# Chapter 5

# Hash Functions and Applications

In this chapter we introduce *cryptographic hash functions* and explore a few of their applications. At the most basic level, a hash function provides a way to map a long input string to a shorter output string sometimes called a *digest*. The primary requirement is to avoid *collisions*, or two inputs that map to the same digest. Collision-resistant hash functions have numerous uses. One example that we will see here is another approach—standardized as HMAC—for achieving domain extension for message authentication codes.

Beyond that, hash functions have become ubiquitous in cryptography, and they are often used in scenarios that require properties much stronger than collision resistance. It has become common to model cryptographic hash functions as being "completely unpredictable" (a.k.a., *random oracles*), and we discuss this framework—and the controversy that surrounds it—in detail later in the chapter. We touch on only a few applications of the random-oracle model here, but will encounter it again when we turn to the setting of public-key cryptography.

Hash functions are intriguing in that they can be viewed as lying between the worlds of private- and public-key cryptography. On the one hand, as we will see in Chapter 6, they are (in practice) constructed using symmetric-key techniques, and many of the canonical applications of hash functions are in the symmetric-key setting. From a theoretical point of view, however, the existence of collision-resistant hash functions appears to represent a qualitatively stronger assumption than the existence of pseudorandom functions (yet a weaker assumption than the existence of public-key encryption).

## 5.1 Definitions

Hash functions are simply functions that take inputs of some length and *compress* them into short, fixed-length outputs. The classic use of hash functions is in data structures, where they can be used to build hash tables that enable $\mathcal{O}(1)$ lookup time when storing a set of elements. Specifically, if the range of the hash function $H$ is of size $N$, then element $x$ is stored in row $H(x)$ of a table of size $N$. To retrieve $x$, it suffices to compute $H(x)$ and probe

that row of the table for the elements stored there. A "good" hash function for this purpose is one that yields few *collisions*, where a collision is a pair of distinct items $x$ and $x'$ for which $H(x) = H(x')$; in this case we also say that $x$ and $x'$ *collide*. (When a collision occurs, two elements end up being stored in the same cell, increasing the lookup time.)

*Collision-resistant* hash functions are similar in spirit. Again, the goal is to avoid collisions. However, there are fundamental differences. For one, the desire to minimize collisions in the setting of data structures becomes a *requirement* to avoid collisions in the setting of cryptography. Furthermore, in the context of data structures we can assume that the set of data elements is chosen independently of the hash function and without any intention to cause collisions. In the context of cryptography, in contrast, we are faced with an adversary who may select elements with the explicit goal of causing collisions. This means that collision-resistant hash functions are much harder to design.

### 5.1.1 Collision Resistance

Informally, a function $H$ is *collision resistant* if it is infeasible for any probabilistic polynomial-time algorithm to find a collision in $H$. We will only be interested in hash functions whose domain is larger than their range. In this case collisions must *exist*, but such collisions should be hard to find.

Formally, we consider *keyed* hash functions. That is, $H$ is a two-input function that takes as input a key $s$ and a string $x$, and outputs a string $H^s(x) \stackrel{\text{def}}{=} H(s, x)$. The requirement is that it must be hard to find a collision in $H^s$ for a randomly generated key $s$. There are at least two differences between keys in this context and keys as we have used them until now. First, not all strings necessarily correspond to valid keys (i.e., $H^s$ may not be defined for certain $s$), and therefore the key $s$ will typically be generated by an algorithm Gen rather than being chosen uniformly. Second, and perhaps more importantly, this key $s$ is (generally) not kept secret, and collision resistance is required even when the adversary is given $s$. In order to emphasize this, we superscript the key and write $H^s$ rather than $H_s$.

***DEFINITION 5.1*** *A* hash function *(with output length $\ell$) is a pair of probabilistic polynomial-time algorithms $(\mathsf{Gen}, H)$ satisfying the following:*

- Gen *is a probabilistic algorithm which takes as input a security parameter $1^n$ and outputs a key $s$. We assume that $1^n$ is implicit in $s$.*
- *$H$ takes as input a key $s$ and a string $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^{\ell(n)}$ (where $n$ is the value of the security parameter implicit in $s$).*

*If $H^s$ is defined only for inputs $x \in \{0,1\}^{\ell'(n)}$ and $\ell'(n) > \ell(n)$, then we say that $(\mathsf{Gen}, H)$ is a* fixed-length hash function for inputs of length $\ell'$. *In this case, we also call $H$ a* compression function.

In the fixed-length case we require that $\ell'$ be greater than $\ell$. This ensures that the function *compresses* its input. In the general case the function takes as input strings of arbitrary length. Thus, it also compresses (albeit only strings of length greater than $\ell(n)$). Note that without compression, collision resistance is trivial (since one can just take the identity function $H^s(x) = x$).

We now proceed to define security. As usual, we first define an experiment for a hash function $\Pi = (\mathsf{Gen}, H)$, an adversary $\mathcal{A}$, and a security parameter $n$:

**The collision-finding experiment** $\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(n)$**:**

1. *A key $s$ is generated by running* $\mathsf{Gen}(1^n)$.
2. *The adversary $\mathcal{A}$ is given $s$ and outputs $x, x'$. (If $\Pi$ is a fixed-length hash function for inputs of length $\ell'(n)$, then we require $x, x' \in \{0,1\}^{\ell'(n)}$.)*
3. *The output of the experiment is defined to be* 1 *if and only if $x \neq x'$ and $H^s(x) = H^s(x')$. In such a case we say that $\mathcal{A}$ has found a collision.*

The definition of collision resistance states that no efficient adversary can find a collision in the above experiment except with negligible probability.

***DEFINITION 5.2*** *A hash function $\Pi = (\mathsf{Gen}, H)$ is* collision resistant *if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there is a negligible function* negl *such that*

$$\Pr\left[\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(n) = 1\right] \leq \mathsf{negl}(n).$$

For simplicity, we sometimes refer to $H$ or $H^s$ as a "collision-resistant hash function," even though technically we should only say that $(\mathsf{Gen}, H)$ is. This should not cause any confusion.

Cryptographic hash functions are designed with the explicit goal of being collision resistant (among other things). We will discuss some common real-world hash functions in Chapter 6. In Section 8.4.2 we will see how it is possible to construct hash functions with proven collision resistance based on an assumption about the hardness of a certain number-theoretic problem.

**Unkeyed hash functions.** Cryptographic hash functions used in practice generally have a fixed output length (just as block ciphers have a fixed key length) and are usually *unkeyed*, meaning that the hash function is just a fixed function $H : \{0,1\}^* \to \{0,1\}^\ell$. This is problematic from a theoretical standpoint since for any such function there is always a constant-time algorithm that outputs a collision in $H$: the algorithm simply outputs a colliding pair $(x, x')$ hardcoded into the algorithm itself. Using keyed hash functions solves this technical issue since it is impossible to hardcode a colliding pair for every possible key using a reasonable amount of space (and in an asymptotic setting, it would be impossible to hardcode a colliding pair for every value of the security parameter).

Notwithstanding the above, the (unkeyed) cryptographic hash functions used in the real world are collision resistant for all practical purposes since colliding pairs are unknown (and computationally difficult to find) even though they must exist. Proofs of security for some construction based on collision resistance of a hash function are meaningful even when an unkeyed hash function $H$ is used, as long as the proof shows that any efficient adversary "breaking" the primitive can be used to efficiently find a collision in $H$. (All the proofs in this book satisfy this condition.) In this case, the interpretation of the proof of security is that if an adversary can break the scheme in practice, then it can be used to find a collision in practice, something that we believe is hard to do.

### 5.1.2 Weaker Notions of Security

In some applications it suffices to rely on security requirements weaker than collision resistance. These include:

- *Second-preimage or target-collision resistance:* Informally, a hash function is second preimage resistant if given $s$ and a uniform $x$ it is infeasible for a PPT adversary to find $x' \neq x$ such that $H^s(x') = H^s(x)$.
- *Preimage resistance:* Informally, a hash function is preimage resistant if given $s$ and a uniform $y$ it is infeasible for a PPT adversary to find a value $x$ such that $H^s(x) = y$. (Looking ahead to Chapter 7, this essentially means that $H^s$ is *one-way*.)

Any hash function that is collision resistant is also second preimage resistant. This holds since if, given a uniform $x$, an adversary can find $x' \neq x$ for which $H^s(x') = H^s(x)$, then it can clearly find a colliding pair $x$ and $x'$. Likewise, any hash function that is second preimage resistant is also preimage resistant. This is due to the fact that if it were possible, given $y$, to find an $x$ such that $H^s(x) = y$, then one could also take a given input $x'$, compute $y := H^s(x')$, and then obtain an $x$ with $H^s(x) = y$. With high probability $x' \neq x$ (relying on the fact that $H$ compresses, and so multiple inputs map to the same output), in which case a second preimage has been found.

We do not formally define the above notions or prove the above implications, since they are not used in the rest of the book. You are asked to formalize the above in Exercise 5.1.

## 5.2 Domain Extension: The Merkle-Damgård Transform

Hash functions are often constructed by first designing a collision-resistant compression function handling fixed-length inputs, and then using *domain*

*extension* to handle arbitrary-length inputs. In this section, we show one solution to the problem of domain extension. We return to the question of designing collision-resistant compression functions in Section 6.3.

The *Merkle–Damgård transform* is a common approach for extending a compression function to a full-fledged hash function, while maintaining the collision-resistance property of the former. It is used extensively in practice for hash functions including MD5 and the SHA family (see Section 6.3). The existence of this transform means that when designing collision-resistant hash functions, we can restrict our attention to the fixed-length case. This, in turn, makes the job of designing collision-resistant hash functions much easier. The Merkle–Damgård transform is also interesting from a theoretical point of view since it implies that compressing by a single bit is as easy (or as hard) as compressing by an arbitrary amount.

For concreteness, assume the compression function $(\mathsf{Gen}, h)$ compresses its input by half; say its input length is $2n$ and its output length is $n$. (The construction works regardless of the input/output lengths, as long as $h$ compresses.) We construct a collision-resistant hash function $(\mathsf{Gen}, H)$ that maps inputs of *arbitrary* length to outputs of length $n$. ($\mathsf{Gen}$ remains unchanged.) The Merkle–Damgård transform is defined in Construction 5.3 and depicted in Figure 5.1. The value $z_0$ used in step 2 of the construction, called the *initialization vector* or $IV$, is arbitrary and can be replaced by any constant.

***CONSTRUCTION 5.3***

Let $(\mathsf{Gen}, h)$ be a fixed-length hash function for inputs of length $2n$ and with output length $n$. Construct hash function $(\mathsf{Gen}, H)$ as follows:

- $\mathsf{Gen}$: remains unchanged.
- $H$: on input a key $s$ and a string $x \in \{0,1\}^*$ of length $L < 2^n$, do the following:
  1. Set $B := \lceil \frac{L}{n} \rceil$ (i.e., the number of blocks in $x$). Pad $x$ with zeros so its length is a multiple of $n$. Parse the padded result as the sequence of $n$-bit blocks $x_1, \ldots, x_B$. Set $x_{B+1} := L$, where $L$ is encoded as an $n$-bit string.
  2. Set $z_0 := 0^n$. (This is also called the $IV$.)
  3. For $i = 1, \ldots, B+1$, compute $z_i := h^s(z_{i-1} \| x_i)$.
  4. Output $z_{B+1}$.

The Merkle–Damgård transform.

***THEOREM 5.4*** *If* $(\mathsf{Gen}, h)$ *is collision resistant, then so is* $(\mathsf{Gen}, H)$.

**PROOF** We show that for any $s$, a collision in $H^s$ yields a collision in $h^s$. Let $x$ and $x'$ be two different strings of length $L$ and $L'$, respectively, such that $H^s(x) = H^s(x')$. Let $x_1, \ldots, x_B$ be the $B$ blocks of the padded $x$, and

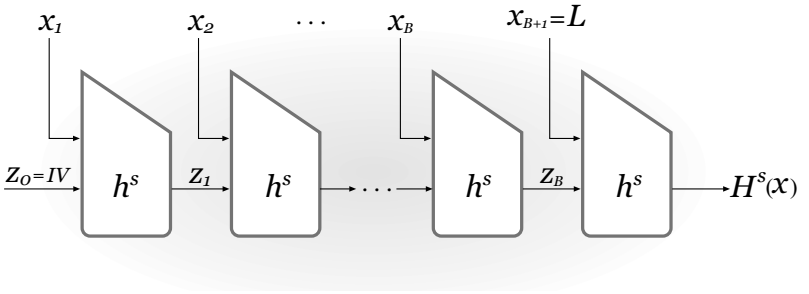


**FIGURE 5.1**: The Merkle–Damgård transform.

let $x'_1, \ldots, x'_{B'}$ be the $B'$ blocks of the padded $x'$. Recall that $x_{B+1} = L$ and $x'_{B'+1} = L'$. There are two cases to consider:

1. *Case 1:* $L \neq L'$. In this case, the last step of the computation of $H^s(x)$ is $z_{B+1} := h^s(z_B \| L)$, and the last step of the computation of $H^s(x')$ is $z'_{B'+1} := h^s(z'_{B'} \| L')$. Since $H^s(x) = H^s(x')$ it follows that $h^s(z_B \| L) = h^s(z'_{B'} \| L')$. However, $L \neq L'$ and so $z_B \| L$ and $z'_{B'} \| L'$ are two different strings that collide under $h^s$.

