CHAPTER 6

The RSA Algorithm

6.1 The RSA Algorithm

Alice wants to send a message to Bob, but they have not had previous contact and they do not want to take the time to send a courier with a key. Therefore, all information that Alice sends to Bob will potentially be obtained by the evil observer Eve. However, it is still possible for a message to be sent in such a way that Bob can read it but Eve cannot.

With all the previously discussed methods, this would be impossible. Alice would have to send a key, which Eve would intercept. She could then decrypt all subsequent messages. The possibility of the present scheme, called a **public key cryptosystem**, was first publicly suggested by Diffie and Hellman in their classic paper [Diffie-Hellman]. However, they did not yet have a practical implementation (although they did present an alternative key exchange procedure that works over public channels; see Section 7.4). In the next few years, several methods were proposed. The most successful, based on the idea that factorization of integers into their prime factors is hard, was proposed by Rivest, Shamir, and Adleman in 1977 and is known as the RSA algorithm.

It had long been claimed that government cryptographic agencies had discovered the RSA algorithm several years earlier, but secrecy rules prevented them from releasing any evidence. Finally, in 1997, documents released by CESG, a British cryptographic agency, showed that in 1970, James

Ellis had discovered public key cryptography, and in 1973, Clifford Cocks had written an internal document describing a version of the RSA algorithm in which the encryption exponent e (see the discussion that follows) was the same as the modulus n.

Here is how the RSA algorithm works. Bob chooses two distinct large primes p and q and multiplies them together to form

$$n = pq.$$

He also chooses an encryption exponent e such that

$$\gcd(e, (p-1)(q-1)) = 1.$$

He sends the pair (n, e) to Alice but keeps the values of p and q secret. In particular, Alice, who could possibly be an enemy of Bob, never needs to know p and q to send her message to Bob securely. Alice writes her message as a number m. If m is larger than n, she breaks the message into blocks, each of which is less than n. However, for simplicity, let's assume for the moment that $m < n$. Alice computes

$$c \equiv m^e \pmod{n}$$

and sends c to Bob. Since Bob knows p and q, he can compute $(p-1)(q-1)$ and therefore can find the decryption exponent d with

$$de \equiv 1 \pmod{(p-1)(q-1)}.$$

As we'll see later,

$$m \equiv c^d \pmod{n},$$

so Bob can read the message.

We summarize the algorithm in the following table.

The RSA Algorithm
1. Bob chooses secret primes p and q and computes $n = pq$.
2. Bob chooses e with $\gcd(e, (p-1)(q-1)) = 1$.
3. Bob computes d with $de \equiv 1 \pmod{(p-1)(q-1)}$.
4. Bob makes n and e public, and keeps p, q, d secret.
5. Alice encrypts m as $c \equiv m^e \pmod{n}$ and sends c to Bob.
6. Bob decrypts by computing $m \equiv c^d \pmod{n}$.

Example. Bob chooses

$$p = 885320963, \quad q = 238855417.$$

Then

$$n = p \cdot q = 211463707796206571.$$

Let the encryption exponent be

$$e = 9007.$$

The values of n and e are sent to Alice.

Alice's message is *cat*. We will depart from our earlier practice of numbering the letters starting with $a = 0$; instead, we start the numbering at $a = 01$ and continue through $z = 26$. We do this because, in the previous method, if the letter a appeared at the beginning of a message, it would yield a message number m starting with 00, so the a would disappear.

The message is therefore

$$m = 30120.$$

Alice computes

$$c \equiv m^e \equiv 30120^{9007} \equiv 113535859035722866 \pmod{n}.$$

She sends c to Bob.

Since Bob knows p and q, he knows $(p-1)(q-1)$. He uses the extended Euclidean algorithm (see Section 3.2) to compute d such that

$$de \equiv 1 \pmod{(p-1)(q-1)}.$$

The answer is

$$d = 116402471153538991.$$

Bob computes

$$c^d \equiv 113535859035722866^{116402471153538991} \equiv 30120 \pmod{n},$$

so he obtains the original message. ∎

There are several aspects that need to be explained, but perhaps the most important is why $m \equiv c^d \pmod{n}$. Recall Euler's theorem (Section 3.6): If $\gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \pmod{n}$. In our case, $\phi(n) = \phi(pq) = (p-1)(q-1)$. Suppose $\gcd(m, n) = 1$. This is very likely the case; since p and q are large, m probably has neither as a factor. Since $de \equiv 1 \pmod{\phi(n)}$, we can write $de = 1 + k\phi(n)$ for some integer k. Therefore,

$$c^d \equiv (m^e)^d \equiv m^{1+k\phi(n)} \equiv m \cdot (m^{\phi(n)})^k \equiv m \cdot 1^k \equiv m \pmod{n}.$$

We have shown that Bob can recover the message. If $\gcd(m, n) \neq 1$, Bob still recovers the message. See Exercise 19.

What does Eve do? She intercepts n, e, c. She does not know p, q, d. We assume that Eve has no way of factoring n. The obvious way of computing d requires knowing $\phi(n)$. We show later that this is equivalent to knowing p and q. Is there another way? We will show that if Eve can find d, then she can probably factor n. Therefore, it is unlikely that Eve finds the decryption exponent d.

Since Eve knows $c \equiv m^e \pmod{n}$, why doesn't she simply take the eth root of c? This works well if we are not working mod n but is very difficult in our case. For example, if you know that $m^3 \equiv 3 \pmod{85}$, you cannot calculate the cube root of 3, namely $1.4422\ldots$, on your calculator and then reduce mod 85. Of course, a case-by-case search would eventually yield $m = 7$, but this method is not feasible for large n.

How does Bob choose p and q? They should be chosen at random, independently of each other. How large depends on the level of security needed, but it seems that they should have at least 100 digits. For reasons that we discuss later, it is perhaps best if they are of slightly different lengths. When we discuss primality testing, we'll see that finding such primes can be done fairly quickly. A few other tests should be done on p and q to make sure they are not bad. For example, if $p - 1$ has only small prime factors, then n is easy to factor by the $p - 1$ method (see Section 6.4), so p should be rejected and replaced with another prime.

Why does Bob require $\gcd(e, (p-1)(q-1)) = 1$? Recall (see Section 3.3) that $de \equiv 1 \pmod{(p-1)(q-1)}$ has a solution d if and only if $\gcd(e, (p-1)(q-1)) = 1$. Therefore, this condition is needed in order for d to exist. The extended Euclidean algorithm can be used to compute d quickly. Since $p - 1$ is even, $e = 2$ cannot be used; one might be tempted to use $e = 3$. However, there are dangers in using small values of e (see Section 6.2, Computer Problem 14, and Section 17.3), so something larger is usually recommended. For example, one could let e be a moderately large prime. Then there is no difficulty ensuring that $\gcd(e, (p-1)(q-1)) = 1$.

In the encryption process, Alice calculates $m^e \pmod{n}$. Recall that this can be done fairly quickly and without large memory, for example, by successive squaring. This is definitely an advantage of modular arithmetic: If Alice tried to calculate m^e first, then reduce mod n, it is possible that recording m^e would overflow her computer's memory. Similarly, the decryption process of calculating $c^d \pmod{n}$ can be done efficiently. Therefore, all the operations needed for encryption and decryption can be done quickly (i.e., in time a power of $\log n$). The security is provided by the assumption that n cannot be factored.

