Chapter 9

Hash Functions and Data Integrity

Contents in Brief

9.1 Introduction

Cryptographic hash functions play a fundamental role in modern cryptography. While related to conventional hash functions commonly used in non-cryptographic computer applications – in both cases, larger domains are mapped to smaller ranges – they differ in several important aspects. Our focus is restricted to cryptographic hash functions (hereafter, simply hash functions), and in particular to their use for data integrity and message authentication.

Hash functions take a message as input and produce an output referred to as a *hash-code*, *hash-result*, *hash-value*, or simply *hash*. More precisely, a hash function h maps bit-strings of arbitrary finite length to strings of fixed length, say n bits. For a domain D and range R with $h : D \to R$ and $|D| > |R|$, the function is many-to-one, implying that the existence of *collisions* (pairs of inputs with identical output) is unavoidable. Indeed, restricting h to a domain of t-bit inputs ($t > n$), if h were "random" in the sense that all outputs were essentially equiprobable, then about 2^{t-n} inputs would map to each output, and two randomly chosen inputs would yield the same output with probability 2^{-n} (independent of t). The basic idea of cryptographic hash functions is that a hash-value serves as a compact representative image (sometimes called an *imprint*, *digital fingerprint*, or *message digest*) of an input string, and can be used as if it were uniquely identifiable with that string.

Hash functions are used for data integrity in conjunction with digital signature schemes, where for several reasons a message is typically hashed first, and then the hash-value, as a representative of the message, is signed in place of the original message (see Chapter 11). A distinct class of hash functions, called message authentication codes (MACs), allows message authentication by symmetric techniques. MAC algorithms may be viewed as hash functions which take two functionally distinct inputs, a message and a secret key, and produce a fixed-size (say n-bit) output, with the design intent that it be infeasible in practice to produce the same output without knowledge of the key. MACs can be used to provide data integrity and symmetric data origin authentication, as well as identification in symmetric-key schemes (see Chapter 10).

A typical usage of (unkeyed) hash functions for data integrity is as follows. The hash-value corresponding to a particular message x is computed at time T_1. The integrity of this hash-value (but not the message itself) is protected in some manner. At a subsequent time T_2, the following test is carried out to determine whether the message has been altered, i.e., whether a message x' is the same as the original message. The hash-value of x' is computed and compared to the protected hash-value; if they are equal, one accepts that the inputs are also equal, and thus that the message has not been altered. The problem of preserving the integrity of a potentially large message is thus reduced to that of a small fixed-size hash-value. Since the existence of collisions is guaranteed in many-to-one mappings, the unique association between inputs and hash-values can, at best, be in the computational sense. A hash-value should be uniquely identifiable with a single input *in practice*, and collisions should be *computationally* difficult to find (essentially never occurring in practice).

Chapter outline

The remainder of this chapter is organized as follows. §9.2 provides a framework including standard definitions, a discussion of the desirable properties of hash functions and MACs, and consideration of one-way functions. §9.3 presents a general model for iterated hash functions, some general construction techniques, and a discussion of security objectives and basic attacks (i.e., strategies an adversary may pursue to defeat the objectives of a hash function). §9.4 considers hash functions based on block ciphers, and a family of functions based on the MD4 algorithm. §9.5 considers MACs, including those based on block ciphers and customized MACs. §9.6 examines various methods of using hash functions to provide data integrity. §9.7 presents advanced attack methods. §9.8 provides chapter notes with references.

9.2 Classification and framework

9.2.1 General classification

At the highest level, hash functions may be split into two classes: *unkeyed hash functions*, whose specification dictates a single input parameter (a message); and *keyed hash functions*, whose specification dictates two distinct inputs, a message and a secret key. To facilitate discussion, a hash function is informally defined as follows.

9.1 Definition A *hash function* (in the unrestricted sense) is a function h which has, as a minimum, the following two properties:

1. *compression* — h maps an input x of arbitrary finite bitlength, to an output $h(x)$ of fixed bitlength n.
2. *ease of computation* — given h and an input x, $h(x)$ is easy to compute.

As defined here, *hash function* implies an unkeyed hash function. On occasion when discussion is at a generic level, this term is abused somewhat to mean both unkeyed and keyed hash functions; hopefully ambiguity is limited by context.

For actual use, a more goal-oriented classification of hash functions (beyond *keyed* vs. *unkeyed*) is necessary, based on further properties they provide and reflecting requirements of specific applications. Of the numerous categories in such a *functional classification*, two types of hash functions are considered in detail in this chapter:

1. *modification detection codes* (MDCs)
   Also known as *manipulation detection codes*, and less commonly as *message integrity codes* (MICs), the purpose of an MDC is (informally) to provide a representative image or *hash* of a message, satisfying additional properties as refined below. The end goal is to facilitate, in conjunction with additional mechanisms (see §9.6.4), data integrity assurances as required by specific applications. MDCs are a subclass of *unkeyed* hash functions, and themselves may be further classified; the specific classes of MDCs of primary focus in this chapter are (cf. Definitions 9.3 and 9.4):
   (i) *one-way hash functions* (OWHFs): for these, finding an input which hashes to a pre-specified hash-value is difficult;
   (ii) *collision resistant hash functions* (CRHFs): for these, finding any two inputs having the same hash-value is difficult.
2. *message authentication codes* (MACs)
   The purpose of a MAC is (informally) to facilitate, without the use of any additional mechanisms, assurances regarding both the source of a message and its integrity (see §9.6.3). MACs have two functionally distinct parameters, a message input and a secret key; they are a subclass of *keyed* hash functions (cf. Definition 9.7).

Figure 9.1 illustrates this simplified classification. Additional applications of unkeyed hash functions are noted in §9.2.6. Additional applications of keyed hash functions include use in challenge-response identification protocols for computing responses which are a function of both a secret key and a challenge message; and for key confirmation (Definition 12.7). Distinction should be made between a MAC algorithm, and the use of an MDC with a secret key included as part of its message input (see §9.5.2).

It is generally assumed that the algorithmic specification of a hash function is public knowledge. Thus in the case of MDCs, given a message as input, anyone may compute the hash-result; and in the case of MACs, given a message as input, anyone with knowledge of the key may compute the hash-result.

9.2.2 Basic properties and definitions

To facilitate further definitions, three potential properties are listed (in addition to *ease of computation* and *compression* as per Definition 9.1), for an unkeyed hash function h with inputs x, x' and outputs y, y'.

1. *preimage resistance* — for essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x' such that $h(x') = y$ when given any y for which a corresponding input is not known.[1]
2. *2nd-preimage resistance* — it is computationally infeasible to find any second input which has the same output as any specified input, i.e., given x, to find a 2nd-preimage $x' \neq x$ such that $h(x) = h(x')$.

[1]This acknowledges that an adversary may easily precompute outputs for any small set of inputs, and thereby invert the hash function trivially for such outputs (cf. Remark 9.35).

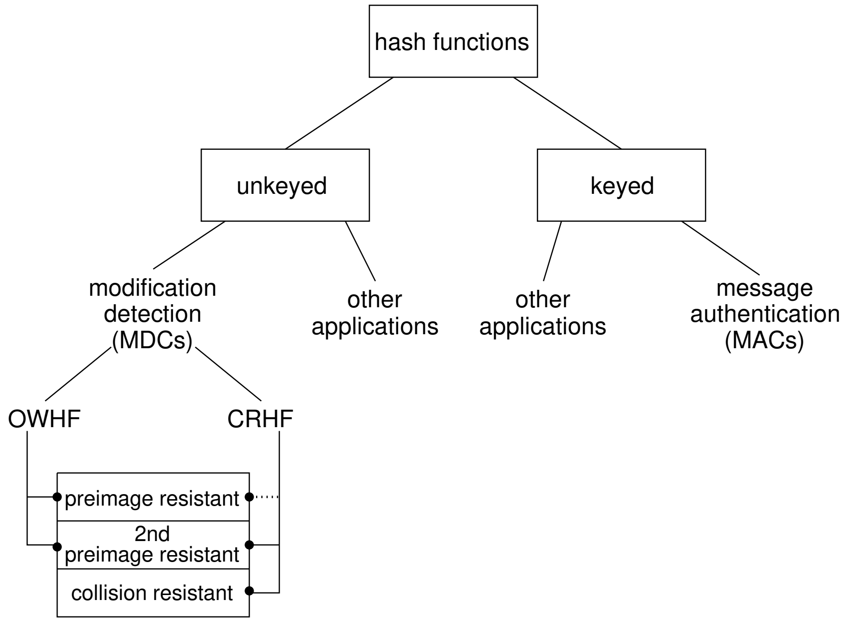

Figure 9.1: *Simplified classification of cryptographic hash functions and applications.*

3. *collision resistance* — it is computationally infeasible to find any two distinct inputs x, x' which hash to the same output, i.e., such that $h(x) = h(x')$. (Note that here there is free choice of both inputs.)

Here and elsewhere, the terms "easy" and "computationally infeasible" (or "hard") are intentionally left without formal definition; it is intended they be interpreted relative to an understood frame of reference. "Easy" might mean polynomial time and space; or more practically, within a certain number of machine operations or time units – perhaps seconds or milliseconds. A more specific definition of "computationally infeasible" might involve super-polynomial effort; require effort far exceeding understood resources; specify a lower bound on the number of operations or memory required in terms of a specified security parameter; or specify the probability that a property is violated be exponentially small. The properties as defined above, however, suffice to allow practical definitions such as Definitions 9.3 and 9.4 below.

9.2 Note (*alternate terminology*) Alternate terms used in the literature are as follows: preimage resistant $\equiv$ *one-way* (cf. Definition 9.9); 2nd-preimage resistance $\equiv$ *weak collision resistance*; collision resistance $\equiv$ *strong collision resistance*.

For context, one motivation for each of the three major properties above is now given. Consider a digital signature scheme wherein the signature is applied to the hash-value $h(x)$ rather than the message x. Here h should be an MDC with 2nd-preimage resistance, otherwise, an adversary C may observe the signature of some party A on $h(x)$, then find an x' such that $h(x) = h(x')$, and claim that A has signed x'. If C is able to actually choose the message which A signs, then C need only find a collision pair (x, x') rather than the harder task of finding a second preimage of x; in this case, collision resistance is also necessary (cf. Remark 9.93). Less obvious is the requirement of preimage resistance for some public-key signature schemes; consider RSA (Chapter 11), where party A has public key

(e, n). C may choose a random value y, compute $z = y^e \bmod n$, and (depending on the particular RSA signature verification process used) claim that y is A's signature on z. This (existential) forgery may be of concern if C can find a preimage x such that $h(x) = z$, and for which x is of practical use.

9.3 Definition A *one-way hash function* (OWHF) is a hash function h as per Definition 9.1 (i.e., offering ease of computation and compression) with the following additional properties, as defined above: preimage resistance, 2nd-preimage resistance.

9.4 Definition A *collision resistant hash function* (CRHF) is a hash function h as per Definition 9.1 (i.e., offering ease of computation and compression) with the following additional properties, as defined above: 2nd-preimage resistance, collision resistance (cf. Fact 9.18).

Although in practice a CRHF almost always has the additional property of preimage resistance, for technical reasons (cf. Note 9.20) this property is not mandated in Definition 9.4.

9.5 Note (*alternate terminology for OWHF, CRHF*) Alternate terms used in the literature are as follows: OWHF $\equiv$ *weak one-way hash function* (but here preimage resistance is often not explicitly considered); CRHF $\equiv$ *strong one-way hash function*.

9.6 Example (*hash function properties*)

(i) A simple modulo-32 checksum (32-bit sum of all 32-bit words of a data string) is an easily computed function which offers compression, but is not preimage resistant.

(ii) The function $g(x)$ of Example 9.11 is preimage resistant but provides neither compression nor 2nd-preimage resistance.

(iii) Example 9.13 presents a function with preimage resistance and 2nd-preimage resistance (but not compression). □

9.7 Definition A *message authentication code (MAC)* algorithm is a family of functions h_k parameterized by a secret key k, with the following properties:

1. *ease of computation* — for a known function h_k, given a value k and an input x, $h_k(x)$ is easy to compute. This result is called the *MAC-value* or *MAC*.
2. *compression* — h_k maps an input x of arbitrary finite bitlength to an output $h_k(x)$ of fixed bitlength n.

   Furthermore, given a description of the function family h, for every fixed allowable value of k (unknown to an adversary), the following property holds:
3. *computation-resistance* — given zero or more text-MAC pairs $(x_i, h_k(x_i))$, it is computationally infeasible to compute any text-MAC pair $(x, h_k(x))$ for any new input $x \neq x_i$ (including possibly for $h_k(x) = h_k(x_i)$ for some i).

If computation-resistance does not hold, a MAC algorithm is subject to *MAC forgery*. While computation-resistance implies the property of *key non-recovery* (it must be computationally infeasible to recover k, given one or more text-MAC pairs $(x_i, h_k(x_i))$ for that k), key non-recovery does not imply computation-resistance (a key need not always actually be recovered to forge new MACs).

9.8 Remark (*MAC resistance when key known*) Definition 9.7 does not dictate whether MACs need be preimage- and collision resistant for parties knowing the key k (as Fact 9.21 implies for parties without k).

(i) Objectives of adversaries vs. MDCs

The objective of an adversary who wishes to "attack" an MDC is as follows:

(a) to attack a OWHF: given a hash-value y, find a preimage x such that $y = h(x)$; or given one such pair $(x, h(x))$, find a second preimage x' such that $h(x') = h(x)$.
(b) to attack a CRHF: find any two inputs x, x', such that $h(x') = h(x)$.

A CRHF must be designed to withstand standard birthday attacks (see Fact 9.33).

(ii) Objectives of adversaries vs. MACs

The corresponding objective of an adversary for a MAC algorithm is as follows:

(c) to attack a MAC: without prior knowledge of a key k, compute a new text-MAC pair $(x, h_k(x))$ for some text $x \neq x_i$, given one or more pairs $(x_i, h_k(x_i))$.

Computation-resistance here should hold whether the texts x_i for which matching MACs are available are given to the adversary, or may be freely chosen by the adversary. Similar to the situation for signature schemes, the following attack scenarios thus exist for MACs, for adversaries with increasing advantages:

1. *known-text attack*. One or more text-MAC pairs $(x_i, h_k(x_i))$ are available.
2. *chosen-text attack*. One or more text-MAC pairs $(x_i, h_k(x_i))$ are available for x_i chosen by the adversary.
3. *adaptive chosen-text attack*. The x_i may be chosen by the adversary as above, now allowing successive choices to be based on the results of prior queries.

As a certificational checkpoint, MACs should withstand adaptive chosen-text attack regardless of whether such an attack may actually be mounted in a particular environment. Some practical applications may limit the number of interactions allowed over a fixed period of time, or may be designed so as to compute MACs only for inputs created within the application itself; others may allow access to an unlimited number of text-MAC pairs, or allow MAC verification of an unlimited number of messages and accept any with a correct MAC for further processing.

(iii) Types of forgery (selective, existential)

When MAC forgery is possible (implying the MAC algorithm has been technically defeated), the severity of the practical consequences may differ depending on the degree of control an adversary has over the value x for which a MAC may be forged. This degree is differentiated by the following classification of forgeries:

1. *selective forgery* – attacks whereby an adversary is able to produce a new text-MAC pair for a text of his choice (or perhaps partially under his control). Note that here the selected value is the text for which a MAC is forged, whereas in a chosen-text attack the chosen value is the text of a text-MAC pair used for analytical purposes (e.g., to forge a MAC on a distinct text).
2. *existential forgery* – attacks whereby an adversary is able to produce a new text-MAC pair, but with no control over the value of that text.

Key recovery of the MAC key itself is the most damaging attack, and trivially allows selective forgery. MAC forgery allows an adversary to have a forged text accepted as authentic. The consequences may be severe even in the existential case. A classic example is the replacement of a monetary amount known to be small by a number randomly distributed between 0 and $2^{32} - 1$. For this reason, messages whose integrity or authenticity is to be verified are often constrained to have pre-determined structure or a high degree of verifiable redundancy, in an attempt to preclude meaningful attacks.

Analogously to MACs, attacks on MDC schemes (primarily 2nd-preimage and collision attacks) may be classified as selective or existential. If the message can be partially controlled, then the attack may be classified as partially selective (e.g., see §9.7.1(iii)).

9.2.3 Hash properties required for specific applications

Because there may be costs associated with specific properties – e.g., CRHFs are in general harder to construct than OWHFs and have hash-values roughly twice the bitlength – it should be understood which properties are actually required for particular applications, and why. Selected techniques whereby hash functions are used for data integrity, and the corresponding properties required thereof by these applications, are summarized in Table 9.1.

In general, an MDC should be a CRHF if an untrusted party has control over the exact content of hash function inputs (see Remark 9.93); a OWHF suffices otherwise, including the case where there is only a single party involved (e.g., a store-and-retrieve application). Control over precise format of inputs may be eliminated by introducing into the message randomization that is uncontrollable by one or both parties. Note, however, that data integrity techniques based on a shared secret key typically involve mutual trust and do not address non-repudiation; in this case, collision resistance may or may not be a requirement.

Hash properties required → Integrity application ↓	Preimage resistant	2nd-preimage	Collision resistant	Details
MDC + asymmetric signature	yes	yes	yes†	page 324
MDC + authentic channel		yes	yes†	page 364
MDC + symmetric encryption				page 365
hash for one-way password file	yes			page 389
MAC (key unknown to attacker)	yes	yes	yes†	page 326
MAC (key known to attacker)		yes‡		page 325

Table 9.1: *Resistance properties required for specified data integrity applications.*
†Resistance required if attacker is able to mount a chosen message attack.
‡Resistance required in rare case of multi-cast authentication (see page 378).

9.2.4 One-way functions and compression functions

Related to Definition 9.3 of a OWHF is the following, which is unrestrictive with respect to a compression property.

9.9 Definition A *one-way function* (OWF) is a function f such that for each x in the domain of f, it is easy to compute $f(x)$; but for essentially all y in the range of f, it is computationally infeasible to find any x such that $y = f(x)$.

9.10 Remark (*OWF vs. domain-restricted OWHF*) A OWF as defined here differs from a OWHF with domain restricted to fixed-size inputs in that Definition 9.9 does not require 2nd-preimage resistance. Many one-way functions are, in fact, non-compressing, in which case most image elements have unique preimages, and for these 2nd-preimage resistance holds vacuously – making the difference minor (but see Example 9.11).

9.11 Example (*one-way functions and modular squaring*) The squaring of integers modulo a prime p, e.g., $f(x) = x^2 - 1 \bmod p$, behaves in many ways like a random mapping. However, $f(x)$ is not a OWF because finding square roots modulo primes is easy (§3.5.1). On the other hand, $g(x) = x^2 \bmod n$ is a OWF (Definition 9.9) for appropriate randomly chosen primes p and q where $n = pq$ and the factorization of n is unknown, as finding a preimage (i.e., computing a square root mod n) is computationally equivalent to factoring (Fact 3.46) and thus intractable. Nonetheless, finding a 2nd-preimage, and, therefore, collisions, is trivial (given x, $-x$ yields a collision), and thus g fits neither the definition of a OWHF nor a CRHF with domain restricted to fixed-size inputs. □

9.12 Remark (*candidate one-way functions*) There are, in fact, no known instances of functions which are provably one-way (with no assumptions); indeed, despite known hash function constructions which are provably as secure as **NP**-complete problems, there is no assurance the latter are difficult. All instances of "one-way functions" to date should thus more properly be qualified as "conjectured" or "candidate" one-way functions. (It thus remains possible, although widely believed most unlikely, that one-way functions do not exist.) A proof of existence would establish $\mathbf{P} \neq \mathbf{NP}$, while non-existence would have devastating cryptographic consequences (see page 377), although not directly implying $\mathbf{P} = \mathbf{NP}$.

Hash functions are often used in applications (cf. §9.2.6) which require the one-way property, but not compression. It is, therefore, useful to distinguish three classes of functions (based on the relative size of inputs and outputs):

1. *(general) hash functions*. These are functions as per Definition 9.1, typically with additional one-way properties, which compress arbitrary-length inputs to n-bit outputs.
2. *compression functions* (fixed-size hash functions). These are functions as per Definition 9.1, typically with additional one-way properties, but with domain restricted to fixed-size inputs – i.e., compressing m-bit inputs to n-bit outputs, $m > n$.
3. *non-compressing one-way functions*. These are fixed-size hash functions as above, except that $n = m$. These include *one-way permutations*, and can be more explicitly described as computationally non-invertible functions.

9.13 Example (*DES-based OWF*) A one-way function can be constructed from DES or any block cipher E which behaves essentially as a random function (see Remark 9.14), as follows: $f(x) = E_k(x) \oplus x$, for any fixed known key k. The one-way nature of this construction can be proven under the assumption that E is a random permutation. An intuitive argument follows. For any choice of y, finding any x (and key k) such that $E_k(x) \oplus x = y$ is difficult because for any chosen x, $E_k(x)$ will be essentially random (for any key k) and thus so will $E_k(x) \oplus x$; hence, this will equal y with no better than random chance. By similar reasoning, if one attempts to use decryption and chooses an x, the probability that $E_k^{-1}(x \oplus y) = x$ is no better than random chance. Thus $f(x)$ appears to be a OWF. While $f(x)$ is not a OWHF (it handles only fixed-length inputs), it can be extended to yield one (see Algorithm 9.41). □

9.14 Remark (*block ciphers and random functions*) Regarding random functions and their properties, see §2.1.6. If a block cipher behaved as a random function, then encryption and decryption would be equivalent to looking up values in a large table of random numbers; for a fixed input, the mapping from a key to an output would behave as a random mapping. However, block ciphers such as DES are bijections, and thus at best exhibit behavior more like random permutations than random functions.

9.15 Example (*one-wayness w.r.t. two inputs*) Consider $f(x,k) = E_k(x)$, where E represents DES. This is not a one-way function of the joint input (x,k), because given any function value $y = f(x,k)$, one can choose any key k' and compute $x' = E_{k'}^{-1}(y)$ yielding a preimage (x',k'). Similarly, $f(x,k)$ is not a one-way function of x if k is known, as given $y = f(x,k)$ and k, decryption of y using k yields x. (However, a "black-box" which computes $f(x,k)$ for fixed, externally-unknown k is a one-way function of x.) In contrast, $f(x,k)$ is a one-way function of k; given $y = f(x,k)$ and x, it is not known how to find a preimage k in less than about 2^{55} operations. (This latter concept is utilized in one-time digital signature schemes – see §11.6.2.) □

9.16 Example (*OWF - multiplication of large primes*) For appropriate choices of primes p and q, $f(p,q) = pq$ is a one-way function: given p and q, computing $n = pq$ is easy, but given n, finding p and q, i.e., *integer factorization*, is difficult. RSA and many other cryptographic systems rely on this property (see Chapter 3, Chapter 8). Note that contrary to many one-way functions, this function f does not have properties resembling a "random" function. □

9.17 Example (*OWF - exponentiation in finite fields*) For most choices of appropriately large primes p and any element $\alpha \in \mathbb{Z}_p^*$ of sufficiently large multiplicative order (e.g., a generator), $f(x) = \alpha^x \bmod p$ is a one-way function. (For example, p must not be such that all the prime divisors of $p-1$ are small, otherwise the discrete log problem is feasible by the Pohlig-Hellman algorithm of §3.6.4.) $f(x)$ is easily computed given α, x, and p using the square-and-multiply technique (Algorithm 2.143), but for most choices p it is difficult, given (y,p,α), to find an x in the range $0 \le x \le p-2$ such that $\alpha^x \bmod p = y$, due to the apparent intractability of the discrete logarithm problem (§3.6). Of course, for specific values of $f(x)$ the function can be inverted trivially. For example, the respective preimages of 1 and -1 are known to be 0 and $(p-1)/2$, and by computing $f(x)$ for any small set of values for x (e.g., $x = 1, 2, \ldots, 10$), these are also known. However, for *essentially* all y in the range, the preimage of y is difficult to find. □

9.2.5 Relationships between properties

In this section several relationships between the hash function properties stated in the preceding section are examined.

9.18 Fact Collision resistance implies 2nd-preimage resistance of hash functions.

Justification. Suppose h has collision resistance. Fix an input x_j. If h does not have 2nd-preimage resistance, then it is feasible to find a distinct input x_i such that $h(x_i) = h(x_j)$, in which case (x_i, x_j) is a pair of distinct inputs hashing to the same output, contradicting collision resistance.

