Part II

Tools

There are many kinds of cryptographic hash functions and we present many definitions. We give a survey of design paradigms and common applications of cryptographic hash functions. We introduce the concept of non-incrementality, which is necessary when we want to use cryptographic hash functions in proofs of knowledge.

Chapter 3

Cryptographic Hash Functions

Typically, cryptography is about encoding and decoding. One can take a message and encode it in such a way that only specific people can reconstruct (decode) the original message from the encoded message (because they have the right secret key). Thus, though messages are mangled in such a way that the ciphertext looks like random noise, it is possible to reconstruct the original message from the ciphertext. This chapter is not about this type of cryptography. This chapter is about cryptographic functions (primitives) which have no inverse: it is impossible (or at least hard) to reconstruct the original message from the ciphertext. Such functions are called *cryptographic hash functions*[1] On first sight it may seem that this type of functions has no sensible application, but this is not the case. In this chapter, we will show a number of well-known applications. Later on in this thesis, we will show a whole new type of application of this type of functions. In Section 3.6 we will elaborate on these new applications, which will be shown in greater detail later on in Chapters 8–10 of this thesis.

In this chapter, many hash function-related concepts necessary for understanding this thesis will be explained. In cases where multiple names for the same concept exist, a footnote at the first use of the concept will list equivalent names of the concept. A more detailed taxonomy of cryptographic hash functions can be found in [Pre93, Pre98]. A highly valuable reflection on the definition of cryptographic hash functions can be found in [And93].

Cryptographic hash functions can be regarded as a special case of hash functions. Therefore, Section 3.1 will explain what a 'normal' hash function actually is, and Section 3.2 will elaborate on what distinguishes cryptographic

[1] Sometimes cryptographic hash functions are called *one-way functions* [DH76].

hash functions from normal hash functions. In Section 3.3 we will explain the *Random Oracle Model*, which can safely be regarded the theoretical ideal of a cryptographic hash function. In Section 3.4 two approaches to construct cryptographic hash functions will be described: the *Merkle-Damgård paradigm*, and the *randomize-then-combine paradigm*. In Section 3.5 we will give brief survey of applications of cryptographic hash functions.

The concept of cryptographic hash functions as they are generally used is not strong enough for our purposes in this thesis. Therefore, we will explain and define the concept of non-incrementality in Section 3.6. In the final section of this chapter, we will briefly summarize what properties of a cryptographic hash function are required for the new applications described later on in this thesis (in Chapters 8–10),

3.1 Normal Hash Functions

In this section, we will give a definition of hash functions, elaborate on this definition, and explain what hash functions are typically used for.

Definition 3.1. *A function is a hash function if it*

1. *takes its input from a large (possibly infinite) domain,*[2]
2. *has a bounded range,*
3. *is designed to minimize* collisions,
4. *is designed to be fast to compute, and*
5. *is deterministic.*

The *input* to a hash function is often called a message or pre-image. The *output* of a hash function is often called the *hash value*, or just simply *the hash*.

If two different pre-images $M_1 \neq M_2$ have the same hash value ($H(M_1) = H(M_2)$), we have a *collision*. Because of the birthday paradox[3], one can be sure that in practical situations collisions will occur.

A few observations about the list of properties of a hash function should be made here. Properties 1 and 2 imply that a hash function is in practical terms always many-to-one. Property 3 implies in practical terms that the output of a hash function should depend on the complete input of the hash function.

[2] In theory, a hash function is a function $H\colon \{0,1\}^* \to \{0,1\}^k$, but in practice the input domain is bounded by a very high number, for example SHA-512 ($H\colon \{0,1\}^{2^{128}} \to \{0,1\}^{512}$) can handle messages with a length of 2^{128} bits [Nat02]. To give some perspective, there are not even 2^{128} bits stored together on all hard disks worldwide.

[3] The birthday paradox states that if there are 23 or more people in a room, the chance that two of them have the same birthday is more than 50%. This number of 23 is much lower than what one might expect from intuition. It stresses that the chances of a "collision" are often underestimated by humans.

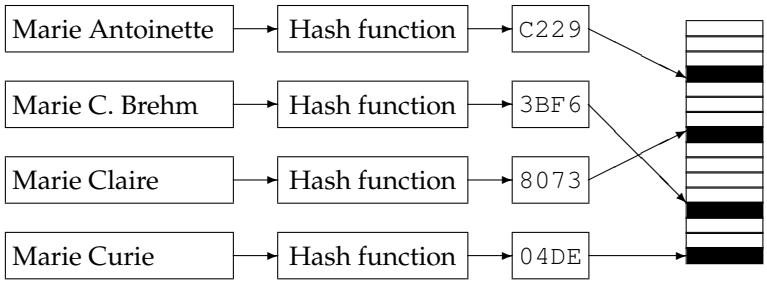


FIGURE 3.1: A 'normal' hash function in action. To the left is a list of (highly non-uniform) names. All of these names are each led through a hash function, resulting in a list of 'more-or-less uniform' hash values. To the right is a data structure with allocated slots (black) and unallocated slots (white). The hash function minimizes the chance that two highly similar names should be stored in the same slot.

Property 4 simply follows from the general goal in computer science to produce efficient computer programs. Later on in this chapter, however, we will see applications where "fast to compute" should apply only to specific uses of hash functions, and not to others, such as, for example, the inverse of a hash function. Property 5 is so obvious for computer scientists that it is often taken for granted, and therefore it is often not considered as a part of the definition. However, since many cryptographic primitives, such as encryption, are often non-deterministic, we deem it important to stress that hash functions are *always and by definition* deterministic.

