Théorie de l'information et codage

Master de cryptographie

Cours 11 : Logarithme discret

27 et 30 mars 2009

Université Rennes 1

The discrete logarithm

Definition

Let G be a (multiplicative) group. Let g an element of G of finite order l (ie $g^l = 1$). Let $H = \left(g^1, g^2, \cdots, g^l\right)$ the subgroup of G generated by g

$$\forall h \in H, \exists n \in [1, \cdots, l] \text{ such that } h = g^n$$

n is said to be the discrete logarithm of h in base g and is denoted $\log_g(h)$.
n est determined modulo l

Examples :

- $(\mathbb{Z}/n\mathbb{Z}, +)$
- The multiplicative group of a finite field : $\mathbb{F}_q^*$
- An elliptic curve
- The Jacobian of an hyperellitic curve

Goal : find a group where finding the discrete logarithm is difficult and use it in cryptography

Diffie-Hellman key exchange

Public parameters : a group G, an element g in G of order l

- A picks a random number a in $[1, l-1]$
- A computes g^a in G and sends it to B
- B picks a random number b in $[1, l-1]$
- B computes g^b in G and sends it to A
- B gets g^a and computes $g^{ab} = (g^a)^b$
- A gets g^b and computes $g^{ab} = \left(g^b\right)^a$
- A and B share a common secret key g^{ab}.

An eavesdropper knows g and intercepts g^a, g^b but cannot deduce g^{ab} without solving a discrete logarithm problem

El-Gamal encryption

Public parameters : a group G, an element g in G of order l

- A chooses a random number k_a in $[1, l-1]$ (her private key)
- A computes $K_a = g^{k_a}$ in G (her public key) and distributes it
- B wants to send a message m to (we assume that $m \in G$)

  B picks a random number k in $[1, l-1]$

  B sends $\left(g^k, mK_a^k\right)$ to A
- A then receives $\left(g^k, mK_a^k\right)$ and can recover m because

$$m = \frac{mK_a^k}{\left(g^k\right)^{k_a}}$$

In fact it is just a Diffie-Hellman but k is a session private key for B

- DLP (Discrete Logarithm Problem)
  Given g and g^a, recover a
- CDH (Computational Diffie-Hellman)
  Given g, g^a and g^b, recover g^{ab}
- DDH (Decisional Diffie Hellman)
  Given g, g^a, g^b and g^c, decide if $g^{ab} = g^c$

$$\text{DLP} > \text{CDH} > \text{DDH}$$

CDH is sufficient to break key-exchange or El-Gamal

DDH is sufficient to weaken El-Gamal (eg if we suspect a message m, we can verify if we are right if DDH is easy)

Computing the discrete logarithm

Definition

An algorithm to compute the discrete log is said to be generic if it uses only the following operations

- the composition of two groups elements
- the inverse of an element
- the equality test

In other words, it can be used on any group

Theorem (Shoup)

Let p be the largest prime number dividing the order l of the element g. Computing a discrete logarithm using a generic algorithm requires at least $O(\sqrt{p})$ operations in the group

Brute force

Compute g^k for all $k < l$ and check if it is equal to $h \rightarrow O(l)$ operations

Polhig-Hellman (from l to p)

We assume, to simplify, that the order l of g equals pq
Given $h \in (g^1, g^2, \cdots, g^l)$, we want n such that $h = g^n$.

Let us write $n = n_p + kp$, so we have :

$$\begin{aligned} h &= g^{n_p+kp} \\ h^q &= g^{q(n_p+kp)} \\ h^q &= g^{qn_p} g^{kl} \\ h^q &= g^{qn_p} \end{aligned}$$

Putting $g' = g^q$ and $h' = h^q$, n_p is the discrete logarithm of h' in base g' and, by construction, g' is an element of order p
Compute n mod q in the same way and recover n from n mod p and n mod q thanks to the CRT

This method can of course be generalized to any l

Conclusion : The complexity of the discrete logarithm problem in a group of size l does not depend on l but on the largest prime dividing l

Baby step, Giant step (Shanks)

Reminder : Given $h \in (g^1, g^2, \cdots, g^l)$, we want n such that $h = g^n$
Let $s = \lceil \sqrt{l} \rceil + 1$, there are $u < s$ and $v < s$ such that $n = u + vs$. Then we have

$$\begin{aligned} h &= g^{u+vs} \\ h &= g^u (g^s)^v \\ h (g^{-1})^u &= (g^s)^v \end{aligned}$$

Algorithm

1. Baby step : Compute and store $h (g^{-1})^u$ in G for $0 \leq u < s$
2. Giant step : For v from 0 to s do
   - compute $(g^s)^v$ in G
   - if $(g^s)^v = h (g^{-1})^u$ for a certain u then return $u + vs$

Complexity : $2\sqrt{l}$ operations in G (optimal)
Drawback : necessary to store $\sqrt{l}$ elements of G

Baby step, giant step : an example

$G = \mathbb{F}_p^*$ with $p = 83$, $\# G = 82 = 2 \times 41$. We choose $g = 3$ (order 41)
We want to compute $\log_3(30)$. We take $s = 7$.

