Cryptographic Hash Functions

Diplomarbeit
von
Christian Knopf

November 2007

Leibniz Universität Hannover
Institut für Theoretische Informatik

Prüfer:
Prof. Dr. Heribert Vollmer
Prof. Dr. Rainer Parchmann

Contents

1 Introduction

1.1 Motivation

With the advent of public key cryptography, the sender of an encrypted message could no longer be authenticated simply based on the fact that he knows the encryption key. When encrypting messages with public keys, it is therefore necessary to separately authenticate the sender. Of course, this problem is not limited to encrypted messages. Methods of authentication became important with the rise of digital communication.
It would be possible to encrypt the message with the senders private key. Either the message is encrypted with the recipients public key once more, or attached to the message, to encrypt both. Then, to authenticate the sender, it can be decrypted using his public key. However, apart from either complicating handling, or unnecessarily doubling the length of a secret message, it has several other disadvantages. For example, the message always has to be decrypted before the authenticity can be verified, and thus can only be verified by the recipient. Also, the authenticity information cannot be detached from the message.
A lot of such problems can be eliminated by not using the whole message as the signature, but instead calculating a form of checksum of the message, the *message digest*. A signature scheme can be devised with this digest and public key cryptography. Thereby, a small, detachable signature can be produced, conveniently handled, and independently confirmed.
The security of such a signature scheme is naturally limited by the strength of the underlying method of encryption. However, the signature has to be able to withstand several kinds of attack. It has to be impossible to forge a signature: nobody should be able to sign any message without the signatories private key. Finding a message that a valid (existing) signature can be attached to must be impossible as well. Additionally, it has to be impossible to alter a signed message in any way, without invalidating the signature. Of course, gaining any information about the message out of the signature should not be possible as well.
All these demands require the generation of the message digest to have specific properties. In essence, it has to be infeasible to generate two messages with the same digest, as well as find a message that has a given digest. Of course, this is just an oversimplified outline.
Hashing can be compared to taking a “fingerprint” of data. The fingerprint is small, and, for a good hash function, all files have a different fingerprint.
Since its operation can be compared to “chopping up” the message, the func-

tion was termed a *hash function*, or, more specifically, a *cryptographic hash function* because of its cryptographic properties. The message digest is also often called a *hash value*, a *hash sum*, or simply a *hash*. They are usually displayed in the form of hexadecimal digits.

MD4	`31d6cfe0 d16ae931 b73c59d7 e0c089c0`
MD5	`d41d8cd9 8f00b204 e9800998 ecf8427e`
SHA-1	`da39a3ee 5e6b4b0d 3255bfef 95601890 afd80709`
RIPEMD-160	`9c1185a5 c5e9fc54 61280897 7ee8f548 b2258d31`
Tiger	`3293ac63 0c13f024 5f92bbb1 766e1616 7a4e5849 2dde73f3`
RadioGatún[64]	`64a9a7fa 139905b5 7bdab35d 33aa2163 70d5eae1 3e77bfcd d8551340 8311a584`
SHA-256	`e3b0c442 98fc1c14 9afbf4c8 996fb924 27ae41e4 649b934c a495991b 7852b855`
Whirlpool	`19fa61d7 5522a466 9b44e39c 1d2e1726 c5302321 30d407f8 9afee096 4997f7a7 3e83be69 8b288feb cf88e3e0 3c4f0757 ea8964e5 9b63d937 08b138cc 42a66eb3`

Some example hash values: hashes of the empty string.

In this document, many aspects of cryptographic hash functions are explained and analyzed. The rest of this chapter covers some basic topics like notations and terminology, and shows some of the many purposes cryptographic hash functions can be used for. The theoretical and cryptographic demands are discussed in chapter 2, while chapter 3 shows some methods and approaches for designing cryptographic hash functions. Several examples of hash functions are presented in chapter 4, along with many cryptanalytic findings. In chapter 5, each hash function is tested for its speed, showing many insightful results. Finally, a conclusion and an outlook are given in chapter 6.
The three appendices give some additional information: naive collision search is covered in appendix A. Detailed results of the speed tests are shown in appendix B, and some example collisions are shown in appendix C.
In the bibliography, a comprehensive collection of papers on the addressed aspects of cryptographic hash functions is assembled; each one is annotated with important contents and a short summary.

1.2 Terminology and Notation

As usual, a byte consists of eight bits. A word can hold 32 bits (a 32-bit word), or 64 bits (a 64-bit word). There are two ways to arrange bytes in a word: if the most significant byte is first in a word, then it is called big-endian, otherwise little-endian. Any data can be represented by a stream (or string) of bits.

The term *computationally infeasible* expresses that something is (nearly) impossible to compute. In the same sense, sometimes a problem is referred to as being *hard*.

This ranges from "could be possible in (or within) 10 or more years given large amounts of money" to "needs more memory space than the number of atoms in the universe and more calculatory steps than the number of nanoseconds in the age of the universe". While problems of complexity 2^{64} have been solved by internet projects [1], generally problems of complexity 2^{80} (of memory or computations) are considered computationally infeasible. However, technological progress suggests, in accordance with Moore's Law, a factor of solvability of about 2 each year. Consequently, a problem of complexity 2^{80} would become solvable in less than 20 years. On the other hand, it is safe to assume that a calculation of size 2^{128} remain impossible at least until the end of the century. The physical limits of feasibility set by the second law of thermodynamics is considerably lower than 2^{256} [2].

Due to the nature of hash functions, all attacks have a finite complexity. Therefore, the Landau notation cannot be used to describe the complexity of such attacks in this document. Instead, they are usually given in the form of n hash function applications.

Unfortunately, in many papers, cryptanalysts state only the size of the problem to solve for an attack. For example, when n conditions of the form "bit $x_m = b_m$" have to be fulfilled, the complexity is taken to be 2^n. This indication disregards the fact that the computational effort needed is not equivalent to applying the hash function the same number of times [3]. Therefore, the complexities given here might be too optimistic in some cases, and should be corrected.

The amount of information of a message is not always identical to the size or length of the message. In information theory, Claude E. Shannon brought forth the concept of *entropy* to describe the information content of a message [4]. This can be translated into "randomness" of data, or the maximum achievable compression ratio of a message – data with little randomness has little entropy and can be compressed well, data with maximum entropy has no bits wasted on redundancy or unnecessary information.

Concatenation of strings or variables is not mostly explicitly denoted. However, in some cases the symbol "$\circ$" is used as the concatenation operator to avoid confusion.
Binary functions can easily be extended to bytes or words by bitwise operation. "$\wedge$" thus denotes bitwise `and`, "$\vee$" denotes bitwise `or`. "$\oplus$" stands for bitwise `xor`. The bitwise complement function, `not`, is expressed by $\bar{x}$ for the variable x.
It is widely accepted to use the braces { and } in two different ways: in the traditional meaning to mark sets, and, with an exponent, as a common way to specify strings over an alphabet. $\{0,1\}^*$ denotes, for example, the set of all finite strings over the alphabet $\{0,1\}$, while $\{0,1\}^n$ is the set of all strings of length n. Finally, $\{0,1\}^\infty$ stands for the set of all binary strings of infinite length.

1.3 Alternative Uses

Cryptographic hash functions are necessary for useful authentication protocols and digital signatures. There are many signature standards, one of the most popular is the digital signature algorithm (DSA) [5].
However, because of their various properties, cryptographic hash functions are well-suited for many other applications. In the digital world, they are in everyday use.
The hash value of a set of data should always be unique. Consequently, any file can be easily identified by its hash value. Additionally, the integrity of data can be verified this way. Many websites list MD5 and SHA-1 hashes for downloadable software packages, to eliminate transmission errors, for example. MD4 is used to ensure file integrity in edonkey downloads.
A hash function is irreversible. Therefore, it can be used to prove knowledge of a secret without revealing it. This is sometimes used, for instance, in the following manner: Someone discovers a security flaw in a software product, and wants to disclose it at a later time (when the flaw has been fixed by the producer of the product), but also wants to assure to be credited as the first to discover the flaw. Then, he can write a document explaining the weakness, and publish only the hash of that document. At a later time, he can therefore prove to have known about the flaw by publishing the full document.
Additionally, the irreversibility is used by saving only the hash of a password. To gain access to a system, the provided password is then hashed and the result is compared to the saved hash. This scheme ensures that even the system administrator has no knowledge of the password, and is, slightly modified at

times, widely used for user authentication.
In addition, a hash function can be utilized to transform a password into a key with specific properties. For example, keys for symmetric ciphers have to have a certain size, like 128 bit. It is desirable for the password to have no such restrictions, but the hash value of the password has a set size, and high entropy.
Another excellent and important use of cryptographic hash functions is the generation of pseudo-random numbers. A simple setup is the hash of a combination of a counter with a seed. This random number generator will have an infinite period, if the counter is allowed to become arbitrarily large. Session identifiers are usually created this way.
The same aspect can be used for encryption. The output of hash functions can be used either to create block ciphers, or even as a stream of bits for stream ciphers, like encryption via a one-time pad.
Due to the properties of cryptographic hash functions, and the fact that they are thoroughly checked by cryptanalysts, they have many more special uses throughout all areas of computing.

2 Theoretical Properties

2.1 The Random Oracle Model

A *random oracle* is defined as the function $R : \{0,1\}^* \to \{0,1\}^\infty$ of which for each input the output is a uniformly and randomly chosen, infinite-length binary string [6]. The output can be truncated to any desired length. For this function, the only way to find the corresponding input to a given output is brute force: trying all possible inputs until the correct one is found.
The random oracle is a theoretical abstraction that cannot be implemented. However, many cryptographic algorithms and protocols can be proven to be secure under the *random oracle model*, and the model is used for other mathematical proofs as well. Here, all parties are given access to the random oracle, and it is substituted for cryptographic primitives like hash functions and ciphers. Although questioning the oracle usually has a complexity of $\mathcal{O}(1)$, it often makes sense to modify this property.
A random oracle, possibly truncated to a reasonable length, possesses all necessary properties of a hash function.

2.2 A Formal Definition

Not all desired properties of a cryptographic hash function can be phrased into a short, comprehensive definition. Nonetheless, covering the basic aspects is helpful for most purposes.
A cryptographic hash function H is a mapping from the set of input data, binary strings of arbitrary length, $\{0,1\}^* = \mathbb{N}$, to the set of hash values $\{0,1\}^d$ which are fixed in size, and considerably small, d is usually a value between 128 and 512.
Computation of the hash function should be very fast. The complexity of a hash calculation should be $\mathcal{O}(n)$ for a string of length n.
Because the input domain is larger than the output domain, hashing cannot be an injective mapping. Therefore, *collisions* – two input values that result in the same output – inevitably occur. Nevertheless, the output domain usually has "enough" elements (for example, $2^{160} \approx 1.4 \cdot 10^{48}$). Therefore, for a function that maps to a uniformly chosen element, the chances of finding a collision are negligible.
Consequently, a cryptographic hash function should be a random, uniform, and surjective mapping to the output domain. In other words, it should not be distinguishable from a (truncated) random oracle. Thus, collisions can-

not easily be found: it is infeasible to find two distinct messages x and x' with $H(x) = H(x')$. This property of hash functions is called *collision resistance*.
A one-way function is a function that is easy to compute, but hard to reverse, or invert: $f(x) \longrightarrow y$ is fast and simple, but $f^{-1}(y) \longrightarrow x$ is extremely difficult. Put another way, $f(x) \in P$, but $f^{-1}(x) \in NP$. Therefore, any efficient algorithm solving a P-problem succeeds in inverting a one-way function f with negligible probability. Proving the existence of one-way functions would imply $P \neq NP$ [7].
A cryptographic hash function is a one-way function, and therefore called *preimage resistant*. Given a hash h, it is impossible to compute any x with $H(x) = h$ faster than by brute force.
A similar property of hash functions is *second preimage resistance*. This means that, given a message and its hash, it is computationally infeasible to find another message that hashes to the same value: given x, finding $x' \neq x$ with $H(x) = H(x')$ is difficult.
In accordance with Kerckhoffs' principle, any hash function algorithm should be fully disclosed so any cryptanalyst can scrutinize it. To go even further, any design considerations should receive the same treatment.
Most of these properties are expressed the same way as in Ralph C. Merkles dissertation, where he laid the groundwork for cryptographic hash functions and devised first authentication schemes [8].

2.3 Collisions and Preimages

Preimage Resistance

If preimages can be created for a hash function, then an attacker is able to create a message that hashes to a specific hash. This means that he can take any signature, create another document with the same hash, and transfer the signature. The signatory could not deny having signed the document, because the signature is valid for both documents.
Therefore, a cryptographic hash function has to be preimage resistant: there is no method that finds a preimage to a given hash faster than by brute force. For a brute force attack on a random oracle truncated to n bits, 2^{n-1} messages would have to be created and hashed on average to find a preimage. Thus, for a secure cryptographic hash function, a preimage attack has a complexity of $\mathcal{O}(2^n)$.

Second Preimage Resistance

From the viewpoint of digital signatures, an attacker is given more information for a second preimage attack than for a preimage attack, he is not only given the hash of a message, but also the message itself. For a hash function, being second preimage resistant roughly means that even slight modifications to a document will always result in a different hash value. Consequently, knowledge of the original message does not facilitate the attack. Performing a second preimage attack on a good cryptographic hash function therefore has the same complexity as a preimage attack, namely $\mathcal{O}(2^n)$, because the same number of messages have to be searched through.

Collision Resistance

Forging a digital signature is easiest before the signature is created. If an attacker can generate two distinct messages with the same hash value, then both will have the same signature. When the attacker has the possibility of creating both messages, for example by varying small aspects of a document like wording and layout, then he has more degrees of freedom for his attack. Then, after one document is signed, the attacker can transfer the signature to the other one. A nice example of this scheme is drawn up in [9].
For this reason, a cryptographic hash function must to be collision resistant: there is no method to find collisions faster than brute force. This property is sometimes misleadingly called "collision free".
However, because the attacker is able to vary both messages, he is able to take advantage of a time-space tradeoff, and perform an attack with a lower time complexity than that of a preimage attack. This is because of a mathematical fact known as the birthday paradox: In a group of 23 people, the probability of two people having the same birthday is greater than 50%. Similarly, the number of random objects out of the domain 2^n needed to get a collision with a probability of 50% is in $\mathcal{O}(\sqrt{2^n})$ (see appendix A). This means that an attacker only has to perform around $2^{n/2}$ hashing operations to find two messages with the same (random) hash value, however, he also has to save $2^{n/2}$ hash values (and their corresponding messages) in the process.
Performing an exhaustive search while taking advantage of the birthday paradox is often called a *birthday attack*.
For a cryptographic hash function to be secure in the near future, it therefore needs to output hashes of at least 160 bit, if not 192 bit. Any hash function with an output of more than 512 bit is excess.

Near Collisions

In hashing cryptanalysis, random collisions receive by far the most attention. Several publications present somewhat weaker results than full collisions.
A *near-collision* consists of two messages which hash to almost identical hashes, only a few bits of the hash values differ.

Implications Between Collisions and Preimages

Some of the general ideas for proving the following implications were taken from [10]. It is important to note, that, although these claims and definitions are not unambiguously formalized, they serve their purpose well. For more information, see [11].

$$\text{preimage resistance} \not\Rightarrow \text{second preimage resistance}$$

Consider the following example: $H(x)$ is a function that hashes all bits of x but the first with a random oracle. Obviously, finding preimages is impossible, but a second preimage is always given by flipping the first bit of x.

$$\text{collision resistance} \not\Rightarrow \text{preimage resistance}$$
$$\text{second preimage resistance} \not\Rightarrow \text{preimage resistance}$$

For half of all hashes of the following hash function, a preimage can be found, but finding a collision or a second preimage is infeasible:
$H(x)$ hashes all messages longer than $m-1$ bits with a random oracle truncated to m bits and appends the result to a zero bit. The hash of all other messages is a one bit appended by the message x, and then filled with a one bit and zeroes as needed.

$$\text{collision resistance} \Rightarrow \text{second preimage resistance}$$

Any second preimage is a collision, so finding random collisions is not more difficult than finding second preimages.

$$\text{preimage resistance} \not\Rightarrow \text{collision resistance}$$
$$\text{second preimage resistance} \not\Rightarrow \text{collision resistance}$$

The function $H(x)$ hashes messages consisting of n one bits or n zero bits to a binary representation of n, all other messages are hashed with a random oracle. In this hash function, many collisions can be found, but there are exponentially more hash values for which neither a preimage nor a second preimage can be found.

2.4 Completeness and the Avalanche Effect

For a random oracle, the output of two different sets of input is completely different, regardless of the differences between the two input strings. A hash function should have the same property. Otherwise, information can be gained on how to construct preimages or collisions. Even small pieces of such information lessen the security of a hash function.
Two separate effects can be used to formalize this property. When even small changes of the input of a hash function result in a significant change of the hash values, the hash function possesses a strong *avalanche effect*. When each input bit affects all output bits, then the hash function is *complete*.
The *strict avalanche criterion combines* both the avalanche effect and the completeness of boolean functions: A cryptographic hash function satisfies the strict avalanche criterion, if changing one bit of the input data results in each output bit changing with a probability of $\frac{1}{2}$.
If the hash function were not complete, then for at least one output bit there would be a (nonempty) subset of input bits that do not contribute to the value of the output. If the probability of each output bit flipping with a modification in input would be different from the probability of not changing, then on average more or less than one half of the output bits would change with a single bitchange of the input.
Any such flaw would heighten the probability of successful attack on the hash function considerably. In a 128-bit hash function, for example, the probability of finding a second preimage by a single bitchange rises to 2^{-94} when each output bit changes with a probability of 0.4 instead of 0.5.

3 Design and Construction

3.1 The Merkle-Damgård Construction

Designing a cryptographically secure hash function that takes strings of arbitrary lengths as input is a complicated task. In [8], Ralph C. Merkle proposed a way to construct such functions. At Crypto '89, the construction scheme was thoroughly discussed. In [12] and [13], the method is proven to be secure under certain assumptions, and several examples of hash functions are given. Originally named "Merkle's Meta Method", this scheme is now mostly called the *Merkle-Damgård construction*.
In order to construct a hash function that takes messages of arbitrary length as input, a *compression function* is required. The compression function has a fixed-length input and a fixed-length output with a smaller size. Although a compression of one bit suffices, a higher level of compression will result in a faster hash function.
First, the message is split into a number of blocks. These blocks can then easily be hashed by the compression function. By iteratively compressing the result of the previous operation concatenated with the next block, all blocks of the message are processed to produce the hash value. Therefore, the operation of a hash function is determined by its compression function.

Every message m can be expressed as the concatenation of its n blocks. Each block has a fixed length l, a common blocksize is 512 bits, or 16 message words. The last block is padded if necessary (with zeroes, for instance). As an additional security measure, a block m_n consisting of the length of the message is appended:

$$m \;=\; m_0 \;\circ\; m_1 \;\circ\; \cdots \;\circ\; m_n$$

The compression function C takes the *intermediate result* or *chaining value* s_{i-1} and a block of input data m_i, and calculates the next intermediate result, s_i. Therefore, s_i is the state of the hash function after message block m_i has been processed. Here, the chaining values s_i have length k.

$$C\colon\;\; \{0,1\}^k \times \{0,1\}^l \;\rightarrow\; \{0,1\}^k$$

The hash function therefore needs an *initialization vector (IV)*, or initial value, the "first intermediate result", s_0.

$$\begin{aligned} s_0 &= IV \\ s_1 &= C(r_0, m_0) \end{aligned}$$

$$
\begin{aligned}
s_2 &= C(s_1, m_1) \\
&\vdots \\
H(m) = s_{n+1} &= C(s_n, m_n) \\
H(m) &= C\Big(\ldots \underbrace{C\Big(\underbrace{C(s_0, m_0)}_{s_1}, m_1\Big)}_{s_2} \ldots \Big)
\end{aligned}
$$

Originally, the hashing scheme described by Ralph C. Merkle was slightly different. Here, hashing begins with $s_1 = C(m_0, m_1)$, so no initialization vector is needed.
The initialization vector is a hash-specific constant that is determined at design. Commonly, a representation of π or zero bits are chosen to assure no hidden properties. Many hash functions also use the sequence `0123456789abcdef` and permutations thereof.
However, parameterizing the initialization vector offers an easy way to introduce *keyed hashes*. Keying extends a hash function H to a family of hash functions H_K, each with a different key K. Because hashing computations start with the initialization vector, each member of the family would supply different hashes for a message. This would serve as an additional security measure. Furthermore, families of hash functions are important for some cryptographic proofs.
For most hash functions, the hash value of the message is the little-endian representation of the last chaining value s_n. Otherwise, a transformation is done by a *finalizing function*. For example, the internal values s_i are often larger than the hash size, so the transformation consists of shortening s_n to obtain the hash result.
Hash functions constructed according to the Merkle-Damgård scheme are often referred to as "iterative hash functions".

3.2 Attacks on Merkle-Damgård Hashes

Over the years, several flaws of the Merkle-Damgård construction scheme have been discovered. Some of them can easily be avoided. Others are rather theoretical attacks with no practical relevance, because they only apply to extremely large messages, for example. However, Merkle-Damgård hash functions do have a small number of undesired properties.
Altogether, the Merkle-Damgård construction can be considered secure in re-

gard to many characteristics.
It is possible to find two messages m and m' along with two intermediate values s and s' with $C(s, m) = C(s', m')$. This is known as a *pseudo-collision*. For a secure hash function, finding pseudo-collisions has the same complexity as finding collisions.
Hashing an additional block containing the length of the message is known as *Merkle-Damgård strengthening*. It increases the security of the hash function considerably.
For example, given the hash h of a message m under a hash function without Merkle-Damgård strengthening, the hash of the string $m \circ m'$ can be computed: $H(m \circ m') = H(h, m)$.
Another example is an attack based on *fixed points*. Fixed points can be found for all hash functions that have a reversible compression function. This is the case for all hashes that follow the Davies-Meyer construction (see next section), for example for MD4, MD5, and SHA-1. As their compression function C, they execute a block cipher $E(k, X)$ with key k and message block X, and combine the result with the state s_i via a group operation $+$:

$$C(s_i, m_i) \;=\; E(m_i, s_i) \;+\; s_i$$

(the state is encrypted with the message block as the key). The operation $+$ can be modular addition or xor, for example. For these compression functions, fixed points p can be found for any message block m_p: $p = E^{-1}(m_p, 0)$, then p satisfies $C(p, m_p) = p$ [14]. Therefore, when the state of the hash function is p, the message block m_p can be hashed without changing the state.
Note that the initialization vector of a hash function can be chosen to be a fixed point for a secret block m_p. Then, collisions and second preimages are easily obtained by anyone who knows the secret: $H(m) = H(m_p \circ m) = H(m_p \circ m_p \circ m)$. Otherwise, an attacker can control the message block, but he has to produce a preimage for the chaining value p for an attack.
Of course, this can be simplified with a birthday attack, computing $2^{n/2}$ fixed points (p_i, m_{p_i}) and $2^{n/2}$ messages m_j to generate an *expandable message* for a n-bit hash function: $H(m_j) = H(m_j \circ m_{p_i})$. Colliding messages of any length are thus elements of the expandable message.
When the length of the message is part of the hash calculation, however, inserting any message block always results in a different hash.

A kind of attack that is not easily avoided is *length extension*: when two messages m and m', having the same length, collide, then so do $m \circ x$ and $m' \circ x$ for any message x. Therefore, an infinite number of collisions can be instantly generated.

Moreover, if collisions can be generated for any chosen intermediate value, then *multicollisions* can be computed in Merkle-Damgård hash functions. If two colliding message pairs (m_1,m_1') and (m_2,m_2') can be found with $C(s_0, m_1) = C(s_0, m_1') = s_1$, and $C(s_1, m_2) = C(s_1, m_2')$, then the four different messages $m_1 \circ m_2$, $m_1 \circ m_2'$, $m_1' \circ m_2$, and $m_1' \circ m_2'$ all have the same hash value, producing a 4-collision. The scheme can be extended to find an arbitrary number of messages that all have the same hash. The complexity to find a 2^n-collision is merely n times as high as finding a single collision.
This fact can be used to show the low increase in security when concatenating two (or more) cryptographic hash functions [15]. Suppose, the results of two hash functions G and H are concatenated to produce a longer hash value. G produces n bit hashes. However, for the hash function H, collisions, and therefore multicollisions, can be generated with little computational work. Then, out of $2^{n/2}$ different messages, which are computed as a $2^{n/2}$-collision, two will collide under G with reasonable probability, even if G is a truncated random oracle. For this reason, the security of the concatenation of two hash functions is not much higher than the security of the stronger one. The same argument can be applied to concatenations of more hash functions.

Expandable messages can be computed without finding fixed points, however, a birthday attack has to be used. Executing many birthday attacks, n message pairs (m_i, m_i') can be found, each message with a chosen length. If all messages m_i have a length of one block, and all messages m_i' have a length of $2^i + 1$ blocks, then a number of messages, colliding without the length block, can be generated, each with any chosen length between i and $2^i + i$ blocks.
With the help of expandable messages, it is possible to generate second preimages for long messages with less computational work than the expected 2^n for n-bit hash functions [14]. A message with 2^k blocks has 2^k intermediate hash values. A birthday attack on these intermediate hash values has a high probability of finding a block which hashes to the same intermediate value s_i as one of the 2^k values of the original message for $k \to n/2$. When searching for collisions, the chaining value is set to the intermediate hash value of the last part of an expandable message. That way, a second preimage can be constructed by concatenating the element of the expandable message with the correct length, the colliding block found by the birthday attack, and blocks $i + 1$ to 2^k of the original message.
However, with 160-bit hash functions, this attack remains purely theoretical, having a complexity of around 2^{n-k+1} for a n-bit hash function and a message of length 2^k.

