رمزنگاری، امنیت اطلاعات و حریم خصوصی

ارائه: دکتر سیدعلی لاجوردی

بخش هفتم

number theory

(Computational)

Computational number theory

- Measure running times of algorithms in terms of the input lengths involved
  - For integer x, we have $\|x\| = O(\log x)$, $x = O(2^{\|x\|})$
- An algorithm taking input x and running in time O(x) is an exponential time algorithm
  - Efficient algorithms run in time $\text{poly}(\|x\|)$

Computational number theory

- Our goal: classify various problems as either “easy” or “hard”
  - I.e., polynomial-time algorithms known or not

- We will not focus on optimizations, although these are very important in practice
  - For “easy” problems: speed up cryptographic implementations
  - For “hard” problems: need to understand concrete hardness for concrete security

Representing integers

- Cryptography involves very large numbers!
- Standard (unsigned) integers (e.g., in C) are small, fixed length (e.g., 16 or 32 bits)
  - For crypto, need to work with integers that are much longer (e.g., 2000 bits)
- Solution: use an array
  - E.g., “bignum” = array of unsigned chars (bytes)
  - May be useful to also maintain a variable indicating the length of the array
  - Or, assume fixed length (which bounds the maximum size of a bignum)

Example: addition

- Add(bignum a, int L1, bignum b, int L2)
  - Use grade-school addition, using AddWithCarry byte-by-byte…

- Running time O(max{L1,L2}) = O(max{‖a‖,‖b‖})
  - If ‖a‖=‖b‖=n then O(n)
  - Is it possible to do better?
    - No – must read input (O(n)) and write output (O(n))

Example: multiplication

- What is the length of the result of a*b?
  - ‖ab‖=O(log ab)=O(log a + log b) =O(‖a‖+‖b‖)

- Use grade-school multiplication…
- Running time O(‖a‖·‖b‖)
  - If ‖a‖=‖b‖=n then O(n2)
- Is it possible to do better?
  - Surprisingly…yes!

Basic arithmetic operations

- As we have seen, addition / subtraction / multiplication can all be done efficiently
  - Using grade-school algorithms (or better)

- Division-with-remainder can also be done efficiently
  - Much harder to implement!

Modular arithmetic

- Notation:
  - [a mod N] is the remainder of a when divided by N
  - Note $0 \leq [a \bmod N] \leq N-1$

- $a = b \bmod N \Leftrightarrow [a \bmod N] = [b \bmod N]$

Modular arithmetic

- Note that
  [a+b mod N] = [[a mod N] + [b mod N] mod N]
  [a-b mod N] = [[a mod N] - [b mod N] mod N]
  and
  [ab mod N] = [[a mod N][b mod N] mod N]

- I.e., can reduce intermediate values
  - This can be used to speed up computations

Modular arithmetic

- Careful: not true for division!

- I.e., [9/3 mod 6] = [3 mod 6] = 3
  but [[9 mod 6]/[3 mod 6] mod 6] = 3/3 = 1
  - We will return to division later…

Modular arithmetic

- Modular reduction can be done efficiently
  - Use division-with-remainder

- Modular addition / subtraction / multiplication can all be done efficiently
  - We will return to division later

Exponentiation

- Compute ab ?
  - $\|ab\| = O(b \cdot \|a\|)$
  - Just writing down the answer takes exponential time!

- Instead, look at modular exponentiation
  - I.e., compute [ab mod N]
  - Size of the answer $\leq \|N\|$
  - How to do it?
    - Computing ab and then reducing modulo N will not work…

Modular exponentiation

- Consider the following algorithm:

```
exp(a, b, N) {
  // assume b ≥ 0
  ans = 1;
  for (i=1, i ≤ b; i++)
    ans = [ans * a mod N];
  return ans;
}
```

  - This runs in time O(b * poly(‖a‖, ‖N‖))
  - This is an exponential-time algorithm!

