta \hat{m}_{14} = 2^{31}$. We need $\delta T_{25} = 0$.
Hence, $\delta f_{25} \doteq 0$.
Conditions: We have no bit differences on the arguments to the f function, so we get $\delta f_{25} = 0$ directly.

Steps 26–33 In all these steps there are no bit differences on the arguments to the f function, and hence no requirements on the step variables.

Steps 34–47 In all these steps the f function is the xor function, and hence a difference (on a certain bit position) on an odd number of arguments to the f function will always produce a difference in the output, and a difference on an even number of arguments will never produce a difference in the output. Hence, we cannot ensure any specific behavior of the f function, but it can be easily verified that the characteristics are always fulfilled during these steps.

Step 48 $Q_{49} \leftarrow Q_{48} + (I(Q_{48}, Q_{47}, Q_{46}) + Q_{45} + \hat{m}_0 + k_{48})^{\lll 6}$
We have $\delta Q_{45} = 2^{31}$. We need $\delta T_{48} = 0$.
Hence, $\delta f_{48} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function – which is now the I function.

Q_t	∇Q_t
Q_{48}	$\langle \pm 31 \rangle$
Q_{47}	$\langle \pm 31 \rangle$
Q_{46}	$\langle \pm 31 \rangle$

Let's take a look at the truth table of the I function:

X	Y	Z	$I(X,Y,Z)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Since we have differences on bit 31 for all arguments, we need to finding a pattern for which flipping all bits always produces a difference on the output. In order to minimize the number of conditions we require that we only add a condition to Q_{48} in this step. The only pattern seems to be whenever $X = Z$, which works since $I(0,0,0) = 1$ but $I(1,1,1) = 0$, and $I(0,1,0) = 0$ but $I(1,0,1) = 1$. Hence, we have the condition $Q_{48}[31] \doteq Q_{46}[31]$.

Step 49 $Q_{50} \leftarrow Q_{49} + (I(Q_{49}, Q_{48}, Q_{47}) + Q_{46} + \hat{m}_7 + k_{49})^{\lll 10}$
We have $\delta Q_{46} = 2^{31}$. We need $\delta T_{49} = 0$.
Hence, $\delta f_{49} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{49}	$\langle \pm 31 \rangle$
Q_{48}	$\langle \pm 31 \rangle$
Q_{47}	$\langle \pm 31 \rangle$

The requirements are the same as in step 48, so the condition is $Q_{49}[31] \doteq Q_{47}[31]$.

Step 50 $Q_{51} \leftarrow Q_{50} + (I(Q_{50}, Q_{49}, Q_{48}) + Q_{47} + \hat{m}_{14} + k_{50})^{\lll 15}$
We have $\delta Q_{47} = 2^{31}$ and $\delta \hat{m}_{14} = 2^{31}$. We need $\delta T_{50} = 0$.
Hence, $\delta f_{50} \doteq 0$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{50}	$\langle \pm 31 \rangle$
Q_{49}	$\langle \pm 31 \rangle$
Q_{48}	$\langle \pm 31 \rangle$

In this step there must be no difference in the output of the f function, so we need to look at the truth table of the I function again. We note that whenever $X = \neg Z$ we have the desired behavior, so we require $Q_{50}[31] \doteq \neg Q_{48}[31]$.

Steps 51–59 In all these steps we have the same situation as in steps 48 and 49, so we require $Q_t[31] \doteq Q_{t-2}[31]$ for $51 \leq t \leq 59$.

Step 60 $Q_{61} \leftarrow Q_{60} + (I(Q_{60}, Q_{59}, Q_{58}) + Q_{57} + \hat{m}_4 + k_{60})^{\lll 6}$
We have $\delta Q_{57} = 2^{31}$ and $\delta \hat{m}_4 = 2^{31}$. We need $\delta T_{60} = 0$.
Hence, $\delta f_{60} \doteq 0$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{60}	$\langle \pm 31 \rangle$
Q_{59}	$\langle \pm 31 \rangle$
Q_{58}	$\langle \pm 31 \rangle$

Again, we need no difference out of the f function, so we require $Q_{60}[31] \doteq \neg Q_{58}[31]$.

Step 61 $Q_{62} \leftarrow Q_{61} + (I(Q_{61}, Q_{60}, Q_{59}) + Q_{58} + \hat{m}_{11} + k_{61})^{\lll 10}$
We have $\delta Q_{58} = 2^{31}$ and $\delta \hat{m}_{11} = -2^{15}$. We need $\delta T_{61} = -2^{15}$.
Hence, $\delta f_{61} \doteq 2^{31}$.

Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{61}	$\langle \pm 31 \rangle$
Q_{60}	$\langle \pm 31 \rangle$
Q_{59}	$\langle \pm 31 \rangle$

Since we need a difference on bit 31 out of the f function we again require $Q_{61}[31] \doteq Q_{59}[31]$. We have an additional condition on Q_{62} from $\nabla Q_{62} \doteq \langle -25, \pm 31 \rangle$.

Step 62 $\quad Q_{63} \leftarrow Q_{62} + (I(Q_{62}, Q_{61}, Q_{60}) + Q_{59} + \hat{m}_2 + k_{62})^{\lll 15}$
We have $\delta Q_{59} = 2^{31}$. We need $\delta T_{62} = 0$.
Hence, $\delta f_{62} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{62}	$\langle -25, \pm 31 \rangle$
Q_{61}	$\langle \pm 31 \rangle$
Q_{60}	$\langle \pm 31 \rangle$

We require $Q_{62}[31] \doteq Q_{60}[31]$, and to prevent the difference on bit 25 of Q_{62} from spreading we also require $Q_{60}[25] \doteq \texttt{0}$. We have an additional condition on Q_{63} from $\nabla Q_{63} \doteq \langle -25, \pm 31 \rangle$.

Step 63 $\quad Q_{64} \leftarrow Q_{63} + (I(Q_{63}, Q_{62}, Q_{61}) + Q_{60} + \hat{m}_9 + k_{63})^{\lll 21}$
We have $\delta Q_{60} = 2^{31}$. We need $\delta T_{63} = 0$.
Hence, $\delta f_{63} \doteq 2^{31}$.
Conditions: We have the following bit differences on the arguments to the f function.

Q_t	∇Q_t
Q_{63}	$\langle -25, \pm 31 \rangle$
Q_{62}	$\langle -25, \pm 31 \rangle$
Q_{61}	$\langle \pm 31 \rangle$

We require $Q_{63}[31] \doteq Q_{61}[31]$, and to prevent a difference on bit 25 we furthermore require $Q_{61}[25] \doteq \texttt{1}$. We have an additional condition on Q_{64} from $\nabla Q_{64} \doteq \langle -25, \pm 31 \rangle$ (note that this condition can be omitted, which is explained in Section 4.7).

4.6 Modifications by Klíma

On March 31, 2005, Vlastimil Klíma published an article [25] that describes modifications to the search for a first block of an MD5 collision, using the differential pattern that was found by Wang et al. Following Klíma's technique, it is possible to find first block near-collisions in a few minutes. He also introduced possible new

techniques for finding the second block, but these did not seem faster than the one described by Wang et al. – at least according to the speed that Wang et al. claimed to have reached. For this reason, only the modifications to finding the first block are covered in this section.

4.6.1 An overview

In his technique, Klíma makes clever use of the fact that in the first block there are no conditions on Q_1 and Q_2, so instead of letting these two values determine the values of m_0 and m_1, we can let m_0 and m_1 be determined by Q_{17} and Q_{20}. This way, we can ensure that the condition on Q_{20} holds, and by selecting new values of Q_{17} (determining Q_{18} and Q_{19}) until all these three values satisfy the (9) conditions on them, we are left with 33 conditions (according to Wang et al., ignoring T-conditions). We have 31 free bits of Q_{20} to try, so on average we should only need to choose new values for $Q_1, \ldots, Q_{19}$ four times before we find a usable near-collision. Of course, because of T-conditions, of which there are 4 with a combined probability of success of about $1/5$, in practice we need more than 4 tries, but the time it takes to choose the first 19 Q-values and compute the relevant message words is still negligible.

4.6.2 Details

The technique is quite simple and elegant, so the best way to describe it might be in the form of an algorithm, see Algorithm 4.2.

In [25], Klíma suggests to compute $m_2, \ldots, m_5$ before starting Q_{20} as a counter, but this is not possible since computing those values requires that Q_1 and Q_2 exist already, and these we cannot compute before we know m_0.

Obviously, if it was possible to perform multi-message modifications in steps later than 19, then we could do this to decrease the complexity even further. But, as mentioned, we cannot perform multi-message modifications later than step 18.

4.7 Possible additional improvements

Some of the conditions on step variables in the second iteration can be changed to improve the attack slightly.

First of all, there is really no need to require $\nabla Q_{64} = \langle -25, \pm 31 \rangle$. For previous step variables we have this requirement in order to make it possible (or at least easier) to control the spreading of bit differences through the f function. However, the last step variable, Q_{64}, is never used in an f function, and we just require a particular modular difference on Q_{64}, namely $-2^{25} + 2^{31}$, so that performing the final addition with the chaining variable cancels out this difference. Hence, we can omit completely the condition $Q_{64}[25] \doteq \mathtt{1}$, speeding up the process by a factor of 2.

Algorithm 4.2 Klíma's MD5 attack

Ensure: M and M' with the difference defined in (4.1) form a near-collision satisfying (4.3).
  repeat
    Choose $Q_3, Q_4, \ldots, Q_{16}$ arbitrarily, but fulfilling conditions (including T-conditions, see [21])
    Compute $m_6, m_7, \ldots, m_{15}$ from the Q-values just chosen
    repeat
      Choose Q_{17} arbitrarily, but fulfilling conditions
      $Q_{18} \leftarrow Q_{17} + (G(Q_{17}, Q_{16}, Q_{15}) + Q_{14} + m_6 + k_{17})^{\lll 9}$
      $Q_{19} \leftarrow Q_{18} + (G(Q_{18}, Q_{17}, Q_{16}) + Q_{15} + m_{11} + k_{18})^{\lll 14}$
    until all conditions on Q_{17}, Q_{18} and Q_{19} are fulfilled
    $m_1 \leftarrow (Q_{17} - Q_{16})^{\ggg 5} - G(Q_{16}, Q_{15}, Q_{14}) - Q_{13} - k_{16}$
    if $Q_{19}[31] = 0$ **then**
      $Q_{20} \leftarrow 0$
      $Z \leftarrow 2^{31} - 1$
    else
      $Q_{20} \leftarrow 2^{31}$
      $Z \leftarrow 2^{32} - 1$
    end if {*This is to ensure that the single condition on Q_{20} is fulfilled*}
    while $Q_{20} \leq Z$ and not all conditions are fulfilled **do**
      $m_0 \leftarrow (Q_{20} - Q_{19})^{\ggg 20} - G(Q_{19}, Q_{18}, Q_{17}) - Q_{16} - k_{19}$
      $Q_1 \leftarrow Q_0 + (F(Q_0, Q_{-1}, Q_{-2}) + Q_{-3} + m_0 + k_0)^{\lll 7}$
      $Q_2 \leftarrow Q_1 + (F(Q_1, Q_0, Q_{-1}) + Q_{-2} + m_1 + k_1)^{\lll 12}$
      Compute m_2, m_3, m_4, m_5 according to steps 2–5
      Compute all remaining step variables. If/when a condition is not fulfilled, let $Q_{20} \leftarrow Q_{20} + 1$ and continue
    end while
  until all conditions are fulfilled

We can even omit the condition on $Q_{63}[25]$, or at least modify the requirements. The f function call in step 63 (the last step) is

$$I(Q_{63}, Q_{62}, Q_{61}),$$

and we already require $Q_{61}[25] \doteq 1$. When $Q_{63}[25] = 1$ as we originally required, then the changes on bit 25 of Q_{63} and Q_{62} cause no change to the (25th bit of the) output of I, since we have

$$I(1,1,1) = I(0,0,1) = 0.$$

If instead $Q_{63}[25] = 0$ then the 25th bit of the output of I still does not change, since

$$I(0,1,1) = I(1,0,1) = 1.$$

However, in this case subtracting 2^{25} from Q_{63} causes a carry to propagate at least to bit 26. Assume that it propagates to bit s, $26 \leq s \leq 31$. Then the output of the f function does not change unless $1 \in Q_{61}[s-26]$, because whenever $Q_{61}[i]$ is 0, then $Q_{63}[i]$ has no influence on the output of the function. Hence, if $Q_{63}[25] = 0$ we require that for increasing i starting from 26, as long as $Q_{63}[i]$ is 0, $Q_{61}[i]$ must also be 0, and for the first i for which $Q_{63}[i]$ is 1, $Q_{61}[i]$ must still be 0. This improves the possibility that the requirement is fulfilled from $\frac{1}{2}$ to about $\frac{2}{3}$.
Note that if the carry propagates to bit 31 and stops there then $Q_{63}[31] = 1$, and since we require $Q_{63}[31] \doteq Q_{61}[31]$, we also know that $Q_{61}[31] = 1$. This case also leads to success (if $Q_{61}[i] = 0$ for $i \in \{26, \ldots, 30\}$), because in bit 31 we need a difference on the output of I, and this is achieved since we have

$$I(1, A, 1) = \neg I(1, \neg A, 0)$$

for any value of A. Of course, the probability of this happening is very low, only 2^{-12}.
We cannot allow the carry to propagate past bit 31, because then we have

$$I(0, A, 0) = I(0, \neg A, 1)$$

for any value of A.

The two improvements described here reduce the complexity of finding the second block of the message by a factor of about $3/8$.

4.8 An implementation

It seems that an optimal implementation of the Wang MD5 attack would consist of two parts: the first part, i.e. finding the first block, would optimally be done using the technique of Klíma [25], and the second part, i.e. finding the second block, would optimally be done using the technique of Wang et al. [45]. Surely it is possible to combine these two techniques, as the two parts are completely independent of one another, except for the chaining values to the second part. There are some conditions on these chaining values, but these are ensured already in the first part, so any first block satisfying the conditions can be used to find a second block.

The attack has been implemented in the C programming language. See Appendix C.

The program was run on a notebook computer with a 1.5 GHz Pentium M processor on the Linux platform. In these settings it found 100 collisions in a little over 44 hours, yielding an average of about $26\frac{1}{2}$ minutes per collision. The first part took on average about 18.4 minutes, and the second part took a little more than 8 minutes on average.
Note that without the improvements described in Section 4.7, the average time of finding a collision would be about 13 minutes longer.

Some of these collisions can be found at
`http://www.student.dtu.dk/~s001856/md5collisions/`.

4.9 Constructing meaningful collisions

The collisions found following the method described in this chapter are not likely to be of any use directly. Because of the message modifications, we have very little room to maneuver towards constructing meaningful colliding messages. However, we can exploit some document programming language constructs to create colliding messages that, when opened using a standard viewer of that sort of document, look authentic. The method described here is due to Magnus Daum [10].

For instance, in the Postscript language, an if-then-else construct can be exploited. This construct has the syntax (where the `typewriter` font means the actual characters, and symbols such as X_1 refer to some string)

$$\texttt{(}X_1\texttt{)}\ \texttt{(}X_2\texttt{)}\ \texttt{eq}\ \{M_1\}\ \{M_2\}\ \texttt{ifelse}$$

– meaning that if the strings X_1 and X_2 are equal, then M_1 is displayed, and otherwise M_2 is displayed.

Hence, let M_1 and M_2 be two versions of a document to be signed. One version, M_1, is the one whose MD5 hash will be signed by some party, S. The other version, M_2, is the version that some forger, F, wants the rest of the world to think that S signed. Hence, F is going to make M_1 and M_2 collide by changing these documents internally, but in a way such that these changes are not visible in a Postscript viewer such as Ghostview. F does this by first creating a one-block message B containing text that is not shown by Ghostview, for instance a comment, followed by a newline and the character '`(`'. He then computes the MD5 hash h_1 of this block without padding, and subsequently finds a collision (m_1, m_2) of MD5 with initial value h_1. He then creates the two Postscript documents

$$B\|m_1\texttt{)}\ \texttt{(}m_1\texttt{)}\ \texttt{eq}\ \{M_1\}\ \{M_2\}\ \texttt{ifelse}$$

and

$$B\|m_2\texttt{)}\ \texttt{(}m_1\texttt{)}\ \texttt{eq}\ \{M_1\}\ \{M_2\}\ \texttt{ifelse}$$

The first one will display M_1 in Ghostview, and the second one will display M_2. The two documents collide in MD5 since they only differ in the first occurrence of m_1 and m_2, and these form a collision when preceded by B.

Note that there is absolutely no restriction on M_1 and M_2, except that they should both contain a complete Postscript specification of a document. They don't even have to have the same size. Furthermore, virtually any collision (m_1, m_2) can be used to construct this forgery. The only condition is that neither m_1 nor m_2 contains a bracket character, '`(`' or '`)`'.

Also note that this kind of exploitation of a collision can be used for other hash functions as well, unless the collision requires a large number of message blocks. Hence, it actually removes the necessity that a collision be meaningful in itself whenever a forger is able to persuade someone to sign a document that he has only seen on a computer screen.

Two examples of Postscript documents that collide under MD5, but have completely different appearances in any Postscript viewer, can be found at `http://www.student.dtu.dk/~s001856/md5collisions/meaningful.html`.

Chapter 5

AES-based hash functions

In 2001 the RIJNDAEL encryption algorithm was selected by NIST as the new Advanced Encryption Standard, AES. Being standardised, implementations exist in virtually any programming language on many different combinations of platforms and architectures, and therefore it would be convenient if a cryptographic hash function based on AES existed. The hash function WHIRLPOOL, although inspired by AES, does not provide the added convenience of re-using AES components. For a description of AES see [19, 42].

AES supports three different key sizes, 128 bits, 192 bits, and 256 bits. By AES_κ we denote the version of AES using keys of κ bits.

In this chapter some suggestions to a new hash function truly based on AES components are given.

5.1 Block cipher-based hash functions in general

There are many advantages of basing a hash function on an existing block cipher. One advantage is, that if the block cipher is generally believed to be secure, then this level of security may be transferred to the hash function based on that block cipher. However, some attacks that are not a real threat to a block cipher may have much greater consequences for a hash function. This is for instance true in cases where a block cipher suffers from some weakness based on some properties of the key, and the hash function uses message blocks as keys for the encryption function. Since an adversary in a collision attack has complete control over the message, he might be able to exploit such weaknesses. Hence, there are extended security requirements for a block cipher that is to be used as the compression function of a hash function. In the following, unless the contrary is explicitly mentioned, it shall be assumed that AES has no weaknesses that can be exploited in mounting attacks on hash functions based on AES. Hence, we need only care about building a secure construction using AES.

Three general constructions of block cipher-based hash functions are usually mentioned in the literature, e.g. in [30, §9.4.1] (E_k is the encryption function of the

block cipher using key k):
The Davies-Meyer construction:

$$h_{i+1} = E_{m_i}(h_i) \oplus h_i \tag{5.1}$$

The Matyas-Meyer-Oseas construction:

$$h_{i+1} = E_{h_i}(m_i) \oplus m_i \tag{5.2}$$

The Miyaguchi-Preneel construction:

$$h_{i+1} = E_{h_i}(m_i) \oplus h_i \oplus m_i \tag{5.3}$$

Other schemes could be considered, of course. If four possibilities of inputs (m_i, h_i, $m_i \oplus h_i$, and some constant) are considered as key, plaintext and feed-forward value, then $4^3 = 64$ schemes can be constructed. Of these it has been shown [35] that assuming no weakness of the underlying block cipher can be exploited, only four are secure (among these are (5.2) and (5.3)). A further eight (among these are (5.1)) have the only weakness that fixed points can be found (see more on this in Section 6.3). Do note that if the underlying block cipher uses keys that are not the same size as the plaintext blocks, then either some transformation of inputs must be applied, or, of the three schemes mentioned, only the Davies-Meyer scheme can be used.

All these constructions are so-called *single-length*, meaning that they produce a hash result of the same size as the underlying block cipher. The rate of the constructions above is 1, signifying that one encryption compresses an n-bit block to produce an n-bit (intermediate) hash. In general, the rate of a block cipher-based hash function is $\rho = \frac{\mu}{n}$, where μ is the number of bits of message that are processed by each application of the compression function, and n is the block size of the underlying block cipher.

Most new hash functions proposed today have a hash result of size at least 256 bits in order to prevent birthday attacks from being effective in years to come. Since AES is a 128-bit block cipher, a single-length construction should not be used to form an AES-based hash function. Instead, a double-length scheme may be considered.

5.1.1 Fast double-length schemes

It is tempting to consider fast double-length schemes such as the Parallel-DM scheme, proposed in [22]:

$$\begin{aligned} h^1_{i+1} &= E_{m^1_i \oplus m^2_i}(h^1_i \oplus m^1_i) \oplus h^1_i \oplus m^1_i \\ h^2_{i+1} &= E_{m^1_i}(h^2_i \oplus m^2_i) \oplus h^2_i \oplus m^2_i \end{aligned}$$

– where E is some block cipher of block length n, and the message is split into 2μ-bit blocks (m^1_i, m^2_i), each m^j_i is μ bits of length. The hash rate of this scheme is

1, since one encryption is needed to process μ bits of the message. However, rate 1 schemes such as Parallel-DM and the general form

$$\begin{aligned} h^1_{i+1} &= E_{A_1}(B_1) \oplus C_1 \\ h^2_{i+1} &= E_{A_2}(B_2) \oplus C_2 \end{aligned} \tag{5.4}$$

(where A_j, B_j and C_j are linear combinations of h^1_i, h^2_i, m^1_i and m^2_i) were broken in [27]. This is not to say that all rate 1 schemes are insecure. At the time of this attack, however, most block ciphers used 64-bit blocks and ~64-bit keys.

5.1.2 DES and MDC-2

A double-length scheme was used in the 128-bit hash function MDC-2 using DES. DES uses 56-bit keys and processes blocks of 64 bits. In each iteration of MDC-2, 64 bits of message are processed by two instances of DES, creating a combined 128-bit output, which is then used to form the 2×56 bits of key for the next iteration:

$$\begin{aligned} l^1_{i+1} \| r^1_{i+1} &= E_{g_1(h^1_i)}(m_i) \oplus m_i \\ l^2_{i+1} \| r^2_{i+1} &= E_{g_2(h^2_i)}(m_i) \oplus m_i, \end{aligned}$$

where $|l^j_{i+1}| = |r^j_{i+1}|$, g_j are simple functions that remove 8 bits of h^j_i and fix two other bits, and

$$\begin{aligned} h^1_i &= l^1_i \| r^2_i \quad \text{and} \\ h^2_i &= l^2_i \| r^1_i . \end{aligned}$$

Hence, MDC-2 has rate $1/2$: 64 bits of message are processed by two applications of DES to produce a 128-bit output.

Other block ciphers than DES could be used, following the MDC-2 scheme. However, a rate $1/2$ scheme may not be fast enough. On the other hand, any scheme should keep the work of [27] in mind, and hence schemes of the form (5.4) are prohibited.

5.1.3 Using AES

AES can be used in a fashion that compresses data by letting a 256-bit message block be used as the key for AES.

Consider the following scheme:

$$\begin{aligned} l^1_{i+1} \| r^1_{i+1} &= E_{m_i}(h^1_i) \oplus h^1_i \\ l^2_{i+1} \| r^2_{i+1} &= E_{m_i}(h^2_i) \oplus h^2_i, \end{aligned} \tag{5.5}$$

where E is the AES_{256} encryption function, $|m_i| = 256$ and

$$\begin{aligned} h^1_i &= l^1_i \| r^2_i \quad \text{and} \\ h^2_i &= l^2_i \| r^1_i . \end{aligned} \tag{5.6}$$

for $1 \leq i < t$. If m consists of t blocks of 256 bits then we define $H(m) = l_t^1 \| r_t^1 \| l_t^2 \| r_t^2$.

h_0^1 and h_0^2 are fixed initial values, but they must be different. Otherwise, $l_i^1 = l_i^2$ and $r_i^1 = r_i^2$, and hence $h_i^1 = h_i^2$ for all $i < t$. In fact, if it happens for some k that $h_k^1 = h_k^2$ then $h_i^1 = h_i^2$ for all i, $k < i < t$, so we may fix some bits of h_i^j as was done in MDC-2. For instance, we might replace (5.6) with

$$\begin{aligned} h_i^1 &= (l_i^1 \| r_i^2) \wedge \neg 1_b \quad \text{and} \\ h_i^2 &= (l_i^2 \| r_i^1) \vee 1_b \end{aligned}$$

for $1 \leq i < t$. We define $H(m)$ as before.

This scheme has rate 1, since a 256-bit message is processed by two applications of AES_{256}, each producing 128 bits of output. The scheme is apparently not broken by [27], because it does not fall under the category of (5.4).

5.1.4 Related-key attack on AES_{256}

A *related-key* attack [13] exists on 9 rounds of AES_{256}. However, this attack requires 2^{77} plaintexts and takes time 2^{224}. The attack exploits the fact that the AES key schedule is far from optimal, and so it may be a good idea to consider new proposals for the key schedule.

It is an open question whether this related-key attack poses a threat to hash functions based on AES_{256}.