2. *Case 2:* $L = L'$. This means that $B = B'$. Let $z_0, \ldots, z_{B+1}$ be the values defined during the computation of $H^s(x)$, let $I_i \stackrel{\text{def}}{=} z_{i-1} \| x_i$ denote the $i$th input to $h^s$, and set $I_{B+2} \stackrel{\text{def}}{=} z_{B+1}$. Define $I'_1, \ldots, I'_{B+2}$ analogously with respect to $x'$. Let $N$ be the *largest* index for which $I_N \neq I'_N$. Since $|x| = |x'|$ but $x \neq x'$, there is an $i$ with $x_i \neq x'_i$ and so such an $N$ certainly exists. Because

$$I_{B+2} = z_{B+1} = H^s(x) = H^s(x') = z'_{B+1} = I'_{B+2},$$

we have $N \leq B+1$. By maximality of $N$, we have $I_{N+1} = I'_{N+1}$ and in particular $z_N = z'_N$. But this means that $I_N, I'_N$ are a collision in $h^s$.

We leave it as an exercise to turn the above into a formal reduction. ■

## 5.3 Message Authentication Using Hash Functions

In the previous chapter, we presented two constructions of message authentication codes for arbitrary-length messages. The first approach was generic, but inefficient. The second, CBC-MAC, was based on pseudorandom functions. Here we will see another approach, which we call "hash-and-MAC," that relies on collision-resistant hashing along with any message authentication code. We then discuss a standardized and widely used construction called HMAC that can be viewed as a specific instantiation of this approach.

### 5.3.1 Hash-and-MAC

The idea behind the hash-and-MAC approach is simple. First, an arbitrarily long message $m$ is hashed down to a fixed-length string $H^s(m)$ using a collision-resistant hash function. Then, a (fixed-length) MAC is applied to the result. See Construction 5.5 for a formal description.

***CONSTRUCTION 5.5***

Let $\Pi = (\mathsf{Mac}, \mathsf{Vrfy})$ be a MAC for messages of length $\ell(n)$, and let $\Pi_H = (\mathsf{Gen}_H, H)$ be a hash function with output length $\ell(n)$. Construct a MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ for arbitrary-length messages as follows:

- $\mathsf{Gen}'$: on input $1^n$, choose uniform $k \in \{0,1\}^n$ and run $\mathsf{Gen}_H(1^n)$ to obtain $s$; the key is $k' := \langle k, s\rangle$.
- $\mathsf{Mac}'$: on input a key $\langle k, s\rangle$ and a message $m \in \{0,1\}^*$, output $t \leftarrow \mathsf{Mac}_k(H^s(m))$.
- $\mathsf{Vrfy}'$: on input a key $\langle k, s\rangle$, a message $m \in \{0,1\}^*$, and a MAC tag $t$, output 1 if and only if $\mathsf{Vrfy}_k(H^s(m), t) \stackrel{?}{=} 1$.

The hash-and-MAC paradigm.

Construction 5.5 is secure if $\Pi$ is a secure MAC for fixed-length messages and $(\mathsf{Gen}, H)$ is collision resistant. Intuitively, since the hash function is collision resistant, authenticating $H^s(m)$ is as good as authenticating $m$ itself: if the sender can ensure that the receiver obtains the correct value $H^s(m)$, collision resistance guarantees that the attacker cannot find a different message $m'$ that hashes to the same value. A bit more formally, say a sender uses Construction 5.5 to authenticate some set of messages $\mathcal{Q}$, and an attacker $\mathcal{A}$ is then able to forge a valid tag on a new message $m^* \notin \mathcal{Q}$. There are two possible cases:

**Case 1:** *there is a message $m \in \mathcal{Q}$ such that $H^s(m^*) = H^s(m)$.* Then $\mathcal{A}$ has found a collision in $H^s$, contradicting the collision resistance of $(\mathsf{Gen}, H)$.

**Case 2:** *for every message $m \in \mathcal{Q}$ it holds that $H^s(m^*) \neq H^s(m)$.* Let $H^s(\mathcal{Q}) \stackrel{\text{def}}{=} \{H^s(m) \mid m \in \mathcal{Q}\}$. Then $H^s(m^*) \notin H^s(\mathcal{Q})$. In this case, $\mathcal{A}$ has forged a valid tag on the "new message" $H^s(m^*)$ with respect to the *fixed-length* message authentication code $\Pi$. This contradicts the assumption that $\Pi$ is a secure MAC.

We now turn the above into a formal proof.

***THEOREM 5.6*** *If $\Pi$ is a secure MAC for messages of length $\ell$ and $\Pi_H$ is collision resistant, then Construction 5.5 is a secure MAC (for arbitrary-length messages).*

**PROOF** Let $\Pi'$ denote Construction 5.5, and let $\mathcal{A}'$ be a PPT adversary attacking $\Pi'$. In an execution of experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$, let $k' = \langle k, s\rangle$ denote the MAC key, let $\mathcal{Q}$ denote the set of messages whose tags were requested by $\mathcal{A}'$, and let $(m^*, t)$ be the final output of $\mathcal{A}'$. We assume without loss of generality that $m^* \notin \mathcal{Q}$. Define $\mathsf{coll}$ to be the event that, in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$, there is an $m \in \mathcal{Q}$ for which $H^s(m^*) = H^s(m)$. We have

$$\begin{aligned}
&\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1] \\
&= \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \mathsf{coll}] + \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\mathsf{coll}}] \\
&\leq \Pr[\mathsf{coll}] + \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1 \wedge \overline{\mathsf{coll}}]. \qquad (5.1)
\end{aligned}$$

We show that both terms in Equation (5.1) are negligible, thus completing the proof. Intuitively, the first term is negligible by collision resistance of $\Pi_H$, and the second term is negligible by security of $\Pi$.

Consider the following algorithm $\mathcal{C}$ for finding a collision in $\Pi_H$:

**Algorithm** $\mathcal{C}$:
The algorithm is given $s$ as input (with $n$ implicit).

- Choose uniform $k \in \{0,1\}^n$.
- Run $\mathcal{A}'(1^n)$. When $\mathcal{A}'$ requests a tag on the $i$th message $m_i \in \{0,1\}^*$, compute $t_i \leftarrow \mathsf{Mac}_k(H^s(m_i))$ and give $t_i$ to $\mathcal{A}'$.
- When $\mathcal{A}'$ outputs $(m^*, t)$, then if there exists an $i$ for which $H^s(m^*) = H^s(m_i)$, output $(m^*, m_i)$.

It is clear that $\mathcal{C}$ runs in polynomial time. Let us analyze its behavior. When the input to $\mathcal{C}$ is generated by running $\mathsf{Gen}_H(1^n)$ to obtain $s$, the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{C}$ is distributed identically to the view of $\mathcal{A}'$ in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$. In particular, the tags given to $\mathcal{A}'$ by $\mathcal{C}$ have the same distribution as the tags that $\mathcal{A}'$ receives in $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$. Since $\mathcal{C}$ outputs a collision exactly when $\mathsf{coll}$ occurs, we have

$$\Pr[\mathsf{Hash\text{-}coll}_{\mathcal{C},\Pi_H}(n) = 1] = \Pr[\mathsf{coll}].$$

Because $\Pi_H$ is collision resistant, we conclude that $\Pr[\mathsf{coll}]$ is negligible.

We now proceed to prove that the second term in Equation (5.1) is negligible. Consider the following adversary $\mathcal{A}$ attacking $\Pi$ in $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$:

**Adversary** $\mathcal{A}$:
The adversary is given access to a MAC oracle $\mathsf{Mac}_k(\cdot)$.

- Compute $\mathsf{Gen}_H(1^n)$ to obtain $s$.
- Run $\mathcal{A}'(1^n)$. When $\mathcal{A}'$ requests a tag on the $i$th message $m_i \in \{0,1\}^*$, then: (1) compute $\hat{m}_i := H^s(m_i)$; (2) obtain a tag $t_i$ on $\hat{m}_i$ from the MAC oracle; and (3) give $t_i$ to $\mathcal{A}'$.
- When $\mathcal{A}'$ outputs $(m^*, t)$, then output $(H^s(m^*), t)$.

Clearly $\mathcal{A}$ runs in polynomial time. Consider experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n)$. In that experiment, the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{A}$ is distributed identically to its view in experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n)$. Furthermore, whenever both $\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) = 1$ and $\mathsf{coll}$ do not occur, $\mathcal{A}$ outputs a valid forgery. (In that case $t$ is a valid tag on $H^s(m^*)$ in scheme $\Pi$ with respect to $k$. The fact that $\mathsf{coll}$ did not occur means that $H^s(m^*)$ was never asked by $\mathcal{A}$ to its own MAC oracle and so this is indeed a forgery.) Therefore,

$$\Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] = \Pr[\mathsf{Mac\text{-}forge}_{\mathcal{A}',\Pi'}(n) \wedge \overline{\mathsf{coll}}],$$

and security of $\Pi$ implies that the former probability is negligible. This concludes the proof of the theorem. ∎

### 5.3.2 HMAC

All the constructions of message authentication codes we have seen so far are ultimately based on some block cipher. Is it possible to construct a secure MAC (for arbitrary-length messages) based directly on a hash function? A first thought might be to define $\mathsf{Mac}_k(m) = H(k\|m)$; we might expect that if $H$ is a "good" hash function then it should be difficult for an attacker to predict the value of $H(k\|m')$ given the value of $H(k\|m)$, for any $m' \neq m$, assuming $k$ is chosen at random (and unknown to the attacker). Unfortunately, if $H$ is constructed using the Merkle–Damgård transform—as most real-world hash functions are—then a MAC designed in this way is completely insecure, as you are asked to show in Exercise 5.10.

Instead, we can try using *two* layers of hashing. See Construction 5.7 for a standardized scheme called *HMAC* based on this idea.

***CONSTRUCTION 5.7***

Let $(\mathsf{Gen}_H, H)$ be a hash function constructed by applying the Merkle–Damgård transform to a compression function $(\mathsf{Gen}_H, h)$ taking inputs of length $n + n'$. (See text.) Let $\mathsf{opad}$ and $\mathsf{ipad}$ be fixed constants of length $n'$. Define a MAC as follows:

- $\mathsf{Gen}$: on input $1^n$, run $\mathsf{Gen}_H(1^n)$ to obtain a key $s$. Also choose uniform $k \in \{0,1\}^{n'}$. Output the key $\langle s, k\rangle$.
- $\mathsf{Mac}$: on input a key $\langle s, k\rangle$ and a message $m \in \{0,1\}^*$, output

$$t := H^s\Big((k \oplus \mathsf{opad}) \,\|\, H^s\big((k \oplus \mathsf{ipad}) \,\|\, m\big)\Big).$$

- $\mathsf{Vrfy}$: on input a key $\langle s, k\rangle$, a message $m \in \{0,1\}^*$, and a tag $t$, output 1 if and only if $t \stackrel{?}{=} H^s\big((k \oplus \mathsf{opad}) \,\|\, H^s\big((k \oplus \mathsf{ipad}) \,\|\, m\big)\big)$.

HMAC.

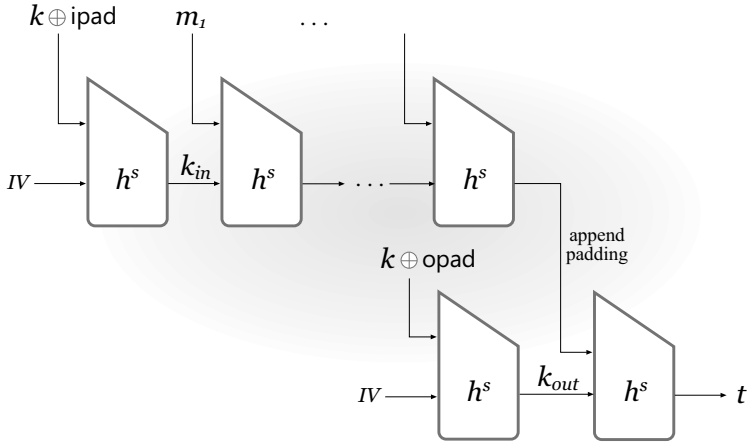


**FIGURE 5.2**: HMAC, pictorially.

Why should we have any confidence that HMAC is secure? One reason is that we can view HMAC as a specific instantiation of the hash-and-MAC paradigm from the previous section. To see this, we will look "under the hood" at what happens when a message is authenticated; see Figure 5.2. We must also specify parameters more carefully and go into a bit more detail regarding the way the Merkle–Damgård transform is implemented in practice.

Say $(\mathsf{Gen}_H, H)$ is constructed based on a compression function $(\mathsf{Gen}_H, h)$ in which $h$ maps inputs of length $n + n'$ to outputs of length $n$ (where, formally, $n'$ is a function of $n$). When we described the Merkle–Damgård transform in Section 5.2, we assumed $n' = n$, but that need not always be the case. We also said that the length of the message being hashed was encoded as an extra message block that is appended to the message. In practice, the length is instead encoded in a *portion* of a block using $\ell < n'$ bits. That is, computation of $H^s(x)$ begins by padding $x$ with zeros to a string of length exactly $\ell$ less than a multiple of $n'$; it then appends the length $L = |x|$, encoded using exactly $\ell$ bits. The hash of the resulting sequence of $n'$-bit blocks $x_1, \ldots$ is then computed as in Construction 5.3. We will assume that $n + \ell \leq n'$. This means, in particular, that if we hash an input $x$ of length $n' + n$ then the padded result (including the length) will be exactly $2n'$ bits long. The proof of Theorem 5.4, showing that $(\mathsf{Gen}_H, H)$ is collision resistant if $(\mathsf{Gen}_H, h)$ is collision resistant, remains unchanged.