We made two claims. We justify them here. Recall that the point of these two claims was that finding $\phi(n)$ or finding the decryption exponent d is essentially as hard as factoring n. Therefore, if factoring is hard, then there should be no fast, clever way of finding d.

Claim 1: Suppose $n = pq$ is the product of two distinct primes. If we know n and $\phi(n)$, then we can quickly find p and q.

Note that

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

Therefore, we know pq and $p+q$. The roots of the polynomial

$$X^2 - (n - \phi(n) + 1)X + n = X^2 - (p+q)X + pq = (X-p)(X-q)$$

are p and q, but they can also be calculated by the quadratic formula:

$$p, q = \frac{(n - \phi(n) + 1) \pm \sqrt{(n - \phi(n) + 1)^2 - 4n}}{2}.$$

This yields p and q.

For example, suppose $n = 221$ and we know that $\phi(n) = 192$. Consider the quadratic equation

$$X^2 - 30X + 221.$$

The roots are

$$p, q = \frac{30 \pm \sqrt{30^2 - 4 \cdot 221}}{2} = 13, \quad 17.$$

Claim 2: If we know d and e, then we can probably factor n.

In the discussion of factorization methods in Section 6.4, we show that if we have a universal exponent $b > 0$ such that $a^b \equiv 1 \pmod{n}$ for all a with $\gcd(a, n) = 1$, then we can probably factor n. Since $de - 1$ is a multiple of $\phi(n)$, say $de - 1 = k\phi(n)$, we have

$$a^{de-1} \equiv (a^{\phi(n)})^k \equiv 1 \pmod{n}$$

whenever $\gcd(a, n) = 1$. The method for universal exponents can now be applied.

One way the RSA algorithm can be used is when there are several banks, for example, that want to be able to send financial data to each other. If there are several thousand banks, then it is impractical for each pair of banks to have a key for secret communication. A better way is the following. Each bank chooses integers n and e as before. These are then published in a public book. Suppose bank A wants to send data to bank B. Then A looks up B's n and e and uses them to send the message. In practice, the RSA algorithm is not quite fast enough for sending massive amounts of data. Therefore, the RSA algorithm is often used to send a key for a faster encryption method such as DES.

PGP (= Pretty Good Privacy) is a popular method for encrypting email. When Alice sends an email message to Bob, she first signs the message using a digital signature algorithm such as those discussed in Chapter 9. She then encrypts the message using a block cipher such as triple DES (other choices are IDEA or CAST-128) with a randomly chosen 128-bit key (a new random key is chosen for each transmission). She then encrypts this key using Bob's public RSA key (other public key methods can also be used). When Bob receives the email, he uses his private RSA exponent to decrypt the random key. Then he uses this random key to decrypt the message, and he checks the signature to verify that the message is from Alice. For more discussion of PGP, see Section 10.6.

6.2 Attacks on RSA

In practice, the RSA algorithm has proven to be effective, as long as it is implemented correctly. We give a few possible implementation mistakes in the Exercises. Here are a few other potential difficulties. For more about attacks on RSA, see [Boneh].

Theorem. *Let $n = pq$ have m digits. If we know the first $m/4$, or the last $m/4$, digits of p, we can efficiently factor n.*

In other words, if p and q have 100 digits, and we know the first 50 digits, or the last 50 digits, of p, then we can factor n. Therefore, if we choose a random starting point to choose our prime p, the method should be such that a large amount of p is not predictable. For example, suppose we take a random 50-digit number N and test numbers of the form $N \cdot 10^{50} + k$, $k = 1, 3, 5, \ldots$, for primality until we find a prime p (which should happen for $k < 1000$). An attacker who knows that this method is used will know 47 of the last 50 digits (they will all be 0 except for the last 3 digits). Trying the method of the theorem for the various values of $k < 1000$ will eventually lead to the factorization of n.

For details of the preceding result, see [Coppersmith2]. A related result is the following.

Theorem. *Suppose (n, e) is an RSA public key and n has m digits. Let d be the decryption exponent. If we have at least the last $m/4$ digits of d, we can efficiently find d in time that is linear in $e \log_2 e$.*

This means that the time to find d is bounded as a function linear in $e \log_2 e$. If e is small, it is therefore quite fast to find d when we know a large part of d. If e is large, perhaps around n, the theorem is no better than a case-by-case search for d. For details, see [Boneh et al.].

6.2.1 Low Exponent Attacks

Low encryption or decryption exponents are tempting because they speed up encryption or decryption. However, there are certain dangers that must be avoided. One pitfall of using $e = 3$ is given in Computer Problem 14. Another difficulty is discussed in Chapter 17 (Lattice Methods). These problems can be avoided by using a somewhat higher exponent. One popular choice is $e = 65537 = 2^{16}+1$. This is prime, so it is likely that it is relatively prime to $(p-1)(q-1)$. Since it is one more than a power of 2, exponentiation to this power can be done quickly: To calculate x^{65537}, square x sixteen times, then multiply the result by x.

The decryption exponent d should of course be chosen large enough that brute force will not find it. However, even more care is needed, as the following result shows. One way to obtain desired properties of d is to choose d first, then find e with $de \equiv 1 \pmod{\phi(n)}$.

Suppose Bob wants to be able to decrypt messages quickly, so he chooses a small value of d. The following theorem of M. Wiener [Wiener] shows that often Eve can then find d easily. In practice, if the inequalities in the hypotheses of the proposition are weakened then Eve can still use the method to obtain d in many cases. Therefore, it is recommended that d be chosen fairly large.

Theorem. *Suppose p, q are primes with $q < p < 2q$. Let $n = pq$ and let $1 \leq d, e < \phi(n)$ satisfy $de \equiv 1 \pmod{(p-1)(q-1)}$. If $d < \frac{1}{3}n^{1/4}$, then d can be calculated quickly (that is, in time polynomial in $\log n$).*

Proof. Since $q^2 < pq = n$, we have $q < \sqrt{n}$. Therefore, since $p < 2q$,

$$n - \phi(n) = pq - (p-1)(q-1) = p + q - 1 < 3q < 3\sqrt{n}.$$

Write $ed = 1 + \phi(n)k$ for some integer $k \geq 1$. Since $e < \phi(n)$, we have

$$\phi(n)k < ed < \frac{1}{3}\phi(n)n^{1/4},$$

so $k < \frac{1}{3}n^{1/4}$. Therefore,

$$kn - ed = k(n - \phi(n)) - 1 < k(n - \phi(n)) < \frac{1}{3}n^{1/4}(3\sqrt{n}) = n^{3/4}.$$

Also, since $k(n - \phi(n)) - 1 > 0$, we have $kn - ed > 0$. Dividing by dn yields

$$0 < \frac{k}{d} - \frac{e}{n} < \frac{1}{dn^{1/4}} < \frac{1}{3d^2},$$

since $3d < n^{1/4}$ by assumption.

We now need a result about continued fractions. Recall from Section 3.12 that if x is a positive real number and k and d are positive integers with

$$\left|\frac{k}{d} - x\right| < \frac{1}{2d^2},$$

then k/d arises from the continued fraction expansion of x. Therefore, in our case, k/d arises from the continued fraction expansion of e/n. Therefore, Eve does the following:

1. Computes the continued fraction of e/n. After each step, she obtains a fraction A/B.

2. Eve uses $k = A$ and $d = B$ to compute $C = (ed - 1)/k$. (Since $ed = 1 + \phi(n)k$, this value if C is a candidate for $\phi(n)$.)

