Generic attacks on iterated hash functions

Thyla Joy van der Merwe

University of Cape Town

Thesis presented for the degree of Master of Science
In the Department of Mathematics and Applied Mathematics
University of Cape Town
February 2009

To my family.

Fall seven times, stand up eight.

I know the meaning of plagiarism and declare that all of the work in this document, save that which is properly acknowledged, is my own.

Signed: Date: 27/05/2009

Signed by candidate

Abstract

We survey the existing generic attacks on hash functions based on the Merkle-Damgård construction: that is, attacks in which the compression function is treated as a black box.

Acknowledgements

I want to thank:

My supervisor Christine Swart for all her hard work, and for constantly pushing me to perform at my best.

The National Research Foundation for supporting me financially.

My family for their continued support and understanding especially you Mom; Greg for his patience and encouragement; Renée, Mieke, Rike, Sofie, Bruce and Monica for their faith in me; Ashley and Shannon for their help with coding; Vasco, Thomas and Mashudu for making the journey memorable; and all those who have contributed in any way to the production of this thesis.

Contents

1 Introduction **9**

2 Probability Theory and Mathematical Tools **15**

2.1 Obtaining a particular result in k trials 15

2.2 The birthday problem 16

2.3 Multicollisions 18

2.4 The birthday problem for collisions between two sets 20

2.4.1 Model A 20

2.4.2 Model B 22

2.4.3 Model C 22

2.5 The geometric distribution 24

2.6 Random functions and cycles 25

2.7 Sequences and partial Orders 26

2.8 Square-free and abelian square-free sequences 28

2.8.1 Square-free words and sequences 29

2.8.2 Abelian square-free words and sequences 29

2.9 A note on big-oh notation and complexity 30

3 Hash Function Basics **31**

3.1 What is a hash function? 31

3.2 Hash function properties 33

3.3 What are hash functions used for? 36

3.3.1 Data integrity and message authentication 36

3.3.2 Confirmation of knowledge 38

3.3.3 Key derivation 41
3.3.4 Pseudorandom number generation 42

4 The Merkle-Damgård Construction for Hash Functions 44
4.1 Definitions and notation 44
4.2 The Merkle-Damgård construction 45
4.3 Merkle's meta method and Damgård's construction 48
4.3.1 Merkle's meta method 48
4.3.2 Damgård's construction 48
4.3.3 A security proof 49
4.4 A more specific classification 51
4.4.1 Hash functions based on block ciphers 51
4.4.2 Customized hash functions 52
4.4.3 Hash functions based on modular arithmetic 56
4.5 BABI: A toy hash function 57

5 Generic Attacks on Hash functions 61
5.1 The birthday attack 61
5.1.1 Finding a collision in one set of messages 61
5.1.2 Finding a collision between two sets of messages 66
5.2 Memoryless techniques for finding collisions 67
5.2.1 Pollard's Rho Method 67
5.2.2 Floyd's cycle finding attack 68
5.2.3 The QD attack 70

6 Generic Attacks on Iterated Hash Functions 72
6.1 The multicollision attack 72
6.1.1 Building multicollisions 73
6.1.2 A consequence of the multicollision attack: Cascaded hash functions 76
6.2 The long-message attack 80
6.3 The long-message attack with expandable messages 81
6.3.1 Expandable messages from fixed points 82

6.3.2 Expandable messages without fixed points 84
6.3.3 Using expandable messages to find second preimages . . . 89
6.3.4 Variations 91
6.4 The herding attack 91
6.4.1 Herding without Merkle-Damgård strengthening 93
6.4.2 Herding with Merkle-Damgård strengthening 95
6.5 Combining herding and the long-message attack 99
6.6 The poisoned block attack 102

7 Generic Attacks on Some Merkle-Damgård Variants 105
7.1 Generalised Sequential Hash Functions 105
7.1.1 Nandi and Stinson's attacks 107
7.2 Dithered hash functions 115
7.2.1 A second preimage attack on dithered hash functions . . . 119

8 Conclusion 125

Bibliography 126

Appendix A: BABI Source Code 132

Appendix B: MD5 Collision Files 142

Chapter 1

Introduction

A cryptographic hash function H takes as input an arbitrary-length message M, and produces a fixed-length output called the *hash value*, or simply just the *hash*, $H(M)$. The hash acts as a commitment for the message itself, because it has some or all of the following properties (given in increasing order of strength):

1. *Preimage resistance*: It is easy to take a message M and compute the hash value $H(M)$, but it is computationally infeasible to take a hash value h and find a message M with $H(M) = h$.

2. *Second preimage resistance*: Given a message M and its hash value $H(M)$, it is computationally infeasible to find a second message M' with the same hash $H(M) = H(M')$.

3. *Collision resistance*: It is computationally infeasible to find two messages M and M' that have the same hash value (although of course, since the output space is finite and the input space is infinite, infinitely many such "collisions" exist).

Note that collisions in a hash function can always be found by brute force: If the hash function produces an n-bit output and one hashes about $2^{n/2}$ random messages, a birthday paradox argument shows that one can expect to find a collision. This attack cannot be avoided, except by making n large enough to prevent it by making the attack computationally infeasible; with modern computing power,

160 bits is enough. Preimages and second preimages can also be found by brute force: This is done by exhaustive search and requires an attacker to compute about 2^n hash values. A hash function is considered to be "broken" if there is a method for finding collisions, preimages or second preimages faster than by brute force.

Hash functions are an integral part of modern cryptography. They are used in applications such as digital signatures: rather than directly signing a message, the signer first hashes the message and then applies the signature to the (typically shorter) hash value. Clearly, if the hash function is not second preimage resistant, then an evil attacker Alice can take a message M signed by Bob, find another message M' with the same hash value, and claim that Bob signed M'. If the hash function is not collision resistant, then Alice can find two messages M and M', sign the first, M, and later claim to have signed M'. Hash functions are also used in conjunction with public-key algorithms for encryption, integrity checking and authentication. Bruce Schneier has been quoted as calling hash functions "the workhorse of modern cryptography".

Some history

The implementation of a one-way function with a cryptographic purpose in mind was first suggested in 1976 by Diffie and Hellman [DH76], who proposed a way to use preimage resistant and second preimage resistant functions for the purpose of authentication. Diffie and Hellman did not use the term "hash function", and the functions mentioned in [DH76] did not necessarily need to be "compressing" in nature. The term *hash function* to imply a compressing, preimage resistant function is probably due to Merkle [Mer79] or Rabin [Rab78]. Neither Merkle nor Rabin mentioned collision resistance, but its importance was soon pointed out by Yuval in [Yuv79].

At about the same time (i.e., around 1979), Rabin, Shamir and Adelman invented the RSA public key cryptosystem [RSA78]. Attacks on the scheme started

to surface shortly after its publication, most notably the chosen message attacks by Davida [Dav82] and Desmedt and Odlyzko [DO86]. In the hope of preventing such attacks, researchers suggested the use of hash functions (see [Den84] and [Win83]). (In addition to improving security, hash functions also made digital signature schemes faster.)

Contini et al. [CSPM07] note that for some time the precise notion of a hash function was disagreed upon by the academic community. In [Den84], Denning required that a hash function be preimage resistant and second preimage resistant only; collision resistance was not mentioned. However, in [Win83], Winternitz stated that hash functions had to be collision resistant if they were to be used in digital signature schemes, yet in turn did not remark on the need for preimage resistance. Contini et al. [CSPM07] propose the reason for this to be that many researches believed that collision resistance implied preimage resistance. Although true for hash functions used in practice, this implication only holds under certain assumptions (the relationship between these two properties will be discussed in more detail in Chapter 3).

Owing to the confusion arising from the use of many different hash function definitions, Preneel [Pre93] attempted to introduce some clarity: Drawing on the work of Rabin [Rab78], Merkle [Mer89] and Damgård [Dr89] he defined a *one-way hash function* to be preimage resistant and second preimage resistant, and a *collision resistant hash function* to be preimage resistant, second preimage resistant and collision resistant. Nowadays we consider the term "cryptographic hash function" to imply a compressing function that satisfies the three properties mentioned above, and consider a hash function to be *ideal* if the best possible way to foil collision resistance is by a birthday attack, and if the best possible way to foil both preimage and second preimage resistance is by brute force.

Until recently, hash functions have not received as much attention from the cryptographic community as other branches of symmetric cryptography like block

ciphers. Interest in the topic suddenly flared after the ingenious attacks of Wang *et al.* [WGLY04], [WY05] on MD4 [Riv92a] and MD5 [Riv92b] were unveiled at CRYPTO[1] in 2004. Since then, cryptographers and cryptanalysts have started to pour energy into the construction of new hash functions, and into discovering the weaknesses present in the hash functions currently used in practice.

Increasing concern about the use of flawed hash functions in industry led the National Institute of Standards and Technology (NIST)[2] in late 2007 to launch a public competition for the development of a new cryptographic hash algorithm [NIS07]. The last time the NIST launched an endeavour of this nature was in 1997; this competition concerned the search for a new, unclassified and publicly disclosed encryption standard that could be used to protect government information. The need for this competition arose because the then current encryption standard, known as the Data Encryption Standard (DES) [NIS99], was no longer secure against modern computing power. As a result of the competition, the Advanced Encryption Standard (AES) [NIS01] came into being. The hash algorithm competition has been launched with a similar purpose in mind: The most commonly used hash functions in industry, namely, MD5 [Riv92b] and SHA-0 [NIS93], have been successfully attacked and thus need to be replaced.

Iterated hash functions and generic attacks

An ideal hash function would behave like a random oracle – all of the properties mentioned above would be satisfied. Unfortunately, in terms of implementation, the best we can do is to build hash functions that "appear" to be random, i.e., hash functions are pseudo-random functions. Most of the hash functions used in industry are built according to a design by Ralph Merkle [Mer89] and Ivan Damgard [Dr89] commonly known as the *Merkle-Damgård construction.* The design makes use of an iterative procedure to process messages one block at a time. Message blocks are processed by what is known as the *compression function.*

[1]CRYPTO is an international cryptology conference held annually in Santa Barbara, California.

[2]NIST is a leading federal standards agency based in the United States of America.

Starting with some pre-specified value and the first message block, the output of the first compression function computation together with the next message block forms the input to the next compression function computation, and so on. We refer to hash functions based on this design as *iterated cryptographic hash functions*.

In 1989 both Merkle [Mer89] and Damgård [Dr89] proved that if no collision could be found in the compression function, then no collision could be found in the hash function. However, this leaves open the question of what would happen if a collision in the compression function could be obtained. As it turns out, finding a collision in the compression function allows one to attack the hash function in a variety of unexpected ways. This means that the Merkle-Damgård construction itself is prone to weaknesses and hash functions based on this construction cannot be as secure as was previously hoped.

We refer to the attacks that highlight the weaknesses of the Merkle-Damgård construction as *generic attacks*. These attacks are applicable to all iterated hash functions. They do not exploit the flaws of a particular compression function, but simply treat the compression function as a black box. The attacks by Wang *et al.* [WY05] (sometimes referred to as the Chinese attacks) are not generic attacks. They attack a specific family of hash functions, namely the MD hash function family, by exploiting flaws in the design of hash functions belonging to this family.

It is in some ways surprising that the generic attacks only started to surface after the Chinese attacks; for instance the very simple and elegant generic multi-collision attack by Joux [Jou04] was only published in 2004. This indicates that until now, hash functions have not been as well studied as other cryptographic primitives, and that hash function research is in fact still in its infancy. The use of hash functions as part of cryptographic schemes was suggested as early as 1976 [DH76], and only now are these cryptographic primitives really coming under scrutiny.

This thesis

In this dissertation we collect the generic attacks on iterated hash functions and describe them in detail. In other words, we focus our attention on what can be done by an attacker who can find a collision in the compression function. The remainder of the thesis is organized as follows: In Chapter 2 we introduce the mathematical tools that will be used throughout the thesis. In Chapter 3 we formally define cryptographic hash functions and discuss the properties desirable in such hash functions, as well as the uses of hash functions in cryptography. In Chapter 4 we discuss the Merkle-Damgård construction for hash functions and describe MD4 [Riv92a] as an example of the use of the construction in practice. Chapter 4 also includes a description of our toy hash function, BABI. In Chapter 5 we describe attacks that are applicable to all hash functions, and in Chapter 6 we present the known generic attacks on iterated hash functions. Owing to these inherent weaknesses in the classical Merkle-Damgård construction, several variations of the construction have been suggested in the hope of improving its security. In Chapter 7 we describe some of these Merkle-Damgård variants and discuss the generic attacks on these constructions. We conclude in Chapter 8.

Chapter 2

Probability Theory and Mathematical Tools

This chapter collects results from probability theory, as well as other relevant branches of mathematics, that will be important to the generic attacks described in Chapters 5, 6 and 7.

2.1 Obtaining a particular result in k trials

Suppose we have an urn containing m balls labeled 1 to m. We draw k balls out of the urn at random and with replacement. What is the probability of drawing a particular ball? (This is equivalent to the probability of throwing a particular number at least once if we roll an m-sided dice k times). The probability of drawing that particular ball each time is $\frac{1}{m}$, so the probability of *not* drawing it in k attempts is

$$q(k) = \left(1 - \frac{1}{m}\right)^k,$$

and hence the probability of drawing it at least once is

$$p(k) = 1 - q(k) = 1 - \left(1 - \frac{1}{m}\right)^k.$$

We now consider the Taylor expansion of e^x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

If $|x|$ is small, then the terms $\frac{x^2}{2!}$, $\frac{x^3}{3!}$, ... become negligible and

$$e^x \approx 1 + x. \tag{2.1}$$

So if $k \ll m$ we can replace $1 - \frac{1}{m}$ by $e^{-\frac{1}{m}}$ (we need m to be large and k to be small in order to use the Taylor approximation, so that we are not approximating too many times) to get

$$p(k) \approx 1 - e^{-\frac{k}{m}}.$$

In particular, if $k = m$ (i.e., the number of trials is the same as the number of balls in the urn) we get

$$p(k) \approx 1 - e^{-1} \approx 63\%.$$

This is relevant for example when we consider the probability of obtaining a particular n-bit hash value (out of the $m = 2^n$ possible hash values) by hashing $k = 2^n$ random messages. Alternatively, we can find the value of k for which the probability of drawing a particular ball is p: then $e^{-\frac{k}{m}} \approx 1 - p$, so $-\frac{k}{m} \approx \ln 1 - p$, or

$$k \approx m \ln \left(\frac{1}{1-p} \right).$$

So for a success probability of $\frac{1}{2}$ we need about $m \ln 2 = 0.69m$ trials.

2.2 The birthday problem

The birthday problem can be stated as follows: Within a group of k people, what is the probability that two or more will share a birthday? For $k = 23$, two people will share the same birthday with probability just greater than $\frac{1}{2}$. This appears to be somewhat counter-intuitive[1], but within a group of twenty-three people there are $\binom{23}{2} = 253$ unordered pairs, each serving as a suitable candidate for a match. It now seems more likely that of the 365 days available in a year, at least one of the 253 pairs could possibly be two people sharing the same birth-date.

[1]Due to the counter-intuitive nature of this result, the birthday problem is also commonly known as the *birthday paradox*. Note that this is not a "paradox" in the mathematical sense of the word.

When computing the probability that in a room full of k people at least two have the same birthday, we disregard any factors that may cause variations in the distribution, such as leap years, and we assume that all 365 possible birth-dates are equally likely.

We start by calculating the probability that all k birthdays are different, i.e. that no two people share the same birthday. We call this probability $q(k)$. The first person could have any of the 365 possible birthdays. The probability that the second person has a different birthday to the first is $\left(1 - \frac{1}{365}\right)$. The probability that the third person has a different birthday to the first and second is $\left(1 - \frac{2}{365}\right)$. We continue in a similar fashion and if $k \leq 365$, then

$$q(k) = \prod_{i=0}^{k-1} \left(1 - \frac{i}{365}\right).$$

Thus, the probability that at least two people share the same birth-date is

$$p(k) = 1 - q(k) = 1 - \prod_{i=0}^{k-1} \left(1 - \frac{i}{365}\right).$$

For $k = 23$, $p(k)$ is roughly 50.7%.

Using approximations

We now generalize the setting and rephrase the problem in terms of urns and balls: Consider an urn containing m balls labeled 1 to m. We draw k balls out of the urn with replacement. What is the probability of obtaining a "collision", i.e., of drawing the same ball twice? An argument similar to the one above shows that this probability is given by

$$p(k) = 1 - q(k) = 1 - \prod_{i=0}^{k-1} \left(1 - \frac{i}{m}\right).$$

Now,

$$\begin{aligned} p(k) &= 1 - \prod_{i=0}^{k-1} \left(1 - \frac{i}{m}\right) \\ &= 1 - 1\left(1 - \frac{1}{m}\right)\left(1 - \frac{2}{m}\right)\ldots\left(1 - \frac{k-1}{m}\right) \end{aligned}$$

and if $k \ll m$ (i.e., $\frac{k}{m}$ is small), we can again use equation (2.1) to replace each $\left(1 - \frac{i}{m}\right)$ by $e^{\frac{-i}{m}}$ to obtain

$$\begin{aligned} p(k) &\approx 1 - e^{\frac{-1}{m}} \cdot e^{\frac{-2}{m}} \ldots e^{\frac{-(k-1)}{m}} \\ &= 1 - e^{\frac{-k(k-1)}{2m}} \\ &\approx 1 - e^{\frac{-k^2}{2m}}. \end{aligned} \tag{2.2}$$

When $k = \sqrt{m}$, this is $1 - e^{-\frac{1}{2}} \approx 39\%$.

Putting it another way, we get a collision with probability p when $e^{\frac{-k^2}{2m}} = 1 - p$, i.e., when the number of balls drawn is

$$k \approx \sqrt{2 \ln \left(\frac{1}{1-p}\right) m}. \tag{2.3}$$

So the collision probability is $\frac{1}{2}$ when $k = \sqrt{2 \ln 2 \cdot m} = 1.18\sqrt{m}$. That is, if we draw $1.18\sqrt{m}$ balls (with replacement) out of an urn containing m balls, we have a 50% chance of drawing the same ball at least twice. In the case of the birthday problem, we set $m = 365$. Hence,

$$k \approx 1.2\sqrt{365} \approx 22.93 \approx 23.$$

This is relevant for example when we consider the probability of obtaining the same n-bit hash value twice when we hash $k = 2^{n/2}$ random messages.

2.3 Multicollisions

We now consider the probability of drawing the same ball r times in k tries. Equivalently, suppose there are k people in a group and m possible birthdays.

We are interested in the probability that some r people have the same birthday (we will call this event an *r-collision*). The details are now more complicated, but it turns out that if $k = m^{(r-1)/r} = \frac{m}{m^{1/r}}$, then there is a high probability of at least r people having the same birthday; we give a heuristic argument for this claim.

There are $\binom{k}{r} = \frac{k(k-1)\cdots(k-r+1)}{r!}$ unordered subsets of r people in the room, each of which has a probability $(\frac{1}{m})^{r-1}$ of being an r-collision, i.e., of all r people having the same birthday. If these probabilities were independent, then the probability of there being at least one r-collision would be

$$p(k,r) = 1 - \left(1 - \frac{1}{m^{r-1}}\right)^{\binom{k}{r}}.$$

If $k \ll m$ then by equation 2.2 this is approximately $1 - e^{-\frac{1}{m^{r-1}}\binom{k}{r}}$, which indeed reduces to our earlier birthday problem approximation of $1 - e^{\frac{-k(k-1)}{2m}}$ if $r = 2$.

Now note that if $r \ll k$ (i.e., the number of people in the multicollision we are looking for is much smaller than the number of people in the room) then $\binom{k}{r} \approx \frac{k^r}{r!}$, and the probability of an r-collision is

$$p(k,r) \approx 1 - e^{-\frac{k^r}{r!m^{r-1}}}. \tag{2.4}$$

If $k = m^{(r-1)/r}$, then equation (2.4) gives

$$p(m^{(r-1)/r}, r) \approx 1 - e^{\frac{-1}{r!}}. \tag{2.5}$$

Alternatively, we can find the value of k that results in an r-collision with probability p:

$$1 - p \approx e^{\frac{-k^r}{r!m^{r-1}}}$$

implies

$$-\frac{k^r}{r!m^{r-1}} \approx \ln(1-p),$$

and hence

$$k \approx \left(r! \ln\left(\frac{1}{1-p}\right)\right)^{1/r} \cdot m^{(r-1)/r}.$$

In particular, an r-collision occurs with probability $p = \frac{1}{2}$ if the number of people is

$$k \approx (r! \ln 2)^{1/r} \cdot m^{(r-1)/r}. \tag{2.6}$$

(Remember, however, that this argument is heuristic, since the probabilities of each subset of r people all having the same birthday are not really independent.)

2.4 The birthday problem for collisions between two sets

We now consider the problem of obtaining a shared birthday between two different groups of people, as opposed to obtaining a shared birthday within one group of people. Consider a group of k_1 men and a group of k_2 women. What is the probability of at least one man and one woman sharing a birthday? (Shared birthdays within each gender group do not concern us). Again, we reformulate the problem in terms of urns and balls: Consider two urns, each containing m balls numbered from 1 to m. We randomly draw k_1 balls out of the first urn and k_2 balls out of the second urn, each time recording the number. What is the probability of the same number occurring in both lists? As pointed out by Nishimura and Sibuya in [NS90] there are actually three distinct cases to consider: the case in which both sets of balls are drawn without replacement, the case in which one set of balls is drawn without replacement and the other is drawn with replacement, and the case in which both sets of balls are drawn with replacement. We refer to these three cases as Model A, Model B and Model C respectively (Model C corresponds to the scenario involving men and women given above).

2.4.1 Model A

We consider the case in which balls are drawn from both urns without replacement (i.e., we are assured that there are no collisions *within* the two sets of drawn balls). Let $q(k_1, k_2)$ be the probability that there are no matches between the sets of balls drawn from the first and second urns. Since k_1 balls have been drawn from the

first urn without replacement, the probability that the first ball drawn from the second urn is different to all the balls in the first set is $1 - \frac{k_1}{m}$ (because of the m possible numbers that can be drawn, k_1 are bad). Since the balls are also drawn without replacement from the second urn, the probability that the second ball drawn from urn two is different to any of the balls in the first set is $1 - \frac{k_1}{m-1}$ (because there are $m-1$ possibilities for the ball drawn out of the second urn, k_1 of which are bad). Continuing, we obtain

$$q(k_1, k_2) = \prod_{i=0}^{k_2-1} \left(1 - \frac{k_1}{m-i}\right).$$

Rearranging and dividing numerator and denominator by m gives

$$q(k_1, k_2) = \prod_{i=0}^{k_2-1} \left(\frac{m-i-k_1}{m-i}\right) = \frac{\prod_{i=0}^{k_2-1}\left(1 - \frac{k_1+i}{m}\right)}{\prod_{i=0}^{k_2-1}\left(1 - \frac{i}{m}\right)}.$$

Thus,

$$\ln q(k_1, k_2) = \sum_{i=0}^{k_2} \ln\left(1 - \frac{k_1+i}{m}\right) - \sum_{i=0}^{k_2} \ln\left(1 - \frac{i}{m}\right).$$

We now consider the Taylor expansion of $\ln(1-x)$:

$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \ldots\ldots \tag{2.7}$$

If $|x|$ is small, then the terms $\frac{x^2}{2}$, $\frac{x^3}{3}$, ... become negligible and

$$\ln(1-x) \approx -x. \tag{2.8}$$

It follows that if $k_1 + k_2 \ll m$ then[2]

$$\ln q(k_1, k_2) \approx \sum_{i=0}^{k_2} \left(-\frac{k_1+i}{m}\right) - \sum_{i=0}^{k_2} \left(-\frac{i}{m}\right) = \sum_{i=0}^{k_2} \left(-\frac{k_1}{m}\right) = -\frac{k_1 k_2}{m}.$$

Hence

$$q(k_1, k_2) \approx e^{-\frac{k_1 k_2}{m}},$$

[2]The value of k_1 has to be significantly smaller than m for us to replace each $\ln\left(1 - \frac{i}{m}\right)$ by $\frac{-i}{m}$, and k_2 has to be significantly smaller than m or the cumulative effect of approximating k_2 times would be too great.

and the probability of there being a match between the first and second set of drawn balls is approximately $1 - e^{-\frac{k_1 k_2}{m}}$. When $k_1 = k_2 = \sqrt{m}$ this is $1 - e^{-1} = 63\%$.

Putting it another way, we get a collision with probability p when $e^{-\frac{k_1 k_2}{m}} = 1 - p$, i.e., when

$$k_1 k_2 = m \ln \left(\frac{1}{1-p} \right).$$

So the collision probability is $\frac{1}{2}$ when $k_1 k_2 = m \ln 2 \approx 0.69m$.

2.4.2 Model B

This is a mixed model in which the balls are drawn without replacement from the first urn, and with replacement from the second urn. The balls drawn out of the first urn comprise k_1 of m possible balls. Since each ball is drawn out of the second urn with replacement, each has an independent probability $\frac{k_1}{m}$ of being identical to a ball drawn out of the first urn. Since k_2 balls are drawn from the second urn, the probability of no balls in the first set being identical to any in the second is

$$q(k_1, k_2) = \left(1 - \frac{k_1}{m} \right)^{k_2}.$$

Using equation (2.1), we replace $1 - x$ with e^{-x}, and if $k_1 + k_2 \ll m$, then

$$q(k_1, k_2) \approx e^{\frac{-k_1 k_2}{m}}.$$

Thus the probability of finding a match between the two sets of drawn balls is again approximately $1 - e^{\frac{-k_1 k_2}{m}}$.

2.4.3 Model C

We now consider the case in which balls are drawn from both urns with replacement, i.e., collisions within both sets of balls drawn are possible. Model C will be of use to us when considering generic attacks on hash functions in later chapters.

This case is more complicated than the previous two cases (because of the possibility of collisions within both sets, which makes them effectively smaller when considering collisions between them), but we can work out an exact expression for the collision probability. Let $S_2(n,t)$ denote the number of ways in which an n-element set can be partitioned into t nonempty subsets (this is called a Stirling number of the second kind, see [Pre93] and [GKP98]). We first note that if T_1 is the number of distinct balls drawn out of the first urn in k_1 tries, then

$$\text{Prob}[T_1 = t_1] = \frac{1}{m^{k_1}} S_2(k_1, t_1) \cdot m(m-1)\cdots(m-t_1+1). \tag{2.9}$$

(To see this, note that T_1 is analogous to the number of distinct birthdays among the men. There are $S_2(k_1,t_1)$ ways in which k_1 men can be divided into t_1 subsets who share the same birthday, and there are $m(m-1)\cdots(m-t_1+1)$ ways in which distinct birthdays can be assigned to these t_1 subsets. Hence in $S_2(k_1,t_1)\cdot m(m-1)\cdots(m-t_1+1)$ of the possible m^{k_1} outcomes, the number of distinct birthdays is t_1.) A similar expression holds for the second urn. It follows that the probability of there being no collision between the two sets of balls is

$$q(k_1,k_2) = \frac{1}{m^{k_1+k_2}} \sum_{t_1=1}^{k_1} \sum_{t_2=1}^{k_2} S_2(k_1,t_1) S_2(k_2,t_2) \cdot m(m-1)\cdots(m-t_1+t_2+1). \tag{2.10}$$

This is because the probability of t_1 distinct balls being drawn from the first urn in k_1 tries is $\frac{1}{m^{k_1}} S_2(k_1,t_1)\cdot m(m-1)\cdots(m-t_1+1)$, the probability of t_2 distinct balls being drawn from the second urn in k_2 tries is $\frac{1}{m^{k_2}} S_2(k_2,t_2)\cdot m(m-1)\cdots(m-t_2+1)$, and the probability that none of the t_2 balls from the second urn has the same number as any of the t_1 balls from the first urn is $\left(\frac{m-t_1}{m}\right)\left(\frac{m-1-t_1}{m-1}\right)\cdots\left(\frac{m-t_2+1-t_1}{m-t_2+1}\right)$.

Returning now to the problem at hand, the probability of at least one collision occurring between the two sets is $p(k_1,k_2) = 1 - q(k_1,k_2)$. The asymptotic analysis is much harder here than in the other cases, but it is shown in [NS88] that for large values of m and $k_1 k_2 \approx m$ we again have

$$p(k) \approx 1 - e^{-\frac{k_1 k_2}{m}}. \tag{2.11}$$

Of course, it is not surprising that if $k_1 + k_2 \ll m$ then all three cases result in the same approximation — the probability of a significant number of collisions

occurring within either of the sets becomes negligible, so whether we are sampling with or without replacement becomes irrelevant.

