Algorithms in Systems Engineering
IE172

Lecture 28

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Section 8.3
- References
  - CLRS Chapter 31
  - Koblitz, *A Course in Number Theory and Cryptography*, Second Edition (1999).

RSA Public Key Encryption Algorithm

- The RSA algorithm is used almost universally to encrypt data on the Internet.
- If you have ever visited a secure site on the Internet, you have used RSA encryption.
- Procedure for creating public and private keys.
  - Randomly choose two large prime numbers p and q such that $p \neq q$.
  - Compute $n = pq$.
  - Select an odd integer e that is relatively prime to $\phi(n) = (p-1)(q-1) = n + 1 - p - q$.
  - Compute d as the multiplicative inverse of e modulo $\phi(n)$.
  - The pair (e, n) is the public key.
  - The pair (d, n) is the private key.
- The encoding function is $f_E(P) = P^e \mod n$.
- The decoding function is $f_D(C) = C^d \mod n$.

Some Questions

- Does the RSA algorithm actually yield a cryptosystem (are the encoding and encoding functions really inverses)?
- How secure is this system, i.e., is the encoding key really a trap door function?
- Can we compute the keys efficiently?
  - Can we find large prime numbers efficiently?
  - Does d always exist and can we compute it?
- Can we encode and decode efficiently?

How Secure is RSA Encryption?

- Can RSA encryption be broken?
- To break the scheme, we need to obtain d from e and n.
- The easiest known algorithm for obtaining d is to factor n.
- Hence, the security of the encryption scheme depends entirely on the difficulty of factoring large numbers.
- So far, no one has discovered a method for factoring large numbers efficiently.
- However, it also hasn't been proven that this *cannot* be done.
- To keep abreast of the current state-of-the-art in factoring, RSA offers cash prizes for factoring large numbers.

Factoring Algorithms

- The easiest algorithm for factoring an odd integer n is *trial division*.
  - Try dividing n by each odd integer less than $\sqrt{n}$.
  - This method works for numbers that have prime factors near $\sqrt{n}$, but is not practical for most purposes.
- *Fermat's factoring algorithm* is based on the observation that that n is the product of two integers if and only if it is the difference of two squares.

$$n = ab = ((a+b)/2)^2 - ((a-b)/2)^2 = t^2 - s^2$$

  - If a and b are close together, then s is small and t is near $\sqrt{n}$.
  - Start with $t = \lceil \sqrt{n} \rceil$ and check whether $t^2 - n$ is a square number.
  - There are only 22 combinations of the last two digits of a square number, so many numbers can be quickly shown not to be square.
  - Continue by increasing t by 1.
- There are more efficient and more complex algorithms based on this general principle, but none are efficient for large numbers.

Generating the Public Key

- Because the security of the system depends on the difficulty of factoring n, we want n to be as large as possible.
- There are tradeoffs, however, because a large n makes the keys harder to compute and also makes the encoding and decoding more difficult.
- In addition, generating large prime numbers can be difficult.
- When choosing the factors p and q, we should endeavor to choose them to be large, but not too close together.
- Large numbers with two prime factors close together are easy to factor by Fermat's Algorithm or others.
- To find e, we can just try some random choices.

Generating Large Prime Numbers

- One can systematically find all primes less than n using a *sieve*.
- This method is not efficient enough to find large primes.
- There really is no efficient direct method for finding large primes.
- In fact, even verifying that a large number *is* prime can be difficult.
- There are tests that ensure it is *highly probable* the n is prime, but do not provide a guarantee.
- The probability that a random integer n is prime is approximately $1/\ln n$.
- In practice, we can just try random large integers and try to prove they are prime.

Proving Primality

- Again, we can try to prove primality by trial division, but this is not practical for large integers.
- Fermat's theorem says that if n is prime, then

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

  for all positive integers $a < n$.
- In fact, the converse is *almost* true.
- Checking to see whether this equation is satisfied for $a = 2$ is a very accurate test.
- This involves computing $2^{n-1} \pmod{n}$.
- This can be done efficiently using *repeated squaring* (Section 31.6).

Generating the Private Key

- The multiplicative inverse of e modulo $\phi(n)$ exists if and only if e is relatively prime to $\phi(n)$ (which we require).
- Then d can be computed by solving a modular equation (Section 31.4).
- This is done using the extended version of Euclid's Algorithm to find the gcd of e and $\phi(n)$ (which we know is one).
- Extended Euclid's Algorithm for finding the multiplicative inverse of m modulo n (assuming m and n are relatively prime).
  - Divide m by n and let r be the remainder.
  - If $r = 0$, then return $(m, 1, 0)$.
  - Otherwise, recursively call the function with arguments n and r to obtain (d', x', y').
  - Return $(d', y', x' - \lfloor m/n \rfloor y')$.
- If the final return values are (d, x, y), where $d = \gcd(m, n) = 1$ and $d = 1 = mx + ny$.
- This means that x is the multiplicative inverse of m modulo n.

Some Messy Details

- Note that in this cryptosystem, the ciphertext alphabet is not quite the same as the plaintext alphabet.
- The ciphertext alphabet depends on the choice of n.
- To take care of this, we need to choose the size of the message units in the plaintext and the ciphertext alphabets properly.
- Our scheme for digital signatures also breaks down with RSA encryption, but this can also be taken care of.