3. If C is not an integer, she proceeds to the next step of the continued fraction.

4. If C is an integer, then she finds the roots r_1, r_2 of $X^2-(n-C+1)X+n$. (Note that this is possibly the equation $X^2 - (n - \phi(n) + 1)X + n = (X - p)(X - q)$ from earlier.) If r_1 and r_2 are integers, then Eve has factored n. If not, then Eve proceeds to the next step of the continued fraction algorithm.

Since the number of steps in the continued fraction expansion of e/n is at most a constant times $\log n$, and since the continued fraction algorithm stops when the fraction e/n is reached, the algorithm terminates quickly. Therefore, Eve finds the factorization of n quickly. □

Remarks. Recall that the rational approximations to a number x arising from the continued fraction algorithm are alternately larger than x and smaller than x. Since $0 < \frac{k}{d} - \frac{e}{n}$, we only need to consider every second fraction arising from the continued fraction.

What happens if Eve reaches e/n without finding the factorization of n? This means that the hypotheses of the proposition are not satisfied. However, it is possible that sometimes the method will yield the factorization of n even when the hypotheses fail.

Example. Let $n = 1966981193543797$ and $e = 323815174542919$. The continued fraction of e/n is

$$[0; 6, 13, 2, 3, 1, 3, 1, 9, 1, 36, 5, 2, 1, 6, 1, 43, 13, 1, 10, 11, 2, 1, 9, 5]$$

$$= \cfrac{1}{6+\cfrac{1}{13+\cfrac{1}{2+\cfrac{1}{3+\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{1+\cdots}}}}}}}.$$

The first fraction is $1/6$, so we try $k = 1, d = 6$. Since d must be odd, we discard this possibility.

By the remark, we may jump to the third fraction:

$$\frac{1}{6+\frac{1}{13+\frac{1}{2}}} = \frac{27}{164}.$$

Again, we discard this since d must be odd.

The fifth fraction is $121/735$. This gives $C = (e \cdot 735 - 1)/121$, which is not an integer.

The seventh fraction is $578/3511$ This gives $C = 1966981103495136$ as the candidate for $\phi(n)$. The roots of

$$X^2 - (n - C + 1)X + n$$

are 37264873 and 52783789, to several decimal places of accuracy. Since

$$n = 37264873 \times 52783789,$$

we have factored n.

6.2.2 Short Plaintext

A common use of RSA is to transmit keys for use in DES or AES. However, a naive implementation could lead to a loss of security. Suppose a 56-bit DES key is written as a number $m \approx 10^{17}$. This is encrypted with RSA to obtain $c \equiv m^e \pmod{n}$. Although m is small, the ciphertext c is probably a number of the same size as n, so perhaps around 200 digits. However, Eve attacks the system as follows. She makes two lists:

1. $cx^{-e} \pmod{n}$ for all x with $1 \leq x \leq 10^9$.
2. $y^e \pmod{n}$ for all y with $1 \leq y \leq 10^9$.

She looks for a match between an element on the first list and an element on the second list. If she finds one, then she has $cx^{-e} \equiv y^e$ for some x, y. This yields

$$c \equiv (xy)^e \pmod{n},$$

so $m \equiv xy \pmod{n}$. Is this attack likely to succeed? Suppose m is the product of two integers x and y, both less than 10^9. Then these x, y will

yield a match for Eve. Not every m will have this property, but many values of m are the product of two integers less than 10^9. For these, Eve will obtain m.

This attack is much more efficient than trying all 10^{17} possibilities for m, which is nearly impossible on one computer, and would take a very long time even with several thousand computers working in parallel. In the present attack, Eve needs to compute and store a list of length 10^9, then compute the elements on the other list and check each one against the first list. Therefore, Eve performs approximately 2×10^9 computations (and compares with the list up to 10^9 times). This is easily possible on a single computer. For more on this attack, see [Boneh-Joux-Nguyen].

It is easy to prevent this attack. Instead of using a small value of m, adjoin some random digits to the beginning and end of m so as to form a much longer plaintext. When Bob decrypts the ciphertext, he simply removes these random digits and obtains m.

A more sophisticated method of preprocessing the plaintext, namely Optimal Asymmetric Encryption Padding (OAEP), was introduced by Bellare and Rogaway [Bellare-Rogaway2] in 1994. Suppose Alice wants to send a message m to Bob, whose RSA public key is (n, e), where n has k bits. Two positive integers k_0 and k_1 are specified in advance, with $k_0 + k_1 < k$. Alice's message is allowed to have $k - k_0 - k_1$ bits. Typical values are $k = 1024$, $k_0 = k_1 = 128$, $k - k_0 - k_1 = 768$. Let G be a function that inputs strings of k_0 bits and outputs strings of $k - k_0$ bits. Let H be a function that inputs $k - k_0$ bits and outputs k_0 bits. The functions G and H are usually constructed from hash functions (see Chapter 8 for a discussion of hash functions). To encrypt m, Alice first expands it to length $k - k_0$ by adjoining k_1 zero bits. The result is denoted $m0^{k_1}$. She then chooses a random string r of k_0 bits and computes

$$x_1 = m0^{k_1} \oplus G(r), \qquad x_2 = r \oplus H(x_1).$$

If the concatenation $x_1||x_2$ is a binary number larger than n, Alice chooses a new random number r and computes new values for x_1 and x_2. As soon as she obtains $x_1||x_2 < n$ (this has a probability of at least $1/2$ of happening for each r, as long as $G(r)$ produces fairly random outputs), she forms the ciphertext

$$E(m) = (x_1||x_2)^e \pmod{n}.$$

To decrypt a ciphertext c, Bob uses his private RSA decryption exponent d to compute $c^d \pmod{n}$. The result is written in the form

$$c^d \pmod{n} = y_1||y_2,$$

where y_1 has $k - k_0$ bits and y_2 has k_0 bits. Bob then computes

$$m0^{k_1} = y_1 \oplus G\big(H(y_1) \oplus y_2\big).$$

The correctness of this decryption can be justified as follows. If the ciphertext is the encryption of m, then

$$y_1 = x_1 = m0^{k_1} \oplus G(r) \quad \text{and} \quad y_2 = x_2 = r \oplus H(x_1).$$

Therefore,

$$H(y_1) \oplus y_2 = H(x_1) \oplus r \oplus H(x_1) = r$$

and

$$y_1 \oplus G\big(H(y_1) \oplus y_2\big) = x_1 \oplus G(r) = m0^{k_1}.$$

Bob removes the k_1 zero bits from the end of $m0^{k_1}$ and obtains m. Also, Bob has check on the integrity of the ciphertext. If there are not k_1 zeros at the end, then the ciphertext does not correspond to a valid encryption.

This method is sometimes called a plaintext-aware encryption. Note that the padding with x_2 depends on the message m and on the random parameter r. This makes chosen ciphertext attacks on the system more difficult.

6.2.3 Timing Attacks

Another type of attack on RSA and similar systems was discovered by Paul Kocher in 1995, while he was an undergraduate at Stanford. He showed that it is possible to discover the decryption exponent by carefully timing the computation times for a series of decryptions. Though there are ways to thwart the attack, this development was unsettling. There had been a general feeling of security since the mathematics was well understood. Kocher's attack demonstrated that a system could still have unexpected weaknesses.

