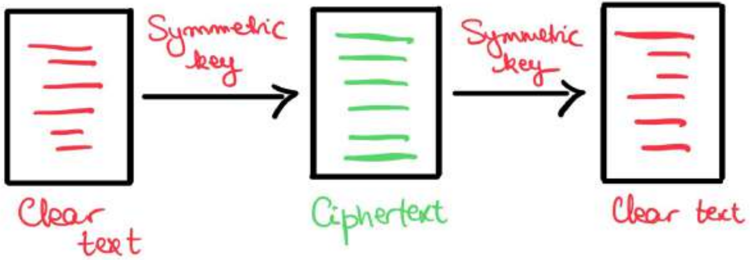
Lecture 8: Number Theory for Public Key Cryptography

TTM4135

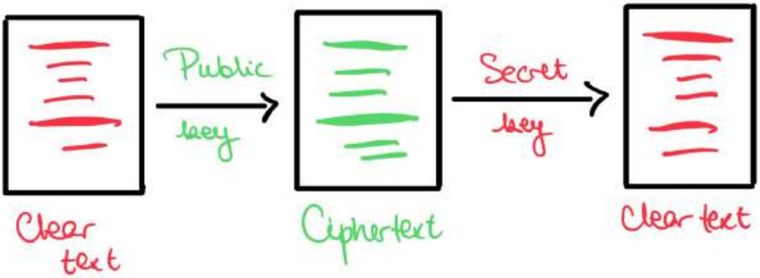
Relates to Stallings Chapters 2 and 5

Spring Semester, 2025

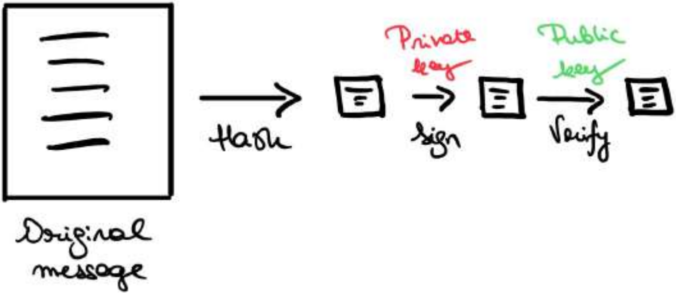
Reminder – symmetric key encryption



Reminder – public key encryption



Reminder – signatures



Motivation

- Number theoretic problems are at the foundation of public key cryptography (e.g encryption, signatures) in use today.
- In order to use these problems we need efficient ways to generate large prime numbers.
- We also need to define hard computational problems which we can base our cryptosystems on.

Outline

- Chinese remainder theorem
- Euler function ϕ
- Testing for primality
  - Fermat Test
  - Miller–Rabin Test
- Some Basic Complexity Theory
- Factorisation problem
- Discrete logarithm problem

Chinese remainder theorem

Theorem

Let $d_1, d_2, \ldots, d_r$ be pairwise relatively prime and $n = d_1 d_2 \ldots d_r$. Given any integers c_i there exists a unique integer x with $0 \leq x < n$ such that

$$\begin{array}{rcll} x & \equiv & c_1 & \pmod{d_1} \\ x & \equiv & c_2 & \pmod{d_2} \\ & \vdots & & \\ x & \equiv & c_r & \pmod{d_r} \end{array}$$

In fact $x \equiv \sum(\frac{n}{d_i})\, y_i\, c_i \pmod{n}$ where $y_i \equiv (\frac{n}{d_i})^{-1} \pmod{d_i}$.