Definition 3.1 does not mention a typical application, but hash functions are normally used to optimize data structures, and are often not 'visible' from the outside of the data structure. Range sizes of such hash functions are typically only a few orders of magnitude larger than the number of elements stored in the data structure (say a few thousand, or maybe a few million at the most).

In data structures it is often needed to store data values together with some *identification* (an *index*) in such a way, that it is efficient to locate the corresponding data value if given an identification. For example, a teacher would like to find the grades of a student quickly, if he is given a student name. To provide this functionality, one needs a look-up-table (LUT), a data structure that maps indexes to the corresponding data. A LUT has to be designed with a balance between storage requirements and search time. The naive solution is to reserve a *memory slot* for each possible index. This solution wastes an incredible amount of storage space, since usually the number of stored data values is only a tiny fraction of the number of possible indexes. Indeed, the domain of indexes may be infinite. A less naive solution could be to reserve a single slot for a number of large chunks of possible indexes, and hope that it will not happen that the corresponding slot will be needed more than once at the same time.

Therefore, one has to be smart in the way the full domain of possible in-

dexes is divided over the allocated slots. This is where a hash function can be used to optimize data structures. A hash function can be used to determine, given an index, in which allocated slot the corresponding data value should be stored. Figure 3.1 shows an example of how a hash function attributes hash values to indexes and how this determines slot usage.

In most applications, the actual indexes have a highly non-uniform distribution over the domain of possible indexes. If no precautions are taken, many indexes will be projected onto a small set of hash values, which results in a lot of collisions and a lot of unused slots. Thus, a hash function should be designed in such a way that even for highly non-uniform input, its output should be more or less uniform.

Moreover, it should be noted that so far, there is nothing secretive about hash functions. Given a specific hash function, it will generally be easy to construct different pre-images in such a way that they will lead to a collision in a given hash function. Also, it may be easy to infer some properties of the pre-image from the output of the hash function.

3.2 Special Properties

From a cryptography point of view, hash functions are interesting if it would be possible to strengthen some of their properties to the extreme. Hash functions need to be strengthened in such a way that, for example, for any set of messages, (1) the set of hash values is indistinguishable from a uniform distribution, and that (2) collisions are not just unlikely, but actually hard to find. If this could be achieved while the hash function is still cheap to compute, it would open up a whole lot of applications, most of which will be described in section 3.5.

The special properties which distinguish *cryptographic* hash functions from 'normal' hash functions are all related to computational complexity. It definitely goes beyond the scope of this chapter, even this thesis, to give full formal definitions of the hardness properties introduced later on in this section. There are many different ways to define them, and the technicalities involved in these definitions are not relevant for our purposes. Nevertheless, we feel that some clear indication of what we mean by *easy* and *hard* should be provided.

The parameters which play a role in the complexity figures of cryptographic hash functions are l, the length in bits of the pre-image of the hash function, and k, the length in bits of the output of the hash function. For a given hash function, k is always given, and l (obviously) depends on the pre-image with which one feeds the hash function. For a hash function to be *easy to compute*, or *computationally feasible*, means that the number of operations is at most polynomial in l.[4] A computation is considered *hard to compute*, or *computationally infeasible* if the number of required operations is superpolynomial in terms of the input. Specifically, we consider a hash function hard to invert if, given a hash value h,

[4] In practical cases, it is common that the number of operations is at most linear in l.

it requires $O(2^k)$ operations to find a (second) pre-image. Of course, an attack that requires less than $O(2^k)$ may in practical terms still be infeasible.

It depends on the application how far, and in what way, the properties of a hash function must be strengthened. There is no such thing as an all-purpose cryptographic hash function, because requirements of one application may be incompatible with requirements of other applications. To choose a type of cryptographic hash function is therefore to select properties from a menu of possible options. Some options are mutually exclusive, some options can be combined, and some options imply others.

Deciding what properties a cryptographic hash function should have is a very complex task, and in [And93] some stunning examples of overlooked, but required properties in certain applications are shown. Terminology is partially to blame, as the operationalization of a concept is often given the name of the concept itself, though the operationalization is often far from perfect. The most blatant example of this is that a hash function that is called 'collision-free' actually has infinitely many collisions. It is not advisable to literally interpret the linguistic meaning of the words which make up the name of a concept.

For a hash function, one has to choose its *keyedness*, its *freedom*, its *key dependency level* and its *incrementality*. The full menu of items and options to choose from is shown in Figure 3.2.

Keyedness The first choice on the menu is whether or not the cryptographic hash function should be *keyed*. A keyed cryptographic hash function is often called a Message Authentication Code, or simply MAC.[5] A MAC takes two inputs: a key and the pre-image. In the context of MACs, the key is a piece of information which makes computing the function actually feasible: without the key, not only the inverse is hard, but also the 'forward' direction of computation is impossible. Any application of a MAC should include a specification of which principals should know the key, and which principals should not. Otherwise, if the key were to be publicly known, the keyed cryptographic hash function would reduce to a non-keyed cryptographic hash function. (In fact, in many circumstances it reduces to even *less* than a non-keyed cryptographic hash function, which will be explained later on in this section.) If it is unspecified whether a cryptographic hash function is keyed or not, a non-keyed cryptographic hash function is assumed.

