Stanford University
Topics in Cryptography (CS 355) **Lecture #5**
Instructor: Lior Rotem **April 18, 2023**

Disclaimer *These lecture notes are aimed to serve as a supplementary resource, and are not written as a replacement for attending the in-person lecture. Some material might appear here in a more sketched form than in the lecture, and vice versa. These notes probably contain typos and might even contain errors. If you find any, please let me know.*

Outline *In this lecture, we ask exactly how hard the discrete log problem is. We present a trivial algorithm and then show that there are better-than-trivial "generic" algorithms that solve the discrete log problem in any group. We then present the index-calculus algorithm for computing discrete log in integer groups.*

1 How hard is discrete log?

On the importance of concrete security. Recall the discrete log problem: Given a cyclic group $\mathbb{G}$ of order p generated by g, and a uniformly-random group element $h \in \mathbb{G}$, the problem is to find an element $x \in \mathbb{Z}_p$ such that $h = g^x$. In previous lectures, we saw a very fundamental assumption in cryptography: that there exist groups in the discrete log problem is hard. We thought of the term "hard" in its asymptotic meaning; that is, the success probability of any PPT algorithm is negligible (in the security parameter, which is roughly $\log |\mathbb{G}|$). But asymptotic guarantees do not tell the whole picture. In practice, we might be tied to groups of concrete sizes due to various reasons (efficiency, legacy implementations, and more). If we are restricted to working in a 256-bit group, then it just might be that the asymptotic do not "kick in" yet for groups of this size. For example, suppose that we have a PPT algorithm that solves the discrete log problem with probability 1 for groups of size $\leq 2^{1024}$ and with probability 0 for groups of size $> 2^{1024}$. Such an algorithm does not contradict that asymptotic discrete log assumption, but it tells us that discrete log is easy in 256-bit groups!

So what we want are *concrete security bounds* for the hardness of discrete log. Given a group $\mathbb{G}$ of order p, and an algorithm A that runs in time t, what is the probability that A solves the discrete log problem in $\mathbb{G}$? In this lecture, we will primarily be interested in upper bounds: That is, we will see algorithms that solve the discrete log problem and analyze their runtime vs. success probability tradeoffs.

The cost model. Throughout the lecture, we will fix $\mathbb{G}, p$ and g. For simplicity of presentation, we will count all operations in $\mathbb{G}$ and in $\mathbb{Z}_p$ as unit-cost. This includes group operations, exponentiations in the group, and arithmetic operations. In groups of interest, all of these can be implemented using $\mathsf{poly}(\log |\mathbb{G}|)$ basic operations. Hence, the bounds that we will derive will be accurate up to a multiplicative $\mathsf{poly}(\log |\mathbb{G}|)$ term, which will be of lower order.

A trivial solution. As a warm-up, consider a brute force algorithm A, whose running time is $t \leq p$. Given as input a group element h, A iterates over $i = 0, \dots, t-1$ and checks if $g^i = h$. If it finds such an i, it outputs it and terminates. If no such i is found, A outputs $\perp$ (implying failure) and terminates. Observe that A indeed runs in time t.

What is the success probability of A? Let $h = g^x$. Then A succeeds if $x \in \{0, \dots, t-1\}$. Since x is chosen at random from $\mathbb{Z}_p$, this happens with probability t/p.

2 A Random Collision-Based Algorithm

Recall Pedersen Commitments we saw in lecture #3. These commitments were parameterized by g and an additional random group element h and were defined by $c = g^r \cdot h^m$ for a message m and randomness r in $\mathbb{Z}_p$. We proved that an adversary that breaks binding – that is, finds $m_0 \neq m_1$ and r_0, r_1 such that $g^{r_0} \cdot h^{m_0} = g^{r_1} \cdot h^{m_1}$ – immediately implies an algorithm that finds the discrete log of h with respect to g by outputting $(r_1 - r_0) \cdot (m_0 - m_1)^{-1}$. We will see three algorithms that use this fact.

The first algorithm simply finds a random collision. Consider the following algorithm A that gets a discrete log challenge h and proceeds as follows:

1. For $i = 1, \dots, t$:
   (a) Sample $m_i, r_i \xleftarrow{\$} \mathbb{Z}_p$.
   (b) If $g^{r_i} \cdot h^{m_i} = g^{r_j} \cdot h^{m_j}$ and $m_i \neq m_j$ for some $j < i$, output $(r_i - r_j) \cdot (m_j - m_i)^{-1}$ and terminate.
2. Output $\perp$ and terminate.

The running time of A is $O(t)$ since it makes t iterations, in each of which it makes a constant number of operations. By the birthday bound, the probability that A finds a collision in Step 1b is $\approx t^2/p$. This means that the success probability of A is $\approx t^2/p$, which is better than the trivial algorithm that we saw before.

