Hash functions: Theory, attacks, and applications

Ilya Mironov
Microsoft Research, Silicon Valley Campus
mironov@microsoft.com

October 24, 2005

1 Introduction

Hash functions, most notably MD5 and SHA-1, initially crafted for use in a handful of cryptographic schemes with specific security requirements, have become standard fare for many developers and protocol designers who treat them as black boxes with magic properties. This practice had been defensible until 2004—both functions appeared to have withstood the test of time and intense scrutiny of cryptanalysts. Starting last year, we have seen an explosive growth in the number and power of attacks on the standard hash functions. In this note we discuss the extent to which the hash functions can be thought of as black boxes, review some recent attacks, and, most importantly, revisit common applications of hash functions in programming practice.

The note assumes no previous background in cryptography. Parts of the text intended for the more mathematically-inclined readers are marked with $\oint$— the sign of the contour integral—and typeset in a smaller font.

2 Theory of hash functions

In this section we introduce notation, define security properties of hash functions, describe basic design principles of modern hash functions and generic attacks.

2.1 Notation

The following notation used in this note is standard in the cryptographic literature:

$\{0,1\}^n$—the set of all binary strings of length n.
$\{0,1\}^*$—the set of all finite binary strings.
$A \times B$—the set of all pairs (v, w), where $v \in A$ and $w \in B$.

$H\colon A \mapsto B$—function H from set A to set B.
$|w|$—the length of string w.
$w||v$—concatenation of strings w and v.
$w \oplus v$—bitwise XOR of binary strings w and v, which are presumed to be of equal length.

For example, $H\colon \{0,1\}^* \times \{0,1\}^\ell \mapsto \{0,1\}^n$ means a function H of two arguments, the first of which is a binary string of arbitrary length, the second is a binary string of length ℓ, returning a binary string of length n.

2.2 Definitions

The disconnect between theory and practice of cryptographic hash functions starts right in the beginning—in the very definition of hash functions. In practice, the hash function (sometimes called the message digest) is a fixed function that maps arbitrary strings into binary strings of fixed length. In theory, we usually consider *keyed* hash functions, as in the following definition:

Definition Let ℓ, n be positive integers. We call f a *hash function* with n-bit output and ℓ-bit key if f is a deterministic function that takes two inputs, the first of arbitrary length, the second of length ℓ bits, and outputs a binary string of length n. Formally, $H\colon \{0,1\}^* \times \{0,1\}^\ell \mapsto \{0,1\}^n$.

In this note we write the key as a subscript: $H_k(x)$ is an expressive shorthand for $H(x,k)$. The key k is assumed to be known unless indicated otherwise (to avoid confusion with cryptographic keys, which typically represent closely guarded secrets, the hash function's key is sometimes called the "index").

We distinguish between three levels of security that hash functions may satisfy to be useful in cryptographic applications. The definitions are framed as games that are infeasible to win by a computationally-bounded adversary. The words "infeasible" and "computationally-bounded" can be formalized in several ways, neither of which we find entirely satisfying for the purpose of this note. Instead, we offer a semi-formal semantic where we say that the problem is computationally infeasible if no program performing less than T elementary operations can solve the problem with probability higher than $T/2^{80}$. It corresponds to the so-called 80-bit security level, which is currently acceptable for most applications. For further discussion on the appropriate level of security we refer the reader to [LV01].

$\oint$ We stress that both the game and the adversary are randomized. The probability of the adversary's success in winning the game is taken over all random choices, including the key of the keyed hash function and the adversary's own coin tosses. Usually we assume the uniform distribution on the input, which is impossible to define when x is an arbitrary finite binary string. In those case it is convenient to break the set of inputs into finite subsets, such as strings of the same length.

Definition Hash function H is *one-way* if, for random key k and an n-bit string w, it is hard for the attacker presented with k, w to find x so that $H_k(x) = w$.

Definition Hash function H is *second-preimage resistant* if it is hard for the attacker presented with a random key k and random string x to find $y \neq x$ so that $H_k(x) = H_k(y)$.

Definition Hash function H is *collision resistant* if it is hard for the attacker presented with a random key k to find x and $y \neq x$ so that $H_k(x) = H_k(y)$.

The last definition is notably harder to formalize for keyless hash functions, which explains why theorists prefer keyed hash functions.

It is easy to see that collision resistance implies second-preimage resistance. Strictly speaking, second-preimage resistance and one-wayness are incomparable (neither property follows from the other), although construction which are one-way but not second-preimage resistant are quite contrived. In practice, collision resistance is the strongest property of all three, hardest to satisfy and easiest to breach, and breaking it is the goal of most attacks on hash functions.

Certificational weakness. Intuitively, a good hash function must satisfy other properties not implied by one-wayness or even collision-resistance. For example, one would expect that flipping a bit of the input would change approximately half the bits of the output (avalanche property) or that no inputs bits can be reliably guessed based on the hash function's output (local one-wayness). Failure to satisfy those or similar properties, such as infeasibility of finding a pseudo-collision, a free-start collision, and a near-collision whose definitions are given in Section 5, is called a *certificational weakness.*

Presence of certificational weaknesses does not amount to a break of a hash function but is enough to cast doubt on its design principles.

2.3 Generic attacks

A generic attack is an attack that applies to all hash functions, no matter how good they are, as opposed to specific attacks that exploit flaws of a particular design. The running time of generic attacks is measured in the number of calls to the hash function, which is treated as a black box.

It is not difficult to see that in the black box model the best strategy for inverting the hash function and for finding a second preimage is the exhaustive search. Suppose the problem is to invert H_k, i.e., given w, k find x, so that $H_k(x) = w$, where k is ℓ-bit key and w is an n-bit string. The only strategy which is guaranteed to work for any hash function is to probe arbitrary chosen strings until a preimage of w is hit. For a random function H it would take on average 2^{n-1} evaluations of H. In the black-box model the problem of finding a second preimage is just as hard as inverting the hash function.

Finding collisions is a different story, the one that goes under the name of the "birthday

paradox." The chances that among 23 randomly chosen people there are two who share the same birthday are almost 51%. The better than even odds appear to be much higher than the intuition would suggest, hence the name. Of course, there is nothing paradoxical (as in being absurd or self-contradictory) about the fact—between 23 people there are 253 pairs, each of which has a one-to-365 odds of being a hit. The reason why the result appears counter-intuitive is because *your* chances of finding among 22 people somebody with the same birthday *as yours* (not just someone else's) to split the cost of the birthday party are strongly against you. This explains why inverting a hash function is much more difficult than finding a collision.