Precomputations $3^{-1} = 28$ mod 83 and $3^7 = 29$ mod 83

Baby step : Compute all the $30\left(3^{-1}\right)^u$ modulo 83 for $0 \leq u < s$

u=0	30
u=1	10
u=2	31
u=3	38
u=4	68
u=5	78
u=6	26

Giant step : For v from 0 to $s-1$ compute $\left(3^7\right)^v$ modulo 83

v=0	1
v=1	29
v=2	11
v=3	70
v=4	38

Then $n = 3 + 4 \times 7 = 31$.

In 10 steps instead of 31 (brute force)

Baby step, giant step : a real example

On a group of size around 2^{80} (security level of 40 bits)

Computation time

On a recent PC, an operation on such a group takes around $10\mu s$
2^{40} operations $\longrightarrow \sim 4$ months

Realizable

Baby step, giant step : a real example

On a group of size around 2^{80} (security level of 40 bits)

Computation time

On a recent PC, an operation on such a group takes around $10\mu s$
2^{40} operations $\rightarrow \sim 4$ months

Realizable

In term of memory usage

80 bits = 10 bytes $\rightarrow$ 20 bytes to store an element of G

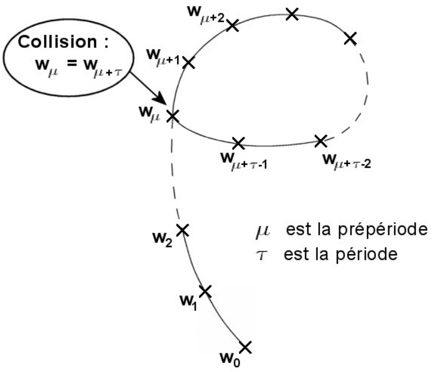
$$20 \times 2^{40} = 20\ 000 \text{ GB approximately}$$

and it must be RAM

The limiting factor is the memory

Birthday paradox : If elements of G are randomly picked, the number of draws before a collision (the last element picked was already picked before) is around $\sqrt{\frac{\pi l}{2}}$.

Principle : Realize a random walk $w_{i+1} = \phi(w_i)$ until a collision happens



$\tau \sim \mu \sim \sqrt{\frac{\pi l}{8}}$

Pollard ρ

A trick to avoid storage :

$$\text{If } i = k\tau \text{ and } i \geq \mu, \text{ then } w_i = w_{2i}$$

We just look for a collision, don't want to compute τ and μ.

Algorithm (Pollard, Floyd)

1. initialization w_0, $z_0 = w_0$
2. Compute $w_{i+1} = \phi(w_i)$ and $z_{i+1} = \phi(\phi(z_i))$
3. If $w_{i+1} = z_{i+1}$ then return i and $2i$, else $i = i + 1$ and repeat

Advantage : No storage and always in $\sqrt{l}$
Drawback : Compute 3 times ϕ. There are improvements (balance between computation cost and frequencies of collision).

Application to discrete logarithm

$$w_i = g^{a_i} h^{b_i}$$

$$w_i = w_j \quad \Rightarrow \quad \begin{aligned} g^{a_i} h^{b_i} &= g^{a_j} h^{b_j} \\ h^{b_i - b_j} &= g^{a_j - a_i} \\ h &= g^{\frac{a_j - a_i}{b_i - b_j}} \end{aligned}$$

$$\text{so } n = \frac{a_j - a_i}{b_i - b_j} \mod l$$

Easy to parallelize. A 109 bits elliptic curve discrete logarithm (55 bits security) was broken in 2002 using this algorithm with 10000 PC running during 549 days
Next challenge : 131 bits (20 000 $)
Available on http ://www.certicom.com/

Example of random walk for the discrete logarithm ($w_i = g^{a_i} h^{b_i}$)

We split G in 3 subset of approximately the same size

$$G = G_1 \cup G_2 \cup G_3$$

$$w_0 = g \quad (a_0 = 1, b_0 = 0)$$

$$w_{i+1} = \phi(w_i) = \begin{cases} hw_i & \text{si} \quad w_i \in G_1 \\ w_i^2 & \text{si} \quad w_i \in G_2 \\ gw_i & \text{si} \quad w_i \in G_3 \end{cases}$$

