CHAPTER 3

Discrete Logarithms

Chapter Goals

- To examine algorithms for solving the discrete logarithm problem.
- To introduce the Pohlig–Hellman algorithm.
- To introduce the Baby-Step/Giant-Step algorithm.
- To explain the methods of Pollard.
- To show how discrete logarithms can be solved in finite fields using algorithms like those used for factoring.
- To describe the known results on the elliptic curve discrete logarithm problem.

3.1. The DLP, DHP and DDH Problems

In Chapter 2 we examined the hard problem of FACTOR. This gave us some (hopefully) one-way functions, namely the RSA function, the squaring function modulo a composite and the function which multiplies two large numbers together. Another important class of problems are those based on the discrete logarithm problem or its variants. Let $(G, \cdot)$ be a finite abelian group of prime order q, such as a subgroup of the multiplicative group of a finite field or the set of points on an elliptic curve over a finite field (see Chapter 4). The discrete logarithm problem, or DLP, in G is: given $g, h \in G$, find an integer $x \in [0, \dots, q)$ (if it exists) such that

$$g^x = h.$$

We write $x = \text{dlog}_g(h)$. A diagram for the security game for the discrete logarithm problem is given in Figure 3.1, for a group G of prime order q. Just as for our factoring-based games we define an advantage function as the probability that the adversary wins the game in Figure 3.1 for a group G, i.e. $\text{Adv}_G^{\mathsf{DLP}}(A) = \Pr[A \text{ wins the } \mathsf{DLP} \text{ game in the group } G]$.

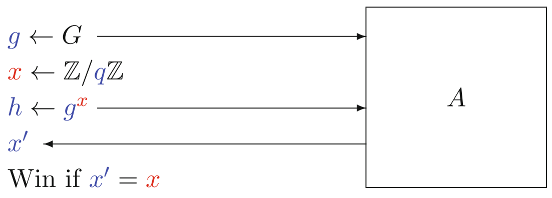

FIGURE 3.1. Security game to define the discrete logarithm problem

For some groups G this problem is easy. For example if we take G to be the integers modulo a number N under addition, then given $g, h \in \mathbb{Z}/N\mathbb{Z}$ we need to solve

$$x \cdot g = h.$$

We have already seen in Chapter 1 that we can easily tell whether such an equation has a solution, and determine its solution when it does, using the extended Euclidean algorithm.

For certain other groups determining discrete logarithms is believed to be hard. For example in the multiplicative group of a finite field the best known algorithm for this task is the Number Field Sieve/Function Field Sieve. The complexity of determining discrete logarithms in this case is given by

$$L_N(1/3, c)$$

for some constant c, depending on the type of the finite field, e.g. whether it is a large prime field or an extension field of small characteristic.

For other groups, such as elliptic curve groups, the discrete logarithm problem is believed to be even harder. The best known algorithm for finding discrete logarithms on a general elliptic curve defined over a finite field $\mathbb{F}_q$ is Pollard's Rho method, a fully exponential algorithm with complexity

$$\sqrt{q} = L_q(1, 1/2).$$

Since determining elliptic curve discrete logarithms is harder than in the case of multiplicative groups of finite fields we are able to use smaller groups. This leads to an advantage in key size. Elliptic curve cryptosystems often have much smaller key sizes (say 256 bits) compared with those based on factoring or discrete logarithms in finite fields (where for both the "equivalent" recommended key size is about 2048 bits).

Later in this chapter we survey the methods known for solving the discrete logarithm problem,

$$h = g^x$$

in various groups G. These algorithms fall into one of two categories: either the algorithms are generic and apply to any finite abelian group or the algorithms are specific to the special group under consideration.

Just as with the FACTOR problem, where we had a number of related problems, with discrete logarithms there are also related problems that we need to discuss. Suppose we are given a finite abelian group $(G, \cdot)$, of prime order q, and $g \in G$. The first of these is the Diffie–Hellman problem.

Definition 3.1 (Computational Diffie–Hellman Problem (DHP))**.** *Given $g \in G$, $a = g^x$ and $b = g^y$, for unknowns x and y chosen at random from $\mathbb{Z}/q\mathbb{Z}$, find c such that $c = g^{x \cdot y}$.*

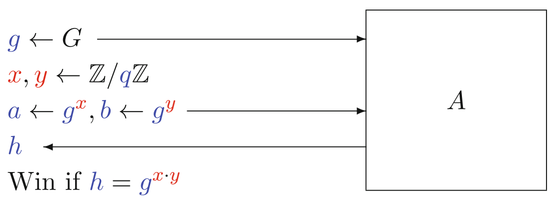
$g \leftarrow G$
$x, y \leftarrow \mathbb{Z}/q\mathbb{Z}$
$a \leftarrow g^x, b \leftarrow g^y$
h
Win if $h = g^{x \cdot y}$
A

FIGURE 3.2. Security game to define the Computational Diffie–Hellman problem

Diagrammatically we can represent the associated security game as in Figure 3.2, and we define the advantage of the adversary A in the game by

$$\mathrm{Adv}_G^{\mathsf{DHP}}(A) = \Pr[A \text{ wins the } \mathsf{DHP} \text{ game in the group } G].$$

We first show how to reduce solving the Diffie–Hellman problem to the discrete logarithm problem. But before doing so we note that in some groups there is a more complicated argument to show that the DHP is in fact equivalent to the DLP. This is done by producing a reduction in the other direction for the specific groups in question.