More careful analysis of the birthday paradox shows that in order to attain probability better than $1/2$ of finding a collision in a hash function with n-bit output, it suffices to evaluate the function on approximately $1.2 \cdot 2^{n/2}$ randomly chosen inputs (notice that $\lceil 1.2 \cdot \sqrt{365} \rceil = 23$).

The running times of generic attacks on different properties of hash functions provide upper bounds on security of any hash function. We say that a hash function has *ideal security* if the best attacks known against it are generic. Cryptanalysts consider a primitive broken if its security is shown to be less than ideal, although it may still be sufficient for some applications. The generic attacks are summarized in Table 1.

Property	**Ideal security**
One-wayness	2^{n-1}
Second preimage-resistance	2^{n-1}
Collision-resistance	$1.2 \cdot 2^{n/2}$

Table 1: Complexity of generic attacks on different properties of hash functions.

$\oint$ A naïve implementation of the birthday attack would store $2^{n/2}$ previously computed elements in a data structure supporting quick stores and look-ups. However, there is profound imbalance between the cost of storage and computation. While 2^{40} CPU cycles are dirty cheap, capacity of high-end hard-drives is still below 2^{40} bytes=1TB, and having commercially available computers with 1TB RAM is still several years away. It is therefore important to note that a birthday attack can be run essentially memoryless (using Floyd's cycle-finding algorithm) with only a modest increase in the number of evaluations of the hash function.

$\oint$ Vice versa, it is sometimes convenient to push some of the computation to the pre-processing stage, storing the results of the off-line computation to facilitate on-line processing. Such rebalancing acts are called time-memory tradeoffs. They are often expressed as equations in two variables: T—time of on-line processing, M—storage requirement. For instance, Hellman's time-memory tradeoff for inverting hash functions is $M^2T = N^2$, where $N = 2^n$ is the size of the hash function's image. Its sweet spot is $M = T = 2^{2/3n}$, although the preprocessing stage takes time proportional to 2^n.

2.4 Constructions

Provably secure constructions of cryptographic hash functions consist of two ingredients, which may be studied independently of each other. The first component is a *compression function* that maps a fixed-length input to a fixed-length output. The second component of a construction is a *domain extender* that, given a compression function, produces a function with arbitrary-length input.

Compression function. From the theorist's point of view, a one-way function is the most basic primitive, from which many other cryptographic tools can be designed. A seminal result due to Simon [Sim98] provides strong evidence that collision-resistant hash functions cannot be constructed based on one-way functions. Instead, we derive collision-resistant hash functions from another cryptographic primitive—a block cipher.

A block cipher is a keyed permutation $E\colon \{0,1\}^n \times \{0,1\}^k \mapsto \{0,1\}^n$. Technically, a block cipher already compresses its input—it maps $k+n$ bits to n bits. As is, however, the block cipher is not even one-way: to invert E on w, fix any key k_0 and decrypt w under this key. If w decrypts to x, then $E(k_0, x) = w$. Nonetheless, as many as 12 simple constructions based on a block cipher result in a collision-resistant compression function (although there are many more that do not, see [BRS02]). Two schemes most often used in hash functions are the following:

Davies-Meyer:	$H(x,y) = E_x(y) \oplus y$
Miyaguchi-Preneel:	$H(x,y) = E_x(y) \oplus x \oplus y.$

Proofs of security of these and similar block cipher-based constructions assume that the underlying cipher is indistinguishable from a certain abstraction, called the ideal cipher, which goes beyond the standard security requirements for block ciphers.

Domain extender. The domain extender is a generic construction that transforms a compression function with fixed-length input into a hash function with arbitrary input. The simplest and most commonly used domain extender is called the Merkle-Damgård construction and it works as follows:

Given: compression function $C\colon \{0,1\}^n \times \{0,1\}^m \mapsto \{0,1\}^n$;
n-bit constant IV.
Input: message M
1. Break M into m-bit blocks $M_1,\ldots,M_k$, padding if necessary;
2. Let M_{k+1} be encoding of $|M|$;
3. Let $h_0 = \text{IV}$;
4. For $i = 1$ to $k+1$ let $h_i = C(h_{i-1}, M_i)$;
5. Output h_{k+1}.

The construction works by iterating the compression function C. The output of the compression function, together with the next block of the message, becomes the input to the next application of the compression function. The hash of the last block, which contains

an encoding of the length of the message, is the hash of the entire message. The temporary storage of the compression function's output, h_i, is called the chaining variable or the internal state (see Figure 1).

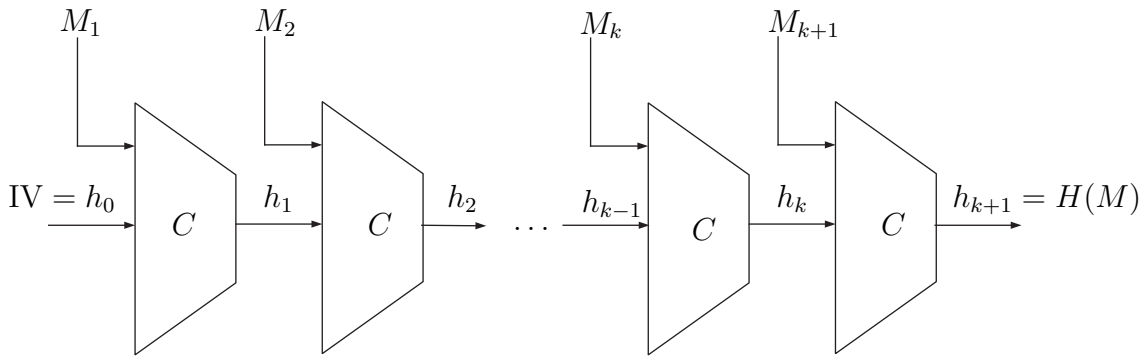

Figure 1: Merkle-Damgård construction.

There is a certain flexibility in the first two steps of the Merkle-Damgård construction. Any encoding will do as long as it satisfies the following three conditions:

- M is encoded as an integral number of m-bit blocks;
- the encoding is collision-free;
- the length of M is encapsulated in the last block.

The Merkle-Damgård construction is compatible with streaming APIs, where a message is fed one block at a time into a cryptographic search engine. Its length need not be known until the last block becomes available. On the other hand, updating even one bit of the message may trigger recomputation of the entire hash.

