The RSA System and Factoring 116

4

The RSA System and Factoring

4.1 Introduction to Public-key Cryptography

In the classical model of cryptography that we have been studying up until now, Alice and Bob secretly choose the key K. K then gives rise to an encryption rule e_K and a decryption rule d_K. In the cryptosystems we have seen so far, d_K is either the same as e_K, or easily derived from it (for example, **DES** decryption is identical to encryption, but the key schedule is reversed). Cryptosystems of this type are known as *private-key* systems, since exposure of e_K renders the system insecure.

One drawback of a private-key system is that it requires the prior communication of the key K between Alice and Bob, using a secure channel, before any ciphertext is transmitted. In practice, this may be very difficult to achieve. For example, suppose Alice and Bob live far away from each other and they decide that they want to communicate electronically, using e-mail. In a situation such as this, Alice and Bob may not have access to a reasonable secure channel.

The idea behind a *public-key* system is that it might be possible to find a cryptosystem where it is computationally infeasible to determine d_K given e_K. If so, then the encryption rule e_K could by made public by publishing it in a directory (hence the term public-key system). The advantage of a public-key system is that Alice (or anyone else) can send an encrypted message to Bob (without the prior communication of a secret key) by using the public encryption rule e_K. Bob will be the only person that can decrypt the ciphertext, using his secret decryption rule d_K.

Consider the following analogy: Alice places an object in a metal box, and then locks it with a combination lock left there by Bob. Bob is the only person who can open the box since only he knows the combination.

The idea of a public-key system was due to Diffie and Hellman in 1976. The first realization of a public-key system came in 1977 by Rivest, Shamir, and Adleman, who invented the well-known **RSA Cryptosystem** which we study in this chapter. Since then, several public-key systems have been proposed, whose security rests on different computational problems. Of these, the most important are the following:

RSA

The security of **RSA** is based on the difficulty of factoring large integers. This system is described in Section 4.3.

Merkle-Hellman Knapsack

This and related systems are based on the difficulty of the subset sum problem (which is **NP**-complete[1]); however, all of the various knapsack systems have been shown to be insecure (with the exception of the **Chor-Rivest Cryptosystem** mentioned below). See Chapter 5 for a discussion of this cryptosystem.

McEliece

The **McEliece Cryptosystem** is based on algebraic coding theory and is still regarded as being secure. It is based on the problem of decoding a linear code (which is also **NP**-complete). (See Chapter 5.)

ElGamal

The **ElGamal Cryptosystem** is based on the difficulty of the discrete logarithm problem for finite fields. (See Chapter 5.)

Chor-Rivest

This is also referred to as a "knapsack" type system, but it is still regarded as being secure.

Elliptic Curve

The **Elliptic Curve Cryptosystems** are modifications of other systems (such as the ElGamal Cryptosystem, for example) that work in the domain of elliptic curves rather than finite fields. The **Elliptic Curve Cryptosystems** appear to remain secure for smaller keys than other public-key cryptosystems. (See Chapter 5.)

One very important observation is that a public-key cryptosystem can never provide unconditional security. This is because an opponent, on observing a ciphertext y, can encrypt each possible plaintext in turn using the public encryption rule e_K until he finds the unique x such that $y = e_K(x)$. This x is the decryption of y. Consequently, we study the computational security of public-key systems.

It is helpful conceptually to think of a public-key system in terms of an abstraction called a trapdoor one-way function. We informally define this notion now.

Bob's public encryption function, e_K, should be easy to compute. We have just noted that computing the inverse function (i.e., decrypting) should be hard (for

[1]The **NP**-complete problems are a large class of problems for which no polynomial-time algorithms are known.

anyone other than Bob). This property of being easy to compute but hard to invert is often called the *one-way* property . Thus, we desire that e_K be an (injective) one-way function.

One-way functions play a central role in cryptography; they are important for constructing public-key cryptosystems and in various other contexts. Unfortunately, although there are many functions that are believed to be one-way, there currently do not exist functions that can be proved to be one-way.

Here is an example of a function which is believed to be one-way. Suppose n is the the product of two large primes p and q, and let b be a positive integer. Then define $f : \mathbb{Z}_n \rightarrow \mathbb{Z}_n$ to be

$$f(x) = x^b \bmod n.$$

(For a suitable choice of b and n, this is in fact the **RSA** encryption function; we will have much more to say about it later.)

If we are to construct a public-key cryptosystem, then it is not sufficient to find a one-way function. We do not want e_K to be a one-way function from Bob's point of view, since he wants to be able to decrypt messages that he receives in an efficient way. Thus, it is necessary that Bob possesses a *trapdoor*, which consists of secret information that permits easy inversion of e_K. That is, Bob can decrypt efficiently because he has some extra secret knowledge about K. So, we say that a function is a *trapdoor one-way* function if it is a one-way function, but it becomes easy to invert with the knowledge of a certain trapdoor.

We will see in Section 4.3 how to find a trapdoor for the function f defined above. This will lead to the **RSA Cryptosystem**.

4.2 More Number Theory

Before describing how **RSA** works, we need to discuss some more facts concerning modular arithmetic and number theory. Two fundamental results that we require are the Euclidean algorithm and the Chinese remainder theorem.

4.2.1 The Euclidean Algorithm

We already observed in Chapter 1 that $\mathbb{Z}_n$ is a ring for any positive integer n. We also proved there that $b \in \mathbb{Z}_n$ has a multiplicative inverse if and only if $\gcd(b, n) = 1$, and that the number of positive integers less than n and relatively prime to n is $\phi(n)$.

The set of residues modulo n that are relatively prime to n is denoted $\mathbb{Z}_n{}^*$. It is not hard to see that $\mathbb{Z}_n{}^*$ forms an abelian group under multiplication. We already have stated that multiplication modulo n is associative and commutative, and that 1 is the multiplicative identity. Any element in $\mathbb{Z}_n{}^*$ will have a multiplicative inverse (which is also in $\mathbb{Z}_n{}^*$). Finally, $\mathbb{Z}_n{}^*$ is closed under multiplication since xy is relatively prime to n whenever x and y are relatively prime to n (prove this!).

At this point, we know that any $b \in \mathbb{Z}_n{}^*$ has a multiplicative inverse, b^{-1}, but we do not yet have an efficient algorithm to compute b^{-1}. Such an algorithm exists; it is called the extended Euclidean algorithm.

First, we describe the Euclidean algorithm, in its basic form, which is used to compute the greatest common divisor of two positive integers, say r_0 and r_1, where $r_0 > r_1$. The Euclidean algorithm consists of performing the following sequence of divisions:

$$\begin{array}{rcll}
r_0 & = & q_1 r_1 + r_2, & 0 < r_2 < r_1 \\
r_1 & = & q_2 r_2 + r_3, & 0 < r_3 < r_2 \\
\vdots & & & \\
r_{m-2} & = & q_{m-1} r_{m-1} + r_m, & 0 < r_m < r_{m-1} \\
r_{m-1} & = & q_m r_m. &
\end{array}$$

Then it is not hard to show that

$$\gcd(r_0, r_1) = \gcd(r_1, r_2) = \ldots = \gcd(r_{m-1}, r_m) = r_m.$$

Hence, it follows that $\gcd(r_0, r_1) = r_m$.

Since the Euclidean algorithm computes greatest common divisors, it can be used to determine if a positive integer $b < n$ has a multiplicative inverse modulo n, by starting with $r_0 = n$ and $r_1 = b$. However, it does not compute the value of the multiplicative inverse (if it exists).

Now, suppose we define a sequence of numbers $t_0, t_1, \ldots, t_m$ according to the following recurrence (where the q_j's are defined as above):

$$\begin{array}{rcll}
t_0 & = & 0 & \\
t_1 & = & 1 & \\
t_j & = & t_{j-2} - q_{j-1} t_{j-1} \bmod r_0, & \text{if } j \geq 2.
\end{array}$$

Then we have the following useful result.

THEOREM 4.1
For $0 \leq j \leq m$, we have that $r_j \equiv t_j r_1 \pmod{r_0}$, where the q_j's and r_j's are defined as in the Euclidean algorithm, and the t_j's are defined in the above recurrence.

PROOF The proof is by induction on j. The assertion is trivially true for $j = 0$ and $j = 1$. Assume the assertion is true for $j = i - 1$ and $i - 2$, where $i \geq 2$; we will prove the assertion is true for $j = i$. By induction, we have that

$$r_{i-2} \equiv t_{i-2} r_1 \pmod{r_0}$$

and

$$r_{i-1} \equiv t_{i-1} r_1 \pmod{r_0}.$$

Now, we compute:

$$\begin{aligned} r_i &= r_{i-2} - q_{i-1}r_{i-1} \\ &\equiv t_{i-2}r_1 - q_{i-1}t_{i-1}r_1 \pmod{r_0} \\ &\equiv (t_{i-2} - q_{i-1}t_{i-1})r_1 \pmod{r_0} \\ &\equiv t_i r_1 \pmod{r_0}. \end{aligned}$$

Hence, the result is true by induction. ∎

The next corollary is an immediate consequence.

COROLLARY 4.2
Suppose $\gcd(r_0, r_1) = 1$. *Then* $t_m = r_1{}^{-1} \bmod r_0$.

Now, the sequence of numbers $t_0, t_1, \ldots t_m$ can be calculated in the Euclidean algorithm at the same time as the q_j's and the r_j's. In Figure 4.1, we present the extended Euclidean algorithm to compute the inverse of b modulo n, if it exists. In this version of the algorithm, we do not use an array to keep track of the q_j's, r_j's and t_j's, since it suffices to remember only the "last" two terms in each of these sequences at any point in the algorithm.

In step 10 of the algorithm, we have written the expression for *temp* in such a way that the reduction modulo n is done with a positive argument. (We mentioned earlier that modular reductions of negative numbers yield negative results in many computer languages; of course, we want to end up with a positive result here.) We also mention that at step 12, it is always the case that $tb \equiv r \pmod{n}$ (this is the result proved in Theorem 4.1).

Here is a small example to illustrate:

Example 4.1
Suppose we wish to compute $28^{-1} \bmod 75$. The Extended Euclidean algorithm proceeds as follows:

$75 = 2 \times 28 + 19$	step 6
$73 \times 28 \bmod 75 = 19$	step 12
$28 = 1 \times 19 + 9$	step 16
$3 \times 28 \bmod 75 = 9$	step 12
$19 = 2 \times 9 + 1$	step 16
$67 \times 28 \bmod 75 = 1$	step 12
$9 = 9 \times 1$	step 16

Hence, $28^{-1} \bmod 75 = 67$. □

FIGURE 4.1
Extended Euclidean algorithm

```
1.  $n_0 = n$
2.  $b_0 = b$
3.  $t_0 = 0$
4.  $t = 1$
5.  $q = \lfloor \frac{n_0}{b_0} \rfloor$
6.  $r = n_0 - q \times b_0$
7.  while $r > 0$ do
8.       $temp = t_0 - q \times t$
9.       if $temp \geq 0$ then $temp = temp \bmod n$
10.      if $temp < 0$ then $temp = n - ((-temp) \bmod n)$
11.      $t_0 = t$
12.      $t = temp$
13.      $n_0 = b_0$
14.      $b_0 = r$
15.      $q = \lfloor \frac{n_0}{b_0} \rfloor$
16.      $r = n_0 - q \times b_0$
17.      if $b_0 \neq 1$ then
             $b$ has no inverse modulo $n$
         else
             $b^{-1} = t \bmod n$
```

4.2.2 The Chinese Remainder Theorem

The Chinese remainder theorem is really a method of solving certain systems of congruences. Suppose $m_1, \ldots, m_r$ are pairwise relatively prime (that is, $\gcd(m_i, m_j) = 1$ if $i \neq j$). Suppose $a_1, \ldots, a_r$ are integers, and consider the following system of congruences:

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ &\vdots \\ x &\equiv a_r \pmod{m_r}. \end{aligned}$$

The Chinese remainder theorem asserts that this system has a unique solution modulo $M = m_1 \times m_2 \times \ldots \times m_r$. We will prove this result in this section, and also describe an efficient algorithm for solving systems of congruences of this type.

It is convenient to study the function $\pi : \mathbb{Z}_M \rightarrow \mathbb{Z}_{m_1} \times \ldots \times \mathbb{Z}_{m_r}$, which we define as follows:

$$\pi(x) = (x \bmod m_1, \ldots, x \bmod m_r).$$

Example 4.2
Suppose $r = 2$, $m_1 = 5$ and $m_2 = 3$, so $M = 15$. Then the function π has the following values:

$$\begin{array}{rclrclrcl}
\pi(0) & = & (0,0) & \pi(1) & = & (1,1) & \pi(2) & = & (2,2) \\
\pi(3) & = & (3,0) & \pi(4) & = & (4,1) & \pi(5) & = & (0,2) \\
\pi(6) & = & (1,0) & \pi(7) & = & (2,1) & \pi(8) & = & (3,2) \\
\pi(9) & = & (4,0) & \pi(10) & = & (0,1) & \pi(11) & = & (1,2) \\
\pi(12) & = & (2,0) & \pi(13) & = & (3,1) & \pi(14) & = & (4,2).
\end{array}$$

□

Proving the Chinese remainder theorem amounts to proving that this function π we have defined is a bijection. In Example 4.2 this is easily seen to be the case. In fact, we will be able to give an explicit general formula for the inverse function π^{-1}.

