Number Theoretical Cryptanalysis

Gerardo Pelosi

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)
Politecnico di Milano

gerardo.pelosi - at - polimi.it

Overview

Lesson contents

- Primality Testing
- Number theoretical cryptanalysis
  - Factoring Algorithms
  - Discrete Log Algorithms

Primality Testing

Primality Testing

Prime Number Theorem (PNT)

Let $\pi(x)$ be the prime-counting function that gives the number of primes less than or equal to x, for any real number x. The limit of the quotient of the two functions $\pi(x)$ and $\frac{x}{\ln(x)}$ as x approaches infinity is 1:

$$\pi(x) \sim \frac{x}{\ln(x)}, \quad x \to \infty$$

Equivalently, the value of the n-th prime number p_n is approximately equal to $n \cdot \ln(n)$, with the approximation error approaching 0 as n approaches infinity.

The probability a generic number $x = p$ is a prime is about $\frac{\pi(x)}{x} = \frac{1}{\ln(x)}$

F.i., picking a random integer encoded with 512 bits, the probability of being prime is: $\approx \frac{1}{\ln(2^{512})} = \frac{1}{355}$ which means that: picking $355/2 = 177$ random odd integers (encoded with 512), one of them is likely to be prime.

Primality Testing

It is enough practical to look for large primes picking candidates at random, as long as we can test primality or composability in an efficient manner!

We know that in $(\mathbb{Z}_p^*, \cdot)$ the following relation is trivially verified:

$$a^{p-1} \equiv 1 \bmod p, \quad \text{for any element } a \in \mathbb{Z}_p^*$$

Composability Criterion

It can be shown that if n is composite then

$$a^{n-1} \not\equiv_n 1 \quad \text{with } 1 < a < n$$

with probability $\geq \frac{1}{2}$, over all possible values of the base a.
The base a is called "Fermat witness" for the compositeness of n

if n is a composite number and for a given value of a it is true that $a^{n-1} \equiv_n 1$ then n is called "Fermat pseudoprime to base a", while a is called "Fermat liar"

Primality Testing: Fermat test

Fermat's primality test

Input: integer to be tested n
probability parameter: $1 \leq k < n$

```
1 begin
2     for i from 0 to k − 1 do
3         Pick a ∈ {2, ..., n − 1}, s.t. gcd(a, n)=1
4         if a^{n−1} ≢ 1 mod n then
5             return "Composite"
6     return "NON composite: with probability ≥ 1 − 1/2^k"
```

- Using algorithms for modular exponentiation, the running time of the Fermat's algorithm is $\mathcal{O}(k \log_2^3 n)$ bit operations, $k \in \{1, \ldots, n-1\}$

Primality Testing: Fermat test

- There are infinitely many composite integers n (Fermat pseudo-primes aka "Carmichael" numbers) that may behave like prime numbers with respect to the previous test
  - A Carmichael number is defined as a composite integer n s.t. $a^{n-1} \equiv_n 1$ for all the choices of a s.t. $\gcd(a, n) = 1$
- While Carmichael numbers are substantially rarer than prime numbers, there are enough of them to justify the choice of a more refined criterium known as **Miller-Rabin test for composability**

Primality Testing: Miller-Rabin test

Lemma

if p is prime then the only two solutions of the equation:

$$x^2 = 1 \bmod p \;\Leftrightarrow\; (x+1)(x-1) = 0 \bmod p$$

are: $x = \pm 1 \bmod p$

Lemma

Given a prime $p>2$, is always possible to write: $p-1=\mathbf{d}\,2^{\mathbf{s}}$, $\mathbf{d}$ odd, $s \geq 1$
Therefore, for each $a \in \mathbb{Z}_p^*$: $a^{p-1}=a^{d\,2^s} \equiv_p 1$ we have either

$$a^d \equiv \pm 1 \bmod p$$

or

$$a^{d\,2^r} \equiv -1 \bmod p \qquad \text{for at least one value } r \text{ s.t. } 1 \leq r \leq s-1$$

Primality Testing: Miller-Rabin test

Miller–Rabin Test

The test is based on the contrapositive clause of the above Lemma
Given an odd integer n, if we can find at least one $a \in \{2, \ldots, n-1\}$ s.t.

$$\begin{cases} a^d & \not\equiv & \pm 1 \bmod n \\ & \text{and} & \\ a^{d\,2^r} & \not\equiv & -1 \bmod n \quad \text{for all } 1 \le r \le s-1 \end{cases}$$

then n is composite (i.e., not a prime number!)

The more bases, a, we test, the better the accuracy of the test. It can be shown that for any odd composite n, at least $\frac{3}{4}$ of the bases $2<a<n$ are witnesses for the composedness of n, therefore:

- The prob. of erroneously declaring n non-composite (i.e., prime) running the test one time is $\frac{1}{4}$; running the test two times: $\frac{1}{4^2}$, ...
- k-rounds of the MR test declares n "prime" (when it is actually prime) with prob. $\geq 1-4^{-k}$

Primality Testing: Miller-Rabin test

Algorithm

Input: odd integer to be tested for primality: n
accuracy parameter: $1 \le k < n$
Output: "Composite" if n is composite,
otherwise "Prime: with prob. $\geq 1 - \frac{1}{4^k}$"

```
1  begin
2      Consider n − 1 = d 2^s, with d odd, s ≥ 1
3      for i from 0 to k − 1 do
4          Pick a ∈ {2, ..., n − 1}
5          x ← a^d mod n
6          if x ≠ ±1 mod n then
7              r ← 1
8              while r < s and x ≠ −1 mod n do
9                  x ← x^2 mod n; r ← r + 1
10             if r = s then
11                 return "Composite"
12     return "Prime: with probability ≥ 1 − 1/4^k"
```


Primality Testing: Miller-Rabin test

- Given an odd integer n, if we keep k=30 and the Miller-Rabin test declares n a prime, the error probability is at most $4^{-30} \approx 10^{-18}$, which is far smaller than the probability of an Hardware fault in every digital circuit!
- The Miller-Rabin Test is a probabilistic procedure with computational complexity $\mathcal{O}(k \log_2^3 n)$ bit operations