9.19 Remark (*one-way vs. preimage and 2nd-preimage resistant*) While the term "one-way" is generally taken to mean preimage resistant, in the hash function literature it is sometimes also used to imply that a function is 2nd-preimage resistant or computationally non-invertible. (*Computationally non-invertible* is a more explicit term for preimage resistance when preimages are unique, e.g., for one-way permutations. In the case that two or more preimages exist, a function fails to be computationally non-invertible if any one can be found.) This causes ambiguity as 2nd-preimage resistance does not guarantee preimage-resistance (Note 9.20), nor does preimage resistance guarantee 2nd-preimage resistance (Example 9.11); see also Remark 9.10. An attempt is thus made to avoid unqualified use of the term "one-way".

9.20 Note (*collision resistance does not guarantee preimage resistance*) Let g be a hash function which is collision resistant and maps arbitrary-length inputs to n-bit outputs. Consider the function h defined as (here and elsewhere, || denotes concatenation):

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

Then h is an $(n+1)$-bit hash function which is collision resistant but not preimage resistant. As a simpler example, the identity function on fixed-length inputs is collision and 2nd-preimage resistant (preimages are unique) but not preimage resistant. While such pathological examples illustrate that collision resistance does not guarantee the difficulty of finding preimages of specific (or even most) hash outputs, for most CRHFs arising in practice it nonetheless appears reasonable to assume that collision resistance does indeed imply preimage resistance.

9.21 Fact (*implications of MAC properties*) Let h_k be a keyed hash function which is a MAC algorithm per Definition 9.7 (and thus has the property of computation-resistance). Then h_k is, against chosen-text attack by an adversary without knowledge of the key k, (i) both 2nd-preimage resistant and collision resistant; and (ii) preimage resistant (with respect to the hash-input).

Justification. For (i), note that computation-resistance implies hash-results should not even be computable by those without secret key k. For (ii), by way of contradiction, assume h were not preimage resistant. Then recovery of the preimage x for a randomly selected hash-output y violates computation-resistance.

9.2.6 Other hash function properties and applications

Most unkeyed hash functions commonly found in practice were originally designed for the purpose of providing data integrity (see §9.6), including digital fingerprinting of messages in conjunction with digital signatures (§9.6.4). The majority of these are, in fact, MDCs designed to have preimage, 2nd-preimage, or collision resistance properties. Because one-way functions are a fundamental cryptographic primitive, many of these MDCs, which typically exhibit behavior informally equated with one-wayness and randomness, have been proposed for use in various applications distinct from data integrity, including, as discussed below:

1. *confirmation of knowledge*
2. *key derivation*
3. *pseudorandom number generation*

Hash functions used for confirmation of knowledge facilitate commitment to data values, or demonstrate possession of data, without revealing such data itself (until possibly a later point in time); verification is possible by parties in possession of the data. This resembles the use of MACs where one also essentially demonstrates knowledge of a secret (but with the demonstration bound to a specific message). The property of hash functions required is preimage resistance (see also partial-preimage resistance below). Specific examples include use in password verification using unencrypted password-image files (Chapter 10); symmetric-key digital signatures (Chapter 11); key confirmation in authenticated key establishment protocols (Chapter 12); and document-dating or timestamping by hash-code registration (Chapter 13).

In general, use of hash functions for purposes other than which they were originally designed requires caution, as such applications may require additional properties (see below) these functions were not designed to provide; see Remark 9.22. Unkeyed hash functions having properties associated with one-way functions have nonetheless been proposed for a wide range of applications, including as noted above:

- *key derivation* – to compute sequences of new keys from prior keys (Chapter 13). A primary example is key derivation in point-of-sale (POS) terminals; here an important requirement is that the compromise of currently active keys must not compromise the security of previous transaction keys. A second example is in the generation of one-time password sequences based on one-way functions (Chapter 10).
- *pseudorandom number generation* – to generate sequences of numbers which have various properties of randomness. (A pseudorandom number generator can be used to construct a symmetric-key block cipher, among other things.) Due to the difficulty of producing cryptographically strong pseudorandom numbers (see Chapter 5), MDCs should not be used for this purpose unless the randomness requirements are clearly understood, and the MDC is verified to satisfy these.

For the applications immediately above, rather than hash functions, the cryptographic primitive which is needed may be a *pseudorandom function* (or keyed pseudorandom function).

9.22 Remark (*use of MDCs*) Many MDCs used in practice may appear to satisfy additional requirements beyond those for which they were originally designed. Nonetheless, the use of arbitrary hash functions cannot be recommended for any applications without careful analysis precisely identifying both the critical properties required by the application and those provided by the function in question (cf. §9.5.2).

Additional properties of one-way hash functions

Additional properties of one-way hash functions called for by the above-mentioned applications include the following.

1. *non-correlation*. Input bits and output bits should not be correlated. Related to this, an avalanche property similar to that of good block ciphers is desirable whereby every input bit affects every output bit. (This rules out hash functions for which preimage resistance fails to imply 2nd-preimage resistance simply due to the function effectively ignoring a subset of input bits.)
2. *near-collision resistance*. It should be hard to find any two inputs x, x' such that $h(x)$ and $h(x')$ differ in only a small number of bits.
3. *partial-preimage resistance* or *local one-wayness*. It should be as difficult to recover any substring as to recover the entire input. Moreover, even if part of the input is known, it should be difficult to find the remainder (e.g., if t input bits remain unknown, it should take on average 2^{t-1} hash operations to find these bits.)

Partial preimage resistance is an implicit requirement in some of the proposed applications of §9.5.2. One example where near-collision resistance is necessary is when only half of the output bits of a hash function are used.

Many of these properties can be summarized as requirements that there be neither local nor global statistical weaknesses; the hash function should not be weaker with respect to some parts of its input or output than others, and all bits should be equally hard. Some of these may be called *certificational properties* – properties which intuitively appear desirable, although they cannot be shown to be directly necessary.

9.3 Basic constructions and general results

9.3.1 General model for iterated hash functions

Most unkeyed hash functions h are designed as iterative processes which hash arbitrary-length inputs by processing successive fixed-size blocks of the input, as illustrated in Figure 9.2.

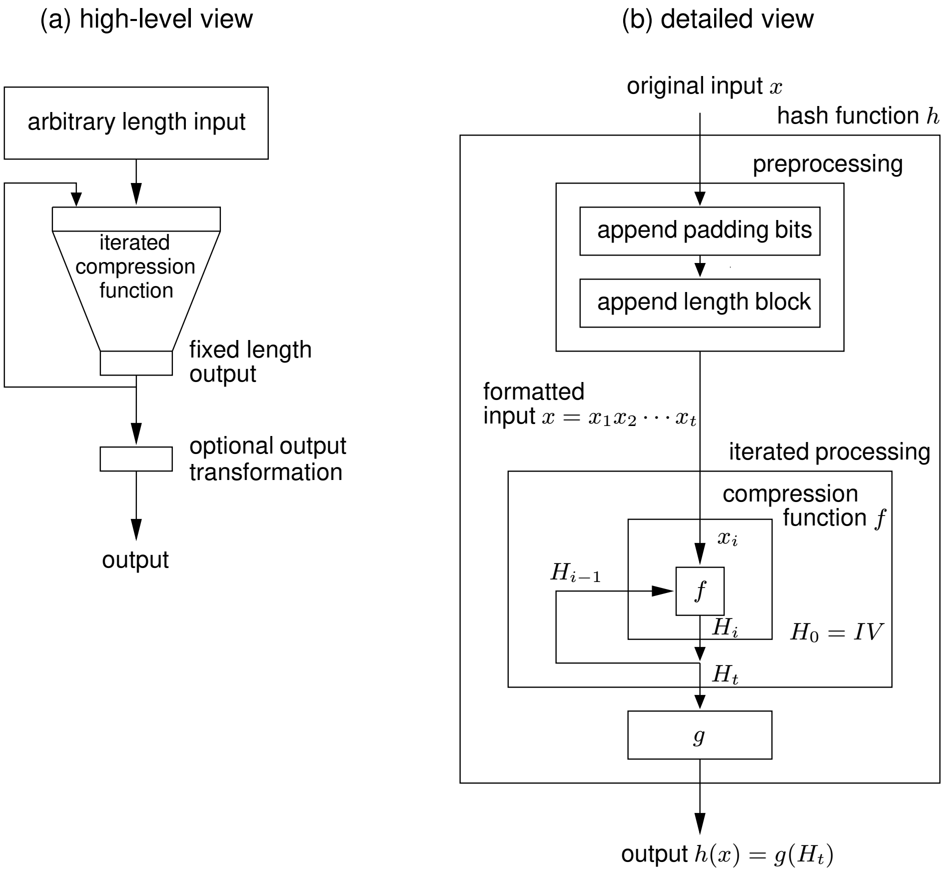

Figure 9.2: *General model for an iterated hash function.*

A hash input x of arbitrary finite length is divided into fixed-length r-bit blocks x_i. This preprocessing typically involves appending extra bits (*padding*) as necessary to attain an overall bitlength which is a multiple of the blocklength r, and often includes (for security reasons – e.g., see Algorithm 9.26) a block or partial block indicating the bitlength of the unpadded input. Each block x_i then serves as input to an internal fixed-size hash function f, the *compression function* of h, which computes a new intermediate result of bitlength n for some fixed n, as a function of the previous n-bit intermediate result and the next input block x_i. Letting H_i denote the partial result after stage i, the general process for an iterated hash function with input $x = x_1x_2 \dots x_t$ can be modeled as follows:

$$H_0 = IV; \qquad H_i = f(H_{i-1}, x_i),\ \ 1 \le i \le t; \qquad h(x) = g(H_t). \qquad (9.1)$$

H_{i-1} serves as the n-bit *chaining variable* between stage $i-1$ and stage i, and H_0 is a pre-defined starting value or *initializing value* (IV). An optional output transformation g (see Figure 9.2) is used in a final step to map the n-bit chaining variable to an m-bit result $g(H_t)$; g is often the identity mapping $g(H_t) = H_t$.

Particular hash functions are distinguished by the nature of the preprocessing, compression function, and output transformation.

9.3.2 General constructions and extensions

To begin, an example demonstrating an insecure construction is given. Several secure general constructions are then discussed.

9.23 Example (*insecure trivial extension of OWHF to CRHF*) In the case that an iterated OWHF h yielding n-bit hash-values is not collision resistant (e.g., when a $2^{n/2}$ birthday collision attack is feasible – see §9.7.1) one might propose constructing from h a CRHF using as output the concatenation of the last two n-bit chaining variables, so that a t-block message has hash-value $H_{t-1}||H_t$ rather than H_t. This is insecure as the final message block x_t can be held fixed along with H_t, reducing the problem to finding a collision on H_{t-1} for h. □

Extending compression functions to hash functions

Fact 9.24 states an important relationship between collision resistant compression functions and collision resistant hash functions. Not only can the former be extended to the latter, but this can be done efficiently using Merkle's meta-method of Algorithm 9.25 (also called the Merkle-Damgård construction). This reduces the problem of finding such a hash function to that of finding such a compression function.

9.24 Fact (*extending compression functions*) Any compression function f which is collision resistant can be extended to a collision resistant hash function h (taking arbitrary length inputs).

9.25 Algorithm Merkle's meta-method for hashing

INPUT: compression function f which is collision resistant.
OUTPUT: unkeyed hash function h which is collision resistant.

1. Suppose f maps $(n+r)$-bit inputs to n-bit outputs (for concreteness, consider $n = 128$ and $r = 512$). Construct a hash function h from f, yielding n-bit hash-values, as follows.
2. Break an input x of bitlength b into blocks $x_1x_2 \dots x_t$ each of bitlength r, padding out the last block x_t with 0-bits if necessary.
3. Define an extra final block x_{t+1}, the length-block, to hold the right-justified binary representation of b (presume that $b < 2^r$).
4. Letting 0^j represent the bitstring of j 0's, define the n-bit hash-value of x to be $h(x) = H_{t+1} = f(H_t \mid\mid x_{t+1})$ computed from:

$$H_0 = 0^n; \qquad H_i = f(H_{i-1} \mid\mid x_i), \quad 1 \le i \le t+1.$$

The proof that the resulting function h is collision resistant follows by a simple argument that a collision for h would imply a collision for f for some stage i. The inclusion of the length-block, which effectively encodes all messages such that no encoded input is the tail end of any other encoded input, is necessary for this reasoning. Adding such a length-block is sometimes called Merkle-Damgård strengthening (*MD-strengthening*), which is now stated separately for future reference.

9.26 Algorithm MD-strengthening

Before hashing a message $x = x_1x_2 \dots x_t$ (where x_i is a block of bitlength r appropriate for the relevant compression function) of bitlength b, append a final length-block, x_{t+1}, containing the (say) right-justified binary representation of b. (This presumes $b < 2^r$.)

Cascading hash functions

9.27 Fact (*cascading hash functions*) If *either* h_1 or h_2 is a collision resistant hash function, then $h(x) = h_1(x) \,||\, h_2(x)$ is a collision resistant hash function.

If both h_1 and h_2 in Fact 9.27 are n-bit hash functions, then h produces $2n$-bit outputs; mapping this back down to an n-bit output by an n-bit collision-resistant hash function (h_1 and h_2 are candidates) would leave the overall mapping collision-resistant. If h_1 and h_2 are independent, then finding a collision for h requires finding a collision for both simultaneously (i.e., on the same input), which one could hope would require the product of the efforts to attack them individually. This provides a simple yet powerful way to (almost surely) increase strength using only available components.

9.3.3 Formatting and initialization details

9.28 Note (*data representation*) As hash-values depend on exact bitstrings, different data representations (e.g., ASCII vs. EBCDIC) must be converted to a common format before computing hash-values.

(i) Padding and length-blocks

For block-by-block hashing methods, extra bits are usually appended to a hash input string before hashing, to pad it out to a number of bits which make it a multiple of the relevant block size. The padding bits need not be transmitted/stored themselves, provided the sender and recipient agree on a convention.

9.29 Algorithm Padding Method 1

INPUT: data x; bitlength n giving blocksize of data input to processing stage.
OUTPUT: padded data x', with bitlength a multiple of n.

1. Append to x as few (possibly zero) 0-bits as necessary to obtain a string x' whose bitlength is a multiple of n.

9.30 Algorithm Padding Method 2

INPUT: data x; bitlength n giving blocksize of data input to processing stage.
OUTPUT: padded data x', with bitlength a multiple of n.

1. Append to x a single 1-bit.

2. Then append as few (possibly zero) 0-bits as necessary to obtain a string x' whose bitlength is a multiple of n.

9.31 Remark (*ambiguous padding*) Padding Method 1 is *ambiguous* – trailing 0-bits of the original data cannot be distinguished from those added during padding. Such methods are acceptable if the length of the data (before padding) is known by the recipient by other means. Padding Method 2 is not ambiguous – each padded string x' corresponds to a unique unpadded string x. When the bitlength of the original data x is already a multiple of n, Padding Method 2 results in the creation of an extra block.

9.32 Remark (*appended length blocks*) Appending a logical length-block prior to hashing prevents collision and pseudo-collision attacks which find second messages of different length, including trivial collisions for random IVs (Example 9.96), long-message attacks (Fact 9.37), and fixed-point attacks (page 374). This further justifies the use of MD-strengthening (Algorithm 9.26).

Trailing length-blocks and padding are often combined. For Padding Method 2, a length field of pre-specified bitlength w may replace the final w 0-bits padded if padding would otherwise cause w or more redundant such bits. By pre-agreed convention, the length field typically specifies the bitlength of the original message. (If used instead to specify the number of padding bits appended, deletion of leading blocks cannot be detected.)

(ii) IVs

Whether the IV is fixed, is randomly chosen per hash function computation, or is a function of the data input, the same IV must be used to generate and verify a hash-value. If not known *a priori* by the verifier, it must be transferred along with the message. In the latter case, this generally should be done with guaranteed integrity (to cut down on the degree of freedom afforded to adversaries, in line with the principle that hash functions should be defined with a fixed or a small set of allowable IVs).

9.3.4 Security objectives and basic attacks

As a framework for evaluating the computational security of hash functions, the objectives of both the hash function designer and an adversary should be understood. Based on Definitions 9.3, 9.4, and 9.7, these are summarized in Table 9.2, and discussed below.

Hash type	Design goal	Ideal strength	Adversary's goal
OWHF	preimage resistance;	2^n	produce preimage;
	2nd-preimage resistance	2^n	find 2nd input, same image
CRHF	collision resistance	$2^{n/2}$	produce any collision
MAC	key non-recovery;	2^t	deduce MAC key;
	computation resistance	$P_f = \max(2^{-t}, 2^{-n})$	produce new (msg, MAC)

Table 9.2: *Design objectives for n-bit hash functions (t-bit MAC key). P_f denotes the probability of forgery by correctly guessing a MAC.*

Given a specific hash function, it is desirable to be able to prove a lower bound on the complexity of attacking it under specified scenarios, with as few or weak a set of assumptions as possible. However, such results are scarce. Typically the best guidance available regarding the security of a particular hash function is the complexity of the (most efficient) applicable known attack, which gives an *upper* bound on security. An attack of *complexity* 2^t is one which requires approximately 2^t operations, each being an appropriate unit of work (e.g., one execution of the compression function or one encryption of an underlying cipher). The storage complexity of an attack (i.e., storage required) should also be considered.

(i) Attacks on the bitsize of an MDC

Given a fixed message x with n-bit hash $h(x)$, a naive method for finding an input colliding with x is to pick a random bitstring x' (of bounded bitlength) and check if $h(x') = h(x)$. The cost may be as little as one compression function evaluation, and memory is negligible. Assuming the hash-code approximates a uniform random variable, the probability of a match is 2^{-n}. The implication of this is Fact 9.33, which also indicates the effort required to find collisions if x may itself be chosen freely. Definition 9.34 is motivated by the design goal that the best possible attack should require no less than such levels of effort, i.e., essentially brute force.

9.33 Fact (*basic hash attacks*) For an n-bit hash function h, one may expect a guessing attack to find a preimage or second preimage within 2^n hashing operations. For an adversary able to choose messages, a birthday attack (see §9.7.1) allows colliding pairs of messages x, x' with $h(x) = h(x')$ to be found in about $2^{n/2}$ operations, and negligible memory.

9.34 Definition An n-bit unkeyed hash function has *ideal security* if both: (1) given a hash output, producing each of a preimage and a 2nd-preimage requires approximately 2^n operations; and (2) producing a collision requires approximately $2^{n/2}$ operations.

(ii) Attacks on the MAC key space

An attempt may be made to determine a MAC key using exhaustive search. With a single known text-MAC pair, an attacker may compute the n-bit MAC on that text under all possible keys, and then check which of the computed MAC-values agrees with that of the known pair. For a t-bit key space this requires 2^t MAC operations, after which one expects $1 + 2^{t-n}$ candidate keys remain. Assuming the MAC behaves as a random mapping, it can be shown that one can expect to reduce this to a unique key by testing the candidate keys using just over t/n text-MAC pairs. Ideally, a MAC key (or information of cryptographically equivalent value) would not be recoverable in fewer than 2^t operations.

As a probabilistic attack on the MAC key space distinct from key recovery, note that for a t-bit key and a fixed input, a randomly guessed key will yield a correct (n-bit) MAC with probability $\approx 2^{-t}$ for $t < n$.

(iii) Attacks on the bitsize of a MAC

MAC forgery involves producing any input x and the corresponding correct MAC without having obtained the latter from anyone with knowledge of the key. For an n-bit MAC algorithm, either guessing a MAC for a given input, or guessing a preimage for a given MAC output, has probability of success about 2^{-n}, as for an MDC. A difference here, however, is that guessed MAC-values cannot be verified off-line without known text-MAC pairs – either knowledge of the key, or a "black-box" which provides MACs for given inputs (i.e., a chosen-text scenario) is required. Since recovering the MAC key trivially allows forgery, an attack on the t-bit key space (see above) must be also be considered here. Ideally, an adversary would be unable to produce new (correct) text-MAC pairs (x, y) with probability significantly better than $\max(2^{-t}, 2^{-n})$, i.e., the better of guessing a key or a MAC-value.

(iv) Attacks using precomputations, multiple targets, and long messages

9.35 Remark (*precomputation of hash values*) For both preimage and second preimage attacks, an opponent who precomputes a large number of hash function input-output pairs may trade off precomputation plus storage for subsequent attack time. For example, for a 64-bit hash value, if one randomly selects 2^{40} inputs, then computes their hash values and stores (hash value, input) pairs indexed by hash value, this precomputation of $O(2^{40})$ time and space allows an adversary to increase the probability of finding a preimage (per one subsequent hash function computation) from 2^{-64} to 2^{-24}. Similarly, the probability of finding a second preimage increases to r times its original value (when no stored pairs are known) if r input-output pairs of a OWHF are precomputed and tabulated.

9.36 Remark (*effect of parallel targets for OWHFs*) In a basic attack, an adversary seeks a second preimage for one fixed target (the image computed from a first preimage). If there are r targets and the goal is to find a second preimage for any one of these r, then the probability of success increases to r times the original probability. One implication is that when using hash functions in conjunction with keyed primitives such as digital signatures, repeated use of the keyed primitive may weaken the security of the combined mechanism in the following sense. If r signed messages are available, the probability of a hash collision increases r-fold (cf. Remark 9.35), and colliding messages yield equivalent signatures, which an opponent could not itself compute off-line.

Fact 9.37 reflects a related attack strategy of potential concern when using iterated hash functions on long messages.

9.37 Fact (*long-message attack for 2nd-preimage*) Let h be an iterated n-bit hash function with compression function f (as in equation (9.1), without MD-strengthening). Let x be a message consisting of t blocks. Then a 2nd-preimage for $h(x)$ can be found in time $(2^n/s)+s$ operations of f, and in space $n(s+\lg(s))$ bits, for any s in the range $1 \le s \le \min(t, 2^{n/2})$.

Justification. The idea is to use a birthday attack on the intermediate hash-results; a sketch for the choice $s = t$ follows. Compute $h(x)$, storing (H_i, i) for each of the t intermediate hash-results H_i corresponding to the t input blocks x_i in a table such that they may be later indexed by value. Compute $h(z)$ for random choices z, checking for a collision involving $h(z)$ in the table, until one is found; approximately $2^n/s$ values z will be required, by the birthday paradox. Identify the index j from the table responsible for the collision; the input $zx_{j+1}x_{j+2}\ldots x_t$ then collides with x.

9.38 Note (*implication of long messages*) Fact 9.37 implies that for "long" messages, a 2nd-preimage is generally easier to find than a preimage (the latter takes at most 2^n operations), becoming moreso with the length of x. For $t \ge 2^{n/2}$, computation is minimized by choosing $s = 2^{n/2}$ in which case a 2nd-preimage costs about $2^{n/2}$ executions of f (comparable to the difficulty of finding a collision).

9.3.5 Bitsizes required for practical security

Suppose that a hash function produces n-bit hash-values, and as a representative benchmark assume that 2^{80} (but not fewer) operations is acceptably beyond computational feasibility.[2] Then the following statements may be made regarding n.

[2]Circa 1996, 2^{40} simple operations is quite feasible, and 2^{56} is considered quite reachable by those with sufficient motivation (possibly using parallelization or customized machines).

1. For a OWHF, $n \geq 80$ is required. Exhaustive off-line attacks require at most 2^n operations; this may be reduced with precomputation (Remark 9.35).
2. For a CRHF, $n \geq 160$ is required. Birthday attacks are applicable (Fact 9.33).
3. For a MAC, $n \geq 64$ along with a MAC key of 64-80 bits is sufficient for most applications and environments (cf. Table 9.1). If a single MAC key remains in use, off-line attacks may be possible given one or more text-MAC pairs; but for a proper MAC algorithm, preimage and 2nd-preimage resistance (as well as collision resistance) should follow directly from lack of knowledge of the key, and thus security with respect to such attacks should depend on the keysize rather than n. For attacks requiring on-line queries, additional controls may be used to limit the number of such queries, constrain the format of MAC inputs, or prevent disclosure of MAC outputs for random (chosen-text) inputs. Given special controls, values as small as $n = 32$ or 40 may be acceptable; but caution is advised, since even with one-time MAC keys, the chance any randomly guessed MAC being correct is 2^{-n}, and the relevant factors are the total number of trials a system is subject to over its lifetime, and the consequences of a single successful forgery.

These guidelines may be relaxed somewhat if a lower threshold of computational infeasibility is assumed (e.g., 2^{64} instead of 2^{80}). However, an additional consideration to be taken into account is that for both a CRHF and a OWHF, not only can off-line attacks be carried out, but these can typically be parallelized. Key search attacks against MACs may also be parallelized.