5.2 Extending AES to support 256-bit blocks

In the original Rijndael design, 256-bit message blocks were supported, but this feature was omitted in the AES standard. It is easy to see how AES could be extended to accept larger message blocks. A 256-bit message block can be represented internally as a 4×8 matrix. Then the functions SubBytes, ShiftRows and MixColumns are extended in a natural manner to operate on this larger matrix. The key expansion is extended to produce twice as much key material, since 15 keys of 256 bits are now needed.

Extending AES to support 256-bit blocks obviously requires a few changes to be made to the basic functions of AES. However, when considering software implementations, most of these changes involve only changes to counters and array sizes. Hardware implementations are a completely different matter, as in most cases new hardware would probably have to be built to accommodate larger block sizes. Note that the hash function suggested in Section 5.1 does *not* require new implementations of AES itself in software nor in hardware.

When a 256-bit implementation of AES exists, the hash function could be defined using e.g. the Miyaguchi-Preneel scheme (5.3).

5.3 Alternative constructions

Other constructions than (5.1)-(5.3) should be considered. In this section, two alternatives are mentioned.

5.3.1 The Lucks scheme

In [29], Stefan Lucks proposed two new constructions to improve the security of the Merkle-Damgård construction: the *wide-pipe* construction and the *double-pipe* construction. These are both extensions of the MD-construction.

The wide-pipe construction requires the existence of two compression functions, $f : \{0,1\}^s \times \{0,1\}^\mu \rightarrow \{0,1\}^s$ and $f' : \{0,1\}^s \rightarrow \{0,1\}^n$, where $s > n$, f is used in the usual way, and f' is a final compression yielding the hash. With $s > n$ this makes finding collisions for the compression function harder, but this construction is not suitable for block cipher-based hash functions, since usually the block size of the cipher is not large enough to substitute s.

The double-pipe construction uses only one compression function, $f : \{0,1\}^n \times \{0,1\}^{n+\mu} \rightarrow \{0,1\}^n$, where $\mu \geq n$ is the size of a message block. The hash function has the outline

$$\begin{aligned} h^1_{i+1} &= f(h^1_i, h^2_i \| m_i) \\ h^2_{i+1} &= f(h^2_i, h^1_i \| m_i), \end{aligned} \tag{5.7}$$

(where (h^1_0, h^2_0) is an initial pair of values) and the final hash value is defined

$$H = f(h^*, h^1_t \| h^2_t \| 0^{\mu - n}) \tag{5.8}$$

for some initial value h^*.

When based on a block cipher B, this scheme has rate $\frac{\mu}{2n}$, and it requires that B support key sizes that are different from the block sizes. Furthermore, the lesser of these sizes, n, must still be large enough that attacks requiring time $2^{n/2}$ are infeasible, i.e. n should preferably be at least 256. As mentioned in the previous section, AES can be extended to support block and key sizes of virtually any multiple of 32 bits, although other combinations than the standard ones have not been thoroughly examined.

5.3.2 The Knudsen-Preneel scheme

In [28], Knudsen and Preneel determine some necessary (but possibly not sufficient) conditions on multiple constructions to achieve a particular level of security, and they propose such constructions based on error-correcting codes.

Following the reasoning and the assumptions of [28], it is possible to create a hash function based on standard ($n = 128$) AES_{128} for which finding collisions has the complexity (at most) 2^{128} by using 3 parallel chains (i.e. $\nu = 3$), and this yields a hash function of rate $1/3$. Several other combinations with the same (upper bound on the) security level are possible.

Basing the construction on error-correcting codes

Knudsen and Preneel propose to base the construction of linear combinations of the chaining values and the message blocks that take part in each iteration on error-correcting codes. This way it is possible to ensure that when two sets of chaining values and message blocks differ, then inputs to at least d encryption functions differ, where d is the minimum distance of the code.

A simple differential attack exists on the Knudsen-Preneel construction when codes of minimum distance greater than 3 are used, since in these cases the conjectured complexities of attacks are too optimistic. However, for $d = 3$ we can hope that the complexity of a collision attack is exactly the conjectured 2^n, which is good enough when the hash function is based on AES.

In the following, two hash functions following the principles of [28] shall be described.

A hash function of 5 parallel chains

Using the $(5,3,3)$ shortened Hamming code over $GF(2^2)$ one obtains a hash function using five parallel chains. Hence, using standard AES_{128}, a single 128-bit message block is processed in each iteration, and so the hash function has rate $1/5$. The conjectured complexity of a collision attack is 2^{128}. Since the result has size 5×128 bits, it may be further compressed to 256 bits by using an *output transformation* $T : \{0,1\}^{640} \rightarrow \{0,1\}^{256}$. This compression function can be slow because of the small amount of data that it needs to compress. See Section 5.3.2 for some proposals.

It is clear that the amount of internal memory needed for a construction such as this is larger than when a small number of parallel chains are used. In this case, we need to store from one iteration to the next the 5 chaining values of 128 bits and the message block also of 128 bits, i.e. a total of 768 bits, or 24 32-bit words. In environments with limited resources this could be a problem.

The field $GF(2^2)$ is defined by the polynomial $p(x) = x^2 + x + 1$ (symbols in **bold** are elements of the field). Let $\mathbf{0} = 0$, $\mathbf{1} = 1$, $\alpha = x$ and $\beta = x + 1$. Then the $(5,3,3)$ code may be defined by the generator matrix

$$G = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \alpha \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \beta \end{bmatrix}.$$

Let us investigate what multiplication by α means in terms of the coefficients. Let $ax + b$ be some element of $GF(2^2)$ defined as above. Then $\alpha(ax+b) = x(ax+b) = ax^2 + bx = a(x+1) + bx = (a+b)x + a$. Hence, the most significant coefficient of the result is $a + b$, and the least significant coefficient is a. In other words, we may define this multiplication as a multiplication of a vector by a matrix as follows:

$$(ax+b)\alpha = \begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix},$$

and hence we define

$$M_\alpha = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

The elements **0** and **1** have the trivial corresponding matrices M_0 and M_1. Since $\alpha + \mathbf{1} = \beta$ in this field, we define the β-matrix as

$$M_\beta = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

We may now write the generator matrix using elements of $GF(2)$ as

$$\hat{G} = \begin{bmatrix} M_1 & M_0 & M_0 & M_1 & M_1 \\ M_0 & M_1 & M_0 & M_1 & M_\alpha \\ M_0 & M_0 & M_1 & M_1 & M_\beta \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

We then define a six-element vector V of chaining values and message blocks, in this case for instance $V = [h_i^1, h_i^2, h_i^3, h_i^4, h_i^5, m_i]$, where h_i^j is the jth chaining value. In order to determine the combinations of chaining values and message blocks used in each chain, we compute $V \times \hat{G}$ yielding

$$(V \times \hat{G})^T = \begin{bmatrix} h_i^1 \\ h_i^2 \\ h_i^3 \\ h_i^4 \\ h_i^5 \\ m_i \\ h_i^1 \oplus h_i^3 \oplus h_i^5 \\ h_i^2 \oplus h_i^4 \oplus m_i \\ h_i^1 \oplus h_i^3 \oplus h_i^4 \oplus m_i \\ h_i^2 \oplus h_i^3 \oplus h_i^5 \oplus m_i \end{bmatrix}.$$

We pair up the elements of this vector to obtain the input to the underlying encryption function, which means that with E_k the AES encryption function using the key k and $m_0, \ldots, m_{t-1}$ the message, we may define the complete hash function (excluding the output transformation) as

$$\left.\begin{array}{lcl} h_{i+1}^1 & \leftarrow & E_{h_i^1}(h_i^2) \\ h_{i+1}^2 & \leftarrow & E_{h_i^3}(h_i^4) \\ h_{i+1}^3 & \leftarrow & E_{h_i^5}(m_i) \\ h_{i+1}^4 & \leftarrow & E_{h_i^1 \oplus h_i^3 \oplus h_i^5}(h_i^2 \oplus h_i^4 \oplus m_i) \\ h_{i+1}^5 & \leftarrow & E_{h_i^1 \oplus h_i^3 \oplus h_i^4 \oplus m_i}(h_i^2 \oplus h_i^3 \oplus h_i^5 \oplus m_i) \end{array}\right\} \quad \text{for } 0 \le i < t$$

where $(h_0^1, h_0^2, h_0^3, h_0^4, h_0^5)$ is some initial value, $h_0^j \neq h_0^{j'}$ for $j \neq j'$ – in fact preferably all the h_0^j should be linearly independent. For instance, one could choose $h_0^j = \mathtt{0}^{j-1}\mathtt{1}\mathtt{0}^{128-j}$.

Note that in this scheme the message has no influence on the first chaining value in the first two iterations, and it has no influence on the second chaining value in the first iteration. Hence, a one-block message has no influence on the first two chaining values, and so the first 256 bits are identical for all one-block messages. Of course, the output transformation should spoil this relation between one-block messages.

Using a large code

In order to improve the rate of the hash function one may wish to choose a larger code over a larger field. For instance, in $GF(2^4)$ there exists a $(17,15,3)$ code, which can be used to form a hash function of rate $13/17$ using 17 parallel chains. Hence, this solution requires 2176 bits of memory to store the chaining values, or 68 32-bit words. This does not seem very practical, and it sure wouldn't be useful in some environments, but when resources are not too limited one may consider using this fairly quick hash function.

A generator matrix for the $(17,15,3)$ code is

$$G = \begin{bmatrix}
\mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\
\mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha \\
\mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^2 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^3 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^4 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^5 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^6 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^7 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^8 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^9 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^{10} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^{11} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \alpha^{12} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \alpha^{13} \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \alpha^{14}
\end{bmatrix}$$

where the field $GF(2^4)$ is defined by the polynomial $p(x) = x^4 + x + 1$, and $\alpha = x$ is a primitive element. Similarly to the previous example, we may exchange each element of $GF(2^4)$ with a 4×4 matrix over $GF(2)$ corresponding to multiplication by that element. For instance, multiplying $\alpha = x$ with the element $ax^3 + bx^2 + cx + d$ yields $ax^4 + bx^3 + cx^2 + dx = a(x+1) + bx^3 + cx^2 + dx = bx^3 + cx^2 + (a+d)x + a$,

so we replace α by the matrix

$$M_\alpha = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

Obviously, the elements **0** and **1** have the trivial corresponding matrices, and $M_{\alpha^j} = \prod_{i=1}^{j} M_\alpha$ for $1 \leq j \leq 14$. This way, we obtain the 4×4 matrices below, where a subscript j means the matrix is M_{α^j}.

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_1 \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}_2 \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}_3$$

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}_4 \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}_5 \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}_6$$

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}_7 \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}_8 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}_9$$

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}_{10} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}_{11} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}_{12}$$

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}_{13} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}_{14}$$

As above, we define a vector V containing the chaining values and the message blocks, but this time we need twice the number of elements as the combined number of chaining values and message blocks, and hence we split these in a left and a right part. The vector hence becomes

$$V = [h_i^{1,L}, h_i^{1,R}, \ldots, h_i^{17,L}, h_i^{17,R}, m_i^{1,L}, m_i^{1,R}, \ldots, m_i^{13,L}, m_i^{13,R}]$$

where $x_i^{j,L}$ means the 64 leftmost bits of x_i^j, and $x_i^{j,R}$ means the 64 rightmost bits of x_i^j. The combination of chaining values and message blocks to choose as input to each encryption function is then computed as $V \times \hat{G}$, where $\hat{G}$ is G with the field elements replaced by their corresponding 4×4 matrices over $GF(2)$. This computation shall not be performed here as the resulting vector becomes quite large. The resulting hash function, however, has the outline of Table 5.1.

C.v.	Input
h_{i+1}^{1}	$\{h_i^1, h_i^2\}$
h_{i+1}^{2}	$\{h_i^3, h_i^4\}$
h_{i+1}^{3}	$\{h_i^5, h_i^6\}$
h_{i+1}^{4}	$\{h_i^7, h_i^8\}$
h_{i+1}^{5}	$\{h_i^9, h_i^{10}\}$
h_{i+1}^{6}	$\{h_i^{11}, h_i^{12}\}$
h_{i+1}^{7}	$\{h_i^{13}, h_i^{14}\}$
h_{i+1}^{8}	$\{h_i^{15}, h_i^{16}\}$
h_{i+1}^{9}	$\{h_i^{17}, m_i^1\}$
h_{i+1}^{10}	$\{m_i^2, m_i^3\}$
h_{i+1}^{11}	$\{m_i^4, m_i^5\}$
h_{i+1}^{12}	$\{m_i^6, m_i^7\}$
h_{i+1}^{13}	$\{m_i^8, m_i^9\}$
h_{i+1}^{14}	$\{m_i^{10}, m_i^{11}\}$
h_{i+1}^{15}	$\{m_i^{12}, m_i^{13}\}$
h_{i+1}^{16}	$\{h_i^1 \oplus h_i^3 \oplus h_i^5 \oplus h_i^7 \oplus h_i^9 \oplus h_i^{11} \oplus h_i^{13} \oplus h_i^{15} \oplus h_i^{17} \oplus m_i^2 \oplus m_i^4 \oplus m_i^6 \oplus m_i^8 \oplus m_i^{10} \oplus m_i^{12}, h_i^2 \oplus h_i^4 \oplus h_i^6 \oplus h_i^8 \oplus h_i^{10} \oplus h_i^{12} \oplus h_i^{14} \oplus h_i^{16} \oplus m_i^1 \oplus m_i^3 \oplus m_i^5 \oplus m_i^7 \oplus m_i^9 \oplus m_i^{11} \oplus m_i^{13}\}$
h_{i+1}^{17}	$\{h_i^1 \oplus (h_i^{3,R} \Vert h_i^{4,L}) \oplus h_i^6 \oplus h_i^7 \oplus (h_i^{8,R} \Vert h_i^{5,L}) \oplus h_i^9 \oplus (h_i^{9,R} \Vert h_i^{7,L}) \oplus (h_i^{11,R} \Vert h_i^{10,L}) \oplus h_i^{12} \oplus h_i^{13} \oplus h_i^{14} \oplus (h_i^{14,R} \Vert h_i^{11,L}) \oplus (h_i^{15,R} \Vert h_i^{12,L}) \oplus (h_i^{16,R} \Vert h_i^{15,L}) \oplus h_i^{17} \oplus m_i^1 \oplus m_i^2 \oplus (m_i^{2,R} \Vert h_i^{16,L}) \oplus (m_i^{3,R} \Vert h_i^{17,L}) \oplus m_i^4 \oplus (m_i^{4,R} \Vert m_i^{2,L}) \oplus m_i^5 \oplus m_i^6 \oplus (m_i^{6,R} \Vert m_i^{3,L}) \oplus m_i^7 \oplus (m_i^{7,R} \Vert m_i^{4,L}) \oplus (m_i^{8,R} \Vert m_i^{5,L}) \oplus m_i^9 \oplus (m_i^{9,R} \Vert m_i^{7,L}) \oplus m_i^{11} \oplus (m_i^{11,R} \Vert m_i^{9,L}) \oplus (m_i^{13,R} \Vert m_i^{12,L}), h_i^2 \oplus (h_i^{3,L} \Vert h_i^{3,L}) \oplus (h_i^{4,R} \Vert h_i^{8,L}) \oplus h_i^5 \oplus (h_i^{5,R} \Vert h_i^{9,L}) \oplus (h_i^{7,R} \Vert h_i^{11,L}) \oplus (h_i^{8,L} \Vert h_i^{11,R}) \oplus (h_i^{9,L} \Vert h_i^{14,L}) \oplus h_i^{10} \oplus (h_i^{10,R} \Vert h_i^{15,L}) \oplus (h_i^{11,R} \Vert h_i^{16,L}) \oplus (h_i^{12,R} \Vert h_i^{16,R}) \oplus h_i^{13} \oplus (h_i^{14,L} \Vert m_i^{2,L}) \oplus (h_i^{15,L} \Vert m_i^{3,L}) \oplus (h_i^{15,R} \Vert m_i^{4,L}) \oplus (h_i^{16,R} \Vert m_i^{4,R}) \oplus h_i^{17} \oplus (h_i^{17,R} \Vert m_i^{6,L}) \oplus m_i^1 \oplus (m_i^{2,L} \Vert m_i^{6,R}) \oplus (m_i^{2,R} \Vert m_i^{7,L}) \oplus (m_i^{3,L} \Vert m_i^{8,L}) \oplus (m_i^{3,R} \Vert m_i^{8,R}) \oplus (m_i^{4,R} \Vert m_i^{9,L}) \oplus m_i^5 \oplus (m_i^{5,R} \Vert m_i^{9,R}) \oplus (m_i^{7,L} \Vert m_i^{11,L}) \oplus (m_i^{7,R} \Vert m_i^{11,R}) \oplus (m_i^{9,R} \Vert m_i^{13,L}) \oplus m_i^{10} \oplus (m_i^{12,R} \Vert m_i^{13,R})\}$

Table 5.1: An outline of the hash function defined by the $(17, 15, 3)$ code. The first column is the chaining value being computed, and the second column is a pair (in braces) representing the input to the encryption function, where the first element is the key and the second element is the message block.

The input to the last chaining value is fairly complex; it consists of 62 XORs. This obviously adds to the complexity of the entire hash function, and furthermore it is not very practical. Another problem is that we need at least 13 message blocks of 128 bits in order to perform a single iteration. This could be achieved by padding, but it would be better to use some message expansion function that (for each iteration) makes each block dependent on all the other blocks. The output transformation $T : \{0,1\}^{2176} \rightarrow \{0,1\}^{256}$ should (again) make all 256 output bits dependent on all chaining values, since it takes 5 iterations for the message to affect the first chaining value. In fact, m_0^{13} never affects h_i^j for $j \in \{6,7,10,11,12,13,14\}$ and any i, no matter how large the message is. To avoid this undesirable feature one could change the order of the elements of V. For instance, the first 13 chaining values could be computed as $h_{i+1}^j \leftarrow E_{m_j}(h_i^{j+2})$, $0 < j \leq 13$, the next two as $h_{i+1}^{14} \leftarrow E_{h_i^1}(h_i^{16})$ and $h_{i+1}^{15} \leftarrow E_{h_i^2}(h_i^{17})$, and last two chaining values would then also have different specifications, according to the new vector V. It would then take at most 9 iterations for any message block to affect all chaining values.

Possible output transformations

As mentioned, the output transformation T should be secure rather than efficient, since it only has to compress a fixed, limited number of bits into 256 bits. Even schemes based on fairly slow mathematical operations may be considered. It should be computationally infeasible to find collisions and (2nd) preimages for T as well.

Hashing the chaining values. Knudsen and Preneel propose some possible output transformations. One such proposal is to compress the ν chaining values using the hash function itself in order to make each bit dependent on every other bit, and then truncate the result. However, as mentioned in the previous section, a message block (in this case a chaining value) does not necessarily affect all "new" chaining values. There are many ways of making sure that the final output depends on all chaining values. For instance, further iterations of the compression function could be applied, but this has to be done carefully to make sure that the bits are mixed properly. Also, it might be a good idea to do the final truncation by dropping all chaining values except the last two, i.e. $h^{\nu-2}$ and $h^{\nu-1}$, because these two chaining values have the largest degree of dependency on message blocks and other chaining values.

A CBC-like application. Applying a CBC-like structure would also have the effect that all output bits depend on all chaining value bits. For instance, let h_t^j be the final chaining values of the "original" hash function, $0 \leq j < \nu$. Define $H_{-1} = 0$ and compute

$$H_j \leftarrow E_{H_{j-1}}(h_t^j) \quad \text{for } 0 \leq j < \nu$$

Repeat this once, i.e. let $\hat{H}_{-1} = 0$ and compute

$$\hat{H}_j \leftarrow E_{\hat{H}_{j-1}}(H_j) \quad \text{for } 0 \leq j < \nu$$

Define $\hat{H}_{\nu-2} \| \hat{H}_{\nu-1}$ as the output. This solution would require an additional 2ν applications of the AES encryption function. The reason for choosing two CBC-like applications is that two output blocks are needed, and only one of the blocks H_j depend on all chaining values h^t_j. To achieve symmetry, the CBC processing is repeated. This method would probably be faster than the one mentioned above, and it is also simpler as one does not have to take into account which chaining values are affected by which message blocks etc.

A method using modular arithmetic. As a final proposal for the output transformation, consider a compression function based on modular arithmetic, for instance the Chaum-van Heijst-Pfitzmann hash function discussed in Section 7.2. The fact that this hash function is quite slow is less significant in this context, since only a fixed and very limited amount of data needs to be hashed. Additionally, it is proved that if one is able to find a collision for the Chaum-van Heijst-Pfitzmann hash function, then one is also able to compute a non-trivial discrete logarithm. However, since we are really looking for a (complete) hash function entirely based on AES, this solution might not be feasible.

5.4 Summary

There are many ways of building a hash function based on AES, but doing so in a secure and efficient manner is not easy. A general problem of block cipher based hash functions stems from the fact that for each call of the underlying block cipher, the key scheduling of the block cipher has to be applied (except in cases where the key is fixed, but none of these schemes are secure according to [35]). In some block ciphers the key scheduling algorithm is rather slow. This is not particularly the case for AES.

One of the quickest proposals of those given in this chapter is that of Section 5.1.3, and this one can even be used with AES as a "black box" whenever this accepts 256-bit keys, which is standard. This version is not trusted quite as much as the 128-bit version, however. Do note that key scheduling only has to be performed once for every two applications of AES, as the two applications in each step use the same key.

The proposal of Section 5.2 does not work with standard AES implementations, as it requires the use of 256-bit blocks. It is quite easy to extend AES to accept this size of blocks, but implementations, especially hardware implementations, still require some inconvenient changes to be made. The speed would probably be about the same as in the proposal of Section 5.1.3.

The two proposals making use of error-correcting codes have quite different properties. The first one uses 5 parallel chains, and hence requires 5×128 bits

of internal memory to pass on the chaining values from step to step. It is not faster than using an MDC-2-like construction with AES_{128}. The second proposal is almost as fast as the two mentioned above, but it requires a significant amount of internal memory, namely 17×128 bits. Both proposals come with the additional challenge of identifying a secure and convenient output transformation. On the other hand, they both have the advantage that they are able to use AES as a black box.

Considering the large number of existing implementations of AES in both software and hardware, and the recent attacks on MD4-like hash functions, it may be worth considering any proposal based on AES that is able to use AES as a black box.

Chapter 6

General results on the Merkle-Damgård construction

In this chapter some general results on all hash functions using the Merkle-Damgård construction with MD-strengthening are presented.

6.1 The motivation for using MD-strengthening

As mentioned in Chapter 2, the Merkle-Damgård construction with MD-strengthening makes it possible to prove Theorem 6.1. Different but equivalent constructions were suggested by Ralph C. Merkle [31] and Ivan Damgård [9] independently.

Theorem 6.1. *Let H be a hash function based on the Merkle-Damgård construction, where the message is padded with a number of bits representing the length of the unpadded message (i.e. MD-strengthening is used), and let f be the compression function of H. Then, a collision for H implies a collision for f.*

Proof. Assume that we have found x and x', $x \neq x'$, such that $H(x) = H(x')$. Let m and m' respectively be the padded versions of x and x'. Note that the two messages are padded in the exact same way if and only if $|x| = |x'|$. Define h_0 as the initial value of the hash function, and let $h_{i+1} = f(m_i, h_i)$, where m_i, $0 \leq i < t$, are the message blocks of m. Define similarly $h'_{i+1} = f(m'_i, h'_i)$, $0 \leq i < t'$.
Since $h_t = h'_{t'}$, either there is a collision for f or $(m_{t-1}, h_{t-1}) = (m'_{t'-1}, h'_{t'-1})$. In the latter case, either there is a collision for f or $(m_{t-2}, h_{t-2}) = (m'_{t'-2}, h'_{t'-2})$. This argument repeats. If $|x| = |x'|$ then either there is a collision for f, or $m_i = m'_i$ for all i, $0 \leq i < t$. In the latter case $x = x'$, which is impossible. If $|x| \neq |x'|$ then at least one of the padding blocks differ for the two messages, and hence $m_{t-1-j} \neq m'_{t'-1-j}$ for some j, where $0 \leq j < \min(t, t')$. Thus, there is a collision for f. Note that in both cases the collision can occur later than the last iteration in which the message blocks differ, but then there is still a collision for f since the chaining values must differ. □

Theorem 6.1 has the implication that if one is able to construct a collision resistant compression function f, then a collision resistant hash function can be easily built from f. However, as we have seen, it is not a simple task to construct a collision resistant compression function.

In the following sections, some general attacks on the Merkle-Damgård construction are described.

6.2 Joux's multicollisions

In [23], Antoine Joux develops a new method for finding multicollisions, i.e. sets of messages $\{M_1, \ldots, M_j\}$, $j > 1$, that all have the same hash value.

The method of Joux assumes that the attacker has access to a collision finder $C : \{0,1\}^n \to \{0,1\}^\mu \times \{0,1\}^\mu$ of the hash function H. C takes as input an initial chaining value v of size n bits and outputs two message blocks M_0 and M_1 that collide under one iteration of the compression function given the initial chaining value v. C can be assumed to run in time T, where e.g. $T = 2^{n/2}$ if C uses the brute force method.