For $1 \leq i \leq r$, define

$$M_i = \frac{M}{m_i}.$$

Then it is not difficult to see that

$$\gcd(M_i, m_i) = 1$$

for $1 \leq i \leq r$. Next, for $1 \leq i \leq r$, define

$$y_i = M_i^{-1} \bmod m_i.$$

(This inverse exists since $\gcd(M_i, m_i) = 1$, and it can be found using the Euclidean algorithm .) Note that

$$M_i y_i \equiv 1 \pmod{m_i}$$

for $1 \leq i \leq r$.

Now, define a function $\rho : \mathbb{Z}_{m_1} \times \ldots \times \mathbb{Z}_{m_r} \to \mathbb{Z}_M$ as follows:

$$\rho(a_1, \ldots, a_r) = \sum_{i=1}^{r} a_i M_i y_i \bmod M.$$

We will show that the function $\rho = \pi^{-1}$, i.e., it provides an explicit formula for solving the original system of congruences.

Denote $X = \rho(a_1, \ldots, a_r)$, and let $1 \leq j \leq r$. Consider a term $a_i M_i y_i$ in the above summation, reduced modulo m_j: If $i = j$, then

$$a_i M_i y_i \equiv a_i \pmod{m_i}$$

since

$$M_i y_i \equiv 1 \pmod{m_i}.$$

On the other hand, if $i \neq j$, then

$$a_i M_i y_i \equiv 0 \pmod{m_j}$$

since $m_j \mid M_i$ in this case. Thus, we have that

$$\begin{aligned} X &\equiv \sum_{i=1}^{r} a_i M_i y_i \pmod{m_j} \\ &\equiv a_j \pmod{m_j}. \end{aligned}$$

Since this is true for all j, $1 \leq j \leq r$, X is a solution to the system of congruences.

At this point, we need to show that the solution X is unique modulo M. But this can be done by simple counting. The function π is a function from a domain of cardinality M to a range of cardinality M. We have just proved that π is a surjective (i.e., onto) function. Hence, π must also be injective (i.e., one-to-one), since the domain and range have the same cardinality. It follows that π is a bijection and $\pi^{-1} = \rho$. Note also that π^{-1} is a linear function of its arguments $a_1, \ldots, a_r$.

Here is a bigger example to illustrate.

Example 4.3
Suppose $r = 3$, $m_1 = 7$, $m_2 = 11$ and $m_3 = 13$. Then $M = 1001$. We compute $M_1 = 143$, $M_2 = 91$ and $M_3 = 77$, and then $y_1 = 5$, $y_2 = 4$ and $y_3 = 12$. Then the function $\pi^{-1} : \mathbb{Z}_7 \times \mathbb{Z}_{11} \times \mathbb{Z}_{13} \to \mathbb{Z}_{1001}$ is the following:

$$\pi^{-1}(a_1, a_2, a_3) = 715a_1 + 364a_2 + 924a_3 \bmod 1001.$$

For example, if $x \equiv 5 \pmod 7$, $x \equiv 3 \pmod{11}$ and $x \equiv 10 \pmod{13}$, then this formula tells us that

$$x = 715 \times 5 + 364 \times 3 + 924 \times 10 \bmod 1001$$

$$= 13907 \bmod 1001$$
$$= 894 \bmod 1001.$$

This can be verified by reducing 894 modulo 7, 11 and 13. □

For future reference, we record the results of this section as a theorem.

THEOREM 4.3 (Chinese Remainder Theorem)
Suppose $m_1, \ldots, m_r$ are pairwise relatively prime positive integers, and let $a_1, \ldots, a_r$ be integers. Then, the system of r congruences $x \equiv a_i \pmod{m_i}$ $(1 \leq i \leq r)$ has a unique solution modulo $M = m_1 \times \ldots \times m_r$, which is given by

$$x = \sum_{i=1}^{r} a_i M_i y_i \bmod M,$$

where $M_i = M/m_i$ and $y_i = {M_i}^{-1} \bmod m_i$, for $1 \leq i \leq r$.

4.2.3 Other Useful Facts

We next mention another result from elementary group theory, called Lagrange's Theorem, that will be relevant in our treatment of the **RSA Cryptosystem**. For a (finite) multiplicative group G, define the *order* of an element $g \in G$ to be the smallest positive integer m such that $g^m = 1$. The following result is fairly simple, but we will not prove it here.

THEOREM 4.4 (Lagrange)
Suppose G is a multiplicative group of order n, and $g \in G$. Then the order of g divides n.

For our purposes, the following corollaries are essential.

COROLLARY 4.5
If $b \in {\mathbb{Z}_n}^*$, then $b^{\phi(n)} \equiv 1 \pmod{n}$.

PROOF ${\mathbb{Z}_n}^*$ is a multiplicative group of order $\phi(n)$. ∎

COROLLARY 4.6 (Fermat)
Suppose p is prime and $b \in \mathbb{Z}_p$. Then $b^p \equiv b \pmod{p}$.

PROOF If p is prime, then $\phi(p) = p - 1$. So, for $b \not\equiv 0 \pmod{p}$, the result follows from Corollary 4.5. For $b \equiv 0 \pmod{p}$, the result is also true since $0^p \equiv 0 \pmod{p}$. ∎

At this point, we know that if p is prime, then $\mathbb{Z}_p{}^*$ is a group of order $p-1$, and any element in $\mathbb{Z}_p{}^*$ has order dividing $p-1$. However, if p is prime, then the group $\mathbb{Z}_p{}^*$ is in fact *cyclic*: there exists an element $\alpha \in \mathbb{Z}_p{}^*$ having order equal to $p-1$. We will not prove this very important fact, but we do record it for future reference:

THEOREM 4.7
If p is prime, then $\mathbb{Z}_p{}^*$ is a cyclic group.

An element α having order $p-1$ is called a *primitive* element modulo p. Observe that α is a primitive element if and only if

$$\{\alpha^i : 0 \leq i \leq p-2\} = \mathbb{Z}_p{}^*.$$

Now, suppose p is prime and α is a primitive element modulo p. Any element $\beta \in \mathbb{Z}_p{}^*$ can be written as $\beta = \alpha^i$, where $0 \leq i \leq p-2$, in a unique way. It is not difficult to prove that the order of $\beta = \alpha^i$ is

$$\frac{p-1}{\gcd(p-1, i)}.$$

Thus β is itself a primitive element if and only if $\gcd(p-1, i) = 1$. It follows that the number of primitive elements modulo p is $\phi(p-1)$.

Example 4.4
Suppose $p = 13$. By computing successive powers of 2, we can verify that 2 is a primitive element modulo 13:

$$\begin{aligned}
2^0 \bmod 13 &= 1\\
2^1 \bmod 13 &= 2\\
2^2 \bmod 13 &= 4\\
2^3 \bmod 13 &= 8\\
2^4 \bmod 13 &= 3\\
2^5 \bmod 13 &= 6\\
2^6 \bmod 13 &= 12\\
2^7 \bmod 13 &= 11\\
2^8 \bmod 13 &= 9\\
2^9 \bmod 13 &= 5\\
2^{10} \bmod 13 &= 10\\
2^{11} \bmod 13 &= 7.
\end{aligned}$$

FIGURE 4.2
RSA Cryptosystem

Let $n = pq$, where p and q are primes. Let $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$, and define

$$\mathcal{K} = \{(n, p, q, a, b) : n = pq, p, q \text{ prime}, ab \equiv 1 \pmod{\phi(n)}\}.$$

For $K = (n, p, q, a, b)$, define

$$e_K(x) = x^b \bmod n$$

and

$$d_K(y) = y^a \bmod n$$

$(x, y \in \mathbb{Z}_n)$. The values n and b are public, and the values p, q, a are secret.

The element 2^i is primitive if and only if $\gcd(i, 12) = 1$; i.e., if and only if $i = 1, 5, 7$ or 11. Hence, the primitive elements modulo 13 are 2, 6, 7 and 11. □

4.3 The RSA Cryptosystem

We can now describe the **RSA Cryptosystem**. This cryptosystem uses computations in $\mathbb{Z}_n$, where n is the product of two distinct odd primes p and q. For such n, note that $\phi(n) = (p-1)(q-1)$.

The formal description of the cryptosystem is given in Figure 4.2. Let's verify that encryption and decryption are inverse operations. Since

$$ab \equiv 1 \pmod{\phi(n)},$$

we have that

$$ab = t\phi(n) + 1$$

for some integer $t \geq 1$. Suppose that $x \in \mathbb{Z}_n{}^*$; then we have

$$\begin{aligned}(x^b)^a &\equiv x^{t\phi(n)+1} \pmod{n}\\ &\equiv (x^{\phi(n)})^t x \pmod{n}\\ &\equiv 1^t x \pmod{n}\\ &\equiv x \pmod{n},\end{aligned}$$

as desired. We leave it as an exercise for the reader to show that $(x^b)^a \equiv x \pmod{n}$ if $x \in \mathbb{Z}_n \backslash {\mathbb{Z}_n}^*$.

Here is a small (insecure) example of the **RSA Cryptosystem**.

Example 4.5

Suppose Bob chooses $p = 101$ and $q = 113$. Then $n = 11413$ and $\phi(n) = 100 \times 112 = 11200$. Since $11200 = 2^6 5^2 7$, an integer b can be used as an encryption exponent if and only if b is not divisible by $2, 5$ or 7. (In practice, however, Bob will not factor $\phi(n)$. He will verify that $\gcd(\phi(n), b) = 1$ using the Euclidean algorithm.) Suppose Bob chooses $b = 3533$. Then the Extended Euclidean algorithm will yield

$$b^{-1} = 6597 \bmod 11200.$$

Hence, Bob's secret decryption exponent is $a = 6597$.

Bob publishes $n = 11413$ and $b = 3533$ in a directory. Now, suppose Alice wants to send the plaintext 9726 to Bob. She will compute

$$9726^{3533} \bmod 11413 = 5761$$

and send the ciphertext 5761 over the channel. When Bob receives the ciphertext 5761, he uses his secret decryption exponent to compute

$$5761^{6597} \bmod 11413 = 9726.$$

(At this point, the encryption and decryption operations might appear to be very complicated, but we will discuss efficient algorithms for these operations in the next section.) □

The security of **RSA** is based on the hope that the encryption function $e_K(x) = x^b \bmod n$ is one-way, so it will be computationally infeasible for an opponent to decrypt a ciphertext. The trapdoor that allows Bob to decrypt is the knowledge of the factorization $n = pq$. Since Bob knows this factorization, he can compute $\phi(n) = (p-1)(q-1)$ and then compute the decryption exponent a using the Extended Euclidean algorithm. We will say more about the security of **RSA** later on.

4.4 Implementing RSA

There are many aspects of the **RSA Cryptosystem** to discuss, including the details of setting up the cryptosystem, the efficiency of encrypting and decrypting, and security issues. In order to set up the system, Bob follows the steps indicated in

FIGURE 4.3
Setting up RSA

1. Bob generates two large primes, p and q
2. Bob computes $n = pq$ and $\phi(n) = (p-1)(q-1)$
3. Bob chooses a random b ($0 < b < \phi(n)$) such that $\gcd(b, \phi(n)) = 1$
4. Bob computes $a = b^{-1} \bmod \phi(n)$ using the Euclidean algorithm
5. Bob publishes n and b in a directory as his public key.

Figure 4.3. How Bob carries out these steps will be discussed later in this chapter.

One obvious attack on the cryptosystem is for a cryptanalyst to attempt to factor n. If this can be done, it is a simple manner to compute $\phi(n) = (p-1)(q-1)$ and then compute the decryption exponent a from b exactly as Bob did. (It has been conjectured that breaking **RSA** is polynomially equivalent[2] to factoring n, but this remains unproved.)

Hence, if the **RSA Cryptosystem** is to be secure, it is certainly necessary that $n = pq$ must be large enough that factoring it will be computationally infeasible. Current factoring algorithms are able to factor numbers having up to 130 decimal digits (for more information on factoring, see Section 4.8). Hence, it is recommended that, to be on the safe side, one should choose p and q to each be primes having about 100 digits; then n will have 200 digits. Several hardware implementations of **RSA** use a modulus which is 512 bits in length. However, a 512-bit modulus corresponds to about 154 decimal digits (since the number of bits in the binary representation of an integer is $\log_2 10$ times the number of decimal digits), and hence it does not offer good long-term security.

Leaving aside for the moment the question of how to find 100 digit primes, let us look now at the arithmetic operations of encryption and decryption. An encryption (or decryption) involves performing one exponentiation modulo n. Since n is very large, we must use multiprecision arithmetic to perform computations in $\mathbb{Z}_n$, and the time required will depend on the number of bits in the binary representation of n.