Efficient modular exponentiation

- Assume b = 2k for simplicity
  - The preceding algorithm roughly corresponds to computing a*a*a*…*a
  - Better: compute (((a2)2)2…)2
  - 2k multiplications vs. k multiplications
    - Note k = O(‖b‖)

Efficient exponentiation

- Consider the following algorithm:

```
exp(a, b, N) {
  // assume b ≥ 0
  x=a, t=1;
  while (b > 0) {
    if (b odd)
      t = [t * x mod N], b = b-1;
    x = [x2 mod N],  b = b/2; }
  return t; }
```

- Why does this work?
  - Invariant: answer is $[t \cdot x^b \bmod N]$
- Running time is polynomial in $\|a\|$, $\|b\|$, $\|N\|$

Primes and divisibility

- Assume you have encountered this before…
- Notation a | b
- If a | b then a is a divisor of b
- p > 1 is prime iff its only divisors are 1 and p
  - p is composite otherwise
- d = gcd(a, b) if both:
  - d | a and d | b
  - d is the largest integer with that property

Computing gcd?

- Can compute gcd(a, b) by factoring a and b and looking for common prime factors…
  - This is not (known to be) efficient!

- Use Euclidean algorithm to compute gcd(a, b)
  - One of the earliest nontrivial algorithms!

Euclidean algorithm

ALGORITHM B.7
The Euclidean algorithm GCD

Input: Integers a, b with $a \geq b > 0$
Output: The greatest common divisor of a and b

```
if b divides a
    return b
else return GCD(b, [a mod b])
```

See book for proof of correctness
and analysis of running time

Proposition

- Given a, b > 0, there exist integers X, Y such that Xa + Yb = gcd(a, b)
- Moreover, d=gcd(a, b) is the smallest positive integer that can be written this way
  - See book for proof

- Can use the extended Euclidean algorithm to compute X, Y
  - See book for details

Modular inverses

- b is invertible modulo N if there exists an integer a such that ab = 1 mod N
  - Let $[b^{-1} \bmod N]$ denote the unique such a that lies in the range {0, …, N-1}

- Division by b modulo N is only defined when b is invertible modulo N
  - Then $[c/b \bmod N]$ is defined to be $[c\, b^{-1} \bmod N]$

Cancellation

- The “expected” cancellation rule applies for invertible elements
- I.e., if ab = cb mod N and b is invertible modulo N, then a = c mod N
  - Proof: multiply both sides by b-1
- Note: this is not true if b is not invertible
  - E.g., 3*2 = 15*2 mod 8 but $3 \neq 15 \bmod 8$

Invertibility

- How to determine whether b is invertible modulo N?
- Thm: b invertible modulo N if gcd(b, N)=1
- To find the inverse, use extended Euclidean algorithm to find X, Y with Xb + YN = 1
  - Then [X mod N] is the inverse of b modulo N
- Conclusion: can efficiently test invertibility and compute inverses!

Group theory

Groups

- Introduce the notion of a group

- Provides a way to reason about objects that share the same mathematical structure
  - Not absolutely needed to understand crypto applications, but does make it conceptually easier

Groups

- An abelian group is a set G and a binary operation $\circ$ defined on G such that:
  - (Closure) For all $g, h \in G$, $g \circ h$ is in G
  - There is an identity $e \in G$ such that $e \circ g = g$ for $g \in G$
  - Every $g \in G$ has an inverse $h \in G$ such that $h \circ g = g \circ h = e$
  - (Associativity) For all $f, g, h \in G$, $f \circ (g \circ h) = (f \circ g) \circ h$
  - (Commutativity) For all $g, h \in G$, $g \circ h = h \circ g$

- The order of a finite group G is the number of elements in G

Examples and non-examples

- $\mathbb{Z}$ under addition
- $\mathbb{Z}$ under multiplication
- $\mathbb{R}$ under addition
- $\mathbb{R}$ under multiplication
- $\mathbb{R}\backslash\{0\}$ under multiplication
- {0,1}* under concatenation
- {0, 1}n under bitwise XOR
- 2 x 2 real, invertible matrices under mult.

