

# Practical Cryptography

Niels Ferguson

Bruce Schneier



WILEY

Wiley Publishing, Inc.

**Executive Publisher: Robert Ipsen**  
**Executive Editor: Carol A. Long**  
**Editorial Manager: Kathryn A. Malm**  
**Managing Editor: Fred Bernardi**

This book is printed on acid-free paper.

Copyright © 2003 by Niels Ferguson and Bruce Schneier. All rights reserved.

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-mail: [permcoordinator@wiley.com](mailto:permcoordinator@wiley.com).

**Limit of Liability/Disclaimer of Warranty:** While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

**Trademarks:** Wiley, the Wiley Publishing logo and related trade dress are trademarks or registered trademarks of Wiley Publishing, Inc., in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

ISBN: 0-471-22894-X (C)

ISBN: 0-471-22357-3 (P)

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

# Chapter 11

## Primes

The following two chapters explain public-key cryptographic systems; unfortunately, this requires quite a bit of mathematics. It is always tempting to dispense with the understanding and only present the formulas and equations, but we feel very strongly that this is a dangerous thing to do. To use a tool, you must understand the properties of that tool. This is easy with something like a hash function. We have an “ideal” model of a hash function, and we require that the actual hash function behave like the ideal model. This is not so easy to do with public-key systems because there are no “ideal” models to work with. In practice, you have to deal with the mathematical properties of the public-key systems, and to do that safely you must understand these properties. There is no shortcut here; you must understand the mathematics. It’s not that difficult; the only background knowledge required is high school math. More specifically: the type of math the authors were taught in high school.

This chapter is about prime numbers. Prime numbers play an important role in mathematics, but we are interested in them because the most important public-key crypto systems are based on prime numbers.

## 11.1 Divisibility and Primes

A number  $a$  is a divisor of  $b$  (notation  $a \mid b$ , pronounced “ $a$  divides  $b$ ”) if you can divide  $b$  by  $a$  without leaving a remainder. For example, 7 is a divisor of 35 so we write  $7 \mid 35$ . We call a number a *prime* number if it has exactly two divisors, namely 1 and itself. For example, 13 is a prime; the two divisors are 1 and 13. The first few primes are easy to find: 2, 3, 5, 7, 11, 13, . . . . Any integer greater than 1 that is not prime is called a composite. The number 1 is neither prime nor composite.

We will use the proper mathematical notation and terminology in the chapters ahead. This will make it much easier to read other texts on this subject. The notation might look difficult and complicated at first, but this part of mathematics is really easy.

Here is a simple lemma about divisibility:

**Lemma 1.** *If  $a \mid b$  and  $b \mid c$  then  $a \mid c$ .*

*Proof.* If  $a \mid b$ , then there is an integer  $s$  such that  $as = b$ . (After all,  $b$  is divisible by  $a$  so it must be a multiple of  $a$ .) And if  $b \mid c$  then there is an integer  $t$  such that  $bt = c$ . But this implies that  $c = bt = (as)t = a(st)$  and therefore  $a$  is a divisor of  $c$ . (To follow this argument, just verify that each of the equal signs is correct. The conclusion is that the first item  $c$  must be equal to the last item  $a(st)$ .)  $\square$

The lemma is a statement of fact. The proof argues why the lemma is true. The little square box signals the end of the proof. Mathematicians love to use lots of symbols.<sup>1</sup> This is a very simple lemma, and the proof should be easy to follow, as long as you remember what the notation  $a \mid b$  means.

Prime numbers have been studied by mathematicians throughout the ages. Even today, if you want to generate all primes below one million, you should use an algorithm developed just over 2000 years ago by Eratosthenes, a friend of Archimedes. (Eratosthenes was also the first person to accurately measure the diameter of the earth. A mere 1700 years later Columbus allegedly used

---

<sup>1</sup>Using symbols has advantages and disadvantages. We’ll use whatever we think is most appropriate for this book.

a much smaller—and wrong—estimate for the size of the earth when he planned to sail to India by going due west.) Euclid, another great Greek mathematician, gave a beautiful proof that showed there are an infinite number of primes. This is such a beautiful proof that we'll include it here. Reading through it will help you reacquaint yourself with the math.

Before we start with the real proof we will give a simple lemma.

**Lemma 2.** *Let  $n$  be a positive number greater than 1. Let  $d$  be the smallest divisor of  $n$  that is greater than 1. Then  $d$  is prime.*

*Proof.* First of all, we have to check that  $d$  is well defined. (If there is a number  $n$  which has no smallest divisor, then  $d$  is not properly defined and the lemma is nonsensical.) We know that  $n$  is a divisor of  $n$ , and  $n > 1$ , so there is at least one divisor of  $n$  that is greater than 1. Therefore, there must also be a smallest divisor greater than 1.

To prove that  $d$  is prime we use a standard mathematician's trick called *reductio ad absurdum* or *proof by contradiction*. To prove a statement  $X$  we first assume that  $X$  is not true, and show that this assumption leads to a contradiction. If assuming that  $X$  is not true leads to a contradiction, then obviously  $X$  must be true.

In our case we will assume that  $d$  is not a prime. If  $d$  is not a prime, it has a divisor  $e$  such that  $1 < e < d$ . But we know from Lemma 1 that if  $e \mid d$  and  $d \mid n$  then  $e \mid n$ , so  $e$  is a divisor of  $n$  and is smaller than  $d$ . But this is a contradiction, because  $d$  was defined as the smallest divisor of  $n$ . Because a contradiction cannot be true, our assumption must be false, and therefore  $d$  must be prime.  $\square$

Don't worry if you find this type of proof a bit confusing; it takes some getting used to.

We can now prove that there are an infinite number of primes.

**Theorem 3 (Euclid).** *There are an infinite number of primes.*

*Proof.* We again assume the opposite of what we try to prove. Here we assume that the number of primes is finite, and therefore that the list of primes is finite. Let's call them  $p_1, p_2, p_3, \dots, p_k$ , where  $k$  is the number of

primes. We define the number  $n := p_1 p_2 p_3 \cdots p_k + 1$ , which is the product of all our primes plus one.

Consider the smallest divisor greater than 1 of  $n$ ; we'll call it  $d$  again. Now  $d$  is prime (by Lemma 2) and  $d \mid n$ . But none of the primes in our finite list of primes is a divisor of  $n$ . After all, they are all divisors of  $n - 1$ , so if you divide  $n$  by one of the  $p_i$ 's in the list you are always left with a remainder of 1. So  $d$  is a prime and it is not in the list. But this is a contradiction, as the list is defined to contain all the primes. Thus, assuming that the number of primes is finite leads to a contradiction. We are left to conclude that the number of primes is infinite.  $\square$

This is basically the proof that Euclid gave over 2000 years ago.

There are many more results on the distribution of primes, but interestingly enough there is no easy formula for the number of primes in a specific interval. Primes seem to occur fairly randomly. There are even very simple conjectures which have never been proven. For example, the Goldbach conjecture is that every even number greater than 2 is the sum of two primes. This is easy to verify with a computer for relatively small even numbers, but mathematicians still don't know whether it is true for all even numbers.

The *fundamental theorem of arithmetic* is also useful to know: any integer greater than 1 can be written in exactly one way as the product of primes (if you disregard the order in which you write the primes). For example,  $15 = 3 \cdot 5$ ;  $255 = 3 \cdot 5 \cdot 17$ ; and  $60 = 2 \cdot 2 \cdot 3 \cdot 5$ . We won't try to prove this here. Check any textbook on number theory if you want to know the details.

## 11.2 Generating Small Primes

Sometimes it is useful to have a list of small primes, so here is the Sieve of Eratosthenes, which is still the best algorithm to generate small primes with.

**function** SMALLPRIMELIST

**input:**  $n$     Limit on primes to generate. Must satisfy  $2 \leq n \leq 2^{20}$ .

**output:**  $P$     List of all primes  $\leq n$ .

*Limit the size of  $n$ . If  $n$  is too large we run out of memory.*

```

assert  $2 \leq n \leq 2^{20}$ 
Initialize a list of flags all set to one.
 $(b_2, b_3, \dots, b_n) \leftarrow (1, 1, \dots, 1)$ 
 $i \leftarrow 2$ 
while  $i^2 \leq n$  do
    We have found a prime  $i$ . Mark all multiples of  $i$  composite.
    for  $j \in 2i, 3i, 4i, \dots, \lfloor n/i \rfloor i$  do
         $b_j \leftarrow 0$ 
    od
    Look for the next prime in our list. It can be shown that this loop
    never results in the condition  $i > n$ , which would access a
    nonexistent  $b_i$ .
    repeat
         $i \leftarrow i + 1$ 
    until  $b_i = 1$ 
od
All our primes are now marked with a one. Collect them in a list.
 $P \leftarrow []$ 
for  $k \in 2, 3, 4, \dots, n$  do
    if  $b_k = 1$  then
         $P \leftarrow P \parallel k$ 
    fi
od
return  $P$ 

```

The algorithm is based on a simple idea. Any composite number  $c$  is divisible by a prime that is smaller than  $c$ . We keep a list of flags, one for each of the numbers up to  $n$ . Each flag indicates whether the corresponding number could be prime. Initially all numbers are marked as potential primes by setting the flag to 1. We start with  $i$  being the first prime 2. Of course, none of the multiples of  $i$  can be prime so we mark  $2i, 3i, 4i$ , etc. as being composite by setting their flag to 0. We then increment  $i$  until we have another candidate prime. Now this candidate is not divisible by any smaller prime, or it would have been marked as a composite already. So the new  $i$  must be the next prime. We keep marking the composite numbers and finding the next prime until  $i^2 > n$ .