2.5 The geometric distribution

We briefly introduce the geometric distribution.

Definition 2.5.1. *We define a Bernoulli trial to be an experiment that can result in only one of two outcomes, 'success' or 'failure'.*

Definition 2.5.2. *The geometric distribution is a discrete probability distribution describing the number of Bernoulli trials needed to obtain one success.*

If p represents the probability of success in each trial then the probability of the first success occurring on the kth trial is

$$\begin{aligned} p(k) &= \text{Prob}(k-1 \text{ failures and then one success}) \\ &= (1-p)^{k-1}p. \end{aligned}$$

In the chapters to come, we will be concerned with how many trials, on average, we need to consider before we obtain a 'success'. This is given by the expected value of a geometrically distributed random variable k, denoted $E(K)$. We derive the formula for $E(K)$:

$$E(K) = \sum_{j=1}^{\infty} jp(j) = \sum_{j=1}^{\infty} j(1-p)^{j-1}p.$$

We note that the series $\sum_{j=1}^{\infty} jx^{j-1}$ is the derivative with respect to x of the series $\sum_{j=1}^{\infty} x^j$, and therefore converges to $\frac{d}{dx}\left(\frac{x}{1-x}\right) = \frac{1}{(1-x)^2}$ if $|x| < 1$. We therefore have

$$E(K) = p\left(\frac{1}{(1-(1-p))^2}\right) = \frac{1}{p}.$$

Consider the following example: Suppose that in a set of ten balls, four are black and six are white. Assume that all of the balls are placed in an opaque urn and that it is impossible to tell white balls from black balls whilst the balls are in the urn. Suppose we want to know how many balls, on average, we need to pull out

of the urn before we obtain a black ball.

The probability p of a randomly drawn ball being black is $4/10 = 0.4$. The number of trials before drawing a black ball follows a geometric distribution. We therefore expect, on average, to draw

$$\frac{1}{p} = \frac{1}{0.4} = 2.5$$

balls before obtaining a black ball.

2.6 Random functions and cycles

A *random function* $f : D \to R$ maps an element of the domain, D, to a randomly selected element of the range, R. A truly random function essentially needs to be described by a look-up table — if it had a short description then it wouldn't be random — which is inefficient if the domain is large. So in practice one uses *pseudo-random* functions. These are functions which emulate random functions; that is, even though they have a short description their outputs "look random". In this section we briefly discuss the behaviour of a pseudo-random function iterated multiple times (i.e., the function is iterated with itself).

Let $f : S \to S$ be a pseudo-random function on a finite set S of cardinality m. For an initial value $x_0 \in S$ we define a sequence of elements of S by

$$x_i = f(x_{i-1}),$$

for $i = 1, 2, \ldots$. Since S is a finite set, the sequence of x_i's must eventually start to cycle. By the birthday paradox argument, the expected number of inputs that need to be considered before this happens is $O(\sqrt{m})$ (see Section 2.2). The rho method derives its name from the fact that the shape of the path 'walked' through the sequence resembles the Greek letter rho. In figure 2.1 the tail (the section of the sequence before the cycle) has length two, and the cycle has length six. (In later chapters we use μ and λ to denote the cycle length and tail length respectively). If we can find the start of the cycle, i.e., values i and j such that

$f^i(x) = f^j(x) = w$, say, but $f^{i-1}(x) \neq f^{j-1}(x)$, we have two different preimages for w and hence have found a collision.

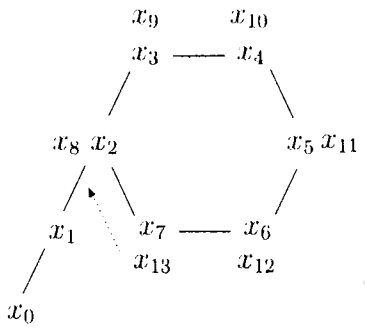


Figure 2.1: Cycle of length six with a tail of length two

2.7 Sequences and partial Orders

In this section we present definitions and concepts from [NS04], to be used in Chapter 7. We stick mainly to the notation used by the authors of the said publications.

Definition 2.7.1. *Let $\alpha = \langle \alpha_1, \alpha_2, \ldots, \alpha_s \rangle$ be a fixed, finite sequence of symbols such that each symbol is an element of the symbol set $\mathcal{S} = \{1, 2, \ldots, l\}$, and such that each symbol in $\mathcal{S}$ occurs at least once in α.*

The length of α is s and is denoted $|\alpha|$. For $1 \leq i \leq j \leq |\alpha|$, we let $\alpha[i, j]$ denote the subsequence of consecutive terms $\langle \alpha_i, \alpha_{i+1}, \ldots, \alpha_j \rangle$, and we call this an interval of the sequence α. A sequence interval of the form $\alpha[1, j]$ is known as an initial interval.

A relation $\prec$ on $\mathcal{S}$, associated with the sequence α, is defined as follows:

Definition 2.7.2. *For $x, y \in \mathcal{S}$, $x \prec y$ if every occurrence of x precedes every occurrence of y in α.*

The relation above is antisymmetric and transitive and thus is a *partial order*. Two elements x and y of $\mathcal{S}$ are said to be *incomparable* if $x \not\prec y$ and $y \not\prec x$. A list of symbols $x_1, x_2, \ldots, x_t$ such that $x_1 \prec x_2 \prec \cdots \prec x_t$ is known as a *chain* of length t. A set of chains is a *chain decompostion of* X if the chains are disjoint and their union is X (i.e., if every element of X appears in exactly one of the chains). There is a classical result called Dilworth's Theorem which applies to any partial order:

Theorem 2.7.1. *Suppose that $\prec$ is a partial order on a finite set X. Then the maximum number of mutually incomparable elements in X is equal to the minimum number of chains in any chain decomposition of $(X, \prec)$.*

We denote the length of the longest chain in α by $\text{maxchain}(\alpha)$. Note that if there are t elements which occur exactly once in α, then $\text{maxchain}(\alpha) \geq t$.

For a symbol $x \in \mathcal{S}$, we define the frequency of x to be the number of times x appears in the sequence α, i.e.,

$$\text{freq}(x) = |\{i : \alpha_i = x\}|.$$

We define the *frequency of the sequence* α to be

$$\text{freq}(\alpha) = \max\{\text{freq}(x) : x \in \mathcal{S}\},$$

i.e., the number of occurrences of the most popular symbol in α.

We now give some examples to help illustrate the notions discussed above. Some of these sequences will be referred to again in Chapter 7.

Example 1. Let $\Psi^{(1,l)} = \langle 1, 2, \ldots, l \rangle$. In this example we have the chain $1 \prec 2 \prec \cdots \prec l$, and $\text{maxchain}(\Psi^{(1,l)}) = l$.

Example 2. Let $\Psi^{(2,l)} = \langle 1, 2, \ldots, l, 1, 2, \ldots, l \rangle$. There is no chain of length two in this sequence, and hence $\text{maxchain}(\Psi^{(2,l)}) = 1$.

Example 3. Let $\{\Theta^l : l = 1, 2, \ldots\}$ be a family of sequences defined iteratively in such a way that $\Theta^1 = \langle 1, 1\rangle$, $\Theta^2 = \langle 1, 2, 1, 2\rangle$ and Θ^{l+1} is obtained from Θ^l by replacing the final element l in Θ^l by $l+1, l, l+1$. (So Θ^l has length $2l$ and every element in $\{1, 2, \ldots, l\}$ appears twice.) For instance,

$$\begin{aligned}\Theta^3 &= \langle 1, 2, 1, 3, 2, 3\rangle,\\ \Theta^4 &= \langle 1, 2, 1, 3, 2, 4, 3, 4\rangle, \text{ and}\\ \Theta^5 &= \langle 1, 2, 1, 3, 2, 4, 3, 5, 4, 5\rangle.\end{aligned}$$

So with respect to the sequence Θ^l, the lists $1 \prec 3 \prec 5 \prec \cdots \prec l$ (if l is odd), or $1 \prec 3 \prec 5 \prec \cdots \prec l-1$ (if l is even) are chains (all occurrences of 1 appear before all occurrences of 3, which occur before all occurrences of 5, etc).
From the above it is evident that $\text{maxchain}(\Theta^l) \geq \lfloor \frac{l+1}{2} \rfloor$. In fact, $\text{maxchain}(\Theta^l) = \lfloor \frac{l+1}{2} \rfloor$. For any $\lfloor \frac{l+1}{2} \rfloor + 1$ elements from $\mathcal{S} = \{1, \ldots, l\}$, there appear two consecutive elements $i, i+1$ of $\mathcal{S}$ by application of the pigeonhole principle. But, due to the nature of the construction of Θ^l, it contains a subsequence $\langle i, i+1, i\rangle$, and thus the $\lfloor \frac{l+1}{2} \rfloor + 1$ elements cannot form a chain.

2.8 Square-free and abelian square-free sequences

In this section we introduce concepts and notation surrounding square-free words and sequences. We are particularly interested in abelian square-free sequences since they will be of use to us in Chapter 7. We stick mainly to the notation used by Rivest and Andreeva *et al.* in [Riv05] and [ABF+08] respectively.

A word ω is a sequence of letters, finite or infinite, over some finite alphabet $\mathcal{A}$. A word may be denoted $\omega = abc$ where a, b and c are elements of $\mathcal{A}$. If a word can be written in the form $\omega = xyz$ where x,y and z are words, with y nonempty, then x is known as a *prefix* of ω, z is known as a *suffix*, and y is known as a *factor* (or subword) of ω; the words x and z may be empty. We let $\text{Fact}_\alpha(l)$ denote the number of factors of length l occurring in the sequence α.

2.8.1 Square-free words and sequences

Definition 2.8.1. *A word ω is a square if it can be written in the form $\omega = xx$ with x nonempty and finite.*

A word ω is said to be *square-free* if it contains no repeated, adjacent letters or subwords. More formally:

Definition 2.8.2. *A word ω is square-free if it contains no factors of the form yy where y is finite and nonempty.*

In a sequence consisting of letters over a binary alphabet, one will not find square-free words of length greater than three (since we are not allowed repeated 0s or 1s, the only square-free words of length three are 010 and 101; but adding 1 or 0 respectively to these words gives a repeated subword 01 or 10). However, if the alphabet is increased by just one symbol, an infinite square-free sequence can be generated, namely,

$$\mu = 2102012101202102012021012102012101 20\ldots$$

This sequence was first exhibited by Alex Thue in 1906 [Ber95]. We will need infinite square-free sequences that can be generated using a small alphabet when we discuss dithered hash functions in Chapter 7.

2.8.2 Abelian square-free words and sequences

Definition 2.8.3. *A word ω is said to be an abelian square if it can be written in the form $\omega = xx'$ where x' is a permutation of x.*

Definition 2.8.4. *A word ω is abelian square-free if it contains no factors of the form yy' where y is finite and nonempty, and where y' is a permutation of y.*

The notion of being abelian square-free is stronger than that of being square-free. If a word or a sequence is abelian square-free, then it is automatically square-free since y' may simply be the result of the identity permutation on y. In 1970 Peter

Pleasants [Ple70] proved that infinite abelian square-free words exist over a five-letter alphabet, and in 1992 Veikko Keränen [Ker92] produced an infinite abelian square-free sequence over a four-letter alphabet, namely,

$$\kappa = \mathsf{abcacdcbcdcadcdbdabacabadbabcbdbcba}\ldots$$

For the remainder of this dissertation we refer to this sequence as the *Keränen sequence*. Further details relating to the generation of this sequence can be found in [Ker03] and [Riv05].

2.9 A note on big-oh notation and complexity

Owing to the fact that we make use of big-oh notation in some sections of this dissertation, we formally introduce it here:

Definition 2.9.1. *Let $f : \mathbb{R} \to \mathbb{R}$ be a function. For $n \in \mathbb{N}$ and for an arbitrary real constant c we define the following:*

1. $O(f) := \{g : \mathbb{R} \to \mathbb{R} | (\exists c)(\forall n) g(n) \leq cf(n) + c\}$.
   This is the class of all functions which grow more slowly than, or just as fast as, f. An equivalent definition yields $O(f) := \{g : \mathbb{R} \to \mathbb{R} | (\exists c)(\exists n_0)(\forall n \geq n_0) g(n) \leq cf(n)\}$.

2. $o(f) := \{g : \mathbb{R} \to \mathbb{R} | (\forall c)(\exists k)(\forall n \geq k) cg(n) + c \leq f(n)\}$.
   This is the class of functions which grow strictly more slowly than f.

3. $\Omega(f) := \{g : \mathbb{R} \to \mathbb{R} | f \in O(g)\}$.
   This is the class of functions which grow faster than, or just as fast as, f.

4. $\Theta(f) = O(f) \cap \Omega(f)$.
   This is the class of functions which have the same growth rate as f.

The most informative measure of the complexity of an algorithm is how long it runs, as a function of the size of the input in bits. In the case of hash functions, we usually define the time complexity as the number of calls to the the compression function. We are usually interested in the average-case complexity although it is also possible to consider the worst-case complexity.

Chapter 3

Hash Function Basics

In this chapter we define hash functions and describe the properties they need for security, and how they are used in cryptography.

3.1 What is a hash function?

A hash function is a function

$$H : \{0,1\}^{\infty} \rightarrow \{0,1\}^{n}$$

where $n \in \mathbb{N}$, that is, a function that compresses an input of arbitrary length to an output of fixed length[1]. We refer to the output of a hash function as a *hash value*, or simply just a *hash*.

Historically, the term *hash function* originates from the field of computer science. Cryptographic hash functions and conventional hash functions are related in the sense that they both map arbitrary-length inputs to fixed-length outputs. Cryptographic hash functions, however, are used in modern day cryptographic applications and therefore have to satisfy certain security properties, whereas conventional hash functions are used in non-cryptographic environments and do

[1] In practice, a hash function H will map from the domain $\{0,1\}^{m}$ for some large m, and not from $\{0,1\}^{\infty}$. This is not really of any consequence since m is larger than any message that we would realistically want to hash.

not need to be secure. Within the field of cryptography the term hash function has become synonymous with the term cryptographic hash function. We adopt this convention and it is assumed that all hash functions referred to are cryptographic hash functions.

Since the domain of a hash function is significantly larger than its range, it is inevitable that many inputs map to the same output. In other words, there always exist numerous collisions, but the point is that two such colliding inputs should be computationally infeasible to find. (Here 'computationally infeasible' could mean requiring super-polynomial effort, or requiring effort far exceeding available resources.)

The basic idea behind hash functions is that a hash value serves as a compact representative image of a specific input, and we want to be able to use this representative image as if it were uniquely identifiable with the corresponding input string. A practical illustration of this idea is a digital signature scheme. The message to be signed is first fed through a hash function and the resulting hash value is signed in place of the original message. Thus each signature is in fact a signature on an infinite number of messages; however, the collision-resistance property ensures that no one knows what the other messages are.

Cryptographic hash functions are often modelled using random oracles. A random oracle can be thought of as a mathematical function that maps every possible input (called a query) in its domain to a truly random output (called the response) in its range. A random oracle consistently responds with the same response for any specific query.

Hash functions can be split into two categories: Keyed hash functions and unkeyed hash functions. An unkeyed hash function takes in a single input, namely the message to be hashed, and a keyed hash function takes in two inputs — the message to be hashed and a secret value known as the key. In general, it is as-

sumed that the algorithmic specifications of hash functions are public knowledge. Since hash functions display the inherent characteristic of ease of computation, if the hash function is unkeyed then anyone can compute the hash value of a given input; for keyed hash functions only the users who know the key can compute the hash. The theory of keyed and unkeyed hash functions is quite dissimilar, and this dissertation will consider only unkeyed hash functions.

3.2 Hash function properties

A good hash function satisfies the following three major properties:

Property 1. ***Preimage resistance***
A hash function H is said to be preimage resistant if, given a hash value y, it is computationally infeasible to find an input x such that $H(x) = y$.

Property 2. ***Second preimage resistance***
A hash function H is second preimage resistant if, given an input value x, it is computationally infeasible to find an input x', with $x \neq x'$, such that $H(x) = H(x')$.

Property 3. ***Collision resistance***
A hash function H is collision resistant if it is computationally infeasible to find any two inputs x and x', with $x \neq x'$, such that $H(x) = H(x')$. (Such a pair (x, x') is known as a collision pair.)

The properties of preimage resistance, second preimage resistance and collision resistance are given in order of increasing strength: in other words, finding a collision is easier for an attacker than finding a second preimage, which is in general (although not always) easier than finding a preimage.

Theorem 3.2.1. *Collision resistance $\Rightarrow$ second preimage resistance.*

Proof. We prove the contrapositive: If H is not second preimage resistant, then for some x we can find an x', with $x \neq x'$, such that $H(x) = H(x')$. We then have a collision pair (x, x') and hence H is not collision resistant. □

Intuitively, we would expect it to be easier for an attacker to find a second preimage x' of $H(x)$, given x, than it is to find a preimage of $H(x)$ given no further information. For example, the hash function $H(x) = x^2 \bmod n$ where n is the product of two large primes p and q and the factorization of n is unknown, finding a preimage is computationally infeasible: we would need to be able to find a square root modulo n and this is computationally equivalent to factoring n, an intractable problem in itself. Finding a second preimage, however, is trivial: given x, $-x$ yields a collision.

The following argument illustrates that it is reasonable to assume that second preimage resistance implies preimage resistance for hash functions used in practice. Let H be an n-bit hash function, and recall that in practice, H takes inputs from a finite set of size 2^m for some large $m \gg n$. If H behaves like a random function, then for every possible hash value y there are about 2^{m-n} values in the domain of H that hash to y. If an attacker can find preimages for a significant fraction of possible hash values, then given $x \in \{0,1\}^m$ he simply computes $H(x) = y$ and then uses his preimage-finding algorithm to find a pre-image x' of y. Then with probability $\frac{(2^{m-n}-1)}{2^{m-n}} = 1 - \frac{2^n}{2^m}$ we have $x' \neq x$ and he has found a second preimage.

The above argument assumes that all possible hash values are equally likely. However, it is possible to construct hash functions for which this is not the case. We now present an example of a hash function that is second preimage resistant but not preimage resistant. Let g be a second preimage resistant hash function such that $g : \{0,1\}^\infty \to \{0,1\}^n$, and consider the hash function H defined as follows:

$$H(x) = \begin{cases} 0\|x & \text{if } x \text{ has bitlength } n, \\ 1\|g(x) & \text{otherwise.} \end{cases}$$

We now have a second preimage resistant hash function $H : \{0,1\}^\infty \to \{0,1\}^{n+1}$ that is not preimage resistant: For a hash value $H(x)$ with left-most bit 0, we know that $H(x)$ has the form $0\|x$. Thus by dropping the leading zero we have

the preimage, x.

In the above example, for hash values whose first bit is 0 there is precisely one preimage, so obviously finding a second preimage is impossible. However, this hash function is somewhat contrived and one would not expect to see such hash functions used in practice. In practice, collision resistance is the strongest of all three of the major hash function properties. It is the hardest to satisfy and the easiest to break. Once collision resistance has been breached in practice, so has second preimage resistance and preimage resistance.

An unkeyed hash function that satisfies preimage resistance and second preimage resistance is known as a *one way hash function (OWHF)*. A hash function that exhibits all three of the aforementioned properties is known as a *collision resistant hash function (CRHF)*. The figure below depicts a simple taxonomy of hash functions according to the three major hash function properties.

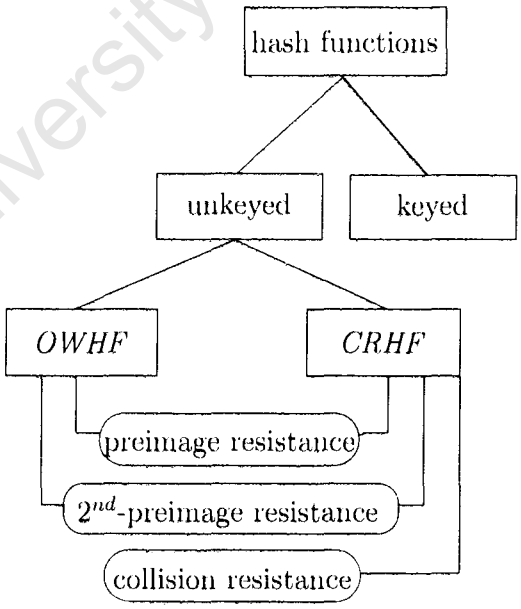


Figure 3.1: Taxonomy of hash functions

3.3 What are hash functions used for?

Originally, hash functions were designed for the purposes of providing *data integrity* and *data origin integrity*. Providing data integrity means ensuring that information has not been altered by unauthorized or unknown means. Providing data origin integrity means verifying the origin of information. Data origin integrity and data integrity cannot be separated. As soon as information has been altered, it effectively has a new source; and if a source cannot be verified, then the issue of integrity cannot be settled. Therefore, verification of integrity implicitly provides data origin authentication and vice versa.

Hash functions used for the purposes of data integrity and message authentication satisfy some, or all three, of the major hash function properties.

3.3.1 Data integrity and message authentication

Digital signatures

Data integrity can be established by using a hash function and a secure channel. Before Alice sends a message x to Bob, she uses a hash function to compute $H(x)$. She sends x to Bob over an unsecured channel and transmits $H(x)$ over a secure channel. Bob hashes the received data and compares his result to the hash result, $H(x)$, received from Alice. If the two are identical, Bob accepts that the information has not been altered by unauthorized or unknown means, i.e., data integrity is established. See Figure 3.2.

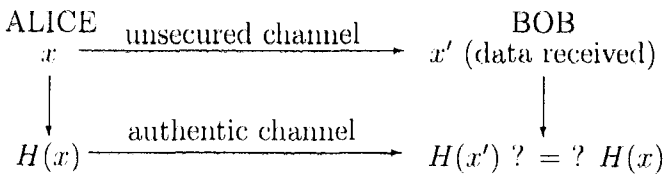


Figure 3.2: Data integrity via a hash function

The secure channel could be provided, for example, by a digital signature scheme. This is a mechanism consisting of a signature generation algorithm and a verification algorithm. The signature generation algorithm produces a digital signature, and the verification algorithm is a method for verifying the authenticity of a digital signature. In the description to follow we denote the signature generation algorithm by sig and the verification algorithm by ver.

Suppose Alice wants to send a signed message, x, to Bob. She first feeds x through a hash function H and obtains $H(x)$. Since $H(x)$ is usually considerably smaller than x, Alice signs $H(x)$ instead of x in order to save time. To sign $H(x)$ Alice computes $sig(H(x))$ by sending $H(x)$ through the signing algorithm. She then sends the pair $(x, sig(H(x)))$ to Bob. Since it is possible for the message to be corrupted during transmission, we denote the received message by x'. Bob receives the pair $(x', sig(H(x)))$, and applies the hash function to x'. He also feeds $sig(H(x))$ into the verification algorithm to obtain $ver(sig(H(x))) = H(x)$. If $H(x') = H(x)$, then Bob has successfully authenticated the signature and he can be assured that the message was sent by Alice. If $H(x) \neq H(x')$, then it possible that the message has been altered since signing, or that someone is trying to impersonate Alice. See Figure 3.3.

For the purposes of security we would like H to be collision resistant. Since Alice signs $H(x)$ instead of x, she essentially also signs all messages that hash to the same value as $H(x)$. If Alice can find a collision pair, (x, x'), then she can later claim she signed x' instead of x. Alternatively, if Bob can find a collision pair (x, x') such that Alice would be willing to sign x, he is later in a position to claim that she signed x'.

Due to the fact that hash functions are modelled on random oracles and behave like one-way functions, they are also used for purposes other than data integrity such as *confirmation of knowledge*, *key derivation* and *pseudorandom number generation*.

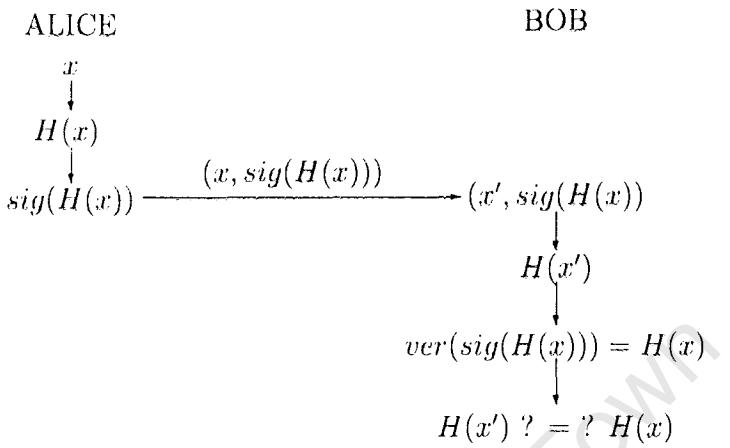


Figure 3.3: Digital signature scheme

3.3.2 Confirmation of knowledge

Mechanisms that employ hash functions for the purpose of confirmation of knowledge include *commitment schemes, password verification* and *trusted timestamping.*

Commitment schemes

A commitment scheme is a mechanism that allows a user to commit to a value while keeping it secret, and permits the user to reveal the value at a later stage. Consider for example an auction, where the parties submit their bids in sealed envelopes to the auctioneer who opens the envelopes and announces the highest bidder. If a bidder, Alice, is to pay her bid (as would be the case in a first-price auction), she is highly motivated to bid just slightly over the runner-up. An unscrupulous auctioneer, Eve, may leak the bids before the auction clears, allowing her accomplice, Alice, to overbid other participants by a minimal amount. One possible solution to this problem is to remove the trust from Eve by letting the parties commit to their bids without disclosing them.

A secure commitment scheme binds the committed party to a value without leaking any information to the auctioneer or to anyone else. Hash functions are

often suggested as a computationally efficient way of constructing commitment schemes: $H(x)$ acts as a commitment to x. A security argument may go along the following lines: If H is collision-resistant, the commitment can only be opened to x (otherwise the cheating party, Alice, is able to find a collision): if H is preimage resistant, recovering x from $H(x)$ is infeasible, which implies the hiding property. This argument as it stands is clearly inadequate. Preimage resistance is only guaranteed in the absence of any prior information, which is plainly false in the case of the input as structured as an auction bid. In particular, a curious adversary may go through all possibilities for the value of the bid, applying the hash function to them and comparing the result with the committed value. Secondly, Alice may just copy someone else's bid, which might be sufficient in some scenarios. However, both these problems can easily be avoided if the commitment scheme is randomised to prevent forward search, and if the protocol requires the inclusion of the committer's identity.

Password verification

In general, for security reasons, user passwords are not stored in plaintext. Instead, a hash value of the password is stored. Whenever a user provides a password, the hash value of the password provided is computed and compared to the stored hash value. If the two hash results are the same, the user is authenticated and access is granted. The hash function clearly has to be preimage resistant; otherwise if a malicious hacker, Eve, obtains the hash value $H(P)$ of Alice's password P, then she might be able to determine a password P' which has the same hash value, and hence would work just as well as P.

Trusted timestamping

Timestamping is the process of recording the time of creation or modification of information. *Trusted* timestamping refers to doing this in a secure manner. According to the widely accepted Internet X.509 Public Key Infrastructure Time-Stamp Protocol (RCF 3161) [ACPZ01], a trusted timestamp is one which is issued by a trusted third party (TTP) acting as a timestamping authority (TSA). The

following scenario motivates the need for timestamping.

Suppose that Alice has produced a medical cure x. Assume that there are many medical researchers working independently, and simultaneously, on this cure. It should not be possible for Alice to backdate the time of creation of her cure, and it should be possible later, when Bob also finds the cure, for her to prove that she did indeed know it by this time.

Alice computes the hash of x, $H(x)$. She sends this to a trusted timestamping authority. The TSA allocates this data a timestamp T_1, concatenates the timestamp to $H(x)$ and computes a hash of this concatenated result (T_1 denotes the version of the timestamp at time 1). This new hash value $H(H(x)\|T_1)$ is digitally signed by the TSA using its private key k. The signed hash value, $sig_k(H(H(x)\|T_1))$, together with the timestamp are sent back to Alice. Alice stores the cure, the signed hash value and the timestamp together in the same place. See Figure 3.4.