Freedom The second choice on the menu is about how obscure the relation between the input and the output should be. In the definitions to come, the parts between square brackets [] give the definitions for MACs. The bitwise exclusive or is written as $\oplus$. In these definitions, *computationally infeasible* means that there exists no polynomial-time function to perform the task mentioned, given the usual assumptions about the complexity hierarchy.

[5] The opposite, a cryptographic hash function that is not keyed, is often called a Manipulation Detection Code, or simply MDC.

One-Way A hash function $H(M)$ [$MAC(K, M)$] is one-way[6] if, for a given hash value h, it is computationally infeasible to construct a message M [and key K] in such a way that $H(M) = h$ [$MAC(K, M) = h$].

Weakly Collision-Free A hash function $H(M)$ [$MAC(K, M)$] is weakly collision-free[7] if, for a given message M_1, it is computationally infeasible to find a message M_2 such that $M_1 \neq M_2$ and $H(M_1) = H(M_2)$ [$MAC(K, M_1) = MAC(K, M_2)$].

Strongly Collision-Free A hash function $H(M)$ [$MAC(K, M)$] is strongly collision-free[8] if it is computationally infeasible to find any two messages M_1 and M_2 such that $M_1 \neq M_2$ and $H(M_1) = H(M_2)$ [$MAC(K, M_1) = MAC(K, M_2)$].

Correlation-Free A hash function $H(M)$ [$MAC(K, M)$] is correlation-free if it is computationally infeasible to find any two messages M_1 and M_2 such that $M_1 \neq M_2$ and the Hamming weight[9] of $H(M_1) \oplus H(M_2)$ [$MAC(K, M_1) \oplus MAC(K, M_2)$] is less than what one would expect if one were to compute $H(M_1) \oplus H(M')$ [$MAC(K, M_1) \oplus MAC(K, M')$] for a lot of randomly chosen M' [Oka93, And93][10].

Correlation freedom means in practical terms that not only collisions are very unlikely and hard to find, but also that *near misses*[11] are unlikely and hard to find. Thus, correlation freedom is a strictly stronger property than strong collision freedom. Moreover, strong collision freedom is a strictly stronger property than weak collision freedom, and weak collision freedom is strictly stronger than one-wayness. This is summarized in Figure 3.2.

Key Dependency For MACs, there are a few more relevant properties, which a MAC is generally assumed to satisfy:

Key-Dependent A MAC $MAC(K, M)$ is key-dependent if, given a pre-image M, it is hard to compute $MAC(K, M)$ (that is, without K)[12].

Chosen Text Attack-Resistant A MAC $MAC(K, M)$ is resistant against a chosen text attack, if given any number of freely chosen pairs $\{M', MAC(K, M')\}$, it is still hard to compute $MAC(K, M)$ for any $M \neq M'$.

---

[6] One-way is also called *first pre-image resistant*.
[7] Weakly collision-free is also called *second pre-image resistant*.
[8] Strongly collision-free is also called *collision-resistant*.
[9] The Hamming weight of a binary string is the number of nonzero bits is the particular string.
[10] Of course, the Hamming weight of the bitwise exclusive or ($\oplus$) of two bitstrings is their Hamming distance.
[11] A near miss is, roughly, that two different messages produce hash values which, though different, are very similar. For example, two hash values are identical except for one or two bits.
[12] Or likewise, one could say that it is hard to guess $MAC(K, M)$ with a chance of success significantly higher than $1/2^k$, where k is the length in bits of the hash value.

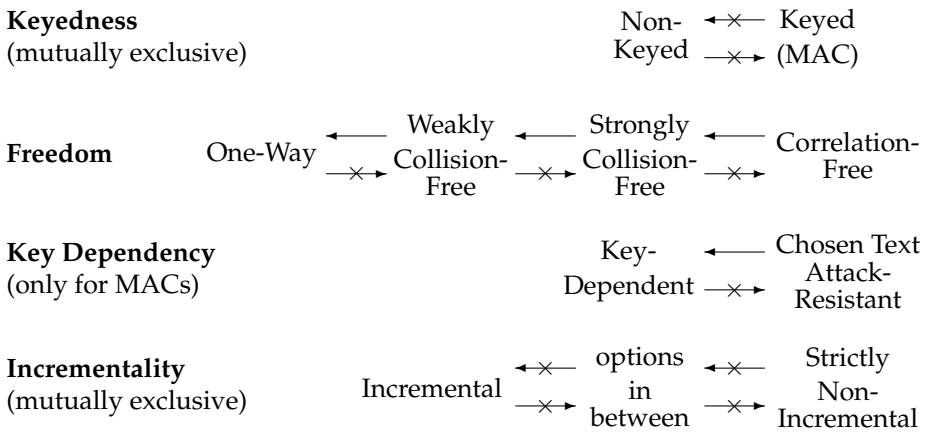


FIGURE 3.2: The relation between various properties of cryptographic hash functions. For the menu items *keyedness* and *incrementality*, the options are mutually exclusive. For the menu items *freedom* and *key-dependency*, the options are increasingly stronger from left to right (e.g. correlation freedom implies strong collision freedom, but not vice versa).

Incrementality For an explanation of the property of incrementality, we refer to Section 3.6. We mention it here for completeness only.