It is clear that no prime will ever be marked as a composite, since we only mark a number as a composite when we know a factor of it. (The loop that marks them as composite loops over  $2i, 3i, \dots$ . Each of these terms has a factor  $i$  and therefore cannot be prime.)

Why can we stop when  $i^2 > n$ ? Well, suppose a number  $k$  is composite, and let  $p$  be its smallest divisor greater than 1. We already know that  $p$  is prime (see Lemma 2). Let  $q := k/p$ . We now have  $p \leq q$ ; otherwise,  $q$  would be a divisor of  $k$  smaller than  $p$ , which contradicts the definition of  $p$ . The crucial observation is that  $p \leq \sqrt{k}$ , because if  $p$  were larger than  $\sqrt{k}$  we would have  $k = p \cdot q > \sqrt{k} \cdot q \geq \sqrt{k} \cdot p > \sqrt{k} \cdot \sqrt{k} = k$ . This last inequality would show that  $k > k$  which is an obvious fallacy. So  $p \leq \sqrt{k}$ .

We have shown that any composite  $k$  is divisible by a prime  $\leq \sqrt{k}$ . So any composite  $\leq n$  is divisible by a prime  $\leq \sqrt{n}$ . When  $i^2 > n$  then  $i > \sqrt{n}$ . But we have already marked the multiples of all the primes less than  $i$  as composite in the list, so every composite  $< n$  has already been marked as such. The numbers in the list that are still marked as primes are really prime.

The final part of the algorithm simply collects them in a list to be returned.

There are several optimizations you can make to this algorithm, but we have left them out to make things simpler. Properly implemented, this algorithm is very fast.

You might wonder why we need the small primes. It turns out that small primes are useful to generate large primes with, something we will get to soon.

### 11.3 Computations Modulo a Prime

The main reason why primes are so useful in cryptography is that you can compute modulo a prime.

Let  $p$  be a prime. When we compute modulo a prime we only use the numbers  $0, 1, \dots, p-1$ . The basic rule for computations modulo a prime is to do the computations using the numbers as integers, just as you normally would, but every time you get a result  $r$  you take it modulo  $p$ . Taking a

modulo is easy: just divide the result  $r$  by  $p$ , throw away the quotient, and keep the remainder as the answer. For example, if you take 25 modulo 7 you divide 25 by 7, which gives us a quotient of 3 with a remainder of 4. The remainder is the answer, so  $(25 \bmod 7) = 4$ . The notation  $(a \bmod b)$  is used to denote an explicit modulo operation, but as modulo computations are used very often, and mathematicians are rather lazy, there are several other notations in general use as well. Often the entire equation will be written without any modulo operations, and then  $(\bmod p)$  will be added at the end of the equation to remind you that the whole thing is to be taken modulo  $p$ . When the situation is clear from the context even this is left out, and you have to remember the modulo yourself.

You don't need to write parentheses around a modulo computation. We could just as well have written  $a \bmod b$ , but as the modulo operator looks very much like normal text this can be a bit confusing for people who are not used to it. To avoid confusion we tend to put  $(a \bmod b)$  in parentheses.

One word of warning: Any integer taken modulo  $p$  is always in the range  $0, \dots, p - 1$ , even if the original integer is negative. Some programming languages have the (for mathematicians very irritating) property that they allow negative results from a modulo operation. If you want to take  $-1$  modulo  $p$ , then the answer is  $p - 1$ . More generally: to compute  $(a \bmod p)$ , find integers  $q$  and  $r$  such that  $a = qp + r$  and  $0 \leq r < p$ . The value of  $(a \bmod p)$  is defined to be  $r$ . If you fill in  $a = -1$  then you find that  $q = -1$  and  $r = p - 1$ .

### 11.3.1 Addition and Subtraction

Addition modulo  $p$  is easy. Just add the two numbers, and subtract  $p$  if the result is greater than or equal to  $p$ . As both inputs are in the range  $0, \dots, p - 1$ , the sum cannot exceed  $2p - 1$ , so you have to subtract  $p$  at most once to get the result back in the proper range.

Subtraction is similar to addition. Subtract the numbers, and add  $p$  if the result is negative.

These rules only work when the two inputs are both modulo  $p$  numbers already. If they are outside the range, you have to do a full reduction modulo  $p$ .

It takes a while to get used to modulo computations. You get equations like  $5 + 3 = 1 \pmod{7}$ . This looks odd at first. You know that 5 plus 3 is not 1. But while  $5 + 3 = 8$  is true in the integer numbers, working modulo 7 we have  $8 \bmod 7 = 1$ , so  $5 + 3 = 1 \pmod{7}$ .

We use modulo arithmetic in real life quite often without realizing it. When computing the time of day, we take the hours modulo 12 (or modulo 24). A bus schedule might state that the bus leaves at 55 minutes past the hour and takes 15 minutes. To find out when the bus arrives, we compute  $55 + 15 = 70 \pmod{60}$ , and determine it arrives at 10 minutes past the hour. For now we will restrict ourselves to computing modulo a prime, but you can do computations modulo any number you like.

### 11.3.2 Multiplication

Multiplication is, as always, more work than addition. To compute  $(ab \bmod p)$  you first compute  $ab$  as an integer, and then take the result modulo  $p$ . Now  $ab$  can be as large as  $(p-1)^2 = p^2 - 2p + 1$ . Here you have to perform a long division to find  $(q, r)$  such that  $ab = qp + r$  and  $0 \leq r < p$ . Throw away the  $q$ ; the  $r$  is the answer.

Let's give you an example: Let  $p = 5$ . When we compute  $3 \cdot 4 \pmod{5}$  the result is 2. After all,  $3 \cdot 4 = 12$ , and  $(12 \bmod 5) = 2$ . So we get  $3 \cdot 4 = 2 \pmod{5}$ .

### 11.3.3 Groups and Finite Fields

Mathematicians call the set of numbers modulo a prime  $p$  a *finite field*, and often refer to it as the “mod  $p$ ” field, or simply “mod  $p$ .” Here are some useful reminders about computations in a mod  $p$  field:

- You can always add or subtract any multiple of  $p$  from your numbers without changing the result.
- All results are always in the range  $0, 1, \dots, p - 1$ .

- You can think of it as doing your entire computation in the integers and only taking the modulo at the very last moment. So all the algebraic rules you learned about the integers (such as  $a(b + c) = ab + ac$ ) still apply.

The finite field of the integers modulo  $p$  is referred to using different notations in different books. We will use the notation  $\mathbb{Z}_p$  to refer to the finite field modulo  $p$ . In other texts you might see  $\text{GF}(p)$  or even  $\mathbb{Z}/p\mathbb{Z}$ .

We also have to introduce the concept of a *group*—another mathematical term, but a simple one. A group is simply a set of numbers together with an operation, such as addition or multiplication.<sup>2</sup> The numbers in  $\mathbb{Z}_p$  form a group together with addition. You can add any two numbers and get a third number in the group. If you want to use multiplication in a group you cannot use the 0. (This has to do with the fact that multiplying by 0 is not very interesting, and that you cannot divide by 0.) However, the numbers  $1, \dots, p - 1$  together with multiplication modulo  $p$  form a group. This group is called the *multiplicative group modulo  $p$* , and is written in various ways; we will use the notation  $\mathbb{Z}_p^*$ . A finite field consists of two groups: the addition group and the multiplication group. In the case of  $\mathbb{Z}_p$  the finite field consists of the addition group, defined by addition modulo  $p$ , and the multiplication group  $\mathbb{Z}_p^*$ .

A group can contain a *subgroup*. A subgroup consists of some of the elements of the full group. If you apply the group operation to two elements of the subgroup, you again get an element of the subgroup. That sounds complicated, so here is an example. The numbers modulo 8 together with addition (modulo 8) form a group. The numbers  $\{0, 2, 4, 6\}$  form a subgroup. You can add any two of these numbers modulo 8 and get another element of the subgroup. The same goes for multiplicative groups. The multiplicative subgroup modulo 7 consists of the numbers  $1, \dots, 6$ , and the operation is multiplication modulo 7. The set  $\{1, 6\}$  forms a subgroup, as does the set  $\{1, 2, 4\}$ . You can check that if you multiply any two elements from the same subgroup modulo 7, you get another element from that subgroup.

We use subgroups to speed up certain cryptographic operations. They can also be used to attack systems, which is why you need to know about them.

---

<sup>2</sup>There are a couple of further requirements, but they are all met by the groups we will be talking about.