Groups

- The group operation can be written additively or multiplicatively
  - I.e., instead of $g \circ h$, write $g+h$ or gh
  - Does not imply that the group operation has anything to do with (integer) addition or multiplication

- Identity denoted by 0 or 1, respectively
- Inverse of g denoted by –g or g-1, respectively
- Group exponentiation: $m \cdot a$ or a^m, respectively

Computations in groups

- When computing with groups, need to fix some representation of the group elements
  - Usually (but not always) some canonical representation
  - Usually want unique representation for each element
- Must be possible to efficiently identify elements in the group
- Must be possible to efficiently perform the group operation
  - $\Rightarrow$ Group exponentiation can be computed efficiently

Useful example

- $\mathbb{Z}_N$ = {0, …, N-1} under addition modulo N
  - Identity is 0
  - Inverse of a is [-a mod N]
  - Associativity, commutativity obvious
  - Order N

Example

- What happens if we consider multiplication modulo N?

- {0, …, N-1} is not a group under this operation!
  - 0 has no inverse
  - Even if we exclude 0, there is, e.g., no inverse of 2 modulo 4

Example

- Consider instead the invertible elements modulo N, under multiplication modulo N
- Define $\mathbb{Z}^*_N = \{0 < x < N : \gcd(x, N) = 1\}$
  - Closure
  - Identity is 1
  - Inverse of a is [a-1 mod N]
  - Associativity, commutativity obvious

$\phi(N)$

- $\phi(N)$ = the number of invertible elements modulo N
- $= |\{a \in \{1, \ldots, N-1\} : \gcd(a, N) = 1\}|$
- = The order of $\mathbb{Z}^*N$

Two special cases

- If p is prime, then 1, 2, 3, …, p-1 are all invertible modulo p
  - $\phi(p) = |\mathbb{Z}^*_p| = p-1$

- If N=pq for p, q distinct primes, then the invertible elements are the integers from 1 to N-1 that are not multiples of p or q
  - $\phi(N) = |\mathbb{Z}^*N| = ?$

Fermat's little theorem

- Let G be a finite group of order m. Then for any $g \in G$,

it holds that $g^m = 1$

  - Proof (abelian case)

Examples

- In $\mathbb{Z}_N$:
  - For all a $\in \mathbb{Z}$N, we have N · a = 0 mod N
  - (Note that N is not a group element!)

- In $\mathbb{Z}^*_N$:
  - For all a $\in \mathbb{Z}$*N, we have aϕ(N) = 1 mod N
  - p prime: for all a $\in \mathbb{Z}$*p, we have ap-1 = 1 mod p

Corollary

- Let G be a finite group of order m. Then for $g \in G$ and integer x, it holds that $g^x = g^{[x \bmod m]}$
  - Proof: Let $x = qm+r$. Then $g^x = g^{qm+r} = (g^m)^q g^r = g^r$

- This can be used for efficient computation…
  - …reduce the exponent modulo the order of the group before computing the exponentiation

Corollary

- Let G be a finite group of order m
- For any positive integer e, define $f_e(g)=g^e$
- Thm: If gcd(e,m)=1, then f_e is a permutation of G. Moreover, if $d = e^{-1} \bmod m$ then f_d is the inverse of f_e
  - Proof: The first part follows from the second.
    And $f_d(f_e(g)) = (g^e)^d = g^{ed} = g^{[ed \bmod m]} = g^1 = g$

Corollary

- Let N=pq for p, q distinct primes
  - So $|\mathbb{Z}^*_N| = \phi(N) = (p-1)(q-1)$
- If $\gcd(e, \phi(N))=1$, then $f_e(x) = [x^e \bmod N]$ is a permutation
  - In that case, let $[y^{1/e} \bmod N]$ be the *unique* $x \in \mathbb{Z}^*_N$ such that $x^e = y \bmod N$
- Moreover, if $d = e^{-1} \bmod \phi(N)$ then f_d is the inverse of f_e
  - So for any x we have $(x^e)^d = x \bmod N$
  - I.e., $[x^{1/e} \bmod N] = [x^d \bmod N]$!