Lemma 3.2. *In an arbitrary finite abelian group G the DHP is no harder than the DLP. In particular for all algorithms A there is an algorithm B such that*

$$\mathrm{Adv}_G^{\mathsf{DLP}}(A) = \mathrm{Adv}_G^{\mathsf{DHP}}(B).$$

PROOF. Suppose we have an oracle/algorithm A which will solve the DLP, i.e. on input of $h = g^x$ it will return x. To solve the DHP on input of $a = g^x$ and $b = g^y$ we compute

(1) $z \leftarrow A(a)$.
(2) $c \leftarrow b^z$.
(3) Output c.

The above reduction clearly runs in polynomial time and will compute the true solution to the DHP, assuming algorithm A returns the correct value, i.e. $z = x$. Hence, the DHP is no harder than the DLP. □

There is a decisional version of the DHP problem, just like there is a decisional version QUADRES of the SQRROOT problem.

Definition 3.3 (Decision Diffie–Hellman problem (DDH))**.** *The adversary is given $g \in G$, $a = g^x$, $b = g^y$, and $c = g^z$, for unknowns x, y and z. The value z is chosen by the challenger to be equal to $x \cdot y$ with probability $1/2$, otherwise it is chosen at random. The goal of the adversary is to determine which case he thinks the challenger picked, i.e. he has to determine whether $z = x \cdot y$.*

Diagrammatically we can represent the associated security game as in Figure 3.3. But when defining the advantage for the DDH problem we need to be a bit careful, as the adversary can always win with probability one half, by just guessing the bit b at random. This is exactly the same situation as we had when looking at the QUADRES problem in Chapter 2. We hence define the advantage by

$$\mathrm{Adv}_G^{\mathsf{DDH}}(A) = 2 \cdot \left| \Pr[A \text{ wins the } \mathsf{DDH} \text{ game in } G] - \frac{1}{2} \right|.$$

Notice that with this definition, just as with the advantage in the QUADRES game, if the adversary just guesses the bit with probability $1/2$ then its advantage is zero as we would expect, since $2 \cdot |1/2 - 1/2| = 0$. If however the adversary is always right, or indeed always wrong, then the advantage is one, since $2 \cdot |1 - 1/2| = 2 \cdot |0 - 1/2| = 1$. Thus, the advantage is normalized to lie between zero and one, with one being always successful and zero being no better than random. Just like Lemma 2.3 we have that another way to write down the advantage is

$$\mathrm{Adv}_G^{\mathsf{DDH}}(A) = \Big| \Pr[b' = 1 | b = 1] - \Pr[b' = 1 | b = 0] \Big|.$$

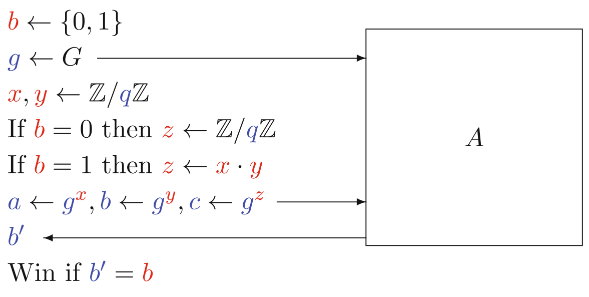
$b \leftarrow \{0,1\}$
$g \leftarrow G$
$x, y \leftarrow \mathbb{Z}/q\mathbb{Z}$
If $b = 0$ then $z \leftarrow \mathbb{Z}/q\mathbb{Z}$
If $b = 1$ then $z \leftarrow x \cdot y$
$a \leftarrow g^x, b \leftarrow g^y, c \leftarrow g^z$
b'
Win if $b' = b$
A

FIGURE 3.3. Security game to define the Decision Diffie–Hellman problem

We now show how to reduce the solution of the Decision Diffie–Hellman problem to the Computational Diffie–Hellman problem, and hence using our previous argument to the discrete logarithm problem.

Lemma 3.4. *In an arbitrary finite abelian group G the DDH is no harder than the DHP. In particular for all algorithms A there is an algorithm B such that*

$$\mathrm{Adv}_G^{\mathsf{DHP}}(A) = \mathrm{Adv}_G^{\mathsf{DDH}}(B).$$

PROOF. Suppose we have an oracle A which on input of g^x and g^y computes the value of $g^{x \cdot y}$. To solve the DDH on input of $a = g^x, b = g^y$ and $c = g^z$ we compute

(1) $d \leftarrow A(a, b)$.
(2) If $d = c$ output one.
(3) Else output zero.

Again the reduction clearly runs in polynomial time, and assuming the output of the oracle is correct then the above reduction will solve the DDH. □

So the Decision Diffie–Hellman problem is no harder than the Computational Diffie–Hellman problem. There are, however, some groups[1] in which we can solve the DDH in polynomial time but the fastest known algorithm to solve the DHP takes sub-exponential time. Hence, of our three discrete-logarithm-based problems, the easiest is DDH, then comes DHP and finally the hardest problem is DLP.

3.2. Pohlig–Hellman

The first observation to make is that the discrete logarithm problem in a group G is only as hard as the discrete logarithm problem in the largest subgroup of prime order in G. This observation is due to Pohlig and Hellman, and it applies in an arbitrary finite abelian group. To explain the Pohlig–Hellman algorithm, suppose we have a finite cyclic abelian group $G = \langle g \rangle$ whose order is given by

$$N = \#G = \prod_{i=1}^{t} p_i^{e_i}.$$

Now suppose we are given $h \in \langle g \rangle$, so there exists an integer x such that

$$h = g^x.$$

Our aim is to find x by first finding it modulo $p_i^{e_i}$ and then using the Chinese Remainder Theorem to recover it modulo N.