So far we've only talked about addition, subtraction, and multiplication modulo a prime. To fully define a multiplicative group you also need the inverse operation of multiplication: division. It turns out that you can define division on the numbers modulo  $p$ . The simple definition is that  $a/b \pmod{p}$  is a number  $c$  such that  $c \cdot b = a \pmod{p}$ . You cannot divide by zero, but it turns out that the division  $a/b \pmod{p}$  is always well defined as long as  $b \neq 0$ .

So how do you compute the quotient of two numbers modulo  $p$ ? This is more complicated and it will take a few pages to explain. We first have to go back more than 2000 years to Euclid again, and to his algorithm for the GCD.

### 11.3.4 The GCD Algorithm

Another high-school math refresher course: The *greatest common divisor* (or GCD) of two numbers  $a$  and  $b$  is the largest  $k$  such that  $k \mid a$  and  $k \mid b$ . In other words,  $\text{gcd}(a, b)$  is the largest number that divides both  $a$  and  $b$ .

Euclid gave an algorithm for computing the GCD of two numbers which is still in use today, thousands of years later. For a detailed discussion of this algorithm see Knuth [55].

**function** GCD

**input:**  $a$     Positive integer.

$b$     Positive integer.

**output:**  $k$     The greatest common divisor of  $a$  and  $b$ .

**assert**  $a \geq 0 \wedge b \geq 0$

**while**  $a \neq 0$  **do**

$(a, b) \leftarrow (b \bmod a, a)$

**od**

**return**  $b$

Why would this work? The first observation is that the assignment does not change the set of common divisors of  $a$  and  $b$ . After all,  $(b \bmod a)$  is just  $b - sa$  for some integer  $s$ . Any number  $k$  that divides both  $a$  and  $b$  will also divide both  $a$  and  $(b \bmod a)$ . (The converse is also true.) And when  $a = 0$ , then  $b$  is a common divisor of  $a$  and  $b$ , and  $b$  is obviously the largest such

common divisor. You can check for yourself that the loop must terminate because  $a$  and  $b$  keep getting smaller and smaller until they reach zero.

Let's compute the GCD of 21 and 30 as an example. We start with  $(a, b) = (21, 30)$ . In the first iteration we compute  $(30 \bmod 21) = 9$ , so we get  $(a, b) = (9, 21)$ . In the next iteration we compute  $(21 \bmod 9) = 3$ , so we get  $(a, b) = (3, 9)$ . In the final iteration we compute  $(9 \bmod 3) = 0$  and get  $(a, b) = (0, 3)$ . The algorithm will return 3, which is indeed the greatest common divisor of 21 and 30.

The GCD has a cousin: the LCM or *least common multiple*. The LCM of  $a$  and  $b$  is the smallest number that is both a multiple of  $a$  and a multiple of  $b$ . For example,  $\text{lcm}(6, 8) = 24$ . The GCD and LCM are tightly related by the equation

$$\text{lcm}(a, b) = \frac{ab}{\text{gcd}(a, b)}$$

which we won't prove here but just state as a fact.

### 11.3.5 The Extended Euclidean Algorithm

This still does not help us to compute division modulo  $p$ . For that we need what is called the extended Euclidean algorithm. The idea is that while computing  $\text{gcd}(a, b)$  we can also find two integers  $u$  and  $v$  such that  $\text{gcd}(a, b) = ua + vb$ . This will allow us to compute  $a/b \pmod{p}$ .

**function** EXTENDEDGCD

**input:**  $a$     Positive integer argument.

$b$     Positive integer argument.

**output:**  $k$     The greatest common divisor of  $a$  and  $b$ .

$(u, v)$     Integers such that  $ua + vb = k$ .

**assert**  $a \geq 0 \wedge b \geq 0$

$(c, d) \leftarrow (a, b)$

$(u_c, v_c, u_d, v_d) \leftarrow (1, 0, 0, 1)$

**while**  $c \neq 0$  **do**

*Invariant:*  $u_c a + v_c b = c \wedge u_d a + v_d b = d$

$q \leftarrow \lfloor d/c \rfloor$

$(c, d) \leftarrow (d - qc, c)$

```

     $(u_c, v_c, u_d, v_d) \leftarrow (u_d - qu_c, v_d - qv_c, u_c, v_c)$ 
od
return  $d, (u_d, v_d)$ 

```

This algorithm is very much like the GCD algorithm. We introduce new variables  $c$  and  $d$  instead of using  $a$  and  $b$  because we need to refer to the original  $a$  and  $b$  in our invariant. If you only look at  $c$  and  $d$ , this is exactly the GCD algorithm. (We've rewritten the  $d \bmod c$  formula slightly, but this gives the same result.) We have added four variables that maintain the given invariant; for each value of  $c$  or  $d$  that we generate, we keep track of how to express that value as a linear combination of  $a$  and  $b$ . For the initialization this is easy, as  $c$  is initialized to  $a$  and  $d$  to  $b$ . When we modify  $c$  and  $d$  in the loop it is not terribly difficult to update the  $u$  and  $v$  variables.

Why bother with the extended Euclidean algorithm? Well, suppose we want to compute  $1/b \bmod p$  where  $1 \leq b < p$ . We use the extended Euclidean algorithm to compute  $\text{EXTENDEDGCD}(b, p)$ . Now, we know that the GCD of  $b$  and  $p$  is 1, because  $p$  is prime and it therefore has no other suitable divisors. But the  $\text{EXTENDEDGCD}$  function also provides two numbers  $u$  and  $v$  such that  $ub + vp = \text{gcd}(b, p) = 1$ . In other words,  $ub = 1 - vp$  or  $ub = 1 \pmod{p}$ . This is the same as saying that  $u = 1/b \pmod{p}$ , the inverse of  $b$  modulo  $p$ . The division  $a/b$  can now be computed by multiplying  $a$  by  $u$ , so we get  $a/b = au \pmod{p}$ , and this last formula is something that we know how to compute.

The extended Euclidean algorithm allows us to compute an inverse modulo a prime, which in turn allows us to compute a division modulo  $p$ . Together with the addition, subtraction, and multiplication modulo  $p$ , this allows us to compute all four elementary operations in the finite field modulo  $p$ .

Note that  $u$  could be negative, so it is probably a good idea to reduce  $u$  modulo  $p$  before using it as the inverse of  $b$ .

If you look carefully at the  $\text{EXTENDEDGCD}$  algorithm, you'll see that if you only want  $u$  as output, you can leave out the  $v_c$  and  $v_d$  variables, as they do not affect the computation of  $u$ . This slightly reduces the amount of work needed to compute a division modulo  $p$ .

+	0	1
0	0	1
1	1	0

·	0	1
0	0	0
1	0	1

Figure 11.1: Addition and multiplication modulo 2

### 11.3.6 Working Modulo 2

An interesting special case is computation modulo 2. After all, 2 is a prime, so we should be able to compute modulo it. If you've done any programming this might look familiar to you. The addition and multiplication tables modulo 2 are shown in figure 11.1. Addition modulo 2 is exactly the exclusive-or (XOR) function you find in programming languages. Multiplication is just a simple AND operation. In the field modulo 2 there is only one inversion possible ( $1/1 = 1$ ) so division is the same operation as multiplication. It shouldn't surprise you that the field  $\mathbb{Z}_2$  is an important tool to analyze certain algorithms used by computers.

## 11.4 Large Primes

Several cryptographic primitives use very large primes, and we're talking about many hundreds of digits here. Don't worry, you won't have to compute with these primes by hand. That's what the computer is for.

To do any computations at all with numbers this large, you need a multi-precision library. You cannot use floating-point numbers, because they do not have several hundred digits of precision. You cannot use normal integers, because in most programming languages they are limited to a dozen digits or so. Few programming languages provide native support for arbitrary precision integers. Writing routines to perform computations with large integers is fascinating. For a good overview, see Knuth [55, section 4.3]. However, implementing a multiprecision library is far more work than you might expect. Not only do you have to get the right answer, but you always strive to compute it as quickly as possible. There are quite a number of special situations you have to deal with carefully. Save your time for

more important things, and download one of the many free libraries from the Internet, or use a language like Python that has built-in large integer support.

For public-key cryptography, the primes we want to generate are 2000–4000 bits long. The basic method of generating a prime that large is surprisingly simple: take a random number and check whether it is prime. There are very good algorithms to determine whether a large number is prime or not. There are also very many primes. In the neighborhood of a number  $n$ , approximately one in every  $\ln n$  numbers is prime. (The natural logarithm of  $n$ , or  $\ln n$  for short, is one of the standard functions on any scientific calculator. To give you an idea of how slowly the logarithm grows when applied to large inputs: the natural logarithm of  $2^k$  is slightly less than  $0.7 \cdot k$ .) A number that is 2000 bits long falls between  $2^{1999}$  and  $2^{2000}$ . In that range about one in every 1386 of the numbers is prime. And this includes a lot of numbers that are trivially composite, such as the even numbers.

Generating a large prime looks something like this:

**function** GENERATELARGEPRIME

**input:**  $l$      Lower bound of range in which prime should lie.  
            $u$      Upper bound of range in which prime should lie.