So

$$(a_{i+1}, b_{i+1}) = \begin{cases} (a_i, b_i + 1) & \text{si} \quad w_i \in G_1 \\ (2a_i, 2b_i) & \text{si} \quad w_i \in G_2 \\ (a_i + 1, b_i) & \text{si} \quad w_i \in G_3 \end{cases}$$

In fact, not random enough and the collision happens later than expected

Summary of the constraints on G

- G must contain a subgroup of prime order p where the discrete log problem will be applied
- If we want a n bits security level, p must have $2n$ bits (because of generic attacks)
- The goal is to find groups such that there are no better attacks than generic ones

A candidate for G : $\mathbb{F}_p^*$

p prime, $\mathbb{F}_p$ finite field
The set of non-zero elements in $\mathbb{F}_p$ is a (multiplicative) group of order $p-1 \rightarrow$ natural candidate for G

Index calculus algorithm can compute the discrete logarithm in such a group in subexponential time

Security level of 80 bits $\rightarrow p \sim 2^{1024}$
Same security as RSA

In practice, we chose p a 1024 bits prime number such that $p-1$ is divisible by a 160 bits prime number l. In this case, the operations take place in $\mathbb{F}_p^*$ but the keys (the exponents) are in $\mathbb{Z}/l\mathbb{Z}$.

Smaller keys than RSA (160 bits instead of 1024).

Diffie-Hellman key-exchange on $\mathbb{F}_p^*$ for 80 bits security

We chose l a 160 bits prime number and p a 1024 bits prime number such that $p - 1 = kl$. Let g be an element in $\mathbb{F}_p^*$ of order l. Public parameters are l, p and g.

- A picks a random number a in $[1, l-1]$
- A computes g^a modulo p and sends it to B
- B picks a random number b in $[1, l-1]$
- B computes g^b modulo p and sends it to A
- B gets g^a and computes $g^{ab} = (g^a)^b$ modulo p
- A gets g^b and computes $g^{ab} = (g^b)^a$ modulo p
- A and B share the common secret key g^{ab}

The standard procedure to generate l, p and g is given by the NIST
http ://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf

$$\begin{aligned} \text{for instance } l &= 2^{160} + 7 \\ p &= 1 + \left(2^{160} + 7\right)\left(2^{864} + 218\right) \sim 2^{1024} \\ g &= 2^{\frac{p-1}{l}} \bmod p \end{aligned}$$

Other candidates

- Other finite fields. In particular those of the form $\mathbb{F}_{2^n}$. Index calculus works in the same way : 1024 bits are necessary for 80 bits of security
- Elliptic curves and genus 2 (hyperelliptic) curves for which nobody knows better attacks than generic ones : 160 bits are sufficient for 80 bits of security
- Curves of larger genus but the Index calculus algorithm can be adapted

Advantages and Drawbacks compared to RSA

- Smaller key size
- Faster decryption (eg 160 bits exponent instead of 1024)
- Slower encryption (if small e is used in RSA)
- Trivial key generation

Principle of Index calculus (Western-Miller, Kraitchik)

We assume, to simplify, that $\# G = l$ (ie all elements of G are a power of g). We want to compute the discrete log of h

1. Construct a "factor basis" made of some particular elements of G, $(g_i)_{i=1..c}$. By definition, we have $g_i = g^{\log_g(g_i)}$
2. Find relations between these elements of the form

$$g^{\alpha_g} h^{\alpha_h} = g_1^{\alpha_1} g_2^{\alpha_2} \cdots g_c^{\alpha_c}$$

   This give relations of the form

$$g^{\alpha_g} g^{\log_g(h)\alpha_h} = g^{\log_g(g_1)\alpha_1} g^{\log_g(g_2)\alpha_2} \ldots g^{\log_g(g_c)\alpha_c}$$

   and then

$$\alpha_g = -\log_g(h)\alpha_h + \log_g(g_1)\alpha_1 + \log_g(g_2)\alpha_2 + \cdots + \log_g(g_c)\alpha_c$$

   which is a linear equation between $\log_g(h)$ and the $\log_g(g_i)$.

Principle of Index calculus (Western-Miller, Kraitchik)

3. When you have $c+1$ independent relations of this form, solve the system (standard linear algebra) assuming that $\log_g(h)$ and the $\log_g(g_i)$ are the unknowns. The solution then gives $\log_g(h)$

For efficiency, must find a balance between step 2 and step 3 (which are contradictory)

This algorithm is generic but is efficient only if a good factor basis can be used

- on $\mathbb{F}_p^*$, we choose the small prime numbers
- on $\mathbb{F}_{2^n}^*$, we choose the polynomials of small degrees
- on large genus curves, we choose elements of small degrees