

# CSCI3390-Lecture 20: Probabilistic Algorithms: Number Theory and Cryptography

## 1 Two Problems

### Problem 1. Generate Primes

Find a prime number  $p$  of between 200 and 1000 decimal digits that has never been found before.

### Problem 2. Factor into Primes

Given  $N = pq$ , where  $p, q$  are primes generated by an algorithm for Problem 1, find  $p$  and  $q$ .

### Comment.

We will see that Problem 1 has an *easy* solution—one that generates a prime in time polynomial in the number of digits required, and which for large numbers of digits (like those given in the problem) produces primes with an microscopically small probability of ever having been encountered before.

A homework problem in Assignment 5 asked you to prove that if  $\mathbf{P} = \mathbf{NP}$ , then Problem 2 has a solution whose running time is polynomial in the number of digits of  $N$ . However, it is widely believed that this problem is intractable.

This question is not merely theoretical! Shown below is the ‘public-key certificate’ for amazon.com. The large value labeled ‘Public key’ is a number of 512 hexadecimal (base 16) digits, corresponding to 2048 bits in binary, or some six hundred decimal digits. It is, indeed, an instance of an integer  $N$  formed by multiplying together two primes produced according to a fast algorithm for Problem

1. We believe that there is no practical method for factoring  $N$ . If there were, encrypted traffic between Amazon and its customers (and many, many other secure websites and their clients) could be decrypted.

To get to the easy algorithm for Problem 1, we have to analyze the time complexity of several number-theoretic algorithms.

## 2 Euclid's Algorithm

Euclid's algorithm computes the greatest common divisor  $d = \gcd(m, n)$  of two integers  $m$  and  $n$ , provided  $m$  and  $n$  are not both zero.

Repeatedly replace  $(n, m)$  by  $(m, n \bmod m)$  until  $n \bmod m = 0$ .  
The gcd is the last value of  $m$ .

For example, with  $n = 114, m = 20$ , the algorithm produces the sequence of pairs

$$(114, 20), (20, 14), (14, 6), (6, 2), (2, 0),$$

so the greatest common divisor is 2. Observe that Euclid's algorithm performs a single division at each step, but throws away the quotient of the division and works exclusively with the remainders. A modification, often called the *Extended Euclid Algorithm* retains the quotients, and uses them to arrive at a pair  $a, b$  of integers such that  $am + bn = d$ .

Each division can be done in time polynomial in the size (number of digits) of  $n$  and  $m$ . How many divisions are performed? In two steps of the algorithm, the larger number  $n$  of the pair is replaced by  $n \bmod m$ . Observe that

$$n \bmod m < \frac{n}{2}$$


(Think of the two cases where  $m \leq \frac{n}{2}$  and  $m > \frac{n}{2}$ .) Thus the larger number in each pair decreases by a factor of at least 2 every 2 divisions. If we perform  $k$  divisions and are still going, then

$$n/2^{\frac{k}{2}} \geq 1,$$

so

$$k \leq 2 \log_2 n,$$

or twice the number of bits in the binary representation of  $n$ . Thus the algorithm runs in time polynomial (in fact something like cubic time) in the number of digits

 **Safari is using an encrypted connection to www.amazon.com.**  
Encryption with a digital certificate keeps information private as it's sent to or from the https website www.amazon.com.

VeriSign Class 3 Public Primary Certification Authority - G5  
VeriSign Class 3 Secure Server CA - G3  
www.amazon.com

Serial Number 10 E2 69 30 ED A9 B9 1B 80 67 09 2A 1F 6F B5 3C  
Version 3

Signature Algorithm SHA-1 with RSA Encryption ( 1.2.840.113549.1.1.5 )  
Parameters none

Not Valid Before Monday, April 27, 2015 at 8:00:00 PM Eastern Daylight Time  
Not Valid After Friday, October 2, 2015 at 7:59:59 PM Eastern Daylight Time