3.3 Building Hash Functions From Block Ciphers

The Merkle-Damgård construction scheme can be easily used in combination with block ciphers to create secure cryptographic hash functions.
The encryption routine of a block cipher can be viewed as a function $E(k, X) = Y$, encrypting a message block X under the key k to produce the ciphertext block Y. Blocksizes are usually the same as keysizes, although this is not necessarily the case. DES, for example, has a keysize of 56 bits and a blocksize of 64 bits, the Rijndael algorithm, which is the underlying block cipher of AES, has variable blocksizes and keysizes of between 128 and 256 bits.
A simple cryptographic hash function can be created from any block cipher. Using a message block m_i as the key and the chaining value s_{i-1} as the plaintext, the encryption routine returns the next intermediate value s_i. However, this construction is insecure, because it is easily reversible. Therefore, for a secure cryptographic hash function, the previous intermediate value is xored with the output of the encryption, using modular addition is also possible:

$$s_{i+1} \;=\; E(m_i, s_i) \;\oplus\; s_i$$

This method is called the *Davies-Meyer* construction scheme. Switching the use of the state and the message block, the *Matyas-Meyer-Oseas* construction scheme is obtained:

$$s_{i+1} \;=\; E(s_i, m_i) \;\oplus\; s_i$$

As an extension, the message block can additionally be used in the last step, this is known as the *Miyaguchi-Preneel* scheme:

$$s_{i+1} \;=\; E(m_i, s_i) \;\oplus\; s_i \;\oplus\; m_i$$
$$s_{i+1} \;=\; E(s_i, m_i) \;\oplus\; s_i \;\oplus\; m_i$$

These constructions work very well with block ciphers with equal key- and blocksizes, however, similar methods are usable for ciphers where the keysize is different from the blocksize by using a function to convert the key to the needed format [16].
There are a number of similar construction schemes which can be proven to be secure [17].

Many of the first cryptographic hash functions were constructed from block ciphers. The security of DES, for example, has been studied extensively. The security of constructions from well known, secure block ciphers can therefore be trusted. Additionally, the hash functions are provably secure under certain

assumptions about the cipher. However, most of these hashes are slow in comparison to dedicated hash functions.
Nonetheless, many dedicated hash functions can be viewed as having a "block cipher like" compression function. Whirlpool uses a slightly modified Rijndael algorithm, and MD4, MD5, and all SHA hash functions also fall under this category.

3.4 The Compression Function

The central part of any iterative hash function is its compression function. Its job is to "randomize" the input as much as possible, but of course deterministically. Ultimately, the main importance for a secure compression function is to show collision resistance and preimage resistance, as these properties are inherited by the hash function. Additionally, its computation has to be fast.
The compression function has an internal state, its variables. In the beginning of the hash calculation, some are set to the initialization vector. During hash calculation, the aforementioned intermediate results are accumulated in these variables, and the final result of the hashing process is read from them. Often, there are additional variables for message preprocessing and storing the input block, as well as variables needed for calculation.
There are a number of basic operations that are commonly used in compression functions. Note that for any of these basic operations, it is important to preserve entropy. Thus, independent and unbiased input results in independent and unbiased output.
The following basic operations are frequently used for hash functions:

- `xor`, `not`, `and`, `or`: basic boolean operators. They are present in all processors and very quick in execution. Most hash functions make extensive use of these functions as bitwise operations over full words. For example, inside the compression functions of MD4, MD5 and the SHA hashes the following function is used: $F(x, y, z) = (x \wedge y) \oplus (\bar{x} \wedge z)$. This can be translated as "if x then y else z" [18, 19, 20].

- Bitshifts and rotations (circular shifts): Rotation operations are usually not implemented in processors, but they can be computed using two shifts. Because they aid in giving a strong avalanche effect, they are also used frequently.

- Addition and subtraction: These are also quick, and present in all processors, but not used as frequently. The utilization of multiplication is even more uncommon.

- Substitution boxes: A substitution box is simply a permutation table. Each input value is substituted for the corresponding output value. Substitution boxes offer a unique way of nonlinear operation. Because operations built with the above functions are all linear, substitution boxes are used often, although not in every cryptographic hash function. They have the disadvantage of requiring additional space (in code and in memory during execution). Especially if they are large, they can slow down the compression function significantly. However, the nonlinearity of the mapping dramatically increases the cryptographic strength of the hash function.

Most hash functions are organized in *rounds*, or *passes*. After one block is read, they compute a number of routines (compositions of the operations above), in a loop. Usually, one message word is used inside the loop. There are different meanings of rounds and passes throughout the literature. As many compression functions use different state updating functions, a round is mostly considered as a loop over a part of the message words using the same function. Thus, a compression function with four rounds executes a loop for each of four such functions. SHA-1 and MD5, for example, have four rounds. A *step* is a smaller computation; it usually denotes the functions carried out for each message word.
Many compression functions have a separate stage called *message expansion*. Here, the message words are combined in some way to produce a longer input. SHA-1 expands its 16-word input into 80 words by creating 64 new words, each round then uses 20 words; MD4 and MD5 merely reorder the message words for the different rounds.
This set-up is a very easy way to achieve an exceptional avalanche effect, while saving code and execution time. Depending on the complexity of one round usually looping 3 to 20 times is sufficient. HAVAL has 3 to 5 passes, while Whirlpool specifies 10. Sometimes the rounds are not all identical, or in between them, other functions are carried out.
Needless to say, the compression function needs to be as fast as possible. Therefore, fast computable functions are used almost exclusively. This also applies to the whole hash function, so it includes initialization and finalization.
Parallel operation is problematic in compression functions. Operations that are designed to run in parallel cannot depend on the output of each other – a significant amount of avalanche is lost. Usually, hash functions are mostly

serial in their operation, having only very small parallelizable parts. An exception are hashes, where two similar "lines" are computed by the compression function, which are added at the end. The RIPEMD hashes are an example of this technique [21, 22]. Also, in the MDC-2 hash function there are two DES encryption routines carried out in the compression function, and the latter halves of the resulting string of each encryption are then interchanged. MDC-4 works similarly [23].

3.5 Preformatting

When taking an arbitrary number of bits as input, it is important to have a set procedure on what to do at the end of the stream. The stream might not end with a full word, or even a full byte, but the compression function needs a complete block as input.
Therefore, cryptographic hash functions pad their last input block. This is done by adding certain bits until the hashed message meets length requirements. Padding could be done by simply adding enough zeroes. But if two messages, differing by the number of trailing zeroes, were padded with this technique, the difference would be lost. Of course, this is undesirable. Padding with other repeating patterns would present the same problem.
Most hash functions thus append a single 1 bit, and then 0 bits as needed for the stream to meet length requirements (there may no 0 bits be added at all). This is the widely accepted method as it is unambiguous, so all current cryptographic hash functions employ this padding scheme.
Finally, Merkle-Damgård strengthening is done. The length of the message is appended after the padding, thereby filling up the last block. A 64-bit, 128-bit, or 256-bit representation of the number of bits in the original message is used in most hash functions (it is safe to assume a maximum message size of 2^{256} bits for all purposes). Of course, the length is computed without the padding bits. It is important to note that for most messages, this procedure does not increase the total number of blocks. In some applications, like hashing streamed data, the size is unknown before the end of the message.
To save computation time, as little data as possible will be added to the original message. The hash function can only operate on full blocks, so padding is necessary, but introducing unneeded blocks to compute would be wasteful. However, the extra security of Merkle-Damgård strengthening is important, so an extra block is accepted.

3.6 Implementation Considerations

Poor – or good – design of the hashing algorithm has a huge impact on the possible implementations.
Cryptographic hashing is usually done on "normal" computer hardware. Since embedded hardware environments are not a concern, hashing algorithms do not need to be specially optimized (or optimizable) for very low memory conditions or other strict hardware restraints. However, in the future this does not necessarily hold true. Especially message authentication (for example via keyed-hash message authentication code (HMAC)) could soon be carried out by "keychain devices" or other small, low-cost systems. Additionally, the hash function could be used for encryption on said systems.
This is a significant difference to other hash functions or checksums, and even to encryption. These are regularly computed on dedicated or minimal hardware, and therefore often have numerous design requirements to adapt them for unusual environments while performing well on standard hardware.
The speed of hashing is a very important factor. In most hash functions this leads to very little resources being used. Also, with a compact hash function that uses few operations and emphasizes on loops and reusable code parts, less problems can be expected in all aspects of hardware and software implementation, and the speed of the hash can reach fairly high values.
Hash functions that (almost) exclusively use simple instructions (like `xor`, `or`, `and`, `not`) and rotations could easily be implemented in hardware like FPGAs (Field-Programmable Gate Array) or ASICs (Application Specific Integrated Circuits). Substitutions are efficient, the organization in rounds and especially permutations easily translate into simple wiring [24].
A cryptographic hash function that is designed exclusively for hardware implementation can make use of several specific features. As a common CPU can only operate on full 32-bit or 64-bit words, hash functions based on smaller wordsizes run considerably slower. At hardware level, there are no such restrictions, and bit-oriented hash functions are able to achieve the same speed. However, cryptographic hashing in hardware components is not (yet) in heavy use.
Access to the RAM is slow compared to cashes and registers, so an optimized hash function reads the input data into the CPU registers, and keeps the amount of data small enough to remain in the cashes. This sets a limit to the blocksize of the hash function, and also to S-boxes and other constants.
Cryptographic hash functions are designed to be portable across the commonly used hardware platforms, which makes them portable to almost any system. Therefore, special care has to be taken about several issues. The most impor-

tant is which CPU instructions are used by the compression function. Each different processor supplies a completely different set of instructions. There are, however, a number of basic instructions present on all processors, which at the same time are very fast. For the same reason, those are the instructions used mostly by hash functions. Other functions may have to be emulated in some way. This might slow down execution of the hash function significantly, so they are chosen carefully. For all cryptographic hash functions, reference implementations have been published in standard C, which provides a certain amount of portability. However, optimization for specific architectures can greatly improve the speed of those implementations. Exploiting properties of specific CPUs like cash management, multithreading, or even pipelining features often results in huge changes [25, 26, 27].
Another issue is the endianness of the platform. Because different architectures have different ways of storing data internally, it has to be ensured that functions have the same result when given the same input on all architectures. This applies to constants, like the initialization vector, as well as to chaining values, input data and the final hash value. Converting between different formats might take a significant amount of time.
Most hash functions have been designed for 32-bit processors, mostly using 32-bit words for calculations. Using 64-bit operations or designing hash functions for 64-bit processors can result in faster computations; disregarding 64-bit CPUs can result in hashing being only half as fast as it could be. For example, the Tiger hash function was the first to be specifically designed for 64-bit computers (mainly for the DEC Alpha processor), while performing well on 32-bit platforms [28]. Most hash functions with a wordlength of 32 bits are by design incapable of taking advantage of speed improvements of 64-bit operations. Cryptographic hashing on 16-bit and 8-bit processors is rarely done (today), thus consideration for these platforms is usually not within the design criteria.

3.7 Hash Lists and Hash Trees

Parallel hashing can be easily done with tree and list constructions. The message is simply divided into "chunks", which are hashed seperately. These chunks should not be smaller than the blocksize for performance reasons, otherwise, they can have any size.
Constructing a list of the hashes of these chunks results in a *hash list*. This can be easily parallelized over n processors with linear speedup.
Additionally, a *hash tree* can be constructed. The leaves are represented by the

hashes of the chunks, and every node is the hash of its children. Any branch of the tree can quickly be verified, as the nodes only depend on their children. This is important in untrusted and distributed environments, for example in peer to peer filesharing applications.
A tree of Tiger hash values is commonly used for this purpose.

4 Some Cryptographic Hash Functions

4.1 A Historical Overview

Many of the first cryptographic hash functions proposed were based on well known, hard problems, because the designs concentrated on provable security. Ivan B. Damgård devised cryptographic hash functions that are based on the NP-complete knapsack problem, and on modular squaring [13], Ralph C. Merkle based the security of his design on the randomness of DES in the sense that the DES functions can be viewed as a lookup in a large table of random numbers [12]. The two hash functions MDC-2 and MDC-4 were also based on DES [23].
However, after 1990, designers quickly moved away from provable security or security under certain assumptions to repeating many rounds of a simple compression function, as mentioned in chapter 3.4. This was largely motivated by efficiency reasons and confidence in the strong security of these constructions.

The first widely used hash was the "Message Authenticator Algorithm", MAA, which was proposed in 1983 and used mainly in communication of financial institutes [29]. A keyed cryptographic hash function was part of the algorithm, the secret key was 64-bit in size. It returned a 32-bit hash, had a special mode of operation for messages longer than 1024 bits, and was intended for authentication via symmetric cryptography. The hash function was, in contrast to other proposals of the time, quite fast.
A cryptanalysis was published in 1997, showing the possibility of both message forgery (2^{24} one-block messages), and key recovery (2^{32} chosen one-block messages and 2^{44} up to more than 2^{51} multiplications, depending on the key), as well as the existence of 2^{33} weak keys that allow easy computation of collisions [30].

Between 1990 and 1992, many other hash functions have been proposed. Many of them showed weaknesses within less than two years of their announcement. For a more exhaustive survey, see [31].
MD4, MD5, and SHA-1 were certainly the most dominant hash functions after 1991.

4.2 MD2

MD2 is the first of a series of cryptographic hash functions designed by Ronald L. Rivest, MD standing for "message digest". MD1 and MD3 have never been published, MD5 is merely an extension of MD4.
All three functions give 128-bit hash values.
MD2 was designed in 1988. It has been published in RFCs 1115 and 1319 [32]. The function was designed to hash *byte*streams, and optimized for 8-bit machines. Its blocksize is 16 bytes. Padding is performed by appending n bytes with value n each (1 to 16 bytes), and then a separately calculated checksum with 16 bytes in length.
At the center of MD2s compression function is a 256 byte s-box substituting bytes for bytes, which is constructed from the digits of π. A 48-byte array (X_k) is used as the internal state, the value of an additional variable t is used as the index for the substitution box, changes are calculated modulo 256. All variables of MD2 are initialized to 0. Only the xor function is used to combine variables. The 16-byte blocks are written to X_k, once normal ($k \in \{16 \dots 31\}$) and once xored with $X_0 \dots X_{15}$ ($k \in \{32 \dots 47\}$). MD2 iterates the following function over all 48 words for 18 rounds:

$$\texttt{Set } t \texttt{ and } X_k \texttt{ to } (X_k \oplus S[t])$$

After each round, t is increased by the number of rounds so far. The resulting hash value of MD2 is $X_0 \dots X_{15}$ after the final block has been processed.

Cryptanalysis of MD2 has shown severe weaknesses: An attack finding compression function collisions was found in 1995, although the intermediate hash value has a huge effect on its complexity [33]. However, it was possible to compute many collisions, since the complexity is $256^{17-z} = 2^{8 \cdot (17-z)}$, where z is the number of trailing zero bytes of the intermediate hash, $z = 16$ for the first block hashed. Surprisingly, the checksum seems to be the only reason why no collisions have been published yet, although a pseudo-collision has been found along with a preimage attack by Lars R. Knudsen and John E. Mathiassen. Furthermore, they show multi-collisions for the compression function and a pseudo-preimage attack [34]. The multi-collision attack is expected to generate eight messages and has a complexity of 2^{72}. Pseudo-collisions can be found with a complexity of 2^{16}, but it is important to note that the checksums for both messages is equal only because the messages were both fixed to 0 and two different intermediate hash values h and h' were calculated, resulting in the pseudo-collision $H(h, m) = H(h', m)$ with $m = 0^{128}$. The complexities are 2^{95} for the pseudo-preimage attack, and at least 2^{97} for the preimage attack,

depending on the desired message length. In light of these attacks, MD2 can no longer be considered a one-way hash function.

4.3 MD4

MD4 was designed in 1990, two years later than MD2. It is a Merkle-Damgård strengthened hash for bitstreams. The algorithm was inspired by the proposals of Ivan B. Damgård and Ralph C. Merkle at Crypto '89 [12, 13]. The MD4 hash function was optimized for 32-bit CPUs, and designed to run very fast. It is described in RFCs 1186 and 1320, but it was originally published at the Crypto '90 conference [18]. An extension to MD4 was proposed in the article to provide for 256-bit hash values, but it did not gain much attention. It is commonly referred to as Extended-MD4.

The method of padding introduced with MD4 has been copied to many other hash functions. One 1 bit is appended to the message, and then as many 0 bits as needed for the message length to be congruent to 448 modulo 512. Next, the length of the message is appended as two 32-bit words, the most significant word is last. If the length of the message is bigger than 2^{64} bits then only the 64 least significant bits are used. This introduces security flaws, but messages with lengths in excess of 16 exabytes are highly unlikely, even by today's standards.

After that, the message can be evenly divided into 512-bit blocks (16 32-bit words), MD4s blocksize.

MD4 uses eight 32-bit words as its internal state, four are used during each round (A, B, C, D), and the remaining four are the chaining variables. They get updated after each block is processed by the compression function: the sum of the chaining variable and its corresponding round variable is saved into both variables. All additions are done without carry (i.e. modulo 2^{32}). The initialization vector is:

A	=	`0x67452301`	`01 23 45 67`
B	=	`0xefcdab89`	`89 ab cd ef`
C	=	`0x98badcfe`	`fe dc ba 98`
D	=	`0x10325476`	`76 54 32 10`
		(hexadecimal)	(little-endian notation)

The following functions are used by MD4s compression function:

$$\begin{aligned} F(x,y,z) &= (x \wedge y) \vee (\bar{x} \wedge z) \\ G(x,y,z) &= (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \\ H(x,y,z) &= x \oplus y \oplus z \end{aligned}$$

For each block, every function is carried out for 16 passes, with three of the four variables as arguments in varying order. Thus, MD4 has three rounds, each consisting of 16 calls to the function F, G, or H, respectively. The sum of the result, an input word, and a constant is added to the fourth variable. After that, the variable is circular-shifted by an odd number.
The operation of each step of MD4 can be summarized as follows:

$$w \;=\; (w \;+\; \phi_i(x,y,z) \;+\; M_{\pi_i(k)} \;+\; C_i)^{\lll s_{i,k}}$$

w, x, y and z are each one of the variables A, B, C and D. i is the current round ($i \in \{1,2,3\}$), k is the current pass ($k \in \{0,\ldots,15\}$) of each round. ϕ_i denotes one of the functions F, G or H. $M_{\pi_i(k)}$ is word $\pi_i(k)$ of the input block (π_i is a simple permutation), and C_i is the additive constant of round i (either `0x0`, `0x5a827999` ($= \lfloor 2^{30} \cdot \sqrt{2} \rfloor$) or `0x6ed9eba1` ($= \lfloor 2^{30} \cdot \sqrt{3} \rfloor$)). $\alpha^{\lll s_{i,k}}$ denotes circular shifting α by $s_{i,k}$ bits ($s_{ik} \in \{3,5,7,9,11,13,15,19\}$), with each round having four different shift distances.
After the last block is processed, the MD4 message digest is $ABCD$ in little-endian notation.
Extended-MD4 uses two parallel instances of MD4. They differ by the initialization vectors and the additive constants. In the first line, MD4 as described above is used, while the other line uses `0x33221100 0x77665544 0xbbaa9988 0xffeeddcc` as its initialization vector, and `0x0`, `0x50a28be6` ($= \lfloor 2^{30} \cdot \sqrt[3]{2} \rfloor$), and `0x5c4dd124` ($= \lfloor 2^{30} \cdot \sqrt[2]{3} \rfloor$) as additive constants. At the end of the compression function, the values of the A variables of the instances are interchanged. Extended-MD4 produces a 256-bit hash value by concatenating the results of both lines.

The first cryptanalysis of MD4 was published in 1991 [35]. It was shown that if the first round is omitted, then collisions can be found easily. The article also claims that collisions can easily be found if omitting the last round. These serious concerns about the security of MD4 led to the deployment of MD5 shortly after.
Many other cryptanalytic results were published over the following 15 years. In 1996, the first collision of the full MD4 was published, as well as a collision of a slightly modified variant of Extended-MD4, where both lines had the

same initialization vector [36]. By 2005, the attacks had become so sophisticated that collisions can be found by "hand calculation", finding collisions with probability close to 1 within three applications of MD4 [37, 38]. Additionally, preimages for the first two rounds of MD4 can be found within an hour, second preimages only take a few minutes [39].
Today, MD4 can be considered the most unsound cryptographic hash function of all in use. Nevertheless, many other hash functions are based on the design of MD4. All of these functions seem to have inherited many of the weak cryptographic properties of the hash function.
MD4 is, despite its flaws, still popular in some areas. Additionally, MD4 remains a target for hashing cryptanalysis.

4.4 MD5

After indications of MD4 sacrificing too much security for speed, the algorithm was modified to compose a more secure hash function, MD5. It was presented at the Crypto '91 rump session and published in RFC 1321 [19]. The MD5 hash function was very popular from 1992 on, and is still in use. It has been used for many applications, for example, it is available as a native tool in almost all versions of Linux and Unix.
MD5 is very similar to MD4 in many aspects. It uses the same message preprocessing method (padding, and appending the length of the message). The blocksize is the same, the initialization vector of the four variables A, B, C and D is also the same. One round was added, along with a new round function. While F and H were kept, G has been exchanged and I has been added. Each round still has 16 passes, but the operation of each step has changed considerably.
A table $T[i]$ $(i \in \{1 \dots 64\})$ of 64 elements is used for different additive constants for each step. The elements of the table are values of the sine function: $T[i] = \lfloor |\sin i| \cdot 2^{32} \rfloor$. Of course, this is most of the time implemented as a table of constants. There are now 16 different shift distances, only eight of them are odd. As in MD4, every fourth pass of each round has the same shifting constant, but the same values are not used in any other round. In every step, the result of the previous operation is added.

MD5s functions are

$$\begin{aligned} F(x,y,z) &= (x \wedge y) \vee (\bar{x} \wedge z) \\ G(x,y,z) &= (x \wedge z) \vee (y \wedge \bar{z}) \\ H(x,y,z) &= x \oplus y \oplus z \\ I(x,y,z) &= y \oplus (x \vee \bar{z}) \end{aligned}$$

Note that G and F are the same function, $G(x,y,z) = F(z,x,y)$, along with H they are used by MD4. The G function of MD4 has not been used in MD5 (as well as in other hash functions that were designed on the basis of MD4), because of its undesired symmetry and its slow performance.
The operation carried out in each step is:

$$w = v + (w + \phi_i(x,y,z) + M_{\pi_i(k)} + T[i \cdot 16 + k])^{\lll s_{i,k}}$$

Here, again, w, x, y and z are each one of the variables A, B, C and D. v is the result of the previous step. i is the current round ($i \in \{1,2,3,4\}$), k is the current pass ($k \in \{0,\ldots,15\}$) of each round. ϕ_i denotes one of the functions F, G, H or I. $M_{\pi_i(k)}$ is word $\pi_i(k)$ of the input block, and $T[k]$ is an additive constant from the table T. $\alpha^{\lll s_{i,k}}$ denotes circular shifting α by $s_{i,k}$ bits, addition is done modulo 2^{32}.
The resulting MD5 hash is, as in MD4, the little endian notation of $ABCD$.

MD5 withstood cryptanalysis for a while, with only a few unpractical attacks published until 2004. Then, without any explanation of the results, collisions for MD4, MD5, HAVAL-128 and RIPEMD were published by Xiaoyun Wang et al. [40]. Ten months later, Xiaoyun Wang had improved her attack, and published a paper explaining most of her methods in respect to MD5 [41]. However, more research had been done by other cryptanalysts, leading to slightly different (and faster) attacks [42]. Since 2006, collisions of MD5 can be computed in less than 30 seconds, for any initialization vector [43]. This is done by using even more sophisticated methods of cryptanalysis, as well as improving and combining other methods of attack. A program to generate collisions is available with its source code [44].

Because of MD5s popularity, plenty of work beyond finding collisions has been done. In [9], two different postscript files are available having the same hash and meaningful, different content. This is done by using commands like

```
if (data1 == data1) then write(text1);
if (data1 == data2) then write(text2);
```

Of course, this can easily be used in executables. However, similar attacks are possible with several other file types, for example in `pdf` documents, `tiff` images and Word files [45].
The hashclash project (`http://www.win.tue.nl/hashclash`) has generated a *target collision* for MD5. It consists of messages $m_1 \circ b_1$, $m_2 \circ b_2$, with $m_1 \neq m_2$, $b_1 \neq b_2$ that hash to the same value: $H(m_1 \circ b_1) = H(m_2 \circ b_2)$. It is called a target collision, because for two distinct intermediate values of the hash function, messages have been found to produce a collision (the term *target collision* has been used with different meanings in hashing cryptanalysis). It is important to note that both messages consist of meaningful data. The project has generated two X.509 certificates for different identities. A X.509 certificate is currently the most important standard for certificates of a public key infrastructure. They are commonly used as certificates for the authenticity of a host when establishing a secure, encrypted communication, for example via TLS (Transport Layer Security).
The two generated certificates have different common names and some identical information (the message parts m_1 and m_2), and then two different RSA moduli (the collision b_1 and b_2). At this point, both messages hash to the same value. Then, another part of identical information is appended to meet the form of the X.509 certificate. This information is not much more than the MD5 hash and the signature. The target collision was achieved by using several (pseudo) near-collisions and it takes up about half of each RSA modulus, the latter half was calculated thereafter to make both certificates contain actual valid RSA moduli. About 2^{50} calls to the compression function of MD5 were executed for the generation of the near-collisions, and was done with the help of BOINC (the Berkeley Open Infrastructure for Network Computing, `http://boinc.berkeley.edu` is an infrastructure for distributed computing projects), with about 1200 computers participating in the search. The whole attack took roughly 6 months to complete. It is described in detail in [46].
This attack extends finding random collisions to producing meaningful data within the collision. Also, the attack is much more sophisticated, and on a considerably higher level than what has been done before. Different to all previous work, the collision consists of more than two blocks. The complexity was much higher than random collision finding (all methods for quickly finding MD5 collisions had been incorporated), but considerably lower than a brute force search (2^{64} without considering that valid RSA moduli would have to be part of the collision, and, more importantly, without consideration for the space complexity of both attacks). All in all, this attack shows very well how random collisions can lead to meaningful collisions – which pose even more of

a threat to digital signatures – and how the attack was extended from equal or fixed starting points (the intermediate hash value) to completely different ones.

