Introduction to Cryptography
Lecture 8

Primality testing, factoring, computing discrete logs

Benny Pinkas

Primality testing

- Why do we need primality testing?
  - Essentially all public key cryptographic algorithms use large prime numbers
  - We therefore need an algorithm for prime number generation
  - Suppose we have an algorithm "PrimalityTest" with a binary output.
  - We can generate random primes as follows

```
GeneratePrime(a,b)

  1. Choose random number x ∈ [a,b]

  2. If PrimalityTest(x) then output "x is
  prime"; otherwise goto line 1.
```

Density of prime numbers

- How long will GeneratePrime run?
- Let $\pi(n)$ specify number of primes $\leq n$.

- Prime number theorem:
  - $\pi(n)$ goes to $n / \ln n$ as n goes to infinity.
- Pretty accurate even for small n (e.g. for $n=2^{30}$ it is off by 6%).

- Corollary: a random number in [1,n] is prime with probability $1/\ln n$. (e.g. for $n=2^{512}$, probability is *1/355*).
  - The GeneratePrime algorithm is expected to take ln n rounds.
  - If we skip even numbers, we cut running time by ½.

Primality testing

- Trial division
  - Try to divide x by every prime integer smaller than $\sqrt{x}$ (sqrt(x)).
  - Infeasible for large x.

- Primality testing is a decision problem: “is x prime or composite?”
- Different than the search problem “find all prime factors of x”.
- In this case, the decision problem has an efficient solution while the search problem does not.

Fermat's test

- Fermat's theorem: if x is prime then for all $1 \le a < x$ it holds that $a^{x-1} = 1 \bmod x$.

- If we can find an a s.t $a^{x-1} \ne 1 \bmod x$, x is surely composite.
  - Surprisingly, the converse is almost always true, and for a large percentage of the choices of a.
  - Suppose we check only for $a=2$.
    - If $2^{x-1} != 1 \bmod x$
      - Then return COMPOSITE /for sure
      - Otherwise, return PRIME /we hope
  - How accurate is this program?

Fermat's test

- Surprisingly, this test is almost always right
  - Wrong for only 22 values of x smaller than 100,000
  - Probability of error goes down to 0 as x grows
    - For |x|=512 bits, probability of error is < 10^{-20}
    - For |x|=1024 bits, probability of error is < 10^{-41}
- The test is therefore sufficient for randomly chosen candidate primes
- But we need a better test if x is not chosen at random
- Cannot eliminate errors by checking for bases ≠ 2
  - x is a Charmichael number if it is composite, but $a^{x-1} = 1 \bmod x$ for all $1 \le a < x$.
  - There are infinitely many Charmichael numbers
  - But they are rare.

Miller-Rabin test

- Works for all numbers (even Charmichael numbers).
  - Checks several randomly chosen bases a
  - If it finds out that $a^{x-1} = 1 \bmod x$, it checks whether a nontrivial root of 1 $(\neq 1,-1)$ was generated in the process. If so, outputs COMPOSITE.

The Miller-Rabin test:

1. Write $x-1=2^c r$ for an odd r. set comp=0.
2. For i=1 to T
   - Pick random a $\in$ [1,x-1]. If gcd(a,x)> 1 set comp=1.
   - Compute $y_0=a^r \bmod x$, $y_i=(y_{i-1})^2 \bmod x$ for $i=1..c$. If $y_c\neq 1$, or $\exists i$, $y_i=1$, $y_{i-1}\neq\pm 1$, set comp=1.
3. If comp=1 return PRIME, else COMPOSITE.

Miller-Rabin test

- Possible values for the sequence $y_0=a^r$, $y_1=a^{2r}$,… $y_c=a^{r2^{\wedge}c}=a^{x-1}$.
  - <…,d>, where $d\neq 1$, decide COMPOSITE.
  - <1,1,…,1>, decide PRIME.
  - <..,-1,1,..,1>, decide PRIME.
  - <…,d,1,…,1>, where $d\neq\pm 1$, decide COMPOSITE.
  - For a composite number x, we denote a base a as non-witness if it results in the output being "PRIME".
- Lemma: if x is an odd composite number then the number of non-witnesses is at most $x/4$.
- Therefore, for any odd integer x, T trials give the wrong answer with probability $< (1/4)^T$.

Integer factorization

- The RSA and Rabin cryptosystems use a modulus N and are insecure if it is possible to factor N.
- Factorization: given N find all prime factors of N.
- Factoring is the search problem corresponding to the primality testing decision problem.

Pollard's Rho method

- Factoring N
- Division by all integers $< N^{1/2}$.
- Pollard's rho method:
  - $O(N^{1/4})$ computation.
  - O(1) memory.
  - A heuristic algorithm.

