Mathematical preliminaries of crypto
Discrete Logarithm

Helger Lipmaa

May 10, 2017

1 Discrete Logarithm and Discrete Logarithm Assumption

Many cryptographic primitives and protocols are based on the assumption that the discrete logarithm problem is secure in some group. We use multiplicative notion in the case of such groups, by letting $a \cdot b$ to be the group operation. In this case,

$$g^x = \underbrace{g \cdot g \cdot \cdots \cdot g}_{x \text{ times}}$$

is the *exponentiation operation*. Moreover, if $h = g^x$ then we say that x is the *discrete logarithm* of h on basis g, and write $x = \log_g h$.

The *discrete logarithm assumption* in a cyclic finite group $\mathbb{G}$ of order p is that given a generator g of $\mathbb{G}$ and a randomly generated element h of $\mathbb{G}$, it is computationally infeasible to find $\log_g h$. To formalize this, we need to define infeasibility. Thus, more formally, the discrete logarithm assumption in $\mathbb{G}$ is $(\tau, \sigma, \varepsilon)$-hard, if for any probabilistic polynomial-time adversary $\mathcal{A}$ that runs in time τ and requires at most σ space (in some fixed machine model), the probability that the next experiment returns 1 is at most ε:

1. Let g be some (possibly fixed) generator of $\mathbb{G}$.
2. Let $h \leftarrow \mathbb{G}$ be a randomly chosen element of $\mathbb{G}$.
3. The adversary $\mathcal{A}$ is given (g, h) as an input, and he returns x.
4. The experiment returns 1 if $h = g^x$, and 0 otherwise.

Thus, infeasibility means that no efficient (running in time τ or less and taking space σ or less) algorithm can produce the discrete logarithm with probability ε or better. In practice, one can choose $\tau = 2^{80}$, $\sigma = 2^{40}$ (since space is a more valuable resource than time!) and $\varepsilon = 2^{-80}$. One also considers a family of groups $\mathbb{G}_\kappa$ indexed by a security parameter κ, and then requires that the discrete logarithm every group $\mathbb{G}_\kappa$ is $(\tau(\kappa), \sigma(\kappa), \varepsilon(\kappa))$ secure, where $\tau(\kappa)$ and $\sigma(\kappa)$ are arbitrary positive polynomials, and $\varepsilon(\kappa)$ is some (positive) negligible function.

Remark 1 *In practice, one often divides the time τ to the precomputation time (which can be performed before the attacker obtains the input (g, h)) and the postcomputation time (which is performed after the input is obtained). We will not formalize this aspect, but we will mention it in several places.*

2 Trivial Examples

Discrete logarithm is not always hard.

Consider the group $\mathbb{Z}_p$ with a prime p, and with group operation $+$. In this case, exponentiation corresponds to multiplication with an integer, $x \cdot g = g + \cdots + g$.

Exercise 1 *Show that computing discrete logarithm in this (additive) group is not $(\Theta(1), \Theta(1), 1)$-hard (in group operations).*

Exhaustive search.

Consider any cyclic finite group $\mathbb{G}$ of order p, and a generator $g \in \mathbb{G}$. The next adversary $\mathcal{A}$ clearly outputs the discrete logarithm x of any element $h \leftarrow \mathbb{G}$:

1. Let (g, h) be the inputs to the discrete logarithm problem
2. Set $g' \leftarrow 1$
3. For $i = 0$ to $p - 1$ do:
   (a) If $g' = h$ then: output i. Exit the program
   (b) Let $g' \leftarrow g' \cdot g$

To see the correctness of this algorithm, note that in the inner loop g' is always equal to g^i. Therefore, this adversary just tests if h is equal to g^i for every $i \in \mathbb{Z}_p$, and if it finds a correct i then it returns it. It is also straightforward to see that this algorithm outputs the correct answer always, thus $\varepsilon = 1$. However, the adversary works in time $\Theta(p \log^2 p)$ (assuming $\Theta(\log^2 p)$ is the time to perform a single multiplication in $\mathbb{Z}_p$) and space $\Theta(\log p)$ (the space needed to memorize i and one element of $\mathbb{Z}_p$). Still, it means that $\mathbb{G}$ is not more than $(\Theta(p \log^2 p), \Theta(\log p), 1)$-secure.

3 Generic Algorithms

In this section, we will present some discrete logarithm algorithms that, like the exhaustive search algorithm, work in arbitrary cyclic groups *grp* of order p. Such algorithms are called *generic*. These algorithms are much faster than the exhaustive search algorithm.