Example

$$\text{Solve} \quad \begin{aligned} x &\equiv 5 \pmod{6} \\ x &\equiv 33 \pmod{35} \end{aligned}$$

Since 6 and 35 are relatively prime we can use CRT. Set $n = 6 \times 35 = 210$.

$$\begin{aligned} \tfrac{210}{6} y_1 &\equiv 1 \pmod{6} & \tfrac{210}{35} y_2 &\equiv 1 \pmod{35} \\ 35 y_1 &\equiv 1 \pmod{6} & 6 y_2 &\equiv 1 \pmod{35} \\ y_1 &\equiv 5 \pmod{6} & y_2 &\equiv 6 \pmod{35} \end{aligned}$$

$$\begin{aligned} x &\equiv \sum (\frac{n}{d_i})\, y_i\, c_i \pmod{n} \\ &\equiv (35 \times 5 \times 5) + (6 \times 6 \times 33) \pmod{210} \\ &\equiv 175 \times 5 + 36 \times 33 \pmod{210} \\ &\equiv 173 \pmod{210} \end{aligned}$$

Euler function ϕ

Definition

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

- Recall that *a* and *b* relatively prime is the same as $\gcd(a, b) = 1$.
- The set of positive integers less than *n* and relatively prime to *n* form the reduced residue class $\mathbb{Z}_n^*$.
  - So in particular, $\phi(n)$ gives us the *size* of $\mathbb{Z}_n^*$.

Example

$\phi(10) = 4$ since 1,3,7,9 are each relatively prime to 10.

$\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$

Properties of $\phi(n)$

1. $\phi(p) = p - 1$ for p prime
2. $\phi(pq) = (p-1)(q-1)$ for p and q distinct primes
3. Let $n = p_1^{e_1} \dots p_t^{e_t}$ where p_i are distinct primes. Then

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

where $\prod$ represents the product

Example

$$\begin{array}{rcccc} \phi(15) & = & 2 \times 4 & = & 8 \\ \phi(24) & = & 2^2(2-1)3^0(3-1) & = & 8 \\ & & (\text{where } 24 = 2^3 \times 3) & & \end{array}$$

Two important theorems

Theorem (Fermat)

Let p be a prime. Then

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

for all integers a with $1 < a < p - 1$

Theorem (Euler)

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

if $\gcd(a, n) = 1$.

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

Testing for primality

- Testing for primality by trial division is not practical except for very small numbers
- There are a number of fast methods which are *probabilistic*: they require random input and can fail in exceptional circumstances
- In 2002, three Indian mathematicians, Agrawal, Saxena and Kayal, found a polynomial time deterministic primality test. Although a huge theoretical breakthrough, the probabilistic methods are still used in practice.
- We examine one of the simplest tests: the *Fermat primality test* and then extend it to the *Miller–Rabin test*

Outline

Fermat primality test

- Recall that Fermat's theorem says that *if* a number p is prime then $a^{p-1} \bmod p = 1$ for all a with $\gcd(a, p) = 1$
- If we examine a number n and find that $a^{n-1} \bmod n \neq 1$ then we know that n is *not* prime
- This is essentially the Fermat primality test: if a number satisfies Fermat's equation then we assume that it is prime
- The Fermat primality test can fail with some probability
- We reduce the failure probability by repeating the test with different base values a

Fermat primality test

Inputs:
- n: a value to test for primality;
- k: a parameter that determines the number of times to test for primality

Output: `composite` if n is composite, otherwise `probable prime`

Algorithm: repeat k times:

1. pick a randomly in the range $1 < a < n - 1$
2. if $a^{n-1} \bmod n \neq 1$ then return `composite`

return `probable prime`

Effectiveness of Fermat test

- If the test outputs `composite` then n is definitely composite
- The test can can output `probable prime` if n is composite. In this case n is called a *pseudoprime*.
- There are some composite numbers for which the test will always output `probable prime` for every a with $\gcd(a, n) = 1$: these are called *Carmichael numbers*
- First few Carmichael numbers are: 561, 1105, 1729, 2465, ...

Carmichael Numbers

A Carmichael number n is a *composite* number that satisfies

$$b^{n-1} \equiv 1 \pmod{n},$$

for all integers b. Carmichael numbers constitute the (rare) instances where the converse of Fermat's theorem does not hold. There are infinitely many such numbers.

