Random Walks Revisited: Extensions of Pollard's Rho Algorithm For Computing Multiple Discrete Logarithms

Fabian Kuhn[1] and René Struik[2]

[1] Departement Informatik, ETH Zentrum, CH-8092 Zürich, Switzerland, fkuhn@iiic.ethz.ch
[2] Certicom Research, 5520 Explorer Drive, 4th Floor, Mississauga, Ontario, Canada L4W 5L1, rstruik@certicom.com

Abstract. This paper extends the analysis of Pollard's rho algorithm for solving a single instance of the discrete logarithm problem in a finite cyclic group G to the case of solving more than one instance of the discrete logarithm problem in the same group G. We analyze Pollard's rho algorithm when used to iteratively solve *all* the instances. We also analyze the situation when the goal is to solve *any one* of the multiple instances using any DLP algorithm.

1 Introduction

The security of many public-key cryptographic systems is based on the discrete logarithm problem (DLP). Examples are the Diffie-Hellman key agreement protocol and the ElGamal encryption and signature schemes.

The DLP can be defined as follows: Let g be a generator of a finite cyclic group $G = \langle g \rangle$ of order N. For the general DLP, we have to find an integer x $(0 \leq x < N)$ such that $g^x = h$, where h is chosen uniformly at random from G (written $h \in_R G$). The integer x is called the discrete logarithm of h to the base g, denoted $\log_g h$. If N is composite, one can compute $x \bmod p^k$ in the subgroup of order p^k for each prime power p^k dividing N. Then, one can compute x by application of the Chinese Remainder Theorem. Further, calculating the discrete logarithm in the subgroup of order p^k can be reduced to finding the discrete logarithm in the group of prime order p (see [7]). For these reasons, we only consider the DLP in groups of prime order N.

Shoup [10] gave a lower bound for the running time for computing discrete logarithms by generic algorithms (probabilistic or deterministic) in groups of prime order. The time needed to solve the DLP with a non-negligible probability is $c\sqrt{N}$ group operations for some constant c. The best algorithm known for solving the general DLP is Pollard's rho algorithm [8]. It does not only match Shoup's lower bound, but also needs very little memory and is parallelizable with a linear speed-up (see [6]). For many groups of cryptographic interest, such as the multiplicative group of a finite field (see [1]), and the Jacobians of hyperelliptic curves of high genus (see [2]), there are subexponential-time algorithms

known for the DLP that are more efficient than Pollard's rho algorithm. However, Pollard's rho algorithm is the best algorithm known for solving the DLP in some groups such as the group of points on an elliptic curve, and the Jacobian of genus 2 and 3 hyperelliptic curves. Thus, the results in this paper are particularly relevant to the DLP in elliptic curve groups and in genus 2 and 3 hyperelliptic curves.

This paper extends the analysis of Pollard's rho algorithm for solving a single instance of the discrete logarithm problem in a finite cyclic group G to the case of solving more than one instance of the discrete logarithm problem in the same group G. Pollard's rho algorithm is reviewed in §2. In §3, we provide a runtime analysis in an idealized model and do an exact analysis of possible time-memory trade-offs for the parallelized version. When using Pollard's rho algorithm to iteratively solve *all* n instances of the DLP in the same group, the data that is gathered during the calculation of a single discrete logarithms can be used to compute subsequent discrete logarithms. Thus, the additional time needed for every new DLP may be smaller than the time needed to solve the one before. A careful analysis for this case is provided in §4. In §5 we consider the case where the goal is to solve *any one* of a set of n DLPs in the same group using any DLP algorithm.

2 Pollard's Rho Algorithm

2.1 Basic Idea

Pollard's rho algorithm is based on the birthday paradox. If we randomly choose elements (with replacement) from a set of N numbered elements, we only need to choose about $\sqrt{N}$ elements until we get one element twice (called a *collision*). This can be applied to find discrete logarithms as follows. By choosing $a, b \in_R [0, N-1]$, one obtains a random group element $g^a h^b$. Such group elements are randomly selected until we get a group element twice. If $g^{a_i} h^{b_i}$ and $g^{a_j} h^{b_j}$ represent the same group element then $a_i + b_i x \equiv a_j + b_j x \pmod{N}$, whence

$$x = (a_j - a_i)(b_i - b_j)^{-1} \bmod N \quad \text{for } b_i \not\equiv b_j \pmod{N}. \tag{1}$$

Let T be the random variable describing the number of group elements chosen until the first collision occurs. We denote the probability that $T > k$ by p_k. We have

$$p_k = 1\left(1 - \frac{1}{N}\right)\left(1 - \frac{2}{N}\right)\cdots\left(1 - \frac{k-1}{N}\right) \approx \left(1 - \frac{k-1}{2N}\right)^k \approx e^{-\frac{k^2}{2N}}. \tag{2}$$

For $k \in \mathrm{O}(\sqrt{N})$, the relative error of the above approximation is $\mathrm{O}(N^{-1/2})$. As shown in Appendix B, the expected value of T is $\mathrm{E}(T) \approx \sqrt{\pi N/2}$. The first collision can be found by simply storing all the randomly selected group elements until a repeat is detected. However, this simple-minded method has an expected storage requirement of $\sqrt{\pi N/2}$ group elements.