4.5 RIPEMD

Another hash function designed after MD4 – or rather Extended-MD4 – is RIPEMD (RIPEMD stands for "RACE Integrity Primitives Evaluation Message Digest", RACE stands for "Research and Development in Advanced Communications Technologies in Europe"). It was designed in 1992 as a strengthened version of MD4 [21].
The RIPEMD hash is little more than two slightly modified lines of MD4 run in parallel. Both lines differ only in the additive constants (one line uses the constants of MD4, the other line uses `0x50a28be6` ($= \lfloor 2^{30} \cdot \sqrt[3]{2} \rfloor$), `0` and `0x5c4dd124` ($= \lfloor 2^{30} \cdot \sqrt[3]{3} \rfloor$)). The distances of the circular-shifts and the permutations of the input words are the same in both lines, but differ from MD4. Each round now has 16 distinct shifts. The initialization vector is the same as in MD4. The resulting variables A^r and A^l, B^r and B^l, C^r and C^l, and D^r and D^l of each line l and r are added to the chaining variables after each run of the compression function.
The operation of each step of RIPEMD is:

$$\begin{aligned} w^l &= (w^l + \phi_i(x^l, y^l, z^l) + M_{\pi_i(k)} + C_i^l)^{\lll s_{i,k}} \\ w^r &= (w^r + \phi_i(x^r, y^r, z^r) + M_{\pi_i(k)} + C_i^r)^{\lll s_{i,k}} \end{aligned}$$

For lines l and r respectively, the words w are computed by summing up the previous word, the output of the function ϕ_i (ϕ_i is one of MD4s functions F, G, or H) of three of the variables x, y, and z, the word $M_{\pi_i(k)}$ of the input, and the constant C_i. The sum is then circular-shifted by the last operation, $\lll s_{i,k}$.
The final hash value of RIPEMD is the value of the chaining variables in little-endian notation.

After some cryptanalysis on MD4, and as a direct consequence of an attack on a reduced-round version [47], RIPEMD has been replaced by a new version. In 2004, a collision for RIPEMD was published [40], and later the attack was explained [37]. Generating collisions has a complexity of about 2^{18}.

4.6 RIPEMD-160

RIPEMD-160 was published in 1996, along with RIPEMD-128, a more secure replacement for the original hash [22]. It is therefore sometimes referred to as RIPEMD-1. The proposal includes four hash functions, with 128-, 160-, 256- and 320-bit hash values. The latter two, RIPEMD-256 and RIPEMD-320, are slightly modified versions of the 128- and 160-bit hashes, and offer no additional security against cryptanalysis. The longer hash values do, of course, diminish the probability of random collisions.
At the time of the proposal, 128-bit hash functions were considered too weak to become standards, as birthday attacks with a complexity of 2^{64} would soon be feasible. Therefore, the emphasis of the proposal is on RIPEMD-160.
The hashes were designed with the goal to change the structure of MD4 and RIPEMD as little as possible, and be secure rather than fast.
All of the strengthened RIPEMD hashes employ two parallel lines of hashing. There are five 32-bit variables used in each line of RIPEMD-160, and the compression function has five rounds and five boolean functions. The padding scheme of MD4 is used, as well as the initial values of MD4, with the fifth being `0xc3d2e1f0` (`f0 e1 d2 c3`). The lines have different shift distances ($s^l_{i,k}$ and $s^r_{i,k}$) and different message word selection permutations (π^l_i and π^r_i), all have been chosen to fulfill specific criteria. The boolean functions are the same as in MD5:

$$\begin{aligned} f_1(x,y,z) &= x \oplus y \oplus z \\ f_2(x,y,z) &= (x \wedge y) \vee (\bar{x} \wedge z) \\ f_3(x,y,z) &= (x \vee \bar{y}) \oplus z \\ f_4(x,y,z) &= (x \wedge z) \vee (y \wedge \bar{z}) \\ f_5(x,y,z) &= x \oplus (y \vee \bar{z}) \end{aligned}$$

$f_4(x,y,z) = f_2(z,x,y)$, and $f_5(x,y,z) = f_3(y,z,x)$.
The additive constants for each round are the integer parts of square roots (the l line) and cube roots (the r line) of the first four primes times 2^{30}, and 0, similar to MD4 and RIPEMD. The operation of each step of RIPEMD-160 is:

$$\begin{aligned} w^l &= (w^l + f_i(x^l, y^l, z^l) + M_{\pi^l_i(k)} + C^l_i)^{\lll s^l_{i,k}} + v^l; \quad y^l = y^l \lll 10 \\ w^r &= (w^r + f_{5-i}(x^r, y^r, z^r) + M_{\pi^r_i(k)} + C^r_i)^{\lll s^r_{i,k}} + v^r; \quad y^r = y^r \lll 10 \end{aligned}$$

for each line l and r, using the same notation as above.
The functions applied in the lines are different, one line uses f_i, the other f_{5-i}

in the round i. One variable is added in each step, another variable is shifted by 10 bits, a value that is not used for any other shift. After all 80 steps, the results of both lines are added to the chaining variables and then discarded, as the two sets of variables are set to the intermediate hash value in the beginning of each pass of the compression function.

RIPEMD-128 uses only four 32-bit variables in each line, and the fifth round is omitted. Thus, the function f_5 is not used, along with the two additive constants of that round. Also, addition of one value (v), and shifting one variable (y) is left out. Therefore, RIPEMD-128 is a considerably different hash function.
The operation of RIPEMD-128 is:

$$\begin{aligned} w^l &= (w^l + f_i(x^l, y^l, z^l) + M_{\pi_i^l(k)} + C_i^l)^{\lll s_{i,k}^l} \\ w^r &= (w^r + f_{4-i}(x^r, y^r, z^r) + M_{\pi_i^r(k)} + C_i^r)^{\lll s_{i,k}^r} \end{aligned}$$

The extensions to 256- and 320-bit versions are achieved by excluding the addition of both lines at the end of the compression function. However, to maintain proper interaction between both lines, after each step a variable of one line is swapped with the corresponding variable of the other line. This means that RIPEMD-256 will run faster than RIPEMD-160, and will probably be less secure.

No successful cryptanalysis of any version of the strengthened RIPEMD hash functions has been published.

4.7 HAVAL

In 1992, HAVAL was published as a cryptographic hash function with variable security [48]. It has 15 different levels of security, as the number of rounds can be chosen between three, four, and five, and the hashlength can be chosen between 128 bit and 256 bit in 32-bit increments. Additionally, HAVAL uses boolean functions with specific properties that had been found the same year. The structure of HAVAL is similar to that of MD4, but many things have been altered.
The blocklength of HAVAL is 1024 bits. The padding method of MD4 has been extended. After a 1 bit and zero or more 0 bits, 16 additional bits composed of the version used (3 bits, only one version has been proposed), the number of passes (3 bits), and the length of the hash (10 bits), are appended before

the 64 bits of the length of the message.
HAVAL uses eight words as its internal state. Because of the bigger blocksize, each round consists of 32 passes over the corresponding boolean function, one for each input word. Additive constants are used in all passes except the first, and are different for each step. The initialization vector and the constants are consecutive words from the fractional part of π.
The five functions used by HAVAL are the following:
($x_i x_j$ denotes $x_i \wedge x_j$)

$$\begin{aligned}
f_1(x_6,x_5,x_4,x_3,x_2,x_1,x_0) &= x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus x_0x_1 \oplus x_0 \\
f_2(x_6,x_5,x_4,x_3,x_2,x_1,x_0) &= x_1x_2x_3 \oplus x_2x_4x_5 \oplus x_1x_2 \oplus x_1x_4 \oplus \\
&\quad x_2x_6 \oplus x_3x_5 \oplus x_4x_5 \oplus x_0x_2 \oplus x_0 \\
f_3(x_6,x_5,x_4,x_3,x_2,x_1,x_0) &= x_1x_2x_3 \oplus x_1x_4 \oplus x_2x_5 \oplus \\
&\quad x_3x_6 \oplus x_0x_3 \oplus x_0 \\
f_4(x_6,x_5,x_4,x_3,x_2,x_1,x_0) &= x_1x_2x_3 \oplus x_2x_4x_5 \oplus x_3x_4x_6 \oplus x_1x_4 \oplus \\
&\quad x_2x_6 \oplus x_3x_4 \oplus x_3x_5 \oplus x_3x_6 \oplus x_4x_5 \oplus \\
&\quad x_4x_6 \oplus x_0x_4 \oplus x_0 \\
f_5(x_6,x_5,x_4,x_3,x_2,x_1,x_0) &= x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus \\
&\quad x_0x_1x_2x_3 \oplus x_0x_5 \oplus x_0
\end{aligned}$$

Depending on the number of rounds, not all functions might be used.
In every step of HAVAL, the following operation is executed:

$$x_0 \;=\; f_i(x_1,x_2,x_3,x_4,x_5,x_6,x_7)^{\lll 7} \;+\; x_0{}^{\lll 11} \;+\; M_{\pi_i(k)} + C_{i,k}$$

The result of the round function f_i is circular-shifted with a constant shift distance, so is the previous value of the variable x_0. Then the sum of the two values, one message word $M_{\pi_i(k)}$, and one constant $C_{i,k}$ is taken as the next intermediate value. Between the steps, the variables are swapped by an additional permutation. There are different permutations for each number of rounds. After at most 160 steps, one message block has been processed.
At the end, a final transformation is applied to the variables to produce the message digest. If a 256-bit output is desired, then all eight words are used, written in little-endian notation. Otherwise, different bytes of variables are added together to produce a shorter output.

From 2000 to 2003, attacks on reduced versions of HAVAL have been published. At Asiacrypt 2003, collisions for three rounds were presented, along with an attack with a complexity of 2^{29} hash calculations [49]. Because of the structure of HAVAL, the attack produces colliding messages for all hashlengths. In

her note, Xiaoyun Wang showed collisions for three-round HAVAL-128 with a claimed computational effort of 2^6 hash computations [40]. The four-round version of HAVAL was broken in 2006, when two different attacks were published. One attack finds colliding messages with a complexity of less than 2^{32} for the first block and less than 2^{29} for the second [50], the other one presents two methods with a complexity of 2^{36} and 2^{43} [51]. An attack on five-round HAVAL is shortly explained in the same paper, it finds collisions with a probability of 2^{-123}.

4.8 SHA-0 and SHA-1

The SHA hashes were designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) as a government standard (as part of the Federal Information Processing Standards (FIPS)). They are specified in FIPS publication 180-2, the Secure Hash Standard (SHS) [20].
SHA-0 and SHA-1 produce 160-bit hashes. The structure of the two functions is similar to MD4 and MD5, but several changes and modifications have been introduced to increase their security.
SHA-0 was the first published SHA hash, made public in 1993. However, in 1995 the NSA suggested minimal changes to the standard because of security issues. The NSA did not disclose any further explanations. This change lead to the hash function SHA-1, which can be considered the most popular and the most widely used hash function yet. It is used in many security applications, and part of many other protocols as well as numerous standards, for example in TLS (Transport Layer Security) / SSL (Secure Sockets Layer), PGP (Pretty Good Privacy), S/MIME (Secure / Multipurpose Internet Mail Extensions), and IPSec (Internet Protocol Security).
The SHA hashes do not employ the same padding method as the MD4 hash. After appending a 1 bit and 0 bits appropriately, the length is appended as a 64-bit integer in big-endian notation. SHA-1 uses 512-bit blocks, 5 32-bit chaining variables and an additional 5-word state inside the compression function. Four of these chaining variables, a, b, c, and d, are initialized to the same values as MD4 and MD5, e to `0xc3d2e1f0` (`f0 e1 d2 c3`), which is the same as in the RIPEMD-160 hash. The following functions are used by the

compression function of SHA-1:

$$\begin{aligned} f_1(x,y,z) &= (x \wedge y) \oplus (\bar{x} \wedge z) \\ f_2(x,y,z) &= x \oplus y \oplus z \\ f_3(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ f_4(x,y,z) &= x \oplus y \oplus z \end{aligned}$$

Before each execution of the compression function of SHA-1, the 16-word input block is expanded into 80 words W_t. The original message block makes up the first 16 words. Then, 64 words are generated by xoring four of the previous words, and circular-shifting the result by one bit:

$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})^{\lll 1} \qquad \text{for } 16 \leq t \leq 79$$

This circular-shift is the only difference from SHA-1 to SHA-0, where no shifting occurred at this stage.
The compression function has 80 steps in four rounds. In each step, one variable is circular-shifted, all variables are interchanged, and the round function is carried out:

$$T = a^{\lll 5} + f_i(b,c,d) + e + C_t + W_t; \quad b = b^{\lll 30}$$

Each round has a unique constant C_t, these are $\lfloor \sqrt{r} \cdot 2^{30} \rfloor, r \in \{2,3,5,10\}$, respectively. After the compression function is completed, the results are added to the chaining variables, which compose the message digest at the end.

The first result of cryptanalysis of SHA-0 was presented at Crypto '98 [52]. The authors state that a collision can be found with complexity 2^{61}. However, the attack is not applicable to SHA-1, thus suggesting that SHA-1 is indeed more secure than SHA-0. In 2004, near-collisions were found [53]. The hashes differ by only 18 bits. The generation of the messages had a complexity of 2^{40}. In the same paper, collisions of a reduced version of SHA-0 with 65 rounds were shown. The attack was generalized shortly after, and collisions for the full SHA-0 were found. The calculation has a complexity of 2^{51}, and finds colliding four-block messages using three near-collisions [54]. Finally, in 2005, collisions were generated with only 2^{39} hash operations [55].
Some of the methods used for the SHA-0 collisions can also be applied to SHA-1 collision search. After different cryptanalysts found several attacks on reduced versions of SHA-1, Xiaoyun Wang and her colleagues presented finding collisions with less than 2^{69} hash operations [56]. Soon, they improved their attack to a complexity of 2^{63} [57]. These results were published in August

2005, at the Crypto rump session. Further developments were published in 2006, including colliding messages for 64-round SHA-1 [58].
In July 2007, a group of austrian researchers started a distributed computing project to find a collision for the full SHA-1 (`http://boinc.iaik.tugraz.at`). Like the target collision of MD5, the calculations are done with the help of many computers which are connected to the search via BOINC. The search for the collision involves finding two related near-collisions where the second collision evens out the differences of the first. The calculations have a roughly estimated complexity of 2^{60} compression function invocations.

4.9 SHA-2

In 2002, the NSA and the NIST added three hash functions to the secure hash standard to incorporate the need for hashes that offer more security, especially longer hash values. Another hash function was added in 2004. These four new SHA versions are also known as SHA-2 hash functions as they are very similar to each other. However, the structure of the new hash functions is considerably different from SHA-1.
SHA-256 uses 32-bit words, SHA-512 uses 64-bit words throughout the operation. The initialization vectors for both functions consist of eight words generated from the square roots of the first eight primes. The compression function uses eight internal variables, in addition to eight chaining variables. Each input block of 16 words (512 or 1024 bit) is extended to 64 words for SHA-256 and 80 words for SHA-512. However, the extension is more complex than in SHA-1. Instead of xor and the circular-shift, modular addition and two additional "mixing functions" are used by the SHA-2 functions:

$$\sigma_i(x) \quad = \quad x^{\lll s_{i,1}} \ \oplus \ x^{\lll s_{i,2}} \ \oplus \ x^{\gg s_{i,3}}, \quad i \in \{0,1\}$$

The shift distances $s_{i,j}$ are different for SHA-256 and SHA-512. In addition to the circular-shift operations, logical-shift is used: $\alpha^{\gg s}$ denotes shifting the bits of α s places to the right, filling the left s bits with the value 0.
Beginning with the unmodified 16 input words, The extension generates 48 / 64 more words by computing σ_0 and σ_1 for two words and summing the result up with two more unmodified words to form each new value:

$$W_t \ = \ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$$

The SHA-2 variants now use two functions f_1 and f_2, along with another two mixing functions Σ_0 and Σ_1.

$$\begin{aligned} f_1(x,y,z) &= (x \wedge y) \oplus (\bar{x} \wedge z) \\ f_2(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\ \Sigma_i(x) &= x^{\ggg s_{i,4}} \oplus x^{\ggg s_{i,5}} \oplus x^{\ggg s_{i,6}}, \quad i \in \{0,1\} \end{aligned}$$

Again, SHA-256 and SHA-512 have different shift distances $s_{i,j}$. SHA-1 also employs the functions f_1 and f_2, but the parity function of SHA-1 ($f(x,y,z) = x \oplus y \oplus z$) was not included in SHA-2.
In difference to SHA-1 and the MD4 and MD5 hashes, however, all four functions are used in every step for computing new values T_1 and T_2:

$$\begin{aligned} T_1 &= h + \Sigma_1(e) + f_1(e,f,g) + C_t + W_t; \\ T_2 &= \Sigma_0(a) + f_2(a,b,c) \end{aligned}$$

After that, the variables of the compression function are interchanged, and T_1 and T_2 are added to form the new value of a, while the value of h is dropped. T_1 is added to another variable. Additionally, the constants C_t are unique for each step. They are generated from the cube roots of the first 64 / 80 primes ($C_t = \lfloor (\sqrt[3]{p_t} - \lfloor \sqrt[3]{p_t} \rfloor) \cdot 2^{32} \rfloor$, p_t denoting the tth prime).
After the step operation has been iterated over all words of the expanded message, the values of a through h are added to the chaining variables. These form the final hash value by concatenation after the calculation is complete.
Another two hash functions, SHA-224 and SHA-384, are also part of the SHA-2 family of hash functions. SHA-384 is essentially a truncated version of SHA-512 with a different set of initialization variables, while SHA-224 is derived from SHA-256 in the same way. To obtain shorter hash values, output consists of only the first 384 or 224 bits. The initialization vectors are created from the square roots of primes 9 through 16. SHA-224 was added in 2004 to supply a hash function with the same security (in respect to collision resistance) as Triple-DES.

Current cryptanalytic results are unable to pose threats to the collision resistance of the SHA-2 hashes, and even more to their preimage resistance. However, there are several papers with interesting results. Henri Gilbert and Helena Handschuh show that most of the previous attacks on hash functions do not apply to SHA-256 and SHA-512, but also that slight modifications of SHA-256 are able to weaken the hash considerably [59]. Another result is that the Σ and σ functions are vital for the security of the SHA-2 hash functions. Without them, collisions can be found with a complexity of 2^{64} [60].

4.10 Tiger

The Tiger hash function was designed in 1995. After several hash functions had been shown to be weak, which implicated other, derived hash functions to be weak as well, it was designed as a hash with a completely different design, independent of the popular hashes derived from MD4. The Tiger hash was published in [28]. It is a 192 bit hash function, but in its specification it is noted to truncate its output to form 160-bit and 128-bit hash functions.
Tiger is specifically designed for 64-bit CPUs, especially the DEC Alpha processor. At the time, the Alpha was one of two available 64-bit CPUs, the other being the MIPS R4000 line of processors. Other 64-bit CPUs were released later (the first Sun UltraSPARC processor was released in 1995, however, PA-RISC by HP, IBMs PowerPC, and Intels Itanium were not available until 1996, 1997, and 2001, respectively), but they were believed to phase out 32-bit CPUs much faster than they actually did. Additionally, other cryptographic hash functions were unable to benefit from 64-bit words and 64-bit instructions. This is, for example, because of their strong serial flow which prevents carrying out two 32-bit operations using one 64-bit instruction, in addition to 32-bit operations like shifts.
Tiger was designed with much more emphasis on the hardware platform than other hash functions before. It makes specific use of several features of the Alpha architecture, however, it was made to work well on all systems, 32-bit as well as 64-bit architectures. Tiger is as fast as SHA-1 on 32-bit hardware. Also, certain parts of the hash can be parallelized, so calculation can be done more efficiently using pipelining capabilities of most CPUs.
The compression function uses four s-boxes of $256 \cdot 8$ bytes each, mapping bytes to 64-bit words. Additionally, it uses multiplications by small constants (5, 7, and 9, for which the Alpha has fast implementations), shifts (no circular shifts), and addition, subtraction, xor, and bit invertation (`not`) operations. These form an internal "block cipher like" function with a 512-bit key and 192-bit input and output. The hash function then follows the Davies-Meyer construction method, where, after encryption of the intermediate hash value with the input block as the key, the next intermediate hash value is combined with the previous. This is done to ensure the non-reversibility of the compression function. All calculations are of course done on 64-bit words, addition, subtraction, and multiplication are done modulo 2^{64}. Padding is done as in SHA-1.
The three variables of Tiger are initialized as follows:

$$
\begin{aligned}
a &= \texttt{0x0123456789abcdef} \\
b &= \texttt{0xfedcba9876543210} \\
c &= \texttt{0xf096a5b4c3b2e187}
\end{aligned}
$$

Another three variables are used as chaining variables. The input block is divided into eight words $x_0 \dots x_7$. The computation of the compression function consists of three rounds. In between the rounds, a key schedule is carried out. This key schedule mixes the input words using 16 steps. One step of each round consists of the following computations, they are carried out for each input word x (possibly mixed by the key schedule).

$$
\begin{aligned}
c &= c \oplus x \\
a &= a - (t_1(c_0) \oplus t_2(c_2) \oplus t_3(c_4) \oplus t_4(c_6)) \\
b &= b + (t_4(c_1) \oplus t_3(c_3) \oplus t_2(c_5) \oplus t_1(c_7)) \\
b &= b \cdot m
\end{aligned}
$$

Here, s-box lookups are denoted by the functions t_i which take a byte as input ($t_i : \{0,1\}^8 \rightarrow \{0,1\}^{64}, \quad i \in \{1, \dots, 4\}$). For this, the variable c is split up into eight bytes, $c = c_0 \dots c_7$. The multiplication factor m is an additional parameter of each round.
The words a, b, and c are interchanged before each round. After execution of the compression function, the variables are combined with the corresponding chaining variables, one via xor, one via addition and one via subtraction.

There has only one cryptanalysis been published for Tiger [61]. However, an attack was found that was able to find collisions of 16-round Tiger with 2^{44} calls to the compression function. Also, pseudo-near-collisions of 20-round Tiger were found with 2^{49} compression function invocations. They had a difference of only six bits.

4.11 Whirlpool

Whirlpool, a cryptographic hash function named after the Whirlpool galaxy, was first published in 2000. It has undergone two small changes since. It is based on a modified version of the block cipher Rijndael, which is used with the Miyaguchi-Preneel scheme to form a hash function. The blocksize and the hashsize both are 512 bits. Whirlpool is a recommended NESSIE (New European Schemes for Signatures, Integrity, and Encryption) primitive, and an ISO standard. It has been published in [62].

The initialization vector of Whirlpool is 0^{512}. Padding is done with one 1 bit and 0 bits as usual, then the length of the input is appended as a 256-bit integer.
The Whirlpool hash function is specified as a series of matrix operations. All matrices are 8×8 matrices over $GF(2^8)$ (bytes). Except for the constants, the algorithm is endianess-independent.
Whirlpool uses the following functions:

- μ and μ^{-1} convert 512-bit strings to the matrix form, and back.
- γ applies Whirlpools s-box to all elements of the matrix individually (this is parallelizable).
- π cyclically permutates the matrix.
- θ, the "linear diffusion layer", mixes the bytes in each row.
- $\sigma[k]$ is the key addition. The key k is also a matrix, and corresponding elements of each matrix are xored to form the output matrix.

With this notation (with $\circ$ as the composition operator of functions, $f \circ g$ specifies $f(g(x))$), the round function of Whirlpool can be described as

$$\rho[k] \;=\; \sigma[k] \;\circ\; \theta \;\circ\; \pi \;\circ\; \gamma$$

A key schedule which expands the 512-bit key to a sequence of round key matrices $K^0, \ldots, K^R$, is used.

$$K^0 \;=\; K; \quad K^r \;=\; \rho[c^r](K^{r-1}), \quad r > 0$$

The matrix c^r is derived from the s-box. Therefore, the round key K^r is encrypted using a constant key to form the next round key K^{r+1}.
The blockcipher W is defined by

$$W[K] \;=\; \rho[K^{10}] \;\circ\; \ldots \;\circ\; \rho[K^1] \;\circ\; \sigma[K^0]$$

Thus, W has ten rounds of encryption with the expanded key ($\rho[K^r]$), combined with xor of the key K. The number of rounds can easily be changed to provide for more security.
To compute the Whirlpool hash of a message M, its blocks m_i form the key for the block cipher W, and the intermediate hash value H_{i-1} is encrypted, and then xored with the message block and the intermediate value H_{i-1} to form H_i.

$$H_i \;=\; W[H_{i-1}](\mu(m_i)) \;\oplus\; H_{i-1} \;\oplus\; \mu(m_i)$$

The Whirlpool message digest, after all m_t input blocks have been processed, is $\mu^{-1}(H_t)$.
For all parts of the hash, specific predetermined security requirements were met. Whirlpool has undergone two changes, both increased cryptographic properties of parts of the hash. The function can easily be changed to more (or less) rounds.
The rather mathematical specification of the hash function can be easily implemented. On 64-bit platforms, eight table lookups and eight xor operations are needed per encryption round ρ. On platforms with smaller word lengths, implementations with similar speed are possible. However, the speed of Whirlpool is considerably less than that of hash functions like MD5 or SHA-1, however, its speed is comparable with hash functions of similar security.

No cryptanalysis of the relatively new algorithm has been published. Most attacks on the block cipher Rijndael are not applicable to Whirlpool, however, no attacks have been found to seriously threaten the security of Rijndael, either.

4.12 RadioGatún

The RadioGatún hash function family consists of 64 functions, RadioGatún[1] through RadioGatún[64], according to their wordsize. The hashes are specified in [63]. The RadioGatún hashes are based on the Panama hash function, which has roots in other hash functions.
The hash functions do not follow the Merkle-Damgård construction scheme. Instead of a compression function, RadioGatún introduces an "iterative mangling function". Rather than using the complete state, it returns only two words as the output. However, the output function can be called as often as desired. Therefore, it can be viewed as truncating the infinite output stream to a proper length. Of course, the output stream is periodic after a certain length, since any algorithm with a state of n bits can output a non-periodic sequence of at most 2^n bits.
The state of RadioGatún is seperated into two parts: the belt and the mill. There are three functions in the operation of the hash, one for input, one for output, and a round function. The round function can be executed by itself, but it is also used by the former two.
The mill consists of 19 words and is the non-linear component of the hash. The belt is comprised of 39 words. Its function can be compared to the message

expansion SHA-like hashes. All variables are initialized to 0.
The input function inserts three words at the start of the belt and the end of the mill. After that, the round function is carried out once. The last set of input words is padded with a 1 bit and 0 bits.
The output function first carries out the round function, and then returns two words of the mill. Between input and output, 16 blank rounds are executed.
The round function consists of four steps. These are the bell function, the mill function, and two feed forward functions. They transfer data from the mill into the belt and vice versa.
The belt function is a simple rotation. The mill to belt feed forward function xors a word of the mill into every third word of the belt. The mill function has four stages. The first provides nonlinearity by carrying out the function $a_i = a_i \oplus (a_{i+1} \vee \overline{a_{i+1}})$ for all words of the mill. Then, the words are circular-shifted and interchanged. The shift distances are different for all words. After that, the third stage computes the xor combination of three words for each word of the mill. At the end, the constant 1 is xored into the first word for asymmetry.
The fourth step of the round function is the belt to mill feed forward, which xors the three last words of the belt into the mill. All of these functions are invertible, therefore the round function is invertible.
The authors claim for RadioGatún[64] to have a security level equivalent to a ideal hash function with 1216 bits. This is given by the size of the mill, which holds 19 words, here, each is 64-bit wide. The security of RadioGatún[32] therefore is at 608 bit, and at 304 bit for RadioGatún[16].

