Lecture 10: Number Theory for Public Key Cryptography

COSC362 Data and Network Security

Book 1: Chapter 2

Spring Semester, 2021

Motivation

- Number theory problems used in public key cryptography.
- Need of efficient ways to generate large prime numbers in order to use such problems.
- Definitions of hard computational problems to base cryptosystems on.

Outline

Outline

Chinese Remainder Theorem

Euler Function

Primality Tests
- Fermat Test
- Miller-Rabin Test

Basic Complexity Theory

Factorisation Problem

Discrete Logarithm Problem

Chinese Remainder Theorem (CRT)

Theorem: (this is a special case!)

- Let p, q be relatively prime.
- Let $n = p \times q$ be the modulus.
- Given integers c_1 and c_2, there exists a unique integer x, $0 \leq x < n$, s.t.:

$$\begin{aligned} x &\equiv c_1 \pmod{p} \\ x &\equiv c_2 \pmod{q} \end{aligned}$$

- $x \equiv \frac{n}{p} y_1 c_1 + \frac{n}{q} y_2 c_2 \pmod{n}$ where:
  - $y_1 \equiv \left(\frac{n}{p}\right)^{-1} \pmod{p} = q^{-1} \pmod{p}$ (rewriting as $q y_1 \equiv 1 \pmod{p}$)
  - $y_2 \equiv \left(\frac{n}{q}\right)^{-1} \pmod{q} = p^{-1} \pmod{q}$ (rewriting as $p y_2 \equiv 1 \pmod{q}$)

Example

Solve $x \equiv 5 \pmod{6}$ and $x \equiv 33 \pmod{35}$:

- $c_1 = 5$ and $c_2 = 33$
- $p = 6$ and $q = 35$ are relatively prime, so CRT can be used
- $n = 6 \times 35 = 210$

$$\begin{array}{r l|r l} \frac{210}{6} y_1 & \equiv 1 \pmod{6} & \frac{210}{35} y_2 & \equiv 1 \pmod{35} \\ 35 y_1 & \equiv 1 \pmod{6} & 6 y_2 & \equiv 1 \pmod{35} \\ y_1 & \equiv 5 \pmod{6} & y_2 & \equiv 6 \pmod{35} \end{array}$$

$$\begin{aligned} x &\equiv \frac{n}{p} y_1 c_1 + \frac{n}{q} y_2 c_2 \pmod{n} \\ &\equiv (35 \times 5 \times 5) + (6 \times 6 \times 33) \pmod{210} \\ &\equiv 175 \times 5 + 36 \times 33 \pmod{210} \equiv 173 \pmod{210} \end{aligned}$$

Outline

Euler Function

Definition: given a positive integer *n*, the Euler function $\phi(n)$ denotes the number of *positive* integers *less than n* and *relatively prime* to *n*.

Example: $\phi(10) = 4$ since $1, 3, 7, 9$ are each relatively prime to 10.
The set of positive integers less than $n = 10$ and relatively prime to $n = 10$ is thus $\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$.

Properties

- $\phi(p) = p - 1$ where p is prime.
- $\phi(pq) = (p-1)(q-1)$ where p, q are distinct primes.
- $n = p_1^{e_1} \cdots p_t^{e_t}$ where p_i are distinct primes, then:

$$\phi(n) = \prod_{i=1}^{t} p_i^{e_i - 1}(p_i - 1)$$

Examples:

$\phi(15) = (3-1)(5-1) = 2 \times 4 = 8$ since $15 = 3 \times 5$
$\phi(24) = 2^2(2-1) \times 3^0(3-1) = 8$ since $24 = 2^3 \times 3$

Important Theorems

Fermat's theorem: Let p be a prime, then for any integer a s.t. $1 < a \leq p - 1$:

$$a^{p-1} \mod p = 1$$

Euler's theorem: If $\gcd(a, n) = 1$ then:

$$a^{\phi(n)} \mod n = 1$$

When p is prime then $\phi(p) = p - 1$, so Fermat's theorem is a special case of Euler's theorem.

Outline

Primality Tests

- Testing for primality by trial division is not practical (except for very small numbers).
- Many *probabilistic* methods:
  - Requiring random inputs
  - Possible failure in exceptional circumstances
- Indian mathematicians Agrawal, Saxena and Kayal found a polynomial time deterministic primality test (2002):
  - Huge theoretical breakthrough
- Probabilistic methods are still used in practice.

Fermat Primality Test

- *Fermat's theorem:* If p is prime then $a^{p-1} \mod p = 1$ for all a s.t. $\gcd(a, p) = 1$.
- If a number n s.t. $a^{n-1} \mod n \neq 1$, then n is NOT prime.
- *Test idea:* If a number satisfies Fermat's equation then we ASSUME that it is prime.
- Failure is possible but very unlikely in practice.
- Reducing the failure probability by repeating the test with different base values a.