Here is how the timing attack works. Suppose Eve is able to observe Bob decrypt several ciphertexts y. She times how long this takes for each y. Knowing each y and the time required for it to be decrypted will allow her to find the decryption exponent d. But first, how could Eve obtain such information? There are several situations where encrypted messages are sent to Bob and his computer automatically decrypts and responds. Measuring the response times suffices for the present purposes.

We need to assume that we know the hardware being used to calculate $y^d \pmod n$. We can use this information to calculate the computation times for various steps that potentially occur in the process.

Let's assume that $y^d \pmod n$ is computed by an algorithm given in Exercise 23 in Chapter 3, which is as follows:

Let $d = b_1 b_2 \dots b_w$ be written in binary (for example, when $x = 1011$, we have $b_1 = 1, b_2 = 0, b_3 = 1, b_4 = 1$). Let y and n be integers. Perform the following procedure:

1. *Start with* $k = 1$ *and* $s_1 = 1$.

2. *If* $b_k = 1$, *let* $r_k \equiv s_k y \pmod{n}$. *If* $b_k = 0$, *let* $r_k = s_k$.

3. *Let* $s_{k+1} \equiv r_k^2 \pmod{n}$.

4. *If* $k = w$, *stop. If* $k < w$, *add 1 to* k *and go to (2).*

Then $r_w \equiv y^d \pmod{n}$.

Note that the multiplication $s_k y$ occurs only when the bit $b_k = 1$. In many situations, there is a reasonably large variation in how long this multiplication takes. We assume this is the case here.

Before we continue, we need a few facts from probability. Suppose we have a random process that produces real numbers t as outputs. For us, t will be the time it takes for the computer to complete a calculation, given a random input y. The mean is the average value of these outputs. If we record outputs $t_1, \dots, t_n$, the mean should be approximately $m = (t_1 + \cdots t_n)/n$. The variance for the random process is approximated by

$$\mathrm{Var}(\{t_i\}) = \frac{(t_1 - m)^2 + \cdots + (t_n - m)^2}{n}.$$

The standard deviation is the square root of the variance and gives a measure of how much variation there is in the values of the t_i's.

The important fact we need is that when two random processes are independent, the variance for the sum of their outputs is the sum of the variances of the two processes. For example, we will break the computation done by the computer into two independent processes, which will take times t' and t''. The total time t will be $t' + t''$. Therefore, $\mathrm{Var}(\{t_i\})$ should be approximately $\mathrm{Var}(\{t_i'\}) + \mathrm{Var}(\{t_i''\})$.

Now assume Eve knows ciphertexts $y_1, \dots, y_n$ and the times that it took to compute each $y_i^d \pmod{n}$. Suppose she knows bits $b_1, \dots, b_{k-1}$ of the exponent d. Since she knows the hardware being used, she knows how much time was used in calculating $r_1, \dots, r_{k-1}$ in the preceding algorithm. Therefore, she knows, for each y_i, the time t_i that it takes to compute $r_k, \dots, r_w$.

Eve wants to determine b_k. If $b_k = 1$, a multiplication $s_k y \pmod{n}$ will take place for each ciphertext y_i that is processed. If $b_k = 0$, there is no such multiplication.

Let t_i' be the amount of time it takes the computer to perform the multiplication $s_k y \pmod{n}$, though Eve does not yet know whether this multiplication actually occurs. Let $t_i'' = t_i - t_i'$. Eve computes $\mathrm{Var}(\{t_i\})$ and $\mathrm{Var}(\{t_i''\})$. If $\mathrm{Var}(\{t_i\}) > \mathrm{Var}(\{t_i''\})$, then Eve concludes that $b_k = 1$. If not, $b_k = 0$. After determining b_k, she proceeds in the same manner to find all the bits.

Why does this work? If the multiplication occurs, t_i'' is the amount of time it takes the computer to complete the calculation after the multiplication. It is reasonable to assume t_i' and t_i'' are outputs that are independent of each other. Therefore,

$$\mathrm{Var}(\{t_i\}) \approx \mathrm{Var}(\{t_i'\}) + \mathrm{Var}(\{t_i''\}) > \mathrm{Var}(\{t_i''\}).$$

If the multiplication does not occur, t_i' is the amount of time for an operation unrelated to the computation, so it is reasonable to assume t_i and t_i' are independent. Therefore,

$$\mathrm{Var}(\{t_i''\}) \approx \mathrm{Var}(\{t_i\}) + \mathrm{Var}(\{-t_i'\}) > \mathrm{Var}(\{t_i\}).$$

Note that we couldn't use the mean in place of the variance, since the mean of $\{-t_i\}$ would be negative, so the last inequality would not hold. All that can be deduced from the mean is the total number of nonzero bits in the binary expansion of d.

The preceding gives a fairly simple version of the method. In practice, various modifications would be needed, depending on the specific situation. But the general strategy remains the same. For more details, see [Kocher].

A similar attack on RSA works by measuring the power consumed during the computations. See [Kocher et al.]. Attacks such as this one and the timing attack can be prevented by appropriate design features in the physical implementation.

6.3 Primality Testing

Suppose we have an integer of 200 digits that we want to test for primality. Why not divide by all the primes less than its square root? There are around 4×10^{97} primes less than 10^{100}. This is significantly more than the number of particles in the universe. Moreover, if the computer can handle 10^9 primes per second, the calculation would take around 10^{81} years. Clearly, better methods are needed. Some of these are discussed in this section.

A very basic idea, one that is behind many factorization methods, is the following.

Basic Principle. *Let n be an integer and suppose there exist integers x and y with $x^2 \equiv y^2 \pmod{n}$, but $x \not\equiv \pm y \pmod{n}$. Then n is composite. Moreover, $\gcd(x-y, n)$ gives a nontrivial factor of n.*

Proof. Let $d = \gcd(x-y, n)$. If $d = n$ then $x \equiv y \pmod{n}$, which is assumed not to happen. Suppose $d = 1$. A basic result on divisibility is that if $a|bc$ and $\gcd(a, b) = 1$, then $a|c$ (see Exercise 7 in Chapter 3). In our

case, since n divides $x^2 - y^2 = (x-y)(x+y)$ and $d = 1$, we must have that n divides $x + y$, which contradicts the assumption that $x \not\equiv -y \pmod{n}$. Therefore, $d \neq 1, n$, so d is a nontrivial factor of n. □

Example. Since $12^2 \equiv 2^2 \pmod{35}$, but $12 \not\equiv \pm 2 \pmod{35}$, we know that 35 is composite. Moreover, $\gcd(12-2, 35) = 5$ is a nontrivial factor of 35. ∎

It might be surprising, but factorization and primality testing are not the same. It is much easier to prove a number is composite than it is to factor it. There are many large integers that are known to be composite but that have not been factored. How can this be done? We give a simple example. We know by Fermat's theorem that if p is prime, then $2^{p-1} \equiv 1 \pmod{p}$. Let's use this to show 35 is not prime. By successive squaring, we find (congruences are mod 35)

$$\begin{aligned} 2^4 &\equiv 16, \\ 2^8 &\equiv 256 \equiv 11 \\ 2^{16} &\equiv 121 \equiv 16 \\ 2^{32} &\equiv 256 \equiv 11. \end{aligned}$$

Therefore,

$$2^{34} \equiv 2^{32}2^2 \equiv 11 \cdot 4 \equiv 9 \not\equiv 1 \pmod{35}.$$

Fermat's theorem says that 35 cannot be prime, so we have proved 35 to be composite without finding a factor.