Example

- Consider N=15
  - Look at table for $f_3(x)$
- N = 33
  - Take e=3, d=7, so 3rd root of 2 is…?
  - e=2; squaring is not a permutation…

Hard problems

- So far, we have only discussed number-theoretic problems that can be solved in polynomial time
  - E.g., addition, multiplication, modular arithmetic, exponentiation, gcd, …
- Some problems are (conjectured to be) *hard*

Factoring

- Multiplying two numbers is easy; factoring a number is hard
  - Given x, y, easy to compute $x \cdot y$
  - Given N, hard (in general) to find x, $y > 1$ such that $x \cdot y = N$

- Compare:
  - Multiply 10101023 and 29100257
  - Find the factors of 293942365262911

Factoring

- It’s not hard to factor *random* numbers
  - 50% of the time, random number is even
  - 1/3 of the time, random number is divisible by 3…
- The hardest numbers to factor are those that are the product of two, equal-length *primes*

Generating primes

- To generate a (random) n-bit prime do:
  - Choose uniform n-bit integer p
  - If p is prime, output it; else, repeat

- Is this efficient?

Generating primes

- For this to be efficient, need two things:
  - Primes should be sufficiently *dense*
    - I.e., probability that a uniform n-bit integer is prime should be sufficiently large
  - Need an efficient way to test primality

Distribution of primes

- Known that primes are sufficiently dense
  - Pr[n-bit number is prime] > 1/3n
  - Probability that a uniform n-bit integer is prime is inverse polynomial
    - If we choose poly(n) uniform n-bit integers, we find a prime with all but negligible probability

Testing primality

- In the ‘70s, *probabilistic* poly-time algorithms for testing primality were developed
  - These are quite efficient
- For decades, a classic example of a problem with an efficient *randomized* algorithm but no known efficient *deterministic* algorithm
- 2002: efficient deterministic algorithm found
  - By undergraduates!
- In practice, randomized algorithms still used

Generating primes

- Summarizing: there are efficient (randomized) algorithms for generating (random) primes
  - These algorithms may fail, but only with negligible probability

The RSA problem

- The factoring problem is not *directly* useful for cryptography

- Instead, introduce a problem related to factoring: the *RSA problem*

The RSA problem

- For the next few slides, N=pq with p and q distinct, odd primes

- $\mathbb{Z}^*_N$ = invertible elements under multiplication modulo N
  - The order of $\mathbb{Z}^*_N$ is $\phi(N) = (p-1)\cdot(q-1)$
- Note:
  - $\phi(N)$ is *easy* to compute if p, q are known
  - $\phi(N)$ is *hard* to compute if p, q are not known
    - In fact, can be shown equivalent to factoring N

The RSA problem

- N defines the group $\mathbb{Z}^*_N$ of order $\phi(N)$
- Fix e with $\gcd(e, \phi(N)) = 1$
  - Raising to the e-th power is a permutation of $\mathbb{Z}^*_N$
- If $ed = 1 \bmod \phi(N)$, raising to the d-th power is the *inverse* of raising to the e-th power
  - I.e., $(x^e)^d = x \bmod N$, $(x^d)^e = x \bmod N$
  - x^d is the *e-th root of x modulo N*

Example

- N=33, e=3

x	x^3 mod 33	x	x^3 mod 33
1	1	17	29
2	8	19	28
4	31	20	14
5	26	23	23
7	13	25	16
8	17	26	20
10	10	28	7
13	19	29	2
14	5	31	25
16	4	32	32

Computing e-th roots

- If p, q are known:
  - $\Rightarrow \phi(N)$ can be computed
  - $\Rightarrow d = e^{-1} \bmod \phi(N)$ can be computed
  - $\Rightarrow$ possible to compute e-th roots modulo N

- If p, q are *not* known:
  - $\Rightarrow$ computing $\phi(N)$ is as hard as factoring N
  - $\Rightarrow$ computing d is as hard as factoring N
  - $\Rightarrow$ appears hard to compute e-th roots modulo N

The RSA problem

- Informally: given N, e, and uniform element $y \in \mathbb{Z}^*_N$, compute the e-th root of y

- RSA assumption: this is a hard problem!