From basic group theory we know that there is a group isomorphism

$$\phi : G \longrightarrow C_{p_1^{e_1}} \times \cdots \times C_{p_t^{e_t}},$$

where C_{p^e} is a cyclic group of prime power order p^e. The projection of ϕ on the component C_{p^e} is given by

$$\phi_p : \begin{cases} G \longrightarrow C_{p^e} \\ f \longmapsto f^{N/p^e}. \end{cases}$$

The map ϕ_p is a group homomorphism, so if we have $h = g^x$ in G then we will have $\phi_p(h) = \phi_p(g)^x$ in C_{p^e}. But the discrete logarithm in C_{p^e} is only determined modulo p^e. So if we could solve the discrete logarithm problem in C_{p^e}, then we would determine x modulo p^e. Doing this for all primes p dividing N would allow us to solve for x using the Chinese Remainder Theorem. In summary suppose we have some oracle $O(g, h, p, e)$ which for $g, h \in C_{p^e}$ will output the discrete logarithm of h with respect to g. We can then solve for x using Algorithm 3.1.

[1]For example supersingular elliptic curves.

Algorithm 3.1: Algorithm to solve the DLP in a group of order N, given an oracle for DLP for prime power divisors of N

```
$S \leftarrow \{\}$.
for all primes $p$ dividing $N$ do
    Compute the largest $e$ such that $T = p^e$ divides $N$.
    $g_1 \leftarrow g^{N/T}$.
    $h_1 \leftarrow h^{N/T}$.
    $z \leftarrow O(g_1, h_1, p, e)$.
    $S \leftarrow S \cup \{(z, T)\}$.
$x \leftarrow$CRT$(S)$
```

The only problem is that we have not shown how to solve the discrete logarithm problem in C_{p^e}. We shall now show how this is done, by reducing to solving e discrete logarithm problems in the group C_p. Suppose $g, h \in C_{p^e}$ and there is an x such that

$$h = g^x.$$

Clearly x is only defined modulo p^e and we can write

$$x = x_0 + x_1 \cdot p + \cdots + x_{e-1} \cdot p^{e-1}.$$

We find $x_0, x_1, \ldots$ in turn, using the following inductive procedure. Suppose we know x', the value of x modulo p^t, i.e.

$$x' = x_0 + \cdots + x_{t-1} \cdot p^{t-1}.$$

We now wish to determine x_t and so compute x modulo p^{t+1}. We write

$$x = x' + p^t \cdot y,$$

so we have that

$$h = g^{x'} \cdot (g^{p^t})^y.$$

Hence, if we set

$$h_1 = h \cdot g^{-x'} \text{ and } g_1 = g^{p^t},$$

then

$$h_1 = {g_1}^y.$$

Now g_1 is an element of order p^{e-t}, so to obtain an element of order p and hence a discrete logarithm problem in C_p, we need to raise the above equation to the power $s = p^{e-t-1}$. So, setting

$$h_2 = h_1^s \text{ and } g_2 = g_1^s,$$

we obtain the discrete logarithm problem in C_p given by

$$h_2 = {g_2}^{x_t}.$$

So assuming we can solve discrete logarithms in C_p we can find x_t and so find x.

Pohlig–Hellman Example: We now illustrate this approach, assuming we can find discrete logarithms in cyclic groups of prime order. We leave to the next two sections how to do this; for now we assume that it is possible. As an example of the Pohlig–Hellman algorithm, consider the multiplicative group of the finite field $\mathbb{F}_{397}$. This group has order

$$N = 396 = 2^2 \cdot 3^2 \cdot 11$$

and a generator of $\mathbb{F}_{397}^*$ is given by

$$g = 5.$$

We wish to solve the discrete logarithm problem given by

$$h = 208 = 5^x \pmod{397}.$$

We first reduce to the three subgroups of prime power order, by raising the above equation to the power $396/p^e$, for each maximal prime power p^e which divides the order of the group 396. Hence, we obtain the three discrete logarithm problems

$$\begin{aligned} 334 = h^{396/4} &= g^{396/4 x_4} = 334^{x_4} \pmod{397}, \\ 286 = h^{396/9} &= g^{396/9 x_9} = 79^{x_9} \pmod{397}, \\ 273 = h^{396/11} &= g^{396/11 x_{11}} = 290^{x_{11}} \pmod{397}. \end{aligned}$$

The value of x_4 is the value of x modulo 4, the value of x_9 is the value of x modulo 9 whilst the value of x_{11} is the value of x modulo 11. Clearly if we can determine these three values then we can determine x modulo 396.

