Chapter 13

Public-Key Cryptosystems in the Random Oracle Model

In the previous two chapters, we have seen constructions of digital signatures and public-key encryption schemes based on a variety of assumptions. For the most part, however, the provably-secure schemes we have discussed and analyzed are not particularly efficient. More to the point:

- In Section 10.4.3 we have seen an encryption scheme that can be proven secure based on the RSA assumption (cf. Theorem 10.17), but the efficiency of this scheme does not come close to the efficiency of the textbook RSA encryption scheme described in Section 10.4.1. In fact, no secure encryption scheme based on RSA with efficiency comparable to the textbook RSA encryption scheme is currently known.
- We have not shown any public-key encryption schemes secure against chosen-ciphertext attacks. Though efficient schemes based on certain assumptions are known (these are, however, beyond the scope of this book), there is no known scheme based on RSA that is even remotely practical.
- We saw only a single example of a digital signature scheme satisfying the desired level of security given in Definition 12.2. This construction, shown in Section 12.6.2, is not very practical. In fact, no secure signature scheme based on RSA with efficiency comparable to the textbook RSA signature scheme is known.

We stress that the above statements remain true even given an efficient pseudorandom function (that could be instantiated in practice using a block cipher such as DES or AES) and/or an efficient collision-resistant hash function (that could be instantiated using a cryptographic hash function such as SHA-1).

We conclude that for most "standard" cryptographic assumptions (such as the RSA, factoring, or DDH assumptions), there are few or no public-key cryptosystems that are both (1) efficient enough to be used in practice, yet (2) can be proven secure based on these assumptions. This state of affairs presents a challenge to cryptographers, who continue to work at improving the efficiency of existing solutions, proposing new assumptions, and showing limitations to the best possible efficiency that can be achieved. In the meanwhile, we are left in practice with the question of what schemes to use. While one might argue

that we should simply choose the "best" schemes that are currently available (according to whatever criteria one likes), the reality appears to be that people for the most part would rather use *nothing* than use an inefficient scheme. (Furthermore, in some cases existing solutions are not remotely practical even if one is willing to sacrifice some efficiency.) Something must give.

One possibility, of course, is simply to use an efficient but completely ad-hoc cryptosystem with no justification for its security other than, perhaps, the fact that the designers tried to attack the scheme and were unsuccessful. This flies in the face of everything we have said so far about the importance of the rigorous, modern approach to cryptography, and it should be clear that this is unacceptable! By using a scheme that merely seems (to us) "hard to break" we potentially leave ourselves open to an adversary who is more clever than us and who *can* break the scheme. A better alternative must be sought.

13.1 The Random Oracle Methodology

Another approach, which has been hugely successful in practice and 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. Though 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 posits the existence of a public, randomly-chosen function H that can be evaluated *only* by "querying" an oracle — which can be thought of as a "magic box" — that returns $H(x)$ when given input x. (We will discuss in the following section exactly how this is to be interpreted.) To differentiate things, the model we have been using until now (where no random oracle is present) is often called the 'standard model.'

It should be stressed that no one seriously claims that any such oracle exists (although there have been suggestions that a random oracle could be implemented in practice by a trusted party). Rather, the random oracle model provides a formal *methodology* that can be used to design and validate cryptographic schemes via the following two-step approach:

1. First, a scheme is designed and proven secure in the random oracle model; that is, we assume that the world contains a random oracle, and construct and analyze a scheme based on this assumption. 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 H in the scheme is *instantiated* with a cryptographic hash function $\hat{H}$ such as SHA-1 or MD5, modified appropriately. 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 simulating a random oracle, so that the security proof given in the first step will carry over to the real-world instantiation of the scheme. A difficulty is that there is currently no theoretical justification for this hope, and in fact there exist (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, as a practical matter it is not clear exactly what it means for a hash function to be "good" at simulating a random oracle, nor is it clear that, as stated, this is an achievable goal. For these reasons, a proof of security for a scheme in the random oracle model should be viewed as providing evidence that the scheme has no "inherent design flaws", but should *not* be taken as 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 13.1.2.

13.1.1 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 — both honest parties as well as adversaries — can interact with the box, where such interaction consists of entering 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.

It is guaranteed that the box 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 function H; i.e., we simply 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. Note that no one "knows" 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.

Security guarantees in the random oracle model are a bit different from the security guarantees we are familiar with from the rest of the book, as we

now discuss. Fix some cryptographic scheme Π that relies on an oracle; that is, the scheme requires the honest parties running Π to query the oracle at various points during their execution. A general definition of (computational) security for Π has the following form:

> *For any polynomial-time adversary $\mathcal{A}$, the probability that some event occurs when $\mathcal{A}$ interacts with Π is negligible.*

The difference is that in all the definitions we have seen thus far, the probability in question is taken over *the random choices made by the honest parties running* Π. (If $\mathcal{A}$ is randomized then the probability is taken over the random choices of $\mathcal{A}$ as well, but this is less important for the current discussion.) In the random oracle model, the probability in question is taken over these random choices *as well as the random choice of a function H for the oracle*. (This should become more clear from the examples given in the following section.) It is for this reason that we speak of H as a *random* oracle.

A brief digression is in order regarding what it means to choose a function (uniformly) at random. Any function H mapping n_1-bit inputs to n_2-bit outputs can be viewed as a table indicating for each possible input $x \in \{0,1\}^{n_1}$ the corresponding output value $H(x) \in \{0,1\}^{n_2}$. Using lexicographic order for the inputs, this means that any such function can be represented by a string of length $2^{n_1} \cdot n_2$ bits, and conversely that every string of this length can be viewed as a function mapping n_1-bit inputs to n_2-bit outputs. An immediate corollary is that there are exactly $U \stackrel{\text{def}}{=} 2^{n_2 \cdot 2^{n_1}}$ different functions having the specified input and output lengths. Picking a function H of this type uniformly at random means choosing H uniformly from among these U possibilities. In the random oracle model as we have been picturing it, this corresponds to initializing the oracle by choosing such an H and having the oracle answer according to H. Note that storing the string/table representing H in any physical device would require an *exponential* (in the input length) number of bits, so even for moderately-sized inputs this is not something we can hope to do in the real world.

An equivalent, but often more convenient, way to think about choosing a function H uniformly at random is to imagine generating random outputs for H "on-the-fly," as needed. Specifically, imagine that the function is defined by a table of pairs $\{(x_i, y_i)\}$ 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 y_i is returned. Otherwise, a random string $y \in \{0,1\}^{n_2}$ is chosen, the answer y is returned, and the oracle stores (x, y) in its table so the same output can be returned if the same input is ever queried again. While one could imagine carrying this out in the real world, this is further from our conception of "fixing" the function H once-and-for-all before beginning to run some cryptographic scheme. From the point of view of the parties interacting with the oracle, however, it is completely identical.

Returning to our discussion of security in the random oracle model, note that even if some scheme Π is proven secure in this model, security is not

guaranteed for any *particular* function H but is instead only guaranteed "on the average" over random choice of H. (This is exactly analogous to the fact that a scheme secure in the standard model is not guaranteed to be secure for any particular set of random choices made by the honest parties but only on average over these random choices.) This indicates one reason why it is difficult to argue that any concrete instantiation of the oracle H yields a real-world implementation of Π that is actually secure.

Simple Illustrations of the Random Oracle Model

At this point some examples may be helpful. The examples given here are rather simple, do not use the full power that the random oracle model affords, and do not really illustrate any of the *limitations* of the random oracle methodology; the intention of including these examples is merely to provide a gentle introduction to the use of the model.

In all that follows, we assume a random oracle mapping n_1-bit inputs to n_2-bit outputs where $n_1, n_2 \geq n$, the security parameter. (Technically speaking, n_1 and n_2 are functions of n.)