The best "deterministic primality test" currently known is called *AKS Algorithm*. It has a computational complexity, for testing a generic integer n, equal to $\mathcal{O}((\log_2 n)^{7.5})$ arithmetic operations

[ref. M. Agrawal, N. Kayal, N. Saxena *PRIMES in P*,
Annals of Mathematics, Vol. 160, 2004. pp.781-793]

Factoring Algorithms

Factoring Algorithms

Factoring methods are usually divided into elementary methods such as

- Trivial division
- Fermat's factorization algorithm
- Pollard's $P-1$ method
- Pollard's rho method

and sub-exponential factoring methods with running time $\mathcal{O}(L_n(\alpha, \beta))$ where $0 < \alpha < 1,\ \beta \geq 1$

$$L_n(\alpha, \beta) = \mathcal{O}(e^{(\beta+o(1))(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}})$$

we'll see only the "linear sieve" strategy ('1978) to show the rough method that is carefully refined by more powerful algorithms up to the General Number Field Sieve (GFNS) with running time: $\mathcal{O}\left(L_n(1/3, \sqrt[3]{64/9})\right)$

Factoring Algorithms: Fermat's method

Fermat's factorization method, named after Pierre de Fermat, is based on the representation of an **odd** integer n as the difference of two squares:

$$n = x^2 - y^2 = (x + y)(x - y)$$

if neither factor equals one, it is a proper factorization of n

Each odd number has such a representation. Indeed, if $n = c\,d$ is a factorization of n, then

$$n = \left(\frac{1}{2}(c + d)\right)^2 - \left(\frac{1}{2}(c - d)\right)^2$$

- since n is odd, then c and d are also odd, so those halves are integers
- a multiple of four is also a difference of squares, with c and d even

Factoring Algorithms: Fermat's method

- The factoring method tries various values of x hoping to find one that yields a square for $x^2 - n$:

```
Input: a composite integer n
Output: two non-trivial factors of n
1 begin
2     x ← ⌈√n⌉
3     y_squared ← x² − n
4     while y_squared is not a square do
5         x ← x + 1
6         y_squared ← y_squared + 2x − 1 // y_squared = x² − n
7     return x − √y_squared; x + √y_squared
```

- The "squaring" test can be done observing that there are only 22 possible values for the last two hexadecimal digits of a perfect square:

  00, e1, e4, 25, o6, and e9

  where 'e' stands for an even number and 'o' for an odd number.
- If factors are not near $\sqrt{n}$, it works slowly: $\mathcal{O}(\sqrt{n})=\mathcal{O}(2^{\frac{\log_2(n)}{2}})$ iterations (Obs.: $\mathcal{O}(2\log_2 n)$ bit op.s is the complexity of a divisibility test (Euclidean alg.) – Trivial division factoring: $\mathcal{O}(2\sqrt{n}\log_2 n)$)

Factoring Algorithms: Pollard's ρ method

The heart of Pollard's rho method is the following:

Basic Observation

1. Let λ be a factor of n, if we can find two other integers (a, b) s.t. $a \equiv_\lambda b$ then $\lambda | a-b$, i.e. λ=gcd$(a-b, n)$
2. two random numbers a and b are congruent modulo λ with probability 0.5 after $\approx \sqrt{\lambda}$ numbers have been randomly gathered (birthday paradox)

Therefore a basic strategy would be:

- Pick $a, b \in \{1, \ldots, n-1\}$ at random and check if gcd$(a - b, n) \neq 1$
- Continue in this way until you find a factor λ=gcd$(a-b, n) > 1$

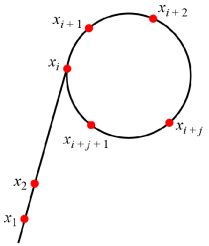
When picking random numbers, it is useful to have a criterion to avoid repeating the same gcd computation
Fortunately, there is method to structure the way in which you pick "random" numbers

Factoring Algorithms: Pollard's ρ method

- Let $f : \{1,\ldots,n{-}1\} \mapsto \{1,\ldots,n{-}1\}$ be an arbitrary function, f.i. $f(x){=}x^2{+}1 \bmod n$, with $x{\in}\{1,\ldots,n{-}1\}$
- We use f to generate a pseudo-random sequence $x_1, x_2{=}f(x_1), x_3{=}f(x_2), \ldots, x_{i+1}{=}f(x_i), \ldots$
- Numbers in this pseudo-random sequence are employed to test if $\gcd(x_i{-}x_j, n){\neq}1$; thus finding a factor of n

Example: Let us consider $n{=}15$ and $f(x){\equiv_n}x^2{+}1$ then

x_1	x_2	x_3	x_4	x_5	x_6	x_7	$\cdots$
1	2	5	11	2	5	11	$\cdots$



Thus,
$\lambda{=}\gcd(x_3{-}x_2, n){=}\gcd(3,15){=}3\ (\neq 1)$

Since $\{1,\ldots,n{-}1\}$ is finite, the sequence $x_1, x_2, x_3, \ldots$ must eventually cycle

Factoring Algorithms: Pollard's ρ method

We need an efficient way to determine if the sequence $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_t}$ from $x_{i+1}{\equiv_n}f(x_i)$, is cycling (...to stop testing gcds in case it occurs) $\Rightarrow$ *Floyd's cycle-finding algorithm*

- The algorithm keeps two pointers named *tortoise* and *hare*. The *tortoise* moves through each element one step at a time; the *hare* starts from the same point x_0, but moves twice as fast as the *tortoise*
- Let x_i mark the tortoise path, and y_i the hare path. Floyd's algorithm increases i by one, at each step, thus moving the *tortoise* one step forward $x_{i+1}{\leftarrow}f(x_i)$ and the *hare* two steps forward $y_{i+1}{\leftarrow}f(f(y_i))$ in the sequence, and then compares the pointed values
- The hare will enter the loop first but, eventually, both will be in the loop. In the loop the hare closes up to the tortoise by one step at each iteration. Since the cycle has finite length, they will meet (i.e., $x_i{\equiv_n}y_i$)... and we will detect the that there is cycle.