Determining x_4: By inspection we see that $x_4 = 1$, but let us labour the point and show how the above algorithm will determine this for us. We write

$$x_4 = x_{4,0} + 2 \cdot x_{4,1},$$

where $x_{4,0}, x_{4,1} \in \{0, 1\}$. Recall that we wish to solve

$$h_1 = 334 = 334^{x_4} = {g_1}^{x_4}.$$

We set $h_2 = h_1^2$ and $g_2 = g_1^2$ and solve the discrete logarithm problem

$$h_2 = {g_2}^{x_{4,0}}$$

in the cyclic group of order two. We find, using our oracle for the discrete logarithm problem in cyclic groups, that $x_{4,0} = 1$. So we now have

$$\frac{h_1}{g_1} = {g_2}^{x_{4,1}} \pmod{397}.$$

Hence we have $1 = 396^{x_{4,1}}$, which is another discrete logarithm in the cyclic group of order two. We find $x_{4,1} = 0$ and, as expected,

$$x_4 = x_{4,0} + 2 \cdot x_{4,1} = 1 + 2 \cdot 0 = 1.$$

Determining x_9: We write

$$x_9 = x_{9,0} + 3 \cdot x_{9,1},$$

where $x_{9,0}, x_{9,1} \in \{0, 1, 2\}$. Recall that we wish to solve

$$h_1 = 286 = 79^{x_9} = {g_1}^{x_9}.$$

We set $h_2 = h_1^3$ and $g_2 = g_1^3$ and solve the discrete logarithm problem

$$h_2 = 34 = {g_2}^{x_{9,0}} = 362^{x_{9,0}}$$

in the cyclic group of order three. We find, using our oracle for the discrete logarithm problem in cyclic groups, that $x_{9,0} = 2$. So we now have

$$\frac{h_1}{g_1^2} = {g_2}^{x_{9,1}} \pmod{397}.$$

Hence we have $1 = 362^{x_{9,1}}$, which is another discrete logarithm in the cyclic group of order three. We find $x_{9,1} = 0$ and so conclude that

$$x_9 = x_{9,0} + 3 \cdot x_{9,1} = 2 + 3 \cdot 0 = 2.$$

Determining x_{11}: We are already in a cyclic group of prime order, so applying our oracle to the discrete logarithm problem

$$273 = 290^{x_{11}} \pmod{397},$$

we find that $x_{11} = 6$.

Summary: So we have determined that if

$$208 = 5^{x} \pmod{397},$$

then x is given by

$$\begin{aligned} x &= 1 \pmod{4}, \\ x &= 2 \pmod{9}, \\ x &= 6 \pmod{11}. \end{aligned}$$

If we apply the Chinese Remainder Theorem to this set of three simultaneous equations, then we obtain that the solution to our discrete logarithm problem is given by $x = 281$.

3.3. Baby-Step/Giant-Step Method

In our above discussion of the Pohlig–Hellman algorithm we assumed we had an oracle to solve the discrete logarithm problem in cyclic groups of prime order. We shall now describe a general method to solve such problems due to Shanks called the Baby-Step/Giant-Step method. We stress that this is a generic method which applies to any cyclic finite abelian group.

Since the intermediate steps in the Pohlig–Hellman algorithm are quite simple, the difficulty of solving a general discrete logarithm problem will be dominated by the time required to solve the discrete logarithm problem in the cyclic subgroups of prime order. Hence, for generic groups the complexity of the Baby-Step/Giant-Step method will dominate the overall complexity of any algorithm. Indeed, one can show that the following method is the best possible method, time-wise, for solving the discrete logarithm problem in an arbitrary group. Of course in any actual group there may be a special purpose algorithm which works faster, but in general the following is provably the best one can do.

We fix notation as follows: We have a public cyclic group $G = \langle g \rangle$, which we can now assume to have prime order p. We are also given an $h \in G$ and are asked to find the value of x modulo p such that

$$h = g^{x}.$$

We assume there is some fixed encoding of the elements of G, so in particular it is easy to store, sort and search a list of elements of G.

The idea behind the Baby-Step/Giant-Step method is a standard divide-and-conquer approach found in many areas of computer science. We write

$$x = x_0 + x_1 \cdot \lceil \sqrt{p} \rceil.$$

Now, since $0 \le x \le p$, we have that $0 \le x_0, x_1 < \lceil \sqrt{p} \rceil$. We first compute the Baby-Steps

$$g_i \leftarrow g^{i} \text{ for } 0 \le i < \lceil \sqrt{p} \rceil.$$

The pairs

$$(g_i, i)$$

are stored in a table so that one can easily search for items indexed by the first entry in the pair. This can be accomplished by sorting the table on the first entry, or more efficiently by the use of hash tables. To compute and store the Baby-Steps clearly requires

$$O(\lceil \sqrt{p} \rceil)$$

time and a similar amount of storage.

We now compute the Giant-Steps $h_j \leftarrow h \cdot g^{-j \cdot \lceil \sqrt{p} \rceil}$, for $0 \leq j < \lceil \sqrt{p} \rceil$, and try to find a match in the table of Baby-Steps, i.e. we try to find a value g_i such that $g_i = h_j$. If such a match occurs we have

$$x_0 = i \text{ and } x_1 = j,$$

since, if $g_i = h_j$, we have

$$g^i = h \cdot g^{-j \cdot \lceil \sqrt{p} \rceil},$$

i.e.

$$g^{i + j \cdot \lceil \sqrt{p} \rceil} = h.$$

Notice that the time to compute the Giant-Steps is at most

$$O(\lceil \sqrt{p} \rceil).$$

Hence, the overall time and space complexity of the Baby-Step/Giant-Step method is

$$O(\sqrt{p}).$$

This means, combining with the Pohlig–Hellman algorithm, that if we wish a discrete logarithm problem in a group G to be as difficult as a work effort of 2^{128} operations, then we need the group G to have a prime order subgroup of size larger than 2^{256}.