**output:**  $p$      A random prime in the interval  $l, \dots, u$

*Check for a sensible range.*

**assert**  $2 < l \leq u$

*Compute maximum number of attempts*

$r \leftarrow 100(\lfloor \log_2 u \rfloor + 1)$

**repeat**

$r \leftarrow r - 1$

**assert**  $r > 0$

*Choose  $n$  randomly in the right interval*

$n \in_{\mathcal{R}} l, \dots, u$

*Continue trying until we find a prime.*

**until** ISPRIME( $n$ )

**return**  $n$

We use the operator  $\in_{\mathcal{R}}$  to indicate a random selection from a set. Of course, this requires some output from the PRNG.

The algorithm is relatively straightforward. We first check that we get a sensible interval. The cases  $l \leq 2$  and  $l \geq u$  are not useful and lead to problems. Note the boundary condition: the case  $l = 2$  is not allowed.<sup>3</sup> Next we compute how many attempts we are going to make to find a prime. There are intervals that do not contain a prime. For example, the interval  $90, \dots, 96$  is prime-free. A proper program should never hang, independent of its inputs, so we limit the number of tries and generate a failure if we exceed this number. How many times should we try? As stated before, in the neighborhood of  $u$  about one in every  $0.7 \log_2 u$  numbers is prime. (The function  $\log_2$  is the logarithm to the base 2. The simplest definition is that  $\log_2(x) := \ln x / \ln 2$ . The number  $\log_2 u$  is difficult to compute but  $\lfloor \log_2 u \rfloor + 1$  is much easier; it is the number of bits necessary to represent  $u$  as a binary number. So if  $u$  is an integer that is 2017 bits long, then  $\lfloor \log_2 u \rfloor + 1 = 2017$ . The factor 100 ensures that it is extremely unlikely that we will not find a prime. For large enough intervals, the probability of a failure due to bad luck is less than  $2^{-128}$ , so we can ignore this risk. At the same time, this limit does ensure that the `GENERATELARGEPRIME` function will terminate. We've been a bit sloppy in our use of an assertion to generate the failure; a proper implementation would generate an error with explanations of what went wrong.

The main loop is simple. After the check that limits the number of tries, we choose a random number and check whether it is prime using the `ISPRIME` function. We will define this function shortly.

Make sure that the number  $n$  you choose is uniformly randomly in the range  $l, \dots, u$ . Also make sure that the range is not too small if you want your prime to be a secret. If the attacker knows the interval you use, and there are fewer than  $2^{128}$  primes in that interval, the attacker could potentially try them all.

If you wish, you can make sure the random number you generate is odd by setting the least significant bit just after you generate a candidate  $n$ .

---

<sup>3</sup>The Rabin-Miller algorithm we use below does not work well when it gets 2 as an argument. That's okay, we already know that 2 is prime so we don't have to generate it here.

As 2 is not in your interval, this will not affect the probability distribution of primes you are choosing, and it will halve the number of attempts you have to make. But this is only safe if  $u$  is odd, otherwise setting the least significant bit might bump  $n$  just outside the allowed range.

The `ISPRIME` function is a two-step filter. The first phase is a simple test where we try to divide  $n$  by all the small primes. This will quickly weed out the great majority of numbers which are composite and divisible by a small prime. If we find no divisors, we employ a heavyweight test called the Rabin-Miller test.

```
function ISPRIME
input:   $n$       Integer  $\geq 3$ .
output:  $b$       Boolean whether  $n$  is prime.
    assert  $n \geq 3$ 
    for  $p \in \{ \text{all primes } \leq 1000 \}$  do
        if  $p$  is a divisor of  $n$  then
            return  $p = n$ 
        fi
    od
    return RABIN-MILLER( $n$ )
```

If you are lazy and don't want to generate the small primes, you can cheat a bit. Instead of trying all the primes, you can try 2 and all odd numbers 3, 5, 7, ..., 999, in that order. This sequence contains all the primes below 1000, but it also contains a lot of useless composite numbers. The order is important to ensure that a small composite number like 9 is properly detected as being composite. The bound of 1000 is arbitrary, and can be chosen for optimal performance.

All that remains to explain is the mysterious Rabin-Miller test that does the hard work.

### 11.4.1 Primality Testing

It turns out to be remarkably easy to test whether a number is prime. At least, it is remarkably easy compared to factoring a number and finding its prime divisors. These easy tests are not perfect. They are all probabilistic.

There is a certain chance they give the wrong answer. By repeatedly running the same test we can reduce the probability of error to an acceptable level.

The primality test of choice is the Rabin-Miller test. The mathematical basis for this test is well beyond the scope of this book, although the outline is fairly simple. The purpose of this test is to determine whether an odd integer  $n$  is prime. We choose a random value  $a$  less than  $n$ , called the *basis*, and check a certain property of  $a$  modulo  $n$  that always hold when  $n$  is prime. However, you can prove that when  $n$  is not a prime, this property holds for at most 25% of all possible basis values. By repeating this test for different random values of  $a$ , you build your confidence in the final result. If  $n$  is a prime, it will always test as a prime. If  $n$  is not a prime, then at least 75% of the possible values for  $a$  will show so, and the chance that  $n$  will pass multiple tests can be made as small as you want. We limit the probability of a false result to  $2^{-128}$  to achieve our required security level.

Here is how it goes:

**function** RABIN-MILLER

**input:**  $n$     An odd number  $\geq 3$ .

**output:**  $b$     Boolean indicating whether  $n$  is prime or not.

**assert**  $n \geq 3 \wedge n \bmod 2 = 1$

*First we compute  $(s, t)$  such that  $s$  is odd and  $2^t s = n - 1$ .*

$(s, t) \leftarrow (n - 1, 0)$

**while**  $s \bmod 2 = 0$  **do**

$(s, t) \leftarrow (s/2, t + 1)$

**od**

*We keep track of the probability of a false result in  $k$ . The probability is at most  $2^{-k}$ . We loop until the probability of a false result is small enough.*

$k \leftarrow 0$

**while**  $k < 128$  **do**

*Choose a random  $a$  such that  $2 \leq a \leq n - 1$ .*

$a \in_{\mathcal{R}} 2, \dots, n - 1$

*The expensive operation: a modular exponentiation.*

$v \leftarrow a^s \bmod n$

*When  $v = 1$ , the number  $n$  passes the test for basis  $a$ .*

**if**  $v \neq 1$  **then**

*The sequence  $v, v^2, \dots, v^{2^t}$  must finish on the value 1, and the last value not equal to 1 must be  $n - 1$  if  $n$  is a prime.*

```

i ← 0
while v ≠ n - 1 do
  if i = t - 1 then
    return false
  else
    (v, i) ← (v2 mod n, i + 1)
  fi
od
fi

```

*When we get to this point,  $n$  has passed the primality test for the basis  $a$ . We have therefore reduced the probability of a false result by a factor of  $2^2$ , so we can add 2 to  $k$ .*

```

k ← k + 2
od
return true

```

This algorithm only works for an odd  $n$  greater or equal to 3, so we test that first. The ISPRIME function should only call this function with a suitable argument, but each function is responsible for checking its own inputs and outputs. You never know how the software will be changed in future.

The basic idea behind the test is known as Fermat's little theorem.<sup>4</sup> For any prime  $n$  and for all  $1 \leq a < n$ , the relation  $a^{n-1} \bmod n = 1$  holds. To fully understand the reasons for this requires more math than we will explain here. A simple test (also called the Fermat primality test) verifies this relation for a number of randomly chosen  $a$  values. Unfortunately, there are some obnoxious numbers called the Carmichael numbers. These are composite but they pass the Fermat test for (almost) all basis  $a$ .

The Rabin-Miller test is a variation of the Fermat test. First we write  $n - 1$  as  $2^t s$ , where  $s$  is an odd number. If you want to compute  $a^{n-1}$  you can first compute  $a^s$  and then square the result  $t$  times to get  $a^{s \cdot 2^t} = a^{n-1}$ . Now if  $a^s = 1 \pmod{n}$  then repeated squaring will not change the result so we

---

<sup>4</sup>There are several theorems named after Fermat. Fermat's last Theorem is the most famous one, involving the equation  $a^n + b^n = c^n$  and a proof too small to fit in the margin of the page.

have  $a^{n-1} = 1 \pmod{n}$ . If  $a^s \neq 1 \pmod{n}$ , then we look at the numbers  $a^s, a^{s \cdot 2}, a^{s \cdot 2^2}, a^{s \cdot 2^3}, \dots, a^{s \cdot 2^t}$  (all modulo  $n$ , of course). If  $n$  is a prime, then we know that the last number must be 1. If  $n$  is a prime, then the only numbers that satisfy  $x^2 = 1 \pmod{n}$  are 1 and  $n - 1$ .<sup>5</sup> So if  $n$  is prime, then one of the numbers in the sequence must be  $n - 1$ , or we could never have the last number be equal to 1. This is really all the Rabin-Miller test checks. If any choice of  $a$  demonstrates that  $n$  is composite, we return immediately. If  $n$  continues to test as a prime, we repeat the test for different  $a$  values until the probability that we have generated a wrong answer and claimed that a composite number is actually prime is less than  $2^{-128}$ .