A random oracle as a one-way function. We first show that a random oracle acts like a one-way function. Note that we do not say that a random oracle *is* a one-way function, since (as discussed in the previous section) a random oracle is not a fixed function. Rather, what we claim is that any polynomial-time adversary $\mathcal{A}$ succeeds with only negligible probability in the following experiment:

1. A random function H is chosen.
2. A random input $x \in \{0,1\}^{n_1}$ is chosen, and $y := H(x)$ is computed.
3. $\mathcal{A}$ is given y, and succeeds if it outputs a value x' such that $H(x') = y$.

We now argue why this is true. Assume without loss of generality that the value x' that $\mathcal{A}$ outputs was previously queried by $\mathcal{A}$ to the oracle. Assume further than if $\mathcal{A}$ ever makes a query x_i with $H(x_i) = y$, then $\mathcal{A}$ succeeds. (This just means that $\mathcal{A}$ does not act stupidly and fail to output a correct answer once it knows one.) Then the success probability of $\mathcal{A}$ in the above experiment is exactly the same as the success probability of $\mathcal{A}$ in the following experiment (to see why, it helps to recall the discussion from the previous section regarding "on-the-fly" selection of a random function):

1. A random $x \in \{0,1\}^{n_1}$ is chosen, and a random value $y \in \{0,1\}^{n_2}$ is given to $\mathcal{A}$.
2. Each time $\mathcal{A}$ makes a query x_i to the random oracle, do:
   - If $x_i = x$, then $\mathcal{A}$ immediately succeeds.

- Otherwise, choose a random $y_i \in \{0,1\}^{n_2}$. If $y_i = y$ then $\mathcal{A}$ immediately succeeds; if not, return y_i to $\mathcal{A}$ as the answer to the query and continue the experiment.

Let ℓ be the number of queries $\mathcal{A}$ makes to the oracle, with $\ell = \mathsf{poly}(n)$ since $\mathcal{A}$ runs in polynomial time. Since y is completely independent of x, the probability that $\mathcal{A}$ succeeds by querying $x_i = x$ for some i is at most $\ell/2^{n_1}$. Furthermore, since the answer y_i is chosen at random when the query x_i is not equal to x, the probability that $\mathcal{A}$ succeeds because $y_i = y$ for some i is at most $\ell/2^{n_2}$. Since $n_1, n_2 \geq n$ the probability that $\mathcal{A}$ succeeds in the latter game is at most $2\ell/2^n = \mathsf{poly}(n)/2^n$, which is negligible.

A random oracle as a collision-resistant hash function. It is not much more difficult to see that a random oracle also acts like a collision-resistant hash function. That is, the success probability of any polynomial-time adversary $\mathcal{A}$ in the following game is negligible:

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

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 oracles queries of $\mathcal{A}$ be $x_1, \ldots, x_\ell$, with $\ell = \mathsf{poly}(n)$, 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 $i \neq j$. Viewing the choice of a random H as being done "on-the-fly", this is exactly equal to the probability that if we pick ℓ strings $y_1, \ldots, y_\ell \in \{0,1\}^{n_2}$ independently and uniformly at random, we have $y_i = y_j$ for some $i \neq j$. The problem has now been transformed into an example of the birthday problem; using the results of Section A.4 we see that $\mathcal{A}$ succeeds with probability $\mathcal{O}(\ell^2/2^{n_2})$, which is negligible.

Constructing a pseudorandom function from a random oracle. It is also rather easy to construct a pseudorandom function in the random oracle model (though the proof is not quite as trivial as in the examples above). Suppose $n_1 = 2n$ and $n_2 = n$. Then define

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

where $|k| = |x| = n$. We claim that this is a pseudorandom function; namely, for any polynomial-time $\mathcal{A}$ the success probability of $\mathcal{A}$ in the following experiment is at most negligibly greater than 1/2:

1. A random function H, a random $k \in \{0,1\}^n$, and a random bit b are chosen.
2. If $b = 0$, the adversary $\mathcal{A}$ is given access to an oracle for $F_k(\cdot)$. 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 $b = b'$.

We stress that $\mathcal{A}$ can access H in addition to the function oracle provided to it by the experiment in step 2. In Exercise 13.1 you are asked to show that the construction above indeed gives a pseudorandom function.

It is worth reflecting that the use of an "oracle" and a "random function" in step 2 (and, indeed, back in Chapter 3 when we first defined pseudorandom functions) is fundamentally *different* from the use of a random oracle/function in the random oracle model. In the context of pseudorandom functions, the oracle and random function are used *as a definitional tool* but need not exist for the primitive itself to be realized. In the random oracle model, in contrast, the random oracle is *used as part of the construction* and so something must take the place of the oracle for the construction to be realized.

An interesting aspect of all the above proofs is that they hold even for *computationally-unbounded* adversaries, as long as such adversaries are limited to only making polynomially-many queries to the oracle. This has no real-world counterpart, where (for example) any function can be inverted by an adversary running for an unlimited amount of time and, moreover, there is no oracle and hence no way to define what it means to "query an oracle/evaluate a hash function" polynomially-many times.

Advanced Proof Techniques in the Random Oracle Model

The preceding examples may not make clear that the random oracle model enables certain proof techniques that have no counterpart in the standard model. We sketch them here, but caution the reader that a full understanding will likely have to wait until later in this chapter when these techniques are used in the proofs of some concrete schemes.

A first distinctive feature of the random oracle model, used already in the previous section, is

> *If an adversary $\mathcal{A}$ has not explicitly queried the oracle on some point x, then the value of $H(x)$ is* completely random *(at least as far as $\mathcal{A}$ is concerned).*

This may seem superficially similar to the guarantee provided by a pseudorandom generator, but it is actually quite different. If G is a pseudorandom generator then $G(x)$ is pseudorandom *assuming x is chosen uniformly at random and is unknown to the observer.* If x is known then trivially $G(x)$ is too. For H a random oracle, however, $H(x)$ is (truly) random as long as the adversary has not queries x. This is true even if x is known. Furthermore, if x is not chosen uniformly at random, but is chosen with enough entropy to make guessing x difficult, then $G(x)$ might be easy to distinguish from random but $H(x)$ will not be.

Say we are trying to prove security of some scheme in the random oracle model. As in the rest of the book, we will often construct a *reduction* showing

how any adversary $\mathcal{A}$ breaking the security of the scheme (in the random oracle model) can be used to violate some cryptographic assumption.[1] As part of the reduction, the random oracle that $\mathcal{A}$ interacts with must be simulated as part of the reduction. That is: $\mathcal{A}$ will submit queries to and receive answers from what it believes to be the oracle, but what is actually the reduction itself. This turns out to give a lot of power. For starters:

> *As part of the reduction, we may choose values for the output of H "on-the-fly" (as long as these values are correctly distributed, i.e., uniformly random).*

This is sometimes called "programmability". Although this may not seem like programmability confers any advantage, it does, as perhaps illustrated best by the proof of Theorem 13.11 will illustrate. Another advantage deriving from the fact that the reduction gets to simulate the random oracle is

> *The reduction gets to "see" the queries that $\mathcal{A}$ makes to the random oracle.*

(Note that this does contradict the fact, mentioned earlier, that queries to the random oracle are supposed to be "private". While that is true in the formal model itself, here we are using $\mathcal{A}$ as a subroutine within a reduction.) This also turns out to be extremely useful, as the proofs of Theorems 13.2 and 13.6 will demonstrate.

13.1.2 Is the Random Oracle Methodology Sound?

With the mechanics of the random oracle model behind us, we can turn to more fundamental questions such as: *What do proofs of security in the random oracle model guarantee in the real world?*, and: *Are proofs in the random oracle model fundamentally different from proofs in the standard model?* We highlight at the outset that these questions do not currently have any definitive answers: there is currently much debate within the cryptographic community regarding the role played by the random oracle model, and an active area of research is to determine what, exactly, a proof of security in the random oracle model *does* guarantee in the real world. We can only hope to give a flavor of all sides of this discussion.

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 Π in the random oracle model implies anything about the security of Π in the real world (i.e., once the random oracle H has been instantiated with any particular hash function $\hat{H}$). These are more than just theoretical

[1] In contrast, the proofs in the previous section were information-theoretic and did not use reductions.

misgivings. A more basic issue is that *no* concrete hash function can ever act as a "true" random oracle. For example, in the random oracle model $H(x)$ is supposed to be "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? For starters, 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 doesn't 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 defined.

