CS 355: Topics in Cryptography **Spring 2019**

Lecture 11: Generic Discrete-Log Algorithms

Dima Kogan

Last lecture, we've seen two examples of real-world cryptanalysis: The first exploited the fact that many RSA public keys shared a common factor (due to insufficient randomness when selecting the primes). The second attack exploited the fact that primes were chosen from a bad distribution.

Today, we'll look at classic cryptanalytic algorithms for the discrete-log problem.

1 Generic Discrete-Log Algorithms

Recall: the discrete-log problem in a cyclic group $\mathbb{G}$ of order $q = |G|$: given a generator $g \in \mathbb{G}$ and a challenge $h = g^x \in \mathbb{G}$ for a random $x \in \mathbb{Z}_q$, find x.

Naive algorithm: exhaustive search. Compute g^a for $0 \le a < q$ and compare with h. Requires $O(q)$ group operations.

Baby-Step Giant-Step Algorithm [1]

Let $B \leftarrow \lceil \sqrt{q} \rceil$. For $h = g^x \in \mathbb{G}$ where $0 \le x < q$, we can write:

$$x = a + B \cdot b \quad \text{where } a, b \in \{0, 1, \ldots, B-1\}.$$

Then

$$h = g^x = g^{a+B\cdot b}$$
$$h \cdot g^{-a} = g^{B\cdot b}.$$

The algorithm finds a and b as following:

compute $B \leftarrow \lceil \sqrt{q} \rceil$.
for $b = 0, 1, \ldots, B-1$, compute g^{Bb} and store in a searchable data structure $\mathcal{T}[g^{Bb}] = b$.
for $a = 0, 1, \ldots, B-1$:
  compute $t \leftarrow h \cdot g^{-a}$
  if $t \in \mathcal{T}$, let $b \leftarrow \mathcal{T}[t]$, and output $x \leftarrow a + B \cdot b$.

The algorithm is deterministic and always finds the discrete log. It's running time is $O(\sqrt{q} \log q)$ bit operations (since group elements are $\log q$ bits each).

What's the main disadvantage of this algorithm? It takes space $O(\sqrt{q} \log)$ bits.

Pollard's Rho Algorithm [2]

The first observation is that if we find $a, b, a', b' \in \mathbb{Z}_q$ such that

$$g^a h^b = g^{a'} h^{b'}$$

and $b \neq b'$, then we can find the discrete log of h as:

$$\log_g(h) = \frac{a - a'}{b - b'} \pmod{q}.$$

Consider now a *random walk* on a group G, which is a sequence

$$u_1 \to u_2 := f(u_1) \to u_3 := f(u_2) \to \ldots \to u_{i+1} := u_i \to \ldots$$

for some random function $f : \mathbb{G} \to \mathbb{G}$. The second observation is that such a random walk of length $\Theta(\sqrt{q})$ will, by the Birthday Paradox, "collide with itself", i.e., $u_{i+\ell} = u_i$ for some $i, l = O(\sqrt{q})$.

The idea behind Pollard's Rho algorithm is to construct a *pseudorandom walk* on $\mathbb{G}$ with the following properties:

1. The starting point is $u_1 = g^{a_1} h^{b_1}$ for some random $a_1, b_1 \xleftarrow{R} \mathbb{Z}_q$.
2. Each step in the walk is a point $u_i = g^{a_i} h^{b_i}$ where we know $a_i, b_i \in \mathbb{Z}_q$.
3. The walk "behaves like a random walk", and thus collides with itself after $\Theta(\sqrt{q})$ steps.

The problem is that if we just choose use some standard hash function as our random function f, then even if we know $u_i = g^{a_i} h^{b_i}$, we don't know the corresponding a_{i+1}, b_{i+1} of $u_{i+1} := f(u_i)$. Instead, we choose an arbitrary partitions of $\mathbb{G}$ to three sets $G = S_0 \cup S_1 \cup S_2$ and define f as following:

$$f(u) = \begin{cases} u^2 & \text{if } u \in S_0 \\ g \cdot u & \text{if } u \in S_1 \\ h \cdot u & \text{if } u \in S_2 \end{cases}.$$

Note that each step in the walk can be computed using a single group operation. A ideal way to define the partitions is by using some hash function $s : \mathbb{G} \to \{0, 1, 2\}$, which would make the walk more "random". But for speed, one often simply looks on the bit representation $b(u)$ of $u \in \mathbb{G}$ as an integer, and then takes $s(u) := b(u) \mod 3$ (or probably uses 4 sets with a different step for each).