Observe that the random collision algorithm has two main drawbacks:

1. **Probabilistic success:** The algorithm succeeds with probability $\approx t^2/p$. We can ensure that it succeeds with constant probability by taking $t = \Omega(\sqrt{p})$, but succeeding with probability 1 means reverting to exhaustive search.
2. **Large memory:** The more serious drawback is that the algorithm, as presented, requires very large memory – of size $O(t)$ which is potentially as large as $O(\sqrt{p})$. There are several ways to combat this fact. We will see one.

3 The Baby-Step Giant-Step Algorithm

We start by presenting an algorithm that runs in time $O(\sqrt{p})$ and succeeds with probability 1. Then we discuss how to generalize it to an algorithm that runs in time $t < \sqrt{p}$ and succeeds with probability $\approx t^2/p$.

Let $h = g^x$. The main observation behind the baby-step giant-step algorithm is that x can be re-written as $x = i \cdot \lceil \sqrt{p} \rceil + j$ for $i, j \in \{0, \ldots, \lceil \sqrt{p} \rceil\}$. So we can write $g^j = h \cdot \left(g^{-\lceil \sqrt{p} \rceil}\right)^i$. Using this observation, given an input $h = g^x$, the Baby-Step Giant-Step algorithm A is defined as follows:

1. Set $m = \lceil \sqrt{p} \rceil$ and $v = g^{-m}$.
2. Compute $g_i = g^i$ for $i = 0, \ldots, m$ and store the results.
3. Set $u_0 = h$.
4. For $j = 0, \ldots, m$:
    (a) If $u_j = g_i$ for some $i \in \{0, \ldots, m\}$ then output $x = i \cdot m + j$ and terminate.
    (b) Set $u_{j+1} \leftarrow u_j \cdot v$.

A time-probability tradeoff. Note that one can set $m = t$. Then, by the same analysis, the algorithm is guaranteed to work whenever $x \leq t^2$. Hence, since x is chosen at random from $\mathbb{Z}_p$, the success probability is t^2/p.

On random self-reducibility. The above analysis guarantees that a t^2/p-fraction of the inputs to A are "good" in the sense that A succeeds in computing their discrete log with probability 1. However, for the rest of the inputs A succeeds with probability 0. What if we want that A to succeed with probability t^2/p for *all inputs* (where the probability is over the random coins of A)? For this, we can leverage the fact that a discrete log is randomly self-reducible. Using the above A, we construct an algorithm B that has the strong guarantee that we want. Given input h, B samples a random $r \overset{\$}{\leftarrow} \mathbb{Z}_p$ and computes $h' = h \cdot g^r$. It then invokes A on h'. Whenever A outputs an exponent x', B outputs $x = x' - r$. Since h' is distributed uniformly in $\mathbb{G}$ (since g is a generator), it is "good" with probability t^2/p.

A persistent caveat: Large memory. Note that the above algorithm still requires that one maintains all the g_i values in memory; this means keeping $\Theta(\sqrt{p})$ elements in memory. Next, we will work our way to a better solution in terms of memory.

4 Pollard's Rho Algorithm

Pollard's Rho algorithm is based on a general technique due to Floyd for finding a cycle in a sequence of values $x_0, f(x_0), f(f(x_0)), f(f(f(x_0))), \ldots$ for some function f. The idea will be to apply this technique to a sequence of the form $u_0 = g^{a_0} \cdot h^{b_0}, u_1 = f(u_0) = g^{a_1} \cdot h^{b_1}, u_2 = f(u_1) = g^{a_2} \cdot h^{b_2}, \ldots$ to find u_i and u_j such that $u_i = u_j$ and $b_i \neq b_j$. Then, we can compute $\log_g(h)$ as before.

We start by assuming that we have a function f as above, that is: (1) random; and (2) allows us to compute the a_i and b_i values for each u_i. Later, we will see how to choose this f. Assume that after ℓ steps, we hit a cycle of length c. That is, for each $i \geq \ell$ it holds that $u_i = u_{i+c \cdot k}$ for every positive integer k.

Let i be the first index greater than ℓ that is a multiple of c; that is $i = k \cdot c \geq \ell$ for some integer k. Since $i \geq \ell$, it is in the cycle, so we know that $u_i = u_{i+k \cdot c} = u_{2i}$. The collision that the algorithm will find will be $u_i = g^{a_i} \cdot h^{b_i}$ and $u_{2i} = g^{a_{2i}} \cdot h^{b_{2i}}$.