For trusted timestamping we require that the hash function H be preimage resistant. When Alice sends $H(x)$ to the TSA, for the sake of confidentiality, the TSA should not be able to obtain x from a knowledge of $H(x)$. Also, Bob can verify that Alice has not predated the creation of the cure and Alice is in a position to prove that she was indeed in possession of the cure at the time of timestamping. This is done as follows:

At the time of verification (some time after time 1) Alice reveals her alleged cure x' and her alleged time stamp T_1'. The hash of x' is calculated and the alleged timestamp is appended to $H(x')$. This result is then sent through the hash function to obtain $H(H(x')\|T_1')$. The digital signature of the TSA is checked by using the public key, k', of the TSA and sending $sig_k(H(H(x)\|T_1))$ through the verification algorithm of the digital signature scheme to obtain $ver_k'(sig_k(H(H(x)\|T_1))) = H(H(x)\|T_1)$. If $H(H(x')\|T_1')) = H(H(x)\|T_1)$, then it is accepted that the timestamp has not been altered and that it was issued by the timestamping authority,

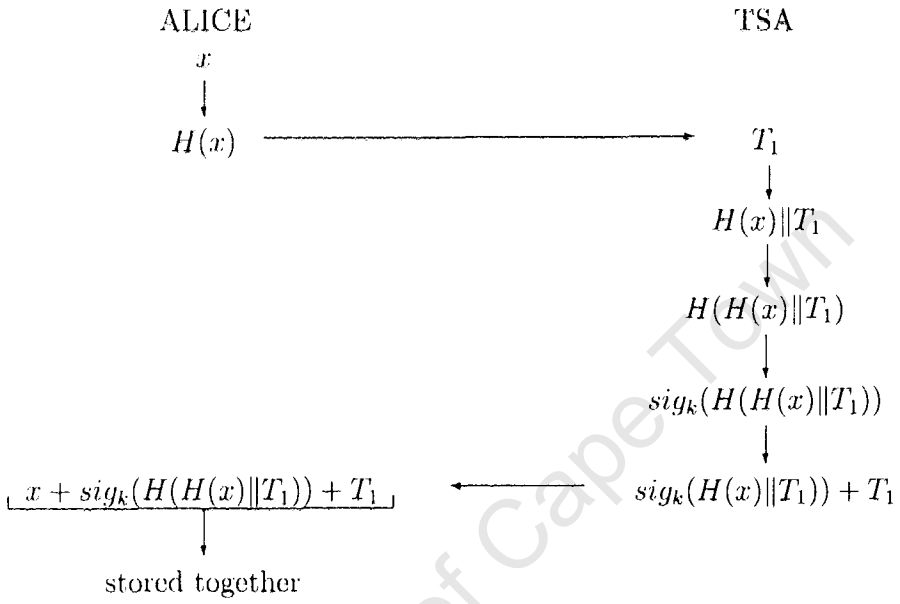


Figure 3.4: Issuing a timestamp

and also that x is correct. If the two hash results are not equal, then it is possible that the timestamp has been altered, or that it was not genuinely signed by the TSA. Figure 3.5 depicts this process.

3.3.3 Key derivation

A *key derivation function* is a function which derives one or more secret keys from an unknown value, or possibly from a piece of known information such as a password or passphrase. An example of such a function is *PBKDF1* (*Password-Based Key Derivation Function 1*), which forms part of the RSA Laboratories' Password-Based Cryptography Standard [Kal00]. *PBKDF1* makes use of a hash function and a cryptographic *salt*. This salt is made up of random bits that are added to a password or passphrase before the password or passphrase is sent through a hash function. The derived key (DK) is obtained by applying the hash

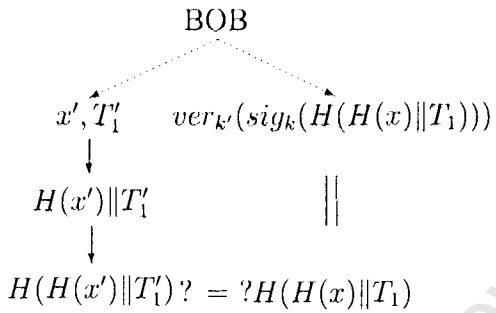


Figure 3.5: Checking the timestamp

function for c iterations to the concatenation of the password P and the salt S.

$$\begin{aligned} k_1 &= H(P\|S) \\ k_2 &= H(k_1) \\ &\vdots \\ k_c &= H(k_{c-1}) \end{aligned}$$

$$DK = len(k_c)$$

The function *len* extracts the first t bits of k_c to obtain a key of length t. The specification of *PBKDF1* allows for it to produce a key with maximum length of 160 bits. *PBKDF2* [Kal00] allows for the production of larger keys. *PBKDF2* makes use of a pseudorandom function such as a hash function or a block cipher to derive the key. The pseudorandom function given as an example in the RSA Laboratories' Password-Based Cryptography Specification document is a pseudorandom function based on a hash function [Kal00].

3.3.4 Pseudorandom number generation

A pseudorandom number generator generates sequences of numbers which satisfy various statistical properties of randomness. Since hash functions are modeled

on random oracles, it is possible to generate pseudorandom numbers by using a hash function. This is done, for example, by applying a hash function to a counter. For $c_1 = \textit{initial value}$, we compute $H(c_1) = a_1$, then $H(c_2) = a_2$ and so on. It is hoped that the sequence $a_1, a_2, \ldots$ is a sequence of pseudorandom numbers. In order for this generating mechanism to be cryptographically secure (by which we mean that given part of the sequence it is not possible to predict the next bit), the initial value of the counter must be both random and secret. Recommendations for random number generation using hash functions from the SHA family can be found in [BK07].

Chapter 4

The Merkle-Damgård Construction for Hash Functions

According to the definition provided in Chapter 3, a hash function needs to be able to accept a string of arbitrary length as input and to output a string of fixed length. The generic construction currently used for most popular hash functions was first described by Ralph Merkle [Mer79] in 1979. In 1989 Ralph Merkle and Ivan Damgård [Dr89] independently proved the construction to be secure under certain assumptions, and hence today we refer to it as the Merkle-Damgård construction for hash functions.

4.1 Definitions and notation

Before the details of the construction are introduced, we present a few definitions and the notation to be used:

Definition 4.1.1. *A compression function is a function that maps an m-bit string to an n-bit string, where $m > n$. More formally, a compression function is a function*

$$f : \{0,1\}^n \times \{0,1\}^l \rightarrow \{0,1\}^n,$$

where $m = n + l$.

Note. The compression function can either be described as taking in two inputs, one of length n bits and one of length l bits, or as taking one input of length $m = n+l$. We will find it convenient to switch loosely between these two descriptions. In other words, for an n-bit value x, an l-bit value y and a compression function f, we will consider feeding the two inputs x and y into f as being identical to feeding the single input $x\|y$ into f; i.e., we consider $f(x, y)$ to be the same as $f(x\|y)$.

We now introduce the notation that will be used throughout the remainder of this dissertation:

F, G, H	hash functions
f, g, h	compression functions of F, G and H respectively
h_0	initialization vector (IV)
m_i	the i^{th} message block, $i \in \mathbb{N}$
h_i	the i^{th} chaining value, $i \in \mathbb{N}$
0^t	the binary string consisting of t zeros
$\|x\|$	bit-length of x
$x\|\|y$	concatenation of x and y
$x + y$	addition modulo 2^{32} of x and y
$x \wedge y$	bitwise 'and' of x and y
$x \vee y$	bitwise 'or' of x and y
$x \oplus y$	bitwise 'xor' of x and y
$\neg x$	bitwise complement of x
$x <<<_s$	rotation of x left by s bits

4.2 The Merkle-Damgård construction

Iterated hash functions operate on messages of arbitrary length by iteratively processing fixed-length pieces of the message to be hashed. We refer to these pieces as *message blocks*, and denote their fixed length by l. Assume we have a

hash function $F : \{0,1\}^{\infty} \rightarrow \{0,1\}^n$. Let $f : \{0,1\}^n \times \{0,1\}^l \rightarrow \{0,1\}^n$ be the compression function of F. If we wish to hash a message of arbitrary length, M, we follow the procedure outlined below. Assume for now that the length of M is a multiple of l.

1. Break M up into k message blocks of length l. We denote these message blocks $m_1, m_2, \ldots, m_{k-1}, m_k$.

2. Set h_0 to some pre-specified n-bit value called the *initialization vector*, or IV.

3. For i from 1 to k let $h_i = f(h_{i-1}, m_i)$.

4. Set $F(M) = h_k$.

For a graphical representation of this process see Figure 4.1. The h_i are called chaining values; the output from the compression function f in round $i-1$ becomes one of the inputs to f in round i. The second input to f in round i is the i^{th} message block, m_i. Also, note that $F(m_1 \| m_2 \| \ldots \| m_{k-1} \| m_k) = f(F(m_1 \| m_2 \| \ldots \| m_{k-1}) \| m_k)$.

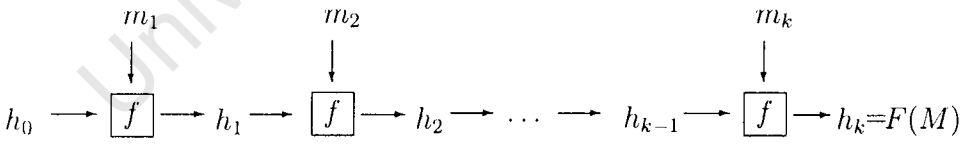


Figure 4.1: Iterative hashing process

In the event that $|M|$ is not a multiple of l, we append extra bits to the end of M to ensure that message blocks of the correct size will be fed into the compression function of F. This is called *padding*. One possible way of padding a message is to append as few 0-bits as possible to the end of the message such that the length of the augmented message is a multiple of l. However, this method is ambiguous — it is not possible to pinpoint where a message trailing with 0-bits ends and where padding begins. This can be avoided by first appending a single

1-bit to the message and thereafter appending the required number of 0-bits. (So if the message length is an integral multiple of l, we will add one block of padding).

After padding has been done, a single message block containing the length of the original message is often appended to the augmented message. This is known as Merkle-Damgård strengthening. The bits representing the message length are usually right justified in the final block. If the length field is larger in bit-size than l, it may be spread over multiple message blocks. The final block is added for security reasons, because it foils a second preimage finding attack known as the *long message attack* (described in Chapter 6). The procedure for hashing an arbitrary-length message M now reads as follows:

1. Pad message M until $|M|$ is a multiple of l.
2. Break the augmented message up into k blocks of length l. We denote the message blocks $m_1, m_2, \ldots, m_{k-1}, m_k$.
3. Append the message block m_{k+1} containing the length of m (right justified).
4. Set h_0 to some pre-specified value IV.
5. For i from 1 to $k+1$ let $h_i = f(h_{i-1}, m_i)$.
6. Set $F(M) = h_{k+1}$.

Figure 4.2 depicts the process outlined above.

A note on initialization vectors

Initialisation vectors could be fixed or randomly chosen, but in either case it is important that the IV is a known value, and that the same IV be used when recomputing a hash result.

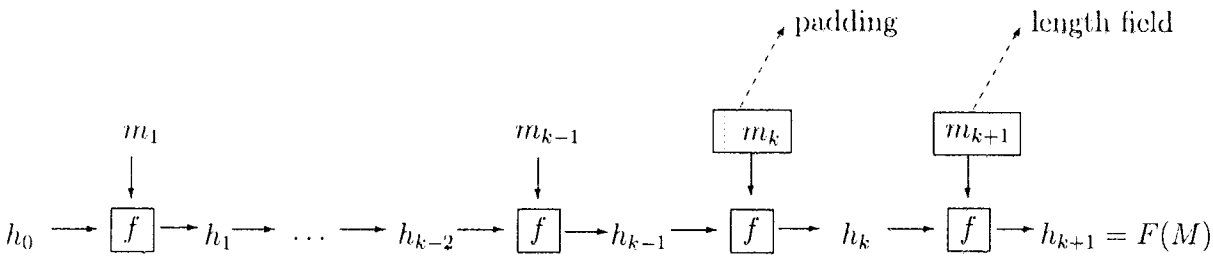


Figure 4.2: Iterative hashing process with padding and Merkle-Damgård strengthening

4.3 Merkle's meta method and Damgård's construction

We now present the constructions suggested by Merkle [Mer79] and Damgård [Dr89]. We then provide a general security proof for the construction mentioned in Section 4.2, and adapt this proof for both the Merkle and Damgård constructions.

4.3.1 Merkle's meta method

The method most commonly used to construct modern-day hash functions (refer to Section 4.2) is largely based on Merkle's Meta Method [Mer79]. His method for hashing an arbitrary length message M is as follows:

For a given compression function f from m bits to n bits, we set the IV to 0^n, pad the message if necessary, and obtain $F(M)$ by following the hashing processes mentioned in the previous section.

4.3.2 Damgård's construction

Damgård's construction is slightly more complicated than the construction provided by Merkle. Also, Damgård's original definitions are phrased in terms of collision resistant *families* of functions[1]. We follow Merkle in working with a

[1] A fixed size hash function family $\mathcal{F}$ is an infinite family of finite sets $\{F_m\}_{m=1}^{\infty}$, and a function $t : \mathbb{N} \rightarrow \mathbb{N}$ such that $t(m) < m$ for all $m \in \mathbb{N}$. For any given $m \in \mathbb{N}$, a member of F_m

single iterated hash function and its compression function. For a compression function f from m bits to n bits, where $m = n + l$ and $l > 1$, Damgård's construction reads as follows:

1. Break the message, M, up into blocks of size $l - 1$ bits. If the last block is incomplete, pad it with zeros.

2. Let d be the number of zeros needed, and let L be the length of the message with padding. We then have message blocks $m_1, m_2, \ldots, m_k$, where $k = \frac{L}{l-1}$.

3. Append an extra $(l - 1)$-bit block m_{k+1} containing the binary representation of d, right justified. (Note that here we are appending the number of padding zeroes, not the entire message length.)

4. For i from 2 to $k + 1$ let

   $h_1 = f(0^{n+1} \| m_1)$, and
   $h_i = f(h_{i-1} \| 1 \| m_i)$.

5. Set $F(M) = h_{k+1}$.

The construction given above is slightly more complicated than the one mentioned previously. The reason for the additional 0 or 1 in the middle of the input to f in Step 4 will become clear in the discussion of the security proof in the next section.

4.3.3 A security proof

We now provide a security proof for the general construction described in Section 4.2. We first consider the construction without Merkle-Damgård strengthening, and prove the following lemma.

Lemma 4.3.1. *Let F be an iterated hash function with compression function f. Assume we can find a collision under the hash function F (without Merkle-Damgård strengthening). If the colliding messages have the same number of*

is a function $f : \{0,1\}^m \to \{0,1\}^{t(m)}$; in other words, F_m is the set of all functions from m bits to $t(m)$ bits [Dr89].

blocks, then we can find a collision under the compression function f. Otherwise we can find either a collision under f or a preimage of the IV under F.

Proof. Assume we have found M and M', with $M \neq M'$, such that $F(M) = F(M')$. Let t denote the number of message blocks in M, and let t' denote the number of message blocks in M', where without loss of generality $t' \geq t$. Since $F(M) = F(M')$ we have

$$F(m_1 \| \ldots \| m_t) = F(m'_1 \| \ldots \| m'_{t'})$$

and thus

$$f(F(m_1 \| \ldots \| m_{t-1}), m_t) = f(F(m'_1 \| \ldots \| m'_{t'-1}), m'_{t'}). \tag{4.1}$$

If either $m_t \neq m'_{t'}$ or $F(m_1 \| \ldots \| m_{t-1}) \neq F(m'_1 \| \ldots \| m'_{t'-1})$ then we have found a collision under f and we are done. Otherwise, if $F(m_1 \| \ldots \| m_{t-1}) = F(m'_1 \| \ldots \| m'_{t'-1})$ and $m_t = m'_{t'}$, then we consider

$$f(F(m_1 \| \ldots \| m_{t-2}), m_{t-1}) = f(F(m'_1 \| \ldots \| m'_{t'-2}), m'_{t'-1}),$$

and repeat the argument. The process will come to a halt either when we find a collision under f or when

$$F(m_1) = F(m'_1 \| \ldots \| m'_{t'-t+1}).$$

If $t' = t$, then this means $m_1 \neq m'_1$ (else we would have $M = M'$) and

$$f(IV, m_1) = f(IV, m'_1)$$

so we have found a collision under f. If $t' > t$, then we have

$$f(IV, m_1) = f(F(m'_1 \| \ldots \| m'_{t'-t}), m'_{t'-t+1}), \tag{4.2}$$

so either we have found a collision under f, or $m_1 = m'_{t'-t+1}$ and

$$IV = F(m'_1 \| \ldots \| m'_{t'-t})$$

and we have found a pre-image of the IV under F. □

We are now in a position to give a security proof for both Merkle's and Damgård's construction.

Theorem 4.3.2. *Let F be an iterated hash function with compression function f. If f is collision resistant, then so is F.*

Proof. Assume that we have found M and M', with $M \neq M'$, such that $F(M) = F(M')$. If the lengths of M and M' are the same then a straightforward extension of Lemma 4.3.1 to the case with Merkle-Damgård strengthening gives us a collision under f. So suppose the message lengths are different.

In the case of Merkle's method, the appending of the length field ensures that the final message blocks of M and M' are different, so equation 4.1 gives us a collision under f. In Damgård's construction, equation 4.2 is replaced by

$$f(0^{n+1}\|m_1) = f(F(m'_1\|\ldots\|m'_{t'-t})\|1\|m_{t'-t+1}). \tag{4.3}$$

The fact that the first message block m_1 is processed differently from the others in Damgard's construction prevents the inputs to f being the same on the left hand side and right hand side: clearly $0^{n+1}\|m_1 \neq F(m'_1\|\ldots\|m'_{t'-t})\|1\|m_{t'-t+1}$ because of the 0 or 1 sandwiched between the two inputs of the compression function. Thus we have found a collision under f, and we are done. □

4.4 A more specific classification

Staying within the Merkle-Damgård paradigm, we now briefly discuss three methods for constructing compression functions. These include: hash functions based on block ciphers, customized hash functions and hash functions based on modular arithmetic.

4.4.1 Hash functions based on block ciphers

An advantage of building hash functions using block ciphers is that many efficient block cipher implementations already exist. Some block-cipher-based constructions are provably secure assuming the underlying block ciphers exhibit certain ideal properties. However, in reality block ciphers are not random functions and often exhibit irregularities and weaknesses when used in practice. It is difficult

to say what properties of block ciphers are needed in practice in order to build secure hash functions, and whether or not good block ciphers guarantee good hash functions.

Definition 4.4.1. *A block cipher is a map* $E : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ *such that for* $k \in \{0,1\}^\kappa$ *and for* $x \in \{0,1\}^n$, *the function* $E_k(x) = E(k,x)$ *is a permutation on* $\{0,1\}^n$. *We say that the* n*-bit plaintext* x *is encrypted under the* κ*-bit key* k. *For a block cipher* E, *there exists* E^{-1} *such that* $E_k^{-1}(y)$ *is a plaintext* x' *for which* $E_k(x') = y$.

The Matyas-Meyer-Oseas, Davies-Meyer and Miyaguchi-Preneel hash functions are all hash functions based on block ciphers. The figure below depicts a single iteration of the Davies-Meyer compression function; the other two hash functions are similar. Each scheme makes use of the following:

1. A generic n-bit block cipher E_k;
2. A function g that maps n-bit inputs to keys, k, for E (note that k will be public); and
3. A fixed IV h_0 that can be used in E.

In the Davies-Meyer construction each intermediate chaining value h_i is obtained by using the current message block to encrypt the previous chaining value h_{i-1}, and then XORing again with this value to prevent inverting the compression function. In other words,

$$h_i = E_{g(m_i)}(h_{i-1}) \oplus h_{i-1}.$$

4.4.2 Customized hash functions

Customized hash functions are designed from scratch so as not to be constrained by the use of existing system components such as block ciphers. They are designed with a view to optimizing performance.

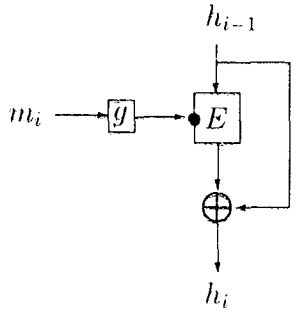


Figure 4.3: The Davies-Meyer compression function

Examples of such functions are those based on the MD4 hash function. This function was invented by Ronald Rivest [Riv92a] in 1990. Owing to security concerns, many versions and adaptations have been invented since, including MD5 [Riv92b], SHA-0 [NIS93] and SHA-1 [NIS95].

We now provide a detailed description of the MD4 algorithm. We include it not only as an example of a customized hash function, but also as an example of the use of the Merkle-Damgård construction in practice.

MD4

The MD4 hash function consists of three rounds of sixteen operations each. Before we describe the hashing process, we introduce the three boolean functions f^*, g^* and h^* that are to be used in rounds one, two and three respectively. Each function takes three 32-bit words as input and produces one 32-bit word as output. We let X, Y and Z denote 32-bit words. The functions are defined as follows:

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

Assume we wish to hash a message of arbitrary length, M. We do the following:

1. **Pad the message and append the length field.**

   The MD4 hash function processes 512-bit message blocks. When padding, we have to allow for the last 64 bits of the augmented message to contain the length[2] of M. Thus, we pad M until it is congruent to 448 mod 512. We employ 'unambiguous' padding, i.e., we first append a single 1-bit and thereafter append the required number of 0-bits. Once this is done, we append a 64-bit binary representation of $|M|$. The length of the augmented message is now a multiple of 512. Note that it is also a multiple of sixteen 32-bit words. We let $W_0, \ldots, W_{L-1}$ denote the 32-bit words comprising the augmented message.

2. **Set the IV**

   A four-word buffer (A, B, C, D) is used throughout the hashing process. The final message digest will be the concatenation of the 32-bit words in the registers A, B, C and D following the last computation of the compression function. We initialize these registers with the following hexadecimal values (in little-endian) to obtain h_0:

$$\begin{aligned} A &:= 67452301 \\ B &:= EFCDAB89 \\ C &:= 98BADCFE \\ D &:= 10325476 \end{aligned}$$

3. **Process the message blocks**

   For each 512-bit message block, we do the following:

   for $i = 0$ **to** $(L/16) - 1$ **do**

   for $j = 0$ **to** 15 **do**

   $X_j = W_{16i+j}$

[2]We thus never hash a message of more than $2^{64} - 1$ bits.

$A' := A$

$B' := B$

$C' := C$

$D' := D$

ROUND 1

Let $[abcd\ k\ s]$ denote the operation of leaving b, c, d, k and s unchanged and changing a as follows:

$$a = (a + f^*(b, c, d) + X_k) <<<_s$$

We perform the following sixteen operations (from left to right):

$[ABCD\ 0\ 3]$	$[DABC\ 1\ 7]$	$[CDAB\ 2\ 11]$	$[BCDA\ 3\ 19]$
$[ABCD\ 4\ 3]$	$[DABC\ 5\ 7]$	$[CDAB\ 6\ 11]$	$[BCDA\ 7\ 19]$
$[ABCD\ 8\ 3]$	$[DABC\ 9\ 7]$	$[CDAB\ 10\ 11]$	$[BCDA\ 11\ 19]$
$[ABCD\ 12\ 3]$	$[DABC\ 13\ 7]$	$[CDAB\ 14\ 11]$	$[BCDA\ 15\ 19]$

ROUND 2

Let $[abcd\ k\ s]$ be defined as follows:

$$a = (a + g^*(b, c, d) + X_k + 5A827999) <<<_s$$

We perform the following sixteen operations (from left to right):

$[ABCD\ 0\ 3]$	$[DABC\ 4\ 5]$	$[CDAB\ 8\ 9]$	$[BCDA\ 12\ 13]$
$[ABCD\ 1\ 3]$	$[DABC\ 5\ 5]$	$[CDAB\ 9\ 9]$	$[BCDA\ 13\ 13]$
$[ABCD\ 2\ 3]$	$[DABC\ 6\ 5]$	$[CDAB\ 10\ 9]$	$[BCDA\ 14\ 13]$
$[ABCD\ 3\ 3]$	$[DABC\ 7\ 5]$	$[CDAB\ 11\ 9]$	$[BCDA\ 15\ 13]$

ROUND 3

Let $[abcd\ k\ s]$ be defined as follows:

$$a = (a + h^*(b, c, d) + X_k + 6ED9EBA1) <<<_s$$

We perform the following sixteen operations (from left to right):

$[ABCD\ 0\ 3]$ $[DABC\ 8\ 9]$ $[CDAB\ 4\ 11]$ $[BCDA\ 12\ 15]$
$[ABCD\ 2\ 3]$ $[DABC\ 10\ 9]$ $[CDAB\ 6\ 11]$ $[BCDA\ 14\ 15]$
$[ABCD\ 1\ 3]$ $[DABC\ 9\ 9]$ $[CDAB\ 5\ 11]$ $[BCDA\ 13\ 15]$
$[ABCD\ 3\ 3]$ $[DABC\ 11\ 9]$ $[CDAB\ 7\ 11]$ $[BCDA\ 15\ 15]$

$A := A + A'$
$B := B + B'$
$C := C + C'$
$D := D + D'$

4. **Output the message digest**

   The final output reads as follows: $MD4(M) = A\|B\|C\|D$. This will be an output of size 128 bits.

The diagram below is a schematic representation of the MD4 hashing process. We denote the length of the augmented message (including padding and the length field) by $|M'|$. The number of 512-bit message blocks is thus $|M'|/512$, denoted l. According to the algorithm above, $h_0 = (A, B, C, D)$ and $h_i = (A + A', B + B', C + C', D + D')$.

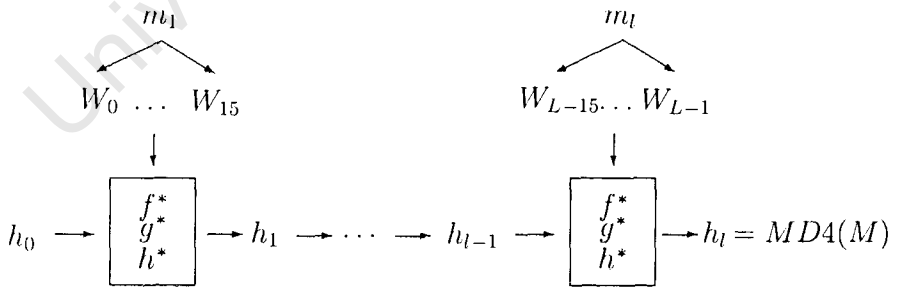


Figure 4.4: The MD4 hashing process

4.4.3 Hash functions based on modular arithmetic

Hash functions based on modular arithmetic are iterated hash functions in which the compression function is made up of operations involving modulo N arithmetic.

Two advantageous aspects of such hash functions include: (i) the ability to scale N and therefore the size of the hash so as to obtain a desired security level, and (ii) the availability of existing software and components for modular arithmetic from other cryptographic protocols. However, hash functions based on modular arithmetic do not include speed as a favourable performance feature and are therefore not commonly used in practice.

4.5 BABI: A toy hash function

All practical illustrations of the generic attacks mentioned in Chapters 5 and 6 will make use of the hash function described below. It has been *Built As a Basic Instance* and will be referred to as BABI from this point onwards. The practical illustrations to come are intended to be as simple and as easy to follow as possible. For this reason, BABI has been built to accommodate only messages of short length, and has not been built to be secure. Due to the nature of many of the attacks in Chapters 5 and 6, it will be important to be able to easily find collisions on the compression function of BABI for the purposes of illustration.

BABI is based on modular arithmetic and has been built according to the specifications suggested by Ivan Damgård in [Dr89]. The suggestion for a secure hash function to be built using modular arithmetic is not surprising given the hardness of extracting roots modulo N, where N is the product of two large primes. Damgård's construction reads as follows:

Let N be the product of two large primes, p and q. Let the length of N be u bits. We choose a proper subset I such that $I \subseteq \{1, 2, \ldots, u\}$. For any u-bit string $b_1, b_2, \ldots, b_u$ we define the function f_I to be the concatenation of all b_i such that $i \in I$.

We define the compression function f, from m bits to n bits, to be the func-

tion

$$f(h_{i-1}, m_i) = f_I((BUFFER \| h_{i-1} \| m_i)^2 \bmod N).$$

The size of each chaining variable h_{i-1} is n bits, the size of each message block m_i is t bits, and hence $m = n + t$. The input variable $BUFFER$ represents the sequence of bits required to ensure that the input to f is greater than $\sqrt{N}$ but less than $N/2$. This helps avoid trivial collisions of the form $x^2 \equiv (-x)^2 \bmod N$, and ensures that modular reduction always takes place, thus foiling attempts to find collisions of small magnitude.