Limitations of the random oracle model become more clear once we examine the proof techniques introduced in Section 13.1.1. As an example, 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. But if we replace the random oracle by a particular hash function $\hat{H}$, this means that 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 in the previous paragraph, the notion of $\mathcal{A}$ performing distinct evaluations of $\hat{H}$ may not even be true and certainly cannot be formally defined.)

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. While for any given scheme Π one could, after analyzing Π, decide to assume that Π is secure when instantiated with SHA-1, it is much less reasonable to assume that SHA-1 can take the place of the random oracle in *any* 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 can be proven secure in the random oracle model, but is completely insecure when the random oracle is replaced by SHA-1. (See Exercise 13.2.)

It is worth emphasizing that an assumption of the form "SHA-1 acts like a random oracle" is significantly different from an assumption of the form "SHA-1 is collision-resistant" or "AES is a pseudorandom function." The problem lies partly with the fact that we do not have a satisfactory *definition* of what the first statement should mean (while we do have such definitions for the latter two statements). In particular, a random oracle is not the same thing as a pseudorandom function: the latter is a *keyed* function that can only be evaluated when the key is known, and is only "random-looking" when the key is *un*known. In contrast, a random oracle is an unkeyed function that can be evaluated by anyone, yet is supposed to remain "random-looking" in some ill-defined sense.

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, and proofs of security in the random oracle model are less desirable and less satisfying than proofs of security in the standard model. The division of the chapters in this book can be taken as an endorsement of this preference.

In support of the random oracle model. Given all the problems with the random oracle model, why do we cover the random oracle model at all? More to the point: why has the random oracle been so influential in the development of modern cryptography, and why does it continue to be so widely used? As we will see, the random oracle model currently 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 widely-deployed schemes having proofs of security in the random oracle model. In addition, proofs in the random oracle model are almost universally recognized as important for schemes being considered as standards. The random oracle model have increased the confidence we have in certain efficient schemes, and has played a major role in the increasing pervasiveness with which cryptographic algorithms are deployed.

The fundamental reason for the above is the belief (with which we concur) that

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

Though some might disagree with the above, we offer the following in support of this conviction:

- A proof of security for a given scheme in the random oracle model indicates that the design is "sound", in the sense that the only possible weaknesses in (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. Said differently, a proof in the random oracle model indicates that the only way to "break" the scheme is to "break" the hash function itself (in some way); thus, if the hash function is "good enough" we have some confidence in the security of the scheme itself. 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 real-world attacks on any "natural" schemes proven secure in the random oracle model* (we do not include here attacks on "contrived" schemes like that of Exercise 13.2). 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. Understanding exactly what such proofs guarantee in the real world remains, in our minds, one of the most important research questions facing cryptographers today.

Instantiating the Random Oracle

Multiple times already in this chapter, we have stated that the random oracle can be instantiated in practice using "an appropriate modification of a cryptographic hash function." In fact, things are complicated by a number of issues such as (to name two)

- Cryptographic hash functions almost all use a Merkle-Damgård construction (cf. Section 4.6.4), and can therefore be distinguished relatively easy from a random oracle taking variable-length inputs. (In contrast, there are no known attacks on such hash function when restricting to fixed-length inputs.)
- Frequently, it is necessary for the output of a random oracle to have a certain form; e.g., the oracle should output elements of $\mathbb{Z}_N^*$ rather than bit-strings. Cryptographic hash functions, of course, output bit-strings only. (When we need such an oracle for our proofs, we will simply assume that one exists.)

A detailed discussion of how these issues can be dealt with in practice is beyond the scope of this book; our aim is merely to alert the reader to the subtleties that can arise.

13.2 Public-Key Encryption in the Random Oracle Model

We show in this section various public-key encryption schemes in the random oracle model. We present these constructions based on the RSA problem, both for convenience as well as because these constructions are most frequently instantiated using RSA in practice. We remark, however, that they can all be instantiated using an *arbitrary* trapdoor permutation (see Section 10.7.1).

13.2.1 Security against Chosen-Plaintext Attacks

The secure public-key encryption scheme we have previously seen based on RSA (cf. Theorem 10.17) was both inefficient and difficult to prove secure (indeed, we offered no proof). In the random oracle model, things become significantly easier. As usual, GenRSA denotes a PPT algorithm that, on input 1^n, outputs a modulus N that is the product of two n-bit primes, along with integers e, d satisfying $ed = 1 \bmod \phi(N)$.

CONSTRUCTION 13.1

Let GenRSA be as in the text, and let $H : \{0,1\}^{2n} \to \{0,1\}^{\ell(n)}$ be a function for ℓ an arbitrary polynomial.

Key generation. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d), where N is of length $2n$ and elements of $\mathbb{Z}_N^*$ are represented by $2n$-bit strings. The public key is $\langle N, e\rangle$ and the private key is (N, d).

Encryption. To encrypt a message $m \in \{0,1\}^{\ell(n)}$ with respect to the public key $\langle N, e\rangle$, choose random $r \leftarrow \mathbb{Z}_N^*$ and output the ciphertext

$$\langle [r^e \bmod N],\ H(r) \oplus m \rangle .$$

Decryption. To decrypt ciphertext $\langle c_1, c_2\rangle$ using private key $\langle N, d\rangle$, compute $r := [c_1^d \bmod N]$ and then output the message $H(r) \oplus c_2$.

CPA-secure RSA encryption in the random oracle model.

We can argue intuitively that the scheme is CPA-secure in the random oracle model (under the RSA assumption) as follows: since r is chosen at random it is infeasible for an eavesdropping adversary to recover r from $c_1 = [r^e \bmod N]$. The adversary will therefore never query r to the random oracle, and so the value $H(r)$ is completely random from the point of view of the adversary. But then c_2 is just a "one-time pad"-like encryption of m using the random value $H(r)$, and so the adversary gets no information about m. This intuition is developed into a formal proof below.

The proof (as indicated by the intuition above) relies heavily on the fact that H is a random oracle, and does not work if H is replaced by, e.g., a pseudorandom generator G. The reason is that the RSA assumption says that an adversary cannot recover r from $[r^e \bmod N]$, *but says nothing about what partial information about r the adversary might recover.* For instance, it may be the case that the adversary *can* compute half the bits of r, and in this case we can no longer claim that $G(r)$ is pseudorandom (since pseudorandomness of $G(r)$ requires r to be completely random). However, when H is a random oracle it does not matter if partial information about r is leaked; $H(r)$ is random as long as r has not been explicitly queried to the oracle.

THEOREM 13.2 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, Construction 13.1 has indistinguishable encryptions under chosen-plaintext attacks.*

PROOF Let Π denote Construction 13.1. As usual, we prove that Π has indistinguishable encryptions in the presence of an eavesdropper; by Theorem 10.10 this implies that Π is CPA-secure.

Let $\mathcal{A}$ be a probabilistic polynomial-time adversary, and define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1].$$

For the reader's convenience, we describe the steps of experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$:

1. *A random function H is chosen.*
2. $\mathsf{GenRSA}(1^n)$ *is run to generate* (N, e, d). $\mathcal{A}$ *is given* $pk = \langle N, e\rangle$, *and may query* $H(\cdot)$. *Eventually,* $\mathcal{A}$ *outputs two messages* $m_0, m_1 \in \{0,1\}^{\ell(n)}$,
3. *A random bit* $b \leftarrow \{0,1\}$ *and a random* $r \leftarrow \mathbb{Z}^*_N$ *are chosen, and* $\mathcal{A}$ *is given the ciphertext* $\langle r^e \bmod N, H(r) \oplus m_b\rangle$. *The adversary may continue to query* $H(\cdot)$.
4. $\mathcal{A}$ *then outputs a bit* b'. *The output of the experiment is defined to be 1 if* $b' = b$, *and 0 otherwise.*

In an execution of experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$, let Query denote the event that, at any point during its execution, $\mathcal{A}$ queries r to the random oracle H. We also use Succ as shorthand for the event that $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$. Then

$$\begin{aligned}\Pr[\mathsf{Succ}] &= \Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Succ} \wedge \mathsf{Query}] \\ &\leq \Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Query}]\end{aligned}$$

where all probabilities are taken over the randomness used in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. We now show that $\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2}$ and that $\Pr[\mathsf{Query}]$ is negligible. The theorem follows.