Public Key Info

Algorithm RSA Encryption ( 1.2.840.113549.1.1.1 )  
Parameters none

Public Key 256 bytes : 9C BC F7 39 29 5E 42 8A FF BF 71 87 BF EF 0C  
08 82 EB C9 AB E7 F0 C3 7C 6C 9C 46 12 3C DE 75 76 7E D3  
42 96 65 85 86 A9 D4 02 F3 B4 DF 24 5A 4C E8 05 79 64 B0  
16 3E D2 89 87 97 3A C1 15 5D 3C 20 07 A1 F4 5F CD BA DA  
0F 69 47 3B 03 49 31 92 59 5A 73 81 E0 78 5C 5F F9 FF 42 77  
BD 45 CE 63 87 EF 51 CC CD 3E 94 F8 29 D4 B4 30 3E 6F BF  
BA DF 75 2F D9 AB F7 A9 81 E2 6D 58 39 E3 19 10 33 C8 6B  
DE 2E 6E 7F F4 00 23 D2 90 0C 8D 8A 99 92 D8 34 C7 A3 A7  
E9 20 D4 F0 A0 7A B9 57 E7 F5 75 57 28 D9 66 BD 77 C7 F0  
8F 75 5F 4C 71 E2 12 B2 7B F7 FD 64 F0 29 8F F9 D7 EF 9E  
67 A4 AE 93 CE 69 10 25 35 90 F6 F0 33 57 6A E4 23 D4 4E 78  
A6 B3 AD DD A7 B3 6A 8D 1F 1A 8D 18 66 F1 4B 5F 74 B4 EC  
2C 31 E4 06 97 09 8F 16 D1 1C D6 87 03 7F 8E F8 5B 40 A9  
8E 2C 58 47 39 27 80 36 FD 57 A7

Exponent 65537

Figure 1: *The public key of amazon.com is an integer  $N > 10^{600}$ , shown here in hexadecimal, that is the product of two large primes. The security of the website depends on the conjectured practical impossibility of recovering the primes.*

of the input. The moral is that Euclid's algorithm is *fast*, and can be performed efficiently with numbers thousands of bits long. The same holds for the extended version, which performs a couple of additional operations for each division step.

### 3 Modular Exponentiation

Modular exponentiation is the problem of computing

$$a^b \bmod c$$

for positive integers  $a, b, c$ . If  $a, b$  are very large (think  $\approx 10^{600}$ , then  $a^b$  is literally too big to write down, but we can compute the value of the expression above by reducing modulo  $c$  at every step. The algorithm looks like:

- Set power  $\leftarrow 1$ .
- Repeat  $b$  times:  
Set power  $\leftarrow (a \times \text{power}) \bmod c$ .

Here the numbers do not get too big to write down, but the repetition of the loop  $10^{600}$  times cannot be carried out in practice. The solution to the dilemma is a simple trick that does many fewer multiplications and divisions than this naïve algorithm.

Let us say that we need to compute  $14^{30} \bmod 33$ . We begin with 14 and repeatedly square the last value and reduce it mod 33. This produces the following values:

$$\begin{aligned}14^2 &= 196 \equiv 31 \pmod{33} \\14^4 &\equiv 31^2 \equiv (-2)^2 = 4 \pmod{33} \\14^8 &\equiv 4^2 = 16 \pmod{33} \\14^{16} &\equiv 16^2 = 256 \equiv 25 \pmod{33}.\end{aligned}$$

This first phase required four multiplications and four divisions. For the second phase, we write the exponent 30 as a sum of distinct powers of 2:

$$30 = 16 + 8 + 4 + 2.$$

We then use this as a recipe to combine the values computed in the first phase:

$$14^{30} = 14^{16} \times 14^8 \equiv 25 \times 16 = 400 \equiv 4 \pmod{33}.$$

$$14^{28} = 14^{24} \times 14^4 \equiv 4 \times 4 = 16 \pmod{33}.$$

$$14^{30} = 14^{28} \times 14^2 \equiv 16 \times 31 = 496 \equiv 1 \pmod{33}.$$

This second phase required three multiplications and four divisions, so we did seven multiplications and divisions in all, compared to the 29 we would have had to perform for the naïve algorithm.

In general, this algorithm takes  $\log_2 b$  multiplications and divisions to compute the successive values  $a^{2^k} \pmod{c}$  in the first phase, and no more than this many operations for the second, so the total number of multiplications is bounded by  $2 \log_2 b$ , which is less than the number of decimal digits of  $b$ . Thus this algorithm can be carried out in time polynomial in the size of the original input.

## 4 Fermat's Theorem

### Statement

Let  $p$  be prime, and let  $1 \leq a < p$ . Then

$$a^{p-1} \equiv 1 \pmod{p}.$$

Let's give an example, which may seem kind of strange at first. We compute  $7^{14} \pmod{15}$ . Of course, 15 is not prime, and Fermat's Theorem says nothing about what the result here should be. We apply our repeated squaring algorithm:

$$7^2 = 49 \equiv 4 \pmod{15}.$$

$$7^4 \equiv 4^2 = 16 \equiv 1 \pmod{15}$$

$$7^8 \equiv 1^2 = 1 \pmod{15}.$$

Then

$$7^{14} = (7^8 \times 7^4) \times 7^2 \equiv 1 \times 1 \times 4 \equiv 4 \pmod{15}.$$

We didn't get 1, but Fermat's Theorem says that if 15 were prime, we would get 1. So 15 is composite. This is a proof that 15 is composite that gives no information about the factors of 15.