Fermat Primality Test

- *Inputs:*
  - a value n to test for primality
  - a parameter k that determines the number of times the test is run
- *Output:* `composite` if n is composite; `probable prime` otherwise.
- *Algorithm:*
  - Pick a at random s.t. $1 < a < n - 1$.
  - If $a^{n-1} \mod n \neq 1$ then return `composite`; otherwise return `probable prime`.

Effectiveness

- If the test outputs `composite` then n is definitely composite.
- The test can output `probable prime` if n is composite:
  - n is called *pseudoprime*
- The test ALWAYS outputs `probable prime` for some composite n:
  - n is called *Carmichael number*
  - First few Carmichael numbers: 561, 1105, 1729, 2465, etc.

Miller-Rabin Test

- Similar idea to Fermat test
- Guaranteeing to detect composite numbers if test run sufficiently many times
- Most widely used test for generating large prime numbers

Square Roots of 1

- A *modular square root of 1* is a number x s.t. $x^2 \mod n = 1$.
- There are 4 square roots of 1 when $n = pq$:
  - 2 are 1 and -1 modulo n
  - 2 are called *non-trivial* square roots of 1
- If x is a non-trivial square root of 1 then $\gcd(x - 1, n)$ is a non-trivial factor of n:
  - Hence, if a non-trivial square root of 1 exists then n is composite

Miller-Rabin Algorithm

Let n and u be odd, and v s.t. $n - 1 = 2^v u$:

1. Pick a at random s.t. $1 < a < n - 1$
2. Set $b = a^u \mod n$
3. If $b = 1$ then return `probable prime`
4. For $j = 0$ to $v - 1$:
   - If $b = -1$ then return `probable prime`
   - Else set $b = b^2 \mod n$
5. Return `composite`

Note: when an output is returned, the algorithm halts.

Effectiveness

- If test returns `composite` then n is composite.
- If test returns `probable prime` then n MAY be composite.
- If n is composite then test returns `probable prime` with probability at most 1/4:
  - Algorithm is run k times
  - Repeat while the output is `probable prime`
  - Output `probable prime` when n is composite with probability no more than $(1/4)^k$
- In practice, error probability is smaller:
  - *Example:* no composites less than 341,550,071,728,321 which pass the test for 7 base values $a = 2, 3, 5, 7, 11, 13, 17$.

Why Does Miller-Rabin Test Work?

- Random a s.t. $0 < a < n - 1$, and $n - 1 = 2^v u$.
- Sequence $a^u, a^{2u}, \cdots, a^{2^{v-1}u}, a^{2^v u} \mod n$.
- Each number in the sequence (after the 1st) is the square of the previous number.
- If n is prime then $a^{2^v u} \mod n = 1$ (Fermat's theorem), and then:
  - Either $a^u \mod n = 1$
  - Or there is a square root of 1 somewhere in the sequence, and the value must be -1
- If a non-trivial square root of 1 is found then n is composite.

Example

- Let $n = 1729$ be a Charmichael number.
- $n - 1 = 1728 = 64 \times 27 = 2^6 \times 27$.
- Hence $v = 6$ and $u = 27$.

1. Choose $a = 2$
2. $b = 2^{27} \mod 1729 = 645$
3. Since $b \neq 1$, continue:
   - $b = 645^2 \mod n = 1065$
   - $b = 1065^2 \mod n = 1$
   - Thus $b = -1$ will never occur
4. Return `composite`

Note: 1065 is a non-trivial square root of 1 modulo 1729, since $\gcd(1729, 1064) = 133$ is a factor of 1729.

Generation of Large Primes

Miller-Rabin test used to generate large primes:

1. Choose a random odd integer r of the same number of bits as the required prime.
2. Test if r is divisible by any of a list of small primes.
3. Apply Miller-Rabin test with 5 random (or fixed) base values a.
4. If r fails any test, then set $r = r + 2$ and return to Step 2.

Note: this *incremental* method does not produce completely random primes.
Instead, start from Step 1 if r is found to be composite. Both options are seen in practice.

Outline

Complexity Theory in Cryptology

- Computational complexity provides a foundation for:
  - Analysing computational requirements of cryptanalytic techniques.
  - Studying the difficulty of breaking ciphers.
- 2 aspects:
  - *Algorithm complexity:* how long does it take to run a particular algorithm?
  - *Problem complexity:* what is the best (known) algorithm to solve a particular problem?