Factoring Algorithms: Pollard's ρ method

- Basic Observation + Floyd's cycle finding
- assume $f(x) = x^2 + 1 \bmod n$ be the random sequence generator
- when $\gcd(\mathrm{abs}(x - y), n) = \gcd(0, n){=}n$ a cycle has been detected – it is useless keep computing the tortoise and hare paths!!!

Pollard's ρ Algorithm

Input: a composite integer n
Output: a non-trivial factor λ of n

```
1  begin
2      x ← 2, y ← 2 // x ← x_1 = 2,  y = x_1 ∈{0,...,n−1}
3      λ ← 1
4      while λ == 1 do
5          x ← x^2 + 1 mod n // x_{i+1} ← f(x_i)
6          y ← (y^2+1 mod n)^2+1 mod n // y_{i+1}←f(f(x_i)), y_i=x_{2i}
7          λ ← gcd(abs(x − y), n)
8          if λ == n then
9              return "Failure (cycle detected)" // restart w/ x, y≠2
10     return λ
```


Factoring Algorithms: Pollard's ρ method

Computational Complexity

It can be shown that the Pollard's ρ method achieves a time complexity of $\mathcal{O}(\sqrt[4]{n}\log_2^2 n)$ bit operations, thus asymptotically better than both Fermat's and trivial division method

Example

Consider $n = 455459$ and apply the Pollard ρ algorithm for factoring it.
$\lambda = \gcd(\mathrm{abs}(x - y), n)$

i	$x \leftarrow x_i$	$y \leftarrow x_{2i}$	λ
0	2	2	
1	5	26	1
2	26	2871	1
3	677	179685	1
4	2871	155260	1
5	44380	416250	1

i	$x \leftarrow x_i$	$y \leftarrow x_{2i}$	λ
6	179685	43670	1
7	121634	164403	1
8	155260	247944	1
9	44567	68343	743

We can write $n = 743 \cdot 613$

Factoring Algorithms: Pollard's $P-1$ Method

Pollard's $P-1$ Method is particularly fast to eventually find the "small" prime factors of a composite number

Smooth Numbers

Let B be an integer. An integer n is called B-smooth if every prime factor p of n is smaller than B.

F.i.: $n = 2^{78} \cdot 3^{89} \cdot 11^3$ is 12-smooth.

- Basically, a smooth number can be quickly factored through trivial divisions

Power Smooth Numbers

Let B be an integer. An integer n is called B-power smooth if every prime power of n is smaller than B

F.i.: $n = 2^5 \cdot 3^3 \cdot 11$ is 33-power smooth.

Factoring Algorithms: Pollard's $P-1$ Method

- Suppose we wish to factor a composite integer defined as $n = p \cdot q$.
- Suppose that we know (by some pure guess) a number **B** s.t. $p-1$ is B-power smooth, while $q-1$ is not!
- We know that $p-1$ divides $(B! = B\,(B-1)\,(B-2)\cdots 2)$ because $(B!)$ will certainly have the prime powers of $p-1$ as factors.
  - therefore, if $\mathbf{a} = \left(\mathbf{2}^{(\mathbf{B!})} \bmod \mathbf{n}\right)$ we have: $a \equiv_p 1$ and $a \not\equiv_q 1$
  - we know that $p \mid a-1$, $q \nmid a-1$ and then we can recover the factor p as: $\mathbf{p} = \gcd(\mathbf{a}-\mathbf{1}, \mathbf{n})$ **if the result is** $\mathbf{1} < \gcd < \mathbf{n}$ (**otherwise we have to guess another value for B**)
- a possible choice for B would be $\log_2^c(n)$ for some c (e.g. $0 < c \leq 6$)
- the computational complexity of the method is $\mathcal{O}(B \log_2(B) \log_2^2(n))$

To hinder Pollard's $P-1$ method, a common recommendation for RSA primes is to pick them so to satisfy: $p{=}2{\cdot}p_1{+}1$, $q{=}2{\cdot}q_1{+}1$, with p_1 and q_1 also primes. However, for large enough primes ($> 2^{512}$) the smoothness bound B is usually so high that the attack is not applicable

Factoring Algorithms: Number Field Sieve - Linear Sieve

(... this is not included in the written exam)

The problem of finding the difference of two squares modulo a integer number n is a key concept of modern factoring algorithms

Given a composite integer $n = p \cdot q$:

- find two other numbers x, y with about the same size of n s.t. $x^2 \equiv y^2 \bmod n \Leftrightarrow (x-y)(x+y) \equiv 0 \bmod (p \cdot q)$, one of the following holds:
  1. $p \mid x-y$ and $q \mid x+y$, i.e. for l—(q − 1)
  2. $p \mid x+y$ and $q \mid x-y$
  3. $p, q \mid (x-y)$ and $p, q \nmid (x+y)$
  4. $p, q \nmid (x-y)$ and $p, q \mid (x+y)$
- compute $d = \gcd(x-y, n)$

Possible values of d are: $1, n, p, q$ with equal probability thus, the chances to factor n in this way are $\frac{1}{2}$

How do we find x, y s.t. $x^2 \equiv y^2 \bmod n$? (i.e., $x^2{-}y^2{=}n\lambda$, for some λ ?)

Factoring Algorithms: Number Field Sieve - Linear Sieve

Let n be the number to be factored and $F = \{p : p \leq B\}$ be a set of "small" prime numbers (randomly chosen), which form the so called "factorbase"
A number which factorizes with all its factors in F is therefore B-smooth

The idea of the linear sieve is: (**Sieving Step**)

- find two integers a and λ such that $b = a + n\lambda$ is B-smooth (i.e. its prime factors are in F)
- Then, we would expect the number a will also be B-smooth:

$$a = \prod_{p \in F} p^{a_p} \quad \text{...verified by trivial division}$$

and

$$b = a + n\lambda = \prod_{p \in F} p^{b_p} \quad \text{...verified by trivial division}$$

- We get a 1st relation in $\mathbb{Z}_n$, say $\texttt{rel}_1$: $\left(\prod_{p \in F} p^{a_p}\right) \equiv \left(\prod_{p \in F} p^{b_p}\right) \bmod n$
- Repeat the above two steps (guessing other values for a and λ) until you collect $B{+}1$ relations