If you apply this test to a random number, the probability of failure of this test is much, much smaller than the bound we use. For almost all composite numbers  $n$ , almost all basis values will show that  $n$  is composite. You will find a lot of libraries that depend on this and perform the test for only 5 or 10 bases or so. This idea is fine, though we would have to investigate how many attempts are needed to reach an error level of  $2^{-128}$  or less. But it only holds as long as you apply the ISPRIME test to *randomly* chosen numbers. Later on we will encounter situations where we apply the primality test to numbers that we received from someone else. These might be maliciously chosen, so the ISPRIME function must achieve a  $2^{-128}$  error bound all by itself.

Doing the full 64 Rabin-Miller tests is necessary when we receive the number to be tested from someone else. It is overkill when we try to generate a prime randomly. But when generating a prime, you spend most of your time rejecting composite numbers. (Almost all composite numbers are rejected by the very first Rabin-Miller test that you do.) As you might have to try hundreds of numbers before you find a prime, doing 64 tests on the final prime is only marginally slower than doing 10 of them.

In an earlier version of this chapter, the Rabin-Miller routine had a second argument that could be used to select the maximum error probability. But it was a perfect example of a needless option, so we removed it. Always doing a good test to a  $2^{-128}$  bound is simpler, and much less likely to be improperly used.

---

<sup>5</sup>It is easy to check that  $(n - 1)^2 = 1 \pmod{n}$ .

There is still a chance of  $2^{-128}$  that our ISPRIME function will give you the wrong answer. To give you an idea of how small this chance actually is, the chance that you will be killed by a meteorite while you read this sentence is far larger. Still alive? Okay, so don't worry about it.

### 11.4.2 Evaluating Powers

The Rabin-Miller test spends most of its time computing  $a^s \bmod n$ . You cannot compute  $a^s$  first and then take it modulo  $n$ . No computer in the world has enough memory to even store  $a^s$ , much less the computing power to compute it; both  $a$  and  $s$  can be thousands of bits long. But we only need  $a^s \bmod n$ ; we can apply the mod  $n$  to all the intermediate results, which stops the numbers from growing too large.

There are several ways of computing  $a^s \bmod n$ , but here is a simple description. To compute  $a^s \bmod n$  use the following rules:

- If  $s = 0$  the answer is 1.
- If  $s > 0$  and  $s$  is even, then first compute  $y := a^{s/2} \bmod n$  using these very same rules. The answer is given by  $a^s \bmod n = y^2 \bmod n$ .
- If  $s > 0$  and  $s$  is odd, then first compute  $y := a^{(s-1)/2} \bmod n$  using these very same rules. The answer is given by  $a^s \bmod n = a \cdot y^2 \bmod n$ .

This is a recursive formulation of the so-called binary algorithm. If you look at the operations performed, it builds up the desired exponent bit by bit from the most significant part of the exponent down to the least significant part. It is also possible to convert this from a recursive algorithm to a loop.

How many multiplications are required to compute  $a^s \bmod n$ ? Let  $k$  be the number of bits of  $s$ ; i.e.,  $2^{k-1} \leq s < 2^k$ . Then this algorithm requires at most  $2k$  multiplications modulo  $n$ . This is not too bad. If we are testing a 2000-bit number for primality, then  $s$  will also be about 2000 bits long and we only need 4000 multiplications. That is still a lot of work, but certainly within the capabilities of most desktop computers.

Many public-key cryptographic systems make use of modular exponentiations like this. Any good multiprecision library will have an optimized

---

routine for evaluating modular exponentiations. A special type of multiplication called Montgomery multiplication is well suited for this task. There are also ways of computing  $a^s$  using fewer multiplications [10, Ch. 4]. Each of these tricks can save 10%–30% of the time it takes to compute a modular exponentiation, so used in combination they can be important.

Straightforward implementations of modular exponentiation are often vulnerable to timing attacks. See section 16.3 for details and possible remedies.



## Chapter 12

# Diffie-Hellman

For the presentation of public-key cryptography we're going to follow the historical path. Public-key cryptography was really started by Whitfield Diffie and Martin Hellman when they published their "New Directions in Cryptography" article in 1976 [22].

So far in this book we've only talked about encryption and authentication with shared secret keys. But where do we get those shared secret keys from? If you have 10 friends you want to communicate with, you can meet them all and exchange a secret key with each of these friends for future use. But like all keys, these keys should be refreshed regularly, so then you have to meet and exchange keys all over again. A total of 45 keys are needed for a group of 10 friends. But as the group gets larger, the number of keys grows quadratically. For 100 people all communicating with each other, you need 4950 keys. This quickly becomes unmanageable.

Diffie and Hellman posed the question of whether it would be possible to do this more efficiently. Suppose you have an encryption algorithm where the encryption and decryption keys are different. You can publish your encryption key and keep your decryption key secret. Anyone can now send you an encrypted message, and only you can decrypt it. This would solve the problem of having to distribute so many different keys.

Diffie and Hellman posed the question, but they could only provide a partial

answer. Their partial solution is today known as the Diffie-Hellman key exchange protocol, often shortened to DH protocol [22].

The DH protocol is a really nifty idea. It turns out that two people communicating over an insecure line can agree on a secret key in such a way that both of them receive the same key without divulging it to someone who is listening in on their conversation.

## 12.1 Groups

If you've read the last chapter, it won't surprise you that primes are involved. For the rest of this chapter,  $p$  is a large prime. Think of  $p$  as being 2000 to 4000 bits long. Most of our computations in this chapter will be modulo  $p$ —in many places we will not specify this again explicitly. The DH protocol uses  $\mathbb{Z}_p^*$ , the multiplicative group modulo  $p$  that we discussed in section 11.3.3.

Choose any  $g$  in the group and consider the numbers  $1, g, g^2, g^3, \dots$ , all modulo  $p$ , of course. This is an infinite sequence of numbers, but there is only a finite set of numbers in  $\mathbb{Z}_p^*$ . (Remember,  $\mathbb{Z}_p^*$  is the numbers  $1, \dots, p-1$  together with the operation of multiplication modulo  $p$ .) At some point the numbers must start to repeat. Let us assume this happens at  $g^i = g^j$  with  $i < j$ . As we can do divisions modulo  $p$ , we can divide each side by  $g^i$  and get  $1 = g^{j-i}$ . In other words, there is a number  $q := j - i$  such that  $g^q = 1 \pmod{p}$ . We call the smallest positive value  $q$  for which  $g^q = 1 \pmod{p}$  the *order* of  $g$ . (Unfortunately, there is quite a bit of terminology associated with this stuff. We feel it is better to use the standard terminology than to invent our own words; otherwise readers will be confused later on when they read other books.)

If we keep on multiplying  $gs$  we can reach the numbers  $1, g, g^2, \dots, g^{q-1}$ . After that, the sequence repeats as  $g^q = 1$ . We say that  $g$  is a generator and that it generates the set  $1, g, g^2, \dots, g^{q-1}$ . The number of elements that can be written as a power of  $g$  is exactly  $q$ , the order of  $g$ .

One property of multiplication modulo  $p$  is that there is at least one  $g$  that generates the entire group. That is, there is at least one  $g$  value for which  $q = p - 1$ . So instead of thinking of  $\mathbb{Z}_p^*$  as the numbers  $1, \dots, p - 1$ , we can

also think of them as  $1, g, g^2, \dots, g^{p-2}$ . A  $g$  that generates the whole group is called a *primitive element* of the group.

Other values of  $g$  can generate smaller sets. Observe that if we multiply two numbers from the set generated by  $g$ , then we get another power of  $g$ , and therefore another element from the set. If you go through all the math, it turns out that the set generated by  $g$  is another group. That is, you can multiply and divide in this group just as you can in the large group modulo  $p$ . These smaller groups are called subgroups (see section 11.3.3). They will be important in various attacks.

There is one last thing to explain. For any element  $g$ , the order of  $g$  is a divisor of  $p - 1$ . This isn't too hard to see. Choose  $g$  to be a primitive element. Let  $h$  be any other element. As  $g$  generates the whole group, there is an  $x$  such that  $h = g^x$ . Now consider the elements generated by  $h$ . These are  $1, h, h^2, h^3, \dots$  which are equal to  $1, g^x, g^{2x}, g^{3x}, \dots$  (All our computations are still modulo  $p$ , of course.) The order of  $h$  is the smallest  $q$  at which  $h^q = 1$ , which is the same as saying that it is the smallest  $q$  such that  $g^{xq} = 1$ . For any  $t$ ,  $g^t = 1$  is the same as saying  $t = 0 \pmod{p - 1}$ . So  $q$  is the smallest  $q$  such that  $xq = 0 \pmod{p - 1}$ . This happens when  $q = (p - 1) / \gcd(x, p - 1)$ . So  $q$  is obviously a factor of  $p - 1$ .