Outline

Miller–Rabin test

- Same idea as Fermat test
- Can be guaranteed to detect composites if 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 with $x^2 \bmod n = 1$
- When $n = pq$ there are 4 square roots of 1
- Two of these are 1 and -1 modulo n
- The other two 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-trival factor of n
- In other words, the existence of a non-trivial square root implies that n is composite

Miller–Rabin algorithm

Assume that n is odd and define u, v such that $n - 1 = 2^v u$, where u is odd

1. Pick a randomly in the range $1 < a < n - 1$
2. Set $b = a^u \bmod n$
3. If $b == 1$ then return `probable prime`
4. For $j = 0$ to $v - 1$
   4.1 If $b == -1$ then return `probable prime`
   4.2 Else set $b = b^2 \bmod n$
5. Return `composite`

Note that when any output is returned the algorithm halts

Effectiveness of Miller–Rabin

- If the test returns `composite` then n is composite
- If the test returns `probable prime` then n *may* be composite
- If n is composite the test returns `probable prime` with probability at most $1/4$
- Therefore we repeat the algorithm k times while the output is `probable prime`
- The k-times algorithm will output `probable prime` when n is composite with probability no more than $(1/4)^k$
- In practice error probability is far smaller
- There are no composites less than 341,550,071,728,321 which pass the test for the seven bases $a = 2, 3, 5, 7, 11, 13, 17$

Why Miller–Rabin works

- Consider the sequence $a^u, a^{2u}, \ldots, a^{2^{v-1}u}, a^{2^v u} \bmod n$, where a is random with $0 < a < n-1$
- Each number in this sequence, after the first, is the square of the previous number
- If n is prime then Fermat's theorem tells us that the final value, $a^{2^v u} \bmod n = 1$
- Therefore if n is prime then either $a^u \bmod n = 1$ or there is a square root of 1 somewhere in this sequence and this value must be -1
- If a non-trivial square root of 1 is found then n is composite.

Example

Let $n = 1729$ which is a Carmichael number. Then $n - 1 = 1728 = 2 \times 864 = 4 \times 432 = 8 \times 216 = 16 \times 108 = 32 \times 54 = 64 \times 27$. So $v = 6$ and $u = 27$.

1. Choose $a = 2$.
2. $b = 2^{27} \bmod 1729 = 645$.
3. Since $b \neq 1$ continue.
4.
   - Next $b = 645^2 \bmod n = 1065$
   - Next $b = 1065^2 \bmod n = 1$
   - Thus $b = -1$ will never occur.
5. The algorithm returns `composite`.

Note that 1065 is a non-trivial square root of 1 modulo 1729. Indeed $\gcd(1729, 1064) = 133$ is a factor of 1729 (see slide 20).

Generating large primes

The Miller–Rabin test can be 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 bases
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. To do so, start from step 1 if r fails in step 4. Both options are seen in practice.

Complexity theory in cryptology

Computational complexity provides a foundation for

- analysing the computational requirements of cryptographic algorithms
- studying the difficulty of breaking ciphers

We can consider two aspects of computational complexity:

- algorithm complexity - how long it takes to run a particular algorithm
- problem complexity - what is the best (known) algorithm to solve a particular problem

Algorithm complexity

- The computational complexity of an algorithm is measured by its time and space requirements as functions of the size of the input m
- A positive function $f(m)$ is typically expressed as an "order of magnitude" of the form $\mathcal{O}(g(m))$ where $g(m)$ is another positive function. This is called "big O" notation.
- We say
$$f(m) = \mathcal{O}(g(m))$$
if there exist constants $c > 0$ and m_0 such that $f(m) \leq c \cdot g(m)$ for $m \geq m_0$
- This means that g is, at least in the long run, an upper bound for f
- Speak of *asymptotic* behaviour

