Computer Science 52

RSA Encryption

Spring Semester, 2017

This document presents the technique of RSA encryption and its mathematical foundation. The implementation details can be found in Sections II, IV, and V. The number theory underlying the technique is in Section III. A broader discussion of the use of RSA encryption appears in Sections VI and VII.

I Encryption: A Quick Overview

Private communication has been a concern throughout history. People have developed many clever ways to send secret messages to their partners in business, politics, war, and adultery. In the age of computer communication, we assume that anyone may examine the bits flying around the internet, and we use mathematical techniques to scramble the bits in our message so that it is intelligible only at the endpoints of the transmission.

Most often, the technique of encryption is widely known. The security of the communication lies in a *key,* some binary data that can be used to scramble, or unscramble, the bits in a message. An encryption system must have three properties:

1. It must be "easy" to encrypt a message if you know the (encryption) key.
2. It must be "easy" to decrypt a message if you know the (decryption) key.
3. It must be "hard" to decrypt a message without the decryption key.

By "easy" we mean "polynomial time." By "hard" we mean "not in polynomial time, for any polynomial." Another way to think of "hard" is that it is not worth the time or effort required to decrypt a message without a key. For many modern schemes, decryption without a key would take many centuries—even using the fastest computers of the day.

A human-readable message is called a *plaintext.* An encrypted version of the message is called a *ciphertext.* There is an encryption function E that takes a plaintext P and an encryption key K_E, and produces the ciphertext C.

$$E(P, K_E) = C$$

There is also a decryption function D that takes a ciphertext and a decryption key K_D, and reproduces the plaintext message.

$$D(C, K_D) = P$$

In a *symmetric* or *private key* system, the encryption and decryption keys are the same. A private key system has the disadvantage that the parties must get together and agree upon a shared key. It has the advantage in that the computational overhead is smaller. Once the key is in place, communication can happen much faster.

In an *asymmetric* or *public key* system, the two keys are different. Each participant has her or his own pair of keys. The encryption keys are known to everyone, but the decryption keys are kept secret. Person A can look up person B's encryption key, encrypt a message with it, and send the result to person B. Only someone with B's decryption key, namely only B, can read the message. An eavesdropper E might intercept the encrypted message but would not be able to decipher it.

The names of the participants A and B are Alice and Bob. The eavesdropper E is Eve. The three of them appear in most of the encryption papers and books written in the last several decades. One wonders why, after all those years, Eve is still interested in Alice and Bob.

Our study will focus on the RSA system, named for the initials of its three inventors. It is a public key system that was invented in 1978 and is widely used.

The original paper, linked from the course web page, is interesting and accessible. R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (February 1978), 120-126. DOI=http://dx.doi.org/10.1145/359340.359342

II RSA Encryption: The Recipe

Here, briefly, are the steps to carry out an RSA encryption scheme.

1. Find two large prime numbers, p and q.
2. Compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$.
3. Find an integer d which is relatively prime to $\varphi(n)$.
4. Use an extension of Euclid's algorithm to obtain its modulo-$\varphi(n)$ inverse e.
5. Declare the pair (e, n) to be your public encryption key and publish it.
6. Keep your decryption key (d, n) secret.

A message m is a string of bits. We can think of it as the binary representation of a large integer. When someone wants to send a confidential message to you, they look up your public key (e, n), compute

$$c = m^e \bmod n,$$

and send c to you. Upon receiving c, you can recover the original message using your private key (d, n):

$$m = c^d \bmod n.$$

This works because of some number theoretic facts that we will discuss in Section III. The scheme is believed to be secure because there is

A few values like $m = 0$ and $m = 1$ are not changed by RSA encryption, but they probably do not correspond to very interesting messages.

no known way to find the private key (d, n) from the public key (e, n) except by discovering the factors p and q of n. If p and q are large enough, finding them would probably take centuries.

If the message m is a number larger than n, then it is necessary to break m into several smaller blocks of bits and encrypt each block separately. We discuss this, and some other implementation details, in Section V.