In order to ensure that f is both secure and efficient, the subset I has to be chosen with some care. In order to avoid an attack by Marc Girault [Gir88], I should specify bit positions that are spread evenly over all u possible positions. However, choosing random bit positions does not optimize efficiency. Allowing for positions that are grouped together makes more sense when keeping efficiency in mind. Also, if $|I|$ is too small, then the set of possible hash values has a maximum size of $2^{|I|}$ and thus an attack applicable to all hash functions known as the birthday attack[3] becomes computationally feasible.

BABI has been built according to the design principle outlined above, but due to the fact that it need not be secure and should be easy to implement, certain conditions have been relaxed. We present the algorithm below. See Appendix A for the JAVA source code. In what follows we let $|m|$ represent the length of the message blocks and $|h|$ the length of the chaining values.

1. **Determine N**

   Two small primes, p and q, are selected such that $N = pq$. BABI allows for the user to select p and q. The primes should be selected to be small in order to aid ease of computation and to make it possible to find collisions in the compression function (since we want the compression function to be easy to break). BABI has a built-in primality test for p and q.

[3]This attack is described in detail in Chapter 5.

2. **Determine the buffer, $|m|$ and $|h|$**

   The value of $BUFFER$ is set to $\lceil\sqrt{N}\rceil$ and we set the auxiliary variable $INTERVAL$ to $\lfloor\frac{N}{2}\rfloor - \lceil\sqrt{N}\rceil$. We let the bit length of the input to the compression function, l, be defined such that $l = bitlength(INTERVAL) - 1$. We let the length of the message blocks be $|m| = \lceil\frac{l}{2}\rceil + 1$ and the length of the chaining values be $|h| = \lfloor\frac{l}{2}\rfloor$ -1; i.e., the message block length will always be larger than the length of the chaining values.

3. **Set the IV**

   We set the IV to $\mathbf{0}$. It is possible to set the IV to any constant of the correct size, say a randomly generated value perhaps.

4. **Split the message into blocks and pad the message**

   As the message comes in, it is split up into blocks of the specified size. All message blocks are stored and processed when necessary. This is admittedly not the most efficient way of computing a hash value; ideally, message blocks should be processed as they are read in. If the message length is a multiple of the message block length, then no padding is added. If the message is not a multiple of $|m|$, then it is padded with a 1 followed by the required number of zeros.

5. **Append the length field**

   The length field is set to be a multiple of the message block length, i.e., $t|m|$ bits are set aside to contain the binary representation of the length of M for some integer t. The code in Appendix A specifies t to be 4. The length representation of M is right justified in this length field. The reason why the length field is set to be a multiple of $|m|$ is to ensure that no additional padding needs to be sandwiched between the partially augmented message and the last $t|m|$ bits. We acknowledge that this is not ideal since messages of longer length can only be accommodated when larger primes are used.

6. **Process the message blocks**

   The overall size of the input to BABI is a multiple of $|m|$. We let k denote the total number of message blocks. For a message block m and a chaining

value h, we define the compression function of BABI to be

$$f(h, m) = f_l((BUFFER + h \| m)^2 \bmod N),$$

where f_l selects the last $|h|$ bits of $((BUFFER + h\|m)^2 \bmod N)$. For i from 1 to k, message blocks are processed as follows

$$h_i = f(h_{i-1}, m_i).$$

7. Output the hash value

We output $BABI(M) = h_k$.

The input to modular squaring in the compression function of BABI is slightly different from the idea suggested by Damgård. However, it still ensures that this particular bit of input is greater than $\sqrt{N}$ and less than $N/2$.

Security was not an important design goal when creating BABI (although it does contain certain security enhancers such as Merkle-Damgård strengthening and the elimination of trivial roots). BABI has largely been built to be relatively insecure so as to be able to easily implement and demonstrate some of the generic attacks in Chapters 5 and 6.

Chapter 5

Generic Attacks on Hash functions

In this chapter we collect and describe attacks that are applicable to all hash functions. These attacks would work even if the hash function in question was a random oracle.

5.1 The birthday attack

The birthday attack is a collision-finding attack: the attacker hopes to find two distinct inputs, M and M', that hash to the same value under some hash function F. The attack was pointed out by Yuval [Yuv79], who justified it by the birthday problem (described in Chapter 2), although Nishimura and Sibuya [NS88] were the first to give a proper mathematical analysis of it in terms of the birthday problem between two sets.

5.1.1 Finding a collision in one set of messages

Suppose that F is an n-bit hash function, and consider a set of $k \ll 2^n$ messages, chosen at random. Assuming all 2^n hash values are equally likely, recall from Chapter 2 that there is a collision in the set with probability

$$p(k) \approx 1 - e^{\frac{-k^2}{2n}}, \tag{2.2}$$

where here $m = 2^n$ is the number of possible hash values. This probability is $\frac{1}{2}$ when

$$k \approx \sqrt{2 \ln 2}\sqrt{m} \approx 1.2 \cdot 2^{n/2}. \tag{2.3}$$

So if an attacker computes the hash values of $1.2 \cdot 2^{n/2}$ distinct messages and searches for two hash values that are the same, he has a 50% chance of producing a collision.

We now consider the expected running time of the birthday attack. A quick upper bound for this expected time is given by the geometric distribution (see Chapter 2): computing the hash values of $2^{n/2}$ messages gives a collision with probability

$$p(2^{n/2}) \approx 1 - e^{-\frac{(2^{n/2})^2}{2 \cdot 2^n}} = 1 - e^{-\frac{1}{2}} \approx 0.4, \tag{5.1}$$

so the expected number of times the attacker would need to run this attack in order for it to be successful is $\frac{1}{0.4} = 2.5$. This would mean computing the hash of $2.5 \cdot 2^{n/2}$ messages (and the number of calls to the compression function would of course depend on the length of the messages). This analysis assumes the attacker computes $2^{n/2}$ values, looks for a collision, and then throws away those values and starts again if there is none. Suppose instead that the attacker computes hash values of random messages until he produces a collision: how long do we expect this to take? The expected number of messages needed is

$$\begin{aligned} &\sum_{j=1}^{\infty} j \cdot \text{Prob(first collision occurs when } k = j) \\ &= \sum_{j=1}^{\infty} j \cdot (p(j) - p(j-1)), \end{aligned}$$

and a careful analysis (Fact 2.27 in [MvOV96], or see [Knu68], section 1.2.11.3) shows that this is approximately

$$\sqrt{\frac{\pi m}{2}} \approx 1.25 \cdot 2^{n/2}.$$

Often in practice the constants 1.25 or 2.5 are ignored, and the average-case work for the birthday attack is taken to be $O(2^{n/2})$ applications of F. (Note that the

worst-case work would be given by the pigeon-hole principle: the attacker can only be assured of finding a collision in $2^n + 1$ attempts.)

The birthday attack is unavoidable; it is applicable to all hash functions (it would even work if F was a truly random function, or a black box). If it is the best known attack on the collision resistance of a particular hash function F, then F is said to have *ideal security* as far as collision resistance is concerned. In general, we speak about F as having a *security level*[1] of $2^{n/2}$.

Example 5.1.1. We now apply the birthday attack to BABI. We ran the attack three times; each time increasing the value of the primes used. All messages are given in hexadecimal. Colliding pairs are indicated in bold.

Attack 1: We let $p = 71$ and we let $q = 73$. This results in an output of length 4 bits. We tried the following $2^{4/2} = 4$ messages

M_{1_1} = 5468652063617420736174206f6e20746865206d617420616e6420626974206d652e

$\mathbf{M_{1_2}}$ = 546865206f6c64206d616e20696e2074686520736561206e6420686974206f782e

M_{1_3} = 4f6c6420646f672072617420696e20746865206c696520616e6420686974206f782e

$\mathbf{M_{1_4}}$ = 43616c2063756c206c6174206574206865772066f726207468612074697320742e,

and find that M_{1_2} and M_{1_4} have the same hash value b.

Attack 2: We let $p = 211$ and we let $q = 281$. This results in an output of

[1] We say that F has ideal security in terms of preimage resistance and second preimage resistance if the best known attack on both properties is a brute force attack, i.e., hashing random messages until we find one with the desired hash — we have a 63% chance of finding one within 2^n tries. In both cases, the security level is 2^n.

length 6 bits. We tried the following $2^{6/2} = 8$ messages

$$
\begin{aligned}
M_{2_1} &= \texttt{2e2020202020202020686f6d726f6d6f6d726f736f6d726f64756d626f636f6d626f} \\
M_{2_2} &= \texttt{6c696d626f6e696d626f636f6d626f72696d626f73696d626f20202020202020202e} \\
M_{2_3} &= \texttt{2e20202020202020206e6f6e7365626c7565736a617a7a79636f757273656c617465} \\
\mathbf{M_{2_4}} &= \texttt{2e202020206c696d626f636f6d626f6b696d626f6e696d626f726f6d626f20202020} \\
M_{2_5} &= \texttt{2e2020202020202020626561647361726d737362616e6473686f706573686f6d6573} \\
M_{2_6} &= \texttt{20202020686f706573686f75765686f7273652e20202020626c7565736a617a7a79} \\
M_{2_7} &= \texttt{5468652063617420736174206f6e20746865206d617420616e6420626974206d652e} \\
\mathbf{M_{2_8}} &= \texttt{20202020202020202e636f6d626f6c696d626f6c696d6d6f7368696d6f6c6f6d6d75},
\end{aligned}
$$

and find that M_{2_4} and M_{2_8} have the same hash value 3b.

Attack 3: We let $p = 839$ and we let $q = 929$. This results in an output of

length 8 bits. We tried the following $2^{8/2} = 16$ messages

$$
\begin{aligned}
M_{3_1} &= 5468652063617420736174206f6e20746865206d617420616e6420626974206d652e \\
M_{3_2} &= 546865206f6c64206d616e20696e2074686520736561206e64206869742206f782e \\
M_{3_3} &= 202020202e62616c6c736d616c6c7374616c6c7363616c6c7320202020736e616b65 \\
\mathbf{M_{3_4}} &= 7261626974686f6269746c6f6269746b6f6269746e6f6d626f2e2020202020202020 \\
M_{3_5} &= 6c696d626f2062696d626f206a696d626f206b696d626f2072696d626f202020202e \\
M_{3_6} &= 67697a6d6f74696e6b6572696d626f2e6c6f6d626f636f6d626f2020202020202020 \\
M_{3_7} &= 546865206f6c6420646f6720696e2074686520736561206e64206c69742069742e \\
M_{3_8} &= 202020202e20202020686f757365686f7273656c6f7573656d6f7573656c696d626f \\
M_{3_9} &= 202020202e686f7273656b696d626f6d6f757365736e616b65736e69706520202020 \\
M_{3_{10}} &= 62696d626f6c696d626f686f7273656272616b6520202020202020202e6e65696768 \\
M_{3_{11}} &= 68616c6c7363616c6c7362616c6c736d616c6c732e2020202020202020686f6c6473 \\
M_{3_{12}} &= 736e616b65202020202e2020202068616b65736d616b65736c696b6573736d696c65 \\
M_{3_{13}} &= 686f6c6473636f6c6473726f6c64736d6f6c64732e2020202020202020686f727365 \\
M_{3_{14}} &= 6c6164656c686164656c726164656c6d6164656c202020202e202020206269726473 \\
M_{3_{15}} &= 70757070796c757070796d75707079637570707 92e2020202020202020206875707079 \\
\mathbf{M_{3_{16}}} &= 2020202074696d626f6e696d626f722e6f6d626f636f6d626f6c696d626f2022020,
\end{aligned}
$$

and find that M_{3_1} and $M_{3_{16}}$ have the same hash value 58.

The table below summarizes our results. Note that small primes have been used for the purposes of illustration.

p	q	N	n	$2^{n/2}$	M	M'	hash value (bits)	hash value (hex)
71	73	5183	4	4	M_{2_2}	M_{2_4}	1011	b
211	281	59291	6	8	M_{3_4}	M_{3_8}	111011	3b
839	929	779431	8	16	M_{4_4}	$M_{4_{16}}$	01011000	58

5.1.2 Finding a collision between two sets of messages

It is important to observe how the birthday attack can be used in practice. If a dishonest signer, Alice, could find a collision between a 'good' message M_1 and a 'bad' (or fraudulent) message M_2, then she is in the position to provide a signature on M_1, and then later claim that she signed M_2 instead. A collision of this nature is of use to Alice because recall that a digital signature scheme signs the hash of a message, and not the message itself (see Section 3.3.1).

For the attack to work, Alice would need to obtain a collision between a set of good messages and a set of bad messages. Recall from Section 2.4 that if the size of both sets of messages is k, then the probability of obtaining a collision between the two sets is $p(k) = 1 - e^{-\frac{k^2}{m}}$. So, for an n-bit hash function F, if we compute the hash values of $2^{n/2}$ good messages and $2^{n/2}$ bad messages, then we will have a probability of $1 - e^{-1} \approx 0.63$ of finding a collision.

To obtain two colliding messages, Alice chooses a good message M_1, and a bad message M_2, and then does the following:

Step 1 She generates $2^{n/2}$ minor modifications of M_1, denoted M_1'. A minor modification could include substituting tab characters for spaces in text files, or perhaps substituting unprintable characters for each other.

Step 2 She hashes each of the $2^{n/2}$ modified messages and stores each result with its associated message.

Step 3 She generates minor modifications M_2' of M_2. As she generates these messages, she computes the hash of each, $F(M_2')$, and checks for any matches between $F(M_2')$ and the hash values computed in Step 2. She keeps going until she finds a match. She expects to find a match within about $2^{n/2}$ candidate messages M_2'.

Assuming that Alice finds M_1^* and M_2^* such that $F(M_1^*) = F(M_2^*)$, she can now provide a signature for the modified good message M_1^* by signing $F(M_1^*)$, and

then later claim that this is a signature for M_2^* since $F(M_1^*) = F(M_2^*)$ and hence the signatures are the same. The work for this attack is about $2 \times 2^{n/2}$ times the length of the messages. (It is comprised of the $2^{n/2}$ hash values that need to be computed in Step 2, and the $2^{n/2}$ hash computations that need to be done in Step 3, each of which involves one call to the compression function for every block of the message.)

5.2 Memoryless techniques for finding collisions

The obvious way to find a collision using the birthday attack is to keep a list of messages and their hash values, and each time we compute a new hash value to check the list to see if it has occurred before; this has a storage requirement of $O(2^{n/2})$. In this section we discuss two refinements of the birthday attack which use much less storage. Both are based on Pollard's *Rho* method[2] [Pol75]. The method essentially works by producing, instead of a sequence of random values of $F(x)$, a sequence of values which "looks" random but which has the advantage that once a value repeats, the sequence will continue to repeat from there.

5.2.1 Pollard's Rho Method

Let F be an n-bit hash function, and consider a restricted version of F (i.e. F with a restricted domain), denoted F_r and defined by

$$F_r : \{1, 2, \ldots, 2^n\} \rightarrow \{1, 2, \ldots, 2^n\}.$$

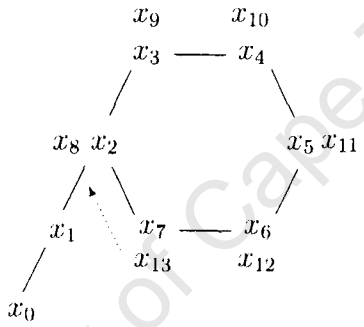
We observe that F_r can be modelled as a pseudo-random function on the set of all n-bit blocks. If we select an n-bit block x_0 at random and consider the sequence

$$x_i = F_r(x_{i-1}),$$

for $i = 1, 2, \ldots$, we obtain a pseudo-random sequence which, since $\{0,1\}^n$ is a finite set, must eventually start to cycle (see Chapter 2, Section 2.6). By the

[2]This method was introduced by J.Pollard in 1975 as a technique for factoring composites of small primes, but its underlying principle is applicable to collision searches.

birthday paradox argument, the expected number of inputs that need to be considered before this happens is about $\sqrt{\frac{\pi m}{2}} = 1.25 \cdot 2^{n/2}$ (see Section 5.1.1). The Rho method derives its name from the fact that the shape of the path 'walked' through the sequence resembles the Greek letter rho. In what follows, we let μ denote the length of the cycle, and we let λ denote the length of the tail (the part of the sequence before the cycle). In figure 2.1 the tail has length two, and the cycle has length six.



If we can find the start of the cycle, i.e., values λ and μ such that $F_r^{\lambda}(x_0) = F_r^{\lambda+\mu}(x_0) = w$, say, but $F_r^{\lambda-1}(x_0) \neq F_r^{\lambda+\mu-1}(x_0)$ (since one is on the tail and the other is on the cycle), we have two different preimages for w and hence have found a collision. Obviously we could keep a list of $F_r^i(x)$ for $i = 0, 1, 2, \ldots$ until we find $F_r^j(x) = F_r^k(x)$ with $j \neq k$, but this would take up a lot of memory.

5.2.2 Floyd's cycle finding attack

In order to reduce the storage requirement needed to run the birthday attack, we can make use of Floyd's cycle-finding algorithm ([Flo67], or see [MvOV96], section 3.8). This algorithm works by running two instances of the sequence at the same time, one twice as fast as the other. In other words, for $i = 1, 2, \ldots$ the algorithm computes $F_r^i(x_0) = F_r(F_r^{i-1}(x_0))$ and $F_r^{2i}(x) = F_r(F_r(F_r^{2(i-1)}(x_0)))$, and compares them. If $F_r^i(x) \neq F_r^{2i}(x)$, then the next pair $(F_r^{i+1}(x), F_r^{2(i+1)}(x))$ is computed, and the previous pair can be discarded. The algorithm halts when it finds the smallest i for which $F_r^i(x) = F_r^{2i}(x)$, and records this pair.

Note that $F_r^i(x) = F_r^{2i}(x)$ if and only if $2i - i = i$ is a multiple of the cycle length μ and i is greater than or equal to the tail length λ. So the value of i found by the algorithm will be the lowest multiple of μ that is greater than or equal to λ, and hence $i \leq \lambda + \mu$. So although the method will probably not detect the cycle as soon as it starts (as the straightforward Rho method does), it will detect it within one cycle length of the start.

Now that we have i we need to find the start of the cycle, in order to obtain our collision $F_r(F_r^{\lambda-1}(x_0)) = F_r(F_r^{\lambda+\mu-1}(x_0))$, and remember that we cannot run the sequence backwards from $F^i(x_0)$ and $F^{2i}(x_0)$ since we cannot invert the hash function. So starting at the positions $x = F_r^0(x)$ and $F_r^i(x)$ in the sequence, we compute and compare $F_r^1(x)$ and $F_r^{i+1}(x)$. If these are not equal, we discard these values and compute and compare the next set values, $F_r^2(x)$ and $F_r^{i+2}(x)$. We continue in this fashion until we arrive at two values $F_r^s(x)$ and $F_r^{i+s}(x)$ such that $F_r^s(x) = F_r^{i+s}(x)$ for some $s \in \mathbb{N}$. We have now found the value of the tail length, $\lambda = s$, and we now know where the sequence starts to cycle.

This attack yields no improvement on the birthday attack in terms of time complexity, but the storage requirements are significantly reduced. On average $\sqrt{\frac{\pi}{2}} \cdot 2^{n/2}$ hash values need to be computed for each instance of the sequence before the collision is found. This keeps the time complexity of the attack at $O(2^{n/2})$. In terms of storage, however, only two values need to be stored at a time. This means a storage requirement of $O(1)$.

There is an improvement on Floyd's cycle finding algorithm by Richard Brent [Bre80], and some more efficient techniques such as the method by Sedgewick, Szymanski and Yao [SSY82].

5.2.3 The QD attack

In 1989 Jean-Jacques Quisquater and Jean-Paul Delescaille [QD89] proposed a different memoryless strategy for finding collisions under a hash function F. We adopt the convention of [Mit07] and refer to this as the QD attack. Paul van Oorschot and Michael Wiener [VOW99] refined the attack by making it both parallelisable and capable of producing meaningful collisions[3]. As with the attack above, this attack yields no improvement on the birthday attack in terms of time complexity, but the storage requirements are significantly reduced.

Quisquater and Delescaille's cycle detection algorithm makes use of *distinguished points*. An n-bit block x_d is considered to be a distinguished point if it satisfies some predefined distinguishing criterion. A criterion such as this could be, for instance, requiring that the s leftmost bits of a distinguished block are all zero. Mitchell [Mit07] suggests that a block be distinguished if the leftmost $n/2 - s$ bits of the block are all zero for some small s. So, one in $2^{n/2-s} = \frac{2^{n/2}}{2^s}$ blocks is distinguished; in $2^{n/2}$ iterations of F_r we expect to find 2^s distinguished points. The distinguishing criterion should be set to enable fast computation. The idea of the method is that once we get a collision in the sequence $x_0, x_1, \ldots$ we will also find that the next distinguished block we encounter has occurred before, so it is only necessary to store the distinguished points. (Enough blocks are distinguished that we expect to have at least one on the cycle.)

In order to find a collision under F we establish a distinguishing criterion, for example, the one suggested by Mitchell, and we do the following:

Step 1 We choose a random n-bit block x.

Step 2 For $i = 1, 2, \ldots$, we compute $F_r^i(x)$. If $F_r^i(x)$ satisfies the distinguishing criterion for some i, then we record the distinguished pair $(i, F_r^i(x))$. If,

[3]The term 'meaningful collisions' refers to colliding messages that are not simply just a collection of random nonsense — they pass off as legitimate messages that could be signed or sent from one party to another.

additionally, there exists a distinguished pair $(j, F_r^j(x))$ such that

$$F_r^j(x) = F_r^i(x)$$

for $j < i$, we terminate the process. (Note that, since we expect to get a distinguished collision in $O(2^{n/2})$ time, we only use $O(2^s)$ storage space for the distinguished points).

The result of the algorithm described directly above is a pair (j, i) such that $F_r^j(x) = F_r^i(x)$ with $1 \leq j < i$, and $i - j$ is the length of the cycle. In the unlikely event that x itself lies on the cycle, then $F_r^{i-j}(x) = x$. This does not give us a collision, but we do have a preimage $F_r^{i-j-1}(x)$ of x. However, since our aim is to locate a collision, in this case we select another random block x' and run the algorithm again. If x is not on the cycle, then for some $t < j$ (where $j - t = \lambda$ is the point at which the cycle starts),

$$F_r^{i-t}(x) = F_r^{j-t}(x) \text{ and } F_r^{i-t-1}(x) \neq F_r^{j-t-1}(x).$$

Hence, we have a collision; $F_r^{i-t-1}(x)$ and $F_r^{j-t-1}(x)$ collide under F_r and therefore also collide under F. We can find $F_r^{i-t-1}(x)$ and $F_r^{j-t-1}(x)$ by reverting back to the previously stored distinguished points and working forwards (since we know i and j).

Chapter 6

Generic Attacks on Iterated Hash Functions

In this chapter we collect and describe generic attacks on iterated hash functions. These attacks work on any hash function built on the Merkle-Damgård construction, even if the compression function is treated as a black box. They highlight the weaknesses of the Merkle-Damgård construction, and show that iterated hash functions do not behave like the random oracles on which they are often modeled.

6.1 The multicollision attack

With the following attack, Antoine Joux [Jou04] answered the long standing open problem of whether or not the concatenation of two independent hash values was more secure than a single hash value. This concatenated construction allegedly appeared first in Preneel's PhD thesis [Pre93] and was called *cascading*. It was presented as a means to increase the security level of a hash function at the cost of decreased performance, since it is obviously easier to use two existing n-bit hash functions than it is to design a new $2n$-bit hash function. Intuitively, two concatenated n-bit hash functions form a $2n$-bit hash function, and one might hope that the collision resistance of the new hash function is improved to $O(2^{2n/2}) = O(2^n)$. See for example Fact 9.27 in [MvOV96] for this view, which is obviously true for random oracles. Surprisingly, Joux proved that it cannot be

true if one of the hash functions is iterated. In order to solve the problem, the simpler task of constructing a multicollision in an iterated hash function had to be addressed. We now introduce the concept of a multicollision.

Definition 6.1.1. *A multicollision is an r-tuple of messages such that all r messages hash to the same value. Such a collision is referred to as an r-collision.*

The multicollision concept introduces a new security property of hash functions known as *r-collision freeness* or *r-collision resistance*. We might hope that having this property would improve the security of a hash function. Intuitively, constructing r different messages that all hash to the same value should be much more difficult than finding only two such messages. This is indeed the case for random oracles, but Joux shows that it is not true for iterated hash functions.

For the purposes of this discussion we consider a hash function F and its compression function f. This hash function is subject to the generic birthday attack, i.e. the probability of finding a 2-collision within roughly $2^{n/2}$ hash values is fairly high. For r-collisions, the collision probability is given by (2.4), and we expect to find an r-collision by hashing about $2^{n\cdot(r-1)/r}$ different messages (see equation (2.6)). Certainly if the attacker was treating the hash function as a random oracle and using a straight birthday attack it would take $O(2^{n\cdot(r-1)/r})$ calls to the hash function. When r is large, this value tends to 2^n. Surprisingly, in the case of iterated hash functions, the following attack by Joux shows that constructing a multicollision is not much harder than constructing an ordinary 2-collision.

6.1.1 Building multicollisions

We now describe Joux's attack [Jou04], which shows that constructing a 2^k-collision only costs k times as much as constructing an ordinary 2-collision. As far as the padding process is concerned, when considering collisions between messages of the same length, padding can be ignored since blocks of padding are identical. As soon as the messages collide on some intermediate chaining value, the chaining values to follow remain equal as long as the message ends are the

same. Thus, for messages of the same length, finding collisions without padding is synonymous to finding collisions with padding.

Joux presents the attack for the case in which it is assumed that the size of the message blocks is bigger than the size of the hash and chaining values (which would usually be the case). It is also assumed that we have access to an algorithm C that finds collisions under the compression with a fixed first input (for instance, C could be the birthday attack). For a hash function F, given a chaining value h as input, C outputs two different message blocks m and m' such that $f(h, m) = f(h, m')$. The algorithm may make use of the birthday attack or any other specific attack based on a weakness of f. It is imperative that C works properly for all chaining values (or at least for a significant proportion of them).

In order to illustrate how the attack works, we first describe how a 4-collision is obtained from two calls to C. Starting with an initial value $h_0 = IV$, our first call to C yields two blocks m_1 and m_1' such that $f(h_0, m_1) = f(h_0, m_1') = z$. We now allow z to be our new chaining value, and our second call to C finds blocks m_2 and m_2' such that $f(z, m_2) = f(z, m_2')$. We now have a 4-collision since

$$f(f(h_0, m_1), m_2) = f(f(h_0, m_1), m_2') = f(f(h_0, m_1'), m_2) = f(f(h_0, m_1'), m_2'),$$

with the messages $(m_1 \| m_2 \| \text{padding})$, $(m_1 \| m_2' \| \text{padding})$, $(m_1' \| m_2 \| \text{padding})$ and $(m_1' \| m_2' \| \text{padding})$ all hashing to the same value.

We now extend the above process to obtain a 2^k-collision from k calls to C:

Step 1 Start with an initial value $IV = h_0$.

Step 2 For i from 1 to k

- call C and find m_i and m_i' such that $f(h_{i-1}, m_i) = f(h_{i-1}, m_i')$
- let $h_i = f(h_{i-1}, m_i)$.

Note that this takes time roughly $k \times 2^{n/2}$. Finding a one-block collision

using the birthday attack takes $2^{n/2}$ calls to the compression function, and we need to find k such collisions.

Step 3 Pad (if necessary) and output 2^k messages in the form $(b_1 \| b_2 \| \ldots \| b_k \| \text{padding})$ where b_i is one of either m_i or m'_i.

Besides all 2^k messages hashing to the same value, we also have that all intermediate chaining values are identical. Below we provide a schematic representation of the multicollision construction.

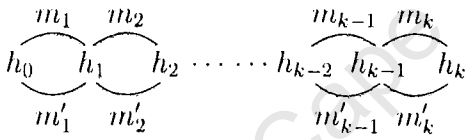


Figure 6.1: A 2^k-collision for F

Example 6.1.1. We now build a 2^2-collision for BABI. We use the primes $p = 457$ and $q = 233$. This results in a message block length of eight bits, i.e., one ASCII character each. All chaining values are six bits long. We build the following multicollision without taking Merkle-Damgård strengthening into account. We use the birthday attack to find message blocks o and q that collide on $f(h_0, \cdot)$, and h and i that collide on $f(h_1, \cdot)$.

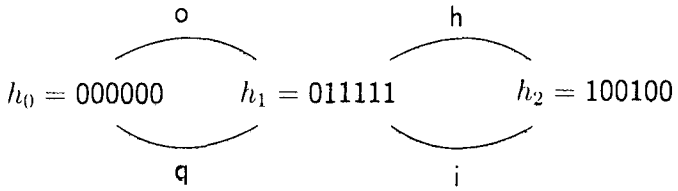


Figure 6.2: A 2^2-collision for BABI

With Merkle-Damgård strengthening, the messages oh, qi, oi and qh all hash to 100100.