To find j messages that all have the same hash under H, perform the operations of Algorithm 6.1, where $f : \{0,1\}^\mu \times \{0,1\}^n \to \{0,1\}^n$ is the compression function of H. Since all messages have the same length, and are built from message blocks

Algorithm 6.1 Multicollision finder

Ensure: The $j = 2^\lambda$ different messages $M_\beta^0 \| M_\beta^1 \| \cdots \| M_\beta^{\lambda-1}$, $\beta \in \{0,1\}$, all have the same hash under H.

Let v be the initial value of H
for $i = 0$ to $\lambda - 1$ **do**
  $(M_0^i, M_1^i) \leftarrow C(v)$
  $v \leftarrow f(M_0^i, v)$
end for

that collide under the compression function given the proper initial value, all the 2^λ different concatenations (in the right order) of the λ collisions have the same hash value under H. This algorithm operates in time λT on average. If H was a random oracle, it would take time $2^{n(j-1)/j}$ to find j messages with the same hash. Even with $T = 2^{n/2}$, which is always possible, $\lambda T < 2^{n(j-1)/j}$ as long as $\lambda > 1$.

This result is interesting for more than the obvious reasons. For example, it has been suggested (for instance by Preneel [34]) that the security of a hash function may be improved by concatenating the hash results of two different hash functions, alternatively two different applications (using for instance different initial values) of the same hash function. This is called a cascaded construction. Let a hash function H be defined as $H(m) = H_1(m) \| H_2(m)$, where H_1 is an n_1-bit hash function, and H_2 is an n_2-bit hash function. Assume without loss of generality that $n_1 \leq n_2$. Find (in time $\sim n_2 2^{n_1/2}$) at least $2^{n_2/2}$ messages that all have the same hash under

H_1. Then, with a probability of about $1/2$, two of these messages also collide under H_2, and hence we have found a collision of H in time $n_2 2^{n_1/2} + 2^{n_2/2}$, including the time it takes to find the pair that also collides under H_2. For a truly random mapping into $(n_1 + n_2)$-bit values, it takes time $2^{(n_1+n_2)/2}$ to find a collision.

Note that if $n_1 = n_2$, and one has access to a collision finder of H_1 running in time T such that $n_2 T < 2^{n_2/2}$, then this method has complexity $2^{n_2/2}$, which is not much better than the case where the multicollisions are found by brute force.

Multicollisions can also be used to improve the speed of finding (2nd) preimages of cascaded constructions. The method will not be described here.

6.3 Kelsey/Schneier's 2nd preimage attack

MD-strengthening was introduced in order to make the hash function more secure. However, as this section intends to prove, it does not add much resistance against 2nd preimage attacks. The results of this section are due to Kelsey and Schneier [24], and are in part inspired by the work of Joux described in the previous section.

Most compression functions have *fixed points* that can easily be found. A fixed point in this context is a pair (m_i, h_i) such that $h_{i+1} = f(m_i, h_i) = h_i$, where f is the compression function of the hash function. Hash functions such as MD4, MD5, SHA-1 etc. are constructed in a way similar to the Davies-Meyer scheme using an invertible block-cipher-like function E, i.e. $f(m_i, h_i) = E_{m_i}(h_i) \oplus h_i$. Fixed points can be found for the compression function of all such hash functions as follows. Given message block m_i compute (easily) $h_i = E_{m_i}^{-1}(\omega)$, where ω is the neutral element with respect to $\oplus$, usually 0. Then $h_{i+1} = E_{m_i}(h_i) \oplus h_i = h_i$, and hence (m_i, h_i) is a fixed point. Note that m_i can be chosen arbitrarily, but it does not give any control over h_i. However, this means that a huge number of fixed points exist for most compression functions, and these can be easily computed.

This fact can be exploited, which we shall see. However, there is another (slightly slower) technique for finding 2nd preimages of hash functions based on the Merkle-Damgård construction that does not require the existence of fixed points in the compression function. For a description of this technique the reader is referred to [24].

6.3.1 Expandable messages

An *expandable message* is a set of messages of different lengths that have the same intermediate hash value, excluding padding and MD-strengthening, under some hash function H. A (k_1, k_2)-expandable message ($k_2 > k_1$) is a set of $k_2 - k_1 + 1$ messages of lengths (number of blocks) $k_1, k_1 + 1, \ldots, k_2$ that all have the same intermediate hash.

Expandable messages can be found relatively quickly as follows. Find $2^{n/2}$ fixed points (m_i, h_i) and the same number of pairs (m'_i, h'_i) where $h'_i = f(m'_i, h_0)$, h_0 is the initial value of the hash function (both take time $2^{n/2}$). With a good

probability there is a match between the h_i and the h'_i, assume that $h'_{j_1} = h_{j_2}$. This means that the messages m'_{j_1}, $m'_{j_1}\|m_{j_2}$, $m'_{j_1}\|m_{j_2}\|m_{j_2}$, $\ldots$ all have the same hash value (excluding the padding block), and hence in time about $2^{(n/2)+1}$ we have found a $(1,k)$-expandable message with $1 < k \leq K$, where K is the largest number of blocks that a message can contain in H.

Note that an expandable message is entirely defined by just two message blocks, so we may denote the expandable message above by (m'_{j_1}, m_{j_2}).

6.3.2 Finding a 2nd preimage

Expandable messages can be used to find 2nd preimages for very long messages as follows. Let the target message be $M = m_0\|\cdots\|m_t$ excluding padding (here, $|m_t|$ may be less than e.g. $|m_0|$). Compute the first t chaining values h_i, $0 < i \leq t$. Find a $(1,t)$-expandable message (μ_0, μ_r), and let ι be the common intermediate hash value of these messages. Keep computing $f(b_j, \iota)$ for random message blocks b_j until some message block B is found such that $f(B, \iota) = h_\ell$ for some ℓ, where $0 < \ell \leq t$ (i.e. this chaining value was reached by processing $m_0\|\cdots\|m_{\ell-1}$). Then, a 2nd preimage of M is the message

$$M' = \mu_0\|\underbrace{\mu_r\|\cdots\|\mu_r}_{\ell-2 \text{ blocks}}\|B\|m_\ell\|m_{\ell+1}\|\cdots\|m_t,$$

which will be padded in the exact same way as M, since it has the same size.

This procedure is expected to take time $2^{(n/2)+1} + 2^{n-\log_2 t}$, since finding the expandable message takes time $2^{(n/2)+1}$, and finding a match on the chaining values takes an expected time of $\frac{2^n}{t}$. Hence, t must be rather large for this attack to be of any use. Nevertheless, the result is fairly strong as finding 2nd preimages using the brute force method requires time 2^n, and this technique can be used for any hash function based on the Merkle-Damgård construction (as long as fixed points can be easily found for the compression function, but, as mentioned, even if they can't there is a technique that is only slightly slower than the one just described).

Chapter 7

Hash functions based on modular arithmetic

Most hash functions are based on logical bitwise operations such as AND, OR and XOR, because these operations are fast in both hardware and software. However, they must be mixed with other kinds of operations in order to introduce non-linearity, and most often modular additions and rotations are used for this purpose. Hash functions that are truly based on modular arithmetic do exist. These have not become very popular, since they are quite inefficient compared to hash functions based on logical operations, but they do have some advantages like easy scalability, and in some cases provable security.

7.1 A simple hash function

A simple example of a hash function entirely based on modular arithmetic is the following. Let p and q be large, secret primes, and let $N = pq$. Define some integer $a > 1$. Let x be the message to be hashed. To compute the hash $H(x)$ perform

$$H(x) = a^x \bmod N.$$

If one is able to find x and x' such that $H(x) = H(x')$, then $\phi(N)$ divides $x - x'$. Hence, two colliding messages are very far apart with respect to modular addition. Furthermore, if we have found say 10 (independent) collisions (x_i, x_i'), then since $\phi(N)$ divides $x_i - x_i'$ for all i, there is a good probability that the greatest common divisor of all these differences is equal to $\phi(N)$ – or at least it should be a small multiple of $\phi(N)$. Knowing $\phi(N)$ it is simple to find p and q. To do this, solve

$$\rho^2 - \rho(N - \phi(N) + 1) + N = 0$$

for ρ. The two solutions are p and q, and hence N is factorised. This demonstrates that finding a small number of collisions is at least as hard as factoring N.

An alternative demonstration is possible if x is limited in size, for instance by requiring $x < 4N$. In this case, any collision will be a small multiple of $\phi(N)$ apart,

and hence only a single collision is needed in practice to find $\phi(N)$, and from that the factorisation of N. Of course, with such a limitation on x, H can be used only as a compression function. The compression rate, however, is not very high, as only two bits of message can be processed in each application of H.

To resist preimage attacks it should be made sure that a^x is always greater than N. This can be done for instance by appending a `1`-bit to x on the left.

This scheme has probably never been used in practice, and it obviously has some disadvantages. Most of these are also present in the hash function MASH-1, which is described in Section 7.3.

7.2 The Chaum-van Heijst-Pfitzmann hash function

One could fear that the main reason why this hash function is not used in practice is its tongue twisting name. It sure does have one clear advantage to other hash functions, which we shall come back to after having defined it.

Definition 7.1 (The Chaum-van Heijst-Pfitzmann hash function)**.** Let p and q be large (and hence odd) primes such that $p-1=2q$. Let $\mathbb{Z}_p$ be the field where addition and multiplication is performed modulo p (and define $\mathbb{Z}_q$ similarly), and let α and β be primitive elements of $\mathbb{Z}_p$. α and β are public, but it is believed that it is computationally infeasible to compute $\log_\alpha \beta$. Let $m=(m_1,m_2)$ be the message to be hashed such that $m_i \in \mathbb{Z}_q$, $i \in \{1,2\}$. The hash of m is the defined as

$$h(m) \leftarrow \alpha^{m_1}\beta^{m_2} \bmod p.$$

Hence, h maps elements of $\mathbb{Z}_{q^2}$ into $\mathbb{Z}_p$, so it might be used as the compression function of some hash function in order to accommodate messages of arbitrary length.

Note that we deliberately select the parameters of the hash function large enough that the discrete logarithm mentioned should be infeasible to compute.
The interesting point of this (admittedly inefficient) hash function is Theorem 7.2.

Theorem 7.2. *If a collision of the Chaum-van Heijst-Pfitzmann hash function is found, then the discrete logarithm* $\log_\alpha \beta$ *can be computed in negligible time.*

Proof. The proof is from [41].
Assume that a collision (m,m'), $h(m)=h(m')$ and $m \neq m'$ has been found. Then

$$\alpha^{m_1}\beta^{m_2} = \alpha^{m'_1}\beta^{m'_2} \pmod{p}$$

meaning that

$$\alpha^{m_1-m'_1} = \beta^{m'_2-m_2} \pmod{p}$$

First note that if $m_1=m'_1$ then $m_2=m'_2$, which is impossible. Hence, $\gcd(m'_2-m_2,q)=1$, since both m_2 and m'_2 are at most $q-1$. Therefore m'_2-m_2 has an inverse, γ, modulo q. I.e.

$$(m'_2-m_2)\gamma = 1 \pmod{q} \Rightarrow (m'_2-m_2)\gamma = kq+1$$

for some integer k. Since β has order $p-1$, and $p-1 = 2q$ it must be the case that $\beta^q = -1 \pmod{p}$. Hence,

$$\beta^{(m'_2 - m_2)\gamma} = \beta^{kq+1} = (-1)^k \beta \pmod{p},$$

and therefore, if k is even (meaning that $(m'_2 - m_2)\gamma$ is odd) then $\log_\alpha \beta = (m_1 - m'_1)\gamma$. If k is odd then, since $\alpha^q = -1 \pmod{p}$ (just as we showed for β), the discrete logarithm can be computed as $\log_\alpha \beta = (m_1 - m'_1)\gamma + q$. □

Computing the discrete logarithm is generally believed to be a difficult task, and since finding collisions has just been proven to be at least as hard, this is a strong result. 2nd preimages are always at least as difficult to find as collisions. However, it is not possible to prove a similar result for preimages, although it does seem difficult to find these except in rare cases such as $h(m) \in \{1, \alpha, \beta\}$.

This hash function can be used as a compression function using the Merkle-Damgård construction. If $|q-1| = n$, then each message block is $n-1$ bits. The $(n+1)$-bit output of the compression function is then concatenated with the next message block, and the combined $2n$ bits are subsequently split into two n-bit entities that play the roles of m_1 and m_2 in Definition 7.1.

7.3 MASH-1 and MASH-2

Another hash function based on modular arithmetic is MASH-1 (see e.g. [30, §9.4.3]). The setup of MASH-1 is far from elegant, and like the Chaum-van Heijst-Pfitzmann hash function it suffers from inefficiency.

Definition 7.3 (MASH-1)**.** Let p and q be two large primes that are kept secret, and define $N = pq$. Let n, the bit length of the hash value, be the greatest multiple of 16 not greater than $|N|$. Define $h_0 = 0$ and $a = \texttt{f0}\ldots\texttt{0}_h$.
If necessary, pad the message with `0`-bits until it has a length that is a multiple of $\frac{n}{2}$. Append an extra $\frac{n}{2}$-bit block containing a representation of the length of the original message. The padded message consists of t blocks $x_0, \ldots, x_{t-1}$.
Expand each message block by preceding each four-bit nibble with $\texttt{1111}_b$, except the last message block for which the four-bit nibble inserted is instead $\texttt{1010}_b$. The expanded message blocks are denoted $m_0, \ldots, m_{t-1}$.
Define

$$h_{i+1} \leftarrow ((((h_i \oplus m_i) \vee a)^2 \bmod N) \bmod 2^n) \oplus h_i \quad \text{for } 0 \le i < t$$

The hash result is h_t.

This hash function has the obvious drawback of inefficiency. It also has a number of other disadvantages. For one, the primes p and q must be chosen by someone, but at the same time they should be secret. There are ways to do this (see e.g. [4]), but it is not a very practical feature of a hash function.

Another problem is the message expansion. Not only does it halve the speed of the hash function, but it is also inconvenient. However, there would be easy ways of finding collisions had it not been introduced.

Finally, the security of the hash function is not based on the size of the modulus, but on the size of the factors, meaning that if p and q are about the same size then a collision can be found in time $\sqrt{p} \approx 2^{n/4}$ and a (2nd) preimage in time $p \approx 2^{n/2}$. A collision is found as follows. First factorise N, which can be done in time much less than $\sqrt{p}$, then in time $\sqrt{p}$ find a collision, say (x, x'), modulo p. Use the trivial collision (x, x) modulo q, and, using the Chinese remainder theorem, obtain a collision modulo $pq = N$. This method also in part demonstrates the motivation for constructing this hash function: if N cannot be factorised, it does not seem straight-forward to find collisions and (2nd) preimages. Do note that this is in no way a proof that if one is able to find for instance a collision, then one can factor N.

A substitute for MASH-1, called MASH-2, has been proposed. Here, the exponent of the compression function is $2^8 + 1$ instead of 2 as in MASH-1.

Chapter 8

The SMASH hash function

SMASH is a hash function proposal by Lars R. Knudsen [26], first presented at FSE 2005. The proposal is broken [33] in the sense that collisions can be found in negligible time. We first describe the hash function, then the collision attack, and finally we try to improve the security of SMASH by making some changes.

8.1 The design of SMASH

The motivation for the development of the SMASH proposal is the fact that, as previous chapters have shown, quite quickly the hash functions of the MD4 family are becoming vulnerable. SMASH introduces a design that is different in both the overall construction and in the compression function.

The idea behind the proposal is to use a bijective mapping in the compression function, which in this construction makes the compression function invertible. Due to feed-forward applications of the bijective mapping in the beginning and in the end, the complete hash function is not invertible. However, it is easy to show that (2nd) preimages can be found in time $2^{n/2}$.

Two different versions of SMASH were proposed; a 256-bit version called SMASH-256 and a 512-bit version called SMASH-512. It shall be made clear when the description of the design refers to a particular version (see also [26] for further details).

8.1.1 The construction

The construction can be expressed as follows. Let $m = m_0 \| m_1 \| \cdots \| m_{t-1}$ be the message to be hashed, padded as in Rule 1, with $u = \frac{n}{2}$. Let v be the all-zero vector of same length as m_i. Then compute

$$h_0 \leftarrow f(v) \oplus v \tag{8.1}$$

$$h_{i+1} \leftarrow f(h_i \oplus m_i) \oplus h_i \oplus \theta m_i \quad \text{for } 0 \leq i < t \tag{8.2}$$

$$h_{t+1} \leftarrow f(h_t) \oplus h_t \tag{8.3}$$

The output of the hash function is h_{t+1}. The bijective mapping $f : \{0,1\}^n \rightarrow \{0,1\}^n$ (also called the *core function*) and the operation θm_i will be explained in the following section. Note that in the first general presentation of this construction it is specified that in a concrete hash function it is not allowed that $\theta = 0$ or $\theta = 1$.

8.1.2 The compression function

The components of the compression function are now described.

The core function

The description of the core function is based on SMASH-256. It consists of a number of rounds of two types, H and L. These shall be described in the following subsection.

The outline of f is

$$\begin{aligned} f(\alpha) &= H_1 \circ H_3 \circ H_2 \circ L \circ H_1 \circ H_2 \circ H_3 \circ L \circ \\ & \quad H_2 \circ H_1 \circ H_3 \circ L \circ H_3 \circ H_2 \circ H_1(\alpha) \end{aligned} \tag{8.4}$$

Let $\alpha = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$, i.e. each a_i is 32 bits.

H-rounds. As unveiled, there are three kinds of H-rounds, H_1, H_2 and H_3. The H_j-round consists of the following 8 steps:

1. $(a_7, a_6, a_5, a_4) \leftarrow S_j(a_7, a_6, a_5, a_4)$
2. $a_{i+4} \leftarrow a_{i+4} + a_i^{\lll r_{i,j}}$ for $0 \leq i < 4$
3. $(a_3, a_2, a_1, a_0) \leftarrow S_j(a_3, a_2, a_1, a_0)$
4. $a_i \leftarrow a_i + a_{i+4}^{\lll r_{i+4,j}}$ for $0 \leq i < 4$
5. $(a_7, a_6, a_5, a_4) \leftarrow S_j(a_7, a_6, a_5, a_4)$
6. $a_{i+4} \leftarrow a_{i+4} + a_i^{\lll r_{i+8,j}}$ for $0 \leq i < 4$
7. $(a_3, a_2, a_1, a_0) \leftarrow S_j(a_3, a_2, a_1, a_0)$
8. $a_i \leftarrow a_i + a_{i+4}^{\lll r_{i+12,j}}$ for $0 \leq i < 4$

Here, S_j is a 4-bit (bijective) S-box lookup in bitslice mode, meaning that the ith bit ($0 \leq i < 32$) of the four inputs are viewed as a single 4-bit entity, with the bit from the first input being the most significant bit. The output of S is a quadruple, assembled in the reverse way of the disassembling of the input. The three S-boxes look as follows.

$S_j(x)$		x: 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	6	13	12	7	15	1	3	10	8	11	5	0	2	4	14	9
j	2	1	11	6	0	14	13	5	10	12	2	9	7	3	8	15	4
	3	4	2	9	12	8	1	14	7	15	5	0	11	6	10	3	13

The rotations are also different in each H-round:

$r_{i,j}$		i															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	1	19	18	17	7	1	7	26	20	0	16	20	5	28	2	20	4
j	2	22	29	12	4	18	2	13	29	26	20	16	29	18	4	10	9
	3	4	21	19	5	24	20	12	16	14	30	3	4	23	15	13	12

***L*-rounds.** There is only one type of L-round, and it is very simple. The operations performed are

$$\begin{array}{lcl} a_3 & \leftarrow & a_3 \oplus \text{ShiftLeft}_8(a_7) \\ a_2 & \leftarrow & a_2 \oplus \text{ShiftLeft}_8(a_6) \\ a_1 & \leftarrow & a_1 \oplus \text{ShiftRight}_8(a_5) \\ a_0 & \leftarrow & a_0 \oplus \text{ShiftRight}_8(a_4) \end{array}$$

– where ShiftLeft_8 and ShiftRight_8 shift the argument by 8 bit positions to the left and to the right, respectively.

Multiplication by θ

The multiplication by θ is performed in $GF(2^n)$, where n is either 256 or 512, depending on the variant of SMASH. Hence, before we can perform the multiplication we must convert the factors to elements of $GF(2^n)$. This conversion can be done in a natural manner by viewing the bits as coefficients to a polynomial, the most significant bit being the coefficient to the term of the highest degree. The field is defined via an irreducible polynomial of degree n. For SMASH-256 this polynomial is

$$p(\theta) = \theta^{256} \oplus \theta^{16} \oplus \theta^3 \oplus \theta \oplus 1.$$

Hence, θm_i is computed by shifting m_i one bit position to the left (yielding $\tilde{m}_i$), and if the most significant bit of m_i is `1`, the polynomial $q(\theta) = \theta^{16} \oplus \theta^3 \oplus \theta \oplus 1$ is added (since then θ^{255} is shifted to θ^{256}, which equals $q(\theta)$ in this field), meaning that we compute $\tilde{m}_i \oplus \mathtt{1000b}_h$. I.e. $\theta m_i = \tilde{m}_i$ if the most significant bit of m_i is `0`, and $\theta m_i = \tilde{m}_i \oplus \mathtt{1000b}_h$ if the most significant bit of m_i is `1`.

For SMASH-512 the field polynomial is

$$p_{512}(\theta) = \theta^{512} \oplus \theta^8 \oplus \theta^5 \oplus \theta^2 \oplus 1.$$

A similar technique to the one above can be used for multiplication by θ in SMASH-512.

8.2 Analysis

In this section a few properties of SMASH are mentioned. They were all presented in [26] as well.

8.2.1 The forward prediction property

The construction of SMASH originates a *forward prediction* property: If $m_i \oplus m'_i = h_i \oplus h'_i = d$, then $h_{i+1} \oplus h'_{i+1} = d \oplus \theta d = d(1 \oplus \theta)$. Repeated applications of this property, i.e. $m_{i+u} \oplus m'_{i+u} = d(1 \oplus \theta)^u$ for $0 < u < t - i$, yields $h_{i+u+1} \oplus h'_{i+u+1} = d(1 \oplus \theta)^{u+1}$. This weakness, observed already in [26], was exploited in the attack of [33]. Note that $d(1 \oplus \theta) \neq 0$ whenever $d \neq 0$, even for other hash functions based on the same construction, since it is required that $\theta \neq 1$.

The final application of f (8.3) helps protect against attacks based on the forward prediction property. If $h_t \oplus h'_t = \tilde{d}$ then the final application of f causes $h_{t+1} \oplus h'_{t+1} = f(h_t) \oplus f(h'_t) \oplus \tilde{d}$, which seems unpredictable if f is close to a truly random function.

8.2.2 Inverting the compression function

Since f is a bijective mapping, it can be inverted. This means that the entire compression function can be inverted. Let h_{i+1} be fixed. Then choose a arbitrarily, and compute $b = f^{-1}(h_{i+1} \oplus a)$. Since $a = h_i \oplus \theta m_i$ and $b = h_i \oplus m_i$, we find m_i and h_i by solving these two simultaneous equations. They always have a solution since $\theta \neq 1$.

8.2.3 Complexity of (2nd) preimage attacks

Preimage and 2nd preimage attacks of SMASH have much lower complexity than the usual 2^n. First $2^{n/2}$ intermediate hash values h_t are precomputed and stored. Then the same number of pairs (a,b), for which $f(a) \oplus b$ equals some fixed value h_{t+1}, is randomly chosen. From a and b, h_t can be found (as above). With probability about $1/2$ there will be a match on the h_t. This meet-in-the-middle attack has complexity about $2^{n/2+1}$. Knudsen argues that if this level of security is not high enough, then a larger hash function should be used anyway, since for any hash function there is a collision attack based on the birthday paradox of complexity $2^{n/2}$.

8.3 An attack

The collision attack [33] that broke SMASH will now be described.

8.3.1 The idea

As mentioned, the attack makes use of the forward prediction property. The team behind [33] first assumes that $1 \oplus \theta$ has order 3 in $GF(2^n)$, i.e. $(1 \oplus \theta)^3 = 1$. This is not the case for the two proposals of SMASH, but it is permitted in the general construction, since it is only required that $\theta \neq 0$ and $\theta \neq 1$. By choosing carefully the differences between the two messages, this variant of SMASH can be broken

with four-block messages, since the difference on the chaining value h_4 becomes $a(1 \oplus (1 \oplus \theta)^3) = 0$. The attack is then extended to break SMASH.

8.3.2 Breaking a variant

In this section the attack on a variant of SMASH, for which $1 \oplus \theta$ has order 3, is described.