III Mathematical Foundations

Quotients and Remainders

If m and n are integers, and $n \neq 0$, then there is a unique pair of integers q and r satisfying

$$m = qn + r \quad \text{and} \quad 0 \leq r < |n|.$$

Not surprisingly, q is the *quotient* and r is the *remainder* when m is divided by n. We say that n *divides* m, or n is a *divisor* of m, when the remainder is zero. The notation $n \mid m$ is used to signify that n divides m.

Be aware that different processors compute remainders differently. The remainder operator % in C or Java may yield a result r which does not satisfy the stated inequality, but this will happen *only* when one or both of the arguments is negative.

Theorem 1 *Suppose that a, b, and c are integers, and $n \mid a$ and $n \mid b$. Then the following equations hold:*

1. $n \mid (a + b)$,
2. $n \mid (-a)$, *and*
3. $n \mid (ac)$.

The proof is straightforward and omitted. Notice that it follows from the theorem that $n \mid (a - bc)$.

An integer p is a *prime number* if $1 < |p|$ and the only divisors of p are ± 1 and $\pm p$. A number that is not prime is *composite* and can be decomposed into two factors. Those factors are either prime or can themselves be decomposed further. A fundamental fact about the integers is that the process will eventually halt with the original number represented as a product of prime factors.

For example, 47 is itself a prime number and is the product of a single prime, while 4914 is the product of six primes (not all of them are different): $4914 = 2 \cdot 3^3 \cdot 7 \cdot 13$.

Theorem 2 (Fundamental Theorem of Arithmetic) *Any positive integer can be written as the product of positive primes. The representation is unique up to the order of the factors.*

Exercise 3.1. Suppose that p and q are different prime numbers ant that $p \mid n$ and $q \mid n$. Use the Fundamental Theorem of Arithmetic to prove that $pq \mid n$.

Modular Arithmetic

We write $a \equiv b \pmod{n}$ if $b - a$ is divisible by n. This notion, called congruence, is an equivalence relation. All the usual laws of arithmetic hold for congruence; for example, we have the distributivity of multiplication over addition and the associativity of multiplication.

$$a(b + c) \equiv ab + ac \pmod{n}$$
$$a(bc) \equiv (ab)c \pmod{n}$$

The notation is potentially confusing. Here, "mod" is not used as an operator; we are not saying that a is equivalent to $(b \bmod n)$. Rather, the equivalence between a and b depends on n. A better notation might be $a \equiv_n b$.

When computing arithmetic results for a congruence, it does not hurt replace a value by its remainder on division by n. This keeps the results in the range between 0 and $|n - 1|$.

For encryption, n is most likely much larger than the largest possible integer. For this, we need multi-word data structures and special library routines for the arithmetic operations.

Exercise 3.2. Suppose that p and q are different prime numbers and that that $a \equiv b \pmod{m}$ and $a \equiv b \pmod{q}$. Prove $a \equiv b \pmod{pq}$. (Hint, use Exercise 3.1.

Greatest Common Divisors

Suppose that a and b are integers, with one or both being non-zero. An integer d is a *greatest common divisor* of a and b if

1. $d \mid a$,
2. $d \mid b$, and
3. if $e \mid a$ and $e \mid b$, then $e \mid d$.

According to this definition, if d is a greatest common divisor of a and b, then so is $-d$. We use the notation $\gcd(a, b)$ to denote the unique *positive* greatest common divisor. The integers a and b are *relatively prime* if $\gcd(a, b) = 1$.

The following theorem characterizes the greatest common divisor. As we shall see, it provides an extremely useful tool.

Theorem 3 *Suppose a and b are integers, at least one of which is non-zero, and let d be the least positive integer of the form $ua + vb$. Then $d = \gcd(a, b)$.*

PROOF: First observe that there are positive integers of the form $ua + vb$. (If a is positive, just take u to be 1 and v to be 0. If a is negative, take u to be -1 and v to be 0. If a is zero, then b is non-zero—take u to be 0 and v to be ± 1.) Therefore, there is a *least* positive integer d of the appropriate form.