6.1.2 A consequence of the multicollision attack: Cascaded hash functions

An obvious way of increasing the size of a hash value is to concatenate two or more smaller hash values coming from two or more completely different hash functions, or from multiple instances of the same hash function. Cascading two existing hash functions is presumably easier than constructing a new hash function of a larger size. In fact, until this attack by Joux it had long been assumed that if both hash functions have ideal security, then so does the function created by the concatenation of these functions.

We consider the concatenation of two independent secure hash functions F and G. We label the compression functions of these two hash functions f and g respectively. It is hoped that the security of the cascaded hash function $H = F\|G$ is of the desired level with respect to collision resistance, preimage resistance and second preimage resistance. In other words, we hope that with respect to each security level, the security of H is the product of the security levels of F and G. In the case of random oracles this is most certainly true since the only available attack is the birthday attack on $F\|G$ which takes $O(2^{\frac{n_f+n_g}{2}})$, where n_f and n_g are the sizes of the outputs of F and G respectively. As far as iterated hash functions are concerned however, by making use of the multi-collision attack, Joux shows that this is not the case. In fact, Joux shows that if F or G is an iterated hash function, then $F\|G$ is no more secure than F or G by itself.

Collision resistance

We let F and G be secure iterated hash functions. We assume that F outputs an n_f-bit hash value and that G outputs an n_g-bit hash value. Thus, in terms of collision resistance, the security level of F is $2^{n_f/2}$ and the security level of G is $2^{n_g/2}$. If $F\|G$ was ideally secure, it would exhibit a security level of $2^{(n_f+n_g)/2}$. However, assuming without loss of generality that $n_f \leq n_g$, there exists an attack on $F\|G$ which runs with a complexity of order $n_g 2^{n_f/2} + 2^{n_g/2}$.

The attack works as follows: We set k equal to $\frac{n_g}{2}$ rounded up, and find a 2^k-multicollision on F using Joux's method. If our collision finding algorithm, C, uses the generic birthday attack to find collisions between blocks, then in total we make roughly $k2^{n_f/2}$ calls to the compression function of F.

We now have a 2^k-collision on F. Since $k \geq \frac{n_g}{2}$, all we need do is apply the birthday attack to the 2^k different messages all hashing to the same value under F, and with reasonable probability we will get a collision under G. Increasing k increases the probability of success.

Work

With regards to the complexity of the attack, we know that finding a 2^k-collision on F costs roughly $k2^k$ calls to the compression function of F. With respect to the work done in finding a collision on G, we must take into account the contribution of applying G to 2^k different messages, each of which has a length of k blocks. Naively, this would cost $k2^k$ applications of the compression function of G. However, the tree structure of the messages helps to reduce the number of calls to g, because to compute the hash of $x\|m$ and $x\|m'$ once the hash of x is already computed only takes two more calls to g. In fact, if the compression functions of F and G act on the same size blocks, then the number of calls to g is roughly 2^k, one for each edge in the graph below. See figure 6.3; here h_0 is the IV of G.

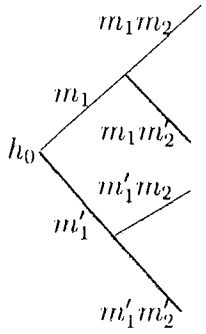


Figure 6.3: Tree structure of the messages for $k = 2$

The number of calls made to the compression function in the diagram above is in fact 2^{k+1} since there are 2^i calls at each level of the tree, and $\sum_{i=1}^{k} 2^i = 2^{k+1} = O(2^k)$.

If f and g do not act on the same size blocks, then additional padding (constant padding at the end of the shorter blocks) may be required in order to synchronize the functions before the tree structure can be of help. We can see that putting together the work involved in finding collisions on F and G respectively, gives the attack a complexity of the order of $n_g 2^{n_f/2} + 2^{n_g/2}$.

It is important to note that the application of this attack does not require G to be an iterated hash function. The attack exploits the fact that F is an iterated hash function; it would still work if G were a random oracle. Of course, in this case, the contribution of evaluating G on 2^k different messages would be $k2^k$ since no simplification from the tree structure is possible.

Example 6.1.2. Since we do have not have two or more completely different toy hash functions, the best we can do in terms of illustrating the above is to consider the hash function $H' = \text{BABI}\|\text{BABI}'$. We use the primes $p = 457$ and $q = 233$ for both BABI and BABI′. This means that message blocks are eight bits long, i.e., one ASCII character, and that chaining values are six bits long. For BABI′, we set the IV to 111111. We build on example 6.1.1 and find a 2^3-collision for BABI. See the diagram below.

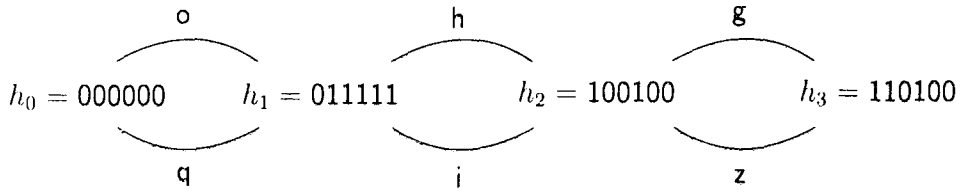


Figure 6.4: A 2^3-collision for BABI

Under BABI the messages ohg, qiz, oig, oiz, qhg, qhz, ohz and qig all hash to 110111. We now search for a collision on BABI′ among these eight messages,

and find that the messages qiz and oig collide under BABI′. Both messages have a hash value under BABI′ of 101000. We therefore conclude that the function $H' = \text{BABI}\|\text{BABI}'$ is no safer than $H = \text{BABI}$.

Preimage resistance of cascaded hash functions

Again, we work with two secure iterated hash functions F and G, and as before, F outputs an n_f-bit hash value and G outputs an n_g-bit hash value. With regards to preimage resistance the level of security of F is 2^{n_f} and the level of security of G is 2^{n_g}. The security of $F\|G$ would be $2^{n_f+n_g}$ if $F\|G$ was ideally secure. However, Joux [Jou04] shows that there exists an attack on the preimage resistance of $F\|G$ with complexity of order $n_g 2^{n_f/2} + 2^{n_f} + 2^{n_g}$.

Assuming that $n_f \leq n_g$, the attack works as follows: We set k equal to n_g and construct a 2^k-collision under F using Joux's method. It costs $k2^{n_f/2}$ calls to f to construct this multicollision (if we use the generic birthday attack — there might of course be a quicker way for that specific compression function). At this stage, the common value of the 2^k different messages under F will almost certainly not align with our predetermined target value. We thus search for an additional block that maps the last chaining value to the target value. When searching for this block, we need to consider messages under F with the padding included. In the worst case, this will cost 2^{n_f} calls to f.

Since $k = n_g$, we expect (with probability 63%) at least one of the 2^k messages to hash to the desired n_g-bit value under G. Obviously, increasing the size of k beyond n_g increases the probability of success. Assuming that f and g act on the same size blocks, the work done using a naive approach to find the desired target value under G costs up to $k2^{n_g}$ $(= n_g 2^{n_g})$ calls to g because we have to find the hash under G of 2^{n_g} n_g-block messages. Once again, making use of the tree structure of the messages helps to reduce this to $O(2^{n_g})$. Again, if the assumption does not hold, additional padding can be used to resynchronize the two functions.

Work

Putting together the contributions of applying f and g where needed gives the attack a complexity of order $n_g 2^{n_f/2} + 2^{n_f} + 2^{n_g}$. Yet again, we observe that G need not be an iterated function for the attack to work; it will also work when G is a random oracle. In this case the complexity would be of the order of $n_g 2^{n_f/2} + 2^{n_f} + n_g 2^{n_g}$.

6.2 The long-message attack

The long-message attack, described in [MvOV96], violates the second preimage resistance of an iterated hash function F that does not use Merkle-Damgard strengthening. We start with an extremely long message, M_{target}, with $2^R + 1$ message blocks, where R is some large number. We now search for a linking message m_{link} that hashes from h_0 to one of the intermediate chaining values of M_{target}. If $f(h_0, m_{link}) = h_j$, then $M' = m_{link} \| m_{j+1} \| m_{j+2} \| \dots \| m_{2^R+1}$ will have the same hash as M_{target}.

Work

The probability of reaching one of the 2^R intermediate hash states from h_0 via m_{link} is $\frac{2^R}{2^n}$. We therefore expect to try roughly $\frac{2^n}{2^R} = 2^{n-R}$ messages in search of m_{link} (see Chapter 2, Section 2.5).

Since M_{target} is very long, this is significantly less than the 2^n work needed to run a brute force attack on second preimage resistance. This also means that for long messages such as M_{target}, finding a second preimage is easier than finding a preimage. This does not hold true for shorter messages since the brute force attack is not significantly improved upon.

The longer M_{target} is, the more significant the saving over the $O(2^n)$ work needed for a brute force attack. If, for example, $R = \frac{n}{2}$, then the amount of work required to find a second preimage is comparable to the work required for finding a

collision (assuming use of the birthday attack). Messages of this length, however, are not practical.

Obviously, Merkle-Damgård strengthening foils this attack since M' and M_{target} are not of the same length, so the last block of M, m_{2^K+1}, will not be an appropriate last block for M'. If there is no Merkle-Damgård strengthening, then an attacker need only find a second preimage of any of the chaining values in order to find a second preimage of M_{target}.

6.3 The long-message attack with expandable messages

In 1999 Richard Dean [Dea99] found a way to get around Merkle-Damgård strengthening in the long-message attack for hash functions whose compression functions admit easily-found *fixed points*, by using *expandable messages* to keep the length of the second preimage message the same. In 2005 John Kelsey and Bruce Schneier [KS05] found a way to get expandable messages without fixed points by using a clever adaptation of Joux's multicollision technique, and thus extended the long-message attack to all Merkle-Damgård hash functions, even with Merkle-Damgård strengthening.

For an n-bit iterated hash function F, the Kelsey-Schneier attack finds a second preimage for a message with 2^k message blocks in roughly $2^{k+1} + 2^{n/2+1} + 2^{n-k}$ calls to the compression function. When messages are long, this is less than the required brute force amount of work. The attack might not be of practical importance — it is at least as expensive as finding a collision, and the target messages for which it is significantly faster than a brute force second preimage search are impractically long. However, it does demonstrate the discrepancies between iterated hash functions and the random oracles on which these functions are often modeled. Its biggest significance is that it shows that iterated hash functions cannot provide an ideal security level of 2^n for second preimage resistance, i.e., they

cannot reach the commonly-claimed security bounds of hash functions against second preimage resistance.

We describe first Dean's technique for constructing expandable messages if you can find many fixed points on the compression function, and then Kelsey and Schneier's more general technique. Finally we show how Dean used expandable messages to find a second preimage of a long message in the presence of Merkle-Damgård strengthening.

6.3.1 Expandable messages from fixed points

We now explain what we mean by an "expandable message".

Definition 6.3.1. *An expandable message is a set of different messages of varying lengths that all collide on the input to the last call of the compression function (i.e., when you hash all of these messages, their second-to-last chaining values will all be the same). An (a, b)-expandable message is an expandable message that can take on any length from a to b message blocks.*

Expandable messages are easy to construct if you can easily find fixed points in the compression function:

Definition 6.3.2. *For a compression function f with $f(h_i, m_{i+1}) = h_{i+1}$, a fixed point is a pair (h_i, m_{i+1}) such that $f(h_i, m_{i+1}) = h_i$.*

Compression functions based on the Davies-Meyer construction (See Chapter 4, figure 4.3) admit easily found fixed points. In this construction each intermediate chaining value is obtained by using the current message block to encrypt the previous chaining value, and then XORing again with this value to prevent inverting the compression function. In other words,

$$h_i = E_{m_i}(h_{i-1}) \oplus h_{i-1}.$$

So (h, m) is a fixed point if and only if $h = E_m(h) \oplus h$, which is the same as saying $E_m(h)$ is the zero vector 0. So to find a fixed point (h, m) for any given m

one simply decrypts the zero vector under the key m, i.e., one finds $h = D_m(0)$. Davies-Meyer hash functions include MD4 [Riv92a], MD5 [Riv92b], Tiger [AB96] and the SHA family [NIS93], [NIS95]. The techniques used to find fixed-points allow for no choice over h_i, although they might over m. Several more such techniques exist and a few of them are discussed in [MOI91] and [PGV93].

Dean's technique builds an expandable message (of a very particular form) with about twice as much work as finding a collision, i.e. with about $2 \times 2^{n/2} = 2^{n/2+1}$ calls to the compression function. The algorithm works as follows:

Step 1 Construct a list of $2^{n/2}$ fixed-points. Split the fixed-point pairs (h, m) into two lists, A and B: on A list all the chaining values, h_i, and on B list all the associated message blocks, m_i, such that $f(h_i, m_i) = h_i$.

- *For i from 1 to* $2^{n/2}$,

A	B
h_1	m_1
h_2	m_2
$\vdots$	$\vdots$
$h_{2^{n/2}}$	$m_{2^{n/2}}$

Step 2 Construct a list of $2^{n/2}$ hash values that can be reached from some initial value h_0 with one message block, i.e. the output of $f(h_0, m)$ for $2^{n/2}$ random values of m. Store the hash values and the associated message blocks on two lists, C and D. List the hash values in C, and the message blocks in D.

- *For i from 1 to* $2^{n/2}$,

C	D
$h_1{}^*$	$m_1{}^*$
$h_2{}^*$	$m_2{}^*$
$\vdots$	$\vdots$
$h_{2^{n/2}}{}^*$	$m_{2^{n/2}}{}^*$

Step 3 Find a match between lists A and C; i.e., find j and l such that $h_j = h_l{}^* = h^*$, say. (We expect such a match by the birthday argument).

Step 4 Return an expandable message $[m_l{}^*, m_j]$ (this notation denotes the use of as many copies of the $m_l{}^*$ block as is necessary, followed by one copy of the m_j block).

A message of desired length, say α blocks, that takes the initial value h_0 to the penultimate chaining value h^*, is easily produced. It consists of one copy of the starting message block from list D and $\alpha - 1$ copies of the fixed-point message block from list B.

Work

In order to construct lists A and B, the compression function needs to be called $2^{n/2}$ times. The same result holds for the construction of lists C and D. Therefore, for an n-bit hash function that allows a maximum of 2^k blocks in its messages, the work needed for this technique to produce a $(1, 2^k)$-expandable message is roughly $2 \times 2^{n/2} = 2^{n/2+1}$. (Note that this doesn't depend on k — because we are using fixed points, producing a message of more blocks doesn't require more calls to the compression function.)

6.3.2 Expandable messages without fixed points

We now describe a different technique by Kelsey and Schneier [KS05] for building expandable messages. This is a generic technique which does not require the compression function to admit easily found fixed points, and in fact works for all iterated hash functions. It is closely related to the multicollision finding technique of Joux (see section 6.1). Joux's technique first finds a sequence of single-message-block collisions. These colliding blocks are then pasted together to provide a set of messages of equal length that all hash to the same value. The technique by Kelsey and Schneier finds a sequence of collisions between messages of different lengths, (i.e., the messages within each collision pair are of differing lengths). These collision pairs are then pasted together to provide a set of mes-

sages of varying lengths that all produce the same penultimate chaining value when fed through a hash function.

We first show how to find a collision between a one block message and another of arbitrary length α; this is simply a birthday attack in which the messages in each set are chosen to have a certain form. For an n-bit iterated hash function we use the IV, h_0, and proceed to do the following:

Step 1 Choose a random message block q, and process it $t = \alpha - 1$ times to reach the chaining value h_{temp}.

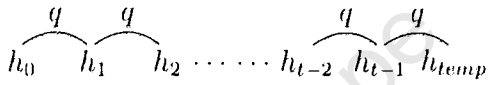


Figure 6.5: Using $\alpha - 1$ copies of q

Step 2 Compute $2^{n/2}$ hash values h_i from h_0 using message blocks, m_i, of length 1. Put these values on list A. Also, compute $2^{n/2}$ hash values $h_i{}^*$ from h_{temp} using single-block messages m_i^*. Put these on list B.

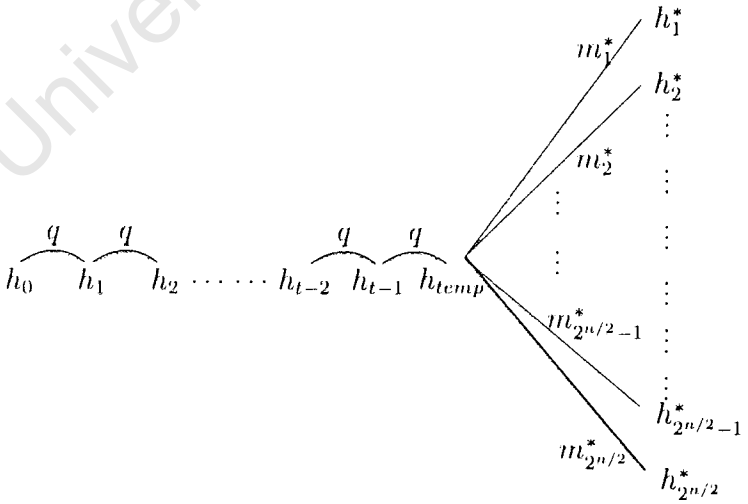


Figure 6.6: Computing $2^{n/2}$ hash values from h_{temp}

- *For i from 1 to* $2^{n/2}$,

A	B
h_1	$h_1{}^*$
h_2	$h_2{}^*$
$\vdots$	$\vdots$
$h_{2^{n/2}}$	$h^*_{2^{n/2}}$

Step 3 Find a match between lists A and B, i.e. find j and l such that $h_j = h_l{}^* = h^*$, say. (We expect such a match by the birthday argument).

Step 4 Return the colliding message pair $(m_j, q\|q\|\ldots\|q\|m_l{}^*)$.

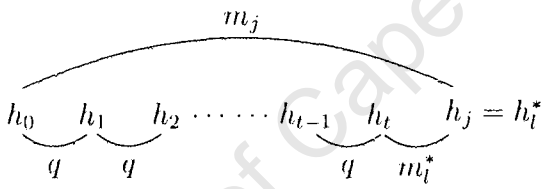


Figure 6.7: A collision between an α-block message and a 1-block message

Work

In Step 1, $\alpha - 1$ calls are made to the compression function, and in order to construct lists A and B, $2 \times 2^{n/2}$ calls to the compression function need to be made. This brings the total work for finding a collision between a one-block and an α-block message to $(\alpha - 1) + 2^{n/2+1}$.

We now show how the above algorithm can be used repeatedly to build an expandable message: In order to build a $(k, k + 2^k - 1)$-expandable message we first follow the process outlined above to find a colliding message pair consisting of a 1-block message and a $(2^{k-1} + 1)$-block message starting from h_0. We then find a second colliding message pair consisting of a 1-block message and a $(2^{k-2} + 1)$-block message starting from h_1. The third message pair consists of a 1-block message and a $(2^{k-3} + 1)$-block message starting from h_2. We continue going until the final message pair consists of a 1-block message and a 2-block

message. See the diagram below: the number of blocks in each colliding message in a message pair is indicated above or below the corresponding message.

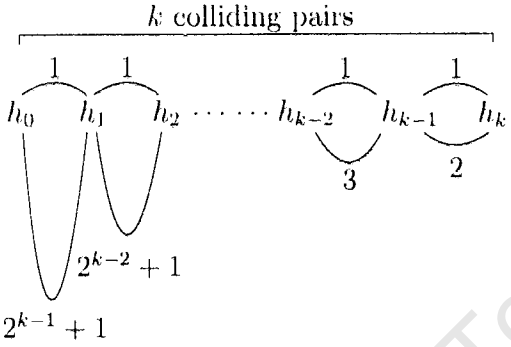


Figure 6.8: Construction of k colliding message pairs

By pasting all of the message pairs together we obtain an expandable message with minimum length k blocks and maximum length $\sum_{j=1}^{k}\left(2^{k-j}+1\right) = k+2^k-1$ blocks.

We label the message pairs as follows:

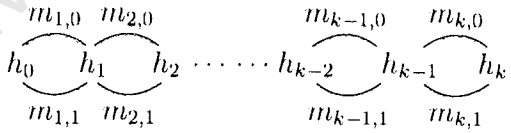


Figure 6.9: Pasting message pairs together

We now show that we can use this pattern to produce a message M of any desired length α between k and $k+2^k-1$. Note that in each message pair $\{m_{i,0}, m_{i,1}\}$, message $m_{i,1}$ is 2^i blocks longer than message $m_{i,0}$. We can therefore build a message of length α by using the bits of the binary representation of $\alpha - k$ to select $m_{i,0}$ or $m_{i,1}$ for each i. If $\alpha - k = \sum_{i=0}^{k-1} s_i 2^i$ (recall $\alpha - k < 2^t$), then for $i = 1, 2, \ldots, k$, choose the short message $m_{i,0}$ if $s_i = 0$, else choose the long message $m_{i,1}$. We build the message M by pasting together the message blocks

selected at each stage, i.e.,

$$M = m(1, j_1) \| m(2, j_2) \| \dots \| m(k, j_k),$$

where $j_i \in \{0, 1\}$.

Work

Finding the ith message pair takes $2^{i-1} + 2^{n/2+1}$ calls to the compression function, so in total the number of calls to the compression function involved in finding a message of length α between k and $k + 2^k - 1$ is

$$\begin{aligned} & k2^{n/2+1} + \sum_{i=0}^{k-1} 2^i \\ &= k2^{n/2+1} + 2^k - 1 \\ &\approx k2^{n/2+1} + 2^k. \end{aligned}$$

(Note this depends on k rather than on α, because we still have to find all k message pairs no matter what α is.) If $k \ll n$ then the number of calls is approximately $k2^{n/2+1}$.

Remark 6.3.1. Both this generic technique and Dean's technique allow for expandable messages to start at any chaining value, which means that we can build messages that start with a desired prefix.

Remark 6.3.2. This technique builds a 2^k-collision, before the final compression function computation is taken into account, by using roughly $k \times 2^{n/2+1}$ calls to the compression function. This is roughly twice as expensive as the Joux technique which builds a 2^k-collision of messages of the same length using $k \times 2^{n/2}$ calls to the compression function. The reason for the difference is that in Joux's attack each colliding message pair starts from the same h_i and we are finding a collision within one set, whereas here we need to find collisions between *two* sets, one starting from h_{temp} and one from h_0.

Remark 6.3.3. There is no reason why we cannot use distinct message blocks $a, b, c, \ldots, z$ instead of repeats of the message block q when building the expandable message. This may allow for more flexibility in the format of the second preimage that we eventually find. However, it may not be entirely possible to make the second preimage meaningful due to the fact that we are forced to use only a few components of the expandable message (depending on the bit pattern of $\alpha - k$) and, as we will see in the next section, α is only determined after the expandable message has been built.

6.3.3 Using expandable messages to find second preimages

Let M_{target} be an extremely long message of 2^k+k+1 message blocks. Attempting to find a second preimage, M', by using the long-message attack described in Section 6.2 will not work because Merkle-Damgård strengthening ensures that the final length fields of M_{target} and M' are different and therefore M_{target} and M' almost certainly do not hash to the same final value. However, combining the long-message attack with an expandable message circumvents Merkle-Damgård strengthening by constructing a second preimage M_{found} of the same length as M_{target}. The attack works as follows:

Step 1 Start with a long message, M_{target}, that is $2^k + k + 1$ blocks long, and compute its intermediate hash values $h_1, h_2, \ldots, h_{2^k+k}$.

Step 2 Make a $(k, 2^k - 1 + k)$ expandable message using the procedure outlined above. (Recall that an expandable message is a set of 2^k possible messages, all of which collide on the penultimate chaining value). The expandable message can start from any chosen initial value h_0^*, and we denote the end hash chaining value of the expandable message by h^*.

Step 3 Carry out the long message attack on M_{target}, but starting from the end of the expandable message constructed in Step 2. In other words, find a linking message m_{link} such that $f(h^*, m_{link}) = h_j$ for some $j \in$

$\{k+1, k+2, \ldots, 2^k + k\}$, where h_j is one of the intermediate chaining values of M_{target}. (The intermediate hash values $h_0, h_1, \ldots, h_k$ need to be excluded when searching for m_{link} because the expandable message from Step 2 has a minimum message length of k blocks).

Step 4 Now use the expandable message to produce a message M^* of $j-1$ blocks, so that $M^* \| m_{link}$ is a message of j blocks which can replace the first j blocks of M_{target} without affecting the intermediate hash values from h_j onwards.

Step 5 Return a second preimage M_{found} where

$$M_{found} = M^* \| m_{link} \| m_{j+1} \| m_{j+2} \| \ldots \| m_{2^k+k+1}.$$

We now have two messages, M_{target} and M_{found}, of the same length, whose final length fields are thus the same, and which therefore hash to the same final value.

Work

The compression function is put to use at three different stages in the process above: in Step 1 when the intermediate hash values are computed, in Step 2 when the expandable message is constructed, and in Step 3 during the search for m_{link}. Adding together the number of calls at each of these stages gives us the total work:

$$\overbrace{(2^k + k)}^{\text{Step 1}} + \overbrace{k \times 2^{n/2+1} + 2^k}^{\text{Step 2}} + \overbrace{\frac{2^n}{2^k + k}}^{\text{Step 3}}.$$

Since $k \ll 2^k$, we regard k as negligible and this simplifies to

$$2^{k+1} + 2^{n/2+1} + 2^{n-k}.$$

The longer the target message the more efficient the attack relative to a brute force attack, until k is so large that that the $k \times 2^{n/2+1}$ term dominates the 2^{n-k} term, at which point the work to find the expandable message becomes greater than the work for the long-message attack.

6.3.4 Variations

It is possible to allow the second preimage to have the first few hundred or thousand messages the same as the target message. We just start the expandable message from the intermediate value just after the last desired identical message block, instead of from h_0.

We can also construct a second preimage to have the last few hundred or thousand message blocks the same as the target message. In this case we start from h_0, and we restrict the intermediate hash states that can be reached with m_{link} to the hash states occurring before the first of the desirably identical message blocks gets processed by the compression function.

6.4 The herding attack

Previously mentioned uses of hash functions include proof of prior knowledge and commitment to a secret whilst keeping the secret hidden. In 2006 John Kelsey and Tadayoshi Kohno [KK06] presented an attack on Merkle-Damgård hash functions in which they find a preimage of a desired form without much more effort than finding a collision.

Consider the following scenario: Ned, a modern-day Nostradamus, claims on day D_1 to know the closing prices of gold (P_1), brent crude oil (P_2) and platinum (P_3) according to some or other financial index on December 31, 2008. He also claims to know the outcome of certain other events that have not yet come to fruition. On this day, D_1, Ned claims that the hash value under F of all of this information (stock prices and other) is h^*. We assume F to be a 'good' hash function in the sense that its output is large enough to defeat preimage attacks; let the output be a bit string of size n. On January 1, 2009, Ned produces a message M such that $F(M) = h^*$. The three closing prices make up the first part of M.

The question we need to consider is whether or not we have sufficient evidence to conclude that Ned did indeed know the value of P_1, P_2 and P_3 at time D_1. If not, then it would seem that Ned would have to know how to find a preimage of h^* under F. However, given that the output of F is large, and assuming that the best preimage attack on F is a brute force attack, then this seems highly unlikely.

The attack provided by Kelsey and Kohno actually violates a less well-known property of hash functions known as *Chosen Target Forced Prefix (CTFP) preimage resistance*. This property reads as follows:

Property 4. ***Chosen Target Forced Prefix preimage resistance***
A hash function F is said to be CTFP preimage resistant if for a value h^* chosen by the attacker, and for a given prefix P (specified after h^* is chosen), it is computationally infeasible to find M such that $F(P\|M) = h^*$.

The attack makes use of repeated application of a collision finding algorithm on F and thus shows that applications such as commitment schemes which were previously thought to be unrelated to collision resistance, are no longer as secure as previously hoped.