Of course, you already knew that 15 is composite! But because of the algorithm for modular exponentiation with repeated squaring, we can apply this algorithm to very large values, and obtain proofs of their compositeness without the necessity of factoring them.

This gives a kind of partial algorithm for testing whether a given integer  $n > 1$  is prime or composite:

- Pick a random value  $a$  such that  $1 \leq a < n$ .
- Compute  $b = a^{n-1} \pmod n$ .
- If  $b \neq 1$ , answer ‘definitely composite’, otherwise answer, ‘might be prime’.

This test can produce false positives. You may verify that  $11^{14} \pmod{15} = 1$ , so the algorithm would give ‘might be prime’ if 11 happened to have been selected as the test value for  $n = 15$ .

How common are false positives? If  $n$  is very large, then false positives are extremely rare: I ran this test with 100,000 randomly selected  $n$  in the range from 2 to  $10^{200}$  and compared the result to the one given by a more reliable primality test. There were *no* false positives—every  $n$  identified as ‘might be prime’ really was prime.

Below we will prove Fermat’s Theorem, and give a kind of explanation of why it works so well. As it turns out, there is a flaw in the algorithm that could undermine its performance (although in practical terms it does not seem to matter), and we will discuss some better alternatives.

## Proof of Fermat’s Theorem

Let  $1 \leq a < p$  and let  $p$  be prime. We first show a property of the set of integers

$$G = \{1, \dots, p - 1\}.$$

It is this: If  $x, y, z \in G$  and

$$xz \equiv yz \pmod p,$$

then

$$x \equiv y \pmod p.$$

To see why, note that if  $xz \equiv yz$ , then  $p \mid xz - yz = z(x - y)$ . Now if a prime number divides a product  $z(x - y)$ , then it must divide either  $z$  or  $x - y$ . Since  $z \in G$ , we cannot have  $p \mid z$ , so  $p \mid (x - y)$ , thus  $x \equiv y \pmod p$ .

In other words, we can ‘divide’ both sides of an equation mod  $p$  by any  $z \in G$ .

Now let

$$H = \{1, a, a^2 \pmod p, \dots\}.$$

$H \subseteq G$ . Since  $H$  is finite, there must be  $1 \leq k < \ell$  such that

$$a^k \equiv a^\ell \pmod p.$$

By the cancellation property we just proved, we can divide both sides of this equation  $k$  times by  $a$ , and get

$$1 \equiv a^j \pmod{p}$$

for some  $j > 0$ . If we let  $j$  denote the least such value, we have that  $H$  consists of exactly  $j$  elements

$$H = \{a, a^2 \pmod{p}, \dots, a^{j-1} \pmod{p}, a^j \pmod{p} = 1\}.$$

If this is all of  $G$ , then  $j = p - 1$ , so  $a^{p-1} \equiv 1 \pmod{p}$ , which is what we wanted to prove. If there is some  $b \in G \setminus H$ , then we consider the elements

$$bH = \{ba^k \pmod{p} : 1 \leq k \leq j\}$$

for  $k = 1, \dots, j$ . By the cancellation principle again, all  $j$  of these elements must be distinct. Furthermore, we can never have an element in both  $bH$  and  $H$ . This is because then

$$ba^k \equiv a^\ell \equiv a^{j+\ell} \pmod{p},$$

and we can cancel  $a^k$  from both sides to get

$$b \equiv a^{j+\ell-k} \pmod{p},$$

which would give  $b \in H$ , a contradiction. If this still does not exhaust all the elements of  $G$ , we pick  $b' \in G \setminus (H \cup bH)$ , and repeat. In the end, we will have  $G$  as the union of a bunch of pairwise disjoint sets,

$$G = H \cup bH \cup b'H \cup \dots,$$

each with  $j$  elements. Thus  $j|p - 1$  so  $p - 1 = js$  for some positive integer  $s$ . Accordingly

$$a^{p-1} = a^{js} = (a^j)^s \equiv 1^s = 1 \pmod{p},$$

completing the proof.