The RSA assumption (informally)

- "Computing e-th roots modulo N is hard"
  - When the factorization of N is *unknown*

- Careful: it is not hard to compute e-th roots of *all* $y \in \mathbb{Z}^*_N$
  - In particular, it is easy when y is an e-th power (over the integers, with no modular reduction)
  - Hard for a *randomly chosen* y

The RSA assumption (formal)

- Let GenRSA be an algorithm that on input 1^n, outputs (N, e, d) with
  - N=pq a product of two distinct n-bit primes
  - $ed = 1 \bmod \phi(N)$

Implementing GenRSA

- One way to implement GenRSA:
  - Generate uniform n-bit primes p, q
  - Set N := pq
  - Compute $\phi(N) := (p-1)(q-1)$
  - Choose arbitrary e with $\gcd(e, \phi(N))=1$
  - Compute $d := [e^{-1} \bmod \phi(N)]$
  - Output (N, e, d)

- Choice of e?
  - Not believed to affect hardness of RSA problem
  - $e = 3$ or $e = 2^{16} + 1$ for efficient exponentiation

The RSA assumption (formal)

- Fix GenRSA and some algorithm A

- Experiment $\text{RSA-inv}_{A,\ \text{GenRSA}}(n)$:
  - Compute $(N, e, d) \leftarrow \text{GenRSA}(1^n)$
  - Choose uniform $y \in \mathbb{Z}^*_N$
  - Run A(N, e, y) to get x
  - Experiment evaluates to 1 iff $x^e = y \bmod N$

The RSA assumption (formal)

- The *RSA problem is hard relative to GenRSA* if for all PPT algorithms A,

$$\Pr[\text{RSA-inv}_{A,\ \text{GenRSA}}(n) = 1] < \text{negl}(n)$$

RSA and factoring

- If factoring moduli output by GenRSA is easy, then the RSA problem is easy relative to GenRSA
  - Factoring is easy $\Rightarrow$ RSA problem is easy

- Hardness of the RSA problem is *not known to be implied* by hardness of factoring
  - Possible factoring is hard but RSA problem is easy
  - Possible both are hard but RSA problem is “easier”
  - Currently, RSA is believed to be as hard as factoring

Cyclic groups

Cyclic groups

- Let G be a finite group of order m (written multiplicatively)
- Let g be some element of G
- Consider the set $\langle g \rangle = \{g^0, g^1, \ldots\}$
  - We know $g^m = 1 = g^0$, so the set has ≤m elements
  - If the set has m elements, then it is all of G !
    - In this case, we say g is a *generator* of G
    - If G has a generator, we say G is *cyclic*
    - Not every element of a cyclic group will be a generator
    - A cyclic group can have more than one generator

Examples

- $\mathbb{Z}_N$
  - Cyclic; 1 is always a generator: {0, 1, 2, …, N-1}

- $\mathbb{Z}_8$
  - Is 3 a generator?
    {0, 3, 6, 1, 4, 7, 2, 5} – yes!
  - Is 2 a generator?
    {0, 2, 4, 6} – no!

Example

- $\mathbb{Z}^*_{11}$
  - Is 3 a generator?
    {1, 3, 9, 5, 4} – no!
  - Is 2 a generator?
    {1, 2, 4, 8, 5, 10, 9, 7, 3, 6} – yes!
  - Is 8 a generator?
    {1, 8, 9, 6, 4, 10, 3, 2, 5, 7} – yes!
    Note that elements are in a different order …

Example

- $\mathbb{Z}^*_{13}$
  - <2> = {1, 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7},
    so 2 is a generator
  - <8> = {1, 8, 12, 5},
    so 8 is not a generator

Important examples

- Theorem: Any group of *prime order* is cyclic, and every non-identity element is a generator

- Theorem: If p is prime, then $\mathbb{Z}^*_p$ is cyclic
  - Note: the order is p-1, which is not prime for p > 3

Uniform sampling

- Given cyclic group G of order q along with generator g, easy to sample a uniform $h \in G$:
  - Choose uniform $x \in \{0, \ldots, q-1\}$; set $h := g^x$