Several security aspects are addressed in the publication of RadioGatún. The Panama hash, on which RadioGatún is based, showed severe weaknesses. They were studied in order to improve the RadioGatún functions.
It is important to point out that several attacks are feasible for RadioGatún[1]. The security claim also allows for brute force collision search against RadioGatún[4], and preimage search against RadioGatún[2], both have a complexity of 2^{38}. Producing random collisions and studying their effects throughout the hash is a good way to find weaknesses. Additionally, different cryptanalytic tools can easily be applied to the small internal state of only 58 bits. However, extending the results of these attacks to versions with a higher wordsize is not as simple.
Currently, no weaknesses of the RadioGatún family are known.

5 The Speed of Cryptographic Hashing

5.1 Introduction and Motivation

Of course, the most important characteristic of a cryptographic hash function are its security properties. For many applications, however, the speed of a hash function is of almost the same importance. The use of the MD4 hash function in current applications shows that, depending on the purpose, the speed might be even more important than specific security properties.
Naturally, there are limits to what is desirable. The data to be hashed has to be read or transmitted, which, apart from applications operating exclusively in memory, rarely exceeds speeds of 1 Gbit/second.
Apart from pure speed, it is often desirable to hash with small processing cost, especially in the case of multitasking systems. Therefore, hash functions should use only very limited resources.

In this chapter, the speed of the addressed cryptographic hash functions is analyzed. After some discussions of possibilities and influences, special implementations and hardware hashing, various platforms, as well as different compilers and their abilities to optimize will be examined.
Of course, the speed of outdated and deprecated hash functions like MD2 on modern systems provides no usable information at first sight. However, it serves as an overview over an important characteristic of the numerous hash functions. Even though about half of these hash functions must be considered broken and unfit for most purposes, a detailed analysis allows for a deep insight into the aspects of the different elements and concepts of the algorithms. Moreover, the temporal cost, and therefore the threat of an attack can be estimated with this data.
Since the execution speed of a hash function depends on many circumstances, it can not be determined by a single test. Like any other statistics, all test results have to be interpreted thoroughly.
The tests presented here focus on comparing the actual speed of the different hash functions in normal environments. Additionally, different hardware environments are compared against each other. Small aspects, like the behavior of the compression functions, round functions, or message expansion are not examined in detail.
Obviously, the number of insightful tests and enlightening results is unlimited. Sadly, only a very limited collection can be presented here.

5.2 Influencing Factors

Needless to say, the speed of a cryptographic hash function depends on many factors. Additionally, measurements are always based on particular circumstances, therefore generalizations about a slow or fast hash function are vague and unspecific.
In software, hashing speed is heavily determined by several factors:

- The hash function and the desired security. In general, short hash functions are faster than longer and more secure hashes. Increasing the number of rounds or choosing longer or more secure variants (for example the round parameter of HAVAL, or SHA-512 compared to SHA-256) increases the running time. Of course, output transformations like truncation have very little impact.

- The software implementation along with the compiler. The capabilities of the compiler and the optimizability of the source code also have an enormous effect on the achievable speed. Most hash functions are implemented in standard C to easily incorporate different hardware platforms and compilers, exchanging speed for easy portability. Therefore, code tailored towards a specific platform can give an advantage, as can a suitable and potent compiler.

- The hardware platform and the CPU. Obviously, the choice of the processor has a huge significance, as it dictates the instruction set and the word size. Byte-order conversions can slow down the execution as much as 10% and more [25]. Furthermore, the clock frequency has a linear influence on the speed of a hash function when comparing CPUs of the same type. However, the processor accounts for several other important aspects:
  The internal registers of the processor are very limited, but they can be accessed very quickly. If not all variables of a hash function can be stored in registers, the execution is slowed down considerably. An Intel x86 CPU has, for example, only four general-purpose registers, which is not enough to hold all variables and temporary results of the internal functions of MD5 or SHA-1. The RISC processors from MIPS, on the other hand, have 32 64-bit-wide general-purpose registers, enough to hold the eight variables of HAVAL, or even two matrices during Whirlpool calculation. Whenever data does not reside in registers, it has to be fetched from the caches. This does not only apply to the rather small set of internal and temporary values, but also to the message words. All SHA versions

expand the input message into 64 or 80 words, and most compression functions have a blocksize of at least 512 bits, so caching is always a factor. Cache sizes vary widely, as do cashing strategies and, of course, access times. The different cache levels (first level, second level and, on some platforms, even third level cache) and their sizes also vary widely. All these are additional factors.

Memory access times and other I/O transfer rates rarely have a noteworthy impact. A 1 GHz Intel Pentium 3 processor, for example, is able to do MD4 hashing (the fastest hash function measured) at a speed of around 970 Mbit/second, while the memory speed is more than twice as high.

5.3 Hardware Implementations

Hardware implementations of cryptographic hash functions are not in heavy use today. However, the UltraSPARC T2 processor by Sun is a perfect example for hardware hashing: it uses dedicated cryptographic units for hashing. Along with other cryptographic routines for encryption, the three hash functions MD5, SHA-1, and SHA-256 are implemented, and hash at over 30 Gbit/second [64].
All commonly used operations of a hash function can easily be implemented in hardware. Permutations, rotations and interchanging variables can be implemented in wiring, and are therefore the fastest operations. For all other operations, standard gates have to be used.
A cryptographic hash function design dedicated to implementation in hardware can take advantage of using any level of parallelization with a linear gain in speed. As already mentioned, no wordsize has to be considered. Instead, a whole block can be used for calculation at once. Virtually only the structure of the hash sets a limit to the achievable speed.
FPGAs can reach around 600 MHz clock frequency, most chips operate on 20 to 200 MHz. The clock frequency of ASICs can go well into the GHz range.
SHA-1 hashing, for example, can be implemented within 1–5 % of common FPGAs, with a throughput of well above 1 Gbit/second. On ASICs, an SHA-1 implementation takes up less than 23000 gates [65].

5.4 Optimized Implementations

As mentioned above, fine tuning a hash function for a specific platform and processor can result in a considerable increase in speed. These adaptations can range from special compiler optimizations and mild changes of the sources to a complete rewrite in assembler language. Through detailed knowledge of the underlying processor and thorough analysis of the hashing algorithm it is possible to gain a speedup factor of more than two in most cases.
The following table lists, in its first column, the speed of heavily optimized assembler implementations of several hash functions. The code takes advantage of many features specific to the Intel Pentium 3 processor like parallelizable execution, while avoiding many performance bottlenecks like register starvation and memory access stalls. Many of the techniques used are counter-intuitive, favoring memory or cache access over registers, for example.
The data was published in [27] along with a detailed explanation. The authors achieved even better results by executing two and three hashes in parallel. However, they are not included here since the parallel hashes are independent of each other.
For comparison, the speed of non-optimized code for the same type of processor is shown in the second column. These values were calculated from results of the speed test explained below. They were obtained with gcc 4.1.1. Out of all tested compiler switches, the optimizations `-O1` and `-O2` delivered the best results, the latter are marked (*).
Both columns list calculatory values of cycles/byte. They have the advantage of being independent of the processors clock frequency, but cannot be used to compare between different CPUs, let alone different architectures.
The third column shows the increase in speed, it is the quotient of the first two.

	Optimized	Standard	Speedup
MD5	5.53	11.44*	2.1
RIPEMD-128	9.41	15.03	1.6
RIPEMD-160	14.23	23.49	1.7
SHA-1	9.73	37.31*	3.8
SHA-256	23.73	45.91*	1.9
SHA-512	40.18	87.68	2.2
Whirlpool	36.52	141.25	3.9

A comparison shows a remarkable speedup in all hash functions. With SHA-1 and Whirlpool it reaches a factor of almost four. For SHA-1, the message expansion is independent of the hashing algorithm and can thus be executed

in parallel. Whirlpool uses even more sophisticated tricks: the 64-byte state was kept in eight MMX registers, and four substitution tables in the first-level cache, while the other four tables were generated when needed.
To conclude, taking steps to optimize a heavily used hash function can be very rewarding when speed is important.

5.5 The Test Environment

Many of the less used hash functions have only a few implementations, mostly for different languages. Heavily used hashes like MD5 and SHA-1, on the other hand, have several implementations for various demands, of course with different hashing rates. Obviously, testing all would be too extensive, therefore, only a few suitable implementations were tested.
Standard C is well suited for writing compact, quick code. With the exception of the SHA hashes, a reference implementation in standard C is part of all hash function proposals. The RFC document for SHA-1 includes an implementation, but no such code was published for the SHA-2 functions. However, most of the reference implementations are not very efficient. There are other implementations that show clear advantages, therefore, a few adequate ones were included in the test, unfortunately, such implementations could not be found for all hash functions. Except for an implementation of SHA-2 by Oliver Gay [66], which is marked with ^{G}, they were all written by Christophe Devine as part of an open-source cryptographic library [67]. They are marked with ^{D}.
The following implementations were used for the test:

- The reference implementations of MD2 [32], MD4 [18], and MD5 [19].
- An implementation of MD2, MD4, and MD5 by Christophe Devine [67], MD2^D, MD4^D, and MD5^D.
- The RFC implementation of SHA-1 [20].
- An implementation of SHA-1 and SHA-2 by Christophe Devine [67], SHA-1^D as well as SHA-256^D and SHA-512^D.
- An implementation of SHA-2 by Oliver Gay [66], SHA-256^G and SHA-512^G.
- The reference implementation of the three HAVAL hashes [48].
- The reference implementation of RIPEMD-160 and RIPEMD-128 [22].

- The reference implementation of Tiger [28].
- The reference implementation of Whirlpool [62].
- The reference implementation of two members of the RadioGatún family, RadioGatún32 and RadioGatún64 [63].

HAVAL basically includes three different hash functions, they were tested as HAVAL3, HAVAL4 and HAVAL5, according to their number of rounds. For the same reason, RIPEMD-128 and RIPEMD-160 were tested individually. For the SHA-2 hash functions, only SHA-256 and SHA-512 were tested, since SHA-224 and SHA-348 possess only slight differences to their longer counterparts that do not alter the hashing rate. Members of the RadioGatún hash function family operate on different wordsizes, RadioGatún32 and RadioGatún64 are suggested by the authors, and were both included here. Unfortunately, the testing code that is part of the published implementation refuses to run on big-endian platforms.
SHA-0 and RIPEMD were not tested, because they were completely replaced by their successors.
Altogether, 21 different hash function tests were administered.

On multitasking systems, an exact measurement of the efficiency of a program is not easy. The best way in standard C to determine processor time is the `clock()` function, as it allows to measure the time a block of code spends on calculations [68]. To a large extent, the return values of the `clock()` function are free of factors disturbing the accuracy of the measurement, like the processor usage of other applications and the operating system. However, the resolution of the measurements obtained is not higher than 10 ms. Unfortunately, a small overhead can not be completely avoided, and can vary based on different factors.
The test routine contains a loop that repeatedly hashes one block of 8192 zero bytes. Therefore, memory transfers should not be necessary, but the size of the buffer also allows the hash function to execute long enough in each cycle. Some preliminary tests showed a drop in speed for blocks smaller than 1 kb and larger than 64 kb.
The following block of C code was used for testing the speed of the hash functions. Since the implementations all have different interfaces (different functions are called with different parameters), it had to be modified to fit each implementation.

```
void TimeTrial() {
  HashContext state;
  unsigned char block[8192], digest[64];
  unsigned int i;
  clock_t t;
  for (i = 0; i < 8192; i++) block[i] = 0;

  t=-clock();
    HashInit(&state);
    for (i = 0; i < TEST_BLOCK_COUNT; i++)
      HashUpdate(&state, block, 8192);
    HashFinal(digest, &state);
  t+=clock();

  printf("Hash speed [kbit/sec]:   \%.1f \n",
    (float)(8192 * 8 * TEST_BLOCK_COUNT) /
      (float)t * CLOCKS_PER_SEC / 1024.0);
}
```

Between the two `clock()` function calls that perform the time measurement, the hashes initialization and finalization routines are each called once. In comparison to the majority of applications, the two routines are underrepresented here. In other scenarios, like computations for an attack, they might be executed even rarer. However, the functions have very little impact on the performance of the hash, but they should be factored in nonetheless.
The constant `TEST_BLOCK_COUNT` was used to easily change the duration of the test, and was set to 65536 for the final evaluations.
For each hash function, 2^{32} bit, or 512 MB were hashed (only 2^{27} bit were used for MD2 because it runs extremely slow). This ensures a running time between just less than 4 seconds (1 Gbit/sec) and 10 minutes (7 Mbit/sec), except for very slow systems. The resulting hash speed is an average over this time period. Therefore, influencing events that occur only for short time periods like a few milliseconds have little impact on the overall performance.
Because circumstances are never quite the same, even for quick tests and a small system load the results exhibit moderate variations between seemingly identical trials. However, these variations are acceptable as long as they do not exceed a certain limit.

To verify the reliability of this approach, it was run several times on the same computer (a 2.2 GHz AMD Athlon 64 3500+), simple optimizations

were used (`-O1`)). The system load was artificially increased between passes. Altogether, it can be said that the testing method delivers accurate, comparable results. The maximum deviation from the average is well below 2%. Moreover, without a change in system load, the results differ by less than 0.5%. The following comparison shows the deviation from the average hashing speed (given in Mbit/second) for each hash function.

	Average	deviation in % for load of			
	speed	0	4	8	16
MD2	42.8	- 0.12	- 0.41	- 0.41	0.94
MD2^D	48.1	- 0.04	0.01	- 0.74	0.77
MD4	1080.1	- 0.46	- 0.20	- 0.20	0.86
MD4^D	2136.2	- 0.65	- 0.13	0.92	- 0.13
MD5	897.3	- 0.04	0.01	- 0.74	0.77
MD5^D	1572.5	- 0.58	- 0.58	- 0.58	1.75
SHA-1	552.2	0.10	- 0.03	0.24	- 0.30
SHA-1^D	1147.4	- 0.84	0.27	- 0.84	1.41
SHA-256^G	612.0	- 0.26	- 0.11	- 0.41	0.79
SHA-256^D	582.2	- 0.07	0.21	- 0.07	- 0.07
SHA-512^G	936.2	0.34	- 0.12	0.34	- 0.57
SHA-512^D	1001.5	0.24	- 0.24	- 0.24	0.24
RIPEMD-128	910.7	- 0.28	0.39	- 0.50	0.39
RIPEMD-160	671.5	0.00	- 0.33	0.00	0.33
HAVAL3	1743.1	- 0.43	- 0.43	- 0.43	1.29
HAVAL4	1210.1	0.14	- 0.44	- 0.44	0.74
HAVAL5	1050.3	- 0.52	- 0.52	- 0.52	1.55
Tiger	1246.9	0.15	- 0.45	0.15	0.15
Whirlpool	194.1	- 0.10	- 0.09	- 0.09	0.29
RadioGatun[32]	268.1	- 0.13	0.39	- 0.65	0.39
RadioGatun[64]	521.3	0.63	0.12	0.12	- 0.88

The GNU Compiler Collection, gcc, was used to generate the executables for most tests. On the Sparc systems, Suns compiler `Sun C 5.8 2005/10/13` was additionally tested, and on the Pentium 3 system, Intels `ICC 10.0 20070809` was used. On a Power4 AIX system, the tests were run with IBMs `xlc 7.0.0.7` compiler. In chapter 5.15, their results are compared to those of gcc.

For a comprehensive test of which hashing speeds are achievable, rather than different compilers, trying different optimization options for each compiler is

much more important.
For gcc and Intels icc, general levels of optimization range from `-O0` to `-O3`, with the addition of `-Os` to minimize the size of the executable. The Sun compiler has six levels (`-xO0` to `-xO5`), so does Intels xlc. These command line arguments are short for a number of options. Moreover, there are additional options to control several more aspects of optimization.
For each system, several options were chosen and tested to determine which provide the best results. Of course, not all combinations could be tested, however, only a slight gain can be expected beyond the chosen optimizations.
In appendix B, all relevant results of all systems are shown. To compare the hashing speeds against each other, however, only the fastest results were chosen for each hash function. Additionally, only the results obtained by using gcc are used, because different compilers could not be tested on more than a few systems.

5.6 Hardware Platforms

Cryptographic hash functions are carried out on many different hardware systems, large and small. For attacks on the hash functions, even supercomputers and clusters can be used. Therefore, many systems are relevant for the speed test.
Results of the test suite are shown for the following systems:

1. An Intel Pentium 3, 1.0 GHz (available since 2000), `weide.unix-ag.uni-hannover.de`.

2. An Intel Pentium 4 (Prescott), running at 3.6 GHz (available since 2004), `basher.snils.de`.

3. An Intel Core 2 Duo, 2.0 GHz (available since 2006), `hex.knolle.no-ip.org`. All tests were run in 64-bit mode.

4. An AMD Athlon 64 X2 3800+, 2.0 GHz (available since 2006), `hiwi.uni-hannover.de`. The processor was also only tested in 64-bit mode.

5. A Sun UltraSPARC II 450 MHz CPU in a Sun Ultra Enterprise 220R (available since 1999), `studserv.stud.uni-hannover.de`.

6. A Sun UltraSPARC T1 (Niagara) processor at 1.0 GHz in a Sun Fire T-2000 (available since 2005), `studserv5.stud.uni-hannover.de`. The test only used one of 32 available threads.

7. A Sun UltraSPARC IIIi 1.5 GHz CPU in a Sun Fire V245 (available since 2006), `studserv2.stud.uni-hannover.de`.

8. A SGI MIPS R14000 processor at 600 MHz in an SGI Onyx (available since 2001), `onyx3.rrzn.uni-hannover.de`.

9. An 833 MHz Alpha CPU in a COMPAQ AlphaServer DS20E (available since 2001), `birke.unix-ag.uni-hannover.de`.

10. An IBM Power5 processor running at 1.65 GHz in an OpenPower 720 (available since 2005), `tick.rz.uni-augsburg.de`.

11. An Intel Itanium 2 processor (McKinley) at 900 MHz in a HP rx2600 (available since 2003), `sollnix.unix-ag.uni-hannover.de`.

12. An IBM Power4 processor running at 1.3 GHz in a pSeries 690 (available since 2002), `hanni.hlrn.de`. Unfortunately, the CPU could only be tested with 32-bit binaries, and is therefore only used in chapter 5.15 to show differences of compilers.

The first system is roughly a "minimal standard", typical personal computers usually have more computing power. The following three computers are state-of-the-art systems, used for many purposes and with fairly high computing power. Additionally, similar kinds of processors can be used for small clusters. The next three systems can be viewed as typical servers. The UltraSPARC II is only slightly older as the Pentium 1.0 GHz. The UltraSPARC IIIi and the UltraSPARC T1 are common, progressive processors used for a broad range of servers. The MIPS CPU is a niche product, but forms an interesting platform nonetheless. The same thing can be said about the Alpha platform today. Both processors are no longer produced.
Systems 10 and 11 are typical processors for high-performance computing. However, for this purpose both are somewhat outdated, and newer versions are available: The Power6 processor was released this year, and there are four newer Itanium 2 cores available. Conforming with Moore's Law, they can be expected to perform about four to eight times as fast.
Systems 1 and 2 are strictly 32-bit computers, all others allow execution of 64-bit code, while some are capable of running 32-bit and 64-bit code at the same time. x86 and x64 systems all use little-endian byte order. The Sparc and MIPS machines exclusively use big-endian byte order, while Alpha, Power4 and Itanium 2 are capable of running either. On Alpha and Itanium, little-endian byte-ordering was used, and big-endian on the Power processor.

5.7 Hashing Speed

Running the test suite on each platform determines the hashing speed for each hash function for each of a number of different compiler options. The following list was compiled by taking only the best results of each platform. However, all results shown here were obtained with gcc. Differences for Intels icc and Suns compiler are discussed in chapter 5.15. For tables with all results, including the other compilers used, see Appendix B.

	1	2	3	4	5	6	7	8	9	10	11
MD2	22.9	69.9	41.1	39.0	10.6	7.2	31.4	18.0	11.6	29.7	17.6
MD2^D	23.7	72.7	50.4	44.0	10.7	7.7	31.8	12.4	12.2	31.9	16.0
MD4	839.3	2844.4	1692.5	1402.7	293.8	236.5	1050.2	384.2	285.4	1003.4	387.2
MD4^D	950.3	4137.4	2167.2	2178.7	475.2	621.5	1788.6	689.6	896.0	1536.4	1101.5
MD5	667.1	2327.3	1183.8	1122.2	228.1	201.2	822.5	301.6	244.6	686.5	324.9
MD5^D	875.2	3690.1	1473.4	1569.3	318.0	443.3	1201.2	456.1	511.4	879.3	683.0
SHA-1	204.5	762.8	687.2	691.9	81.0	65.3	301.6	136.6	160.1	325.4	170.0
SHA-1^D	354.6	787.7	1101.1	1052.9	194.7	216.6	865.9	361.5	551.0	1219.3	559.9
SHA-256^G	166.2	751.6	611.3	617.8	66.5	78.3	288.4	93.8	192.1	387.3	196.8
SHA-256^D	235.9	636.0	572.9	510.7	63.9	103.8	378.6	153.2	235.0	615.0	294.8
SHA-512^G	87.0	210.9	935.1	948.1	105.2	120.8	431.2	154.3	325.4	589.9	409.5
SHA-512^D	107.8	224.7	1092.3	1083.6	113.6	136.2	480.8	199.2	357.4	573.0	466.6
RIPEMD-128	507.6	1861.8	954.8	1021.4	220.9	254.7	996.6	370.3	548.7	993.9	570.3
RIPEMD-160	324.8	817.6	646.0	747.4	136.7	166.5	613.2	237.9	398.4	718.2	394.7
HAVAL3	548.3	1134.6	1678.7	2155.8	243.8	274.9	991.8	416.3	540.2	845.6	542.1
HAVAL4	414.1	856.9	1208.2	1422.2	181.2	217.9	694.2	300.7	409.7	658.4	414.7
HAVAL5	252.5	786.2	1008.9	1187.2	150.5	177.9	616.9	267.2	352.6	558.5	350.5
Tiger	267.4	689.6	1197.6	1194.2	199.4	155.6	795.3	273.2	1213.6	667.9	638.6
Whirlpool	54.0	152.2	314.3	197.4	18.3	16.0	72.2	61.3	98.0	188.4	129.8
RadioGatun[32]	66.2	166.7	217.1	244.9					87.5		173.8
RadioGatun[64]	36.0	108.9	434.2	487.5					175.9		379.9

Hashing speeds in Mbit/second

5.8 The MD Family

On all tested systems, MD2 was by far the slowest hash function. Its speed barely exceeds 70 Mbit/second on the fastest system, and stays at around 10 Mbit/second on both the UltraSPARC II and the UltraSPARC T1.
As MD2 emphasizes on byte-operations, this is no surprise. However, even when MD2s speed was four or eight times as high (in correspondence to the wordsize), its speed barely reaches that of other hash functions. This is due to the compression function having 18 rounds over the extended message of 48 bytes. The substitution box lookup done in each step is slow, and other hash functions compute considerably less steps for each block.
The implementation by Christophe Devine is only slightly faster on all systems, on most of the non-x86 computers it runs even slower than the reference implementation.
However, the slow speed of MD2 was one of the main reasons of the development of MD4 and other hash functions.

The speed of MD5 and especially MD4 is very high. The optimization of these hashes for 32-bit platforms was quite successful. The MD4 hash is, apart from one exception (the Tiger hash on the Alpha processor), the fastest hash function on all systems that were tested.
Often reaching rates of above 1 Gbit/second, even the slower systems (Pentium 3, UltraSPARC II and UltraSPARC N1) are able to perform MD5 hashing at well above 300 Mbit/second. The difference between MD5 and MD2 is a factor of well above 20, averaging almost 40 for Christophe Devines implementations.
The reference implementation of MD4 hashes around 25% faster than that of MD5, the alternate versions differ by about 45% on average, however, on the 32-bit x86 platforms (systems 1 and 2), the difference is merely 10%.
The value of over 4 Gbit/second on a Pentium 4 processor is remarkable, and can be accounted to the high clock frequency of the processor, fast caches and good 32-bit performance.
The main difference between MD4 and MD5 is the added fourth round, and in each step, another addition is carried out. This roughly explains the speed difference.
Christophe Devines implementations deliver higher results on all processors. The speedup lies at over 85% and 60% on average for MD4 and MD5, respectively. With the exception of the Pentium 3 and the Core 2 processor, it is higher than 40% on all systems, often reaching 200% or even 300%. This is the case on the Alpha, Itanium 2, and UltraSPARC T1 processors. The imple-

mentations nicely show that a significantly higher speed can be achieved over the reference implementations, and that different implementations can have a widely varying speed difference.
However, the MD4 and MD5 hash functions must be considered both fast and insecure in equal measure.