Suppose that $d = u_0a + v_0b$. Divide a by d to obtain a quotient q and remainder r: $a = qd + r$ and $0 \leq r < d$. Then $r = a - qd = (1 - qu_0)a + (-qv_0)b$, so r is of the form $ua + vb$. But d was the least positive value of that form, and r is less than d. Therefore r cannot be positive. The only possibility is for r to be zero, and in that case, d divides a. In the same way, we conclude that d divides b.

If e divides both a and b, then e divides the combination $u_0a + v_0b$. Hence e divides d. This completes the proof that $d = \gcd(a, b)$.

We can add, subtract, and multiply in modulo-n arithmetic. We can *sometimes* divide. The next corollary tells us when.

Corollary 4 *If* $\gcd(a, n) = 1$*, then there is an integer* u *such that* $ua \equiv 1 \pmod{n}$.

PROOF: There are integers u and v such that $ua + vn = 1$. We see that $ua - 1 = (-v)n$, so n divides $ua - 1$ and $ua \equiv 1 \pmod{n}$.

Corollary 5 *If* $\gcd(a, n) = \gcd(b, n) = 1$*, then* $\gcd(ab, n) = 1$.

PROOF: Again, there are integers u and v such that $ua + vn = 1$. Similarly, there are u' and v' such that $u'b + v'n = 1$. Multiply the left hand sides of the two equations to get another expression that equals 1:

$$(uu')ab + (uav' + u'bv + vv')n = 1.$$

The least positive combination of ab and n is therefore 1, and $\gcd(ab, n) = 1$.

Exercise 3.3. Show, directly from the definition, that $\gcd(a, b)$ is unique.

Exercise 3.4. Suppose that a and n are relatively prime and n divides ab. Show that n divides b.

Exercise 3.5. Prove that a is relatively prime to b if and only if there are integers u and v such that $ua + vb = 1$.

Exercise 3.6. Suppose that $ua \equiv 1 \pmod{n}$. Prove that a is relatively prime to n.

Euler's Totient Function

A useful idea is Euler's *totient* function, which is defined as follows. Let n be a non-zero integer. Then $\varphi(n)$ is the number of integers between 1 and $n-1$ (inclusive) which are relatively prime to n. That is, $\varphi(n)$ is the cardinality of the set $\{j \mid 1 \leq j < n \text{ and } \gcd(j,n) = 1\}$.

For example, $\varphi(12) = 4$ because 1, 5, 7, and 11 are relatively prime to 12 but the other values in the range from 1 to 11 which are not.

Exercise 3.7. Prove that p is a prime number if and only if $\varphi(p) = p - 1$.

Exercise 3.8. If p and q are different primes, show that $\varphi(pq) = (p-1)(q-1)$.

Exercise 3.9. If p is a prime number and k is positive, what is the value of $\varphi(p^k)$?

Euclid's Algorithm

Theorem 3 is essential for characterizing the greatest common divisor, but it does not directly give a very efficient algorithm for computing the gcd. The following properties of the gcd function lead to Euclid's algorithm for computing the greatest common divisor.

Theorem 6 *For integers a and b, not both zero, we have the following properties:*

1. $\gcd(a,0) = |a|$.
2. $\gcd(a,b) = \gcd(b,a)$.
3. $\gcd(a,b) = \gcd(-a,b)$.
4. *If $b = qa + r$, then* $\gcd(a,b) = \gcd(r,a)$.

PROOF: The first three properties can be proved directly from the definition of the greatest common divisor. For the fourth, let $d = \gcd(a,b)$. we know that d divides both a and b. A consequence of Theorem 1 is that d also divides $r = b - qa$. Similarly, if e divides both r and a, then e also divides b, and hence e divides d. Therefore, $d = \gcd(r,a)$.

Euclid's algorithm uses the properties of Theorem 6 to preserve loop invariants. Suppose that the we wish to compute the greatest common divisor of two integers whose values are stored in the C variables `a0` and `b0`. The pseudo-code below starts by establishing three invariants:

$0 \leq$ `a`
$0 \leq$ `b`
$\gcd($`a`$,$ `b`$) = \gcd($`a0`$,$ `b0`$)$.

