Cryptography and RSA

1 RSA

Cryptography is concerned with the following scenario: Two people (we shall call them Alice and Bob) wish to communicate in the presence of an eavesdropper (Eve). Suppose that Alice wants to send to Bob secret message x. Cryptography provides solutions of the following form: Alice computes a function of x, $e(x)$, using a secret key, and sends $e(x)$ along the channel tapped by Eve. Bob receives $e(x)$, and using his key (which in traditional cryptography is the same as Alice's, but in modern cryptography it is not) computes a function $d(e(x)) = x$, recovering the message. Eve is presumably unable to recover x from $e(x)$ because she does not have Bob's key.

The most classical cryptographic method is the *one-time pad.* Alice and Bob have the same key k, as long as the message x (suppose without loss of generality that they are both bitstrings). Then Alice encodes x as $e(x) = x \oplus k$, and Bob decodes $e(x)$ as $d(e(x)) = (x \oplus k) \oplus k) = x$. Here $\oplus$ is the bit-wise exclusive or. One-time pad is nice and secure (can you prove that x can be recovered from $e(x)$ only if Eve knows k?), except for two crucial drawbacks:

- k has to be as long as the whole message. The message cannot be broken down to small fragments that will be encoded using the same key, because such repeated use of the key makes the scheme insecure, subject to frequency attacks (e.g., you can detect appearances of the letter e, the most frequent letter in English, in the same position).
- How do you exchange keys? Do you encrypt them, and how?

The *Data Encryption Standard* (DES), a U. S. government sponsored cryptographic method proposed in 1976, uses a key with 64 bits (although only 56 bits are really used) to compute, through an extremely complicated sequence of bit operations, an encoding (or decoding) function of any input bitstring with 64 bits (to encode longer files, you break them in 8-byte blocks). The proponents of DES claim that it is secure, its detractors suspect that, in subtle ways, it is not. Key size of 56 is widely believed now to be inadequate for serious cryptography, since 2^{56} is no longer a very large number.

Around the same time the DES was invented, a very exciting new idea was proposed: *Public-key cryptography.* Bob would have two keys, his private key k_d, known only to him, and his public key, k_e. k_e is known to everybody (it is on Bob's homepage, for example). If anybody, for example Alice, wants to send a message x to Bob, then she encrypts into $e(x)$ it using the public key. Bob decrypts it using his private key. The clever part is this: (1) Decryption is correct, that is, $d(e(x)) = x$; (2) You cannot feasibly compute Bob's private key from his public key, or x from $e(x)$, so the protocol is secure.

RSA (from the initials of Ron Rivest, Adi Shamir, and Len Adleman, the three computer scientists who invented it) is a clever way to realize the public key idea. Here is how it works:

- *Key Generation* Bob finds two large primes p and q. ("Large" nowadays means a couple of hundreds of decimal digits. To find such primes, Bob repeatedly generates odd integers in this range, and submits them to the Fermat test, until two of them pass it.) Then Bob computes $n = p \cdot q$. Also, Bob computes a random integer $e < n$, with the only restriction that e must be prime to $(p-1)$ and $(q-1)$. *The pair (n, e) is now Bob's* *public key*, and Bob announces it. (In practice, to make encoding easier (see below), e is often taken 3. Of course, we have to reject primes that are 1 mod 3.)

Now Bob must generate his secret key. All he has to do is compute (by Euclid's algorithm) $d = e^{-1} \bmod (p-1) \cdot (q-1)$. *The pair* (n, d) *is Bob's private key.*

- Operation Whenever Alice (or anybody else) wants to send a message to Bob, she does the following:
  - She first breaks the message into bitstrings of length $\lfloor \log n \rfloor$. She then encodes each bitstring by the following algorithm.
  - Let x be such a bitstring, and consider it an integer $\bmod n$. Alice computes x^e mod n. This is the encoded message $e(x)$ Alice sends to Bob.
  - Bob, upon receipt of $e(x)$, computes $e(x)^d \bmod n$. A little algebra (and recalling fact 4 from the previous lecture) gives:

    $$e(x)^d = x^{d \cdot e} = x^{1+m\cdot(p-1)\cdot(q-1)} = x \bmod n$$

    Notice that this sequence of equations establishes that Bob decodes correctly. The first equation recalls the definition of $e(n)$. The second follows from the fact that d is the inverse of e mod $(p-1)\cdot(q-1)$, and thus if you multiply d with e you get 1 plus a multiple of $(p-1)\cdot(q-1)$. The last inequality recalls fact 4: $x^{(p-1)\cdot(q-1)} = 1 \bmod n$ (unless of course x is a multiple of p or q, a very unlikely event); so, multiples of $(p-1)\cdot(q-1)$ in the exponent can be ignored.

So, Alice can encode using Bob's public key. Bob can decode, using the private key only he knows. How about an eavesdropper? If Eve gets $e(x) = x^e \bmod n$, she can do several things: She could try all possible x's encode them with Bob's public key, and find the correct one —that takes far too long. Or, she can factor n and compute d —nobody knows how to do this fast. Or, she could use her own ingenious method that decodes $e(x)$ without factoring n —it is widely believed that no such method exists.

So, in all available evidence, RSA is safe.with suitably large n.

2 A Few Important Ideas

In these first lectures we saw, besides a few algorithms like linear median-finding and RSA that are of great interest by themselves, several ideas that are both sophisticated and central to our business in this class. We hope you didn't miss them:

- *Rates of Growth.* We usually express the running time of algorithms in terms of its *rate of growth*, the $O(\cdot)$ notation. In analyzing algorithms we typically ignore constants. Constants are of course important in implementing an algorithm, but no useful theory can be based on them.

- *Polynomial vs. exponential.* Among all rates of growth, *polynomials* like $O(n^2)$ and $O(n^3)$ are considered satisfactory. In contrast, problems for which we can only find algorithms whose running time is an exponential function, like 2^n, $n!$, $2^{\sqrt{n}}$, etc., may be *intractible*, not solvable by algorithms in a satisfactory, practical way. Here n is the number of bits of the input. We have to be especially careful when the input is an integer N; in this case the size of the input is $\log N$ —so the $O(\sqrt{N})$ algorithm for testing and iterative exponentiation are *not* polynomial algorithms. In contrast, recursive exponentiation modulo a number, and therefore RSA, are polynomial algorithms.

There is a huge difference between exponential and polynomial algorithms. Here is a way of looking at this difference: Moore's law says that computing power doubles every 18 months. This means that every 18 months the size of the instance we are able to solve in reasonable time by a polynomial algorithm increases by a percentage (the percentage varies with the polynomial, e.g., 100% for a linear algorithm and about 25% for an $O(n^3)$ algorithm). But the instances that an 2^n algorithm can handle increase only by *one bit* every 18 months —even less for $n!$ algorithms.

- *Divide-and-conquer.* Some problems are endowed with such a favorable structure that they can be solved by recursion. Recursion works like magic, you just divide and glue, and somehow the work is done. The running time of such algorithms can be analyzed by the "master theorem." Unfortunately, there are very few problems that can be solved by this "dream" method.

- *Randomized algorithms.* It is often useful to employ randomization —algorithms that flip coins. Such algorithms are in no way inferior or less rigorous. One way of using randomization is for the purpose of analysis, as in median finding. A more radical use is when an algorithm may return a wrong answer, but the probability of this can be made so small that it does not matter, as in primality.