Suppose n has k bits in its binary representation; i.e., $k = \lfloor \log_2 n \rfloor + 1$. Using standard "grade-school" arithmetic techniques, it is not difficult to see that an addition of two k-bit integers can be done in time $O(k)$, and a multiplication can be done in time $O(k^2)$. Also, a reduction modulo n of an integer having at most

[2]Two problems are said to be *polynomially equivalent* if the existence of a polynomial-time algorithm for either problem implies the existence of a polynomial-time algorithm for the other problem.

FIGURE 4.4
The square-and-multiply algorithm to compute $x^b \bmod n$

```
1.  $z = 1$
2.  for $i = \ell - 1$ downto 0 do
3.      $z = z^2 \bmod n$
4.      if $b_i = 1$ then $z = z \times x \bmod n$
```

$2k$ bits can be performed in time $O(k^2)$ (this amounts to doing long division and retaining the remainder). Now, suppose that $x, y \in \mathbb{Z}_n$ (where we are assuming that $0 \leq x, y \leq n - 1$). Then $xy \bmod n$ can be computed by first calculating the product xy (which is a $2k$-bit integer), and then reducing it modulo n. These two steps can be peformed in time $O(k^2)$. We call this computation *modular multiplication.*

We now consider *modular exponentiation*, i.e., computation of a function of the form $x^c \bmod n$. As noted above, both the encryption and the decryption operations in **RSA** are modular exponentiations. Computation of $x^c \bmod n$ can be done using $c - 1$ modular multiplications; however, this is very inefficient if c is large. Note that c might be as big as $\phi(n) - 1$, which is exponentially large compared to k.

The well-known "square-and-multiply" approach reduces the number of modular multiplications required to compute $x^c \bmod n$ to at most 2ℓ, where ℓ is the number of bits in the binary representation of c. Since $\ell \leq k$, it follows that $x^c \bmod n$ can be computed in time $O(k^3)$. Hence, **RSA** encryption and decryption can both be done in polynomial time (as a function of k, which is the number of bits in one plaintext (or ciphertext) character).

Square-and-multiply assumes that the exponent, b say, is represented in binary notation, say

$$b = \sum_{i=0}^{\ell-1} b_i 2^i,$$

where $b_i = 0$ or 1, $0 \leq i \leq \ell - 1$. The algorithm to compute $z = x^b \bmod n$ is presented in Figure 4.4. It is easy to count the number of modular multiplications performed by the square-and-multiply algorithm. There are always ℓ squarings performed (step 3). The number of modular multiplcations in step 4 is equal to the number of 1's in the binary representation of b, which is an integer between 0 and ℓ. Thus, the total number of modular multiplications is at least ℓ and at most 2ℓ.

We will illustrate the use of square-and-multiply by returning to Example 4.5.

Example 4.5 ***(Cont.)***
Recall that $n = 11413$, and the public encryption exponent is $b = 3533$. Alice encrypts the plaintext 9726 by computing $9726^{3533} \bmod 11413$, using the square-and multiply algorithm, as follows:

i	b_i	z
11	1	$1^2 \times 9726 = 9726$
10	1	$9726^2 \times 9726 = 2659$
9	0	$2659^2 = 5634$
8	1	$5634^2 \times 9726 = 9167$
7	1	$9167^2 \times 9726 = 4958$
6	1	$4958^2 \times 9726 = 7783$
5	0	$7783^2 = 6298$
4	0	$6298^2 = 4629$
3	1	$4629^2 \times 9726 = 10185$
2	1	$10185^2 \times 9726 = 105$
1	0	$105^2 = 11025$
0	1	$11025^2 \times 9726 = 5761$

Hence, as stated earlier, the ciphertext is 5761. □

It should be emphasized that the most efficient current hardware implementations of **RSA** achieve encryption rates of about 600 Kbits per second (using a 512 bit modulus n), as compared to 1 Gbit per second for **DES**. Stated another way, **RSA** is roughly 1500 times slower than **DES**.

At this point we have discussed the encryption and decryption operations for **RSA**. In terms of setting up **RSA**, the generation of the primes p and q (Step 1) will be discussed in the next section. Step 2 is straightforward and can be done in time $O((\log n)^2)$. Steps 3 and 4 involve the Euclidean algorithm, so let's briefly consider its complexity.

Suppose we compute the greatest common divisor of r_0 and r_1, where $r_0 > r_1$. In each iteration of the algorithm, we compute a quotient and remainder, which can be done in time $O((\log r_0)^2)$. If we can obtain an upper bound on the number of iterations, then we will have a bound on the complexity of the algorithm. There is a well-known result, known as Lamé's Theorem, that provides such a bound. It asserts that if s is the number of iterations, then $f_{s+2} \leq r_0$, where f_i denotes the ith Fibonacci number. Since

$$f_i \approx \left(\frac{1+\sqrt{5}}{2}\right)^i,$$

it follows that s is $O(\log r_0)$.

This shows that the running time of the Euclidean algorithm is $O((\log n)^3)$. (Actually, a more careful analysis can be used to show that the running time is, in fact, $O((\log n)^2)$.)

4.5 Probabilistic Primality Testing

In setting up the **RSA Cryptosystem**, it is necessary to generate large (e.g., 80 digit) "random primes." In practice, the way this is done is to generate large random numbers, and then test them for primality using a probabilistic polynomial-time Monte Carlo algorithm such as the Solovay-Strassen or Miller-Rabin algorithm , both of which we will present in this section. These algorithms are fast (i.e., an integer n can be tested in time that is polynomial in $\log_2 n$, the number of bits in the binary representation of n), but there is a possibility that the algorithm may claim that n is prime when it is not. However, by running the algorithm enough times, the error probability can be reduced below any desired threshold. (We will discuss this in more detail a bit later.)

The other pertinent question is how many random integers (of a specified size) will need to be tested until we find one that is prime. A famous result in number theory, called the Prime number theorem, states that the number of primes not exceeding N is approximately $N/\ln N$. Hence, if p is chosen at random, the probability that it is prime is about $1/\ln p$. For a 512 bit modulus, we have $1/\ln p \approx 1/177$. That is, on average, of 177 random integers p of the appropriate size, one will be prime (of course, if we restrict our attention to odd integers, the probability doubles, to about $2/177$). So it is indeed practical to generate sufficiently large random numbers that are "probably prime," and hence it is practical to set up the **RSA Cryptosystem**. We proceed to describe how this is done.

A *decision problem* is a problem in which a question is to be answered "yes" or "no." A *probabilistic* algorithm is any algorithm that uses random numbers (in contrast, an algorithm that does not use random numbers is called a *deterministic* algorithm). The following definitions pertain to probabilistic algorithms for decision problems.

DEFINITION 4.1 A ***yes-biased Monte Carlo algorithm*** *is a probabilistic algorithm for a decision problem in which a "yes" answer is (always) correct, but a "no" answer may be incorrect. A* ***no-biased*** *Monte Carlo algorithm is defined in the obvious way. We say that a yes-biased Monte Carlo algorithm has* ***error probability*** *equal to ϵ if, for any instance in which the answer is "yes," the algorithm will give the (incorrect) answer "no" with probability at most ϵ. (This probability is computed over all possible random choices made by the algorithm when it is run with a given input.)*

The decision problem called **Composites** is described in Figure 4.5.

Note that an algorithm for a decision problem only has to answer "yes" or "no." In particular, in the case of the problem **Composites**, we do not require the algorithm to find a factorization in the case that n is composite.

We will first describe the Solovay-Strassen algorithm, which is a yes-biased

FIGURE 4.5
Composites

Problem Instance	A positive integer $n \geq 2$.
Question	Is n composite?

FIGURE 4.6
Quadratic Residues

Problem Instance	An odd prime p, and an integer x such that $0 \leq x \leq p - 1$.
Question	Is x a quadratic residue modulo p?

Monte Carlo algorithm for **Composites** with error probability $1/2$. Hence, if the algorithm answers "yes," then n is composite; conversely, if n is composite, then the algorithm answers "yes" with probability at least $1/2$.

Although the Miller-Rabin algorithm (which we will discuss later) is faster than Solovay-Strassen, we begin by looking at the Solovay-Strassen algorithm because it is easier to understand conceptually and because it involves some number-theoretic concepts that will be useful in later chapters of the book. We begin by developing some further background from number theory before describing the algorithm.

DEFINITION 4.2 Suppose p is an odd prime and x is an integer, $1 \leq x \leq p - 1$. x is defined to be a ***quadratic residue*** *modulo p if the congruence $y^2 \equiv x \pmod{p}$ has a solution $y \in \mathbb{Z}_p$. x is defined to be a* **quadratic non-residue** *modulo p if $x \not\equiv 0 \pmod{p}$ and x is not a quadratic residue modulo p.*

Example 4.6
The quadratic residues modulo 11 are $1, 3, 4, 5$ and 9. Note that $(\pm 1)^2 = 1$, $(\pm 4)^2 = 3$, $(\pm 2)^2 = 4$, $(\pm 4)^2 = 5$ and $(\pm 3)^2 = 9$ (where all arithmetic is in $\mathbb{Z}_{11}$). $\square$

The decision problem **Quadratic Residues** is defined in Figure 4.6 in the obvious way:

We prove a result, known as *Euler's criterion*, that will give rise to a polynomial-time deterministic algorithm for **Quadratic Residues**.

THEOREM 4.8 (Euler's Criterion)
Let p be prime. Then x is a quadratic residue modulo p if and only if

$$x^{(p-1)/2} \equiv 1 \pmod{p}.$$

PROOF First, suppose $x \equiv y^2 \pmod{p}$. Recall from Corollary 4.6 that if p is prime, then $x^{p-1} \equiv 1 \pmod{p}$ for any $x \not\equiv 0 \pmod{p}$. Thus we have

$$\begin{aligned} x^{(p-1)/2} &\equiv (y^2)^{(p-1)/2} \pmod{p} \\ &\equiv y^{p-1} \pmod{p} \\ &\equiv 1 \pmod{p}. \end{aligned}$$

Conversely, suppose $x^{(p-1)/2} \equiv 1 \pmod{p}$. Let b be a primitive element modulo p. Then $x \equiv b^i \pmod{p}$ for some i. Then we have

$$\begin{aligned} x^{(p-1)/2} &\equiv (b^i)^{(p-1)/2} \pmod{p} \\ &\equiv b^{i(p-1)/2} \pmod{p}. \end{aligned}$$

Since b has order $p-1$, it must be the case that $p-1$ divides $i(p-1)/2$. Hence, i is even, and then the square roots of x are $\pm b^{i/2}$. ▮

Theorem 4.8 yields a polynomial-time algorithm for **Quadratic Residues**, by using the "square-and-multiply" technique for exponentiation modulo p. The complexity of the algorithm will be $O((\log p)^3)$.

We now need to give some further definitions from number theory.

DEFINITION 4.3 *Suppose p is an odd prime. For any integer $a \geq 0$, we define the* ***Legendre symbol*** $\left(\frac{a}{p}\right)$ *as follows:*

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{p} \\ 1 & \text{if } a \text{ is a quadratic residue modulo } p \\ -1 & \text{if } a \text{ is a quadratic non-residue modulo } p. \end{cases}$$

We have already seen that $a^{(p-1)/2} \equiv 1 \pmod{p}$ if and only if a is a quadratic residue modulo p. If a is a multiple of p, then it is clear that $a^{(p-1)/2} \equiv 0 \pmod{p}$. Finally, if a is a quadratic non-residue modulo p, then $a^{(p-1)/2} \equiv -1 \pmod{p}$ since $a^{p-1} \equiv 1 \pmod{p}$. Hence, we have the following result, which provides an efficient algorithm to evaluate Legendre symbols:

THEOREM 4.9
Suppose p is prime. Then

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}.$$

Next, we define a generalization of the Legendre symbol.

DEFINITION 4.4 *Suppose n is an odd positive integer, and the prime power factorization of n is ${p_1}^{e_1} \dots {p_k}^{e_k}$. Let $a \geq 0$ be an integer. The **Jacobi symbol** $\left(\frac{a}{n}\right)$ is defined to be*

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i}.$$

Example 4.7
Consider the Jacobi symbol $\left(\frac{6278}{9975}\right)$. The prime power factorization of 9975 is $9975 = 3 \times 5^2 \times 7 \times 19$. Thus we have

$$\begin{aligned}\left(\frac{6278}{9975}\right) &= \left(\frac{6278}{3}\right)\left(\frac{6278}{5}\right)^2\left(\frac{6278}{7}\right)\left(\frac{6278}{19}\right)\\ &= \left(\frac{2}{3}\right)\left(\frac{3}{5}\right)^2\left(\frac{6}{7}\right)\left(\frac{8}{19}\right)\\ &= (-1)(-1)^2(-1)(-1)\\ &= -1.\end{aligned}$$

□

Suppose $n > 1$ is odd. If n is prime then $\left(\frac{a}{n}\right) \equiv a^{(n-1)/2} \pmod{n}$ for any a. On the other hand, if n is composite, it may or may not be the case that $\left(\frac{a}{n}\right) \equiv a^{(n-1)/2} \pmod{n}$. If this equation holds, then a is called an *Euler pseudo-prime* to the base n. For example, 10 is an Euler pseudo-prime to the base 91, since

$$\left(\frac{10}{91}\right) = -1 = 10^{45} \bmod 91.$$

However, it can be shown that, for any odd composite n, at most half of the integers a such that $1 \leq a \leq n-1$ are Euler pseudo-primes to base n (see the exercises). This fact shows that the Solovay-Strassen primality test, which we present in Figure 4.7, is a yes-biased Monte Carlo algorithm with error probability at most 1/2. At this point it is not clear that the algorithm is a polynomial-time algorithm. We already know how to evaluate $a^{(n-1)/2} \bmod n$ in time $O((\log n)^3)$, but how do we compute Jacobi symbols efficiently? It might appear to be necessary to first factor n, since the Jacobi symbol $\left(\frac{a}{n}\right)$ is defined in terms of the factorization of n. But, if we could factor n, we would already know if it is prime, so this approach ends up in a vicious circle.