The same reasoning gives us the following.

Fermat Primality Test. *Let $n > 1$ be an integer. Choose a random integer a with $1 < a < n-1$. If $a^{n-1} \not\equiv 1 \pmod{n}$, then n is composite. If $a^{n-1} \equiv 1 \pmod{n}$, then n is probably prime.*

Although this and similar tests are usually called "primality tests," they are actually "compositeness tests," since they give a completely certain answer only in the case when n is composite. The Fermat test is quite accurate for large n. If it declares a number to be composite, then this is guaranteed to be true. If it declares a number to be probably prime, then empirical results show that this is very likely true. Moreover, since modular exponentiation is fast, the Fermat test can be carried out quickly.

Recall that modular exponentiation is accomplished by successive squaring. If we are careful about how we do this successive squaring, the Fermat test can be combined with the Basic Principle to yield the following stronger result.

Miller-Rabin Primality Test. *Let $n > 1$ be an odd integer. Write $n - 1 = 2^k m$ with m odd. Choose a random integer a with $1 < a < n - 1$. Compute $b_0 \equiv a^m \pmod{n}$. If $b_0 \equiv \pm 1 \pmod{n}$, then stop and declare that n is probably prime. Otherwise, let $b_1 \equiv b_0^2 \pmod{n}$. If $b_1 \equiv 1 \pmod{n}$, then n is composite (and $\gcd(b_0 - 1, n)$ gives a nontrivial factor of n). If $b_1 \equiv -1 \pmod{n}$, then stop and declare that n is probably prime. Otherwise, let $b_2 \equiv b_1^2 \pmod{n}$. If $b_2 \equiv 1 \pmod{n}$, then n is composite. If $b_2 \equiv -1 \pmod{n}$, then stop and declare that n is probably prime. Continue in this way until stopping or reaching b_{k-1}. If $b_{k-1} \not\equiv -1 \pmod{n}$, then n is composite.*

Example. Let $n = 561$. Then $n - 1 = 560 = 16 \cdot 35$, so $2^k = 2^4$ and $m = 35$. Let $a = 2$. Then

$$\begin{aligned} b_0 &\equiv 2^{35} \equiv 263 \pmod{561} \\ b_1 &\equiv b_0^2 \equiv 166 \pmod{561} \\ b_2 &\equiv b_1^2 \equiv 67 \pmod{561} \\ b_3 &\equiv b_2^2 \equiv 1 \pmod{561}. \end{aligned}$$

Since $b_3 \equiv 1 \pmod{561}$, we conclude that 561 is composite. Moreover, $\gcd(b_2 - 1, 561) = 33$, which is a nontrivial factor of 561. ∎

If n is composite and $a^{n-1} \equiv 1 \pmod{n}$, then we say that n is a pseudoprime for the base a. If a and n are such that n passes the Miller-Rabin test, we say that n is a strong pseudoprime for the base a. We showed in Section 3.6 that $2^{560} \equiv 1 \pmod{561}$, so 561 is a pseudoprime for the base 2. However, the preceding calculation shows that 561 is not a strong pseudoprime for the base 2. For a given base, strong pseudoprimes are much more rare than pseudoprimes.

Up to 10^{10}, there are 455052511 primes. There are 14884 pseudoprimes for the base 2, and 3291 strong pseudoprimes for the base 2. Therefore, calculating $2^{n-1} \pmod{n}$ will fail to recognize a composite in this range with probability less than 1 out of 30 thousand, and using the Miller-Rabin test with $a = 2$ will fail with probability less than 1 out of 100 thousand.

It can be shown that the probability that the Miller-Rabin test fails to recognize a composite for a randomly chosen a is at most $1/4$. In fact, it fails much less frequently than this. See [Damgård et al.]. If we repeat the test 10 times, say, with randomly chosen values of a, then we expect that the probability of certifying a composite number as prime is at most $(1/4)^{10} \simeq 10^{-6}$. In practice, using the test for a single a is fairly accurate.

Though strong pseudoprimes are rare, it has been proved (see [Alford et al.]) that, for any finite set B of bases, there are infinitely many integers

that are strong pseudoprimes for all $b \in B$. The first strong pseudoprime for all the bases $b = 2, 3, 5, 7$ is 3215031751. There is a 337-digit number that is a strong pseudoprime for all bases that are primes < 200.

Suppose we need to find a prime of around 100 digits. The prime number theorem asserts that the density of primes around x is approximately $1/\ln x$. When $x = 10^{100}$, this gives a density of around $1/\ln(10^{100}) = 1/230$. Since we can skip the even numbers, this can be raised to $1/115$. Pick a random starting point, and throw out the even numbers (and multiples of other small primes). Test each remaining number in succession by the Miller-Rabin test. This will tend to eliminate all the composites. On average, it will take less than 100 uses of the Miller-Rabin test to find a likely candidate for a prime, so this can be done fairly quickly. If we need to be completely certain that the number in question is prime, there are more sophisticated primality tests that can test a number of 100 digits in a few seconds.

Why does the test work? Suppose, for example, that $b_3 \equiv 1 \pmod{n}$. This means that $b_2^2 \equiv 1^2 \pmod{n}$. Apply the Basic Principle from before. Either $b_2 \equiv \pm 1 \pmod{n}$, or $b_2 \not\equiv \pm 1 \pmod{n}$ and n is composite. In the latter case, $\gcd(b_2 - 1, n)$ gives a nontrivial factor of n. In the former case, the algorithm would have stopped by the previous step. If we reach b_{k-1}, we have computed $b_{k-1} \equiv a^{(n-1)/2} \pmod{n}$. The square of this is a^{n-1}, which must be 1 (mod n) if n is prime, by Fermat's theorem. Therefore, if n is prime, $b_{k-1} \equiv \pm 1 \pmod{n}$. All other choices mean that n is composite. Moreover, if $b_{k-1} \equiv 1$, then, if we didn't stop at an earlier step, $b_{k-2}^2 \equiv 1^2 \pmod{n}$ with $b_{k-2} \not\equiv \pm 1 \pmod{n}$. This means that n is composite (and we can factor n).

In practice, if n is composite, usually we reach b_{k-1} and it is not $\pm 1 \pmod{n}$. In fact, usually $a^{n-1} \not\equiv 1 \pmod{n}$. This means that Fermat's theorem fails, so n is not prime.

For example, let $n = 299$ and $a = 2$. Since $2^{298} \equiv 140 \pmod{299}$, Fermat's theorem and also the Miller-Rabin test say that 299 is not prime (without factoring it). The reason this happens is the following. Note that $299 = 13 \times 23$. An easy calculation shows that $2^{12} \equiv 1 \pmod{13}$ and no smaller exponent works. In fact, $2^j \equiv 1 \pmod{13}$ if and only if j is a multiple of 12. Since 298 is not a multiple of 12, we have $2^{298} \not\equiv 1 \pmod{13}$, and therefore also $2^{298} \not\equiv 1 \pmod{299}$. Similarly, $2^j \equiv 1 \pmod{23}$ if and only if j is a multiple of 11, from which we can again deduce that $2^{298} \not\equiv 1 \pmod{299}$. If Fermat's theorem (and the Miller-Rabin test) were to give us the wrong answer in this case, we would have needed $13 \cdot 23 - 1$ to be a multiple of $12 \cdot 11$.