Coming back to HMAC, and looking at Figure 5.2, we can see that the general form of HMAC involves hashing an arbitrary-length message down to a short string $y \stackrel{\text{def}}{=} H^s(\,(k \oplus \mathsf{ipad}) \,\|\, m)$, and then computing the (secretly keyed) function $H^s((k \oplus \mathsf{opad}) \,\|\, y)$ of the result. But we can say more than this. Note first that the "inner" computation

$$\widetilde{H}^s(m) \stackrel{\text{def}}{=} H^s(\,(k \oplus \mathsf{ipad}) \,\|\, m)$$

is collision resistant (assuming $h$ is), for any value of $k \oplus \mathsf{ipad}$. Moreover, the first step in the "outer" computation $H^s((k \oplus \mathsf{opad}) \| y)$ is to compute a value $k_{out} \stackrel{\text{def}}{=} h^s(IV \| (k \oplus \mathsf{opad}))$. Then, we evaluate $h^s(k_{out} \| \hat{y})$ where $\hat{y}$ refers to the padded value of $y$ (i.e., including the length of $(k \oplus \mathsf{opad}) \| y$, which is always $n' + n$ bits, encoded using exactly $\ell$ bits). Thus, if we treat $k_{out}$ as uniform—we will be more formal about this below—and assume that

$$\widetilde{\mathsf{Mac}}_k(y) \stackrel{\text{def}}{=} h^s(k \| \hat{y}) \tag{5.2}$$

is a secure *fixed-length* MAC, then HMAC can be viewed as an instantiation of the hash-and-MAC approach with

$$\text{HMAC}_{s,k}(m) = \widetilde{\mathsf{Mac}}_{k_{out}}(\tilde{H}^s(m)) \tag{5.3}$$

(where $k_{out} = h^s(IV \| (k \oplus \mathsf{opad}))$). Because of the way the compression function $h$ is typically designed (see Section 6.3.1), the assumption that $\widetilde{\mathsf{Mac}}$ is a secure fixed-length MAC is a reasonable one.

**The roles of ipad and opad.** Given the above, one might wonder why it is necessary to incorporate $k$ in the "inner" computation $H^s((k \oplus \mathsf{ipad}) \| m)$. (In particular, for the hash-and-MAC approach to be secure we require collision resistance in the first step, which does not require any secret key.) The reason is that this allows security of HMAC to be based on the potentially weaker assumption that $(\mathsf{Gen}_H, H)$ is *weakly* collision resistant, where weak collision resistance is defined by the following experiment: a key $s$ is generated using $\mathsf{Gen}_H$ and a uniform *secret* $k_{in} \in \{0,1\}^n$ is chosen. Then the adversary is allowed to interact with a "hash oracle" that returns $H^s_{k_{in}}(m)$ in response to the query $m$, where $H^s_{k_{in}}$ refers to computation of $H^s$ using the Merkle–Damgård transform applied to $h^s$, but using the secret value $k_{in}$ as the $IV$. (Refer again to Figure 5.2.) The adversary succeeds if it can output distinct values $m, m'$ such that $H^s_{k_{in}}(m) = H^s_{k_{in}}(m')$, and we say that $(\mathsf{Gen}_H, H)$ is *weakly collision resistant* if every PPT $\mathcal{A}$ succeeds in this experiment with only negligible probability. If $(\mathsf{Gen}_H, H)$ is collision resistant then it is clearly weakly collision resistant; the latter, however, is a weaker condition that is potentially easier to satisfy. This is a good example of sound security engineering. This defensive design strategy paid off when it was discovered that the hash function MD5 (see Section 6.3.2) was *not* collision resistant. The collision-finding attacks on MD5 did not violate weak collision resistance, and HMAC-MD5 was not broken even though MD5 was. This gave developers time to replace MD5 in HMAC implementations, without immediate fear of attack. (Despite this, HMAC-MD5 should no longer be used now that weaknesses in MD5 are known.)

The above discussion suggests that *independent* keys should be used in the outer and inner computations. For reasons of efficiency, a single key $k$ is used for HMAC, but the key is used in combination with ipad and opad to derive

two other keys. Define

$$G^s(k) \stackrel{\text{def}}{=} h^s\left(IV \| (k \oplus \mathsf{opad})\right) \| h^s\left(IV \| (k \oplus \mathsf{ipad})\right) = k_{out} \| k_{in}. \qquad (5.4)$$

If we assume that $G^s$ is a pseudorandom generator for any $s$, then $k_{out}$ and $k_{in}$ can be treated as independent and uniform keys when $k$ is uniform. Security of HMAC then reduces to the security of the following construction:

$$\mathsf{Mac}_{s,k_{in},k_{out}}(m) = h^s\left(k_{out} \| H^s_{k_{in}}(m)\right).$$

(Compare to Equation (5.3).) As noted earlier, this construction can be proven secure (using a variant of the proof for the hash-and-MAC approach) if $H$ is weakly collision resistant and the MAC defined in Equation (5.2) is a secure fixed-length MAC.

***THEOREM 5.8*** *Assume $G^s$ as defined in Equation (5.4) is a pseudorandom generator for any $s$, the MAC defined in Equation (5.2) is a secure fixed-length MAC for messages of length $n$, and $(\mathsf{Gen}_H, H)$ is weakly collision resistant. Then HMAC is a secure MAC (for arbitrary-length messages).*

**HMAC in practice.** HMAC is an industry standard and is widely used in practice. It is highly efficient and easy to implement, and is supported by a proof of security based on assumptions that are believed to hold for practical hash functions. The importance of HMAC is partially due to the timeliness of its appearance. Before the introduction of HMAC, many practitioners refused to use CBC-MAC (with the claim that it was "too slow") and instead used heuristic constructions that were insecure. HMAC provided a standardized, secure way of doing message authentication based on hash functions.

## 5.4 Generic Attacks on Hash Functions

What is the best security we can hope for a hash function $H$ to provide? We explore this question by showing two attacks that are *generic* in the sense that they apply to *arbitrary* hash functions. The existence of these attacks implies lower bounds on the output length of $H$ needed to achieve some desired level of security, and therefore has important practical ramifications.

### 5.4.1 Birthday Attacks for Finding Collisions

Let $H : \{0,1\}^* \to \{0,1\}^\ell$ be a hash function. (Here and in the rest of the chapter, we drop explicit mention of the hash key $s$ since it is not directly relevant. One can also view $s$ as being generated and fixed before these

algorithms are applied.) There is a trivial collision-finding attack running in time $\mathcal{O}(2^\ell)$: simply evaluate $H$ on $2^\ell + 1$ distinct inputs; by the pigeonhole principle, two of the outputs must be equal. Can we do better?

Generalizing the above algorithm, say we choose $q$ distinct inputs $x_1, \ldots, x_q$, compute $y_i := H(x_i)$, and check whether any of the two $y_i$ values are equal. What is the probability that this algorithm finds a collision? As we have just said, if $q > 2^\ell$ then a collision occurs with probability 1. What is the probability of a collision when $q$ is smaller? It is somewhat difficult to analyze this probability exactly, and so we will instead analyze an idealized case in which $H$ is treated as a random function.[1] That is, for each $i$ we assume that the value $y_i = H(x_i)$ is uniformly distributed in $\{0,1\}^\ell$ and independent of any of the previous output values $\{y_j\}_{j<i}$ (recall we assume all $\{x_i\}$ are distinct). We have thus reduced our problem to the following one: if we choose values $y_1, \ldots, y_q \in \{0,1\}^\ell$ uniformly at random, what is the probability that there exist distinct $i, j$ with $y_i = y_j$?

This problem has been extensively studied, and is related to the so-called *birthday problem* discussed in detail in Appendix A.4. For this reason, the collision-finding algorithm we have described is often called a *birthday attack*. The birthday problem is the following: if $q$ people are in a room, what is the probability that two of them have the same birthday? (Assume birthdays are uniformly and independently distributed among the 365 days of a non-leap year.) This is exactly analogous to our problem: if $y_i$ represents the birthday of person $i$, then we have $y_1, \ldots, y_q \in \{1, \ldots, 365\}$ chosen uniformly, and matching birthdays correspond to distinct $i, j$ with $y_i = y_j$ (i.e., matching birthdays correspond to collisions).

In Appendix A.4 we show that for $y_1, \ldots, y_q$ chosen uniformly in $\{1, \ldots, N\}$, the probability of a collision is roughly $1/2$ when $q = \Theta(N^{1/2})$. In the case of birthdays, once there are only 23 people the probability that some two of them have the same birthday is greater than $1/2$. In our setting, this means that when the hash function has output length $\ell$ (and so the range is of size $2^\ell$), then taking $q = \Theta(2^{\ell/2})$ yields a collision with probability roughly $1/2$.

From a concrete-security perspective, the above means that for a hash function to resist collision-finding attacks that run in time $T$ (where we take the time to evaluate $H$ as our unit of time), the output length of the hash function needs to be at least $2 \log T$ bits (since $2^{(2 \log T)/2} = T$). Taking specific parameters, this means that if we want finding collisions to be as difficult as an exhaustive search over 128-bit keys, then we need the output length of the hash function to be at least 256 bits. We stress that having an output this long is only a *necessary* condition, not a sufficient one. We also note that birthday attacks work only for finding collisions. There are no generic attacks

[1] It can be shown that this is (essentially) the worst case, and collisions occur with higher probability if $H$ deviates from random and the $\{x_i\}$ are chosen uniformly.

for second preimage resistance or preimage resistance of a hash functions $H$ that require fewer than $2^\ell$ evaluations of $H$ (though see Section 5.4.3).

**Finding meaningful collisions.** The birthday attack just described gives a collision that is not necessarily very useful. But the same idea can be used to find "meaningful" collisions as well. Assume Alice wishes to find two messages $x$ and $x'$ such that $H(x) = H(x')$, and furthermore $x$ should be a letter from her employer explaining why she was fired from work, while $x'$ should be a flattering letter of recommendation. (This might allow Alice to forge an appropriate tag on a letter of recommendation if the hash-and-MAC approach is being used by her employer to authenticate messages.) The observation is that the birthday attack only requires the hash inputs $x_1, \ldots, x_q$ to be distinct; they do not need to be random. Alice can carry out a birthday-type attack by generating $q = \Theta(2^{\ell/2})$ messages of the first type and $q$ messages of the second type, and then looking for collisions between messages of the two types. A small change to the analysis from Appendix A.4 shows that this gives a collision between messages of different types with probability roughly 1/2. A little thought shows that it is easy to write the same message in many different ways. For example, consider the following:

> It is *hard/difficult/challenging/impossible* to *imagine/believe* that we will *find/locate/hire* another *employee/person* having similar *abilities/skills/character* as Alice. She has done a *great/super* job.

Any combination of the italicized words is possible, and expresses the same point. Thus, the sentence can be written in $4 \cdot 2 \cdot 3 \cdot 2 \cdot 3 \cdot 2 = 288$ different ways. This is just one sentence and so it is actually easy to generate a message that can be rewritten in $2^{64}$ different ways—all that is needed are 64 words with one synonym each. Alice can prepare $2^{\ell/2}$ letters explaining why she was fired and another $2^{\ell/2}$ letters of recommendation; with good probability, a collision between the two types of letters will be found.

### 5.4.2 Small-Space Birthday Attacks

The birthday attacks described above require a large amount of memory; specifically, they require the attacker to store all $\mathcal{O}(q) = \mathcal{O}(2^{\ell/2})$ values $\{y_i\}$, because the attacker does not know in advance which pair of values will yield a collision. This is a significant drawback because memory is, in general, a scarcer resource than time. It is arguably more difficult to allocate and manage storage for $2^{60}$ bytes than to execute $2^{60}$ CPU instructions. Furthermore, one can always let a computation run indefinitely, whereas the memory requirements of an algorithm must be satisfied as soon as that amount of memory is needed.

We show here a better birthday attack with drastically reduced memory requirements. In fact, it has similar time complexity and success probability as before, but uses only *constant* memory. The attack begins by choosing a

random value $x_0$ and then computing $x_i := H(x_{i-1})$ and $x_{2i} := H(H(x_{2(i-1)}))$ for $i = 1, 2, \ldots$. (Note that $x_i = H^{(i)}(x_0)$ for all $i$, where $H^{(i)}$ refers to $i$-fold iteration of $H$.) In each step the values $x_i$ and $x_{2i}$ are compared; if they are equal then there is a collision somewhere in the sequence $x_0, x_1, \ldots, x_{2i-1}$. The algorithm then finds the least value of $j$ for which $x_j = x_{j+i}$ (note that $j \leq i$ since $j = i$ works), and outputs $x_{j-1}, x_{j+i-1}$ as a collision. This attack, described formally as Algorithm 5.9 and analyzed below, only requires storage of two hash values in each iteration.