3.1 Shanks's Babystep-Giantstep Algorithm

Let g be some generator of $\mathbb{G}$. Consider the next algorithm that gets (g, h) as inputs:

1. Let $n \leftarrow 1 + \lfloor \sqrt{p} \rfloor$. Let $u \leftarrow g^{-n}$
2. Create two lists:
   (a) List 1: $g^0 = 1$, $g^1 = g$, g^2, ..., g^n
   `// multiplication by` g `is the ''baby step''`
   (b) List 2: h, $h \cdot u$, $h \cdot u^2$, $h \cdot u^3$, ..., $h \cdot u^n$
   `// multiplication by` g^{-n} `is the ''giant step''`
3. Find a match between the two lists, that is, a pair (i, j) such that $g^i = h \cdot g^{-jn}$
4. Return $x \leftarrow i + jn$ as a solution to $h = g^x$

Clearly, if such a pair (i, j) exists then since $g^i = h\dot{g}^{-jn}$, we get $h = g^{i+jn}$. If this match exists, it can be found in $\Theta(n \log n)$ steps by sorting two lists and then doing a standard search over sorted lists. Creating unsorted lists takes approximately $2n$ multiplications, and thus the whole process is dominated by $\Theta(n \log n) = \Theta(\sqrt{p} \log p)$ group operations. However, differently from the exhaustive search algorithm, there is also a considerable memory requirement of $\Theta(\sqrt{p} \log p)$ bits.

Let us now show that lists 1 and lists 2 always have a match. For this, let $h = g^x$ for unknown x, and write $x = q \cdot n + r$ for $0 \leq r < n$. Since $1 < x < p$, then also $q < n$. Thus, $h = g^{qn+r}$, or $g^r = h \cdot g^{-qn}$ for some $0 \leq q, r \leq n$. Therefore, a match exists always.

Thus, putting everything together, we see that the Babystep-Giantstep algorithm breaks the discrete logarithm in time $\Theta(\sqrt{p} \log p)$ (group operations) and space $\Theta(\sqrt{p} \log p)$ (bits) with certainty $\varepsilon = 1$.

3.2 Pohlig-Hellman Algorithm

Assume that the group order is equal to

$$N = p_1^{e_1} \dots q_t^{e_t} \tag{1}$$

for some primes $p_i \neq p_j$ and $e_i > 0$. For example, consider the equation $g^x \equiv h \mod q$ for a prime q. While here q is prime, the discrete logarithm needs to be computed in $\mathbb{Z}_q^*$ and the order of the group $\mathbb{Z}_q^*$ is $N := q - 1$, which is *not* a prime. The following Pohlig-Hellman algorithm demonstrates that for security reasons, the group order $N := q - 1$ must have at least one large prime factor.

Assume that Eq. (1) holds. Assume that if $g \in \mathbb{G}$ has prime power order $p_i^{e_i}$, then we can solve $g^x = h$ in $O(S_{p_i^{e_i}})$ steps. For example, by using the Babystep-Giantstep algorithm, $S_{p_i^{e_i}} = p_i^{e_i/2}$. Now, assume that $g \in \mathbb{G}$ has order N. We show that $g^x = h$ can be solved in $O(\sum_{i=1}^{t} S_{q_i^{e_i}} + \log N)$ steps by using the next Pohlig-Hellman algorithm:

1. For each $1 \le i \le t$
   (a) Let $g_i \leftarrow g^{N/q_i^{e_i}}$ and $h_i \leftarrow h^{N/q_i^{e_i}}$
   (b) Since g_i has prime power order $q_i^{e_i}$, use the given algorithm to solve the discrete logarithm problem $g_i^y = h_i$. Let $y_i = y$ be the corresponding solution.
2. Use the Chinese Remainder theorem to solve $x \equiv y_1 \pmod{q_1^{e_1}}$, $\ldots$, $x \equiv y_1 \pmod{q_t^{e_t}}$ for x. Return x

Let us now prove that this algorithm works in time claimed. The running time is clear, since the first step takes time $\Theta(\sum S_{q_i^{e_i}})$ (group operations), and the second step takes time $\Theta(\log N)$ (group operations).

Let us next show that the algorithm gives a correct solution. Let x be the output of the Pohlig-Hellman algorithm. Then for each i, there exists z_i such that

$$x = y_i + q_i^{e_i} z_i \ .$$

Thus

$$\begin{aligned}
(g^x)^{N/q_i^{e_i}} &= (g^{y_i + q_i^{e_i} z_i})^{N/q_i^{e_i}} \\
&= (g^{N/q_i^{e_i}})^{y_i} \underbrace{g^{N z_i}}_{g^N = 1} \\
&= ((g^{N/q_i^{e_i}})^{y_i}) \\
&= g_i^{y_i} \\
&= h_i \\
&= h^{N/q_i^{e_i}}
\end{aligned}$$

We can rewrite this as

$$\frac{N}{q_i^{e_i}} \cdot x \equiv \frac{N}{q_i^{e_i}} \cdot \log_g h \pmod{N} \ . \tag{2}$$