2.2 The Single Processor Case

The question now is how to detect a collision without having to store $\sqrt{\pi N/2}$ group elements. In Pollard's rho algorithm, this is done by means of a random function[1] $f : G \to G$. For actual implementations, f is chosen such that it approximates a random function as closely as possible. Further, it should be calculated with a single group multiplication and map an element $g^a h^b$ to an element $g^c h^d$ so that c and d can easily be computed from a and b. The originally suggested function by Pollard (for $\mathbb{Z}_p^*$) can be generalized towards arbitrary cyclic groups as

$$f(x) = \begin{cases} hx \text{ if } x \in \mathcal{S}_1; \\ x^2 \text{ if } x \in \mathcal{S}_2; \\ gx \text{ if } x \in \mathcal{S}_3. \end{cases}$$

Here, $\mathcal{S}_1, \mathcal{S}_2$ and $\mathcal{S}_3$ are three sets of roughly the same size which form a partition of G. In [12, 13], Teske shows that this function is not random enough and gives a better function:

$$f(x) = x \cdot g^{m_s} h^{n_s}, \text{ if } x \in \mathcal{M}_s \quad \text{for } s \in \{1, \ldots, r\} \text{ and } r \approx 20.$$

Here again, the $\mathcal{M}_s$ are of roughly the same size and form a partition of G. But this time, G is partitioned into more than three subsets. For both functions, it is of course necessary that determining the subset $\mathcal{M}_i$, resp. $\mathcal{S}_i$, to which a group element belongs is very efficient.

By starting at a random point $g^{a_0} h^{b_0}$ and iteratively applying a random function, random points $g^{a_i} h^{b_i}$ are generated. Because the group is finite, we eventually arrive at a point for the second time. The sequence of subsequent points then cycle forever. From §2.1 we know that the first repeat happens after an expected $\mathrm{E}(T) \approx \sqrt{\pi N/2}$ function applications. With very little time and space overhead, it is possible to detect such a cycle with Floyd's cycle-finding algorithm (or with an improved variant by Brent [3]).

2.3 Parallelization of Pollard's Rho Algorithm

Unfortunately, iteratively applying a function is an inherently serial process and cannot efficiently be parallelized. If m processors run the Pollard-rho algorithm as described above, the speed-up when compared to the single processor case is only about $\sqrt{m}$. For, if the processors run the algorithm individually, the probability that none of them has found a collision after k steps is $p_k^m \approx e^{-k^2 m/(2N)}$. This leads to an expected time of $(\sqrt{\pi N/2})/\sqrt{m}$ for finding the first collision. If, however, the processors communicate with each other, we can do better. If we could detect and use any collision which occurs between two processors, the speed-up to the single processor case would be a factor m because m processors calculate m times as many points as a single processor does.

[1] A random function is a function that is chosen uniformly at random from the set of all functions $f : G \to G$.

In [6], Wiener and van Oorschot presented a very elegant way of parallelizing Pollard's rho alorithm which is based on distinguished points. A distinguished point is a group element with an easy testable property. An often used distinguishing property is whether a point's binary representation has a certain number of leading zeros. Each processor starts the iteration at a different random element (but all have the same iteration function). As soon as the iteration hits a distinguished point, this point will be sent to a central server and the processor starts a new iteration. The server stores all collected points (a_i, b_i and $y_i = g^{a_i} h^{b_i}$) in a hash table. As soon as the server has received the same point twice, it has two representations $g^{a_i} h^{b_i}$ and $g^{a_j} h^{b_j}$ for a group element and can calculate the discrete logarithm x of h as given in (1).

As soon as a point occurs in two iterations, the remainder of those two iteration trails will be the same and thus lead to the same distinguished point. Therefore, by performing the iterations, all processors calculate random group elements of the form $g^a h^b$ and as soon as the same element has been calculated twice, we are going to get the same distinguished point twice, as well. If the two representations of the point, where the trails collided, are different, the representations of this distinguished point are different too, and we are therefore able to calculate x.

3 Analysis of the Parallelized Pollard's Rho Algorithm

For our analysis, we make the following assumptions (cf. §3.3):

1. The iterative function really behaves like a random mapping and thus generates uniformly distributed random group elements.
2. All collisions are useful, i.e. the collision reveals two representations $g^{a_i} h^{b_i}$ and $g^{a_j} h^{b_j}$ of a group element with $b_i \not\equiv b_j \pmod{N}$.
3. All trails lead to distinguished points (i.e., we neglect the existence of iteration paths which eventually run into a cycle that does not contain a distinguished point).

We denote the number of processors by m. The proportion of the points that constitute distinguished points is called θ (i.e., there are θN distinguished points). Additionally, for the analysis, we assume that all processors operate at the same speed.

3.1 Running Time

The runtime of the Pollard-rho algorithm can be divided into two statistically independent phases. First, all processors have to calculate points until a collision occurs. We already know that an expected $\sqrt{\pi N/2}$ points must be calculated for this part of the algorithm. Because all m processors calculate their points independently, the expected time for this part is $\sqrt{\pi N/2}/m$ function iterations.

After a collision, the iteration has to be continued until it arrives at a distinguished point. Because for each function application, the probability to come to

a distinguished point is θ, the number of steps from a collision to its detection is a geometrically distributed random variable with expected value $1/\theta$.

The iteration function is such that the time for one function application is equal to the time of one group operation plus a negligible overhead. Thus, the overall expected value for the running time of the parallel Pollard-rho algorithm is $\mathrm{E}(T) = (\sqrt{\pi N/2})/m + 1/\theta$ group operations.

3.2 Memory Requirements

Essentially, the only memory needed for the parallel version of Pollard's rho algorithm is that for storing the distinguished points on the server[2]. For every iteration, the server has to store one distinguished point. The length of a trail portion between distinguished points is geometrically distributed with parameter θ. Therefore, the expected length of such an iteration trail is $\frac{1}{\theta}$. This means that for the whole duration of the algorithm, all processors will send a distinguished point to the server every $\frac{1}{\theta}$ group operations on average. Therefore, because the time until a collision occurs and the average length of the trails are assumed to be statistically independent, the expected space needed on the server is $\mathrm{E}(S) = m\theta\mathrm{E}(T) = \theta\sqrt{\pi N/2} + m$ distinguished points. Note that for each distinguished point, we have to store the group element $g^a h^b$ and the integers a and b. Therefore, the actual space needed to store one distinguished point is $\mathrm{O}(\log N)$ bits.

