Discrete logarithm problem II

Brute-force seach and BSGS

Tanja Lange

Eindhoven University of Technology

2MMC10 – Cryptology

Notation and target

Setting: cyclic group $G = \langle P \rangle$ with n elements;
group operation written additively on these slides.

Given $R \in G$ as well as P and how to compute in G.

Target: Compute $a = \log_P R$, i.e. a such that $R = aP$.
a is called the discrete logarithm of R to base P.
We typically take $0 \leq a < n$.

For elliptic curves, point counting over $\mathbf{F}_p$ runs in time polynomial in $\log p$.

Number of points in $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$.

The group is isomorphic to $\mathbf{Z}/m \times \mathbf{Z}/n$, where $m \mid n$ and $m \mid (p - 1)$.

Running example

$p = 1000003$, Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.

This curve has

Running example

$p = 1000003$, Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points.
$P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

Running example

$p = 1000003$, Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points.
$P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

Can we find an integer $a \in \{1, 2, 3, \ldots, 500001\}$ such that $aP = (670366, 740819)$?

This point was generated as a multiple of P.
Random point could also be outside cyclic group.

Could find a by brute force.

Running example

$p = 1000003$, Weierstrass curve $y^2 = x^3 - x$ over $\mathbf{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points.
$P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

Can we find an integer $a \in \{1, 2, 3, \ldots, 500001\}$ such that
$aP = (670366, 740819)$?

This point was generated as a multiple of P.
Random point could also be outside cyclic group.

Could find a by brute force.
$1P = (101384, 614510)$, $2P = (102361, 628914)$, $3P = (77571, 87643)$,
$4P = (650289, 31313)$, $\ldots$, $500001P = (101384, 385493) = -P$.
$500002P = \infty$.

At some point we'll find a with $aP = (670366, 740819)$.

Maximum cost of computation: ≤ 500001 additions of P;

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Computation has a good chance of finishing earlier.

Chance scales linearly:
$1/2$ chance of $n/2$ cost; $1/10$ chance of $n/10$ cost; etc.

"So users should choose large a."

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Computation has a good chance of finishing earlier.

Chance scales linearly:
$1/2$ chance of $n/2$ cost; $1/10$ chance of $n/10$ cost; etc.

"So users should choose large a."

Actually, no. We can apply "random self-reduction":
choose random r, compute rP, compute $(a+r)P = aP + rP$,
compute discrete log of this point.

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Computation has a good chance of finishing earlier.

Chance scales linearly:
$1/2$ chance of $n/2$ cost; $1/10$ chance of $n/10$ cost; etc.

"So users should choose large a."

Actually, no. We can apply "random self-reduction":
choose random r, compute rP, compute $(a+r)P = aP + rP$,
compute discrete log of this point. Obtain a from $a+r$ by subtracting r.

Example: $r = 69961$, compute $rP = (593450, 987590)$, compute
$(a+r)P = (670366, 740819) + (593450, 987590)$.

Smarter brute force attacks

This naive brute force attack takes up to n steps in general.

Computation has a good chance of finishing earlier.

Chance scales linearly:
$1/2$ chance of $n/2$ cost; $1/10$ chance of $n/10$ cost; etc.

"So users should choose large a."

Actually, no. We can apply "random self-reduction":
choose random r, compute rP, compute $(a+r)P = aP + rP$,
compute discrete log of this point. Obtain a from $a+r$ by subtracting r.

Example: $r = 69961$, compute $rP = (593450, 987590)$, compute
$(a+r)P = (670366, 740819) + (593450, 987590)$.

Choosing large, random r moves target around.

Many targets

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n - 1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Many targets

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n - 1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Many targets out of one

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n - 1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Can turn one DLP into 100 to benefit: compute $mP = \lfloor n/100 \rfloor P$.
New targets: $aP, (a+m)P, (a+2m)P, \ldots (a+99m)P$.

Many targets out of one

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n-1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Can turn one DLP into 100 to benefit: compute $mP = \lfloor n/100 \rfloor P$.
New targets: $aP, (a+m)P, (a+2m)P, \ldots (a+99m)P$.
Save a factor 100.

Many targets out of one

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n-1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Can turn one DLP into 100 to benefit: compute $mP = \lfloor n/100 \rfloor P$.
New targets: $aP, (a+m)P, (a+2m)P, \ldots (a+99m)P$.
Save a factor 100. Save factor 200, 300, 400, ...1000, 1100, 1200,

Many targets out of one

Computation can be applied to many targets at once.

Given 100 DL targets a_1P, a_2P, ..., $a_{100}P$:
Can find *all* of $a_1, a_2, \ldots, a_{100}$ with $\leq n-1$ ADDs.

Simplest approach:
First build a sorted table containing $a_1P, a_2P, \ldots, a_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems costs about as much as solving one.

Interesting consequence #2:
Solving *at least one* out of 100 DL problems is much faster than solving one specific DL problem.

First DL found after roughly $n/100$ ADDs.

Can turn one DLP into 100 to benefit: compute $mP = \lfloor n/100 \rfloor P$.
New targets: $aP, (a+m)P, (a+2m)P, \ldots (a+99m)P$.
Save a factor 100. Save factor 200, 300, 400, ...1000, ~~1100, 1200 ...~~
Cannot save more than $\sqrt{n}$, else computing targets dominates cost.

Baby-step giant-step algorithm

Systematize and optimize this approach.
Let $0 < m < n$ then there exist $0 \leq a_0 < m$ and $0 \leq a_1 \leq n/m + 1$ with

$$a = a_0 + ma_1.$$

Compute all possibilities for $a_0 P$. This costs m steps and m space.

We first do the small steps to get mP with just one addition.

$a = a_0 + ma_1$ means $aP = (a_0 + ma_1)P$, thus

$$a_0 P = aP - a_1 mP$$

Each step adds -mP (remember, $-(x, y) = (x, -y)$ on Weierstrass).

Baby-step giant-step algorithm

Systematize and optimize this approach.
Let $0 < m < n$ then there exist $0 \leq a_0 < m$ and $0 \leq a_1 \leq n/m + 1$ with

$$a = a_0 + ma_1.$$

Compute all possibilities for $a_0 P$. This costs m steps and m space.

We first do the small steps to get mP with just one addition.

$a = a_0 + ma_1$ means $aP = (a_0 + ma_1)P$, thus

$$a_0 P = aP - a_1 mP$$

Each step adds -mP (remember, $-(x, y) = (x, -y)$ on Weierstrass).
Finding this match takes at most $n/m + 1$ steps.

These targets are evenly spaced, so we are guaranteed a match.

We balance both for $m \sim \sqrt{n}$ to get to a total cost of $2\sqrt{n}+$ some small constant expenses, hence the cost of $O(\sqrt{n})$.

Note that this also costs $\sqrt{n}$ in storage.