5.9 The SHA Family

SHA-1 reaches a speed of between 200 Mbit/second and 1.1 Gbit/second. Although SHA-1 is not a 64-bit function, the Pentium 4 is outperformed by both the Core 2 and the Athlon 64 processor by more than 30% regarding the faster implementation. The reference implementation values differ only by 10%, here, the Pentium hashes faster.
The implementation of Christophe Devine reaches even higher differences to the reference implementation than the MD4 and MD5 hashes, coming to a factor of over 2.4 in all non-x86 systems. The difference on the Pentium 4 is very low, on the other x86 systems, however, the factor is higher than 1.7.
While comparisons between the reference implementations of MD5 and SHA-1 show a decrease in speed by about 30%–60% (the average is at 50%), the alternate implementations do not completely follow this decrease. The average hashing speed is at 73% when comparing Christophe Devines MD5 and SHA-1 implementation, and, surprisingly, on the Alpha and the Power4 processor, SHA-1 hashing is done faster than MD5. SHA-1 even hashes almost as fast as MD4 on the Power4 CPU.
The lower speed of SHA-1 is a result of the message expansion, and the noticeably more complicated routine of each step than in MD5. Its higher security is paid by slower hashing rates, the difference is not unexpected.

On newer computers like the Core 2 and the Athlon 64, SHA-512 hashes at above 1 Gbit/second, SHA-256 is only half as fast.
When comparing the SHA-2 hash functions, it is important to note that SHA-256 uses a 32-bit wordsize, while SHA-512 uses 64-bit words. Therefore, the speed of SHA-512 is faster on 64-bit systems, and should reach around twice the speed of SHA-256. There is no further difference between the two hashes that would have an effect on the hashing rate. For the 32-bit systems, however, the 64-bit operations have to be emulated in some way.
The SHA-256 hashing rate on the Pentium 3 is 1.9–2.2 times as high as the rate of SHA-512. On the Pentium 4 on the other hand, the difference is a factor of over 3 for the implementations of Oliver Gay, 2.8 for Christophe Devines

implementation, SHA-512 showing a significant disadvantage over SHA-256, probably because of 64-bit emulation methods.
For the 64-bit systems, SHA-512 hashes faster by a factor of between 1.3 and 2.1, with an average of 1.6 for both implementations. Especially for the UltraSPARC II and T1 and the SGI MIPS CPUs, the difference is very small. The x86-64 systems show a behavior close to the naively expected, with SHA-512 hashing at 190% and 210% of the speed of SHA-256.
The differences between the two implementations are altogether not too big. Christphe Devines implementation of SHA-512 is slightly faster on all systems, with a higher speed of 15% on average, for SHA-256, it is slightly slower on some systems, but on others it is more than 50% faster.
SHA-1 runs up to 3.5 times faster than both SHA-2 functions on all systems. However, on the 64-bit systems, the difference of SHA-512 and SHA-1 has an average of 50%, SHA-1 runs at about twice the speed of SHA-256. This makes SHA-512 a suitable replacement for SHA-1 on the 64-bit systems, and would perfectly justify a 64-bit wordsize, 256-bit SHA-2 function, as 64-bit computing can now be considered standard.

5.10 RIPEMD-160

The speed of RIPEMD-128 is about 55% higher than that of RIPEMD-160. For the Pentium 4, however, the difference is almost 130% as the speed of RIPEMD-128 is 1.8 Gbit/second.
The difference between the two hashes is that RIPEMD-160 uses an additional variable and an additional round. Also, in each step of RIPEMD-128, two less operations are carried out. The omited round only accounts for roughly 20% of the speed difference.
The structure of the RIPEMD hashes was designed to be similar to two lines of MD4. RIPEMD-128 is closer to MD4 than RIPEMD-160. The speed of the MD4 implemetation of Christophe Devine is about twice as high as that of RIPEMD-128, the reference implementation runs between 40% and 75% faster. Considering that each line of RIPEMD-128 is more complex than MD4, the expected speed would be even lower. Of course, this is heavily dependant on the implementation.
However, one reason for the comparatively good performance of RIPEMD-128 might be that most processors are able to compute both lines with only a slightly lower speed because of parallelization within the processor. The MIPS and the Power5 CPUs, for example, hash RIPEMD-128 with almost the same speed as the reference implementation of MD4. Both are RISC processors with

many registers, which can speed up such computations significantly. Nonetheless, this would require a very high level of optimization to be done by the compiler, and may as well be based on poor performance of the MD4 hashes.

5.11 HAVAL

The three HAVAL hash functions differ by the number of rounds. On average, HAVAL3 performs about 30% faster than HAVAL4, and 60% faster than HAVAL5. This is directly related to the number of rounds. The deviation from the average is very low on all processors except for two. The Pentium 3 computes HAVAL3 hashes about 2.2 times faster than HAVAL5 hashes. The Athlon 64 has, in comparison to HAVAL3, considerably low values in both HAVAL4 and HAVAL5.
The boolean functions used by HAVAL have an acceptable performance altogether. HAVAL3 reaches 1.5–2 Gbit/second on the fast systems, here, the versions with more rounds both stay above 1 Gbit/second. The slower systems 1, 8, 9, and 11 hash HAVAL3 at around 400–550 Mbit/second, HAVAL4 at around 300–400 Mbit/second and HAVAL5 at around 250–350 Mbit/second.
For some reason, the Core 2 and the Athlon 64 outperform the Pentium 4, while they were both outperformed by the Pentium 4 in MD4 and MD5. While the speed difference of MD4 and MD5 was low between Core 2 and Athlon 64, there is a considerable difference in all HAVAL versions. This shows the vast differences between the two processors: although their computing power is similar, they have different features, and therefore noticeable advantages or disadvantages in certain areas.
In the publication of HAVAL [48], the authors claim the speed of HAVAL3 to be 160% of MD5, HAVAL4 115%, and HAVAL5 100%. This is clearly not the case on the systems tested here. Even when comparing to the slower reference implementation, HAVAL3, HAVAL4, and HAVAL5 mostly reach 130%, 100%, and around 80%, respectively. Nonetheless, HAVAL5 is substantially faster than SHA-256.

5.12 Tiger

The Tiger hash was specifically designed for the Alpha processor, resulting in a very high hashing speed on this system of over 1.2 Gbit/second. MD4 hashes at only 900 Mbit/second on the same system. On the Itanium 2, Tiger is also very fast, possibly indicating similarities to the Alpha processor. On all

other systems, the speed of Tiger matches that of HAVAL5. Also, Tiger is on average 10% slower than Christophe Devines MD4.
Being a 64-bit hash function, Tiger is executed slow on the 32-bit CPUs. The Core 2 and the Athlon both fail to achieve a higher hashing speed than the five year older Alpha.
While Tiger is not extremely slow on other systems than the Alpha, which was one of the design goals, there are other hash functions that perform better. However, it is shown that tailoring a hash function for a specific processor can be rewarded by a very high speed.

5.13 Whirlpool

On all tested systems, Whirlpool hashes quite slowly. A speed of more than 200 Mbit/second is only reached by the Core 2 CPU, on most systems Whirlpool performs only 2-3 times as fast as MD2, and is barely able to reach 5% of the speed of MD4. However, the systems 3, 9, 10, and 11 stand out with Whirlpool hashing at a rate of 10%–15% of the hashing speed of MD4.
Comparing the two 512-bit hash functions, shows similar results, as the speed of SHA-512 is 3–4 times as high as the speed of Whirlpool for most systems, all UltraSPARC processors even hash SHA-512 six and eight times as fast.
The implementation of Whirlpool mainly uses table lookups, obviously the approach delivers quite poor performance in comparison to most other hash functions. However, it has to be noted that Whirlpool is a hash function directly based on a block cipher. Other such constructions have similarly low speeds. On the other hand, the security of the hash, is rarely diminished.

5.14 RadioGatún

The difference between RadioGatún[32] and RadioGatún[64] is only the wordsize of the hash function. The 32-bit systems compute RadioGatún[32] hashes with 150%–180% of the speed of RadioGatún[64]. On the 64-bit systems RadioGatún[64] hashes at exactly twice the speed of the 32-bit version. Only on the Itanium 2 RadioGatún[64] is 120% faster.
The speed of the faster of the RadioGatún versions tested roughly performs 50%–80% as good as SHA-512.
The test routine of RadioGatún only tests the input speed. However, the output function should have the same speed, and is mostly used to generate sequences of less than 1024 bytes.

If the security of the hash functions is not reduced by cryptanalysis, then the RadioGatún family delivers quite high performance in respect to its output length.

5.15 Compiler Efficiency

The speed of a cryptographic hash function can be increased by use of a compiler that optimizes well. The gnu gcc compiler is known not to reach the efficiency of other proprietary compilers in many circumstances.

Comparisons of Intels icc to the gcc on a Pentium 3 system show that Intels compiler produces faster code in almost all cases. Interestingly, gcc can optimize the SHA-2 implementations of Oliver Gay better than icc. The fastest hash rates were all obtained with the Intel compiler though.
The rates of Christophe Devines SHA-1 implementation, as well as Tiger and HAVAL5 are around 50% higher for the icc binaries. The average difference of the two is almost 14%.

The xlc compiler by IBM delivers similar results on the Power4 processor. However, SHA-1 and especially RIPEMD hashing is faster with gcc.
Whirlpool hashes around 70% faster when xlc is used, HAVAL, Tiger and the reference imlementations of MD4 and MD5 gain 20%–35%.
A comparison between gcc, icc and xlc efficiency of Christophe Devines implementations suggest, that the high speed of the latter is achieved because gcc can optimize the code very well on these systems.
The difference of xlc compared to gcc is 12.5% on average.

On the UltraSPARC II system, hash function binaries compiled using Suns compiler run 19% faster on average than those compiled using gcc. For many hash functions, the speedup is higher than 40%, Whirlpool even executes almost twice as fast. Where gcc delivers better results, the difference is less than 10%.
The differences on the UltraSPARC T1 are less high. Most hash functions run less than between 5% slower and 10% faster when using Suns compiler. The average speedup is only 5.6%.
The results of the UltraSPARC III system lay in between, many hash functions run 20% faster. Here, too, the difference of the two Whirlpool versions is at 70%. RIPEMD-160 runs 20% slower when Suns compiler is used.

	1 icc	1 gcc	12 xlc	12 gcc	5 cc	5 gcc	6 cc	6 gcc	7 cc	7 gcc
MD2	23.2	22.9	24.3	23.4	13.4	10.6	7.2	7.2	38.4	31.4
MD2^D	24.5	23.7	24.8	24.2	14.0	10.7	7.8	7.7	41.2	31.8
MD4	839.3	839.3	906.2	629.2	390.5	293.8	247.0	236.5	1308.6	1050.2
MD4^D	1215.4	950.3	979.9	908.2	517.8	475.2	606.8	621.5	2027.7	1788.6
MD5	644.0	667.1	586.0	451.1	299.0	228.1	210.6	201.2	1003.9	822.5
MD5^D	920.4	875.2	558.8	557.3	374.4	318.0	439.5	443.3	1347.4	1201.2
SHA-1	202.9	204.5	343.9	303.6	114.4	81.0	75.4	65.3	375.1	301.6
SHA-1^D	580.2	354.6	781.7	855.1	216.1	194.7	252.7	216.6	820.8	865.9
SHA-256^G	160.8	166.2	353.7	245.3	97.6	66.5	98.3	78.3	359.0	288.4
SHA-256^D	259.9	235.9	440.9	437.6	93.1	63.9	114.2	103.8	353.4	378.6
SHA-512^G	82.0	87.0	220.3	156.3	151.1	105.2	142.7	120.8	539.7	431.2
SHA-512^D	113.0	107.8	232.3	193.0	162.0	113.6	143.8	136.2	571.3	480.8
RIPEMD-128	512.0	507.6	390.1	644.0	254.4	220.9	290.9	254.7	946.0	996.6
RIPEMD-160	376.5	324.8	297.2	467.0	125.7	136.7	187.4	166.5	498.3	613.2
HAVAL3	675.9	548.3	757.1	627.2	252.8	243.8	271.6	274.9	984.6	991.8
HAVAL4	492.9	414.1	595.3	464.9	166.2	181.2	210.8	217.9	748.8	694.2
HAVAL5	414.1	252.5	496.5	383.2	142.1	150.5	163.3	177.9	638.0	616.9
Tiger	469.2	267.4	460.7	356.8	196.5	199.4	146.3	155.6	1013.9	795.3
Whirlpool	77.6	54.0	131.4	64.0	36.4	18.3	17.6	16.0	156.5	72.2
Avg. diff. [%]	13.84		12.54		19.26		5.61		13.58	

Hashing speeds in Mbit/second

6 Conclusion

Over the past 15 years, cryptographic hashing has undergone many changes. As a fairly new part of cryptography, several methods of analysis and attack have been added to the toolbox of hashing cryptanalysis in the past years. Some methods of analyzing block ciphers, like linear cryptanalysis and differential cryptanalysis, took some time to be applied to hash functions. Additionally, several new concepts of constructing secure cryptographic hash functions, as well as methods of increasing the strength of cryptographic properties of hash functions or existing constructions, have been found after Ralph C. Merkle first proposed the concept of hashing.

In the past few years, all important hash functions of the early design have been broken. All hash functions based on MD4 have been shown to be insecure, except for the strengthened RIPEMD hashes. For SHA-1, collisions will probably be found within one year. However, because of considerable differences between SHA-1 and the SHA-2 hash functions, SHA-256 and SHA-512 are not affected and seem a suitable replacement for now.
While a lot of hash functions have been proposed, many of them have lasted less than a year before serious flaws were discovered. Examples of this include FFT-Hash (1990), N-Hash (1990), and Smash (2005).

Apart from the achievements of cryptanalysis, the computing power of today's computers and networks rule out 128-bit hash functions for environments where collision resistance is a requirement. At least within the next 20 years, 160-bit hash functions should be phased out for the same reason. On the other hand, 512-bit hash functions have an extra security margin for possible attacks, as brute force attacks even on 256-bit hash functions can be considered impossible for the foreseeable future. Nevertheless, there are many applications of cryptographic hash functions where collision resistance is not a required property.

For all hash functions in use, the threat of preimage calculation is low, as there is no feasible preimage attack for most hash functions. Additionally, preimages or second preimages would have to be computed separately for each hash, although for message authentication the feasibility of calculating preimages, especially meaningful second preimages, would be disastrous.
Collisions, however, can be used for many different attacks. Any collision can serve for this purpose, as some of the examples of MD5 collisions show. Despite this, efficient collision search of a hash function might make other attacks

possible. An important application of this are message pairs that are collisions for multiple hash functions.

Currently, the United States National Institute for Standards and Technology (NIST) is in the process of choosing a new standard to propose as the AHS, the Advanced Hashing Standard. The process is similar to that of the AES, the Advanced Encryption Standard. The preliminary timeline spans from 2007 to 2012, so it is at a very early stage at the moment. During the process, hash functions will be proposed, and – much more importantly – evaluated, by the open cryptographic community. There will be at least two rounds of evaluations of the proposals, each lasting at least one year.
There will be a number of requirements for the new hash functions. Apart from the hashsize, the speed of implementations in various processors (8-bit, 32-bit, and 64-bit CPUs) and in hardware will be evaluated [69]. Also, at least one parameter to vary the level of security should be present (the number of rounds, for example). However, the full list of requirements has not been compiled yet. Also, many requirements are desired for specific applications, many of which are conflicting.
The RadioGatún hash function family is one of the candidates in this process. Recently, several other new hashes have been designed.
Certainly, the whole procedure will bring forth very interesting results, along with new hash functions, possibly new or more sophisticated attacks, and several new methods for cryptographic hashing.

An extensive test of the speeds of cryptographic hash functions has shown many interesting results. Even on systems with similar processors and architectures, the hashing rates vary widely. Each hardware platform has its own advantages, so different hash functions are better suited for different systems. From fast execution of a specific hash on a system, no generalization can be made about the expected speed of other hash functions.
Moreover, the compiler options have a considerable impact on the speed of a hash function. Surprisingly, the highest level of optimization does not always yield the fastest hash function. This is a result of the hash algorithms using very basic operations in dense code. Therefore, the compiler does not always chose the fastest arrangement of operations and registers. Quite often, optimizing for a small executable also results in hash function that performs best. Tuning a hash function towards a different processor, as was done in the Tiger hash, can result in exceptional speeds. However, a cryptographic hash function should be designed for around 15–20 years of operation. During this time, processors undergo many changes, so the intended architecture might not be

used any more after 5 years.
The security of a hash function cannot be measured. Many flaws in hashes were discovered long after their publication. Therefore, measuring the speed of a hash based on its length is not a suitable approach.

The performance of processors can not be directly compared. Especially CPUs of different hardware platforms have advantages in very different computing areas (integer performance, floating point performance in single or double precision, I/O transfer, to name just a few). Therefore, the performance is measured by benchmark programs, and then indicated by a score. There are many different benchmarks for all areas. 10 to 20 years ago, results of the dhrystone benchmark were widely accepted as an accurate measurement for the speed of a system. Other benchmarks, like SPECs integer benchmark, are used today, but license fees have to be paid for the test.
Dividing the speed of a hash by the result of a suitable benchmark would make it possible to directly compare the performance of a hash over all systems. As it turns out, however, the dhrystone benchmark is absolutely unsuited for such a comparison.
Additionally, dividing hashing results by the average achieved on the system, which would represent a "hash performance" benchmark, results in misleading and not representative values.

In general, this document merely scratches the surface of the vast field of cryptographic hash functions, especially hashing cryptanalysis. Unfortunately, the time and scope of the document did not permit a deep look into interesting methods like linear and differential cryptanalysis.

In summary, as with all cryptographic applications, careful considerations have to be made when choosing a hash function for a specific purpose. The process of designing a new cryptographic hash function is extremely complicated, and many aspects have to be examined. The past has not only shown that cryptographers have been too optimistic about the security of hash functions and that flaws can be found more or less in any hash function, but also that the understanding of cryptographic hashing is still in its infancy.
Hopefully, this document provided some insight over the wide range of cryptographic hashing to the reader.

A Random Collision Search

When taking n random numbers $r_1, \ldots, r_n$ from 0 to $d-1$, the probability $p = 1 - \bar{p}$ of having a collision ($r_i = r_j, i \neq j$) can be calculated as follows.

$$\begin{aligned} \bar{p} &= 1 \cdot \left(\frac{d-1}{d}\right) \cdot \left(\frac{d-2}{d}\right) \cdot \ \cdots \ \cdot \left(\frac{d-(n-1)}{d}\right) \\ &= \prod_{i=0}^{n-1} \left(1 - \frac{i}{d}\right) \\ &= \frac{d!}{d^n \ (d-n)!} \end{aligned}$$

Since

$$e^{-i/d} = 1 - \frac{i}{d} + \frac{i^2}{2!\,d^2} + \frac{i^3}{3!\,d^3} + \ \cdots$$

$\bar{p}$ can be expressed as (approximating $e^{-i/d} \approx 1 - \frac{i}{d}$, which is fine for $i \ll d$)

$$\begin{aligned} \bar{p} &\approx 1 \cdot e^{-1/d} \cdot e^{-2/d} \cdot \ \cdots \ \cdot e^{-(n-1)/d} \\ \bar{p} &\approx e^{-(n(n-1))/2\,d} \end{aligned}$$

and finally

$$p \approx 1 - e^{-(n^2/2\,d)}$$

Inverting gives a formula for calculating n, the number of random numbers needed to achieve a collision with probability p

$$n \approx \sqrt{-2\,d \cdot \ln(1-p)}$$

So, to expect a collision with a probability of 50%, about $1.177 \cdot \sqrt{d}$ random objects have to be tried. When taking exactly $\sqrt{d}$ numbers, the probability of finding a collision is about 39.35%.
Therefore, finding random collisions in cryptographic hash functions is sometimes called a *birthday attack*. For $d = 2^m$, the complexity of this attack obviously is $\mathcal{O}(2^{m/2}) = \mathcal{O}(\sqrt{d})$.
However, the space required for finding random collisions when applying this method also is $\mathcal{O}(2^{m/2})$, since every result has to be compared to all previous results. Also, the output values have to be mapped to the corresponding input

values. Altogether, the attack is a simple time / space tradeoff.

Further improvements, however, lead to a more efficient collision search method that needs far less memory, and is parallelizable. The idea behind this attack is similar to the Pollard rho method. Additionally, distinguished points are used for an even faster parallel attack [70].

B Complete Results of the Speed Tests

For all systems, the test suit has collected multiple results for each hash function by varying the compiler options. All relevant results are shown here.
The highest values of each hash function and each system was used in chapter 5. These are marked in dark gray in all tables. Additionally, values that are within 97% of the maximum are marked in gray. The slowest result of each hash is shown in light gray:

maximum	high	minimum

The systems are shown in chapter 5.6, and are numbered accordingly. All hashing speeds are given in Mbit/second.
The following list shows all compiler options used. To save space, the columns are numbered with the corresponding value from this list.

1. `-O0`
2. `-O1`
3. `-O2`
4. `-O3`
5. `-Os`
6. `-O1 -march=pentium3`
7. `-O2 -march=pentium3`
8. `-O1 -march=pentium4`
9. `-O2 -march=pentium4`
10. `-O1 -march=prescott`
11. `-O2 -march=prescott`
12. `-O1 -march=nocona`
13. `-O2 -march=nocona`
14. `-O1 -march=k8`
15. `-O2 -march=k8`

16. `-O1 -march=opteron`
17. `-O2 -march=opteron`
18. `-O0 -m64`
19. `-O1 -m64`
20. `-O2 -m64`
21. `-O3 -m64`
22. `-Os -m64`
23. `-fast`
24. `-xO5`
25. `-xtarget=native`
26. `-xtarget=native64`
27. `-xtarget=native64 -fast`
28. `-O2 -mcpu=ultrasparc`
29. `-O2 -mcpu=ultrasparc3`
30. `-O2 -m64 -mcpu=ultrasparc`
31. `-O2 -m64 -mcpu=ultrasparc3`
32. `-qO0`
33. `-qO1`
34. `-qO2`
35. `-qO3`
36. `-qO4`
37. `-qO5`
38. `-qOs`

System 1: Pentium 3

	1	2	3	4	5	6	7
MD2	7.4	22.9	22.9	22.3	20.5	21.6	21.3
MD2^D	7.7	20.2	23.7	22.3	22.8	19.0	22.4
MD4	221.0	786.2	839.3	591.0	765.6	732.7	795.3
MD4^D	435.7	950.3	910.2	884.7	910.2	886.6	846.3
MD5	168.3	632.1	667.1	494.7	611.3	581.0	618.7
MD5^D	316.0	875.2	778.7	734.0	713.6	817.6	721.1
SHA-1	87.4	191.8	204.5	198.6	179.5	177.6	193.1
SHA-1^D	327.2	354.6	341.3	334.4	311.2	329.5	325.6
SHA-256^G	105.6	153.4	166.2	127.6	152.7	144.5	155.2
SHA-256^D	176.8	235.9	215.0	217.6	207.0	216.8	205.5
SHA-512^G	63.9	87.0	80.6	75.9	82.7	81.5	78.2
SHA-512^D	81.2	107.8	97.1	97.9	96.2	97.9	94.8
RIPEMD-128	168.5	507.6	459.2	478.5	429.8	464.9	435.7
RIPEMD-160	136.1	324.8	280.2	280.0	253.6	310.3	271.6
HAVAL3	374.1	548.3	400.0	394.6	373.0	513.9	399.2
HAVAL4	290.7	414.1	309.8	302.3	318.5	387.5	303.9
HAVAL5	252.5	224.3	99.5	96.9	104.9	209.0	98.7
Tiger	192.0	267.4	245.0	240.9	243.2	249.9	239.0
Whirlpool	43.5	54.0	49.3	48.2	48.5	50.2	50.0
RadioGatun[32]	52.8	66.2	43.0	42.7	53.0	61.6	40.3
RadioGatun[64]	27.5	36.0	33.1	31.4	31.3	33.0	31.1

System 1: Pentium 3 (Intel compiler)

	1	2	3	4	5	6	7
MD2	19.4	23.2	19.5	19.3	18.5	22.4	19.2
MD2^D	20.1	24.5	20.2	19.4	22.6	24.0	20.4
MD4	807.9	784.7	806.3	793.8	839.3	668.2	768.5
MD4^D	1208.2	1166.9	1215.4	1011.3	1042.2	1140.9	1194.2
MD5	609.5	617.8	611.3	601.5	644.0	575.3	559.6
MD5^D	918.4	910.2	920.4	806.3	781.7	882.7	912.2
SHA-1	202.1	194.4	202.9	185.0	180.6	186.6	180.0
SHA-1^D	572.9	580.2	575.3	561.9	496.5	565.0	502.0
SHA-256^G	154.4	160.8	154.6	149.7	149.2	158.2	141.4
SHA-256^D	258.9	259.9	259.7	253.3	258.9	246.3	257.8
SHA-512^G	82.0	80.6	82.0	78.6	79.2	77.8	79.5
SHA-512^D	85.8	113.0	86.1	77.3	110.9	112.1	86.5
RIPEMD-128	511.4	506.3	511.4	498.3	508.2	508.2	512.0
RIPEMD-160	376.5	371.3	371.7	368.0	364.1	364.1	372.4
HAVAL3	675.9	645.0	671.5	649.1	623.4	630.1	673.7
HAVAL4	489.4	473.5	492.9	476.8	458.7	465.4	492.9
HAVAL5	408.8	389.0	414.1	400.0	384.2	383.2	412.5
Tiger	415.4	405.9	413.7	402.0	397.3	385.0	469.2
Whirlpool	77.5	73.7	77.6	74.5	66.3	56.5	76.5
RadioGatun[32]	70.6	68.3	70.9	69.1	67.9	39.4	41.5
RadioGatun[64]	41.0	41.8	41.3	39.3	37.5	25.0	29.0