CLAIM 13.3 *If H is modeled as a random oracle,* $\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2}$.

If $\Pr[\overline{\mathsf{Query}}] = 0$ then the claim is immediate. Otherwise, we have

$$\begin{aligned}\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] &= \Pr\left[\mathsf{Succ} \mid \overline{\mathsf{Query}}\right] \cdot \Pr[\overline{\mathsf{Query}}] \\ &\leq \Pr\left[\mathsf{Succ} \mid \overline{\mathsf{Query}}\right].\end{aligned}$$

Furthermore, $\Pr\left[\mathsf{Succ} \mid \overline{\mathsf{Query}}\right] = \frac{1}{2}$. This is an immediate consequence of what we said earlier: namely, that if $\mathcal{A}$ does not explicitly query r to the oracle then $H(r)$ is completely random from $\mathcal{A}$'s point of view, and so $\mathcal{A}$ has no information as to whether m_0 or m_1 was encrypted. (This is exactly as in the case of the one-time pad encryption scheme.) Therefore, the probability that $b' = b$ when Query does not occur is exactly $\frac{1}{2}$. The reader should convince him or herself that this intuition can be turned into a formal proof.

CLAIM 13.4 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, then* $\Pr[\mathsf{Query}]$ *is negligible.*

The intuition here is that if Query is not negligible then we can use $\mathcal{A}$ to solve the RSA problem with non-negligible probability as follows: given inputs N, e, and $c_1 \in \mathbb{Z}_N^*$, give to $\mathcal{A}$ the public key $\langle N, e\rangle$ and ciphertext $\langle c_1, c_2\rangle$, where $c_2 \leftarrow \{0,1\}^{\ell(n)}$ is chosen at random. Then *monitor all the queries that $\mathcal{A}$ makes to the random oracle*. (See the discussion in Section 13.1.1.) If Query occurs then one of $\mathcal{A}$'s queries r satisfies $r^e = c_1 \bmod N$, and so we can output r as the answer. We therefore solve the RSA problem with probability exactly $\Pr[\mathsf{Query}]$, which must be negligible because the RSA problem is hard relative to GenRSA.

Formally, consider the following algorithm $\mathcal{A}'$:

Algorithm $\mathcal{A}'(N, e, c_1)$

1. Choose random $k^* \leftarrow \{0,1\}^{\ell(n)}$. (We imagine that $\mathcal{A}'$ implicitly sets $H(r) = k^*$, where $r \stackrel{\text{def}}{=} [c_1^{1/e} \bmod N]$. Note, however, that $\mathcal{A}'$ does not know r.)
2. Run $\mathcal{A}$ on input the public key $pk = \langle N, e\rangle$. Store pairs of strings $(\cdot, \cdot)$ in a table, initially empty. When $\mathcal{A}$ makes random oracle query $H(x)$, answer it as follows:
   - If there is an entry (x, k) in the table, return k.
   - If $x^e = c_1 \bmod N$, return k^* and store (x, k^*) in the table. (Note that in this case we have $x = r$, for r defined as above.)
   - Otherwise, choose a random $k \leftarrow \{0,1\}^{\ell(n)}$, return k to $\mathcal{A}$, and store (x, k) in the table.
3. At some point, $\mathcal{A}$ outputs messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$.
4. Choose random $b \leftarrow \{0,1\}$ and set $c_2 := k^* \oplus m_b$. Give $\mathcal{A}$ the ciphertext $\langle c_1, c_2\rangle$.
5. At the end of $\mathcal{A}$'s execution (after it has output its guess b'), let $x_1, \ldots, x_p$ be the list of all oracle queries made by $\mathcal{A}$. If there exists an i for which $x_i^e = c_1 \bmod N$, output x_i.

It is immediate that $\mathcal{A}'$ runs in polynomial time. Say the input to $\mathcal{A}'$ is generated by running $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d) and then choosing $c_1 \leftarrow \mathbb{Z}_N^*$ at random. (See Definition 7.46.) Then the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$. (In each case $\langle N, e\rangle$ is generated the same way; c_1 is equal to $[r^e \bmod N]$ for a randomly-chosen $r \leftarrow \mathbb{Z}_N^*$; and the random oracle queries of $\mathcal{A}$ are answered with random strings.) Thus, the probability of event Query remains unchanged. Furthermore, $\mathcal{A}'$ correctly solves the given RSA instance whenever Query occurs. That is,

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n) = 1] = \Pr[\mathsf{Query}].$$

Since the RSA problem is hard relative to GenRSA, it must be the case that $\Pr[\mathsf{Query}]$ is negligible. This concludes the proof of the claim, and hence the proof of the theorem. ■

13.2.2 Security Against Chosen-Ciphertext Attacks

We have not yet seen any examples of public-key encryption schemes secure against chosen-ciphertext attacks. Although such schemes exist, they are somewhat complex. Moreover, no *practical* schemes are known that can be based on, e.g., the RSA or factoring assumptions. (There are, however, practical CCA-secure public-key encryption schemes based on the DDH assumption.) Once again, the situation becomes much simpler in the random oracle model, and we show a construction based on the RSA assumption here.

Let GenRSA be as in the previous section, and let $\Pi' = (\mathsf{Enc}', \mathsf{Dec}')$ be a *private-key* encryption scheme for messages of length $\ell(n)$ whose keys are (without loss of generality) of length n. Looking ahead, we will want Π' to be secure against chosen-ciphertext attacks; as shown in Section 3.7, efficient private-key encryption schemes satisfying this notion can be constructed relatively easily.

CONSTRUCTION 13.5

Let Π' be a private-key encryption scheme as described in the text, and GenRSA be as in the previous section. $H : \{0,1\}^{2n} \to \{0,1\}^n$ is a function.

Key generation. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.

Encryption. To encrypt a message $m \in \{0,1\}^{\ell(n)}$ with respect to the public key $\langle N, e\rangle$, first choose random $r \leftarrow \mathbb{Z}_N^*$ and compute $k := H(r)$. Output the ciphertext

$$\left\langle [r^e \bmod N],\ \mathsf{Enc}'_k(m)\right\rangle.$$

Decryption. To decrypt ciphertext $\langle c_1, c_2\rangle$ using private key $\langle N, d\rangle$, first compute $r := [c_1^d \bmod N]$ and set $k := H(r)$. Output $\mathsf{Dec}'_k(c_2)$.

CCA-secure RSA encryption in the random oracle model.

It is instructive to compare the above with Construction 13.1. In both cases, the sender chooses a random $r \leftarrow \mathbb{Z}_N^*$ and sends $c_1 = [r^e \bmod N]$ as part of the ciphertext. The difference between the two schemes, on a conceptual level, is that in Construction 13.1 the sender encrypts the message m using the key $H(r)$ and a private-key encryption scheme that has *indistinguishable encryptions in the presence of an eavesdropper*. (Construction 13.1 utilizes the

one-time pad encryption scheme since, letting $k := H(r)$ there, the second component of the ciphertext is computed as $k \oplus m$. We remark that an analogous proof can be given when Construction 13.1 is instantiated with an arbitrary CPA-secure private-key encryption scheme; see Exercise 13.3.) Here, in contrast, the sender encrypts the message m using the key $k = H(r)$ and a private-key encryption scheme that has *indistinguishable encryptions under a chosen-ciphertext attack*.