This is easily done with the two assignments Properties 2 and 3 from Theorem 6 show that the invariant is established.

```
a = abs(a0);
b = abs(b0);
while (a != 0)
      (a,b) = (b % a, a);
return b;
```

The invariants are preserved by the loop above: Even though a or b may be changed by an iteration of the loop, both variables remain non-negative. Moreover, property 4 from Theorem 6 guarantees that the gcd does not change.

On each iteration, either a is made smaller. The value of a must eventually reach zero, and the loop will terminate. When it does, b contains the greatest common divisor by property 1.

Picky point: If at the start of the loop $b < a$, then the first iteration simply reverses the values of a and b. After that, the value of a always decreases.

For encryption, we often want to express the greatest common divisor in terms of the original numbers a0 and b0,

```
gcd = u*a0 + v*b0.
```

In fact, the coefficients u and v are often more important than the value of the gcd itself.

It is easy to modify the Euclid's algorithm to provide the extra information. We simply use four new variables and maintain two additional invariants.

$$a = ua * a0 + va * b0$$

$$b = ub * a0 + vb * b0$$

If we initialize the four variables correctly and preserve the invariants inside the loop, then the pair ub and vb will be the desired coefficients when b is returned as the greatest common divisor at the completion of the loop.

Exercise 3.10. Fill in the details in extending Euclid's algorithm to find u and v such that u*a0 + v*b0 is the greatest common divisor of a0 and b0.

Exercise 3.11. Show that Euclid's algorithm makes at most $2(\lg a0 + \lg b0)$ loop iterations. (In two iterations, either a or b becomes at least one bit shorter. Can you find a better bound?)

Remember that lg is the base-2 logarithm.

Fermat's Theorem

The following theorem is the heart of the RSA algorithm.

Theorem 7 (Fermat's Theorem) *If $n \neq 0$ and $\gcd(m, n) = 1$, then*

$$m^{\varphi(n)} \equiv 1 \pmod{n}.$$

PROOF: Consider the integers between 1 and $|n| - 1$ which are relatively prime to n. There are $\varphi(n)$ of them, and we can list them:

(1) $a_1, a_2, \ldots a_{\varphi(n)}$.

For each i satisfying $1 \leq i \leq \varphi(n)$, let a'_i be the remainder upon division of ma_i by n. Both m and a_i are relatively prime to n, so by Corollary 5 the product ma_i is relatively prime to n. Further, by property 4 of Theorem 6, the remainder a'_i is relatively prime to n. Therefore, a'_i occurs among the elements of the list (1).

We next show that the elements in the list

(2) $a'_1, a'_2, \ldots a'_{\varphi(n)}$

are all different. If $a'_j = a'_k$, then $ma_j \equiv ma_k \pmod{n}$. We have that n divides $m(a_j - a_k)$, so by the result of Exercise 2, n divides $a_j - a_k$. Because the distance between a_j and a_k is less than n, this means that $a_j - a_k = 0$, or equivalently, that $j = k$. Therefore, the lists (1) and (2) contain exactly the same numbers, although perhaps in different orders.

Multiplying the numbers in either list gives the same result:

$$a_1 a_2 \ldots a_{\varphi(n)} = a'_1 a'_2 \ldots a'_{\varphi(n)},$$

or

$$\begin{aligned} a_1 a_2 \ldots a_{\varphi(n)} &\equiv ma_1\, ma_2 \ldots ma_{\varphi(n)} \pmod{n} \\ &\equiv m^{\varphi(n)}\, a_1 a_2 \ldots a_{\varphi(n)} \pmod{n}. \end{aligned} \tag{3}$$

Each term in the product $a_1 a_2 \ldots a_{\varphi(n)}$ is relatively prime to n. Using Corollary 5 inductively, the whole product is relatively prime to n. By Corollary 4, there is a number u such that $u a_1 a_2 \ldots a_{\varphi(n)} \equiv 1 \pmod{n}$. Multiplication by u effectively "cancels" $a_1 a_2 \ldots a_{\varphi(n)}$ from the members of Equation (3), leaving the desired result $1 \equiv m^{\varphi(n)} \pmod{n}$.