Note that if $u_i = g^{a_i} h^{b_i}$, then

$$(a_{i+1}, b_{i=1}) = \begin{cases} (2a_i, 2b_i) & \text{if } u \in S_0 \\ (a_i + 1, b_i) & \text{if } u \in S_1 \\ (a_i, b_i + 1) & \text{if } u \in S_2 \end{cases},$$

so Property 2 above holds.

The walk is deterministic and not really random, but heuristically it behaves like a random walk and so after $\Theta(\sqrt{q})$ steps, we have:

$$g^{a_i} h^{b_i} = g^{a_j} h^{b_j},$$

and we can recover $\log_g(h)$.

Why is it important that f above depends only on u_i and not on the specific representation (a_i, b_i)? Because if the function would have depended on the representation, than two colliding segments of the walk would have diverged rather than "merging" after the collision.

A naive implementation would still require $\widetilde{O}(\sqrt{q})$ space, but we can do much better using Floyd's cycle finding algorithm ("the turtle and the hare "): at each step we only maintain (u_i, a_i, b_i) and (u_{2i}, a_{2i}, b_{2i}) (and we can compute $u_{2i} \to u_{2(i+1)}$ by taking two consecutive steps of the walk), and we continue until $u_i = u_{2i}$ at which point we recover the discrete log as above. This uses only $O(\log q)$ space.

2 Lower Bound for Generic Algorithms

All of the above discrete-log algorithms are *generic*: they do not exploit any properties of the representation of the group elements, but rather only use the group operation as a black box.[1] We would like to formally model this type of algorithms.

Definition 1. An encoding of a group $\mathbb{G}$ of prime order q is an injective function $\sigma : \mathbb{G} \to \{0,1\}^{\lceil t \log q \rceil}$ for some $t > 0$.

We think of σ as assigning labels to the group elements, and we denote the set of labels $\mathcal{L} = \{0,1\}^{\lceil t \log q \rceil}$. A generic discrete-log algorithm for a group $\mathbb{G}$ of order q is a probabilistic algorithm that:

- Gets as input the order of the group q, as well as the labels $\sigma(g), \sigma(h) \in \mathcal{L}$.
- Gets black box access to the group operation. More specifically, the algorithm gets oracle access to an oracle $\mathcal{O} : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$, such that $\mathcal{O}(\sigma(g), \sigma(h)) = \sigma(gh)$.
- Outputs an integer $x \in \mathbb{Z}_q$.

The success probability of a generic discrete-log algorithm $\mathcal{A}$ for a group $\mathbb{G}$ is:

$$\mathsf{Succ}(\mathcal{A}, \mathbb{G}) := \Pr\left[\mathcal{A}(q, \sigma(g), \sigma(g^x)) = x \colon \sigma \xleftarrow{R} \mathcal{L}^{\mathbb{G}} \text{ injective}, g \xleftarrow{R} \mathbb{G}, x \xleftarrow{R} \mathbb{Z}_q\right], \tag{1}$$

where by $\mathcal{L}^{\mathbb{G}}$ is shorthand for the set of all functions from $\mathbb{G}$ to $\mathcal{L}$, and we further restrict ourselves to injective functions only.

What about oracle access to computing inverses? We can allow the algorithm oracle access to computing inverses as well (and the result below still holds), or we can assume that the algorithm can compute the inverse $g^{-1} = g^{q-1}$ using $O(\log q)$ multiplications (by repeated squaring).

Theorem 2 ([3])**.** *If $\mathcal{A}$ is a generic discrete-log algorithm that makes m queries to the group oracle, then* $\mathsf{Succ}(A, q) \leq O(m^2/q)$.

Proof. Consider the following two games:

[1] An example of a non-generic algorithm would be a discrete-log algorithm for $\mathbb{F}_p^*$ that uses the structure of $\mathbb{F}_p$. Such an algorithm could for example use addition and subtraction in $\mathbb{F}_p$. You will see such an algorithm in the next homework assignment.