With the complete menu given, we can introduce some commonly used terms for cryptographic hash functions: A *one-way* hash function (OWHF) is one that is not keyed, one-way and weakly collision-free. A *collision resistant* hash function (CRHF) is an OWHF that is also strongly collision-free. There also exists something like a *universal one-way* hash function (UOWHF), which is stronger than an OWHF but weaker than a CRHF [NY89]. We omit its definition for reasons of simplicity.[13]

A Message Authentication Code (MAC) is considered to be key-dependent and resistant against a chosen text attack. It should be noted that key-dependency and resistance against a chosen text attack jointly imply that a keyed hash function is (strongly) collision-free and one-way for someone who does not know the key K. However, a MAC may, by design, actually not be one-way and not collision-free for someone who *does* know the key K. Thus, a MAC $MAC(K, M)$, with a publicly known key K, should not be considered equivalent to a CRHF (or even a OWHF) $H(M)$: the MAC may have none of the interesting properties of the CRHF (!)[14]

[13] For completeness, there are also things like *universal*$_n$, *strongly universal*$_n$ and *strongly universal*$_\omega$ classes of hash functions [CW79, WC81].

[14] For this reason, many experts feel that a MAC should not be considered a cryptographic hash function at all. For completeness and clarity, we have chosen to include MACs in this survey.

3.3 The Random Oracle Model

It has been proven that strong collision freedom is an insufficient property to guarantee *information hiding* and *randomness* [And93]. Information hiding and randomness are, stated informally:

Information Hiding The hash value ($H(x)$) does not leak any information on the pre-image (x).

Randomness The output of the hash function is indistinguishable from random noise.

The history of the definition of cryptographic hash functions is littered with problems popping up every now and then. One cycle of history typically includes: (1) the formal definition of one of the abovementioned properties, (2) the use of a function satisfying the property in some protocol, (3) finding out that the protocol can be broken by some attack on the hash function, and (4) adjusting the definition of a hash function to defy the attack. The properties of one-wayness, weak collision resistance, strong collision resistance and correlation freedom should be regarded as iterations of progressive insight. Not so long ago, strong collision resistance was considered sufficient for many applications, whereas now for the same applications correlation freedom is considered necessary. I would not at all be surprised if correlation freedom will at some point in the near future be proven insufficient for many applications as well.

In fact Preneel, one of the most respected researchers in the field of cryptographic hash functions, recently stated that "we understand very little about the security of hash functions" and "designers have been too optimistic (over and over again...)" [Pre05].

This symptomatic practice leads to the more fundamental question of what notion it is we would like to actually define. What 'real, practical' properties do we believe a 'real' cryptographic hash function actually has? The ideal cryptographic hash function differs from a random function only in that it is deterministic and easy to compute. This defies any formal expression, and the *random oracle model* is the next-best thing one can get. We will introduce this model now.

The purpose of the random oracle model [BR93], introduced by Bellare and Rogaway, is to provide protocol designers with a clear definition of what they can expect from an 'ideal' cryptographic hash function (the random oracle). Whether such ideal cryptographic hash functions actually exist, is a completely different question. The random oracle satisfies 'any property one generally addresses to the notion of a cryptographic hash function'. When designing a protocol, this is of course very useful. A random oracle is defined as follows:

Definition 3.2 (Bellare-Rogaway)**.** *A Random Oracle* $R\colon \{0,1\}^* \rightarrow \{0,1\}^\infty$ *is a map available to all parties*[15]*, good and evil, and every party is allowed to ask the*

[15] There are no such things as 'private oracles'.

oracle only polynomially many questions. Each bit of $R(x)$ is chosen uniformly and independently.

In practical terms, when an oracle is given a question q, it does the following:

1. If the oracle has seen the question q before from whatever party, it gives the answer it gave upon the previous time when it was asked question q.
2. If the oracle has never seen the question q before, it returns a random string of infinite length.

The poser of the question may (and in all practical cases will) instruct the oracle not to physically return the full length random string, but just a prefix of this string of a certain given length.

It is easy to see that a Random Oracle satisfies the properties of information hiding and randomness, as well as all the properties given in Section 3.2.

When using the Random Oracle Model, all calls to a cryptographic hash function are replaced with calls to the oracle (which one might call a black box [Sch98]). Then the protocol is proven correct within this setting. Because such oracles do not seem to exist in real life, the oracle consult has to be replaced again by a call to a 'suitable' cryptographic hash function[16], when such a protocol is deployed in real life.

The Random Oracle methodology is not sound: though it can help detecting protocol flaws, protocols proven secure in the Random Oracle Model cannot be assumed secure when the oracle is replaced by an implementation of a cryptographic hash function [CGH98]. Moreover, it is shown that to replace the oracle with an implementation, one faces some very serious problems [BGI+01]. Nevertheless the Random Oracle methodology is a very valuable one. It provided the best formalization of the properties addressed to cryptographic hash function so far. It has been of great value to protocol design and analysis. Protocols proven correct in the Random Oracle methodology can 'in real life' only be broken by an attack on the internal structure of the hash function, within the setting of protocol interactions [BM97].

3.4 Design Paradigms

The operational design of a hash function is as complex as its definition. But how are cryptographic hash functions actually designed? There are currently two paradigms, the *Merkle-Damgård paradigm*, and the *randomize-then-combine paradigm*. Both paradigms chop up the pre-image of the hash function in a series of fixed-length blocks of bits, and then combine these blocks in some specific way. Depending on the paradigm, the combining specifics differ.[17]

[16] The notion of 'suitable' has not yet been formalized in the literature.

[17] Furthermore, when these paradigms are compared with modes used in cryptography, one can see the Merkle-Damgård paradigm has some similarities to the cipher-block chaining (CBC) mode, and the randomize-then-combine paradigm has some similarities to the electronic codebook (ECB) mode.