Factoring Algorithms: Number Field Sieve - Linear Sieve

Linear algebra Step: Consider the relations previously gathered

$$\begin{array}{lrcl}
\texttt{rel}_1: & p_1^{a_{1,1}} p_2^{a_{1,2}} \cdots p_t^{a_{1,t}} & \equiv & p_1^{b_{1,1}} p_2^{b_{1,2}} \cdots p_s^{b_{1,s}} \bmod n \\
\texttt{rel}_2: & p_1^{a_{2,1}} p_2^{a_{2,2}} \cdots p_t^{a_{2,t}} & \equiv & p_1^{b_{2,1}} p_2^{b_{2,2}} \cdots p_s^{b_{2,s}} \bmod n \\
\cdots & & & \\
\texttt{rel}_i: & p_1^{a_{i,1}} p_2^{a_{i,2}} \cdots p_t^{a_{i,t}} & \equiv & p_1^{b_{i,1}} p_2^{b_{i,2}} \cdots p_s^{b_{i,s}} \bmod n \\
\cdots & & &
\end{array}$$

- Select from the above relations a subset of equivalences ($S' = \{\texttt{rel}_{j_1}, \ldots, \texttt{rel}_{j_2}\}$) s.t. their member-wise product gives a relation $X \equiv Y \bmod n$ with $X \leftarrow \left(\prod_{p \in F} p^{a_p}\right)$, and $Y \leftarrow \left(\prod_{p \in F} p^{b_p}\right)$ where the exponents over each prime factor, both in X and Y, is **even**
- In such a way X and Y are guaranteed to be two perfect squares (i.e. $X = x^2$, $Y = y^2$), thus we have $x^2 \equiv y^2 \bmod n$, and we can get a non-trivial factor of n as:

$$\lambda \leftarrow \gcd(\sqrt{X} + \sqrt{Y}, n)$$

Factoring Algorithms: Number Field Sieve - Linear Sieve

The previous step is called **Linear algebra Step** because of the actual method to select the subsets of relations: $S', S'', \ldots$ Each relation $\texttt{rel}_j$ is mapped to a binary row vector as: $(a_{j,1} \bmod 2; a_{j,2} \bmod 2; \ldots; b_{j,1} \bmod 2; b_{j,2} \bmod 2; \ldots)$

$$\begin{array}{lrcl}
\texttt{rel}_1: & p_1^{a_{1,1}} p_2^{a_{1,2}} \cdots p_t^{a_{1,t}} & \equiv & p_1^{b_{1,1}} p_2^{b_{1,2}} \cdots p_s^{b_{1,s}} \bmod n \\
\texttt{rel}_2: & p_1^{a_{2,1}} p_2^{a_{2,2}} \cdots p_t^{a_{2,t}} & \equiv & p_1^{b_{2,1}} p_2^{b_{2,2}} \cdots p_s^{b_{2,s}} \bmod n \\
\cdots & & & \\
\texttt{rel}_i: & p_1^{a_{i,1}} p_2^{a_{i,2}} \cdots p_t^{a_{i,t}} & \equiv & p_1^{b_{i,1}} p_2^{b_{i,2}} \cdots p_s^{b_{i,s}} \bmod n \\
\cdots & & &
\end{array}$$

$$M = \begin{bmatrix}
a_{1,1} \bmod 2 & a_{1,2} \bmod 2 & \ldots & b_{1,1} \bmod 2 & b_{1,2} \bmod 2 & \ldots \\
a_{2,1} \bmod 2 & a_{2,2} \bmod 2 & \ldots & b_{2,1} \bmod 2 & b_{2,2} \bmod 2 & \ldots \\
\quad \cdots & & & & & \\
a_{i,1} \bmod 2 & a_{i,2} \bmod 2 & \ldots & b_{i,1} \bmod 2 & b_{i,2} \bmod 2 & \ldots \\
\quad \cdots & & & & &
\end{bmatrix}$$

The problem of finding the target relations is mapped into the problem of finding the linearly dependent rows of M, which is a well-known linear algebra problem

Factoring Algorithms: Number Field Sieve

- The linear sieve is simply not good enough to factor large numbers
- Indeed, the linear sieve was never proposed as a real factoring algorithm
- However the scheme more efficient methods is the same:
  - choice of an initial condition to drive the search of the target relations (this is $x^2 \equiv y^2 \bmod n$ or another one, which plays the same role w.r.t. the employed strategy)
  - selection of a factorbase ($F = \{p : p \leq B\}$, for a proper B)
  - a sieving step to gather enough relations (the way the relations are found is the most different part w.r.t. the "linear sieve")
  - a linear algebra step (the same of the "linear sieve" one)

Factoring Algorithms: Number Field Sieve

Depending on the number of digits that compose the number n to be factored one of the following methods is selected

- **Elliptic Curve Method** This has a sub-exponential complexity $\mathcal{O}\left(L_p(1/2, \sqrt{2})\right)$, where p denotes the smaller factor of n. It is appropriate when $p \leq 2^{83}$
- **Quadratic Sieve** This is the fastest method for factoring integers of between $2^{240} \leq n \leq 2^{300}$. It has complexity $\mathcal{O}\left(L_n(1/2, 1)\right)$
- **General Number Field Sieve** This is currently the most successful method for numbers greater then 2^{300} and has complexity $\mathcal{O}\left(L_n(1/3, 1.923)\right)$

Discrete Log Extraction Algorithms

Discrete Log Extraction Algorithms

Given a generic finite group G with order n

- Baby-Step/Giant-Step: $\mathcal{O}(\sqrt{n})$ time and $\mathcal{O}(\sqrt{n})$ space
- Pollard-Rho: $\mathcal{O}(\sqrt{n})$ time and $\mathcal{O}(1)$ space
- Pohlig-Hellman: maps DLOG to smaller prime order subgroups of G

The above methods are among the best possible ones for a generic cyclic group, in particular for solving the DLP on Elliptic Curve systems

For groups $(\mathbb{F}_{p^m}, \cdot)$, there are several Sub-exp Methods for solving the DLP

- They are in a close relationship with the sub-exp methods for factoring
- Running time complexity: $\mathcal{O}(L_p(1/3, c))$ with $c = 1.587$ in $\mathbb{F}_{2^m}$ and $c = 1.923$ in $\mathbb{F}_{p^m}, p > 2$
- The sub-exponential algorithms for finite fields are often referred to as "index-calculus"

We will describe in some detail only the case for prime fields: $\mathbb{F}_p$

DL Algorithms: Baby-Step/Giant-Step

The BSGS is a meet-in-the-middle algorithm computing the discrete logarithm in any finite cyclic group G.
It is a fairly simple modification of the naive (brute-force) method of finding discrete logarithms.