System 2: Pentium 4

	1	2	3	4	5	6	7	8	9	10	11
MD2	23.4	68.1	60.1	62.7	61.0	68.4	57.9	69.9	69.9	69.9	64.6
MD2^D	24.4	63.1	61.8	63.4	55.7	63.4	63.1	67.7	72.3	72.7	67.7
MD4	658.5	2805.5	2238.3	1685.6	2844.4	2694.7	2659.7	2528.4	2592.4	2844.4	2625.6
MD4^D	924.6	2482.4	3303.2	2989.8	3792.6	2467.5	3250.8	4137.4	1828.6	2482.4	3471.2
MD5	423.1	2068.7	2301.1	1517.0	2048.0	2100.5	2327.3	2263.0	2214.1	2250.5	2238.3
MD5^D	918.4	3011.8	2340.6	2381.4	2226.1	3011.8	2381.4	3690.1	3079.7	3056.7	2659.7
SHA-1	282.3	693.1	714.8	762.8	630.2	711.1	721.1	632.1	742.0	691.9	751.6
SHA-1^D	628.2	786.2	669.3	674.8	695.4	787.7	675.9	764.2	647.1	778.7	662.8
SHA-256^G	395.7	681.5	734.1	618.7	751.6	538.9	734.1	677.0	706.2	678.1	713.6
SHA-256^D	439.0	621.5	630.2	616.9	618.7	629.2	632.1	614.1	595.3	636.0	594.5
SHA-512^G	190.1	210.9	205.0	199.4	187.4	208.8	204.5	210.2	194.4	210.1	194.3
SHA-512^D	173.6	224.4	195.3	195.9	201.8	224.7	195.3	224.2	189.9	219.7	189.5
RIPEMD-128	318.5	1861.8	1505.9	1534.1	1545.7	1836.8	1522.7	1788.6	1735.6	1820.4	1557.4
RIPEMD-160	241.1	771.4	650.2	660.6	561.1	755.7	663.9	798.4	817.6	754.3	605.9
HAVAL3	898.2	1134.6	675.9	706.2	725.0	1110.0	707.4	1131.5	890.4	1131.5	656.4
HAVAL4	630.2	837.6	638.0	646.1	607.7	853.3	632.1	792.3	562.6	856.9	591.9
HAVAL5	514.6	581.0	252.8	265.6	282.7	580.2	267.4	786.2	444.3	544.7	242.8
Tiger	499.5	686.1	588.5	615.0	628.2	650.2	617.8	675.9	625.3	689.6	631.1
Whirlpool	128.8	150.5	146.4	146.2	150.8	151.9	150.1	152.2	147.5	128.4	144.0
RadioGatun[32]	141.2	166.7	121.0	117.7	132.8	164.0	121.0	164.6	125.1	152.9	123.0
RadioGatun[64]	76.7	108.1	98.8	98.9	102.4	108.9	98.7	107.5	103.7	108.6	102.3

System 3: Core 2 Duo (only 64-bit binaries were generated)

	1	2	3	4	5	12	13
MD2	14.9	40.9	40.7	41.1	40.2	40.9	40.4
MD2^D	16.2	50.4	49.8	49.0	49.6	50.4	49.6
MD4	381.0	1365.3	1342.9	1692.5	1272.0	1321.3	1356.3
MD4^D	869.6	2167.2	2058.3	2058.3	2017.7	2167.2	2058.3
MD5	281.9	1024.0	1024.0	1183.8	957.0	1013.8	1016.4
MD5^D	609.5	1473.4	1402.7	1402.7	1383.8	1468.1	1393.2
SHA-1	224.9	651.2	659.6	656.4	629.2	624.4	687.2
SHA-1^D	607.7	1101.1	1016.4	1016.4	1036.9	1101.1	972.9
SHA-256^G	261.6	610.4	575.3	525.1	602.3	611.3	576.9
SHA-256^D	323.8	564.2	572.9	572.9	568.9	564.2	564.2
SHA-512^G	400.0	935.1	880.8	888.5	920.4	924.6	875.2
SHA-512^D	455.6	1089.3	1080.7	1080.7	1072.2	1092.2	1089.3
RIPEMD-128	338.5	941.6	954.8	954.8	918.4	941.6	954.8
RIPEMD-160	264.9	632.1	645.0	646.0	636.0	630.1	635.0
HAVAL3	809.5	1651.6	1658.3	1638.4	1494.9	1625.4	1678.7
HAVAL4	625.3	1187.2	1183.8	1183.8	1072.2	1208.2	1187.2
HAVAL5	531.9	975.2	972.9	972.9	920.4	975.2	1008.9
Tiger	628.2	1119.1	1150.5	1144.1	1197.6	1113.0	1173.6
Whirlpool	188.1	314.3	266.5	261.9	252.7	314.3	259.2
RadioGatun[32]	127.6	217.1	217.1	217.1	204.2	217.1	205.1
RadioGatun[64]	254.6	434.2	434.2	434.2	411.7	434.2	411.7

System 4: Athlon 64 (only 64-bit binaries were generated)

	1	2	3	4	5	14	15	16	17
MD2	13.7	37.5	38.8	37.5	39.0	37.4	38.8	37.3	38.8
MD2^D	11.3	43.5	43.5	44.0	43.2	43.1	43.5	43.1	43.5
MD4	314.6	1170.3	1190.7	1402.7	1180.4	1153.8	1190.7	1157.0	1190.7
MD4^D	778.7	2178.7	1870.3	1878.9	1905.1	2167.2	1878.9	2167.2	1878.9
MD5	250.4	963.8	977.6	1122.2	975.2	957.0	977.6	954.8	977.6
MD5^D	568.9	1557.4	1417.3	1417.3	1442.2	1569.3	1417.3	1563.3	1417.3
SHA-1	150.2	556.5	686.1	691.9	516.5	558.0	680.4	558.0	680.4
SHA-1^D	599.7	1052.9	1050.2	1052.9	1052.9	1050.2	1050.2	1050.2	1047.6
SHA-256^G	170.8	572.9	617.8	565.7	615.9	575.3	614.1	577.7	615.0
SHA-256^D	304.8	503.2	510.7	508.8	509.4	504.4	508.2	503.8	471.9
SHA-512^G	262.9	888.5	948.1	939.4	941.6	871.5	945.9	871.5	943.8
SHA-512^D	291.3	1052.9	1055.7	1052.9	1083.6	1050.2	1063.9	1047.6	1061.1
RIPEMD-128	315.6	970.6	999.0	1021.4	979.9	966.0	999.0	968.3	999.0
RIPEMD-160	246.6	723.7	731.4	747.4	724.9	722.4	732.7	721.1	734.0
HAVAL3	723.7	2155.8	2027.7	2027.7	1651.6	2068.7	1950.4	2079.2	1950.4
HAVAL4	584.3	1422.2	1383.8	1383.8	1197.6	1388.5	1356.3	1383.8	1351.8
HAVAL5	506.9	1187.2	1134.6	1134.6	1006.4	1160.3	1110.0	1160.3	1110.0
Tiger	332.2	1011.3	1131.5	1131.5	1194.2	1008.9	1125.3	1008.9	1128.4
Whirlpool	85.2	195.7	160.8	158.5	166.4	197.4	174.1	197.4	173.9
RadioGatun[32]	127.3	243.7	206.7	205.1	219.9	244.9	204.2	243.7	205.1
RadioGatun[64]	250.8	480.7	411.7	413.4	436.1	487.5	413.4	485.2	413.4

System 5: UltraSPARC II

	1	2	3	4	5	18	19	20	21	22
MD2	2.1	9.0	10.4	10.5	10.6	7.6	9.9	10.6	10.5	1.8
MD2^D	2.0	9.2	9.4	9.4	9.3	7.6	10.3	10.7	10.1	1.8
MD4	53.9	274.9	293.8	185.3	287.6	112.9	226.9	242.8	235.3	38.3
MD4^D	100.3	381.4	475.2	451.1	475.2	353.1	404.7	418.0	415.4	96.4
MD5	40.4	202.0	228.1	134.3	220.7	155.0	185.0	196.6	189.5	31.0
MD5^D	78.2	251.6	318.0	293.0	317.5	239.3	265.8	102.3	266.8	74.7
SHA-1	17.5	54.8	81.0	69.1	63.4	48.4	68.8	67.6	63.6	16.0
SHA-1^D	51.8	133.6	184.8	194.7	180.6	129.0	173.2	178.9	179.2	50.1
SHA-256^G	17.8	48.9	63.9	66.5	65.9	45.2	64.5	65.5	66.4	16.2
SHA-256^D	22.0	59.0	63.9	52.5	62.4	52.8	36.7	32.0	37.3	15.0
SHA-512^G	12.7	19.8	25.0	24.5	24.5	69.8	99.2	105.2	101.3	25.3
SHA-512^D	15.2	26.9	28.1	25.7	27.4	74.0	107.4	111.0	113.6	29.5
RIPEMD-128	37.7	154.6	219.7	220.9	209.2	142.6	206.3	215.0	206.1	34.9
RIPEMD-160	25.5	94.2	136.7	129.3	128.5	87.7	126.1	130.6	128.3	22.7
HAVAL3	62.3	200.2	236.2	231.9	231.4	192.5	238.3	243.8	224.7	59.6
HAVAL4	41.6	147.3	180.8	175.8	175.2	143.9	181.2	179.4	180.0	40.4
HAVAL5	33.1	125.0	148.3	150.5	143.4	116.7	113.0	120.3	124.2	32.0
Tiger	32.3	86.2	87.6	89.0	87.5	143.5	190.0	184.8	199.4	42.0
Whirlpool	9.2	13.0	5.1	2.3	5.2	18.3	11.0	10.9	11.1	8.9

System 5: UltraSPARC II (Suns compiler)

	1	3	23	24	25	26	27
MD2	3.3	13.4	13.0	13.0	3.2	4.6	10.4
MD2^D	5.2	11.4	14.0	13.3	5.3	4.4	11.0
MD4	162.6	327.2	390.5	372.7	165.5	122.7	346.8
MD4^D	313.1	502.6	517.8	517.8	310.1	279.2	490.5
MD5	124.3	263.7	299.0	293.2	126.6	107.3	259.4
MD5^D	229.3	353.1	374.4	358.7	233.5	204.3	341.9
SHA-1	26.7	68.7	94.9	70.5	26.5	24.4	114.4
SHA-1^D	71.6	194.1	216.1	198.9	72.2	88.2	208.8
SHA-256^G	27.1	75.6	89.9	73.4	27.4	30.6	97.6
SHA-256^D	23.5	87.4	81.5	93.1	23.6	21.1	78.9
SHA-512^G	19.3	96.0	101.7	100.4	19.1	47.0	151.1
SHA-512^D	19.5	98.4	103.2	99.5	22.4	57.0	162.0
RIPEMD-128	105.2	217.5	251.6	120.3	105.0	126.3	254.4
RIPEMD-160	48.1	118.9	125.7	124.2	49.5	70.5	123.5
HAVAL3	148.5	231.9	252.8	222.2	148.5	182.9	248.8
HAVAL4	106.2	162.7	166.2	162.5	105.9	135.3	161.4
HAVAL5	85.6	142.1	122.7	140.5	85.9	110.8	121.5
Tiger	34.5	164.0	141.3	196.5	35.4	57.2	140.7
Whirlpool	8.7	31.4	34.3	19.1	8.6	11.5	36.4

System 6: UltraSPARC T1, part 1

	1	2	3	4	5	28	29
MD2	1.8	6.9	6.9	6.8	6.6	7.1	6.8
MD2^D	1.8	7.5	7.4	6.0	7.1	7.5	7.3
MD4	56.7	236.5	235.5	232.7	220.2	220.3	223.3
MD4^D	122.1	621.5	595.3	517.2	582.6	348.3	565.7
MD5	43.9	198.7	201.2	197.4	191.4	181.6	192.8
MD5^D	96.7	443.3	434.8	307.7	354.0	293.4	431.2
SHA-1	16.6	61.1	59.4	52.8	55.2	64.8	65.3
SHA-1^D	43.6	216.6	170.0	180.5	212.4	131.2	206.8
SHA-256^G	16.1	74.5	70.6	61.5	71.5	71.9	74.0
SHA-256^D	23.5	103.8	63.6	50.7	64.2	77.8	62.8
SHA-512^G	9.3	13.1	15.3	14.7	16.5	14.4	13.3
SHA-512^D	10.9	22.1	16.8	15.3	16.5	14.2	15.0
RIPEMD-128	40.8	247.9	245.4	244.8	244.5	227.7	247.2
RIPEMD-160	23.3	160.3	166.5	157.0	153.4	128.5	134.4
HAVAL3	64.3	274.0	272.2	274.9	247.2	206.4	198.1
HAVAL4	37.8	215.8	215.7	205.2	204.1	118.7	138.0
HAVAL5	29.9	175.7	177.9	170.9	165.0	107.8	110.1
Tiger	26.2	51.0	47.9	48.8	47.2	44.7	41.7
Whirlpool	7.4	8.8	2.9	2.9	2.8	3.6	3.3

System 6: UltraSPARC T1, part 2

	18	19	20	21	22	30	31
MD2	1.2	7.1	6.9	6.9	6.9	7.0	7.2
MD2^D	1.6	7.4	7.3	7.3	7.3	7.5	7.7
MD4	30.5	145.5	151.5	151.5	151.0	148.8	144.2
MD4^D	104.7	588.5	556.5	556.5	568.9	312.2	512.0
MD5	27.3	132.3	137.6	137.6	136.5	134.0	130.3
MD5^D	81.7	419.7	353.1	343.6	345.9	230.6	299.4
SHA-1	15.5	57.9	61.2	64.7	54.9	61.3	60.2
SHA-1^D	39.1	183.5	191.8	196.9	193.9	122.2	159.5
SHA-256^G	18.3	72.0	75.6	75.3	78.3	71.7	74.5
SHA-256^D	13.3	90.3	39.5	35.3	41.0	43.5	50.2
SHA-512^G	28.3	115.8	97.5	119.3	120.8	117.7	114.5
SHA-512^D	37.1	132.3	135.8	136.2	135.8	116.6	124.0
RIPEMD-128	37.7	254.7	218.8	249.8	247.3	225.6	201.6
RIPEMD-160	21.1	164.6	147.6	156.6	163.1	136.9	124.9
HAVAL3	62.0	246.2	254.7	259.9	245.0	220.7	218.8
HAVAL4	39.9	217.9	174.1	160.0	156.6	143.0	140.7
HAVAL5	29.9	175.9	126.7	128.0	136.5	102.0	106.4
Tiger	45.8	155.6	149.3	143.4	147.6	152.8	121.3
Whirlpool	7.7	16.0	6.6	6.7	6.5	10.1	7.6

System 6: UltraSPARC T1 (Suns compiler)

	1	3	23	24	25	26	27
MD2	2.3	7.0	7.2	7.1	2.3	3.7	7.2
MD2^D	3.8	7.6	7.6	7.5	3.9	5.1	7.8
MD4	182.4	218.1	247.0	245.1	204.1	133.2	162.6
MD4^D	392.3	606.8	515.9	593.6	372.4	406.8	535.4
MD5	112.2	199.6	210.6	208.0	121.1	112.6	136.4
MD5^D	302.7	439.5	424.9	416.7	312.2	332.7	399.6
SHA-1	24.9	61.8	65.9	63.0	25.2	26.5	75.4
SHA-1^D	88.8	248.1	208.2	252.7	96.5	101.1	196.2
SHA-256^G	28.3	74.6	85.5	76.2	29.3	38.4	98.3
SHA-256^D	27.3	114.2	99.7	94.5	27.4	23.7	94.5
SHA-512^G	16.8	106.9	84.1	95.1	18.0	49.6	142.7
SHA-512^D	20.4	117.3	113.4	110.1	20.8	71.5	143.8
RIPEMD-128	145.0	290.9	252.5	255.0	149.7	200.7	256.3
RIPEMD-160	60.8	187.4	175.8	160.4	65.0	103.8	181.4
HAVAL3	169.3	260.1	261.1	265.6	207.1	232.6	271.6
HAVAL4	147.4	201.0	169.7	201.1	151.6	153.9	210.8
HAVAL5	110.2	163.3	128.0	151.4	116.0	154.8	162.3
Tiger	26.9	136.8	114.8	140.6	27.7	82.3	146.3
Whirlpool	6.6	17.6	16.6	12.8	7.8	12.0	17.5

System 7: UltraSPARC III, part 1

	1	2	3	4	5	28	29
MD2	8.0	21.7	27.0	27.2	26.9	31.3	31.4
MD2^D	8.2	24.6	24.5	24.6	24.8	28.0	28.0
MD4	170.0	848.0	952.5	920.4	918.4	1050.2	1044.9
MD4^D	320.5	1296.2	1685.6	1575.4	1631.8	1665.0	1788.6
MD5	140.0	656.4	752.9	699.0	738.0	806.3	822.5
MD5^D	256.5	875.2	1092.3	1058.4	1113.0	1140.9	1201.2
SHA-1	56.8	184.0	258.6	249.3	230.9	301.6	293.8
SHA-1^D	195.4	506.9	703.8	752.9	706.2	793.8	865.9
SHA-256^G	59.2	194.5	258.7	256.8	260.9	288.0	288.4
SHA-256^D	114.9	247.9	325.9	296.4	330.9	378.6	373.7
SHA-512^G	54.3	65.0	116.7	114.7	120.9	91.1	122.1
SHA-512^D	58.4	113.2	98.4	108.0	96.5	69.7	81.8
RIPEMD-128	120.6	550.5	848.0	827.5	814.3	924.6	996.6
RIPEMD-160	86.7	365.4	563.4	538.2	535.4	576.9	613.2
HAVAL3	197.9	747.4	914.3	877.1	862.3	910.2	975.2
HAVAL4	148.6	546.9	687.2	657.5	642.0	681.5	694.2
HAVAL5	129.4	475.2	559.6	572.1	554.3	572.9	616.9
Tiger	87.7	248.2	275.1	247.6	248.2	232.9	238.0
Whirlpool	35.8	47.6	18.9	18.2	19.5	23.6	20.8

System 7: UltraSPARC III, part 2

	18	19	20	21	22
MD2	7.1	23.6	30.5	26.9	30.5
MD2^D	6.6	22.6	28.4	31.8	28.8
MD4	131.0	650.2	765.6	750.2	754.3
MD4^D	303.4	1219.0	1651.6	1545.7	1645.0
MD5	104.4	483.6	638.0	625.3	626.3
MD5^D	237.4	803.1	1044.9	1016.4	1029.1
SHA-1	48.6	161.1	257.6	252.5	184.6
SHA-1^D	141.6	453.6	686.1	700.2	683.8
SHA-256^G	57.1	188.9	281.3	270.2	276.2
SHA-256^D	81.4	244.4	262.1	240.7	264.3
SHA-512^G	88.5	293.6	431.2	410.8	430.7
SHA-512^D	97.7	315.6	465.5	470.8	480.8
RIPEMD-128	114.2	541.8	856.9	829.1	844.5
RIPEMD-160	85.8	356.8	557.3	558.0	547.6
HAVAL3	198.5	750.2	922.5	991.8	866.0
HAVAL4	149.4	546.9	680.4	693.1	644.0
HAVAL5	130.8	473.5	533.3	553.5	528.5
Tiger	143.2	623.4	795.3	769.9	717.3
Whirlpool	35.4	72.2	34.4	33.5	33.7

System 7: UltraSPARC III (Suns compiler)

	1	3	23	24	25	26	27
MD2	14.3	38.1	37.4	38.4	14.5	14.0	31.7
MD2^D	16.6	34.4	36.0	41.2	16.7	13.4	32.7
MD4	751.6	1170.3	1308.6	1248.8	761.3	434.8	1089.4
MD4^D	1018.9	1932.1	1978.7	2027.7	1029.1	904.2	1796.5
MD5	364.4	950.3	1003.9	999.0	364.7	380.7	851.6
MD5^D	757.1	1308.6	1347.4	1343.0	772.8	674.8	1241.2
SHA-1	89.3	211.1	343.0	224.8	89.2	97.3	375.1
SHA-1^D	312.2	804.7	743.4	820.8	310.1	304.1	772.8
SHA-256^G	101.6	297.2	313.4	293.2	102.2	123.6	359.0
SHA-256^D	119.5	353.4	337.1	353.1	119.7	146.4	331.9
SHA-512^G	62.7	410.4	376.5	410.4	62.8	190.3	539.7
SHA-512^D	69.8	412.1	415.8	425.3	70.0	189.1	571.3
RIPEMD-128	353.7	858.7	916.3	946.0	355.2	454.1	937.3
RIPEMD-160	178.4	494.7	497.7	498.3	176.6	264.6	491.1
HAVAL3	550.5	968.3	957.0	984.6	553.5	666.0	957.0
HAVAL4	412.9	711.1	725.0	748.8	412.9	489.4	739.4
HAVAL5	344.5	608.6	618.7	638.0	344.8	424.9	618.7
Tiger	103.2	771.4	789.2	1013.9	103.6	235.0	894.3
Whirlpool	30.9	132.5	146.7	82.3	31.0	40.7	156.5

System 8: SGI MIPS R14000

	1	2	3	4	5
MD2	4.5	18.0	11.8	12.0	11.7
MD2^D	4.3	12.4	12.4	12.4	12.3
MD4	103.9	374.1	381.7	380.3	384.2
MD4^D	184.0	612.2	683.8	682.7	689.6
MD5	74.0	295.1	301.6	301.2	301.4
MD5^D	112.6	424.0	455.6	454.1	456.1
SHA-1	42.4	125.2	101.8	136.6	105.5
SHA-1^D	177.7	352.8	361.5	359.0	361.5
SHA-256^G	45.6	91.3	92.6	93.8	90.6
SHA-256^D	99.6	153.2	143.0	138.8	144.7
SHA-512^G	69.0	149.3	154.3	105.4	148.1
SHA-512^D	83.7	195.5	198.9	199.2	189.5
RIPEMD-128	56.1	245.3	366.7	370.3	364.7
RIPEMD-160	46.3	179.0	235.8	237.9	232.3
HAVAL3	121.0	416.3	412.5	410.4	412.9
HAVAL4	91.9	299.8	300.7	300.1	299.8
HAVAL5	72.1	267.2	262.6	262.6	262.4
Tiger	94.8	273.2	262.9	255.7	253.3
Whirlpool	30.0	61.3	20.7	20.9	20.1

System 9: Alpha

	1	2	3	4	5
MD2	2.3	10.3	11.5	11.6	11.5
MD2^D	2.8	12.1	12.2	12.2	12.1
MD4	90.7	285.4	241.5	275.7	267.5
MD4^D	170.5	772.4	888.9	896.0	893.7
MD5	59.3	244.6	211.1	238.5	229.6
MD5^D	145.6	495.9	511.4	506.6	510.5
SHA-1	48.2	153.6	117.6	117.5	160.1
SHA-1^D	109.3	448.2	530.1	551.0	532.8
SHA-256^G	51.2	173.7	176.7	192.1	180.3
SHA-256^D	80.0	235.0	165.0	204.2	170.3
SHA-512^G	95.1	299.2	322.0	325.4	282.2
SHA-512^D	89.6	314.1	348.2	357.4	326.8
RIPEMD-128	59.3	322.5	548.7	528.9	529.7
RIPEMD-160	43.9	246.2	398.4	393.4	392.6
HAVAL3	169.5	540.2	535.2	530.6	531.0
HAVAL4	114.5	404.9	406.2	409.7	409.2
HAVAL5	87.6	352.6	334.1	336.1	331.8
Tiger	147.3	1077.4	1070.3	1107.0	1213.6
Whirlpool	39.6	98.0	61.3	61.0	56.6
RadioGatun[32]	38.2	87.5	60.8	64.3	61.9
RadioGatun[64]	74.9	175.9	90.2	138.1	115.2

System 10: Power5

	1	2	3	4	5	18	19	20	21	22
MD2	4.0	29.0	26.4	27.1	29.4	5.4	22.0	21.6	21.4	29.3
MD2^D	4.1	30.3	31.0	31.1	29.6	4.2	30.3	31.1	31.0	31.2
MD4	203.0	867.8	829.1	979.9	853.3	151.5	514.6	555.0	556.5	543.2
MD4^D	324.8	1360.8	1500.4	1500.4	1457.7	298.3	941.6	1055.7	1055.7	1044.9
MD5	133.5	616.9	591.9	670.4	605.0	108.4	412.9	437.6	441.4	429.8
MD5^D	258.3	806.3	858.7	858.7	849.8	248.7	656.4	642.0	642.0	666.0
SHA-1	72.3	272.0	317.8	306.4	279.6	47.8	235.9	260.4	278.8	263.9
SHA-1^D	246.9	1003.9	1190.7	1190.7	1147.3	209.0	580.2	636.0	636.0	627.3
SHA-256^G	91.9	378.2	368.7	365.7	390.5	61.7	337.4	339.4	339.4	356.5
SHA-256^D	199.1	537.5	600.6	600.6	592.8	138.1	315.8	370.3	370.7	362.5
SHA-512^G	67.9	158.0	171.1	154.4	187.8	129.8	569.7	576.1	572.9	608.6
SHA-512^D	77.9	184.8	216.7	216.5	212.8	155.9	528.5	558.8	559.6	557.3
RIPEMD-128	121.8	532.6	924.6	970.6	884.7	108.0	377.9	678.1	679.3	691.9
RIPEMD-160	102.9	368.7	654.3	701.4	644.0	94.6	278.6	502.6	502.6	507.6
HAVAL3	371.4	793.8	825.8	825.8	800.0	346.2	643.0	641.0	641.0	619.7
HAVAL4	289.7	608.6	642.0	643.0	633.1	263.6	482.4	484.7	484.2	481.3
HAVAL5	240.7	536.1	545.4	545.4	548.3	221.2	403.1	403.1	403.1	392.3
Tiger	108.5	313.9	367.7	369.0	389.7	130.7	652.2	617.8	617.8	615.9
Whirlpool	45.9	83.3	54.7	52.7	53.6	54.6	184.0	86.0	86.6	91.6

System 11: Itanium 2

	1	2	3	4	5
MD2	4.8	14.1	15.5	17.6	15.5
MD2^D	4.5	13.0	16.0	16.0	15.9
MD4	113.9	348.0	383.2	387.2	382.8
MD4^D	295.8	824.2	1101.2	1092.9	1101.5
MD5	95.3	296.6	322.1	324.9	321.9
MD5^D	223.4	560.7	682.9	680.9	683.0
SHA-1	36.7	115.2	170.0	163.4	159.1
SHA-1^D	120.6	334.0	559.9	551.4	559.9
SHA-256^G	32.0	89.8	196.8	182.8	196.8
SHA-256^D	60.4	142.3	292.3	294.8	293.2
SHA-512^G	59.7	205.2	409.5	394.8	409.3
SHA-512^D	80.7	237.8	466.5	466.6	466.4
RIPEMD-128	104.1	324.9	570.2	570.3	568.7
RIPEMD-160	71.3	209.0	394.7	391.5	394.0
HAVAL3	175.7	430.8	531.2	541.1	542.1
HAVAL4	128.6	315.6	412.5	411.2	414.7
HAVAL5	108.4	269.9	349.1	349.4	350.5
Tiger	119.8	414.7	619.7	623.3	638.6
Whirlpool	22.3	57.3	124.1	129.8	113.8
RadioGatun[32]	16.7	42.6	74.2	79.1	173.8
RadioGatun[64]	38.9	139.9	161.8	161.9	379.9

System 12: Power 4

	1	2	3	4	5
MD2	7.0	21.1	21.3	21.1	23.4
MD2^D	7.5	21.7	21.7	21.6	24.2
MD4	135.9	629.2	605.9	604.1	587.7
MD4^D	267.4	902.2	908.2	906.2	906.2
MD5	97.4	451.1	446.2	426.2	428.0
MD5^D	216.5	549.1	557.3	556.5	554.3
SHA-1	68.1	238.3	303.6	270.9	275.8
SHA-1^D	269.5	550.5	815.9	839.3	855.1
SHA-256^G	88.7	213.9	242.9	245.3	236.9
SHA-256^D	184.6	394.6	385.0	398.4	437.6
SHA-512^G	80.6	156.3	88.7	89.1	85.5
SHA-512^D	88.1	193.0	76.0	74.8	73.9
RIPEMD-128	101.2	349.2	640.0	644.0	609.5
RIPEMD-160	84.4	248.2	464.4	467.0	458.2
HAVAL3	369.0	556.5	622.5	627.2	622.5
HAVAL4	297.0	416.7	461.8	463.9	464.9
HAVAL5	254.7	344.2	382.4	383.2	379.6
Tiger	134.6	356.8	196.0	199.5	237.7
Whirlpool	38.9	64.0	28.9	32.3	31.7

System 12: Power 4 (IBMs compiler)

	32	33	34	35	36	37	38
MD2	4.4	4.4	24.0	22.5	24.3	24.1	23.9
MD2^D	4.8	4.8	24.8	24.2	23.9	23.9	24.8
MD4	282.3	282.3	672.6	754.3	837.6	906.2	672.6
MD4^D	637.0	634.0	898.2	902.2	908.2	979.9	898.2
MD5	233.4	234.3	465.4	501.3	586.0	558.0	465.4
MD5^D	446.2	445.2	557.3	558.8	555.8	540.4	556.5
SHA-1	73.5	73.4	272.7	259.1	343.9	329.8	273.1
SHA-1^D	218.6	218.8	582.6	590.2	781.7	781.7	583.5
SHA-256^G	116.2	116.4	351.3	348.9	353.7	351.9	350.4
SHA-256^D	165.4	165.3	428.4	433.9	440.9	430.7	427.6
SHA-512^G	111.3	111.0	187.2	189.2	217.4	220.3	186.5
SHA-512^D	125.0	125.0	225.3	230.6	232.3	225.8	225.1
RIPEMD-128	237.9	237.2	364.4	362.2	389.3	390.1	365.1
RIPEMD-160	123.3	123.4	254.4	256.2	296.6	297.2	254.4
HAVAL3	399.2	400.0	563.4	576.9	757.1	752.9	562.6
HAVAL4	303.0	303.2	420.1	425.3	595.3	583.5	419.7
HAVAL5	262.7	262.4	349.2	354.3	496.5	491.1	348.9
Tiger	95.2	95.2	384.6	401.2	460.7	457.1	375.4
Whirlpool	51.0	51.1	130.7	128.0	110.0	114.6	131.4

C Collision Examples

Examples of colliding messages for different hash functions are collected in this appendix. Traditionally, the message words of collisions are given, so the notation presented here is big-endian.
To save space, messages with more than 32 words were abbreviated. The first message is shown fully, but for the second message, only the differences are shown.
Every fourth message word is numbered for clarity.