If the compression function is collision-resistant, so is the resulting construction. However, the Merkle-Damgård construction produces a function with many structural properties, creating a number of unexpected vulnerabilities as illustrated in Section 4.

In fact, the Merkle-Damgård construction is the single most important reason why it is **wrong** (**dangerous**, **reckless**, **ignorant**) to think of hash functions as black boxes. The iterative construction was designed to meet a very modest goal, that of extending the domain of a collision-resistant function, and should not be expected to provide security guarantees beyond that.

2.5 Algebraic hash function

$\oint$ Collision-resistant hash functions can be based on the same hardness assumptions as public-key cryptography. Such functions are largely confined to theoretical papers, being algebraic and

therefore orders of magnitude slower than ad-hoc or block cipher-based hash functions. The main reason why we discuss algebraic hash functions in this note is to correct the popular reference [Sch95, Section 18.12], which does not give two standard constructions and describes two incorrect ones instead.

$\oint$ Discrete logarithm problem in group G of prime order p is to solve the equation $g^x = h$ for x given two group elements $g, h \in G$. Discrete-logarithm based hash function can be designed as follows:

$$H(x, y) = g^x h^y,$$

where g, h are elements of a group where the discrete logarithm problem is hard. It is easy to verify that if the inputs to H are defined modulo p, finding a collision amounts to solving the discrete logarithm problem.

$\oint$ The RSA-based hash function is defined for arbitrary strings. If $N = pq$ is product of two unknown primes, and $g \neq 1$ is an element co-prime with N, then the function defined as

$$H(x) = g^x \mod N$$

is collision-resistant under the hardness of factoring N.

3 Practical Hash Functions

This section covers hash functions that are most likely to be used in practice: MD5, SHA-1, SHA-256, Whirlpool and their close relatives. For their detailed description we refer the reader to the documents issued by standardization bodies.

MD4 and MD5. MD4 was proposed by Ron Rivest in 1990 and MD5 [Riv92] followed shortly thereafter as its stronger version. Their design had great influence on subsequent constructions of hash function. The letters "MD" stand for "message digest" and the numerals refer to the functions being the fourth and fifth designs from the same hash-function family. MD5 follows the design principles outlined in Section 2.4: its compression function is (implicitly) based on a block-cipher and the domain extender is the Merkle-Damgård construction.

The compression function of MD5 operates on 512-bit blocks further subdivided into sixteen 32-bit subblocks. The size of the internal state (i. e., the chaining variable) and its output are both 128 bits. One important parameter of the compression function is the number of *rounds*—the number of sequential updates of the internal state. The compression function of MD5 has 64 rounds organized in an unbalanced Feistel network (for comparison, DES is a Feistel cipher with 16 rounds) each using a 32-bit message subblock to update the internal state via a non-linear mix of boolean and arithmetic operations. Every 32-bit subblock is used four times by the compression function.

MD5 allocates 64 bits in the last block to encode the message's length and it pads the message so that its length is congruent to 448 modulo 512. The padding procedure expands

the message by at least one bit, so the largest message fitting into one block is 447 bits.

SHA-0 and SHA-1. The Secure Hash Algorithm (SHA) initially appeared as part of the design specification of the Digital Signature Standard (DSS) in 1993. Two years later the standard was updated to become what is currently known as SHA-1 [NIST95]. The first version of SHA is referred in the cryptographic literature as SHA-0, although it has never been its official designation. SHA-1 differs from SHA-0 by exactly one additional instruction, which is nonetheless extremely important from the cryptanalytic perspective. Since there were no reasons to prefer the initial version of the standard, SHA-1 replaced SHA-0 in all but most antiquated applications.

SHA-1 is closely modeled after MD4, taking some cues from MD5. It uses the same padding algorithm, breaking the message into 512-bit blocks and encoding the length as a 64-bit number. The size of its internal state and its output length are 160 bits, which is substantially longer than MD5's 128 bits. Although its round functions are simpler and less varied than those of MD5, there are more of them—80 instead of 64. SHA-1 uses a more complex procedure for deriving 32-bit subblocks from the 512-bit message. If one bit of the message is flipped, more than half of the subblocks get changed (as opposed to just four in the case of MD5). It is interesting to note that the cipher, which operates inside the compression function, has never been given any officially recognition. It did not stop the cryptanalytic community, which dubbed the cipher SHACAL, from isolating and studying it (no attacks have been found).

SHA-256, SHA-384, and SHA-512. The new standard issued by NIST in August 2002 adds three members to the SHA family of functions [NIST02]. The connections between the NIST-approved functions are following: SHA-256 and SHA-512 have similar designs, with SHA-256 operating on 32-bit words and SHA-512 operating on 64-bit words. Both designs bear strong resemblance to SHA-1, although they are much closer to each other than to their predecessor. SHA-384 is a trivial modification of SHA-512, which consists of trimming the output to 384 bits and changing the initial value of the chaining variable.

The most important difference between the three new functions and SHA-1 is the new message schedule (procedure for deriving subblocks from one block of the message).

Whirlpool. Whirlpool was designed by Paulo Barreto and Vincent Rijmen (the latter is of AES's fame) and submitted in response to the call for cryptographic primitives issued by NESSIE (New European Schemes for Signature, Integrity, and Encryption) in 2000. Whirlpool was selected together with SHA-256,384,512 as part of NESSIE's portfolio.

Whirlpool's design combines the Merkle-Damgård domain extender with a blockcipher-based compression function. The blockcipher is a variant of AES, which is radically different from SHACAL, and it is converted into a compression function using the Miyaguchi-Preneel construction (Section 2.4). Whirlpool does not target any particular architecture, although 32- or 64-bit processors permit some optimizations impossible in 8-bit implementations.

We summarize in Table 2 parameters of the standard hash functions. Table 3 presents

Name	Block size (bits)	Word size (bits)	Output size (bits)	Rounds	Year of the standard
MD4	512	32	128	48	1990
MD5	512	32	128	64	1992
SHA-0	512	32	160	80	1993
SHA-1	512	32	160	80	1995
SHA-256	512	32	256	64	2002
SHA-384	1024	64	384	80	2002
SHA-512	1024	64	512	80	2002
Whirlpool	512	—	512	10	2003

Table 2: Standard hash functions in a glance.