Students who have done some abstract algebra will recognize that  $G$  is a finite group,  $H$  a subgroup of  $G$ , and  $bH$  the cosets of  $H$ . We have given the standard proof that the number of elements in a subgroup divides the number of elements in the group.

### What is the probability of a false positive?

The test for primality that we have given above can produce false positives: In the example above this happened when 15 was erroneously identified as a prime because we had the bad luck to choose 11 as the base. Experiments with very large candidate primes show that when testing very large numbers at random, this hardly ever happens. Let us try to quantify the likelihood of this kind of error, and see what we can do about it.

Let  $n > 1$  be a composite number. The set

$$G = \{a : 1 \leq a < n \text{ and } \gcd(a, n) = 1\}$$

has the same cancellation properties as the set of integers  $1 \leq a < p$  when  $a$  is prime, and it is also closed under multiplication modulo  $n$ . (That is, if  $a, b \in G$ , then  $ab \bmod n \in G$ . In the language of abstract algebra,  $G$  is a group, just as it was in the case where  $n$  is prime.)

Let

$$H = \{a \in G : a^{n-1} \equiv 1 \pmod{n}\}.$$

This is the set of bases that produce false positives.  $H$  is nonempty (it includes 1) and is also closed under multiplication, because if  $a, b \in H$ ,

$$(ab)^{n-1} = a^{n-1}b^{n-1} \equiv 1 \cdot 1 = 1 \pmod{n}.$$

In the case  $n = 15$ , we have

$$G = \{1, 2, 4, 7, 8, 11, 13, 14\}, H = \{1, 4, 11, 14\}.$$

If  $G$  contains some element  $b$  that is not in  $H$ , then we can argue just as before that  $bH$  has the same number of elements as  $H$  and is disjoint from it. Thus

$$n - 1 \geq |G| \geq 2 \cdot |H|,$$

and thus  $|H| < \frac{(n-1)}{2}$ .

This means that the probability of a randomly chosen positive integer less than  $n$  being a false positive is less than  $\frac{1}{2}$ . Less than  $\frac{1}{2}$  doesn't sound all that great, but we can amplify the probability that we have not made an error by repeating the test many times:

repeat 100 times:



- pick  $1 \leq a < n$  at random
- compute  $b = a^{n-1} \bmod n$
- if  $b \neq 1$ , return ‘composite’

return ‘probably prime’

Every integer  $n$  that this test identifies as composite actually *is* composite. The test falsely identifies a composite as prime only if we get 100 successive false witnesses to primality. Since the probability of a false witness at each draw is less than  $\frac{1}{2}$ , such a succession of errors would occur with probability  $< 2^{-100}$ , probably a *lot* less.

Now  $2^{-100} \approx 10^{-30}$  is so very small, that the probability of some *other* kind of error causing the calculation to fail (hardware glitch?) is vastly larger. Thus for all practical purposes this always works.

Except...notice that in our calculation of the probability we assumed that  $H$  was properly contained in  $G$ . What if  $H = G$ ? This would mean that for every positive value of  $a$  less than  $n$  and relatively prime to  $n$ ,  $a^{n-1} \bmod n = 1$ .

Numbers  $n$  with this property are called *Carmichael numbers*. There are infinitely many of them, but they are extremely rare, so if you were hunting for primes at random, it would be very unlikely that you happened upon one of them by chance. Furthermore, a Carmichael number would not necessarily give a false positive to the above test, since we might choose  $a$  to be an integer that has a factor in common with  $n$ , in which case we can correctly identify  $n$  as composite. But our probability bound fails to hold in this case.

## 5 A Better Test, and a Better Test?

An improved test, in the same spirit, avoids this problem: If we find  $a^{n-1} \equiv 1 \pmod n$ , we don’t stop there and go on to the next  $a$ . The number  $n$  that we are testing must be odd (otherwise we wouldn’t do any elaborate calculation to determine whether  $n$  is prime) so we set  $k = \frac{n-1}{2}$  and try again, computing  $b = a^k \bmod n$ . Observe that  $a^k$  is then a square root of 1 modulo  $n$ .

If we now again have  $b = 1$ , and  $k$  even, we repeat the test again, dividing  $k$  by 2. Eventually we will reach a point where we can go no further, either because the exponent  $k$  is not divisible by 2, or because  $a^k \bmod p \neq 1$ . In all cases, we stop at a value  $b$  such that  $b$  is a square root of 1 mod  $n$ . If  $b$  is different from both

1 and  $n - 1$ , then

$$n|b^2 - 1 = (b - 1)(b + 1),$$

and neither  $b - 1$  nor  $b + 1$  is divisible by  $n$ . This means that  $n$  must be composite. In all other cases, we treat  $n$  as a possible prime, and move on to the next random  $a$ .