In order to make use of the forward prediction property, there must be a well-defined difference between the two messages, m and m'. Assume without loss of generality that the first difference occurs in the first block, i.e. in m_0. Let

$$d = m_0 \oplus m'_0.$$

Then the difference on the chaining value h_1 is

$$a = h_1 \oplus h'_1 = f(h_0 \oplus m_0) \oplus f(h_0 \oplus m_0 \oplus d) \oplus \theta d. \tag{8.5}$$

If we require $m_1 \oplus m'_1 \doteq a$, then the difference on the chaining value h_2 becomes

$$h_2 \oplus h'_2 = a \oplus \theta a = a(1 \oplus \theta).$$

Extending this to the third message block, we make sure that $m_2 \oplus m'_2 \doteq a(1 \oplus \theta)$, and hence we get

$$h_3 \oplus h'_3 = a(1 \oplus \theta) \oplus \theta(a(1 \oplus \theta)) = a(1 \oplus \theta)^2.$$

Now, for the fourth message block we must deviate from this method, since we want $h_4 = h'_4$. We achieve this if $h_4 \oplus h'_4 = a \oplus a(1 \oplus \theta)^3$. This, in turn, is achieved if the two inputs to the f function are the same as in the first message block, i.e. if $m_3 = h_0 \oplus m_0 \oplus h_3$ and $m'_3 = h_0 \oplus m_0 \oplus d \oplus h'_3$. To see why, first note that this causes the same output difference on f as in (8.5), namely $a \oplus \theta d$. The difference on m_3 is

$$m_3 \oplus m'_3 = d \oplus h_3 \oplus h'_3 = d \oplus a(1 \oplus \theta)^2.$$

Hence, we have

$$\begin{aligned}
h_4 \oplus h'_4 &= a \oplus \theta d \oplus h_3 \oplus h'_3 \oplus \theta(m_3 \oplus m'_3) \\
&= a \oplus \theta d \oplus a(1 \oplus \theta)^2 \oplus \theta(d \oplus a(1 \oplus \theta)^2) \\
&= a \oplus \theta d \oplus (1 \oplus \theta)(a(1 \oplus \theta)^2) \oplus \theta d \\
&= a \oplus a(1 \oplus \theta)^3,
\end{aligned}$$

and since the order of $1 \oplus \theta$ is 3, we have $h_4 = h'_4$, and hence a collision.

8.3.3 Breaking SMASH

For SMASH-256 the order of $1 \oplus \theta$ is (according to [33]) $(2^{256}-1)/5$, and hence using the technique above would require extremely long messages – in fact they would be much longer than the maximum message length of the hash function. Instead, a slightly different technique is used. Note that if we achieve $h_t \oplus h'_t = ap(\theta)$, then we have a collision after the tth message block, since $ap(\theta) = 0 \pmod{p(\theta)}$.

For SMASH-256 we have

$$p(\theta) = \theta^{256} \oplus \theta^{16} \oplus \theta^3 \oplus \theta \oplus 1, \tag{8.6}$$

which can be expressed instead in terms of $1 \oplus \theta$ as

$$p(\theta) = (1\oplus\theta)^{256} \oplus (1\oplus\theta)^{16} \oplus (1\oplus\theta)^3 \oplus (1\oplus\theta)^2 \oplus (1\oplus\theta)^0. \tag{8.7}$$

In the attack above, if we were to add another message block requiring that $m_4 \oplus m'_4 \doteq a \oplus a(1\oplus\theta)^3$, then we would get

$$h_5 \oplus h'_5 = a \oplus a(1\oplus\theta)^3 \oplus \theta(a \oplus a(1\oplus\theta)^3) = a(1\oplus\theta) \oplus a(1\oplus\theta)^4.$$

This shows that we are able to produce any desired polynomial in $1\oplus\theta$ (e.g. specifically a multiple of (8.7)) as difference in h_t: For each block m_i that is processed we can make sure that the difference on h_{i+1} equals the difference on h_i multiplied by $(1\oplus\theta)$ (by placing conditions on the difference $m_i \oplus m'_i$ as just shown for m_4), and optionally we can make sure that a $(= h_1 \oplus h'_1 = a(1\oplus\theta)^0)$ is added to this difference by ensuring (as for m_3 in Section 8.3.2) that the input to f is exactly the same as in the very first step, i.e. when processing m_0.

More specifically, to achieve $h_t \oplus h'_t = ap(\theta)$ we need at least $t = 257$ message blocks, and for this value of t we need $m_0 \oplus m'_0 = d \neq 0$ (which at the end will become the term $(1\oplus\theta)^{256}$ of (8.7)). Then, for $1 \leq i < 240$ $(= 256-16)$ we require $m_i \oplus m'_i \doteq a(1\oplus\theta)^{i-1}$, and in order to produce the term $(1\oplus\theta)^{16}$ of (8.7) we require

$$m_{240} \doteq m_0 \oplus h_0 \oplus h_{240}$$

and

$$m'_{240} \doteq m_0 \oplus d \oplus h_0 \oplus h'_{240}.$$

Similarly to the attack on the order-3 variant of SMASH this can be shown to produce exactly $h_{241} \oplus h'_{241} = a \oplus a(1\oplus\theta)^{240}$.

Now, for the next 12 message blocks, i.e. for $240 < i < 253$, we require $m_i \oplus m'_i \doteq a(1\oplus\theta)^{i-1-240} \oplus a(1\oplus\theta)^{i-1}$. In order to produce the term $(1\oplus\theta)^3$ of (8.7) we require

$$m_{253} \doteq m_0 \oplus h_0 \oplus h_{253}$$

and

$$m'_{253} \doteq m_0 \oplus d \oplus h_0 \oplus h'_{253}.$$

Repeating the arguments we get the following final requirements:

$$
\begin{aligned}
m_{254} &\doteq m_0 \oplus h_0 \oplus h_{254} \\
m'_{254} &\doteq m_0 \oplus h_0 \oplus d \oplus h'_{254} \\
m_{255} \oplus m'_{255} &\doteq a \oplus a(1 \oplus \theta) \oplus a(1 \oplus \theta)^{14} \oplus a(1 \oplus \theta)^{254} \\
m_{256} &\doteq m_0 \oplus h_0 \oplus h_{256} \\
m'_{256} &\doteq m_0 \oplus h_0 \oplus d \oplus h'_{256}
\end{aligned}
$$

This causes $h_{257} \oplus h'_{257} = ap(\theta) = 0 \pmod{p(\theta)}$, and hence $h_{257} = h'_{257}$. I.e. using 257 message blocks ($\approx$ 64KB of data) we have found a collision of SMASH-256. Note that for the first message, 4 message blocks are determined by the attack, and the rest can be chosen freely. For the second message, only one block, m_0, can be chosen freely.

This attack obviously extends to SMASH-512 as well, since in this case the field polynomial can be written

$$p_{512}(\theta) = (1\oplus\theta)^{512} \oplus (1\oplus\theta)^{8} \oplus (1\oplus\theta)^{5} \oplus (1\oplus\theta)^{4} \oplus (1\oplus\theta)^{2} \oplus (1\oplus\theta)^{1} \oplus (1\oplus\theta)^{0}.$$

This attack requires at least 513 blocks, i.e. approximately 256KB of data.

8.4 Possible improvements to SMASH

In the published version of the design document [26], Knudsen suggests a number of possible improvements to SMASH that might prevent attacks based on the forward prediction property from being possible. We now look at these, and also try to identify other possible improvements. When it makes any difference, the discussion refers to SMASH-256, but all proposals should be quite easily extendible to SMASH-512.

First we have to consider which properties of SMASH were exploited in the attack. First of all, the forward prediction property proved to be a weakness. This was in part due to the mathematical property of multiplication by θ. Secondly, and related to the forward prediction property, an attacker is able to choose his differences on the message blocks (and in fact the entire input to the core function) completely independently, since a message block is only used in one application of the compression function. This might not be a weakness in itself, but it should be noted since it gives an adversary greater freedom in mounting attacks.

Before we begin, a useful concept is defined.

Definition 8.1. Let $f : V \to V$ be a bijective mapping. A *difference fixed-point* for f is a pair (x, d), $d \neq 0$, such that $f(x \oplus d) = f(x) \oplus d$.

As it turns out, we would like to be able to construct bijections that have no difference fixed-points, or for which finding difference fixed points is infeasible.

8.4.1 Using the secure compression function for every j steps

One proposal from [26] for improving the security of the hash function is to apply the secure compression function f as in (8.3) for every j steps, with j equal to e.g. 8 or 16.

This change would definitely thwart the specific attack of Section 8.3 on SMASH-256 whenever j is at most 256, since for any such j, the application of f with feed-forward would break the required difference on the following h_i, and subsequent differences (let alone chaining values) would seemingly be impossible to predict.

One may argue that this solution is not very elegant. However, it does not introduce much complexity, and the speed of the hash function is not reduced significantly for values of j like 16 or above. This is a simple fix, but it is unclear exactly how much the security of the hash function is improved. For instance, there might be other collision attacks that need less than j message blocks.

8.4.2 Using different f functions in each step

The idea to use different f functions in each step was actually first suggested in [33]. This might complicate the attack of Section 8.3. We shall look at a few different variants.

Alternating between two f functions

Assume that two versions of the core function, f_0 and f_1 exist, and replace (8.2) with

$$h_{i+1} \leftarrow f_{i \bmod 2}(h_i \oplus m_i) \oplus h_i \oplus \theta m_i \quad \text{for } 0 \leq i < t.$$

This solution does not rid SMASH of the forward prediction property, but the attack above would fail in the processing of m_{253}, since in this step we would use f_1, whereas we used f_0 when processing m_0, and hence we cannot predict the output of the core function as is needed in the attack.
However, this is only true if it is difficult to find values x and x' such that $f_0(x) \oplus f_0(x') = f_1(x) \oplus f_1(x')$. If one finds just a single such pair, the attack will work with $m_0 = h_0 \oplus x$ and $m_0' = h_0 \oplus x'$ and, of course, $d = x \oplus x'$ (thus leaving us with less freedom of choice). Whether such pairs exist and can be found obviously depends on the choices of f_0 and f_1.
In any case an attacker might find a multiple of $p(\theta)$ expressed in terms of $1 \oplus \theta$ where no odd exponents exist, and then the same technique as above could be used to find a collision.

Adding some dependency on the step number

Another possibility is to let the argument to f depend on a third value, e.g. some function of the step number. The most simple way to do this would be to replace

(8.2) with

$$h_{i+1} \leftarrow f(h_i \oplus m_i \oplus g(i)) \oplus h_i \oplus \theta m_i \quad \text{for } 0 \le i < t,$$

where g is some simple function.
However, this solution does not provide any resistance against the attack described in the previous section. Independently of the choice of g one can produce a collision in the same way, except that some of the message blocks get different values because $g(i)$ must be added. This is because $g(i)$ does not have any influence on the difference between the two messages. Thus, using $g(i)$ in other ways, for instance added to m_i before multiplication by θ, does not improve the security either.

Letting f accept a second argument

As a final proposal of this type, one could change the f function by making it dependent on a second argument, for instance i. This argument could affect the S-boxes used, or the rotation values, or both. For instance, if there were 8 different H-rounds, and we still only wanted to use 3 different H-rounds in each call to the f function, there would be $\binom{8}{3} = 56$ possible combinations, and hence we might use as a second argument to f the value i mod 56, and define some simple relation between this value and the combination of S-boxes and rotation vectors to use.
There are even more combinations if we accept any triple of H-rounds (see (8.4)) to be used “in between” L-rounds (as long as they don’t repeat). To be specific, there are $336 \times 335 \times 334 \times 333 \approx 2^{33}$ such combinations in SMASH-256, so one might simply supply the last 32 bits of i as an extra argument to f in this case.
Of course, any of these proposals adds to the complexity of the hash function. Furthermore, it should be noted that there may not be 8 equally good S-boxes and/or rotation vectors, which might also be a concern. Finally, this solution does not rid the hash function of the forward prediction property, although it does make it resistant against the direct use of the attack previously described.

8.4.3 Further dependency on the message

In an attempt to get rid of the forward prediction property and in addition reduce the freedom that an adversary has in choosing message blocks to fit his needs, a new variant is proposed. The idea is to replace (8.2) with

$$h_{i+1} \leftarrow f(h_i \oplus m_i \oplus g(m_{i-1})) \oplus h_i \oplus \theta m_i \quad \text{for } 0 \le i < t, \tag{8.8}$$

where g should be a bijective mapping and m_{-1} could be the all-zero vector.

By making the compression function depend on two different message blocks, an adversary has much less freedom in the choice of message blocks.
In the attack of Section 8.3, the difference between the two messages is chosen such that in many steps there is no difference on the input to the core function. The reason for placing $g(m_{i-1})$ inside the core function in this new proposal is to ensure that message block differences must be more complicated for the input to the core

function to be identical for the two messages. In this respect, g (when not equal to the identity function) also seems to increase the complexity of finding collisions. The forward prediction property is essentially eliminated, and even more so if g does not contain difference fixed-points, or if such are difficult to find.

In addition, the bijective mapping g should not contain any mathematical properties that could be exploited. g could be a rotation, or it could be a function that adds three differently rotated versions of the input (cf. [17, the functions (4.4) and (4.5)]). For 256-bit values, bijective versions exist. Whether they contain (easily found) difference fixed-points is another matter.

It may also be a good idea to choose g such that it does not contain fixed-points. This could be achieved by adding some constant to one of the SHA-256 functions mentioned. For the sake of symmetry it might also be a good idea to append a zero-block to the end of the message, such that the padding block m_{t-1} is used twice, just as all other message blocks are. I.e. the final application of (8.8) becomes

$$h_{t+1} \leftarrow f(h_t \oplus g(m_{t-1})) \oplus h_t.$$

Let us analyse this proposal independently of the choice of g. It can be inverted as follows. Given h_{i+1} choose a at random and compute $b = f^{-1}(h_{i+1} \oplus a)$. Then solve the system of equations

$$\begin{pmatrix} a & b \end{pmatrix} = \begin{pmatrix} h_i & m_i & g(m_{i-1}) \end{pmatrix} \begin{pmatrix} 1 & 1 \\ \theta & 1 \\ 0 & 1 \end{pmatrix}. \tag{8.9}$$

This system has an infinite number of solutions. In fact, one can even choose h_i and m_i independently, and then base one's choice of a on these values. Then b and m_{i-1} are fixed (m_{i-1} is easily computed from $g(m_{i-1})$). However, one cannot choose both h_i and m_{i-1} and compute the m_i that links these choices to h_{i+1}. Of course, one can start off by fixing m_{i-1} such that $g(m_{i-1}) = c$, and then solve the resulting system of equations

$$\begin{pmatrix} a & b \oplus c \end{pmatrix} = \begin{pmatrix} h_i & m_i \end{pmatrix} \begin{pmatrix} 1 & 1 \\ \theta & 1 \end{pmatrix}.$$

This means that as for SMASH, (2nd) preimages can be found in time $2^{n/2}$ by a meet-in-the-middle attack as the one described in Section 8.2.3, except that one message block is fixed in the computation of each pool of h_t-values. It also means that collisions for the compression function can be easily found. However, it seems difficult to find useful collisions, since given h_i and m_{i-1}, there is no easy way to solve (8.9) for m_i.

Some people would reject this proposal with sole reference to the invertibility of the compression function. The straight-forward invertibility can be removed by using as compression function

$$h_{i+1} \leftarrow f(h_i \oplus m_i \oplus g(m_{i-1})) \oplus h_i \oplus m_{i-1} \oplus g(m_i).$$

Now an adversary is faced with the task of finding (m_i, m_{i-1}) such that $m_i \oplus g(m_i) \oplus m_{i-1} \oplus g(m_{i-1}) = f^{-1}(a \oplus h_{i+1}) \oplus a$ for a given h_{i+1} and a chosen a. He can choose one of m_i or m_{i-1}, for instance m_i, and then look for an m_{i-1} such that

$$m_{i-1} \oplus g(m_{i-1}) = f^{-1}(a \oplus h_{i+1}) \oplus a \oplus m_i \oplus g(m_i).$$

The difficulty of this task depends on g, and it seems that g might have to be as strong as f in order to prevent the compression function from being invertible. This would render the hash function two to three times slower than SMASH.

In the following section we consider proposals that use two strong, bijective mappings, but without the dependency on more than one message block.

8.4.4 Using more than one fixed, bijective mapping

The last proposal made in [26] to improve the security of SMASH is to use more than one fixed, bijective mapping in the compression function. This solution will be given more attention than the previous ones.

Of course, there are many possibilities for using more than one fixed, bijective mapping. We shall look at two proposals. The first one replaces multiplication by θ with some bijective mapping g:

Proposal 8.1. Let $m_0 \| \cdots \| m_{t-1}$ be the padded message, and let v be the all-zero word. Compute

$$\begin{aligned} h_0 &\leftarrow f(v) \oplus v \\ h_{i+1} &\leftarrow f(h_i \oplus m_i) \oplus g(m_i) \oplus h_i \quad \text{for } 0 \le i < t \\ h_{t+1} &\leftarrow f(h_t) \oplus h_t \end{aligned}$$

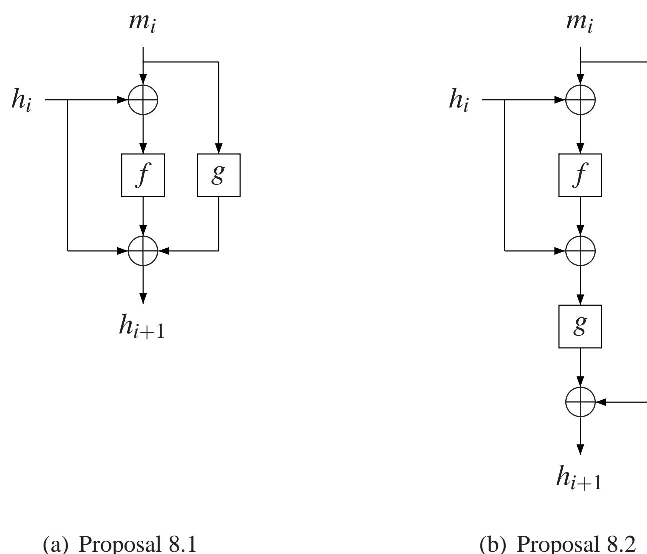
where g is some bijective mapping, for instance a variant of f. The hash value is defined as h_{t+1}. See also Figure 8.1(a).

The evaluation of f and g can be done in parallel. The second proposal does not share this property, but may have other advantages, which we shall look at.

Proposal 8.2. Let $m_0 \| \cdots \| m_{t-1}$ be the padded message, and let v be the all-zero word. Compute

$$\begin{aligned} h_0 &\leftarrow f(v) \oplus v \\ h_{i+1} &\leftarrow g(f(h_i \oplus m_i) \oplus h_i) \oplus m_i \quad \text{for } 0 \le i < t \\ h_{t+1} &\leftarrow f(h_t) \oplus h_t \end{aligned}$$

where g is some bijective mapping, for instance a variant of f. The hash value is defined as h_{t+1}. See also Figure 8.1(b).



(a) Proposal 8.1 (b) Proposal 8.2

Figure 8.1: The compression function of the two proposals for improving SMASH.

Both proposals reduce the speed of the hash function compared to SMASH, at least whenever g takes about as much time as f. One may consider as g a faster variant of f such as g_1 (compare with (8.4)):

$$g_1(\alpha) = H_2 \circ H_3 \circ H_1 \circ L \circ H_3 \circ H_1 \circ H_2(\alpha^{\lll 128})$$

It is not recommended to choose $g = f$, since this might give an attacker additional options.

Choosing $g = g_1$ would add about 50% to the running time of the algorithm. The rotation of the input is added because there is a slight difference in significance of high-order and low-order bytes in (either version of) f, and this rotation might make attacks based on the internal structures of f and g less likely to succeed. Also note that the order in which the different H-rounds are used differ from f.

Note that g_1 is just one possibility of how g could be chosen. In the following we shall try to identify some requirements on the g function.

Finding collisions

One way to go about findings collisions of variants of SMASH with two bijective functions is to ensure that there is no difference in the input to one of the functions for the two messages that are to form a collision. This reduces the problem to that of finding a collision for the remaining part.

Analysis of Proposal 8.1. If the input to f is constant, then $h_i \oplus h'_i = m_i \oplus m'_i = d \neq 0$, and hence g must have a difference fixed-point (m_i, d) in order to achieve a

collision. Similarly, if the input to g is kept constant, then $m_i = m'_i$ and $h_i \oplus h'_i = d \neq 0$, and hence f must have a difference fixed-point $(h_i \oplus m_i, d)$. This means that if it is infeasible to find difference fixed-points for both f and g, then this technique cannot be applied.

Analysis of Proposal 8.2. If the input to f is kept constant, then $h_i \oplus h'_i = m_i \oplus m'_i = d \neq 0$ and hence g must have the difference fixed-point $(f(h_i \oplus m_i) \oplus h_i, d)$. Similarly, if the input to g is kept constant, then $m_i = m'_i$ and $h_i \oplus h'_i = d \neq 0$, and hence f must have the difference fixed-point $(h_i \oplus m_i, d)$. Hence, we can draw the same conclusion as for Proposal 8.1.

The short analyses above show that the two proposals seem equally vulnerable to this sort of attack, and if f and g can be found such that difference fixed-points are infeasible to find, then both proposals are secure to this kind of attack. Also note that the analyses prove that the forward prediction property no longer holds, or at least it would require that the attacker find some property of f or g that can be exploited.

It is easy to create a function that does not have any difference fixed-points. Let p be a bijective mapping, and define $q(x) = p(x) \oplus x$. Then q has no difference fixed-points, since $q(x \oplus d) = p(x \oplus d) \oplus x \oplus d$, and $p(x \oplus d) \neq p(x)$ for $d \neq 0$ because p is bijective. However, these functions are not necessarily bijective. Whether they can be safely used in place of f and g in the two proposals is an open question.

In Proposal 8.1 it seems that the only alternative kinds of collisions of the compression function have a difference on all three "terms", i.e. $\Delta f \neq 0$, $\Delta g \neq 0$ and $\Delta h_i \neq 0$. For Proposal 8.2 we may have $\Delta g = \Delta m_i \neq 0$, which can be achieved in three ways. One of them is mentioned above ($\Delta h_i = \Delta m_i$), another has $\Delta h_i = 0$, and a third has $0 \neq \Delta h_i \neq \Delta m_i$. For all possibilities it is necessary to go into the details of the g and/or f functions in order to find a solution.
Of course, we have not discussed the possibility of finding collisions of the final application of f. This also requires finding a difference fixed point of f, but even if this is possible it may not be a problem, since an attacker has little control over the input h_t.

Inverting the compression function

In the original SMASH, the compression function can be inverted because the system of equations

$$\begin{pmatrix} \theta & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} m_i \\ h_i \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

can be solved no matter the choice of a and b, since $\theta \neq 1$. Let us investigate if Proposals 8.1 and 8.2 are invertible.

Inverting Proposal 8.1. Using the same method as in Section 8.2.2, one obtains the system of equations

$$\begin{aligned} h_i \oplus g(m_i) &= a \\ h_i \oplus m_i &= b \end{aligned}$$

If g is non-linear (which is recommended), this is not a system of linear equations, and hence to solve it one must apparently find m_i such that $g(m_i) \oplus m_i = a \oplus b$. There does not seem to be any other way, unless one is able to exploit some weakness of g. This indicates that we may want to choose g such that it is as strong as f, and hence g_1 might not be a good choice.

Inverting Proposal 8.2. To invert Proposal 8.2 one must choose m_i, compute $a = g^{-1}(h_{i+1} \oplus m_i)$, and then find h_i such that $f(h_i \oplus m_i) \oplus h_i = a$. If the g used in Proposal 8.1 is as strong as f, inverting Proposal 8.2 looks about as difficult as inverting Proposal 8.1.

If f and g were both truly random bijective mappings, then inverting the compression function would take time 2^n for both proposals. Hence, the (2nd) preimage attack of Section 8.2.3 would have about the same complexity as a brute-force attack.

Summary

We have not found any significant differences with regards to security between Proposal 8.1 and Proposal 8.2. They both have some advantages that the original proposal of SMASH does not have, including the absence of the forward prediction property and the difficulty in inverting the compression function. Hence, it seems that an attack would require some analysis of the bijective mappings, possibly including a search for difference fixed-points.
The disadvantage of the proposals is that they are both much slower, about half the speed of SMASH itself. In this respect, as mentioned, Proposal 8.1 has the advantage that the applications of f and g can be parallelised, which would make the hash function as fast as SMASH.

Chapter 9

Future directions

At this point it should be quite clear that discussions about the future direction of hash functions are needed. Do we trust hash functions based on insecure hash functions? Should we gain security by reducing efficiency? Should we abandon the Merkle-Damgård construction, or modify it? These are merely a few of the topics that should be discussed.

9.1 A brief discussion on strategy

When a car breaks down the owner has the choice of having it repaired, or buying a new car. The choice depends on the severity of the defects of the car. If it is too expensive or risky to have it repaired, the owner might choose to buy a new one instead. On the other hand, if the repairs are simple, and the car can be expected to be as good as new afterwards, there is probably no need to replace it. We might use the same approach when deciding on a new hash function. For instance, SHA-256 can be seen as a repaired version of SHA-1, whereas WHIRLPOOL is a "new car". The important difference between hash functions and cars is that there is no warranty on a new hash function, and therefore it might be even more risky to choose a new hash function than to continue with a repaired version of a broken hash function. At the moment there is a general belief in the cryptographic community that we still do not know enough about hash functions to make proper decisions. However, let us try to identify some properties of the two strategies mentioned.