On input h the algorithm is defined as follows:

1. Sample $a_0, b_0 \overset{\$}{\leftarrow} \mathbb{Z}_p$ and set $u_0 \leftarrow g^{a_0} \cdot h^{b_0}$ and $v_0 \leftarrow u_0$.
2. For $i = 1, 2, \ldots$
   (a) Set $u_i = g^{a_i} \cdot h^{b_i} \leftarrow f(u_{i-1})$.
   (b) Set $v_i = f(f(v_{i-1}))$. // note that $v_i = u_{2i} = g^{a_{2i}} \cdot h^{b_{2i}}$.
   (c) If $v_i = u_i$ and $b_i \neq b_{2i}$ output $x = (a_i - a_{2i})/(b_{2i} - b_i) \mod p$.

Since f is random, the birthday paradox says that with high probability, we should have a collision within $O(\sqrt{p})$ steps. Moreover, with probability $1 - 1/p$, it holds that $b_i \neq b_{2i}$.

How to choose f? We assumed that f is random (this was used by the birthday paradox analysis). One tempting option is to set $f = H$ for some cryptographic hash function H; that is $u_{i+1} = H(u_i)$. If we think of this hash function as "complicated" enough, then we can hope that it performs similarly to a random hash function. The problem is that this does not allow us to know what a_{i+1} and b_{i+1} are, even if we know a_i and b_i. Another option is to let and $f(u_i) = g^{a_{i+1}, b_{i+1}}$ and $(a_{i+1}, b_{i+1}) = H(u_i)$. This is a reasonable choice, though it asks quite a lot of the hash function: note that we want it to "act random" on potentially $\approx \sqrt{p}$ inputs.

Another option which is often used in to partition $\mathbb{G}$ into m subsets $\mathcal{S}_1, \ldots, \mathcal{S}_m$, for some parameter m (in practice, m can be quite small). At the beginning of its execution, the algorithm samples s pairs $(c_i, d_i) \overset{\$}{\leftarrow} \mathbb{Z}_p \times \mathbb{Z}_p$ and computes $\delta_i \leftarrow g^{c_i} \cdot h^{d_i}$. Then, f is defined as $f(u) = u \cdot \delta_i$, where i is the index satisfying $u \in \mathcal{S}_i$. There are options for the choice of f which we will not cover in this lecture.

It should be noted that there are ways to de-randomize the choice of f such that it *we can prove* that it behaves almost as good as a truly uniformly-sampled f. By derandmizing the choice of f, we mean sampling it from a family of function which is smaller than the family of all function from $\mathbb{G}$ to $\mathbb{G}$. Alas, such choices of f give us functions which are either too slow to compute, or take up too much memory.

The memory usage. Ignoring the memory needed to implement f, Pollard's algorithm only requires that we store u_i and v_i at each point in time, so we only need $O(\log p)$ bits of memory. This is much better than the $\Theta(\sqrt{p})$ we had before! If we choose to implement f using a cryptographic hash function, no added memory is needed (apart from the memory for computing H, which we can typically ignore). Using the precomputed δ values method, we need to remember additional m values, so we need a memory of $O(m \cdot \log p)$ bits.

5 Optimality of the t^2/p tradeoff

The algorithm that we just saw is "generic" in the sense that it does not use any specific information about the group $\mathbb{G}$ or the representation of group elements. In particular, it would perform just the same in *any* cyclic group of order p. Interestingly, Victor Shoup proved that a success probability of t^2/p is the best we can hope for from a generic algorithm. We will not see the proof in this talk, but students are encouraged to take a look at the paper (a link is on the course website).

6 Index Calculus in Integer Groups

This is an algorithm that does better than the generic ones for groups of integers. Concretely, it runs in strictly sub-exponential time, whereas the generic algorithms that we saw run in exponential time. If the group is of order $\approx 2^\lambda$, exponential time means $2^{\Theta(\lambda)}$ whereas subexponential time means $\Theta(2^{\lambda^\epsilon})$ for some $\epsilon < 1$.

We consider the group $\mathbb{G} = \mathbb{Z}_q^*$ for a prime q. This is always a cyclic group of order $q-1$. We start with an informal overview of the algorithm and then present it in more detail. The Index Calculus algorithm is parameterized by a "smoothness" bound B. Let $\{p_1, \ldots, p_\ell\}$ be the set of all primes up to B (including B). Say that we are getting a group element h and we have to find x such that $h = g^x$ (as before, g is the generator of the group at hand). If we sample a random $e \overset{\$}{\leftarrow} \mathbb{Z}_q$ and compute $u = g^e/h \in \mathbb{Z}_q^*$, it might just be that u, lifted to the integers, has all its prime factors in B. Such an integer is called B-smooth. If this happens, we find these factors in time complexity that depends on B (e.g., by trial division), which we will analyze later. Let us say that this is the case, so we can write $u = \prod_i p_i^{e_i}$ for some $e_1, \ldots, e_\ell$. Taking the discrete log mod q (with respect to g) and rearranging, we get the equation

$$e = \log h + e_1 \log p_1 + \cdots + e_\ell \log p_\ell \mod q-1$$

Unfortuentaely, we do not know $\{\log p_i\}_i$, so we cannot compute $\log h$ from this equation. What we can do is repeated the process m times, until we get a system of the form

$$\begin{aligned} e_1 &= \log h + e_{1,1} \log p_1 + \cdots + e_{1,\ell} \log p_\ell \mod q-1 \\ &\vdots \\ e_m &= \log h + e_{m,1} \log p_1 + \cdots + e_{m,\ell} \log p_\ell \mod q-1 \end{aligned}$$

