

## 20. More on Secret Codes

In this Chapter we will describe some other schemes for secret coding, consider some questions about how long it should take to find primes, and go into further detail about the steps needed to construct an RSA code.

### 1. Other Public Key Algorithms

The RSA algorithm is based upon the difficulty of factoring large numbers. Its public key includes a number  $N$  of the form  $pq$  and a power  $m$  relatively prime to  $(p-1)(q-1)$ . Anyone can then encode in your code by raising their message to the power  $m \bmod N$ .

If you know  $p$ , you can compute  $q$ , and  $(p-1)(q-1)$  and can use Euclid's Algorithm starting from  $(p-1)(q-1)$  and  $m$  to find the multiplicative inverse  $z$ , of  $m \bmod (p-1)(q-1)$ .

This information tells you enough to be able to recover the message; since you will be able to do so by raising the encoded message to the power  $z \bmod N$ .

Thus, if you can factor  $N$ , you can break this code.

On the other hand, factoring  $N$  and breaking the code are not exactly equivalent. You could conceivably break the code without ever factoring  $N$ .

There is a variant of this algorithm, due to Rabin, that has the property that breaking it is exactly equivalent to factoring  $N$ ; we will not describe it here, but note that finding out what it is, describing it and constructing a coder decoder for it is a plausible term paper for this course.

There is another class of public key codes that is based on the difficulty of solving the problem mod a prime  $p$  of finding the power  $k$  which will take  $q$  into  $q^k \bmod p$ .

In the ElGamal discrete logarithm coding scheme, you publish a large prime  $p$  and a large  $q$  and  $q^k$  but keep  $k$  secret. We assume that it is very difficult to recover  $k$  from these two numbers, mod  $p$ .

Given a message  $m$ , a sender picks a random number  $s$  less than  $p-1$ , and sends

two numbers:  $q^s \bmod p$ , and  $m \cdot q^{ks} \bmod p$ , where  $m$  is the message. The sender can do this because it knows  $q$  and  $q^k$  and can raise each of these to the power  $s$ , and can multiply the latter by  $m$ , all computations mod  $p$ .

The receiver can then raise the first of these,  $q^s \bmod p$  to the power  $k$ , and divide the second number,  $q^{sk} \cdot m \bmod p$ , by the result, and that will give  $m$ .

Again, nobody knows how difficult it is to solve the problem of retrieving  $k$  from  $q^k \bmod p$  given  $q$ , and it is not clear that you have to be able to retrieve  $k$  to break this code.

This algorithm is based on the fact that it is easy to raise a number  $q$  to a high power mod  $p$  but not easy to find the power  $k$  given  $q$  and  $q^k$ .  $k$  here is the logarithm of  $q^k$  to base  $q$  mod  $p$ , and the hard problem here is that of finding the logarithm of a number mod  $p$  given the base, the number and  $p$ .  $q$  here should be relatively prime to  $p-1$ .

The hard problem here is called **the discrete logarithm problem**.

## 2. How rare are primes?

All the algorithms discussed here involve, in their construction, finding large prime numbers. A relevant question to finding them is how common are they?

Thus, if we look at numbers near  $N$ , we can ask, what proportion of them will be primes?

The answer to this question is given by the famous prime number theorem, which tells us that this proportion, called the density of primes of size near  $N$ , is roughly  $1/\ln(N)$ .

We can also ask, what proportion of the non-prime numbers near  $N$  will have no prime factor under  $M$ ? or what proportion of the numbers near  $N$  will have all their prime factors under  $M$ ? These questions will both be of interest to us.

The first of these two questions is of interest because we look for primes by first weeding out those having a factor under  $M$  for  $M$  as large as we can. The answer to the first question, along with the density of primes tells us then how many survivors of this weeding we can expect to test before finding a prime.

The answer to the second question will be of interest to our attempts to factor numbers.

How can we address these questions:

Notice that half the numbers near  $N$  are divisible by 2; among the rest,  $1/3$  are divisible by 3, of the rest,  $1/5$  are divisible by 5, and so on, up to  $N^{1/2}$ ; if a number has magnitude near  $N$  and no factor of cardinality less than  $N^{1/2}$ , it will have to be prime, (otherwise it will have two factors and one at least has to be at most its square root).