First we look at some advantages of using the *repair* strategy:

- We know (some of) the weaknesses of the underlying hash function, and hence we can try to eliminate these
- Extensive analysis has often been performed on the broken hash function, and hence we may feel confident that some kinds of attacks are inapplicable
- Existing software implementations can often simply be slightly modified to yield the repaired version. Even hardware implementations may be much easier and cheaper to update.

Among the disadvantages are the following:

- The repaired version may suffer from the same fundamental weaknesses as the broken hash function
- An intended fix might introduce new weaknesses
- Repeated repairs might yield less efficient hash functions.

Of course, the advantages and disadvantages of using the *replace* strategy are to some extent the complements of the above, but we might add that another advantage of the *replace* strategy is that we get the opportunity to rethink the entire construction, and hence we might come up with an alternative to the Merkle-Damgård construction that does not contain the weaknesses described in Chapter 6.

It seems that the risks of the two strategies are comparable. We might not know much more about the security of a repaired version than we know about the security of a new hash function – most often we would use well-known components for a new hash function anyway. In a sense this brings the discussion to the topic of the following section.

The strategy of constructing a new hash function based on an existing block cipher such as AES does not fall directly into either of the two categories mentioned. It can be thought of as a combination, and it surely does have some advantages of both the mentioned strategies. However, as should be clear from Chapter 5, it is not a simple matter to produce a "more-than-single-length" block cipher based hash function, and such solutions often become inconvenient.

9.2 Merkle-Damgård or not Merkle-Damgård?

Two attacks on the general Merkle-Damgård construction were described in Chapter 6. The multicollision attack of Joux proved that it is not much harder to find a large number of messages with the same hash, than it is to find two such messages when the Merkle-Damgård construction is used. Hence, if it is infeasible to find collisions of a hash function based on the Merkle-Damgård construction, then the results of Joux are of no practical relevance.

The 2nd preimage attack of Kelsey and Schneier proves that it is easier to find a 2nd preimage in a hash function based on the Merkle-Damgård construction than by exhaustive search, but the attack only works for very long messages. What's more, the attack still requires the ability to find collisions. Thus, again, if it is infeasible to find collisions of a hash function based on the Merkle-Damgård construction then the attack has no practical relevance, and even when collisions *can* be found, in practice a 2nd preimage cannot be found for most messages with this method.

The strength of these results lie in the fact that they are directed towards a general construction, and not towards a specific hash function. The attacks have significantly reduced the amount of faith in the Merkle-Damgård construction. It is feared that the techniques could be further improved. Some people argue that (2nd)

preimage resistance is much more important than collision resistance, and hence it is a serious weakness that 2nd preimages in some cases can be found in time not much longer than collisions. Others would say that in the future we just have to make sure that it is truly infeasible to find collisions of a hash function, then no existing attacks on the Merkle-Damgård construction are of any use.

Some "quick fixes" of the construction have been proposed. For instance, as suggested by Rivest [39], we might let a (small) fixed number of input bits to the compression function be determined by some function of the iteration number (in particular Rivest suggests a so-called Abelian square-free sequence), such that the size of the input to the compression function does not change, but the simple repeated use of a fixed point is no longer possible.
Another similar proposal by Biham [1] uses the number of message bits processed thus far as an extra input variable to the compression function.

The latest attack on the Merkle-Damgård construction [24] is less than a year old, and hence we should probably expect improvements to this attack in the near future. The attacks may also have motivated many cryptographers to direct their attention towards finding other weaknesses in the construction. Hence, completely ignoring existing attacks is probably not a good idea.

9.3 A discussion on efficiency

The first properties required for a cryptographic hash function (as mentioned in [11]) were one-wayness and efficiency. Surely a hash function must be efficiently computable, but what is "efficient"?

A hash function is used in many different environments, some of which have very limited memory and/or processing capabilities. It is the goal of most hash functions that it be usable in all such environments. Existing hash functions seem to meet this goal, and hence we may compare the efficiency of any new proposal with the efficiency of for instance SHA-256. However, according to Moore's law we should expect a hash function with twice the running time of SHA-256 to be as usable in 18 months as SHA-256 is today[1]. Using twice the amount of work we can double the hash size with the same "per-bit" security, or we could aim for greater per-bit security. Note that adding just a single bit to the hash size doubles the complexity of brute force (2nd) preimage attacks, and adding two bits to the hash size doubles the complexity of a birthday (collision) attack. However, a 256-bit hash function will most probably be resistant to all brute force attacks for many years to come (using again Moore's law we should expect a birthday attack to be infeasible for at least 80 years), and therefore, assuming that we are allowed to construct a hash function that has only half the speed of SHA-256, it is probably better to aim for a greater per-bit security. As suggested in [1] it might be a good idea to focus more on ensuring a large degree of dependency on each input bit

[1] This does not take into account hardware implementations which often do not benefit from technological advances for many years.

than has traditionally been the case when constructing hash functions. The AES competition showed that in the area of block ciphers such focus is present, and there is no reason why this should not extend to the area of hash functions.

The industry requests efficient hash functions. It also requests hash functions with a long lifespan, hash functions that are easily implemented in both software and hardware, and it does not want to rely on agility, such as for instance scalability. It may be difficult, if not impossible, to satisfy all these needs at once. Hence, it makes sense to focus more on security and durability than on efficiency. This can be seen as more of a long-term investment: while a less efficient hash function may be somewhat problematic at the moment, if it is durable enough it will "earn back its price" many times in the long run.

9.4 The NIST Cryptographic Hash Workshop

On October 31 and November 1, 2005, NIST hosted the Cryptographic Hash Workshop with the aim of assessing the status of SHA-256 and other NIST-approved hash functions, and of discussing both short and long term actions to the recent attacks on hash functions of the MD4 family.

Although there were many different opinions, it was, of course, suggested that the industry and others using hash functions stop using MD5 and SHA-1 for any new product. Although most people seemed to believe that SHA-256 will be broken, at least theoretically, within the next 10 years, and since there does not seem to be any alternative equally efficient but more secure hash function, most people suggested switching to SHA-256. This sparked a great deal of concern from some representatives of the industry, since building completely new hardware is a much more costly affair than, for instance, extending SHA-1 to 160 rounds.

In general, the urgency of the matter seems to be by far the greatest problem: it is crucial that MD5 and SHA-1 are phased out as quickly as possible, but since these changes are difficult and expensive to make, it is important that the substitute can be expected to have a long lifespan, at least 10 years and preferably more than that. In the light of these issues, it seems odd that NIST chose to approve a hash function such as SHA-256, which is based on SHA-1, but has fewer rounds. Hence, each round of SHA-256 must produce twice as many bits of hash output as each round of SHA-1. Sure, the SHA-256 round contains a greater amount of processing than the SHA-1 round (and we have not mentioned the message expansion), but it does not look like the ultimate replacement.

No new attacks on hash functions were presented at the workshop. Xaioyun Wang presented her 2^{63} attack [44] on SHA-1, but gave no additional details, and she had no paper in the workshop program. Some new hash function proposals were presented; among these a fairly quick proposal [7] for which finding collisions is as difficult as factoring a large composite integer. Here, "fairly quick" means about 26 times slower than SHA-1.

There were suggestions to initiate a hash function competition similar to the

AES competition, but at the same time most people agreed that the community is not ready for such a competition at the moment. The AES competition process was very educational, but when the competition started, the state of research of block ciphers was much further ahead than the state of research of hash functions is today. Hence, it was suggested that first a number of additional hash function workshops be organised.

It was discussed whether or not it would be a good idea to approve hash functions solely for specific purposes, but most people thought this would not be the way to go, as hash functions are and always will be used for a huge number of different purposes, most of which could never have been predicted. Hash functions were referred to by an attendee of the workshop as the "work horse of cryptography". NIST suggested that a list of all currently known applications of hash functions be made, so as to better identify the security criteria for new hash functions. The process of producing such a list could be very instructive indeed, but the list itself may not be very useful for the reason just mentioned.

NIST itself did not reveal much about its intentions, except that it seemed like NIST is also very much in a state of uncertainty. Another hash workshop organised by NIST is supposed to take place at the end of the Crypto 2006 conference, and hopefully by then NIST and the community is in a better position to make proper decisions.

9.5 Summary

To sum up, it seems that we can currently make the following statements about the status of hash functions and requirements for the future.

- MD5 and SHA-1 should no longer be used, at least in environments where collisions are a threat. Even elsewhere, the use of these hash functions should be phased out.

- Instead of MD5 and SHA-1, it is advised that a switch to SHA-256 is made, but it is stressed that this may not be a long-lasting solution.

- Further research on hash functions is desperately needed. It is important that researchers understand the weaknesses of broken hash functions very well, and that they identify better design criteria.

- New hash function proposals are needed. Preferably these should differ significantly from the hash functions of the MD4 family, and their efficiency should be comparable with that of SHA-256. The publication should be preceded with a significant amount of analysis.

- The community should probably try to come to some sort of agreement regarding whether or not to worry about the attacks on the Merkle-Damgård

construction. This more or less reduces to a decision on whether it is sufficient that a hash function is collision resistant, or whether it should always be much harder to find (2nd) preimages and multicollisions than to find ordinary collisions.

- Alternatives and/or fixes to the Merkle-Damgård construction should be developed.

As we already knew, and as this list shows, a lot of work is needed in the area of cryptographic hash functions before we can expect to regain confidence in our claims regarding the security of hash functions. It should be very interesting to follow the progress in the next few years.

Appendix A

Conditions on step variables in the Wang MD5 attack

In this appendix the conditions on step variables in the Wang MD5 attack are listed. The symbol _ means there is no condition on the bit. The symbol ▲ means the bit must have the same value as the bit directly above it in the table. All bits with the same letter symbol must have the same value. Sometimes there are further conditions on the relations between letter symbols; in these cases they are stated in the table caption.

A.1 First iteration

The conditions in the first iteration are taken from [21], except that we ignore what is referred to as *Case 2*.

	Conditions on Q_t																														
t	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
2	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
3	_	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_	_	_	_	_	0	_	_	_	_	0	_	_	_	_	_	_
4	C	_	_	_	_	_	_	_	0	▲	▲	▲	1	▲	▲	▲	▲	▲	▲	▲	1	▲	▲	▲	▲	0	_	_	_	_	_	_
5	C	_	_	_	1	_	0	_	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	_	_	1	_	1
6	B	▲	▲	▲	0	▲	1	▲	0	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	▲	▲	0	▲	1
7	A	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	0	1	1	_	_	1	0	0	0	1	0	_	0	_	0	1	0	1	0	1	0	0	0	0	0	0
9	E	1	1	1	1	0	1	1	_	_	_	1	0	0	0	0	0	_	1	▲	1	1	1	1	0	0	1	1	1	1	0	1
10	A	1	_	_	_	_	_	_	0	_	_	1	1	1	1	1	1	_	0	1	_	_	_	0	0	1	_	_	_	_	0	0
11	A	0	_	_	_	_	_	_	_	_	_	_	0	0	0	1	1	▲	0	0	_	_	_	0	1	1	_	_	_	_	1	0
12	A	0	_	_	_	_	▲	▲	_	_	_	_	1	0	0	0	0	0	0	1	_	_	_	1	0	_	_	_	_	_	_	_
13	A	1	_	_	_	_	0	1	_	_	_	_	1	1	1	1	1	1	1	_	_	_	_	0	0	_	_	_	1	_	_	_
14	A	_	0	_	_	_	0	0	_	_	_	_	1	0	1	1	1	1	1	_	_	_	_	1	1	_	_	_	1	_	_	_
15	H	_	1	_	_	_	0	1	_	_	_	_	_	_	_	_	1	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_
16	H	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_

Table A.1: Conditions in the first iteration, steps 1–16. If $\mathtt{A} = \mathtt{B}$ then $\mathtt{C} \doteq \mathtt{1}$, otherwise $\mathtt{C} \doteq \mathtt{0}$. Another condition is $\mathtt{E} \doteq \neg\mathtt{A}$

	Conditions on Q_t																															
t	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15	H	_	1	_	_	_	0	1	_	_	_	_	_	_	_	_	1	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_
16	H	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
17	H	_	_	_	_	_	_	_	_	_	_	_	_	_	0	_	▲	_	_	_	_	_	_	_	_	_	_	_	▲	_	_	_
18	H	_	▲	_	_	_	_	_	_	_	_	_	_	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
19	H	_	_	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
20	H	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
21	H	_	_	_	_	_	_	_	_	_	_	_	_	_	▲	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
22	H	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
23	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
24	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
25-45	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
46	I	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
47	J	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
48	I	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
49	J	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
50	K	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
51	J	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
52	K	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
53	J	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
54	K	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
55	J	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
56	K	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
57	J	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
58	K	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
59	J	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
60	I	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
61	J	_	_	_	_	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
62	I	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
63	J	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
64	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
−3	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
−2	L	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
−1	L	_	_	_	_	0	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
0	L	_	_	_	_	0	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_	_	_

Table A.2: Conditions in the first iteration, steps 15–64, and conditions on the chaining values that serve as initial values for the second iteration (Q_{-3}–Q_0). It is a condition that $\texttt{I} \doteq \neg\texttt{K}$.

A.2 Second iteration

The conditions in the second iteration are derived in Section 4.5.2.

t	Conditions on Q_t																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
−3	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
−2	A	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
−1	A	_	_	_	_	0	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
0	A	_	_	_	_	0	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_	_	_
1	B	_	_	_	0	1	0	_	_	_	1	_	_	_	_	_	_	_	_	_	0	_	_	_	_	_	0	_	_	_	_	_
2	B	▲	▲	▲	1	1	0	_	_	_	0	▲	▲	▲	▲	▲	_	_	_	▲	1	_	_	_	▲	▲	0	_	_	0	0	_
3	B	0	1	1	1	1	1	_	_	_	0	1	1	1	1	1	_	_	_	0	1	_	_	1	0	1	1	▲	▲	1	1	_
4	B	0	1	1	1	0	1	_	_	_	0	0	0	1	0	0	_	_	_	0	0	▲	▲	0	0	0	0	1	0	0	0	▲
5	A	1	0	0	1	0	_	_	_	_	1	0	1	1	1	1	_	_	_	0	1	1	1	0	0	1	0	1	0	0	0	0
6	A	_	_	0	0	1	0	_	_	_	1	0	_	_	1	0	_	_	_	0	1	1	0	0	0	1	0	1	0	1	1	0
7	B	_	_	1	0	1	1	▲	▲	_	0	0	_	_	0	1	▲	_	_	1	1	1	1	0	0	0	_	_	_	_	_	1
8	B	_	_	0	0	1	0	0	0	_	1	1	_	_	1	0	1	_	_	_	_	_	1	1	1	1	_	_	_	_	▲	0
9	B	_	_	1	1	1	0	0	0	_	_	_	_	_	0	1	0	_	_	▲	_	_	0	1	1	1	_	_	_	_	0	1
10	B	_	_	_	_	1	1	1	1	_	_	_	_	0	1	1	1	_	_	0	_	_	1	1	1	1	_	_	_	_	0	0
11	B	_	_	_	_	_	_	_	_	_	_	_	▲	1	0	1	1	▲	▲	0	_	_	1	1	1	1	_	_	_	_	1	1
12	B	▲	▲	▲	▲	▲	▲	▲	_	_	_	_	1	0	0	0	0	0	0	1	_	_	_	_	1	_	_	_	_	_	_	_
13	A	0	1	1	1	1	1	1	_	_	_	_	1	1	1	1	1	1	1	_	_	_	_	_	0	_	_	_	1	_	_	_
14	A	1	0	0	0	0	0	0	_	_	_	_	1	0	1	1	1	1	1	_	_	_	_	_	1	_	_	_	1	_	_	_
15	C	1	1	1	1	1	0	1	_	_	_	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_
16	C	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_

Table A.3: Conditions in the second iteration, chaining values and steps 1–16. It is a condition that $\mathtt{B} \doteq \neg\mathtt{A}$.

t	Conditions on Q_t																														
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
15	C	1	1	1	1	1	0	1	_	_	_	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_
16	C	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
17	C	_	_	_	_	_	_	_	_	_	_	_	_	_	0	_	▲	_	_	_	_	_	_	_	_	_	_	_	▲	_	_	_
18	C	_	▲	_	_	_	_	_	_	_	_	_	_	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
19	C	_	_	_	_	_	_	_	_	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
20	C	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
21	C	_	_	_	_	_	_	_	_	_	_	_	_	_	▲	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
22	C	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
23	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
24	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
25-45	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
46	D	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
47	E	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
48	D	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
49	E	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
50	F	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
51	E	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
52	F	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
53	E	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
54	F	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
55	E	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
56	F	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
57	E	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
58	F	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
59	E	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
60	D	_	_	_	_	_	0	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
61	E	_	_	_	_	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
62	D	_	_	_	_	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
63	E	_	_	_	_	_	1	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_
64	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_	_

Table A.4: Conditions in the second iteration, steps 15–64. It is a condition that $\mathtt{F} \doteq \neg\mathtt{D}$. The actual requirement on $Q_{63}[25]$ is somewhat complicated, see Section 4.7.

Appendix B

An implementation of the Dobbertin MD4 attack

The code below is a C program that finds collisions of MD4.

```
#include <stdio.h>
#include <time.h>

typedef unsigned long word;
typedef unsigned char byte;

word m[16], Q[37], Qp[37];
word Q15, Q16, Q17, Q18, Q19, Q20;
word a, b, c, d, A, B, C, D;

#define F(x,y,z) (((x)&(y)) | ((~(x))&(z)))
#define G(x,y,z) (((x)&(y)) | ((x)&(z)) | ((y)&(z)))
#define H(x,y,z) ((x)^(y)^(z))
#define rot(x, s) (((x)<<(s))^((x)>>(32-(s))))

#define random_bit() (1<<(rand()&0x1f))

#define random_word() rand()
#define IVA 0x67452301
#define IVB 0xefcdab89
#define IVC 0x98badcfe
#define IVD 0x10325476
#define K1 0x5a827999
#define K2 0x6ed9eba1
#define N 1 // Number of collisions to find

double current_time_micros() {
```

```
  /* returns current time in microseconds */
  struct timeval curtime;
  gettimeofday(&curtime, 0);
  return ((double) curtime.tv_sec)* 1000000 +
    ((double) curtime.tv_usec);
}

#define collision() \
  Q[21] = rot(Q[17]+G(Q[20],Q[19],Q[18])+m[1]+K1, 3);\
  Q[22] = rot(Q[18]+G(Q[21],Q[20],Q[19])+m[5]+K1, 5);\
  Qp[22] = rot(Q[18]+G(Q[21],Qp[20],Qp[19])+m[5]+K1, 5);\
  if (Qp[22]-Q[22]) continue;\
  Q[23] = rot(Q[19]+G(Q[22],Q[21],Q[20])+m[9]+K1, 9);\
  Qp[23] = rot(Qp[19]+G(Qp[22],Q[21],Qp[20])+m[9]+K1, 9);\
  if (Qp[23]-Q[23] != 0x4000) continue;\
  Q[24] = rot(Q[20]+G(Q[23],Q[22],Q[21])+m[13]+K1, 13);\
  Qp[24] = rot(Qp[20]+G(Qp[23],Qp[22],Q[21])+m[13]+K1, 13);\
  if (Qp[24]-Q[24] != 0xffffffc0) continue;\
\
  Q[25] = rot(Q[21]+G(Q[24],Q[23],Q[22])+m[2]+K1, 3);\
  Qp[25] = rot(Q[21]+G(Qp[24],Qp[23],Qp[22])+m[2]+K1, 3);\
  if (Qp[25]-Q[25]) continue;\
  Q[26] = rot(Q[22]+G(Q[25],Q[24],Q[23])+m[6]+K1, 5);\
  Qp[26] = rot(Qp[22]+G(Qp[25],Qp[24],Qp[23])+m[6]+K1, 5);\
  if (Qp[26]-Q[26]) continue;\
  Q[27] = rot(Q[23]+G(Q[26],Q[25],Q[24])+m[10]+K1, 9);\
  Qp[27] = rot(Qp[23]+G(Qp[26],Qp[25],Qp[24])+m[10]+K1, 9);\
  Q[28] = rot(Q[24]+G(Q[27],Q[26],Q[25])+m[14]+K1, 13);\
  Qp[28] = rot(Qp[24]+G(Qp[27],Qp[26],Qp[25])+m[14]+K1, 13);\
\
  Q[29] = rot(Q[25]+G(Q[28],Q[27],Q[26])+m[3]+K1, 3);\
  Qp[29] = rot(Qp[25]+G(Qp[28],Qp[27],Qp[26])+m[3]+K1, 3);\
  Q[30] = rot(Q[26]+G(Q[29],Q[28],Q[27])+m[7]+K1, 5);\
  Qp[30] = rot(Qp[26]+G(Qp[29],Qp[28],Qp[27])+m[7]+K1, 5);\
  Q[31] = rot(Q[27]+G(Q[30],Q[29],Q[28])+m[11]+K1, 9);\
  Qp[31] = rot(Qp[27]+G(Qp[30],Qp[29],Qp[28])+m[11]+K1, 9);\
  Q[32] = rot(Q[28]+G(Q[31],Q[30],Q[29])+m[15]+K1, 13);\
  Qp[32] = rot(Qp[28]+G(Qp[31],Qp[30],Qp[29])+m[15]+K1, 13);\
\
  Q[33] = rot(Q[29]+H(Q[32],Q[31],Q[30])+m[0]+K2, 3);\
  Qp[33] = rot(Qp[29]+H(Qp[32],Qp[31],Qp[30])+m[0]+K2, 3);\
  Q[34] = rot(Q[30]+H(Q[33],Q[32],Q[31])+m[8]+K2, 9);\
  Qp[34] = rot(Qp[30]+H(Qp[33],Qp[32],Qp[31])+m[8]+K2, 9);\
  Q[35] = rot(Q[31]+H(Q[34],Q[33],Q[32])+m[4]+K2, 11);\
```

```
  Qp[35] = rot(Qp[31]+H(Qp[34],Qp[33],Qp[32])+m[4]+K2, 11);\
  Q[36] = rot(Q[32]+H(Q[35],Q[34],Q[33])+m[12]+K2, 15);\
  Qp[36] = rot(Qp[32]+H(Qp[35],Qp[34],Qp[33])+m[12]+1+K2, 15);\
  if (Qp[36]==Q[36]) break;

int main() {
  int i, ctr, trials, step;
  word t, k, Z, Z1, Z2;
  double tm;
  int seed = time(0)&0xffff;
  srand(seed);
  printf("Seed: %d\n", seed);
  trials = 0;
  tm = current_time_micros();

  for (i = 0; i < N; i++) {

    while (1) {
      step = ctr = 0;

      while (1) {
        if ((++ctr>700 && step<2) || (ctr > 100000)) {
          step = ctr = 0;
        }

        if (!step) {
          Q15 = random_word();
          Q16 = random_word();
          Q17 = random_word();
          Q18 = random_word();
          Q19 = random_word();
          Q20 = random_word();
        }

        Q[15] = Q15 ^ random_bit();
        Q[16] = Q16 ^ random_bit();
        Q[17] = Q17 ^ random_bit();
        Q[18] = Q18 ^ random_bit();
        Q[19] = Q19 ^ random_bit();
        Q[20] = Q20 ^ random_bit();

        Qp[20] = Q[20]-(1<<25);
        Qp[19] = Q[19]+(1<<5);
        Qp[16] = Q[16]-G(Qp[19],Q[18],Q[17]) +
```

```
      G(Q[19],Q[18],Q[17])+rot(Qp[20],19)-rot(Q[20],19)-1;
    Qp[15] = Q[15]-G(Q[18],Q[17],Qp[16]) +
      G(Q[18],Q[17],Q[16])+rot(Qp[19],23)-rot(Q[19],23);
    Q[14] = rot(Q[15],21)-rot(Qp[15],21);
    Qp[14] = Q[14] - G(Q[17],Qp[16],Qp[15]) +
      G(Q[17],Q[16],Q[15]);

    if (G(Q[16],Q[15],Q[14])-G(Qp[16],Qp[15],Qp[14])-1)
      continue;
    if (!step) {
      step = 1;
      Q15 = Q[15]; Q16 = Q[16]; Q17 = Q[17];
      Q18 = Q[18]; Q19 = Q[19]; Q20 = Q[20];
    }

    Z = F(Qp[15],Qp[14],0)-F(Q[15],Q[14],-1)-
      rot(Qp[16],13)+rot(Q[16],13);

    Z2 = Z<<(28-(step<<2));
    Z1 = Z2<<4;

    if (!Z1) {
      if (!Z2) {
        if (++step == 8) {
          if (G(Q[20],Q[19],Q[18])==G(Qp[20],Qp[19],Q[18]))
            break;
          step = ctr = 0;
        }
      }
      Q15 = Q[15]; Q16 = Q[16]; Q17 = Q[17];
      Q18 = Q[18]; Q19 = Q[19]; Q20 = Q[20];
    }
  }

  ctr = 0;

  m[13] = random_word();
  Q[11] = rot(Qp[14],25)-rot(Q[14],25);
  m[12] = rot(Q[20],19)-Q[16]-G(Q[19],Q[18],Q[17])-K1;
  m[14] = rot(Q[15],21)-Q[11]-Q[14];
  m[15] = rot(Q[16],13)-F(Q[15],Q[14],-1);
  m[0]  = rot(Q[17],29)+1-G(Q[16],Q[15],Q[14])-K1;
  m[4]  = rot(Q[18],27)-Q[14]-G(Q[17],Q[16],Q[15])-K1;
  m[8]  = rot(Q[19],23)-Q[15]-G(Q[18],Q[17],Q[16])-K1;
```

```
      Q[10] = rot(Q[14],25)-m[13];
      Q[9]  = -1-Q[10]-m[12];

      do {
        k = 0;
        if (++ctr > 500000) break;
        trials++;
        k = 1;
        m[1]  = random_word();
        m[2]  = random_word();
        m[3]  = random_word();
        m[5]  = random_word();

        Q[1]  = rot(IVA+F(IVB,IVC,IVD)+m[0], 3);
        Q[2]  = rot(IVD+F(Q[1],IVB,IVC)+m[1], 7);
        Q[3]  = rot(IVC+F(Q[2],Q[1],IVB)+m[2], 11);
        Q[4]  = rot(IVB+F(Q[3],Q[2],Q[1])+m[3], 19);

        Q[5]  = rot(Q[1]+F(Q[4],Q[3],Q[2])+m[4], 3);
        Q[6]  = rot(Q[2]+F(Q[5],Q[4],Q[3])+m[5], 7);

        t = rot(Q[9],29)-Q[5]-m[8];
        m[6]  = rot(t,21)-Q[3]-F(Q[6],Q[5],Q[4]);
        m[7]  = -1-Q[4]-F(t,Q[6],Q[5]);
        m[9]  = rot(Q[10],25)-Q[6]-F(Q[9],-1,t);
        m[10] = rot(Q[11],21)-t-F(Q[10],Q[9],-1);
        m[11] = rot(Q[12],13)+1-F(Q[11],Q[10],Q[9]);

        collision();
      } while (1);
      if (k) break;
    }
  }

  tm = current_time_micros()-tm;
  printf("%d collision(s) found in %.2f secs,\n", N, tm/1e6);
  printf("%.2f secs on average\n", tm/1e6/N);

  printf("\nThe last collision found was:\n");
  for (i = 0; i < 16; i++) {
    printf("m[%2d] = 0x%08lx;\n", i, m[i]);
  }
  printf("\n(m[12] can be exchanged with [m12]+1)\n\n");
  return 0;
```

```
}
```