Here is an illustration of the test, with  $n = 561$ , assuming we have picked  $a = 35$ .

$$35^{560} \equiv 1 \pmod{561}$$

$$35^{280} \equiv 1 \pmod{561}$$

$$35^{140} \equiv 1 \pmod{561}$$

$$35^{70} \equiv 1 \pmod{561}$$

$$35^{35} \equiv 494 \pmod{561}$$

The last line shows that 561 is composite. Significantly, 561 is a Carmichael number, but we still get a proof of compositeness for these values as well.

It can be shown once again that the probability that a randomly-chosen  $a$  is a false witness for primality is less than  $\frac{1}{2}$ , and this holds without the additional caveat we had in the Fermat test—it works for every  $n$ .

This test was first discovered by Gary Miller in 1976. He did not describe it as an algorithm with random selection of  $a$ . Instead he showed that *if* a certain conjecture in number theory, the *Extended Riemann Hypothesis* is true, then it is sufficient to test all  $a \leq 2 \ln n$ , and thus the test determines primality with *no* possibility of error in time polynomial in the number of digits of  $n$ . It was Michael Rabin who suggested the method's practical use as a randomized algorithm, and the test is now known as the Miller-Rabin test.

So to summarize: The Miller-Rabin test as it is commonly used is a polynomial-time *probabilistic algorithm* for testing primality; whenever it answers 'composite', the input is composite; whenever it answers 'prime', the input is prime with very high probability, and the probability of error falls off exponentially with the number of repetitions of the main loop of the test. *If* a certain number-theoretic conjecture is true, the Miller-Rabin test can be turned into a polynomial-time *deterministic* algorithm for testing primality.

In 2002, a deterministic polynomial-time algorithm for testing primality was discovered, the AKS algorithm (Aggarwal-Kayal-Saxena). In practice, Miller-Rabin is much faster, so this has not supplanted probabilistic tests for practical work.

## 6 The Distribution of Primes

While we have described an efficient algorithm for testing if a given integer  $n$  is prime, we have not justified our original claim that Problem 1 is easy. While it may be easy to test whether a given integer is prime, *finding* a prime in a given range might be very hard. What if there were only a million primes that have 100 decimal digits? Randomly selecting 100-digit numbers and testing them for primality would then never yield a prime in practice, since the probability of hitting a prime would then be about  $10^{-93}$ . Moreover, the probability that any prime that you did happen to find had not been found before would be greater than  $10^{-6}$ , a small probability, to be sure, but not small enough, since we could very well have one million different parties trying to generate such primes for use in secure websites.

An examination of a table of primes less than, say, 1000, might lead you to worry that a scenario like the above is realistic, because the primes really do seem to thin out as you go to larger and larger numbers. Let us denote by  $\pi(n)$  the number of primes less than  $n$ . So, for example,  $\pi(12) = 5$ , because  $\{2, 3, 5, 7, 11\}$  is the set of primes less than 12. There are infinitely many primes, a fact known since antiquity, so

$$\lim_{n \rightarrow \infty} \pi(n) = +\infty.$$

The fact that the primes thin out is expressed by the equation

$$\lim_{n \rightarrow \infty} \pi(n)/n = 0.$$

That's the bad news. The good news is that this ratio does not approach 0 rapidly, but more like  $\frac{1}{\ln n}$ . The exact statement is

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n/\ln n} = 1.$$

This fact is called the *Prime Number Theorem*. Let's see what this means in practice. The number of primes with 1000 bits (approximately the size of the primes used to generate amazon.com's public key) is the number of primes between  $2^{999}$  and  $2^{1000}$ . By the Prime Number Theorem, the number of primes less than  $2^{1000}$  is about

$$2^{1000}/(1000 \ln 2) \approx 2^{1000}/693.$$

Thus, roughly speaking, more than one in every 700 numbers less than  $2^{1000}$  is prime. You get a same result for  $2^{999}$ . So an algorithm for finding primes in this

range is to follow 1 by 999 randomly selected bits, and test for primality. If you do this repeatedly, you will find on the average one prime for every 700 attempts. If you simplify matters by testing only odd integers, the average number of tries goes down to one out of 350. Moreover, the probability of getting the same prime twice by this procedure is for all practical purposes 0.