Baby-Step/Giant-Step Example: As an example we take the subgroup of order 101 in the multiplicative group of the finite field $\mathbb{F}_{607}$, generated by $g = 64$. Suppose we are given the discrete logarithm problem

$$h = 182 = 64^x \pmod{607}.$$

We first compute the Baby-Steps

$$g_i = 64^i \pmod{607} \text{ for } 0 \leq i < \lceil \sqrt{101} \rceil = 11.$$

We compute

i	$64^i \pmod{607}$	i	$64^i \pmod{607}$
0	1	6	330
1	64	7	482
2	454	8	498
3	527	9	308
4	**343**	10	288
5	100		

Now we compute the Giant-Steps,

$$h_j = 182 \cdot 64^{-11 \cdot j} \pmod{607} \text{ for } 0 \leq j < 11,$$

and check when we obtain a Giant-Step which occurs in our table of Baby-Steps:

j	$182 \cdot 64^{-11 \cdot j} \pmod{607}$	j	$182 \cdot 64^{-11 \cdot j} \pmod{607}$
0	182	6	60
1	143	7	394
2	69	8	483
3	271	9	76
4	**343**	10	580
5	573		

So we obtain a match when $i = 4$ and $j = 4$, which means that

$$x = 4 + 11 \cdot 4 = 48,$$

which we can verify to be the correct answer to the earlier discrete logarithm problem by computing $64^{48} \pmod{607} = 182$.

3.4. Pollard-Type Methods

The trouble with the Baby-Step/Giant-Step method is that, although its run time is bounded by $O(\sqrt{p})$, it required $O(\sqrt{p})$ space. In practice this space requirement is more of a hindrance than the time requirement. Hence, one could ask whether one could trade the large space requirement for a smaller space requirement, but still obtain a time complexity of $O(\sqrt{p})$? Well we can, but we will now obtain only an expected running time rather than an absolute bound on the running time; thus technically we obtain a Las Vegas-style algorithm as opposed to a deterministic one. There are a number of algorithms which achieve this reduced space requirement all of which are due to ideas of Pollard.

3.4.1. Pollard's Rho Algorithm: Suppose $f : S \to S$ is a random mapping between a set S and itself, where the size of S is n. Now pick a random value $x_0 \in S$ and compute

$$x_{i+1} \leftarrow f(x_i) \text{ for } i \geq 0.$$

We consider the values $x_0, x_1, x_2, \ldots$ as a deterministic random walk. By this statement we mean that each step $x_{i+1} = f(x_i)$ of the walk is a deterministic function of the current position x_i, but we are assuming that the sequence $x_0, x_1, x_2, \ldots$ *behaves* as a random sequence would. Another name for a deterministic random walk is a pseudo-random walk.

The goal of many of Pollard's algorithms is to find a collision in a random mapping like the one above, where a collision is finding a pair of values x_i and x_j with $i \neq j$ such that

$$x_i = x_j.$$

From the birthday paradox from Section 1.4.2, we obtain a collision after an expected number of

$$\sqrt{\pi \cdot n/2}$$

iterations of the map f. Hence, finding a collision using the birthday paradox in a naive way would require $O(\sqrt{n})$ time and $O(\sqrt{n})$ memory. But this large memory requirement is exactly the problem with the Baby-Step/Giant-Step method we were trying to avoid.

But, since S if finite, we must eventually obtain $x_i = x_j$ for *some* values of i and j, and so

$$x_{i+1} = f(x_i) = f(x_j) = x_{j+1}.$$

Hence, the sequence $x_0, x_1, x_2, \ldots,$ will eventually become cyclic. If we "draw" such a sequence then it looks like the Greek letter rho, ρ. In other words there is a cyclic part and an initial tail. It can be shown, using much the same reasoning as for the birthday bound above, that for a random mapping, the tail has expected length $\sqrt{\pi \cdot n/8}$, whilst the cycle also has expected length $\sqrt{\pi \cdot n/8}$. It is this observation which will allow us to reduce the memory requirement to constant space.

To find a collision and make use of the rho shape of the random walk, we use a technique called Floyd's cycle-finding algorithm: Given (x_1, x_2) we compute (x_2, x_4) and then (x_3, x_6) and so on, i.e. given the pair (x_i, x_{2i}) we compute

$$(x_{i+1}, x_{2i+2}) = (f(x_i), f(f(x_{2i}))).$$

We stop when we find

$$x_m = x_{2m}.$$

If the tail of the sequence $x_0, x_1, x_2, \ldots$ has length λ and the cycle has length μ then it can be shown that we obtain such a value of m when

$$m = \mu \cdot (1 + \lfloor \lambda/\mu \rfloor).$$

Since $\lambda < m \le \lambda + \mu$ we see that

$$m = O(\sqrt{n}),$$

and this will be an accurate complexity estimate if the mapping f behaves suitably like a random function. Hence, we can detect a collision with virtually no storage.