MD2

A collision for the compression function of MD2 [33]:

m	`2ec90abb 41fcd859 ae7e83a8 d02b835b`
m'	`0c7f5f73 82dab197 5f5d7a8c bf588b86`
h	`c3cf8e71 74519cde 8d363fe0 d987078a`

A pseudo-collision for MD2 [34, 71]:

iv	`a614f187 8e643669 b4e3bc59 942b02d1`
iv'	`ccd5ab32 c19abb4b 093bde42 7b072492`
m	`00000000 00000000 00000000 00000000`
h	`80fc01c0 e1f96cb9 44953f3a 1a444f57`

When the initialization vector is either iv or iv', and the message m is hashed, the hash value is h (without the checksum block).

MD4

A collision of MD4, the two messages differ by only one byte, in which four bit are different [36]:

m	0	13985e12 748a810b 4d1df15a 181d1516
	4	2d6e09ac 4b6dbdb9 6464b0c8 fba1c097
	8	abe17be0 ed1ed4b3 4120abf5 20771029
	12	2077102**7** fdfffbff ffffbffb 6774bed2
m'	0	13985e12 748a810b 4d1df15a 181d1516
	4	2d6e09ac 4b6dbdb9 6464b0c8 fba1c097
	8	abe17be0 ed1ed4b3 4120abf5 20771029
	12	2077102**8** fdfffbff ffffbffb 6774bed2
h		711ad51b bbab5e22 618b1c76 17c15892

Another collision, the four-bit difference of the messages is spread across three words [37, 40]:

m	0	4d7a9c83 **5**6cb927a **b**9d5a578 57a7a5ee
	4	de748a3c dcc366b3 b683a020 3b2a5d9f
	8	c69d71b3 f9e99198 d79f805e a63bb2e8
	12	45d**d**8e31 97e31fe5 2794bf08 b9e8c3e9
m'	0	4d7a9c83 **d**6cb927a **2**9d5a578 57a7a5ee
	4	de748a3c dcc366b3 b683a020 3b2a5d9f
	8	c69d71b3 f9e99198 d79f805e a63bb2e8
	12	45d**c**8e31 97e31fe5 2794bf08 b9e8c3e9
h		4d7e6a1d efa93d2d de05b45d 864c429b

A multicollision for MD4, generated with the help of [72]:

m_1	0	c13112f8 **7**ee5aebf **1**48bc8e5 07f94887
	4	e4c12c0f d7a1de87 a432afc3 86f61c51
	8	88e6ff83 978665c7 7519cf76 d9e5aec6
	12	0e2**5**0f87 e240693a 7540f6f8 843e7d18
m_1'	0	c13112f8 **f**ee5aebf **8**48bc8e5 07f94887
	4	e4c12c0f d7a1de87 a432afc3 86f61c51
	8	88e6ff83 978665c7 7519cf76 d9e5aec6
	12	0e2**4**0f87 e240693a 7540f6f8 843e7d18
m_2	0	b81cb046 **7**744fcbf **7**aef5ed1 4eeba6a8
	4	a5749516 6355c25e c0ba38a7 b67135bd
	8	ab9882fa 62daa4ba 1d9d0e58 2902b4cc
	12	e99**9**fb50 a3dc9de8 d15b1859 098db7fd
m_2'	0	b81cb046 **f**744fcbf **e**aef5ed1 4eeba6a8
	4	a5749516 6355c25e c0ba38a7 b67135bd
	8	ab9882fa 62daa4ba 1d9d0e58 2902b4cc
	12	e99**8**fb50 a3dc9de8 d15b1859 098db7fd
h		d09817bb bda086d0 bcb7ab28 20faec2d

The four messages $m_1 \circ m_2$, $m_1' \circ m_2$, $m_1 \circ m_2'$, and $m_1' \circ m_2'$ all have the same hash value h.

A collision for a variant of Extended-MD4, in which the initialization vectors of both lines are set to `3106724a 187c28f6 6db5f180 afdad375` [36]:

m	0	`51737d99 527507ef 69ea5e67 6a7e3c3d`
	4	`8171ebe6 453ef355 0535803b 2c885e93`
	8	`fec3fc24 74fdd294 28566835 0ec55879`
	12	`9a213c15 2069ff64 ffffbffb 2fa86b00`
m'	0	`51737d99 527507ef 69ea5e67 6a7e3c3d`
	4	`8171ebe6 453ef355 0535803b 2c885e93`
	8	`fec3fc24 74fdd294 28566835 0ec55879`
	12	`9a213c16 2069ff64 ffffbffb 2fa86b00`

A preimage for MD4 reduced to the first two rounds [39]:

m	0	`b5b6ac17 b5b6ac17 b5b6ac17 85574a58`
	4	`4353212d 4353212d 4353212d 3e30333e`
	8	`ff9405c3 ff9405c3 ff9405c3 c26ea1d5`
	12	`015cb5d0 81bbd193 1def9763 ade9028b`
	16	`814f4825 814f4825 814f4825 814f4825`
	20	`9919c508 9919c508 9919c508 2fd7b0f9`
	24	`847064ad 05ddd0f5 d462fa71 56a79dec`
	28	`00000080 00000080 000003a0 00000000`
h		`00000000 00000000 00000000 00000000`

The three underlined words represent the data that is appended to the message in accordance to the padding rule of MD4, thus completing the second block.

MD5

The first published collision of MD5 [40, 41]:

m	0	02dd31d1 c4eee6c5 069a3d69 5cf9af98
	4	**8**7b5ca2f ab7e4612 3e580440 897ffbb8
	8	0634ad55 02b3f409 8388e483 5a41**7**125
	12	e8255108 9fc9cdf7 **f**2bd1dd9 5b3c3780
	16	d11d0b96 9c7b41dc f497d8e4 d555655a
	20	**c**79a7335 0cfdebf0 66f12930 8fb109d1
	24	797f2775 eb5cd530 baade822 5c15**c**c79
	28	ddcb74ed 6dd3c55f **d**80a9bb1 e3a7cc35
m'	0	02dd31d1 c4eee6c5 069a3d69 5cf9af98
	4	**0**7b5ca2f ab7e4612 3e580440 897ffbb8
	8	0634ad55 02b3f409 8388e483 5a41**f**125
	12	e8255108 9fc9cdf7 **7**2bd1dd9 5b3c3780
	16	d11d0b96 9c7b41dc f497d8e4 d555655a
	20	**4**79a7335 0cfdebf0 66f12930 8fb109d1
	24	797f2775 eb5cd530 baade822 5c15**4**c79
	28	ddcb74ed 6dd3c55f **5**80a9bb1 e3a7cc35
h		a4c0d35c 95a63a80 5915367d cfe6b751

The messages each have two blocks due to the construction of the collision.

A multicollision for MD5. The messages were found by using [44]:

m_1	0	518fd248 f1bcabf6 c1173e4b c10929b7
	4	e864a401 16d43407 93973a4a 21ff6eb2
	8	0634acd5 84540e06 829ec0d3 1c724f33
	12	1bc18ed2 5d6a64d4 f88b77da 8b19e93b
	16	1bfe0d0b 0b48781d b70a51dd 9378ffda
	20	95f7148c dd43e44c 97b564f2 7f9d284c
	24	9473cfad 464b5ce9 8a5d2952 ddb92489
	28	4111b8c7 00b3a1b1 83323e4b 2870c101
m'_1	0	518fd248 f1bcabf6 c1173e4b c10929b7
	4	6864a401 16d43407 93973a4a 21ff6eb2
	8	0634acd5 84540e06 829ec0d3 1c72cf33
	12	1bc18ed2 5d6a64d4 788b77da 8b19e93b
	16	1bfe0d0b 0b48781d b70a51dd 9378ffda
	20	15f7148c dd43e44c 97b564f2 7f9d284c
	24	9473cfad 464b5ce9 8a5d2952 ddb8a489
	28	4111b8c7 00b3a1b1 03323e4b 2870c101
m_2	0	db2af7aa 38ea285c 0097a42b 818d01dd
	4	7f4ae347 7e8f0705 1e5d82d6 e985b935
	8	0533ed55 8213f209 817cf1ef db18bd35
	12	62bb5ecd 15b921df 60f53bcd 02a3fb05
	16	7f88c3bf 3bff3a9f 88cb9268 a3bacbac
	20	a56fadd8 ed89e30c e8207914 9d6da8cd
	24	eef38049 e62f984f 6a1dc476 3c95b882
	28	2bbaa8cb 406be5ae 88018230 fa778d78
m'_2	0	db2af7aa 38ea285c 0097a42b 818d01dd
	4	ff4ae347 7e8f0705 1e5d82d6 e985b935
	8	0533ed55 8213f209 817cf1ef db1936d35
	12	62bb5ecd 15b921df e0f53bcd 02a3fb05
	16	7f88c3bf 3bff3a9f 88cb9268 a3bacbac
	20	256fadd8 ed89e30c e8207914 9d6da8cd
	24	eef38049 e62f984f 6a1dc476 3c9538882
	28	2bbaa8cb 406be5ae 08018230 fa778d78
h		198754ae 9d230b3c 188d7718 d47275f8

h is the MD5 hash value of all four messages $m_1 \circ m_2$, $m'_1 \circ m_2$, $m_1 \circ m'_2$, and $m'_1 \circ m'_2$.

RIPEMD

A collision for the RIPEMD function [37, 40]:

m	0	579faf8e 09ecf579 574a6aba 78**4**13511
	4	a2b410a4 ad2f6c9f 0b56202c 4d757911
	8	0bdeaae7 78bc91f2 **4**7**bc**6d7d 9abdd1b1
	12	a45d2015 817104ff 264758a8 **6**1064ea5
m'	0	579faf8e 09ecf579 574a6aba 78**5**13511
	4	a2b410a4 ad2f6c9f 0b56202c 4d757911
	8	0bdeaae7 78bc91f2 **c**7**c0**6d7d 9abdd1b1
	12	a45d2015 817104ff 264758a8 **e**1064ea5
h		dd6478dd 9a7d821c aa018648 e5e792e9

HAVAL

A collision for 3-round HAVAL [49]:

m	0	94c0875e dd25f63e f5d09361 b51db8b2
	4	b00c36e4 bad7de19 32a68bb5 c5aff25d
	8	ad0dea24 a7e1ee7c 617b92dd f9da283d
	12	b2844d83 b8d498eb c72fec88 8f467c05
	16	507ea2c1 c2d94121 cb1af394 036daf20
	20	bba7fb8c 6daee6aa 04fc029f d37c05f4
	24	993aea13 3ccfab88 41ab9931 3c7cae0c
	28	f704baf**c** b60635de f0000000 00000000
m'	0	94c0875e dd25f63e f5d09361 b51db8b2
	4	b00c36e4 bad7de19 32a68bb5 c5aff25d
	8	ad0dea24 a7e1ee7c 617b92dd f9da283d
	12	b2844d83 b8d498eb c72fec88 8f467c05
	16	507ea2c1 c2d94121 cb1af394 036daf20
	20	bba7fb8c 6daee6aa 04fc029f d37c05f4
	24	993aea13 3ccfab88 41ab9931 3c7cae0c
	28	f704baf**d** b60635de f0000000 00000000
HAVAL 128		c90611d1 ebe21d10 0c0ca404 c46716f3
HAVAL 160		c954b561 58386c71 3342320b 65e89595 3a8024d8
HAVAL 192		60be9c82 469dcb7f 27339395 2babcd61 18c37c18 5de862d4
HAVAL 224		a7b0d3de bc89a60d 551d832d e9cc506b daaaf584 c89dac4f 2573859a
HAVAL 256		e2381e14 7ccff9e8 1dd7cc16 b51d54b6 b7665e4b ef5f986d 07a903f6 7862477d

A collision for 4-round HAVAL [50]:

m	0	7a6825d3 1cbc99ad b5fa99a6 f3a55ed5
	4	937d8fe2 ba3562be e58f4b87 aefb7823
	8	e0000**4**10 e0000000 000008e0 4020086f
	12	03bfc7f0 7df8806f fffffff1 fdbffff0
	16	**7**dffffff fdfbffef e3ffffef effffbff
	20	ffff9000 f1ffbff0 ffdc0000 ffffc800
	24	ffffc000 00000000 00000000 00200800
	28	ffdfbbef ffffc000 00008010 d075e0b0
	32	1e062c01 efda9c87 90cddbaa ad2dc583
	36	080efe3b d5719eb7 da5bea4a ce7292f5
	40	e000f**e**10 e00ffc00 000006e0 4021066f
	44	03bfbdf0 7df88031 fffffe01 fdbfffe0
	48	**f**dffffe0 fdfbfdef e3ffffff effffbdf
	52	ffff9001 f1ffbff0 ffdc0000 ffffc800
	54	ffffc000 00000000 00000000 00200800
	56	ffdfbbef ffffc000 0000a000 fdab519a
m'	8	e0000**8**10 e0000000 000008e0 4020086f
	16	**f**dffffff fdfbffef e3ffffef effffbff
	40	e000f**a**10 e00ffc00 000006e0 4021066f
	48	**7**dffffe0 fdfbfdef e3ffffff effffbdf
HAVAL 128		8f05ff39 b80c854c 6f6b1177 d9e27b9c
HAVAL 160		04a7195f ff5da070 2217963d 2d65d22f 5c1bb7a9
HAVAL 192		33bc2821 f9aca4de 2b19fc62 2019d070 d5b1c7bc 28e8d2ed
HAVAL 224		dc0f38f6 cdde9d73 58b85e66 aa18a62d a5852c7f 76afc97b c933cada
HAVAL 256		ef760664 9d6afb71 b2443ba8 917e9acd 83f63594 f8f8cf11 52dfb6aa c997b489

SHA-0

A collision for SHA-0 [55]:

m_0	0	5c4fc265 f6890f0c 77de78d4 455225ef
	4	1f3aae83 08e5962a 6a66522c 5aad6f0d
	8	d9909f9d 1e2882eb eb398221 c7fbe134
	12	24d0845c 2f1cadf7 141a1dd4 18dc753b
m_1	16	**bb**0442**4**7 ffa3303b **08**9b7e**f**1 **74**08fa3f
	20	**7**a37266b **01**dcab18 9**3**eb20**d**3 e**9**eb41**b**3
	24	**5**c0f4813 a63a5d**c**a **8**8bdf3**b**9 2**d**1a92**2**1
	28	a**1**fc8540 **5**9e665**e**b 0c57ac**5**1 **e**5aae8**5**4
m_1'	16	**f9**0442**c**7 ffa3303b **4a**9b7e**7**1 **34**08fa3f
	20	**3**a37266b **43**dcab18 9**1**eb20**5**3 e**b**eb41**3**3
	24	**1**c0f4813 a63a5d**4**a **c**8bdf3**3**9 2**f**1a92**a**1
	28	a**3**fc8540 **1**9e665**6**b 0c57ac**d**1 **a**5aae8**d**4
h		66d65a5b 7e8677a9 882ee92f 132cf181
		c8b93803

The messages $m_0 \circ m_1$ and $m_0 \circ m_1'$ collide.

SHA-1

A collision of 64-step SHA-1 with partial meaningful text [58].

m_0	0	65682049	79626572	6c6f7320	6c6e6d65
	4	72702079	73696d6f	6f742065	6e696620
	8	20687369	5020796d	74204468	69736568
	12	79622073	65687420	646e6520	20666f20
m_1	16	35303032	ea0a0a20	9e02d7cb	21982 19f
	20	92fff0f0	f43de413	694aca07	50687306
	24	777c397f	c145dfdd	b00aac52	cf11159d
	28	78dc5fa1	21864d9f	f3411d5d	6a3ca7c2
	32	a1d3b5c2	ee7dbb1e	b57fc2ff	5c53315c
	36	ce3db18f	894b6ac2	60d28 20e	fb31e78c
	40	d9243b38	a9ecfb37	b690e3f5	d51523c1
	44	be8aa4c1	dfc1d39a	c950d5f6	2d57d96b
m'_1	16	36303032	b80a0a60	de02d78b	73982 17f
	20	92fff050	b43de493	684aca27	30687326
	24	767c39ff	8345df9d	f30aac92	ed1115dd
	28	7bdc5fa1	63864ddf	b0411d9d	483ca702
	32	a2d3b5c2	bc7dbb5e	f57fc2bf	0e5331bc
	36	ce3db12f	c94b6a42	61d2822e	9b31e7ac
	40	d8243bb8	ebecfb77	f590e335	f7152381
	44	bd8aa4c1	9dc1d3da	8a50d536	0f57d9ab
h		e9069cca	b770ec16	f9ed4e3a	d6fd5a86
		6f829f0c			

The block m_0 is the ASCII text "I hereby solemnly promise to finish my PhD thesis by the end of ", the first four bytes of m_1 read "2005" whereas the first four bytes of m'_1 read "2006".
The messages $m_0 \circ m_1$ and $m_0 \circ m'_1$ collide under SHA-1 reduced to 64 steps.

A collision for 70-step SHA-1 [3].

m	0	ae3ab33b bbcbae85 1784a857 9ccb3781
	4	2092e94d c7126f5b 48d96b72 b8e9f6e3
	8	99776023 1d2f9b23 946bc7aa 1e9a00e8
	12	71e84dc2 d8307c5b f5590300 31edf990
	16	e2beddab c70aa242 4de015a9 27b06350
	20	9a98df4d f70c02e0 f4c0fd7f a7e0efef
	24	f0c2fb0f bf16dec8 75e6bb81 cb294425
	28	c6a2375f d36319cd b91ccaff 56cb4296
m'	0	de3ab3ab e8cbae35 1f84a867 dfcb3781
	4	5292e99d d7126feb 2ad96b82 fae9f623
	8	a9776023 5f2f9bc3 f46bc78a 5f9a0008
	12	21e84de2 99307c9b 875903e0 32edf930
	16	92bedd3b 940aa2f2 45e01599 64b06350
	20	e898df9d e70c0250 96c0fd8f e5e0ef2f
	24	c0c2fb0f fd16de28 15e6bba1 8a2944c5
	28	96a2377f 926319 0d cb1cca1f 55cb4236
h		151866d5 f7940d84 28e73685 c4d97e18 97da712b

References

[1] David McNett, Distributed.net completes RC5-64 project, September 2002.
`http://www.distributed.net/pressroom/news-20020926.txt`

[2] Bruce Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley & Sons, Inc., 1995.
Probably one of the most referenced books in cryptography, the work by Bruce Schneier gives a detailed look into all relevant topics. Many references, examples, and algorithms are given.

[3] Christophe De Cannière, Florian Mendel, and Christian Rechberger, Collisions for 70-step SHA-1: On the Full Cost of Collision Search, Selected Areas in Cryptography 2007, August 2007, to appear in Lecture Notes in Computer Science.
The authors summarize different methods for SHA collision search and their complexity, and additionally give a collision for 70-step SHA-1. The given complexities of attacks often are too optimistic estimations, they conclude. Therefore, they propose measurement by comparison with a standard implementation of the hash function on the same platform. Their collision search thus has a complexity of 2^{44} compression function equivalents.

[4] Claude Elwood Shannon, A Mathematical Theory of Communication, The Bell System Technical Journal, Volume 27, pp. 379–423, 623–656, October 1948.
`http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf`
In this landmark publication, Claude E. Shannon lays the groundwork of information theory, concentrating on encoding and decoding information to be transmitted over a noisy channel. He first introduces the idea of information entropy as a measure of uncertainty, or information content, similar to that of thermodynamics.

[5] Wikipedia, Digital Signature Algorithm.
`http://en.wikipedia.org/wiki/Digital_Signature_Algorithm`
Cited February 29, 2008.

[6] Mihir Bellare and Phillip Rogaway, Random Oracles are Practical: A Paradigm for Designing Efficient Protocols, Proceedings of the 1st ACM conference on Computer and Communications Security, ACM, pp. 62–73, October 1993.
http://www-cse.ucsd.edu/users/mihir/papers/ro.pdf
In this paper, the substitution of random oracles for hash functions is justified as a tool for cryptographic proofs.

[7] Eric W. Weisstein, Mathworld, One-Way Function.
http://mathworld.wolfram.com/One-WayFunction.html
Cited February 29, 2008.

[8] Ralph Charles Merkle, Secrecy, Authentication and Public Key Systems, Dissertation, Stanford University, June 1979.
http://www.merkle.com/papers/Thesis1979.pdf
In his dissertational thesis, Ralph C. Merkle covers serveral aspects of public key cryptography and digital signatures. He devises a public key cryptosystem using the knapsack problem, and was the first to define properties of cryptographic hash functions.

[9] Magnus Daum and Stefan Lucks, Attacking Hash Functions by Poisoned Messages "The Story of Alice and her Boss", Eurocrypt 2005 Rump Session, June 2005.
http://www.cits.rub.de/MD5Collisions/
Magnus Daum and Stefan Lucks explain a few aspects of the basics of cryptographic hashing and the implications of hash collisions. They present two postscript documents with the same MD5 hash but completely different, meaningful content.

[10] Bart Preneel, Generic Constructions for Iterated Hash Functions, presentation at the Ecrypt PhD Summer School, April 2007.
http://ecrypt-ss07.rhul.ac.uk/Slides/Monday/preneel-samos07.pdf
The slides of Bart Preneels presentation address many different topics in hashing cryptanalysis.

[11] Phillip Rogaway and Thomas Eric Shrimpton, Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance, Lecture Notes in Computer Science, Volume 3017, Fast Software Encryption 2004, pp. 371–388, February 2004.
http://eprint.iacr.org/2004/035.pdf
With seven different definitions for collisions and preimages, the authors formalize the resistances of cryptographic hash functions. They show two kinds of relationships between the resistances, and prove all of them.

[12] Ralph C. Merkle, One-way hash functions and DES. Lecture Notes in Computer Science, Volume 435, Advances in Cryptology – Crypto '89 Proceedings, pp. 428–446, 1989.
Using DES, Ralph C. Merkle constructs three hash functions that base their strength on the security of the block cipher. The security equivalence of DES and the hash functions is proven. The most efficient function of the three is able to hash 18 bits per application of DES.

[13] Ivan B. Damgård, A Design Principle for Hash Functions, Lecture Notes in Computer Science, Volume 435, Advances in Cryptology – Crypto '89 Proceedings, pp. 416–427, 1989.
In his important paper, Ivan B. Damgård concentrates on the mathematical proofs of the basic principles of hash functions, thereby proving that the Merkle-Damgård construction scheme is sound. Additionally, parallel computations are addressed, and three different examples are constructed, including a fast hash function based on the knapsack problem.