## 7 Public-key Cryptography

So now we know that Problem 1 at the beginning of these notes is easy. We believe that Problem 2 is hard. Thus generating two large primes by the above algorithm in effect locks up a secret that cannot be unlocked by someone who does not know the primes. How can we use this in practice?

The following is reproduced verbatim from the textbook for the Logic and Computation course. This is available on the canvas site, I am just repeating it here for your convenience.

Two parties, Alice and Bob, want to communicate privately, so that their messages cannot be read by an eavesdropper, Eve. Alice and Bob thus decide to encrypt their communications. To do this, they previously agreed on some secret information—a *key*. To send a message  $M$  (the *plaintext*) to Bob, Alice combines it with the key  $K$  and *encrypts* it using an encryption algorithm  $E$ . Bob receives the output of this algorithm, the *ciphertext*  $C = E(M, K)$ . To recover the plaintext, Bob combines the ciphertext with the key and applies a decryption algorithm  $D$ , and finds  $M = D(C, K)$ . This setup is called *symmetric encryption*: Alice and Bob share the same secret information  $K$ , and Bob can reply to Alice by encrypting his own plaintext messages  $M$  using the encryption algorithm  $E$ . (The encryption and decryption algorithms  $D$  and  $E$  themselves are *not* secret.)

This begs the question of exactly how Alice and Bob agree on the key  $K$  in the first place. In the kinds of practical applications of cryptography that you use all the time, the two parties are typically an online retailer and a customer who have not previously had any contact. The retailer cannot simply send the key  $K$  to the customer in unencrypted form, since then any eavesdropper will be able to recover  $K$  and encrypt and decrypt all subsequent communications between the two parties.

The amazing solution to this dilemma is that the retailer actually *does* send  $K$  to the customer.  $K$  is public information. The trick is that encryption and decryption are not carried out using the same key. Instead there are two keys, the public one  $K$  used for encryption, and a secret key  $K'$  used for decryption. The

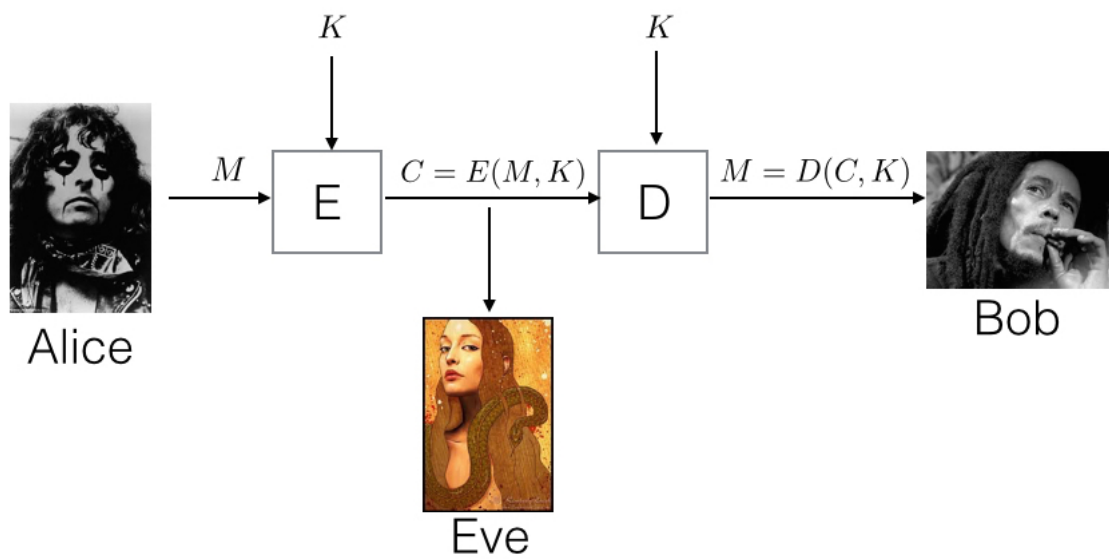


Figure 2: Symmetric encryption: Eavesdropper Eve intercepts the ciphertext  $C$  and knows the encryption and decryption algorithms  $E$  and  $D$ , but without access to the shared secret key  $K$ , she cannot recover the plaintext message  $M$ .