Here's a simple example. Let's choose  $p = 7$ . If we choose  $g = 3$  then  $g$  is a generator because  $1, g, g^2, \dots, g^6 = 1, 3, 2, 6, 4, 5$ . (Again, all computations modulo  $p$ .) The element  $h = 2$  generates the subgroup  $1, h, h^2 = 1, 2, 4$  because  $h^3 = 2^3 \pmod{7} = 1$ . The element  $h = 6$  generates the subgroup  $1, 6$ . These subgroups have sizes 3 and 2 respectively, which are both divisors of  $p - 1$ .

This also explains parts of the Fermat test we talked about in section 11.4.1. Fermat's test is based on the fact that for any  $a$  we have  $a^{p-1} = 1$ . This is easy to check. Let  $g$  be a generator of  $\mathbb{Z}_p^*$ , and let  $x$  be such that  $g^x = a$ . As  $g$  is a generator of the whole group, there is always such an  $x$ . But now  $a^{p-1} = g^{x(p-1)} = (g^{p-1})^x = 1^x = 1$ .

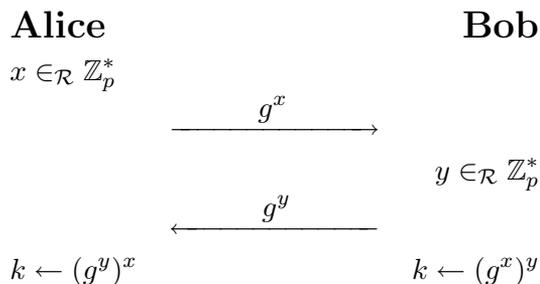


Figure 12.1: The original Diffie-Hellman protocol.

## 12.2 Basic DH

For the original DH protocol, we first choose a large prime  $p$ , and a primitive element  $g$  which generates the whole group  $\mathbb{Z}_p^*$ . Both  $p$  and  $g$  are public constants in this protocol, and we assume that all parties, including the attackers, know them. The protocol is shown in figure 12.1. This is one of the usual ways in which we write cryptographic protocols. There are two parties involved: Alice and Bob. Time progresses from the top to the bottom. First Alice chooses a random  $x$  in  $\mathbb{Z}_p^*$ , which is the same as choosing a random number in  $1, \dots, p-1$ . She computes  $g^x \bmod p$  and sends the result to Bob. Bob in turn chooses a random  $y$  in  $\mathbb{Z}_p^*$ . He computes  $g^y \bmod p$  and sends the result to Alice. The final result  $k$  is defined as  $g^{xy}$ . Alice can compute this by raising the  $g^y$  she got from Bob to the power  $x$  that she knows. (High-school math:  $(g^y)^x = g^{xy}$ .) Similarly, Bob can compute  $k$  as  $(g^x)^y$ . They both end up with the same value  $k$  which they can use as a secret key.

But what about an attacker? The attacker gets to see  $g^x$  and  $g^y$ , but not  $x$  or  $y$ . The problem of computing  $g^{xy}$  given  $g^x$  and  $g^y$  is known as the Diffie-Hellman problem, or DH problem for short. As long as  $p$  and  $g$  are chosen correctly, there is no efficient algorithm to compute this—at least, there is none that we know of. The best method known is to first compute  $x$  from  $g^x$ , after which the attacker can compute  $k$  as  $(g^y)^x$  just like Alice did. In the real numbers, computing  $x$  from  $g^x$  is called the logarithm function, which you find on any scientific calculator. In the finite field  $\mathbb{Z}_p^*$ , it is called

a *discrete logarithm*, and in general the problem of computing  $x$  from  $g^x$  in a finite group is known as the discrete logarithm problem, or DL problem.

The original DH protocol can be used in many ways. We've written it as an exchange of messages between two parties. Another way of using it is to let everybody choose a random  $x$ , and publish  $g^x \pmod{p}$  in the digital equivalent of a phone book. If Alice now wants to communicate with Bob securely, she gets  $g^y$  from the phone book, and using her  $x$ , computes  $g^{xy}$ . Bob can similarly compute  $g^{xy}$  without any interaction with Alice. This makes the system usable in settings such as e-mail where there is no direct interaction.

## 12.3 Man in the Middle

The one thing that DH does not protect against is the man in the middle. Look back at the protocol. Alice knows she is communicating with somebody, but she does not know whom she is communicating with. Eve can sit in the middle of the protocol and pretend to be Bob when speaking to Alice, and pretend to be Alice when speaking to Bob. This is shown in figure 12.2. To Alice, this protocol looks just like the original DH protocol. There is no way in which Alice can detect she is talking to Eve, not Bob. The same holds for Bob. Eve can keep up these pretenses for as long as she likes. Suppose Alice and Bob start to communicate using the secret key they think they have set up. All Eve needs to do is forward all the communications between Alice and Bob. Of course, Eve has to decrypt all the data she gets from Alice that was encrypted with key  $k$ , and then encrypt it again with key  $k'$  to send to Bob. She has to do the same with the traffic in the other direction, but that is not a lot of work.

With a digital phone book this attack is harder. As long as the publisher of the book verifies the identity of everybody when they send in their  $g^x$ , Alice knows she is using Bob's  $g^x$ . We'll discuss other solutions when we talk about digital signatures and PKIs later on in this book.

There is one setting where the man-in-the-middle attack can be addressed without further infrastructure. If the key  $k$  is used to encrypt a phone conversation (or a video link), Alice can talk to Bob and recognize him by

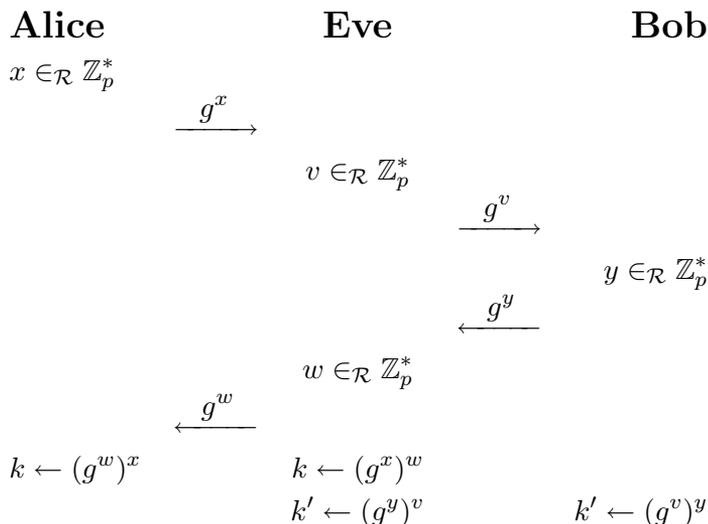


Figure 12.2: Diffie-Hellman protocol with a man in the middle.

his voice. Let  $h$  be a hash function of some sort. If Bob reads the first few digits of  $h(k)$  to Alice, then Alice can verify that Bob is using the same key as she is. Alice can read the next few digits of  $h(k)$  to Bob to allow Bob to do the same verification. This works, but only in situations where you can tie knowledge of the key  $k$  to the actual person on the other side. In most computer communications, this solution is not possible. And if Eve ever succeeds in building a speech synthesizer that can emulate Bob, it all falls apart. Finally, the biggest problem with this solution is that it requires discipline from the users. But users regularly ignore security procedures.

## 12.4 Pitfalls

Implementing the DH protocol can be a bit tricky. For example, if Eve intercepts the communications and replaces both  $g^x$  and  $g^y$  with the number 1, then both Alice and Bob will end up with  $k = 1$ . The result is a key negotiation protocol that looks as if it completed successfully, except that

Eve knows the resulting key. That is bad, and we will have to prevent this attack in some way.

A second problem is if the generator  $g$  is not a primitive element of  $\mathbb{Z}_p^*$  but rather generates only a small subgroup. Maybe  $g$  has an order of one million. In that case the set  $\{1, g, g^2, \dots, g^{q-1}\}$  only contains a million elements. As  $k$  is in this set, Eve can easily search for the correct key. Obviously, one of the requirements is that  $g$  must have a high order. But who chooses  $p$  and  $g$ ? All users are using the same values, so most of them get these values from someone else. To be safe, they have to verify that  $p$  and  $g$  are chosen properly. Alice and Bob should each check that  $p$  is prime, and that  $g$  is a primitive element modulo  $p$ .

The subgroups modulo  $p$  form a separate problem. Eve's attack of replacing  $g^x$  with the number 1 is easy to counter by having Bob check for this. But Eve could also replace  $g^x$  with the number  $h$ , where  $h$  has a small order. The key that Bob derives now comes from the small set generated by  $h$ , and Eve can try all possible values to find  $k$ . (Of course, Eve can play the same attack against Alice.) What both Alice and Bob have to do is verify that the numbers they receive do not generate small subgroups.

Let's have a look at the subgroups. Working modulo a prime, all (multiplicative) subgroups can be generated from a single element. The entire group  $\mathbb{Z}_p^*$  consists of the elements  $1, \dots, p-1$  for a total of  $p-1$  elements. Each subgroup is of the form  $1, h, h^2, h^3, \dots, h^{q-1}$  for some  $h$  and where  $q$  is the order of  $h$ . As we discussed earlier, it turns out that  $q$  must be a divisor of  $p-1$ . In other words: the size of any subgroup is a divisor of  $p-1$ . The converse also holds: for any divisor  $d$  of  $p-1$  there is a single subgroup of size  $d$ . If we don't want any small subgroups, then we must avoid small divisors of  $p-1$ .

