example—in fact, the application originally considered by Hellman—is a key-recovery attack on an arbitrary block cipher F. Define $H(k) \overset{\text{def}}{=} F_k(m)$ where m is some arbitrary input that is used for building the table. If an attacker can subsequently obtain $F_k(m)$ for an unknown key k—either via a chosen-plaintext attack or by choosing m such that $F_k(m)$ 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.

6.5 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 is 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 existing approaches that have not been proven secure. (This is a major issue 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 entirely 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.

A 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 birthday attacks, 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 sometimes called the "standard model," although at this point the random-oracle model itself is considered quite standard in the literature.

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 fixed and its code is known. 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 6.5.2.

6.5.1 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 "black box" that takes a bit-string as input and returns a bit-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 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 view H as 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 ℓ), 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 Π 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* Π *and those of the adversary* $\mathcal{A}$. Assuming the honest parties who use Π 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 Π.

In the random-oracle model, in contrast, a scheme Π may rely on an oracle H. As before, Π 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 Π and those of the adversary $\mathcal{A}$. When using Π in the real world, some (instantiation of) H must be fixed. Unfortunately, security of Π 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 some fixed 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 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 of the random-oracle model are especially useful; we sketch them informally here, and show some simple applications of them in what follows, but caution that a full understanding will likely have to wait until we present formal proofs in the random-oracle model in later chapters.

A 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 of the random-oracle model; they are intended merely to provide a gentle introduction. In what follows, we assume a random oracle mapping ℓ_{in}-bit inputs to ℓ_{out}-bit outputs, where $\ell_{in}, \ell_{out} > n$, the security parameter (so ℓ_{in}, ℓ_{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 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.

As a proof sketch, let S denote the set of points on which $\mathcal{A}$ queries H; of course, $|S|$ is polynomial in n. Observe that in the second case, the probability that $x \in S$ is negligible—this is because $\mathcal{A}$ starts with no information about x (note that $H(x)$ by itself reveals nothing about x because H is a random function), and 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 $H(x) = H(x')$.