***ALGORITHM 5.9***
**A small-space birthday attack**

**Input:** A hash function $H : \{0,1\}^* \to \{0,1\}^\ell$
**Output:** Distinct $x, x'$ with $H(x) = H(x')$

$x_0 \leftarrow \{0,1\}^{\ell+1}$
$x' := x := x_0$
**for** $i = 1, 2, \ldots$ **do**:
  $x := H(x)$
  $x' := H(H(x'))$
  // now $x = H^{(i)}(x_0)$ and $x' = H^{(2i)}(x_0)$
  **if** $x = x'$ **break**
$x' := x$, $x := x_0$
**for** $j = 1$ to $i$:
  **if** $H(x) = H(x')$ **return** $x, x'$ and **halt**
  **else** $x := H(x)$, $x' := H(x')$
  // now $x = H^{(j)}(x_0)$ and $x' = H^{(i+j)}(x_0)$

How many iterations of the first loop do we expect before $x' = x$? Consider the sequence of values $x_1, x_2, \ldots$, where $x_i = H^{(i)}(x_0)$ as defined before. If we model $H$ as a random function, then each of these values is uniformly and independently distributed in $\{0,1\}^\ell$ until the first repeat occurs. Thus, we expect a repeat to occur with probability $1/2$ in the first $q = \Theta(2^{\ell/2})$ terms of the sequence. We show that when there is a repeat in the first $q$ elements, the algorithm finds a repeat in at most $q$ iterations of the first loop:

***CLAIM 5.10*** *Let $x_1, \ldots, x_q$ be a sequence of values with $x_m = H(x_{m-1})$. If $x_I = x_J$ with $1 \leq I < J \leq q$, then there is an $i < J$ such that $x_i = x_{2i}$.*

**PROOF** The sequence $x_I, x_{I+1}, \ldots$ repeats with period $\Delta \stackrel{\text{def}}{=} J - I$. That is, for all $i \geq I$ and $k \geq 0$ it holds that $x_i = x_{i+k\cdot\Delta}$. Let $i$ be the smallest multiple of $\Delta$ that is also greater than or equal to $I$. We have $i < J$ since the sequence of $\Delta$ values $I, I+1, \ldots I+(\Delta-1) = J-1$ contains a multiple of $\Delta$. Since $i \geq I$ and $2i - i = i$ is a multiple of $\Delta$, it follows that $x_i = x_{2i}$. ■

Thus, if there is a repeated value in the sequence $x_1, \ldots, x_q$, then there is some $i < q$ for which $x_i = x_{2i}$. But then in iteration $i$ of our algorithm, we have $x = x'$ and the algorithm breaks out of the first loop. At that point in the algorithm, we know that $x_i = x_{i+i}$. The algorithm then sets $x' := x(= x_i)$ and $x := x_0$, and proceeds to find the *smallest* $j \geq 0$ for which $x_j = x_{j+i}$. (Note $j \neq 0$ because $|x_0| = \ell + 1$.) It outputs $x_{j-1}, x_{j+i-1}$ as a collision.

**Finding meaningful collisions.** The algorithm just described may not seem amenable to finding meaningful collisions since it has no control over the elements sampled. Nevertheless, we show how finding meaningful collisions is possible. The trick is to find a collision in the right function!

Assume, as before, that Alice wishes to find a collision between messages of two different "types," e.g., a letter explaining why Alice was fired and a flattering letter of recommendation that both hash to the same value. Then, Alice writes each message so that there are $\ell - 1$ interchangeable words in each; i.e., there are $2^{\ell-1}$ messages of each type. Define the one-to-one function $g : \{0,1\}^\ell \to \{0,1\}^*$ such that the $\ell$th bit of the input selects between messages of type 0 or type 1, and the $i$th bit (for $1 \leq i \leq \ell - 1$) selects between options for the $i$th interchangeable word in messages of the appropriate type. For example, consider the sentences:

0: Bob is a *good/hardworking* and *honest/trustworthy worker/employee.*
1: Bob is a *difficult/problematic* and *taxing/irritating worker/employee.*

Define a function $g$ that takes 4-bit inputs, where the last bit determines the type of sentence output, and the initial three bits determine the choice of words in that sentence. For example:

$$g(0000) = \text{Bob is a good and honest worker.}$$
$$g(0001) = \text{Bob is a difficult and taxing worker.}$$
$$g(1010) = \text{Bob is a hardworking and honest employee.}$$
$$g(1011) = \text{Bob is a problematic and taxing employee.}$$

Now define $f : \{0,1\}^\ell \to \{0,1\}^\ell$ by $f(x) \stackrel{\text{def}}{=} H(g(x))$. Alice can find a collision in $f$ using the small-space birthday attack shown earlier. The point here is that any collision $x, x'$ in $f$ yields two messages $g(x), g(x')$ that collide under $H$. If $x, x'$ is a random collision then we expect that with probability $1/2$ the colliding messages $g(x), g(x')$ will be of different types (since $x$ and $x'$ differ in their final bit with that probability). If the colliding messages are not of different types, the process can be repeated again from scratch.

### 5.4.3 *Time/Space Tradeoffs for Inverting Functions

In this section we consider the question of preimage resistance, i.e., we are interested in algorithms for the problem of function inversion. Here, an algorithm is given $y = H(x)$ for uniform $x$, and the goal is to find any $x'$ such

that $H(x') = y$. We begin by assuming that the input and output lengths of $H$ are equal, and briefly consider the more general case at the end.

Let $H : \{0,1\}^\ell \to \{0,1\}^\ell$ be a function. Without exploiting any weaknesses of $H$, finding a preimage of a point $y$ can be done in time $\mathcal{O}(2^\ell)$ via an exhaustive search over the domain. We show that with significant preprocessing, and a relatively large amount of memory, it is possible to do better.

To be clear: we view preprocessing as a one-time operation and we will not be overly concerned with its cost. We are instead interested in the *on-line* time required to invert $H$ at a point $y$, after the preprocessing has been done. This is justified if the cost of preprocessing can be amortized over the inversion of many points, or if we are willing to invest computational resources for preprocessing before $y$ is known for the benefit of faster inversion afterwards.

In fact, it is trivial to use preprocessing to enable function inversion in very little time. All we need to do is evaluate $H$ on every point during the preprocessing phase, and then store the pairs $\{(x, H(x))\}$ in a table, sorted by their second entry. Upon receiving any point $y$, a preimage of $y$ can be found easily by searching the table for a pair with second entry $y$. The drawback here is that we need to allocate space for storing $\mathcal{O}(2^\ell)$ pairs in the table, which can be prohibitive, if not impossible for large $\ell$ (e.g., $\ell = 80$).

The initial brute-force attack uses constant memory and $\mathcal{O}(2^\ell)$ time, while the attack just described stores $\mathcal{O}(2^\ell)$ points and enables inversion in essentially constant time. We now present an approach that allows an attacker to *trade off* time and memory. Specifically, we show how to store $\mathcal{O}(2^{2\ell/3})$ points and find preimages in time $\mathcal{O}(2^{2\ell/3})$; other tradeoffs are possible.

**A warmup.** We begin by considering a simple case where the function $H$ defines a cycle, meaning that $x, H(x), H(H(x)), \ldots$ covers all of $\{0,1\}^\ell$ for any starting point $x$ (note that most functions do not define a cycle, but we assume this in order to demonstrate the idea in a very simple case). For clarity, let $N = 2^\ell$ denote the domain size.

In the preprocessing phase, the attacker simply exhausts the entire cycle, beginning at an arbitrary starting point $x_0$ and computing $x_1 := H(x_0)$, $x_2 := H(H(x_0))$, up to $x_N = H^{(N)}(x_0)$, where $H^{(i)}$ refers to $i$-fold evaluation of $H$. Let $x_i \stackrel{\text{def}}{=} H^{(i)}(x_0)$. We imagine partitioning the cycle into $\sqrt{N}$ segments of length $\sqrt{N}$ each, and having the attacker store the points at the beginning and end of each such segment. That is, the attacker stores in a table pairs of the form $(x_{i\cdot\sqrt{N}},\ x_{(i+1)\cdot\sqrt{N}})$, for $i = 0$ to $\sqrt{N} - 1$, sorted by the second component of each pair. The resulting table contains $\mathcal{O}(\sqrt{N})$ points.

When the attacker is given a point $y$ to invert in the on-line phase, it checks which of $y$, $H(y)$, $H^{(2)}(y)$, ... corresponds to the endpoint of a segment. (Each check just involves a table lookup on the second component of the stored pairs.) Since $y$ lies in some segment, it is guaranteed to hit an endpoint within $\sqrt{N}$ steps. Once an endpoint $x = x_{(i+1)\cdot\sqrt{N}}$ is identified, the attacker takes the starting point $x' = x_{i\cdot\sqrt{N}}$ of the corresponding segment

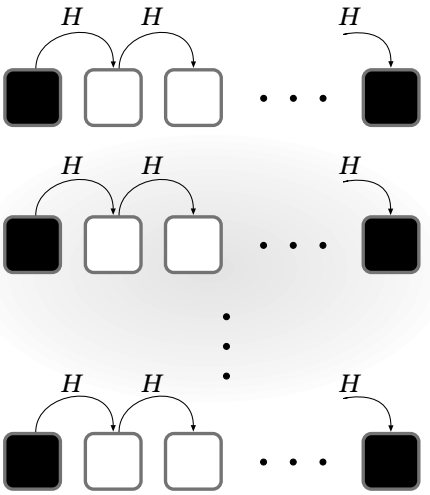


**FIGURE 5.3**: Table generation. Only the $(SP_i, EP_i)$ pairs are stored.

and computes $H(x')$, $H^{(2)}(x')$, ... until $y$ is reached; this immediately gives the desired preimage. Observe that this takes at most $\sqrt{N}$ evaluations of $H$.

In summary, this attack stores $\mathcal{O}(\sqrt{N})$ points and finds preimages with probability 1 using $\mathcal{O}(\sqrt{N})$ hash computations.

**Hellman's time/space tradeoff.** Martin Hellman introduced a more general time/space tradeoff applicable to an arbitrary function $H$ (though the analysis treats $H$ as a random function). Hellman's attack still stores the starting point and endpoint of several segments, but in this case the segments are "independent" rather than being part of one large cycle. In more detail: let $s, t$ be parameters we will set later. The attack first chooses $s$ uniform starting points $SP_1, \ldots, SP_s \in \{0,1\}^\ell$. For each such point $SP_i$, it computes a corresponding endpoint $EP_i := H^{(t)}(SP_i)$ using $t$-fold application of $H$. (See Figure 5.3.) The attacker then stores the values $\{(SP_i, EP_i)\}_{i=1}^s$ in a table, sorted by the second entry of each pair.

Upon receiving a value $y$ to invert, the attack proceeds as in the simple case discussed earlier. Specifically, it checks if any of $y$, $H(y)$, ..., $H^{(t-1)}(y)$ is equal to the endpoint of some segment (stopping as soon as the first such match is found). It is possible that none of these values is equal to an endpoint (as we discuss below). However, if $H^{(j)}(y) = EP_i = H^{(t)}(SP_i)$ for some $i, j$, then the attacker computes $H^{(t-j-1)}(SP_i)$ and checks whether this is a preimage of $y$. The entire process requires at most $t$ evaluations of $H$.

This seems to work, but there are several subtleties we have ignored. First, it may happen that none of $y$, $H(y)$, ..., $H^{(t-1)}(y)$ is the endpoint of a segment. This can happen if $y$ is not in the collection of $s \cdot t$ values (not counting the starting points) obtained during the initial process of generating the table. We can set $s \cdot t \geq N$ in an attempt to include every $\ell$-bit string in the

table, but this does not solve the problem since there can be collisions in the table itself—in fact, for $s \cdot t \geq N^{1/2}$ our previous analysis of the birthday problem tells us that collisions are likely—which will reduce the number of distinct points in the collection of values. A second problem, which arises even if $y$ *is* in the table, is that even if we find a matching endpoint, and so $H^{(j)}(y) = EP_i = H^{(t)}(SP_i)$ for some $i, j$, this does not guarantee that $H^{(t-j-1)}(SP_i)$ is a preimage of $y$. The issue here is that the segment $y, H(y), \ldots, H^{(t-1)}(y)$ might collide with the $i$th segment even though $y$ itself is not in that segment; see Figure 5.4. (Even if $y$ lies in some segment, the first matching endpoint may not be in that segment.) We call this a *false positive*. One might think this is unlikely to occur if $H$ is collision resistant; again, however, we are dealing with a situation where more than $\sqrt{N}$ points are involved and so collisions actually become likely.

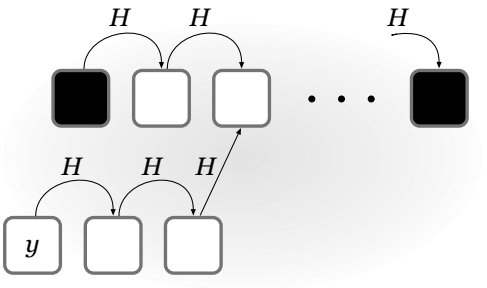


**FIGURE 5.4**: Colliding in the on-line phase.

The problem of false positives can be addressed by modifying the algorithm so that it always computes the entire sequence $y, H(y), \ldots, H^{(t-1)}(y)$, and checks whether $H^{(t-j-1)}(SP_i)$ is a preimage of $y$ for *every* $i, j$ such that $H^{(j)}(y) = EP_i$. This is guaranteed to find a preimage as long as $y$ is in the collection of values (not including the starting points) generated during preprocessing. A concern now is that the running time of the algorithm might increase, since each false positive incurs an additional $\mathcal{O}(t)$ hash evaluations. One can show that the expected number of false positives is $\mathcal{O}(st^2/N)$. (There are $t$ values in the sequence $y, H(y), \ldots, H^{(t-1)}(y)$ and at most $st$ distinct points in the table. Treating $H$ as a random function, the probability that any point in the sequence equals some point in the table is $1/N$. The expected number of false positives is thus $t \cdot st \cdot 1/N = st^2/N$.) Thus, as long as $st^2 \approx N$, which we will ensure for other reasons below, the expected number of false positives is constant and dealing with false positives is expected to require only $\mathcal{O}(t)$ additional hash computations.