For the memory requirements to be as small as possible, we have to choose θ as small as possible. But of course, if θ gets smaller, the time overhead $\frac{1}{\theta}$ to detect a collision gets bigger. In order to keep the overall running time in $\mathrm{O}(\sqrt{N}/m)$, we have to choose θ in $\mathrm{O}(m/\sqrt{N})$. Therefore, we choose θ as $\theta = \alpha m/(\sqrt{\pi N/2})$. The expected values for time and space then become $\mathrm{E}(T) = (1 + \frac{1}{\alpha})\sqrt{\pi N/2}/m$ and $\mathrm{E}(S) = m(1 + \alpha)$. We see that there is a time-space trade-off. But even if we choose the constant α quite big, the space requirements are still small. Therefore, the limiting factor for solving discrete logarithms with the parallel rho algorithm is definitely time.

Remark 1. We have assumed that all distinguished points are collected by a single server. However, it is possible to parallelize the server side with no communication overhead. Assume that k servers collect the distinguished points. One could split up the distinguished point set $\mathcal{D}$ into k disjoint subsets $\mathcal{D}_i$ of roughly the same size. Server i would then only collect the points of $\mathcal{D}_i$. When a client gets a distinguished point, it would have to check to which subset $\mathcal{D}_i$ it belongs and send it to the appropriate server. Checking if a new distinguished point has already been computed previously can be done independently on each server.

[2] All clients also need to store a description of the iteration function. This, however, requires only $\mathrm{O}(\log N)$ bits per client.

3.3 Assumptions of the Analysis

At the beginning of §3, we made three assumptions on which we based our time and space analysis. We will now elaborate on how realistic these assumptions are in an actual implementation.

Randomness of the function: For our analysis, we assumed that the iteration function is perfectly random and therefore produces uniformly distributed group elements. In [12, 13], Teske shows that the function suggested by her behaves practically like a truly random function if the group elements are partitioned into about 20 subsets.

All collisions are useful: A collision reveals two representations of the form $g^{a_i}h^{b_i}$ and $g^{a_j}h^{b_j}$ of the same group element. If $b_i \not\equiv b_j \pmod{N}$, the collision can be used to calculate x. Because the b_i are random elements of $\mathbb{Z}_N$, the probability for this is $1 - \frac{1}{N}$. Therefore, the probability that a collision is not useful is $\frac{1}{N}$ and thus negligible.

Each iteration reaches a distinguished point: In [9], Schulte-Geers shows that the distinguished point set must be at least of size $c\sqrt{N}$ while c should not be too small. This is intuitively clear, since the only way for an iteration not to arrive at a distinguished point is to end up in a cycle without distinguished points, the expected length of which is $\sqrt{\pi N/8}$. The condition is certainly met by our distinguished point set (c is $\alpha m\sqrt{2/\pi}$ in our case). Schulte-Geers also finds that if we choose θ as described in §3.2, the proportion of starting points with iterations that end up in distinguished points is $1/(1 + \frac{\pi\mathcal{N}(0,1)^2}{2\alpha^2 m^2})$, where $\mathcal{N}(0,1)$ is a standard normally distributed random variable.

Further, Schulte-Geers shows that if $\theta \gg 1/\sqrt{N}$, only a negligible number of starting points will miss the distinguished point set. We could meet this requirement by setting α to $\mathrm{O}(\log N)$. The space requirements still remain very small.

Additionally, van Oorschot and Wiener [6] suggest to abandon all trail portions without a single distinguished point that are longer than k/θ, k times their expected lengths. The proportion of time wasted through abandoned trails can be estimated[3] as $k(1-\theta)^{k/\theta} \approx ke^{-k}$ which is very small.

3.4 Statistical Analysis

Until now, we have only considered expected values for time and space. We will now have a look at the probability distributions of these.

As already explained, the time for finding a discrete logarithm with parallel Pollard-rho can be divided in two phases, the time until a collision occurs and

[3] Here, we assume the length of the trail portions between subsequent distinguished points to be geometrically distributed. Note that this model is slightly inaccurate since it implies that all such trail portions eventually lead to a distinguished point. For reasonably chosen values of θ, the model will do, however, since the probability of ending up in cycles without distinguished points is, indeed, very small.

the time needed for its detection. We will first treat those phases individually. As seen in §2.1, the probability that more than l points are needed for a collision is $p_l \approx e^{-l^2/2N}$. Because in time k, mk points are calculated, the probability that the time T_1 for the first phase is longer than k is $\Pr\{T_1 > k\} = p_{mk} \approx e^{-\frac{(mk)^2}{2N}}$. Because the time T_2 for the second phase of detecting a collision is geometrically distributed, the probability that $T_2 > k$ is $\Pr\{T_2 > k\} = (1-\theta)^k$. Therefore, the probabilities that T_1, resp. T_2 are bigger than β times their expected values is:

$$\Pr\{T_1 > (\beta\sqrt{\pi N/2})/m\} \approx e^{-\beta^2\pi/4} \quad \text{and} \quad \Pr\{T_2 > \beta/\theta\} = (1-\theta)^{\beta/\theta} \approx e^{-\beta}. \tag{3}$$

For the probability for T_2, given in (3), note that θ is very small and that $\lim_{x\downarrow 0}(1-x)^{\beta/x} = e^{-\beta}$.

We want to avoid having to calculate exact probabilities for the overall time $T = T_1 + T_2$. Therefore, we assume that α in §3.2 is chosen sufficiently large to achieve a good running time. In this case, T_1 dominates the time T and we can approximate the probability that $T > \beta\mathrm{E}(T)$ with $\Pr\{T_1 > \beta\mathrm{E}(T_1)\}$. Taking Equation (3), we then get:

$$\Pr\{T > \beta\mathrm{E}(T)\} \approx e^{-\beta^2\pi/4}. \tag{4}$$

Table 1 gives samples of the probabilities for various values of β.

β	1/100	1/10	1/3	1/2	1	3/2	2	3
$\Pr\{T > \beta\mathrm{E}(T)\}$	1.000	0.992	0.916	0.822	0.456	0.171	0.043	0.001

Table 1. Probabilities for the running time of Pollard's rho algorithm.

