Cryptography

Lecturer: Michel Goemans

1 Public Key Cryptosystems

In these notes, we will be concerned with constructing *secret codes*. A sender would like to encrypt his message to protect its content while the message is in transit. The receiver would like to easily decode the received message. Our coding scheme should contain the following properties:

1. encoding is easy to perform;
2. decoding is extremely difficult (for protection against eavesdroppers);
3. decoding is easy if you are in possession of some secret "key".

The traditional way of creating secret codes (used in various degrees of sophistication for centuries) is that both the sender and the receiver share a secret, called a key. The sender scrambles the message in a complicated way that depends on the key. The receiver then uses the key to unscramble the message. If these codes are constructed properly (something which is surprisingly hard to do), it seems virtually impossible for somebody without the key to decode the message, even if they have many examples of pairs of messages and their encodings to work from in trying to deduce the key.

The drawback of this method is ensuring that every pair of people who need to communicate secretly have a shared secret key. Distributing and managing these keys is surprisingly difficult even when these keys are used for espionage. This would be extremely difficult for secret communication over the internet, when you may wish to send your credit card securely to a store you have never before heard of. Luckily, there is another way to proceed.

Diffie and Hellman in 1976 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, say Bob or B, publishes in a well defined place (a kind of telephone directory) a description of his function, f_B from this class; this is Bob's *public key*. Bob knows also the inverse of f_B, this is his *private key*. The assumption is that this inverse is extremely difficult to compute if one does not know some private information.

Suppose now that someone, say Alice or A, would like to send a message m to B. She looks up Bob's public key and sends $m' = f_B(m)$. Since Bob knows his own private key, he can recover $m = f_B^{-1}(m')$. The problem here is that Bob has no guarantee that Alice sent the message. Maybe someone else claiming to be Alice sent it. So, instead, suppose that Alice sends the message $m' = f_B(f_A^{-1}(m))$ to Bob. Notice that Alice needs to know Bob's public key (which she can find in the diretory) and also her known private key, which is known only to her. Having received this message, Bob can look up Alice's public key and recover m by computing $m = f_A(f_B^{-1}(m'))$; again, for this purpose, knowledge of Bob's private key and Alice's public key is sufficient. 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 is the basic setup for a public-key cryptosystem. One can also use it for digital signatures. If Alice wants to show to anyone that she wrote message m, she can publish or send $f_A^{-1}(m)$, and anyone can test it came from Alice by computing $f_A(m')$.

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 and is known as the RSA public key cryptosystem. It was invented by three MIT people, Ron Rivest, Adi Shamir and Len Adleman in 1977. Their scheme is based on the fact that it is easy to multiply two large numbers together, but it appears to be hard to factor a large number. The record so far for factoring has been to factor a 768-bit number (i.e., 232 digits) given in an RSA challenge, and this took the equivalent of 20,000 years of computing on a single-core machine... The task of factoring a 1024-bit number appears to be 1,000 harder with the current algorithms.

2 The RSA code

For this code, choose two very large prime numbers (say with several hundred digits), p and q, and form the product $N = pq$. Choose a number $z < N$ such that z is relatively prime to $(p-1)(q-1) = N - p - q + 1$. Knowing p and q (or $(p-1)(q-1)$) we can find the multiplicative inverse y to z modulo $(p-1)(q-1)$ by the extended Euclidean algorithm. The pair (N, z) constitutes the public key, and (N, y) constitutes the private key.

If we want to encode a message, we first view it as a number in base N. Every digit is a number between 0 and $N-1$ and we will encode each digit $0 \leq m < N$ separately. The sender computes $s = m^z \bmod N$ and transmits s. Upon receiving s, the receiver, who knows the private key, computes $s^y \bmod N$. The claim is that this is precisely m, i.e. $m = s^y \bmod N$.

If one could factor N into $N = pq$ then one can easily compute y from z (by the extended Euclid algorithm) and therefore break this public-key cryptosystem. However, as we said previously, factoring large numbers appears to be very challenging.

Why does this scheme work? Let $x = s^y$; we want to show that $m = x \bmod N$. We have that

$$x = s^y m^{yz} \pmod{N},$$

and since z and y are multiplicative inverses modulo $(p-1)(q-1)$, we get that

$$x = m^{1+k(p-1)(q-1)} = m m^{k(p-1)(q-1)} \pmod{N}.$$

We want to prove that this is equal to $m \bmod N$. We will first show that $x \equiv m \pmod{p}$ and $x \equiv m \pmod{q}$ and then deduce from the Chinese remainder theorem that $x \equiv m \pmod{pq}$. To show that $x \equiv m \pmod{p}$, we need to consider two cases. First, if m is a multiple of p, then x is also a multiple of p, and so $x \equiv m \equiv 0 \pmod{p}$. Otherwise, if m is not a multiple of p then it must be relatively prime to p (since p is prime): $\gcd(m, p) = 1$. Thus we can apply Fermat's Little Theorem, which tells us that $m^{p-1} \equiv 1 \pmod{p}$, and thus

$$m^{k(p-1)(q-1)} \equiv 1^{k(q-1)} \equiv 1 \pmod{p}.$$

Multiplying by m, we indeed obtain $x \equiv m \pmod{p}$.