The proportion  $P_N$  of primes then should be on the order of

$$(1-1/2)(1-1/3)(1-1/5) \dots (1-1/p_j) \dots (1-1/p_z)$$

where  $p_z$  is roughly  $N^{1/2}$ , and the product is over all primes up to  $p_z$ .

If we take the logarithm of  $P_N$ , we get a sum over all primes  $p_j$  up to  $p_z$  of

$$\ln (1-1/p_j).$$

For large  $p_j$  such terms each are roughly  $-1/p_j$  (this is the first term in the power series expansion of this logarithm).

Thus, apart from the first terms a constant error from corrections coming from the the other terms in series expansions, we get the sum of  $-1/p_j$  for the log of proportion of primes here.

For large  $j$  we can use the fact that the sum of  $-1/p_j$  is like the sum of  $dx/x$  multiplied by the proportion of primes among numbers near  $x$ .

By the prime number theorem this proportion is  $1/\ln(x)$ , and our sum then differs by a constant from the integral of  $dx/(x \ln x)$  up to  $x=N^{1/2}$ .

This integral differs by a constant from  $-\ln((\ln N)/2)$ , or  $\ln(1/\ln N) + \ln 2$ .

We conclude that the proportion of primes differs by a constant from  $c/(\ln N)$  for some  $c$ , which is what the prime number theorem predicts with  $c=1$ , so we have verified its consistency with our claims.

But now we can ask, what proportion of numbers of size  $N$  have no prime factors under  $M$  but do have prime factors above  $M$ . To compute this for these we must end the integration at  $M$  instead of  $N^{1/2}$ . We get then, for large  $M$ ,  $c/2 \ln M$ ,

This tells us that the proportion of primes among numbers without prime factors under  $M$  will be one per  $(2 \ln M)/\ln N$ .

Thus, if we were to eliminate primes up to a million, with  $\log_{10} M = 6$ , the ratio of 100 digit numbers not eliminated would be like  $100/12$  or roughly one in 8.

This means we can expect to look at only a bearable number of these to find a prime.

If on the other hand we look for primes on the order of a billion and eliminate those with factors under 1000 we find that  $2/3$  will be primes.

We can use the same approach to answer the question:

What proportion of numbers of size roughly  $N$  will have all their prime factors at most  $M$ ?

Then to find the logarithm of our answer we would integrate instead from  $M$  to  $N$  since we only want to

eliminate factors in this range, and we get something like  $\ln(\ln M / \ln N)$  for the logarithm of the proportion of these, which means that a proportion of the numbers like  $\ln M / \ln N$  have no prime factors greater than  $M$ . This includes both primes, and numbers all of whose prime factors are less than  $M$

The proportion of these with all prime factors less than  $M$  will be roughly  $(\ln M - 1) / \ln N$ , by the prime number theorem, since by it a proportion  $1 / \ln N$  will be primes.

This statement will be useful to us eventually.

In the remainder of this Chapter we will review what you have to do to find primes and set up an encoder and decoder for an RSA code.

### 3 Multiplying Numbers.

A number can be considered a polynomial evaluated somewhere, with coefficients in a specific range.

For example  $3x^2 + 7x + 1$  at  $x=10$  is the number 371. Typically we describe numbers by requiring that the coefficients, here 3 7 and 1, are all between 0 and  $x-1$ , here 9.

If we find instead the polynomial  $3x^2 + 17x + 21$ . We re-write the 20 as  $2x$ . This carries a 2 into the 17 making it 19. Then we re-write the  $10x$  as  $1x^2$  and this carries the 1 into the 3 making it 4. We would then change our polynomial to  $4x^2 + 9x + 1$ .

Carrying consists of the following operations: starting from the least significant coefficient, you divide it by  $x$ ; the remainder stays, the dividend gets added to the next higher coefficient and the process is repeated for it.

When we multiply numbers together very typically we produce coefficients which are outside the normal range in the answer, and must perform carrying. (Those of you who completed your elementary education are doubtless aware of this fact.)

Now let us consider the question: suppose we have two  $k$  digit numbers,  $A$  and  $B$ , and want their product mod some other  $k$  digit number  $C$  (that is larger than each).

However the machine at our disposal does not keep enough precision to simply multiply the two numbers together, which will produce a  $2k$  or so digit number.

We therefore want the simplest procedure for computing the  $AB \bmod C$  here.