For space, exactly the same analysis holds. In fact, the space needed is very close to $m\theta T$ where T is the actual running time. This is because the length of every iteration trail is geometrically distributed with parameter θ and the lengths of different trails are statistically independent. By application of the limit theorem, we get that the average length of the trails is very close to the expected length.

4 Solving Multiple Instances of the DLP

In this section, we consider the situation where one wants to solve multiple, say L, discrete logarithms in the same group (using the same generator). Hence, we have a set of L group elements $h_i = g^{x_i}$ (where $1 \le i \le L$) and we would like to find all exponents x_i. This can be done by solving each of the discrete logarithms individually, using the rho algorithm. A better approach, however, is to take advantage of the distinguished points gathered during the solution of the first k discrete logarithm problems using the rho algorithm, to speed up the

solution of the $(k+1)^{st}$ discrete logarithm. As soon as we find a discrete logarithm $x_i = \log_g h_i$, we have a representation of the form g^c for all distinguished points $g^{a_j} h_i^{b_j}$ that were calculated in order to find x_i. The value of c is $c = (a_j + x_i b_j) \bmod N$. If we now find a collision between a distinguished point g^c and a new one of the form $g^a h_k^b$, we can calculate x_k as $x_k = (c-a)b^{-1} \bmod N$. This method was also suggested by Silverman and Stapleton [11], although a precise analysis has not been published. It seems obvious that the number of operations required for solving each new logarithm will become smaller, if one takes advantage of information gathered during previous computations. In this section, we will provide an exact analysis for this case.

The number of points we have to calculate with the rho algorithm to find L discrete logarithms is equal to the number of points we have to choose with replacement out of a set with N numbers until we have chosen L numbers at least twice (i.e. there are L collisions). We denote the expected value for the number of draws W to find L collisions by $\mathrm{E}(W) = E_L$.

Theorem 1. *We have* $E_L \approx \sqrt{\pi N/2} \sum_{t=0}^{L-1} \frac{\binom{2t}{t}}{4^t}$ *for* $L \ll \sqrt[4]{N}$.

Proof: Suppose that an urn has N differently numbered balls. We consider an experiment where one uniformly draws n balls from this urn one at a time, with replacement, and lists the numbers. It is clear that if one obtains $k < n$ different numbers after n draws, then $n-k$ balls must have been drawn more than once (counting multiplicity), i.e., $n-k$ 'collisions' must have occurred. We will be mainly interested in the probability distribution of the number of collisions as a function of the number of draws.

Let $q_{n,k}$ denote the probability that one obtains exactly k differently numbered outcomes after n draws. For any fixed k-set, the number of ways to choose precisely k differently numbered balls in n draws equals $a(n,k)$, the number of surjections from an n-set to a k-set. Hence, the number of possibilities to choose exactly k different balls in n draws equals $\binom{N}{k} a(n,k)$ and, therefore,

$$q_{n,k} = \binom{N}{k} a(n,k)/N^n = \frac{a(n,k)}{k! N^{n-k}} \frac{N(N-1)\cdots(N-k+1)}{N \cdot N \cdots N} = \frac{S(n,k)}{N^{n-k}} p_k,$$

where $p_k = (1-1/N)(1-2/N)\cdots(1-(k-1)/N)$ is the probability of drawing k differently numbered balls in k draws, and where $S(n,k) := a(n,k)/k!$ is a Stirling number of the second type (cf. Appendix A).

We now compute the expected number E_L of draws until one obtains precisely L collisions. Let $Q^+_{n,n-L}$ denote the probability that one requires more than n draws in order to obtain L collisions. Hence

$$Q^+_{n,n-L} = \sum_{t=0}^{L-1} q_{n,n-t}. \tag{5}$$

Now, the probability that one needs exactly n draws in order to obtain L collisions is given by $Q^+_{n-1,n-1-L} - Q^+_{n,n-L}$. As a result, the expected number of

draws that one needs in order to obtain L collisions is given by

$$E_L = \sum_{n=L}^{\infty} n(Q^+_{n-1,n-1-L} - Q^+_{n,n-L}) = (L-1) + \sum_{n=L-1}^{\infty} Q^+_{n,n-L}.$$

From equation (5) we infer that $Q^+_{n,n-(L+1)} = Q^+_{n,n-L} + q_{n,n-L}$, hence one obtains

$$E_{L+1} - E_L = 1 + \sum_{n=L}^{\infty} Q^+_{n,n-(L+1)} - \sum_{n=L-1}^{\infty} Q^+_{n,n-L} \tag{6}$$

$$= \sum_{n=L}^{\infty} q_{n,n-L} = \sum_{k=0}^{\infty} \frac{S(k+L,k)}{N^L} p_k. \tag{7}$$

Obviously, one has $E_0 = 0$, hence we can compute E_L via

$$E_L = \sum_{t=0}^{L-1} \sum_{k=0}^{\infty} \frac{S(k+t,k)}{N^t} p_k. \tag{8}$$

We will now approximate E_L based upon an approximation for $E_{t+1} - E_t$ (for $t < L$). It will turn out that the relative error of our approximation is negligible if $L < c_N \sqrt[4]{N}$ (here $0 < c_N < 1$ is a small constant). We will use the fact that for any fixed value of L, the Stirling number $S(k+L,k)$ is a polynomial in k of degree $2L$. More specifically, one has (cf. Lemma 1 of Appendix A) that

$$S(k+L,k) = \frac{1}{2^L L!} \sum_{j=0}^{2L} \varphi_j(L) k^{2L-j}, \quad \text{where } \varphi_j(L) \in \mathbb{Q}[x] \text{ has degree at most } 2j.$$