9.4 Unkeyed hash functions (MDCs)

A move from general properties and constructions to specific hash functions is now made, and in this section the subclass of unkeyed hash functions known as modification detection codes (MDCs) is considered. From a structural viewpoint, these may be categorized based on the nature of the operations comprising their internal compression functions. From this viewpoint, the three broadest categories of iterated hash functions studied to date are hash functions *based on block ciphers*, *customized hash functions*, and hash functions *based on modular arithmetic*. Customized hash functions are those designed specifically for hashing, with speed in mind and independent of other system subcomponents (e.g., block cipher or modular multiplication subcomponents which may already be present for non-hashing purposes).

Table 9.3 summarizes the conjectured security of a subset of the MDCs subsequently discussed in this section. Similar to the case of block ciphers for encryption (e.g. 8- or 12-round DES vs. 16-round DES), security of MDCs often comes at the expense of speed, and tradeoffs are typically made. In the particular case of block-cipher-based MDCs, a provably secure scheme of Merkle (see page 378) with rate 0.276 (see Definition 9.40) is known but little-used, while MDC-2 is widely believed to be (but not provably) secure, has rate $= 0.5$, and receives much greater attention in practice.

9.4.1 Hash functions based on block ciphers

A practical motivation for constructing hash functions from block ciphers is that if an efficient implementation of a block cipher is already available within a system (either in hardware or software), then using it as the central component for a hash function may provide

↓Hash function	n	m	Preimage	Collision	Comments
Matyas-Meyer-Oseas[a]	n	n	2^n	$2^{n/2}$	for keylength $= n$
MDC-2 (with DES)[b]	64	128	$2 \cdot 2^{82}$	$2 \cdot 2^{54}$	rate 0.5
MDC-4 (with DES)	64	128	2^{109}	$4 \cdot 2^{54}$	rate 0.25
Merkle (with DES)	106	128	2^{112}	2^{56}	rate 0.276
MD4	512	128	2^{128}	2^{20}	Remark 9.50
MD5	512	128	2^{128}	2^{64}	Remark 9.52
RIPEMD-128	512	128	2^{128}	2^{64}	–
SHA-1, RIPEMD-160	512	160	2^{160}	2^{80}	–

[a]The same strength is conjectured for Davies-Meyer and Miyaguchi-Preneel hash functions.
[b]Strength could be increased using a cipher with keylength equal to cipher blocklength.

Table 9.3: *Upper bounds on strength of selected hash functions. n-bit message blocks are processed to produce m-bit hash-values. Number of cipher or compression function operations currently believed necessary to find preimages and collisions are specified, assuming no underlying weaknesses for block ciphers (figures for MDC-2 and MDC-4 account for DES complementation and weak key properties). Regarding rate, see Definition 9.40.*

the latter functionality at little additional cost. The (not always well-founded) hope is that a good block cipher may serve as a building block for the creation of a hash function with properties suitable for various applications.

Constructions for hash functions have been given which are "provably secure" assuming certain ideal properties of the underlying block cipher. However, block ciphers do not possess the properties of random functions (for example, they are invertible – see Remark 9.14). Moreover, in practice block ciphers typically exhibit additional regularities or weaknesses (see §9.7.4). For example, for a block cipher E, double encryption using an encrypt-decrypt (E-D) cascade with keys K_1, K_2 results in the identity mapping when $K_1 = K_2$. In summary, while various necessary conditions are known, it is unclear exactly what requirements of a block cipher are sufficient to construct a secure hash function, and properties adequate for a block cipher (e.g., resistance to chosen-text attack) may not guarantee a good hash function.

In the constructions which follow, Definition 9.39 is used.

9.39 Definition An *(n,r) block cipher* is a block cipher defining an invertible function from n-bit plaintexts to n-bit ciphertexts using an r-bit key. If E is such a cipher, then $E_k(x)$ denotes the encryption of x under key k.

Discussion of hash functions constructed from n-bit block ciphers is divided between those producing *single-length* (n-bit) and *double-length* ($2n$-bit) hash-values, where single and double are relative to the size of the block cipher output. Under the assumption that computations of 2^{64} operations are infeasible,[3] the objective of single-length hash functions is to provide a OWHF for ciphers of blocklength near $n = 64$, or to provide CRHFs for cipher blocklengths near $n = 128$. The motivation for double-length hash functions is that many n-bit block ciphers exist of size approximately $n = 64$, and single-length hash-codes of this size are not collision resistant. For such ciphers, the goal is to obtain hash-codes of bitlength $2n$ which are CRHFs.

In the simplest case, the size of the key used in such hash functions is approximately the same as the blocklength of the cipher (i.e., n bits). In other cases, hash functions use

[3]The discussion here is easily altered for a more conservative bound, e.g., 2^{80} operations as used in §9.3.5. Here 2^{64} is more convenient for discussion, due to the omnipresence of 64-bit block ciphers.

larger (e.g., double-length) keys. Another characteristic to be noted in such hash functions is the number of block cipher operations required to produce a hash output of blocklength equal to that of the cipher, motivating the following definition.

9.40 Definition Let h be an iterated hash function constructed from a block cipher, with compression function f which performs s block encryptions to process each successive n-bit message block. Then the *rate* of h is $1/s$.

The hash functions discussed in this section are summarized in Table 9.4. The Matyas-Meyer-Oseas and MDC-2 algorithms are the basis, respectively, of the two generic hash functions in ISO standard 10118-2, each allowing use of any n-bit block cipher E and providing hash-codes of bitlength $m \leq n$ and $m \leq 2n$, respectively.

Hash function	(n,k,m)	Rate
Matyas-Meyer-Oseas	(n,k,n)	1
Davies-Meyer	(n,k,n)	k/n
Miyaguchi-Preneel	(n,k,n)	1
MDC-2 (with DES)	$(64,56,128)$	$1/2$
MDC-4 (with DES)	$(64,56,128)$	$1/4$

Table 9.4: *Summary of selected hash functions based on n-bit block ciphers. k = key bitsize (approximate); function yields m-bit hash-values.*

(i) Single-length MDCs of rate 1

The first three schemes described below, and illustrated in Figure 9.3, are closely related single-length hash functions based on block ciphers. These make use of the following predefined components:

1. a generic n-bit block cipher E_K parametrized by a symmetric key K;
2. a function g which maps n-bit inputs to keys K suitable for E (if keys for E are also of length n, g might be the identity function); and
3. a fixed (usually n-bit) initial value IV, suitable for use with E.

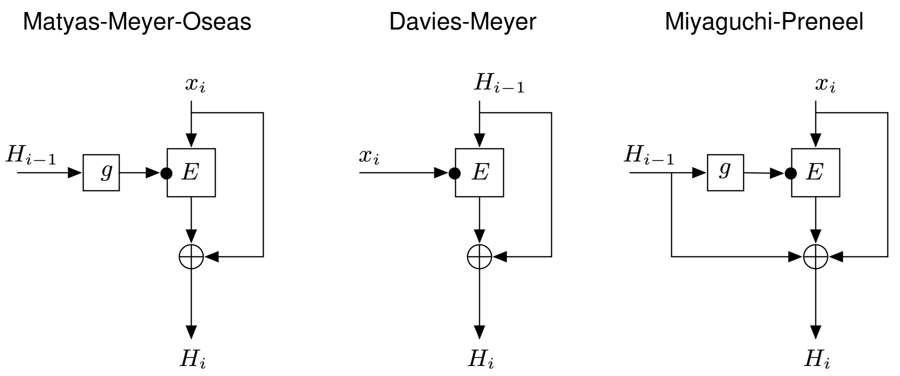

Figure 9.3: *Three single-length, rate-one MDCs based on block ciphers.*

9.41 Algorithm Matyas-Meyer-Oseas hash

INPUT: bitstring x.
OUTPUT: n-bit hash-code of x.

1. Input x is divided into n-bit blocks and padded, if necessary, to complete last block. Denote the padded message consisting of t n-bit blocks: $x_1x_2\dots x_t$. A constant n-bit initial value IV must be pre-specified.
2. The output is H_t defined by: $H_0 = IV$; $H_i = E_{g(H_{i-1})}(x_i)\oplus x_i,\ 1 \le i \le t$.

9.42 Algorithm Davies-Meyer hash

INPUT: bitstring x.
OUTPUT: n-bit hash-code of x.

1. Input x is divided into k-bit blocks where k is the keysize, and padded, if necessary, to complete last block. Denote the padded message consisting of t k-bit blocks: $x_1x_2 \dots x_t$. A constant n-bit initial value IV must be pre-specified.
2. The output is H_t defined by: $H_0 = IV$; $H_i = E_{x_i}(H_{i-1})\oplus H_{i-1},\ 1 \le i \le t$.

9.43 Algorithm Miyaguchi-Preneel hash

This scheme is identical to that of Algorithm 9.41, except the output H_{i-1} from the previous stage is also XORed to that of the current stage. More precisely, H_i is redefined as: $H_0 = IV$; $H_i = E_{g(H_{i-1})}(x_i)\oplus x_i \oplus H_{i-1},\ 1 \le i \le t$.

9.44 Remark (*dual schemes*) The Davies-Meyer hash may be viewed as the 'dual' of the Matyas-Meyer-Oseas hash, in the sense that x_i and H_{i-1} play reversed roles. When DES is used as the block cipher in Davies-Meyer, the input is processed in 56-bit blocks (yielding rate $56/64 < 1$), whereas Matyas-Meyer-Oseas and Miyaguchi-Preneel process 64-bit blocks.

9.45 Remark (*black-box security*) Aside from heuristic arguments as given in Example 9.13, it appears that all three of Algorithms 9.41, 9.42, and 9.43 yield hash functions which are provably secure under an appropriate "black-box" model (e.g., assuming E has the required randomness properties, and that attacks may not make use of any special properties or internal details of E). "Secure" here means that finding preimages and collisions (in fact, pseudo-preimages and pseudo-collisions – see §9.7.2) require on the order of 2^n and $2^{n/2}$ n-bit block cipher operations, respectively. Due to their single-length nature, none of these three is collision resistant for underlying ciphers of relatively small blocklength (e.g., DES, which yields 64-bit hash-codes).

Several double-length hash functions based on block ciphers are considered next.

(ii) Double-length MDCs: MDC-2 and MDC-4

MDC-2 and MDC-4 are manipulation detection codes requiring 2 and 4, respectively, block cipher operations per block of hash input. They employ a combination of either 2 or 4 iterations of the Matyas-Meyer-Oseas (single-length) scheme to produce a double-length hash. When used as originally specified, using DES as the underlying block cipher, they produce 128-bit hash-codes. The general construction, however, can be used with other block ciphers. MDC-2 and MDC-4 make use of the following pre-specified components:

1. DES as the block cipher E_K of bitlength $n = 64$ parameterized by a 56-bit key K;
2. two functions g and $\tilde{g}$ which map 64-bit values U to suitable 56-bit DES keys as follows. For $U = u_1u_2 \dots u_{64}$, delete every eighth bit starting with u_8, and set the 2nd and 3rd bits to '10' for g, and '01' for $\tilde{g}$:

$$g(U) = u_1\ 1\ 0\ u_4u_5u_6u_7u_9u_{10} \dots u_{63}.$$
$$\tilde{g}(U) = u_1\ 0\ 1\ u_4u_5u_6u_7u_9u_{10} \dots u_{63}.$$

(The resulting values are guaranteed not to be weak or semi-weak DES keys, as all such keys have bit 2 = bit 3; see page 375. Also, this guarantees the security requirement that $g(IV) \neq \tilde{g}(\widetilde{IV})$.)

MDC-2 is specified in Algorithm 9.46 and illustrated in Figure 9.4.

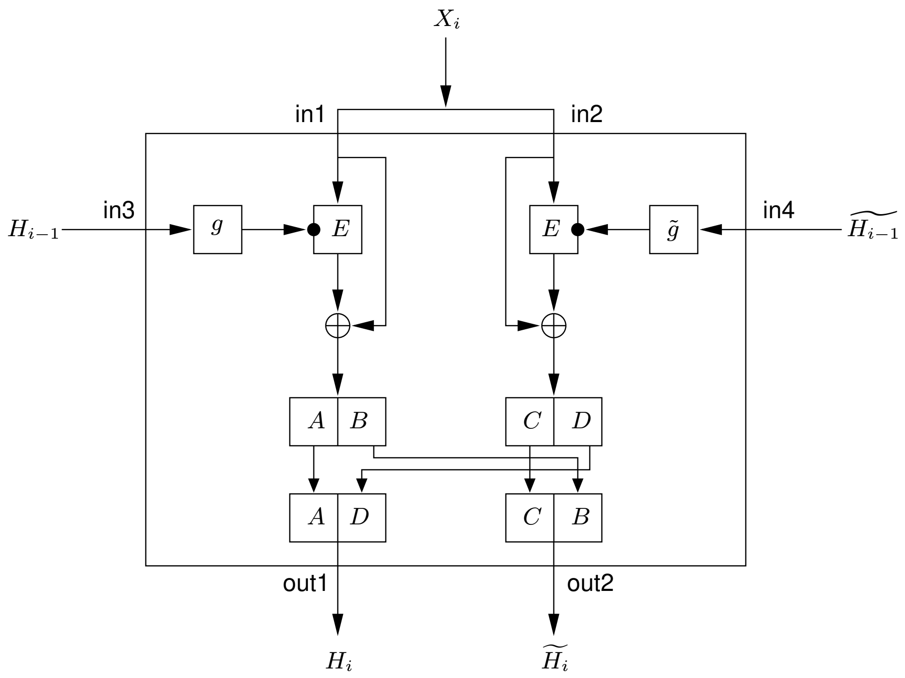

Figure 9.4: *Compression function of MDC-2 hash function. E = DES.*

9.46 Algorithm MDC-2 hash function (DES-based)

INPUT: string x of bitlength $r = 64t$ for $t \geq 2$.
OUTPUT: 128-bit hash-code of x.

1. Partition x into 64-bit blocks x_i: $x = x_1x_2 \dots x_t$.
2. Choose the 64-bit non-secret constants IV, $\widetilde{IV}$ (the same constants must be used for MDC verification) from a set of recommended prescribed values. A default set of prescribed values is (in hexadecimal): IV = `0x5252525252525252`, $\widetilde{IV}$ = `0x2525252525252525`.

3. Let $||$ denote concatenation, and C_i^L, C_i^R the left and right 32-bit halves of C_i. The output is $h(x) = H_t \mid\mid \widetilde{H_t}$ defined as follows (for $1 \le i \le t$):

$$H_0 = IV; \quad k_i = g(H_{i-1}); \quad C_i = E_{k_i}(x_i) \oplus x_i; \quad H_i = C_i^L \mid\mid \widetilde{C_i}^R$$
$$\widetilde{H_0} = \widetilde{IV}; \quad \widetilde{k_i} = \widetilde{g}(\widetilde{H_{i-1}}); \quad \widetilde{C_i} = E_{\widetilde{k_i}}(x_i) \oplus x_i; \quad \widetilde{H_i} = \widetilde{C_i}^L \mid\mid C_i^R .$$

In Algorithm 9.46, padding may be necessary to meet the bitlength constraint on the input x. In this case, an unambiguous padding method may be used (see Remark 9.31), possibly including MD-strengthening (see Remark 9.32).

MDC-4 (see Algorithm 9.47 and Figure 9.5) is constructed using the MDC-2 compression function. One iteration of the MDC-4 compression function consists of two sequential executions of the MDC-2 compression function, where:

1. the two 64-bit data inputs to the first MDC-2 compression are both the same next 64-bit message block;
2. the keys for the first MDC-2 compression are derived from the outputs (chaining variables) of the previous MDC-4 compression;
3. the keys for the second MDC-2 compression are derived from the outputs (chaining variables) of the first MDC-2 compression; and
4. the two 64-bit data inputs for the second MDC-2 compression are the outputs (chaining variables) from the opposite sides of the previous MDC-4 compression.

9.47 Algorithm MDC-4 hash function (DES-based)

INPUT: string x of bitlength $r = 64t$ for $t \ge 2$. (See MDC-2 above regarding padding.)
OUTPUT: 128-bit hash-code of x.

1. As in step 1 of MDC-2 above.
2. As in step 2 of MDC-2 above.
3. With notation as in MDC-2, the output is $h(x) = G_t \mid\mid \widetilde{G_t}$ defined as follows (for $1 \le i \le t$):

$$G_0 = IV; \quad \widetilde{G_0} = \widetilde{IV};$$
$$k_i = g(G_{i-1}); \quad C_i = E_{k_i}(x_i) \oplus x_i; \quad H_i = C_i^L \mid\mid \widetilde{C_i}^R$$
$$\widetilde{k_i} = \widetilde{g}(\widetilde{G_{i-1}}); \quad \widetilde{C_i} = E_{\widetilde{k_i}}(x_i) \oplus x_i; \quad \widetilde{H_i} = \widetilde{C_i}^L \mid\mid C_i^R$$
$$j_i = g(H_i); \quad D_i = E_{j_i}(\widetilde{G_{i-1}}) \oplus \widetilde{G_{i-1}}; \quad G_i = D_i^L \mid\mid \widetilde{D_i}^R$$
$$\widetilde{j_i} = \widetilde{g}(\widetilde{H_i}); \quad \widetilde{D_i} = E_{\widetilde{j_i}}(G_{i-1}) \oplus G_{i-1}; \quad \widetilde{G_i} = \widetilde{D_i}^L \mid\mid D_i^R .$$

9.4.2 Customized hash functions based on MD4

Customized hash functions are those which are specifically designed "from scratch" for the explicit purpose of hashing, with optimized performance in mind, and without being constrained to reusing existing system components such as block ciphers or modular arithmetic. Those having received the greatest attention in practice are based on the MD4 hash function.

Number 4 in a series of hash functions (*Message Digest* algorithms), MD4 was designed specifically for software implementation on 32-bit machines. Security concerns motivated the design of MD5 shortly thereafter, as a more conservative variation of MD4.

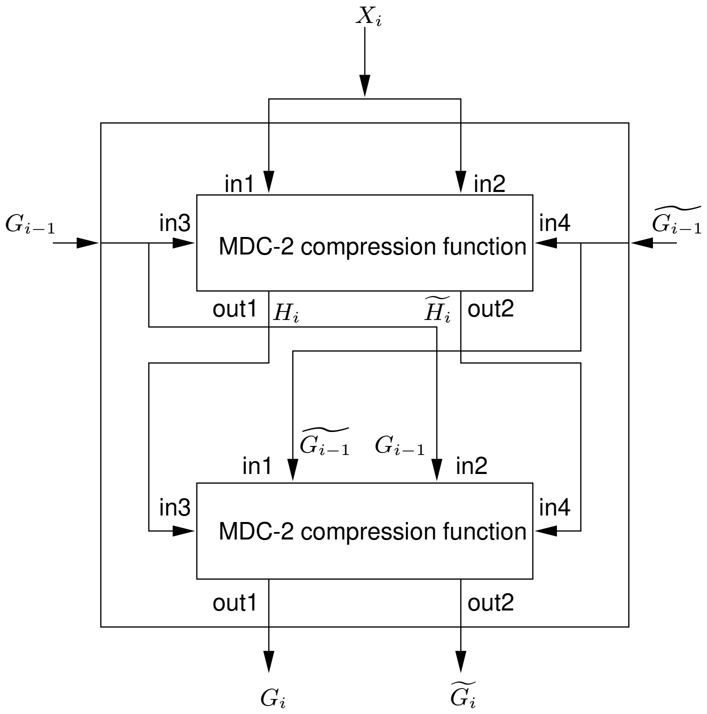

Figure 9.5: *Compression function of MDC-4 hash function*

Other important subsequent variants include the Secure Hash Algorithm (SHA-1), the hash function RIPEMD, and its strengthened variants RIPEMD-128 and RIPEMD-160. Parameters for these hash functions are summarized in Table 9.5. "Rounds $\times$ Steps per round" refers to operations performed on input blocks within the corresponding compression function. Table 9.6 specifies test vectors for a subset of these hash functions.

Notation for description of MD4-family algorithms

Table 9.7 defines the notation for the description of MD4-family algorithms described below. Note 9.48 addresses the implementation issue of converting strings of bytes to words in an unambiguous manner.

9.48 Note (*little-endian vs. big-endian*) For interoperable implementations involving byte-to-word conversions on different processors (e.g., converting between 32-bit words and groups of four 8-bit bytes), an unambiguous convention must be specified. Consider a stream of bytes B_i with increasing memory addresses i, to be interpreted as a 32-bit word with numerical value W. In *little-endian* architectures, the byte with the lowest memory address (B_1) is the least significant byte: $W = 2^{24}B_4 + 2^{16}B_3 + 2^8B_2 + B_1$. In *big-endian* architectures, the byte with the lowest address (B_1) is the most significant byte: $W = 2^{24}B_1 + 2^{16}B_2 + 2^8B_3 + B_4$.

(i) MD4

MD4 (Algorithm 9.49) is a 128-bit hash function. The original MD4 design goals were that breaking it should require roughly brute-force effort: finding distinct messages with the same hash-value should take about 2^{64} operations, and finding a message yielding a

Name	Bitlength	Rounds × Steps per round	Relative speed
MD4	128	3×16	1.00
MD5	128	4×16	0.68
RIPEMD-128	128	4×16 twice (in parallel)	0.39
SHA-1	160	4×20	0.28
RIPEMD-160	160	5×16 twice (in parallel)	0.24

Table 9.5: *Summary of selected hash functions based on MD4.*

Name	String	Hash value (as a hex byte string)
MD4	“”	31d6cfe0d16ae931b73c59d7e0c089c0
	“a”	bde52cb31de33e46245e05fbdbd6fb24
	“abc”	a448017aaf21d8525fc10ae87aa6729d
	“abcdefghijklmnopqrstuvwxyz”	d79e1c308aa5bbcdeea8ed63df412da9
MD5	“”	d41d8cd98f00b204e9800998ecf8427e
	“a”	0cc175b9c0f1b6a831c399e269772661
	“abc”	900150983cd24fb0d6963f7d28e17f72
	“abcdefghijklmnopqrstuvwxyz”	c3fcd3d76192e4007dfb496cca67e13b
SHA-1	“”	da39a3ee5e6b4b0d3255bfef95601890afd80709
	“a”	86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
	“abc”	a9993e364706816aba3e25717850c26c9cd0d89d
	“abcdefghijklmnopqrstuvwxyz”	32d10c7b8cf96570ca04ce37f2a19d84240d3a89
RIPEMD-160	“”	9c1185a5c5e9fc54612808977ee8f548b2258d31
	“a”	0bdc9d2d256b3ee9daae347be6f4dc835a467ffe
	“abc”	8eb208f7e05d987a9b044a8e98c6b087f15a0bfc
	“abcdefghijklmnopqrstuvwxyz”	f71c27109c692c1b56bbdceb5b9d2865b3708dbc

Table 9.6: *Test vectors for selected hash functions.*

Notation	Meaning
u, v, w	variables representing 32-bit quantities
0x67452301	hexadecimal 32-bit integer (least significant byte: 01)
$+$	addition modulo 2^{32}
$\overline{u}$	bitwise complement
$u \hookleftarrow s$	result of rotating u left through s positions
uv	bitwise AND
$u \vee v$	bitwise inclusive-OR
$u \oplus v$	bitwise exclusive-OR
$f(u, v, w)$	$uv \vee \overline{u}w$
$g(u, v, w)$	$uv \vee uw \vee vw$
$h(u, v, w)$	$u \oplus v \oplus w$
$(X_1, \ldots, X_j) \leftarrow (Y_1, \ldots, Y_j)$	simultaneous assignments ($X_i \leftarrow Y_i$), where $(Y_1, \ldots, Y_j)$ is evaluated prior to any assignments

Table 9.7: *Notation for MD4-family algorithms.*

pre-specified hash-value about 2^{128} operations. It is now known that MD4 fails to meet this goal (Remark 9.50). Nonetheless, a full description of MD4 is included as Algorithm 9.49 for historical and cryptanalytic reference. It also serves as a convenient reference for describing, and allowing comparisons between, other hash functions in this family.

9.49 Algorithm MD4 hash function

INPUT: bitstring x of arbitrary bitlength $b \geq 0$. (For notation see Table 9.7.)
OUTPUT: 128-bit hash-code of x. (See Table 9.6 for test vectors.)