For large enough m, the system will determine $\log h$, which we can then find using linear algebra.[1]

Runtime anaylsis. To analyze the running time of the algorithm, we will need to rely on a number-theoretic fact (we will not see a proof): Assume $B < q$ (this will be the case for us). Asymptotically, the fraction of B-smooth numbers in $\{1, \ldots, q-1\}$ is $\approx 1/u^u$ for $u = \log q / \log B$.

Using this fact, we can turn to analyze the cost of each of the algorithm's steps:

- Finding all primes $\leq B$ can be done in time $\tilde{O}(B)$.
- Finding a single equation takes time $\approx Bu^u$. Why? Testing whether or not an integer is B smooth can be done in time $\approx B$. We sample e's until $g^e/h \in \mathbb{Z}_q^*$ is B-smooth. By the fact above, in expectation, we sample u^u such e's until we hit a B-smooth number. Overall, finding one equation takes expected time $\approx Bu^u$.

[1]Note that $q-1$ is not a prime, so we are likely to encounter zero divisors when doing Gaussian elimination; there are standard ways for solving this problem which we will not see.

- It can be shown that it is sufficient to find $m \approx B$ equations. So overall, finding m equations takes time $\approx B^2 u^u$.
- Gaussian elimination takes time $< B^3$.

Overall, the runtime can be bounded by $\approx B^3 + B^2 u^u$. Now it is time to set B. For that, we will introduce L-notation: we denote $L_n[\alpha, c] = e^{c(\log n)^{\alpha}(\log\log n)^{1-\alpha}}$. So it can be shown that choosing $B = L_q[1/2, 1] = e^{\sqrt{\log q \log\log q}}$ we have that $u^u \lessapprox B$. So the total running time is

$$B^3 = L_q[1/2, 3] = e^{3\sqrt{\log q \log\log q}}.$$

Note that this is sub-exponential in the security parameter $\lambda \approx \log q$.

Preprocessing. Note that the p_is are independent of h. Therefore, one can split the computation to two parts. In the first part, which can be done ahead of time, one finds enough equations to compute $\{\log p_i\}_i$. Then, in an "online phase", when the challenge h becomes known, a single equation is sufficient to compute $\log h$. The overall computation still takes B^3 time, but the online computation only takes $B^2 = L_q(1/2, 2)$ time.

Best known algorithm. Improvements to the algorithm that we just described are known. These take the runtime down to $\approx L_q[1/3, 2] = e^{2(\log q)^{1/3}(\log\log q)^{2/3}}$.

Implications to prime order subgroups. It is generally not a good idea to work over $\mathbb{Z}_q^*$, since a group element $h = g^x$ leaks information about its discrete log x (try to figure out what information). Hence, when working over integer groups, we would typically take $q = 2p + 1$ where p and q are both primes (q is called a "safe prime" and q is called a Sophie Germain prime after the French mathematician). Then, $\mathbb{Z}_q^*$ contains a subgroup of order p and it is better to work in such a subgroup. The attack can be extended to work in such subgroups.

7 Conclusion

Integer groups are perhaps to most general to work over, and indeed, these were the main groups used in the early days of public-key cryptography. However, as we saw, such groups are vulnerable to sub-exponential discrete log computations. Let us see why this is a problem in more depth. Say that we want $\approx$ 128-bits of security (this is a standard choice). This means that to solve discrete log with constant probability, an adversary has to run in time $\approx 2^{128}$. If we work over an integer group of order q, we have to set q to be roughly 2^{2048} to get $\approx$ 120-bit security. If we insist on $>$ 128-bit security, we need to set $q = 2^{4096}$ to get yields $\approx$ 130-bit security! This means that each group element requires thousands of bits to represent! Worse still, to get just 10 more bits of security, we had to double the representation length of group elements.

In contrast, if we work in a group in which the best-known discrete log algorithms are the generic algorithms described above, then to get 128-bit security, it is enough to use a group of order $\approx 2^{256}$. For any additional bit of security, we need to add just 2 more bits for the representation. In the next lecture, we will explore such groups.