Given a cyclic group $G = \langle g \rangle$ of order n, with generator g, and a group element $\beta = g^x$ for some unknown x: the BSGS rewrites x as $x = i \cdot \lceil \sqrt{n} \rceil + j$, with $0 \leq i, j < \lceil \sqrt{n} \rceil$. Therefore, we have:

$$\underbrace{\beta \cdot (g^{-\lceil \sqrt{n} \rceil})^i}_{\text{Giant-Step}} = \underbrace{g^j}_{\text{Baby-Step}}$$

The algorithm pre-computes g^j for several values of j. Then it tries values of i in the left-hand side of the congruence above, in the manner of trial multiplication.

DL Algorithms: Baby-Step/Giant-Step

BSGS Algorithm

Input: $G = \langle g \rangle$ of order n and an element β
Output: A value x s.t. $g^x = \beta$

```
1 begin
2     m ← ⌈√n⌉
3     for j ← 0 to m do
4         Table[j] ← g^j
5     y ← β
6     for i ← 0 to m do
7         if y is in Table then
8             return (i · m + j) mod n
9         y ← y · g^{-m} // Obs.  g^{-m} is pre-computed
```

- The algorithm still works also if $n \geq |G|$ is merely an upper bound on $|G|$
- Running time: $\mathcal{O}(\sqrt{n})$ group op.s; storage: $\mathcal{O}(\sqrt{n})$ group elements.
  This is much better than $\mathcal{O}(n)$ (run time of the brute force calculation)

DL Algorithms: Baby-Step/Giant-Step

Example

Consider $(\mathbb{F}_{31}^*, \cdot)$, generator $g = 3$ and $\beta = 2$
Compute $x = \log_g \beta = \log_3 2$

$n = |\mathbb{F}_{31}^*| = 30,\ m = \lceil \sqrt{n} \rceil = 6,\ g^{-m} = 3^{-6} \bmod 31 \equiv_{31} 21^6 \equiv_{31} 2$

Baby-steps

j:	0	1	2	3	4	5	6
g^j:	1	3	9	27	19	26	16

Giant-steps

i :	0	1	2	3	4	5	6
$\beta(g^{-m})^i$:	$\beta = 2$	4	8	16	$\cdots$	$\cdots$	$\cdots$

Thus, $x = \log_3(2) = (i \cdot m + j) \bmod n \equiv_{30} 3 \cdot 6 + 6 = 24$

DL Algorithms: Baby-Step/Giant-Step

Example

Consider $(\mathbb{F}_{3^3}^*, \cdot)$, where $g = \alpha$ is a primitive root of
$f(x) = x^3 + 2x + 1 \in \mathbb{F}_3[x]$ and $\beta = \alpha^2 + 1$
Compute $x = \log_g \beta = \log_\alpha(\alpha^2 + 1)$

$n = |\mathbb{F}_{3^3}^*| = 26,\ m = \lceil \sqrt{n} \rceil = 6,\ g^{-m} = \alpha^{-6} \equiv (2\alpha^2 + 1)^6 \equiv 2\alpha^2 + \alpha + 1$

Baby-steps

j:	0	1	2	3	4	5	6
g^j:	1	α	α^2	$\alpha+2$	$\alpha^2+2\alpha$	$2\alpha^2+\alpha+2$	$\alpha^2+\alpha+1$

Giant-steps

i :	0	1	2	3	4	5	6
$\beta(g^{-m})^i$:	α^2+1	$2\alpha^2$	$\alpha+1$	$\alpha+2$	$\cdots$	$\cdots$	$\cdots$

Thus, $x = \log_\alpha(\alpha^2+1) = (i \cdot m + j) \bmod n \equiv_{26} 3 \cdot 6 + 3 = 21$

DL Algorithms: Pollard's ρ

- Remember that the BSGS method has $\mathcal{O}(\sqrt{n})$ time complexity with also $\mathcal{O}(\sqrt{n})$ storage requirement
- The Pollard's ρ method, adapted to solve the discrete log in a generic cyclic group, employs only constant storage ($\mathcal{O}(1)$) with an average running time of $\mathcal{O}(\sqrt{n})$ group operations

DL Algorithms: Pollard's ρ

Let $G = \langle g \rangle$ denote a finite cyclic group of order $n{=}|G|$ and let $\mathbf{x}$ be the discrete logarithm we want to find:

$$\mathbf{x} \equiv_n \log_g^{\mathrm{D}} h \ \Leftrightarrow\ h = g^{\mathbf{x}}$$

We partition the group G into three sets, i.e.: $G{=}S_1 \cup S_2 \cup S_3$ (not subgroups), where we assume $1 \notin S_2$, and then define the following random walk $x_0, x_1, x_2, \ldots$ on the group G:

$$x_{i+1} = f(x_i) = \begin{cases} h \cdot x_i, & x_i \in S_1 \\ x_i^2, & x_i \in S_2 \\ g \cdot x_i, & x_i \in S_3 \end{cases} \quad \text{starting from } x_0 = 1$$

In practice, we assume $x_i = g^{a_i} \cdot h^{b_i}$ and actually keep track of three pieces of information (x_i, a_i, b_i) starting from $(x_i, a_i, b_i) = (1, 0, 0)$:

$$a_{i+1} = \begin{cases} a_i \bmod n, & x_i \in S_1 \\ 2\,a_i \bmod n, & x_i \in S_2 \\ a_i + 1 \bmod n, & x_i \in S_3 \end{cases} \qquad b_{i+1} = \begin{cases} b_i + 1 \bmod n, & x_i \in S_1 \\ 2\,b_i \bmod n, & x_i \in S_2 \\ b_i \bmod n, & x_i \in S_3 \end{cases}$$

$$\forall\ i,\ \log_g(x_i) = a_i + b_i \cdot \log_g h = a_i + b_i \cdot \mathbf{x}$$