Appendix C

An implementation of the Wang MD5 attack

The code below is a C program that finds collisions of MD5.

```
#include <stdio.h>
#include <time.h>

typedef unsigned long word;

static word Q[65];
static word T, T2, stop;
static word iva, ivb, ivc, ivd;
static word myIVA, myIVB, myIVC, myIVD;

static int ok, i, start_over, changed, ctr;

#define TRY2 0x10000
#define TRY1 0xffff

#define IVA 0x67452301
#define IVB 0xefcdab89
#define IVC 0x98badcfe
#define IVD 0x10325476

#define t0 0xd76aa478
#define t1 0xe8c7b756
#define t2 0x242070db
#define t3 0xc1bdceee
#define t4 0xf57c0faf
#define t5 0x4787c62a
#define t6 0xa8304613
```

```
#define t7 0xfd469501
#define t8 0x698098d8
#define t9 0x8b44f7af
#define t10 0xffff5bb1
#define t11 0x895cd7be
#define t12 0x6b901122
#define t13 0xfd987193
#define t14 0xa679438e
#define t15 0x49b40821
#define t16 0xf61e2562
#define t17 0xc040b340
#define t18 0x265e5a51
#define t19 0xe9b6c7aa
#define t20 0xd62f105d
#define t21 0x02441453
#define t22 0xd8a1e681
#define t23 0xe7d3fbc8
#define t24 0x21e1cde6
#define t25 0xc33707d6
#define t26 0xf4d50d87
#define t27 0x455a14ed
#define t28 0xa9e3e905
#define t29 0xfcefa3f8
#define t30 0x676f02d9
#define t31 0x8d2a4c8a
#define t32 0xfffa3942
#define t33 0x8771f681
#define t34 0x6d9d6122
#define t35 0xfde5380c
#define t36 0xa4beea44
#define t37 0x4bdecfa9
#define t38 0xf6bb4b60
#define t39 0xbebfbc70
#define t40 0x289b7ec6
#define t41 0xeaa127fa
#define t42 0xd4ef3085
#define t43 0x04881d05
#define t44 0xd9d4d039
#define t45 0xe6db99e5
#define t46 0x1fa27cf8
#define t47 0xc4ac5665
#define t48 0xf4292244
#define t49 0x432aff97
#define t50 0xab9423a7
```

```
#define t51 0xfc93a039
#define t52 0x655b59c3
#define t53 0x8f0ccc92
#define t54 0xffeff47d
#define t55 0x85845dd1
#define t56 0x6fa87e4f
#define t57 0xfe2ce6e0
#define t58 0xa3014314
#define t59 0x4e0811a1
#define t60 0xf7537e82
#define t61 0xbd3af235
#define t62 0x2ad7d2bb
#define t63 0xeb86d391

word m[16], m1[16];

#if defined(_WIN32) || defined(__INTEL_COMPILER)
#define rot _lrotl
#define rotr _lrotr
#else
#define rot(x, s) (((x)<<(s))^((x)>>(32-(s))))
#define rotr(x, s) (((x)<<(32-(s)))^((x)>>(s)))
#endif

#define F(u, v, w) ((w) ^ ((u) & ((v) ^ (w))))
#define G(u, v, w) ((v) ^ ((w) & ((u) ^ (v))))
#define H(u, v, w) ((u) ^ (v) ^ (w))
#define I(u, v, w) ((v) ^ ((u) | ~(w)))

#define msb_equal(x, y) (((~x)^(y))&0x80000000)
#define bitsequal(x, n, y) (((~(x)^(y))>>n)&1)
#define random_word() (random()) // mrand48()

double current_time_micros() {
  /* returns current time in microseconds */
  struct timeval curtime;
  gettimeofday(&curtime, 0);
  return ((double) curtime.tv_sec)* 1000000 +
    ((double) curtime.tv_usec);
}

// FIRST BLOCK:

inline void search1() {
```

```
//   int i;
do {
  ok = 0;

  Q[3] = mrand48();
  Q[3] &= 0xfff7f7bf;

  Q[4] = random_word(); // bit 32 = 0
  Q[4] &= 0xff7fffbf;
  Q[4] |= 0x00080800;
  Q[4] ^= (Q[3]^Q[4])&0x77f780;

  Q[5] = random_word();
  Q[5] &= 0xfd40003f;
  Q[5] |= 0x08400025;
  Q[5] ^= (Q[4]^Q[5])&0x80000000;
  if (!(rotr(Q[5]-Q[4], 7)>>31)) continue; // T_4

  Q[6] = mrand48();
  Q[6] &= 0xf77fbc5b;
  Q[6] |= 0x827fbc41; // 32 = 1
  Q[6] ^= (Q[5]^Q[6])&0x7500001a;
  if (rotr(Q[6]-Q[5],12)&0x80000) continue; // T_5

  Q[7] = 0x03fef820;
  T = rotr(Q[7]-Q[6], 17); // T_6
  if (T&0x4000 || !((T>>10)&0xf)) continue;
  m[6] = T - F(Q[6],Q[5],Q[4]) - Q[3] - t6;

  Q[8] = random_word();
  Q[8] &= 0x01f15540;
  Q[8] |= 0x01910540;
  T = rotr(Q[8]-Q[7], 22); // T_7
  // - these T-conditions are always satisfied
  m[7] = T - F(Q[7],Q[6],Q[5]) - Q[4] - t7;

  do {
    do {
      Q[9] = random_word();
      Q[9] &= 0xfbf07f3d;
      Q[9] |= 0x7b102f3d;
      Q[9] ^= (Q[8]^Q[9])&0x1000;
      Q[9] ^= (~Q[7]^Q[9])&0x80000000;
      T = rotr(Q[9]-Q[8],7); // T_8
```

```
        if (!(T>>31) || !(T&0x1000000)) continue;
        m[8] = T - F(Q[8],Q[7],Q[6]) - Q[5] - t8;

        Q[10] = random_word();
        Q[10] &= 0xff7fde7c;
        Q[10] |= 0x401f9040;
        Q[10] ^= (~Q[9]^Q[10])&0x80000000;
        T = rotr(Q[10]-Q[9], 12); // T_9
        if (!(~T>>27)) continue;
        m[9] = T - F(Q[9],Q[8],Q[7]) - Q[6] - t9;

        Q[11] = random_word();
        Q[11] &= 0xbff1cefc;
        Q[11] |= 0x000180c2;
        Q[11] ^= (Q[10]^Q[11])&0x80004000;
        T = rotr(Q[11]-Q[10], 17); // T_10
      } while (!(~T&0x6000) || !(T>>27));
      m[10] = T - F(Q[10],Q[9],Q[8]) - Q[7] - t10;

      Q[12] = random_word();
      Q[12] &= 0xbff81f7f;
      Q[12] |= 0x00081100;
      Q[12] ^= (Q[11]^Q[12])&0x83000000;
      T = rotr(Q[12]-Q[11], 22); // T_11
    } while (!(T&0x300) || !(T>>24));

    m[11] = T - F(Q[11],Q[10],Q[9]) - Q[8] - t11;

    // No strict conditions on T_12
    Q[13] = random_word();
    Q[13] &= 0xfdfffe7f;
    Q[13] |= 0x410fe008;
    Q[13] ^= (Q[12]^Q[13])&0x80000000;
    m[12] = rotr(Q[13]-Q[12], 7) -
      F(Q[12],Q[11],Q[10]) - Q[9] - t12;

    do {
      // No strict conditions on T_13
      Q[14] = random_word();
      Q[14] &= 0xdcfbffff;
      Q[14] |= 0x000be188;
      Q[14] ^= (Q[13]^Q[14])&0x80000000;
      m[13] = rotr(Q[14]-Q[13], 12) -
        F(Q[13],Q[12],Q[11]) - Q[10] - t13;
```

```
    Q[15] = mrand48();
    Q[15] &= 0xfdfffff7; // Removed 31 = 0
    Q[15] |= 0x21008000;
    T = rotr(Q[15]-Q[14], 17); // T_14
    if (!(T>>30)) continue;
    m[14] = T - F(Q[14],Q[13],Q[12]) - Q[11] - t14;

    Q[16] = random_word();
    Q[16] |= 0x20000000;
    Q[16] ^= (Q[15]^Q[16])&0x80000000;

    T = rotr(Q[16]-Q[15], 22); // T_15
    m[15] = T - F(Q[15],Q[14],Q[13]) - Q[12] - t15;
    if ((T&0x380) && (~T>>26)) break;

} while (1);

ctr = 0;
do {
    ctr++;
    if (ctr > TRY1) break;
    Q[17] = random_word();
    Q[17] &= 0xfffdffff;
    Q[17] ^= (Q[16]^Q[17])&0x80008008;
    Q[18] = Q[17] +
      rot(G(Q[17],Q[16],Q[15])+Q[14]+m[6]+t17, 9);
    Q[19] = Q[18] +
      rot(G(Q[18],Q[17],Q[16])+Q[15]+m[11]+t18, 14);
}
while (!((Q[18]>>17)&1) ||
        (Q[17]^Q[18])&0xa0000000 ||
        (Q[19]&0x20000) ||
        (Q[18]^Q[19])&0x80000000);
if (ctr > TRY1) continue;

T = rotr(Q[17]-Q[16], 5); // T_16
if (T&0x1000000) continue;
m[1] = T - G(Q[16],Q[15],Q[14]) - Q[13] - t16;

ok = 0;
stop = ~Q[19]&0x80000000;
for (Q[20] = Q[19]&0x80000000; Q[20] != stop; Q[20]++) {
    T = rotr(Q[20]-Q[19], 20); // T_19
```

```
if (!(T>>29)) continue;
m[0] = T - G(Q[19],Q[18],Q[17])-Q[16]-t19;

Q[1] = IVB + rot(0xd76aa477 + m[0], 7);
Q[2] = Q[1] +
    rot(IVD + F(Q[1], IVB, IVC) + t1 + m[1], 12);
m[2] = rotr(Q[3]-Q[2], 17) - F(Q[2],Q[1],IVB) -
   IVC - t2;
m[3] = rotr(Q[4]-Q[3], 22) - F(Q[3],Q[2],Q[1]) -
   IVB - t3;
m[4] = rotr(Q[5]-Q[4], 7) - F(Q[4],Q[3],Q[2]) -
   Q[1] - t4;
m[5] = rotr(Q[6]-Q[5], 12) - F(Q[5],Q[4],Q[3]) -
   Q[2] - t5;

// check rest
Q[21] = Q[20] +
    rot(Q[17] + G(Q[20], Q[19], Q[18]) + t20 + m[5], 5);
if (!bitsequal(Q[21],17,Q[20]) ||
      !msb_equal(Q[21], Q[20])) continue;


Q[22] = Q[21] +
    rot(Q[18] + G(Q[21], Q[20], Q[19]) + t21 + m[10], 9);
if (!msb_equal(Q[22], Q[21])) continue;

// T_22
T = Q[19] + G(Q[22], Q[21], Q[20]) + t22 + m[15];
if (T&0x20000) continue;

Q[23] = Q[22] + rot(T, 14);
if (Q[23]>>31) continue;

Q[24] = Q[23] +
    rot(Q[20] + G(Q[23], Q[22], Q[21]) + t23 + m[4], 20);
if (!(Q[24]>>31)) continue;


Q[25] = Q[24] +
    rot(Q[21] + G(Q[24], Q[23], Q[22]) + t24 + m[9], 5);
Q[26] = Q[25] +
    rot(Q[22] + G(Q[25], Q[24], Q[23]) + t25 + m[14], 9);
Q[27] = Q[26] +
    rot(Q[23] + G(Q[26], Q[25], Q[24]) + t26 + m[3], 14);
```

```
Q[28] = Q[27] +
  rot(Q[24] + G(Q[27], Q[26], Q[25]) + t27 + m[8], 20);

Q[29] = Q[28] +
  rot(Q[25] + G(Q[28], Q[27], Q[26]) + t28 + m[13], 5);
Q[30] = Q[29] +
  rot(Q[26] + G(Q[29], Q[28], Q[27]) + t29 + m[2], 9);
Q[31] = Q[30] +
  rot(Q[27] + G(Q[30], Q[29], Q[28]) + t30 + m[7], 14);
Q[32] = Q[31] +
  rot(Q[28] + G(Q[31], Q[30], Q[29]) + t31 + m[12], 20);

Q[33] = Q[32] +
  rot(Q[29] + H(Q[32], Q[31], Q[30]) + t32 + m[5], 4);
Q[34] = Q[33] +
  rot(Q[30] + H(Q[33], Q[32], Q[31]) + t33 + m[8], 11);

// T_34
T = Q[31] + H(Q[34], Q[33], Q[32]) + t34 + m[11];
if (T&0x8000) continue;

Q[35] = Q[34] + rot(T, 16);
Q[36] = Q[35] +
  rot(Q[32] + H(Q[35], Q[34], Q[33]) + t35 + m[14], 23);

Q[37] = Q[36] +
  rot(Q[33] + H(Q[36], Q[35], Q[34]) + t36 + m[1], 4);
Q[38] = Q[37] +
  rot(Q[34] + H(Q[37], Q[36], Q[35]) + t37 + m[4], 11);
Q[39] = Q[38] +
  rot(Q[35] + H(Q[38], Q[37], Q[36]) + t38 + m[7], 16);
Q[40] = Q[39] +
  rot(Q[36] + H(Q[39], Q[38], Q[37]) + t39 + m[10], 23);

Q[41] = Q[40] +
  rot(Q[37] + H(Q[40], Q[39], Q[38]) + t40 + m[13], 4);
Q[42] = Q[41] +
  rot(Q[38] + H(Q[41], Q[40], Q[39]) + t41 + m[0], 11);
Q[43] = Q[42] +
  rot(Q[39] + H(Q[42], Q[41], Q[40]) + t42 + m[3], 16);
Q[44] = Q[43] +
  rot(Q[40] + H(Q[43], Q[42], Q[41]) + t43 + m[6], 23);

Q[45] = Q[44] +
```

```
    rot(Q[41] + H(Q[44], Q[43], Q[42]) + t44 + m[9], 4);
Q[46] = Q[45] +
    rot(Q[42] + H(Q[45], Q[44], Q[43]) + t45 + m[12], 11);
Q[47] = Q[46] +
    rot(Q[43] + H(Q[46], Q[45], Q[44]) + t46 + m[15], 16);
Q[48] = Q[47] +
    rot(Q[44] + H(Q[47], Q[46], Q[45]) + t47 + m[2], 23);
if ((Q[48]^Q[46])&0x80000000) continue;

Q[49] = Q[48] +
    rot(Q[45] + I(Q[48], Q[47], Q[46]) + t48 + m[0], 6);
if ((Q[49]^Q[47])&0x80000000) continue;

Q[50] = Q[49] +
    rot(Q[46] + I(Q[49], Q[48], Q[47]) + t49 + m[7], 10);
if ((~Q[50]^Q[46])&0x80000000) continue;

Q[51] = Q[50] +
    rot(Q[47] + I(Q[50], Q[49], Q[48]) + t50 + m[14], 15);
if ((Q[51]^Q[49])&0x80000000) continue;

Q[52] = Q[51] +
    rot(Q[48] + I(Q[51], Q[50], Q[49]) + t51 + m[5], 21);
if ((Q[52]^Q[50])&0x80000000) continue;

Q[53] = Q[52] +
    rot(Q[49] + I(Q[52], Q[51], Q[50]) + t52 + m[12], 6);
if ((Q[53]^Q[51])&0x80000000) continue;

Q[54] = Q[53] +
    rot(Q[50] + I(Q[53], Q[52], Q[51]) + t53 + m[3], 10);
if ((Q[54]^Q[52])&0x80000000) continue;

Q[55] = Q[54] +
    rot(Q[51] + I(Q[54], Q[53], Q[52]) + t54 + m[10], 15);
if ((Q[55]^Q[53])&0x80000000) continue;

Q[56] = Q[55] +
    rot(Q[52] + I(Q[55], Q[54], Q[53]) + t55 + m[1], 21);
if ((Q[56]^Q[54])&0x80000000) continue;

Q[57] = Q[56] +
    rot(Q[53] + I(Q[56], Q[55], Q[54]) + t56 + m[8], 6);
if ((Q[57]^Q[55])&0x80000000) continue;
```

```
Q[58] = Q[57] +
  rot(Q[54] + I(Q[57], Q[56], Q[55]) + t57 + m[15], 10);
if ((Q[58]^Q[56])&0x80000000) continue;

Q[59] = Q[58] +
  rot(Q[55] + I(Q[58], Q[57], Q[56]) + t58 + m[6], 15);
if ((Q[59]^Q[57])&0x80000000) continue;

Q[60] = Q[59] +
  rot(Q[56] + I(Q[59], Q[58], Q[57]) + t59 + m[13], 21);
if ((Q[60]&0x2000000) || ((~Q[60]^Q[58])&0x80000000))
  continue;

Q[61] = Q[60] +
  rot(Q[57] + I(Q[60], Q[59], Q[58]) + t60 + m[4], 6);
if ((!(Q[61]&0x2000000)) || ((Q[61]^Q[59])&0x80000000))
  continue;

iva = IVA + Q[61];

T = Q[58] + I(Q[61], Q[60], Q[59]) + t61 + m[11];
if (!(~T&0x3f8000)) continue;

Q[62] = Q[61] + rot(T, 10);
if ((Q[62]&0x2000000) || ((Q[62]^Q[60])&0x80000000))
  continue;

ivd = IVD + Q[62];
if (ivd&0x2000000) continue;

Q[63] = Q[62] +
  rot(Q[59] + I(Q[62], Q[61], Q[60]) + t62 + m[2], 15);
if ((Q[63]&0x2000000) || ((Q[63]^Q[61])&0x80000000))
  continue;

ivc = IVC + Q[63];
if ((!(ivc&0x2000000)) || (ivc&0x4000000) ||
    ((ivc^ivd)&0x80000000))
  continue;

printf("."); fflush(stdout);

Q[64] = Q[63] +
```

```
        rot(Q[60] + I(Q[63], Q[62], Q[61]) + t63 + m[9], 21);
      ivb = IVB + Q[64];
      if ((ivb&0x06000020) || ((ivb^ivc)&0x80000000)) continue;

      myIVA = iva; myIVB = ivb; myIVC = ivc; myIVD = ivd;
      ok = 1;
      break;
    }

  } while (!ok);

}



// SECOND BLOCK:

inline void modify_0_13() {
  do {
    Q[1] = random_word();
    Q[1] &= 0xf5fff7df;
    //      Q[1] |= 0x04200040;
    Q[1] |= 0x04200000;
    Q[1] ^= (~Q[1]^myIVB)&0x80000000;

    Q[2] = random_word();
    Q[2] &= 0xfddfffd9;
    //      Q[2] |= 0x0c000840;
    Q[2] |= 0x0c000800;
    //      Q[2] ^= (Q[1]^Q[2])&0xf01f1080;
    Q[2] ^= (Q[1]^Q[2])&0xf01f10c0;

    T = rotr(Q[2]-Q[1], 12);
    if (!(~T>>25)) continue;
    m[1] = T - myIVD - F(Q[1],myIVB,myIVC) - t1;

    Q[3] = random_word();
    Q[3] &= 0xbfdfef7f;
    Q[3] |= 0x3e1f0966;
    Q[3] ^= (Q[2]^Q[3])&0x80000018;

    T = rotr(Q[3]-Q[2], 17);
    if ((T>>31) || !((~T>>26)&0x1f)) continue;
    m[2] = T - myIVC - F(Q[2],Q[1],myIVB) - t2;
```

```
Q[4] = random_word();
Q[4] &= 0xbbc4e611;
Q[4] |= 0x3a040010;
Q[4] ^= (Q[3]^Q[4])&0x80000601;

T = rotr(Q[4]-Q[3], 22);
if (!(T>>27)) continue;
m[3] = T - myIVB - F(Q[3],Q[2],Q[1]) - t3;

Q[5] = random_word();
Q[5] &= 0xcbefee50;
Q[5] |= 0x482f0e50;
Q[5] ^= (~Q[4]^Q[5])&0x80000000;

T = rotr(Q[5]-Q[4], 7);
if ((T>>31) || !(T&0x40000000)) continue;
m[4] = T - Q[1] - F(Q[4],Q[3],Q[2]) - t4;

Q[6] = random_word();
Q[6] &= 0xe5eeec56;
Q[6] |= 0x04220c56;
Q[6] ^= (Q[5]^Q[6])&0x80000000;

T = rotr(Q[6]-Q[5], 12);
if (!(T>>30)) continue;
m[5] = T - Q[2] - F(Q[5],Q[4],Q[3]) - t5;

Q[7] = random_word();
Q[7] &= 0xf7cdfe3f;
Q[7] |= 0x16011e01;
Q[7] ^= (~Q[6]^Q[7])&0x80000000;
Q[7] ^= (Q[6]^Q[7])&0x1808000;

T = rotr(Q[7]-Q[6], 17);
if ((T>>31) || !(T&0x7c00)) continue;
m[6] = T - Q[3] - F(Q[6],Q[5],Q[4]) - t6;

Q[8] = random_word();
Q[8] &= 0xe47efffe;
Q[8] |= 0x043283e0; // Added Q_8[5] = 1 *
Q[8] ^= (Q[7]^Q[8])&0x80000002;

T = rotr(Q[8]-Q[7], 22);
```

```
if (!(~T&0x3f0) || !(~T>>27)) continue;
m[7] = T - Q[4] - F(Q[7],Q[6],Q[5]) - t7;

Q[9] = random_word();
Q[9] &= 0xfc7d7ddd; // Added Q_9[5] = 0 *
Q[9] |= 0x1c0101c1;
Q[9] ^= (Q[8]^Q[9])&0x80001000;
// * so that Q[9]-Q[8] produces a carry into bit 6

T = rotr(Q[9]-Q[8], 7);
if (!(T>>31) || !(~T&0x7e000000)) continue;
m[8] = T - Q[5] - F(Q[8],Q[7],Q[6]) - t8;

Q[10] = random_word();
Q[10] &= 0xfffbeffc;
Q[10] |= 0x078383c0;
// Added Q[10],3 != Q[9],3
// so correction in step 19 works:
Q[10] ^= (~Q[9]^Q[10])&0x8;
Q[10] ^= (Q[9]^Q[10])&0x80000000;
T = rotr(Q[10]-Q[9],12);
if (!(T>>27)) continue;
m[9] = T - Q[6] - F(Q[9],Q[8],Q[7]) - t9;

Q[11] = random_word();
Q[11] &= 0xfffdefff;
Q[11] |= 0x000583c3;
Q[11] ^= (Q[10]^Q[11])&0x80086000;

T = rotr(Q[11]-Q[10],17);
if (!(T>>27)) continue;
m[10] = T - Q[7] - F(Q[10],Q[9],Q[8]) - t10;

Q[12] = random_word();
Q[12] &= 0xfff81fff;
Q[12] |= 0x00081080;
Q[12] ^= (Q[12]^Q[11])&0xff000000;

Q[13] = random_word();
Q[13] &= 0xbfffff7f;
Q[13] |= 0x3f0fe008;
Q[13] ^= (~Q[12]^Q[13])&0x80000000;

Q[14] = random_word();
```

```
    Q[14] &= 0xc0fbffff;
    Q[14] |= 0x400be088;
    Q[14] ^= (Q[13]^Q[14])&0x80000000;

    T = rotr(Q[14]-Q[13], 12);
    if (!((T>>12)&0xff)) continue;
    m[13] = T - Q[10] - F(Q[13],Q[12],Q[11]) - t13;

    m[0] = rotr(Q[1]-myIVB, 7) - myIVA -
      F(myIVB,myIVC,myIVD) - t0;
    m[11] = rotr(Q[12]-Q[11], 22) - Q[8] -
      F(Q[11],Q[10],Q[9]) - t11;
    m[12] = rotr(Q[13]-Q[12], 7) - Q[9] -
      F(Q[12],Q[11],Q[10]) - t12;
    m[13] = rotr(Q[14]-Q[13], 12) - Q[10] -
      F(Q[13],Q[12],Q[11]) - t13;

    break;
  } while (1);
}

inline void checkrest() {
  ctr = 0;
  start_over = 1;
  do {
    ctr++;

    Q[15] = random_word();
    Q[15] &= 0x7dff7ff7;
    Q[15] |= 0x7d000000;

    T = rotr(Q[15]-Q[14], 17);
    if (!(~T>>30)) continue;

    m[14] = T - Q[11] - F(Q[14],Q[13],Q[12]) - t14;

    Q[16] = random_word();
    Q[16] &= 0x7fffffff;
    Q[16] |= 0x20000000;
//      Q[16] ^= (Q[15]^Q[16])&0x80000000;

    T = rotr(Q[16]-Q[15], 22);
    if (!(T&0x380) || !(T>>26)) continue;
```

```
m[15] = T - Q[12] - F(Q[15],Q[14],Q[13]) - t15;

changed = 0;
Q[17] = Q[16] +
   rot(G(Q[16], Q[15], Q[14]) + Q[13] + m[1] + t16, 5);
if (!bitsequal(Q[17],3,Q[16])) {
  m[1] += ((Q[2]>>10)&1) ? -0x40000000 : 0x40000000;
  Q[2] ^= 0x400;
  Q[17] = Q[16] +
    rot(G(Q[16], Q[15], Q[14]) + Q[13] + m[1] + t16, 5);
  changed = 1;
}
if (!bitsequal(Q[17],15,Q[16])) {
  m[1] += ((Q[2]>>22)&1) ? -0x400 : 0x400;
  Q[2] ^= 0x400000;
  Q[17] = Q[16] +
    rot(G(Q[16], Q[15], Q[14]) + Q[13] + m[1] + t16, 5);
  changed = 1;
}
if (Q[17]&0x20000) {
  m[1] += ((Q[2]>>24)&1) ? -0x1000 : 0x1000;
  Q[2] ^= 0x1000000;
  Q[17] = Q[16] +
    rot(G(Q[16], Q[15], Q[14]) + Q[13] + m[1] + t16, 5);
  changed = 1;
}
if (changed) {
  T = rotr(Q[2]-Q[1], 12);
  if (!(~T>>25)) return;
  T = rotr(Q[3]-Q[2], 17);
  if ((T>>31) || !((~T>>26)&0x1f)) return;
  m[2] = T - myIVC - F(Q[2],Q[1],myIVB) - t2;
  m[3] = rotr(Q[4]-Q[3], 22) - myIVB -
    F(Q[3],Q[2],Q[1]) - t3;
  m[4] = rotr(Q[5]-Q[4], 7) - Q[1] -
    F(Q[4],Q[3],Q[2]) - t4;
  m[5] = rotr(Q[6]-Q[5], 12) - Q[2] -
    F(Q[5],Q[4],Q[3]) - t5;
}
if (Q[17]>>31) continue;

if ((Q[17]-Q[16])>>29 == 0x7) continue;

Q[18] = Q[17] +
```

```
  rot(G(Q[17], Q[16], Q[15]) + Q[14] + t17 + m[6], 9);
// Seems impossible to correct:
if (!(Q[18]&0x20000)) continue;

if (!bitsequal(Q[18],29,Q[17])) {
  m[6] += ((Q[7]>>5)&1) ? -0x100000 : 0x100000;
  Q[7] ^= 0x20;
  Q[18] = Q[17] +
    rot(G(Q[17], Q[16], Q[15]) + Q[14] + t17 + m[6], 9);

  T = rotr(Q[7]-Q[6], 17);
  if ((T>>31) || !(T&0x7c00)) return;

  T = rotr(Q[8]-Q[7], 22);
  if (!(~T&0x3f0) || !(~T>>27)) return;
  m[7] = T - Q[4] - F(Q[7],Q[6],Q[5]) - t7;

  m[8] = rotr(Q[9]-Q[8], 7) - Q[5] -
    F(Q[8],Q[7],Q[6]) - t8;
  m[9] = rotr(Q[10]-Q[9], 12) - Q[6] -
    F(Q[9],Q[8],Q[7]) - t9;
  m[10] = rotr(Q[11]-Q[10], 17) - Q[7] -
    F(Q[10],Q[9],Q[8]) - t10;
}
if (Q[18]>>31) {
  Q[3] ^= 0x400000;

  T = rotr(Q[3]-Q[2], 17);
  if ((T>>31) || !((~T>>26)&0x1f)) return;
  m[2] = T - myIVC - F(Q[2],Q[1],myIVB) - t2;

  T = rotr(Q[4]-Q[3], 22);
  if (!(T>>27)) return;
  m[3] = T - myIVB - F(Q[3],Q[2],Q[1]) - t3;

  m[4] = rotr(Q[5]-Q[4], 7) - Q[1] -
    F(Q[4],Q[3],Q[2]) - t4;
  m[5] = rotr(Q[6]-Q[5], 12) - Q[2] -
    F(Q[5],Q[4],Q[3]) - t5;
  m[6] = rotr(Q[7]-Q[6], 17) - Q[3] -
    F(Q[6],Q[5],Q[4]) - t6;

  Q[18] = Q[17] +
    rot(G(Q[17], Q[16], Q[15]) + Q[14] + t17 + m[6], 9);
```

```
}

Q[19] = Q[18] +
   rot(Q[15] + G(Q[18], Q[17], Q[16]) + t18 + m[11], 14);
if (Q[19]&0x20000) {
  Q[11] ^= 0x8;
  T = rotr(Q[11]-Q[10],17);
  if (!(T>>27)) return;
  m[10] = T - Q[7] - F(Q[10],Q[9],Q[8]) - t10;
  m[11] = rotr(Q[12]-Q[11], 22) - Q[8] -
     F(Q[11],Q[10],Q[9]) - t11;
  m[12] = rotr(Q[13]-Q[12], 7) - Q[9] -
     F(Q[12],Q[11],Q[10]) - t12;
  m[13] = rotr(Q[14]-Q[13], 12) - Q[10] -
     F(Q[13],Q[12],Q[11]) - t13;
  m[14] = rotr(Q[15]-Q[14], 17) - Q[11] -
     F(Q[14],Q[13],Q[12]) - t14;
  Q[19] = Q[18] + rot(Q[15] + G(Q[18], Q[17], Q[16]) +
                       t18 + m[11], 14);
}
if (Q[19]>>31) continue;

// check rest
T = Q[16] + G(Q[19], Q[18], Q[17]) + t19 + m[0];
if (!(T>>29)) continue;
Q[20] = Q[19] + rot(T, 20);
if (Q[20]>>31) continue;

Q[21] = Q[20] +
   rot(Q[17] + G(Q[20], Q[19], Q[18]) + t20 + m[5], 5);
if (!bitsequal(Q[21],17,Q[20]) || Q[21]>>31) continue;

Q[22] = Q[21] +
   rot(Q[18] + G(Q[21], Q[20], Q[19]) + t21 + m[10], 9);
if (Q[22]>>31) continue;

T = Q[19] + G(Q[22], Q[21], Q[20]) + t22 + m[15];
if (T&0x20000) continue; // p = 1/2

Q[23] = Q[22] + rot(T, 14);
if (Q[23]>>31) continue;

Q[24] = Q[23] +
   rot(Q[20] + G(Q[23], Q[22], Q[21]) + t23 + m[4], 20);
```

```
if (!(Q[24]>>31)) continue;

Q[25] = Q[24] +
  rot(Q[21] + G(Q[24], Q[23], Q[22]) + t24 + m[9], 5);
Q[26] = Q[25] +
  rot(Q[22] + G(Q[25], Q[24], Q[23]) + t25 + m[14], 9);
Q[27] = Q[26] +
  rot(Q[23] + G(Q[26], Q[25], Q[24]) + t26 + m[3], 14);
Q[28] = Q[27] +
  rot(Q[24] + G(Q[27], Q[26], Q[25]) + t27 + m[8], 20);

Q[29] = Q[28] +
  rot(Q[25] + G(Q[28], Q[27], Q[26]) + t28 + m[13], 5);
Q[30] = Q[29] +
  rot(Q[26] + G(Q[29], Q[28], Q[27]) + t29 + m[2], 9);
Q[31] = Q[30] +
  rot(Q[27] + G(Q[30], Q[29], Q[28]) + t30 + m[7], 14);
Q[32] = Q[31] +
  rot(Q[28] + G(Q[31], Q[30], Q[29]) + t31 + m[12], 20);

Q[33] = Q[32] +
  rot(Q[29] + H(Q[32], Q[31], Q[30]) + t32 + m[5], 4);
Q[34] = Q[33] +
  rot(Q[30] + H(Q[33], Q[32], Q[31]) + t33 + m[8], 11);

// Step 34 T-check
T = Q[31] + H(Q[34], Q[33], Q[32]) + t34 + m[11];
if (!(T&0x8000)) continue; // p = 1/2

Q[35] = Q[34] + rot(T, 16);
Q[36] = Q[35] +
  rot(Q[32] + H(Q[35], Q[34], Q[33]) + t35 + m[14], 23);

Q[37] = Q[36] +
  rot(Q[33] + H(Q[36], Q[35], Q[34]) + t36 + m[1], 4);
Q[38] = Q[37] +
  rot(Q[34] + H(Q[37], Q[36], Q[35]) + t37 + m[4], 11);
Q[39] = Q[38] +
  rot(Q[35] + H(Q[38], Q[37], Q[36]) + t38 + m[7], 16);
Q[40] = Q[39] +
  rot(Q[36] + H(Q[39], Q[38], Q[37]) + t39 + m[10], 23);

Q[41] = Q[40] +
  rot(Q[37] + H(Q[40], Q[39], Q[38]) + t40 + m[13], 4);
```

```
Q[42] = Q[41] +
  rot(Q[38] + H(Q[41], Q[40], Q[39]) + t41 + m[0], 11);
Q[43] = Q[42] +
  rot(Q[39] + H(Q[42], Q[41], Q[40]) + t42 + m[3], 16);
Q[44] = Q[43] +
  rot(Q[40] + H(Q[43], Q[42], Q[41]) + t43 + m[6], 23);

Q[45] = Q[44] +
  rot(Q[41] + H(Q[44], Q[43], Q[42]) + t44 + m[9], 4);
Q[46] = Q[45] +
  rot(Q[42] + H(Q[45], Q[44], Q[43]) + t45 + m[12], 11);
Q[47] = Q[46] +
  rot(Q[43] + H(Q[46], Q[45], Q[44]) + t46 + m[15], 16);
Q[48] = Q[47] +
  rot(Q[44] + H(Q[47], Q[46], Q[45]) + t47 + m[2], 23);
if ((Q[48]^Q[46])&0x80000000) continue;

Q[49] = Q[48] +
  rot(Q[45] + I(Q[48], Q[47], Q[46]) + t48 + m[0], 6);
if ((Q[49]^Q[47])&0x80000000) continue;

Q[50] = Q[49] +
  rot(Q[46] + I(Q[49], Q[48], Q[47]) + t49 + m[7], 10);
if ((~Q[50]^Q[46])&0x80000000) continue;

Q[51] = Q[50] +
  rot(Q[47] + I(Q[50], Q[49], Q[48]) + t50 + m[14], 15);
if ((Q[51]^Q[49])&0x80000000) continue;

Q[52] = Q[51] +
  rot(Q[48] + I(Q[51], Q[50], Q[49]) + t51 + m[5], 21);
if ((Q[52]^Q[50])&0x80000000) continue;

Q[53] = Q[52] +
  rot(Q[49] + I(Q[52], Q[51], Q[50]) + t52 + m[12], 6);
if ((Q[53]^Q[51])&0x80000000) continue;

Q[54] = Q[53] +
  rot(Q[50] + I(Q[53], Q[52], Q[51]) + t53 + m[3], 10);
if ((Q[54]^Q[52])&0x80000000) continue;

Q[55] = Q[54] +
  rot(Q[51] + I(Q[54], Q[53], Q[52]) + t54 + m[10], 15);
if ((Q[55]^Q[53])&0x80000000) continue;
```

```
Q[56] = Q[55] +
   rot(Q[52] + I(Q[55], Q[54], Q[53]) + t55 + m[1], 21);
if ((Q[56]^Q[54])&0x80000000) continue;

Q[57] = Q[56] +
   rot(Q[53] + I(Q[56], Q[55], Q[54]) + t56 + m[8], 6);
if ((Q[57]^Q[55])&0x80000000) continue;

Q[58] = Q[57] +
   rot(Q[54] + I(Q[57], Q[56], Q[55]) + t57 + m[15], 10);
if ((Q[58]^Q[56])&0x80000000) continue;

Q[59] = Q[58] +
   rot(Q[55] + I(Q[58], Q[57], Q[56]) + t58 + m[6], 15);
if ((Q[59]^Q[57])&0x80000000) continue;

Q[60] = Q[59] +
   rot(Q[56] + I(Q[59], Q[58], Q[57]) + t59 + m[13], 21);
if ((Q[60]&0x2000000) || (~Q[60]^Q[58])&0x80000000)
   continue;

Q[61] = Q[60] +
   rot(Q[57] + I(Q[60], Q[59], Q[58]) + t60 + m[4], 6);
if ((!(Q[61]&0x2000000)) || ((Q[61]^Q[59])&0x80000000))
   continue;

T = Q[58] + I(Q[61], Q[60], Q[59]) + t61 + m[11];
if (!(T&0x3f8000)) continue;

Q[62] = Q[61] + rot(T, 10);
if (!(Q[62]&0x2000000) || ((Q[62]^Q[60])&0x80000000))
   continue;

printf(":"); fflush(stdout);
Q[63] = Q[62] +
   rot(Q[59] + I(Q[62], Q[61], Q[60]) + t62 + m[2], 15);
if ((Q[63]^Q[61])&0x80000000) continue;

if (!(Q[63]&0x2000000)) {
  T = Q[63]>>26;
  T2 = Q[61]>>26;
  if (!T) continue;
```

```
      while (!(T&1)) {
        if (T2&1) break;
        T>>=1; T2>>=1;
      }
      if (!(T&1) || T2&1) continue;
    }

    printf("\n");

    start_over = 0;
    break;
  } while (ctr < TRY2);
}

inline void search2() {
  do {
    modify_0_13();
    checkrest();
  } while (start_over);
}

int main(int argc, char* argv[]) {
  int j;
  double t, tfirst, tsecond;
  FILE *fp1;
  srandom(time(0));
  srand48(time(0));

  t = current_time_micros();
  search1(); // Find first block
  tfirst = current_time_micros()-t;
  for (j = 0; j < 16; j++) m1[j] = m[j];

  search2(); // Find second block
  t = current_time_micros()-t;
  tsecond = t-tfirst;

  printf("Collision found in %.2f minutes\n", t/1e6/60);
  printf("Time of first block  : %.2f minutes\n",
         tfirst/1e6/60);
  printf("Time of second block : %.2f minutes\n",
         tsecond/1e6/60);
  printf("NOTE: Since not all conditions are checked,\n");
  printf("this might in fact NOT be a true collision.\n");
```

```
printf("Please verify...\n\n");
printf("First block:\n");
for (j = 0; j < 16; j++)
  printf("m[%d] = 0x%08lx;\n", j, m1[j]);
printf("\nThe colliding message has m[4]+2^31, m[11]+2^15,\n");
printf("m[14]+2^31\n\n");
printf("Second block:\n");
for (j = 0; j < 16; j++)
  printf("m[%d] = 0x%08lx;\n", j, m[j]);
printf("\nThe colliding message has m[4]+2^31, m[11]-2^15,\n");
printf("m[14]+2^31\n\n");
printf("The two messages have been saved as msg1.txt\n");
printf("and msg2.txt. To verify on a UNIX-like system,\n");
printf("use 'md5sum msg1.txt msg2.txt'.\n\n");

fp1 = fopen("msg1.txt", "w");
for (j = 0; j < 16; j++) {
  fprintf(fp1, "%c",(char)(m1[j]&0xff));
  fprintf(fp1, "%c",(char)((m1[j]>>8)&0xff));
  fprintf(fp1, "%c",(char)((m1[j]>>16)&0xff));
  fprintf(fp1, "%c",(char)(m1[j]>>24));
}
for (j = 0; j < 16; j++) {
  fprintf(fp1, "%c",(char)(m[j]&0xff));
  fprintf(fp1, "%c",(char)((m[j]>>8)&0xff));
  fprintf(fp1, "%c",(char)((m[j]>>16)&0xff));
  fprintf(fp1, "%c",(char)(m[j]>>24));
}
fclose(fp1);

m1[4] += 0x80000000;
m1[11] += 0x8000;
m1[14] += 0x80000000;
m[4] += 0x80000000;
m[11] -= 0x8000;
m[14] += 0x80000000;

fp1 = fopen("msg2.txt", "w");
for (j = 0; j < 16; j++) {
  fprintf(fp1, "%c",(char)(m1[j]&0xff));
  fprintf(fp1, "%c",(char)((m1[j]>>8)&0xff));
  fprintf(fp1, "%c",(char)((m1[j]>>16)&0xff));
  fprintf(fp1, "%c",(char)(m1[j]>>24));
}
```

```
  for (j = 0; j < 16; j++) {
    fprintf(fp1, "%c",(char)(m[j]&0xff));
    fprintf(fp1, "%c",(char)((m[j]>>8)&0xff));
    fprintf(fp1, "%c",(char)((m[j]>>16)&0xff));
    fprintf(fp1, "%c",(char)(m[j]>>24));
  }
  fclose(fp1);

  return 0;
}
```