Fortunately, we can evaluate a Jacobi symbol without factoring n by using some results from number theory, the most important of which is a generalization of

FIGURE 4.7
The Solovay-Strassen primality test for an odd integer n

```
1.  choose a random integer $a$, $1 \le a \le n-1$
2.  if $\left(\frac{a}{n}\right) \equiv a^{(n-1)/2} \pmod{n}$ then
        answer "$n$ is prime"
    else
        answer "$n$ is composite"
```

the law of quadratic reciprocity (property 4 below). We now enumerate these properties without proof:

1. If n is an odd integer and $m_1 \equiv m_2 \pmod{n}$, then

$$\left(\frac{m_1}{n}\right) = \left(\frac{m_2}{n}\right).$$

2. If n is an odd integer, then

$$\left(\frac{2}{n}\right) = \begin{cases} 1 & \text{if } n \equiv \pm 1 \pmod 8 \\ -1 & \text{if } n \equiv \pm 3 \pmod 8. \end{cases}$$

3. If n is an odd integer then

$$\left(\frac{m_1 m_2}{n}\right) = \left(\frac{m_1}{n}\right)\left(\frac{m_2}{n}\right).$$

In particular, if $m = 2^k t$, where t is odd, then

$$\left(\frac{m}{n}\right) = \left(\frac{2}{n}\right)^k \left(\frac{t}{n}\right).$$

4. Suppose m and n are odd integers. Then

$$\left(\frac{m}{n}\right) = \begin{cases} -\left(\frac{n}{m}\right) & \text{if } m \equiv n \equiv 3 \pmod 4 \\ \left(\frac{n}{m}\right) & \text{otherwise.} \end{cases}$$

Example 4.8
As an illustration of the application of these properties, we evaluate the Jacobi

Previous Chapter Previous Page Home Next page Next Chapter

symbol $\left(\frac{7411}{9283}\right)$ as follows:

$$\begin{aligned}
\left(\frac{7411}{9283}\right) &= -\left(\frac{9283}{7411}\right) && \text{by property 4}\\
&= -\left(\frac{1872}{7411}\right) && \text{by property 1}\\
&= -\left(\frac{2}{7411}\right)^4\left(\frac{117}{7411}\right) && \text{by property 3}\\
&= -\left(\frac{117}{7411}\right) && \text{by property 2}\\
&= -\left(\frac{7411}{117}\right) && \text{by property 4}\\
&= -\left(\frac{40}{117}\right) && \text{by property 1}\\
&= -\left(\frac{2}{117}\right)^3\left(\frac{5}{117}\right) && \text{by property 3}\\
&= \left(\frac{5}{117}\right) && \text{by property 2}\\
&= \left(\frac{117}{5}\right) && \text{by property 4}\\
&= \left(\frac{2}{5}\right) && \text{by property 1}\\
&= -1 && \text{by property 2}
\end{aligned}$$

Notice that we successively apply properties 4, 1, 3, and 2 in this computation.
□

In general, by applying these four properties, it is possible to compute a Jacobi symbol $\left(\frac{m}{n}\right)$ in polynomial time. The only arithmetic operations that are required are modular reductions and factoring out powers of two. Note that if an integer is represented in binary notation, then factoring out powers of two amounts to determining the number of trailing zeroes. So, the complexity of the algorithm is determined by the number of modular reductions that must be done. It is not difficult to show that at most $O(\log n)$ modular reductions are performed, each of which can be done in time $O((\log n)^2)$. This shows that the complexity is $O((\log n)^3)$, which is polynomial in $\log n$. (In fact, the complexity can be shown to be $O((\log n)^2)$ by more precise analysis.)

Suppose that we have generated a random number n and tested it for primality using the Solovay-Strassen algorithm. If we have run the algorithm m times, what is our confidence that n is prime? It is tempting to conclude that the probability that such an integer n is prime is $1 - 2^{-m}$. This conclusion is often stated in both textbooks and technical articles, but it cannot be inferred from the given data.

We need to be careful about our use of probabilities. We will define the following random variables: **a** denotes the event

"a random odd integer n of a specified size is composite,"

and **b** denotes the event

"the algorithm answers 'n is prime' m times in succession."

It is certainly the case that $prob(\mathbf{b}|\mathbf{a}) \leq 2^{-m}$. However, the probability that we are really interested is $prob(\mathbf{a}|\mathbf{b})$, which is usually not the same as $prob(\mathbf{b}|\mathbf{a})$.

We can compute $prob(\mathbf{a}|\mathbf{b})$ using Bayes' theorem (Theorem 2.1). In order to do this, we need to know $prob(\mathbf{a})$. Suppose $N \leq n \leq 2N$. Applying the Prime number theorem, the number of (odd) primes between N and $2N$ is approximately

$$\begin{aligned} \frac{2N}{\ln 2N} - \frac{N}{\ln N} &\approx \frac{N}{\ln N} \\ &\approx \frac{n}{\ln n}. \end{aligned}$$

Since there are $N/2 \approx n/2$ odd integers between N and $2N$, we will use the estimate

$$prob(\mathbf{a}) \approx 1 - \frac{2}{\ln n}.$$

Then we can compute as follows:

$$\begin{aligned} prob(\mathbf{a}|\mathbf{b}) &= \frac{prob(\mathbf{b}|\mathbf{a})prob(\mathbf{a})}{prob(\mathbf{b})} \\ &= \frac{prob(\mathbf{b}|\mathbf{a})prob(\mathbf{a})}{prob(\mathbf{b}|\mathbf{a})prob(\mathbf{a}) + prob(\mathbf{b}|\overline{\mathbf{a}})prob(\overline{\mathbf{a}})} \\ &\approx \frac{prob(\mathbf{b}|\mathbf{a})\left(1 - \frac{2}{\ln n}\right)}{prob(\mathbf{b}|\mathbf{a})\left(1 - \frac{2}{\ln n}\right) + \frac{2}{\ln n}} \\ &= \frac{prob(\mathbf{b}|\mathbf{a})(\ln n - 2)}{prob(\mathbf{b}|\mathbf{a})(\ln n - 2) + 2} \\ &\leq \frac{2^{-m}(\ln n - 2)}{2^{-m}(\ln n - 2) + 2} \\ &= \frac{\ln n - 2}{\ln n - 2 + 2^{m+1}}. \end{aligned}$$

Note that in this computation, $\overline{\mathbf{a}}$ denotes the event

FIGURE 4.8
Error probabilities for the Solovay-Strassen test

m	2^{-m}	$\frac{175}{175+2^{m+1}}$
1	.500	.978
2	.250	.956
5	$.312 \times 10^{-1}$	.732
10	$.977 \times 10^{-3}$	$.787 \times 10^{-1}$
20	$.954 \times 10^{-6}$	$.834 \times 10^{-4}$
30	$.931 \times 10^{-9}$	$.815 \times 10^{-7}$
50	$.888 \times 10^{-15}$	$.777 \times 10^{-13}$
100	$.789 \times 10^{-30}$	$.690 \times 10^{-28}$

"a random odd integer n is prime."

It is interesting to compare the two quantities $(\ln n - 2)/(\ln n - 2 + 2^{m+1})$ and 2^{-m} as a function of m. Suppose that $n \approx 2^{256} \approx e^{177}$, since these are the sizes of primes that we seek for use in **RSA**. Then the first function is roughly $175/(175 + 2^{m+1})$. We tabulate the two functions for some values of m in Figure 4.8.

Although $175/(175+2^{m+1})$ approaches zero exponentially quickly, it does not do so as quickly as 2^{-m}. In practice, however, one would take m to be something like 50 or 100, which will reduce the probability of error to a very small quantity.

We conclude this section with another Monte Carlo algorithm for **Composites** which is known as the Miller-Rabin algorithm (it is also known as the "strong pseudo-prime test"). This algorithm is presented in Figure 4.9. It is clearly a polynomial-time algorithm: its complexity is $O((\log n)^3)$, which is the same as the Solovay-Strassen test. In fact, the Miller-Rabin algorithm performs better in practice than the Solovay-Strassen algorithm.

We show now that this algorithm cannot answer "n is composite" if n is prime, i.e., the algorithm is yes-biased.

THEOREM 4.10

The Miller-Rabin algorithm for **Composites** *is a yes-biased Monte Carlo algorithm.*

PROOF We will prove this by assuming the algorithm answers "n is composite" for some prime integer n, and obtain a contradiction. Since the algorithm answers "n is composite," it must be the case that $a^m \not\equiv 1 \pmod{n}$. Now consider the sequence of values b tested in step 2 of the algorithm. Since b is squared in each iteration of the **for** loop, we are testing the values $a^m, a^{2m}, \ldots, a^{2^{k-1}m}$. Since the

Previous Chapter Previous Page Home Next page Next Chapter

FIGURE 4.9
The Miller-Rabin primality test for an odd integer n

```
1.  write $n - 1 = 2^k m$, where $m$ is odd
2.  choose a random integer $a$, $1 \le a \le n - 1$
3.  compute $b = a^m \bmod n$
4.  if $b \equiv 1 \pmod{n}$ then
          answer "$n$ is prime" and quit
5.  for $i = 0$ to $k - 1$ do
6.        if $b \equiv -1 \pmod{n}$ then
                answer "$n$ is prime" and quit
          else
                $b = b^2 \bmod n$
7.  answer "$n$ is composite"
```

algorithm answers "n is composite," we conclude that

$$a^{2^i m} \not\equiv -1 \pmod{n}$$

for $0 \le i \le k - 1$.

Now, using the assumption that n is prime, Fermat's theorem (Corollary 4.6) tells us that

$$a^{2^k m} \equiv 1 \pmod{n}$$

since $n - 1 = 2^k m$. Then $a^{2^{k-1} m}$ is a square root of 1 modulo n. Since n is prime, there are only two square roots of 1 modulo n, namely, $\pm 1 \bmod n$. This can be seen as follows: x is a square root of 1 modulo n if and only if

$$n \mid (x - 1)(x + 1).$$

Since n is prime, either $n \mid (x - 1)$ (i.e., $x \equiv 1 \pmod{n}$) or $n \mid (x + 1)$ (i.e., $x \equiv -1 \pmod{n}$).

We have that

$$a^{2^{k-1} m} \not\equiv -1 \pmod{n},$$

so it follows that

$$a^{2^{k-1} m} \equiv 1 \pmod{n}.$$

Then $a^{2^{k-2}m}$ must be a square root of 1. By the same argument,

$$a^{2^{k-2}m} \equiv 1 \pmod{n}.$$

Repeating this argument, we eventually obtain

$$a^m \equiv 1 \pmod{n},$$

which is a contradiction, since the algorithm would have answered "n is prime" in this case. ∎

It remains to consider the error probability of the Miller-Rabin algorithm. Although we will not prove it here, the error probability can be shown to be, at most, $1/4$.

4.6 Attacks On RSA

In this section, we address the question: are there possible attacks on RSA other than factoring n? Let us first observe that it is sufficient for the cryptanalyst to compute $\phi(n)$. For, if n and $\phi(n)$ are known, and n is the product of two primes p, q, then n can be easily factored, by solving the two equations

$$n = pq$$
$$\phi(n) = (p-1)(q-1)$$

for the two "unknowns" p and q. If we substitute $q = n/p$ into the second equation, we obtain a quadratic equation in the unknown value p:

$$p^2 - (n - \phi(n) + 1)p + n = 0.$$

The two roots of this equation will be p and q, the factors of n. Hence, if a cryptanalyst can learn the value of $\phi(n)$, then he can factor n and break the system. In other words, computing $\phi(n)$ is no easier than factoring n.

Here is an example to illustrate.

Example 4.9
Suppose the cryptanalyst has learned that $n = 84773093$ and $\phi(n) = 84754668$. This information gives rise to the following quadratic equation:

$$p^2 - 18426p + 84773093 = 0.$$

This can be solved by the quadratic formula, yielding the two roots 9539 and 8887. These are the two factors of n. □

4.6.1 The Decryption Exponent

We will now prove the very interesting result that any algorithm which computes the decryption exponent a can be used as a subroutine (or *oracle*) in a probabilistic algorithm that factors n. So we can say that computing a is no easier than factoring n. However, this does not rule out the possibility of breaking the cryptosystem without computing a.