Name	PIII C, Visual C 6.0	Xeon C, gcc 3.2	PIII assembly, MASM 6.15
MD4	4.8	6.4	—
MD5	7.2	9.4	3.7
SHA-1	19	25	8.3
SHA-256	56	39	20.6
SHA-512	80	135	40.2
Whirlpool	82	112	36.5

Table 3: Performance in cycles/byte. Sources— [P+03] (C code) and [NM02] (assembly).

performance of some of the hash functions on selected processors compiled from two studies [P+03, NM02]. The first report considered portable C implementations of the hash functions limiting optimization to a choice between different combinations of the compiler's options. The second study undertook a painstaking optimization of the assembly language code for Pentium III, taking advantage of the MMX registers and carefully orchestrated pipeline scheduling.

4 Generic Attacks

In this section we discuss generic attacks against schemes that use iterative hash functions based on the Merkle-Damgård construction, highlighting pitfalls of using such functions as black boxes, and review specific attacks against standard hash functions.

Black box abstraction. Software engineers and researchers alike are trained to reduce complexity of systems by thinking of the systems' components as "black boxes"—modules with well-defined interface, separating functionality from implementation details. It is also common to treat programming objects as real-life models of mathematical abstractions. For

instance, we use the `double` type to represent real numbers, or the `rand()` function as a source of randomness. This pragmatic approach is productive as long as its limits are well understood. To continue our example, `double` represents reals with a certain machine-dependent precision and `rand()` from the standard library has a relatively short cycle—both facts are well documented and known to most C/C++ developers.

It is convenient (but wrong!) to think of a concrete hash function such as SHA-1 as a real-world instantiation of a random function. One important characteristic of a random function is that the only way to learn its value on some input is to evaluate the function on precisely this input. As we will show in this section, this is not the case for iterative hash functions (i. e., functions based on the Merkle-Damgård paradigm).

We stress that the generic attacks in this sections are attacks against schemes that uncritically use iterative hash functions, not against hash functions themselves.

4.1 MAC construction

A common application of cryptographic hash functions is to use them as building blocks for message authentication code (MAC) constructions. A MAC is a keyed hash function that may be used to verify the integrity and authenticity of information. Consider two players, Alice and Bob, who share a secret key k. Whenever Alice transmits a message M to Bob, she appends an authentication tag $v = H_k(M)$, which is recomputed on arrival and checked by Bob against the received value. The goal is to prevent the adversary from forging a message-tag pair that would be accepted by Bob and fool him into believing that the message originated from Alice.

A secure MAC is a keyed hash function with the property that the adversary capable of mounting an active attack—querying the function on inputs of his own choosing—is not able to evaluate the function on any other input with a non-negligible probability of success.

Constructing a MAC given an ideal hash function I is a no-brainer—define $H_k(M) := I(k||M)$, where $||$ denotes string concatenation. It is easy to verify that H_k is a perfect MAC. Indeed, unless the adversary gets hold of the key k, he is not able to evaluate H_k himself, and no amount of valid MACs output by Alice and Bob helps the adversary to find k faster than by exhaustive search.

What if we instantiate the ideal function I with SHA-1? Assume for concreteness that the length of the key k is 80 bits and the adversary intercepts a message M of length 256 bits with a valid 512-bit tag t. By definition the tag is equal to

$$t = \mathrm{SHA1}(k||M) = C(k||M||\underbrace{\texttt{10}\ldots\texttt{0}}_{\text{112 bits}}||\underbrace{\texttt{0}\ldots\texttt{0101010000}}_{\text{64 bits}}),$$

where C is the compression function of SHA-1. To forge a valid MAC tag, construct a new message M', which includes M, 112 bits of padding, the 64-bit encoding of 336 followed by

arbitrary text T. The new message M' is thus defined as $M' = M||\texttt{10}\ldots\texttt{0101010000}||T$. Recall the Merkle-Damgård construction and consider its effect on $k||M'$:

$$\text{SHA1}(k||M') = C(C(k||M||\texttt{10}\ldots\texttt{0101010000}), T||\text{padding} + \text{length}) = \\ C(t, T||\text{padding} + \text{length}).$$

In other words, we may compute a valid tag on M' by applying the compression function to t, T and some padding, all of which are known! Therefore, the naïve MAC construction based on an iterative hash function is totally broken, although it may be proved secure using the black box abstraction.

4.2 Security of cascaded hash functions

A generic collision-finding attack takes time of the order of $2^{n/2}$, where n is the length of the hash output. Suppose we have two functions $G, H\colon \{0,1\}^* \mapsto \{0,1\}^n$, each having ideal security $2^{n/2}$, and we would like to construct a collision-resistant function with security 2^n. An obvious solution is to consider the function

$$F(M) := G(M)||H(M),$$

which has output length $2n$. Indeed, if G and H are ideal, so is F, hence it has 2^n security against a collision-finding attack. We claim that this reasoning fails if one of the two functions is iterative.

Assume that one of the two hash functions, say G, is based on the Merkle-Damgård construction. Let C be the compression function of G. Consider a generic collision-finding attack on one application of the compression function (Section 2.3). In time $2^{n/2}$ it finds two messages $M_1^{(0)}$ and $M_1^{(1)}$ so that $h_1 = C(h_0, M_1^{(0)}) = C(h_0, M_1^{(1)})$, where $h_0 = \text{IV}$. Repeat the process, using h_1 instead of h_0. After k iterations we have a list of k pairs satisfying

$$C(h_i, M_{i+1}^{(0)}) = C(h_i, M_{i+1}^{(1)}) = h_{i+1}, \text{ where } 1 \le i \le k.$$

The total running time of this attack is $k2^{n/2}$, as expected. More remarkably, however, is that we now possess a treasure trove of collisions. Any message that has the form $M_1^{(b_1)}||M_2^{(b_2)}||\ldots||M_k^{(b_k)}$, where $b_1, \ldots, b_k \in \{0,1\}$, hashes down to h_k (see Figure 2). There are 2^k such messages, many times more than one would be able to find in time $k2^{n/2}$ had the function G been an ideal hash.

Let $k = n/2$. Making no assumptions about the internal structure of function H, the birthday paradox (Section 2.3) guarantees that with high probability there is a collision under function H among $2^k = 2^{n/2}$ messages of the form $M_1^{(b_1)}||M_2^{(b_2)}||\ldots||M_k^{(b_k)}$. By construction, all of these messages already collide under G (they all hash to h_k), hence the pair colliding under H is also a collision in F.

