

19. Coding for Secrecy

1. Introduction

It has always been necessary for enterprises of all sorts and especially governments, to keep some of the information in their possession secret from the prying eyes of rivals.

They have often attempted to do so by using codes, with the hopes that encrypted messages would be unreadable to others. There is a vast amount of lore on coding schemes that have been used successfully or unsuccessfully in the past and on techniques for attempting to break these schemes.

The modern use of the internet for communication has led to much greater needs for secret codes. If you wish to send a private message to someone, it will go out in some manner that you do not know, through various sorts of communications links at which it may be read by others unknown. You have the problem then of encrypting it so that it cannot be read by undesirables.

Diffie and Hellman, years ago, came up with a scheme for handling such communications, called a public key cryptosystem. It is based on the assumption that there is a wide class of functions that are relatively easy to compute but extraordinarily difficult to invert unless you possess a secret.

According to this scheme, each communicator or recipient, A, publishes in a well defined place a description of his or her function, f_A from this class.

When you, B, want to send a message, m , to A, you send $f_A(f_B^{-1}(m))$ which you can compute because you know how to invert your function f_B , and how to compute the recipient's function f_A .

A can then invert f_A and compute f_B of the result, and discover the message. Anyone else would have to solve the said-to-be-extraordinarily-difficult task of inverting the action of one or another of these functions on some message in order to read the message or alter it in any way at all.

This kind of encryption has two vulnerabilities.

First an adversary could learn the secret of inverting your function f_B by espionage. The adversary can then intercept all your messages and send message posing as you.

Second, an adversary C could possibly convince A that he is B and convince B that he is A. Then A would send a message of the form $f_C((f_A^{-1}(m))$ supposedly to B which C can intercept and read by performing $f_A f_C^{-1}$ on it and send on $f_B(f_C^{-1}(m'))$ to B. Here m' may be m or may be any modification of it. So long as B believes that the

message is coming from A, B will likely respond back to C, and C can read or alter all messages at will.

Roughly the same sort of problem exists even for such direct interaction devices as ATM machines. Recently two individuals were arrested for putting fake devices over several of them so that users gave their passwords and cards to the fakes. Thus by convincing a user that the faker is the bank, the faker can learn the secret that the user employs to communicate with the bank, and can then exploit that information to steal from the user, by convincing the bank that the faker is the user. Such things are of course known to happen with live human interaction as well.

On the internet, those who need secure transactions have introduced certificates to avoid this kind of attack. We will not discuss this further here.

We will discuss one of the classes of problems that have been suggested and deployed for communications of this public key type. It was actually developed here at M.I.T., a number of years ago.

The hard problem that is more or less equivalent to breaking this code is that of factoring a rather large number. (There are variants of the coding scheme in which the inversion problem is exactly equivalent to factoring such a number.)

To describe it, we address the following questions:

1. What is the encryption scheme, and how can it be set up?

The answer to this question is that encryption will involve raising the message, described as a number m , to a certain power z mod another number, N , with N the product of two large primes, p and q . This leads to the next questions:

2. How do we find large primes p and q ?
3. How do we raise a number to a power mod N ?
4. How can we invert the encryption to find the message?

After considering these questions, we will ask: how difficult is it to break this encryption scheme? In particular we examine how hard it seems to be to factor numbers.

2. RSA Encryption

As we have noted just above, **this encryption scheme involves first finding two large primes, then taking their product, then raising numbers to an appropriate power mod that product.**

So the first task involved in implementing such a scheme is to find a large prime number and in fact two of them. Of course to create your encryption key you will need to do this only twice, so you can devote some effort to it.

Suppose, for example, we sought a 25 decimal digit prime number. How can we find one?

There is a standard approach to doing this which goes as follows.

Choose the most significant 22 digits you would like to have after the first, at random.

Then examine all numbers with these most significant bits, and rule out the ones divisible by any number under k , for as high a k as you can, conveniently.

(We do this because for each prime p that we test, we only have to divide it into the first of our large numbers to be able to determine which of all these numbers are divisible by it. So we only have to do one tedious division per small prime.)

Then, test each of those still potentially prime for primality until you find a prime.

Which brings us to two more questions:

How can we test a number for primality?

How many numbers will we have to test, typically, before we find a prime?

We now address these questions, but first introduce two tools that will help us here and beyond.

3. Euclid's Algorithm

Given two numbers positive integers, A and B , with $A > B$, we can produce a third number C , by the definition $C = B \bmod A$ (or in spreadsheet terms, $C = \text{mod}(A, B)$).

