Cryptography

The idea of cryptography is to protect data by transforming into a representation from which the original is hard to recover. These days many networking technologies (internet, wireless) allow many agents to see a piece of data as it moves from its source to its destination. Those agents can capture the data in the representation sent across the network, and use it for their own purposes. Increasingly, people send data of great value (e.g. credit card numbers, secrets) through networks, so there is plenty of need for crytpography. In the future, most monetary transactions will probably go across the standard internet, and cryptography is an essential part of doing those transactions safely. There are two common types of cryptography:

Public-Key systems In a public-key system, the message M is encrypted using a key e which is public. That is, to encrypt the message compute $X = E(M, e)$ where E is the encryption function. To decrypt the encrypted message X, you compute $M = D(X, d)$, where d is the decryption key corresponding to e. The point of public-key systems is that knowing the encryption key e doesnt help a spy to discover the decryption key d.

Private-Key systems In a private-key system, the message M is encrypted using a key e which is known only to the sender and receiver. Once again, to encrypt the message compute $X = E(M, e)$ where E is the encryption function. To decrypt X, you compute $M = D(X, d)$, where d is the decryption key corresponding to e. In a private-key system, there is usually a simple relationship between the encryption and decryption keys, so knowing e would make it easy for a spy to intercept and decrypt a message. Probably the most widely used secret-key system is DES (the Data Encryption Standard).

RSA: A public-key crypto-system

The most famous public-key system is called RSA after its inventors Ron Rivest, Adi Shamir, and Len Adleman. Its very easy to describe RSA given what we know about additive and multiplicative groups of $\mathbb{Z}_n$. First of all, we assume the message is broken into chunks of the right size, say 1024 bits. In what follows, assume M is at most 1024 bits.

1. Generate a number n of at least 1024 bits which is a product of two large primes p and q. i.e. generate two primes of at least 512 bits and multiply them together.
2. Given p and q, recall that $\phi(n) = (p-1)(q-1)$ so it is easy to compute $\phi(n)$.
3. For the encryption key e, choose a value s.t. $\gcd(e, \phi(n)) = 1$.
4. Using the extended Euclid algorithm, find the multiplicative inverse of $e (\mathrm{mod}\ \ \phi(n))$, that is, find x such that $ex + \phi(n)y = 1$. This inverse is the decryption key d.

5. The public (encryption) key is the pair (e, n), while the decription key, which only the receiver knows is (d, n).

To send a message using RSA, the sender computes

$$X = M^e (\text{mod } n)$$

And then to decrypt the message, the receiver computes:

$$M_0 = X^d (\text{mod } n) = M^{ed} (\text{mod } n)$$

Now since $M^{\phi(n)} (\text{mod } n) = 1$ and ed is 1 + some multiple of $\phi(n)$,

$$M^{ed} (\text{mod } n) = M^1 (\text{mod } n) = M$$

So the recovered message M_0 is indeed equal to the original message M. Notice that the encryption and decryption functions are identical, that is:

$$E(X, (k, n)) = D(X, (k, n)) = X^k (\text{mod } n)$$

Complexity of RSA

We should check that we can do all the steps in RSA efficiently. Let's defer choosing the primes p and q for a moment. All the powering and gcd calculations are clearly in polynomial time in the number of bits of n. The other task is to find a number e such that $\gcd(e, \phi(n)) = 1$. From last time we know that the fraction of elements which are relatively prime to N is $\Omega(1/\log N)$. So setting $N = \phi(n)$, after $O(\log N)$ random trials for e, we should be able to get an e which is prime to $\phi(n)$. This is still all polynomial in the number of bits of n.

For generating primes, we can generate random numbers in the appropriate range and test them for primality. The prime number theorem asserts that about $1/\ln n$ of the numbers less than n are prime. So about one in $\ln n$ of the integers near n is a prime. Thus if we make $O(\log n)$ random choices, we will have high probability that one of our choices is a prime. So it is enough to show that there is an efficient test for primality. There are quite a few of these, but we will present one which is self-contained given what we know so far:

Algorithm Primality

Input: Odd number n and t

Output: PRIME or COMPOSITE

1. If n is a perfect power, then return COMPOSITE
2. Choose $b_1, b_2, \ldots, b_t$ independently and uniformly at random from $\mathbb{Z}_n - \{0\}$
3. If for any b_i, $\gcd(b_i, n) \neq 1$ then return COMPOSITE
4. Compute $r_i = b_i^{(n-1)/2} (\text{mod } n)$ for $i = 1, \ldots, t$

5. If for any i, $r_i \neq \pm 1 (\mathrm{mod}\ n)$, then return COMPOSITE

6. If for all i, $r_i = 1 (\mathrm{mod}\ n)$, then return COMPOSITE
   else return PRIME

Theorem
The probability that algorithm **Primality** makes an error is $O(1/2^t)$.

Proof
Clearly all the steps from 1 to 5 are correct. So we are left with checking step 6 in the two cases when n is prime or composite.

Suppose n is prime. We can output COMPOSITE if all of the r_i's evaluate to 1. But we know that for randomly chosen b_i's only half of them (those which are even powers of a generator) would give +1 when raised to the $(n-1)/2$. The probability that all the b_i's we chose happen to be even powers of a generator would be $1/2^t$. Thus we output the wrong answer with probability $1/2^t$.

If n is composite, the proof is more difficult and we wont give it here. But it can be shown that in that case, the probability of a wrong answer is $1/2^{t-1}$. QED

Finally, we should say something about checking if n is a perfect power. We want to check if $n = l^k$ for some integer l and k. We can do that by trying each $k = 1, 2, \ldots, \log(n)$. For each k, we compute the k^{th} root of n by Newton's method or bisection, which are both polynomial time in the number of bits of n. Overal this takes time polynomial in $\log n$.

Security of RSA

We would like to be convinced that RSA is a secure cryptographic scheme. For it to be so, we need a hypothesis:

Factoring is Hard It is believed to be very difficult to factor an integer n in the worst case. The worst case is where n comprises a small number of large factors. This has not been shown to be NP-complete, but the problem has resisted years of effort at finding efficient algorithms.

To see how factoring would help break RSA, notice that knowledge of the factors p and q of n is all that was needed in the key generation procedure described above. That is, given p and q, you can compute $\phi(n)$, and given $\phi(n)$ you can compute the decryption key from the encryption key using the extended Euclid algorithm.

So if factoring is easy, RSA is easy to break. But its possible that RSA is easy even if factoring is hard, because its not known how to go in the other direction - how to reduce factoring to breaking RSA.