Consider the general case $n = pq$, a product of two primes. For simplicity, consider the case where $p > q$ and suppose $a^k \equiv 1 \pmod{p}$ if and only if $k \equiv 0 \pmod{p-1}$. This means that a is a primitive root mod p; there are

$\phi(p-1)$ such $a \bmod p$. Since $0 < q-1 < p-1$, we have

$$n-1 \equiv pq-1 \equiv q(p-1)+q-1 \not\equiv 0 \pmod{p-1}.$$

Therefore, $a^{n-1} \not\equiv 1 \pmod{p}$ by our choice of a, which implies that $a^{n-1} \not\equiv 1 \pmod{n}$. Similar reasoning shows that usually $a^{n-1} \not\equiv 1 \pmod{n}$ for many other choices of a, too.

But suppose we are in a case where $a^{n-1} \equiv 1 \pmod{n}$. What happens? Let's look at the example of $n = 561$. Since $561 = 3 \times 11 \times 17$, we consider what is happening to the sequence b_0, b_1, b_2, b_3 mod 3, mod 11, and mod 17:

$$\begin{aligned}
b_0 &\equiv -1 \pmod{3}, & &\equiv -1 \pmod{11}, & &\equiv 8 \pmod{17} \\
b_1 &\equiv 1 \pmod{3}, & &\equiv 1 \pmod{11}, & &\equiv -4 \pmod{17} \\
b_2 &\equiv 1 \pmod{3}, & &\equiv 1 \pmod{11}, & &\equiv -1 \pmod{17} \\
b_3 &\equiv 1 \pmod{3}, & &\equiv 1 \pmod{11}, & &\equiv 1 \pmod{17}.
\end{aligned}$$

Since $b_3 \equiv 1 \pmod{561}$, we have $b_2^2 \equiv b_3 \equiv 1$ mod all three primes. But there is no reason that b_3 is the first time we get $b_i \equiv 1$ mod a particular prime. We already have $b_1 \equiv 1$ mod 3 and mod 11, but we have to wait for b_3 when working mod 17. Therefore, $b_2^2 \equiv b_3 \equiv 1$ mod 3, mod 11, and mod 17, but b_2 is congruent to 1 only mod 3 and mod 11. Therefore, $b_2 - 1$ contains the factors 3 and 11, but not 17. This is why $\gcd(b_2 - 1, 561)$ finds the factor 33 of 561. The reason we could factor 561 by this method is that the sequence $b_0, b_1, \ldots$ reached 1 mod the primes not all at the same time.

More generally, consider the case $n = pq$ (a product of several primes is similar) and suppose $a^{n-1} \equiv 1 \pmod{n}$. As pointed out previously, it is very unlikely that this is the case; but if it does happen, look at what is happening mod p and mod q. It is likely that the sequences $b_i \pmod{p}$ and $b_i \pmod{q}$ reach -1 and then 1 at different times, just as in the example of 561. In this case, we will be have $b_i \equiv -1 \pmod{p}$ but $b_i \equiv 1 \pmod{q}$ for some i; therefore, $b_i^2 \equiv 1 \pmod{n}$ but $b_i \not\equiv \pm 1 \pmod{n}$. Therefore, we'll be able to factor n.

The only way that n can pass the Miller-Rabin test is to have $a^{n-1} \equiv 1 \pmod{n}$ and also to have the sequences $b_i \pmod{p}$ and $b_i \pmod{q}$ reach 1 at the same time. This rarely happens.

Another primality test of a nature similar to the Miller-Rabin test is the following, which uses the Jacobi symbol (see Section 3.10).

Solovay-Strassen Primality Test. *Let n be an odd integer. Choose several random integers a with $1 < a < n-1$. If*

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

for some a, then n is composite. If

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

for all a, then n is probably prime.

Note that if n is prime, then the test will declare n to be a probable prime. This is because of the Proposition in Section 3.10.

The Jacobi symbol can be evaluated quickly, as in Section 3.10. The modular exponentiation can also be performed quickly.

For example,

$$\left(\frac{2}{15}\right) = -1 \not\equiv 23 \equiv 2^{(15-1)/2} \pmod{15},$$

so 15 is not prime. As in the Miller-Rabin tests, we usually do not get ± 1 for $a^{(n-1)/2} \pmod{n}$. Here is a case where it happens:

$$\left(\frac{2}{341}\right) = -1 \not\equiv +1 \equiv 2^{(341-1)/2} \pmod{341}.$$

Therefore, 341 is composite.

Both the Miller-Rabin and the Solovay-Strassen tests work quickly in practice, but, when p is prime, they do not give rigorous proofs that p is prime. There are tests that actually prove the primality of p, but they are somewhat slower and are used only when it is essential that the number be proved to be prime. Most of these methods are probabilistic, in the sense that they work with very high probability in any given case, but success is not guaranteed. In 2002, Agrawal, Kayal, and Saxena [Agrawal et al.] gave what is known as a deterministic polynomial time algorithm for deciding whether or not a number is prime. This means that the computation time is always, rather than probably, bounded by a constant times a power of $\log p$. This was a great theoretical advance, but their algorithm has not yet been improved to the point that it competes with the probabilistic algorithms.

For more on primality testing and its history, see [Williams].

6.4 Factoring

We now turn to factoring. The basic method of dividing an integer n by all primes $p \leq \sqrt{n}$ is much too slow for most purposes. For many years, people have worked on developing more efficient algorithms. We present some of them here. In Chapter 16, we'll also cover a method using elliptic

curves, and in Chapter 19, we'll show how a quantum computer, if built, could factor efficiently.

One method, which is also too slow, is usually called the **Fermat factorization** method. The idea is to express n as a difference of two squares: $n = x^2 - y^2$. Then $n = (x+y)(x-y)$ gives a factorization of n. For example, suppose we want to factor $n = 295927$. Compute $n + 1^2$, $n + 2^2$, $n + 3^2$, ..., until we find a square. In this case, $295927 + 3^2 = 295936 = 544^2$. Therefore,

$$295927 = (544 + 3)(544 - 3) = 547 \cdot 541.$$

The Fermat method works well when n is the product of two primes that are very close together. If $n = pq$, it takes $|p - q|/2$ steps to find the factorization. But if p and q are two randomly selected 100-digit primes, it is likely that $|p - q|$ will be very large, probably around 100-digits, too. So Fermat factorization is unlikely to work. Just to be safe, however, the primes for an RSA modulus are often chosen to be of slightly different sizes.

We now turn to more modern methods. If one of the prime factors of n has a special property, it is sometimes easier to factor n. For example, if p divides n and $p - 1$ has only small prime factors, the following method is effective. It was invented by Pollard in 1974.