We can apply the same argument to q, to obtain that $x \equiv m \pmod{q}$ also. Thus the Chinese remainder theorem tells us that $x \equiv m \pmod{N}$, and we have proved the correctness of the RSA scheme.

In order to use RSA, we need to show how to generate large primes, and also how to efficiently compute $m^z \bmod N$ (or $s^y \bmod N$) when z is very large.

3 Raising a Number to a High Power

In this section, we show how to raise a number to a high power modulo N efficiently, say $m^z \bmod N$. The technique is known as repeated squaring and the idea is very simple.

If z is a power of 2, say 2^k, we can compute $m^z \bmod N$ by starting with m, and repeatedly (k times) squaring it and taking it modulo N. This requires only k multiplications. For example, to compute $3^{32} \bmod 83$, we first compute $3^2 = 9 \pmod{83}$ then $3^4 = 9^2 = 81 \pmod{83}$, then $3^8 = 81^2 = (-2)^2 = 4 \pmod{83}$ then $3^{16} = 4^2 = 16 \pmod{83}$, and finally $3^{32} = 16^2 = 256 = 7 \pmod{83}$. Even though 3^{32} is more than 10^{15}, the fact that we did every operation modulo 83 meant that we never had to deal with large numbers.

In general, when z is not necessarily a power of 2, we write z's binary representation. Suppose that z requires d bits, and let z_k denote the k leading bits of z. Observe that $z_k = 2z_{k-1}$ or $z_k = 2z_{k-1}+1$ depending on the kth bit of z. We compute $m^{z_k} \bmod N$ for $k = 1, \cdots, d$. For $k = 1$, this is simply $z^2 \bmod N$. Once we have computed $a_{k-1} = m^{z_{k-1}} \bmod N$, it is easy to compute a_k. Square a_{k-1}, multiply it by m if the kth leading bit is a 1, and do these operations modulo N. We then get a_k and we can repeat.

Example. Suppose $z = 201$. Then z can be respresented by 11001001 in binary. There are 8 steps, in which (i) we square and multiply by m in steps 1, 2, 5, 8 and (ii) we only square in steps 3, 4, 6, and 7.

4 Primality Testing

To be able to use RSA, we need to be able to generate large primes. The *prime number theorem* states that among integers near n, when n is large, approximately one in $\ln n$ is a prime. To generate a large prime, we can then generate a random number with the appropriate number of digits or bits, and check if it is prime. If it is, we are done, else we increment it, and try again until we find a prime number.

We therefore need to be able to efficiently check if a number is prime. This is known as *primality testing*. We could try whether it is divisible by any of the small primes, say all primes up to 30. This would detect a good fraction of the composite numbers, but clearly not all of them. Checking all possible factors up to the square root of the number n is *extremely* slow if n is large, and we are interested in numbers with hundreds of digits.

One approach is based on Fermat's little theorem, and so is called the *Fermat primality test*.

4.1 Fermat primality test

Suppose we are given a large number n, and we want to determine if it's prime.

Let a be any positive number less than n; then by Fermat's Little Theorem, if n is indeed prime, then since $\gcd(a, n) = 1$ we have that

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

If n is not prime, on the other hand, this doesn't have to be true (though it might happen, depending on the specific a and n we choose). So here is a test: choose a large number of randomly chosen values $a_1, a_2, \ldots, a_N$, all positive and less than n, and calculate $a_i^{n-1} \pmod{n}$ for each. This we can do very quickly by repeated squaring, as we saw already. If we obtain a value other than 1 for *any* a, then n is not prime; that a acts a a certificate ("proof") that n is not prime. We call such an a a *Fermat witness* for n; if $a^{n-1} \equiv 1 \pmod{n}$ (and n is composite) then we instead call a a "Fermat liar" for n.