Corollary 8 (Fermat's Little Theorem) *If p is a prime number, then for all integers m, $m \equiv m^p \pmod{p}$.*

Exercise 3.12. Prove Corollary 8, Fermat's Little Theorem. (Hint: Recall that $\varphi(p) = p - 1$.)

Exercise 3.13. The specific fact upon which the RSA algorithm rests concerns the product of two primes. If n is the product of two distinct primes p and q, prove that $m \equiv m^{1+k\varphi(n)} \pmod{n}$, for all integers m and k. (A proof of this fact is sketched in the RSA paper.)

IV Finding Primes and Inverses

1. Find two large prime numbers, p and q.
2. Compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$.
3. Find an integer d which is relatively prime to $\varphi(n)$.
4. Use an extension of Euclid's algorithm to obtain its modulo-$\varphi(n)$ inverse e.
5. Declare the pair (e, n) to be your public encryption key and publish it.
6. Keep your decryption key (d, n) secret.

Recall from Section II our "recipe" for RSA encryption, repeated at the right. Steps 2, 5, and 6 are straightforward, and we have by now explained the extension to Euclid's algorithm in step 4. The only gaps are in steps 1 and 3, both of which begin with the word "find."

We resort to probabilistic techniques to find large prime numbers and numbers that are relatively prime to $\varphi(n)$. Although we incur the risk of an error, the probability of the error can be made tiny.

To use the probabilistic techniques, we need a source of random numbers. For our purposes in the assignment, the random number generator built into SML is sufficient. For "real" cryptography, we would need a much better source of random bits. We discuss random number generators further in Section V.

Industrial Strength Primes

Suppose that we have a number p that we think is prime. If we choose a random positive number a less than p, we can compute $a^p \bmod p$. If the result is different from a, then by Corollary 8, Fermat's Little Theorem, we know that p is *not* prime. If the result is a, then p may, or may not, be prime.

Now suppose that we repeat the test for many different values of a. If the two values are ever unequal, then we can declare with certainty that p is not prime. If, after several tests, we do not discover evidence that p is non-prime, it is tempting to declare p to be a prime number. However, there is a chance that we will make a mistake and declare a number to be prime when it is in fact not. But by making many such tests, we can make the probability of such an error very low. A number declared prime on the basis of such a test is called an *industrial strength prime.*

Nearly all non-prime numbers will, with high probability, be exposed by this process, but there are a few exceptions. Non-primes that will always be declared industrial strength primes by our method are called *Carmichael numbers.* Although there is an infinite number of them, they are rare—meaning that they are distributed sparsely among the integers. The smallest Carmichael number is 561. Using more sophisticated number theory, people have devised similar tests that do not have such a defect. These tests satisfy two properties:

- If p is prime, then every test will report it as prime.

- If p is not prime, then a given test will report it as prime with probability less than $1/2$. Therefore, if we make k tests, the probability of an error is at most $1/2^k$, a very small number.

Observe that we are *not* making a statement about "the probability that p is prime." A number is either prime or it is not; there is no chance involved. We are saying that, if a number is not prime, there is a small possibility that the test will give wrong information.

To find an industrial strength prime number, we simply generate a random number and test it, as above, several times. If it survives, we have our result. If not, we try again with a different random number. By increasing the number of tests, we decrease the probability of coming up with a non-prime.

Exercise 4.1. A question for thought: The strategy here is "guess a number and see if it is prime; if not, guess again." This seems as bad, or worse, than a brute-force search for primes. Why do we find primes after only a "reasonable" number of guesses?

For many years, industrial strength primality testing was the only technology available. More recently, in 2002, the AKS test for primality was developed. Refinements of it produce a deterministic test that tells us whether or not a k-bit integer is prime in $O(k^6)$ time. In practice, the algorithm turns out to be significantly slower than its probabilistic industrial strength counterpart.

There is, however, still no known way of *factoring* composite numbers in deterministic polynomial time. The security of RSA encryption rests on the difficulty of factoring n.