DL Algorithms: Pollard's ρ

Therefore we have three random-walk maps $f(\cdot), f^{(a)}(\cdot,\cdot), f^{(b)}(\cdot,\cdot)$:

$$a_{i+1} = f^{(a)}(a_i, x_i),\ b_{i+1} = f^{(b)}(b_i, x_i),\ x_{i+1} = f(x_i)$$

Considering the tortoise ($x_{i+1} \leftarrow f(x_i)$) and hare pointers ($y_{i+1} \leftarrow f(f((x_i)), y_0 = x_0$) along the random walk, the Floyd's cycle finding condition gives us that, for a certain step $s{>}0$:

$$x_s = y_{2s} \Leftrightarrow g^{a_s} \cdot h^{b_s} = g^{a_{2s}} \cdot h^{b_{2s}}$$

keeping into account that $h = g^x$ we obtain:

$$-(a_{2s} - a_m) \equiv_n \mathbf{x}\,(b_{2s} - b_s) \ \Leftrightarrow\ \mathbf{x} \equiv_n -\frac{a_{2s} - a_s}{b_{2s} - b_m}$$

Note that, if it happens that $b_{2s} - b_s \equiv_n 0$ then the method fails. However for large n, the probability of such an event is negligible.

DL Algorithms: Pollard's ρ

Pollard's ρ Algorithm

Input: $G = \langle g \rangle$ of order n and an element $h \in G$
Output: A value x s.t. $g^x = h$

begin
  $a \leftarrow 0,\ b \leftarrow 0,\ x \leftarrow 1$
  $da \leftarrow 0,\ db \leftarrow 0,\ dx \leftarrow 1,\ i \leftarrow 0$
  while true do
    $a \leftarrow f^{(a)}(a, x),\ b \leftarrow f^{(b)}(b, x)\ x \leftarrow f(x)$
    $da \leftarrow f^{(a)}(f^{(a)}(da, dx), f(dx)),\ db \leftarrow f^{(b)}(f^{(b)}(db, dx), f(dx))$
    $dx \leftarrow f(f(dx))$
    if $b = db$ **then**
      return *"Failure"*
    if $x = dx$ **then**
      return $\left(-\frac{da-a}{db-b}\right) \bmod n$
    $i \leftarrow i + 1$

DL Algorithms: Pollard's ρ

Example

Let $G{=}\langle g \rangle$ be a subgroup of $(\mathbb{F}^*_{607}, \cdot)$ with $g{=}64$ and order $n{=}101$. Consider $h{=}122{\in}G$, find $x{\equiv_n}\log_g h$, i.e., x s.t. $h{\equiv}g^x{\Leftrightarrow}122{\equiv_{607}}64^x$

Assume
$S_1 = \{d \in \mathbb{F}^*_{607} : 1 \le x \le 201\}$
$S_2 = \{d \in \mathbb{F}^*_{607} : 202 \le x \le 403\}$
$S_3 = \{d \in \mathbb{F}^*_{607} : 404 \le x \le 606\}$

Starting from $(x_0, a_0, b_0) = (1, 0, 0)$ we stop when $x_i{=}x_{2i}$ and rename the index value as m. Thus, $m = 14$,

$$g^{a_m} \cdot h^{b_m} = g^{a_{2m}} \cdot h^{b_{2m}} \Rightarrow$$

$$x \equiv_n -\frac{a_{2m} - a_m}{b_{2m} - b_m} \equiv_{101} \frac{64}{6} \equiv_{101} 78$$

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
1	122	0	1	316	0	2
2	316	0	2	172	0	8
3	308	0	4	137	0	18
4	172	0	8	7	0	38
5	346	0	9	309	0	78
6	137	0	18	352	0	56
7	325	0	19	167	0	12
8	7	0	38	498	0	26
9	247	0	39	172	2	52
10	309	0	78	137	4	5
11	182	0	55	7	8	12
12	352	0	56	309	16	26
13	76	0	11	352	32	53
14	**167**	0	12	**167**	64	6

DL Algorithms: Pohlig-Hellman Method

Lemma

Let $G{=}\langle g \rangle$ be a finite cyclic group with generator g, order $\mathbf{n}{=}|G|$. If we know that $\mathbf{n}$ is B-smooth, with $B{\le}\log_2^c(n)$, then also its prime factorization is known, $\mathbf{n}{=}\prod_i^s p_i^{e_i}$, $s{\ge}1$, $e_i{\ge}1$, and the computation of $x{=}\log_g(\beta)$, $\beta{\in}G$ will require $O(\sqrt{B})$ operations.

Proof.

We want to compute: $x \equiv_n \log_g(\beta),\ \beta \in G$

since $\mathbf{n} = \prod_{\mathbf{i}}^{\mathbf{s}} \mathbf{p_i^{e_i}}$, $s \ge 1, e_i \ge 1$, and $p_i < B$, the DL problem would be equivalent to the following congruence system, in case x_is were known:

$$\begin{cases} x \equiv x_1 \bmod p_1^{e_1} \\ x \equiv x_2 \bmod p_2^{e_2} \\ \dots \\ x \equiv x_s \bmod p_s^{e_s} \end{cases} \overset{\text{CRT}}{\Rightarrow} x \equiv_n \sum_{i=1}^{s} x_i \left(\frac{n}{p_i^{e_i}}\right)\left(\left(\frac{n}{p_i^{e_i}}\right)^{-1} \bmod p_i^{e_i}\right)$$

DL Algorithms: Pohlig-Hellman Method

Proof... cont.

Therefore the DL problem is now split up in **s** smaller problems:

$$x_i = x \bmod p_i^{e_i} = \log_g(\beta) \bmod p_i^{e_i}, \qquad 0 \le i \le s$$

How do we find the values $\mathbf{x_i} \equiv \mathbf{x} \bmod \mathbf{p_i^{e_i}}$?