The intuition for the proof of security, when H is modeled as a random oracle, is roughly as in the previous section. In the proof, we will again distinguish between the case when the adversary does not query r to the random oracle H and the case when it does. In the first case, the adversary learns nothing about the key $k = H(r)$ and so we have reduced the security of our construction to the security of the private-key encryption scheme Π'. We then argue that the second case occurs with only negligible probability if the RSA problem is hard relative to GenRSA. The proof of this fact is a bit more complex than in the previous section because we must now show how it is possible to simulate decryption oracle queries without knowing the private key. We show how to do this by "programming" the random oracle in an appropriate way.

THEOREM 13.6 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, Construction 13.5 has indistinguishable encryptions under a chosen-ciphertext attack.*

PROOF Let Π denote Construction 13.5, and let $\mathcal{A}$ be a probabilistic polynomial-time adversary. Define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n) = 1].$$

For the reader's convenience, we describe the steps of experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$:

1. *A random function H is chosen.*
2. $\mathsf{GenRSA}(1^n)$ *is run to obtain (N, e, d). $\mathcal{A}$ is given $pk = \langle N, e\rangle$ and may query $H(\cdot)$ and the decryption oracle $\mathsf{Dec}_{\langle N,d\rangle}(\cdot)$. Eventually $\mathcal{A}$ outputs two messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$,*
3. *Random $b \leftarrow \{0,1\}$ and $r \leftarrow \mathbb{Z}^*_N$ are chosen, and $\mathcal{A}$ is given the ciphertext $\langle r^e \bmod N, \mathsf{Enc}'_{H(r)}(m_b)\rangle$. Adversary $\mathcal{A}$ may continue to query $H(\cdot)$ and the decryption oracle, though it may not query the latter on the ciphertext it was given.*
4. *$\mathcal{A}$ then outputs a bit b'. The output of the experiment is defined to be 1 if $b' = b$, and 0 otherwise.*

In an execution of experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$, let Query denote the event that, at any point during its execution, $\mathcal{A}$ queries r to the random oracle H. We

also use Succ as shorthand for the event that $b' = b$. Then

$$\begin{aligned}\Pr[\mathsf{Succ}] &= \Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Succ} \wedge \mathsf{Query}] \\ &\leq \Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] + \Pr[\mathsf{Query}],\end{aligned}$$

where all probabilities are taken over the randomness used in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$. We now show that there exists a negligible function negl such that

$$\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2} + \mathsf{negl}(n),$$

and that $\Pr[\mathsf{Query}]$ is negligible. The theorem follows.

CLAIM 13.7 *If private-key encryption scheme Π' has indistinguishable encryptions under a chosen-ciphertext attack and H is modeled as a random oracle, then there exists a negligible function negl such that*

$$\Pr\left[\mathsf{Succ} \wedge \overline{\mathsf{Query}}\right] \leq \frac{1}{2} + \mathsf{negl}(n).$$

The proof now is much more involved than the proof of the corresponding claim in the previous section. This is because, as discussed in the intuition preceding this theorem, Construction 13.1 uses the (perfectly-secret) one-time pad encryption scheme as its "private-key component", whereas Construction 13.5 uses a computationally-secure private-key encryption scheme Π'.

Consider the following adversary $\mathcal{A}'$ carrying out a chosen-ciphertext attack on Π' (cf. Definition 3.31):

Adversary $\mathcal{A}'(1^n)$

$\mathcal{A}'$ has access to an encryption oracle $\mathsf{Enc}'_k(\cdot)$ and a decryption oracle $\mathsf{Dec}'_k(\cdot)$ for some key k

1. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). Choose $r \leftarrow \mathbb{Z}^*_N$ and set $c_1 := [r^e \bmod N]$.
   / We will imagine that $\mathcal{A}'$ implicitly sets $H(r) = k$, though $\mathcal{A}'$ does not know k. */*
2. Run $\mathcal{A}$ on input $pk := \langle N, e\rangle$. Pairs of strings $(\cdot, \cdot)$ are stored in a table, initially empty. When $\mathcal{A}$ makes a query $\langle \bar{c}_1, \bar{c}_2\rangle$ to its decryption oracle, answer it as follows:
   - If $\bar{c}_1 = c_1$, then $\mathcal{A}'$ queries $\bar{c}_2$ to its own decryption oracle and returns the result to $\mathcal{A}$.
   - If $\bar{c}_1 \neq c_1$, compute $\bar{r} := [\bar{c}_1^d \bmod N]$. Then compute $\bar{k} := H(\bar{r})$ using the procedure discussed below. Return the result $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$ to $\mathcal{A}$.

   When the value $H(\bar{r})$ is needed, either in response to a query by $\mathcal{A}$ to the random oracle or in the course of answering a query by $\mathcal{A}$ to its decryption oracle, compute $H(\bar{r})$ as follows:

- If there is an entry $(\bar{r}, \bar{k})$ in the table, return $\bar{k}$.
- Otherwise, choose a random $\bar{k} \leftarrow \{0,1\}^n$ return it, and store $(\bar{r}, \bar{k})$ in the table.

3. At some point, $\mathcal{A}$ outputs $m_0, m_1 \in \{0,1\}^{\ell(n)}$. Adversary $\mathcal{A}'$ outputs these same messages, and is given in return a ciphertext c_2. Then $\mathcal{A}'$ gives the ciphertext $\langle c_1, c_2 \rangle$ to $\mathcal{A}$, and continues to answer the oracle queries of $\mathcal{A}$ as before.
4. When $\mathcal{A}$ outputs its guess b', this value is output by $\mathcal{A}'$.

It is immediate that $\mathcal{A}'$ runs in polynomial time. Furthermore, $\mathcal{A}'$ never submits the ciphertext c_2 to its own decryption oracle after it is given this ciphertext in step 3; the only way this could happen would be if $\mathcal{A}$ submitted $\langle c_1, c_2 \rangle$ to *its* decryption oracle, but this is not allowed.

Let $\Pr'[\cdot]$ refer to the probability of an event in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A}',\Pi'}(n)$. Define Succ and Query as above; that is, Succ is the event that $b' = b$, and Query is the event that $\mathcal{A}$ queries r to the random oracle. The key observation is that the view of $\mathcal{A}'$ when run as a subroutine by $\mathcal{A}$ (in experiment $\mathsf{PrivK}^{\mathsf{cca}}_{\mathcal{A}',\Pi'}(n)$) is distributed identically to the view of $\mathcal{A}'$ in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$, until event Query occurs. So

$$\Pr'[\mathsf{Succ}] \geq \Pr'[\mathsf{Succ} \wedge \mathsf{Query}] = \Pr[\mathsf{Succ} \wedge \mathsf{Query}].$$

(The inequality is trivial, and the equality follows from the observation we just made.) Because Π' is CCA-secure, there exists a negligible function negl such that

$$\frac{1}{2} + \mathsf{negl}(n) \geq \Pr'[\mathsf{Succ}] \geq \Pr[\mathsf{Succ} \wedge \mathsf{Query}],$$

completing the proof of the claim.

CLAIM 13.8 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, then* $\Pr[\mathsf{Query}]$ *is negligible.*

Intuitively, $\Pr[\mathsf{Query}]$ is negligible for the same reason as in the proof of Theorem 13.2. In the formal proof, however, additional difficulties arise due to the fact that the decryption queries of $\mathcal{A}$ must somehow be answered *without knowledge of the private (decryption) key.* Fortunately, the random oracle model enables a solution: to decrypt a ciphertext $\langle \bar{c}_1, \bar{c}_2 \rangle$ (where no prior decryption query was made using the same initial component $\bar{c}_1$), we generate a random key $\bar{k}$ and return the message $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$. We then *implicitly* set $H(\bar{r}) = \bar{k}$, where $\bar{r} \stackrel{\text{def}}{=} [\bar{c}_1^{1/e} \bmod N]$. (Note that $\bar{r}$ may be unknown at this time, and we do not know how to compute it without the factorization of N.)