Moreover, we can iterate this step, (with $B = A_2$) by defining

$$A_{j+1} = \text{mod}(A_j, A_{j-1})$$

and if we do so we are performing Euclid's algorithm.

There are two important things to say about this algorithm.

First, it eventually ends, when A_j is a factor of A_{j-1} , and when this happens, A_j is the greatest common factor of the original two numbers.

Second, we can work this procedure backwards to find an expression for this greatest common factor (or greatest common divisor or gcd) as a linear combination of the original two numbers.

For example, suppose we start with 237 and 177. We get

$$\begin{aligned} 237 &= 177*1 + 60 \\ 177 &= 60*2 + 57 \\ 60 &= 57*1 + 3 \\ 57 &= 3*19 + 0 \end{aligned}$$

Since A_j , here 3, is a factor of A_{j-1} , all the terms on the right hand side of the previous equation are multiples of A_{j-1} , so the left hand number is such a multiple as well, and this same argument holds true up the line to the original two numbers, and this is the first fact.

The next to last equation above gives an expression for A_j , (here 3) in terms of the previous two A 's.

We can then use the previous equation to eliminate the smaller of these two A 's, and so on up the line until we get an expression for A_j as a linear combination of A_1 and A_2 with integer coefficients.

The arithmetic of it here is:

$$\begin{aligned} 60 - 57*1 &= 3 \\ 177 - 60*2 &= 57 \\ 237 - 177*1 &= 60 \end{aligned}$$

from which we can deduce

$$\begin{aligned} 60 - (177-60*2)*1 &= 60*3 - 177 = 3 \\ (237 - 177)*3 - 177 &= 277*3 - 177*4 = 3 \end{aligned}$$

When the gcd is 1, this algorithm gives us the multiplicative inverse of A_2 mod A_1 , which is the coefficient of A_2 in the last equation (here it is -4), since it tells us that this coefficient multiplied by A_2 is congruent to 1 mod A_1 . In our particular case the gcd is 3 and 177 has no multiplicative inverse mod 237.

The rule for doing this is,

Suppose we get from Euclid: $A_j = A_{j-2} - q_j A_{j-1}$ for each j . Then we have two coefficients, here 1, and $-q_j$ in our expression for A_j , the first being the coefficient of the larger number the second of the smaller number.

Call these coefficients b_1 and c_1 respectively.

We then get $b_{k+1} = -c_k$, and $c_{j+1} = -b_k q_{j-k}$ or something like this.

4. The Chinese Remainder Theorem

Suppose A and B are relatively prime numbers, and we are interested in remainders of integers x on dividing by the product AB.

Any integer x has two alternative characterizations; one is its remainder upon dividing by AB, and the other is its pair of remainders upon dividing x by A and by B separately.

The Chinese remainder theorem is the statement that **these two characterizations are equivalent, and the equivalence is preserved under arithmetic operations.**

Consider an example; suppose we are considering remainders on dividing by $7 \cdot 13$ or 91. Then we can describe 164 say by its remainder upon dividing by 91, which is 73, or by the pair (3,8) which are its remainders on dividing by 7 and 13 respectively.

That the equivalence here noted between 73 and (3,8) is preserved under multiplication can be illustrated by multiplying 73 by 11 which is (4,11) or (4,-2) the result is $(3 \cdot 4, -2 \cdot 8)$ or (12,-16) or (5,10). Also it is 803 which is $75 \pmod{91}$ which is also (5,10). The arithmetic functions work similarly.

We can prove this theorem first by noting that

1. every possible remainder on dividing by the product has a pair of remainders on dividing by the factors;
2. the number of possible remainders on dividing by the product is the same as the number of pairs of remainders on dividing by the factors; and
3. no two distinct remainders on dividing by the product can have the same pair of remainders on dividing by the factors. (their difference would otherwise have 0 remainder on dividing by each factor, and hence on dividing by their product which makes them the same mod the product)

Also, taking remainders on dividing by the product has no effect on remainders on dividing by the factors, so that the preservation of the correspondence by arithmetic operations here is an immediate consequence of the consistency of arithmetic on taking remainders with arithmetic before taking remainders. In other words the correspondence is preserved because both representations have to agree with ordinary arithmetic before taking remainders.

It is not generally easy to find one representation of x given the other, but the fact that they both exist will be very useful to us.

5. Taking Products of Numbers mod N

The method we will use for encryption decryption and testing all involve raising numbers to high powers mod some integer.

At first glance, the idea of raising a huge number to a huge power seems ridiculous. The number of digits will grow outrageously and get rapidly out of hand. However if we are limiting ourselves to computations of remainders on dividing by some N , then there are no worries of this kind.