1. *Definition of constants.* Define four 32-bit initial chaining values (IVs):
   $h_1 =$ 0x67452301, $h_2 =$ 0xefcdab89, $h_3 =$ 0x98badcfe, $h_4 =$ 0x10325476.
   Define additive 32-bit constants:
   $y[j] = 0$, $0 \leq j \leq 15$;
   $y[j] =$ 0x5a827999, $16 \leq j \leq 31$; (constant = square-root of 2)
   $y[j] =$ 0x6ed9eba1, $32 \leq j \leq 47$; (constant = square-root of 3)
   Define order for accessing source words (each list contains 0 through 15):
   $z[0..15] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$,
   $z[16..31] = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15]$,
   $z[32..47] = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]$.
   Finally define the number of bit positions for left shifts (rotates):
   $s[0..15] = [3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19, 3, 7, 11, 19]$,
   $s[16..31] = [3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13, 3, 5, 9, 13]$,
   $s[32..47] = [3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15, 3, 9, 11, 15]$.
2. *Preprocessing.* Pad x such that its bitlength is a multiple of 512, as follows. Append a single 1-bit, then append $r-1$ (≥ 0) 0-bits for the smallest r resulting in a bitlength 64 less than a multiple of 512. Finally append the 64-bit representation of $b \bmod 2^{64}$, as two 32-bit words with least significant word first. (Regarding converting between streams of bytes and 32-bit words, the convention is little-endian; see Note 9.48.) Let m be the number of 512-bit blocks in the resulting string ($b + r + 64 = 512m = 32 \cdot 16m$). The formatted input consists of $16m$ 32-bit words: $x_0 x_1 \ldots x_{16m-1}$. Initialize: $(H_1, H_2, H_3, H_4) \leftarrow (h_1, h_2, h_3, h_4)$.
3. *Processing.* For each i from 0 to $m - 1$, copy the i^{th} block of 16 32-bit words into temporary storage: $X[j] \leftarrow x_{16i+j}$, $0 \leq j \leq 15$, then process these as below in three 16-step rounds before updating the chaining variables:
   (*initialize working variables*) $(A, B, C, D) \leftarrow (H_1, H_2, H_3, H_4)$.
   (*Round 1*) For j from 0 to 15 do the following:
   $t \leftarrow (A + f(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j], B, C)$.
   (*Round 2*) For j from 16 to 31 do the following:
   $t \leftarrow (A + g(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j]), B, C)$.
   (*Round 3*) For j from 32 to 47 do the following:
   $t \leftarrow (A + h(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, t \hookleftarrow s[j]), B, C)$.
   (*update chaining values*) $(H_1, H_2, H_3, H_4) \leftarrow (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$.
4. *Completion.* The final hash-value is the concatenation: $H_1||H_2||H_3||H_4$
   (with first and last bytes the low- and high-order bytes of H_1, H_4, respectively).

9.50 Remark (*MD4 collisions*) Collisions have been found for MD4 in 2^{20} compression function computations (cf. Table 9.3). For this reason, MD4 is no longer recommended for use as a collision-resistant hash function. While its utility as a one-way function has not been studied in light of this result, it is prudent to expect a preimage attack on MD4 requiring fewer than 2^{128} operations will be found.

(ii) MD5

MD5 (Algorithm 9.51) was designed as a strengthened version of MD4, prior to actual MD4 collisions being found. It has enjoyed widespread use in practice. It has also now been found to have weaknesses (Remark 9.52).

The changes made to obtain MD5 from MD4 are as follows:

1. addition of a fourth round of 16 steps, and a Round 4 function
2. replacement of the Round 2 function by a new function
3. modification of the access order for message words in Rounds 2 and 3
4. modification of the shift amounts (such that shifts differ in distinct rounds)
5. use of unique additive constants in each of the 4×16 steps, based on the integer part of $2^{32} \cdot \sin(j)$ for step j (requiring overall, 256 bytes of storage)
6. addition of output from the previous step into each of the 64 steps.

9.51 Algorithm MD5 hash function

INPUT: bitstring x of arbitrary bitlength $b \geq 0$. (For notation, see Table 9.7.)
OUTPUT: 128-bit hash-code of x. (See Table 9.6 for test vectors.)

MD5 is obtained from MD4 by making the following changes.

1. *Notation.* Replace the Round 2 function by: $g(u, v, w) \stackrel{\text{def}}{=} uw \ \vee \ v\overline{w}$.
   Define a Round 4 function: $k(u, v, w) \stackrel{\text{def}}{=} v \oplus (u \ \vee \ \overline{w})$.
2. *Definition of constants.* Redefine unique additive constants:
   $y[j]$ = first 32 bits of binary value $\text{abs}(\sin(j+1))$, $0 \leq j \leq 63$, where j is in radians and "abs" denotes absolute value. Redefine access order for words in Rounds 2 and 3, and define for Round 4:
   $z[16..31] = [1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12]$,
   $z[32..47] = [5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2]$,
   $z[48..63] = [0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9]$.
   Redefine number of bit positions for left shifts (rotates):
   $s[0..15] = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22]$,
   $s[16..31] = [5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20]$,
   $s[32..47] = [4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23]$,
   $s[48..63] = [6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]$.
3. *Preprocessing.* As in MD4.
4. *Processing.* In each of Rounds 1, 2, and 3, replace "$B \leftarrow (t \hookleftarrow s[j])$" by "$B \leftarrow B + (t \hookleftarrow s[j])$". Also, immediately following Round 3 add:
   (*Round 4*) For j from 48 to 63 do the following:
   $t \leftarrow (A+k(B, C, D)+X[z[j]]+y[j])$, $(A, B, C, D) \leftarrow (D, B+(t \hookleftarrow s[j]), B, C)$.
5. *Completion.* As in MD4.

9.52 Remark (*MD5 compression function collisions*) While no collisions for MD5 have yet been found (cf. Table 9.3), collisions have been found for the MD5 compression function. More specifically, these are called collisions for random IV. (See §9.7.2, and in particular Definition 9.97 and Note 9.98.)

(iii) SHA-1

The Secure Hash Algorithm (SHA-1), based on MD4, was proposed by the U.S. National Institute for Standards and Technology (NIST) for certain U.S. federal government applications. The main differences of SHA-1 from MD4 are as follows:

1. The hash-value is 160 bits, and five (vs. four) 32-bit chaining variables are used.
2. The compression function has four rounds instead of three, using the MD4 step functions f, g, and h as follows: f in the first, g in the third, and h in both the second and fourth rounds. Each round has 20 steps instead of 16.
3. Within the compression function, each 16-word message block is expanded to an 80-word block, by a process whereby each of the last 64 of the 80 words is the XOR of 4 words from earlier positions in the expanded block. These 80 words are then input one-word-per-step to the 80 steps.
4. The core step is modified as follows: the only rotate used is a constant 5-bit rotate; the fifth working variable is added into each step result; message words from the expanded message block are accessed sequentially; and C is updated as B rotated left 30 bits, rather than simply B.
5. SHA-1 uses four non-zero additive constants, whereas MD4 used three constants only two of which were non-zero.

The byte ordering used for converting between streams of bytes and 32-bit words in the official SHA-1 specification is big-endian (see Note 9.48); this differs from MD4 which is little-endian.

9.53 Algorithm Secure Hash Algorithm – revised (SHA-1)

INPUT: bitstring x of bitlength $b \geq 0$. (For notation, see Table 9.7.)
OUTPUT: 160-bit hash-code of x. (See Table 9.6 for test vectors.)

SHA-1 is defined (with reference to MD4) by making the following changes.

1. *Notation.* As in MD4.
2. *Definition of constants.* Define a fifth IV to match those in MD4: $h_5 = $ 0xc3d2e1f0. Define per-round integer additive constants: $y_1 = $ 0x5a827999, $y_2 = $ 0x6ed9eba1, $y_3 = $ 0x8f1bbcdc, $y_4 = $ 0xca62c1d6. (No order for accessing source words, or specification of bit positions for left shifts is required.)
3. *Overall preprocessing.* Pad as in MD4, except the final two 32-bit words specifying the bitlength b is appended with most significant word preceding least significant. As in MD4, the formatted input is $16m$ 32-bit words: $x_0 x_1 \ldots x_{16m-1}$. Initialize chaining variables: $(H_1, H_2, H_3, H_4, H_5) \leftarrow (h_1, h_2, h_3, h_4, h_5)$.
4. *Processing.* For each i from 0 to $m-1$, copy the i^{th} block of sixteen 32-bit words into temporary storage: $X[j] \leftarrow x_{16i+j}$, $0 \leq j \leq 15$, and process these as below in four 20-step rounds before updating the chaining variables:
   (expand 16-word block into 80-word block; let X_j denote $X[j]$)
   for j from 16 to 79, $X_j \leftarrow ((\ X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}\ ) \hookleftarrow 1)$.
   (*initialize working variables*) $(A, B, C, D, E) \leftarrow (H_1, H_2, H_3, H_4, H_5)$.
   (*Round 1*) For j from 0 to 19 do the following:
   $t \leftarrow ((A \hookleftarrow 5) + f(B, C, D) + E + X_j + y_1)$,
   $(A, B, C, D, E) \leftarrow (t, A, B \hookleftarrow 30, C, D)$.
   (*Round 2*) For j from 20 to 39 do the following:
   $t \leftarrow ((A \hookleftarrow 5) + h(B, C, D) + E + X_j + y_2)$,
   $(A, B, C, D, E) \leftarrow (t, A, B \hookleftarrow 30, C, D)$.

(*Round 3*) For j from 40 to 59 do the following:
$t \leftarrow ((A \hookleftarrow 5) + g(B, C, D) + E + X_j + y_3)$,
$(A, B, C, D, E) \leftarrow (t, A, B \hookleftarrow 30, C, D)$.
(*Round 4*) For j from 60 to 79 do the following:
$t \leftarrow ((A \hookleftarrow 5) + h(B, C, D) + E + X_j + y_4)$,
$(A, B, C, D, E) \leftarrow (t, A, B \hookleftarrow 30, C, D)$.
(*update chaining values*)
$(H_1, H_2, H_3, H_4, H_5) \leftarrow (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E)$.

5. *Completion.* The hash-value is: $H_1||H_2||H_3||H_4||H_5$
(with first and last bytes the high- and low-order bytes of H_1, H_5, respectively).

9.54 Remark (*security of SHA-1*) Compared to 128-bit hash functions, the 160-bit hash-value of SHA-1 provides increased security against brute-force attacks. SHA-1 and RIPEMD-160 (see §9.4.2(iv)) presently appear to be of comparable strength; both are considered stronger than MD5 (Remark 9.52). In SHA-1, a significant effect of the expansion of 16-word message blocks to 80 words in the compression function is that any two distinct 16-word blocks yield 80-word values which differ in a larger number of bit positions, significantly expanding the number of bit differences among message words input to the compression function. The redundancy added by this preprocessing evidently adds strength.

(iv) RIPEMD-160

RIPEMD-160 (Algorithm 9.55) is a hash function based on MD4, taking into account knowledge gained in the analysis of MD4, MD5, and RIPEMD. The overall RIPEMD-160 compression function maps 21-word inputs (5-word chaining variable plus 16-word message block, with 32-bit words) to 5-word outputs. Each input block is processed in parallel by distinct versions (the *left line* and *right line*) of the compression function. The 160-bit outputs of the separate lines are combined to give a single 160-bit output.

Notation	Definition
$f(u, v, w)$	$u \oplus v \oplus w$
$g(u, v, w)$	$uv \vee \overline{u}w$
$h(u, v, w)$	$(u \vee \overline{v}) \oplus w$
$k(u, v, w)$	$uw \vee v\overline{w}$
$l(u, v, w)$	$u \oplus (v \vee \overline{w})$

Table 9.8: *RIPEMD-160 round function definitions.*

The RIPEMD-160 compression function differs from MD4 in the number of words of chaining variable, the number of rounds, the round functions themselves (Table 9.8), the order in which the input words are accessed, and the amounts by which results are rotated. The left and and right computation lines differ from each other in these last two items, in their additive constants, and in the order in which the round functions are applied. This design is intended to improve resistance against known attack strategies. Each of the parallel lines uses the same IV as SHA-1. When writing the IV as a bitstring, little-endian ordering is used for RIPEMD-160 as in MD4 (vs. big-endian in SHA-1; see Note 9.48).

9.55 Algorithm RIPEMD-160 hash function

INPUT: bitstring x of bitlength $b \geq 0$.
OUTPUT: 160-bit hash-code of x. (See Table 9.6 for test vectors.)

RIPEMD-160 is defined (with reference to MD4) by making the following changes.

1. *Notation.* See Table 9.7, with MD4 round functions f, g, h redefined per Table 9.8 (which also defines the new round functions k, l).
2. *Definition of constants.* Define a fifth IV: $h_5 = $ 0xc3d2e1f0. In addition:
   (a) Use the MD4 additive constants for the left line, renamed: $y_L[j] = 0$, $0 \leq j \leq 15$; $y_L[j] = $ 0x5a827999, $16 \leq j \leq 31$; $y_L[j] = $ 0x6ed9eba1, $32 \leq j \leq 47$. Define two further constants (square roots of 5,7): $y_L[j] = $ 0x8f1bbcdc, $48 \leq j \leq 63$; $y_L[j] = $ 0xa953fd4e, $64 \leq j \leq 79$.
   (b) Define five new additive constants for the right line (cube roots of 2,3,5,7): $y_R[j] = $ 0x50a28be6, $0 \leq j \leq 15$; $y_R[j] = $ 0x5c4dd124, $16 \leq j \leq 31$; $y_R[j] = $ 0x6d703ef3, $32 \leq j \leq 47$; $y_R[j] = $ 0x7a6d76e9, $48 \leq j \leq 63$; $y_R[j] = 0$, $64 \leq j \leq 79$.
   (c) See Table 9.9 for constants for step j of the compression function: $z_L[j]$, $z_R[j]$ specify the access order for source words in the left and right lines; $s_L[j]$, $s_R[j]$ the number of bit positions for rotates (see below).
3. *Preprocessing.* As in MD4, with addition of a fifth chaining variable: $H_5 \leftarrow h_5$.
4. *Processing.* For each i from 0 to $m-1$, copy the i^{th} block of sixteen 32-bit words into temporary storage: $X[j] \leftarrow x_{16i+j}$, $0 \leq j \leq 15$. Then:
   (a) Execute five 16-step rounds of the left line as follows:
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (H_1, H_2, H_3, H_4, H_5)$.
   (*left Round 1*) For j from 0 to 15 do the following:
   $t \leftarrow (A_L + f(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (*left Round 2*) For j from 16 to 31 do the following:
   $t \leftarrow (A_L + g(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (*left Round 3*) For j from 32 to 47 do the following:
   $t \leftarrow (A_L + h(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (*left Round 4*) For j from 48 to 63 do the following:
   $t \leftarrow (A_L + k(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (*left Round 5*) For j from 64 to 79 do the following:
   $t \leftarrow (A_L + l(B_L, C_L, D_L) + X[z_L[j]] + y_L[j])$,
   $(A_L, B_L, C_L, D_L, E_L) \leftarrow (E_L, E_L + (t \hookleftarrow s_L[j]), B_L, C_L \hookleftarrow 10, D_L)$.
   (b) Execute in parallel with the above five rounds an analogous right line with $(A_R, B_R, C_R, D_R, E_R)$, $y_R[j]$, $z_R[j]$, $s_R[j]$ replacing the corresponding quantities with subscript L; and the order of the round functions reversed so that their order is: l, k, h, g, and f. Start by initializing the right line working variables: $(A_R, B_R, C_R, D_R, E_R) \leftarrow (H_1, H_2, H_3, H_4, H_5)$.
   (c) After executing both the left and right lines above, update the chaining values as follows: $t \leftarrow H_1$, $H_1 \leftarrow H_2 + C_L + D_R$, $H_2 \leftarrow H_3 + D_L + E_R$, $H_3 \leftarrow H_4 + E_L + A_R$, $H_4 \leftarrow H_5 + A_L + B_R$, $H_5 \leftarrow t + B_L + C_R$.
5. *Completion.* The final hash-value is the concatenation: $H_1||H_2||H_3||H_4||H_5$ (with first and last bytes the low- and high-order bytes of H_1, H_5, respectively).

Variable	Value
z_L[0..15]	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15]
z_L[16..31]	[7, 4,13, 1,10, 6,15, 3,12, 0, 9, 5, 2,14,11, 8]
z_L[32..47]	[3,10,14, 4, 9,15, 8, 1, 2, 7, 0, 6,13,11, 5,12]
z_L[48..63]	[1, 9,11,10, 0, 8,12, 4,13, 3, 7,15,14, 5, 6, 2]
z_L[64..79]	[4, 0, 5, 9, 7,12, 2,10,14, 1, 3, 8,11, 6,15,13]
z_R[0..15]	[5,14, 7, 0, 9, 2,11, 4,13, 6,15, 8, 1,10, 3,12]
z_R[16..31]	[6,11, 3, 7, 0,13, 5,10,14,15, 8,12, 4, 9, 1, 2]
z_R[32..47]	[15, 5, 1, 3, 7,14, 6, 9,11, 8,12, 2,10, 0, 4,13]
z_R[48..63]	[8, 6, 4, 1, 3,11,15, 0, 5,12, 2,13, 9, 7,10,14]
z_R[64..79]	[12,15,10, 4, 1, 5, 8, 7, 6, 2,13,14, 0, 3, 9,11]
s_L[0..15]	[11,14,15,12, 5, 8, 7, 9,11,13,14,15, 6, 7, 9, 8]
s_L[16..31]	[7, 6, 8,13,11, 9, 7,15, 7,12,15, 9,11, 7,13,12]
s_L[32..47]	[11,13, 6, 7,14, 9,13,15,14, 8,13, 6, 5,12, 7, 5]
s_L[48..63]	[11,12,14,15,14,15, 9, 8, 9,14, 5, 6, 8, 6, 5,12]
s_L[64..79]	[9,15, 5,11, 6, 8,13,12, 5,12,13,14,11, 8, 5, 6]
s_R[0..15]	[8, 9, 9,11,13,15,15, 5, 7, 7, 8,11,14,14,12, 6]
s_R[16..31]	[9,13,15, 7,12, 8, 9,11, 7, 7,12, 7, 6,15,13,11]
s_R[32..47]	[9, 7,15,11, 8, 6, 6,14,12,13, 5,14,13,13, 7, 5]
s_R[48..63]	[15, 5, 8,11,14,14, 6,14, 6, 9,12, 9,12, 5,15, 8]
s_R[64..79]	[8, 5,12, 9,12, 5,14, 6, 8,13, 6, 5,15,13,11,11]

Table 9.9: *RIPEMD-160 word-access orders and rotate counts (cf. Algorithm 9.55).*

9.4.3 Hash functions based on modular arithmetic

The basic idea of hash functions based on modular arithmetic is to construct an iterated hash function using mod M arithmetic as the basis of a compression function. Two motivating factors are re-use of existing software or hardware (in public-key systems) for modular arithmetic, and scalability to match required security levels. Significant disadvantages, however, include speed (e.g., relative to the customized hash functions of §9.4.2), and an embarrassing history of insecure proposals.

MASH

MASH-1 (*Modular Arithmetic Secure Hash, algorithm 1*) is a hash function based on modular arithmetic. It has been proposed for inclusion in a draft ISO/IEC standard. MASH-1 involves use of an RSA-like modulus M, whose bitlength affects the security. M should be difficult to factor, and for M of unknown factorization, the security is based in part on the difficulty of extracting modular roots (§3.5.2). The bitlength of M also determines the blocksize for processing messages, and the size of the hash-result (e.g., a 1025-bit modulus yields a 1024-bit hash-result). As a recent proposal, its security remains open to question (page 381). Techniques for reducing the size of the final hash-result have also been proposed, but their security is again undetermined as yet.

9.56 Algorithm MASH-1 (version of Nov. 1995)

INPUT: data x of bitlength $0 \leq b < 2^{n/2}$.
OUTPUT: n-bit hash of x (n is approximately the bitlength of the modulus M).

1. *System setup and constant definitions.* Fix an RSA-like modulus $M = pq$ of bitlength m, where p and q are randomly chosen secret primes such that the factorization of M is intractable. Define the bitlength n of the hash-result to be the largest multiple of 16 less than m (i.e., $n = 16n' < m$). $H_0 = 0$ is defined as an IV, and an n-bit integer constant $A = $ 0xf0...0. "$\vee$" denotes bitwise inclusive-OR; "$\oplus$" denotes bitwise exclusive-OR.
2. *Padding, blocking, and MD-strengthening.* Pad x with 0-bits, if necessary, to obtain a string of bitlength $t \cdot n/2$ for the smallest possible $t \geq 1$. Divide the padded text into ($n/2$)-bit blocks $x_1, \ldots, x_t$, and append a final block x_{t+1} containing the ($n/2$)-bit representation of b.
3. *Expansion.* Expand each x_i to an n-bit block y_i by partitioning it into (4-bit) nibbles and inserting four 1-bits preceding each, except for y_{t+1} wherein the inserted nibble is 1010 (not 1111).
4. *Compression function processing.* For $1 \leq i \leq t+1$, map two n-bit inputs (H_{i-1}, y_i) to one n-bit output as follows: $H_i \leftarrow ((((H_{i-1} \oplus y_i) \vee A)^2 \bmod M) \dashv n) \oplus H_{i-1}$. Here $\dashv n$ denotes keeping the rightmost n bits of the m-bit result to its left.
5. *Completion.* The hash is the n-bit block H_{t+1}.

MASH-2 is defined as per MASH-1 with the exponent $e = 2$ used for squaring in the compression function processing stage (step 4) replaced with $e = 2^8 + 1$.

9.5 Keyed hash functions (MACs)

Keyed hash functions whose specific purpose is message authentication are called message authentication code (MAC) algorithms. Compared to the large number of MDC algorithms, prior to 1995 relatively few MAC algorithms had been proposed, presumably because the original proposals, which were widely adopted in practice, were adequate. Many of these are for historical reasons block-cipher based. Those with relatively short MAC bitlengths (e.g., 32-bits for MAA) or short keys (e.g., 56 bits for MACs based on DES-CBC) may still offer adequate security, depending on the computational resources available to adversaries, and the particular environment of application.

Many iterated MACs can be described as iterated hash functions (see Figure 9.2, and equation (9.1) on page 333). In this case, the MAC key is generally part of the output transformation g; it may also be an input to the compression function in the first iteration, and be involved in the compression function f at every stage.

Fact 9.57 is a general result giving an upper bound on the security of MACs.

9.57 Fact (*birthday attack on MACs*) Let h be a MAC algorithm based on an iterated compression function, which has n bits of internal chaining variable, and is deterministic (i.e., the m-bit result is fully determined by the message). Then MAC forgery is possible using $O(2^{n/2})$ known text-MAC pairs plus a number v of chosen text-MAC pairs which (depending on h) is between 1 and about 2^{n-m}.

9.5.1 MACs based on block ciphers

CBC-based MACs

The most commonly used MAC algorithm based on a block cipher makes use of cipher-block-chaining (§7.2.2(ii)). When DES is used as the block cipher E, $n = 64$ in what follows, and the MAC key is a 56-bit DES key.

9.58 Algorithm CBC-MAC

INPUT: data x; specification of block cipher E; secret MAC key k for E.
OUTPUT: n-bit MAC on x (n is the blocklength of E).

1. *Padding and blocking.* Pad x if necessary (e.g., using Algorithm 9.30). Divide the padded text into n-bit blocks denoted $x_1, \ldots, x_t$.
2. *CBC processing.* Letting E_k denote encryption using E with key k, compute the block H_t as follows: $H_1 \leftarrow E_k(x_1)$; $H_i \leftarrow E_k(H_{i-1} \oplus x_i)$, $2 \leq i \leq t$. (This is standard cipher-block-chaining, $IV = 0$, discarding ciphertext blocks $C_i = H_i$.)
3. *Optional process to increase strength of MAC.* Using a second secret key $k' \neq k$, optionally compute: $H'_t \leftarrow E^{-1}_{k'}(H_t)$, $H_t \leftarrow E_k(H'_t)$. (This amounts to using two-key triple-encryption on the last block; see Remark 9.59.)
4. *Completion.* The MAC is the n-bit block H_t.