Notice that this result is of much more than theoretical interest. It tells us that if a is revealed, then the value n is also compromised. If this happens, it is not sufficient for Bob to choose a new encryption exponent; he must also choose a new modulus n.

The algorithm we are going to describe is a probabilistic algorithm of the Las Vegas type. Here is the definition:

DEFINITION 4.5 Suppose $0 \leq \epsilon < 1$ is a real number. A ***Las Vegas*** *algorithm is a probabilistic algorithm such that, for any problem instance I, the algorithm may fail to give an answer with some probability ϵ (i.e., it can terminate with the message "no answer"). However, if the algorithm does return an answer, then the answer must be correct.*

REMARK A Las Vegas algorithm may not give an answer, but any answer it gives is correct. In contrast, a Monte Carlo algorithm always gives an answer, but the answer may be incorrect. ▮

If we have a Las Vegas algorithm to solve a problem, then we simply run the algorithm over and over again until it finds an answer. The probability that the algorithm will return "no answer" m times in succession is ϵ^m. The average (i.e., expected) number of times the algorithm must be run in order to obtain an answer is in fact $1/\epsilon$ (see the exercises).

Suppose that **A** is a hypothetical algorithm that computes the decryption exponent a from b. We will describe a Las Vegas algorithm that uses **A** as an oracle. This algorithm will factor n with probability at least $1/2$. Hence, if the algorithm is run m times, then n will be factored with probability at least $1 - 1/2^m$.

The algorithm is based on certain facts concerning square roots of 1 modulo n, where $n = pq$ is the product of two distinct odd primes. Recall that the congruence $x^2 \equiv 1 \pmod p$ has two solutions modulo p, namely $x = \pm 1 \bmod p$. Similarly, the congruence $x^2 \equiv 1 \pmod q$ has two solutions, namely $x = \pm 1 \bmod q$.

Now, since $x^2 \equiv 1 \pmod n$ if and only if $x^2 \equiv 1 \pmod p$ and $x^2 \equiv 1 \pmod q$, it follows that $x^2 \equiv 1 \pmod n$ if and only if $x = \pm 1 \bmod p$ and $x = \pm 1 \bmod q$. Hence, there are four square roots of 1 modulo n, and they can be found using the Chinese remainder theorem. Two of these solutions are $x = \pm 1 \bmod n$; these are called the *trivial* square roots of 1 modulo p. The other two square roots are called *non-trivial*, and they are negatives of each other modulo n.

Here is a small example to illustrate.

Example 4.10
Suppose $n = 403 = 13 \times 31$. The four square roots of 1 modulo 403 are $1, 92, 311$ and 402. The square root 92 is obtained by solving the system $x \equiv 1 \pmod{13}$, $x \equiv -1 \pmod{31}$ using the Chinese remainder theorem. Having found this non-trivial square root, the other non-trivial square root must be $403 - 92 = 311$. It is the solution to the system $x \equiv -1 \pmod{13}$, $x \equiv 1 \pmod{31}$. □

Suppose x is a non-trivial square root of 1 modulo n. Then we have

$$n \mid (x-1)(x+1),$$

but n divides neither factor on the right side. It follows that $\gcd(x+1, n) = p$ or q (and similarly, $\gcd(x-1, n) = p$ or q). Of course, a greatest common divisor can be computed using the Euclidean algorithm, without knowing the factorization of n. Hence, knowledge of a non-trivial square root of 1 modulo n yields the factorization of n with only a polynomial amount of computation. This important fact is the basis of many results in cryptography.

In Example 4.10 above, $\gcd(93, 403) = 31$ and $\gcd(312, 403) = 13$.

In Figure 4.10, we present an algorithm which, using the hypothetical algorithm **A** as a subroutine, attempts to factor n by finding a non-trivial square root of 1 modulo n. (Recall that **A** computes the decryption exponent a corresponding to the encryption exponent b.) We first do an example to illustrate the application of this algorithm.

Example 4.11
Suppose $n = 89855713$, $b = 34986517$ and $a = 82330933$, and the random value $w = 5$. We have

$$ab - 1 = 2^3 \times 360059073378795.$$

In step 6, $v = 85877701$, and in step 10, $v = 1$. In step 12, we compute

$$\gcd(85877702, n) = 9103.$$

This is one factor of n; the other is $n/9103 = 9871$. □

Let's now proceed to the analysis of the algorithm. First, observe that if we are lucky enough to choose w to be a multiple of p or q, then we can factor n immediately. This is detected in step 2. If w is relatively prime to n, then we compute $w^r, w^{2r}, w^{4r}, \ldots$, by successive squaring, until

$$w^{2^t r} \equiv 1 \pmod{n}$$

FIGURE 4.10
Factoring algorithm, given the decryption exponent a

```
1.  choose $w$ at random such that $1 \leq w \leq n-1$
2.  compute $x = \gcd(w, n)$
3.  if $1 < x < n$ then quit (success: $x = p$ or $x = q$)
4.  compute $a = \mathbf{A}(b)$
5.  write $ab - 1 = 2^s r$, $r$ odd
6.  compute $v = w^r \bmod n$
7.  if $v \equiv 1 \pmod{n}$ then quit (failure)
8.  while $v \not\equiv 1 \pmod{n}$ do
9.           $v_0 = v$
10.          $v = v^2 \bmod n$
11. if $v_0 \equiv -1 \pmod{n}$ then
             quit (failure)
    else
             compute $x = \gcd(v_0 + 1, n)$ (success: $x = p$ or $x = q$)
```

for some t. Since

$$ab - 1 = 2^s r \equiv 0 \pmod{\phi(n)},$$

we know that $w^{2^s r} \equiv 1 \pmod{n}$. Hence, the **while** loop terminates after at most s iterations. At the end of the **while** loop, we have found a value v_0 such that $v_0^2 \equiv 1 \pmod{n}$ but $v_0 \not\equiv 1 \pmod{n}$. If $v_0 \equiv -1 \pmod{n}$, then the algorithm fails; otherwise, v_0 is a non-trivial square root of 1 modulo n and we are able to factor n (step 12).

The main task facing us now is to prove that the algorithm succeeds with probability at least $1/2$. There are two ways in which the algorithm can fail to factor n:

1. $w^r \equiv 1 \pmod{n}$ (step 7)
2. $w^{2^t r} \equiv -1 \pmod{n}$ for some t, $0 \leq t \leq s-1$ (step 11)

We have $s+1$ congruences to consider. If a random value w is a solution to at least one of these $s+1$ congruences, then it is a "bad" choice, and the algorithm fails. So we proceed by counting the number of solutions to each of these congruences.

First, consider the congruence $w^r \equiv 1 \pmod{n}$. The way to analyze a congruence such as this is to consider solutions modulo p and modulo q separately, and then combine them using the Chinese remainder theorem. Observe that $x \equiv 1 \pmod{n}$ if and only if $x \equiv 1 \pmod{p}$ and $x \equiv 1 \pmod{q}$.

So, we first consider $w^r \equiv 1 \pmod{p}$. Since p is prime, $\mathbb{Z}_p{}^*$ is a cyclic group by Theorem 4.7. Let g be a primitive element modulo p. We can write $w = g^u$ for a unique integer u, $0 \leq u \leq p-2$. Then we have

$$\begin{aligned} w^r &\equiv 1 \pmod{p} \\ g^{ur} &\equiv 1 \pmod{p} \\ (p-1) &\mid ur. \end{aligned}$$

Let us write

$$p - 1 = 2^i p_1$$

where p_1 is odd, and

$$q - 1 = 2^j q_1$$

where q_1 is odd. Since

$$\phi(n) = (p-1)(q-1) \mid (ab-1) = 2^s r,$$

we have that

$$2^{i+j} p_1 q_1 \mid 2^s r.$$

Hence

$$i + j \leq s$$

and

$$p_1 q_1 \mid r.$$

Now, the condition $(p-1) \mid ur$ becomes $2^i p_1 \mid ur$. Since $p_1 \mid r$ and r is odd, it is necessary and sufficient that $2^i \mid u$. Hence, $u = k2^i$, $0 \leq k \leq p_1 - 1$, and the number of solutions to the congruence $w^r \equiv 1 \pmod{p}$ is p_1.

By an identical argument, the congruence $w^r \equiv 1 \pmod{q}$ has exactly q_1 solutions. We can combine any solution modulo p with any solution modulo q to obtain a unique solution modulo n, using the Chinese remainder theorem. Consequently, the number of solutions to the congruence $w^r \equiv 1 \pmod{n}$ is $p_1 q_1$.

The next step is to consider a congruence $w^{2^t r} \equiv -1 \pmod{n}$ for a fixed value t (where $0 \leq t \leq s-1$). Again, we first look at the congruence modulo p and then modulo q (note that $w^{2^t r} \equiv -1 \pmod{n}$ if and only if $w^{2^t r} \equiv -1 \pmod{p}$ and $w^{2^t r} \equiv -1 \pmod{q}$). First, consider $w^{2^t r} \equiv -1 \pmod{p}$. Writing $w = g^u$, as above, we get

$$g^{u 2^t r} \equiv -1 \pmod{p}.$$

Since $g^{(p-1)/2} \equiv -1 \pmod{p}$, we have that

$$\begin{aligned} u2^t r &\equiv \frac{p-1}{2} \pmod{p-1} \\ (p-1) &\mid \left(u2^t r - \frac{p-1}{2}\right) \\ 2(p-1) &\mid (u2^{t+1}r - (p-1)). \end{aligned}$$

Since $p - 1 = 2^i p_1$, we get

$$2^{i+1}p_1 \mid (u2^{t+1}r - 2^i p_1).$$

Taking out a common factor of p_1, this becomes

$$2^{i+1} \mid \left(\frac{u2^{t+1}r}{p_1} - 2^i\right).$$

Now, if $t \geq i$, then there can be no solutions since $2^{i+1} \mid 2^{t+1}$ but $2^{i+1} \not| 2^i$. On the other hand, if $t \leq i-1$, then u is a solution if and only if u is an odd multiple of 2^{i-t-1} (note that r/p_1 is an odd integer). So, the number of solutions in this case is

$$\frac{p-1}{2^{i-t-1}} \times \frac{1}{2} = 2^t p_1.$$

By similar reasoning, the congruence $w^{2^t r} \equiv -1 \pmod{q}$ has no solutions if $t \geq j$, and $2^t q_1$ solutions if $t \leq j-1$. Using the Chinese remainder theorem, we see that the number of solutions of $w^{2^t r} \equiv -1 \pmod{n}$ is

$$\begin{array}{ll} 0 & \text{if } t \geq \min\{i,j\} \\ 2^{2t}p_1 q_1 & \text{if } t \leq \min\{i,j\} - 1. \end{array}$$

Now, t can range from 0 to $s-1$. Without loss of generality, suppose $i \leq j$; then the number of solutions is 0 if $t \geq i$. The total number of "bad" choices for w is *at most*

$$\begin{aligned} p_1q_1 + p_1q_1(1 + 2^2 + 2^4 + \ldots + 2^{2i-2}) &= p_1q_1\left(1 + \frac{2^{2i}-1}{3}\right) \\ &= p_1q_1\left(\frac{2}{3} + \frac{2^{2i}}{3}\right). \end{aligned}$$

Recall that $p - 1 = 2^i p_1$ and $q - 1 = 2^j q_1$. Now, $j \geq i \geq 1$, so $p_1q_1 < n/4$. We also have that

$$2^{2i}p_1q_1 \leq 2^{i+j}p_1q_1 = (p-1)(q-1) < n.$$

Hence, we obtain

$$p_1q_1\left(\frac{2}{3} + \frac{2^{2i}}{3}\right) < \frac{n}{6} + \frac{n}{3}$$

$$= \frac{n}{2}.$$

Since at most $(n-1)/2$ choices for w are "bad," it follows that at least $(n-1)/2$ choices are "good" and hence the probability of success of the algorithm is at least $1/2$.

4.6.2 Partial Information Concerning Plaintext Bits

The other result we will discuss concerns partial information about the plaintext that might be "leaked" by an **RSA** encryption. Two examples of partial information that we consider are the following:

1. given $y = e_K(x)$, compute *parity*(y), where *parity*(y) denotes the low-order bit of x
2. given $y = e_K(x)$, compute *half*(y), where *half*$(y) = 0$ if $0 \leq x < n/2$ and *half*$(y) = 1$ if $n/2 < x \leq n-1$.

We will prove that, given $y = e_K(x)$, any algorithm that computes *parity*(y) or *half*(y) can be used as an oracle to construct an algorithm that computes the plaintext x. What this means is that, given a ciphertext, computing the low-order bit of the plaintext is polynomially equivalent to determining the whole plaintext!

First, we prove that computing *parity*(y) is polynomially equivalent to computing *half*(y). This follows from the following two easily proved identities (see the exercises):

$$half(y) = parity(y \times e_K(2) \bmod n) \tag{4.1}$$

$$parity(y) = half(y \times e_K(2^{-1}) \bmod n) \tag{4.2}$$

and from the multiplicative rule $e_K(x_1)e_K(x_2) = e_K(x_1x_2)$.