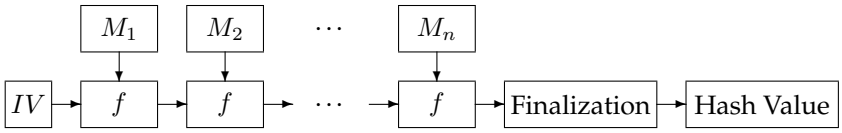


FIGURE 3.3: A Merkle-Damgård hash function. The message M consists of n blocks of the same size, $M_1, M_2, \ldots M_n$. The initialization vector IV has a fixed value. The function f is called the *compression function*.

Because the length of individual blocks is fixed, the original message has to be modified (padded) in such a way that its length is a multiple of the block length. This can be done by adding zeroes at the end of the message until its length is a multiple of the block length. Care has to be taken to make sure that this padding procedure does not weaken the hash function by mapping different pre-images to the same modified pre-image. This problem is solved by encoding the length of the pre-image into the padded message.[18]

Merkle-Damgård In the Merkle-Damgård paradigm [Mer90b, Dam90], the individual blocks are combined by means of a *compression function* f, which takes as input the 'hash so far' and the next message block. At the beginning of the message, the 'hash so far' is a constant *initialization vector* (IV). When all blocks have been processed, the 'hash so far' may undergo some *finalization*, which results in the hash value. In Figure 3.3 the information flow of a hash function using this design is shown graphically.

A hash function in the Merkle-Damgård paradigm is *chaining* (or iterative): blocks earlier in the sequence influence how the blocks later in the sequence are processed. One implication of this is that this type of hash functions cannot be parallellized: adding more hardware cannot speed up the computation of a hash value. Examples of hash functions using this paradigm are MD5, SHA-1 and RIPEMD-160 [DBP96]. Merkle and Damgård have proven that if the compression function is strongly collision-free, then so is the whole hash function [Mer90b, Dam90].

Randomize-then-combine Another approach to cryptographic hash function design is provided by the *randomize-then-combine paradigm* by Bellare et al [BM97, BGG95, BGG94]. Instead of sequentially processing all blocks of the message, the n blocks are processed independently by a *randomizing function* g[19]. The n obtained results are then combined using some well-chosen *combining function* $\odot$. This combining function $\odot$ is chosen in such a way that it is associative[20] and commutative[21] within a group.

[18] This technique is sometimes called MD-strengthening.

[19] In [BM97] the randomizing function is denoted with h, but we will use g instead. The randomizing function could be considered the equivalent of the compression function f in the Merkle-Damgård paradigm.

[20] The parentheses may be moved or removed, e.g. $((x + y) + z) = (x + (y + z)) = x + y + z$.

[21] The terms may be re-ordered, e.g. $x + y = x + y$.

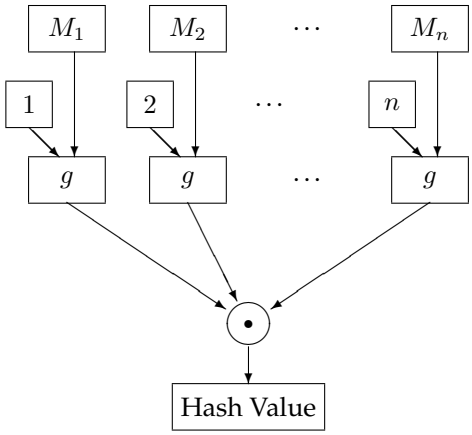


FIGURE 3.4: A hash function of the randomize-then-combine paradigm. The message M consists of n blocks of the same size, $M_1, M_2, \ldots M_n$. The *randomizing function* g combines a message block M_i with its sequence number i. The *combining function* $\odot$ combines the results of all invocations of the randomizing function g into one hash value.

Thus, there is also an identity element[22] 1 and an inverse[23] $^{-1}$ with respect to the combining function $\odot$. The function g takes as input not only a message block, but also the sequence number of the message block. This has to do with the commutative nature of the combining function: without the sequence number it would be trivial to construct collisions for the hash function as a whole. A graphical picture of the information flow in a hash function of this paradigm is shown in Figure 3.4.

There are various options for choosing a combining function, such as multiplication within a suitable group G (MuHASH) or modular addition (AdHASH). Bellare and Micciancio have proven that if the discrete logarithm in G is hard and g is "ideal", then MuHASH is strongly collision-free; and that AdHASH is a UOWHF if the *weighted knapsack problem* (which they define)[24] is hard and g is "ideal" [BM97]. Similar results have been obtained by Impagliazzo and Naor [IN96]. Computationally these combining operations are cheap. The bitwise exclusive or ($\oplus$) is no good candidate for the combining function in the case of a non-keyed hash: it can easily be proven insecure [BM97]. However, within a MAC the bitwise exclusive or *can* be applied [BGR95].

[22] $x \odot 1 = x$

[23] $x \odot x^{-1} = 1$

[24] The *weighted knapsack problem* is a modification of the *subset sum problem*, which is a special case of the *knapsack problem*. These are all problems in combinatorial optimization. The subset sum problem and the knapsack problem are well known NP-complete problems. The weighted knapsack problem is defined in [BM97] and is assumed to be NP-complete.

hash function	bit size	year of publication	reference	status
MD5	128	1992	[Riv92]	obsoleted by [WY05]
RIPEMD-160	160	1996	[DBP96]	
SHA-1	160	1992	[Nat92]	obsoleted by [WYY05]
SHA-224	224	2004	[Nat04]	
SHA-256	256	2002	[Nat02]	
SHA-384	384	2002	[Nat02]	
SHA-512	512	2002	[Nat02]	

TABLE 3.1: Some commonly used cryptographic hash functions and the size of the hash values they produce.