The $p - 1$ Factoring Algorithm. *Choose an integer $a > 1$. Often $a = 2$ is used. Choose a bound B. Compute $b \equiv a^{B!} \pmod{n}$ as follows. Let $b_1 \equiv a \pmod{n}$ and $b_j \equiv b_{j-1}^j \pmod{n}$. Then $b_B \equiv b \pmod{n}$. Let $d = \gcd(b - 1, n)$. If $1 < d < n$, we have found a nontrivial factor of n.*

Suppose p is a prime factor of n such that $p - 1$ has only small prime factors. Then it is likely that $p - 1$ will divide $B!$, say $B! = (p - 1)k$. By Fermat's theorem, $b \equiv a^{B!} \equiv (a^{p-1})^k \equiv 1 \pmod{p}$, so p will occur in the greatest common divisor of $b - 1$ and n. If q is another prime factor of n, it is unlikely that $b \equiv 1 \pmod{q}$, unless $q - 1$ also has only small prime factors. If $d = n$, not all is lost. In this case, we have an exponent r (namely $B!$) and an a such that $a^r \equiv 1 \pmod{n}$. There is a good chance that the exponent factorization method (explained later in this section) will factor n. Alternatively, we could choose a smaller value of B and repeat the calculation.

How do we choose the bound B? If we choose a small B, then the algorithm will run quickly but will have a very small chance of success. If we choose a very large B, then the algorithm will be very slow. The actual value used will depend on the situation at hand.

In the applications, we will use integers that are products of two primes, say $n = pq$, but that are hard to factor. Therefore, we should ensure that $p-1$ has at least one large prime factor. This is easy to accomplish. Suppose we want p to have around 100 digits. Choose a large prime p_0, perhaps

around 10^{40}. Look at integers of the form $kp_0 + 1$, with k running through some integers around 10^{60}. Test $kp_0 + 1$ for primality by the Miller-Rabin test, as before. On the average, this should produce a desired value of p in less than 100 steps. Now choose a large prime q_0 and follow the same procedure to obtain q. Then $n = pq$ will be hard to factor by the $p - 1$ method.

The elliptic curve factorization method (see Section 16.3) gives a generalization of the $p - 1$ method. However, it uses some random numbers near $p - 1$ and only requires at least one of them to have only small prime factors. This allows the method to detect many more primes p, not just those where $p - 1$ has only small prime factors.

6.4.1 The Quadratic Sieve

Since it is the basis of the best current factorization methods, we repeat the following result from Section 6.3.

Basic Principle. *Let n be an integer and suppose there exist integers x and y with $x^2 \equiv y^2 \pmod{n}$, but $x \not\equiv \pm y \pmod{n}$. Then n is composite. Moreover, $\gcd(x - y, n)$ gives a nontrivial factor of n.*

Suppose we want to factor $n = 3837523$. Observe the following:

$$\begin{aligned} 9398^2 &\equiv 5^5 \cdot 19 \pmod{3837523} \\ 19095^2 &\equiv 2^2 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \pmod{3837523} \\ 1964^2 &\equiv 3^2 \cdot 13^3 \pmod{3837523} \\ 17078^2 &\equiv 2^6 \cdot 3^2 \cdot 11 \pmod{3837523}. \end{aligned}$$

If we multiply the relations, we obtain

$$\begin{aligned} (9398 \cdot 19095 \cdot 1964 \cdot 17078)^2 &\equiv (2^4 \cdot 3^2 \cdot 5^3 \cdot 11 \cdot 13^2 \cdot 19)^2 \\ 2230387^2 &\equiv 2586705^2. \end{aligned}$$

Since $2230387 \not\equiv \pm 2586705 \pmod{3837523}$, we now can factor 3837523 by calculating

$$\gcd(2230387 - 2586705, 3837523) = 1093.$$

The other factor is $3837523/1093 = 3511$.

Here is a way of looking at the calculations we just did. First, we generate squares such that when they are reduced mod n =3837523 they can be written as products of small primes (in the present case, primes less than 20). This set of primes is called our **factor base**. We'll discuss how to generate such squares shortly. Each of these squares gives a row in a matrix, where the entries are the exponents of the primes 2, 3, 5, 7, 11, 13, 17, 19.

For example, the relation $17078^2 \equiv 2^0 \cdot 3^2 \cdot 11 \pmod{3837523}$ gives the row 6, 2, 0, 0, 1, 0, 0, 0.

In addition to the preceding relations, suppose that we have also found the following relations:

$$\begin{aligned} 8077^2 &\equiv 2 \cdot 19 \pmod{3837523} \\ 3397^2 &\equiv 2^5 \cdot 5 \cdot 13^2 \pmod{3837523} \\ 14262^2 &\equiv 5^2 \cdot 7^2 \cdot 13 \pmod{3837523}. \end{aligned}$$

We obtain the matrix

9398	0	0	5	0	0	0	0	1
19095	2	0	1	0	1	1	0	1
1964	0	2	0	0	0	3	0	0
17078	6	2	0	0	1	0	0	0
8077	1	0	0	0	0	0	0	1
3397	5	0	1	0	0	2	0	0
14262	0	0	2	2	0	1	0	0

Now look for linear dependencies mod 2 among the rows. Here are three of them:

1. 1st + 5th + 6th = (6,0,6,0,0,2,0,2)$\equiv$ 0 (mod 2)
2. 1st + 2nd + 3rd + 4th = (8,4,6,0,2,4,0,2)$\equiv$ 0 (mod 2)
3. 3rd + 7th = (0,2,2,2,0,4,0,0)$\equiv$ 0 (mod 2)

When we have such a dependency, the product of the numbers yields a square. For example, these three yield

1. $(9398 \cdot 8077 \cdot 3397)^2 \equiv 2^6 \cdot 5^6 \cdot 13^2 \cdot 19^2 \equiv (2^3 \cdot 5^3 \cdot 13 \cdot 19)^2$
2. $(9398 \cdot 19095 \cdot 1964 \cdot 17078)^2 \equiv (2^3 \cdot 3^2 \cdot 5^3 \cdot 11 \cdot 13^2 \cdot 19)^2$
3. $(1964 \cdot 14262)^2 \equiv (3 \cdot 5 \cdot 7 \cdot 13^2)^2$

Therefore, we have $x^2 \equiv y^2 \pmod{n}$ for various values of x and y. If $x \not\equiv \pm y \pmod{n}$, then $\gcd(x - y, n)$ yields a nontrivial factor of n. If $x \equiv \pm y \pmod{n}$, then $\gcd(x - y, n) = 1$ or n, so we don't obtain a factorization. In our three examples, we have

1. $3590523^2 \equiv 247000^2$, but $3590523 \equiv -247000 \pmod{3837523}$
2. $2230387^2 \equiv 2586705^2$ and $\gcd(2230387 - 2586705, 3837523) = 1093$
3. $1147907^2 \equiv 17745^2$ and $\gcd(1147907 - 17745, 3837523) = 1093$

Year	Number of Digits
1964	20
1974	45
1984	71
1994	129
1999	155
2003	174
2005	200

Table 6.1: Factorization Records

We now return to the basic question: How do we find the numbers 9398, 19095, etc.? The idea is to produce squares that are slightly larger than a multiple of n, so they are small mod n. This means that there is a good chance they are products of small primes. An easy way is to look at numbers of the form $[\sqrt{in}+j]$ for small j and for various values of i. Here $[x]$ denotes the greatest integer less than or equal to x. The square of such a number is approximately $in + 2j\sqrt{in} + j^2$, which is approximately $2j\sqrt{in} + j^2$ mod n. As long as i is not too large, this number is fairly small, hence there is a good chance it is a product of small primes.

In the preceding calculation, we have $8077 = [\sqrt{17n} + 1]$ and $9398 = [\sqrt{23n} + 4]$, for example.