The only "catch" is that we must ensure consistency with both prior and later queries of $\mathcal{A}$ to the random oracle. But this is relatively simple:

- When decrypting, we first check for any prior random oracle query $H(\bar{r})$ such that $\bar{c}_1 = r^e \bmod N$ (and, if so, use for k the value previously returned in response to the previous random oracle query $H(\bar{r})$).
- When answering a random oracle query $H(\bar{r})$, we compute $\bar{c}_1 := [\bar{r}^e \bmod N]$ and check whether any previous decryption query used $\bar{c}_1$ as the first component of the ciphertext (and, if so, return the value $\bar{k}$ that was previously used to answer this prior decryption oracle query).

Actually, a simple data structure handles both cases: maintain a table storing all the random oracle queries and answers as in the proof of Theorem 13.2 (and as in the proof of the previous claim), except that now the table will contain *triples* rather than pairs. Two types of entries will appear in the table:

- The first type of entry has the form $(\hat{r}, \hat{c}_1, \hat{k})$ with $\hat{c}_1 = [\hat{r}^e \bmod N]$. This entry means that we have defined $H(\hat{r}) = \hat{k}$.
- The second type of entry has the form $(\star, \hat{c}_1, \hat{k})$, which means that the value $\hat{r} \stackrel{\text{def}}{=} [\hat{c}_1^{1/e} \bmod N]$ is not yet known. (Again, we are not able to compute this value without the factorization of N.) An entry of this sort indicates that we are implicitly setting $H(\bar{r}) = \hat{k}$; in particular, when answering a decryption oracle query $\langle \bar{c}_1, \bar{c}_2 \rangle$ by $\mathcal{A}$, we return $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$. If $\mathcal{A}$ ever asks the random oracle query $H(\bar{r})$ we will return the correct answer $\hat{k}$ because we will check the table for any entry having $[\bar{r}^e \bmod N]$ as its second component.

We implement the above ideas as the following algorithm $\mathcal{A}'$:

Algorithm $\mathcal{A}'(N, e, c_1)$

1. Choose random $k \in \{0,1\}^n$. Triples $(\cdot, \cdot, \cdot)$ are stored in a table that initially contains only $(\star, c_1, k)$.
2. Run $\mathcal{A}$ on input $pk := \langle N, e \rangle$. When $\mathcal{A}$ makes a query $\langle \bar{c}_1, \bar{c}_2 \rangle$ to the decryption oracle, answer it as follows:
   - If there is an entry in the table whose second component is $\bar{c}_1$, let $\bar{k}$ be the third component of this entry. (That is, the entry is either of the form $(\bar{r}, \bar{c}_1, \bar{k})$ with $\bar{r}^e = \bar{c}_1 \bmod N$, or of the form $(\star, \bar{c}_1, \bar{k})$.) Return $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$.
   - Otherwise, choose a random $\bar{k} \leftarrow \{0,1\}^n$, return $\mathsf{Dec}'_{\bar{k}}(\bar{c}_2)$, and store $(\star, \bar{c}_1, \bar{k})$ in the table.

   When $\mathcal{A}$ makes a query $\hat{r}$ to the random oracle, compute $\bar{c}_1 := [\bar{r}^e \bmod N]$ and answer the query as follows:
   - If there is an entry of the form $(\bar{r}, \bar{c}_1, \bar{k})$ in the table, return $\bar{k}$.

- If there is an entry of the form $(\star, \bar{c}_1, \bar{k})$ in the table, return $\bar{k}$ and store $(\bar{r}, \bar{c}_1, \bar{k})$ in the table.
- Otherwise, choose random $\bar{k} \leftarrow \{0,1\}^n$, return $\bar{k}$, and store $(\bar{r}, \bar{c}_1, \bar{k})$ in the table.

3. At some point, $\mathcal{A}$ outputs messages $m_0, m_1 \in \{0,1\}^{\ell(n)}$. Choose a random bit $b \leftarrow \{0,1\}$ and set $c_2 := \mathsf{Enc}'_k(m_b)$. Give to $\mathcal{A}$ the ciphertext $\langle c_1, c_2 \rangle$, and continue to answer the oracle queries of $\mathcal{A}$ as before.
4. At the end of $\mathcal{A}$'s execution, if there is an entry in the table of the form (r, c_1, k) then output r.

Algorithm $\mathcal{A}'$ exactly carries out the strategy outlined earlier, with the only addition being that a random key k is chosen at the beginning of the experiment and $\mathcal{A}'$ implicitly sets $H([c_1^{1/e} \bmod N]) = k$.

It is immediate that $\mathcal{A}'$ runs in polynomial time. Say the input to $\mathcal{A}'$ is generated by running $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d) and then choosing $c_1 \leftarrow \mathbb{Z}_N^*$ at random from $\mathbb{Z}_N^*$. (See Definition 7.46.) Then the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n)$. Thus, the probability of event Query remains unchanged. Furthermore, $\mathcal{A}'$ correctly solves the given RSA instance whenever Query occurs. That is,

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n) = 1] = \Pr[\mathsf{Query}].$$

Since the RSA problem is hard relative to GenRSA, it must be the case that $\Pr[\mathsf{Query}]$ is negligible. This concludes the proof of the claim, and hence the proof of the theorem. ■

13.2.3 OAEP

The public-key encryption scheme given in Section 13.2.2 offers a fairly efficient way to achieve security against chosen-ciphertext attacks based on the RSA assumption, in the random oracle model. (Moreover, as we noted earlier, the general paradigm shown there can be instantiated using any trapdoor permutation and so can be used to construct a scheme with similar security based on the hardness of factoring.) For certain applications, though, even more efficient schemes are desirable. The main drawback of the previous scheme is that the ciphertext expansion is relatively significant when very short messages are encrypted.

The *optimal asymmetric encryption padding* (OAEP) technique eliminates this drawback, and the ciphertext includes only a single element of $\mathbb{Z}_N^*$ when short messages are encrypted. (To encrypt longer messages, hybrid encryption would be used as discussed in Section 10.3.) Technically, OAEP is a padding method and not an encryption scheme, though encryption schemes that use

this padding are often simply called OAEP themselves. We denote by RSA-OAEP the combination of OAEP padding with textbook RSA encryption (as will become clear from the discussion below and Construction 13.9).

OAEP is a reversible, randomized method for encoding a plaintext message m of length[2] $n/2$ as a string $\hat{m}$ of length $2n$. OAEP uses two functions G and H that are modeled as independent random oracles in the analysis. Though the existence of more than one random oracle was not discussed when we introduced the random oracle model in Section 13.1.1, this is interpreted in the natural way. In fact it is quite easy to use a single random oracle $\bar{H}$ to implement two independent random oracles G, H by setting $G(x) \stackrel{\text{def}}{=} \bar{H}(0x)$ and $H(x) \stackrel{\text{def}}{=} \bar{H}(1x)$.

In RSA-OAEP, encryption of a message m relative to a public key $\langle N, e\rangle$ (with $\|N\| > 2n$) is done by encoding m as $\hat{m}$ and then computing the ciphertext $c := [\hat{m}^d \bmod N]$. To decrypt, the receiver recovers $\hat{m}$ using its private key, and then reverses the encoding to recover the message m. A full description is given as Construction 13.9.

CONSTRUCTION 13.9

Let GenRSA be as in the previous sections, and let $G : \{0,1\}^n \to \{0,1\}^n$ and $H : \{0,1\}^n \to \{0,1\}^n$ be functions.

Key generation. On input 1^n, run $\mathsf{GenRSA}(1^{n+1})$ to obtain (N, e, d) with $\|N\| > 2n$.[a] The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.

Encryption. To encrypt message $m \in \{0,1\}^{n/2}$ with respect to the public key $\langle N, e\rangle$, first choose random $r \leftarrow \{0,1\}^n$. Set $m' := m \| 0^{n/2}$, compute $\hat{m}_1 := G(r) \oplus m'$, and then set

$$\hat{m} := \hat{m}_1 \,\|\, (r \oplus H(\hat{m}_1))$$

and interpret $\hat{m}$ as an element of $\mathbb{Z}_N^*$ in the natural way. Output the ciphertext $c := [\hat{m}^e \bmod N]$.