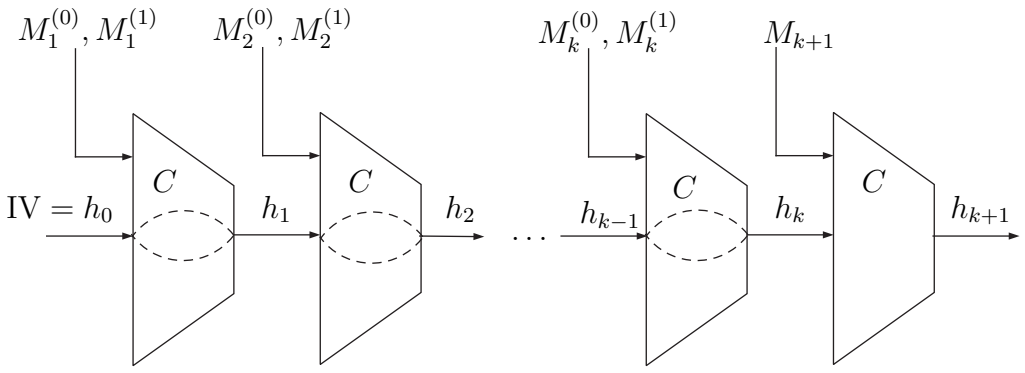

Figure 2: Joux attack.

We demonstrated that a cascading hash function has security less than $n2^{n/2}$, if one of the two functions it is composed of is iterative. The attack was published by Antoine Joux in 2004 [Jou04].

Although the attack is strikingly elegant, it is fairly straightforward. A collection of several messages hashing to the same value is called a multi-collision. The ideal security of a hash function with an n-bit output against m-way collision finding attack is proportional to $2^{n(m-1)/m}$. The argument above demonstrates that the multicollision-resistance of an iterative hash function is much smaller than the ideal security—for $m = 2^k$ we produced an m-way collision in time $k \cdot 2^{n/2}$. Joux's ingenuity was turn this observation into an efficient attack on a well-known construction.

4.3 Meaningful collisions

Although hash functions are expected to be collision resistant, an actual collision-finding attack is often written off as "theoretical" and "irrelevant in practice," since the colliding strings are almost always meaningless and most certainly would be rejected by any natural protocol. It is therefore reasoned that should the adversary prevail in finding a collision, he would have hard time compromising any real system. We disagree with this argument.

In this section we describe two scenarios in which collision-finding attacks on the compression function of an iterative hash function may be converted into collisions of perfectly meaningful messages.

Poisoned block attack. Recall that the Merkle-Damgård construction iterates the compression function of the form $C\colon \{0,1\}^n \times \{0,1\}^m \mapsto \{0,1\}^n$. Suppose there exists an algorithm that given x of length n bits finds two m-bit strings M and M' such that $C(x, M) = C(x, M')$. Most attacks on the real world hash functions do just that: since the Merkle-Damgård construction starts with setting $h_0 := \mathrm{IV}$, where IV is a constant hard-

wired into the scheme, the attacker essentially has no control over the first argument of the function C.

Consider a message $M = M_1||M_2||\ldots||M_k$ formatted into m-bit blocks. Set $h_0 := \text{IV}$ and $h_i := C(h_{i-1}, M_i)$. Fix some $j \in \{0, \ldots, k\}$ and find two strings N and N' such that $C(h_j, N) = C(h_j, N')$. It is easy to check that the two messages that differ in the jth block

$$\begin{aligned} M &= M_1||\ldots||M_{j-1}||N\,||M_{j+1}||\ldots||M_k, \\ M &= M_1||\ldots||M_{j-1}||N'||M_{j+1}||\ldots||M_k \end{aligned}$$

collide under the hash function built based on C.

Notice that although N and N' may be complete gibberish, they are now part of much longer messages that can be carefully constructed in order to inconspicuously accommodate them.

We call this attack the "poisoned block attack" to emphasize the fact that it works in the block level, which is an artifact of the Merkle-Damgård construction.

The effectiveness of the attack was demonstrated by Magnus Daum and Stefan Lucks in 2005, who produced two valid Postscript files with identical MD5 hashes [DL05]. Their attack makes use of the following Postscript construction:

$$(R1)\ (R2)\ \texttt{eq}\ \{\text{instruction set 1}\}\ \{\text{instruction set 2}\}\ \texttt{ifelse}$$

This Postscript code executes the first instruction set if $R1 = R2$, otherwise the second instruction set is run. To launch the attack, align the message so that the poisoned block is inside $R1$, and set the constant $R2$ to contain the other element of the colliding pair. By switching between $R1$ and $R2$ in the first argument of the boolean formula, the attacker controls the execution path, while keeping the value of the hash function unchanged.

$\oint$ **X.509 certificate.** The attack of Arjen Lenstra et el. [LWW05] allows one to construct two X.509 public key certificates [H+02] that collide under MD5. Normally, a certificate issued by a trusted certification authority (CA) binds a subject's public key to its identity, and guarantees that the subject owns the associated private key. The binding is asserted by having the CA digitally sign a specially formatted message that includes, among other things, the subjects's identity and its public key. Even though the standard does not specify an exact signature scheme, most common signature standards hash the message using some dedicated hash function as their first step (see Section 6.1). Therefore, if one is able to produce two X.509 messages that collide under the hash function used by the signature scheme, the semantic of the certificate is compromised: the CA effectively issues a certificate on two public keys, one it has examined and one it has not. Although the attack does to a certain extent undermine credibility of the certification process, it is less damaging than it would appear, since both public keys must be created by the same entity (hence the binding property is still preserved).

$\oint$ At a technical level, the attack works by constructing two RSA moduli with known factorization that share a common postfix. The poisoned block, as in the attack above, consists of one of the

two distinct prefixes, computed by a collision-finding algorithm. The attack allows to efficiently find such moduli of length 2048 bits together with their prime factors. The prime factors are unbalanced, i.e., in contrast with regular RSA, where p and q have the same length (for 2048-bit RSA it would be 1024 bits each), the prime factors generated by the attack are 512- and 1536-bit long. RSA with unbalanced factors is still believed to be secure.

5 Specific Attacks

Table 4 summarizes attacks on standard hash functions that appeared in the public literature as of October 2005. Some of the attacks are trivial to interpret, as they expose collisions in the hash function, some call for more definitions.

Definition We say that (h, x) and (h', x') is a *pseudo-collision* for compression function $C\colon \{0,1\}^n \times \{0,1\}^m \mapsto \{0,1\}^n$ if $C(h, x) = C(h', x')$ and $(h, x) \neq (h', x')$.