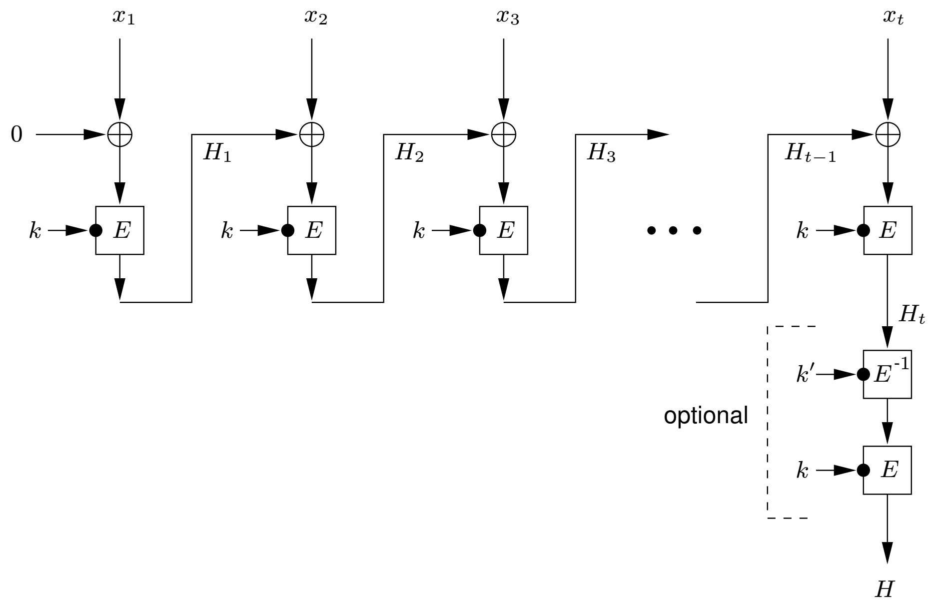

Figure 9.6: *CBC-based MAC algorithm.*

For CBC-MAC with $n = 64 = m$, Fact 9.57 applies with $v = 1$.

9.59 Remark (*CBC-MAC strengthening*) The optional process reduces the threat of exhaustive key search, and prevents chosen-text existential forgery (Example 9.62), without impacting the efficiency of the intermediate stages as would using two-key triple-encryption throughout. Alternatives to combat such forgery include prepending the input with a length block before the MAC computation; or using key K to encrypt the length m yielding $K' = E_K(m)$, before using K' as the key to MAC the message.

9.60 Remark (*truncated MAC outputs*) Exhaustive attack may, depending on the unicity distance of the MAC, be precluded (information-theoretically) by using less than n bits of the final output as the m-bit MAC. (This must be traded off against an increase in the probability of randomly guessing the MAC: 2^{-m}.) For $m = 32$ and $E =$ DES, an exhaustive attack reduces the key space to about 2^{24} possibilities. However, even for $m < n$, a second text-MAC pair almost certainly determines a unique MAC key.

9.61 Remark (*CBC-MAC IV*) While a random IV in CBC encryption serves to prevent a codebook attack on the first ciphertext block, this is not a concern in a MAC algorithm.

9.62 Example (*existential forgery of CBC-MAC*) While CBC-MAC is secure for messages of a fixed number t of blocks, additional measures (beyond simply adding a trailing length-block) are required if variable length messages are allowed, otherwise (adaptive chosen-text) existential forgery is possible as follows. Assume x_i is an n-bit block, and let $\perp b$ denote the n-bit binary representation of b. Let (x_1, M_1) be a known text-MAC pair, and request the MAC M_2 for the one-block message $x_2 = M_1$; then $M_2 = E_k(E_k(x_1))$ is also the MAC for the 2-block message $(x_1||\perp 0)$. As a less trivial example, given two known text-MAC pairs (x_1, H_1), (x_2, H_2) for one-block messages x_1, x_2, and requesting the MAC M on a chosen 2-block third message $(x_1||z)$ for a third text-MAC pair $((x_1||z), M)$, then $H_i = E_k(x_i)$, $M = E_k(H_1 \oplus z)$, and the MAC for the new 2-block message $X = x_2||(H_1 \oplus z \oplus H_2)$ is known – it is M also. Moreover, MD-strengthening (Algorithm 9.26) does not address the problem: assume padding by Algorithm 9.29, replace the third message above by the 3-block message $(x_1||\perp 64||z)$, note

$$H_i' = E_k(E_k(x_i) \oplus \perp 64), \quad M_3 = E_k(E_k(E_k(E_k(x_1) \oplus \perp 64) \oplus z) \oplus \perp 192),$$

and M_3 is also the MAC for the new 3-block message $X = (x_2||\perp 64||H_1' \oplus H_2' \oplus z)$. □

9.63 Example (*RIPE-MAC*) RIPE-MAC is a variant of CBC-MAC. Two versions RIPE-MAC1 and RIPE-MAC3, both producing 64-bit MACs, differ in their internal encryption function E being either single DES or two-key triple-DES, respectively, requiring a 56- or 112-bit key k (cf. Remark 9.59). Differences from Algorithm 9.58 are as follows: the compression function uses a non-invertible chaining best described as CBC with data feed-forward: $H_i \leftarrow E_k(H_{i-1} \oplus x_i) \oplus x_i$; after padding using Algorithm 9.30, a final 64-bit length-block (giving bitlength of original input) is appended; the optional process of Algorithm 9.58 is mandatory with final output block encrypted using key k' derived by complementing alternating nibbles of k: for $k = k_0 \dots k_{63}$ a 56-bit DES key with parity bits $k_7 k_{15} \dots k_{63}$, $k' = k \oplus$ 0xf0f0f0f0f0f0f0f0. □

9.5.2 Constructing MACs from MDCs

A common suggestion is to construct a MAC algorithm from an MDC algorithm, by simply including a secret key k as part of the MDC input. A concern with this approach is that implicit but unverified assumptions are often made about the properties that MDCs have; in particular, while most MDCs are designed to provide one-wayness or collision resistance, the requirements of a MAC algorithm differ (Definition 9.7). Even in the case that a one-way hash function precludes recovery of a secret key used as a partial message input (cf. partial-preimage resistance, page 331), this does not guarantee the infeasibility of producing MACs for new inputs. The following examples suggest that construction of a MAC from a hash function requires careful analysis.

9.64 Example (*secret prefix method*) Consider a message $x = x_1x_2 \dots x_t$ and an iterated MDC h with compression function f, with definition: $H_0 = IV, H_i = f(H_{i-1}, x_i)$; $h(x) = H_t$. (1) Suppose one attempts to use h as a MAC algorithm by prepending a secret key k, so that the proposed MAC on x is $M = h(k||x)$. Then, extending the message x by an arbitrary single block y, one may deduce $M' = h(k||x||y)$ as $f(M, y)$ without knowing the secret key k (the original MAC M serves as chaining variable). This is true even for hash functions whose preprocessing pads inputs with length indicators (e.g., MD5); in this case, the padding/length-block z for the original message x would appear as part of the extended message, $x||z||y$, but a forged MAC on the latter may nonetheless be deduced. (2) For similar reasons, it is insecure to use an MDC to construct a MAC algorithm by using the secret MAC key k as IV. If k comprises the entire first block, then for efficiency $f(IV, k)$ may be precomputed, illustrating that an adversary need only find a k' (not necessarily k) such that $f(IV, k) = f(IV, k')$; this is equivalent to using a secret IV. □

9.65 Example (*secret suffix method*) An alternative proposal is to use a secret key as a suffix, i.e., the n-bit MAC on x is $M = h(x||k)$. In this case, a birthday attack applies (§9.7.1). An adversary free to choose the message x (or a prefix thereof) may, in O($2^{n/2}$) operations, find a pair of messages x, x' for which $h(x) = h(x')$. (This can be done off-line, and does not require knowledge of k; the assumption here is that n is the size of both the chaining variable and the final output.) Obtaining a MAC M on x by legitimate means then allows an adversary to produce a correct text-MAC pair (x', M) for a new message x'. Note that this method essentially hashes and then encrypts the hash-value in the final iteration; in this weak form of MAC, the MAC-value depends only on the last chaining value, and the key is used in only one step. □

The above examples suggest that a MAC key should be involved at both the start and the end of MAC computations, leading to Example 9.66.

9.66 Example (*envelope method with padding*) For a key k and MDC h, compute the MAC on a message x as: $h_k(x) = h(k \,||\, p \,||\, x \,||\, k)$. Here p is a string used to pad k to the length of one block, to ensure that the internal computation involves at least two iterations. For example, if h is MD5 and k is 128 bits, p is a 384-bit pad string. □

Due to both a certificational attack against the MAC construction of Example 9.66 and theoretical support for that of Example 9.67 (see page 382), the latter construction is favored.

9.67 Example (*hash-based MAC*) For a key k and MDC h, compute the MAC on a message x as HMAC$(x) = h(k \,||\, p_1 \,||\, h(k \,||\, p_2 \,||\, x))$, where p_1, p_2 are distinct strings of sufficient length to pad k out to a full block for the compression function. The overall construction is quite efficient despite two calls to h, since the outer execution processes only (e.g., if h is MD5) a two-block input, independent of the length of x. □

Additional suggestions for achieving MAC-like functionality by combining MDCs and encryption are discussed in §9.6.5.

9.5.3 Customized MACs

Two algorithms designed for the specific purpose of message authentication are discussed in this section: MAA and MD5-MAC.

Message Authenticator Algorithm (MAA)

The Message Authenticator Algorithm (MAA), dating from 1983, is a customized MAC algorithm for 32-bit machines, involving 32-bit operations throughout. It is specified as Algorithm 9.68 and illustrated in Figure 9.7. The main loop consists of two parallel interdependent streams of computation. Messages are processed in 4-byte blocks using 8 bytes of chaining variable. The execution time (excluding key expansion) is proportional to message length; as a rough guideline, MAA is twice as slow as MD4.

9.68 Algorithm Message Authenticator Algorithm (MAA)

INPUT: data x of bitlength $32j$, $1 \leq j \leq 10^6$; secret 64-bit MAC key $Z = Z[1]..Z[8]$.
OUTPUT: 32-bit MAC on x.

1. *Message-independent key expansion.* Expand key Z to six 32-bit quantities X, Y, V, W, S, T (X, Y are initial values; V, W are main loop variables; S, T are appended to the message) as follows.
   1.1 First replace any bytes 0x00 or 0xff in Z as follows. $P \leftarrow 0$; for i from 1 to 8 ($P \leftarrow 2P$; if $Z[i] =$ 0x00 or 0xff then ($P \leftarrow P + 1$; $Z[i] \leftarrow Z[i]$ OR P)).
   1.2 Let J and K be the first 4 bytes and last 4 bytes of Z, and compute:[4]
   $X \leftarrow J^4 \pmod{2^{32}-1} \oplus J^4 \pmod{2^{32}-2}$
   $Y \leftarrow [K^5 \pmod{2^{32}-1} \oplus K^5 \pmod{2^{32}-2}](1+P)^2 \pmod{2^{32}-2}$
   $V \leftarrow J^6 \pmod{2^{32}-1} \oplus J^6 \pmod{2^{32}-2}$
   $W \leftarrow K^7 \pmod{2^{32}-1} \oplus K^7 \pmod{2^{32}-2}$
   $S \leftarrow J^8 \pmod{2^{32}-1} \oplus J^8 \pmod{2^{32}-2}$
   $T \leftarrow K^9 \pmod{2^{32}-1} \oplus K^9 \pmod{2^{32}-2}$
   1.3 Process the 3 resulting pairs (X, Y), (V, W), (S, T) to remove any bytes 0x00, 0xff as for Z earlier. Define the AND-OR constants: $A =$ 0x02040801, $B =$ 0x00804021, $C =$ 0xbfef7fdf, $D =$ 0x7dfefbff.
2. *Initialization and preprocessing.* Initialize the rotating vector: $v \leftarrow V$, and the chaining variables: $H_1 \leftarrow X$, $H_2 \leftarrow Y$. Append the key-derived blocks S, T to x, and let $x_1 \ldots x_t$ denote the resulting augmented segment of 32-bit blocks. (The final 2 blocks of the segment thus involve key-derived secrets.)
3. *Block processing.* Process each 32-bit block x_i (for i from 1 to t) as follows.
   $v \leftarrow (v \hookleftarrow 1)$, $U \leftarrow (v \oplus W)$
   $t_1 \leftarrow (H_1 \oplus x_i) \times_1 (((H_2 \oplus x_i) + U) \text{ OR } A) \text{ AND } C)$
   $t_2 \leftarrow (H_2 \oplus x_i) \times_2 (((H_1 \oplus x_i) + U) \text{ OR } B) \text{ AND } D)$
   $H_1 \leftarrow t_1$, $H_2 \leftarrow t_2$
   where $\times_i$ denotes special multiplication mod $2^{32} - i$ as noted above ($i = 1$ or 2); "+" is addition mod 2^{32}; and "$\hookleftarrow 1$" denotes rotation left one bit. (Each combined AND-OR operation on a 32-bit quantity sets 4 bits to 1, and 4 to 0, precluding 0-multipliers.)
4. *Completion.* The resulting MAC is: $H = H_1 \oplus H_2$.

[4] In ISO 8731-2, a well-defined but unconventional definition of multiplication mod $2^{32} - 2$ is specified, producing 32-bit results which in some cases are $2^{32} - 1$ or $2^{32} - 2$; for this reason, specifying e.g., J^6 here may be ambiguous; the standard should be consulted for exact details.

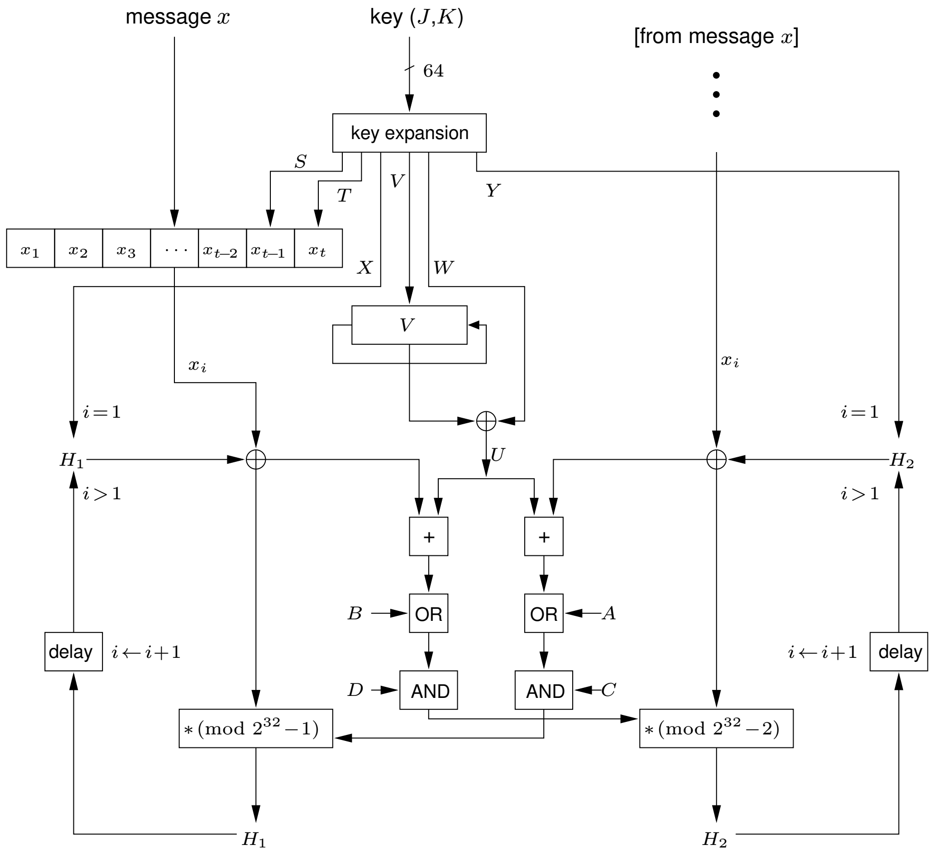

Figure 9.7: *The Message Authenticator Algorithm (MAA).*

Since the relatively complex key expansion stage is independent of the message, a one-time computation suffices for a fixed key. The mixing of various operations (arithmetic mod $2^{32} - i$, for $i = 0, 1$ and 2; XOR; and nonlinear AND-OR computations) is intended to strengthen the algorithm against arithmetic cryptanalytic attacks.

MD5-MAC

A more conservative approach (cf. Example 9.66) to building a MAC from an MDC is to arrange that the MAC compression function itself depend on k, implying the secret key be involved in all intervening iterations; this provides additional protection in the case that weaknesses of the underlying hash function become known. Algorithm 9.69 is such a technique, constructed using MD5. It provides performance close to that of MD5 (5-20% slower in software).

9.69 Algorithm MD5-MAC

INPUT: bitstring x of arbitrary bitlength $b \geq 0$; key k of bitlength ≤ 128.
OUTPUT: 64-bit MAC-value of x.

MD5-MAC is obtained from MD5 (Algorithm 9.51) by the following changes.

1. *Constants.* The constants U_i and T_i are as defined in Example 9.70.
2. *Key expansion.*
   (a) If k is shorter than 128 bits, concatenate k to itself a sufficient number of times, and redefine k to be the leftmost 128 bits.
   (b) Let $\overline{\text{MD5}}$ denote MD5 with both padding and appended length omitted. Expand k into three 16-byte subkeys K_0, K_1, and K_2 as follows: for i from 0 to 2, $K_i \leftarrow \overline{\text{MD5}}(k \,\|\, U_i \,\|\, k)$.
   (c) Partition each of K_0 and K_1 into four 32-bit substrings $K_j[i]$, $0 \leq i \leq 3$.
3. K_0 replaces the four 32-bit IV's of MD5 (i.e., $h_i = K_0[i]$).
4. $K_1[i]$ is added $\bmod\ 2^{32}$ to each constant $y[j]$ used in Round i of MD5.
5. K_2 is used to construct the following 512-bit block, which is appended to the padded input x subsequent to the regular padding and length block as defined by MD5: $K_2 \,\|\, K_2 \oplus T_0 \,\|\, K_2 \oplus T_1 \,\|\, K_2 \oplus T_2$.
6. The MAC-value is the leftmost 64 bits of the 128-bit output from hashing this padded and extended input string using MD5 with the above modifications.

9.70 Example (*MD5-MAC constants/test vectors*) The 16-byte constants T_i and three test vectors (x, MD5-MAC(x)) for key k = `00112233445566778899aabbccddeeff` are given below. (The T_i themselves are derived using MD5 on pre-defined constants.) With subscripts in T_i taken mod 3, the 96-byte constants U_0, U_1, U_2 are defined: $U_i = T_i \,\|\, T_{i+1} \,\|\, T_{i+2} \,\|\, T_i \,\|\, T_{i+1} \,\|\, T_{i+2}$.

```
T0:   97 ef 45 ac 29 0f 43 cd 45 7e 1b 55 1c 80 11 34
T1:   b1 77 ce 96 2e 72 8e 7c 5f 5a ab 0a 36 43 be 18
T2:   9d 21 b4 21 bc 87 b9 4d a2 9d 27 bd c7 5b d7 c3
("",                            1f1ef2375cc0e0844f98e7e811a34da8)
("abc",                         e8013c11f7209d1328c0caa04fd012a6)
("abcdefghijklmnopqrstuvwxyz", 9172867eb60017884c6fa8cc88ebe7c9)
```
□

9.5.4 MACs for stream ciphers

Providing data origin authentication and data integrity guarantees for stream ciphers is particularly important due to the fact that bit manipulations in additive stream-ciphers may directly result in predictable modifications of the underlying plaintext (e.g., Example 9.83). While iterated hash functions process message data a block at a time (§9.3.1), MACs designed for use with stream ciphers process messages either one bit or one symbol (block) at a time, and those which may be implemented using linear feedback shift registers (LFSRs) are desirable for reasons of efficiency.

One such MAC technique, Algorithm 9.72 below, is based on cyclic redundancy codes (cf. Example 9.80). In this case, the polynomial division may be implemented using an LFSR. The following definition is of use in what follows.

9.71 Definition A (b, m) *hash-family* $\mathcal{H}$ is a collection of hash functions mapping b-bit messages to m-bit hash-values. A (b, m) hash-family is ε-*balanced* if for all messages $B \neq 0$ and all m-bit hash-values c, $prob_h(h(B) = c)) \leq \varepsilon$, where the probability is over all randomly selected functions $h \in \mathcal{H}$.

9.72 Algorithm CRC-based MAC

INPUT: b-bit message B; shared key (see below) between MAC source and verifier.
OUTPUT: m-bit MAC-value on B (e.g., $m = 64$).

1. *Notation.* Associate $B = B_{b-1} \dots B_1 B_0$ with the polynomial $B(x) = \sum_{i=0}^{b-1} B_i x^i$.
2. *Selection of MAC key.*
   (a) Select a random binary irreducible polynomial $p(x)$ of degree m. (This represents randomly drawing a function h from a (b, m) hash-family.)
   (b) Select a random m-bit one-time key k (to be used as a one-time pad).

   The secret MAC key consists of $p(x)$ and k, both of which must be shared a priori between the MAC originator and verifier.
3. Compute $h(B) = coef(B(x) \cdot x^m \bmod p(x))$, the m-bit string of coefficients from the degree $m - 1$ remainder polynomial after dividing $B(x) \cdot x^m$ by $p(x)$.
4. The m-bit MAC-value for B is: $h(B) \oplus k$.

9.73 Fact (*security of CRC-based MAC*) For any values b and $m > 1$, the hash-family resulting from Algorithm 9.72 is ε-balanced for $\varepsilon = (b + m)/(2^{m-1})$, and the probability of MAC forgery is at most ε.

9.74 Remark (*polynomial reuse*) The hash function h in Algorithm 9.72 is determined by the irreducible polynomial $p(x)$. In practice, $p(x)$ may be re-used for different messages (e.g., within a session), but for each message a new random key k should be used.

9.6 Data integrity and message authentication

This section considers the use of hash functions for data integrity and message authentication. Following preliminary subsections, respectively, providing background definitions and distinguishing non-malicious from malicious threats to data integrity, three subsequent subsections consider three basic approaches to providing data integrity using hash functions, as summarized in Figure 9.8.

9.6.1 Background and definitions

This subsection discusses data integrity, data origin authentication (message authentication), and transaction authentication.

Assurances are typically required both that data actually came from its reputed source (data origin authentication), and that its state is unaltered (data integrity). These issues cannot be separated – data which has been altered effectively has a new source; and if a source cannot be determined, then the question of alteration cannot be settled (without reference to a source). Integrity mechanisms thus implicitly provide data origin authentication, and vice versa.

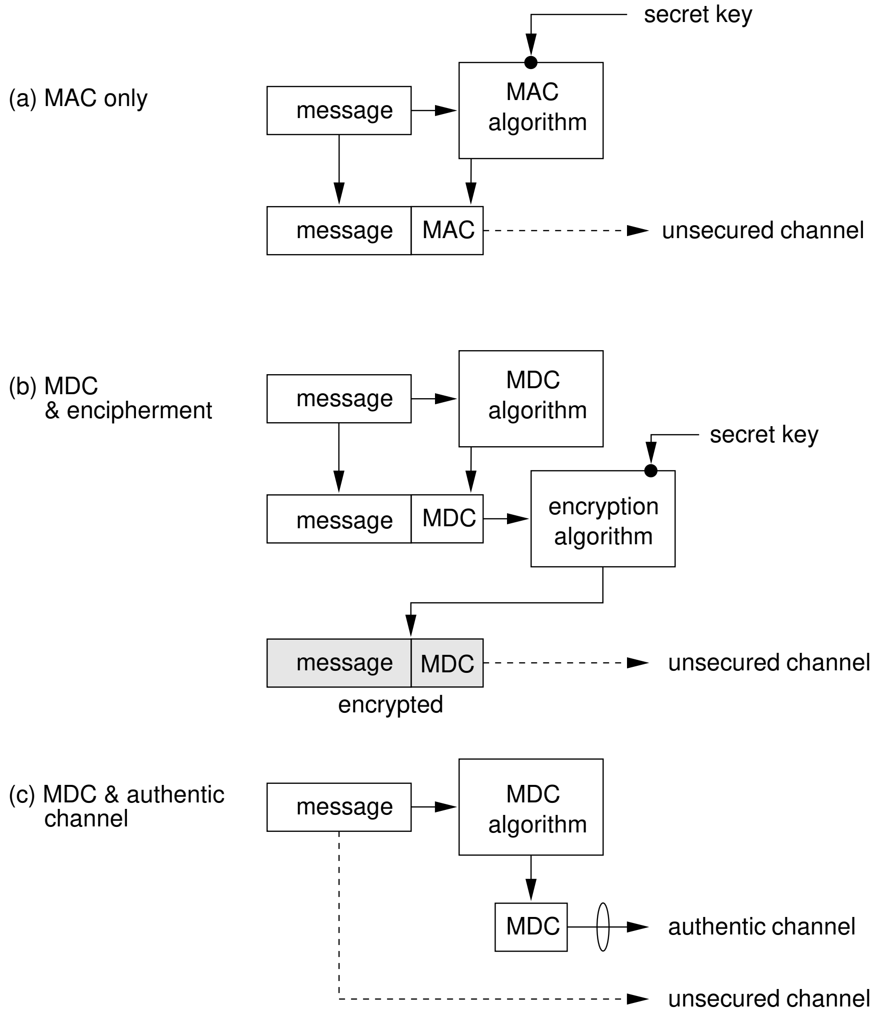

Figure 9.8: *Three methods for providing data integrity using hash functions. The second method provides encipherment simultaneously.*

(i) Data integrity

9.75 Definition *Data integrity* is the property whereby data has not been altered in an unauthorized manner since the time it was created, transmitted, or stored by an authorized source.