Appendix D

An implementation of the Wang MD4 attack

The code below is a C program that finds collisions of MD4 using the technique of [43].

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

typedef unsigned long word;

word a[13], b[13], c[13], d[13];
word aa[13], bb[13], cc[13], dd[13];
word T;
#define IVA 0x67452301
#define IVB 0xefcdab89
#define IVC 0x98badcfe
#define IVD 0x10325476

word m[16];
int ok = 0;

#define F(u, v, w) ((w) ^ ((u) & ((v) ^ (w))))
#define G(u, v, w) (((u) & (v)) ^ ((w) & ((u) ^ (v))))
#define H(u, v, w) ((u) ^ (v) ^ (w))
#define rot(x, s) (((x)<<(s))^((x)>>(32-(s))))
#define rotr(x, s) (((x)<<(32-(s)))^((x)>>(s)))
#define K1 (0x5a827999)
#define K2 (0x6ed9eba1)
#define N 10000
```

```
double current_time_micros() {
  /* returns current time in microseconds */
  struct timeval curtime;
  gettimeofday(&curtime, 0);
  return ((double) curtime.tv_sec)* 1000000 +
    ((double) curtime.tv_usec);
}

inline void search() {
  ok = 0;
  do {
    // Added b1[14,21,22,24,27] = 0
    b[1] = random();
    b[1] &= 0xf94fdb7f;
    b[1] |= 0x00000040;

    a[2] = random();
    a[2] &= 0xfdffffff;
    a[2] |= 0x480;
    a[2] ^= (a[2]^b[1])&0x2000;

    // Added extra conditions: d2,[17,18,23] = 0
    d[2] = random();
    d[2] &= 0xffffdfff;
    d[2] |= 0x02000000;
    d[2] ^= (d[2]^a[2])&0x003c0000;

    // Added c2[17] = 0
    c[2] = random();
    c[2] &= 0xffd2dfff;
    c[2] |= 0x00100000;
    c[2] ^= (d[2]^c[2])&0x00005000;

    // Added extra conditions: b2[17,18,23] = 0
    b[2] = random();
    b[2] &= 0xff80bfff;
    b[2] |= 0x00003000;
    b[2] ^= (b[2]^c[2])&0x00010000;
    m[7] = rotr(b[2], 19) - b[1] - F(c[2],d[2],a[2]);

    a[3] = random();
    a[3] &= 0xffe2ffff;
    a[3] |= 0x00207000;
    a[3] ^= (a[3]^b[2])&0x02400000;
```

```
m[8] = rotr(a[3], 3) - a[2] - F(b[2],c[2],d[2]);

d[3] = random();
d[3] &= 0xffb6ffff;
d[3] |= 0x02307000;
d[3] ^= (d[3]^a[3])&0x20000000;
m[9] = rotr(d[3], 7) - d[2] - F(a[3],b[2],c[2]);

c[3] = random();
c[3] &= 0xfd87ffff;
c[3] |= 0x20010000;
c[3] ^= (c[3]^d[3])&0x80000000;
m[10] = rotr(c[3], 11) - c[2] - F(d[3],a[3],b[2]);

b[3] = random();
b[3] &= 0x5fb7ffff;
b[3] |= 0x02300000;
m[11] = rotr(b[3], 19) - b[2] - F(c[3],d[3],a[3]);

a[4] = random();
a[4] &= 0x7dbfffff;
a[4] |= 0x20000000;
a[4] ^= (a[4]^b[3])&0x14000000;
m[12] = rotr(a[4], 3) - a[3] - F(b[3],c[3],d[3]);

do {
  d[4] = random();
  d[4] &= 0xddbfffff;
  d[4] |= 0x94000000;
  m[13] = rotr(d[4], 7) - d[3] - F(a[4],b[3],c[3]);

  c[4] = random();
  c[4] &= 0xcbffffff;
  c[4] |= 0x02400000;
  c[4] ^= (c[4]^d[4])&0x40000;
  m[14] = rotr(c[4], 11) - c[3] - F(d[4],a[4],b[3]);

  // Added extra condition: b4[32] = c4[32]
  // (missing in paper!)
  b[4] = random();
  b[4] &= 0xdffbffff;
  b[4] |= 0x16000000;
  b[4] ^= (b[4]^c[4])&0x80000000;
  m[15] = rotr(b[4], 19) - b[3] - F(c[4],d[4],a[4]);
```

```
    a[5] = random();
    a[5] &= 0xfbffffff;
    a[5] |= 0x92000000;
    a[5] ^= 0x40000&(c[4]^a[5]);
    m[0] = rotr(a[5], 3) - a[4] - G(b[4],c[4],d[4]) - K1;
    a[1] = rot(a[0] + F(b[0],c[0],d[0]) + m[0], 3);
    d[1] = random();
    d[1] &= 0xffffffbf;
    d[1] ^= (d[1]^a[1])&0x480;
    c[1] = random();
    c[1] &= 0xfffffbff;
    c[1] |= 0xc0;
    c[1] ^= (c[1]^d[1])&0x02000000;
    m[4] = rotr(a[2], 3) - a[1] - F(b[1],c[1],d[1]);
    m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
    m[6] = rotr(c[2], 11) - c[1] - F(d[2],a[2],b[1]);
} while (0x40&(a[1]^b[0]));

m[1] = rotr(d[1], 7) - d[0] - F(a[1],b[0],c[0]);
m[2] = rotr(c[1], 11) - c[0] - F(d[1],a[1],b[0]);
m[3] = rotr(b[1], 19) - b[0] - F(c[1],d[1],a[1]);

d[5] = rot(d[4] + G(a[5],b[4],c[4]) + m[4] + K1, 5);
// For all possible incorrect bits, b1 points at d1
if ((d[5]^a[5])&0x40000) {
    m[4] += (d[1]&0x2000) ? 0x2000 : -(0x2000);
    d[1] ^= 0x2000;
    d[5] = rot(d[4] + G(a[5],b[4],c[4]) + m[4] + K1, 5);
    m[1] = rotr(d[1], 7) - d[0] - F(a[1],b[0],c[0]);
    m[2] = rotr(c[1], 11) - c[0] - F(d[1],a[1],b[0]);
    m[3] = rotr(b[1], 19) - b[0] - F(c[1],d[1],a[1]);
    m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
}

if ((d[5]^b[4])&0x2000000) {
    m[4] += (d[1]&0x100000) ? 0x100000 : -(0x100000);
    d[1] ^= 0x100000;
    d[5] = rot(d[4] + G(a[5],b[4],c[4]) + m[4] + K1, 5);
    m[1] = rotr(d[1], 7) - d[0] - F(a[1],b[0],c[0]);
    m[2] = rotr(c[1], 11) - c[0] - F(d[1],a[1],b[0]);
    m[3] = rotr(b[1], 19) - b[0] - F(c[1],d[1],a[1]);
    m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
}
```