A substitution in Equation (6) now yields

$$E_{L+1} - E_L = \frac{1}{2^L L!} \sum_{j=0}^{2L} \frac{\varphi_j(L)}{\sqrt{N}^j} \sum_{k=0}^{\infty} \left(\frac{k}{\sqrt{N}}\right)^{2L-j} p_k. \tag{9}$$

We will now approximate this expression, using approximations for p_k and the function $\varphi_j(L)$. The inner summation can be approximated, using the approximation $p_k \approx e^{-k^2/2N}$ and a Riemann integral. We have

$$\sum_{k=0}^{\infty} \left(\frac{k}{\sqrt{N}}\right)^{2L-j} p_k \approx \sum_{k=0}^{\infty} \left(\frac{k}{\sqrt{N}}\right)^{2L-j} e^{-k^2/2N} \approx \sqrt{N} \int_{x=0}^{\infty} x^{2L-j} e^{-x^2/2} dx = \sqrt{N} I_{2L-j},$$

where I_t is the value of the integral determined in Lemma 2.[4] Substitution of this approximation in Equation (9) now yields

$$E_{L+1} - E_L \approx \sqrt{N} \frac{1}{2^L L!} \sum_{j=0}^{2L} \frac{\varphi_j(L)}{\sqrt{N}^j} I_{2L-j} = \sqrt{N} \frac{1}{2^L L!} \left(\frac{(2L)!}{L! 2^L} \sqrt{\pi/2} + \sum_{j=1}^{2L} \frac{\varphi_j(L)}{\sqrt{N}^j} I_{2L-j} \right)$$

[4] It turns out that the relative error of this approximation is $O(\log(N)/\sqrt{N})$. For details, cf. Lemma 4 and its subsequent remark.

$$= \sqrt{\pi N/2}\frac{\binom{2L}{L}}{4^L}(1+o(1)) \approx \sqrt{\pi N/2}\frac{\binom{2L}{L}}{4^L}.$$

The latter approximation follows from the fact that $\varphi_j(L) = 1$ and that for $j > 0$, $\varphi_j(L)$ is a polynomial in L of degree at most $2j$ without a constant term and, hence, $\varphi_j(L)/(\sqrt{N})^j \approx 0$ if $L \ll \sqrt[4]{N}$. Substituting this approximation in Equation (8), we now find that

$$E_L \approx \sum_{t=0}^{L-1} \sqrt{\pi N/2}\frac{\binom{2t}{t}}{4^t} = \sqrt{\pi N/2}\sum_{t=0}^{L-1}\frac{\binom{2t}{t}}{4^t}$$
$$= \sqrt{\pi N/2}(2L-1)\frac{\binom{2L-2}{L-1}}{4^{L-1}} \approx (2/\sqrt{\pi})\sqrt{L}\sqrt{\pi N/2} = \sqrt{2LN}.$$

□

Remark 2. The above result gives a highly accurate estimate of the expected time required to solve multiple instances of the discrete logarithm problem in the same underlying group. Unfortunately, this does not give direct insight in the probability distribution hereof. We should mention, however, that the same techniques used above to estimate expected values can also be used to estimate the vaiance of the probability distribution. It turns out that the variance, when compared to the expected time, is relatively low, especially if the number L of discrete logarithm problems one considers is not too small. Thus, the expected value of the running time of Theorem 1 is a good approximation of practically observed values (for L not too small). Full details will be provided in the full paper, space permitting.

We can conclude from Theorem 1 that computing discrete logarithms iteratively, rather than independently, is advantageous, since the workload involved in computing the $(t+1)^{st}$ discrete logarithm, once the first t of these have been solved, now becomes only $4^{-t}\binom{2t}{t} \approx 1/\sqrt{\pi t}$ times as much as the workload $\sqrt{\pi N/2}$ required for computing a single discrete logarithm. Thus, we arrived at a total workload for computing L discrete logarithms iteratively of approximately $\sqrt{2NL}$ group operations, which is $(2/\sqrt{\pi})\cdot\sqrt{L} \approx 1.128\sqrt{L}$ times as much as the workload for computing a single discrete logarithm. Thus, economies of scale apply: computing L discrete logarithms iteratively comes at an average cost per discrete logarithm of roughly $\sqrt{2N/L}$ group operations, rather than of approximately $\sqrt{\pi N/2}$ group operations (as is the case when computing discrete logarithms independently). Our results hold for $0 < L < c_N\sqrt[4]{N}$, where $0 < c_N < 1$ is some small constant.

Our extension of Pollard's rho algorithm is a generic algorithm for solving multiple instances of the discrete logarithm problem in finite cyclic groups. The low average workload [5] we obtained for computing multiple discrete logarithms

[5] The average workload per discrete logarithm is $O(N^{3/8})$ group operations if one solves $L \approx c_N\sqrt[4]{N}$ discrete logarithm problems iteratively.

seems to be counter-intuitive, since it seems to contradict Shoup's result [10], which gives a lower bound of $\Omega(\sqrt{N})$ group operations required by generic algorithms solving the discrete logarithm problem in groups of prime order N. The result is explained by observing that the low average workload is due to the fact that solving subsequent discrete logarithm problems requires relatively few operations, once the first few discrete logarithms have been computed. Thus, the bottleneck remains the computation of, e.g., the first discrete logarithm, which in our case requires roughly $\sqrt{\pi N/2} = \Omega(\sqrt{N})$ group operations. It should be noted, that Shoup's result does not apply directly, since he addresses the scenario of a single instance of the discrete logarithm problem, rather than that of multiple instances hereof, which we address. Thus, one cannot a priori rule out the existence of other generic algorithms that, given L instances of the discrete logarithm problem in a group of prime order N, solve an arbitrary one of these using only $O(\sqrt{N/L})$ group operations.

