CS174 Spr 99 **Lecture 25 Summary** **John Canny**

More about RSA

Recall that the key generation for RSA proceeds like this:

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

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

And then to decrypt the message, the receiver computes:

$$M_0 = X^d (\mathrm{mod}\ n) = M^{ed} (\mathrm{mod}\ n) = M^1 (\mathrm{mod}\ n)$$

Limitations

RSA, like any cryptographic scheme, must be applied very carefully. There are several basic weakness that you should be aware of:

Avoid small messages Remember that since your encryption key is public, anyone can generate encrypted messages of real text and match them against what you send. e.g. if you sent a message that was only “YES” or “NO”, a spy could precompute $E(“YES'', e)$ and compare it against what you sent. A more subtle version of this is where you fill out a long form and the message consists of a lot of text from the form (which is known) and a few answers that you fill in. A spy need only enumerate the answers and leave the form text intact to decode what you sent. One general answer to this is to pad your message with random bits. e.g. send a 512 bit string of real text and 512 bits of random data.

Beware pseudo-random number generators In some practical systems (e.g. netscape, IE), the key data like p and q and e are generated by a pseudo-random number generator. This random number generator is a deterministic program that outputs a sequence of numbers $r_1, r_2, \ldots$ which depends only on a seed value. The seeds are typically things like the CPU clock or some other dynamic system data. If too few seed bits are used, a spy can enumerate all possible seeds and therefore find out the p q and e that defined the real users key. (This actually happened a couple of years ago).

Publishing public keys is tricky A key idea of public-key systems is that the keys can be published "in the open" where anyone can read them. But on the internet, this can be tricky. Suppose you want to send a message to a friend Alice, and you look up her public key on a server. But in between you and the server is an insecure network node operated by Phiber Optic which intercepts Alice's key and sends you instead a different encryption key, for which Phiber knows the decryption key. If your encrypted message goes through Phiber's node again, he can intercept it and decode it. Many people place their public keys in their "finger" files on unix systems, but this makes them even more susceptible to the above attack. That's because anyone who wants to send them a private message will usually look up their key and send the message over the same network route, meaning that any of the intermediate nodes can both intercept and replace the key, and then intercept the encrypted message.

Finally, 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 $r_i = b_i^{(n-1)/2} \pmod{n}$ 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 $\log n$.

Secure Hash Algorithms

Another basic tool for cryptography is a secure hash algorithm. Unlike encryption, given a variable-length message x, a secure hash algorithm computes a function $h(x)$ which has a fixed number of bits. So it is not possible to recover x from its hash value. The hash function is secure if it is hard to get information about the x's that hash to a particular value. This normally specified in terms of the following properties:

1. A hash function $h(x)$ is said to be *one-way* if given y it is hard to find an x such that $h(x) = y$.

2. A hash function $h(x)$ is said to be *weakly collision-free* if given a message x_1 it is hard to find another message x_2 such that $h(x_1) = h(x_2)$.

3. A hash function $h(x)$ is said to be *strongly collision-free* if it is hard to find any pair of messages x_1, x_2 such that $h(x_1) = h(x_2)$.

and as the terminology suggests, a function which is strongly collision-free is also weakly collision-free.

One of the most popular secure hash algorithms is called MD5. It was invented by Ron Rivest, the R in RSA. The details of MD5 aren't very enlightening, and we wont go over it here. It is completely described in the Wayner book. At high level, it processes 512-bit blocks of data and produces 128-bit hash values. Thus it reduces the length of a large message by a factor of 4. To arrive at a hash value of fixed size, you iterate the process until the output is a single 128-bit value. As far as is known, MD5 is one-way and strongly collision-free.

Hash functions are very much like the fingerprint functions we used earlier. But one difference is that the simple modular functions we used for string matching are very easy to fool - given a string and its fingerprint, it is easy to generate other strings with the same fingerprint. That must not be the case for a secure hash function.