This is all very well, but we have not shown how to relate this to the discrete logarithm problem. Let G denote a group of order n and let the discrete logarithm problem be given by

$$h = g^x.$$

We partition the group into three sets S_1, S_2, S_3, where we assume $1 \notin S_2$, and then define the following random walk on the group G,

$$x_{i+1} \leftarrow f(x_i) = \begin{cases} h \cdot x_i & x_i \in S_1, \\ x_i^2 & x_i \in S_2, \\ g \cdot x_i & x_i \in S_3. \end{cases}$$

The condition that $1 \notin S_2$ is to ensure that the function f has no stationary points. In practice we actually keep track of three pieces of information

$$(x_i, a_i, b_i) \in G \times \mathbb{Z} \times \mathbb{Z}$$

where

$$a_{i+1} \leftarrow \begin{cases} a_i & & x_i \in S_1, \\ 2 \cdot a_i \pmod{n} & & x_i \in S_2, \\ a_i + 1 \pmod{n} & & x_i \in S_3, \end{cases}$$

and

$$b_{i+1} \leftarrow \begin{cases} b_i + 1 \pmod{n} & & x_i \in S_1, \\ 2 \cdot b_i \pmod{n} & & x_i \in S_2, \\ b_i & & x_i \in S_3. \end{cases}$$

If we start with the triple

$$(x_0, a_0, b_0) = (1, 0, 0)$$

then we have, for all i,

$$\log_g(x_i) = a_i + b_i \cdot \log_g(h) = a_i + b_i \cdot x.$$

Applying Floyd's cycle-finding algorithm we obtain a collision, and so find a value of m such that

$$x_m = x_{2m}.$$

This leads us to deduce the following equality of discrete logarithms

$$\begin{aligned} a_m + b_m \cdot x &= a_m + b_m \cdot \log_g(h) \\ &= \log_g(x_m) \\ &= \log_g(x_{2m}) \\ &= a_{2m} + b_{2m} \cdot \log_g(h) \\ &= a_{2m} + b_{2m} \cdot x. \end{aligned}$$

Rearranging we see that

$$(b_m - b_{2m}) \cdot x = a_{2m} - a_m,$$

and so, if $b_m \neq b_{2m}$, we obtain

$$x = \frac{a_{2m} - a_m}{b_m - b_{2m}} \pmod{n}.$$

The probability that we have $b_m = b_{2m}$ is small enough to be ignored for large n. Thus the above algorithm will find the discrete logarithm in expected time $O(\sqrt{n})$.

Pollard's Rho Example: As an example consider the subgroup G of $\mathbb{F}_{607}^*$ of order $n = 101$ generated by the element $g = 64$ and the discrete logarithm problem

$$h = 122 = 64^x.$$

We define the sets S_1, S_2, S_3 as follows:

$$\begin{aligned} S_1 &= \{x \in \mathbb{F}_{607}^* : x \leq 201\}, \\ S_2 &= \{x \in \mathbb{F}_{607}^* : 202 \leq x \leq 403\}, \\ S_3 &= \{x \in \mathbb{F}_{607}^* : 404 \leq x \leq 606\}. \end{aligned}$$

Applying Pollard's Rho method we obtain the following data

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
0	1	0	0	1	0	0
1	122	0	1	316	0	2
2	316	0	2	172	0	8
3	308	0	4	137	0	18
4	172	0	8	7	0	38
5	346	0	9	309	0	78
6	137	0	18	352	0	56
7	325	0	19	167	0	12
8	7	0	38	498	0	26
9	247	0	39	172	2	52
10	309	0	78	137	4	5
11	182	0	55	7	8	12
12	352	0	56	309	16	26
13	76	0	11	352	32	53
14	**167**	0	12	**167**	64	6

So we obtain a collision, using Floyd's cycle-finding algorithm, when $m = 14$. We see that

$$g^0 \cdot h^{12} = g^{64} \cdot h^6$$

which implies

$$12 \cdot x = 64 + 6 \cdot x \pmod{101}.$$

Consequently

$$x = \frac{64}{12 - 6} \pmod{101} = 78.$$

3.4.2. Pollard's Lambda Method: Pollard's Lambda method is like the Rho method, in that we use a deterministic random walk and a small amount of storage to solve the discrete logarithm problem. However, the Lambda method is particularly tuned to the situation where we know that the discrete logarithm lies in a certain interval

$$x \in [a, \dots, b].$$

In the Rho method we used one random walk, which turned into the shape of the Greek letter ρ, whilst in the Lambda method we use two walks which end up in the shape of the Greek letter lambda, i.e. λ, hence giving the method its name. Another name for this method is Pollard's Kangaroo method as it was originally described with the two walks being performed by kangaroos.

Let $w = b - a$ denote the length of the interval in which the discrete logarithm x is known to lie. We define a set

$$S = \{s_0, \dots, s_{k-1}\}$$

of integers in non-decreasing order. The mean m of the set should be around $N = \lfloor \sqrt{w} \rfloor$. It is common to choose

$$s_i = 2^i \text{ for } 0 \le i < k,$$

which implies that the mean of the set is

$$\frac{2^k}{k},$$

and so we choose

$$k \approx \frac{1}{2} \cdot \log_2(w).$$

We divide the group up into k sets S_i, for $i = 0, \dots, k-1$, and define the following deterministic random walk:

$$x_{i+1} = x_i \cdot g^{s_j} \text{ if } x_i \in S_j.$$

We first compute the deterministic random walk starting from the end of the interval $g_0 = g^b$ by setting

$$g_{i+1} = g_i \cdot g^{s_j} \text{ if } g_i \in S_j,$$

for $i = 1, \dots, N = \lfloor \sqrt{w} \rfloor$. We also set $c_0 = b$ and $c_{i+1} = c_i + s_j \pmod q$. We store g_N and notice that we have computed the discrete logarithm of g_N with respect to g,

$$c_N = \log_g(g_N).$$

We now compute our second deterministic random walk, starting from the point in the interval corresponding to the unknown x; we set $h_0 = h = g^x$ and compute

$$h_{i+1} = h_i \cdot g^{s_j} \text{ if } h_i \in S_j.$$

We also set $d_0 = 0$ and $d_{i+1} = d_i + s_j \pmod q$. Notice that we have

$$\log_g(h_i) = x + d_i.$$

If the path of the h_i ever meets that of the path of the g_i then the h_i will carry on the path of the g_i, and so eventually reach the point g_N. Thus, we are able to find a value M where h_M equals our stored point g_N. We then have

$$c_N = \log_g(g_N) = \log_g(h_M) = x + d_M,$$

and so the solution to our discrete logarithm problem is given by

$$x = c_N - d_M \pmod q.$$

If we do not get a collision then we can increase N and continue both walks in a similar manner until a collision does occur.