A pseudo-collision for a compression function C invalidates the proof of the Merkle-Damgård construction (Section 2.4) that states that if the compression function C is collision-resistant, so is the hash function H based on C. It does not, however, translate into an attack on H, since the attacker does not directly control the value of the chaining variable (first argument of the compression function).

Definition We say that (h, x) and (h, x') is a *free-start* collision for compression function $C\colon \{0,1\}^n \times \{0,1\}^m \mapsto \{0,1\}^n$ if $C(h, x) = C(h, x')$ and $x \neq x'$.

Notice that a free-start attack is stronger than a pseudo-collision attack, since the attacker is forced to use the same value of the chaining variable for both colliding arguments in the free-start attack. This attack still falls short of a collision-finding attack on C-based hash function H for the same reason as before—the Merkle-Damgård paradigm fixes the initial value of the chaining variable, which is unlikely to coincide with the value anticipated by the adversary, and subsequent values of the chaining variables cannot be easily chosen either.

Definition We say that x and x' is a *near-collision* for the hash function H if the Hamming distance between $H(x)$ and $H(x')$ is small (typically, a few bits).

Once again, this attack is not as strong as a collision finder, although it sometimes serves as a precursor to a full attack (as was the case of SHA-0).

Reduced-round hash functions are weaker primitives that share important characteristics with their full-round variants. Cryptanalysts often attack reduced-round hash functions first since such attacks may provide insight into potential attack vectors and can be tested using modest computational resources.

hash	attack			
	author	**type**	**complexity**	**year**
MD4	Dobbertin	collision	2^{22}	1996
	Wang et. al	collision	2^{8}	2005
MD5	dan Boer & Bosselaers	pseudo-collision	2^{16}	1993
	Dobbertin	free-start	2^{34}	1996
	Wang et. al	collision	2^{39}	2005
SHA-0	Chabaud & Joux	collision	2^{61} (theory)	1998
	Biham & Chen	near-collision	2^{40}	2004
	Biham et. al	collision	2^{51}	2005
	Wang et. al	collision	2^{39}	2005
SHA-1	Biham et. al	collision (40 rounds)	very low	2005
	Biham et. al	collision (58 rounds)	2^{75} (theory)	2005
	Wang et. al	collision (58 rounds)	2^{33}	2005
	Wang et. al	collision	2^{63} (theory)	2005

Table 4: Attacks on standard hash functions.

6 Applications

Applications of hash functions are abundant in cryptography and programming practice. We describe several scenarios, which can be used as models for assessing security of real-life applications.

6.1 Digital signatures

Historically, digital signatures were the first application of cryptographically secure hash functions. Hash functions serve a dual role in practical signature schemes: they expand the domain of messages that can be signed by a scheme and they are an essential element of the scheme's security.

The first role arises from the fact that it is much easier to design a signature scheme which is secure for signing messages of a fixed length. Consider, for example, the RSA scheme, where the signature of a message $0 \leq M < N$ is $\sigma(M) = M^d \mod N$. Notice that the restriction that $0 \leq M < N$ is necessary, since $\sigma(M) = \sigma(M+N)$. A collision-resistant hash function that hashes arbitrary-length strings into numbers between 0 and N lets the signer sign any message M by exponentiating (raising to a secret power) the hash of the message $\sigma(H(M)) = H(M)^d \mod N$.

This is a generic mechanism that transforms any signature scheme good for signing messages of a fixed length and a collision-resistant hash function into a signature scheme that

can handle arbitrary-length messages. The mechanism is called the hash-and-sign paradigm and is the basis for all modern practical signature schemes.

It is also quite remarkable that virtually all practical signature schemes are either outright *insecure* or lack proof of security in any reasonable model for signing even fixed-length messages if the message is not hashed. For instance, there is a trivial attack against the RSA signature scheme without a hash: any valid signature $C = M^d \mod N$ enables the attacker to compute a signature on $M^2 \mod N$, which is $C^2 \bmod N = (M^2)^d \bmod N$. The second role of hash functions is to prevent this or similar attacks. It is not well understood which properties of the hash function (akin to one-wayness or collision-resistance) are sufficient to fulfil this role.

A rare example of a signature scheme that uses a hash function only for domain extension (i.e., it does not require a hash function for signing short messages) is the Cramer-Shoup signature scheme [CS00].

For any hash-and-sign signature scheme, such as FIPS-approved RSA, DSA, and ECDSA [NIST00], a practical collision-finding attack on the underlying hash function is devastating. Indeed, if the attacker can prepare two documents M_1 and M_2, whose hashes are identical, then a hash-and-sign signature on M_1 is also a valid signature on M_2: $\sigma(H(M_1)) = \sigma(H(M_2))$. It means that if the attacker can obtain a signature on M_1, he automatically gets a signature on M_2, which the signer has never examined and certainly did not expect to sign.

$\oint$ The hash-and-sign paradigm can be symbolically represented as $\sigma(H(M))$, where H is a hash function and $\sigma(\cdot)$ is the output of a signature scheme. It is imperative that the hash function be collision resistant if we are to stay within this paradigm. Alternatively, we may combine a signature scheme with a keyed hash function satisfying the following definition:

Definition We say that the keyed function $H\colon \{0,1\}^* \times \{0,1\}^\ell \mapsto \{0,1\}^n$ is *target-collision resistant* (TCR) if it is hard for the attacker to win the following game:

1. The attacker chooses $x \in \{0,1\}^*$.
2. A random key k is chosen from $\{0,1\}^\ell$.
3. The attacker outputs $y \in \{0,1\}^*$. The game is won if and only if $H_k(x) = H_k(y)$.

$\oint$ Consider the following signature scheme, symbolically represented by $(k, \sigma(k||H_k(M))$. To sign a message M, the signer generates a random key k and signs the concatenation of $k||H_k(M)$. The signature is then constructed as a pair that consists of k and $\sigma(k||H_k(M))$. It is easy to verify that such a signature scheme is secure as long as the hash function is TCR and $\sigma(\cdot)$ is unforgeable for messages of length n.

$\oint$ There is compelling theoretical evidence that TCR hash functions might be easier to construct than keyless collision-resistant hash functions in the mold of MD5 or SHA-1, although no standard TCR functions currently exist.

6.2 MAC

Message-authentication code (MAC), introduced in Section 4.1, is a keyed hash function satisfying certain cryptographic properties. Not surprisingly, a MAC can be based on a collision-resistant keyless hash function (although a naïve way of doing so can be easily broken, as described in Section 4.1). In fact, the most commonly used MAC construction, called HMAC, falls into this category. Both TLS and IPSec standards suggest using HMAC based on either MD5 or SHA-1. Assuming that H is a collision-resistant hash function, HMAC with secret key $k \in \{0,1\}^\ell$ is defined as follows:

$$\mathrm{HMAC}_k(M) = H(k \oplus c_1 || H(k \oplus c_2 || M)),$$

where c_1 and c_2 are two ℓ-bit constants.