An important property of secure hash functions, like any hash function, is that they should uniformly cover their range. That is, if you place a uniform distribution on the inputs, the output probabilities from the hash function should be uniform. But we typically hope for much more. Namely that for any "reasonable" probability distribution on the inputs, the output probabilities should still be uniform. e.g. if the inputs consists of ASCII representation of normal english text (which is very non-uniform compared to random binary strings) the output distribution from the hash function should still be uniform. We will henceforth assume that is true.

Digital Signatures

The purpose of a digital signature is similar to a physical signature. That is, you endorse some document in a way that: (i) others can verify that you signed that particular document (ii) it is difficult for someone else to forge your signature. The simplest signature schemes allow anyone to verify both the document you signed, and who you are.

Here is a simple signature scheme. It assumes that you, the signer, have a published RSA key e, and a corresponding private key d. Given a document x, first compute a hash of it using MD5, $y = h(x)$. Your signature will be the RSA encryption of y by your *secret key*, which we write as $R(y, d)$ (we wont distinguish between encryption and decryption functions for RSA since they are the same, i.e. exponentiation by the key mod n). That is, your signature is

$$s \;=\; R(h(x), d)$$

Now anyone else that has access to the document x and your signature s can verify that you have signed it by computing $R(s, e)$ using your public key e. Since e and d are inverses,

$$R(R(h(x), d), e) \;=\; h(x)$$

Now the verifier (the person checking your signature) compares this value with the hash value that they compute directly from the document using MD5, which will also be $h(x)$.

Because MD5 is a secure hash function, it is impractical to construct another document with the same hash value. So the verifier knows that only this document could have been signed. Furthermore, because it is difficult for anyone else to discover your private key, the fact that you were able to compute $R(h(x), d)$ convinces the verifier that you know what d is. That is, you are the person you claim to be, and that you intended to sign this document.

Discrete Log Signatures

As we have seen, RSA cryptography fails if efficient methods are found for either factoring or the discrete log problem. There are other signature schemes that depend only on the hardness of the discrete log problem. Here is one such scheme. We assume that the full text x has already been shortened to an MD5 hash value m.

1. Let a be a secret key known only to you, the signer. Let p be a large prime, and g be a generator of $\mathbb{Z}_p^*$. You can publish $(g, p, g^a \pmod p)$ as your public key to identify who you are to the world.

2. In order to sign m, choose a random w and compute c as the hash of
   $c = h(m^a \pmod p), m^w \pmod p), g^w \pmod p))$

3. Let $r = ca + w$, you publish the digital signature which is m together with
   $(r, m^a \pmod p), m^w \pmod p), g^w \pmod p))$

4. To check the signature, a verifier first computes c as the hash of the values
   $m^a \pmod p), m^w \pmod p), g^w \pmod p)$
   which were published with the signature. Then the verifier checks that
   $g^r \pmod p) = (g^a)^c \times (g^w) \pmod p)$ and
   $m^r \pmod p) = (m^a)^c \times (m^w) \pmod p)$

Your goal is to convince the verifier that you know what a is. This protocol is a form of "zero-knowledge proof". You must convince the verifier that you know a without giving it away. Instead you give away an r which depends on a but which doesnt help the verifier learn a because you have added a random value w to it, which will make its distribution be completely random.

You can safely tell the verifier $g^a \pmod p)$ and $g^w \pmod p)$ because discrete log is hard, and so those values dont help the verifier discover a or w. The value c is really a "challenge" to you to prove that you know a. It is computed from a "random" hash function and is out of your control. If you didnt know a then when you were challenged with a c, the value $(g^a)^c \times g^w \pmod p)$ could be any element of $\mathbb{Z}_p^*$. So trying to find r satisfying

$$g^r \pmod p) = (g^a)^c \times g^w \pmod p)$$

is a general instance of the discrete log problem which is very hard. So the tests on g establish that you know a and that you are who you claim to be. By similar reasoning, only the person who knows a could construct the powers of m that are published as the signature. Thus those tests establish that you deliberately signed that document.