The expected running time of this method is $\sqrt{w}$ and again the storage can be seen to be constant. The Lambda method can be used when the discrete logarithm is only known to lie in the full interval $[0, \dots, q-1]$. But in this situation, whilst the asymptotic complexity is the same as the Rho method, the Rho method is better due to the implied constants.

Pollard's Lambda Example: As an example we again consider the subgroup G of $\mathbb{F}_{607}^*$ of order $n = 101$ generated by the element $g = 64$, but now we look at the discrete logarithm problem

$$h = 524 = 64^x.$$

We are given that the discrete logarithm x lies in the interval $[60, \dots, 80]$. As our set of multipliers s_i we take $s_i = 2^i$ for $i = 0, 1, 2, 3$. The subsets $S_0, \dots, S_3$ of G we define by

$$S_i = \{g \in G : g \pmod 4 = i\}.$$

We first compute the deterministic random walk g_i and the discrete logarithms $c_i = \log_g(g_i)$, for $i = 0, \dots, N = \lfloor\sqrt{80-40}\rfloor = 4$.

i	g_i	c_i
0	151	80
1	537	88
2	391	90
3	478	98
4	64	1

Now we compute the second deterministic random walk

i	h_i	$d_i = \log_g(h_i) - x$
0	524	0
1	151	1
2	537	9
3	391	11
4	478	19
5	64	23

Hence, we obtain the collision $h_5 = g_4$ and so

$$x = 1 - 23 \pmod{101} = 79.$$

Note that examining the above tables we see that we had earlier collisions between our two walks. However, we are unable to use these since we do not store g_0, g_1, g_2 or g_3. We have only stored the value of g_4.

3.4.3. Parallel Pollard's Rho: In real life when we use random-walk-based techniques to solve discrete logarithm problems we use a parallel version, to exploit the computing resources of a number of sites across the Internet. Suppose we are given the discrete logarithm problem

$$h = g^x$$

in a group G of prime order q. We first decide on an easily computable function

$$H : G \longrightarrow \{1, \dots, k\},$$

where k is usually around 20. Then we define a set of multipliers m_i. These are produced by generating random integers $a_i, b_i \in [0, \dots, q-1]$ and then setting

$$m_i = g^{a_i} \cdot h^{b_i}.$$

To start a deterministic random walk we pick random $s_0, t_0 \in [0, \dots, q-1]$ and compute

$$g_0 = g^{s_0} \cdot h^{t_0},$$

the deterministic random walk is then defined on the triples (g_i, s_i, t_i) where

$$\begin{aligned} g_{i+1} &= g_i \cdot m_{H(g_i)}, \\ s_{i+1} &= s_i + a_{H(g_i)} \pmod{q}, \\ t_{i+1} &= t_i + b_{H(g_i)} \pmod{q}. \end{aligned}$$

Hence, for every g_i we record the values of s_i and t_i such that

$$g_i = g^{s_i} \cdot h^{t_i}.$$

Suppose we have m processors: each processor starts a different deterministic random walk from a different starting position using the same algorithm to determine the next element in the walk. When another processor, or even the same processor, meets an element of the group that has been seen before then we obtain an equation

$$g^{s_i} \cdot h^{t_i} = g^{s'_j} \cdot h^{t'_j}$$

which we can solve for the discrete logarithm x. Hence, we expect that after $O(\sqrt{(\pi \cdot q)/(2 \cdot m^2)})$ iterations of these parallel walks we will find a collision and so solve the discrete logarithm problem.

However, as described this means that each processor needs to return every element in its computed deterministic random walk to a central server which then stores all the computed elements. This is highly inefficient as the storage requirements will be very large, namely $O(\sqrt{(\pi \cdot q)/2})$. We can reduce the storage to any required value as follows: We first define a function d on the group

$$d : G \longrightarrow \{0, 1\}$$

such that $d(g) = 1$ around $1/2^t$ of the time for any $g \in G$. The function d is often defined by returning $d(g) = 1$ if a certain subset of t of the bits representing the group element $g \in G$ are set to zero for example. The elements in G for which $d(g) = 1$ will be called distinguished.

It is only the distinguished group elements that are now transmitted back to the central server. This means that we expect the deterministic random walks to need to continue another 2^t steps before a collision is detected between two deterministic random walks. Hence, the computing time now becomes

$$O\left(\sqrt{(\pi \cdot q)/(2 \cdot m^2)}/ + 2^t\right),$$

whilst the storage becomes

$$O\left(\sqrt{(\pi \cdot q)/2^{2 \cdot t+1}}\right).$$

This allows the storage to be reduced to any manageable amount, at the expense of a little extra computation. We do not give an example, since the method only really becomes useful as q becomes large (say $q > 2^{20}$).