Before the attack is described, we introduce the concept of a *diamond structure*. This is essentially a cleverly devised data structure that allows us to 'herd' any given prefix to a certain hash value by walking down the branches of the diamond. It is this 'herding' ability that gives the attack its name. For a parameter k, the diamond structure produces a 2^k multi-collision in a rather different manner from that of Joux [Jou04]. Figure 6.10 depicts a diamond structure with $k = 3$. We use the hash function's IV, h_0, and apply the compression function f of F to 2^k distinct message blocks m_{0i} for i from 1 to 2^k. We then have 2^k intermediate hash values h_{1i} for i from 1 to 2^k. We now search for distinct message blocks m_{1i} and m_{1i+1} such that $f(h_{1i}, m_{1i}) = f(h_{1(i+1)}, m_{1(i+1)}) = h_{2\frac{i+1}{2}}$ (say) for $i \in \{1, 3, 5, \ldots, 2^k - 1\}$, creating a "subdiamond" $(h_0, h_{1i}, h_{2\frac{i+1}{2}}, h_{1(i+1)}, h_0)$. By the birthday argument we expect to find such a collision in roughly $2 \times 2^{n/2}$ calls

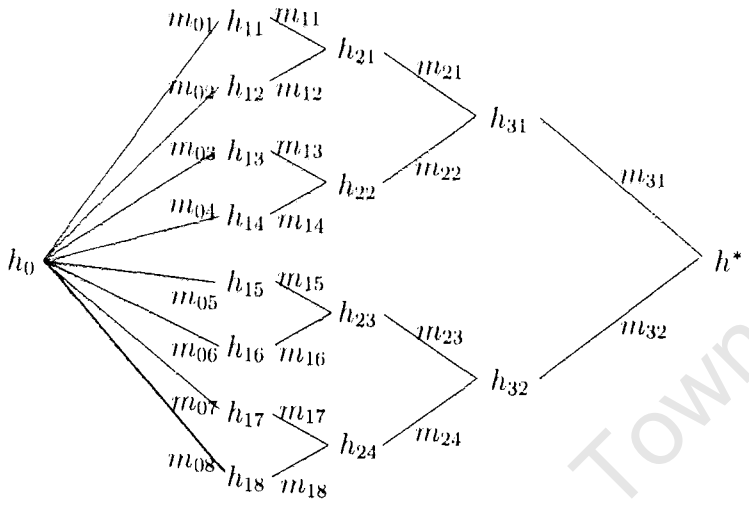


Figure 6.10: Diamond structure with $k = 3$

to the compression function. We repeat this process for the pairs $h_{2i}, h_{2(i+1)}$ for odd i, obtaining 2^{k-1} larger diamonds, $(h_0, h_{2i}, h_{3\frac{i+1}{2}}, h_{2(i+1)}, h_0)$. We continue to do this until we reach a final hash value h^* (which is published by the attacker). Since every collision closes a subdiamond from h_0, and there are $2^k - 1$ subdiamonds in our diamond structure (one for each pair of adjacent $m_0 i, m_{0(i+1)}$), the expected total work for this construction is

$$2 \cdot 2^{n/2}(2^k - 1) \approx 2^{n/2+k+1}$$

calls to the compression function.

We are now in the position to "herd" F. Note that the attacker does not choose h^*; this value is dependent on the diamond structure.

6.4.1 Herding without Merkle-Damgård strengthening

We first ignore Merkle-Damgård strengthening and herd F by doing the following:

Step 1 At time D_1 we build a diamond structure and determine the value h^* that we will claim to be the hash value of our predictions.

Step 2 At time D_2 we gain knowledge of the prefix $P = P_1 \| P_2 \| \ldots \| P_l$. This constitutes the first part of our prediction. (In Ned's case, he obtains the value P_1, P_2 and P_3). We feed P through F to get an intermediate hash value h_{tmp}.

Step 3 We search for a linking message block m_{link}, that when appended to P and fed through F, yields one of the intermediate hash values in our diamond, i.e., we feed random messages m_{link} into $f(h_{tmp}, \cdot)$ until we find one whose output is one of the $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$ intermediate hash values in the diamond structure we expect to try about $2^n/2^{k+1} = 2^{n-k-1}$ messages before finding m_{link}. See figure 6.11.

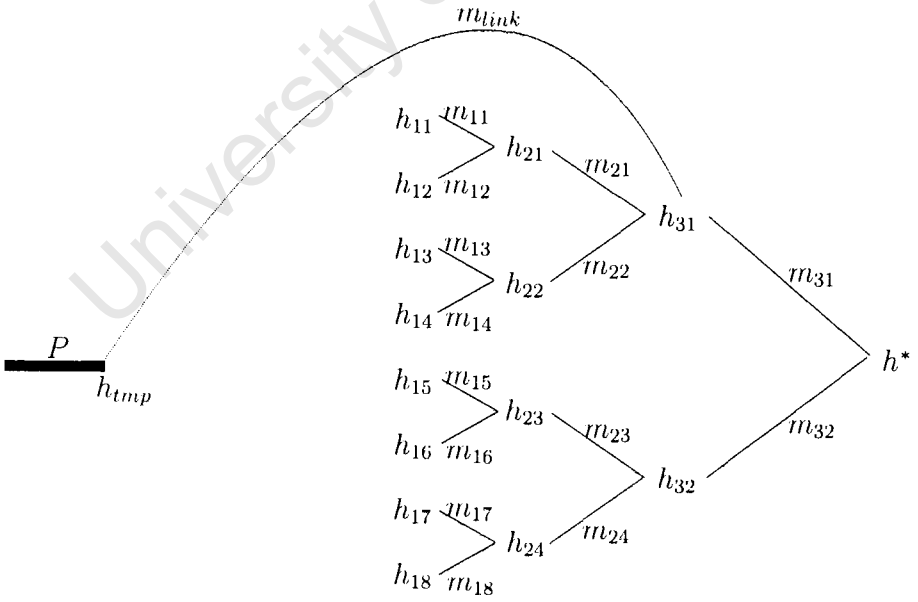


Figure 6.11: Herding without Merkle-Damgård strengthening

Step 4 After we have found m_{link} and h_{ji} such that $f(h_{tmp}, m_{link}) = h_{ji}$, we produce the sequence of message blocks M' by collecting the relevant message

blocks as we walk down the branches of the diamond from h_{ji} to h^*. We now have a message $M = P_1 \| \ldots \| P_l \| m_{link} \| M'$ such that $F(M) = h^*$.

By following the procedure outlined above, we are able to first commit to a hash value h^* and then produce a message which starts with any given prefix, P, and hashes to h^*. See figure 6.11. The message $M = P \| m_{link} \| m_{31}$ hashes to h^*.

6.4.2 Herding with Merkle-Damgård strengthening

There are two possible ways to deal with Merkle-Damgård strengthening. One way involves running the algorithm as given above, and then appending a $(1, k)$-expandable message (see Section ??) to the end of the diamond structure. This ensures that all possible messages built using the diamond structure can still be made to have the same length, and hence to produce the same value when fed through a hash function that employs Merkle-Damgård strengthening. The other possibility involves choosing m_{link} by only considering values h_{1i} in the widest layer of the diamond structure. We would need to know an upper bound for $|P|$ before we start — say, l blocks. Then we can ensure the final message block is $l+k+1$ blocks long by choosing the length of m_{link} appropriately and only searching for $f(h_{tmp}, m_{link})$ in the widest layer of the diamond. Of course in this case after we find the final hash h^* of the diamond we will publish $f(h^*, \textit{length-field})$ as our hash value.

Work

The work needed to run the attack without Merkle-Damgård strengthening can be broken down as follows:

$$\overbrace{2^{n/2+k+1}}^{\text{Step 1}} + \overbrace{2^{n-k-1}}^{\text{Step 3}}. \tag{6.1}$$

We consider the work needed to complete steps 2 and 4 to be negligible.

The work needed to run the attack with Merkle-Damgård strengthening and

an expandable message is

$$\overbrace{2^{n/2+k+1}}^{\text{Step 1}} + \overbrace{2^{n-k-1}}^{\text{Step 3}} + \overbrace{k \times 2^{n/2+1}}^{\text{Step } *} \tag{6.2}$$

The work described for Step $*$ is what is needed to build a $(1, k)$-expandable message (see Section ??).

The work needed to run the attack with Merkle-Damgård strengthening and using the widest layer of the diamond only is approximately

$$\overbrace{2^{n/2+k+1}}^{\text{Step 1}} + \overbrace{2^{n-k}}^{\text{Step 3}} \tag{6.3}$$

To minimize the total work needed for the attack, we set the work needed for building the diamond structure equal to the work needed for finding m_{link} (so that neither phase forms a "bottleneck"), and we solve for k. (Step $*$ requires much less work than Step 1, and can be ignored.) In the case of 6.1

$$k = \frac{n-4}{4},$$

and in the case of 6.3

$$k = \frac{n-2}{4}.$$

In the case without Merkle-Damgård strengthening, the work required is approximately 2^{n-k} since we have considered both terms of 6.1 to contribute equally to the work. If for example F is MD4 (see Section 4.4.2 and [Riv92a]) and we run this version of the attack, then

$$k = \frac{128-4}{4} = \frac{124}{4} \approx 31.$$

Therefore, the work required to find a preimage under MD4 is roughly $2^{128-31} = 2^{97}$. Although this is still too large for the attack to be practical, it is much better than the 2^{128} work required for a brute force preimage attack.

Remark 6.4.1. If it is possible to predetermine the set of possible prefixes, then the diamond structure can be built by using these in its widest layer, i.e., as the m_{1i}. In this case, the total work for the attack is comprised only of the amount needed to build the diamond.

Remark 6.4.2. If the collisions are found using the birthday attack, then the attacker can choose his m_{ij} to be meaningful message blocks.

Some applications of this attack mentioned in [KK06] include stealing credit for inventions, modification of signed documents and random number fixing[1].

Example 6.4.1. We now exhibit the herding attack using BABI. We let $p = 457$ and we let $q = 233$. Message blocks are eight bits long, i.e., one ASCII character, and all chaining values are six bits long. Assume that a week ago we claimed to know what the stock prices of gold, brent crude oil and platinum would be today. We also claimed to know the outcome of certain events that are to happen in the distant future. As proof of this knowledge, we presented the hash value h^*. We claimed that this was the hash value of all the information under BABI and we did not use Merkle-Damgård strengthening. Of course, one week ago we had no idea what the stock prices were. We followed the steps outlined above and started by building the diamond structure displayed in figure 6.12 (which took three collisions).

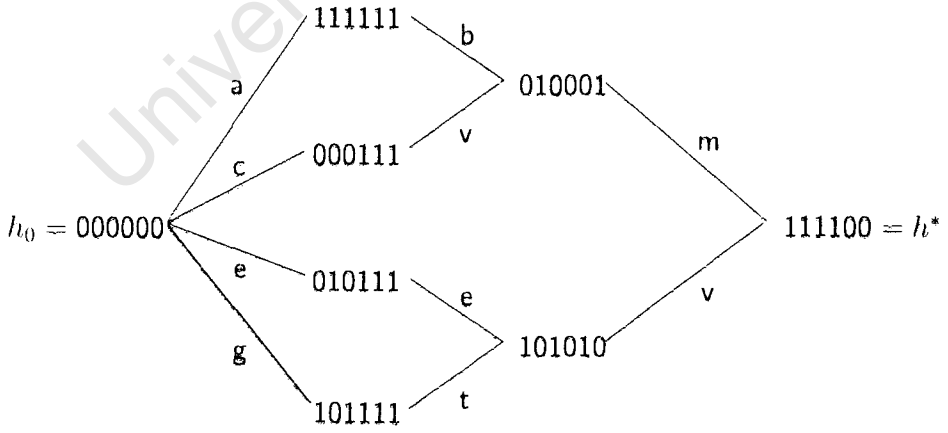


Figure 6.12: Diamond structure with $k = 2$

[1]Alice and Bob wish to play a game that requires them to agree on a certain sequence of random bits. By using the herding attack, Alice can exert a great deal of control over the resulting sequence.

A minute ago we gained knowledge of the relevant stock prices: the price of gold is g, the price of oil is o and the price of platinum is p. We compute the intermediate chaining value, $h_3 = 001000$, by feeding g‖o‖p through BABI. Note that the message length is ignored. We then consider roughly $2^{6-2} = 16$ candidate message blocks and find that the message block n links from h_3 to the third chaining value, 010111, in the widest layer of our diamond. See figure 6.13.

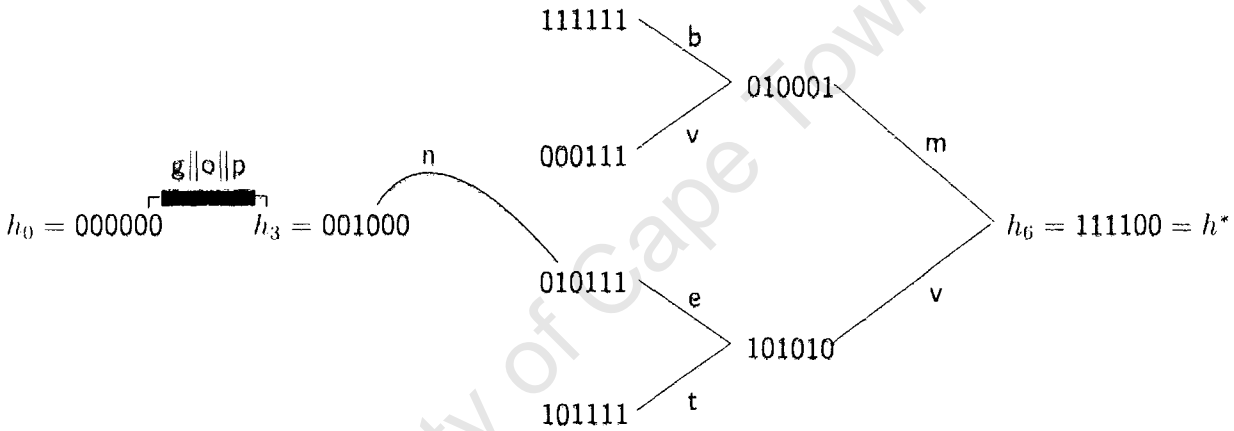


Figure 6.13: Herding BABI

If we pass $M =$ g‖o‖p‖n‖e‖v as input to BABI, we see that the hash value, h_6, without taking message length into account, is $111100 = h^*$. Hence, we have successfully herded BABI. The issue of length can be easily resolved: We know that the input to BABI will be five message blocks (assuming we only look for $f(h_3, m_{link})$ among the widest layer of the intermediate hash values). Thus, from h^* we just hash the appropriate length field and obtain the hash value to be published.

Remark 6.4.3. In [KK06], Kelsey and Kohno mention that the diamond can be built more efficiently with work $2^{n/2} + k/2 + 2$. In what follows, we assume this to be the work needed for constructing a diamond.

6.5 Combining herding and the long-message attack

In this section we describe an attack by Andreeva *et al.* [ABF+08]. It combines the herding attack of Kelsey and Khono [KK06] with the second preimage attack of Kelsey and Schneier [KS05] to produce a different long-message second preimage attack on iterated hash functions. As will be seen, this attack requires slightly more work than is needed for the second preimage attack of Kelsey and Schneier. However, this attack is also applicable to a Merkle-Damgård variant, namely *dithered hash functions*, whereas the attack by Kelsey and Schneier is not. (The application of this attack to dithered hash functions will be described in Chapter 7). Since the herding attack and the Kelsey-Schneier attack have both been described previously, we skip over their details and move straight into describing the combined attack.

For a hash function F, we start with a target message M_{target}, where M_{target} is made up of 2^l message blocks; $M = m_1 \| m_2 \| \dots \| m_{2^l-1} \| m_{2^l}$ and $F(M) = h_{2^l}$. We do the following:

Step 1. We build a directed diamond structure of depth w (i.e., there are 2^w intermediate hash states in the diamond's widest layer), such that all possible paths of w message blocks lead from the IV h_0 to the value h^*. For construction of the diamond see Section 6.4.

Step 2. We search for a linking message block, m_{link}, that connects h^* to one of the intermediate hash values produced during the computation of $F(M_{target})$. We do this by randomly considering options for m_{link} until $f(h^*, m_{link}) = h_i$ for some $w + 1 \leq i \leq 2^l$.

Step 3. We generate a prefix, P, of size $i - w - 1$ blocks such that P hashes from h_0 to one of the 2^w hash values in the widest layer of our diamond - say h_{1j}. We let D denote the chain of w message blocks leading from h_{1j} to h^*.

Step 4. We output the message $M' = P\|D\|m_{link}\|m_{i+1}\|\ldots\|m_{2^l}$.

The messages M_{target} and M' hash to the same value under F. They are both of the same length and therefore still hash to the same value when Merkle-Damgård strengthening is applied. In the event that $i = 2^l$, no message blocks of M will be used in M'.

Work

Building the diamond in Step 1 requires $2^{n/2+w/2+2}$ work [KS05]. Finding m_{link} in Step 2 requires roughly 2^{n-l} work, and Step 3 requires roughly 2^{n-w} work. Therefore, the total work for the attack amounts to

$$\overbrace{2^{n/2+w/2+2}}^{\text{Step 1}} + \overbrace{2^{n-l}}^{\text{Step 2}} + \overbrace{2^{n-w}}^{\text{Step 3}}.$$

This is minimized when $w = (n-2)/3$ [ABF+08], so the total work is $5 \cdot 2^{\frac{2n}{3}} + 2^{n-w}$.

Although this attack is slightly more expensive than the Kelsey-Schneier attack, it does allow the attacker more flexibility when constructing M' because he has more control over P than the attacker has over the expandable message in the Kelsey-Schneier attack, and can therefore make P meaningful.

Finally we note that part of this attack can be viewed as a new way of constructing expandable messages. A prefix of the appropriate length is selected and then connected to the first level of a diamond, from where it is herded to h^*. The work required to do this is $2^{n/2+w/2+2} + 2^{n-w}$ where w is the depth of the diamond. This method is more flexible than the technique described in Section 6.4 because an attacker has much more control over P.

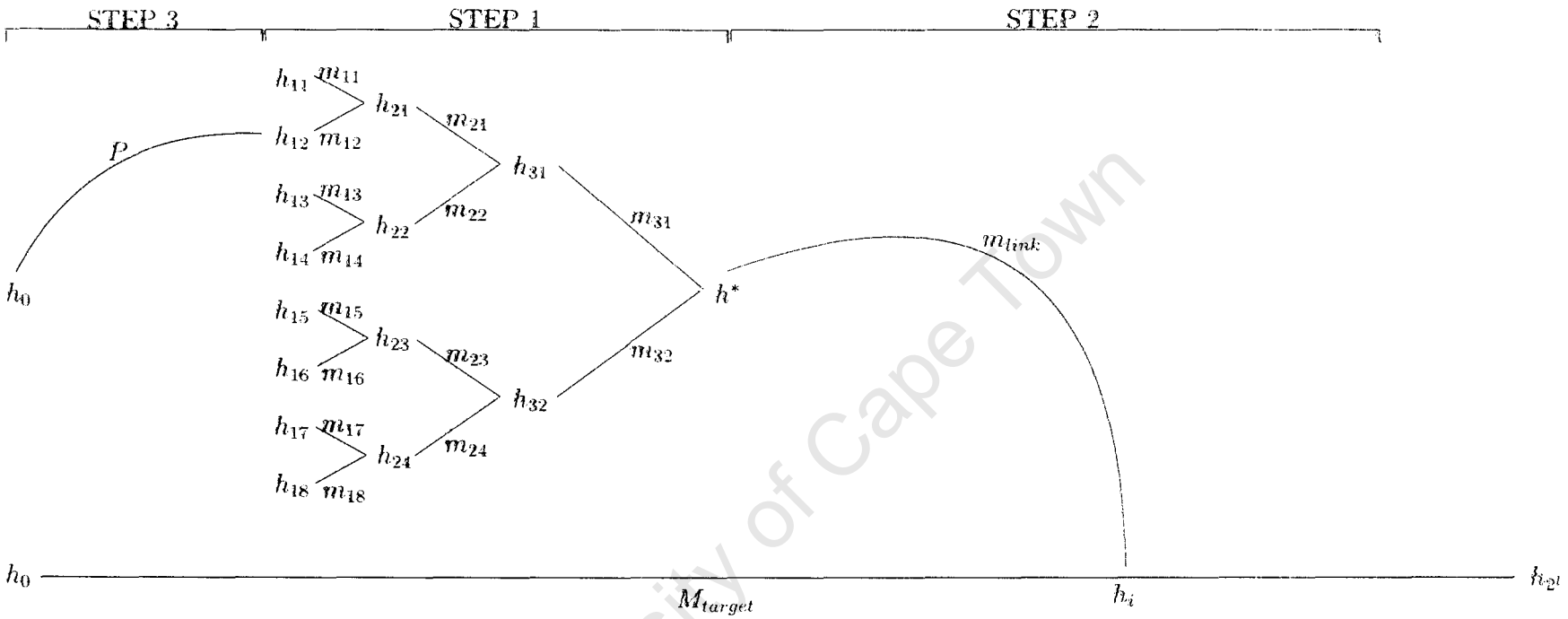


Figure 6.14: Herding and the long-message attack

6.6 The poisoned block attack

It frequently used to be argued that collision attacks were irrelevant in real applications because colliding messages produced by most practical attacks (such as the differential attack by Wang *et al* [WGLY04]) are usually meaningless, i.e., the attacker has little control over the colliding blocks he finds. The poisoned block attack, however, allows us to use a random collision on the compression function of an iterated hash function to construct a meaningful collision on the hash function.

Consider an iterative hash function F and its compression function f. Assume there exists a collision-finding algorithm C that, given some chaining value h, produces two distinct message blocks, m and m', such that $f(h, m) = f(h, m')$. According to the Merkle-Damgärd construction, for a message $M = m_1 \| m_2 \| \dots \| m_l$, we have $h_i = f(h_{i-1}, m_i)$, where h_0 is the IV of the hash function. In the poisoned block attack we fix some $j \in \{0, 1, \dots, l\}$ and run C in order to find two message blocks N and N' such that $f(h_j, N) = f(h_j, N')$. We now notice that the two messages

$$M = m_1 \| m_2 \| \dots \| m_j \| N \| m_{j+2} \| \dots \| m_l,$$

and

$$M' = m_1 \| m_2 \| \dots \| m_j \| N' \| m_{j+2} \| \dots \| m_l$$

collide under F. The blocks N and N' may be meaningless, but they are now part of longer messages that can be carefully constructed so as to inconspicuously incorporate them.

Example 6.6.1. We do not break BABI using the poisoned block attack. This is because BABI has been designed to take in simple messages from the console, i.e., the program cannot process files. Instead, we provide an example by Magnus Daum and Stefan Lucks [DL05] that aptly demonstrates the effectiveness of the poisoned block attack.

Daum and Lucks provide two postscript files that hash to the same value under MD5. One of the files is a letter of recommendation for an evil employee Alice, and the other is a letter granting Alice access to secret documents. In short, Alice gets her boss, Bob, to sign the letter of recommendation, thereby providing her with a digital signature for the letter granting her access.

The example makes use of the following postscript construction:

$$(R_1)(R_2) \text{ eq } \{\text{instruction set 1}\}\{\text{instruction set 2}\} \text{ ifelse.}$$

This syntax commands the execution of instruction set 1 if $R_1 = R_2$ and the execution of instruction set 2 otherwise. Assume that instruction set 1 commands the display of the letter of recommendation for Alice, and that instruction set 2 commands the display of the letter granting Alice access to privileged information.

Alice makes use of the differential attack on MD5 by Xiaoyun Wang and Hongbo Yu [WY05][2] to find R_1 and R_2 such that $R_1 \neq R_2$ and $\text{MD5}(R_1) = \text{MD5}(R_2)$. The attacker has little control over the messages R_1 and R_2, so they will probably look like meaningless garbage. Alice is now in a position to produce two documents

$$\begin{aligned} D &= (R_2)(R_2) \text{ eq } \{\text{instruction set 1}\}\{\text{instruction set 2}\} \text{ ifelse} \quad \text{and} \\ D' &= (R_1)(R_2) \text{ eq } \{\text{instruction set 1}\}\{\text{instruction set 2}\} \text{ ifelse,} \end{aligned}$$

such that $\text{MD5}(D) = \text{MD5}(D')$. The postscript code for both documents is included in Appendix B. Document D is the letter of recommendation, and D' is the letter granting Alice access to secret documents. Both D and D' hash to a25f7f0b29ee0b3968c860738533a4b9 under MD5.

Alice sends D to Bob and requests a digital signature. When Bob opens D he sees the letter of recommendation and signs it using the signature scheme reliant on MD5. Due to the collision of D and D' under MD5, this signature is also

[2]The attack by Wang and Yu is not generic, and is therefore not described in this dissertation.

valid for D'. Alice may now present D' accompanied with a valid signature.

Although R_1 and R_2 are meaningless (see Appendix B), the exploitation of the 'if-then-else' construction in the postscript language allows for these meaningless, colliding elements to produce a very meaningful collision. Of course, the deception is easily detected upon direct examination of the source code. However, in practice, programs interpreting advanced document languages such as the postscript language are usually trusted.

Chapter 7

Generic Attacks on Some Merkle-Damgård Variants

In response to the generic attacks on iterated hash functions described in the previous chapter, several variations of the classical Merkle-Damgård construction have been suggested in the hope of improving security. We now present some of these variants, and show that security against multicollisions is not significantly improved by the suggested modifications. This means, for example, that cascading these hash functions does not improve their security against collisions.

7.1 Generalised Sequential Hash Functions

Joux's multicollision attack [Jou04] obtains a 2^k-collision on a classical iterated hash function with only slightly more effort than finding an ordinary 2-collision; this raises the question of how the Merkle-Damgård construction might be modified so as to prevent the multicollision attack. A natural approach is to try using message blocks more than once. Nandi and Stinson considered this approach in [NS04], in which they defined a more general class of hash functions called *generalised sequential hash functions*, and immediately showed that it is possible to find multicollisions on this class of hash functions efficiently if each message block is used at most twice in computing the hash. Hoch and Shamir [HS06a] extended these results to find multicollisions on generalised sequential hash functions in

which message blocks are processed more than twice, thus ruling out this class of hash functions as a suitable replacement for the Merkle-Damgård construction.

We define generalised sequential hash functions now:
Let f be a compression function. We define a hash function F based on f as follows: For every possible message length of l blocks, we define a sequence $\alpha^l = \langle \alpha_1^l, \alpha_2^l, \ldots, \alpha_s^l \rangle$ of length s, where each $\alpha_i^l \in \mathcal{S}_l = \{1, 2, \ldots, l\}$ and each number in $\mathcal{S}_l$ appears in α^l at least once (so $s \geq l$). To hash a message $M = m_1 \| m_2 \| \ldots \| m_l$ we compute

$$h_i = f(h_{i-1}, m_{\alpha_i}), \qquad i = 1, 2, \ldots, s,$$

where h_0 is an initial value and where h_s the final hash value $F(M)$. (See figure 7.1.) We refer to $m_{\alpha_1} \| m_{\alpha_2} \| \ldots \| m_{\alpha_s}$ as the *expanded message*.

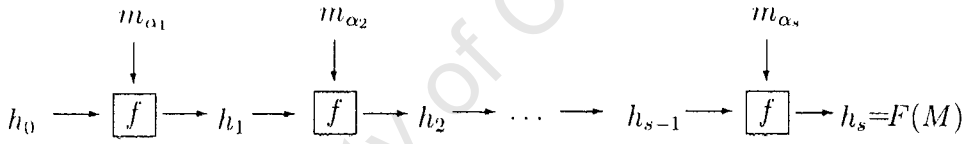


Figure 7.1: A hash function based on the sequence α

This construction processes each message block at least once, in an order determined by the sequence α^l, whereas the classical construction uses each message block exactly once. If the sequence α^l is the sequence $\Psi^{(1,l)} = \langle 1, 2, \ldots, l \rangle$, i.e., $\alpha_i = i$ for $i = 1, 2, \ldots, l$, then we get the classical Merkle-Damgård construction. If α^l is $\Psi^{(2,l)} = \langle 1, 2, \ldots, l, 1, 2, \ldots, l \rangle$, then each message block is processed in order exactly twice. If α^l is the sequence Θ^l described in section 2.7, then each block is still processed twice, but in a different order. Note that Joux's multicollision attack will no longer work, because a pair of message blocks m_i and m_i' that collide on $f(h_i, \cdot)$ will almost certainly not collide on $f(h_j, \cdot)$ when the message blocks have to be used again at step j.

7.1.1 Nandi and Stinson's attacks

Nandi and Stinson describe two methods for finding k-collisions on a generalised sequential hash function in [NS04]: the first works if the sequence α contains a long enough chain, and the second works if no terms appears in α more than twice and α contains an initial interval in which enough terms appear only once. They then prove that if the hash function is based on sequences α^l with frequency at most 2, then l can be chosen so that at least one of the attack methods can be applied.