Big O notation, illustrated

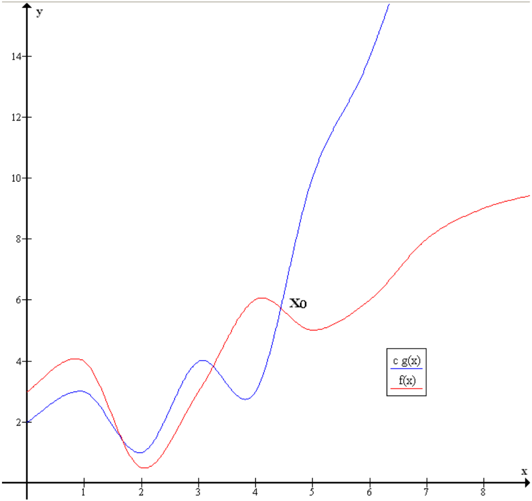


Image source: https://en.wikipedia.org/wiki/Big_O_notation

Polynomial and exponential functions

- A function $f(m)$ for which $f(m) = \mathcal{O}\left(m^t\right)$ for some positive integer t is said to be a *polynomial time function*.
- In cryptography we normally think of a polynomial time function as *efficient*.
- A function $f(m)$ for which $f(m) = \mathcal{O}\left(b^m\right)$ for some number $b > 1$ is said to be an *exponential time function*.
- In cryptography we normally think of a problem whose best solution is an exponential time function as *hard*
- 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. If $f(m) = 17m + 10$ then
$$f(m) = \mathcal{O}(m)$$
since $17m + 10 \leq 18m$ for $m \geq 10$
2. If $f(m)$ is a polynomial:
$$f(m) = a_0 + a_1 \cdot m + \ldots + a_t m^t$$
then
$$f(m) = \mathcal{O}\left(m^t\right)$$
3. If $f(m) = \mathcal{O}\left(m^t\right)$ then it is also true that $f(m) = \mathcal{O}\left(m^{t+1}\right)$

Problem complexity

A problem is classified according to the minimum time and space needed to solve the hardest instances of the problem on a deterministic computer

1. Multiplication of two $m \times m$ matrices, with fixed size entries, using the obvious algorithm is $\mathcal{O}\left(m^3\right)$
2. Sorting a set of integers into ascending order is $\mathcal{O}\left(m \cdot \log_2 m\right)$ with algorithms such as Quicksort

Two important problems

1. **Integer factorisation**: given an integer, find its prime factors
2. **Discrete logarithm problem** (with base g): given a prime p and an integer y with $0 < y < p$, find x such that

$$y = g^x \bmod p$$

- Best known algorithms to solve these problems on conventional computers are *sub-exponential:* slower than polynomial but faster than any exponential
- Fast algorithms exist using *quantum computers*

Integer factorisation

- Factorisation by trial division is an exponential time algorithm and is hopeless for numbers of a few hundred bits
- A number of special purpose methods exist, which apply if the integer to be factorised has special properties
- The best current general method is known as the *number field sieve*
- The number field sieve is a *sub-exponential* time algorithm

Some 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	2000
240	795	Dec 2019	900
250	829	Feb 2020	2700

- All records used number field sieve
- The records are for numbers with only two large factors, so-called RSA numbers

Discrete logarithm problem (DLP)

Let $\mathbb{G}$ be a cyclic group with generator g. The *discrete log problem* (DLP) in $\mathbb{G}$ is:

$$\boxed{\text{given } y \text{ in } \mathbb{G}\text{, find } x \text{ with } y = g^x}$$

- The best known algorithm for solving DLP in $\mathbb{Z}_p^*$ is a variant of the *number field sieve* (also used for factorisation) — a *subexponential* algorithm in the length of p
- The DLP can also be defined on elliptic curve groups (see later lecture)
- Best known DLP algorithms on elliptic curves are *exponential*

Example in $\mathbb{Z}_{19}^*$

$g^x \bmod p$	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

- Integers mod 19: $\mathbb{Z}_{19}^*$
- Generator $g = 2$
- When $y = g^x \bmod p$ then $\log_g y = x$
- For example, $\log_2 3 = 13$

Comparing brute-force key search, factorisation and discrete log in $\mathbb{Z}_p^*$

Symmetric key length	Length of $n = pq$	Length of prime p in $\mathbb{Z}_p^*$
80	1024	1024
112	2048	2048
128	3072	3072
192	7680	7680
256	15360	15360

- For example, brute force search of 128-bit keys for AES takes roughly same computational effort as factorisation of 3072-bit number with two factors of roughly equal size, or finding discrete logs in $\mathbb{Z}_p^*$ with a p of length 3072
- Source: NIST SP 800-57 Part 1 (2016)