This is a problem. If  $p$  is a large prime, then  $p-1$  is always even, and therefore divisible by 2. Thus there is a subgroup with two elements; it consists of the elements 1 and  $p-1$ . But apart from this subgroup that is always present, we could avoid other small subgroups by insisting that  $p-1$  has no other small factors.

## 12.5 Safe Primes

One solution is to use a *safe prime* for  $p$ . A safe prime is a (large enough) prime  $p$  of the form  $2q + 1$  where  $q$  is also prime. The multiplicative group  $\mathbb{Z}_p^*$  now has the following subgroups:

- The trivial subgroup consisting only of the number 1.
- The subgroup of size 2, consisting of 1 and  $p - 1$ .
- The subgroup of size  $q$ .
- The full group of size  $2q$ .

The first two are trivial to avoid. The third is the group we want to use. The full group has one remaining problem. Consider the set of all numbers modulo  $p$  that can be written as a square of some other number (modulo  $p$ , of course). It turns out that exactly half the numbers in  $1, \dots, p - 1$  are squares, and the other half are non-squares. Any generator of the entire group is a non-square. (If it were a square, then raising it to some power could never generate a non-square, so it does not generate the whole group.)

There is a mathematical function called the Legendre symbol that determines whether a number modulo  $p$  is a square or not, without ever needing to find the root. There are efficient algorithms for computing the Legendre symbol. So if  $g$  is a non-square and you send out  $g^x$ , then any observer, such as Eve, can immediately determine whether  $x$  is even or odd. If  $x$  is even, then  $g^x$  is a square. If  $x$  is odd, then  $g^x$  is a non-square. As Eve can determine the square-ness of a number using the Legendre symbol function, she can determine whether  $x$  is odd or even. This is exceptional behavior; Eve cannot learn the value  $x$ , except for the least significant bit. The solution to avoid this problem is to use only the squares modulo  $p$ . This is exactly the subgroup of order  $q$ . Another nice property is that  $q$  is prime, so there are no further subgroups we have to worry about.

Here is how to use a safe prime. Choose  $(p, q)$  such that  $p = 2q + 1$  and both  $p$  and  $q$  are prime. (You can use the `ISPRIME` function to do this on a trial-and-error basis.) Choose a random number  $\alpha$  in the range  $2, \dots, p - 2$  and set  $g = \alpha^2 \pmod{p}$ . Check that  $g \neq 1$  and  $g \neq p - 1$ . (If  $g$  is one

of these forbidden values, choose another  $\alpha$  and try again.) The resulting parameter set  $(p, q, g)$  is suitable for use in the Diffie-Hellman protocol.

Every time Alice (or Bob) receives a value that is supposed to be a power of  $g$ , she (or he) must check that the value received is indeed in the subgroup generated by  $g$ . When you use a safe prime as described above, you can use the Legendre symbol function to check for proper subgroup membership. There is also a simpler but slower method. A number  $r$  is a square if and only if  $r^q = 1 \pmod{p}$ . You also want to forbid the value 1, as its use always leads to problems. So the full test is:  $r \neq 1 \wedge r^q \pmod{p} = 1$ .

## 12.6 Using a Smaller Subgroup

The disadvantage of using the safe prime approach is that it is inefficient. If the prime  $p$  is  $n$  bits long, then  $q$  is  $n-1$  bits long and so all exponents are  $n-1$  bits long. The average exponentiation will take about  $3n/2$  multiplications of numbers modulo  $p$ . For large primes  $p$ , this is quite a lot of work.

The standard solution is to use a smaller subgroup. Here is how that is done. We start by choosing  $q$  as a 256-bit prime. (In other words:  $2^{255} < q < 2^{256}$ .) Next we find a (much) larger prime  $p$  such that  $p = Nq+1$  for some arbitrary value  $N$ . To do this, we choose  $N$  randomly in the suitable range, compute  $p$  as  $Nq+1$ , and check whether  $p$  is prime. As  $p$  must be odd, it is easy to see that  $N$  must be even. The prime  $p$  will be thousands of bits long.

Next we have to find an element of order  $q$ . We do that in a similar fashion to the safe prime case. Choose a random  $\alpha$  in  $\mathbb{Z}_p^*$  and set  $g := \alpha^N$ . Now verify that  $g \neq 1$  and  $g^q = 1$ . (The case  $g = p-1$  is covered by the second test, as  $q$  is odd.) If  $g$  is not satisfactory, choose a different  $\alpha$  and try again. The resulting parameter set  $(p, q, g)$  is suitable for use in the Diffie-Hellman protocol.

When we use this smaller subgroup, the values that Alice and Bob will exchange are all in the subgroup generated by  $g$ . But Eve could interfere and substitute a completely different value. Therefore, every time Alice or Bob receives a value that is supposed to be in the subgroup generated by  $g$ , they should check that it actually is. This check is the same as in the safe prime case. A number  $r$  is in the proper subgroup if  $r \neq 1 \wedge r^q \pmod{p} = 1$ .

Of course, they should also check that  $r$  is not outside the set of modulo- $p$  numbers, so the full check becomes  $1 < r < p \wedge r^q = 1$ .

For all numbers  $r$  in the subgroup generated by  $g$  we have that  $r^q = 1$ . So if you ever need to raise number  $r$  to a power  $e$ , you only have to compute  $r^{e \bmod q}$ , which can be considerably less work if  $e$  is much larger than  $q$ .

How much more efficient is the subgroup case? The large prime  $p$  is at least 2000 bits long. In the safe-prime situation, computing a general  $g^x$  takes about 3000 multiplications. In our subgroup case,  $g^x$  takes about 384 multiplies because  $x$  can be reduced modulo  $q$  and is therefore only 256 bits long. This is a savings of a factor of nearly eight. When  $p$  grows larger, the savings increase further. This is the reason that subgroups are widely used.

## 12.7 The Size of $p$

Choosing the right sizes for the parameters of a DH system is difficult. Up to now, we have been using the requirement that an attacker has to spend  $2^{128}$  steps to attack the system. That was an easy target for all the symmetric key primitives. Public-key operations like the DH system are far more expensive to start with, and the computational cost grows much more quickly with the desired security level.

If we keep to our requirement of forcing the attacker to use  $2^{128}$  steps to attack the system, the prime  $p$  should be about 6800 bits long. In practical systems today that will be a real problem from a performance point of view.

There is a big difference between key sizes for symmetric primitives and key sizes for public-key primitives like DH. Never, ever fall into the trap of comparing a symmetric key size (such as 128 or 256 bits) to the size of a public key that can be thousands of bits. The public-key sizes are always much larger than the symmetric key sizes.<sup>1</sup>

The public-key operations are far slower than encryption and authentication functions we presented earlier. In most systems, the symmetric-key operations are insignificant, whereas the public-key operations can have a real

---

<sup>1</sup>This holds for the public-key schemes we discuss in this book. Other public-key schemes, such as those based on elliptic curves, can have completely different key size parameters.

effect on performance. We must therefore look much more closely at the performance aspects of public-key operations.

Symmetric key sizes are typically fixed in a system. Once you design your system to use a particular block cipher and hash function, you also fix the key size. That means that the symmetric key size is fixed for the life of the system. Public-key sizes, on the other hand, are almost always variable. This makes it much easier to change the key size. We set out to design a system that will be used for 30 years, and the data must be kept secure for 20 years after it was first processed. The symmetric key size must be chosen large enough to protect the data up to 50 years from now. But the variable-sized public keys only have to protect the data for the next 20 years. After all, all keys have a limited lifetime. A public key might be valid for one year, and should protect data for 20 more years. This means that the public key only needs to protect data 21 years, rather than the 50 years needed for symmetric keys. Each year you generate a new public key, and you can choose larger public keys as progress in computing technology requires.

The best estimates of how large your prime  $p$  needs to be can be found in [63]. A prime of 2048 bits can be expected to secure data until around 2022; 3072 bits is secure until 2038; and 4096 bits until 2050. The 6800 bits we mentioned above are derived from the same formulas used in [63]. That is the size of  $p$  if you want to force the attacker to perform  $2^{128}$  steps in an attack.

Be very careful with these types of predictions. There is some reasonable basis for these numbers, but predicting the future is always dangerous. We might be able to make some sensible predictions about key sizes for the next 10 years, but making predictions about what things will be like 50 years from now is really rather silly. Just compare the current state of the art in computers and cryptography with the situation 50 years ago. The predictions in [63] are by far the best estimates we have, but don't put too much faith in them.

So what are we to do? As cryptographic designers, we have to choose a key size which will be secure for at least the next 20 years. Obviously 2048 bits is a lower bound. Larger is better, but larger keys have a significant extra cost. In the face of so much uncertainty, we would like to be conservative. So here is our advice: use 2048 bits as an absolute minimum. (And don't

forget that as time passes this minimum will grow.) If at all possible from a performance point of view, use 4096 bits, or as close to 4096 bits as you can afford. Furthermore, make absolutely sure that your system can handle sizes up to 8192 bits. This will save the day if there are unexpected developments in attacking public-key systems. Improvements in cryptanalysis will most likely lead to attacks on the smaller key sizes. Switching to a very much larger key size can be done while the system is in the field. It will cost some performance, but the basic operation of the system will be preserved. This is far better than losing all security and having to reengineer the system, which is what you would have to do if the system cannot use larger keys.