Verification of data integrity requires that only a subset of all candidate data items satisfies particular criteria distinguishing the acceptable from the unacceptable. Criteria allowing recognizability of data integrity include appropriate redundancy or expectation with respect to format. Cryptographic techniques for data integrity rely on either secret information or authentic channels (§9.6.4).

The specific focus of data integrity is on the bitwise composition of data (cf. transaction authentication below). Operations which invalidate integrity include: insertion of bits, including entirely new data items from fraudulent sources; deletion of bits (short of deleting entire data items); re-ordering of bits or groups of bits; inversion or substitution of bits; and any combination of these, such as message splicing (re-use of proper substrings to construct new or altered data items). Data integrity includes the notion that data items are complete. For items split into multiple blocks, the above alterations apply analogously with blocks envisioned as substrings of a contiguous data string.

(ii) Data origin authentication (message authentication)

9.76 Definition *Data origin authentication* is a type of authentication whereby a party is corroborated as the (original) source of specified data created at some (typically unspecified) time in the past.

By definition, data origin authentication includes data integrity.

9.77 Definition *Message authentication* is a term used analogously with data origin authentication. It provides data origin authentication with respect to the original message source (and data integrity, but no uniqueness and timeliness guarantees).

Methods for providing data origin authentication include the following:

1. message authentication codes (MACs)
2. digital signature schemes
3. appending (prior to encryption) a secret authenticator value to encrypted text.[5]

Data origin authentication mechanisms based on shared secret keys (e.g., MACs) do not allow a distinction to be made between the parties sharing the key, and thus (as opposed to digital signatures) do not provide non-repudiation of data origin – either party can equally originate a message using the shared key. If resolution of subsequent disputes is a potential requirement, either an on-line trusted third party in a notary role, or asymmetric techniques (see Chapter 11) may be used.

While MACs and digital signatures may be used to establish that data was generated by a specified party at some time in the past, they provide no inherent uniqueness or timeliness guarantees. These techniques alone thus cannot detect message re-use or replay, which is necessary in environments where messages may have renewed effect on second or subsequent use. Such message authentication techniques may, however, be augmented to provide these guarantees, as next discussed.

---

[5] Such a *sealed authenticator* (cf. a MAC, sometimes called an *appended authenticator*) is used along with an encryption method which provides error extension. While this resembles the technique of using encryption and an MDC (§9.6.5), whereas the MDC is a (known) function of the plaintext, a sealed authenticator is itself secret.

(iii) Transaction authentication

9.78 Definition *Transaction authentication* denotes message authentication augmented to additionally provide uniqueness and timeliness guarantees on data (thus preventing undetectable message replay).

The uniqueness and timeliness guarantees of Definition 9.78 are typically provided by appropriate use of time-variant parameters (TVPs). These include random numbers in challenge-response protocols, sequence numbers, and timestamps as discussed in §10.3.1. This may be viewed as a combination of message authentication and entity authentication (Definition 10.1). Loosely speaking,

$$\text{message authentication} + \text{TVP} = \text{transaction authentication}.$$

As a simple example, sequence numbers included within the data of messages authenticated by a MAC or digital signature algorithm allow replay detection (see Remark 9.79), and thus provide transaction authentication.

As a second example, for exchanges between two parties involving two or more messages, transaction authentication on each of the second and subsequent messages may be provided by including in the message data covered by a MAC a random number sent by the other party in the previous message. This chaining of messages through random numbers prevents message replay, since any MAC values in replayed messages would be incorrect (due to disagreement between the random number in the replayed message, and the most recent random number of the verifier).

Table 9.10 summarizes the properties of these and other types of authentication. Authentication in the broadest sense encompasses not only data integrity and data origin authentication, but also protection from all active attacks including fraudulent representation and message replay. In contrast, encryption provides protection only from passive attacks.

→ Property ↓ Type of authentication	identification of source	data integrity	timeliness or uniqueness	defined in
message authentication	yes	yes	—	§9.6.1
transaction authentication	yes	yes	yes	§9.6.1
entity authentication	yes	—	yes	§10.1.1
key authentication	yes	yes	desirable	§12.2.1

Table 9.10: *Properties of various types of authentication.*

9.79 Remark (*sequence numbers and authentication*) Sequence numbers may provide uniqueness, but not (real-time) timeliness, and thus are more appropriate to detect message replay than for entity authentication. Sequence numbers may also be used to detect the deletion of entire messages; they thus allow data integrity to be checked over an ongoing sequence of messages, in addition to individual messages.

9.6.2 Non-malicious vs. malicious threats to data integrity

The techniques required to provide data integrity on noisy channels differ substantially from those required on channels subject to manipulation by adversaries.

Checksums provide protection against accidental or non-malicious errors on channels which are subject to transmission errors. The protection is non-cryptographic, in the sense that neither secret keys nor secured channels are used. Checksums generalize the idea of a parity bit by appending a (small) constant amount of message-specific redundancy. Both the data and the checksum are transmitted to a receiver, at which point the same redundancy computation is carried out on the received data and compared to the received checksum. Checksums can be used either for error detection or in association with higher-level error-recovery strategies (e.g., protocols involving acknowledgements and retransmission upon failure). Trivial examples include an arithmetic checksum (compute the running 32-bit sum of all 32-bit data words, discarding high-order carries), and a simple XOR (XOR all 32-bit words in a data string). *Error-correcting codes* go one step further than error-detecting codes, offering the capability to actually correct a limited number of errors without retransmission; this is sometimes called *forward error correction*.

9.80 Example (*CRCs*) *Cyclic redundancy codes* or CRCs are commonly used checksums. A k-bit CRC algorithm maps arbitrary length inputs into k-bit imprints, and provides significantly better error-detection capability than k-bit arithmetic checksums. The algorithm is based on a carefully chosen $(k+1)$-bit vector represented as a binary polynomial; for $k = 16$, a commonly used polynomial (CRC-16) is $g(x) = 1 + x^2 + x^{15} + x^{16}$. A t-bit data input is represented as a binary polynomial $d(x)$ of degree $t - 1$, and the CRC-value corresponding to $d(x)$ is the 16-bit string represented by the polynomial remainder $c(x)$ when $x^{16} \cdot d(x)$ is divided by $g(x)$;[6] polynomial remaindering is analogous to computing integer remainders by long division. For all messages $d(x)$ with $t < 32\ 768$, CRC-16 can detect all errors that consist of only a single bit, two bits, three bits, or any odd number of bits, all burst errors of bitlength 16 or less, 99.997% $(1 - 2^{-15})$ of 17-bit burst errors, and 99.998% $(1 - 2^{-16})$ of all bursts 18 bits or longer. (A *burst error* of bitlength b is any bitstring of exactly b bits beginning and ending with a 1.) Analogous to the integer case, other data strings $d'(x)$ yielding the same remainder as $d(x)$ can be trivially found by adding multiples of the divisor $g(x)$ to $d(x)$, or inserting extra blocks representing a multiple of $g(x)$. CRCs thus do not provide one-wayness as required for MDCs; in fact, CRCs are a class of linear (error correcting) codes, with one-wayness comparable to an XOR-sum. □

While of use for detection of random errors, k-bit checksums are not of cryptographic use, because typically a data string checksumming to any target value can be easily created. One method is to simply insert or append to any data string of choice a k-bit correcting-block c which has the effect of correcting the overall checksum to the desired value. For example, for the trivial XOR checksum, if the target checksum is c', insert as block c the XOR of c' and the XOR of all other blocks.

In contrast to checksums, data integrity mechanisms based on (cryptographic) hash functions are specifically designed to preclude undetectable intentional modification. The hash-values resulting are sometimes called *integrity check values (ICV)*, or *cryptographic check values* in the case of keyed hash functions. Semantically, it should not be possible for an adversary to take advantage of the willingness of users to associate a given hash output with a single, specific input, despite the fact that each such output typically corresponds to a large set of inputs. Hash functions should exhibit no predictable relationships or correlations between inputs and outputs, as these may allow adversaries to orchestrate unintended associations.

[6] A modification is typically used in practice (e.g., complementing $c(x)$) to address the combination of an input $d(x) = 0$ and a "stuck-at-zero" communications fault yielding a successful CRC check.

9.6.3 Data integrity using a MAC alone

Message Authentication Codes (MACs) as discussed earlier are designed specifically for applications where data integrity (but not necessarily privacy) is required. The originator of a message x computes a MAC $h_k(x)$ over the message using a secret MAC key k shared with the intended recipient, and sends both (effectively $x \mid\mid h_k(x)$). The recipient determines by some means (e.g., a plaintext identifier field) the claimed source identity, separates the received MAC from the received data, independently computes a MAC over this data using the shared MAC key, and compares the computed MAC to the received MAC. The recipient interprets the agreement of these values to mean the data is authentic and has integrity – that is, it originated from the other party which knows the shared key, and has not been altered in transit. This corresponds to Figure 9.8(a).

9.6.4 Data integrity using an MDC and an authentic channel

The use of a secret key is not essential in order to provide data integrity. It may be eliminated by hashing a message and protecting the authenticity of the hash via an authentic (but not necessarily private) channel. The originator computes a hash-code using an MDC over the message data, transmits the data to a recipient over an unsecured channel, and transmits the hash-code over an independent channel known to provide data origin authentication. Such authentic channels may include telephone (authenticity through voice recognition), any data medium (e.g., floppy disk, piece of paper) stored in a trusted place (e.g., locked safe), or publication over any difficult-to-forge public medium (e.g., daily newspaper). The recipient independently hashes the received data, and compares the hash-code to that received. If these values agree, the recipient accepts the data as having integrity. This corresponds to Figure 9.8(c).

Example applications include virus protection of software, and distribution of software or public keys via untrusted networks. For virus checking of computer source or object code, this technique is preferable to one resulting in encrypted text. A common example of combining an MDC with an authentic channel to provide data integrity is digital signature schemes such as RSA, which typically involve the use of MDCs, with the asymmetric signature providing the authentic channel.

9.6.5 Data integrity combined with encryption

Whereas digital signatures provide assurances regarding both integrity and authentication, in general, encryption alone provides neither. This issue is first examined, and then the question of how hash functions may be employed in conjunction with encryption to provide data integrity.

(i) Encryption alone does not guarantee data integrity

A common misconception is that encryption provides data origin authentication and data integrity, under the argument that if a message is decrypted with a key shared only with party A, and if the decrypted message is meaningful, then it must have originated from A. Here "meaningful" means the message contains sufficient redundancy or meets some other a priori expectation. While the intuition is that an attacker must know the secret key in order to manipulate messages, this is not always true. In some cases he may be able to choose the plaintext message, while in other cases he may be able to effectively manipulate plaintext despite not being able to control its specific content. The extent to which encrypted messages can be manipulated undetectably depends on many factors, as illustrated by the following examples.

9.81 Example (*re-ordering ECB blocks*) The ciphertext blocks of any block cipher used only in ECB mode are subject to re-ordering. □

9.82 Example (*encryption of random data*) If the plaintext corresponding to a given ciphertext contains no redundancy (e.g., a random key), then all attempted decryptions thereof are meaningful, and data integrity cannot be verified. Thus, some form of redundancy is always required to allow verification of integrity; moreover, to facilitate verification in practice, explicit redundancy verifiable by automated means is required. □

9.83 Example (*bit manipulations in additive stream ciphers*) Despite the fact that the one-time pad offers unconditional secrecy, an attacker can change any single bit of plaintext by modifying the corresponding bit of ciphertext. For known-plaintext attacks, this allows an attacker to substitute selected segments of plaintext by plaintext of his own choosing. An example target bit is the high-order bit in a numeric field known to represent a dollar value. Similar comments apply to any additive stream cipher, including the OFB mode of any block cipher. □

9.84 Example (*bit manipulation in DES ciphertext blocks*) Several standard modes of operation for any block cipher are subject to selective bit manipulation. Modifying the last ciphertext block in a CFB chain is undetectable. A ciphertext block in CFB mode which yields random noise upon decryption is an indication of possible selective bit-manipulation of the preceding ciphertext block. A ciphertext block in CBC mode which yields random noise upon decryption is an indication of possible selective bit-manipulation of the following ciphertext block. For further discussion regarding error extension in standard modes of operation, see §7.2.2. □

(ii) Data integrity using encryption and an MDC

If both confidentiality and integrity are required, then the following data integrity technique employing an m-bit MDC h may be used. The originator of a message x computes a hash value $H = h(x)$ over the message, appends it to the data, and encrypts the augmented message using a symmetric encryption algorithm E with shared key k, producing ciphertext

$$C = E_k(x \mid\mid h(x)) \tag{9.2}$$

(Note that this differs subtlely from enciphering the message and the hash separately as $(E_k(x), E_k(h(x)))$, which e.g. using CBC requires two IVs.) This is transmitted to a recipient, who determines (e.g., by a plaintext identifier field) which key to use for decryption, and separates the recovered data x' from the recovered hash H'. The recipient then independently computes the hash $h(x')$ of the received data x', and compares this to the recovered hash H'. If these agree, the recovered data is accepted as both being authentic and having integrity. This corresponds to Figure 9.8(b).

The intention is that the encryption protects the appended hash, and that it be infeasible for an attacker without the encryption key to alter the message without disrupting the correspondence between the decrypted plaintext and the recovered MDC. The properties required of the MDC here may be notably weaker, in general, than for an MDC used in conjunction with, say, digital signatures. Here the requirement, effectively a joint condition on the MDC and encryption algorithm, is that it not be feasible for an adversary to manipulate or create new ciphertext blocks so as to produce a new ciphertext C' which upon decryption will yield plaintext blocks having the same MDC as that recovered, with probability significantly greater than 1 in 2^m.

9.85 Remark (*separation of integrity and privacy*) While this approach appears to separate privacy and data integrity from a functional viewpoint, the two are not independent with respect to security. The security of the integrity mechanism is, at most, that of the encryption algorithm regardless of the strength of the MDC (consider exhaustive search of the encryption key). Thought should, therefore, be given to the relative strengths of the components.

9.86 Remark (*vulnerability to known-plaintext attack*) In environments where known-plaintext attacks are possible, the technique of equation (9.2) should not be used in conjunction with additive stream ciphers unless additional integrity techniques are used. In this scenario, an attacker can recover the key stream, then make plaintext changes, recompute a new MDC, and re-encrypt the modified message. Note this attack compromises the manner in which the MDC is used, rather than the MDC or encryption algorithm directly.

If confidentiality is not essential other than to support the requirement of integrity, an apparent option is to encrypt only either the message x or the MDC $h(x)$. Neither approach is common, for reasons including Remark 9.85, and the general undesirability to utilize encryption primitives in systems requiring only integrity or authentication services. The following further comments apply:

1. *encrypting the hash-code only*: $(x,\ E_k(h(x)))$
Applying the key to the hash-value only (cf. Example 9.65) results in a property (typical for public-key signatures but) atypical for MACs: pairs of inputs x, x' with colliding outputs (MAC-values here) can be verifiably pre-determined without knowledge of k. Thus h must be collision-resistant. Other issues include: pairs of inputs having the same MAC-value under one key also do under other keys; if the blocklength of the cipher E_k is less than the bitlength m of the hash-value, splitting the latter across encryption blocks may weaken security; k must be reserved exclusively for this integrity function (otherwise chosen-text attacks on encryption allow selective MAC forgery); and E_k must not be an additive stream cipher (see Remark 9.86).
2. *encrypting the plaintext only*: $(E_k(x),\ h(x))$
This offers little computational savings over encrypting both message and hash (except for very short messages) and, as above, $h(x)$ must be collision-resistant and thus twice the typical MAC bitlength. Correct guesses of the plaintext x may be confirmed (candidate values x' for x can be checked by comparing $h(x')$ to $h(x)$).

(iii) Data integrity using encryption and a MAC

It is sometimes suggested to use a MAC rather than the MDC in the mechanism of equation (9.2) on page 365. In this case, a MAC algorithm $h_{k'}$ replaces the MDC h, and rather than $C = E_k(x \,||\, h(x))$, the message sent is

$$C' = E_k(x \,||\, h_{k'}(x)) \qquad (9.3)$$

The use of a MAC here offers the advantage (over an MDC) that should the encryption algorithm be defeated, the MAC still provides integrity. A drawback is the requirement of managing both an encryption key and a MAC key. Care must be exercised to ensure that dependencies between the MAC and encryption algorithms do not lead to security weaknesses, and as a general recommendation these algorithms should be independent (see Example 9.88).

9.87 Remark (*precluding exhaustive MAC search*) Encryption of the MAC-value in equation (9.3) precludes an exhaustive key search attack on the MAC key.

Two alternatives here include encrypting the plaintext first and then computing a MAC over the ciphertext, and encrypting the message and MAC separately. These are discussed in turn.

1. *computing a MAC over the ciphertext*: $(E_k(x),\ h_{k'}(E_k(x)))$.
   This allows message authentication without knowledge of the plaintext x (or ciphertext key). However, as the message authentication is on the ciphertext rather than the plaintext directly, there are no guarantees that the party creating the MAC knew the plaintext x. The recipient, therefore, must be careful about conclusions drawn – for example, if E_k is public-key encryption, the originator of x may be independent of the party sharing the key k' with the recipient.
2. *separate encryption and MAC*: $(E_k(x),\ h_{k'}(x))$.
   This alternative requires that neither the encryption nor the MAC algorithm compromises the objectives of the other. In particular, in this case an additional requirement on the algorithm is that the MAC on x must not compromise the confidentiality of x (cf. Definition 9.7). Keys (k, k') should also be independent here, e.g., to preclude exhaustive search on the weaker algorithm compromising the other (cf. Example 9.88). If k and k' are not independent, exhaustive key search is theoretically possible even without known plaintext.

(iv) Data integrity using encryption – examples

9.88 Example (*improper combination of CBC-MAC and CBC encryption*) Consider using the data integrity mechanism of equation (9.3) with E_k being CBC-encryption with key k and initialization vector IV, $h_{k'}(x)$ being CBC-MAC with k' and IV', and $k = k'$, $IV = IV'$. The data $x = x_1x_2 \dots x_t$ can then be processed in a single CBC pass, since the CBC-MAC is equal to the last ciphertext block $c_t = E_k(c_{t-1} \oplus x_t)$, and the last data block is $x_{t+1} = c_t$, yielding final ciphertext block $c_{t+1} = E_k(c_t \oplus x_{t+1}) = E_k(0)$. The encrypted MAC is thus independent of both plaintext and ciphertext, rendering the integrity mechanism completely insecure. Care should thus be taken in combining a MAC with an encryption scheme. In general, it is recommended that distinct (and ideally, independent) keys be used. In some cases, one key may be derived from the other by a simple technique; a common suggestion for DES keys is complementation of every other nibble. However, arguments favoring independent keys include the danger of encryption algorithm weaknesses compromising authentication (or vice-versa), and differences between authentication and encryption keys with respect to key management life cycle. See also Remark 13.32. □

An efficiency drawback in using distinct keys for secrecy and integrity is the cost of two separate passes over the data. Example 9.89 illustrates a proposed data integrity mechanism (which appeared in a preliminary draft of U.S. Federal Standard 1026) which attempts this by using an essentially zero-cost linear checksum; it is, however, insecure.

9.89 Example (*CBC encryption with XOR checksum – CBCC*) Consider using the data integrity mechanism of equation (9.2) with E_k being CBC-encryption with key k, $x = x_1x_2 \dots x_t$ a message of t blocks, and as MDC function the simple XOR of all plaintext blocks, $h(x) = \bigoplus_{i=1}^{i=t} x_i$. The quantity $M = h(x)$ which serves as MDC then becomes plaintext block x_{t+1}. The resulting ciphertext blocks using CBC encryption with $c_0 = IV$ are $c_i = E_k(x_i \oplus c_{i-1})$, $1 \le i \le t+1$. In the absence of manipulation, the recovered plaintext is $x_i = c_{i-1} \oplus D_k(c_i)$. To see that this scheme is insecure as an integrity mechanism, let c'_i denote the actual ciphertext blocks received by a recipient, resulting from possibly manipulated blocks c_i, and let x'_i denote the plaintext recovered by the recipient by CBC decryption with the proper IV. The MDC computed over the recovered plaintext blocks is then

$$M' = h(x') = \bigoplus_{i=1}^{i=t} x'_i = \bigoplus_{i=1}^{i=t} (c'_{i-1} \oplus D_k(c'_i)) = IV \oplus (\bigoplus_{i=1}^{i=t-1} c'_i) \oplus (\bigoplus_{i=1}^{i=t} D_k(c'_i)\ )$$

M' is compared for equality with $x'_{t+1} (= c'_t \oplus D_k(c'_{t+1}))$ as a check for data integrity, or equivalently, that $S = M' \oplus x'_{t+1} = 0$. By construction, $S = 0$ if there is no manipulation (i.e., if $c'_i = c_i$, which implies $x'_i = x_i$). Moreover, the sum S is invariant under any permutation of the values $c'_i, 1 \le i \le t$ (since $D_k(c_{t+1})$ appears as a term in S, but c_{t+1} does not, c_{t+1} must be excluded from the permutable set). Thus, any of the first t ciphertext blocks can be permuted without affecting the successful verification of the MDC. Furthermore, insertion into the ciphertext stream of any random block c^*_j twice, or any set of such pairs, will cancel itself out in the sum S, and thus also cannot be detected. □

9.90 Example (*CBC encryption with mod* $2^n - 1$ *checksum*) Consider as an alternative to Example 9.89 the simple MDC function $h(x) = \sum_{i=1}^{t} x_i$, the sum of plaintext blocks as n-bit integers with wrap-around carry (add overflow bits back into units bit), i.e., the sum modulo $2^n - 1$; consider $n = 64$ for ciphers of blocklength 64. The sum S from Example 9.89 in this case involves both XOR and addition modulo $2^n - 1$; both permutations of ciphertext blocks and insertions of pairs of identical random blocks are now detected. (This technique should not, however, be used in environments subject to chosen-plaintext attack.) □

9.91 Example (*PCBC encryption with mod* 2^n *checksum*) A non-standard, non-self-synchronizing mode of DES known as *plaintext-ciphertext block chaining* (PCBC) is defined as follows, for $i \ge 0$ and plaintext $x = x_1 x_2 \ldots x_t$: $c_{i+1} = E_k(x_{i+1} \oplus G_i)$ where $G_0 = IV$, $G_i = g(x_i, c_i)$ for $i \ge 1$, and g a simple function such as $g(x_i, c_i) = (x_i + c_i)$ mod 2^{64}. A one-pass technique providing both encryption and integrity, which exploits the error-propagation property of this mode, is as follows. Append an additional plaintext block to provide redundancy, e.g., $x_{t+1} = IV$ (alternatively: a fixed constant or x_1). Encrypt all blocks of the augmented plaintext using PCBC encryption as defined above. The quantity $c_{t+1} = E_k(x_{t+1} \oplus g(x_t, c_t))$ serves as MAC. Upon decipherment of c_{t+1}, the receiver accepts the message as having integrity if the expected redundancy is evident in the recovered block x_{t+1}. (To avoid a known-plaintext attack, the function g in PCBC should not be a simple XOR for this integrity application.) □

9.7 Advanced attacks on hash functions

A deeper understanding of hash function security can be obtained through consideration of various general attack strategies. The resistance of a particular hash function to known general attacks provides a (partial) measure of security. A selection of prominent attack strategies is presented in this section, with the intention of providing an introduction sufficient to establish that designing (good) cryptographic hash functions is not an easily mastered art. Many other attack methods and variations exist; some are general methods, while others rely on peculiar properties of the internal workings of specific hash functions.

9.7.1 Birthday attacks

Algorithm-independent attacks are those which can be applied to any hash function, treating it as a black-box whose only significant characteristics are the output bitlength n (and MAC key bitlength for MACs), and the running time for one hash operation. It is typically assumed the hash output approximates a uniform random variable. Attacks falling under this category include those based on hash-result bitsize (page 336); exhaustive MAC key search (page 336); and birthday attacks on hash functions (including memoryless variations) as discussed below.

(i) Yuval's birthday attack on hash functions

Yuval's birthday attack was one of the first (and perhaps the most well-known) of many cryptographic applications of the birthday paradox arising from the classical occupancy distribution (§2.1.5): when drawing elements randomly, with replacement, from a set of N elements, with high probability a repeated element will be encountered after O($\sqrt{N}$) selections. Such attacks are among those called *square-root attacks*.