Given the above modification, the probability of inverting $y = H(x)$ is at least the probability that $x$ is in the collection of points (not including the endpoints) generated during preprocessing. We now lower bound this probability, taken over the randomness of the preprocessing stage as well as

uniform choice of $x$, treating $H$ as a random function in the analysis. We first compute the expected number of distinct points in the table. Consider what happens when the $i$th row of the table is generated. The starting point $SP_i$ is uniform and there are at most $(i-1)\cdot t$ distinct points (not including the endpoints) in the table already, so the probability that $SP_i$ is "new" (i.e., not equal to any previous value) is at least $1-(i-1)\cdot t/N$. What is the probability that $H(SP_i)$ is new? If $SP_i$ is not new, then almost surely neither is $H(SP_i)$. On the other hand, if $SP_i$ *is* new then $H(SP_i)$ is uniform (because we treat $H$ as a random function) and so is new with probability at least $1-((i-1)\cdot t+1)/N$. (We now have the additional point $SP_i$.) Thus, the probability that $H(SP_i)$ is new is at least

$$\begin{aligned}\Pr[SP_i \text{ is new}] &\cdot \Pr[H(SP_i) \text{ is new} \mid SP_i \text{ is new}] \\ &\geq \left(1-\frac{(i-1)\cdot t}{N}\right)\cdot\left(1-\frac{(i-1)\cdot t+1}{N}\right) \\ &> \left(1-\frac{(i-1)\cdot t+1}{N}\right)^2.\end{aligned}$$

Continuing in this way, the probability that $H^{(t-1)}(SP_i)$ is new is at least

$$\left(1-\frac{i\cdot t}{N}\right)^t = \left[\left(1-\frac{i\cdot t}{N}\right)^{\frac{N}{i\cdot t}}\right]^{\frac{i\cdot t^2}{N}} \approx e^{-it^2/N}.$$

The thing to notice here is that when $it^2 \leq N/2$, this probability is at least $1/2$; on the other hand, once $it^2 > N$ the probability is rather small. Considering the last row, when $i=s$, this means that we will not gain much additional coverage if $st^2 > N$. A good setting of the parameters is thus $st^2 = N/2$. Assuming this, the expected number of distinct points in the table is

$$\sum_{i=1}^{s}\sum_{j=0}^{t-1}\Pr\left[H^{(j)}(SP_i) \text{ is new}\right] \geq \sum_{i=1}^{s}\sum_{j=0}^{t-1}\frac{1}{2} = \frac{st}{2}.$$

The probability that $x$ is "covered" is then at least $\frac{st}{2N} = \frac{1}{4t}$.

This gives a weak time/space tradeoff, in which we can use more space (and consequently less time) at the expense of decreasing the probability of inverting $y$. But we can do better by generating $T = 4t$ "independent" tables. (This increases both the space and time by a factor of $T$.) As long as we can treat the probabilities of $x$ being in each of the associated tables as independent, the probability that *at least one* of these tables contains $x$ is

$$1-\Pr[\text{no table contains } x] = 1-\left(1-\frac{1}{4t}\right)^{4t} \approx 1-e^{-1} = 0.63.$$

The only remaining question is how to generate an independent table. (Note that generating a table exactly as before is the same as adding $s$ additional

rows to our original table, which we have already seen does not help.) We can do this for the $i$th such table by applying some function $F_i$ after every evaluation of $H$, where $F_1, \ldots, F_T$ are all distinct. (A good choice might be to set $F_i(x) = x \oplus c_i$ for some fixed constant $c_i$ that is different in each table.) Let $H_i \stackrel{\text{def}}{=} F_i \circ H$, i.e., $H_i(x) = F_i(H(x))$. Then for the $i$th table we again choose $s$ random starting points, but for each such point we now compute $H_i(SP), H_i^{(2)}(SP)$, and so on. Upon receiving a value $y = H(x)$ to invert, the attacker first computes $y' = F_i(y)$ and then checks if any of $y'$, $H_i(y')$, $\ldots$, $H_i^{(t-1)}(y')$ corresponds to an endpoint in the $i$th table; this is repeated for $i = 1, \ldots, T$. (We omit further details.) While it is difficult to argue independence formally, this approach leads to good results in practice.

**Choosing parameters.** Summarizing the above discussion, we see that as long as $st^2 = N/2$ we have an algorithm that stores $\mathcal{O}(s \cdot T) = \mathcal{O}(s \cdot t) = \mathcal{O}(N/t)$ points during a preprocessing phase, and can then invert $y$ with constant probability in time $\mathcal{O}(t \cdot T) = \mathcal{O}(t^2)$. One setting of the parameters is $t = N^{1/3} = 2^{\ell/3}$, in which case we have an algorithm storing $\mathcal{O}(2^{2\ell/3})$ points that finds preimages using $\mathcal{O}(2^{2\ell/3})$ hash computations. If a hash function with 80 bits of output is used, then this is feasible in practice.

**Handling different domain and range.** In practice, it is common to be faced with a situation in which the domain and range of $H$ are different. One example is in the context of password cracking (see Section 5.6.3), where an attacker has $H(pw)$ but $|pw| \ll \ell$. In the general case, say $x$ is chosen from some domain $D$ which may be larger or smaller than $\{0,1\}^\ell$. While it is, of course, possible to artificially expand the domain/range to make them match, this will *not* be useful for the attack described above. To see why, consider the password example. For the attack to succeed we want $pw$ to be in some table of values generated during preprocessing. If we generate each row of the table by simply computing $H(SP), H^{(2)}(SP), \ldots$, for $SP \in D$, then *none* of these values (except possibly $SP$ itself) will be equal to $pw$.

We can address this by applying a function $F_i$, as before, between each evaluation of $H$, though now we choose $F_i$ mapping $\{0,1\}^\ell$ to $D$. This solves the above issue, since $F_i(H(SP)), (F_i \circ H)^{(2)}(SP), \ldots$ now all lie in $D$.

**Applications to key-recovery attacks.** Time/space tradeoffs give attacks on cryptographic primitives other than hash functions. One canonical application—in fact, the application originally considered by Hellman—is an attack on an arbitrary block cipher $F$ that leads to recovery of the key. Define $H(k) \stackrel{\text{def}}{=} F_k(x)$ where $x$ is some arbitrary, but fixed, input that will be used for building the table. If an attacker can obtain $F_k(x)$ for an unknown key $k$—either via a chosen-plaintext attack or by choosing $x$ such that $F_k(x)$ is likely to be obtained in a known-plaintext attack—then by inverting $H$ the attacker learns (a candidate value for) $k$. Note that it is possible for the key length of $F$ to differ from its block length, but in this case we can use the technique just described for handling $H$ with different domain and range.

## 5.5 The Random-Oracle Model

There are several examples of constructions based on cryptographic hash functions that cannot be proven secure based only on the assumption that the hash function is collision or preimage resistant. (We will see some in the following section.) In many cases, there appears to be *no* simple and reasonable assumption regarding the hash function that would be sufficient for proving the construction secure.

Faced with this situation, there are several options. One is to look for schemes that *can* be proven secure based on some reasonable assumption about the underlying hash function. This is a good approach, but it leaves open the question of what to do until such schemes are found. Also, provably secure constructions may be significantly less efficient than other approaches that have not been proven secure. (This is a prominent issue that we will encounter in the setting of public-key cryptography.)

Another possibility, of course, is to use an existing cryptosystem even if it has no justification for its security other than, perhaps, the fact that the designers tried to attack it and were unsuccessful. This flies in the face of everything we have said about the importance of the rigorous, modern approach to cryptography, and it should be clear that this is unacceptable.

An approach that has been hugely successful in practice, and which offers a "middle ground" between a fully rigorous proof of security on the one hand and no proof whatsoever on the other, is to introduce an idealized model in which to prove the security of cryptographic schemes. Although the idealization may not be an accurate reflection of reality, we can at least derive some measure of confidence in the soundness of a scheme's design from a proof within the idealized model. As long as the model is reasonable, such proofs are certainly better than no proofs at all.

The most popular example of this approach is the *random-oracle model*, which treats a cryptographic hash function $H$ as a truly random function. (We have already seen an example of this in our discussion of time/space tradeoffs, although there we were analyzing an attack rather than a construction.) More specifically, the random-oracle model posits the existence of a public, random function $H$ that can be evaluated *only* by "querying" an oracle—which can be thought of as a "black box"—that returns $H(x)$ when given input $x$. (We will discuss how this is to be interpreted in the following section.) To differentiate things, the model we have been using until now (where no random oracle is present) is often called the "standard model."

No one claims that a random oracle exists, although there have been suggestions that a random oracle could be implemented in practice using a trusted party (i.e., some server on the Internet). Rather, the random-oracle model provides a formal *methodology* that can be used to design and validate cryptographic schemes using the following two-step approach:

1. First, a scheme is designed and proven secure in the random-oracle model. That is, we assume the world contains a random oracle, and construct and analyze a cryptographic scheme within this model. Standard cryptographic assumptions of the type we have seen until now may be utilized in the proof of security as well.
2. When we want to implement the scheme in the real world, a random oracle is not available. Instead, the random oracle is *instantiated* with an appropriately designed cryptographic hash function $\hat{H}$. (We return to this point at the end of this section.) That is, at each point where the scheme dictates that a party should query the oracle for the value $H(x)$, the party instead computes $\hat{H}(x)$ on its own.

The hope is that the cryptographic hash function used in the second step is "sufficiently good" at emulating a random oracle, so that the security proof given in the first step will carry over to the real-world instantiation of the scheme. The difficulty here is that there is no theoretical justification for this hope, and in fact there are (contrived) schemes that can be proven secure in the random-oracle model but are insecure *no matter how the random oracle is instantiated* in the second step. Furthermore, it is not clear (mathematically or heuristically) what it means for a hash function to be "sufficiently good" at emulating a random oracle, nor is it clear that this is an achievable goal. In particular, no concrete instantiation $\hat{H}$ can ever behave like a random function, since $\hat{H}$ is deterministic and fixed. For these reasons, a proof of security in the random-oracle model should be viewed as providing evidence that a scheme has no "inherent design flaws," but is *not* a rigorous proof that any real-world instantiation of the scheme is secure. Further discussion on how to interpret proofs in the random-oracle model is given in Section 5.5.2.

### 5.5.1 The Random-Oracle Model in Detail

Before continuing, let us pin down exactly what the random-oracle model entails. A good way to think about the random-oracle model is as follows: The "oracle" is simply a box that takes a binary string as input and returns a binary string as output. The internal workings of the box are unknown and inscrutable. Everyone—honest parties as well as the adversary—can interact with the box, where such interaction consists of feeding in a binary string $x$ as input and receiving a binary string $y$ as output; we refer to this as *querying the oracle on* $x$, and call $x$ itself a *query* made to the oracle. Queries to the oracle are assumed to be private so that if some party queries the oracle on input $x$ then no one else learns $x$, or even learns that this party queried the oracle at all. This makes sense, because calls to the oracle correspond (in the real-world instantiation) to local evaluations of a cryptographic hash function.

An important property of this "box" is that it is *consistent*. That is, if the box ever outputs $y$ for a particular input $x$, then it always outputs the same answer $y$ when given the same input $x$ again. This means that we can view the

box as implementing a well-defined function $H$; i.e., we define the function $H$ in terms of the input/output characteristics of the box. For convenience, we thus speak of "querying $H$" rather than querying the box. No one "knows" the entire function $H$ (except the box itself); at best, all that is known are the values of $H$ on the strings that have been explicitly queried thus far.

We have already discussed in Chapter 3 what it means to choose a random function $H$. We only reiterate here that there are two equivalent ways to think about the uniform selection of $H$: either picture $H$ being chosen "in one shot" uniformly from the set of all functions on some specified domain and range, or imagine generating outputs for $H$ "on-the-fly," as needed. Specifically, in the second case we can view the function as being defined by a table that is initially empty. When the oracle receives a query $x$ it first checks whether $x = x_i$ for some pair $(x_i, y_i)$ in the table; if so, the corresponding value $y_i$ is returned. Otherwise, a *uniform* string $y \in \{0,1\}^\ell$ is chosen (for some specified $\ell$), the answer $y$ is returned, and the oracle stores $(x, y)$ in its table. This second viewpoint is often conceptually easier to reason about, and is also technically easier to deal with if $H$ is defined over an infinite domain (e.g., $\{0,1\}^*$).

When we defined pseudorandom functions in Section 3.5.1, we also considered algorithms having oracle access to a random function. Lest there be any confusion, we note that the usage of a random function there is very different from the usage of a random function here. There, a random function was used *as a way of defining* what it means for a (concrete) keyed function to be pseudorandom. In the random-oracle model, in contrast, the random function is used *as part of a construction itself* and must somehow be instantiated in the real world if we want a concrete realization of the construction. A pseudorandom function is not a random oracle because it is only pseudorandom if the key is *secret*. However, in the random-oracle model all parties need to be able to compute the function; thus there can be no secret key.