Because of the independency of the computation of g, and the nature of the combining function $\odot$, the computation of a hash value can be done in parallel. There is another advantage of the randomize-then-combine paradigm: it is *incremental*. This roughly means that once a hash value $h = H(x)$ is computed, and the pre-image x is modified into x', the time required to compute $h' = H(x')$ is "proportional" to the "amount of difference" between x and x' [BGG94]. The implications of incrementality will be addressed in Section 3.6.

In both paradigms, there is a strong dependency on the strength of an internal function: the compression function f in case of the Merkle-Damgård paradigm, and the randomizing function g in case of the randomize-then-combine paradigm. Neither paradigm gives specific instructions on how to construct this function, except that it should be strongly collision-free. In practice, these functions are chosen to be complex myriads of bitwise operations such as shifts, rotations, ors, exclusive ors, ands and negations. That such a function f or g constructed in this way is strongly collision-free, is no more than a bold claim. For the two most-used hash functions MD5 and SHA-1, the conjectures that its compression functions are strongly collision-free, have been shown to be overly optimistic [WY05, WYY05].

The fixed size of the hash value influences the strength of the corresponding cryptographic hash function: it is trivial to find collisions for a hash function that produces hash values of, say, only 8 bits in length, and it will be impossible to find collisions on a good cryptographic hash function which creates hash values of 2^{16} bits long. What hash sizes are practical and whether it is computationally feasible to find collisions depends on the state of the art of computing power: if the hash size is chosen too small, collisions are easily detected, and if it is chosen too large, the computational and storage requirements grow too large. For illustrational purposes, Table 3.1 shows the hash sizes for various hash functions, commonly known and used in 2006.

3.5 Common Applications

In this thesis, we will introduce a new application domain of cryptographic hash functions. To be able to see how our application differs from existing applications of cryptographic hash functions, we will describe the spectrum of uses for cryptographic hash functions:

1. password protection
2. manipulation detection
3. optimization of existing cryptographic signature schemes
4. creation of new cryptographic signature schemes
5. random-number generation
6. creation of (symmetric) encryption schemes
7. commitment schemes
8. computational currency and spam protection

In all applications, the description of the hash function is public. We will explain the applications one by one in more detail:

1. Probably the oldest application of cryptographic hash functions is protection of the *password file*. On a computer system with password-protected user accounts, one needs to store passwords in such a way that (1) the passwords cannot be derived from the password file, but (2) the password file should contain enough information to positively identify someone who claims to know a specific user password. The solution is to store the user name and the hash value of the password together in the password file[25].

2. The best known application of cryptographic hash functions is to protect data from being manipulated by a malicious party: a hash value can be used to establish message integrity. Computing hash values and communicating these over the same communication channel as the message itself will however not help anything, since the hash value is subject to the same manipulation powers of the adversary as the original message.

   When a MAC is used, the integrity of the message depends on the secrecy of the key used: it depends on the quality of the key management[26]. When a non-keyed cryptographic hash function is used, the integrity of the message is transferred to the integrity of the hash value: the hash value needs to be protected against manipulation in some way or another. One way to accomplish this is to use a separate authenticated communication channel, another way is to cryptographically sign or encrypt the

[25] This solution is not without problems, though. Badly chosen passwords can be detected by dictionary attacks.

[26] Note that the keys used in MAC are *symmetric*: the sender and the verifier share the same key. The method for message authentication presented by Tsudik in [Tsu92] is roughly equivalent to the use of a MAC.

hash value, thereby reducing the problem of integrity to a problem of key management.

3. Transferring the integrity of a large amount of data to the integrity of a smaller amount of data (as explained in the previous application) can also help to optimize schemes in which large amounts of data must be cryptographically signed. Methods for cryptographic signatures are generally computationally expensive, and computational costs often grow superlinearly in the size of the data to be signed. This makes cryptographically signing a message of one megabyte in size very — if not prohibitively — expensive. Instead, one can compute the hash value of the same message, and sign the hash value [Dam88]. Using this two-step signature scheme, the cost of signing a message is linear in size of the input instead of superlinear. Moreover, this scheme saves storage space, since a cryptographic signature is generally about the same size as the message signed. When using two-step signing, the signature is about the size of the hash value, instead of the size of the original message.

   In communication, cryptographic signatures facilitate integrity, authenticity (the receiver can verify that the sender is who he claims he is) and non-repudiation (the sender cannot deny having sent the message). In software protection, cryptographic signatures facilitate the detection of *malware* and *Trojan horses*.[27] Theoretically, cryptographic hash functions are not required for these applications, but cryptographic hash function help to optimize these applications up to the point that they are actually feasible.

4. It is also possible to build cryptographic signature schemes from cryptographic hash functions alone, but these schemes are merely a proof of concept and they are not very practical because they require a large public storage. There are four known hash-based digital signature schemes: Diffie-Lamport [DH76, page 35], Winternitz [Mer90a, page 227], Merkle [Mer90a] and Rabin [Rab78].