5 On the Complexity of DLP-like Problems

In the previous sections, we discussed the workload required for solving multiple instances of the discrete logarithm problem with respect to a fixed generator of a finite cyclic group of order N, using extensions of Pollard's rho algorithm. We found that computing discrete logarithms iteratively, rather than independently, is advantageous. In §5.1 we will consider the problem of solving 1 out of n instances of the discrete logarithm problem and several other relaxations of the classical discrete logarithm problem (DLP) and consider the computational complexity hereof. It turns out that these problems are all computationally as hard as DLP. In particular, it follows that generic algorithms for solving each of these relaxations of the discrete logarithm problem in a prime order group require $\Omega(\sqrt{N})$ group operations. In §5.2 we consider the generalization of the classical discrete logarithm problem of solving k instances hereof (coined kDLP). Again, we consider several relaxations of this so-called kDLP and discuss their computational complexity. It turns out that, similar to the case $k = 1$, these problems are all computationally as hard as solving kDLP. We end the section with a conjectured lower bound $\Omega(\sqrt{kN})$ on the complexity of generic algorithms for solving kDLP, which – if true – would generalize Shoup's result for DLP towards kDLP.

5.1 Complexity of Solving 1 of Multiple Instances of the DLP

We consider the following variations of the discrete logarithm problem:

1. (DLP-1) Solving a single instance of the discrete logarithm problem:

System:	Cyclic group G; generator g for G.
Input:	Group element $h \in_R G$.
Output:	Integer x such that $h = g^x$.

2. (DLP-2) Solving a single instance of the discrete logarithm problem (selected arbitrarily from a set of n instances of the discrete logarithm problem):

   System: Cyclic group G; generator g for G.
   Input: Group elements $h_1, \ldots, h_n \in_R G$.
   Output: Pair (j, x_j) such that $h_j = g^{x_j}$ and such that $1 \leq j \leq n$.

3. (DLP-3) Finding a discrete logarithm with respect to an arbitrary basis element (selected from a set of m basis elements):

   System: Cyclic group G; arbitrary generators $g_1, \ldots, g_m$ for G.
   Input: Group element $h \in_R G$.
   Output: Pair (i, x) such that $h = g_i^x$ and such that $1 \leq i \leq m$.

4. (DLP-4) Finding a linear equation in terms of the discrete logarithms of all group elements of a set of n instances of the discrete logarithm problem:

   System: Cyclic group G; generator g for G.
   Input: Group elements $h_1, \ldots, h_n \in_R G$.
   Output: A linear equation $\sum_{j=1}^{n} a_j \log_g h_j = b$ (with known values of $a_1, \ldots, a_n$ and b).

5. (DLP-5) Finding the differences of two discrete logarithms (selected arbitrarily from of a set of n instances of the discrete logarithm problem):

   System: Cyclic group G; generator g for G.
   Input: Group elements $h_1, \ldots, h_n \in_R G$.
   Output: Triple $(i, j, \log_g h_i - \log_g h_j)$, where $0 \leq i \neq j \leq n$ and where $h_0 := g$.

The following theorem relates the expected workloads required by optimal algorithms for solving the discrete logarithm problem (DLP-1) and for solving arbitrarily 1 out n instances of the discrete logarithm problem (DLP-2).

Theorem 2. *Let T_{DLP}, resp. $T_{DLP(1:n)}$, be the expected workload of an optimal algorithm for solving the discrete logarithm problem, resp. for arbitrary solving 1 out of n instances of the discrete logarithm problem. Then, one has $T_{DLP(1:n)} \leq T_{DLP} \leq T_{DLP(1:n)} + n$ (in group operations).*

Proof: The inequality $T_{DLP} \leq T_{DLP(1:n)} + n$ follows from a reduction of an instance of the DLP to an instance of the DLP(1:n). The other inequality follows from the observation that DLP=DLP(1:1). Let $h := g^x$ be a problem instance of DLP. We will reduce this to a problem instance $h_1, \ldots, h_n$ of DLP(1:n) as follows: for all $i, 1 \leq i \leq n$, select the numbers r_i uniformly at random from the set $\{0, \ldots, N-1\}$ and define $h_i := g^{r_i} h = g^{x+r_i}$. Note that all h_i are random, since all $x + r_i$ are random and independent. Now apply an oracle that solves DLP(1:n), to produce an output (j, x_j), with $x_j := \log_g h_j$ and with $1 \leq j \leq n$. Since $h_j = g^{r_j} h$ and since r_j is known, we get the required discrete logarithm x as $x \equiv x_j - r_j (\text{mod } N)$. □

Corollary 1. *The problem of solving arbitrarily 1 out of n instances of the discrete logarithm is computationally as hard as solving the discrete logarithm problem, provided $n \ll T_{DLP}$. Moreover, any generic algorithm that solves this problem in a group of prime order N requires at least $\Omega(\sqrt{N})$ group operations.*

Proof: The bound $T_{DLP(1:n)} = \Omega(T_{DLP})$ follows from Theorem 2 and the inequality $n \ll T_{DLP}$. The lower bound on the required workload for a generic algorithm that solves[6] the relaxed discrete logarithm problem DLP(1:n) in groups of prime order N follows from the corresponding result for the discrete logarithm problem [10]. □

Remark 3. In fact, one can show that Theorem 2 and Corollary 1 easily generalize to each of the problems DLP-1, ..., DLP-5 above. In particular, one has that each of the problems DLP-1, ..., DLP-5 is as hard as solving the discrete logarithm problem, provided $n, m \ll T_{DLP}$. Moreover, any generic algorithm that solves any of the the problems DLP-1, ..., DLP-5 in a group of prime order N requires at least $\Omega(\sqrt{N})$ group operations.

Remark 4. One can show that each of the problems DLP-1, ..., DLP-5 can be solved directly using Pollard's rho algorithm, with starting points of the randomized walks that are tailored to the specific problem at hand. In each case, the resulting workload is roughly $\sqrt{\pi N/2}$ group operations.