### Definitions and Proofs in the Random-Oracle Model

Definitions in the random-oracle model are slightly different from their counterparts in the standard model because the probability spaces considered in each case are not the same. In the standard model a scheme $\Pi$ is secure if for all PPT adversaries $\mathcal{A}$ the probability of some event is below some threshold, where *this probability is taken over the random choices of the parties running* $\Pi$ *and those of the adversary* $\mathcal{A}$. Assuming the honest parties who use $\Pi$ in the real world make random choices as directed by the scheme, satisfying a definition of this sort guarantees security for real-world usage of $\Pi$.

In the random-oracle model, in contrast, a scheme $\Pi$ may rely on an oracle $H$. As before, $\Pi$ is secure if for all PPT adversaries $\mathcal{A}$ the probability of some event is below some threshold, but now *this probability is taken over random choice of* $H$ as well as the random choices of the parties running $\Pi$ and those of the adversary $\mathcal{A}$. When using $\Pi$ in the real world, some (instantiation of) $H$ must be fixed. Unfortunately, security of $\Pi$ is not guaranteed

for any *particular* choice of $H$. This indicates one reason why it is difficult to argue that any concrete instantiation of the oracle $H$ by a deterministic function yields a secure scheme. (An additional, technical, difficulty is that once a concrete function $H$ is fixed, the adversary $\mathcal{A}$ is no longer restricted to querying $H$ as an oracle but can instead look at and use the *code* of $H$ in the course of its attack.)

Proofs in the random-oracle model can exploit the fact that $H$ is chosen at random, and that the only way to evaluate $H(x)$ is to explicitly query $x$ to $H$. Three properties in particular are especially useful; we sketch them informally here, and show some simple applications of them below and in the next section, but caution that a full understanding will likely have to wait until we present formal proofs in the random-oracle model in later chapters.

The first useful property of the random-oracle model is:

> *If $x$ has not been queried to $H$, then the value of $H(x)$ is* **uniform**.

This may seem superficially similar to the guarantee provided by a pseudorandom generator, but is actually much stronger. If $G$ is a pseudorandom generator then $G(x)$ is pseudorandom to an observer *assuming $x$ is chosen uniformly at random and is completely unknown to the observer*. If $H$ is a random oracle, however, then $H(x)$ is truly uniform to an observer as long as the observer has not queried $x$. This is true even if $x$ is known, or if $x$ is not uniform but *is* hard to guess. (For example, if $x$ is an $n$-bit string where the first half of $x$ is known and the last half is random then $G(x)$ might be easy to distinguish from random but $H(x)$ will not be.)

The remaining two properties relate explicitly to *proofs by reduction* in the random-oracle model. (It may be helpful here to review Section 3.3.2.) As part of the reduction, the random oracle that the adversary $\mathcal{A}$ interacts with must be simulated. That is: $\mathcal{A}$ will submit queries to, and receive answers from, what it believes to be the oracle, but the reduction itself must now answer these queries. This turns out to give a lot of power. For starters:

> *If $\mathcal{A}$ queries $x$ to $H$, the reduction can* **see this query** *and learn $x$.*

This is sometimes called "extractability." (This does not contradict the fact, mentioned earlier, that queries to the random oracle are "private." While that is true in the random-oracle model itself, here we are using $\mathcal{A}$ as a subroutine within a reduction that is simulating the random oracle for $\mathcal{A}$.) Finally:

> *The reduction can* **set** *the value of $H(x)$ (i.e., the response to query $x$) to a value of its choice, as long as this value is correctly distributed, i.e., uniform.*

This is called "programmability." There is no counterpart to extractability or programmability once $H$ is instantiated with any concrete function.

### Simple Illustrations of the Random-Oracle Model

At this point some examples may be helpful. The examples given here are relatively simple, and do not use the full power that the random-oracle model affords. Rather, these examples are presented merely to provide a gentle introduction to the model. In what follows, we assume a random oracle mapping $\ell_{in}$-bit inputs to $\ell_{out}$-bit outputs, where $\ell_{in}, \ell_{out} > n$, the security parameter (so $\ell_{in}, \ell_{out}$ are functions of $n$).

**A random oracle as a pseudorandom generator.** We first show that, for $\ell_{out} > \ell_{in}$, a random oracle can be used as a pseudorandom generator. (We do not say that a random oracle *is* a pseudorandom generator, since a random oracle is not a fixed function.) Formally, we claim that for any PPT adversary $\mathcal{A}$, there is a negligible function $\mathsf{negl}$ such that

$$\left| \Pr[\mathcal{A}^{H(\cdot)}(y) = 1] - \Pr[\mathcal{A}^{H(\cdot)}(H(x)) = 1] \right| \leq \mathsf{negl}(n),$$

where in the first case the probability is taken over uniform choice of $H$, uniform choice of $y \in \{0,1\}^{\ell_{out}(n)}$, and the randomness of $\mathcal{A}$, and in the second case the probability is taken over uniform choice of $H$, uniform choice of $x \in \{0,1\}^{\ell_{in}(n)}$, and the randomness of $\mathcal{A}$. We have explicitly indicated that $\mathcal{A}$ has oracle access to $H$ in each case; once $H$ has been chosen then $\mathcal{A}$ can freely make queries to it.

Let $S$ denote the set of points on which $\mathcal{A}$ has queried $H$; of course, $|S|$ is polynomial in $n$. Observe that in the second case, the probability that $x \in S$ is negligible. This holds since $\mathcal{A}$ starts with no information about $x$ (note that $H(x)$ by itself reveals nothing about $x$ because $H$ is a random function), and because $S$ is exponentially smaller than $\{0,1\}^{\ell_{in}}$. Moreover, conditioned on $x \notin S$ in the second case, $\mathcal{A}$'s input in each case is a uniform string that is independent of the answers to $\mathcal{A}$'s queries.

**A random oracle as a collision-resistant hash function.** If $\ell_{out} < \ell_{in}$, a random oracle is collision resistant. That is, the success probability of any PPT adversary $\mathcal{A}$ in the following experiment is negligible:

1. A random function $H$ is chosen.
2. $\mathcal{A}$ succeeds if it outputs distinct $x, x'$ with $H(x) = H(x')$.

To see this, assume without loss of generality that $\mathcal{A}$ only outputs values $x, x'$ that it had previously queried to the oracle, and that $\mathcal{A}$ never makes the same query to the oracle twice. Letting the oracle queries of $\mathcal{A}$ be $x_1, \ldots, x_q$, with $q = \mathsf{poly}(n)$, it is clear that the probability that $\mathcal{A}$ succeeds is upper-bounded by the probability that $H(x_i) = H(x_j)$ for some $i \neq j$. But this is exactly equal to the probability that if we pick $q$ strings $y_1, \ldots, y_q \in \{0,1\}^{\ell_{out}}$ independently and uniformly at random, we have $y_i = y_j$ for some $i \neq j$. This is exactly the birthday problem, and so using the results of Appendix A.4 we have that $\mathcal{A}$ succeeds with negligible probability $\mathcal{O}(q^2/2^{\ell_{out}})$.

**Constructing a pseudorandom function from a random oracle.** It is also rather easy to construct a pseudorandom function in the random-oracle model. Suppose $\ell_{in}(n) = 2n$ and $\ell_{out}(n) = n$, and define

$$F_k(x) \stackrel{\text{def}}{=} H(k\|x),$$

where $|k| = |x| = n$. In Exercise 5.11 you are asked to show that this is a pseudorandom function, namely, that for any polynomial-time $\mathcal{A}$ the success probability of $\mathcal{A}$ in the following experiment is not more than $1/2$ plus a negligible function:

1. A function $H$ and values $k \in \{0,1\}^n$ and $b \in \{0,1\}$ are chosen uniformly.
2. If $b = 0$, the adversary $\mathcal{A}$ is given access to an oracle for $F_k(\cdot) = H(k\|\cdot)$. If $b = 1$, then $\mathcal{A}$ is given access to a random function mapping $n$-bit inputs to $n$-bit outputs. (This random function is *independent* of $H$.)
3. $\mathcal{A}$ outputs a bit $b'$, and succeeds if $b' = b$.

In step 2, $\mathcal{A}$ can access $H$ in addition to the function oracle provided to it by the experiment. (A pseudorandom function in the random-oracle model must be indistinguishable from a random function that is independent of $H$.)

An interesting aspect of all the above claims is that they make no computational assumptions; they hold even for computationally unbounded adversaries as long as those adversaries are limited to making polynomially many queries to the oracle. This has no counterpart in the real world, where we have seen that computational assumptions are necessary.

### 5.5.2 Is the Random-Oracle Methodology Sound?

Schemes designed in the random-oracle model are implemented in the real world by instantiating $H$ with some concrete function. With the mechanics of the random-oracle model behind us, we turn to a more fundamental question:

> *What do proofs of security in the random-oracle model guarantee as far as security of any real-world instantiation?*

This question does not have a definitive answer: there is currently debate within the cryptographic community regarding how to interpret proofs in the random-oracle model, and an active area of research is to determine what, precisely, a proof of security in the random-oracle model implies vis-a-vis the real world. We can only hope to give a flavor of both sides of the debate.

**Objections to the random-oracle model.** The starting point for arguments against using random oracles is simple: as we have already noted, there is no formal or rigorous justification for believing that a proof of security for some scheme $\Pi$ in the random-oracle model says anything about the security of $\Pi$ in the real world, once the random oracle $H$ has been instantiated with

any particular hash function $\hat{H}$. This is more than just theoretical uneasiness. A little thought shows that *no* concrete hash function can ever act as a "true" random oracle. For example, in the random-oracle model the value $H(x)$ is "completely random" if $x$ was not explicitly queried. The counterpart would be to require that $\hat{H}(x)$ is random (or pseudorandom) if $\hat{H}$ was not explicitly evaluated on $x$. How are we to interpret this in the real world? It is not even clear what it means to "explicitly evaluate" $\hat{H}$: what if an adversary knows some shortcut for computing $\hat{H}$ that does not involve running the actual code for $\hat{H}$? Moreover, $\hat{H}(x)$ cannot possibly be random (or even pseudorandom) since once the adversary learns the description of $\hat{H}$, the value of that function on *all* inputs is immediately determined.

Limitations of the random-oracle model become clearer once we examine the proof techniques introduced earlier. Recall that one proof technique is to use the fact that a reduction can "see" the queries that an adversary $\mathcal{A}$ makes to the random oracle. If we replace the random oracle by a particular hash function $\hat{H}$, this means we must provide a description of $\hat{H}$ to the adversary at the beginning of the experiment. But then $\mathcal{A}$ can evaluate $\hat{H}$ on its own, without making any *explicit* queries, and so a reduction will no longer have the ability to "see" any queries made by $\mathcal{A}$. (In fact, as noted previously, the notion of $\mathcal{A}$ performing explicit evaluations of $\hat{H}$ may not be true and certainly cannot be formally defined.) Likewise, proofs of security in the random-oracle model allow the reduction to choose the outputs of $H$ as it wishes, something that is clearly not possible when a concrete function is used.

Even if we are willing to overlook the above theoretical concerns, a practical problem is that we do not currently have a very good understanding of what it means for a concrete hash function to be "sufficiently good" at instantiating a random oracle. For concreteness, say we want to instantiate the random oracle using some appropriate modification of SHA-1 (SHA-1 is a cryptographic hash function discussed in Section 6.3.3). While for some particular scheme $\Pi$ it might be reasonable to assume that $\Pi$ is secure when instantiated using SHA-1, it is much less reasonable to assume that SHA-1 can take the place of a random oracle in *every* scheme designed in the random-oracle model. Indeed, as we have said earlier, we *know* that SHA-1 is not a random oracle. And it is not hard to design a scheme that is secure in the random-oracle model, but is insecure when the random oracle is replaced by SHA-1.

We emphasize that an assumption of the form "SHA-1 acts like a random oracle" is qualitatively different from assumptions such as "SHA-1 is collision resistant" or "AES is a pseudorandom function." The problem lies partly with the fact that there is no satisfactory *definition* of what the first statement means, while we do have such definitions for the latter two statements.

Because of this, using the random-oracle model to prove security of a scheme is *qualitatively* different from, e.g., introducing a new cryptographic assumption in order to prove a scheme secure in the standard model. Therefore, proofs of security in the random-oracle model are less satisfying than proofs of security in the standard model.

**Support for the random-oracle model.** Given all the problems with the random-oracle model, why use it at all? More to the point: why has the random-oracle model been so influential in the development of modern cryptography (especially current practical usage of cryptography), and why does it continue to be so widely used? As we will see, the random-oracle model enables the design of substantially more efficient schemes than those we know how to construct in the standard model. As such, there are few (if any) public-key cryptosystems used today having proofs of security in the standard model, while there are numerous deployed schemes having proofs of security in the random-oracle model. In addition, proofs in the random-oracle model are almost universally recognized as lending confidence to the security of schemes being considered for standardization.