Algorithm Complexity

- Computational complexity of an algorithm is measured by its time and space requirements, measured as functions of the size of the input m.
- "big O" notation:
  - $f(m), g(m)$ are 2 positive functions
  - $f(m)$ is expressed as an "order of magnitude" of the form $O(g(m))$
  - $f(m) = O(g(m))$ if there are constants $c > 0$ and m_0 s.t. $f(m) \leq c \cdot g(m)$ for $m \geq m_0$

Polynomial and Exponential Functions

- Polynomial time function: function $f(m) = O(m^t)$ for some positive integer t:
  - Polynomial time function is seen as *efficient* in cryptography.
- Exponential time function: function $f(m) = O(b^m)$ for some number $b > 1$:
  - A problem whose the best solution is an exponential time function is seen as *hard* in cryptography.
- Brute force key search is *exponential* as a function of the key length:
  - An m-bit key length allows 2^m keys.

Examples of Algorithm Complexity

1. Let $f(m) = 17m + 10$, then $f(m) = O(m)$:
   - $17m + 10 \leq 18m$ for $m \geq 10$
2. Let $f(m) = a_0 + a_1 m + \cdots + a_t m^t$ be a polynomial, then $f(m) = O(m^t)$

Problem Complexity

- A problem is classified according to the minimum time and space needed to solve its hardest instances on a deterministic computer.
- *Examples:* polynomial time problems
  - Multiplication of 2 $m \times m$ matrices, with fixed size entries, using the obvious algorithm is $O(m^3)$.
  - Sorting a set of integers into ascending order is $O(m \cdot \log_2 m)$ with algorithms such as Quicksort.

Hard Problems

- *Integer factorisation:* given an integer of m bits, find its prime factors.
- *Discrete logarithm problem (with base 2):* given a prime p of m bits and an integer y s.t. $0 < y < p$, find x s.t. $y = 2^x \mod p$.
- There are no known polynomial algorithms to solve these problems.
- The best known algorithms are *sub-exponential*:
  - Slower than any polynomial algorithm but faster than any exponential algorithm.

Outline

Integer Factorisation:

- Factorisation by trial division is an exponential time algorithm:
    - Hopeless for numbers of few hundred bits.
- Several methods exist:
    - They apply if the integer to be factorised has SPECIAL properties.
- The best current general method is *number field sieve*:
    - Sub-exponential algorithm.

Factorisation Records

Decimal digits	Bits	Date	CPU years
140	467	Feb. 1999	?
155	512	Aug. 1999	?
160	533	Mar. 2003	2.7
174	576	Dec. 2003	13.2
200	667	May. 2005	121
232	768	Dec. 2009	3300

- All records used number field sieve method.
- Assuming 1 GHz CPU.

`http://en.wikipedia.org/wiki/RSA_numbers`

Comparing Brute Force Search and Factorisation

Symmetric key length	Length of $n = pq$
80	1024
112	2048
128	3072
192	7680
256	15360

- *Example:* Brute force key search of 128-bit keys for AES takes roughly the same computational effort as factorisation of a 3072-bit number with 2 factors of roughly equal size.
- NIST SP 800-57 Part 1: Recommendations for Key Management (2016).

Outline

Discrete Logarithm Problem

- g is a generator of $\mathbb{Z}_p^*$ for a prime p.
- Discrete logarithm problem over $\mathbb{Z}_p^*$ is:
  - Given $y \in \mathbb{Z}_p^*$, find x s.t. $y = g^x \mod p$.
- If p is large enough, then the problem is believed to be hard.
- The best known algorithm is a variant of the number field sieve.
- Length of modulus p should be chosen of same length as RSA modulus for the same security level (at least 2048 bits).

Example

$g^x \bmod p$	x	$g^x \bmod p$	x
1	18	10	17
2	1	11	12
3	13	12	15
4	2	13	5
5	16	14	7
6	14	15	11
7	6	16	4
8	3	17	10
9	8	18	9

- Set of non-zero integers modulo 19 is $\mathbb{Z}_{19}^*$
- Generator is $g = 2$
- When $y = g^x \bmod p$ then $\log_g(y) = x$
- Example: $\log_2(3) = 13$

Comparing Brute Force Search, Factorisation and DL

Symmetric key length	Length of $n = pq$	Length of discrete log modulus p
80	1024	1024
112	2048	2048
128	3072	3072
192	7680	7680
256	15360	15360

- *Example:* brute force key search of 128-bit keys for AES takes roughly the same computational effort as factorisation of a 3072-bit number with 2 factors of roughly equal size, or finding discrete logs with a 3072-bit modulus.
- NIST SP 800-57 Part 1 (2016).