5.2 Complexity of Solving Multiple Instances of the DLP

In the previous section, we related the workload involved in solving various relaxations of the classical discrete logarithm problem. The main result was that the problem of solving arbitrarily 1 out of n instances of the discrete logarithm is computationally as hard as solving a single instance of the discrete logarithm problem. In this section, we consider the similar problem where we are faced with solving k given instances of the discrete logarithm problem.
We consider the following variations of the discrete logarithm problem kDLP:

1. (kDLP-1) Solving k instances of the discrete logarithm problem:

System:	Cyclic group G; generator g for G.
Input:	Group elements $h_1, \ldots, h_k \in_R G$.
Output:	k pairs (i, x_i) such that $h_i = g^{x_i}$.

2. (kDLP-2) Solving k instances of the discrete logarithm problem (selected arbitrarily from a set of n instances of the discrete logarithm problem):

System:	Cyclic group G; generator g for G.
Input:	Group elements $h_1, \ldots, h_n \in_R G$.
Output:	k pairs (j, x_j) such that $h_j = g^{x_j}$, where $j \in J$ and where J is a k-subset of $\{1, \ldots, n\}$.

[6] with a probability bounded away from zero

3. (kDLP-3) Finding k discrete logarithms with respect to k arbitrary basis elements (selected from a set of m basis elements):

System:	Cyclic group G; arbitrary generators $g_1, \ldots, g_m$ for G.
Input:	Group element $h \in_R G$.
Output:	k pairs (i, x_i) such that $h = g_i^{x_i}$, where $i \in I$ and where I is a k-subset of $\{1, \ldots, m\}$.

4. (kDLP-4) Finding k linear equations in terms of the discrete logarithms of all group elements of a set of n instances of the discrete logarithm problem:

System:	Cyclic group G; generator g for G.
Input:	Group elements $h_1, \ldots, h_n \in_R G$.
Output:	A set of k linear equations $\sum_{j=1}^{n} a_{ij} \log_g h_j = b_i$ (with known values of a_{ij} and b_i).

5. (kDLP-5) Finding k differences of two discrete logarithms (selected arbitrarily from of a set of n instances of the discrete logarithm problem):

System:	Cyclic group G; generator g for G.
Input:	Group elements $h_1, \ldots, h_n \in_R G$.
Output:	A set of k triples $(i, j, \log_g h_i - \log_g h_j)$, where $0 \leq i \neq j \leq n$ and where $h_0 := g$.

One can show that the results of the previous subsection carry over to this section, as follows:

- Each of the problems kDLP-1, ..., kDLP-5 is as hard as solving k instances of the discrete logarithm problem, provided $kn, km, k^2 \ll T_{DLP}$.
- Any generic algorithm for solving k instances of the discrete logarithm problem in a group of prime order N require at least $\Omega(\sqrt{N})$ group operations.
- Each of the problems kDLP-1, ..., kDLP-5 can be solved directly using the extension of Pollard's rho algorithm presented in §4, with starting points of the randomized walks that are tailored to the specific problem at hand. In each case, the resulting workload is roughly $\sqrt{2Nk}$ group operations.

The proofs use constructions based on maximum distance separable codes (cf., e.g., [5]). Details will be included in the full paper.

The lower bound on the required workload for a generic algorithm that solves k instances of the discrete logarithm problem is not very impressive: it derives directly from Shoup's lower bound $\Omega(\sqrt{N})$ for solving a single discrete logarithm (i.e., $k = 1$) . It would be of interest to find a stronger lower bound in this case. Based on the workload involved in computing k discrete logarithm problems iteratively, we postulate that the 'true' lower bound is $\Omega(\sqrt{kN})$. We suggest this as an open problem.

Research Problem Show that any generic algorithm that solves, with a probability bounded away from zero, k instances of the discrete logarithm problem in groups of prime order N requires at least $\Omega(\sqrt{kN})$ group operations.

References

1. L. Adleman and J. De Marrais, A Subexponential Algorithm for Discrete Logarithms over All Finite Fields, in *Advances of Cryptology –CRYPTO'93*, D.R. Stinson, Ed., pp. 147-158, Lecture Notes in Computer Science, Vol. 773, Berlin: Springer, 1993.
2. L. Adleman, J. DeMarrais and M. Huang, A Subexponential Algorithm for Discrete Logarithms over the Rational Subgroup of the Jacobians of Large genus Hyperelliptic Curves over Finite Fields, in *Algorithmic Number Theory*, pp. 28-40, Lecture Notes in Computer Science, Vol. 877, Berlin: Springer, 1994.
3. R.P. Brent, An Improved Monte Carlo Factorization Algorithm, *j-BIT*, Vol. 20, No. 2, pp. 176-184, 1980.
4. J.H. van Lint and R.M. Wilson, *A Course in Combinatorics*, Cambridge: Cambridge University Press, 1992.
5. F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error-Correcting Codes*, Amsterdam: North-Holland, 1977.
6. P.C. van Oorschot and M.J. Wiener, Parallel Collision Search with Cryptanalytic Applications, *J. of Cryptology*, Vol. 12, pp. 1-28, 1999.
7. S. Pohlig and M. Hellman, An Improved Algorithm for Computing Logarithms in $GF(p)$ and its Cryptographic Significance, *IEEE Trans. Inform. Theory*, Vol. IT-24, pp. 106-111, January 1978.
8. J.M. Pollard, Monte Carlo Methods for Index Computation (mod p), *Mathematics of Computation*, Vol. 32, No. 143, pp. 918-924, July 1978.
9. E. Schulte-Geers, Collision Search in a Random Mapping: Some Asymptotic Results, presented at ECC 2000 – The Fourth Workshop on Elliptic Curve Cryptography, University of Essen, Germany, October 4-6, 2000.
10. V. Shoup, Lower Bounds for Discrete Logarithms and Related Problems, in *Advances in Cryptology – EUROCRYPT '97*, W. Fumy, Ed., Lecture Notes in Computer Science, Vol. 1233, pp. 256-266, Berlin: Springer, 1997.
11. R. Silverman and J. Stapleton, Contribution to ANSI X9F1 working group, December 1997.
12. E. Teske, Speeding up Pollard's Rho Method for Computing Discrete Logarithms, in *Proceedings of ANTS III – The 3rd International Symposium on Algorithmic Number Theory*, J.P. Buhler, Ed., Lecture Notes in Computer Science, Vol. 1423, pp. 351-357, Berlin: Springer, 1998.
13. E. Teske, On Random Walks for Pollard's Rho Method, *Mathematics of Computation*, Vol. 70, pp. 809-825, 2001.
14. E. Teske, Square-Root Algorithms for the Discrete Logarithm Problem (A Survey), Technical Report CORR 2001-07, Centre for Applied Cryptographic Research, University of Waterloo, 2001.
15. M.J. Wiener and R.J. Zuccherato, Faster Attacks on Elliptic Curve Cryptosystems, in *Proceedings of SAC'98 – Fifth Annual Workshop on Selected Areas in Cryptography*, E. Tavares, H. Meijer, Eds., Lecture Notes in Computer Science, Vol. 1556, pp. 190-200, Berlin: Springer, 1998.