As an example, let's apply this test to $n = 1591$, using $a = 2$. By repeated squaring, we get

- $2^1 \equiv 2 \pmod{1591}$,
- $2^3 \equiv 2^2 \cdot 2 \equiv 8 \pmod{1591}$
- $2^6 \equiv 8^2 \equiv 64 \pmod{1591}$
- $2^{12} \equiv 64^2 \equiv 914 \pmod{1591}$
- $2^{24} \equiv 914^2 \equiv 121 \pmod{1591}$
- $2^{49} \equiv 121^2 \cdot 2 \equiv 644 \pmod{1591}$
- $2^{99} \equiv 644^2 \cdot 2 \equiv 561 \pmod{1591}$
- $2^{198} \equiv 561^2 \equiv 1294 \pmod{1591}$
- $2^{397} \equiv 1294^2 \cdot 2 \equiv 1408 \pmod{1591}$
- $2^{795} \equiv 1408^2 \cdot 2 \equiv 156 \pmod{1591}$
- $2^{1590} \equiv 156^2 \equiv 471 \pmod{1591}$,

and this proves that 1591 is not a prime. Indeed, $1591 = 37 \cdot 43$.

However, we might obtain that $2^{n-1} \equiv 1 \bmod n$ even though n is not prime; this happens for 22 values of n below 10 000. But we can apply the test for multiple values of a, chosen randomly. Let's define precisely our primality test as follows:

Fermat primality test for an integer n

1. Pick $a \in \{1, 2, 3, \ldots, n-1\}$ uniformly at random.
2. Calculate (efficiently via repeated squaring) the value $a^{n-1} \bmod n$. If this is not 1, output "not prime"; otherwise output "maybe prime".

We can repeat this test many times; if it outputs "not prime" at least once, we can be sure that it is indeed not prime; if it returns "maybe prime" each time, then perhaps we can conclude that n is very likely to be prime?

This turns out to be almost, but not quite, true. There are certain special composite numbers, called *Carmichael numbers*, that do a very good job of fooling this test.

Definition 1. A positive integer n is a *Carmichael number* if it is composite and $a^{n-1} \equiv 1 \pmod{n}$ for all a relatively prime to n.

If we apply Fermat's test to a Carmichael number, then the only way it can spot that it's not prime is if the a chosen happens to share a common factor with n. If the factors of n are all large, then there are few such numbers (compared to n) and we're very unlikely to pick one of them.

It's not obvious that these numbers exist, but they do, and there are infinitely many of them; the smallest is 561. They are very rare however (much much rarer than primes), and if you pick a large random n, you'd have to be very unlucky to pick a Carmichael number. What we will prove now is that the Fermat test *does* work on all numbers except for Carmichael numbers.

Theorem 1. *If $n > 2$ is composite and not a Carmichael number, then the probability that the Fermat test works is at least $1/2$.*

Proof. Let

$$L = \{b \in \mathbb{Z}_n^* : b^{n-1} \equiv 1 \bmod n\},$$

i.e., the set of Fermat liars for n. Let $W = \mathbb{Z}_n^* \setminus L$, the set of Fermat witnesses that are relatively prime to n. Since n is not a Carmichael number, W is nonempty. (Note that all elements of $\mathbb{Z}_n \setminus \mathbb{Z}_n^*$ are also Fermat witnesses.)

We will show that $|L| \leq |W|$, from which the theorem follows. Since then the probability we pick $a \in L$ is not more than $1/2$; and so the probability of choosing a Fermat witness is at least $1/2$.

The plan is to find a $1-1$ map ϕ from L to W, which immediately gives us that $|L| \leq |W|$. The map is very simple: pick an arbitrary $a \in W$ (recall it's nonempty), and define

$$\phi(b) = ab \bmod n \qquad \forall b \in L.$$

First, let's see that it is a map from L to W, and not merely a map from L to $\mathbb{Z}_n$. For any $b \in L$,

$$\phi(b)^{n-1} = a^{n-1}b^{n-1} \equiv a^{n-1} \not\equiv 1 \pmod{n}.$$

So indeed $\phi(b) \in W$. It's also clearly $1-1$: if $\phi(b) = \phi(b')$, then multiplying by a^{-1} gives $b = b'$.

This concludes the proof. □

We can conclude that the probability that if we run the Fermat test K times on a number n that is composite and not a Carmichael number, the probability that the Fermat test outputs "maybe prime" on all tries is at most 2^{-K}; by choosing K large enough, we can make the error probability tiny.

The defect of the Fermat primality test, that it does not work on Carmichael numbers, can be rectified by more advanced tests, in particular the *Miller-Rabin test*. This is based on a different certificate of compositeness: if you can find a number $a \in \mathbb{Z}_n$ so that $a^2 \equiv 1 \pmod{n}$, but $a \not\equiv \pm 1 \pmod{n}$, then n is not prime.

Exercise. Prove this fact.

There is also a (much more complicated) *deterministic* primality test, due to Agrawal, Kayal and Saxena.

18.310 Principles of Discrete Applied Mathematics
Fall 2013