The relevance to hash functions is that it is easier to find collisions for a one-way hash function than to find pre-images or second preimages of specific hash-values. As a result, signature schemes which employ one-way hash functions may be vulnerable to Yuval's attack outlined below. The attack is applicable to all unkeyed hash functions (cf. Fact 9.33), with running time $O(2^{m/2})$ varying with the bitlength m of the hash-value.

9.92 Algorithm Yuval's birthday attack

INPUT: legitimate message x_1; fraudulent message x_2; m-bit one-way hash function h.
OUTPUT: x_1', x_2' resulting from minor modifications of x_1, x_2 with $h(x_1') = h(x_2')$ (thus a signature on x_1' serves as a valid signature on x_2').

1. Generate $t = 2^{m/2}$ minor modifications x_1' of x_1.
2. Hash each such modified message, and store the hash-values (grouped with corresponding message) such that they can be subsequently searched on hash-value. (This can done in O(t) total time using conventional hashing.)
3. Generate minor modifications x_2' of x_2, computing $h(x_2')$ for each and checking for matches with any x_1' above; continue until a match is found. (Each table lookup will require constant time; a match can be expected after about t candidates x_2'.)

9.93 Remark (*application of birthday attack*) The idea of this attack can be used by a dishonest signer who provides to an unsuspecting party his signature on x_1' and later repudiates signing that message, claiming instead that the message signed was x_2'; or by a dishonest verifier, who is able to convince an unsuspecting party to sign a prepared message x_1', and later claim that party's signature on x_2'. This remark generalizes to other schemes in which the hash of a message is taken to represent the message itself.

Regarding practicality, the collisions produced by the birthday attack are "real" (vs. pseudo-collisions or compression function collisions), and moreover of direct practical consequence when messages are constructed to be meaningful. The latter may often be done as follows: alter inputs via individual minor modifications which create semantically equivalent messages (e.g., substituting tab characters in text files for spaces, unprintable characters for each other, etc.). For 128-bit hash functions, 64 such potential modification points are required to allow 2^{64} variations. The attack then requires O(2^{64}) time (feasible with extreme parallelization); and while it requires space for O(2^{64}) messages (which is impractical), the memory requirement can be addressed as discussed below.

(ii) Memoryless variation of birthday attack

To remove the memory requirement of Algorithm 9.92, a deterministic mapping may be used which approximates a random walk through the hash-value space. By the birthday paradox, in a random walk through a space of 2^m points, one expects to encounter some point a second time (i.e., obtain a collision) after $O(2^{m/2})$ steps, after which the walk will repeat its previous path (and begin to cycle). General memoryless cycle-finding techniques may then be used to find this collision. (Here *memoryless* means requiring negligible memory, rather than in the stochastic sense.) These include Floyd's cycle-finding algorithm (§3.2.2) and improvements to it.

Following Algorithm 9.92, let g be a function such that $g(x_1, H) = x'_1$ is a minor modification, determined by the hash-value H, of message x_1 (each bit of H might define whether or not to modify x_1 at a pre-determined modification point). If x_1 is fixed, then g essentially maps a hash-result to a message and it is convenient to write $g_{x_1}(H) = x'_1$. Moreover, let g be injective so that distinct hashes H result in distinct x'_1. Then, with fixed messages x_1, x_2, and using some easily distinguishable property (e.g., parity) which splits the space of hash-values into two roughly equal-sized subsets, define a function r mapping hash-results to hash-results by:

$$r(H) = \begin{cases} h(g_{x_1}(H)) & \text{if } H \text{ is even} \\ h(g_{x_2}(H)) & \text{if } H \text{ is odd} \end{cases} \tag{9.4}$$

The memoryless collision search technique (see above) is then used to find two inputs to r which map to the same output (i.e., collide). If h behaves statistically as a random mapping then, with probability 0.5, the parity will differ in H and H' for the colliding inputs, in which case without loss of generality $h(g_{x_1}(H)) = h(g_{x_2}(H'))$. This yields a colliding pair of variations $x'_1 = g_{x_1}(H)$, $x'_2 = g_{x_2}(H')$ of distinct messages x_1, x_2, respectively, such that $h(x'_1) = h(x'_2)$, as per the output of Algorithm 9.92.

(iii) Illustrative application to MD5

Actual application of the above generic attack to a specific hash function raises additional technicalities. To illustrate how these may be addressed, such application is now examined, with assumptions and choices made for exposition only. Let h be an iterated hash function processing messages in 512-bit blocks and producing 128-bit hashes (e.g., MD5, RIPEMD-128). To minimize computational expense, restrict r (effectively g and h) in equation (9.4) to single 512-bit blocks of x_i, such that each iteration of r involves only the compression function f on inputs one message block and the current chaining variable.

Let the legitimate message input x_1 consist of s 512-bit blocks ($s \geq 1$, prior to MD-strengthening). Create a fraudulent message x_2 of equal bitlength. Allow x_2 to differ from x_1 up to and including the j^{th} block, for any fixed $j \leq s-1$. Use the $(j+1)^{\text{st}}$ block of x_i, denoted B_i ($i = 1, 2$), as a matching/replacement block, to be replaced by the 512-bit blocks resulting from the collision search. Set all blocks in x_2 subsequent to B_i identically equal to those in x_1; x'_i will then differ from x_i only in the single block $(j+1)$. For maximum freedom in the construction of x_2, choose $j = s-1$. Let c_1, c_2 be the respective 128-bit intermediate results (chaining variables) after the iterated hash operates on the first j blocks of x_1, x_2. Compression function f maps $(128 + 512 =)$ 640-bit inputs to 128-bit outputs. Since the chaining variables depend on x_i, $g_{x_i}(= g)$ may be defined independent of x_i here (cf. equation (9.4)); assume both entire blocks B_i may be replaced without practical implication. Let $g(H) = B$ denote an injective mapping from the space of 128-bit hash-values to the space of 512-bit potential replacement blocks, defined as follows: map each two-bit segment of H to one of four 8-bit values in the replacement block B. (A practical motivation for this is that if x_i is an ASCII message to be printed, and the four 8-bit values are selected to represent non-printable characters, then upon printing, the resulting blocks B are all indistinguishable, leaving no evidence of adversarial manipulation.)

The collision-finding function r for this specific example (corresponding to the generic equation (9.4)) is then:

$$r(H) = \begin{cases} f(c_1,\ g(H)) & \text{if } H \text{ is even} \\ f(c_2,\ g(H)) & \text{if } H \text{ is odd} \end{cases}$$

Collisions for MD5 (and similar hash functions) can thus be found in $O(2^{64})$ operations and without significant storage requirements.

9.7.2 Pseudo-collisions and compression function attacks

The exhaustive or brute force methods discussed in §9.3.4, producing preimages, 2nd-preimages, and collisions for hash functions, are always theoretically possible. They are not considered true "attacks" unless the number of operations required is significantly less than both the strength conjectured by the hash function designer and that of hash functions of similar parameters with ideal strength. An attack requiring such a reduced number of operations is informally said to *break* the hash function, whether or not this computational effort is feasible in practice. Any attack method which demonstrates that conjectured properties do not hold must be taken seriously; when this occurs, one must admit the possibility of additional weaknesses.

In addition to considering the complexity of finding (ordinary) preimages and collisions, it is common to examine the feasibility of attacks on slightly modified versions of the hash function in question, for reasons explained below. The most common case is examination of the difficulty of finding preimages or collisions if one allows free choice of IVs. Attacks on hash functions with unconstrained IVs dictate upper bounds on the security of the actual algorithms. Vulnerabilities found, while not direct weaknesses in the overall hash function, are nonetheless considered certificational weaknesses and cast suspicion on overall security. In some cases, restricted attacks can be extended to full attacks by standard techniques.

Table 9.11 lists the most commonly examined variations, including *pseudo-collisions* – collisions allowing different IVs for the different message inputs. In contrast to preimages and collisions, pseudo-preimages and pseudo-collisions are of limited direct practical significance.

9.94 Note (*alternate names for collision and preimage attacks*) Alternate names for those in Table 9.11 are as follows: preimage or 2nd-preimage $\equiv$ *target attack*; pseudo-preimage $\equiv$ *free-start target attack*; collision (fixed IV) $\equiv$ *collision attack*; collision (random IV) $\equiv$ *semi-free-start collision attack*; pseudo-collision $\equiv$ *free-start collision attack*.

9.95 Note (*relative difficulty of attacks*) Finding a collision can be no harder than finding a 2nd-preimage. Similarly, finding a pseudo-collision can be no harder than finding (two distinct) pseudo-preimages.

↓Type of attack	V	V'	x	x'	y	Find ...
preimage	V_0	—	$*$	—	y_0	x: $h(V_0, x) = y_0$
pseudo-preimage	$*$	—	$*$	—	y_0	x, V: $h(V, x) = y_0$
2nd-preimage	V_0	V_0	x_0	$*$	$h(V_0, x_0)$	x': $h(V_0, x_0) = h(V_0, x')$
collision (fixed IV)	V_0	V_0	$*$	$*$	—	x, x': $h(V_0, x) = h(V_0, x')$
collision (random IV)	$*$	V	$*$	$*$	—	x, x', V: $h(V, x) = h(V, x')$
pseudo-collision	$*$	$*$	$*$	$*$	—	x, x', V, V': $h(V, x) = h(V', x')$

Table 9.11: *Definition of preimage and collision attacks. V and V' denote (potentially different) IVs used for MDC h applied to inputs x and x', respectively; V_0 denotes the IV pre-specified in the definition of h, x_0 a pre-specified target input, and $y = y_0$ a pre-specified target output. * Denotes IVs or inputs which may be freely chosen by an attacker; $h(V_0, x_0)$ denotes the hash-code resulting from applying h with fixed IV $V = V_0$ to input $x = x_0$. — Means not applicable.*

9.96 Example (*trivial collisions for random IVs*) If free choice of IV is allowed, then trivial pseudo-collisions can be found by deleting leading blocks from a target message. For example, for an iterated hash (cf. equation (9.1) on page 333), $h(IV, x_1x_2) = f(f(IV, x_1), x_2)$. Thus, for $IV' = f(IV, x_1)$, $h(IV', x_2) = h(IV, x_1x_2)$ yields a pseudo-collision of h, independent of the strength of f. (MD-strengthening as per Algorithm 9.26 precludes this.) □

Another common analysis technique is to consider the strength of weakened variants of an algorithm, or attack specific subcomponents, akin to cryptanalyzing an 8-round version of DES in place of the full 16 rounds.

9.97 Definition An *attack on the compression function* of an iterated hash function is any attack as per Table 9.11 with $f(H_{i-1}, x_i)$ replacing $h(V_0, x)$ – the compression function f in place of hash function h, chaining variable H_{i-1} in place of initializing value V, and a single input block x_i in place of the arbitrary-length message x.

An attack on a compression function focuses on one fixed step i of the iterative function of equation (9.1). The entire message consists of a single block $x_i = x$ (without MD-strengthening), and the hash output is taken to be the compression function output so $h(x) = H_i$. The importance of such attacks arises from the following.

9.98 Note (*compression function vs. hash function attacks*) Any of the six attacks of Table 9.11 which is found for the compression function of an iterated hash can be extended to a similar attack of roughly equal complexity on the overall hash. An iterated hash function is thus in this regard at most as strong as its compression function. (However note, for example, an overall pseudo-collision is not always of practical concern, since most hash functions specify a fixed IV.)

For example, consider a message $x = x_1x_2 \dots x_t$. Suppose a successful 2nd-preimage attack on compression function f yields a 2nd-preimage $x'_1 \neq x_1$ such that $f(IV, x'_1) = f(IV, x_1)$. Then, $x' = x'_1x_2 \dots x_t$ is a preimage of $h(x)$.

More positively, if MD-strengthening is used, the strength of an iterated hash with respect to the attacks of Table 9.11 is the same as that of its compression function (cf.

Fact 9.24). However, an iterated hash may certainly be weaker than its compression function (e.g., Example 9.96; Fact 9.37).

In summary, a compression function secure against preimage, 2nd-preimage, and collision (fixed IV) attacks is necessary and sometimes, but not always, sufficient for a secure iterated hash; and security against the other (i.e., free-start) attacks of Table 9.11 is desirable, but not always necessary for a secure hash function in practice. For this reason, compression functions are analyzed in isolation, and attacks on compression functions as per Definition 9.97 are considered. A further result motivating the study of pseudo-preimages is the following.

9.99 Fact (*pseudo-preimages yielding preimages*) If the compression function f of an n-bit iterated hash function h does not have ideal computational security (2^n) against pseudo-preimage attacks, then preimages for h can be found in fewer than 2^n operations (cf. §9.3.4, Table 9.2). This result is true even if h has MD-strengthening.

Justification. The attack requires messages of 3 or more blocks, with 2 or more unconstrained to allow a meet-in-the-middle attack (page 374). If pseudo-preimages can be found in 2^s operations, then $2^{(n+s)/2}$ forward points and $2^{(n-s)/2}$ backward points are employed (fewer backward points are used since they are more costly). Preimages can thus be found in $2 \cdot 2^{(n+s)/2}$ operations.

9.7.3 Chaining attacks

Chaining attacks are those which are based on the iterative nature of hash functions and, in particular, the use of chaining variables. These focus on the compression function f rather than the overall hash function h, and may be further classified as below. An example for context is first given.

9.100 Example (*chaining attack*) Consider a (candidate) collision resistant iterative hash function h producing a 128-bit hash-result, with a compression function f taking as inputs a 512-bit message block x_i and 128-bit chaining variable H_i ($H_0 = IV$) and producing output $H_{i+1} = f(H_i, x_i)$. For a fixed 10-block message x (640 bytes), consider $H = h(x)$. Suppose one picks any one of the 10 blocks, and wishes to replace it with another block without affecting the hash H. If h behaves like a random mapping, the number of such 512-bit blocks is approximately $2^{512}/2^{128} = 2^{384}$. Any efficient method for finding any one of these 2^{384} blocks distinct from the original constitutes an attack on h. The challenge is that such blocks are a sparse subset of all possible blocks, about 1 in 2^{128}. □

(i) Correcting-block chaining attacks

Using the example above for context, one could attempt to (totally) replace a message x with a new message x', such that $h(x) = h(x')$, by using a single unconstrained "correcting" block in x', designated ahead of time, to be determined later such that it produces a chaining value which results in the overall hash being equal to target value $h(x)$. Such a *correcting block attack* can be used to find both preimages and collisions. If the unconstrained block is the first (last) block in the message, it is called a *correcting first (last) block attack*. These attacks may be precluded by requiring per-block redundancy, but this results in an undesirable bandwidth penalty. Example 9.101 illustrates a correcting first block attack. The extension of Yuval's birthday attack (page 369), with message alterations restricted to the last block of candidate messages, resembles a correcting last block attack applied simultaneously to two messages, seeking a (birthday) collision rather than a fixed overall target hash-value.

9.101 Example (*correcting block attack on CBC cipher mode*) The CBC mode of encryption with non-secret key ($H_0 = IV$; $H_i = E_k(H_{i-1} \oplus x_i)$) is unsuitable as an MDC algorithm, because it fails to be one-way – the compression function is reversible when the encryption key is known. A message x', of unconstrained length (say t blocks) can be constructed to have any specified target hash-value H as follows. Let $x'_2, \dots x'_t$ be $t-1$ blocks chosen freely. Set $H'_t \leftarrow H$, then for i from t to 1 compute $H'_{i-1} \leftarrow D_k(H'_i) \oplus x'_i$. Finally, compute $x^*_1 \leftarrow D_k(H'_1) \oplus IV$. Then, for $x' = x^*_1 x'_2 \dots x'_t$, $h(x') = H$ and all but block x^*_1 (which will appear random) can be freely chosen by an adversary; even this minor drawback can be partially addressed by a meet-in-the-middle strategy (see below). Analogous remarks apply to the CFB mode. □

(ii) Meet-in-the-middle chaining attacks

These are birthday attacks similar to Yuval's (and which can be made essentially memoryless) but which seek collisions on intermediate results (i.e., chaining variables) rather than the overall hash-result. When applicable, they allow (unlike Yuval's attack) one to find a message with a pre-specified hash-result, for either a 2nd-preimage or a collision. An attack point is identified between blocks of a candidate (fraudulent) message. Variations of the blocks preceding and succeeding this point are generated. The variations are hashed forward from the algorithm-specified IV (computing $H_i = f(H_{i-1}, x_i)$ as usual) and backward from the target final hash-result (computing $H_i = f^{-1}(H_{i+1}, x_{i+1})$ for some H_{i+1}, x_{i+1}, ideally for x_{i+1} chosen by the adversary), seeking a collision in the chaining variable H_i at the attack point. For the attack to work, the attacker must be able to efficiently go backwards through the chain (certainly moreso than by brute force – e.g., see Example 9.102), i.e., invert the compression function in the following manner: given a value H_{i+1}, find a pair (H_i, x_{i+1}) such that $f(H_i, x_{i+1}) = H_{i+1}$.

9.102 Example (*meet-in-the-middle attack on invertible key chaining modes*) Chaining modes which allow easily derived stage keys result in reversible compression functions unsuitable for use in MDCs due to lack of one-wayness (cf. Example 9.101). An example of such *invertible key chaining* methods is Bitzer's scheme: $H_0 = IV$, $H_i = f(H_{i-1}, x_i) = E_{k_i}(H_{i-1})$ where $k_i = x_i \oplus s(H_{i-1})$ and $s(H_{i-1})$ is a function mapping chaining variables to the key space. For exposition, let s be the identity function. This compression function is unsuitable because it falls to a meet-in-the-middle attack as outlined above. The ability to move backwards through chaining variables, as required by such an attack, is possible here with the chaining variable H_i computed from H_{i+1} as follows. Choose a fixed value $k_{i+1} \leftarrow k$, compute $H_i \leftarrow D_k(H_{i+1})$, then choose as message block $x_{i+1} \leftarrow k \oplus H_i$. □

(iii) Fixed-point chaining attacks

A *fixed point* of a compression function is a pair (H_{i-1}, x_i) such that $f(H_{i-1}, x_i) = H_{i-1}$. For such a pair of message block and chaining value, the overall hash on a message is unchanged upon insertion of an arbitrary number of identical blocks x_i at the chain point at which that chaining value arises. Such attacks are thus of concern if it can be arranged that the chaining variable has a value for which a fixed point is known. This includes the following cases: if fixed points can be found and it can be easily arranged that the chaining variable take on a specific value; or if for arbitrary chaining values H_{i-1}, blocks x_i can be found which result in fixed-points. Fixed points allow 2nd-preimages and collisions to be produced; their effect can be countered by inclusion of a trailing length-block (Algorithm 9.26).

(iv) Differential chaining attacks

Differential cryptanalysis has proven to be a powerful tool for the cryptanalysis of not only block ciphers but also of hash functions (including MACs). For multi-round block ciphers this attack method examines input differences (XORs) to round functions and the corresponding output differences, searching for statistical anomalies. For hash functions, the examination is of input differences to compression functions and the corresponding output differences; a collision corresponds to an output difference of zero.

9.7.4 Attacks based on properties of underlying cipher

The implications of certain properties of block ciphers, which may be of no practical concern when used for encryption, must be carefully examined when such ciphers are used to construct iterated hash functions. The general danger is that such properties may facilitate adversarial manipulation of compression function inputs so as to allow prediction or greater control of outputs or relations between outputs of successive iterations. Included among block cipher properties of possible concern are the following (cf. Chapter 7):

1. *complementation property*: $y = E_k(x) \iff \overline{y} = E_{\overline{k}}(\overline{x})$, where $\overline{x}$ denotes bitwise complement. This makes it trivial to find key-message pairs of block cipher inputs whose outputs differ in a pre-determined manner. For example, for such a block cipher E, the compression function $f(H_{i-1}, x_i) = E_{H_{i-1} \oplus x_i}(x_i) \oplus x_i$ (a linear transformation of the Matyas-Meyer-Oseas function) produces the same output for x_i and its bitwise complement $\overline{x_i}$.
2. *weak keys*: $E_k(E_k(x)) = x$ (for all x). This property of involution of the block cipher may allow an adversary to easily create a two-step fixed point of the compression function f in the case that message blocks x_i have direct influence on the block cipher key input (e.g., if $f = E_{x_i}(H_{i-1})$, insert 2 blocks x_i containing a weak key). The threat is similar for *semi-weak keys*, where $E_{k'}(E_k(x)) = x$.
3. *fixed points*: $E_k(x) = x$. Block cipher fixed points may facilitate fixed-point attacks if an adversary can control the block cipher key input. For example, for the Davies-Meyer compression function $f(H_{i-1}, x_i) = E_{x_i}(H_{i-1}) \oplus H_{i-1}$, if H_{i-1} is a fixed point of the block cipher for key x_i (i.e., $E_{x_i}(H_{i-1}) = H_{i-1}$), then this yields a predictable compression function output $f(H_{i-1}, x_i) = 0$.
4. *key collisions*: $E_k(x) = E_{k'}(x)$. These may allow compression function collisions.

Although they may serve as distinguishing metrics, attacks which appear purely certificational in nature should be noted separately from others; for example, fixed point attacks appear to be of limited practical consequence.

9.103 Example (*DES-based hash functions*) Consider DES as the block cipher in question (see §7.4). DES has the complementation property; has 4 weak keys and 6 pairs of semi-weak keys (each with bit 2 equal to bit 3); each weak key has 2^{32} fixed points (thus a random plaintext is a fixed point of some weak key with probability 2^{-30}), as do 4 of the semi-weak keys; and key collisions can be found in 2^{32} operations. The security implications of these properties must be taken into account in the design of any DES-based hash function. Concerns regarding both weak keys and the complementation property can be eliminated by forcing key bits 2 and 3 to be 10 or 01 within the compression function. □

9.8 Notes and further references

§9.1

The definitive reference for cryptographic hash functions, and an invaluable source for the material in this chapter (including many otherwise unattributed results), is the comprehensive treatment of Preneel [1003, 1004]; see also the surveys of Preneel [1002] and Preneel, Govaerts, and Vandewalle [1006]. Davies and Price [308] also provide a solid treatment of message authentication and data integrity. An extensive treatment of conventional hashing, including historical discussion tracing origins back to IBM in 1953, is given by Knuth [693, p.506-549]. Independent of cryptographic application, *universal classes of hash functions* were introduced by Carter and Wegman [234] in the late 1970s, the idea being to find a class of hash functions such that for every pair of inputs, the probability was low that a randomly chosen function from the class resulted in that pair colliding. Shortly thereafter, Wegman and Carter [1234] noted the cryptographic utility of these hash functions, when combined with secret keys, for (unconditionally secure) *message authentication tag systems*; they formalized this concept, earlier considered by Gilbert, MacWilliams, and Sloane [454] (predating the concept of digital signatures) who attribute the problem to Simmons. Simmons ([1138],[1144]; see also Chapter 10 of Stinson [1178]) independently developed a general theory of unconditionally secure message authentication schemes and the subject of *authentication codes* (see also §9.5 below).

Rabin [1022, 1023] first suggested employing a one-way hash function (constructed by using successive message blocks to key an iterated block encryption) in conjunction with a one-time signature scheme and later in a public-key signature scheme; Rabin essentially noted the requirements of 2nd-preimage resistance and collision resistance. Merkle [850] explored further uses of one-way hash functions for authentication, including the idea of *tree authentication* [852] for both one-time signatures and authentication of public files.

§9.2

Merkle [850] (partially published as [853]) was the first to give a substantial (informal) definition of one-way hash functions in 1979, specifying the properties of preimage and 2nd-preimage resistance. Foreshadowing UOWHFs (see below), he suggested countering the effect of Remark 9.36 by using slightly different hash functions h over time; Merkle [850, p.16-18] also proposed a public key distribution method based on a one-way hash function (effectively used as a one-way pseudo-permutation) and the birthday paradox, in a precursor to his "puzzle system" (see page 537). The first formal definition of a CRHF was given in 1988 by Damgård [295] (an informal definition was later given by Merkle [855, 854]; see also [853]), who was first to explore collision resistant hash functions in a complexity-theoretic setting. Working from the idea of *claw-resistant pairs of trapdoor permutations* due to Goldwasser, Micali, and Rivest [484], Damgård defined *claw-resistant families of permutations* (without the trapdoor property). The term *claw-resistant* (originally: *claw-free*) originates from the pictorial representation of a functional mapping showing two distinct domain elements being mapped to the same range element under distinct functions $f^{(i)}$ and $f^{(j)}$ (colliding at $z = f^{(i)}(x) = f^{(j)}(y)$), thereby tracing out a claw.