The first Nandi and Stinson attack

The idea of the first attack is to identify k message blocks that form a chain, and then to vary only these message blocks to find a sequence of k internal collisions which can be pasted together in different ways to form a multicollision. This attack works, for example, on hash functions F based on the sequence Θ^l. We describe it using

$$\Theta^5 =< 1, 2, 1, 3, 2, 4, 3, 5, 4, 5 > .$$

We denote the compression function by f.

Step 1 We identify $1 \prec 3 \prec 5$ as a longest chain in Θ^5, and we let $k = \text{maxchain}(\Theta^5) = 3$. (In general, $\text{maxchain}(\Theta^l) = \lfloor \frac{l+1}{2} \rfloor$, see section 2.7). This means that all occurrences of 1 in Θ^5 occur before all occurrences of 3, which occur before all occurrences of 5. This allows us to break the sequence Θ^5 into three sections, each of which contains only *one* of $\{1, 3, 5\}$. So the computation of the hash can be split into three stages in such a way that message blocks m_1, m_3 and m_5 are each used in only one stage. We will find a collision at the end of each stage by varying only one of the message blocks m_1, m_3 and m_5 at a time.

Step 2 We choose a random block Z and set m_2 and m_4 equal to Z.

Step 3 We apply the birthday attack to find two distinct blocks m_1 and m_1' such that

$$f(f(f(h_0, m_1), Z), m_1) = f(f(f(h_0, m_1'), Z), m_1') = h_a,$$

i.e., the expanded messages $m_1\|Z\|m_1$ and $m_1'\|Z\|m_1'$ collide on the third chaining value starting from h_0.

Step 4 We apply another birthday attack to find two distinct blocks m_3 and m_3' such that

$$f(f(f(f(h_a, m_3), Z), Z), m_3) = f(f(f(f(h_a, m_3'), Z), Z), m_3') = h_b,$$

i.e., the expanded messages $m_3\|Z\|Z\|m_3$ and $m_3'\|Z\|Z\|m_3'$ collide on the fourth chaining value starting from h_a.

Step 5 We apply a final birthday attack to find two distinct blocks m_5 and m_5' such that

$$f(f(f(h_b, m_5), Z), m_5) = f(f(f(h_b, m_5'), Z), m_5') = h_c,$$

i.e., the expanded messages $m_5\|Z\|m_5$ and $m_5'\|Z\|m_5'$ collide on the third chaining value starting from h_b.

Finally we build a 2^3-multicollision for the hash function based on Θ^5 by pasting together the message block collisions produced in steps 3, 4 and 5: the colliding messages are $m_1^*\|Z\|m_3^*\|Z\|m_5^*$, where m_i^* can be either m_i or m_i'. For a schematic representation of this attack, see figure 7.2.

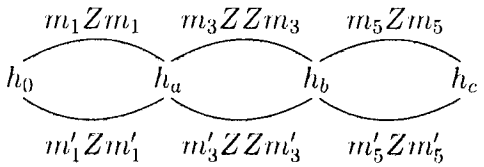


Figure 7.2: A 2^3-multicollision for the hash function based on Θ^5

Work

In order to find a 2^3-collision on the hash function based on Θ^5 we run $k = 3$ birthday attacks. The first and third birthday attack find a collision between messages of three blocks, and therefore require three calls to the compression

function for each candidate message, and we expect to need at most $2.5 \times 2^{n/2}$ candidate messages (see Chapter 5). So, the total work for these birthday attacks is $3 \times 2.5 \times 2^{n/2}$ calls to the compression function. The second birthday attack involves four calls to the compression function for each candidate message, and therefore takes about $4 \times 2.5 \times 2^{n/2}$ calls to f. So the total expected number of calls to f is at most

$$(3 + 4 + 3) \times 2.5 \times 2^{n/2} = 25 \times 2^{n/2}.$$

In general, for a hash function based on Θ^l, we can find a 2^k-collision by choosing $l = 2k - 1$ (so that $\text{maxchain}(\Theta^l) = \lfloor \frac{l+1}{2} \rfloor = k$) and running k birthday attacks, in total calling the compression function about $2.5 \times 2^{n/2} \times |\Theta^l|$ times. Since $|\Theta^l| = 2l$, this is $5l \times 2^{n/2}$ times, which is $O(k2^{n/2})$.

Generalizing the above attack in a natural way results in the following theorem:

Theorem 7.1.1. [NS04] *Let F be a generalized sequential hash function based on a sequence α such that maxchain(α)= k. There exists an attack on F that produces a 2^k-collision in expected time $O(s2^{n/2})$, where $s = |\alpha|$.*

We also note that for classical iterated hash functions, $\alpha = \Psi^{(1,l)}$ and a maximum length chain is $1 \prec 2 \prec 3 \ldots \prec l$, so in the first Nandi and Stinson attack we would find an internal collision for *each* intermediate hash value by varying *each* message block. Thus the attack reduces to the Joux multicollision attack in this case.

The second Nandi and Stinson attack

The above attack method does not work against the "doubly iterated" hash function based on $\Psi^{(2,l)}$, because maxchain($\Psi^{(2,l)}$)= 1 (so the attack produces a 2-collision and is no better than the birthday attack). In the same paper [NS04] Nandi and Stinson give a different method of attack that produces a 2^k-collision for a hash function based on the sequence $\alpha^l = \Psi^{(2,l)}$. We call this hash function F and label its compression function f. In order to find a k-collision (where $k \ll n$) we do the following:

Step 1 We first choose the length l of the 2^k colliding messages we are going to find, by setting

$$t = \left\lceil \frac{n+1}{2} + \frac{\ln \ln 2k}{2 \ln 2} \right\rceil$$

and letting $l = kt$. (We will see shortly that for this t a birthday attack on a set of 2^t elements succeeds with probability $1 - \frac{1}{2k}$.)

Step 2 We then apply Joux's multicollision attack to a hash function based on the first half of α^l (i.e., the ordinary hash function based on $\Psi^{(1,l)}$), to obtain l pairs

$$(m_1, m_1'), (m_2, m_2'), \ldots, (m_l, m_l'),$$

such that

$$f(h_{i-1}, m_i) = f(h_{i-1}, m_i') = h_i, \qquad i = 1, 2, \ldots, l.$$

We now have a 2^l-collision on this hash function and a collision set

$$\mathcal{C} = \{m_1, m_1'\} \times \{m_2, m_2'\} \times \cdots \times \{m_l, m_l'\},$$

i.e., we have 2^l l-block messages of the form $m_1^* \| m_2^* \| \ldots \| m_l^*$ that collide on every intermediate chaining value from h_0 to h_l.

Step 3 We now divide the remaining index interval $[l+1, 2l]$ into k consecutive intervals of t elements each, and we search for a collision on each of these intervals: in other words, for each i from 1 to k, we search among the 2^t messages $m_{(i-1)t+1}^* \| m_{(i-1)t+2}^* \| \ldots \| m_{it}^*$ for two (say M_i and M_i') that take $h_{l+(i-1)t}$ to h_{l+it}. (Note that we can use the tree structure of these 2^t messages to find a collision between them (if it exists) with only $2 + 2^2 + 2^3 + \ldots + 2^t = 2(2^t - 1)$ calls to the compression function – two calls to find the two possible values of $h_{l+(i-1)t+1} = f(h_{l+(i-1)t}, m_{(i-1)t+1}^*)$, four calls to find the two possible values of $h_{l+(i-1)t+2}$ starting from each of these, and so on.)

If this step fails for any i, we return to Step 2 and search for a new collision set $\mathcal{C}$.

Step 4 If all k birthday attacks in the previous step succeed, then pasting together the collisions yields a set of 2^k messages of the form $M_1^* \| M_2^* \| \dots \| M_k^*$ that collide on each of the first l chaining values, and then in every tth chaining value starting at h_l, including the final value h_{2l}. In other words, the set

$$\mathcal{C}^* = \{M_1, M_1'\} \times \{M_2, M_2'\} \times \cdots \times \{M_k, M_k'\}$$

provides a 2^k-collision for a hash function based on $\Psi^{(2,l)}$. In the diagram below, $k = 3$ and $t = 2$, hence $l = 6$.

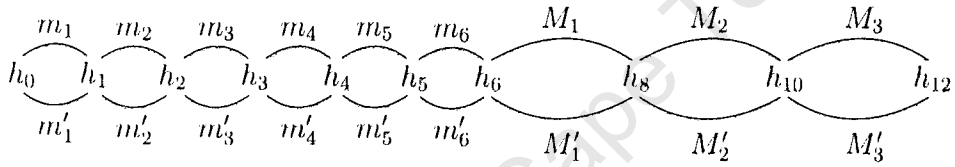


Figure 7.3: A 2^3-collision for the hash function based on $\Psi^{(2,l)}$

Work

By equation 2.2 the success probability for each birthday attack in Step 3 is approximately

$$1 - e^{-\frac{\left(2^t\right)^2}{2^{n+1}}} = 1 - e^{-2^{2t-(n+1)}}.$$

But since

$$t = \left\lceil \frac{n+1}{2} + \frac{\ln \ln 2k}{2 \ln 2} \right\rceil,$$

we have

$$2t \approx n + 1 + \log_2 (\ln 2k),$$

and

$$2^{2t-(n+1)} \approx \ln 2k,$$

so the success probability for each attack is approximately

$$1 - e^{-\ln 2k} = 1 - \frac{1}{2k}.$$

The probability that all k birthday attacks succeed is therefore

$$\left(1 - \frac{1}{2k}\right)^k > \frac{1}{2}.$$

The expected work for the multicollision attack in Step 2 is at most $2.5tk2^{n/2}$ (since it requires finding tk collisions, each of which can be expected to take at most $2.52^{n/2}$ calls to the compression function). In Step 3, the tree-structure of the 2^t possible messages in each interval means that finding a collision between them (if it exists) takes $2(2^t - 1)$ calls to f. Therefore, the work required for all k birthday attacks is about $2k2^t$, assuming they all succeed.

Step 3 is dependant on Step 2; if Step 3 fails, Step 2 has to be repeated. Since Step 3 succeeds with probability greater than $\frac{1}{2}$, we expect steps 2 and 3 to be repeated at most twice (using a geometric distribution argument), and the overall average case complexity of the attack is upper bounded by

$$2 \times (2.5kt2^{n/2} + 2k2^t).$$

Since t is $\Theta(n+\ln\ln k)$ and $2^t = 2^{\frac{n+1}{2}} \cdot 2^{\frac{1}{2}\log_2(\ln 2k)} = 2^{\frac{n+1}{2}} \cdot (\ln 2k)^{\frac{1}{2}}$ is $\Theta(2^{n/2}(\ln k)^{\frac{1}{2}})$, the overall expected complexity of the attack is $\mathrm{O}(k(n + \ln k)2^{n/2})$.

Generalising this second attack in a natural way gives the following theorem.

Theorem 7.1.2. *Let α be a sequence of length s on symbols $\{1, 2, \ldots, l\}$. Assume that the following hold:*

1. $\mathrm{freq}(\alpha) \leq 2$, *and*
2. *there is some initial interval $\alpha[1, w]$ of α in which there are kt symbols which appear exactly once, where $t = \lceil (n + 1)/2 + (\ln\ln 2k)/(2\ln 2) \rceil$.*

Then there exists an attack on any n-bit hash function based on α that produces a 2^k-collision in expected time $O(s \ln k 2^{n/2})$.

The attack works as follows. We choose kt "active" indices $i \leq w$ such that α_i appears only once in $\alpha[1, w]$; the corresponding active message blocks are used exactly once in the first w calls to the compression function; after that (since $\mathrm{freq}(\alpha) \leq 2$) some number j of them are not used again, and the others are used

exactly once more. We call them *active* blocks because they are the only blocks we will be varying to obtain the multicollision; we set all other message blocks to 0.

We begin by finding a 2-collision $f(h_{i-1}, m_i) = f(h_{i-1}, m_i') = h_i$ for each of the active indices i. This also involves evaluating the compression function for the inactive message blocks, and takes time $O((w - kt) + kt2^{\frac{n}{2}})$. (This gives us 2^{kt} l-block messages which collide on the first w chaining values, and out of these we need to find 2^k that collide on the final hash value h_s.) We can divide the remaining interval $[w, s]$ into $k - j$ intervals each containing at least t active message blocks. We now search for a collision on each of these intervals among the (at least) 2^t possibilities provided by choosing m_i or m_i' for each of the active blocks in the interval. If this birthday attack succeeds then it takes time $O(2^t)$ (using the tree-structure of the messages) plus the time taken for evaluating the compression function on the inactive message blocks in that interval, and we are left with two possibilities for that set of (at least) t active blocks. If one of these $k - j$ birthday attacks fails then we need to go back and find different collisions for those active indices, but t has been chosen so that this happens with probability less than $\frac{1}{2}$, so we expect to have to run the attack at most twice.

Once all $k - j$ birthday attacks have succeeded, we are left with two possibilities for each of the j active blocks that are only used once, and two possibilities for each of $k - j$ subsets of the other active blocks. Thus we have found $2^j \times 2^{k-j} = 2^k$ messages which collide on the first w chaining values and on k further chaining values, including the final one h_s. The attack takes time $O((w - kt) + kt2^{\frac{n}{2}} + (k - j)2^t + (s - w - (k - j)))$, which (since $kt \leq s$ and by our choice of t) is $O(s \ln k 2^{n/2})$.

We now show that if α is any sequence (containing a sufficiently large number of symbols) with $\text{freq}(\alpha) \leq 2$, then either the conditions of Theorem 7.1.1 or the conditions of Theorem 7.1.2 hold.

Theorem 7.1.3. *Let α be a sequence on $\mathcal{S} = \{1, 2, \ldots, l\}$ in which $1 \leq \text{freq}(x) \leq$*

2 for all $x \in \mathcal{S}$ (i.e., every element in S appears in α, but not more than twice). If $l \geq uv$, then one of the following holds:

1. *maxchain(α) $\geq u$, or*
2. *there is some initial interval $\alpha[1, w]$ of α in which there are v symbols which appear exactly once.*

Proof. Let maxchain(α) $= u_0$. If $u_0 \geq u$, then we are done, so suppose $u_0 < u$. Let v_0 denote the maximum number of mutually incomparable elements in S; then by Dilworth's Theorem 2.7.1 there is a chain decomposition of S into v_0 chains (i.e., every element of S appears in exactly one of the chains). But every chain contains at most maxchain(α) $= u_0$ symbols, so $|S| = l \leq u_0 v_0$, and it follows that $v_0 > l/u \geq v$. It is therefore possible to find v mutually incomparable elements in α, say $x_{i_1}, x_{i_2}, \ldots, x_{i_v}$. Let the first occurrence of x_{i_j} be at position a_j of α, for $i = 1, \ldots, v$, where $a_1 < \ldots < a_v$. Note that each of the symbols x_{i_j} for $j = 1, 2, \ldots, v-1$ must occur again in the sequence after position a_v, or we would have $x_{i_j} \prec x_{i_v}$. Since we know every symbol occurs at most twice in α, it follows that $x_{i_1}, x_{i_2}, \ldots, x_{i_v}$ all occur exactly once in the initial interval $\alpha[1, v]$. □

We are now ready to state, and prove, the main result by Nandi and Stinson, namely that there is a multi-collision attack for any generalised sequential hash function with frequency at most two.

Theorem 7.1.4. *Let F be a generalised sequential hash function based on the sequences $\langle \alpha^1, \alpha^2, \ldots \rangle$ such that freq(α^l) ≤ 2 for all $l \geq 1$. Then for any $k \ll n$ there exists a multicollision attack on F that produces a 2^k-collision in expected time $O(k^2 \ln k (n + \ln \ln k) 2^{n/2})$.*

Proof. Let $t = \lceil (n+1)/2 + (\ln \ln 2k)/(2 \ln 2) \rceil$, and note that $t = \mathrm{O}(n + \ln \ln k)$. Let $l = k^2 t$. By Theorem 7.1.3 one of the following holds:

1. maxchain(α)$\geq k$, or
2. there is some initial interval $\alpha[1, w]$ of α in which there are kt symbols which appear exactly once.

If condition 1 holds, then by Theorem 7.1.1 we can use Nandi and Stinson's first attack technique to obtain a 2^k-collision on F' in expected time $O(s2^{n/2})$, where $s = |\alpha^l|$. If condition 2 holds, then by Theorem 7.1.2 we can use Nandi and Stinson's second attack technique to obtain a 2^k-collision on F' in expected time $O(s \ln k 2^{n/2})$. Since $s \leq 2l = 2k^2t$, both attacks are $O(k^2 \ln k(n+\ln \ln k))$. □

The work by Nandi and Stinson shows that generalised sequential hash functions are insecure against multicollision attacks if the message blocks are processed at most twice. In fact, Hoch and Shamir show in [HS06a] how to construct a multicollision if message blocks are processed up to e times for some fixed e. So generalised sequential hash functions are not secure against multicollision attacks.

7.2 Dithered hash functions

Dithered hash functions were invented by Ron Rivest [Riv05] as a response to the second preimage attacks using expandable messages, namely, the attack by Dean [Dea99], and more recently, the attack by Kelsey and Schneier [KS05]. These attacks exploit the repetition of the same message block to produce second preimages with less than the expected brute-force amount of work.

The idea behind dithered hashing is to amend the hashing process to include an additional input to the compression function. This input is determined by a fixed dithering[1] sequence and aims to put a halt to the use of repetitive compression function inputs. By making use of a dithering sequence, the intermediate hash value of a message block becomes dependant on the block's position within the message. This means that if the attacker finds a collision in the compression function it doesn't help him unless the dithering value is also correct.

[1]The term *dithering* originates from the field of image-processing. It refers to the representation of a variety of gray or coloured shades by mixing together pixels of a small number of basic shades. These pixels are combined randomly so as to prevent distinct patterns of colour from being visible.

Since a hash function should be able to process messages of arbitrary length, it is reasonable to consider infinite sequences as candidates for the dithering sequence d. We let d be a infinite word over a finite alphabet $\mathcal{A}$, and denote the ith element of d by d_i. Staying within the Merkle-Damgård paradigm, we obtain a dithered iterated hash function F' by setting

$$h_i = f(h_{i-1}, m_i, d_i).$$

Figure 7.4 depicts the basic structure of an iterated hash function with dithering. In the figure $M = m_1 \| m_2 \| \dots \| m_k$.

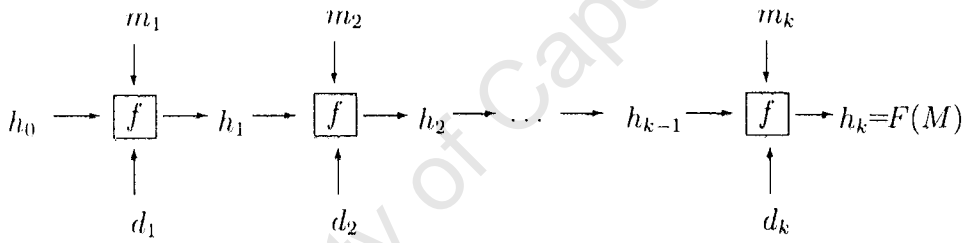


Figure 7.4: Iterated hash function with dithering

We now discuss various options for dithering:

Dithering with a counter

A simple way of dithering is to let the dithering sequence d be nothing more than a simple running counter, i.e.,

$$d_i = i. \tag{7.1}$$

Kelsey and Schneier suggest this in [KS05]. However, Rivest [Riv05] points out that the compression function f would need to be able to accept arbitrarily large inputs i since M is allowed to be of arbitrary length. He suggests that it would be easier if the dithering input d_i was restricted to a small finite alphabet.

Before we move on to discuss the next dithering option, we briefly mention a

construction by Eli Biham and Orr Dunkelman known as HAIFA [BD06]. At any time t, the compression function of HAIFA takes in as additional input the number of bits hashed up to time t, including the message block that is processed by the compression function at time t. This is, in essence, identical to the counter dithering scheme with the counter internally tracking the length of the message processed thus far.

Dithering by alternating with 0's and 1's

Another fairly simple way to dither is to set

$$d_i = \begin{cases} 0 & \text{if } i \text{ is even, and} \\ 1 & \text{otherwise.} \end{cases}$$

Although effective in preventing the use of repeated single message blocks, this dithering sequence does not prevent the use of repeated pairs of blocks. This will not be of any value against an adversary that can find fixed "pairs", i.e., message block pairs (m_1, m_2) for which $f(f(h_0, m_1), m_2) = h_0$. (This is like the attacker treating two message blocks as one block and two iterations of the compression function as one function). Also, if the attacker is able to build expandable messages by using pairs of message blocks, for instance, using a pair of message blocks every time a single block is used in the Kelsey-Schneier [KS05] technique (see Section 6.3.2), then this scheme is of little use.

Dithering with a pseudorandom sequence

We could generate a pseudorandom sequence $s = s_1, s_2, \ldots$ and set $d_i = s_i$ for all relevant i. However, this would only protect weakly against the use of repeated message blocks. The longer the sequence, the more likely the occurrence of repeating subsequences; once the cycle of the sequence is found, it may be possible to repeat chunks of message blocks with size μ (the cycle length) to build expandable messages, or even find fixed "chunks", i.e., a collection of message blocks $(m_1, m_2, \ldots, m_\mu)$ such that $f(\ldots(f(f(h_0, m_1), m_2))\ldots, m_\mu) = h_0$.

Dithering with abelian square-free sequences

Due to the apparent inadequacies inherent in all of the dithering methods described thus far, Rivest [Riv05] suggests the use of an abelian square-free sequence (see Chapter 2, Section 2.8) for d. These are aperiodic sequences over a finite alphabet which have no repeated subwords, and additionally, contain no permutations of any subword in the sequence. So use of such sequences prevents an attacker from being able to use message block repetition to his or her advantage, even by repeating a chunk of several blocks. Square-free sequences, i.e., those containing no repeat subwords, would work perfectly well as dithering sequences. However, Rivest suggests the use of abelian square-free sequences since they are more "repetition-free" than ordinary square-free sequences, and no harder to generate.

In [Riv05], Rivest puts forward two dithered hash function proposals that make use of abelian square-free sequences. The first suggests the direct use of the Keränen sequence (see Chapter 2, Section 2.8.2) as the dithering sequence, d. Dithering inputs are taken from the alphabet $\mathcal{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$, and each symbol is encoded by two bits:

$$\mathsf{a} \rightarrow \mathsf{00}, \mathsf{b} \rightarrow \mathsf{01}, \mathsf{c} \rightarrow \mathsf{10}, \mathsf{d} \rightarrow \mathsf{11}.$$

The number of bits in each message block is thus reduced by two bits in order to make room for the two-bit dither input. Each element of the Keränen sequence can be generated online (symbol by symbol) and can thus be processed efficiently by the compression function.

Rivest imposes a slight modification on the scheme suggested above and proposes a second hash function, henceforth referred to as RCP (Rivest's Concrete Proposal): For a k-block message M, each dithering input d_i is sixteen bits long and has the form

$$d_i = \left(0, \kappa_j, i \bmod 2^{13}\right) \in \{0,1\} \times \mathcal{A} \times \{0,1\}^{13},$$

where $j = \lfloor i/2^{13} \rfloor$, for i running from 1 to $k-1$, and where κ_j is the j^{th} term of the

Keränen sequence. In this construction, the next letter of the Keränen sequence is only utilized when the counter j increases by 1 (so κ_j stays the same for 2^{13} values of i). This means that the dithering sequence used in RCP is generated 2^{13} times faster than the Keränen sequence.

The last dithering symbol is slightly different to take care of padding, and has the form

$$d_k = (1, |M| \bmod |m^*|) \in \{0,1\} \times \{0,1\}^{15},$$

where $|m^*|$ denotes the number of message bits (as opposed to padding bits) in a message block.

The use of the last dithering input, d_k, also eliminates the need for Merkle-Damgård strengthening because the last dithering symbol depends depends on both the number of messages blocks and the number of message bits in the last block, and will therefore be affected by any change in the message length.

7.2.1 A second preimage attack on dithered hash functions

In this section we describe how Andreeva *et al.* [ABF+08] adapt their second preimage attack (see Chapter 5, Section 6.5) to work on dithered hash functions. Recall that the attack mentioned in Chapter 5 combines the herding attack of Kelsey and Schneier [KS05] with the long-message attack described in [MvOV96]. This modified attack follows roughly the same procedure, but contains alterations to accommodate for some dithering sequence d.

When building the diamond, the attacker chooses a dithering symbol for each layer in the structure, i.e., for a diamond of depth k he fixes a k-tuple of dithering symbols $(\omega_1, \omega_2, \ldots, \omega_k)$ (only positions in the long message that use the subsequence $\omega = \omega_1\omega_2 \ldots \omega_k$ as dithering symbols will be candidates for him to join his diamond to). He also chooses an additional dithering symbol to be used when trying to connect to the target message M_{target}. Figure 7.5 depicts a diamond

structure with $k = 3$ and $\omega = \sigma\gamma\mu\delta$.

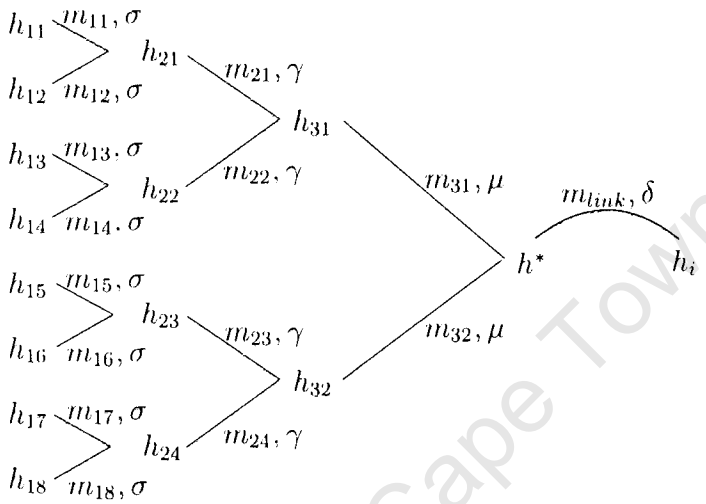


Figure 7.5: Diamond structure with $k = 3$ and $\omega = \sigma\gamma\mu\delta$

For a dithered hash function F, the dithering sequence ensures that the hash value of a message block is dependant on its position within the message. This means that our diamond can only be connected to M at the positions where ω and d match. The set of matching positions is given by

$$\mathcal{R} = \{i \in \mathbb{N} | i \geq k + 1 | (d_{i-k} \dots d_i) = \omega)\},$$

and the attacker knows $\mathcal{R}$ because he knows the dithering sequence d. To ensure that $\mathcal{R}$ is non-empty, ω has to be a factor of d, i.e., $\omega_1\omega_2, \dots, \omega_{k+1}$ has to occur somewhere in the dithering sequence. In fact, we would like ω to be the most frequently occurring factor of size $k+1$ in d. This is because the cost of the attack ultimately depends on the number of message blocks tried when searching for a message block that connects the diamond to some intermediate hash value, h_i, produced during the computation of $F(M)$. The chances that $i \in \mathcal{R}$ are higher if ω is the most popular factor of d.

We have a dithered hash function F with a dithering sequence d and we want to

find a second preimage M' for a target message $M_{target} = m_1 \| m_2 \| \ldots \| m_{2^l-1} \| m_{2^l}$ such that $F(M) = h_{2^l}$. The attack proceeds in almost the same way as the undithered attack described in Section 6.5, which used a diamond structure to produce an M' of the form $M' = P \| D \| m_{link} \| m_{i+1} \| \ldots \| m_{2^l}$ which has the same length as M and hence the same length field.

To make the attack work against dithered hash functions, we need to make sure that when we find our linking message m_{link} and h_i, the k dithering symbols used just before that, and the dithering symbol used for the calculation of m_{link}, are the same in both message paths in figure 6.14. So we're going to need $\omega_{i-k}, \omega_{i-k+1}, \ldots, \omega_i$ to be the same as the dithering symbols $d_{i-k}, d_{i-k+1}, \ldots, d_i$ used when constructing the diamond — this is the essence of the new attack. (The prefix P has the same length as the segment of M_{target} before h_i, and so will automatically have the correct dithering sequence.) We do the following:

Step 1 We select the most popular $(k+1)$-sized factor of d. We denote this factor by ω.

Step 2 We use elements of ω as dithering symbols and build a diamond structure of depth k, i.e., there are 2^k intermediate hash states in the diamond's widest layer. All possible paths of k message block and dithering symbol pairs lead to the value h^*. For construction of the diamond see Chapter 6, Section 6.4.