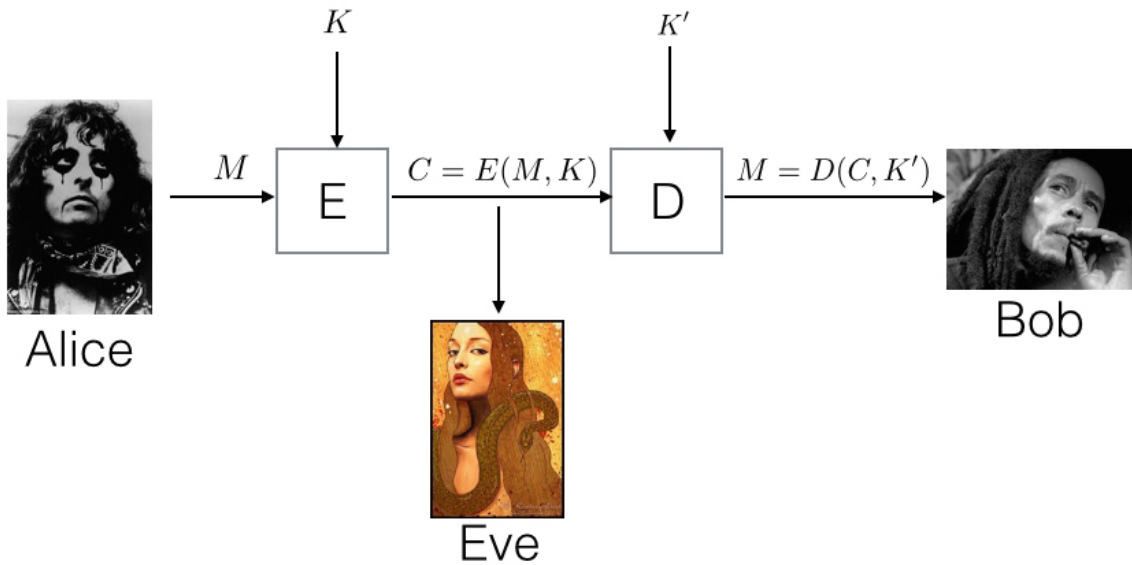


Figure 3: Public key encryption: Eavesdropper Eve intercepts the ciphertext  $C$  and knows both encryption and decryption algorithms  $E$  and  $D$ , and Bob’s public encryption key  $K$ , but without access to Bob’s secret decryption key  $K'$ , she cannot recover the plaintext message  $M$ .

customer encrypts a plaintext message  $M$  by computing

$$C = E(M, K),$$

and the retailer decrypts it by computing

$$M = D(C, K').$$

This setup is called *asymmetric*, or *public-key* cryptography.

How is such a thing possible? If both the encryption algorithm and the encryption key are publicly known, why can’t the adversary reverse-engineer the algorithm to recover the plaintext  $M$  from  $E(M, K)$ ?

## 7.1 The RSA algorithm

How can we harness the difficulty of factoring to create a public-key cryptographic system? The first proposal for doing so, and the method still in widest use, is

called RSA (after its inventors Rivest, Shamir, and Adleman). We'll describe the steps of the algorithm, work through an example with artificially small parameters, and then prove that the algorithm works.

### 7.1.1 Key Generation

The recipient, Bob, secretly generates two large primes  $p \neq q$  and forms the products

$$N = pq, K = (p - 1)(q - 1).$$

He chooses an integer  $e$  so that  $e$  is relatively prime to  $K$ , and determines integers  $d$  and  $c$  such that  $de = cK + 1$ . This is an easy problem in light of Euclid's Algorithm and the extended Euclid Algorithm: we may have to test several different candidate values of  $e$  before we find one that is relatively prime to  $K$ . Observe that  $d$  can be chosen so that  $0 < d < K$ , since if  $d$  were outside this range, we can add multiples of  $K$  to ensure this. So we can assume  $d > 0, c \geq 0$ .

Bob publishes the pair  $(e, N)$ : this is the public key. The integer  $d$  is the private key.

### 7.1.2 Encryption

Alice copies Bob's public key. The message to be sent must be encoded as an integer  $M < N$ . Alice then uses the public information and fast modular exponentiation to compute

$$C = M^e \bmod N.$$

The ciphertext  $C$  is sent to Bob.

### 7.1.3 Decryption

Bob uses his secret information and fast modular exponentiation to compute

$$C^d \bmod N.$$

We will prove below that this is identical to the original plaintext message  $M$ .

### 7.1.4 Example

We'll illustrate the algorithm with an example that uses small integers, so you can see what every step looks like. Of course, in practice, the algorithm is never used with such small values.