Decryption. To decrypt ciphertext c using private key $\langle N, d\rangle$, first compute $\hat{m} := [c^d \bmod N]$ and parse $\hat{m}$ as $\hat{m}_1 \| \hat{m}_2$ with $|\hat{m}_1| = |\hat{m}_2| = n$. Next compute $r := H(\hat{m}_1) \oplus \hat{m}_2$, followed by $m' := \hat{m}_1 \oplus G(r)$. If the final $n/2$ bits of m' are not $0^{n/2}$, output $\perp$. Otherwise, output the first $n/2$ bits of m'.

[a] This explains our unusual choice to run GenRSA with input 1^{n+1} rather than input 1^n.

The RSA-OAEP encryption scheme.

[2] This matches Construction 13.9, but can be generalized for other message/encoding lengths.

The above is actually somewhat of a simplification, in that certain details are omitted and other choices of the parameters are possible. The reader interested in implementing RSA-OAEP is referred to the references given in the notes at the end of this chapter. A proof of security for the above construction is beyond the scope of the book; we only mention that if the RSA problem is hard relative to GenRSA and G and H are modeled as independent random oracles, then RSA-OAEP can be proven secure for certain types of public exponents e (including the common case when $e = 3$). Variants of OAEP suitable for use with arbitrary public exponents or, more generally, with other trapdoor permutations, are also known; see the references at the end of this chapter.

13.3 RSA Signatures in the Random Oracle Model

Having completed our discussion of public-key encryption in the random oracle model, we now turn our attention to constructions of digital signatures. The *full-domain hash* (FDH) signature scheme is perhaps the simplest to analyze. Though this, too, may be instantiated with any trapdoor permutation, we once again describe the RSA-FDH scheme which is based on RSA.

CONSTRUCTION 13.10

Let GenRSA be as in the previous sections, and let $H : \{0,1\}^* \to \{0,1\}^{2n}$ be a function.

Key generation. Run $\mathsf{GenRSA}(1^n)$ to compute (N, e, d). The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.

Signing. To sin message $m \in \{0,1\}^*$ using the secret key $\langle N, d\rangle$, compute

$$\sigma := [H(m)^d \bmod N].$$

Verification. Given a signature σ on a message m with respect to the public key $\langle N, e\rangle$, output 1 iff $\sigma^e \stackrel{?}{=} H(m) \bmod N$.

The RSA-FDH signature scheme.

We have actually seen the RSA-FDH scheme previously in Section 12.3.2, where it was called *hashed RSA*. Hashed RSA was obtained by applying the textbook RSA signature scheme to a *hash* of the message, rather than the message itself. To review: in the textbook RSA signature scheme, a message $m \in \mathbb{Z}_N^*$ was signed by computing $\sigma := m^d \bmod N$. (As usual, the private key is $\langle N, d\rangle$ where (N, e, d) were output by some algorithm GenRSA.) Textbook

RSA is completely insecure, and in particular was shown in Section 12.3.1 to be vulnerable to the following attacks:

- An adversary can choose arbitrary σ, compute $m := [\sigma^e \bmod N]$, and output (m, σ) as a forgery.
- Given (legitimately-generated) signatures σ_1 and σ_2 on messages m_1 and m_2, respectively, it is possible to compute a valid signature $\sigma := [\sigma_1 \cdot \sigma_2 \bmod N]$ on the message $m := [m_1 \cdot m_2 \bmod N]$.

In RSA-FDH (i.e., hashed RSA), the signer *hashes* m before signing it; that is, a signature on a message m is computed as $\sigma := [H(m)^d \bmod N]$. (See Construction 13.10.) In Section 12.3.2 we argued informally why this modification prevents the above attacks; we can now see why the attacks do not apply when H is modeled as a random oracle.

- Given σ, it is hard to find an m such that $H(m) = [\sigma^e \bmod N]$. (See, e.g., the discussion in Section 13.1.1 regarding why a random oracle acts like a one-way function.)
- If σ_1 and σ_2 are signatures on messages m_1 and m_2, respectively, this means that $H(m_1) = \sigma_1^e \bmod N$ and $H(m_2) = \sigma_2^e \bmod N$. It is not likely, however, that $\sigma = [\sigma_1 \cdot \sigma_2 \bmod N]$ is a valid signature on $m = [m_1 \cdot m_2 \bmod N]$ since there is no reason to believe that $H(m_1 \cdot m_2) = H(m_1) \cdot H(m_2) \bmod N$. (And if H is a random oracle, the latter will happen with only negligible probability.)

We stress that the above merely serves as intuition, while in fact RSA-FDH is *provably* resistent to the above attacks as a consequence of Theorem 13.11 that we will prove below. We stress also that the above informal arguments can only be proven when H is modeled as a random oracle; we do not know how to prove anything like the above if H is "only" collision-resistant, say.

On the face of it, RSA-FDH may seem like an exact instantiation of the "hash-and-sign" paradigm (Section 12.4) using the textbook RSA signature scheme. The crucial difference is that in Section 12.4 we showed that the "hash-and-sign" paradigm converts a signature scheme Π that handles "short" messages into a signature scheme Π' that handles messages of arbitrary length, when the hash function being used is *collision resistant* and the original signature scheme Π is *existentially unforgeable under an adaptive chosen-message attack*. In contrast, here we are converting a *completely insecure scheme* Π (namely, textbook RSA signatures) into a *secure* scheme Π', but only by modeling the hash function as a *random oracle*.

We prove below that RSA-FDH is existentially unforgeable under an adaptive chosen-message attack, under the RSA assumption in the random oracle model. Toward intuition for this result, first consider the case of existential unforgeability under a *no-message attack*; i.e., when the adversary cannot request any signatures. Here the adversary is limited only to making queries to

the random oracle, and we can assume without loss of generality that if the adversary outputs a purported forgery (m, σ) then the adversary had at some point previously queried $H(m)$. Letting $y_1, \ldots, y_q$ denote the answers that the adversary received in response to its q queries to the random oracle, we see that each y_i is completely random; furthermore, forging a valid signature on some message requires computing an eth root of one of these values. It is thus not hard to see that, under the RSA assumption, the adversary outputs a valid forgery with only negligible probability.

Formally, starting with an adversary $\mathcal{A}$ forging a valid signature in a no-message attack we construct an algorithm $\mathcal{A}'$ solving the RSA problem. Given input (N, e, y), algorithm $\mathcal{A}'$ first runs $\mathcal{A}$ on the public key $pk = \langle N, e \rangle$. It answers the random oracle queries of $\mathcal{A}$ with random elements of $\mathbb{Z}_N^*$ except for one query (chosen at random from among the q random oracle queries of $\mathcal{A}$) that is answered with y. Say $\mathcal{A}$ outputs (m, σ) with $\sigma^e = H(m) \bmod N$ (i.e., $\mathcal{A}$ outputs a forgery). If the input to $\mathcal{A}'$ was generated by (in particular) choosing y at random from $\mathbb{Z}_N^*$, then the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is identically distributed to the view of $\mathcal{A}$ when it attacks the signature scheme; furthermore, $\mathcal{A}$ has no information regarding which oracle query was answered with y. So with probability $1/q$ it will be the case that the query $H(m)$ was the one that was answered with y, in which case $\mathcal{A}'$ solves the given instance of the RSA problem by outputting $\sigma = y^{1/e} \bmod N$. We see that if $\mathcal{A}$ succeeds with probability ε, then $\mathcal{A}'$ solves the RSA problem with probability ε/q, and hence ε must be negligible if the RSA assumption holds.

Handling the case when the adversary is allowed to request signatures on messages of its choice is more difficult. The complication arises since our reduction $\mathcal{A}'$ above does not, of course, know the decryption exponent d, yet now has to compute valid signatures on messages chosen by $\mathcal{A}$. This seems impossible (and possibly even contradictory!) until we realize that $\mathcal{A}'$ *can* correctly compute a signature on a message m as long as it sets $H(m)$ equal to $[\sigma^e \bmod N]$ for some known value σ. Note that if σ is chosen uniformly at random then $[\sigma^e \bmod N]$ is uniformly distributed as well, and so the random oracle is still emulated "properly" by $\mathcal{A}'$.