5. Yet another use of a cryptographic hash function is to use it to build a pseudorandom number generator (PRNG) or even a *cryptographically secure* pseudorandom number generator (CSPRNG). In the latter case it is required to keep the pre-image[28] secret.

6. The output of a CSPRNG can in turn be used as a keystream for a one-time pad (OTP)[29], which is an algorithm for symmetric encryption[30]. An algorithm for symmetric encryption can also be built directly from a cryptographic hash function, by making it the F-function in a Feistel cipher

[27] The best known example of such a malware detection software is Tripwire, which runs on UNIX systems.

[28] In the design of PRNGs, the pre-image is also called the *seed*.

[29] The OTP is also called the Vernam cipher.

[30] Of course, a *true* OTP uses a keystream that is truly random, and does not depend on a seed of a fixed size. Therefore, an OTP using a hash function based keystream is not unbreakable, as opposed to a *true* OTP.

[Fei73, FNS75][31].

7. Commitment schemes can also be built using cryptographic hash functions. In a game or protocol, a player can commit to a particular chosen value M without instantaneously disclosing the value by publishing only the hash value $H(M)$. Later on in the game the player must disclose M. Since the player has published $H(M)$, he cannot choose to publish another message M' since $H(M') \neq H(M)$ [DPP94, Dam97, CMR98].

8. A relatively recent application of cryptographic hash functions is to create some kind of *computational currency unit*. The idea is this: the difficulty of finding a collision in a cryptographic hash function depends on the bit length of the hash values it produces. Thus, by trimming the size of a hash value to an appropriate length, it is possible to create puzzles that are moderately hard to solve, but at the same time still tractable (cf. [DN93]). For example, it is feasible to create two pre-images such that the first 20 bits of their hash values are the same[32]. We call this a *partial hash collision*. The interesting feature here is that though it is costly to *find* partial collisions, it is very cheap to *verify* them. By showing two pre-images which result in a partial hash collision, one can 'prove' to have invested a certain amount of computation time. This can be used to combat spam, by accepting only emails for which sufficient computation time has been invested (e.g. Hashcash [Bac02]). It can also be used to construct a transferable currency unit in peer-to-peer (P2P) systems [GH05].

As can be seen from the examples just described, cryptographic hash functions have a lot of applications. This does not mean that every application of a cryptographic hash function is legitimate in the sense that it offers cryptographic guarantees. For example, the hash values used to identify files in P2P filesharing systems do not provide any guarantees on the files exchanged via such a P2P filesharing system. Fortunately, no such guarantees are suggested by P2P systems. That it is easy to illegitimately assume guarantees from the use of a cryptographic hash function is demonstrated by security expert Schneier. His first explanation on cryptographic hash functions reads [Sch96, page 31]:

> "If you want to verify someone has a particular file (that you also have), but you don't want him to send it to you, then you ask him for the hash value. If he sends you the correct hash value, then it is almost certain that he has that file."

[31] In general, when choosing an encryption algorithm, the option of a hash-function based one may not be the best choice in terms of efficiency. Nevertheless, there may be legal reasons to choose for such a function. The United States of America have strict rules for the export of cryptographic software; cryptographic software is considered a type of military arms. Within these export rules, cryptographic hash functions are sort-of neglected. Thus, to legally circumvent these export limitations, it can help to use encryption algorithms based on cryptographic hash functions. In the years just before '9/11', these export rules have been relaxed somewhat, but there are still limitations.

[32] By feasible we mean here that such a collision can be found within approximately one second on workstation hardware current in 2006.

Equivalent claims are made in [BAN89b, AvdHdV01]. The precise claim in [BAN89b] will be the subject of Chapter 5. Unfortunately, the claim is false. The problem is that in the above situation sketch, there is no mention that the file should be kept secret. If there is a third person who is willing to publish the hash value of the file, anybody can 'prove' possession of the file.

However, it *is* possible to use cryptographic hash functions to prove possession of specific information. It is more complex than Schneier assumed. Chapter 8 will address the specifics of this application, which are far from trivial. Chapters 9 and 10 of this thesis will actually show protocols for this application. These protocols require a specific kind of cryptographic hash function: the *non-incremental* hash function. In the next section, we will explain and define what that is.

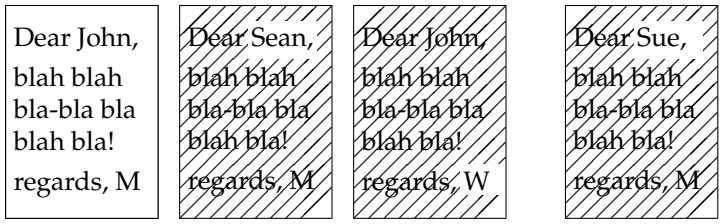
3.6 (Non-) Incrementality

Bellare, Goldreich and Goldwasser have introduced the randomize-then-combine paradigm to show that *incremental* cryptographic hash functions can be built. The concept of incrementality deserves attention, because it shows that certain cryptographic hash functions possess properties we do not desire for our purposes in this thesis. In this section, we will explain what we *do* require. To give some context to our definition, we will first informally explain what incrementality is.

Suppose one wants to send to many different people a signed message that is essentially the same, except for some small parts, such as the addressee field or the date. For every single individual message, the signature has to be recomputed. As the number of similar signed messages grows, the cost of computing the signatures grows as well. Does this mean that signing one million messages is one million times as expensive as signing just one message? In fact, it need not be. Just as buying one million copies of the same book is not one million times more expensive than buying just one copy, it is possible to get a quantity discount on the computational cost of cryptographic signatures.