The first issue we must face is how we multiply two numbers mod N . This is not a problem if we have a machine which is willing to use whatever precision arithmetic we want. Otherwise we have to use some procedure to do it.

To do so we can write the factors to be multiplied as binary sequences (or in any other base), construct the product sequence as we constructed the product of polynomials (except we do not do calculations mod 2), and then take the remainder of the result in the same way we found the remainder of our polynomials.

The major difference is we will not have $2=0$ and will have to use carrying to get our final answer.

To remind you of what our method was, we created a remainder table which gave us the remainders of the individual powers, and took the dot product of the table with the received word. We then applied $2=0$.

Here we can express all our numbers in as sums of powers of 2 multiplied by coefficients, construct a remainder table for the weight one binary numbers exactly as before (without the mod 2 but using carrying instead) and form the dot product of the number whose remainder we seek with the remainder table.

Both in forming the product and in taking dot products coefficients larger than 1 may be formed, and these must be reduced by the end, through carrying.

Suppose for example, that N is a k digit number to whatever base we are using. Then the product of two remainders mod N will in general be a number requiring at most $2k-1$ coefficients of powers of the base to describe. These coefficients will be convolutions of the coefficients of the two factors as they are in all multiplications.

The remainders of all powers of the base that are smaller than N will be themselves. The remainders of higher powers can be computed by the same general approach that we used to compute the remainder table, except that we cannot use $2=0$ and should carry coefficients that are larger than the base into the next power, as we normally do in multiplication.

We will not go into details here.

6. Raising a Number to a Power.

Once we can multiply, we can raise a number x to a power $M \bmod N$ by performing a sequence of between $\log_2 M$ and twice that number of multiplications mod N .

We can do this by writing M out as a binary sequence. We then, starting with 1, raise x successively to powers given by the initial segments of this binary sequence.

For example, if M were 26, we write it as 11010, and will raise x to the powers 1, 11 or 3, 110 or 6, 1101 or 13 and finally 11010 or 26.

To do this we need only answer two questions:

Given x^A how do we form x^B when the binary representation of B is the same as that of A with one extra 0 on its right? And how do we do the same thing when the binary representation of B is the same as that of A with one extra 1 on its right?

The answer to the first question is: square A ; the answer to the second is square A and then multiply the result by x .

This tells us then how we can raise x to the 26^{th} power.

Start with 1; square it and multiply by x ; then square again and multiply by x (we now have x^3) now square again; then square again and multiply by x (now we have x^{13}), square again and we are done.

The total number of multiplications is the number of binary bits needed to represent M , which is the number of squarings we must perform; and also the number of 1's in the binary expression for M which is the number of times we must multiply by x (less one of each if you don't count the first ones).

One thing to notice about this procedure, is that if M is even, the last operation here is taking a square mod N , and so is the preceding operation if M is divisible by 4, and so on.

7. Testing a Number for Primality

We begin by discussing some generalities and noticing how primes and non primes differ.

Given any number N , we can consider the numbers mod N , by which we mean the remainders obtained on dividing integers by N .

The numbers less than N that are relatively prime to N form a group under the operation of multiplication, as remainders. That is, the product of any two such numbers is still relatively prime to N and subtracting any multiple of N from the product will leave the resulting remainder relatively prime to N . The set of such non-zero numbers is therefore closed under multiplication and therefore form a group.

When N is a prime, then the order of this group is $N-1$. Otherwise, if N has no square factors, it is the product over N 's prime factors p_j of (p_j-1) . (For example, for $N=22$ there are 10 numbers less than N relatively prime to it, for $N=15$ there are 8, for example.)

Now recall that the order of an element x of a group, which is the power you must raise x to get the identity, is the order of the subgroup that x generates and so, by Lagrange's theorem, it must be a divisor of the order of the group.

This means here that if you take any non-zero $x \pmod N$, and raise it to the power $(N-1)$, the remainder of the result mod N will be 1, if N is a prime.

For most non prime N this statement will not be true. For example, if N is the product of two primes, p and q , then there will be x 's whose order mod N are $p-1$ and $q-1$, and the statement that $p-1$ is a divisor of $pq-1$ is the statement that $p-1$ divides $q-1$, and vice versa, which cannot both be true.

However, there are non-prime numbers, called Carmichael numbers for which any x raised to the power $(N-1)$ is $1 \pmod N$, so that testing x 's to see if they obey this condition is not a completely reliable test for primality.

However, primes differ from non-primes in another respect: the remainders on dividing by a prime p form a field, and in it every quadratic equation, such as $x^2-1=0 \pmod N$ has at most two solutions, as was true when we considered remainders on dividing by polynomials.