Discrete-logarithm (dlog) problem

- Fix cyclic group G of order q, and generator g

- We know that $\{g^0, g^1, \ldots, g^{q-1}\} = G$
  - For every $h \in G$, there is a <u>unique</u> $x \in \mathbb{Z}_q$ s.t. $g^x = h$
  - Define $\log_g h$ to be this x – *the discrete logarithm of h with respect to g* (in the group G)

Examples

- In $\mathbb{Z}^*_{11}$
  - What is $\log_2 9$?
    - $\langle 2 \rangle = \{1, 2, 4, 8, 5, 10, 9, 7, 3, 6\}$, so $\log_2 9 = 6$
  - What is $\log_8 9$?
    - $\langle 8 \rangle = \{1, 8, 9, 6, 4, 10, 3, 2, 5, 7\}$, so $\log_8 9 = 2$

- In $\mathbb{Z}^*_{13}$
  - What is $\log_2 9$?
    - $\langle 2 \rangle = \{1, 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7\}$, so $\log_2 9 = 8$

Discrete-logarithm problem (informal)

- dlog problem in G: Given generator g and element h, compute $\log_g h$

- dlog assumption in G: Solving the discrete log problem in G is hard
  - Careful: not hard to compute $\log_g h$ for *all* h, but should be hard for a uniform h

Example

- In $\mathbb{Z}^*_{3092091139}$
  - What is $\log_2 1656755742$?

Discrete-logarithm problem

- Let $\mathcal{G}$ be a group-generation algorithm
  - On input 1^n, outputs a (description of a) cyclic group G, its order q (with $\|q\| \geq n$), and a generator g

- For algorithm A, define exp't $\text{Dlog}_{A,\mathcal{G}}(n)$:
  - Compute $(G, q, g) \leftarrow \mathcal{G}(1^n)$
  - Choose uniform $h \in G$
  - Run A(G, q, g, h) to get x
  - Experiment evaluates to 1 if $g^x = h$

- Note: easy to check correctness of the answer

Discrete-logarithm problem

- The *discrete-logarithm problem is hard relative to* $\mathcal{G}$ if for all PPT algorithms A,

$$\Pr[\mathrm{Dlog}_{A,\mathcal{G}}(n) = 1] \leq \mathrm{negl}(n)$$

Diffie-Hellman problems

- Fix cyclic group G and generator g
- Define $DH_g(h_1, h_2) = DH_g(g^x, g^y) = g^{xy}$
  $= (h_1)^y = (h_2)^x$

Diffie-Hellman assumptions

- *Computational* Diffie-Hellman (CDH) problem:
  - Given g, h_1, h_2, compute $DH_g(h_1, h_2)$
- *Decisional* Diffie-Hellman (DDH) problem:
  - Given g, h_1, h_2, distinguish $DH_g(h_1, h_2)$ from a uniform element of G

Example

- In $\mathbb{Z}^*_{11}$
  - <2> = {1, 2, 4, 8, 5, 10, 9, 7, 3, 6}
  - So $DH_2(7, 5)$ = ?
- In $\mathbb{Z}^*_{3092091139}$
  - What is $DH_2(1656755742, 938640663)$?
  - Is 1994993011 the answer, or is that just a uniform element of $\mathbb{Z}^*_{3092091139}$?

DDH problem

- Let $\mathcal{G}$ be a group-generation algorithm
  - On input 1^n, outputs a cyclic group G, its order q (with $\|q\|=n$), and a generator g

- The *DDH problem is hard relative to* $\mathcal{G}$ if for all PPT algorithms A:

  $| \Pr[A(G, q, g, g^x, g^y, g^{xy})=1] - \Pr[A(G, q, g, g^x, g^y, g^z)=1] | \leq \varepsilon(n)$

Relating the Diffie-Hellman problems

- Relative to $\mathcal{G}$:
  - If the discrete-logarithm problem is easy, so is the CDH problem
    - CDH problem is potentially easier than dlog problem
    - I.e., CDH assumption is *stronger* than dlog assumption
  - If the CDH problem is easy, so is the DDH problem
    - DDH problem is potentially easier than CDH problem
    - I.e., DDH assumption is *stronger* than CDH assumption

Group selection

- The discrete logarithm problem is not hard in all groups!
  - For example, it is easy in $\mathbb{Z}_N$ (for any N, and for any generator)

- Nevertheless, there are certain groups where the problem is believed to be hard
  - All cyclic groups of the same order are isomorphic, but the group representation matters!