Pollard's rho method

```
1. i=1; x_1∈[1,n-1]; y=x_1;
2. i = i+1.
3. x_i = ((x_{i-1})^2 - 1) mod n.
4. d = gcd(y-x_i, n)
5. If d>1 then print d, and stop.
6. If i is a power of 2, then y=x_i
7. Goto line 2.
```

Always a factor of n

- x_i is a series of numbers in *0..n-1*.
- y takes the values of $x_1, x_2, x_4, x_8, \ldots, x_{2^i}, \ldots$
- If $(y-x_i) = 0 \bmod p$, then most likely $\gcd(y-x_i, n)=p$.

Pollard's rho method

- The running time is not guaranteed, but is expected to be sqrt(p).
- The sequence x_i is in 1..n.
  - x_i depends only on x_{i-1} $\quad (x_i = ((x_{i-1})^2 - 1) \bmod n)$
  - The sequence is shaped like the letter Rho.
  - Assume that $f_n(x)=x^2-1 \bmod n$ behaves like a random function. Then the tail and the circle are about sqrt(n) long.
- Let $x'_i = x_i \bmod p$, where p factors n.
- $x'_{i+1} = x_{i+1} \bmod p = (x_i^2-1 \bmod n) \bmod p = x_i^2-1 \bmod p = (x_i')^2-1 \bmod p$
- The sequence x'_i therefore follows x_i, but is in $0..p-1$. Its tail and circle are about sqrt(p) long.

Pollard's rho method

- The sequence x'_i:
  - Let t be the first repeated value in x'_i
  - Let u be the length of the cycle
  - $x'_{t+i} = x'_{t+i+u}$
  - Therefore $x_{t+i} = x_{t+i+u} \bmod p$
  - $gcd(x_{t+i} - x_{t+i+u}, n) = cp.$

- Once the algorithm saves $y=x_j$ for $j>t$, it is on the circle. If the circle length u is smaller than j, the algorithm computes $gcd(x_{j+u}-x_j, n)$ and factors n.

- The algorithm fails if
  - The cycle and tail are long -> running time is slow.
  - The cycle and tail are of the same length for both p and q.

Modern factoring algorithms

- The number-theoretic running time function $L_n(a,c)$

$$L_n(a,c) = e^{c(\ln n)^a (\ln\ln n)^{1-a}}$$

  - For a=0, the running time is polynomial in ln(n).
  - For a=1, the running time is exponential in ln(n).
  - For 0<a<1, the running time is subexponential.

- Factoring algorithms
  - Quadratic field sieve: $L_n(1/2, 1)$
  - General number field sieve: $L_n(1/3, 1.9323)$
  - Elliptic curve method $L_p(1/2, 1.41)$ (preferable only if p<<sqrt(n))

Modulus size recommendations

- Factoring algorithms are run on massively distributed networks of computers (running in their idle time).
- RSA published a list of factoring challenges.
- A 512 bit challenge was factored in 1999.
- The largest factored number $n=pq$.
  - 576 bits (RSA-576)
  - Factored on Dec 3, 2003 using the NFS

- Typical current choices:
  - At least 768-bit RSA moduli should be used
  - For better security, 1024-bit RSA moduli are used
  - For more sensitive applications, key lengths of 2048 bits (or higher) are used

Discrete log algorithms

- Input: *(g,y)* in a finite group G. Output: x s.t. $g^x = y$ in G.
- Generic vs. special purpose algorithms: generic algorithms do not exploit the representation of group elements.

- Algorithms
  - Baby-step giant-step: Generic. |G| can be unknown. Sqrt(|G|) running time and memory.
  - Pollard's rho method: Generic. |G| must be known. Sqrt(|G|) running time and O(1) memory.
  - No generic algorithm can do better than O(sqrt(q)), where q is the largest prime factor of |G|
  - Pohlig-Hellman: Generic. |G| and its factorization must be known. O(sqrt(q) ln q), where q is largest prime factor of |G|.
  - Therefore for Z^*_p, p-1 must have a large prime factor.
  - Index calculus algorithm for Z^*_p: L(1/2, c)
  - Number field size for Z^*_p: L(1/3, 1.923)

Baby-step giant-step DL algorithm

- Let t=sqrt(|G|).
- x can be represented as $x=ut-v$, where u,v < sqrt(|G|).

- The algorithm:
  - Giant step: compute the pairs $(j, g^{j \cdot t})$, for $0 \le j \le t$. Store in a table keyed by $g^{j \cdot t}$.
  - Baby step: compute $y \cdot g^i$ for $i=0,1,2\ldots$, until you hit an item $(j, g^{j \cdot t})$ in the table. x $= jt - i$.

- Memory and running time are O(sqrt|G|).

Baby-step giant-step DL algorithm