However, if N has two or more distinct prime factors, then this particular equation will have at least 4 solutions.

How come? Let A and B be relatively prime factors whose product is N . Then the four numbers whose representations as pairs of remainders on dividing by A and by B are $(-1,-1), (1,1), (1,-1)$ and $(-1,1)$ will all obey the equation $x^2=1$, because they do so mod A and mod B , by the Chinese Remainder Theorem.. And there must be 8 such solutions if N has three distinct prime factors, and so on. $(-1,-1)$ is -1 , and $(1,1)$ is 1 here.

These facts imply that we can find a test for the primality of that is much better than merely raising a random number x to the power $N-1 \pmod N$ and seeing if the result is 1, without doing any more work.

What we can do is to raise x to the power $N-1 \pmod N$ using the procedure described in the last section, and examine the last intermediate result that is not 1.

Of course if the last answer of all is not 1, then we know that N cannot be a prime, since the order of x does not divide $N-1$ when this happens.

But even if the final answer is 1, N can fail the primality test if the last predecessor of a 1 other than a 1 was not -1 and the 1 was obtained by squaring. That would imply that we have encountered a number other than 1 or -1 whose square is 1, which is impossible $\pmod N$ when N is a prime.

We can show, as a matter of fact that at least half of the time, even for Carmichael numbers, a random x will cause N to fail this test, as long as N not a prime.

In practice, of course, when N is not a prime, it will usually fail this test by giving a final answer that is not 1.

How then can we test a number for primality?

Until recently, the best known way was as follows: pick an x , say 2; form

$$x^{N-1} \pmod N.$$

If the result is not 1 or a square root of 1 other than 1 or -1 is found along the way in raising x to this power, we deduce that N is not prime. Otherwise we pick another x and try again. If N passes the test each time, say for x given by each of the first 20 primes, we declare N a prime.

The idea here was that the probability that N is not a prime given these results is less than one in a million, even in the unlikely event that N is a Carmichael number.

Why is this probability so small? Well, there are only two ways N could pass the test if it is not prime. All the square roots encountered back to the last initial segment of the binary representation of M that ended in 1 are 1 or -1 . Or some squaring produces a -1 whose square is then 1 in the formation of $x^{N-1} \pmod N$.

In the first case multiplying x by $(1,-1)$ will make the odd power of x square root of 1 in out sequence into $(1,-1)$ or $(-1,1)$, so that we can pair any x for which the test is passed with one for which it fails.

In the second case, we know that x to some even power Q is $(-1,-1)$. This implies that there is a y such that $y^Q = -1 \pmod A$. From this we can find a z such that $z^Q = (1,-1)$ and then we can pair y with yz to pair a number x which fails the test to every one that passes the test, in a one to one manner.

This implies that if x is chosen at random, at least half the time the test will fail, even for Carmichael numbers, and so if the test is repeated independently 20 times, the chance of passage every time is less than 2^{-20} .

Only several months ago someone has announced an algorithm for primality testing that provably always works without making random choices repeatedly as done here, but this way works fine in practice.

7. Implementing RSA

With the ammunition we have developed we can straightforwardly implement the RSA encryption algorithm.

First we find two large primes by the procedure just described. Call them p and q . Then we take their product, N , and find the number $(p-1)(q-1)$ which is the order of the group of remainders on dividing by N that are relatively prime to N .

Next we find a large number $z \bmod N$ that is relatively prime to $(p-1)(q-1)$. (how? Guess one and test it using Euclid's algorithm)

Then we use Euclid's algorithm to find the multiplicative inverse of $z \bmod (p-1)(q-1)$. Call it y , so that we have $yz=1 \bmod (p-1)(q-1)$.

Now the public key will be given by the numbers N and z , and encoding will be described by $c(m) = m^z \bmod N$.

You will know p and q and hence y , and can decode by using

$$m = c(m)^y \bmod N.$$

Exercises:

1. Set up a spreadsheet that finds the gcd of two inputted numbers, and expresses that gcd as a linear combination of the two numbers.
2. Set up a spreadsheet that raises two numbers to a power mod N for the highest value of N possible on your machine (but no more than 20 digits) using ordinary multiplication of numbers. (you have to multiply numbers which requires twice the precision of the length of N)
3. Set up a spreadsheet that checks 200 numbers in a row to eliminate those with any small prime factors (where small means under 200)
4. Set up a spreadsheet that examines candidates for primality from your 3 spreadsheet to test them for primality (expand $N-1$ in binary and raise some x to the power $N-1 \bmod N$ by the procedure described in the notes.)