The above intuition forms the basis for the proof of the following:

THEOREM 13.11 *If the RSA problem is hard relative to* GenRSA *and H is modeled as a random oracle, Construction 13.10 is existentially unforgeable under an adaptive chosen-message attack.*

PROOF Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ denote Construction 13.10, and let $\mathcal{A}$ be a probabilistic polynomial-time adversary. Define

$$\varepsilon(n) \stackrel{\text{def}}{=} \Pr[\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n) = 1].$$

For the reader's convenience, we describe the steps of experiment $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n)$:

1. *A random function H is chosen.*
2. $\mathsf{GenRSA}(1^n)$ *is run to generate* (N, e, d)*. The adversary* $\mathcal{A}$ *is given* $pk = \langle N, e\rangle$*, and may query* $H(\cdot)$ *and the signing oracle* $\mathsf{Sign}_{\langle N,d\rangle}(\cdot)$*. When* $\mathcal{A}$ *requests a signature on a message* m*, it is given* $\sigma := [H(m)^d \bmod N]$ *in return.*
3. *Eventually,* $\mathcal{A}$ *outputs a pair* (m, σ) *where* $\mathcal{A}$ *had not previously requested a signature on* m*. The output of the experiment is 1 if* $\sigma^e = H(m)$*, and 0 otherwise.*

Since we have already discussed the intuition above, we jump right into the formal proof. To simplify matters, we assume without loss of generality that (1) $\mathcal{A}$ never makes the same random oracle query twice; (2) if $\mathcal{A}$ requests a signature on a message m then it had previously queried $H(m)$; and (3) if $\mathcal{A}$ outputs (m, σ) then it had previously queried $H(m)$.

Let $q = q(n)$ be a (polynomial) upper-bound on the number of random oracle queries made by $\mathcal{A}$. Consider the following algorithm $\mathcal{A}'$:

Algorithm $\mathcal{A}'(N, e, y^*)$

1. Choose $j \leftarrow \{1, \ldots, q\}$.
2. Run $\mathcal{A}$ on input the public key $pk = \langle N, e\rangle$. Store triples $(\cdot, \cdot, \cdot)$ in a table, initially empty. An entry (m_i, σ_i, y_i) indicates that $\mathcal{A}'$ has set $H(m_i) = y_i$, and furthermore it will be the case that $\sigma_i^e = y_i \bmod N$. When $\mathcal{A}$ makes its ith random oracle query $H(m_i)$, answer it as follows:
   - If $i = j$, return y^*.
   - Otherwise, choose random $\sigma_i \leftarrow \mathbb{Z}_N^*$, compute $y_i := [\sigma_i^e \bmod N]$, return y_i as the answer to the query, and store (m_i, σ_i, y_i) in the table.

   When $\mathcal{A}$ requests a signature on message m, let i be such that[3] $m = m_i$ and answer the query as follows:
   - If $i \neq j$ then there is an entry (m_i, σ_i, y_i) in the table. Return σ_i.
   - If $i = j$ then abort the experiment.
3. At the end of $\mathcal{A}$'s execution, it outputs (m, σ). If $m = m_j$ and $\sigma^e = y^* \bmod N$, then output σ.

It is immediate that $\mathcal{A}'$ runs in polynomial time. Say the input to $\mathcal{A}'$ is generated by running $\mathsf{GenRSA}(1^n)$ to obtain (N, e, d) and then choosing $y^* \leftarrow$

[3]Here m_i denotes the ith query made to the random oracle. Recall our assumption that if $\mathcal{A}$ requests a signature on a message, then it had previously queried the random oracle on that message.

$\mathbb{Z}_N^*$ uniformly at random. The index j chosen by $\mathcal{A}'$ in the first step represents a guess as to which oracle query of $\mathcal{A}$ will correspond to the eventual output of $\mathcal{A}$. When this guess is correct, the view of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ in experiment $\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n)$ is distributed identically to the view of $\mathcal{A}$ in experiment $\mathsf{Sig\text{-}forge}^{\mathsf{cma}}_{\mathcal{A},\Pi}(n)$. This is, in part, because each of the q random oracle queries of $\mathcal{A}$ when run as a subroutine by $\mathcal{A}'$ is indeed answered with a random value:

- The query $H(m_j)$ is answered with y^*, a value chosen at random from $\mathbb{Z}_N^*$.
- Queries $H(m_i)$ with $i \neq j$ are answered with $y_i = [\sigma_i^e \bmod N]$; since σ_i is chosen uniformly at random and RSA is a permutation, this means that y_i is uniformly distributed as well.

Moreover, j is independent of the view of $\mathcal{A}$ unless $\mathcal{A}$ happens to request a signature on m_j. But in this case the guess of $\mathcal{A}'$ was wrong (since $\mathcal{A}$ cannot output a forgery on m_j once it requests a signature on m_j).

When $\mathcal{A}'$ guesses correctly and $\mathcal{A}$ outputs a forgery, then $\mathcal{A}'$ solves the given instance of the RSA problem. Since $\mathcal{A}'$ guesses correctly with probability $1/q$, we have that

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n) = 1] = \varepsilon(n)/q(n).$$

Because the RSA problem is hard relative to GenRSA, the exists a negligible function negl such that

$$\Pr[\mathsf{RSA\text{-}inv}_{\mathcal{A}',\mathsf{GenRSA}}(n) = 1] \leq \mathsf{negl}(n).$$

Since q is polynomial, we conclude that $\varepsilon(n)$ is negligible as well, completing the proof. ∎

References and Additional Reading

The first formal treatment of the random oracle model was given by Bellare and Rogaway [19], though the idea of using a "random-looking" function in cryptographic applications had been suggested previously, most notably by [55]. Proper instantiation of a random oracle based on concrete cryptographic hash functions is discussed in [19, 20, 21, 63, 40].

The seminal negative result concerning the random oracle model is given by Canetti et al. [34], who show (contrived) schemes that can be proven secure in the random oracle model but demonstrably insecure for *any* concrete instantiation of the random oracle.

OAEP was introduced by Bellare and Rogaway [20] and later standardized as PKCS #1 v2.1 (available from `http://www.rsa.com/rsalabs`). The

original proof of OAEP was later found to be flawed; the interested reader is referred to [31, 59, 116] for further details.

The full-domain hash signature scheme was proposed by Bellare are Rogaway in their original paper on the random oracle model [19]. Later improvements include [21, 38, 39, 63], the first of which has been standardized as part of PKCS #1 v2.1.

Exercises

13.1 Prove that the pseudorandom function construction in Section 13.1.1 is indeed secure in the random oracle model.

13.2 In this exercise we show a scheme that can be proven secure in the random oracle model, but is insecure when the random oracle is instantiated with SHA-1. Let Π be a signature scheme that is secure in the standard model. Construct a signature scheme Π_y where signing is done as follows: if $H(0) = y$, then output the secret key; if $H(0) \neq y$, then return a signature computed using Π.

(a) Prove that for *any* value y, the scheme Π_y is secure in the random oracle model.

(b) Show that there exists a particular y for which Π_y is not secure when the random oracle is instantiated using SHA-1.

13.3 Let $\Pi' = (\mathsf{Enc}', \mathsf{Dec}')$ be a private-key encryption scheme, and modify Construction 13.1 so that encryption is done as follows: To encrypt a message $m \in \{0,1\}^{\ell(n)}$ with respect to the public key $\langle N, e\rangle$, choose random $r \leftarrow \mathbb{Z}_N^*$, set $k := H(r)$, and output the ciphertext

$$\left\langle [r^e \bmod N],\ \mathsf{Enc}'_k(m) \right\rangle .$$

(Decryption is done in the obvious way.) Prove that if the RSA problem is hard relative to GenRSA, Π' has indistinguishable encryptions under a chosen-plaintext attack, and H is modeled as a random oracle, this modified construction is a CPA-secure public-key encryption scheme.