It is always possible to express any number in radix p_i through a finite number of steps. Indeed, we can think the x_i value as:

$$x_i = l_0 + l_1\, p_i + l_2\, p_i^2 + l_3\, p_i^3 + \ldots + l_j\, p_i^j + \ldots + 0\, p_i^{e_i}, \qquad 0 \le l_j < p_i$$

Compute $\eta \leftarrow g^{\frac{n}{p_i}}, \gamma_0 = 1$ and consider $\delta_0 = \left(\beta\, \gamma_0^{-1}\right)^{\frac{n}{p_i^1}}$:

$$\delta_0 = \left(g^x\right)^{\frac{n}{p_i^1}} = g^{x_i\, \frac{n}{p_i^1}} = g^{(l_0 + l_1\, p_i + \ldots)\, \frac{n}{p_i^1}} = g^{l_0\, \frac{n}{p_i^1}} \Rightarrow \mathbf{l_0} \equiv_{\mathbf{p_i}} \log_\eta(\delta_0)$$

Note that: $x = \mathtt{xq} \cdot p_i^{e_i} + x_i$ for some value of $\mathtt{xq}$, and $g^{\mathtt{xq} \cdot p_i^{e_i} \cdot \frac{n}{p_i}} = 1$

DL Algorithms: Pohlig-Hellman Method

Proof... cont.

Compute $\gamma_1 \leftarrow \gamma_0\, g^{l_0\, p_i^0}$ and consider $\delta_1 = \left(\beta\, \gamma_1^{-1}\right)^{\frac{n}{p_i^2}}$:

$$\delta_1 = \left(g^{x - l_0}\right)^{\frac{n}{p_i^2}} = g^{(l_1\, p_i + l_2\, p_i^2 + l_3\, p_i^3 + \ldots)\, \frac{n}{p_i^2}} = g^{l_1\, \frac{n}{p_i}} \Rightarrow \mathbf{l_1} \equiv_{\mathbf{p_i}} \log_\eta(\delta_1)$$

Compute $\gamma_2 \leftarrow \gamma_1\, g^{l_1\, p_i^1} = g^{l_0 + l_1 p_i}$ and consider $\delta_2 = \left(\beta\, \gamma_2^{-1}\right)^{\frac{n}{p_i^3}}$:

$$\delta_2 = \left(g^{x - l_0 - l_1 p_i}\right)^{\frac{n}{p_i^3}} = g^{(l_2\, p_i^2 + l_3\, p_i^3 + \ldots)\, \frac{n}{p_i^3}} = g^{l_2\, \frac{n}{p_i}} \Rightarrow \mathbf{l_2} \equiv_{\mathbf{p_i}} \log_\eta(\delta_2)$$

etc... $\delta_{e_i - 1} = \ldots \Rightarrow \mathbf{l_{e_i - 1}} \equiv_{\mathbf{p_i}} \log_\eta(\delta_{\mathbf{e_i - 1}})$

In this way, it is possible to recover the value of $x_i = x \bmod p_i^{e_i}$, solving e_i dlogs in $\mathbb{Z}_{p_i}$ for all the prime factors of the modulus n (that is for all $0 \le i \le s$) and subsequently apply the CRT and recover original dlog x.

DL Algorithms: Pohlig-Hellman Method

Pohlig-Hellman Algorithm

Input: Group G of order n with $n = \prod_i^s p_i^{e_i}$, B-smooth (with B "small") and an element $\beta \in G$
Output: A value x s.t. $g^x = \beta$

begin
  for $i \leftarrow 0$ **to** s **do**
    $\gamma \leftarrow 1,\ \eta \leftarrow g^{\frac{n}{p_i}}$ `// assume` $l_{-1} = 0$
    for $j \leftarrow 0$ **to** $e_i - 1$ **do**
      $\gamma \leftarrow \gamma \cdot g^{l_{j-1}\, p_i^{j-1}}$
      $\delta \leftarrow \left(\beta \cdot \gamma^{-1}\right)^{\frac{n}{p_i^{j+1}}}$
      $l_j \leftarrow \log_\eta \delta$ `// f.i., through the BSGS`
    Reconstruct the integer value:
    $x_i \leftarrow l_0 + l_1\, p_i + l_2\, p_i^2 + l_3\, p_i^3 + \ldots + l_{e_i - 1} p_i^{e_i - 1}$
  Apply the CRT to the congruence system:
  $x \equiv x_i \bmod p_i^{e_i} \quad \forall\, 0 \le i \le s$
  return x

DL Algorithms: Pohlig-Hellman

- Since the intermediate steps in the Pohlig–Hellman algorithm are quite simple, the difficulty of solving a DLP is dominated by the time required to solve the DLP in the **cyclic subgroups of prime order**
- Hence, for generic groups the complexity of the BSGS method in prime subgroups ($\mathcal{O}(\sqrt{p_i})$) will dominate the overall complexity
- Given the order of the group $n = |G|$, if $n = \prod_{i=1}^s p_i^{e_i}$ has a B-smooth factorization then the Running Time of the Pohlig-Hellman algorithm is given by:

$$\mathcal{O}\left(\sum_{i=1}^{s} e_i \cdot (\log_2 p_i + \sqrt{p_i})\right) = \mathcal{O}\left(s \cdot \max\{e_i\}_{i=1}^s \cdot \left(\log_2 n + \sqrt{B}\right)\right)$$

bit operations

DL Algorithms: Pohlig-Hellman

Example

Consider $G = (\mathbb{Z}^*_{251}, \cdot)$ with generator $g = 71$.
Find $x \equiv_{251} \log_g \beta$, $\beta = 210$

- $n = |G| = 250$, $n = 2^1 \cdot 5^3$, B-smooth with $B = 6$,
  G is cyclic as it coincides with the multiplicative group of a finite field
  (the DL exists for sure!)
- Consider the computation of $x_1 \equiv x \bmod 2^1$
  Obs. x_1 will be composed by $e_1 = 1$ digit in base $p_1 = 2$, that is
  $x_1 \equiv x \bmod 2^1 = l_0 \cdot 2^0 + 0 \cdot 2^1$. We need to find l_0:
  $\eta = g^{\frac{n}{p_1}} = 71^{\frac{250}{2}} \equiv_{251} 71^{125} \equiv_{251} 250$

  $\gamma_0 = 1$;
  $\delta_0 = \left(\beta\gamma_0^{-1}\right)^{\frac{n}{p_1}} \equiv_{251} 210^{\frac{250}{2}} \equiv_{251} 250$
  $l_0 \equiv_{p_1} \log_\eta \delta_0 \equiv_2 \log_{250}(250) \equiv_2 1$ (BSGS is trivial)

  $x_1 \equiv x \bmod 2^1 \equiv 1$

DL Algorithms: Pohlig-Hellman

Example ... (cont.)