Inverses

If we have a number d, it is easy to see if $\gcd(d, \varphi(n)) = 1$. Just compute the greatest common divisor using Euclid's algorithm. To find an integer relatively prime to $\varphi(n)$, simply generate a random value between 3 and $\varphi(n) - 1$ and test it using Euclid's algorithm. If the result is 1, take that value as d in step 3. If the result is not 1, generate a new random value and repeat. Eventually, you will obtain a good value for d, and you can, in step 4, apply the extended version of Euclid's algorithm.

In practice, it pays off to combine the two steps by using the extended version of Euclid's algorithm in the first place. If the test succeeds, you have the inverse e right away. If not, you have wasted only a small amount of time in the extra overhead of the extended algorithm.

Exercise 4.2. Suppose that a has a multiplicative inverse in modulo-n arithmetic. Prove that a and n are relatively prime.

V Implementation: Details and Pitfalls

We are now prepared to show that we can decrypt encrypted messages. We can find a pair of large prime numbers p and q, compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$, find d which is relatively prime to $\varphi(n)$, and compute the value e for which $de = 1 \pmod{\varphi(n)}$. we know that $de-1$ is divisible by $\varphi(n)$, so there is a number k satisfying $de = 1 + k\varphi(n)$.

Recall from Section II that (e, n) is the encryption key and (d, n) is the decryption key. If m is a plaintext message, then the ciphertext is

$$c = m^e \bmod n.$$

To decrypt, we compute $c^d \bmod n$ to obtain

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m^{de} \bmod n = m^{1+k\varphi(n)} \bmod n.$$

The result of Exercise 3.13 tells us that

$$m \equiv m^{1+k\varphi(n)} \pmod{n},$$

so if $0 \le m < n$, computing $c^d \bmod n$ gets us back to m, as desired. But what if m is a long message and not less than n?

Blocking

If m is too long, we can break it into *blocks.* In effect, we can represent it in base-n notation.

$$m = m_0 + m_1 n + m_2 n^2 + \ldots + m_\ell n^\ell$$

Each of the blocks, m_0 through m_ℓ, are less than n and we can encrypt them separately to obtain a sequence c_0 through c_ℓ of ciphertexts. If we like, we can combine the result into one long ciphertext:

$$c = c_0 + c_1 n + c_2 n^2 + \ldots + c_\ell n^\ell.$$

This simple idea is called the *electronic code book mode.*

The electronic code book mode is sometimes sufficient, but it has potential weaknesses. For example, if many of the blocks of the plaintext are identical, as they would be if the message were a graphic with a large monochrome background, then the many identical c_i's would reveal something about the plaintext.

An alternative is *cipher block chaining.* We begin by selecting a random value less than n, calling it iv for *initialization vector.* Then we compute

$$c_0 = (iv \oplus m_0)^e \bmod n, \quad \text{and}$$
$$c_{i+1} = (c_i \oplus m_{i+1})^e \bmod n \text{ for } 0 \le i < \ell.$$

We are "randomizing" each plaintext block before encrypting it. The ciphertext is the sequence $iv, c_0, \ldots c_\ell$. To decrypt, we reverse the process.

$$m_0 = iv \oplus (c_0^d \bmod n), \quad \text{and}$$
$$m_{i+1} = c_i \oplus (c_{i+1}^d \bmod n) \text{ for } 0 \le i < \ell.$$

Recall an identity for bitwise xor: $a \oplus (a \oplus b) = b$.

The cipher block chaining mode also guards against a danger when messages are short. If m is very short, then it is likely that one can decipher the communication by brute force. In electronic code book mode, one must replace the trailing zeroes in the last block by random bits.

Random Number Generators

To carry out a probabilistic algorithm to find primes and inverses, and to generate the initialization vector in cipher block chaining mode, we need a source of random data, a random number generator.

A random number generator produces "unpredictable" values. In reality, software functions use a deterministic algorithm that generates more or less unpredictable values. Such functions are valuable for statistical sampling and similar uses, but they are not suitable for serious cryptographic work.

Most software random number generators start with a *seed,* a piece of data that determines the random numbers produced. If one starts with the same seed, one will get the same sequence of "random" numbers. To avoid repeating values, programmers will sometimes use the low order bits of the computer's clock, representing microseconds, as the seed.