The fundamental reason for this is the belief that:

> *A proof of security in the random-oracle model is significantly better than no proof at all.*

Although some disagree, we offer the following in support of this assertion:

- A proof of security for a scheme in the random-oracle model indicates that the scheme's design is "sound," in the sense that the only possible attacks on a real-world instantiation of the scheme are those that arise due to a weakness in the hash function used to instantiate the random oracle. Thus, if a "good enough" hash function is used to instantiate the random oracle, we should have confidence in the security of the scheme. Moreover, if a given instantiation of the scheme *is* successfully attacked, we can simply replace the hash function being used with a "better" one.
- Importantly, *there have been no successful real-world attacks on schemes proven secure in the random-oracle model*, when the random oracle was instantiated properly. (We do not include here attacks on "contrived" schemes, but remark that great care must be taken in instantiating the random oracle, as indicated by Exercise 5.10.) This gives evidence to the usefulness of the random-oracle model in designing practical schemes.

Nevertheless, the above ultimately represent only intuitive speculation as to the usefulness of proofs in the random-oracle model and—all else being equal—proofs without random oracles are preferable.

## Instantiating the Random Oracle

Properly instantiating a random oracle is subtle, and a full discussion is beyond the scope of this book. Here we only alert the reader that using an "off-the-shelf" cryptographic hash function without modification is not, generally speaking, a sound approach. For one thing, most cryptographic hash functions are constructed using the Merkle–Damgård paradigm (cf. Section 5.2), which can be distinguished easily from a random oracle when variable-length inputs

are allowed. (See Exercise 5.10.) Also, in some constructions it is necessary for the output of the random oracle to lie in a certain range (e.g., the oracle should output elements of some group), which results in additional complications.

## 5.6 Additional Applications of Hash Functions

We conclude this chapter with a brief discussion of some additional applications of cryptographic hash functions in cryptography and computer security.

### 5.6.1 Fingerprinting and Deduplication

When using a collision-resistant hash function $H$, the hash (or *digest*) of a file serves as a unique identifier for that file. (If any other file is found to have the same identifier, this implies a collision in $H$). The hash $H(x)$ of a file $x$ is like a fingerprint, and one can check whether two files are equal by comparing their digests. This simple idea has many applications.

- *Virus fingerprinting:* Virus scanners identify viruses and block or quarantine them. One of the most basic steps toward this goal is to store a database containing the hashes of known viruses, and then to look up the hash of a downloaded application or email attachment in this database. Since only a short string needs to be recorded (and/or distributed) for each virus, the overhead involved is feasible.

- *Deduplication:* Data deduplication is used to eliminate duplicate copies of data, especially in the context of cloud storage where multiple users rely on a single cloud service to store their data. The observation here is that if multiple users wish to store the same file (e.g., a popular video), then the file only needs to be stored once and need not be uploaded separately by each user. Deduplication can be achieved by first having a user upload a hash of the new file they want to store; if a file with this hash is already stored in the cloud, then the cloud-storage provider can simply add a pointer to the existing file to indicate that this specific user has also stored this file. This saves both communication and storage, and the soundness of the methodology follows from the collision resistance of the hash function.

- *Peer-to-peer (P2P) file sharing*: In P2P file-sharing systems, tables are held by servers to provide a file-lookup service. These tables contain the hashes of the available files, once again providing a unique identifier without using much memory.

It may be surprising that a small digest can uniquely identify every file in the world. But this is the guarantee provided by collision-resistant hash functions, which makes them useful in the settings above.

### 5.6.2 Merkle Trees

Consider a client who uploads a file $x$ to a server. When the client later retrieves $x$, it wants to make sure that the server returns the original, unmodified file. The client could simply store $x$ and check that the retrieved file is equal to $x$, but that defeats the purpose of using the server in the first place. We are looking for a solution in which the storage of the client is small.

A natural solution is to use the "fingerprinting" approach described above. The client can locally store the short digest $h := H(x)$; when the server returns a candidate file $x'$ the client need only check that $H(x') \stackrel{?}{=} h$.

What happens if we want to extend this solution to *multiple* files $x_1, \ldots, x_t$? There are two obvious ways of doing this. One is to simply hash each file independently; the client will locally store the digests $h_1, \ldots, h_t$, and verify retrieved files as before. This has the disadvantage that the client's storage grows linearly in $t$. Another possibility is to hash all the files together. That is, the client can compute $h := H(x_1, \ldots, x_t)$ and store only $h$. The drawback now is that when the client wants to retrieve and verify correctness of the $i$th file $x_i$, it needs to retrieve *all* the files in order to recompute the digest.

*Merkle trees*, introduced by Ralph Merkle, give a tradeoff between these extremes. A Merkle tree computed over input values $x_1, \ldots, x_t$ is simply a binary tree of depth $\log t$ in which the inputs are placed at the leaves, and the value of each internal node is the hash of the values of its two children; see Figure 5.5. (We assume $t$ is a power of 2; if not, then we can fix some input values to null or use an incomplete binary tree, depending on the application.)

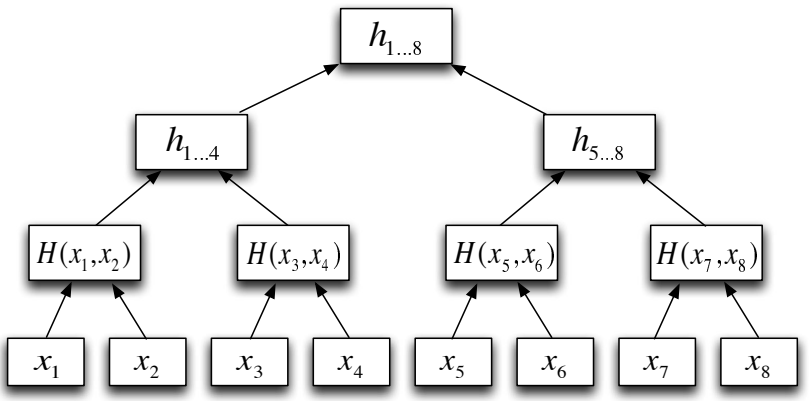


**FIGURE 5.5**: A Merkle tree.

Fixing some hash function $H$, we denote by $\mathcal{MT}_t$ the function that takes $t$ input values $x_1, \ldots, x_t$, computes the resulting Merkle tree, and outputs the value of the root of the tree. (A keyed hash function yields a keyed function $\mathcal{MT}_t$ in the obvious way.) We have:

***THEOREM 5.11*** *Let* $(\mathsf{Gen}_H, H)$ *be collision resistant. Then* $(\mathsf{Gen}_H, \mathcal{MT}_t)$ *is also collision resistant for any fixed* $t$*.*

Merkle trees thus provide an alternative to the Merkle–Damgård transform for achieving domain extension for collision-resistant hash functions. (As described, however, Merkle trees are *not* collision resistant if the number of input values $t$ is allowed to vary.)

Merkle trees provide an efficient solution to our original problem, since they allow verification of any of the original $t$ inputs using $\mathcal{O}(\log t)$ communication. The client computes $h := \mathcal{MT}_t(x_1, \ldots, x_t)$, uploads $x_1, \ldots, x_t$ to the server, and stores $h$ (along with the number of files $t$) locally. When the client retrieves the $i$th file, the server sends $x_i$ along with a "proof" $\pi_i$ that this is the correct value. This proof consists of the values of the nodes in the Merkle tree adjacent to the path from $x_i$ to the root. From these values the client can recompute the value of the root and verify that it is equal to the stored value $h$. As an example, consider the Merkle tree in Figure 5.5. The client computes $h_{1\ldots8} := \mathcal{MT}_8(x_1, \ldots, x_8)$, uploads $x_1, \ldots, x_8$ to the server, and stores $h_{1\ldots8}$ locally. When the client retrieves $x_3$, the server sends $x_3$ along with $x_4$, $h_{1\ldots2} = H(x_1, x_2)$, and $h_{5\ldots8} = H(H(x_5, x_6), H(x_7, x_8))$. (If files are large we may wish to avoid sending any file other than the one the client has requested. That can easily be done if we define the Merkle tree over the *hashes* of the files rather than the files themselves. We omit the details.) The client computes $h'_{1\ldots4} := H(h_{1\ldots2}, H(x_3, x_4))$ and $h'_{1\ldots8} := H(h'_{1\ldots4}, h_{5\ldots8})$, and then verifies that $h_{1\ldots8} \stackrel{?}{=} h'_{1\ldots8}$.

If $H$ is collision resistant, it is infeasible for the server to send an incorrect file (and any proof) that will cause verification to succeed. Using this approach, the client's local storage is *constant* (independent of the number of files $t$), and the communication from server to client is proportional to $\log t$.

### 5.6.3 Password Hashing

One of the most common and important uses of hash functions in computer security is for password protection. Consider a user typing in a password before using their laptop. To authenticate the user, some form of the user's password must be stored somewhere on their laptop. If the user's password is stored in the clear, then an adversary who steals the laptop can read the user's password off the hard drive and then login as that user. (It may seem pointless to try to hide one's password from an attacker who can already read the contents of the hard drive. However, files on the hard drive may be

encrypted with a key derived from the user's password, and would thus only be accessible after the password is entered. In addition, the user is likely to use the same password at other sites.)

This risk can be mitigated by storing a *hash of the password* instead of the password itself. That is, the hard drive stores the value $hpw = H(pw)$ in a password file; later, when the user enters its password $pw$, the operating system checks whether $H(pw) \stackrel{?}{=} hpw$ before granting access. The same basic approach is also used for password-based authentication on the web. Now, if an attacker steals the hard drive (or breaks into a web server), all it obtains is the hash of the password and not the password itself.

If the password is chosen from some relatively small space $D$ of possibilities (e.g., $D$ might be a dictionary of English words, in which case $|D| \approx 80,000$), an attacker can enumerate all possible passwords $pw_1, pw_2, \ldots \in D$ and, for each candidate $pw_i$, check whether $H(pw_i) = hpw$. We would like to claim that an attacker can do no better than this. (This would also ensure that the adversary could not learn the password of any user who chose a *strong* password from a large space.) Unfortunately, preimage resistance (i.e., one-wayness) of $H$ is not sufficient to imply what we want. For one thing, preimage resistance only says that $H(x)$ is hard to invert when $x$ is chosen uniformly from a large domain like $\{0,1\}^n$. It says nothing about the hardness of inverting $H$ when $x$ is chosen from some other space, or when $x$ is chosen according to some other distribution. Moreover, preimage resistance says nothing about the *concrete* amount of time needed to find a preimage. For example, a hash function $H$ for which computing $x \in \{0,1\}^n$ given $H(x)$ requires time $2^{n/2}$ could still qualify as preimage resistant, yet this would mean that a 30-bit password could be recovered in only $2^{15}$ time.

If we model $H$ as a random oracle, then we can formally prove the security we want, namely, recovering $pw$ from $hpw$ (assuming $pw$ is chosen uniformly from $D$) requires $|D|/2$ evaluations of $H$, on average.

The above discussion assumes no preprocessing is done by the attacker. As we have seen in Section 5.4.3, though, preprocessing can be used to generate large tables that enable inversion (even of a random function!) faster than exhaustive search. This is a significant concern in practice: even if a user chooses their password as a random combination of 8 alphanumeric characters—giving a password space of size $N = 62^8 \approx 2^{47.6}$—there is an attack using time and space $N^{2/3} \approx 2^{32}$ that will be highly effective. The tables only need to be generated once, and can be used to crack hundreds of thousands of passwords in case of a server breach. Such attacks are routinely carried out in practice.

**Mitigation.** We briefly describe two mechanisms used to mitigate the threat of password cracking; further discussion can be found in texts on computer security. One technique is to use "slow" hash functions, or to slow down existing hash functions by using multiple iterations (i.e., computing $H^{(I)}(pw)$ for $I \gg 1$). This has the effect of slowing down legitimate users by a factor of $I$, which is not a problem if $I$ is set to some "moderate" value (e.g., 1,000).

On the other hand, it has a significant impact on an adversary attempting to crack thousands of passwords at once.

A second mechanism is to introduce a *salt*. When a user registers their password, the laptop/server will generate a long random value $s$ (a "salt") unique to that user, and store $(s, hpw = H(s, pw))$ instead of merely storing $H(pw)$ as before. Since $s$ is unknown to the attacker in advance, preprocessing is ineffective and the best an attacker can do is to wait until it obtains the password file and then do a linear-time exhaustive search over the domain $D$ as discussed before. Note also that since a different salt is used for each stored password, a separate brute-force search is needed to recover each password.

### 5.6.4 Key Derivation

All the symmetric-key cryptosystems we have seen require a *uniformly distributed* bit-string for the secret key. Often, however, it is more convenient for two parties to rely on shared information such as a password or biometric data that is *not* uniformly distributed. (Jumping ahead, in Chapter 10 we will see how parties can interact to generate a high-entropy shared secret that is *not* uniformly distributed.) The parties could try to use their shared information directly as a secret key, but in general this will not be secure (since, e.g., private-key schemes all assume a uniformly distributed key). Moreover, the shared data may not even have the correct format to be used as a secret key (it may be too long, for example).

Truncating the shared secret, or mapping it in some other ad hoc way to a string of the correct length, may lose a significant amount of entropy. (We define one notion of entropy more formally below, but for now one can think of entropy as the logarithm of the space of possible shared secrets.) For example, imagine two parties share a password composed of 28 random uppercase letters, and want to use a cryptosystem with a 128-bit key. Since there are 26 possibilities for each character, there are $26^{28} > 2^{130}$ possible passwords. If the password is shared in ASCII format, each character is stored using 8 bits, and so the total length of the password is 224 bits. If the parties truncate their password to the first 128 bits, they will be using only the first 16 characters of their password. However, this will not be a uniformly distributed 128-bit string! In fact, the ASCII representations of the letters A–Z lie between 0x41 and 0x5A; in particular, the first 3 bits of every byte are always 010. This means that *37.5% of the bits of the resulting key will be fixed*, and the 128-bit key the parties derive will have only about 75 bits of entropy (i.e., there are only $2^{75}$ or so possibilities for the key).