Adversary		**Challenger**
		$g \xleftarrow{R} \mathbb{G}$, $x \xleftarrow{R} \mathbb{Z}_q$, $\sigma \xleftarrow{R} \mathcal{L}^{\mathbb{G}}$ injective
	$\sigma(g), \sigma(g^x)$ ⟵	
Repeat m times {	$\sigma(u_i), \sigma(v_i)$ ⟶	
	$\sigma(u_i v_i)$ ⟵	
	$y \in \mathbb{Z}_q$ ⟶	
		Check $x \stackrel{?}{=} y$

Game 1

Adversary		**Challenger**
		Initialize empty table $T \in \mathcal{L} \times \mathbb{Z}_q[X]$
		$\sigma_1, \sigma_x \xleftarrow{R} \mathcal{L}$ (distinct)
		add mappings $(\sigma_1, 1)$ and (σ_x, X) to T
	σ_1, σ_x ⟵	
	σ_u, σ_u ⟶	
		Lookup (σ_u, u), (σ_v, v) in T
		Check if the polynomial $u+v$ is in T:
		if yes:
		reply with σ_{u+v}
Repeat m times		if no:
		choose $\sigma_{u+v} \xleftarrow{R} \mathcal{L}$ from all unused labels
		add $(\sigma_{u+v}, u+v)$ to T
		reply with σ_{u+v}
	σ_{u+v} ⟵	
	$y \in \mathbb{Z}_q$ ⟶	
		Choose $x \xleftarrow{R} \mathbb{Z}_q$ and check $x \stackrel{?}{=} y$

Game 2

Game 1 corresponds to the execution of a generic discrete log algorithm.
In Game 2, $x \xleftarrow{R} \mathbb{Z}_q$ is chosen independently at random at the end of the game, so:

$$\leq \Pr_{x,\sigma}[\mathcal{A} \text{ wins in Game 2}] = 1/q\,.$$

Moreover, observe that view of the adversary in Game 2 is "almost identical" to his view in Game 1: the only problem is that if there exists two polynomials f and f' in T such that $f(x) = f'(x)$, then this means that there are two queries to which the challenger has responded inconsistently. Otherwise, if no such pair of polynomials exists, then all responses are consistent, and the view of the adversary is identical to Game 1.

Since all polynomials in T are linear, the probability over x that $f(x) = f'(x)$ is at most $1/q$ (since $f - f'$ has at most one root over F_q). Let B be the the "bad" event that the games are inconsistent. Therefore

$$\Pr_{x,\sigma}[B] = \Pr_{x,\sigma}[\bigcup_{\substack{f,f' \in T \\ f \neq f'}} f(x) = f'(x)] \leq \frac{(m+2)(m+1)}{2} \cdot 1/q\,.$$

Overall, the probability that the adversary wins in Game 1 is at most the probability that the adversary wins in Game 2 plus the probability that the games are different. More formally,

$$\begin{aligned}
\Pr_{x,\sigma}[\mathcal{A} \text{ wins in Game 1}] &= \Pr_{x,\sigma}[\mathcal{A} \text{ wins in Game 1} \wedge \neg B] + \Pr_{x,\sigma}[\mathcal{A} \text{ wins in Game 1} \wedge B] \\
&= \Pr_{x,\sigma}[\mathcal{A} \text{ wins in Game 2} \wedge \neg B] + \Pr_{x,\sigma}[\mathcal{A} \text{ wins in Game 1} \wedge B] \\
&\leq \Pr_{x,\sigma}[\mathcal{A} \text{ wins in Game 2}] + \Pr_{x,\sigma}[B] \\
&\leq 1/q + O(m^2/q)\,.
\end{aligned}$$

□

Non generic attacks

On the next homework assignment, you will see an example of a non-generic discrete-log algorithm that runs in subexponential time $\exp(O(\sqrt{\log q \log\log q}))$.

References

[1] D. Shanks, "Class number, a theory of factorization, and genera," in *Proc. of Symp. Math. Soc., 1971*, vol. 20, 1971, pp. 41–440.

[2] J. M. Pollard, "Monte Carlo methods for index computation mod p," *Mathematics of Computation*, vol. 32, pp. 918–924, 1978.

[3] V. Shoup, "Lower bounds for discrete logarithms and related problems." in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 1233. Springer, 1997, pp. 256–266.

[4] S. D. Galbraith, *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012. [Online]. Available: https://www.math.auckland.ac.nz/~sgal018/crypto-book/crypto-book.html

[5] A. Sutherland, "Lecture notes on elliptic curves," 2019, https://math.mit.edu/classes/18.783/2019/LectureNotes10.pdf.