The terms and conditions of the discount are as follows: the signature should use a two-step signature scheme, and the hash function involved should be *incremental*. The original message must be hashed in the 'old-fashioned' way at least once. To obtain a hash value for every consecutive message, a procedure should be followed, which uses the hash value of the original message, and a description of how the current message differs from the original message. The amount of discount you get on the signature consecutive message is inversely correlated to the size of the *difference description*. Thus, if the body of messages you wish to sign is highly homogeneous, you will get a large discount. If on the other hand the messages are unrelated, your discount will be very small. Similarly, the discount one may get when ordering one million different books will never be as big as when one orders one million copies of the same book. In Figure 3.5 the idea of incremental hash functions is shown.

The practical advantage of incremental hashes and signatures is question-



message number	1	2	3	⋯	∞
individual cost	$\text{size}(M_1)$	$\text{size}(\Delta_2)$	$\text{size}(\Delta_3)$		$\text{size}(\Delta_i)$
incremental average cost	$\text{size}(M_1)$	$\frac{\text{size}(M_1)+\text{size}(\Delta_2)}{2}$	$\frac{\text{size}(M_1)+2\cdot\overline{\text{size}(\Delta)}}{3}$		$\overline{\text{size}(\Delta)}$

FIGURE 3.5: Incremental hash function in action. For the first message, the full message must be consulted. For the next messages, the computation of the hash depends only on the amount of difference of the current message to the first message. As the number of messages grows to infinity, the average cost of hashing a message converges to the average amount of difference. The description of the difference between M_1 and M_i is denoted Δ_i, and $\overline{\text{size}(\Delta)}$ denotes the running average of $\text{size}(\Delta)$. The size of Δ is roughly proportional to the size of the 'not-wiped out' part of messages 2, 3 and onwards.

able. The cost of physically sending a signed message is still proportional to the length of the message. Speeding up the process of creating the signature cannot change that, and will therefore not shorten the whole process of signing and sending a message by more than a constant factor. In fact, the advantages of incremental hashing and signing do only matter if the person generating the hash of the message does not send the message as often as he signs it. In [BGG94, BGG95] some application domains of incremental hashes and signatures are described, including virus protection and signed streaming media.

Incrementality offers some slight advantages, but it also has some clear disadvantages. Consider the following situation:

> Bob wants Alice to prove she possesses M, but Bob doesn't want Alice to send the full message M. Bob knows that the hash value h of M is publicly known, and therefore Bob certainly cannot ask Alice to send just h. Bob decides to create a 'difference description' Δ, which describes how to transform M into M'.
>
> Bob asks Alice to present the hash value h' of M'.

This protocol is faulty: Alice can 'prove' possession of M by exploiting the incrementality of the hash function. That is, Alice can construct $H(M')$ without possessing M or being able to construct M. Thus, for a protocol like this one to be correct, it is necessary that the hash function is not incremental. In that case, the only way to compute $H(M')$ is to construct M' first, and then compute its hash value.

It is not trivial to give a precise formal definition of non-incrementality.[33] Let us give an informal description of the notion of non-incrementality:

Non-Incremental A cryptographic hash function is non-incremental, if it is always necessary to have the full pre-image at hand to compute the hash value of this pre-image.

Hash functions from the randomize-then-combine paradigm are obviously incremental, so that kind of hash functions will not do for our purposes. Hash functions from the Merkle-Damgård paradigm are sort-of incremental, but in a less obvious way. To verify this, see that to compute $H(M_1 \cdot M_2)$, the algorithm computing the hash value passes through a local state where M_1 has been read, but M_2 not yet. This local state can be stored. To compute $H(M_1 \cdot M_2')$, where $M_2 \neq M_2'$, it is sufficient to know the local state and M_2'. This problem can be circumvented by defining $H(M)$ to be equal to $H_{MD}(M \cdot M)$: the pre-image M is 'fed' twice through the Merkle-Damgård hash function $H_{MD}(\cdot)$. As a result of *chaining* (blocks earlier in the sequence influence how the blocks later in the sequence are processed), all blocks of the first iteration of message M, influence the processing of all blocks of the second iteration of M. Therefore, it is not possible to store a local state that allows one to compute $H(M_1 \cdot M_2)$ without knowing M_1.

The solution we propose, to feed the pre-image twice through the hash function is not totally new; Ferguson and Schneier have proposed this solution and similar solutions to fix the length-extension weakness of the Merkle-Damgård paradigm [FS03, pages 90–94]. This is not unexpectedly, as length-extension and incrementality are interrelated properties.[34]

3.7 Conclusion

In this chapter, we have explained what a cryptographic hash function is, and elaborated on some of the difficulties in defining cryptographic hash functions. Moreover, we have shown that incrementality, a property often considered a *feature* of some cryptographic hash functions, can actually be a *drawback*. We have informally but precisely defined non-incrementality, a property that guarantees the specific function does not have this drawback.

In this thesis, we will give a new application of cryptographic hash functions. For this application, we require that it is

1. non-keyed,
2. correlation-free and
3. non-incremental.

We will prove our protocols correct in the random oracle model.

[33] Given the general history of hash function definitions which is sparkled with erroneous definitions, we do not feel safe enough yet to give a formal definition at this point.

[34] It is beyond the scope of this chapter to explain the length extension weakness of the Merkle-Damgård paradigm, consult [FS03, pages 90–94] for a good explanation.