Here is one way to proceed. Suppose for convenience that  $k$  is even and is  $2j$ .

We write  $A$  as  $A_1x + A_0$  where  $x=10^j$ , and each of the  $A_j$  are  $j$  digit numbers, and do the same for  $B$ .

Then we get  $AB$  is  $A_1B_1x^2 + (A_0B_1 + A_1B_0)x + A_0B_0$ , and all of these products are only  $k$  digit numbers, so we can compute them accurately.

We then need to know the remainders of the powers of 10 up to the  $(2k)$ -th, which can arise when we substitute  $10^j$  for  $x$ .

Suppose for example that  $k=4$  so that  $x$  is  $10^2$  and our resulting polynomial is  $1245x^2 + 5198x + 3124$  (these coefficients were chosen at random) This has the same meaning as  $1 \cdot 10^7 + 2 \cdot 10^6 + 4 \cdot 10^5 + 5 \cdot 10^4 + 5 \cdot 10^5 + 1 \cdot 10^4 + 9 \cdot 10^3 + 8 \cdot 10^2 + 3124$ .

If we consolidate these (several powers appear more than once) and multiply the resulting coefficients by the remainders of the corresponding power of  $10 \bmod C$ , and add up the results, we get the product, with coefficients that will probably require carrying.

An appropriate multiple of  $C$  can be subtracted from the result before carrying, and then the carrying can be performed to give the desired remainder.

We have here broken our numbers up into two coefficients of powers of  $10^{k/2}$ . We could handle much larger numbers similarly by breaking it into more pieces and again treating the coefficients separately.

Notice that getting the remainder here is essentially taking the dot product of the coefficients of the powers of 10 with the table of remainders of those powers, just the same operation as in the remainder table.

The only remaining question is how we find the remainder table. The answer is exactly as in the polynomial problems we encountered for error correcting codes. Namely, multiplying by 10 shifts all remainders up by 10 until you reach the greatest power of 10 less than  $C$ , say  $q$ . The remainder of  $10^q$  is itself, but multiplying it by 10 produces  $10^{q+1}$  and that is whichever of  $10^{q+1} - aC$  is positive and less than  $C$ .

This is exactly what we encountered with our primitive polynomial remainder tables, except here we do not do things mod 2 and if we choose to limit our coefficients, we must do carrying.

#### **4. Raising a Number to a High Power**

If we can multiply conveniently raising  $x$  to the power  $y$  mod  $z$  is a fun thing to do.

In one column we can give the binary expansion of  $y$ , least significant bit first, as we have done before. You can leave space to do this for a huge number by extending the procedure for hundreds or thousands of rows. (the non-zero bits will end of course at  $\log_2 y$ .)

You can then start another column at the bottom with 1, and in the next column give its square mod  $z$ . above the first of these two columns you can enter the statement: if the binary bit in that row is 0 give the entry below in the second column, otherwise multiply it by  $x$  mod  $z$ . By copying these last instructions upward to the top you get  $x^y \text{ mod } z$ .

#### **5. Sieving and doing Euclid's Algorithm**

Sieving to find candidates for primes near  $N$  that have no small prime factors, is lots of fun

You can enter in columns in the first row all the odd numbers up to  $M$  (better only the primes but that takes slightly more work)

Then you can enter in another column say all the odd numbers from  $N-c$  to  $N+c$ , in order, and compute in the row of  $N-c$ :  $\text{=mod}(N-c, (\text{column odd number or prime}))$ .

The you can enter  $\text{=mod}(\text{above entry} + 2, \text{column odd number or prime})$  and copy that into all the

succeeding rows.

You then have a candidate in every row for which min of its entries is non-zero.

Euclid's algorithm forward can be accomplished by one instruction copied appropriately. You must keep track of the coefficients carefully when you work your way backwards to find the coefficient of the gcd as a sum of the original numbers each multiplied by an integer.

If you end with a negative coefficient for the smaller number you can convert it to a positive one by adding the larger number to it.

**Exercise: Set up an actual RSA encoding scheme, using primes that are at least 5000; find a public key including the product of your primes and an appropriate power. Produce a spreadsheet that encodes by raising input messages to that power mod  $pq$ . Also produce a decoder using the inverse of that power mod  $(p-1)(q-1)$  as the decoding power mod  $pq$ .**

**See that you get your message back when you do both!**