Group selection

- For cryptographic applications, best to use *prime-order* groups
  - The dlog problem becomes easier if the order of the group has small prime factors
  - Prime-order groups have several nice features
    - E.g., every element except the identity is a generator
  - Avoids some trivial DDH algorithms

- Two common choices of groups for cryptography…

Group selection: choice 1

- Prime-order subgroup of $\mathbb{Z}^*_p$, p prime
  - E.g., let p = kq + 1 for p, q prime
    - So $\mathbb{Z}^*_p$ has order p-1 = kq
  - Take the subgroup of k^{th} powers, i.e.,
    $$G = \{ [x^k \bmod p] \mid x \in \mathbb{Z}^*_p \} \subset \mathbb{Z}^*_p$$
    - G is a group
    - Can show that it has order (p-1)/k = q
    - Since q is prime, G must be cyclic
- Generalizations based on finite fields also

Group selection: choice 2

- Prime-order subgroup of an *elliptic-curve* group
  - See book for the basic details…

- These have the advantage of giving stronger security with smaller parameters (for reasons to be explained shortly)

Group selection

- We will describe cryptographic schemes in an “abstract” cyclic group
  - Can ignore the details of the underlying group in the analysis
  - Can instantiate with any (appropriate) group in an implementation

Concrete parameters?

- We have discussed two classes of cryptographic assumptions
  - Factoring-based (factoring, RSA assumptions)
  - dlog-based (dlog, CDH, and DDH assumptions)
    - In two classes of groups
- All these problems are believed to be “hard,” i.e., to have no polynomial-time algorithms
  - But how hard are they, concretely?

Disclaimer

- The goal here is just to give an idea as to how parameters are calculated, and what relevant parameters are

- In practice, other important considerations come into play

Security

- Recall: For symmetric-key algorithms…
  - Block cipher with n-bit key ≈ security against 2^n-time attacks = n-bit security
  - Hash function with 2n-bit output ≈ security against 2^n-time attacks = n-bit security
- Factoring a modulus $N \approx 2^n$ (i.e., length n) using exhaustive search takes $\approx 2^{n/2}$ time
- Computing discrete logarithms in a group of order $\approx 2^n$ using exhaustive search takes $\approx 2^n$ time
  - Are these the best possible algorithms?

Algorithms for factoring

- There exist algorithms factoring an integer N that run in *much less* than $2^{\|N\|/2}$ time

- Best known algorithm (asymptotically): *general number field sieve*
  - Running time (heuristic): $2^{O(\|N\|^{1/3} \log^{2/3} \|N\|)}$
  - Makes a huge difference in practice!
  - Exact constant term is also important!

Algorithms for dlog

- Two classes of algorithms:
  - Ones that work for *arbitrary* (“generic”) groups
  - Ones that target *specific* groups
    - Recall that in some groups the problem is not even hard

Algorithms for dlog

- Best *generic* dlog algorithms in a group of order $\approx 2^n$ take time $\approx 2^{n/2}$
  - This is known to be optimal (for generic algorithms)

Algorithms for dlog

- Best known algorithm for (subgroups of) $\mathbb{Z}^*_p$: *number field sieve*
  - Running time (heuristic): $2^{O(\|p\|^{1/3} \log^{2/3} \|p\|)}$

- For (appropriately chosen) elliptic-curve groups, nothing better than generic algorithms is known!
  - This is why elliptic-curve groups can allow for more-efficient cryptography

Choosing parameters

- As recommended by NIST (112-bit security):
  - Factoring: 2048-bit modulus
  - Dlog, order-q subgroup of $\mathbb{Z}^*_p$: ‖q‖=224, ‖p‖=2048
    - Addresses both generic and specific algorithms
  - Dlog, elliptic-curve group of order q: ‖q‖=224 bits

- Much longer than for symmetric-key algorithms!
  - Explains in part why public-key crypto is less efficient than symmetric-key crypto