3.5. Sub-exponential Methods for Finite Fields

There is a close relationship between the sub-exponential methods for factoring and the sub-exponential methods for solving the discrete logarithm problem in finite fields. We shall only consider the case of prime fields $\mathbb{F}_p$ but similar considerations apply to finite fields of small characteristic; here we use a special algorithm called the Function Field Sieve. The sub-exponential algorithms for finite fields are often referred to as index-calculus algorithms, as an index is an old name for a discrete logarithm.

We assume we are given $g, h \in \mathbb{F}_p^*$ such that

$$h = g^x.$$

We choose a factorbase F of elements, usually small prime numbers, and then, using one of the sieving strategies used for factoring, we obtain a large number of relations of the form

$$\prod_{p_i \in F} p_i^{e_i} = 1 \pmod{p}.$$

These relations translate into the following equations for discrete logarithms,

$$\sum_{p_i \in F} e_i \cdot \log_g(p_i) = 0 \pmod{p-1}.$$

Once enough equations like the one above have been found we can solve for the discrete logarithm of every element in the factorbase, i.e. we can determine

$$x_i = \log_g(p_i).$$

The value of x_i is sometimes called the index of p_i with respect to g. This calculation is performed using linear algebra modulo $p-1$, which is more complicated than the linear algebra modulo two performed in factoring algorithms. However similar tricks, to those deployed in the linear algebra stage of factoring algorithms can be deployed to keep storage requirements down to manageable levels.

This linear algebra calculation only needs to be done once for each generator g, and the results can then be used for many values of h. When we wish to solve a particular discrete logarithm problem $h = g^x$, we use a sieving technique, or simple trial and error, to write

$$h = \prod_{p_i \in F} p_i^{h_i} \pmod{p},$$

e.g. we could compute

$$T = h \cdot \prod_{p_i \in F} p_i^{f_i} \pmod{p}$$

and see whether it factors in the form

$$T = \prod_{p_i \in F} p_i^{g_i}.$$

If it does then we have

$$h = \prod_{p_i \in F} p_i{}^{g_i - f_i} \pmod{p}.$$

We can then compute the discrete logarithm x from

$$\begin{aligned} x = \log_g(h) &= \log_g\left(\prod_{p_i \in F} p_i^{h_i}\right) \\ &= \sum_{p_i \in F} h_i \cdot \log_g(p_i) \pmod{p-1} \\ &= \sum_{p_i \in F} h_i \cdot x_i \pmod{p-1}. \end{aligned}$$

This means that, once one discrete logarithm has been found, determining the next one is easier since we have already computed the values of the x_i.

The best of the methods to find the relations between the factorbase elements is the Number Field Sieve. This gives an overall running time of $O(L_p(1/3, c))$ for some constant c. This is roughly the same complexity as the algorithms to factor large numbers, although the real practical problem is that the matrix algorithms now need to work modulo $p-1$ and not modulo 2 as they did in the factoring algorithms.

The upshot of these sub-exponential methods is that the size of p for finite field discrete-logarithm-based systems needs to be of the same order of magnitude as a factoring modulus, i.e. $p \geq 2^{2048}$. Even though p has to be very large we still need to guard against generic attacks, hence $p-1$ should have a prime factor q of order greater than 2^{256}. In fact, for finite-field-based systems we usually work in the subgroup of $\mathbb{F}_p^*$ of order q.

In 2013, Antoine Joux and others showed that discrete logarithms in finite fields of characteristic two can be determined in quasi-polynomial time. This result is striking, but the methods used do not seem to generalize to higher-characteristic fields. This confirms the long-standing belief that discrete-logarithm-based cryptographic systems in fields of low characteristic should be avoided.

Chapter Summary

- We covered the DLP, DHP and DDH problems and the relationships between them.
- Due to the Pohlig–Hellman algorithm a hard discrete logarithm problem should be set in a group where the order has a large prime factor.
- Generic algorithms such as the Baby-Step/Giant-Step algorithm mean that to achieve the same security as a 128-bit block cipher, the size of the large prime factor of the group order should be at least 256 bits.
- The Baby-Step/Giant-Step algorithm is a generic algorithm and its running time can be absolutely bounded by $O(\sqrt{q})$, where q is the size of the large prime factor of $\#G$. However, the storage requirements of the Baby-Step/Giant-Step algorithm are also $O(\sqrt{q})$.
- There are a number of techniques, due to Pollard, based on deterministic random walks in a group. These are generic algorithms which require little storage but which solve the discrete logarithm problem in expected time $O(\sqrt{q})$.
- For finite fields a number of index calculus algorithms exist which run in sub-exponential time. These mean that one needs to take large finite fields $\mathbb{F}_{p^t}$ with $p^t \geq 2^{2048}$ to obtain a hard discrete logarithm problem. Due to the new quasi-polynomial-time attacks on low-characteristic fields we should select the characteristic to not be "too small".

Further Reading

There are a number of good surveys on the discrete logarithm problem. I would recommend the ones by McCurley and Odlyzko. For a more modern perspective see the article by Odlyzko, Pierrot and Joux.

K. McCurley. *The discrete logarithm problem.* In *Cryptology and Computational Number Theory*, Proc. Symposia in Applied Maths, Volume 42, 47–94, AMS, 1990.

A. Odlyzko. *Discrete logarithms: The past and the future.* Designs, Codes and Cryptography, **19**, 129–145, 2000.

A. Odlyzko, C. Pierrot and A. Joux. *The past, evolving present, and future of the discrete logarithm.* In *Open Problems in Mathematics and Computational Science*, Springer, 2014.