The method just used is the basis of many of the best current factorization methods. The main step is to produce congruence relations

$$x^2 \equiv \text{product of small primes.}$$

An improved version of the above method is called the quadratic sieve. A recent method, the number field sieve, uses more sophisticated techniques to produce such relations and is somewhat faster in many situations. See [Pomerance] for a description of these two methods and for a discussion of the history of factorization methods. See also Exercise 28.

Once we have several congruence relations, they are put into a matrix, as before. If we have more rows than columns in the matrix, we are guaranteed to have a linear dependence relation mod 2 among the rows. This leads to a congruence $x^2 \equiv y^2 \pmod{n}$. Of course, as in the case of 1st + 5th + 6th $\equiv 0 \pmod 2$ considered previously, we might end up with $x \equiv \pm y$, in which case we don't obtain a factorization. But this situation is expected to occur at most half the time. So if we have enough relations – for example, if there are several more rows than columns – then we should have a relation that

yields $x^2 \equiv y^2$ with $x \not\equiv \pm y$. In this case $\gcd(x-y, n)$ is a nontrivial factor of n.

In the last half of the twentieth century, there was dramatic progress in factoring. This was partly due to the development of computers and partly due to improved algorithms. A major impetus was provided by the use of factoring in cryptology, especially the RSA algorithm. Table 6.1 gives the factorization records (in terms of the number of decimal digits) for various years.

6.4.2 Theoretical Methods

On the surface, the Miller-Rabin test looks like it might factor n quite often; but what usually happens is that b_{k-1} is reached without ever having $b_u \equiv \pm 1 \pmod{n}$. The problem is that usually $a^{n-1} \not\equiv 1 \pmod{n}$. Suppose, on the other hand, that we have some exponent r, maybe not $n-1$, such that $a^r \equiv 1 \pmod{n}$ for all a with $\gcd(a, n) = 1$. Then it is often possible to factor n. We note that such an exponent r must be even (if $n > 2$); since we can take $a \equiv -1 \pmod{n}$, we need $(-1)^r \equiv 1$.

Universal Exponent Factorization Method. *Suppose we have an exponent $r > 0$ such that $a^r \equiv 1 \pmod{n}$ for all a with $\gcd(a, n) = 1$. Write $r = 2^k m$ with m odd. Choose a random a with $1 < a < n-1$. If $\gcd(a, n) \not\equiv 1$, we have a factor of n, so assume $\gcd(a, n) = 1$. Let $b_0 \equiv a^m \pmod{n}$, and successively define $b_{u+1} \equiv b_u^2 \pmod{n}$ for $0 \le u \le k-1$. If $b_0 \equiv 1 \pmod{n}$, then stop and try a different a. If, for some u, we have $b_u \equiv -1 \pmod{n}$, stop and try a different a. If, for some u we have $b_{u+1} \equiv 1 \pmod{n}$ but $b_u \not\equiv \pm 1 \pmod{n}$, then $\gcd(b_u - 1, n)$ gives a nontrivial factor of n.*

This looks very similar to the Miller-Rabin test. The difference is that the existence of r guarantees that we have $b_{u+1} \equiv 1 \pmod{n}$ for some u, which doesn't happen as often in the Miller-Rabin situation. Trying a few values of a has a very high probability of factoring n.

Of course, we might ask how we can find an exponent r. Generally, this seems to be very difficult, and this test cannot be used in practice. However, it is useful in showing that knowing the decryption exponent in the RSA algorithm allows us to factor the modulus.

In some situations, we don't know a universal exponent, but we know an exponent r that works for one value of a. Sometimes this allows us to factor n.

Exponent Factorization Method. *Suppose we have an exponent $r > 0$ and an integer a such that $a^r \equiv 1 \pmod{n}$. Write $r = 2^k m$ with m odd. Let*

$b_0 \equiv a^m \pmod n$, and successively define $b_{u+1} \equiv b_u^2 \pmod n$ for $0 \le u \le k-1$. If $b_0 \equiv 1 \pmod n$, then stop; the procedure has failed to factor n. If, for some u, we have $b_u \equiv -1 \pmod n$, stop; the procedure has failed to factor n. If, for some u, we have $b_{u+1} \equiv 1 \pmod n$ but $b_u \not\equiv \pm 1 \pmod n$, then $\gcd(b_u - 1, n)$ gives a nontrivial factor of n.

Of course, if we take $a = 1$, then any r works. But then $b_0 = 1$, so the method fails. But if a and r are found by some reasonably sensible method, there is a good chance that this method will factor n.

6.5 The RSA Challenge

When the RSA algorithm was first made public in 1977, the authors made the following challenge.

Let the RSA modulus be

$$n = 1143816257578888676692357799761466120102182967212423625625618429357069352457338978305971235639587050589890751475992900268795435 41$$

and let $e = 9007$ be the encryption exponent. The ciphertext is

$$c = 9686961375462206147714092225435588290575999112457431987469512093081629822514570835693147662288398962801339199055182994515781515 4.$$

Find the message.

The only known way of finding the plaintext is to factor n. In 1977, it was estimated that the then-current factorization methods would take 4×10^{16} years to do this, so the authors felt safe in offering \$100 to anyone who could decipher the message before April 1, 1982. However, techniques have improved, and in 1994, Atkins, Graff, Lenstra, and Leyland succeeded in factoring n.

They used 524339 "small" primes, namely those less than 16333610, plus they allowed factorizations to include up to two "large" primes between 16333610 and 2^{30}. The idea of allowing large primes is the following: If one large prime q appears in two different relations, these can be multiplied to produce a relation with q squared. Multiplying by $q^{-2} \pmod n$ yields a relation involving only small primes. In the same way, if there are several

relations, each with the same two large primes, a similar process yields a relation with only small primes. The "birthday paradox" (see Section 8.4) implies that there should be several cases where a large prime occurs in more than one relation.

Six hundred people, with a total of 1600 computers working in spare time, found congruence relations of the desired type. These were sent by e-mail to a central machine, which removed repetitions and stored the results in a large matrix. After 7 months, they obtained a matrix with 524339 columns and 569466 rows. Fortunately, the matrix was sparse, in the sense that most of the entries of the matrix were 0s , so it could be stored efficiently. Gaussian elimination reduced the matrix to a nonsparse matrix with 188160 columns and 188614 rows. This took a little less than 12 hours. With another 45 hours of computation, they found 205 dependencies. The first three yielded the trivial factorization of n, but the fourth yielded the factors

$p =$
3490529510847650949147849619903898133417764638493387843990820577,

$q =$
3276913299326670954996198819083446141317764296799294253979828853 3.

Computing $9007^{-1} \pmod{(p-1)(q-1)}$ gave the decryption exponent

$d =$
1066986143685780244428687713289201547807099066339378628012262244966310631259117744708733401685974623065539685445132771090536060 95.

Calculating $c^d \pmod{n}$ yielded the plaintext message

2008050013010709030023151804190001180500191721050113091908001519190906180107 05,

which, when changed back to letters using $a = 01, b = 02, \ldots,$ blank $= 00$, yielded

the magic words are squeamish ossifrage

(a squeamish ossifrage is an oversensitive hawk; the message was chosen so that no one could decrypt the message by guessing the plaintext and showing that it encrypted to the ciphertext). For more details of this factorization, see [Atkins et al.].