Some applications require data to be kept secret for much longer than 20 years. In these cases you need to use the larger keys now.

## 12.8 Practical Rules

Here are our practical rules for setting up a subgroup that you can use for the DH protocol.

Choose  $q$  as a 256-bit prime. (There are collision-style attacks on the exponent in DH, so all our exponents should be 256 bits long to force the attacker to use at least  $2^{128}$  operations.) Choose  $p$  as a large prime of the form  $Nq + 1$  for some integer  $N$ . (See section 12.7 for a discussion of how large  $p$  should be. Computing the corresponding range for  $N$  is trivial.) Choose a random  $g$  such that  $g \neq 1$  and  $g^q = 1$ . (The easy way to do this is to choose a random  $\alpha$ , set  $g = \alpha^N$ , and check  $g$  for suitability. Try another  $\alpha$  if  $g$  fails the criteria.)

Any party receiving the subgroup description  $(p, q, g)$  should verify that:

- Both  $p$  and  $q$  are prime,  $q$  is 256 bits long, and  $p$  is sufficiently large. (Don't trust keys that are too small.)
- $q$  is a divisor of  $(p - 1)$ .
- $g \neq 1$  and  $g^q = 1$ .

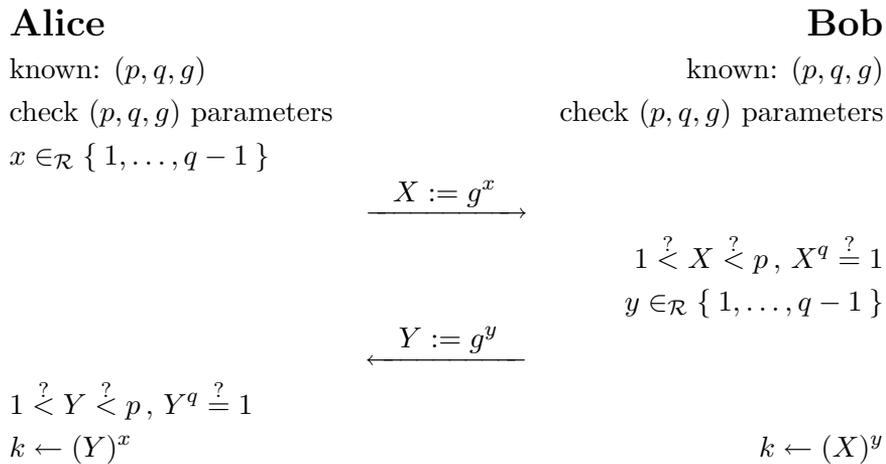


Figure 12.3: Diffie-Hellman in a subgroup.

This should be done even if the description is provided by a trusted source. You would be amazed at how often systems fail in some interesting way, especially when they are under attack. Checking a set  $(p, q, g)$  takes a little time, but in most systems the same subgroup is used for a long time, so these checks need only be performed once.

Any time a party receives a number  $r$  that is supposed to be in the subgroup, it should be verified that  $1 < r < p$  and  $r^q = 1$ . Note that  $r = 1$  is *not* allowed.

Using these rules, we get the version of the Diffie-Hellman protocol shown in figure 12.3. Both parties start by checking the group parameters. Each of them only has to do this once at start-up, not every time they run a DH protocol. (They should do it after every reboot or reinitialization, however, because the parameters could have changed.)

The rest of the protocol is very much the same as the original DH protocol in figure 12.1. Alice and Bob now use the subgroup, so the two exponents  $x$  and  $y$  are in the range  $1, \dots, q-1$ . Both Alice and Bob check that the number they receive is in the proper subgroup to avoid any small-subgroup attacks by Eve.

The notation that we use for the checks is a relational operator (such as  $=$  or  $<$ ) with a question mark above it. This means that Alice (or Bob) should check that the relation holds. If it does, then everything is all right. If the relation is not correct, then Alice has to assume that she is under attack. The standard behavior is to stop the execution of the protocol, not send any other messages, and destroy all protocol-specific data. For example, in this protocol Alice should destroy  $x$  and  $Y$  if the last set of checks fails. See section 14.5.5 for a detailed discussion of how to handle these failures.

This protocol describes a secure variant of DH, but it should not be used in exactly this form. The result  $k$  has to be hashed before it is used by the rest of the system. See section 15.6 for a more detailed discussion.

## 12.9 What Could Go Wrong

Very few books or articles talk about the importance of checking that the numbers you receive are in the correct subgroup. Niels first found this problem in the Internet Key Exchange (IKE) protocol of IPsec [41]. Some of the IKE protocols include a DH exchange. As IKE has to operate in the real world, it has to deal with lost messages. So IKE specifies that if Bob receives no answer, he should resend his last message. IKE does not specify how Alice should process the message that Bob sent again. And it is easy for Alice to make a serious mistake.

For simplicity, let us suppose Alice and Bob use the DH protocol in the subgroup illustrated in figure 12.3 without checking that  $X$  and  $Y$  are proper values. Furthermore, after this exchange Alice starts using the new key  $k$  to send an encrypted and authenticated message to Bob which contains some further protocol data. (This is a very usual situation, and similar situations can occur in IKE.)

Here is the dangerous behavior by Alice: when she receives a resend of the second message containing  $Y$ , she simply recomputes the key  $k$  and sends the appropriate reply to Bob. Sounds entirely harmless, right? But the attacker Eve can now start to play games. Let  $d$  be a small divisor of  $(p - 1)$ . Eve can replace  $Y$  by an element of order  $d$ . Alice's key  $k$  is now limited to  $d$  possible values, and is completely determined by  $Y$  and  $(x \bmod d)$ . Eve

tries all possible values for  $(x \bmod d)$ , computes the key  $k$  that Alice would have gotten, and tries to decrypt the next message that Alice sends. If Eve guesses  $(x \bmod d)$  correctly, this message will decrypt properly, and Eve has learned  $(x \bmod d)$ .

But what if  $p - 1$  contains a number of small factors  $(d_1, d_2, \dots, d_k)$ ? Then Eve can run this attack repeatedly for each of these factors and learn  $(x \bmod d_1), \dots, (x \bmod d_k)$ . Using the general form of the Chinese Remainder Theorem (see section 13.2) she can combine this knowledge to  $(x \bmod d_1 d_2 d_3 \cdots d_k)$ . So if the product of all small divisors of  $p - 1$  is large, Eve can get a significant amount of information about  $x$ . As  $x$  is supposed to be secret, this is always a bad development. In this particular case, Eve can finish by forwarding the original  $Y$  to Alice and letting Alice and Bob complete the protocol. But Eve has collected enough information about  $x$  that she can now find the key  $k$  that Alice and Bob use.

To be quite clear: this is not an attack on IKE. It is an attack on an implementation of IKE that is allowed by the standard. Still, in our opinion the protocol should include enough information for a competent programmer to create a secure implementation. Leaving this type of information out is dangerous, as somebody somewhere will implement it the wrong way.

Eve has to be lucky that  $p - 1$  has enough small divisors. We are designing against an adversary that can perform  $2^{128}$  steps of computing. This allows Eve to take advantage of all divisors of  $p - 1$  up to about  $2^{128}$  or so. We've never seen a good analysis of the probabilities of how much information Eve could get, but a quick estimate indicates that on average Eve will be able to get about 128 bits of information about  $x$  from the factors smaller than  $2^{128}$ . She can then attack the unknown part of  $x$  using a collision-style attack, and as  $x$  is only 256 bits long, this leads to a real attack. At least, it would if we didn't check that  $X$  and  $Y$  were in the proper subgroup.

The attack becomes even easier if Eve was the person selecting the subgroup  $(p, q, g)$ . She may have put the small divisors into  $p - 1$  herself when she selected  $p$  in the first place. Or maybe she sat on the committee that recommended certain parameters for a standard. This isn't as crazy as it seems. The U.S. government, in the form of NIST, helpfully provides primes that can be used with DSA, a signature scheme that uses subgroups like this. Other parts of that same U.S. government (e.g., NSA, CIA, FBI) have a

vested interest in being able to break into private communications. We certainly don't want to imply that these primes are bad, but it is something that you would want to check before you use them. This is easy to do; in fact, NIST published an algorithm for choosing parameters that does not insert additional small factors, and you can check whether the algorithm was indeed followed. But few people ever do.

In the end, the simplest solution is to check that every value you receive is in the proper subgroup. All other ways of stopping small subgroup attacks are much more complicated. You could try to detect the small factors of  $p - 1$  directly, but that is way too complicated. You could require the person who generated the parameter set to provide the factorization of  $p - 1$ , but that adds lots of complexity to the whole system. Verifying that the received values are in the right subgroup is a bit of work, but it is by far the simplest and most robust solution.