We will show how to compute $x = d_K(y)$, given a hypothetical algorithm (oracle) which computes *half*(y). The algorithm is presented in Figure 4.11. In steps 2–4, we compute

$$y_i = half(y \times (e_K(2))^i) = half(e_K(x \times 2^i)),$$

for $0 \leq i \leq \log_2 n$. We observe that

$$half(e_K(x)) = 0 \Leftrightarrow x \in \left[0, \frac{n}{2}\right)$$

$$half(e_K(2x)) = 0 \Leftrightarrow x \in \left[0, \frac{n}{4}\right) \cup \left[\frac{n}{2}, \frac{3n}{4}\right)$$

$$half(e_K(4x)) = 0 \Leftrightarrow x \in \left[0, \frac{n}{8}\right) \cup \left[\frac{n}{4}, \frac{3n}{8}\right) \cup \left[\frac{n}{2}, \frac{5n}{8}\right) \cup \left[\frac{3n}{4}, \frac{7n}{8}\right),$$

and so on. Hence, we can find x by a binary search technique, which is done in steps 7–11. Here is a small example to illustrate.

FIGURE 4.11
Decrypting RSA ciphertext, given an oracle for computing *half*(y)

```
1.  denote $k = \lfloor \log_2 n \rfloor$
2.  for $i = 0$ to $k$ do
3.      $y_i = half(y)$
4.      $y = (y \times e_K(2)) \bmod n$
5.  $lo = 0$
6.  $hi = n$
7.  for $i = 0$ to $k$ do
8.      $mid = (hi + lo)/2$
9.      if $y_i = 1$ then
            $lo = mid$
        else
            $hi = mid$
10. $x = \lfloor hi \rfloor$
```

Example 4.12
Suppose $n = 1457$, $b = 779$, and we have a ciphertext $y = 722$. $e_K(2)$ is computed to be 946. Suppose, using our oracle for *half*, that we obtain the following values y_i in step 3 of the algorithm:

i	0	1	2	3	4	5	6	7	8	9	10
y_i	1	0	1	0	1	1	1	1	1	0	0

Then the binary search proceeds as shown in Figure 4.12. Hence, the plaintext is $x = \lfloor 999.55 \rfloor = 999$. □

4.7 The Rabin Cryptosystem

In this section, we describe the **Rabin Cryptosystem**, which is computationally secure against a chosen-plaintext attack provided that the modulus $n = pq$ cannot be factored. The system is described in Figure 4.13.

We will show that the encryption function e_K is not an injection, so decryption cannot be done in an unambiguous fashion. In fact, there are four possible

FIGURE 4.12
Binary search for RSA decryption

i	lo	mid	hi
0	0.00	728.50	1457.00
1	728.50	1092.75	1457.00
2	728.50	910.62	1092.75
3	910.62	1001.69	1092.75
4	910.62	956.16	1001.69
5	956.16	978.92	1001.69
6	978.92	990.30	1001.69
7	990.30	996.00	1001.69
8	996.00	998.84	1001.69
9	998.84	1000.26	1001.69
10	998.84	999.55	1000.26
	998.84	999.55	999.55

FIGURE 4.13
Rabin Cryptosystem

Let n be the product of two distinct primes p and q, $p, q \equiv 3 \pmod 4$. Let $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$, and define

$$\mathcal{K} = \{(n, p, q, B) : 0 \leq B \leq n - 1\}.$$

For $K = (n, p, q, B)$, define

$$e_K(x) = x(x + B) \bmod n$$

and

$$d_K(y) = \sqrt{\frac{B^2}{4} + y} - \frac{B}{2}.$$

The values n and B are public, while p and q are secret.

plaintexts that could be the encryption of any given ciphertext. More precisely, let ω be one of the four square roots of 1 modulo n. Let $x \in \mathbb{Z}_n$. Then, we can verify the following equations:

$$\begin{aligned} e_K\left(\omega\left(x+\frac{B}{2}\right)-\frac{B}{2}\right) &= \left(\omega\left(x+\frac{B}{2}\right)-\frac{B}{2}\right)\left(\omega\left(x+\frac{B}{2}\right)+\frac{B}{2}\right) \\ &= \omega^2\left(x+\frac{B}{2}\right)^2-\left(\frac{B}{2}\right)^2 \\ &= x^2+Bx+\frac{B^2}{4}-\frac{B^2}{4} \\ &= x^2+Bx \\ &= e_K(x). \end{aligned}$$

(Note that all arithmetic is being done in $\mathbb{Z}_n$, and division by 2 and 4 is the same as multiplication by 2^{-1} and 4^{-1} modulo n, respectively.)

The four plaintexts that encrypt to $e_K(x)$ are x, $-x-B$, $\omega(x+B/2)$ and $-\omega(x+B/2)$, where ω is a non-trivial square root of 1 modulo n. In general, there will be no way for Bob to distinguish which of these four possible plaintexts is the "right" plaintext, unless the plaintext contains sufficient redundancy to eliminate three of these four possible values.

Let us look at the decryption problem from Bob's point of view. He is given a ciphertext y and wants to determine x such that

$$x^2+bx \equiv y \pmod{n}.$$

This is a quadratic equation in the unknown x. We can eliminate the linear term by making the substitution $x_1 = x + B/2$, or equivalently, $x = x_1 - B/2$. Then the equation becomes

$$x_1^2 - Bx_1 + \frac{B^2}{4} + Bx_1 - \frac{B^2}{2} - y \equiv 0 \pmod{n},$$

or

$$x_1^2 \equiv \frac{B^2}{4} + y \pmod{n}.$$

If we define $C = B^2/4 + y$, then we can rewrite the congruence as

$$x_1^2 \equiv C \pmod{n}.$$

So, decryption reduces to extracting square roots modulo n. This is equivalent to solving the two congruences

$$x_1^2 \equiv C \pmod{p}$$

and

$$x_1^2 \equiv C \pmod{q}.$$

(There are two square roots of C modulo p and two square roots modulo q. Using the Chinese remainder theorem, these can be combined to yield four solutions modulo n.) We can use Euler's criterion to determine if C is a quadratic residue modulo p (and modulo q). In fact, C will be a quadratic residue modulo p and modulo q if encryption was performed correctly. But Euler's criterion does not help us find the square roots of C; it yields only an answer "yes" or "no."

When $p \equiv 3 \pmod 4$, there is a simple formula to compute square roots of quadratic residues modulo p. Suppose C is a quadratic residue and $p \equiv 3 \pmod 4$. Then we have that

$$\begin{aligned}(\pm C^{(p+1)/4})^2 &\equiv C^{(p+1)/2} \pmod p\\ &\equiv C^{(p-1)/2} C \pmod p\\ &\equiv C \pmod p.\end{aligned}$$

Here we again make use of Euler's criterion, which says that if C is a quadratic residue modulo p, then $C^{(p-1)/2} \equiv 1 \pmod p$. Hence, the two square roots of C modulo p are $\pm C^{(p+1)/4} \bmod p$. In a similar fashion, the two square roots of C modulo q are $\pm C^{(q+1)/4} \bmod q$. It is then straightforward to obtain the four square roots x_1 of C modulo n using the Chinese remainder theorem.

REMARK It is interesting that for $p \equiv 1 \pmod 4$ there is no known polynomial-time deterministic algorithm to compute square roots of quadratic residues modulo p. There is a polynomial-time Las Vegas algorithm, however. ∎

Once we have determined the four possible values for x_1, we compute x from the equation $x = x_1 - B/2$ to get the four possible plaintexts. This yields the decryption formula

$$d_K(y) = \sqrt{\frac{B^2}{4} + y} - \frac{B}{2}.$$

Example 4.13

Let's illustrate the encryption and decryption procedures for the **Rabin Cryptosystem** with a toy example. Suppose $n = 77 = 7 \times 11$ and $B = 9$. Then the encryption function is

$$e_K(x) = x^2 + 9x \bmod 77$$

and the decryption function is

$$d_K(y) = \sqrt{1+y} - 43 \bmod 77.$$

Suppose Bob wants to decrypt the ciphertext $y = 22$. It is first necessary to find the square roots of 23 modulo 7 and modulo 11. Since 7 and 11 are both congruent

Previous Chapter Previous Page Home Next page Next Chapter

FIGURE 4.14
Factoring a Rabin modulus, given a decryption oracle

```
1.  choose a random $r$, $1 \leq r \leq n-1$
2.  compute $y = r^2 - B^2/4 \bmod n$
3.  call $\mathbf{A}(y)$, obtaining a decryption $x$
4.  compute $x_1 = x + B/2$
5.  if $x_1 \equiv \pm r \pmod{n}$ then
        quit (failure)
    else
        $\gcd(x_1 + r, n) = p$ or $q$ (success)
```

to 3 modulo 4, we use our formula:

$$23^{(7+1)/4} \equiv 2^2 \equiv 4 \bmod 7$$

and

$$23^{(11+1)/4} \equiv 1^3 \equiv 1 \bmod 11.$$

Using the Chinese remainder theorem, we compute the four square roots of 23 modulo 77 to be $\pm 10, \pm 32 \bmod 77$. Finally, the four possible plaintexts are:

$$10 - 43 \bmod 77 = 44$$

$$67 - 43 \bmod 77 = 24$$

$$32 - 43 \bmod 77 = 66$$

$$45 - 43 \bmod 77 = 2.$$

It can be verified that each of these plaintexts encrypts to the ciphertext 22. □

We now discuss the security of the **Rabin Cryptosystem**. We will prove that any hypothetical decryption algorithm **A** can be used as an oracle in a Las Vegas algorithm that factors the modulus n with probability at least $1/2$. This algorithm is depicted in Figure 4.14.

There are several points of explanation needed. First, observe that

$$y = e_K\left(r - \frac{B}{2}\right),$$

so a value x will be returned in step 3. Next, we look at step 4 and note that $x_1^2 \equiv r^2 \pmod{n}$. It follows that $x_1 \equiv \pm r \pmod{n}$ or $x_1 \equiv \pm\omega r \pmod{n}$, where ω is one of the non-trivial square roots of 1 modulo n. In the second case, we have

$$n \mid (x_1 - r)(x_1 + r),$$

but n does not divide either factor on the right side. Hence, computation of $\gcd(x_1 + r, n)$ (or $\gcd(x_1 - r, n)$) must yield either p or q, and the factorization of n is accomplished.

Let's compute the probability of success of this algorithm, over all $n - 1$ choices for the random value r. For two non-zero residues r_1 and r_2, define

$$r_1 \sim r_2 \Leftrightarrow r_1^2 \equiv r_2^2 \pmod{n}.$$

It is easy to see that $r \sim r$ for all r; $r_1 \sim r_2$ implies $r_2 \sim r_1$; and $r_1 \sim r_2$ and $r_2 \sim r_3$ together imply $r_1 \sim r_3$. This says that the relation $\sim$ is an *equivalence relation*. The equivalence classes of $\mathbb{Z}_n \backslash \{0\}$ all have cardinality four: the equivalence class containing r is the set

$$[r] = \{\pm r, \pm\omega r \bmod n\},$$

where ω is a non-trivial square root of 1 modulo n.

In the algorithm presented in Figure 4.14, any two values r in the same equivalence class will yield the same value y. Now consider the value x returned by the oracle **A** when given y. We have

$$[y] = \{\pm y, \pm\omega y\}.$$

If $r = \pm y$, then the algorithm fails; while it succeeds if $r = \pm\omega y$. Since r is chosen at random, it is equally likely to be any of these four possible values. We conclude that the probability of success of the algorithm is $1/2$.

It is interesting that the **Rabin Cryptosystem** is provably secure against a chosen plaintext attack. However, the system is completely insecure against a chosen ciphertext attack. In fact the algorithm in Figure 4.14, that we used to prove security against a chosen plaintext attack, also can be used to break the **Rabin Cryptosystem** in a chosen ciphertext attack! In the chosen ciphertext attack, the oracle **A** is replaced by Bob's decryption algorithm.

4.8 Factoring Algorithms

There is a huge amount of literature on factoring algorithms, and a careful treatment would require more pages than we have in this book. We will just try to give a brief overview here, including an informal discussion of the best current factoring algorithms and their use in practice. The three algorithms that are most effective

FIGURE 4.15
The $p - 1$ factoring algorithm

```
   Input: $n$ and $B$
1. $a = 2$
2. **for** $j = 2$ **to** $B$ **do**
       $a = a^j \bmod n$
3. $d = \gcd(a - 1, n)$
4. $1 < d < n$ **then**
       $d$ is a factor of $n$ (success)
   **else**
       no factor of $n$ is found (failure)
```

on very large numbers are the quadratic sieve, the elliptic curve algorithm and the number field sieve. Other well-known algorithms that were precursors include Pollard's rho-method and $p-1$ algorithm, Williams' $p+1$ algorithm, the continued fraction algorithm, and of course, trial division.

Throughout this section, we suppose that the integer n that we wish to factor is odd. *Trial division* consists of dividing n by every odd integer up to $\lfloor\sqrt{n}\rfloor$. If $n < 10^{12}$, say, this is a perfectly reasonable factorization method, but for larger n we generally need to use more sophisticated techniques.