```
if ((d[5]^b[4])&0x4000000) {
  m[4] += (d[1]&0x200000) ? 0x200000 : -(0x200000);
  d[1] ^= 0x200000;
  d[5] = rot(d[4] + G(a[5],b[4],c[4]) + m[4] + K1, 5);
  m[1] = rotr(d[1], 7) - d[0] - F(a[1],b[0],c[0]);
  m[2] = rotr(c[1], 11) - c[0] - F(d[1],a[1],b[0]);
  m[3] = rotr(b[1], 19) - b[0] - F(c[1],d[1],a[1]);
  m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
}

if ((d[5]^b[4])&0x10000000) {
  m[4] += (d[1]&0x800000) ? 0x800000 : -(0x800000);
  d[1] ^= 0x800000;
  d[5] = rot(d[4] + G(a[5],b[4],c[4]) + m[4] + K1, 5);
  m[1] = rotr(d[1], 7) - d[0] - F(a[1],b[0],c[0]);
  m[2] = rotr(c[1], 11) - c[0] - F(d[1],a[1],b[0]);
  m[3] = rotr(b[1], 19) - b[0] - F(c[1],d[1],a[1]);
  m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
}

if ((d[5]^b[4])&0x80000000) {
  m[4] += (d[1]&0x4000000) ? 0x4000000 : -(0x4000000);
  d[1] ^= 0x4000000;
  d[5] = rot(d[4] + G(a[5],b[4],c[4]) + m[4] + K1, 5);
  m[1] = rotr(d[1], 7) - d[0] - F(a[1],b[0],c[0]);
  m[2] = rotr(c[1], 11) - c[0] - F(d[1],a[1],b[0]);
  m[3] = rotr(b[1], 19) - b[0] - F(c[1],d[1],a[1]);
  m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
}

c[5] = rot(c[4] + G(d[5],a[5],b[4]) + m[8] + K1, 9);
if ((c[5]^d[5])&0x2000000) {
  m[8] += (d[2]&0x10000) ? 0x10000 : -0x10000;
  d[2] ^= 0x10000;
  c[5] = rot(c[4] + G(d[5],a[5],b[4]) + m[8] + K1, 9);
  m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
  m[6] = rotr(c[2], 11) - c[1] - F(d[2],a[2],b[1]);
  m[7] = rotr(b[2], 19) - b[1] - F(c[2],d[2],a[2]);
  m[9] = rotr(d[3], 7) - d[2] - F(a[3],b[2],c[2]);
}

if ((c[5]^d[5])&0x4000000) {
  m[8] += (d[2]&0x20000) ? 0x20000 : -0x20000;
```

```
    d[2] ^= 0x20000;
    c[5] = rot(c[4] + G(d[5],a[5],b[4]) + m[8] + K1, 9);
    m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
    m[6] = rotr(c[2], 11) - c[1] - F(d[2],a[2],b[1]);
    m[7] = rotr(b[2], 19) - b[1] - F(c[2],d[2],a[2]);
    m[9] = rotr(d[3], 7) - d[2] - F(a[3],b[2],c[2]);
  }

  // dangerous to correct this one...
  if ((c[5]^d[5])&0x10000000) {
    continue;
  }

  if ((c[5]^d[5])&0x20000000) {
    m[8] += (a[3]&0x800000) ? -0x100000 : 0x100000;
    a[3] ^= 0x800000;
    c[5] = rot(c[4] + G(d[5],a[5],b[4]) + m[8] + K1, 9);
    m[9] = rotr(d[3], 7) - d[2] - F(a[3],b[2],c[2]);
    m[10] = rotr(c[3], 11) - c[2] - F(d[3],a[3],b[2]);
    m[11] = rotr(b[3], 19) - b[2] - F(c[3],d[3],a[3]);
    m[12] = rotr(a[4], 3) - a[3] - F(b[3],c[3],d[3]);
  }

  if ((c[5]^d[5])&0x80000000) {
    m[8] += (d[2]&0x400000) ? 0x400000 : -0x400000;
    d[2] ^= 0x400000;
    c[5] = rot(c[4] + G(d[5],a[5],b[4]) + m[8] + K1, 9);
    m[5] = rotr(d[2], 7) - d[1] - F(a[2],b[1],c[1]);
    m[6] = rotr(c[2], 11) - c[1] - F(d[2],a[2],b[1]);
    m[7] = rotr(b[2], 19) - b[1] - F(c[2],d[2],a[2]);
    m[9] = rotr(d[3], 7) - d[2] - F(a[3],b[2],c[2]);
  }

  b[5] = rot(b[4] + G(c[5],d[5],a[5]) + m[12] + K1, 13);
  // No conditions on b3,16, c3,16 and d3,16.
  // I.e. flip the bit that b3,16 points at.
  if ((b[5]^c[5])&0x10000000) {
    if (b[3]&0x8000) {
      m[12] += (c[3]&0x8000) ? 0x8000 : -0x8000;
      c[3] ^= 0x8000;
      b[5] = rot(b[4] + G(c[5],d[5],a[5]) + m[12] + K1, 13);
      m[10] = rotr(c[3], 11) - c[2] - F(d[3],a[3],b[2]);
      m[11] = rotr(b[3], 19) - b[2] - F(c[3],d[3],a[3]);
      m[13] = rotr(d[4], 7) - d[3] - F(a[4],b[3],c[3]);
```

```
    m[14] = rotr(c[4], 11) - c[3] - F(d[4],a[4],b[3]);
  }
  else {
    m[12] += (d[3]&0x8000) ? 0x8000 : -0x8000;
    d[3] ^= 0x8000;
    b[5] = rot(b[4] + G(c[5],d[5],a[5]) + m[12] + K1, 13);
    m[9] = rotr(d[3], 7) - d[2] - F(a[3],b[2],c[2]);
    m[10] = rotr(c[3], 11) - c[2] - F(d[3],a[3],b[2]);
    m[11] = rotr(b[3], 19) - b[2] - F(c[3],d[3],a[3]);
    m[13] = rotr(d[4], 7) - d[3] - F(a[4],b[3],c[3]);
  }
}

// This trick only works since c3,17 != d3,17
if (!(b[5]&0x20000000)) {
  m[12] += (b[3]&0x10000) ? 0x10000 : -0x10000;
  b[3] ^= 0x10000;
  b[5] = rot(b[4] + G(c[5],d[5],a[5]) + m[12] + K1, 13);
  m[11] = rotr(b[3], 19) - b[2] - F(c[3],d[3],a[3]);
  m[13] = rotr(d[4], 7) - d[3] - F(a[4],b[3],c[3]);
  m[14] = rotr(c[4], 11) - c[3] - F(d[4],a[4],b[3]);
  m[15] = rotr(b[4], 19) - b[3] - F(c[4],d[4],a[4]);
}

// No conditions on b3,19, c3,19 and d3,19.
// I.e. flip the bit that b3,19 points at.
if (b[5]&0x80000000) {
  if (b[3]&0x40000) {
    m[12] += (c[3]&0x40000) ? 0x40000 : -0x40000;
    c[3] ^= 0x40000;
    b[5] = rot(b[4] + G(c[5],d[5],a[5]) + m[12] + K1, 13);
    m[10] = rotr(c[3], 11) - c[2] - F(d[3],a[3],b[2]);
    m[11] = rotr(b[3], 19) - b[2] - F(c[3],d[3],a[3]);
    m[13] = rotr(d[4], 7) - d[3] - F(a[4],b[3],c[3]);
    m[14] = rotr(c[4], 11) - c[3] - F(d[4],a[4],b[3]);
  }
  else {
    m[12] += (d[3]&0x40000) ? 0x40000 : -0x40000;
    d[3] ^= 0x40000;
    b[5] = rot(b[4] + G(c[5],d[5],a[5]) + m[12] + K1, 13);
    m[9] = rotr(d[3], 7) - d[2] - F(a[3],b[2],c[2]);
    m[10] = rotr(c[3], 11) - c[2] - F(d[3],a[3],b[2]);
    m[11] = rotr(b[3], 19) - b[2] - F(c[3],d[3],a[3]);
    m[13] = rotr(d[4], 7) - d[3] - F(a[4],b[3],c[3]);
```

```
      }
    }

    a[6] = rot(a[5] + G(b[5],c[5],d[5]) + m[1] + K1, 3);
    if (!(a[6]&0x10000000)) continue;
    if (a[6]&0x20000000) continue;
    if (!(a[6]&0x80000000)) continue;

    d[6] = rot(d[5] + G(a[6],b[5],c[5]) + m[5] + K1, 5);
    if ((d[6]^b[5])&0x10000000) continue;

    c[6] = rot(c[5] + G(d[6],a[6],b[5]) + m[9] + K1, 9);
    if ((c[6]^d[6])&0x10000000) continue;
    if (!((c[6]^d[6])&0x20000000)) continue;
    if (!((c[6]^d[6])&0x80000000)) continue;

    b[6] = rot(b[5] + G(c[6],d[6],a[6]) + m[13] + K1, 13);
    a[7] = rot(a[6] + G(b[6],c[6],d[6]) + m[2] + K1, 3);
    d[7] = rot(d[6] + G(a[7],b[6],c[6]) + m[6] + K1, 5);
    c[7] = rot(c[6] + G(d[7],a[7],b[6]) + m[10] + K1, 9);
    b[7] = rot(b[6] + G(c[7],d[7],a[7]) + m[14] + K1, 13);
    a[8] = rot(a[7] + G(b[7],c[7],d[7]) + m[3] + K1, 3);
    d[8] = rot(d[7] + G(a[8],b[7],c[7]) + m[7] + K1, 5);
    c[8] = rot(c[7] + G(d[8],a[8],b[7]) + m[11] + K1, 9);
    b[8] = rot(b[7] + G(c[8],d[8],a[8]) + m[15] + K1, 13);

    a[9] = rot(a[8] + H(b[8],c[8],d[8]) + m[0] + K2, 3);
    d[9] = rot(d[8] + H(a[9],b[8],c[8]) + m[8] + K2, 9);
    c[9] = rot(c[8] + H(d[9],a[9],b[8]) + m[4] + K2, 11);
    T = b[8] + H(c[9],d[9],a[9]) + m[12] + K2;
    if (!(T&0x10000)) continue;
    b[9] = rot(T, 15);

    a[10] = rot(a[9] + H(b[9],c[9],d[9]) + m[2] + K2, 3);
    if ((a[10]^b[9])&0x80000000) continue;

    ok = 1;
  } while (!ok);
}


int main() {
  int i, error = 0;
  double tm;
```

```
  int seed = time(0)&0xffff;

  srandom(seed);
  printf("Seed: %d\n", seed);
  tm = current_time_micros();

  for (i = 0; i < N; i++) {
    a[0] = IVA; b[0] = IVB; c[0] = IVC; d[0] = IVD;
    search();
  }
  tm = current_time_micros()-tm;
  printf("%d collision(s) found in %.2f secs,\n", N, tm/1e6);
  printf("%.2f ms on average\n", tm/1e3/N);
  a[0] = IVA; b[0] = IVB; c[0] = IVC; d[0] = IVD;

  printf("\nThe last collision found was:\n");
  for (i = 0; i < 16; i++) {
    printf("m[%2d] = 0x%08lx;\n", i, m[i]);
  }
  printf("\nReplacing\n m[1] with m[1]+0x80000000=0x%08lx,\n",
         m[1]+0x80000000);
  printf(" m[2] with m[2]+0x70000000=0x%08lx, and\n",
         m[2]+0x70000000);
  printf(" m[12] with m[12]-0x10000=0x%08lx\n", m[12]-0x10000);
  printf("does not change the MD4 hash of this message.\n\n");
  return 0;
}
```

Appendix E

The individual steps of MD4

For convenience, the 48 individual steps of MD4 are written out below. The notation is the same as in Chapter 3.

Step no.	Step operation
0	$Q_1 \leftarrow (F(Q_0, Q_{-1}, Q_{-2}) + Q_{-3} + m_0)^{\lll 3}$
1	$Q_2 \leftarrow (F(Q_1, Q_0, Q_{-1}) + Q_{-2} + m_1)^{\lll 7}$
2	$Q_3 \leftarrow (F(Q_2, Q_1, Q_0) + Q_{-1} + m_2)^{\lll 11}$
3	$Q_4 \leftarrow (F(Q_3, Q_2, Q_1) + Q_0 + m_3)^{\lll 19}$
4	$Q_5 \leftarrow (F(Q_4, Q_3, Q_2) + Q_1 + m_4)^{\lll 3}$
5	$Q_6 \leftarrow (F(Q_5, Q_4, Q_3) + Q_2 + m_5)^{\lll 7}$
6	$Q_7 \leftarrow (F(Q_6, Q_5, Q_4) + Q_3 + m_6)^{\lll 11}$
7	$Q_8 \leftarrow (F(Q_7, Q_6, Q_5) + Q_4 + m_7)^{\lll 19}$
8	$Q_9 \leftarrow (F(Q_8, Q_7, Q_6) + Q_5 + m_8)^{\lll 3}$
9	$Q_{10} \leftarrow (F(Q_9, Q_8, Q_7) + Q_6 + m_9)^{\lll 7}$
10	$Q_{11} \leftarrow (F(Q_{10}, Q_9, Q_8) + Q_7 + m_{10})^{\lll 11}$
11	$Q_{12} \leftarrow (F(Q_{11}, Q_{10}, Q_9) + Q_8 + m_{11})^{\lll 19}$
12	$Q_{13} \leftarrow (F(Q_{12}, Q_{11}, Q_{10}) + Q_9 + m_{12})^{\lll 3}$
13	$Q_{14} \leftarrow (F(Q_{13}, Q_{12}, Q_{11}) + Q_{10} + m_{13})^{\lll 7}$
14	$Q_{15} \leftarrow (F(Q_{14}, Q_{13}, Q_{12}) + Q_{11} + m_{14})^{\lll 11}$
15	$Q_{16} \leftarrow (F(Q_{15}, Q_{14}, Q_{13}) + Q_{12} + m_{15})^{\lll 19}$
16	$Q_{17} \leftarrow (G(Q_{16}, Q_{15}, Q_{14}) + Q_{13} + k_1 + m_0)^{\lll 3}$
17	$Q_{18} \leftarrow (G(Q_{17}, Q_{16}, Q_{15}) + Q_{14} + k_1 + m_4)^{\lll 5}$
18	$Q_{19} \leftarrow (G(Q_{18}, Q_{17}, Q_{16}) + Q_{15} + k_1 + m_8)^{\lll 9}$
19	$Q_{20} \leftarrow (G(Q_{19}, Q_{18}, Q_{17}) + Q_{16} + k_1 + m_{12})^{\lll 13}$
20	$Q_{21} \leftarrow (G(Q_{20}, Q_{19}, Q_{18}) + Q_{17} + k_1 + m_1)^{\lll 3}$
21	$Q_{22} \leftarrow (G(Q_{21}, Q_{20}, Q_{19}) + Q_{18} + k_1 + m_5)^{\lll 5}$
22	$Q_{23} \leftarrow (G(Q_{22}, Q_{21}, Q_{20}) + Q_{19} + k_1 + m_9)^{\lll 9}$
23	$Q_{24} \leftarrow (G(Q_{23}, Q_{22}, Q_{21}) + Q_{20} + k_1 + m_{13})^{\lll 13}$
24	$Q_{25} \leftarrow (G(Q_{24}, Q_{23}, Q_{22}) + Q_{21} + k_1 + m_2)^{\lll 3}$
25	$Q_{26} \leftarrow (G(Q_{25}, Q_{24}, Q_{23}) + Q_{22} + k_1 + m_6)^{\lll 5}$
26	$Q_{27} \leftarrow (G(Q_{26}, Q_{25}, Q_{24}) + Q_{23} + k_1 + m_{10})^{\lll 9}$

27	$Q_{28} \leftarrow (G(Q_{27}, Q_{26}, Q_{25}) + Q_{24} + k_1 + m_{14})^{\lll 13}$
28	$Q_{29} \leftarrow (G(Q_{28}, Q_{27}, Q_{26}) + Q_{25} + k_1 + m_3)^{\lll 3}$
29	$Q_{30} \leftarrow (G(Q_{29}, Q_{28}, Q_{27}) + Q_{26} + k_1 + m_7)^{\lll 5}$
30	$Q_{31} \leftarrow (G(Q_{30}, Q_{29}, Q_{28}) + Q_{27} + k_1 + m_{11})^{\lll 9}$
31	$Q_{32} \leftarrow (G(Q_{31}, Q_{30}, Q_{29}) + Q_{28} + k_1 + m_{15})^{\lll 13}$
32	$Q_{33} \leftarrow (H(Q_{32}, Q_{31}, Q_{30}) + Q_{29} + k_2 + m_0)^{\lll 3}$
33	$Q_{34} \leftarrow (H(Q_{33}, Q_{32}, Q_{31}) + Q_{30} + k_2 + m_8)^{\lll 9}$
34	$Q_{35} \leftarrow (H(Q_{34}, Q_{33}, Q_{32}) + Q_{31} + k_2 + m_4)^{\lll 11}$
35	$Q_{36} \leftarrow (H(Q_{35}, Q_{34}, Q_{33}) + Q_{32} + k_2 + m_{12})^{\lll 15}$
36	$Q_{37} \leftarrow (H(Q_{36}, Q_{35}, Q_{34}) + Q_{33} + k_2 + m_2)^{\lll 3}$
37	$Q_{38} \leftarrow (H(Q_{37}, Q_{36}, Q_{35}) + Q_{34} + k_2 + m_{10})^{\lll 9}$
38	$Q_{39} \leftarrow (H(Q_{38}, Q_{37}, Q_{36}) + Q_{35} + k_2 + m_6)^{\lll 11}$
39	$Q_{40} \leftarrow (H(Q_{39}, Q_{38}, Q_{37}) + Q_{36} + k_2 + m_{14})^{\lll 15}$
40	$Q_{41} \leftarrow (H(Q_{40}, Q_{39}, Q_{38}) + Q_{37} + k_2 + m_1)^{\lll 3}$
41	$Q_{42} \leftarrow (H(Q_{41}, Q_{40}, Q_{39}) + Q_{38} + k_2 + m_9)^{\lll 9}$
42	$Q_{43} \leftarrow (H(Q_{42}, Q_{41}, Q_{40}) + Q_{39} + k_2 + m_5)^{\lll 11}$
43	$Q_{44} \leftarrow (H(Q_{43}, Q_{42}, Q_{41}) + Q_{40} + k_2 + m_{13})^{\lll 15}$
44	$Q_{45} \leftarrow (H(Q_{44}, Q_{43}, Q_{42}) + Q_{41} + k_2 + m_3)^{\lll 3}$
45	$Q_{46} \leftarrow (H(Q_{45}, Q_{44}, Q_{43}) + Q_{42} + k_2 + m_{11})^{\lll 9}$
46	$Q_{47} \leftarrow (H(Q_{46}, Q_{45}, Q_{44}) + Q_{43} + k_2 + m_7)^{\lll 11}$
47	$Q_{48} \leftarrow (H(Q_{47}, Q_{46}, Q_{45}) + Q_{44} + k_2 + m_{15})^{\lll 15}$

Bibliography

[1] E. Biham. Recent Advances in Hash Functions: The Way to Go. Presentation at the Hash Functions Workshop, Krakow, June 2005. Available at `http://www.cs.technion.ac.il/~biham/Reports/Slides/hash-func-krakow-2005.ps.gz`.

[2] E. Biham, R. Chen, A. Joux, P. Carribault, C. Lemuet, and W. Jalby. Collisions of SHA-0 and Reduced SHA-1. In Cramer [8], pages 36–57.

[3] E. Biham and A. Shamir. Differential Cryptanalysis of DES-like Cryptosystems. In A. Menezes and S. A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990.

[4] D. Boneh and M. K. Franklin. Efficient Generation of Shared RSA Keys (Extended Abstract). In B. S. K. Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 1997.

[5] B. O. Brachtl, D. Coppersmith, M. M. Hyden, S. M. Matyas Jr., C. H. W. Meyer, J. Oseas, S. Pilpel, and M. Schilling. Data authentication using modification detection codes based on a public one-way encryption function. U.S. Patent # 4,908,861, March 1990.

[6] F. Chabaud and A. Joux. Differential Collisions in SHA-0. In H. Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 1998.

[7] S. Contini, A. K. Lenstra, and R. Steinfeld. VSH, an Efficient and Provable Collision Resistant Hash Function. Cryptology ePrint Archive, Report 2005/193, 2005. `http://eprint.iacr.org/`.

[8] R. Cramer, editor. *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

[9] I. Damgård. A Design Principle for Hash Functions. In G. Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989.

[10] M. Daum. *Cryptanalysis of Hash Functions of the MD4-Family*. PhD thesis, Ruhr-Universität Bochum, June 2005.

[11] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[12] H. Dobbertin. Cryptanalysis of MD4. *Journal of Cryptology*, 11(4):253–271, 1998.

[13] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved Cryptanalysis of Rijndael. In B. Schneier, editor, *Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2000.

[14] FIPS 46, Data Encryption Standard. Federal Information Processing Standards Publication 46, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, January 1977. Revised as FIPS 46-1 (1988) and FIPS 46-2 (1993).

[15] FIPS 180, Secure Hash Standard. Federal Information Processing Standards Publication 180, U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, May 1993.

[16] FIPS 180-1, Secure Hash Standard. Federal Information Processing Standards Publication 180-1, U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, April 1995. Supersedes FIPS 180.

[17] FIPS 180-2, Secure Hash Standard. Federal Information Processing Standards Publication 180-2, U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, August 2002. Supersedes FIPS 180 and FIPS 180-1.

[18] FIPS 186, Digital Signature Standard. Federal Information Processing Standards Publication 186, U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, May 1994. Revised as FIPS 186-1 (1998) and FIPS 186-2 (2000).

[19] FIPS 197, Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, November 2001.

[20] The *Hashcash* website, `http://www.hashcash.org/`.

[21] P. Hawkes, M. Paddon, and G. G. Rose. Musings on the Wang et al. MD5 Collision. Cryptology ePrint Archive, Report 2004/264, 2004. `http://eprint.iacr.org/`.

[22] W. Hohl, X. Lai, T. Meier, and C. Waldvogel. Security of Iterated Hash Functions Based on Block Ciphers. In Stinson [40], pages 379–390.

[23] A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In M. K. Franklin, editor, *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[24] J. Kelsey and B. Schneier. Second Preimages on n-bit Hash Functions for Much Less than 2^n Work. In Cramer [8], pages 474–490.

[25] V. Klima. Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications. Cryptology ePrint Archive, Report 2005/102, 2005. `http://eprint.iacr.org/`.

[26] L. R. Knudsen. SMASH – A Cryptographic Hash Function. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption*, volume 3557 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2005.

[27] L. R. Knudsen, X. Lai, and B. Preneel. Attacks on Fast Double Block Length Hash Functions. *Journal of Cryptology*, 11(1):59–72, 1998.

[28] L. R. Knudsen and B. Preneel. Construction of Secure and Fast Hash Functions Using Nonbinary Error-Correcting Codes. *IEEE Transactions on Information Theory*, 48(9):2524–2539, September 2002.

[29] S. Lucks. Design Principles for Iterated Hash Functions. Cryptology ePrint Archive, Report 2004/253, 2004. `http://eprint.iacr.org/`.

[30] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[31] R. C. Merkle. A Fast Software One-Way Hash Function. *Journal of Cryptology*, 3(1):43–58, 1990.

[32] C. H. Meyer and M. Schilling. Secure program load with Manipulation Detection Code. In *Proceedings of the 6th Worldwide Congress on Computer and Communications Security and Protection (SECURICOM'88)*, pages 111–130, 1988.

[33] N. Pramstaller, C. Rechberger, and V. Rijmen. Smashing SMASH. Cryptology ePrint Archive, Report 2005/081, 2005. `http://eprint.iacr.org/`.

[34] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, January 1993.

[35] B. Preneel, R. Govaerts, and J. Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In Stinson [40], pages 368–378.

[36] RFC 1186, The MD4 Message Digest Algorithm. Internet Request for Comments 1186, R. Rivest, October 1990.

[37] RFC 1319, The MD2 Message-Digest Algorithm. Internet Request for Comments 1319, B. Kaliski, April 1992.

[38] RFC 1321, The MD5 Message-Digest Algorithm. Internet Request for Comments 1321, R. Rivest, April 1992.

[39] R. L. Rivest. Abelian square-free dithering for iterated hash functions. Available at `http://theory.lcs.mit.edu/~rivest/Rivest-AbelianSquareFreeDitheringForIteratedHashFunctions.pdf`, August 2005.

[40] D. R. Stinson, editor. *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*. Springer, 1994.

[41] D. R. Stinson. *Cryptography: Theory and Practice*. CRC Press, first edition, 1995.

[42] D. R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, second edition, 2002.

[43] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Cramer [8], pages 1–18.

[44] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. To appear.

[45] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In Cramer [8], pages 19–35.