[14] John Kelsey and Bruce Schneier, Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work, Lecture Notes in Computer Science, Volume 3494, Advances in Cryptology – Eurocrypt 2005, pp. 474–490, May 2005.
http://eprint.iacr.org/2004/304.pdf
Messages of different chosen lengths with the same hash value can be found by random collision search. The authors use the term "expandable message" for many pairs of such collisions with different lengths. Using these, they are able to produce second preimages with a complexity of about $k \cdot 2^{n/2+1}$ for messages with 2^k blocks.

[15] Antoine Joux, Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions, Lecture Notes in Computer Science, Volume 3152, Advances in Cryptology – Crypto 2004 Proceedings, pp. 306–316, December 2004.
Studying concatenation of iterated hash functions, Antoine Joux proves their security not to be higher than the security of the stronger hash function. This is a result of the possibility of finding many different collisions for the weaker hash function. In addition, he shows the same weak security in respect to preimages.

[16] Wikipedia, One-way compression function.
http://en.wikipedia.org/wiki/One-way_compression_function
Cited February 29, 2008

[17] John R. Black, Phillip Rogaway, and Thomas Eric Shrimpton, Black-Box Analysis of the Block-Cipher-Based Hash-Function Construction from PGV, Lecture Notes in Computer Science, Volume 2442, Advances in Cryptology – Crypto 2002 Proceedings, pp. 103–118, May 2002.
http://www.cs.ucdavis.edu/~rogaway/papers/hash.pdf
The authors give 12 secure schemes to construct a cryptographic hash function from a block cipher, proving the collision resistance of each. An additional 8 schemes are also shown to be slightly less collision resistant.

[18] Ronald L. Rivest, The MD4 Message Digest Algorithm, Lecture Notes in Computer Science, Volume 537, Advances in Cryptology – Crypto '90 Proceedings, pp. 303–311, 1991.
Also: Ronald L. Rivest, The MD4 Message-Digest Algorithm, RFC 1320, April 1992.
http://tools.ietf.org/html/rfc1320
Ronald L. Rivest gives a complete description of MD4, along with design goals, example hashes, and speed samples. An extension to provide for 256-bit hash values is also described. A publicly available implementation in C is referenced, it is appended to the RFC document.

[19] Ronald L. Rivest, The MD5 Message-Digest Algorithm, RFC 1321, April 1992.
http://tools.ietf.org/html/rfc1321
The description for MD5 is given in RFC 1321. A C implementation is part of the document.

[20] Secure Hash Standard, FIPS 180-2, National Institute of Standards and Technology, August 2002.
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf
Also: Donald E. Eastlake, 3rd and Paul E. Jones, US Secure Hash Algorithm 1 (SHA1), RFC 3174, September 2001.
http://tools.ietf.org/html/rfc3174
The full specification of SHA explains all SHA functions in detail. All internally used functions, along with their constants, are given. Furthermore, for each of the five hash functions, two extensive examples are given, in which all changes of the variables are denoted for every step. The RFC document only covers the SHA-1 function, and includes an implementation in C.

[21] RIPEMD, Research and Development in Advanced Communication Technologies in Europe, RIPE Integrity Primitives: Final Report of RACE Integrity Primitives Evaluation (R1040), RACE, June 1992.
Also: RIPEMD, Integrity Primitives for Secure Information Systems, Final Report of RACE Integrity Primitives Evaluation – RIPE-RACE 1040, Lecture Notes in Computer Science, Volume 1007, pp. 69–111, 1995.
The definition of RIPEMD includes an extensive description of the hash function along with security and performance evaluations, and an implementation in C.

[22] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel: RIPEMD-160 – A Strengthened Version of RIPEMD, Lecture Notes on Computer Science, Volume 1039, Fast Software Encryption 1996, pp. 71–82, April 1996.
http://homes.esat.kuleuven.be/~cosicart/pdf/AB-9601/AB-9601.pdf
The paper designing RIPMD-160 first explains the need for a new hash function with increased security. Then, RIPEMD-160 is briefly described, along with other versions of the new RIPEMD hash algorithm. Specific design decisions are explained, and pseudo-code as well as example hashes are given.

[23] Bart Preneel, Analysis and Design of Cryptographic Hash Functions, Dissertation, Katholieke Universiteit Leuven, February 1993.
http://homes.esat.kuleuven.be/~preneel/phd_preneel_feb1993.pdf
In his dissertation, Bart Preneel covers every aspect of cryptographic hash functions in detail. He attends to the efficient and practical point of view rather than the impractical. Construction methods, as well as attack methods are attended to, and many cryptographic hash functions of the time are explained, along with results of cryptanalysis.

[24] Bart Preneel, Design principles for dedicated hash functions, Lecture Notes in Computer Science, Volume 809, Fast Software Encryption 1993, pp. 71–82, 1993.
http://www.cosic.esat.kuleuven.be/publications/article-47.pdf
Bart Preneel summarizes the more general results of hashing cryptanalysis. He reviews many aspects of hash functions, proposing guidelines for the design of new hashes. Additionally, he mentions some points about the implementation efficiency of hash functions, especially about hardware versus software implementation and memory access issues.

[25] Antoon Bosselaers, René Govaerts, and Joos Vandewalle, Fast hashing on the Pentium, Lecture Notes in Computer Science, Volume 1109, Advances in Cryptology – Crypto '96 Proceedings, pp. 298–312, May 1996.
http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9600.pdf
By carefully using many processor-specific optimizations, especially parallel execution capabilities, the authors show how different MD4-based hash functions can be efficiently implemented for the Pentium processor. Compared to implementations in C, the speedup can be more than 100%.

[26] Antoon Bosselaers, Even faster hashing on the Pentium, Eurocrypt 1997 Rump Session, May 1997.
http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9701.pdf
In a follow-up note to [25], Antoon Bosselaers describes another optimization of hashing code, further increasing the speed of the implementation by a factor of around 1.15.

[27] Junko Nakajima and Mitsuru Matsui, Performance Analysis and Parallel Implementation of Dedicated Hash Functions, Lecture Notes in Computer Science, Volume 2332, Advances in Cryptology – Eurocrypt 2002 Proceedings, pp. 165–180, May 2002.
With parallelization and pipelining, Junko Nakajima and Mitsuru Matsui optimize hash function execution on the Pentium 3 processor. They implement different levels of parallelization using MMX registers and instructions, hashing up to three blocks at once for MD5, RIPEMD and SHA hash functions. Because of their 64-bit operations and data, such parallelization could not be realized for SHA-512 and Whirlpool.

[28] Ross Anderson and Eli Biham, Tiger: A Fast New Cryptographic Hash Function, Lecture Notes in Computer Science, Volume 1039, Fast Software Encryption 1996, pp. 89–97, February 1996.
http://www.cs.technion.ac.il/~biham/Reports/Tiger
The authors explain the design requirements, and the specification of the Tiger hash function, also briefly showing some security aspects. Of course, a reference implementation is also given.

[29] Donald W. Davies and David O. Clayden, A Message Authenticator Algorithm Suitable for a Main Frame Computer, NPL Report DITC 17/83, February 1983
http://www.compulink.co.uk/~klockstone/maa.pdf
The MAA, which includes a keyed hash function, was described in 1983 by Donald W. Davies and David O. Clayden. It is dedicated to authenticating financial messages by using a secret, pre-shared key with a size of 64 bits.

[30] Bart Preneel, Vincent Rijmen, and Paul C. van Oorschot, Security Analysis of the Message Authenticator Algorithm (MAA), European Transactions on Telecommunications, Volume 8, No. 5, pp. 455–470, April 1997.
http://www.scs.carleton.ca/~paulv/papers/MAA-ETT.pdf
In their extensive work which was presented at the 1996 Eurocrypt conference, the authors describe the first computationally feasible attacks on MAA. They were able to conduct both MAC (authentication) forgery and key recovery.

[31] Paulo S.L.M. Barreto, The Hash Function Lounge, January 2007.
http://paginas.terra.com.br/informatica/paulobarreto/hflounge.html

[32] B. Kaliski, The MD2 Message-Digest Algorithm, RFC 1319, April 1992.
`http://tools.ietf.org/html/rfc1319`
In RFC 1319, MD2, the first of the three MD hash functions is described, including an example implementation in C.

[33] N. Rogier and Pascal Chauvaud, MD2 is not Secure Without the Checksum Byte, Designs, Codes and Cryptography, Volume 12, Number 3, pp. 245–251, November 1997.
Originally found in 1995, the authors present an attack on the compression function of MD2. However, there are restrictions on the 16-byte intermediate hash for the attack to work. The complexity of $2^{8 \cdot (17-z)}$ depends on the number of trailing zero bytes z. The intermediate hash value is initialized to zero bytes, thus finding colliding first blocks has a complexity of 2^8.

[34] Lars R. Knudsen and John E. Mathiassen, Preimage and Collision Attacks on MD2, Lecture Notes in Computer Science, Volume 3557, Fast Software Encryption 2005, pp. 255–267, 2005.
With a preimage attack on MD2 with a complexity of 2^{97}, Lars R. Knudsen and John E. Mathiassen improve an attack published earlier. They are able to generate preimages of variable lengths, finding many different preimages, as well as pseudo-collisions for MD2, which can be computed with a complexity of 2^{16}.

[35] Bert den Boer and Antoon Bosselaers, An Attack on the Last Two Rounds of MD4, Lecture Notes in Computer Science, Volume 576, Advances in Cryptology – Crypto '91 Proceedings, pp. 194–203, October 1991.
`http://homes.esat.kuleuven.be/~cosicart/pdf/AB-9100.pdf`
When omitting the first of three rounds of MD4, collisions can be found for the hash function, as Bert den Boer and Antoon Bosselaers show. They can be calculated very fast. Additionally, it is mentioned that Ralph C. Merkle achieved similar results when leaving out the last round of MD4.

[36] Hans Dobbertin, Cryptanalysis of MD4, Lecture Notes in Computer Science, Volume 1039, Fast Software Encryption 1996, pp. 53–69, February 1996.
http://www.epanastasi.com/texts/md4dobbertin.pdf
The first collision of MD4 was presented by Hans Dobbertin, applying the same methods he used to attack RIPEMD. The two colliding blocks differ by four bits, and take 2^{20} invocations of the compression function of MD4 to compute. Furthermore, a collision for a modified version of Extended-MD4, where the initialization vectors of both lines are changed, is given.

[37] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu, Cryptanalysis of the Hash Functions MD4 and RIPEMD, Lecture Notes in Computer Science, Volume 3494, Advances in Cryptology – Eurocrypt 2005 Proceedings, pp. 1–18, May 2005.
http://www.infosec.sdu.edu.cn/paper/md4-ripemd-attck.pdf
Along with some new analytical techniques, Xiaoyun Wang and her colleagues present attacks at MD4 and RIPEMD. For MD4, collisions can be found with a probability of more than 2^{-6} within 2^8 applications of MD4. The attack on RIPEMD yields a collision with a probability of 2^{-16} within 2^{18} RIPEMD computations. Another attack on MD4 computes second preimages, but is only applicable to very rare weak messages.

[38] Yusuke Naito, Yu Sasaki, Noboru Kunihiro, and Kazuo Ohta, Improved Collision Attack on MD4, Cryptology ePrint Archive, Report 2005/151, May 2005.
http://eprint.iacr.org/2005/151.pdf
The authors improve the results of [37] to find collisions even faster. The new attack finds collisions with a higher probability than 2^{-2} within three calculations of MD4.

[39] Hans Dobbertin, The First Two Rounds of MD4 are Not One-Way, Lecture Notes in Computer Science, Volume 1372, Fast Software Encryption 1998, pp. 284–292, March 1997.
http://www-cse.ucsd.edu/users/bsy/dobbertin-md4.ps
Hans Dobbertin uses new methods for finding collisions to construct preimages. With his attack on the first two rounds on MD4, preimages can be found in less than an hour, second preimages in minutes. A preimage is given for 0^{128}.
Unfortunately, no comprehensive explanation of the attack was ever published.

[40] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu, Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, Cryptology ePrint Archive, Report 2004/199, Crypto 2004 Rump Session, August 2004.
http://eprint.iacr.org/2004/199.pdf
The chinese researchers list, without any explanation, collisions for MD5, HAVAL-128, MD4, and RIPEMD. Finding collisions for MD5 took about one hour, collisions for MD4 can be found with "hand calculation". Collisions of HAVAL-128 can be found with 2^6 hash computations. In an additional remark, they state that collisions of SHA-0 can be found with 2^{40} computations, and collisions of HAVAL-160 can be found with a probability of 2^{-32}.

[41] Xiaoyun Wang and Hongbo Yu, How to Break MD5 and Other Hash Functions, Lecture Notes in Computer Science, Volume 3494, Advances in Cryptology – Eurocrypt 2005 Proceedings, pp. 19–35, May 2005.
http://www.infosec.sdu.edu.cn/paper/md5-attack.pdf
With a new measure of difference of a differential attack, Xiaoyun Wang and Hongbo Yu construct an algorithm that finds two-block collisions. Finding the first block has a complexity of 2^{39} MD5 executions, the second block takes only 2^{32} MD5 operations. Similar and fast attacks on HAVAL-128, RIPEMD, SHA-0 and MD4 are also mentioned.

[42] Vlastimil Klima, Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications, Cryptology ePrint Archive: Report 2005/102, March 2005.
http://eprint.iacr.org/2005/102.pdf
From work based on the collisions given by [40], Vlastimil Klima constructs fast techniques to find similar collisions of MD5. With effective message modification methods he is able to find colliding messages with a computational complexity of 2^{33} and 2^{29} for the two blocks.

[43] Vlastimil Klima, Tunnels in Hash Functions: MD5 Collisions Within a Minute, Cryptology ePrint Archive, Report 2006/105, April 2006.
http://eprint.iacr.org/2006/105.pdf
Introducing the concept of tunnels, Vlastimil Klima is able to increase the search for collisions in MD5 exponentially. The complexity of the search is not given, but it is stated that the calculations of generating a collision take less than a minute on a standard PC.

[44] Vlastimil Klima, Tunnels in Hash Functions: MD5 Collisions Within a Minute, source code, March 2006.
`http://cryptography.hyperlink.cz/2006/web_version_1.zip`
In addition to [43] Vlastimil Klima demonstrates his attack by giving a program for finding MD5 collisions. Its full source code is included.

[45] Max Gebhardt, Gerog Illies, and Werner Schindler, A Note on the Practical Value of Single Hash Collisions for Special File Formats, presented at the 2005 NIST Cryptographic Hash Workshop, October 2005.
`http://csrc.nist.gov/pki/HashWorkshop/2005/Oct31_Presentations/Illies_NIST_05.pdf`
In their presentation and paper, the authors show how the ability to calculate collisions for any initialization vector leads to colliding files with meaningful content. Max Gebhardt, Gerog Illies, and Werner Schindler were able to construct pairs of `PDF` files, `TIFF` images, as well as Word97 documents that display different content, but have the same MD5 hash.

[46] Marc Stevens, Arjen Lenstra, and Benne de Weger, Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities Lecture Notes in Computer Science, Volume 4515, Advances in Cryptology – Eurocrypt 2007, pp. 1–22, May 2007.
`http://www.win.tue.nl/hashclash/EC07v2.0.pdf`
Marc Stevens, Arjen Lenstra, and Benne de Weger present their construction of two valid X.509 certificates with identical MD5 hashes with the help of distributed computing. They improve some methods as well as use several previously found methods to speed up collision generation. The attack has a complexity of about 2^{50} calls to the MD5 function. Additionally, several potential applications of chosen-prefix collisions are discussed.

[47] Hans Dobbertin, RIPEMD with Two-Round Compress Function is Not Collision-Free, Journal of Cryptology 1997, Volume 10, Number 1, pp. 51–69, 1997.
First shown in 1995, Hans Dobbertin proves that collisions can be found for RIPEMD reduced to either the first two or the last two rounds. He uses a three-step method to find collisions. About 2^{31} applications of the reduced compression function are necessary to compute a collision, taking about one day on a 486 with 66 MHz, standard hardware at the time.

[48] Yuliang Zheng, Josef Pieprzyk, and Jennifer Seberry, HAVAL – A One-Way Hashing Algorithm with Variable Length of Output, Lecture Notes in Computer Science, Volume 718, Advances in Cryptology – Auscrypt '92, pp. 83–104, December 1993.
`http://labs.calyptix.com/files/haval-paper.pdf`
With newly discovered non-linear boolean functions, three researchers from the australian university of Wollongong propose HAVAL, a hash function with 15 different levels of security (3-5 passes and 5 different lengths of output). The full specification of the hash function is followed by a design rationale and security considerations, possible extensions are also discussed.

[49] Bart Van Rompay, Alex Biryukov, Bart Preneel, and Joos Vandewalle, Cryptanalysis of 3-Pass HAVAL, Lecture Notes in Computer Science, Volume 1894, Advances in Cryptology – Asiacrypt 2003, pp. 228–245, December 2003.
The authors show how to construct collisions for HAVAL with three passes and any length. The attack has a complexity of 2^{29} calls to HAVALs compression function.

[50] Zhangyi Wang, Huanguo Zhang, Zhongping Qin, and Qingshu Meng, Cryptanalysis of 4-Pass HAVAL, Cryptology ePrint Archive, Report 2006/161, April 2006.
`http://eprint.iacr.org/2006/161.pdf`
A two-block collision computation for 4-pass HAVAL is presented by the authors. The complexity is about 2^{32} for the first, and 2^{29} for the second block.

[51] Hongbo Yu, Xiaoyun Wang, Aaram Yun, and Sangwoo Park, Cryptanalysis of the Full HAVAL with 4 and 5 Passes, Lecture Notes in Computer Science, Volume 4047, Fast Software Encryption 2006, pp. 89–110, March 2006.
The chinese cryptanalysts around Hongbo Yu and Xiaoyun Wang develop two different two-block collision attacks on 4-pass HAVAL, along with a theoretical attack on the 5-pass version with probability 2^{-123}. The attacks on 4-pass HAVAL find messages differing in one message word (2^{43} computations of HAVAL), and in two message words (2^{36} computations).

[52] Florent Chabaud and Antoine Joux, Differential Collisions in SHA-0, Lecture Notes in Computer Science, Volume 1462, Advances in Cryptology – Crypto '98 Proceedings, pp. 56–71, August 1998.
`http://fchabaud.free.fr/English/Publications/sha.ps`
Florent Chabaud and Antoine Joux study variants of SHA-0 that have been weakened by replacing the non-linear round functions and addition with the linear xor. With this they are able to find a collision of 35-round SHA-0, their attack on the full version of the hash function has a complexity of 2^{61}. Their method is inefficient when applied to SHA-1.

[53] Eli Biham and Rafi Chen, Near-Collisions of SHA-0, Lecture Notes in Computer Science, Volume 3152, Advances in Cryptology – Crypto 2004 Proceedings, pp. 290–305, August 2004.
`http://eprint.iacr.org/2004/146.ps`
Eli Biham and Rafi Chen considerably improve a previous attack on SHA-0 [52]. They are able to generate near collisions of SHA-0, where only 18 bits of the hash values of two messages differ. Additionally, they present collisions for SHA-0 reduced to 65-rounds, and state the change in security for a changed number of rounds.

[54] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby, Collisions of SHA-0 and Reduced SHA-1, Lecture Notes in Computer Science, Volume 3494, Advances in Cryptology – Eurocrypt 2005, pp. 36–57, May 2005.
Using previous work and several new techniques, the authors publish the first results on SHA-1, including collisions for reduced versions. In addition to a four-block collision for the full SHA-0 with a complexity of roughly 2^{51} hash function applications, colliding two-block messages for SHA-1 with up to 40 rounds are presented. The calculations of the SHA-0 collision were done with optimized code on a supercomputer, and took about 80 000 hours of CPU time, according to [55].

[55] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin, Efficient Collision Search Attacks on SHA-0, Lecture Notes in Computer Science, Volume 3621, Advances in Cryptology – Crypto 2005 Proceedings, pp. 1–16, August 2005.
http://www.infosec.sdu.edu.cn/paper/sha0-crypto-author-new.pdf
Along with new methods, the authors improve previous techniques to find collisions of SHA-0 within 2^{39} hash calculations and near collisions with 2^{33} hash calculations. The colliding one-block message pairs are preceded by a block to obtain certain properties of the intermediate hash value. It is the same for both messages.

[56] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, Finding Collisions in the Full SHA-1, Lecture Notes in Computer Science, Volume 3621, Advances in Cryptology – Crypto 2005 Proceedings, pp. 17–36, August 2005.
http://evan.stasis.org/odds/sha1-crypto-auth-new-2-yao.pdf
By combining many different results of the analysis of several hash functions, Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu are able to present an attack on SHA-1 with a complexity of 2^{69} calls to the hash function. Collisions for 58-round SHA-1 can be generated with a complexity of 2^{33} hash calculations.

[57] Xiaoyun Wang, Andrew Yao, and Frances Yao, communicated by Adi Shamir, New Collision Search for SHA-1, Crypto 2005 Rump Session, August 2005.
http://www.iacr.org/conferences/crypto2005/r/2.pdf
In the short presentation by Adi Shamir about results of Xiaoyun Wang, Andrew Yao, and Frances Yao, a new collision path is announced. It lowers the complexity of finding collisions for SHA-1 to 2^{63}.
Unfortunately, little more information is given in the announcement, and no supporting paper has been published so far.

[58] Christophe De Cannière and Christian Rechberger, SHA-1 Collisions: Partial Meaningful at No Extra Cost?, Crypto 2006 Rump Session, August 2006.
http://www.iaik.tugraz.at/aboutus/people/rechberger/talks/SHA1CollisionsMeaningful.pdf
While not giving any details, Christophe De Cannière and Christian Rechberger show a meaningful collision of 64-round SHA-1. Their method consists of finding collision paths based on the conditions determined by the pre-set message part. The two two-block messages have seven bytes of meaningful text at the beginning, two of them differ by one bit each. No paper or further information was made public yet.

[59] Henri Gilbert and Helena Handschuh, Security Analysis of SHA-256 and Sisters, Lecture Notes in Computer Science, Volume 3006, Selected Areas in Cryptography 2003, pp. 175–193, May 2004.
Henri Gilbert and Helena Handschuh apply several cryptanalytic attacks to the SHA-2 hash functions. They conclude that the techniques yield no usable results. In addition they demonstrate that a symmetric variant of SHA-2 (symmetric constants and xor instead of addition) hashes symmetric input to symmetric hashes.

[60] Krystian Matusiewicz, Josef Pieprzyk, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen, Analysis of simplified variants of SHA-256. Western European Workshop on Research in Cryptology (WEWoRC) 2005, LNI P-74, pp. 123–134, 2005.
http://www.iaik.tu-graz.ac.at/aboutus/people/rechberger/written/AnalysisofsimplifiedVariantsofSHA256.pdf
Taking a closer look at the Σ and σ functions of SHA-256, the authors identify them to be necessary for the security of the hash function. For a SHA-256 variant without the functions, a collision can be found in 2^{64} hash function applications.

[61] John Kelsey and Stefan Lucks, Collisions and Near-Collisions for Reduced-Round Tiger, Lecture Notes in Computer Science, Volume 4047, Fast Software Encryption 2006, pp. 111–125, March 2006.
http://th.informatik.uni-mannheim.de/People/Lucks/papers/Tiger_FSE_v10.pdf
John Kelsey and Stefan Lucks apply the cryptanalytic techniques of breaking MD4 and its descendants to the Tiger hash function. Collisions for 16-round Tiger can be found within 2^{44} hash function applications with their attack. Pseudo-near-collisions of 20-round Tiger take 2^{48} hash function calls and differ in six bits.

[62] Paulo S.L.M. Barreto and Vincent Rijmen, The Whirlpool Hash Function, First open NESSIE Workshop, November 2000.
http://paginas.terra.com.br/informatica/paulobarreto/whirlpool.zip
The official publication of the Whirlpool hash function includes a mathematical description of each step, proposed goals, and a first cryptanalysis of the hash function. Additionally, a design rationale for all aspects, especially for the substitution box, is included, as well as several different optimization suggestions, and full source code in C.

[63] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, RadioGatún, a belt-and-mill hash function, Second Cryptographic Hash Workshop of NIST, August 2006
http://eprint.iacr.org/2006/369.pdf
The authors describe a RadioGatún, a cryptographic hash function that does not follow the Merkle-Damgård principle. It is based on the Panama hash. The hash function has a large internal state that is divided into two parts, the belt and the mill. Results of the cryptanalysis of the hash are also included.

[64] Lawrence Spracklen, UltraSPARC T2 Crypto performance, August 2007.
http://blogs.sun.com/sprack/entry/ultrasparc_t2_crypto_performance

[65] Helion Technology, SHA-1 hashing cores.
http://www.heliontech.com/sha1.htm

[66] Olivier Gay, A fast software implementation in C of the FIPS 180-2 hash algorithms SHA-224, SHA-256, SHA-384 and SHA-512, 2007.
http://www.ouah.org/ogay/sha2/

[67] Christophe Devine, The XySSL project, a quality, open-source cryptographic library written in C and targeted at embedded systems, 2007.
http://xyssl.org

[68] Free Software Foundation, The GNU C Library, Processor And CPU Time, CPU Time Inquiry, August 2007.
http://www.gnu.org/software/libc/manual/html_node/CPU-Time.html

[69] National Institute of Standards and Technology, Announcing the Development of New Hash Algorithm(s) for the Revision of Federal Information Processing Standard (FIPS) 1802, Secure Hash Standard, Federal Register, Volume 72 No. 14, pp. 2861–2863 January 2007.
http://edocket.access.gpo.gov/2007/pdf/E7-927.pdf

[70] Paul C. van Oorschot and Michael J. Wiener, Parallel Collision Search with Cryptanalytic Applications, Journal of Cryptology 1999, Volume 12, Number 1, pp. 1–28, January 1999.
http://www.scs.carleton.ca/~paulv/papers/JoC97.pdf
With distinguished points and Pollards rho method, the authors present a method for effectively parallelizing collision search, reducing the memory needed for such an attack.

[71] Søren Steffen Thomsen, Personal Communication, August 2007.

[72] Patrick Stach, MD5 and MD4 Collision Generators, 2007.
http://www.stachliu.com/research_collisions.html

Digital Data

There are several files attached to this document in digital form:

- source code of the speed test suite
- the colliding messages given in appendix C
- all available publications from the bibliography