Goldwasser et al. [484] established that the intractability of factoring suffices for the existence of claw-resistant pairs of permutations. Damgård showed that the intractability of the discrete logarithm problem likewise suffices. Using several reasonably efficient number-theoretic constructions for families of claw-resistant permutations, he gave the first provably collision resistant hash functions, under such intractability assumptions (for discrete logarithms, the assumption required is that taking *specific* discrete logarithms be difficult). Russell [1088] subsequently established that a collection of collision resistant hash functions exists if and only if there exists a collection of *claw-resistant pairs of pseudo-permutations*; a pseudo-permutation on a set is a function computationally indistinguishable from a permutation (pairs of elements demonstrating non-injectivity are hard to find). It remains open whether the existence of one-way functions suffices for the existence of collision resistant hash functions.

The definition of a one-way function (Definition 9.9) was given in the seminal paper of Diffie and Hellman [345], along with the use of the discrete exponential function modulo a prime as a candidate OWF, which they credit to Gill. The idea of providing the hash-value of some data, to indicate prior commitment to (or knowledge of) that data, was utilized in Lamport's one-time signature scheme (circa 1976); see page 485. The OWF of Example 9.13 was known to Matyas and Meyer circa 1979. As noted by Massey [786], the idea of one-wayness was published in 1873 by J.S. Jevons, who noted (preceding RSA by a century) that multiplying two primes is easy whereas factoring the result is not. Published work dated 1968 records the use of ciphers essentially as one-way functions (decryption was not required) in a technique to avoid storing cleartext computer account passwords in time-shared systems. These were referred to as *one-way ciphers* by Wilkes [1244] (p.91-93 in 1968 or 1972 editions; p.147 in 1975 edition), who credits Needham with the idea and an implementation thereof. The first proposal of a non-invertible function for the same purpose appears to be that of Evans, Kantrowitz, and Weiss [375], while Purdy [1012] proposed extremely high-degree, sparse polynomials over a prime field as a class of functions which were computationally difficult to invert. Foreshadowing later research into collision resistance, Purdy also defined the *degeneracy* of such a function to be the maximum number of preimages than any image could have, noting that "if the degeneracy is catastrophically large there may be no security at all".

Naor and Yung [920] introduced the cryptographic primitive known as a *universal one-way hash function (UOWHF)* family, and give a provably secure construction for a one-way hash function from a one-way hash function which compresses by a single bit ($t+1$ to t bits); the main property of a UOWHF family is 2nd-preimage resistance as for a OWHF, but here an instance of the function is picked at random from a family of hash functions after fixing an input, as might be modeled in practice by using a random IV with a OWHF. Naor and Yung [920] also prove by construction that UOWHFs exist if and only if one-way permutations do, and show how to use UOWHFs to construct provably secure digital signature schemes assuming the existence of any one-way permutation. Building on this, Rompel [1068] showed how to construct a UOWHF family from any one-way function, and based signature schemes on such hash functions; combining this with the fact that a one-way function can be constructed from any secure signature scheme, the result is that the existence of one-way functions is necessary and sufficient for the existence of secure digital signature schemes. De Santis and Yung [318] proceed with more efficient reductions from one-way functions to UOWHFs, and show the equivalence of a number of complexity-theoretic definitions regarding collision resistance. Impagliazzo and Naor [569] give an efficient construction for a UOWHF and prove security equivalent to the subset-sum problem (an **NP**-hard problem whose corresponding decision problem is **NP**-complete); for parameters for which a random instance of subset-sum is hard, they argue that this UOWHF is secure (cf. Remark 9.12). Impagliazzo, Levin, and Luby [568] prove the existence of one-way functions is necessary and sufficient for that of secure pseudorandom generators.

Application-specific (often unprovable) hash function properties beyond collision resistance (but short of preimage resistance) may often be identified as necessary, e.g., for ordinary RSA signatures computed directly after hashing, the multiplicative RSA property dictates that for the hash function h used it be infeasible to find messages x, x_1, x_2 such that $h(x) = h(x_1) \cdot h(x_2)$. Anderson [27] discusses such additional requirements on hash functions. For a summary of requirements on a MAC in the special case of multi-cast authentication, see Preneel [1003]. Bellare and Rogaway [93] include discussion of issues related to the random nature of practical hash functions, and cryptographic uses thereof. Damgård [295] showed that the security of a digital signature scheme which is not existentially forgeable under an adaptive chosen-message attack will not be decreased if used in conjunction with a collision-resistant hash function.

Bellare, Goldreich, and Goldwasser [88] (see also [89]) introduce the idea of *incremental hashing*, involving computing a hash value over data and then updating the hash-value after changing the data; the objective is that the computation required for the update be proportional to the amount of change.

§9.3

Merkle's meta-method [854] (Algorithm 9.25) was based on ideas from his 1979 Ph.D. thesis [850]. An equivalent construction was given by Damgård [296], which Gibson [450] remarks on again yielding Merkle's method. Naor and Yung [920] give a related construction for a UOWHF. See Preneel [1003] for fundamental results (cf. Remarks 9.35 and 9.36, and Fact 9.27 on cascading hash functions which follow similar results on stream ciphers by Maurer and Massey [822]). The padding method of Algorithms 9.29 and 9.30 originated from ISO/IEC 10118-4 [608]. The basic idea of the long-message attack (Fact 9.37) is from Winternitz [1250].

§9.4

The hash function of Algorithm 9.42 and referred to as Davies-Meyer (as cited per Quisquater and Girault [1019]) has been attributed by Davies to Meyer; apparently known to Meyer and Matyas circa 1979, it was published along with Algorithm 9.41 by Matyas, Meyer, and Oseas [805]. The Miyaguchi-Preneel scheme (Algorithm 9.43) was proposed circa 1989 by Preneel [1003], and independently by Miyaguchi, Ohta, and Iwata [886]. The three single-length rate-one schemes discussed (Remark 9.44) are among 12 compression functions employing *non-invertible chaining* found through systematic analysis by Preneel et al. [1007] to be provably secure under black-box analysis, 8 being certificationally vulnerable to fixed-point attack nonetheless. These 12 are linear transformations on the message block and chaining variable (i.e., $[x', H'] = A[x, H]$ for any of the 6 invertible 2×2 binary matrices A) of the Matyas-Meyer-Oseas (Algorithm 9.41) and Miyaguchi-Preneel schemes; these latter two themselves are among the 4 recommended when the underlying cipher is resistant to differential cryptanalysis (e.g., DES), while Davies-Meyer is among the remaining 8 recommended otherwise (e.g., for FEAL). MDC-2 and MDC-4 are of IBM origin, proposed by Brachtl et al. [184], and reported by Meyer and Schilling [860]; details of MDC-2 are also reported by Matyas [803]. For a description of MDC-4, see Bosselaers and Preneel [178].

The DES-based hash function of Merkle [855] which is mentioned uses the meta-method and employs a compression function f mapping 119-bit input to 112-bit output in 2 DES operations, allowing 7-bit message blocks to be processed (with rate 0.055). An optimized version maps 234 bits to 128 bits in 6 DES operations, processing 106-bit message blocks (with rate 0.276); unfortunately, overheads related to "bit chopping" and the inconvenient block size are substantial in practice. This construction is provably as secure as the underlying block cipher assuming an unflawed cipher (cf. Table 9.3; Preneel [1003] shows that accounting for DES weak keys and complementation drops the rate slightly to 0.266). Winternitz [1250] considers the security of the Davies-Meyer hash under a black-box model (cf. Remark 9.45).

The search for secure double-length hash functions of rate 1 is ongoing, the goal being security better than single-length Matyas-Meyer-Oseas and approaching that of MDC-2. Quisquater and Girault [1019] proposed two functions, one (QG-original) appearing in the Abstracts of Eurocrypt'89 and a second (QG-revised) in the final proceedings altered to counter an attack of Coppersmith [276] on the first. The attack, restricted to the case of DES as underlying block cipher, uses fixed points resulting from weak keys to find collisions in 2^{36} DES operations. A general attack of Knudsen and Lai [688], which (unfortunately) applies to a large class of double-length (i.e., $2n$-bit) rate-one block-cipher-based hashes including QG-original, finds preimages in about 2^n operations plus 2^n storage. The systematic method used to establish this result was earlier used by Hohl et al. [560] to prove that pseudo-preimage and pseudo-collision attacks on a large class of double-length hash functions of rate 1/2 and 1, including MDC-2, are no more difficult than on the single-length rate-one Davies-Meyer hash; related results are summarized by Lai and Knudsen [727]. A second attack due to Coppersmith [276], not restricted to DES, employs 88 correcting blocks to find collisions for QG-revised in 2^{40} steps. Another modification of QG-original, the LOKI Double Hash Function (LOKI-DBH) of Brown, Pieprzyk, and Seberry [215], appears as a general construction to offer the same security as QG-revised (provided the underlying block cipher is not LOKI).

The speeds in Table 9.5 are normalized from the timings reported by Dobbertin, Bosselaers, and Preneel [355], relative to an assembly code MD4 implementation optimized for the Pentium processor, with a throughput (90 MHz clock) of 165.7 Mbit/s (optimized C code was roughly a factor of 2 slower). See Bosselaers, Govaerts, and Vandewalle [177] for a detailed MD5 implementation discussion.

MD4 and MD5 (Algorithms 9.49, 9.51) were designed by Rivest [1055, 1035]. An Australian extension of MD5 known as HAVAL has also been proposed by Zheng, Pieprzyk, and Seberry [1268]. The first published partial attack on MD4 was by den Boer and Bosselaers [324], who demonstrated collisions could be found when Round 1 (of the three) was omitted from the compression function, and confirmed unpublished work of Merkle showing that collisions could be found (for input pairs differing in only 3 bits) in under a millisecond on a personal computer if Round 3 was omitted. More devastating was the partial attack by Vaudenay [1215] on the full MD4, which provided only near-collisions, but allowed sets of inputs to be found for which, of the corresponding four 32-bit output words, three are constant while the remaining word takes on all possible 32-bit values. This revealed the word access-order in MD4 to be an unfortunate choice. Finally, late in 1995, using techniques related to those which earlier allowed a partial attack on RIPEMD (see below), Dobbertin [354] broke MD4 as a CRHF by finding not only collisions as stated in Remark 9.50 (taking a few seconds on a personal computer), but collisions for meaningful messages (in under one hour, requiring 20 free bytes at the start of the messages).

A first partial attack on MD5 was published by den Boer and Bosselaers [325], who found pseudo-collisions for its compression function f, which maps a 128-bit chaining variable and sixteen 32-bit words down to 128-bits; using 2^{16} operations, they found a 16-word message X and chaining variables $S_1 \neq S_2$ (these differing only in 4 bits, the most significant of each word), such that $f(S_1, X) = f(S_2, X)$. Because this specialized internal pseudo-collision could not be extended to an external collision due to the fixed initial chaining values (and due to the special relation between the inputs), this attack was considered by many to have little practical significance, although exhibiting a violation of the design goal to build a CRHF from a collision resistant compression function. But in May of 1996, using techniques related to his attack on MD4 above, Dobbertin (rump session, Eurocrypt'96) found MD5 compression function collisions (Remark 9.52) in 10 hours on a personal computer (about 2^{34} compress function computations).

Anticipating the feasibility of 2^{64} operations, Rivest [1055] proposed a method to extend MD4 to 256 bits by running two copies of MD4 in parallel over the input, with different initial chaining values and constants for the second, swapping the values of the variable A with the first after processing each 16-word block and, upon completion, concatenating the 128-bit hash-values from each copy. However, in October of 1995 Dobbertin [352] found collisions for the compression function of extended MD4 in 2^{26} compress function operations, and conjectured that a more sophisticated attack could find a collision for extended MD4 itself in $O(2^{40})$ operations.

MD2, an earlier and slower hash function, was designed in 1988 by Rivest; see Kaliski [1033] for a description. Rogier and Chauvaud [1067] demonstrated that collisions can be efficiently found for the compression function of MD2, and that the MD2 checksum block is necessary to preclude overall MD2 collisions.

RIPEMD [178] was designed in 1992 by den Boer and others under the European RACE Integrity Primitives Evaluation (RIPE) project. A version of MD4 strengthened to counter known attacks, its compression function has two parallel computation lines of three 16-step rounds. Nonetheless, early in 1995, Dobbertin [353] demonstrated that if the first or last (parallel) round of the 3-round RIPEMD compress function is omitted, collisions can be found in 2^{31} compress function computations (one day on a 66 MHz personal computer). This result coupled with concern about inherent limitations of 128-bit hash results motivated RIPEMD-160 (Algorithm 9.55) by Dobbertin, Bosselaers, and Preneel [355]; but for corrections, see the directory `/pub/COSIC/bosselae/ripemd/` at ftp site `ftp.esat.kuleuven.ac.be`. Increased security is provided by five rounds (each with two lines) and greater independence between the parallel lines, at a performance penalty of a factor of 2. RIPEMD-128 (with 128-bit result and chaining variable) was simultaneously proposed as a drop-in upgrade for RIPEMD; it scales RIPEMD-160 back to four rounds (each with two lines).

SHA-1 (Algorithm 9.53) is a U.S. government standard [404]. It differs from the original standard SHA [403], which it supersedes, only in the inclusion of the 1-bit rotation in the block expansion from 16 to 80 words. For discussion of how this expansion in SHA is related to linear error correcting codes, see Preneel [1004].

Lai and Massey [729] proposed two hash functions of rate 1/2 with $2m$-bit hash values, *Tandem Davies-Meyer* and *Abreast Davies-Meyer*, based on an m-bit block cipher with $2m$-bit key (e.g., IDEA), and a third m-bit hash function using a similar block cipher. Merkle's public-domain hash function Snefru [854] and the FEAL-based N-Hash proposed by Miyaguchi, Ohta, and Iwata [886] are other hash functions which have attracted considerable attention. Snefru, one of the earliest proposals, is based on the idea of Algorithm 9.41, (typically) using as E the first 128 bits of output of a custom-designed symmetric 512-bit block cipher with fixed key $k = 0$. Differential cryptanalysis has been used by Biham and Shamir [137] to find collisions for Snefru with 2 passes, and is feasible for Snefru with 4 passes; Merkle currently recommends 8 passes (impacting performance). Cryptanalysis of the 128-bit hash N-Hash has been carried out by Biham and Shamir [136], with attacks on 3, 6, 9, and 12 rounds being of respective complexity 2^8, 2^{24}, 2^{40}, and 2^{56} for the more secure of the two proposed variations.

Despite many proposals, few hash functions based on modular arithmetic have withstood attack, and most that have (including those which are provably secure) tend to be relatively inefficient. MASH-1 (Algorithm 9.56), from Committee Draft ISO/IEC 10118-4 [608], evolved from a long line of related proposals successively broken and repaired, including contributions by Jueneman; Davies and Price; A. Jung; Girault [457] (which includes a summary); and members of ISO SC27/WG2 circa 1994-95 (e.g., in response to the cryptanalysis of the 1994 draft proposal, by Coppersmith and Preneel, in ISO/IEC JTC1/SC27 N1055, Attachment 12, "Comments on MASH-1 and MASH-2 (Feb.21 1995)"). Most prominent among prior proposals was the *sqmodn* algorithm (due to Jung) in informative Annex D of CCITT Recommendation X.509 (1988 version), which despite suffering ignominy at the hands of Coppersmith [275], was resurrected with modifications as the basis for MASH-1.

§9.5

Simmons [1146] notes that techniques for message authentication without secrecy (today called MACs) were known to Simmons, Stewart, and Stokes already in the early 1970s. In the open literature, the idea of using DES to provide a MAC was presented already in Feb. 1977 by Campbell [230], who wrote "... Each group of 64 message bits is passed through the algorithm after being combined with the output of the previous pass. The final DES output is thus a residue which is a cryptographic function of the entire message", and noted that to detect message replay or deletion each message could be made unique by using per-message keys or cryptographically protected sequence numbers. Page 121 of this same publication describes the use of encryption in conjunction with an appended redundancy check code for manipulation detection (cf. Figure 9.8(b)).

The term *MAC* itself evolved in the period 1979-1982 during development of ANSI X9.9 [36], where it is defined as "an eight-digit number in hexadecimal format which is the result of passing a financial message through the authentication algorithm using a specific key." FIPS 81 [398] standardizes MACs based on CBC and CFB modes (CFB-based MACs are little-used, having some disadvantages over CBC-MAC and apparently no advantages); see also FIPS 113 [400]. Algorithm 9.58 is generalized by ISO/IEC 9797 [597] to a CBC-based MAC for an n-bit block cipher providing an m-bit MAC, $m \leq n$, including an alternative to the optional strengthening process of Algorithm 9.58: a second key k' (possibly dependent on k) is used to encrypt the final output block. As discussed in Chapter 15, using ISO/IEC 9797 with DES to produce a 32-bit MAC and Algorithm 9.29 for padding is equivalent to the MAC specified in ISO 8731-1, ANSI X9.9 and required by ANSI X9.17. Regarding RIPE-MAC (Example 9.63) [178], other than the 2^{-64} probability of guessing a 64-bit MAC, and MAC forgery as applicable to all iterated MACs (see below), the best known attacks providing key recovery are linear cryptanalysis using 2^{42} known plaintexts for RIPE-MAC1, and a 2^{112} exhaustive search for RIPE-MAC3. Bellare, Kilian, and Rogaway [91] formally examine the security of CBC-based MACs and provide justification, establishing (via exact rather than asymptotic arguments) that pseudorandom functions are preserved under cipher block chaining; they also propose solutions to the problem of Example 9.62 (cf. Remark 9.59).

The MAA (Algorithm 9.68) was developed in response to a request by the Bankers Automated Clearing Services (U.K.), and first appeared as a U.K. National Physical Laboratory Report (NPL Report DITC 17/83 February 1983). It has been part of an ISO banking standard [577] since 1987, and is due to Davies and Clayden [306]; comments on its security (see also below) are offered by Preneel [1003], Davies [304], and Davies and Price [308], who note that its design follows the general principles of the Decimal Shift and Add (DSA) algorithm proposed by Sievi in 1980. As a consequence of the conjecture that MAA may show weaknesses in the case of very long messages, ISO 8731-2 specifies a special mode of operation for messages over 1024 bytes. For more recent results on MAA including exploration of a key recovery attack, see Preneel and van Oorschot [1010].

Methods for constructing a MAC algorithm from an MDC, including the secret prefix, suffix, and envelope methods, are discussed by Tsudik [1196]; Galvin, McCloghrie, and Davin [438] suggest addressing the message extension problem (Example 9.65) in the secret suffix method by using a prepended length field (this requires two passes over the message if the length is not known *a priori*). Preneel and van Oorschot [1009] compare the security of these methods; propose MD5-MAC (Algorithm 9.69) and similar constructions for customized MAC functions based on RIPEMD and SHA; and provide Fact 9.57, which applies to MAA ($n = 64 = 2m$) with $u = 2^{32.5}$ and $v = 2^{32.3}$, while for MD5-MAC ($n = 128 = 2m$) both u and v are on the order of 2^{64}. Remark 9.60 notwithstanding, the use of an n-bit internal chaining variable with a MAC-value of bitlength $m = n/2$ is supported by these results.

The envelope method with padding (Example 9.66) is discussed by Kaliski and Robshaw (*CryptoBytes* vol.1 no.1, Spring 1995). Preneel and van Oorschot [1010] proposed a key recovery attack on this method, which although clearly impractical by requiring over 2^{64} known text-MAC pairs (for MD5 with 128-bit key), reveals an architectural flaw. Bellare, Canetti, and Krawczyk [86] rigorously examined the security of a nested MAC construction (NMAC), and the practical variation HMAC thereof (Example 9.67), proving HMAC to be secure provided the hash function used exhibits certain appropriate characteristics. Prior to this, the related construction $h(k_1||h(k_2||x))$ was considered in the note of Kaliski and Robshaw (see above).

Other recent proposals for practical MACs include the bucket hashing construction of Rogaway [1065], and the XOR MAC scheme of Bellare, Guérin, and Rogaway [90]. The latter is a provably secure construction for MACs under the assumption of the availability of a finite pseudorandom function, which in practice is instantiated by a block cipher or hash function; advantages include that it is parallelizable and incremental.

MACs intended to provide unconditional security are often called *authentication codes* (cf. §9.1 above), with an *authentication tag* (cf. MAC value) accompanying data to provide origin authentication (including data integrity). More formally, an authentication code involves finite sets $\mathcal{S}$ of source states (plaintext), $\mathcal{A}$ of authentication tags, and $\mathcal{K}$ of secret keys, and a set of rules such that each $k \in \mathcal{K}$ defines a mapping $e_K : \mathcal{S} \rightarrow \mathcal{A}$. An (authenticated) message, consisting of a source state and a tag, can be verified only by the intended recipient (as for MACs) possessing a pre-shared key. Wegman and Carter [1234] first combined one-time pads with hash functions for message authentication; this approach was pursued by Brassard [191] trading unconditional security for short keys.

This approach was further refined by Krawczyk [714] (see also [717]), whose CRC-based scheme (Algorithm 9.72) is a minor modification of a construction by Rabin [1026]. A second LFSR-based scheme proposed by Krawczyk for producing m-bit hashes (again combined with one-time pads as per Algorithm 9.72) improves on a technique of Wegman and Carter, and involves matrix-vector multiplication by an $m \times b$ binary *Toeplitz matrix* A (each left-to-right diagonal is fixed: $A_{i,j} = A_{k,l}$ for $k - i = l - j$), itself generated from a random binary irreducible polynomial of degree m (defining the LFSR), and m bits of initial state. Krawczyk proves that the probability of successful MAC forgery here for a b-bit message is at most $b/2^{m-1}$, e.g., less than 2^{-30} even for $m = 64$ and a 1 Gbyte message (cf. Fact 9.73). Earlier, Bierbrauer et al. [127] explored the relations between coding theory, universal hashing, and practical authentication codes with relatively short keys (see also Johansson, Kabatianskii, and Smeets [638]; and the survey of van Tilborg [1211]). These and other MAC constructions suitable for use with stream ciphers are very fast, scalable, and information-theoretically secure when the short keys they require are used as one-time pads; when used with key streams generated by pseudorandom generators, their security is dependent on the stream and (at best) computationally secure.

Desmedt [335] investigated authenticity in stream ciphers, and proposed both unconditionally secure authentication systems and stream ciphers providing authenticity. Lai, Rueppel, and Woollven [731] define an efficient MAC for use with stream ciphers (but see Preneel [1003] regarding a modification to address tampering with ends of messages). Part of an initial secret key is used to seed a key stream generator, each bit of which selectively routes message bits to one of two feedback shift registers (FSRs), the initial states of which are part of the secret key and the final states of which comprise the MAC. The number of pseudorandom bits required equals the number of message bits. Taylor [1189] proposes an alternate MAC technique for use with stream ciphers.

§9.6

Simmons [1144] notes the use of sealed authenticators by the U.S. military. An early presentation of MACs and authentication is given by Meyer and Matyas [859]; the third or later printings are recommended, and include the one-pass PCBC encryption-integrity method of Example 9.91. Example 9.89 was initially proposed by the U.S. National Bureau of Standards, and was subsequently found by Jueneman to have deficiencies; this is included in the extensive discussion by Jueneman, Matyas, and Meyer [645] of using MDCs for integrity, along with the idea of Example 9.90, which Davies and Price [308, p.124] also consider for $n = 16$. Later work by Jueneman [644] considers both MDCs and MACs; see also Meyer and Schilling [860]. Davies and Price also provide an excellent discussion of transaction authentication, noting additional techniques (cf. §9.6.1) addressing message replay including use of MAC values themselves from immediately preceding messages as chaining values in place of random number chaining. Subtle flaws in various fielded data integrity techniques are discussed by Stubblebine and Gligor [1179].

§9.7

The taxonomy of preimages and collisions is from Preneel [1003]. The alternate terminology of Note 9.94 is from Lai and Massey [729], who published the first systematic treatment of attacks on iterated hash functions, including relationships between fixed-start and free-start attacks, considered *ideal security*, and re-examined MD-strengthening. The idea of Algorithm 9.92 was published by Yuval [1262], but the implications of the birthday paradox were known to others at the time, e.g., see Merkle [850, p.12-13]. The details of the memoryless version are from van Oorschot and Wiener [1207], who also show the process can be perfectly parallelized (i.e., attaining a factor r speedup with r processors) using parallel collision search methods; related independent work (unpublished) has been reported by Quisquater.

Meet-in-the-middle chaining attacks can be extended to handle additional constraints and otherwise generalized. A "triple birthday" chaining attack, applicable when the compression function is invertible, is given by Coppersmith [267] and generalized by Girault, Cohen, Campana [460]; see also Jueneman [644]. For additional discussion of differential cryptanalysis of hash functions based on block ciphers, see Biham and Shamir [138], Preneel, Govaerts, and Vandewalle [1005], and Rijmen and Preneel [1050].