Since the numbers $N/q_1^{e_1}, \ldots, N/q_t^{e_t}$ have no nontrivial common factors, we can use the extended Euclidean algorithm to find integers $c_1, \ldots, c_t$, such that

$$\frac{N}{q_1^{e_1}} \cdot c_1 + \cdots + \frac{N}{q_t^{e_t}} \cdot c_t = 1 \ . \tag{3}$$

Now, multiplying the both sides of Eq. (2) by c_i and summing over $i \in \{1, \ldots, t\}$, we get

$$\underbrace{\sum_{i=1}^{t} \frac{N}{q_i^{e_i}} \cdot c_i}_{=1} \cdot x \equiv \underbrace{\frac{N}{q_i^{e_i}} \cdot c_i}_{=1} \cdot \log_g h \pmod{N} \ .$$

Thus,

$$x \equiv \log_g h \pmod{N} \ ,$$

and therefore the Pohlig-Hellman algorithm outputs the correct value of x.

Important Lesson.

Due to the Pohlig-Hellman algorithm, computing discrete logarithms in a group of order N is basically as hard as computing discrete logarithms modulo $p_i^{e_i}$, where $p_i^{e_i}$ is the largest prime power factor of N. Thus, one does not gain much by considering the general choice of N, but instead one can just choose N to be a prime power. (By choosing more general N we do not gain in security, but we loose in efficiency since together with N, also the cost of group operations increases.) In the following subsubsection, we show that one should always pick a prime N.

Pohlig-Hellman Algorithm: Prime-Power Case

Assume that q is a prime and that we know an algorithm that takes S_q steps to solve the discrete logarithm problem $g^x = h$ in $\mathbb{G}$ if g has order q. Let $g \in \mathbb{G}$ be an element of order q^e with $e \geq 1$. We now show how to solve discrete logarithm problem $g^x = h$ n $O(eS_q)$ steps. The key idea is to write the unknown exponent x as

$$x = x_0 + x_1 q + \cdots + x_{e-1} q^{e-1}$$

with $0 \leq x_i < q$, and then determine successively x_0, x_1, x_2, ... First, observe that $g^{q^{e-1}}$ has order q. Thus,

$$\begin{aligned} h^{q^{e-1}} =& (g^x)^{q^{e-1}} = (g^{x_0+\cdots+x_{e-1}q^{e-1}})^{q^{e-1}} \\ & g^{x_0 q^{e-1}} \cdot (\underbrace{g^{q^e}}_{=1})^{x_1+x_2 q+\cdots+x_{e-1}q^{e-2}} = (g^{q^{e-1}})^{x_0} \end{aligned}$$

Since $g^{q^{e-1}}$ has order q, the equation

$$(g^{q^{e-1}})^{x_0} = h^{q^{e-1}}$$

is a discrete logarithm problem with the base element having order q, and thus x_0 can be found in time S_q. Once x_0 is found, we know x_0 such that $g^{x_0 q^{e-1}} = h^{q^{e-1}}$ in $\mathbb{G}$.

We now compute

$$\begin{aligned} h^{q^{e-2}} =& (g^x)^{q^{e-2}} = (g^{x_0+\cdots+x_{e-1}q^{e-1}})^{q^{e-2}} \\ & g^{x_0 q^{e-2}} \cdot g^{x_1 q^{e-1}} \cdot (\underbrace{g^{q^e}}_{=1})^{x_2+\cdots+x_{e-1}q^{e-3}} = g^{x_0 q^{e-2}} \cdot g^{x_1 q^{e-1}} \quad . \end{aligned}$$

Since we already know x_0, we can now determine x_1 exactly as we determined x_0 in time S_q.

Continuing similarly, we can determine all coefficients x_i one by one (and thus also the discrete logarithm x), in total time $O(eS_q)$.

Exercise 2 *Use the Pohlig-Hellman algorithm to solve* $5448^x = 6909$ *in* $\mathbb{Z}_{11251}^*$. *Write down all intermediate steps.*

Important Lesson.

Due to the Pohlig-Hellman algorithm, computing discrete logarithms in a group of order N is basically as hard as computing discrete logarithms modulo p_i, where p_i is the largest prime factor of N. Thus, one does not gain much by considering the general choice of N, but instead one can just choose N to be a prime. (By choosing more general N we do not gain in security, but we loose in efficiency since together with N, also the cost of group operations increases.) In practice, one of the possible choices of $\mathbb{G}$ is to choose an order q subgroup of $\mathbb{Z}_p^*$, where p is a large prime (≥ 2048 bits) and q is a smaller prime (≥ 160 bits). Since the order of $\mathbb{G}$ is q, q has to be at least 2^{160} elements to be secure against Babystep-Giantstep and Pohlig-Hellman algorithms. The choice $|p| \geq 2048$ is required to obtain security against the index calculus algorithm (see the next lecture).