If someone can discover a seed, or know a few values from a software generator, they can predict subsequent values and be able to replay, with some accuracy, the steps in the creation of keys. Further, if the so-called random values repeat, it may happen that a prime is repeated. If n_A and n_B share a prime p, then someone can compute $\gcd(n_A, n_B) = p$ and compromise

"True" random number generators use natural processes, like cosmic rays or atmospheric noise, to generate random sequences of bits.

There are several web sites that provide true random numbers. Many operating systems have programs that generate random bits by collecting "randomness," or noise, from other processes on the system and the network.

Since our work on the assignment is only a demonstration of the RSA method, we will be satisfied with the more convenient pseudo-random number generators embedded in software.

Bad Keys

Considerable research has gone into the nature of keys, and in particular, the factorability of n. It turns out that there are some "bad choices" for keys. For example, if d is small, someone might find it by a brute-force search. (A small value of e is not a problem, and might actually speed up the encryption process.) If p and q are close to one another, then it is possible to factor n efficiently. Further, if $p - 1$ and $q - 1$ contain only "small" factors, then there is another technique to factor n efficiently.

The lesson is that in cryptography, or computer security more generally, one must pay close attention to *all* the details.

VI Security Guarantees

In Section I, we said that encryption and decryption should be "easy" for those in possession of the key. Decryption without the key should be "hard." The security of the RSA encryption system rests on the belief that finding the prime factors of large integers is "hard."

If you choose a key in which n is between, say 500 and 1000 bits, it would probably take someone years or centuries to find the factors. Of course, you might be careless and choose a bad key, or your adversary might be (incredibly) lucky and guess a factor, but the chances of that happening are tiny. If you are uncomfortable with your sense of security, you can always choose larger primes and produce a larger value of n.

The time required to factor goes up drastically with the size of n. Each additional bit in an integer doubles the number of possible values. The time required by state-of-the-art factoring algorithms does not increase quite that fast, but adding even ten bits would change a year into a century.

In the recent past, 1024 bit keys were considered fairly secure. Today, people are moving to 2048 or 4096 bit keys. As computers get faster,

the size of keys will increase. The choice of key length is made based on

- an estimate of the value of the data or resource being protected, and
- an estimate of the computing power of one's opponent.

No one has proved a lower bound for the time it takes to factor. It is possible, but would surprise almost all computer scientists, that someone will discover a fast factoring algorithm.

It is also possible, but unlikely, that someone will discover a way of deducing (d, n) from (e, n) without factoring n. (To crack an RSA key, it is actually enough to find $\varphi(n)$, but it is only a short step from there to computing the factors of n.)

Finally, there is the promise—or threat—of quantum computers. A device built on quantum mechanical principles can, in theory, factor a b bit number in $O(b^3)$ time. Small quantum computers have been built, and they have factored numbers like 15 and 21. It is unknown whether the technology will scale up to larger machines.

If someone comes up with a miraculous algorithm, or if large-scale quantum computers become a reality, the computing world will have to reject RSA and adopt encryption systems whose security is based on other "hard" problems.

VII Other Uses of RSA

In this section, we give an overview of some of the other uses of the RSA ideas. Be aware that the summary is superficial; the real-world implementations are much more intricate because they need to guard against many kinds of attacks.

For simplicity in the examples, let us assume that the principals are A and B, that their public encryption functions are E_A and E_B respectively, and that their private decryption functions are D_A and D_B. The public and private keys are buried in the functions and are usually not explicitly stated. When necessary, we will use subscripts on e, d, and n.

Digital Signatures

It is possible to electronically sign a message, a contract for example. To sign a message m, party A computes $s = D_A(m)$ and sends it to B. (Notice the use of the private *decryption* function.) Upon receipt, B

recovers $m = E_A(s)$. This works because encryption and decryption can be applied in either order. Party B now has the pair (m, s). It is irrefutable evidence that A signed the document, because anyone can verify that $m = E_A(s)$, and A is the only one who could have generated a value s with that property.