4.8.1 The $p - 1$ Method

As an example of a simple algorithm that can sometimes be applied to larger integers, we describe Pollard's $p - 1$ algorithm, which dates from 1974. This algorithm, presented in Figure 4.15, has two inputs: the (odd) integer n to be factored, and a "bound" B. Here is what is taking place in the $p - 1$ algorithm: Suppose p is a prime divisor of n, and $q \leq B$ for every prime power $q \mid (p - 1)$. Then it must be the case that

$$(p - 1) \mid B!$$

At the end of the **for** loop (step 2),

$$a \equiv 2^{B!} \pmod{n},$$

so

$$a \equiv 2^{B!} \pmod{p}$$

since $p \mid n$. Now,

$$2^{p-1} \equiv 1 \pmod p$$

by Fermat's theorem. Since $(p-1) \mid B!$, we have that

$$a \equiv 1 \pmod p$$

(in step 3). Thus, in step 4,

$$p \mid (a-1)$$

and

$$p \mid n,$$

so

$$p \mid d = \gcd(a-1, n).$$

The integer d will be a non-trivial divisor of n (unless $a = 1$ in step 3). Having found a non-trivial factor d, we would then proceed to attempt to factor d and n/d if they are composite.

Here is an example to illustrate.

Example 4.14
Suppose $n = 15770708441$. If we apply the $p-1$ algorithm with $B = 180$, then we find that $a = 11620221425$ in step 3, and d is computed to be 135979. In fact, the complete factorization of n into primes is

$$15770708441 = 135979 \times 115979.$$

in this case, the factorization succeeds because 135978 has only "small" prime factors:

$$135978 = 2 \times 3 \times 131 \times 173.$$

Hence, by taking $B \geq 173$, it will be the case that $135978 \mid B!$, as desired. ▯

In the algorithm, there are $B-1$ modular exponentiations, each requring at most $2 \log_2 B$ modular multiplications using square-and-multiply. The gcd computation can be done in time $O((\log n)^3)$ using the Euclidean algorithm. Hence, the complexity of the algorithm is $O(B \log B (\log n)^2 + (\log n)^3)$. If B is $O((\log n)^i)$ for some fixed integer i, then the algorithm is indeed a polynomial-time algorithm; however, for such a choice of B the probability of success will be very small. On the other hand, if we increase the size of B drastically, say to $\sqrt{n}$, then the algorithm will be successful, but it will be no faster than trial division.

Thus, the drawback of this method is that it requires n to have a prime factor p such that $p-1$ has only "small" prime factors. It would be very easy to construct an **RSA** modulus $n = pq$ which would resist factorization by this method. One would start by finding a large prime p_1 such that $p = 2p_1 + 1$ is also prime, and a large prime q_1 such that $q = 2q_1 + 1$ is also prime (using one of the Monte Carlo primality testing algorithms discussed in Section 4.5). Then the **RSA** modulus $n = pq$ will be resistant to factorization using the $p - 1$ method.

The more powerful elliptic curve algorithm, developed by Lenstra in the mid-1980's, is in fact a generalization of the $p - 1$ method. We will not discuss the theory at all here, but we do mention that the success of the elliptic curve method depends on the more likely situation that an integer "close to" p has only "small" prime factors. Whereas the $p - 1$ method depends on a relation that holds in the group $\mathbb{Z}_p$, the elliptic curve method involves groups defined on elliptic curves modulo p.

4.8.2 Dixon's Algorithm and the Quadratic Sieve

Dixon's algorithm is based on a very simple idea that we already saw in connection with the **Rabin Cryptosystem**. Namely, if we can find $x \not\equiv \pm y \pmod{n}$ such that $x^2 \equiv y^2 \pmod{n}$, then $\gcd(x - y, n)$ is a non-trivial factor of n.

The method uses a *factor base*, which is a set $\mathcal{B}$ of "small" primes. We first obtain several integers x such that all the prime factors of $x^2 \bmod n$ occur in the factor base $\mathcal{B}$. (How this is done will be discussed a bit later.) The idea is to then take the product of several of these x's in such a way that every prime in the factor base is used an even number of times. This then gives us a congruence of the desired type $x^2 \equiv y^2 \pmod{n}$, which (we hope) will lead to a factorization of n.

We illustrate with a carefully contrived example.

Example 4.15

Suppose $n = 15770708441$ (this was the same n that we used in Example 4.14). Let $\mathcal{B} = \{2, 3, 5, 7, 11, 13\}$. Consider the three congruences:

$$8340934156^2 \equiv 3 \times 7 \pmod{n}$$

$$12044942944^2 \equiv 2 \times 7 \times 13 \pmod{n}$$

$$2773700011^2 \equiv 2 \times 3 \times 13 \pmod{n}.$$

If we take the product of these three congruences, then we have

$$(8340934156 \times 12044942944 \times 2773700011)^2 \equiv (2 \times 3 \times 7 \times 13)^2 \pmod{n}.$$

Reducing the expressions inside the parentheses modulo n, we have

$$9503435785^2 \equiv 546^2 \pmod{n}.$$

Then we compute

$$\gcd(9503435785 - 546, 15770708441) = 115759,$$

finding the factor 115759 of n. ▯

Suppose $\mathcal{B} = \{p_1, \dots, p_B\}$ is the factor base. Let C be slightly larger than B (say $C = B + 10$), and suppose we have obtained C congruences:

$$x_j{}^2 \equiv p_1{}^{\alpha_{1j}} \times p_2{}^{\alpha_{2j}} \dots \times p_B{}^{\alpha_{Bj}} \pmod{n},$$

for $1 \leq j \leq C$. For each j, consider the vector

$$a_j = (\alpha_{1j} \bmod 2, \dots, \alpha_{Bj} \bmod 2) \in (\mathbb{Z}_2)^B.$$

If we can find a subset of the a_j's that sum modulo 2 to the vector $(0, \dots, 0)$, then the product of the corresponding x_j's will use each factor in $\mathcal{B}$ an even number of times.

We illustrate by returning to Example 4.15, where there exists a dependence even though $C < B$ in this case.

Example 4.15 ***(Cont.)***
The three vectors a_1, a_2, a_3 are as follows:

$$\begin{aligned} a_1 &= (0,1,0,1,0,0) \\ a_2 &= (1,0,0,1,0,1) \\ a_3 &= (1,1,0,0,0,1). \end{aligned}$$

It is easy to see that

$$a_1 + a_2 + a_3 = (0,0,0,0,0,0) \bmod 2.$$

This gives rise to the congruence we saw earlier that successfully factored n. ▯

Observe that finding a subset of the C vectors $a_1, \dots, a_C$ that sums modulo 2 to the all-zero vector is nothing more than finding a linear dependence (over $\mathbb{Z}_2$) of these vectors. Provided $C > B$, such a linear dependence must exist, and it can be found easily using the standard method of Gaussian elimination. The reason why we take $C > B + 1$ is that there is no guarantee that any given congruence will yield the factorization of n. Approximately 50% of the time it will turn out that $x \equiv \pm y \pmod{n}$. But if $C > B+1$, then we can obtain several such congruences (arising from different linear dependencies among the a_j's). Hopefully, at least one of the resulting congruences will yield the factorization.

It remains to discuss how we obtain integers x_j such that the values $x_j{}^2 \bmod n$ factor completely over the factor base $\mathcal{B}$. There are several methods of doing this. One common approach is the Quadratic Sieve due to Pomerance, which uses integers of the form $x_j = j + \lfloor\sqrt{n}\rfloor$, $j = 1, 2, \dots$. The name "quadratic sieve" comes from a sieving procedure (which we will not describe here) that is used to determine those x_j's that factor over $\mathcal{B}$.

There is, of course, a trade-off here: if $B = |\mathcal{B}|$ is large, then it is more likely that an integer x_j factors over $\mathcal{B}$. But the larger B is, the more congruences we need to accumulate before we are able to find a dependence relation. The optimal choice for B is approximately

$$\sqrt{e^{\sqrt{\ln n \ln \ln n}}},$$

and this leads to an expected running time of

$$O\left(e^{(1+o(1))\sqrt{\ln n \ln \ln n}}\right).$$

The number field sieve is a more recent factoring algorithm from the late 1980's. It also factors n by constructing a congruence $x^2 \equiv y^2 \pmod{n}$, but it does so by means of computations in rings of algebraic integers.

4.8.3 Factoring Algorithms in Practice

The asymptotic running times of the quadratic sieve, elliptic curve and number field sieve are as follows:

quadratic sieve	$O\left(e^{(1+o(1))\sqrt{\ln n \ln \ln n}}\right)$
elliptic curve	$O\left(e^{(1+o(1))\sqrt{2 \ln p \ln \ln p}}\right)$
number field sieve	$O\left(e^{(1.92+o(1))(\ln n)^{1/3}(\ln \ln n)^{2/3}}\right)$

The notation $o(1)$ denotes a function of n that approaches 0 as $n \to \infty$, and p denotes the smallest prime factor of n.

In the worst case, $p \approx \sqrt{n}$ and the asymptotic running times of the quadratic sieve and elliptic curve algorithms are essentially the same. But in such a situation, quadratic sieve generally outperforms elliptic curve. The elliptic curve method is more useful if the prime factors of n are of differing size. One very large number that was factored using the elliptic curve method was the Fermat number $2^{2^{11}} - 1$ in 1988 by Brent.

For factoring **RSA** moduli (where $n = pq$, p, q are prime, and p and q are roughly the same size), the quadratic sieve is currently the most successful algorithm. Some notable milestones have included the following factorizations. In 1983, the quadratic sieve successfully factored a 69-digit number that was a (composite) factor of $2^{251} - 1$ (this computation was done by Davis, Holdredge, and Simmons). Progress continued throughout the 1980's, and by 1989, numbers having up to 106 digits were factored by this method by Lenstra and Manasse, by distributing the computations to hundreds of widely separated workstations (they called this approach "factoring by electronic mail").

More recently, in April 1994, a 129-digit number known as **RSA**-129 was factored by Atkins, Graff, Lenstra, and Leyland using the quadratic sieve. (The numbers **RSA**-100, **RSA**-110, . . ., **RSA**-500 are a list of **RSA** moduli publicized on the Internet as "challenge" numbers for factoring algorithms. Each number **RSA**-d is a d-digit number that is the product of two primes of approximately the same length.) The factorization of **RSA**-129 required 5000 MIPS-years of computing time donated by over 600 researchers around the world.

The number field sieve is the most recent of the three algorithms. It seems to have great potential since its asymptotic running time is faster than either quadratic sieve or the elliptic curve. It is still in developmental stages, but people have speculated that number field sieve might prove to be faster for numbers having more than about 125–130 digits. In 1990, the number field sieve was used by Lenstra, Lenstra, Manasse, and Pollard to factor $2^{2^9} - 1$ into three primes having 7, 49 and 99 digits.

4.9 Notes and References

The idea of public-key cryptography was introduced by Diffie and Hellman in 1976. Although [DH76A] is the most cited reference, the conference paper [DH76] actually appeared a bit earlier. The **RSA Cryptosystem** was discovered by Rivest, Shamir and Adleman [RSA78]. The **Rabin Cryptosystem** was described in Rabin [RA79]; a similar provably secure system in which decryption is unambiguous was found by Williams [WI80]. For a general survey article on public-key cryptography, we recommend Diffie [DI92].

The Solovay-Strassen test was first described in [SS77]. The Miller-Rabin test was given in [MI76] and [RA80]. Our discussion of error probabilities is motivated by observations of Brassard and Bratley [BB88A, §8.6] (see also [BBCGP88]). The best current bounds on the error probability of the Miller-Rabin algorithm can be found in [DLP93].

The material in Section 4.6 is based on the treatment by Salomaa [SA90, pp. 143–154]. The factorization of n given the decryption exponent was proved in [DE84]; the results on partial information revealed by **RSA** is from [GMT82].

As mentioned earlier, there are many sources of information on factoring algorithms. Pomerance [PO90] is a good survey on factoring, and Lenstra and Lenstra [LL90] is a good article on number-theoretic algorithms in general. Bressoud [BR89] is an elementary textbook devoted to factoring and primality testing. Cryptography textbooks that emphasize number theory include Koblitz [KO87] and Kranakis [KR86]. Lenstra and Lenstra [LL93] is a monograph on the number field sieve.

Exercises 4.7–4.9 give some examples of protocol failures. For a nice article on this subject, see Moore [MO92].

Exercises

4.1 Use the Extended Euclidean algorithm to compute the following multiplicative inverses:

(a) $17^{-1} \bmod 101$

(b) $357^{-1} \bmod 1234$

(c) $3125^{-1} \bmod 9987$.

4.2 Solve the following system of congruences:

$$x \equiv 12 \pmod{25}$$
$$x \equiv 9 \pmod{26}$$
$$x \equiv 23 \pmod{27}.$$

4.3 Solve the following system of congruences:

$$13x \equiv 4 \pmod{99}$$
$$15x \equiv 56 \pmod{101}.$$

HINT First use the Extended Euclidean algorithm, and then apply the Chinese remainder theorem.