HMAC is provably secure under the assumption that H is collision-resistant and the compression function $C(h, x)$ is pseudo-random if h is unknown. None of the existing attacks on hash functions works if the initial part of the hash function's input is unknown, although such attacks are impossible to rule out in the future.

It is important to note that MAC can be based on other primitives than collision-resistant hashes. One recently standardized construction, CMAC, proposes to use a block cipher, such as AES, in the CBC mode [NIST05]. Another construction, which can be as fast as 1.5 cycles/byte (cf. Table 3), is called UMAC and currently exists in the form of an Internet draft; its RFC is forthcoming [B+05].

6.3 Password tables

A common method of client authentication is to require the client to present a password previously registered with the server. Storing passwords of all users on the server poses an obvious security risk. Fortunately, the server need not know the passwords—it may store their hashes (together with some salt to frustrate dictionary attacks) and use the information to match it with the hashes of alleged passwords [MT79].

With this scheme in place, the adversary succeeds in breaking into the system if he is able to construct any string that has the same hash as any of the original passwords. This should be difficult even if the adversary has access to the password file. It is easy to see that if the hash function is one-way on the set of potential passwords (say, alpha-numeric strings of some reasonable length concatenated with salt), the attacker is forced to do exhaustive search.

Inverting a hash function is harder (and quite different) from finding a collision in the same function. With the exception of MD4, whose reduced-round version has been shown to be invertible, one-wayness of other standard hash functions has not been compromised.

6.4 Key derivation, GUID, hash tables

One common use of hash functions is to "destroy" any structure that may exist in the input, while preserving most of its entropy. Validity of using hash functions for entropy extraction is not based on their cryptographic properties but rather on our belief that a good hash function destroys most of the dependencies that may exist in the bits of its input.

Key derivation. Consider, for example, the Diffie-Hellman key exchange protocol. Alice and Bob exchange values of g^a and g^b for some g, which is an element of a group G of order p. Both Alice and Bob can now compute their shared secret g^{ab}, sometimes called the premaster secret. Under some well-defined hardness assumptions, any eavesdropper does not know anything about g^{ab} other than the fact that $g^{ab} \in G$. Can Alice and Bob use g^{ab} directly as their shared secret key? The answer depends on the group representation, and likely to be no. Indeed, for many cryptographic groups (such as the first two Oakley groups [HC98]) the length of a group element is much longer than its order p. Security of most keyed cryptographic primitives is argued under the assumption that the key is a truly random binary string of some fixed length. An element of group G cannot possibly satisfy the assumption if the size of the group is smaller than its representation.

We need a method that would "extract" entropy in the following scenario: given an element x chosen uniformly at random from $S \subset \{0,1\}^n$, where $2^m \leq |S| < 2^n$, compute a string $h(x) \in \{0,1\}^m$ such that $h(x)$ is near-uniform in $\{0,1\}^m$. If the size of S is sufficiently close (but no less than) 2^m, $h(\cdot)$ transforms a high-entropy but unusable distribution (S may be only a small fraction of $\{0,1\}^n$) into a much more useful distribution with almost the same entropy.

Although there exists a provable method (see the end of the section) for entropy extraction with near-optimal parameters, in practice we content with applying a hash function. That a concrete hash function extracts entropy is a heuristic, unprovable in theory but extremely reliable in practice. For example, the TLS protocol [DA99] takes a conservative approach of XORing together outputs of MD5- and SHA-1-based HMACs (Section 6.2) to derive a master secret from the premaster secret and public randomness.

UUID/GUID. A Universally Unique Identifier (UUID), also known as Globally Unique Identifier (GUID), is a 128-bit long string which is unique (with extremely high probability) across space and time [LMS05]. Early versions of the standard encapsulated an accurate clock reading for temporal uniqueness and an IEEE 802 address for spacial uniqueness. This approach was rightfully criticized for unnecessarily exposing the time and place of the identifier's generation, which can lead to a privacy compromise (as it did on at least one occasion, when the Melissa worm was traced back to its origin via a GUID it carried).

The current version of the standard allows using a hash function (such as MD5 or SHA-1) for privacy protection as well as for UUID generation in the absence of a unique network address. In order to generate a UUID, one can compose a "node ID" from a variety of sources, each of which may be guessable or repetitive, but possessing together enough entropy to

ensure both privacy and uniqueness. The UUID is based on the output of the hash function applied to the node ID.

Hash tables. A hash table is a data structure that supports efficient store and look-up operations. Many variants of hash tables exist [CLRS01, Chapter 11]; here we discuss one of the simplest. Internally the hash table is an array of pointers of size N. To store a key-value pair (k, v), the hash table computes $H(k) = i$ that maps the key to a number between 0 and $N - 1$, and stores the pair in the ith entry of the array. If the entry is already occupied, the pair is added to the list associated with the entry (other collision-resolution strategies are possible and work well in practice).

The only requirement for the hash function H is that it should minimize the number of collisions (keys mapped to the same array's entry) on a typical input. The output of a hash function is a number between 0 and $N-1$ (since the array must fit in memory, N is likely to be less than 2^{40}). If H is based on a cryptographic hash function, such as in this example:

$$H(k) = \text{MD5}(k) \bmod N,$$

then finding collisions in H is only marginally harder than doing so for a non-cryptographic function with similar parameters. Because cryptographically strong hash functions impose a substantial computational overhead without adding any tangible performance guarantees, we do not recommend using them in hash tables implementations.

$\oint$ A non-cryptographic solution to the problem of designing a hash function that "randomizes" its input, which was hinted at earlier in this section, is based on the following definition:

Definition A keyed function $F\colon A \times K \mapsto B$ is called *2-universal family* if for any two nonequal $x, y \in A$ and any (not necessary nonequal) $a, b \in B$ the probability over random $k \in K$ that $F_k(x) = a$ and $F_k(y) = b$ is exactly $1/|B|^2$.

In other words for a randomly chosen key $k \in K$ and *any* $x, y \in A$ the function F_k behaves like a truly random function on the pair $(F_k(x), F_k(y))$.

There are two well-known practical methods of constructing 2-universal families.