Step 3 We search for a linking message block, m_{link}, that connects h^* to one of the 2^l hash values produced during the computation of $F(M_{target})$. To do this, we use ω_{k+1} as the dithering symbol and randomly consider options for m_{link} until $f(h^*, m_{link}, \omega_{k+1}) = h_i$, for some $i \in [k+1, 2^l]$, and such that $i \in \mathcal{R}$, i.e., such that $d_{i-k,\ldots,di} = \omega$.

Step 4 We generate a prefix P of size $i-k-1$ blocks such that P hashes from h_0 to one of the 2^k hash values in the widest layer of our diamond. We do this by trying different random values of P with the known dithering

sequence $d_1, d_2, \ldots, d_{k-i-1}$. We let D denote the chain of k message block and dithering symbol pairs leading from $F(P)$ to h^*.

Step 5 We output the message $M' = P\|D\|m_{link}\|m_{i+1}\|\ldots\|m_{2^l}$.

In the event that $i = 2^l$, no message blocks of M will be used in M'. The prefix selected in Step 3 is $2^l - l - 1$ blocks long and $M' = P\|D\|m_{link}$.

Work

In the worst case for the attacker, all factors of size $k+1$ occur in d with the same frequency. This means that the probability of a $(k+1)$-letter subword being ω is $1/Fact_d(k+1)$, where $Fact_d(k+1)$ is the number of factors of length $k+1$ in the sequence d. In other words, the probability that i is an element of $\mathcal{R}$ in Step 3 is $1/Fact_d(k+1)$. This means that in the worst case, we will have to repeat the search for m_{link} $Fact_d(k+1)$ times because all factors of length $(k+1)$ are equally likely, and thus each are ω with probability $1/Fact_d(k+1)$. We will therefore need to consider at most $1/(1/Fact_d(k+1)) = Fact_d(k+1)$ candidates for ω (see Chapter 2, Section 2.5). Since we have to consider 2^{n-l} possibilities for m_{link} each time, this brings the total work for Step 3 to

$$Fact_d(k+1) \times 2^{n-l}.$$

Building the diamond structure in Step 2 requires requires $2^{n/2+k/2+2}$ work (see Chapter 5, Section 6.4), and generating the prefix in Step 4 can be done with 2^{n-k} work. This brings the total work for the attack to

$$\begin{array}{ccccc} & \text{Step 2} & & \text{Step 3} & & \text{Step 4} \\ 2^{n/2+k/2+2} & + & Fact_d(k+1) \times 2^{n-l} & + & 2^{n-k}. \end{array}$$

In the case of Rivest's first proposal, the cost of the attack depends on the number $Fact_\kappa(k+1)$ of distinct factors of length $(k+1)$ occurring in the Keränen sequence κ. This number is fairly low due to the rather regular structure of κ. For $k \leq 85$ it is shown in [ABF+08] that

$$Fact_\kappa(k) \leq 8k + 332,$$

and therefore

$$Fact_\kappa(k+1) \leq 8k + 340.$$

If $n \approx 3l$, then the work required for Step 2 is roughly equal to the work required for Steps 3 and 4 respectively. Note that this approximation is more accurate if l and k are of similar magnitude. If $n \gg 3l$, then the work for the attack is dominated by Steps 3 and 4, and we can ignore the work required for Step 1. Assuming this to be the case, and by setting $k = l - 3$, the work for the attack on Rivest's first proposal is

$$(8(l-3) + 340) \times 2^{n-l} + 2^{n-l+3} = (l + 40.5) \times 2^{n-l+3}.$$

This is much smaller than the expected brute-force amount of 2^n.

In the case of RCP, the amount of work required is larger due to the higher number of distinct factors of length $(k+1)$ occurring in the dithering sequence used. This sequence d, derived from the Keränen sequence by diluting it with a 13-bit counter (i.e., we only move on to the next symbol of the Keränen sequence when the 13-bit counter overflows), has a complexity of

$$Fact_d(k) = 8k + 32760$$

for all $0 \leq l < 2^{13}$. Therefore, the work required to run the attack on RCP is

$$2^{n/2+k/2+2} + (8k + 32768) \times 2^{n-l} + 2^{n-k}.$$

Again assuming that $n \gg 3l$, and by setting $k = l - 3$, we obtain a complexity of roughly

$$(l + 4094) \times 2^{n-l+3} \approx 2^{n-k+15};$$

yet again beating a brute-force attack.

What is evident from the above is that the use of a diluting counter increases the complexity of the attack; in the worst case we have more distinct factors of length $(k+1)$ occurring with equal probability, and hence we have to try more

candidate messages for ω (see Chapter 2, section 2.5). If we use the dithering sequence given by equation 7.1 for a counter over t bits, then for $t \leq k$, the number of factors of size k is $Fact_d(k) = 2^t$, and the complexity of the attack becomes

$$2^{n/2+k/2+2} + 2^{n-l+t} + 2^{n-k}. \tag{7.2}$$

If we assume, as is common in practice, that the work is dominated by the second term of equation 7.2, then by setting $t = l$ the complexity becomes 2^n and we do no better than a brute force attack. This is essentially how HAIFA is constructed and is therefore immune to the Andreeva *et al.* attack. As a means of improving Rivest's constructions, the authors of [ABF+08] suggest the use of a dithering sequence of high complexity over a small alphabet. They prove the existence of an abelian square-free sequence over six letters with $(k+1)$-complexity greater than $2^{k/2}$, and claim that the cost of an online attack (an attack that processes inputs in turn, without having the entire input available from the start) is roughly $2^{n-2l/3}$ for $k = 2l/3$.

Andreeva *et al.* [ABF+08] modify the attack to work on any dithering scheme over a small alphabet. This is done by replacing the diamond with a more complicated search structure know as a *kite generator*. They make use of a time-memory tradeoff that helps to produce a second preimage with only a small amount of online computation; although there is an expensive precomputation stage. In the worst case, this modified attack finds a second preimage of a 2^l-block message in time $max\left(2^l, 2^{(n-l)/2}\right)$.

Chapter 8

Conclusion

In this dissertation we present a body of evidence that shows that the Merkle-Damgård hash function design is not as secure as was previously hoped. Moreover, two of the most natural ways of attempting to fix the construction have been shown not to prevent the attacks. It is thus questionable whether or not an iterative process is optimal for hash function design.

In response to the call for proposals by the NIST for a new hash algorithm, a team from MIT led by Ronald Rivest has recently submitted MD6 [Riv08]. MD6 is not based on the Merkle-Damgård construction for hash functions, but rather, is a tree-based hash function. The input to the compression function of MD6 is very large, and the intermediate chaining values are 1024 bits long. The final output of MD6 has a length of 512 bits, which means that it truncates the final chaining value computed when producing the hash value.

To date, there has not been sufficient time for the cryptographic community to review MD6 in the hope of exposing the algorithm's potential weaknesses. It might be that there are none, in which case algorithms such as these could prove to be the future of hash function design.

Bibliography

[AB96] Ross Anderson and Eli Biham. Tiger - A Fast New Hash Function. In *Fast Software Encryption Workshop Proceedings - FSE 1996*, volume 1039 of *Lecture Notes in Computer Science*, pages 89–97. Springer-Verlag, 1996.

[ABF+08] Elena Andreeva, Charles Bouillaguet, Pierre-Alain Fouque, Jonathan J. Hoch, John Kelsey, Adi Shamir, and Sebastien Zimmer. Second Preimage Attacks on Dithered Hash functions. In *Advances in Cryptology - EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

[ACPZ01] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato. Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). Technical Report RCF 3161, The Internet Engineering Task Force, August 2001. Available at http://tools.ietf.org.

[BD06] Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions - HAIFA. Presented at the Second NIST Hash Workshop, August 2006.

[Ber95] J. Berstel. *Axel Thue's Papers on Repetitions in Words: a Translation.* Number 20 in Publications du Laboratoire de Combinatoire et d'Informatique Mathématique. Université du Quebec a Montreal, Montreal, Canada, 1995.

[BK07] Elain Barker and John Kelsey. Recommendation for Random Number Generation using Deterministic Random Bit Generators (Revised). Technical Report SP 800-90, National Institute of Standards and Technology, March 2007.

[Bre80] Richard M. Brent. An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics*, 20:176–184, 1980.

[CDMP05] Jean-Sebastien Coron, Yevgeniy Dodis, Cecil Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[CSPM07] Scott Contini, Ron Steinfeld, Josef Pieprzyk, and Krystian Matusiewicz. A Critical Look at Cryptographic Hash Function Literature. In *ECRYPT Hash Workshop*, 2007.

[Dav82] G. Davida. Chosen signature cryptanalysis of the RSA (MIT) public key cryptosystem. Technical Report TC-CS-82-2, 1982.

[Dea99] Richard D. Dean. *Formal Aspects of Mobile Code Security.* PhD thesis, Princeton University, January 1999.

[Den84] D.E. Denning. Digital signatures with RSA and other public-key cryptosystems. *Communications of the ACM*, 27(4):388–392, 1984.

[DH76] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[DL05] Magnus Daum and Stefan Lucks. The Story of Alice and her Boss. Presented at the Rump Session of EUROCRYPT 2005 held from 22-26 May, Aurhus, Denmark, 2005.

[DO86] Y.G. Desmedt and A.M. Odlyzko. A Chosen Attack on the RSA Cryptosystem and Some Discrete Logarithm Schemes. In *Advances in Cryptology - CRYTPO 1985*, volume 218 of *Lecture Notes in Computer Science*, pages 512–522. Springer-Verlag, 1986.

[Dr89] Ivan B. Damgå rd. A Design Principle for Hash Functions. In *Advances in Cryptology - CRYTPO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1989.

[Flo67] R.W. Floyd. Non-deterministic Algorithms. *Journal of the Association for Computing Machinery*, 14(4):636–644, 1967.

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography.* Wiley Publishing, Inc., 2003.

[Gir88] M. Girault. Hash Functions Using Modulo-n Operations. In *Advances in Cryptology - EuroCrypt 1987*, volume 304 of *Lecture Notes in Computer Sciene*. Springer, 1988.

[GKP98] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley Publishing Company, Inc., New York, United States of America, second edition, 1998.

[HS06a] Jonathan J. Hoch and Adi Shamir. Breaking the ICE - Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions. In *Fast Software Encryption – FSE 2006*, volume 4047 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2006.

[HS06b] Jonathan J. Hoch and Adi Shamir. Breaking the ICE - Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions. Presented at Fast Software Encryption held from 15 -17 of March, 2005 in Graz, Austria., March 2006.

[Jou04] Antoinne Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In *Advances in Cryptology - CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.

[Kal00] B. Kaliski. PKCS5 v2.0: Password-Based Cryptography Specification Version 2.0. Technical Report RFC 2898, RSA Laboratories, Bedford, MA, September 2000.

[Ker92] Viekko Keränen. Abelian Squares are Avoidable on 4 Letters. In W. Kuich, editor, *19th International Colloquium on Automata, Languages and Programming - ICALP 1992*, volume 623 of *Lecture Notes in Computer Science*, pages 41–52. Springer, July 1992.

[Ker03] Viekko Keränen. On abelian square-free DT0L-languages over 4 letters. In T. Harju, editor, *4th International Conference on Words - Words 2003*,

volume 27 of *TUCS General Publication*, pages 95–109. Turku Centre for Computer Science, 2003.

[KK06] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In *Advances in Cryptology - Eurocrypt 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[Knu68] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1968.

[KS05] John Kelsey and Bruce Schneir. Second Preimages on n-bit Hash Functions for Much Less than 2^n Work. In *Advances in Cryptology - Eurocrypt 2005*, volume 3494 of *Lecture Notes in Computer Science*. Springer, 2005.

[Mer79] Ralph C. Merkle. *Security, Athentication and Public Key Systems*. PhD thesis, Stanford University, 1979.

[Mer89] Ralph C. Merkle. One Way Hash Functions and DES. In *Advances in Cryptology - CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[Mir05] Ilya Mironov. Hash functions: Theory, attacks, and applications. Technical Report MSR-TR-2005-187, Microsoft Research, 2005. Available at http://research.microsoft.com/research/pubs/view.aspx?

[Mit07] C.J. Mitchell. Generic collision attacks on hash-functions and HMAC. Presented at The Claude Shannon Workshop of Coding and Cryptography held on the 21st and 22nd of May, 2007 at the University of Cork, Ireland., May 2007.

[MOI91] Shoji Miyaguchi, Kazuo Ohta, and Masahiko Iwata. Confirmation that Some Hash Functions are Not Collision Free. In *Advances in Cryptology - EUROCRYPT 1990*, volume 473 of *Lecture Notes in Computer Science*, pages 326–343. Springer-Verlag, 1991.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[NIS93] Secure Hash Standard. Technical Report FIPS 180, National Institute of Standards and Technology, May 1993. Available from http://www.csrc.nist.gov.

[NIS95] Secure Hash Standard. Technical Report FIPS 180-1, National Institute of Standards and Technology, April 1995. Available from http://www.csrc.nist.gov.

[NIS99] Data Encryption Standard (DES). Technical Report FIPS PUB 46-3, National Institute of Standards and Technology, October 1999. Publication reaffirmed 25 October 1999.

[NIS01] Introducing the Advanced Encryption Standard(AES). Technical Report FIPS PUB 197, National Institute of Standards and Technology, November 2001. Publication dated 26 November 2001.

[NIS07] Federal Register Notice. *Federal Register*, 72(212), November 2007. Notice dated 2 November 2007.

[NS88] K. Nishimura and M. Sibuya. Occupancy with Two Types of Balls. *Ann. Inst. Statist. Math.*, 44(1):77–91, 1988.

[NS90] K. Nishimura and M. Sibuya. Probability To Meet in the Middle. *Journal of Cryptology*, 2(1):13–22, 1990.

[NS04] M. Nandi and D. Stinson. Multicollision Attacks on Generalized Sequential Hash Functions. Report 2004/330, IACR Cryptology ePrint Archive, 2004. http://eprint.iacr.org/2004/330.

[PGV93] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In *Advances in Cryptology - Crypto 1993*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378, New York, NY, USA, 1993. Springer-Verlag.

[Ple70] P.A.B. Pleasants. Non-repetative sequences. *Proc. Cambridge Phil. Soc.*, 68:267–274, 1970.

[Pol75] J.M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15:331–334, 1975.

[Pre93] Bart Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit, Leuven, Febuary 1993.

[QD89] Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search? Application to DES. In *Advances in Cryptology - EUROCRYPT 1989*, volume 434 of *Lecture Notes in Computer Science*, pages 429 434. Springer-Verlag, 1989.

[Rab78] M. Rabin. Digitalized signatures. In *Foundations of Secure Communications*, pages 155 168. Academic Press, 1978.

[Riv92a] Ronald L. Rivest. The MD4 Message Digest Algorithm. Technical Report RFC1320, MIT Laboratory for Computer Science and RSA Data Security Inc., April 1992. Available at http://www.ietf.org/rfc/rfc1320.txt.

[Riv92b] Ronald L. Rivest. The MD5 Message Digest Algorithm. Technical Report RFC1321, MIT Laboratory for Computer Science and RSA Data Security Inc., April 1992. Available at http://www.ietf.org/rfc/rfc1321.txt.

[Riv05] Ronald L. Rivest. Abelian square-free dithering for iterated hash functions. Presented at the ECrypt Hash Function Workshop on June 21, 2005 in Cracow, and at the Cryptographic Hash Workshop on November 1, 2005 in Gaithersburg, Maryland., August 2005.

[Riv08] Ronald L. Rivest. The MD6 Hash Function (aka "Pumpkin Hash"). Presented as an invited talk on 20 August at CRYPTO 2008 in Santa Barbara, California, August 2008.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120 126, 1978.

[SSY82] R. Sedgewick, T.G. Szymanski, and A.C. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376 390, February 1982.

[Sti06] Douglas R. Stinson. *Cryptography: Theory and Practice*. Chapman and Hall/CRC, third edition, 2006.

[VOW99] Paul C. Van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12:1 28, 1999.

[WGLY04] X. Wang, F. Guo, X. Lai, and H. Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Presented at the CRYPTO 2004 rump session on August 17, 2004. Available at http://eprint.iacr.org/2004/199., August 2004.

[Win83] R.S. Winternitz. Producing a one-way hash function from DES. In *Advances in Cryptology - CRYPTO 1983*, Lecture Notes in Computer Science, pages 207 2 7. Springer-Verlag, 1983.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In *Advances in Cryptology - EUROCRYPT 2005*, volume 4965 of *Lecture Notes in Computer Science*, pages 19-35. Springer, 2005.

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

Appendix A: BABI Source Code

We present the Java source code for BABI. The method 'BigSquareRoot' was taken from http://www.merriampark.com/bigsqrt.htm.

```
import java.math.BigDecimal;
import java.math.BigInteger;
import java.math.MathContext;
import java.math.RoundingMode;
import java.util.Scanner;
import java.util.Vector;
public class BABI {

   public static BigInteger p,q,N;

   public static Scanner input;

   public static BigInteger buffer;
       public static int msgBlockLength, chainVarLength, inputLength,
       initialMsgLength, lengthField;

   public static String msg, IV = "";

   public static Vector<String> msgBlocks;


//=========================================================
   /* This method checks for primality of the user input for p and
    q.*/
   public static BigInteger getPrime()
```

```
    {
        BigInteger curr = new BigInteger("4");

        while (!curr.isProbablePrime(10))
        {
            if (curr.toString().compareTo("4") != 0)
            {
                                System.out.println("Number is not prime, please
                                re-enter");
            }
            curr = new BigInteger(input.next());
        }
        return curr;
    }

//==========================================================

//==========================================================
    public static void calcBufferAndLengths()
    {
        BigSquareRoot app = new BigSquareRoot ();
        BigDecimal sqrt = app.get(N);
                BigDecimal bufferDecimal = sqrt.round(new
                MathContext(1,RoundingMode.CEILING));
        buffer = bufferDecimal.toBigInteger();

BigDecimal NDivide2 = new BigDecimal(N).divide(new
BigDecimal("2")).round(new MathContext(1,RoundingMode.FLOOR));
BigInteger interval = NDivide2.subtract(bufferDecimal).toBigInteger();
```

```
            inputLength = interval.bitLength()-1;

        msgBlockLength = (int) Math.ceil(inputLength/2) + 1;

        chainVarLength = (int) Math.floor(inputLength/2) - 1;

        lengthField = (msgBlockLength * 4);

        for (int i = 0; i < chainVarLength; i++)
        {
           IV += "0";
        }

        //IV = "";

            System.out.println("Length of chaining variable: " +
            chainVarLength);
            System.out.println("Length of message block: " + msgBlockLength);
    }

//=============================================================

//=============================================================
     /* This method appends leading zeros onto bytes where necessary, adds
     padding if required and appends the length to the end of the message.*/
    public static String messageModification(byte[] msgBytes)
    {
        Byte newS;
        String msgBits = "";

        System.out.println("The hex value of your message is: ");
```

```
   for (int i = 0; i < msgBytes.length; i++)
   {
      newS = (Byte)(msgBytes[?] );
System.out.print(Integer.toHexString(newS.intValue()) + " ");
   }
   System.out.println("");
   //Appends leading zeros if necessary.
   for (int i = 0; i < msgBytes.length; i++)
   {
newS = (Byte)(msgBytes[?] ); //Casts each byte to a byte object.

            //Storing the binary representation of each byte in a string buffer.
            StringBuffer tempMsgBits = new
            StringBuffer(Integer.toBinaryString(newS.intValue()));

      //Appending leading zeros.
      for (int j = 0; j < 8 - tempMsgBits.length(); j++) {
         tempMsgBits.insert(j, "0");
      }

            msgBits += tempMsgBits.toString(); /*Augmenting the message
            with the required number of zeros.*/
   }

            /*Getting a binary representation of the message length and
            appending zeros so that it satisfies the length field.*/
   initialMsgLength = msgBits.length();
            StringBuffer tempMsgLength = new
            StringBuffer(Integer.toBinaryString(msgBits.length()));
   int tempLength = tempMsgLength.length();
   for (int j = 0; j < (lengthField - tempLength); j++)
```

```
    {
       tempMsgLength.insert(j, "0");
    }
    int remainder = (initialMsgLength % msgBlockLength);

            if ( remainder == 0) /*If the message is a multiple of
            msgBlockLength, then no superficial padding is added, just the
            length.*/
    {
       msgBits += tempMsgLength.toString();
    }
            else /*Superficial padding is added, followed by the length.*/
    {
       msgBits += "1";

    for (int i = 0; i < (msgBlockLength - remainder - 1); i++)
       {
          msgBits += "0";
       }
       msgBits += tempMsgLength.toString();
    }


    return msgBits;
  }

//=============================================================

//=============================================================
  public static void storeMessage(String message)
  {
```

```
        StringBuffer tempMsg = new StringBuffer(message);
        tempMsg.insert(0, "0");

        msgBlocks = new Vector<String>();
        String tempS = "";

        for (int i = 1; i < tempMsg.length(); i++) {
            if(i % msgBlockLength == 0)
            {
                tempS += tempMsg.charAt(i);
                msgBlocks.add(tempS);
                tempS = "";
            }
            else
            {
                tempS += tempMsg.charAt(i);
            }
        }

        for (int i = 0; i < msgBlocks.size(); i++) {
            System.out.print(msgBlocks.elementAt(i) + " ");
        }
        System.out.println("");
    }

//==========================================================

//==========================================================
public static String compressionFuction(String hiMinus1, String mi)
    {
        String paramConcat = hiMinus1 + mi;
```

```
        int decimalRep = 0;

    for (int i = 0; i < paramConcat.length(); i++) {
     if(paramConcat.charAt(i) == '1')
     {
     decimalRep += Math.pow(2.0, (paramConcat.length() - 1 - i));
     }
     }

         long preCompressed = (long)Math.pow((double)(buffer.intValue() +
         decimalRep), 2.0);

                 preCompressed = preCompressed % N.intValue();
                 StringBuffer compressedValue = new
                 StringBuffer(Long.toBinaryString(preCompressed));


           StringBuffer hi = new StringBuffer("");

for (int i = compressedValue.length()-1; i
>(compressedValue.length()-1-chainVarLength); i--)
       {
          hi.insert(0,compressedValue.charAt(i));
       }

       return hi.toString();
    }

//=============================================================

//=============================================================
```

```
public static String babiHash()
{
    String hi = IV,
        tempH = "";
    for (int i = 0; i < msgBlocks.size(); i++)
    {
        tempH = compressionFuction(hi,msgBlocks.elementAt(i));
        hi = tempH;
        System.out.println(hi + " ");
        System.out.println();
    }
    System.out.println();
    return hi;
}

//=============================================================
    /**
     * @param args
     */
    public static void main(String[] args) {

        input = new Scanner(System.in);

        //Entering in p and q, and determining N.
        //=====================================
        System.out.println("Enter p: ");
        p = getPrime();

        System.out.println("Enter q: ");
        q = getPrime();
```

```
        N = p.multiply(q);
        //=============================================

        System.out.println(p + " " + q + " " + N);

                /*Calculating the buffer, the message block length and chaining
                value length.*/
        //=============================================
        calcBufferAndLengths();
        //=============================================

        msg = input.nextLine();
        msg = "";


        while (msg.compareTo("quit") != 0)
        {
           System.out.println("Enter the message: ");
           msg = input.nextLine();

           byte[] msgBytes = msg.getBytes();
           storeMessage(messageModification(msgBytes));

           System.out.println(babiHash());
           System.out.println("");

        }

    }
}
```

Appendix B: MD5 Collision Files

We present the postscript code (in ascii) for the documents D and D' mentioned in Chapter 5, Section 5.8. $R2'$ is indicated in bold and the second instruction set is indicated in italics.

D:

```
%!PS-Adobe-1.0
%%BoundingBox: 0 0 612 792
(hB&j^M□`U _xe□A'/J7□
F~*@□T>1{□□Em3□□|S`[o □□#xRmZ3N6□
□□nEZR□yk/=□□uWev:1?*□
YJ5sYH2t}
,9#vT6- }o;□!r}kJ6□S□XkHt□tNa|□□`Oo□5□□)(hB&j^M□`U _xe□A'/J7□
F~*@□T>1{□□Em3□□|S`[o □□#xRmZ3N6□
□□nEZR□yk/=□□uWev:1?*□
YJ5sYH2t}
,9#vT6- }o;□!r}kJ6□S□XkHt□tNa|□□`Oo□5□□)eq{/Times-Roman findfont 20
scalefont setfont
300 700 moveto (Julius. Caesar) show
300 680 moveto (Via Appia 1) show
300 660 moveto (Rome, The Roman Empire) show
25 500 moveto (May, 22, 2005) show
25 450 moveto (To Whom it May Concern:) show

25 400 moveto
(Alice Falbala fulfilled all the requirements of the Roman Empire)
show
25 380 moveto
(intern position. She was excellent at translating roman into her gaul)
show
25 360 moveto
(native language, learned very rapidly, and worked with considerable)
show
25 340 moveto
(independence and confidence.)
show

25 300 moveto
(Her basic work habits such as punctuality, interpersonal deportment,)
show
25 280 moveto
(communication skills, and completing assigned and self-determined)
show
25 260 moveto
(goals were all excellent.)
show
```

```
25 220 moveto
(I recommend Alice for challenging positions in which creativity,)
show
25 200 moveto
(reliability, and language skills are required.)
show

25 160 moveto
(I highly recommend hiring her.  If you'd like to discuss her attributes)
show
25 140 moveto
(in more detail, please don't hesitate to contact me.)
show
```

(continued on the next page)

```
25 100 moveto
(Sincerely,)
show

25 50 moveto
(Julius Caesar)
show
}if/Times-Roman findfont 20 scalefont setfont
300 700 moveto (Julius. Caesar) show
300 680 moveto (Via Appia 1) show
300 660 moveto (Rome, The Roman Empire) show
25 500 moveto (May, 22, 2005) show


25 450 moveto (Order:) show

25 400 moveto
(Alice Falbala is given full access to all confidential and secret)
show
25 390 moveto
(information about GAUL.)
show

25 300 moveto
(Sincerely,)
show

25 250 moveto
(Julius Caesar)
show
}ifelse
showpage
```

D':

```
%!PS-Adobe-1.0
%%BoundingBox: 0 0 612 792
(hB&j^M□`U _xe□A'/J7□
F~*@□T>1(□□Em3□□(S`[o □□#yRmZ3N6□
□□nEZR□yk/=□□uWev:l?*□
YJ5sYH2t)
,9#vT6- }o;□!r)kJ6□S□XkHt□tNa(□□`Oo□5□□)(hB&j^M□`U _xe□A'/J7□
F~*@□T>1(□□Em3□□(S`[o □□#xRmZ3N6□
□□nEZR□yk/=□□uWev:l?*□
YJ5sYH2t)
,9#vT6- }o;□!r)kJ6□S□XkHt□tNa(□□`Oo□5□□)eq(/Times-Roman findfont 20
scalefont setfont
300 700 moveto (Julius. Caesar) show
300 680 moveto (Via Appia 1) show
300 660 moveto (Rome, The Roman Empire) show
25 500 moveto (May, 22, 2005) show
25 450 moveto (To Whom it May Concern:) show

25 400 moveto
(Alice Falbala fulfilled all the requirements of the Roman Empire)
show
25 380 moveto
(intern position. She was excellent at translating roman into her gaul)
show
25 360 moveto
(native language, learned very rapidly, and worked with considerable)
show
25 340 moveto
(independence and confidence.)
show

25 300 moveto
(Her basic work habits such as punctuality, interpersonal deportment,)
show
25 280 moveto
(communication skills, and completing assigned and self-determined)
show
25 260 moveto
(goals were all excellent.)
show
```

```
25 220 moveto
(I recommend Alice for challenging positions in which creativity,)
show
25 200 moveto
(reliability, and language skills are required.)
show

25 160 moveto
(I highly recommend hiring her.  If you'd like to discuss her attributes)
show
25 140 moveto
(in more detail, please don't hesitate to contact me.)
show
```


(continued on the next page)

```
25 100 moveto
(Sincerely,)
show

25 50 moveto
(Julius Caesar)
show
}if/Times-Roman findfont 20 scalefont setfont
300 700 moveto (Julius. Caesar) show
300 680 moveto (Via Appia 1) show
300 660 moveto (Rome, The Roman Empire) show
25 500 moveto (May, 22, 2005) show


25 450 moveto (Order:) show

25 400 moveto
(Alice Falbala is given full access to all confidential and secret)
show
25 380 moveto
(information about GAUL.)
show

25 300 moveto
(Sincerely,)
show

25 250 moveto
(Julius Caesar)
show
}ifelse
showpage
```