One problem with our description is that it is dangerous to sign entire messages. Suppose that A intercepted an encrypted message c sent to B. Then A can find two random looking values u and v with the property that $uv \equiv c \pmod{n_B}$. If A asks B to sign u and v, then A will have $D_B(u)$ and $D_B(v)$. Because decryption is done with exponentiation, $D_B(u)D_B(v) \equiv D_B(uv) \pmod{n_B}$, and A will have the plaintext corresponding to c.

The solution to the problem is that one never signs an actual message. Rather one signs a value derived from that message. A *cryptographic hash function* is a function that computes a *message authentication code* from a message. The message authentication code is of fixed size, typically 160 of 512 bits long. The function is designed so that it is extremely unlikely that two different messages will correspond to the same code. You may have seen references to the commonly used hash functions `MD5`, `SHA-1`, and `SHA-256`. Suppose that H is a cryptographic hash function. To sign a message m, party A computes $h = D_A(H(m))$ and sends $E_B(m, h)$ to B. Party B now has evidence that A signed m because $E_A(h) = H(m)$, and A is the only one who could have generated a value h with that property.

By (m, h) we mean the simple concatenation of the two blocks of bits.

Cryptographic hash functions can also be used to verify the integrity of a computer file. Often, when you download a file from the internet, the provider of the file displays the value of one of the hash functions. You can compute the hash function on the file you received and check that the hashes match. If they do not match, then the file may have been corrupted in transmission, or someone may have been replaced the original file with malware.

Identity and Certificates

The same idea can be used to verify identity. Suppose that A generates a "message" r which is a block of random bits. Party A sends $E_B(r)$ to party B, who recovers r and sends $E_A(r)$ back to A. Party A is now confident that the party on the other side is B, because only someone who knows D_B could have discovered r.

A random block of bits is often called a *nonce.*

It sounds simple, but there is a problem—one that is also present in our description of digital signatures. We have been assuming that everyone knows everyone else's public key. That may be true within a

small circle of friends, but we need better guarantees in the wilds of the internet.

We need some way to identify a key with its owner, and for that, we need a trusted authority. The authority verifies that the key (e_A, n_A) is associated with the person (or business or server) A by signing the message (e_A, n_A, A). That signed message is a *certificate.* Everyone knows the public key of the trusted authority and can verify the validity of the certificate. Party A can present that certificate as evidence that the public key in fact belongs to A.

There are actually many trusted authorities in the world. The largest one in the United States is Symantec (formerly VeriSign). We computer users never see the authorities' keys because they are built into operating systems and browsers.

A real-world certificate contains much more information than the simple example above. It will, at least, contain an expiration date and probably much more information. There is a hierarchy of authorities, with those at a higher lever attesting to the responsibility of others to issue certificates.

Key Exchange

As we mentioned earlier, RSA and other forms of public key encryption are computationally intensive. In practice, public key systems are used only to establish identity and to agree upon a key for a symmetric encryption system to be used for the actual communication. That key is called a *session key,* and it may be changed at regular intervals—sometimes as frequently as every five minutes.

Key exchange, and key management more generally, are complicated subjects. There are many ways in which information can leak out, or someone can impersonate another. Here is an overly simplified version of a key exchange system. It assumes that the two parties are already satisfied with one another's identity.

Party A creates a session key $k_{session}$ and sends $E_B(A, D_A(k_{session}))$ to B. Effectively, A is signing the session key to verify its source. The two parties now share a key and can use it to communicate privately.

So far, so good. But what if B is not confident that A has chosen a secure session key? In that case, both parties want a role in selecting the session key. Party A creates a key k_A and sends it to B as above, and party B responds symmetrically by creating a key k_B and sending it to A. The session key is then the bitwise xor of the two keys, $k_A \oplus k_B$.

Communication protocols are used when you access a secure web service, like `amazon.com` or a bank, through `https`. the protocols are carefully designed and are surprisingly delicate. Shortcomings and implementation errors are often found and corrected—a reason for keeping your software up to date. Although complex and sophiasticated, the protocols are made from components like the ones we have described here.