What we need is a generic solution for deriving a key from a high-entropy (but not necessarily uniform) shared secret. Before continuing, we define the notion of entropy we consider here.

***DEFINITION 5.12*** *A probability distribution $\mathcal{X}$ has $m$* bits of min-entropy *if for every fixed value $x$ it holds that $\Pr_{X \leftarrow \mathcal{X}}[X = x] \leq 2^{-m}$. That is, even the most likely outcome occurs with probability at most $2^{-m}$.*

The uniform distribution over a set of size $S$ has min-entropy $\log S$. A distribution in which one element occurs with probability $1/10$ and 90 elements each occur with probability $1/100$ has min-entropy $\log 10 \approx 3.3$. The min-entropy of a distribution measures the probability with which an attacker can guess a value sampled from that distribution; the attacker's best strategy is to guess the most likely value, and so if the distribution has min-entropy $m$ the attacker guesses correctly with probability at most $2^{-m}$. This explains why min-entropy (rather than other notions of entropy) is useful in our context. An extension of min-entropy, called *computational min-entropy*, is defined as above except that the distribution is only required to be *computationally indistinguishable* from a distribution with the given min-entropy. (The notion of computational indistinguishability is formally defined in Section 7.8.)

A *key-derivation function* provides a way to obtain a uniformly distributed string from any distribution with high (computational) min-entropy. It is not hard to see that if we model a hash function $H$ as a random oracle, then $H$ serves as a good key-derivation function. Consider an attacker's uncertainty about $H(X)$, where $X$ is sampled from a distribution with min-entropy $m$ (as a technical point, we require the distribution to be independent of $H$). Each of the attacker's queries to $H$ can be viewed as a "guess" for the value of $X$; by assumption on the min-entropy of the distribution, an attacker making $q$ queries to $H$ will query $H(X)$ with probability at most $q \cdot 2^{-m}$. If the attacker does not query $X$ to $H$, then $H(X)$ is a uniform string.

It is also possible to design key-derivation functions, without relying on the random-oracle model, using keyed hash functions called *(strong) extractors*. The key for the extractor must be uniform, but need not be kept secret. One standard for this is called HKDF; see the references at the end of the chapter.

### 5.6.5 Commitment Schemes

A *commitment scheme* allows one party to "commit" to a message $m$ by sending a commitment value com, while obtaining the following seemingly contradictory properties:

- *Hiding:* the commitment reveals nothing about $m$.
- *Binding:* it is infeasible for the committer to output a commitment com that it can later "open" as two different messages $m, m'$. (In this sense, com truly "commits" the committer to some well-defined value.)

A commitment scheme can be seen as a digital envelope: sealing a message in an envelope and handing it over to another party provides privacy (until the

envelope is opened) and binding (since the envelope is sealed).

Formally, a (non-interactive) commitment scheme is defined by a randomized algorithm $\mathsf{Gen}$ that outputs public parameters $\mathsf{params}$ and an algorithm $\mathsf{Com}$ that takes $\mathsf{params}$ and a message $m \in \{0,1\}^n$ and outputs a commitment $\mathsf{com}$; we will make the randomness used by $\mathsf{Com}$ explicit, and denote it by $r$. A sender commits to $m$ by choosing uniform $r$, computing $\mathsf{com} := \mathsf{Com}(\mathsf{params}, m; r)$, and sending it to a receiver. The sender can later decommit $\mathsf{com}$ and reveal $m$ by sending $m, r$ to the receiver; the receiver verifies this by checking that $\mathsf{Com}(\mathsf{params}, m; r) \stackrel{?}{=} \mathsf{com}$.

Hiding, informally, means that $\mathsf{com}$ reveals nothing about $m$; binding means that it is impossible to output a commitment $\mathsf{com}$ that can be opened two different ways. We define these properties formally now.

**The commitment hiding experiment** $\mathsf{Hiding}_{\mathcal{A},\mathsf{Com}}(n)$**:**

1. *Parameters* $\mathsf{params} \leftarrow \mathsf{Gen}(1^n)$ *are generated.*
2. *The adversary* $\mathcal{A}$ *is given input* $\mathsf{params}$*, and outputs a pair of messages* $m_0, m_1 \in \{0,1\}^n$*.*
3. *A uniform* $b \in \{0,1\}$ *is chosen and* $\mathsf{com} \leftarrow \mathsf{Com}(\mathsf{params}, m_b; r)$ *is computed.*
4. *The adversary* $\mathcal{A}$ *is given* $\mathsf{com}$ *and outputs a bit* $b'$*.*
5. *The output of the experiment is* 1 *if and only if* $b' = b$*.*

**The commitment binding experiment** $\mathsf{Binding}_{\mathcal{A},\mathsf{Com}}(n)$**:**

1. *Parameters* $\mathsf{params} \leftarrow \mathsf{Gen}(1^n)$ *are generated.*
2. $\mathcal{A}$ *is given input* $\mathsf{params}$ *and outputs* $(\mathsf{com}, m, r, m', r')$*.*
3. *The output of the experiment is defined to be* 1 *if and only if* $m \neq m'$ *and* $\mathsf{Com}(\mathsf{params}, m; r) = \mathsf{com} = \mathsf{Com}(\mathsf{params}, m'; r')$*.*

***DEFINITION 5.13*** *A commitment scheme* $\mathsf{Com}$ *is* secure *if for all* PPT *adversaries* $\mathcal{A}$ *there is a negligible function* $\mathsf{negl}$ *such that*

$$\Pr\left[\mathsf{Hiding}_{\mathcal{A},\mathsf{Com}}(n) = 1\right] \leq \tfrac{1}{2} + \mathsf{negl}(n)$$

and

$$\Pr\left[\mathsf{Binding}_{\mathcal{A},\mathsf{Com}}(n) = 1\right] \leq \mathsf{negl}(n).$$

It is easy to construct a secure commitment scheme from a random oracle $H$. To commit to a message $m$, the sender chooses uniform $r \in \{0,1\}^n$ and outputs $\mathsf{com} := H(m\|r)$. (In the random-oracle model, $\mathsf{Gen}$ and $\mathsf{params}$ are not needed since $H$, in effect, serves as the public parameters of the scheme.) Intuitively, hiding follows from the fact that an adversary queries $H(\star\|r)$ with

only negligible probability (since $r$ is a uniform $n$-bit string); if it never makes a query of this form then $H(m\|r)$ reveals nothing about $m$. Binding follows from the fact that $H$ is collision resistant.

Commitment schemes can be constructed without random oracles (in fact, from one-way functions), but the details are beyond the scope of this book.

## References and Additional Reading

Collision-resistant hash functions were formally defined by Damgård [52]. Additional discussion regarding notions of security for hash functions besides collision resistance can be found in [120, 150]. The Merkle–Damgård transform was introduced independently by Damgård and Merkle [53, 123]

HMAC was introduced by Bellare et al. [14] and later standardized [131].

The small-space birthday attack described in Section 5.4.2 relies on a cycle-finding algorithm of Floyd. Related algorithms and results are described at `http://en.wikipedia.org/wiki/Cycle_detection`. The idea for finding meaningful collisions using the small-space attack is by Yuval [180]. The possibility of parallelizing collision-finding attacks, which can offer significant speedups in practice, is discussed in [170]. Time/space tradeoffs for function inversion were introduced by Hellman [87], with practical improvements—not discussed here—given by Rivest (unpublished) and by Oechslin [134].

The first formal treatment of the random-oracle model was given by Bellare and Rogaway [21], although the idea of using a "random-looking" function in cryptographic applications had been suggested previously, most notably by Fiat and Shamir [65]. Proper instantiation of a random oracle based on concrete cryptographic hash functions is discussed in [21, 22, 23, 48]. The seminal negative result concerning the random-oracle model is that of Canetti et al. [41], who show (contrived) schemes that are secure in the random-oracle model but are insecure for *any* concrete instantiation of the random oracle.

Merkle trees were introduced in [121]. Key-derivation functions used in practice include HKDF, PBKDF2, and bcrypt. See [109] for a formal treatment of the problem and an analysis of HKDF.

## Exercises

5.1 Provide formal definitions for second preimage resistance and preimage resistance. Prove that any hash function that is collision resistant is second preimage resistant, and any hash function that is second preimage resistant is preimage resistant.

5.2 Let $(\mathsf{Gen}_1, H_1)$ and $(\mathsf{Gen}_2, H_2)$ be two hash functions. Define $(\mathsf{Gen}, H)$ so that $\mathsf{Gen}$ runs $\mathsf{Gen}_1$ and $\mathsf{Gen}_2$ to obtain keys $s_1$ and $s_2$, respectively. Then define $H^{s_1,s_2}(x) = H_1^{s_1}(x) \| H_2^{s_2}(x)$.

(a) Prove that if at least one of $(\mathsf{Gen}_1, H_1)$ and $(\mathsf{Gen}_2, H_2)$ is collision resistant, then $(\mathsf{Gen}, H)$ is collision resistant.

(b) Determine whether an analogous claim holds for second preimage resistance and preimage resistance, respectively. Prove your answer in each case.

5.3 Let $(\mathsf{Gen}, H)$ be a collision-resistant hash function. Is $(\mathsf{Gen}, \hat{H})$ defined by $\hat{H}^s(x) \stackrel{\text{def}}{=} H^s(H^s(x))$ necessarily collision resistant?

5.4 Provide a formal proof of Theorem 5.4 (i.e., describe the reduction).

5.5 Generalize the Merkle–Damgård transform (Construction 5.3) for the case when the fixed-length hash function $h$ has input length $n + \kappa$ (with $\kappa > 0$) and output length $n$, and the length of the input to $H$ should be encoded as an $\ell$-bit value (as discussed in Section 5.3.2). Prove collision resistance of $(\mathsf{Gen}, H)$, assuming collision resistance of $(\mathsf{Gen}, h)$.

5.6 For each of the following modifications to the Merkle–Damgård transform (Construction 5.3), determine whether the result is collision resistant. If yes, provide a proof; if not, demonstrate an attack.

(a) Modify the construction so that the input length is not included at all (i.e., output $z_B$ and not $z_{B+1} = h^s(z_B \| L)$). (Assume the resulting hash function is only defined for inputs whose length is an integer multiple of the block length.)

(b) Modify the construction so that instead of outputting $z = h^s(z_B \| L)$, the algorithm outputs $z_B \| L$.

(c) Instead of using an $IV$, just start the computation from $x_1$. That is, define $z_1 := x_1$ and then compute $z_i := h^s(z_{i-1} \| x_i)$ for $i = 2, \ldots, B+1$ and output $z_{B+1}$ as before.

(d) Instead of using a fixed $IV$, set $z_0 := L$ and then compute $z_i := h^s(z_{i-1} \| x_i)$ for $i = 1, \ldots, B$ and output $z_B$.

5.7 Assume collision-resistant hash functions exist. Show a construction of a fixed-length hash function $(\mathsf{Gen}, h)$ that is *not* collision resistant, but such that the hash function $(\mathsf{Gen}, H)$ obtained from the Merkle–Damgård transform to $(\mathsf{Gen}, h)$ as in Construction 5.3 *is* collision resistant.

5.8 Prove or disprove: if $(\mathsf{Gen}, h)$ is preimage resistant, then so is the hash function $(\mathsf{Gen}, H)$ obtained by applying the Merkle–Damgård transform to $(\mathsf{Gen}, h)$ as in Construction 5.3.

5.9 Prove or disprove: if $(\mathsf{Gen}, h)$ is second preimage resistant, then so is the hash function $(\mathsf{Gen}, H)$ obtained by applying the Merkle–Damgård transform to $(\mathsf{Gen}, h)$ as in Construction 5.3.

5.10 Before HMAC, it was common to define a MAC for arbitrary-length messages by $\mathsf{Mac}_{s,k}(m) = H^s(k\|m)$ where $H$ is a collision-resistant hash function.

(a) Show that this is never a secure MAC when $H$ is constructed via the Merkle–Damgård transform. (Assume the hash key $s$ is known to the attacker, and only $k$ is kept secret.)

(b) Prove that this is a secure MAC if $H$ is modeled as a random oracle.

5.11 Prove that the construction of a pseudorandom function given in Section 5.5.1 is secure in the random-oracle model.

5.12 Prove Theorem 5.11.

5.13 Show how to find a collision in the Merkle tree construction if $t$ is not fixed. Specifically, show how to find two sets of inputs $x_1, \ldots, x_t$ and $x'_1, \ldots, x'_{2t}$ such that $\mathcal{MT}_t(x_1, \ldots, x_t) = \mathcal{MT}_{2t}(x'_1, \ldots, x'_{2t})$.

5.14 Consider the scenario introduced in Section 5.6.2 in which a client stores files on a server and wants to verify that files are returned unmodified.

(a) Provide a formal definition of security for this setting.

(b) Formalize the construction based on Merkle trees as discussed in Section 5.6.2.

(c) Prove that your construction is secure relative to your definition under the assumption that $(\mathsf{Gen}_H, H)$ is collision resistant.

5.15 Prove that the commitment scheme discussed in Section 5.6.5 is secure in the random-oracle model.