- Consider the computation of $x_2 \equiv x \bmod 5^3$, this will be composed by $e_2 = 3$ digits in radix $p_2 = 5$, that is $x_2 = l_0 + l_1 \cdot 5^1 + l_2 \cdot 5^2$.
  We need to find $l_0, l_1, l_2 \in \{0, \ldots, 4\}$, thus:

  $\eta = g^{\frac{n}{p_2}} = 71^{\frac{250}{5}} \equiv_{251} 71^{50} \equiv_{251} 20$

  $\gamma_0 = 1$;
  $\delta_0 = \left(\beta\gamma_0^{-1}\right)^{\frac{n}{p_2}} \equiv_{251} 210^{\frac{250}{5}} \equiv_{251} 149$
  $\mathbf{l_0} \equiv_{p_2} \log_\eta \delta_0 \equiv_5 \log_{20}(149) \equiv_5 \mathbf{2}$ (by brute-force: $20^2 \equiv_{251} 149$, $20^3, 20^4$)

  $\gamma_1 = \gamma_0 \cdot g^{l_0}$;
  $\delta_1 = \left(\beta\gamma_1^{-1}\right)^{\frac{n}{p_2^2}} \equiv_{251} (210 \cdot 71)^{\frac{250}{25}} \equiv_{251} 113$
  $\mathbf{l_1} \equiv_{p_2} \log_\eta \delta_1 \equiv_5 \log_{20}(113) \equiv_5 \ldots$ (brute-force) $\ldots \equiv_5 \mathbf{4}$

DL Algorithms: Pohlig-Hellman

Example ... (cont.)

$\gamma_2 = \gamma_1 \cdot g^{l_1 p_2}$;
$\delta_2 = \left(\beta\gamma_1^{-1}\right)^{\frac{n}{p_2^3}} \equiv_{251} (210 \cdot 115)^{\frac{250}{125}} \equiv_{251} 149$
$\mathbf{l_2} \equiv_{p_2} \log_\eta \delta_2 \equiv_5 \log_{20}(149) \equiv_5 \ldots$ (brute-force) $\ldots \equiv_5 \mathbf{2}$

x_2 will be composed by $e_2 = 3$ digits in base $p_2 = 5$:
$x_2 = l_0 + l_1 5 + l_2 5^2 = 72$

$$x = \log_g(\beta) \bmod (n) \Leftrightarrow \begin{cases} x \equiv_{p_1^{e_1}} x_1 \\ x \equiv_{p_2^{e_2}} x_2 \end{cases} \Leftrightarrow \begin{cases} x \equiv_2 1 \\ x \equiv_{125} 72 \end{cases} \Rightarrow$$

$$x = \log_g(\beta) \bmod (n) \equiv_{250} 1 \cdot 125 \cdot (125^{-1} \bmod 2) + 72 \cdot 2 \cdot (2^{-1} \bmod 125)$$

$$x = \log_{71}(210) \bmod 250 \equiv_{250} 9197 \equiv_{250} 197$$

Sub-exp DLP in Finite Fields: Index Calculus (1)

...this is not included in the written exam

Let $G = (\mathbb{F}_{q^m}, \cdot) = \langle g \rangle$, ($q$ a prime power) be a cyclic group of order n

- Choose a subset $S = \{s_1, \ldots, s_t\} \subset G$ called "factor base" or "decomposition base"

Sieving Step

- The set S is built in such a way that a large number of elements in G can be written as a product of elements in S. This implies that a generic element $g^k \in G$ can be factorized in S, with high probability: $g^k = \prod_{i=1}^{t} s_i^{c_i}$, $c_i \geq 1$
- Collect at least $t + 1$ relations as the aforementioned one

Linear algebra Step

- Set up the congruence system: $\left\{ k_i = \sum_{j=1}^{t} c_i \log_g(s_i) \bmod (q-1) \right\}_{i=1}^{t}$
- Solve the system assuming $x_i = \log_g(s_i)$, $\forall\, i$ as unknowns and fill a look-up table $\mathbf{T} = \langle \mathbf{s_i}, \log_\mathbf{g}(\mathbf{s_i}) \rangle$

Sub-exp DLP in Finite Fields: Index Calculus (2)

Discrete Log extraction

- Given a generic element $\beta \in G$
- Pick $l \overset{\text{Random}}{\leftarrow} \mathbb{Z}_q^*$
- Compute $(\beta \cdot g^l)$
- Try to factor $(\beta \cdot g^l)$ with the factorbase elements (like in the sieving step):

$$(\beta \cdot g^l) = \prod_{i=1}^{t} (s_i)^{d_i}, \quad d_i \geq 1$$

- If the factorization is possible, then take the $\log_g(\cdot)$ of both members:

$$\log_g \beta \equiv \sum_{i=1}^{t} d_i \log_g(s_i) - l \bmod (q-1)$$

recover the values $\log_g(s_i)$ through indexing the table T

Sub-exp DLP in Finite Fields: Index Calculus (3)

Several specialized versions of this methods exist: tailoring the sieving step of the GNFS algorithm[a] (for factoring integers)

- General Number Field Sieve for $\mathbb{F}_{2^m}$: $\mathcal{O}\left(L_{2^m}(1/3, 1.587)\right)$
- General Number Field Sieve for $\mathbb{F}_p$, $p > 2$:
  $\mathcal{O}\left(L_p(1/3, \sqrt[3]{\frac{64}{9}} = 1.923)\right)$
- Function Field Sieve for $\mathbb{F}_{p^m}$, with p "small":
  $\mathcal{O}\left(L_p(1/3, \sqrt[3]{\frac{32}{9}} = 1.5263)\right)$

[a] A C++ implementation of the GNFS for factoring can be found to http://pGNFS.org