4.4 Here we investigate some properties of primitive roots.

(a) The integer 97 is prime. Prove that $x \neq 0$ is a primitive root modulo 97 if and only if $x^{32} \not\equiv 1 \pmod{97}$ and $x^{48} \not\equiv 1 \pmod{97}$.

(b) Use this method to find the smallest primitive root modulo 97.

(c) Suppose p is prime, and $p - 1$ has prime power factorization

$$p - 1 = \prod_{i=1}^{n} p_i^{e_i},$$

where the p_i's are distinct primes. Prove that $x \neq 0$ is a primitive root modulo p if and only if $x^{(p-1)/p_i} \not\equiv 1 \pmod{p}$ for $1 \leq i \leq n$.

4.5 Suppose that $n = pq$, where p and q are distinct odd primes and $ab \equiv 1 \pmod{(p-1)(q-1)}$. The **RSA** encryption operation is $e(x) = x^b \bmod n$ and the decryption operation is $d(y) = y^a \bmod n$. We proved that $d(e(x)) = x$ if $x \in \mathbb{Z}_n{}^*$. Prove that the same statement is true for any $x \in \mathbb{Z}_n$.

HINT Use the fact that $x_1 \equiv x_2 \pmod{pq}$ if and only if $x_1 \equiv x_2 \pmod{p}$ and $x_1 \equiv x_2 \pmod{q}$. This follows from the Chinese remainder theorem.

4.6 Two samples of **RSA** ciphertext are presented in Tables 4.1 and 4.2. Your task is to decrypt them. The public parameters of the system are $n = 18923$ and $b = 1261$ (for Table 4.1) and $n = 31313$ and $b = 4913$ (for Table 4.2). This can be accomplished as follows. First, factor n (which is easy because it is so small). Then compute the exponent a from $\phi(n)$, and, finally, decrypt the ciphertext. Use the square-and-multiply algorithm to exponentiate modulo n.

In order to translate the plaintext back into ordinary English text, you need to know how alphabetic characters are "encoded" as elements in $\mathbb{Z}_n$. Each element of

Previous Chapter Previous Page Home Next page Next Chapter

TABLE 4.1
RSA Ciphertext

12423	11524	7243	7459	14303	6127	10964	16399
9792	13629	14407	18817	18830	13556	3159	16647
5300	13951	81	8986	8007	13167	10022	17213
2264	961	17459	4101	2999	14569	17183	15827
12693	9553	18194	3830	2664	13998	12501	18873
12161	13071	16900	7233	8270	17086	9792	14266
13236	5300	13951	8850	12129	6091	18110	3332
15061	12347	7817	7946	11675	13924	13892	18031
2620	6276	8500	201	8850	11178	16477	10161
3533	13842	7537	12259	18110	44	2364	15570
3460	9886	8687	4481	11231	7547	11383	17910
12867	13203	5102	4742	5053	15407	2976	9330
12192	56	2471	15334	841	13995	17592	13297
2430	9741	11675	424	6686	738	13874	8168
7913	6246	14301	1144	9056	15967	7328	13203
796	195	9872	16979	15404	14130	9105	2001
9792	14251	1498	11296	1105	4502	16979	1105
56	4118	11302	5988	3363	15827	6928	4191
4277	10617	874	13211	11821	3090	18110	44
2364	15570	3460	9886	9988	3798	1158	9872
16979	15404	6127	9872	3652	14838	7437	2540
1367	2512	14407	5053	1521	297	10935	17137
2186	9433	13293	7555	13618	13000	6490	5310
18676	4782	11374	446	4165	11634	3846	14611
2364	6789	11634	4493	4063	4576	17955	7965
11748	14616	11453	17666	925	56	4118	18031
9522	14838	7437	3880	11476	8305	5102	2999
18628	14326	9175	9061	650	18110	8720	15404
2951	722	15334	841	15610	2443	11056	2186

$\mathbb{Z}_n$ represents three alphabetic characters as in the following examples:

$$\begin{array}{lcrcr} DOG & \rightarrow & 3 \times 26^2 + 14 \times 26 + 6 & = & 2398 \\ CAT & \rightarrow & 2 \times 26^2 + 0 \times 26 + 19 & = & 1371 \\ ZZZ & \rightarrow & 25 \times 26^2 + 25 \times 26 + 25 & = & 17575. \end{array}$$

You will have to invert this process as the final step in your program.

The first plaintext was taken from "The Diary of Samuel Marchbanks," by Robertson Davies, 1947, and the second was taken from "Lake Wobegon Days," by Garrison Keillor, 1985.

4.7 This exercise exhibits what is called a *protocol failure*. It provides an example where ciphertext can be decrypted by an opponent, without determining the key, if a cryptosystem is used in a careless way. (Since the opponent does not determine the key, it is not accurate to call it cryptanalysis.) The moral is that it is not sufficient to use a "secure" cryptosystem in order to guarantee "secure" communication.

Suppose Bob has an **RSA Cryptosystem** with a large modulus n for which the factorization cannot be found in a reasonable amount of time. Suppose Alice sends

Previous Chapter Previous Page Home Next page Next Chapter

TABLE 4.2
RSA Ciphertext

6340	8309	14010	8936	27358	25023	16481	25809
23614	7135	24996	30590	27570	26486	30388	9395
27584	14999	4517	12146	29421	26439	1606	17881
25774	7647	23901	7372	25774	18436	12056	13547
7908	8635	2149	1908	22076	7372	8686	1304
4082	11803	5314	107	7359	22470	7372	22827
15698	30317	4685	14696	30388	8671	29956	15705
1417	26905	25809	28347	26277	7897	20240	21519
12437	1108	27106	18743	24144	10685	25234	30155
23005	8267	9917	7994	9694	2149	10042	27705
15930	29748	8635	23645	11738	24591	20240	27212
27486	9741	2149	29329	2149	5501	14015	30155
18154	22319	27705	20321	23254	13624	3249	5443
2149	16975	16087	14600	27705	19386	7325	26277
19554	23614	7553	4734	8091	23973	14015	107
3183	17347	25234	4595	21498	6360	19837	8463
6000	31280	29413	2066	369	23204	8425	7792
25973	4477	30989					

a message to Bob by representing each alphabetic character as an integer between 0 and 25 (i.e., $A \leftrightarrow 0$, $B \leftrightarrow 1$, etc.), and then encrypting each residue modulo 26 as a separate plaintext character.

(a) Describe how Oscar can easily decrypt a message which is encrypted in this way.

(b) Illustrate this attack by decrypting the following ciphertext (which was encrypted using an **RSA Cryptosystem** with $n = 18721$ and $b = 25$) without factoring the modulus:

$$365, 0, 4845, 14930, 2608, 2608, 0.$$

4.8 This exercise illustrates another example of a protocol failure (due to Simmons) involving **RSA**; it is called the *common modulus* protocol failure. Suppose Bob has an **RSA Crpyotsystem** with modulus n and encryption exponent b_1, and Charlie has an **RSA Cryptosystem** with (the same) modulus n and decryption exponent b_2. Suppose also that $\gcd(b_1, b_2) = 1$. Now, consider the situation that arises if Alice encrypts the same plaintext x to send to both Bob and Charlie. Thus, she computes $y_1 = x^{b_1} \bmod n$ and $y_2 = x^{b_2} \bmod n$, and then she sends y_1 to Bob and y_2 to Charlie. Suppose Oscar intercepts y_1 and y_2, and performs the computations indicated in Figure 4.16.

(a) Prove that the value x_1 computed in step 3 of Figure 4.16 is in fact Alice's plaintext, x. Thus, Oscar can decrypt the message Alice sent, even though the cryptosystem may be "secure."

(b) Illustrate the attack by computing x by this method if $n = 18721, b_1 = 945$, $b_2 = 7717, y_1 = 12677$ and $y_2 = 14702$.

4.9 We give yet another protocol failure involving **RSA**. Suppose that three users in a network, say Bob, Bart and Bert, all have public encryption exponents $b = 3$.

FIGURE 4.16
RSA common modulus protocol failure

```
   Input: $n, b_1, b_2, y_1, y_2$
1. compute $c_1 = b_1^{-1} \bmod b_2$
2. compute $c_2 = (c_1 b_1 - 1)/b_2$
3. compute $x_1 = y_1^{c_1}(y_2^{c_2})^{-1} \bmod n$
```

Let their moduli be denoted by n_1, n_2, n_3. Now suppose Alice encrypts the same plaintext x to send to Bob, Bart and Bert. That is, Alice computes $y_i = x^3 \bmod n_i$, $1 \le i \le 3$. Describe how Oscar can compute x, given y_1, y_2 and y_3, without factoring any of the moduli.

4.10 A plaintext x is said to be *fixed* if $e_K(x) = x$. Show that, for the **RSA Cryptosystem**, the number of fixed plaintexts $x \in \mathbb{Z}_n{}^*$ is equal to $\gcd(b-1, p-1) \times \gcd(b-1, q-1)$.

HINT Consider the system of two congruences $e_K(x) \equiv x \pmod p$, $e_K(x) \equiv x \pmod q$.

4.11 Suppose A is a deterministic algorithm which is given as input an **RSA** modulus n and a ciphertext y. A will either decrypt y or return no answer. Supposing that there are $\epsilon \times n$ ciphertexts which A is able to decrypt, show how to use **A** as an oracle in a Las Vegas decryption algorithm having failure probability ϵ.

HINT Use the multiplicative property of **RSA** that $e_K(x_1)e_K(x_2) = e_K(x_1x_2)$, where all arithmetic operations are modulo n.

4.12 Write a program to evaluate Jacobi symbols using the four properties presented in Section 4.5. The program should not do any factoring, other than dividing out powers of two. Test your program by computing the following Jacobi symbols:

$$\left(\frac{610}{987}\right), \left(\frac{20964}{1987}\right), \left(\frac{1234567}{11111111}\right).$$

4.13 Write a program that computes the number of Euler pseudo-primes to the bases 837, 851, and 1189.

4.14 The purpose of this question is to prove that the error probability of the Solovay-Strassen primality test is at most 1/2. Let $\mathbb{Z}_n{}^*$ denote the group of units modulo n. Define

$$G(n) = \left\{a : a \in \mathbb{Z}_n{}^*, \left(\frac{a}{n}\right) \equiv a^{(n-1)/2} \pmod n\right\}.$$

(a) Prove that $G(n)$ is a subgroup of $\mathbb{Z}_n{}^*$. Hence, by Lagrange's theorem, if $G(n) \neq \mathbb{Z}_n{}^*$, then

$$|G(n)| \le \frac{|\mathbb{Z}_n{}^*|}{2} \le \frac{n-1}{2}.$$

(b) Suppose $n = p^k q$, where p and q are odd, p is prime, $k \ge 2$, and $\gcd(p, q) = 1$. Let $a = 1 + p^{k-1}q$. Prove that

$$\left(\frac{a}{n}\right) \not\equiv a^{(n-1)/2} \pmod n.$$

HINT Use the binomial theorem to compute $a^{(n-1)/2}$.

(c) Suppose $n = p_1 \ldots p_s$, where the p_i's are distinct odd primes. Suppose $a \equiv u \pmod{p_1}$ and $a \equiv 1 \pmod{p_2 p_3 \ldots p_s}$, where u is a quadratic non-residue modulo p_1 (note that such an a exists by the Chinese remainder theorem). Prove that
$$\left(\frac{a}{n}\right) \equiv -1 \pmod{n},$$
but
$$a^{(n-1)/2} \equiv 1 \pmod{p_2 p_3 \ldots p_s},$$
so
$$a^{(n-1)/2} \not\equiv -1 \pmod{n}.$$

(d) If n is odd and composite, prove that $|G(n)| \leq (n-1)/2$.

(e) Summarize the above: prove that the error probability of the Solovay-Strassen primality test is at most $1/2$.

4.15 Suppose we have a Las Vegas algorithm with failure probability ϵ.

(a) Prove that the probability of first achieving success on the nth trial is $p_n = \epsilon^{n-1}(1-\epsilon)$.

(b) The average (expected) number of trials to achieve success is
$$\sum_{n=1}^{\infty}(n \times p_n).$$
Show that this average is equal to $1/(1-\epsilon)$.

(c) Let δ be a positive real number less than 1. Show that the number of iterations required in order to reduce the probaility of failure to at most δ is
$$\left\lfloor \frac{\log_2 \delta}{\log_2 \epsilon} \right\rfloor.$$

4.16 Suppose Bob has carelessly revealed his decryption exponent to be $a = 14039$ in an **RSA Cryptosystem** with public key $n = 36581$ and $b = 4679$. Implement the probablistic algorithm to factor n given this information. Test your algorithm with the "random" choices $w = 9983$ and $w = 13461$. Show all computations.

4.17 Prove Equations 4.1 and 4.2 relating the functions *half* and *parity*.

4.18 Suppose $p = 199$, $q = 211$ and $B = 1357$ in the **Rabin Cryptosystem**. Perform the following computations.

(a) Determine the four square roots of 1 modulo n, where $n = pq$.

(b) Compute the encryption $y = e_K(32767)$.

(c) Determine the four possible decryptions of this given ciphertext y.

4.19 Factor 262063 and 9420457 using the $p-1$ method. How big does B have to be in each case to be successful?