Method 1. Let K be the set of all $m \times n$ binary matrices. For any $x \in \{0,1\}^n$ and $M \in K$, let $H_M(x) = Mx$, which is the usual matrix-vector product of M and (transposed) x. Let $A = \{0,1\}^n \setminus (0, \ldots, 0)$ (set of all n-bit non-zero strings). We claim without proof that $H\colon A \times K \mapsto \{0,1\}^m$ is a 2-universal family of functions.

Method 2. Let $A = \{0, 1, \ldots, n-1\}$ and $B = \{0, 1, \ldots, m-1\}$. Choose any prime number $p \geq n$. Let $K = \{(a, b) \mid a, b \in \{0, 1, \ldots, p-1\}, a \neq 0\}$. For any $(a, b) \in K$ and $x \in A$ define

$$H_{a,b}(x) = ((ax + b) \bmod p) \bmod n.$$

Again, it is not difficult to prove that the family of functions $H\colon A \times K \mapsto B$ is 2-universal.

Entropy extraction given a family of 2-universal functions is straightforward: if x is uniformly distributed in a set $S \subset \{0,1\}^m$ of size at least 2^n and $H_k\colon \{0,1\}^m \mapsto \{0,1\}^n$ is a 2-universal function, then for a random $k \in K$ the value $H_k(x)$ is going to be near-uniform on $\{0,1\}^n$ *even if k is public*. This result is known as the "leftover hash lemma."

Consider the hash table example. Rather than applying an expensive cryptographic hash function, such as SHA-1, generate at random two numbers a and b and compute in advance a prime number p. The number of collisions for $H_{a,b}(x)$ defined as above for *any* incoming stream of keys will be on average (over choices of a and b) exactly the same as for SHA-1.

7 Further reading

As a good starting point for basic theory of hash functions and classic results we recommend "Handbook of Applied Cryptography" [MOV96, Chapter 9], currently available on-line.

Research in cryptographic hash functions has been active and, in recent years, explosive. We will undoubtedly see new proposals, tweaks of existing designs, and attacks in years to come. There is no single source to track new development in the field, although many new results first appear on-line on cryptology ePrint archive maintained by IACR: `eprint.iacr.org`.

References

[B+05] John R. Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway, UMAC—Message Authentication Webpage, `www.cs.ucdavis.edu/~rogaway/umac/`

[BRS02] John Black, Phillip Rogaway, and Tom Shrimpton, "Black-box analysis of the block-cipher-based hash-functions constructions from PGV," Proc. of CRYPTO'02, Lecture Notes in Computer Science 2442, Springer, pp. 320–335, 2002. Available from `www.cs.ucdavis.edu/~rogaway/papers/hash.htm`

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, **Introduction to Algorithms, Second Edition**, MIT Press, 2001.

[CS00] Ronald Cramer and Victor Shoup, "Signature schemes based on the Strong RSA assumption," ACM Transactions on Information and System Security, vol. 3(3), pp. 161–185, 2000. Available from `www.zurich.ibm.com/security/ace/`

[DL05] Magnus Daum and Stefan Lucks, "Attacking hash functions by poisoned messages," EUROCRYPT rump session, 2005. Available from `www.cits.rub.de/MD5Collisions/`

[DA99] T. Dierks and C. Allen, "RFC 2246—The TLS protocol version 1.0," IETF RFC 2246, 1999. Available from `www.ietf.org/rfc/rfc2246.txt`

[HC98] D. Harkins and D. Carrel, "RFC 2409—The Internet Key Exchange (IKE)," IETF RFC 2409, 1998. Available from `www.ietf.org/rfc/rfc2409.txt`

[H+02] R. Houseley, W. Polk, W. Ford, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) profile," IETF RFC 3280, 2002. Available from `www.ietf.org/rfc/rfc3280.txt`

[Jou04] Antoine Joux, "Multicollisions in iterated hash functions. Application to cascaded constructions," Proc. of CRYPTO 2004, Lecture Notes in Computer Science 3152, Springer, pp. 306–316, 2004.

[MOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, **Handbook of Applied Cryptography**, CRC Press, 1996. Available from `www.cacr.math.uwaterloo.ca/hac/`

[LMS05] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," IETF RFC 4122, 2005. Available from `www.ietf.org/rfc/rfc4122.txt`

[LV01] Arjen K. Lenstra and Eric R. Verheul, "Selecting cryptographic key sizes," Journal of Cryptology, vol. 14(4), pp. 255–293, 2001. Available from `www.win.tue.nl/~klenstra/key.pdf`

[LWW05] Arjen K. Lenstra, Xiaoyun Wang, and Benne de Weger, "Colliding X.509 Certificates," Cryptology ePrint Archive, Report 2005/067. Available from `eprint.iacr.org/2005/067`

[MT79] Robert Morris and Ken Thompson, "Password security: A case history," Communications of ACM, vol. 22(11), pp. 594–597, 1979.

[NM02] Junko Nakajima and Mitsuru Matsui, "Performance analysis and parallel implementation of dedicated hash functions," Proc. of EUROCRYPT 2002, Lecture Notes in Computer Science 2332, Springer, pp. 165–180, 2002.

[P+03] Bart Preneel et al., "Performance of optimized implementations of the NESSIE primitives," NES/DOC/TEC/WP6/D21/2, Available from `www.cosic.esat.kuleuven.be/nessie/deliverables/D21-v2.pdf`

[Riv92] Ronald L. Rivest, "The MD5 message-digest algorithm," IETF RFC 1321, 1992. Available from `www.ietf.org/rfc/rfc1321.txt`

[Sch95] Bruce Schneier, **Applied cryptography: Protocols, algorithms, and source code in C, Second edition,** Wiley, 1995.

[NIST95] "Secure hash standard," FIPS PUB 180-1, 1995. Available from `www.itl.nist.gov/fipspubs/fip180-1.htm`

[NIST00] "Digital signature standard," FIPS PUB 186-2, 2000. Available from www.csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf

[NIST02] "Secure hash standard," FIPS PUB 180-2, 2002. Available from www.csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[NIST05] "Recommendation for block cipher modes of operation: The CMAC mode for authentication," NIST Special Publication 800-38B, 2005. Available from www.csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf

[Sim98] Daniel R. Simon, "Finding collisions on a one-way street: Can secure hash functions be based on general assumptions?" Proc. of EUROCRYPT'98, Lecture Notes in Computer Science 1403, Springer, pp. 334–345, 1998. Available from research.microsoft.com/crypto/dansimon/me.htm