Let's choose  $p = 53, q = 61$  as our primes. Then

$$N = pq = 3233, K = (p - 1)(q - 1) = 3120.$$

A little trial and error shows  $\gcd(7, K) = 1$ , and we can apply the extended Euclid algorithm to find

$$\begin{aligned} 3120 &= 7 \cdot 445 + 5 \\ 7 &= 1 \cdot 5 + 2 \\ 5 &= 2 \cdot 2 + 1 \end{aligned}$$

so

$$\begin{aligned} 1 &= 5 - 2 \cdot 2 \\ &= (3120 - 7 \cdot 445) - 2 \cdot (7 - (3120 - 7 \cdot 445)) \\ &= 3 \cdot 3120 - 1337 \cdot 7 \end{aligned}$$

This would give  $d = -1337$ , so we adjust by adding 3120, and get  $d = 1783$ . This is the secret key. The pair  $(7, 3233)$  is the public key.

For purposes of this small example let's suppose that the sender encrypts two-letter pairs, first by encoding the pair as a four-digit integer. For example, the pair 'TB' would be encoded as 2002, since T is the 20th letter and B the 2nd letter of the alphabet.

To encrypt this using the public key, the sender computes

$$2002^7 \bmod 3233.$$

We have already seen how to do this quickly using repeated squaring. We'll let the built-in Python function `pow` do the work here:

```
>>> pow(2002, 7, 3233)
2817
```

The recipient takes the ciphertext 2817 and uses the secret information to decrypt:

$$2817^{1783} \bmod 3233.$$

We use `pow` again to get the result, which is the original plaintext.

```
>>> pow(2817, 1783, 3233)
2002
```



### 7.1.5 Proof of correctness of RSA

We have to show that encryption followed by decryption recovers the original plaintext, in other words, that

$$C^d \bmod N = M^{de} \bmod N = M.$$

To do this we will show

$$M^{de} \equiv M \pmod{p}$$

and

$$M^{de} \equiv M \pmod{q}.$$

This implies that both  $p$  and  $q$  divide  $M^{de} - M$ . Since  $p$  and  $q$  are distinct primes, we have  $N = pq$  also divides  $M^{de} - M$ . Thus

$$M^{de} \equiv M \pmod{N},$$

so  $M^{de} \bmod N = M$ .

To establish the claims above, note that if  $p|M$ , then  $p|M^{de}$ , so

$$M^{de} \equiv 0 \equiv M \pmod{p}.$$

If  $p \nmid M$ , then  $M \bmod p = a$ , where  $1 \leq a < p$ . Since  $de = cK + 1$  for some integer  $c$ , we have:

$$\begin{aligned} M^{de} &= M^{cK+1} \\ &= M^{c(p-1)(q-1)} M \\ &= (M^{p-1})^{c(q-1)} \cdot M \\ &\equiv (a^{p-1})^{c(q-1)} \cdot M \text{ (by Fermat's Theorem)} \\ &\equiv 1 \cdot M \\ &= M \pmod{p} \end{aligned}$$

The proof that  $M^{de} \equiv M \pmod{q}$  is identical.

### 7.1.6 Security and Efficiency of RSA

Since the exponent  $N$  in RSA is part of the public key, anyone who can find the prime factors of  $N$  will be able to reproduce the computation of the secret exponent  $d$  and decipher all subsequent messages. Knowledge of the secret exponent

can also be used to factor  $N$ . So RSA is only secure to the extent that factoring is infeasible. Currently, the best factoring algorithms, coupled with massive computational effort, can factor integers of approximately 200 decimal digits, or about 660 bits in binary. Public RSA keys in wide use are typically 1024 or 2048 bits long, with the latter becoming more common.

The conjecture that factoring is intrinsically hard does not itself guarantee the security of RSA: no one has ruled out the possibility that there is some efficient method for decryption that does not recover the secret exponent.

Thanks to tools like fast modular exponentiation, Euclid's algorithm, and probabilistic primality testing, RSA is 'fast', in the sense that the algorithms for key generation and encryption can be carried out reasonably quickly on very large numbers. Still, encryption with RSA is much slower than with conventional symmetric encryption, so it is not well-suited for large volumes of traffic. For this reason, encryption on the Internet is a two-phase process: when you contact a secure website, your browser downloads the RSA public key  $(e, N)$ , generates a symmetric key  $K$ , encrypts  $K$  with the RSA key and sends this to the server. Now both you and the server share the symmetric key  $K$ , which is used to encrypt all subsequent communication for the session.