A Stirling Numbers

In this section, we introduce Stirling numbers. These numbers play an important role in several parts of combinatorics. We mention several properties of these numbers that will be used in the paper. For a detailed discussion, we refer to [4].

In the sequel, n and k denote non-negative integers. Let $a(n,k)$ denote the number of surjections from an n-set to a k-set. Obviously, one has $a(n,k) = 0$ if $k > n$ and $a(n,k) = n!$ if $n = k$. In general, one can use the principle of inclusion-exclusion to show that

$$a(n,k) = \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n.$$

Let $S(n,k)$ denote the number of ways to partition an n-set into k nonempty subsets. The numbers $S(n,k)$ are called Stirling numbers of the second kind. Obviously, one has that $S(n,k) = a(n,k)/k!$. Moreover, by a simple counting argument, one can show that

$$S(n,k) = \sum_{\substack{1a_1 + 2a_2 + \cdots + na_n = n \\ a_1 + a_2 + \cdots + a_n = k}} \frac{n!}{(1!)^{a_1}(2!)^{a_2}\ldots(n!)^{a_n} a_1! a_2! \ldots a_n!}. \tag{10}$$

Our main interest is in those Stirling numbers $S(n,k)$ for which $n-k$ is relatively small. The first few of these are

$$\begin{aligned}
&S(k,k) = 1;\\
&S(k+1,k) = \tfrac{1}{2}(k+1)k = \tfrac{1}{2}(k^2+k);\\
&S(k+2,k) = \tfrac{1}{8}(k+2)(k+1)k(k+\tfrac{1}{3}) = \tfrac{1}{8}(k^4 + \tfrac{10}{3}k^3 + 3k^2 + \tfrac{2}{3}k);\\
&S(k+3,k) = \tfrac{1}{48}(k+3)(k+2)(k+1)k(k^2+k) = \tfrac{1}{48}(k^6 + 7k^5 + 17k^4 + 17k^3 + 6k^2);\\
&S(k+4,k) = \tfrac{1}{384}(k+4)(k+3)(k+2)(k+1)k(k^3 + 2k^2 + \tfrac{1}{3}k - \tfrac{2}{15})\\
&\qquad\quad = \tfrac{1}{384}(k^8 + 12k^7 + \tfrac{166}{3}k^6 + \tfrac{616}{5}k^5 + \tfrac{403}{3}k^4 + 60k^3 + \tfrac{4}{3}k^2 - \tfrac{16}{5}k).
\end{aligned}$$

Computing $S(k+L,k)$ for bigger values of L is quite cumbersome. Fortunately, however, one can express $S(k+L,k)$ as a polynomial in k with coefficients in L, as is demonstrated by the following lemma.

Lemma 1. *For all $k, L \geq 0$, one has $S(k+L,k) = \frac{1}{2^L L!} \sum_{j=0}^{2L} \varphi_j(L) k^{2L-j}$, where $\varphi_j(x) \in \mathbb{Q}[x]$ has degree at most $2j$ $(0 \leq j \leq 2L)$. For $j > 0$, one has $x | \varphi_j(x)$. The first few coefficients are*

$$\varphi_0(x) = 1, \quad \varphi_1(x) = \frac{2}{3}x^2 + \frac{1}{3}x, \quad \varphi_2(x) = \frac{2}{9}x^4 + \frac{2}{3}x^3 - 2\frac{1}{18}x^2 - \frac{7}{16}x.$$

Proof: To be provided in the full version of this paper. □

B Approximations of Combinatorial Expressions

In this section, we provide approximations of some combinatorial expressions that will be used in the paper and indicate the accuracy of these approximations.

Lemma 2. *For all $t \geq 0$, define $I_t := \int_{x=0}^{\infty} x^t e^{-x^2/2} dx$. Then*

$$I_{2t} = \frac{(2t)!}{t!2^t}\sqrt{\pi/2} \text{ and } I_{2t-1} = (t-1)!2^{t-1}.$$

Proof: The result follows using partial integration and an induction argument. For $t > 1$, one has $I_t = (t-1)I_{t-2}$, since

$$I_t = \int_{x=0}^{\infty} x^t e^{-x^2/2} dx = -\int_{x=0}^{\infty} x^{t-1} d(e^{-x^2/2})$$

$$= [-x^{t-1}e^{-x^2/2}]_{x=0}^{\infty} + (t-1)\int_{x=0}^{\infty} x^{t-2}e^{-x^2/2}dx = (t-1)I_{t-2}.$$

Moreover, $I_0 = \sqrt{\pi/2}$ and $I_1 = 1$, since

$$I_0^2 = (\int_{x=0}^{\infty} e^{-x^2/2}dx)^2 = \int_{\varphi=0}^{\pi/2}\int_{r=0}^{\infty} e^{-r^2/2} r dr d\varphi = \pi/2 \text{ and } I_1 = -\int_{x=0}^{\infty} d(e^{-x^2/2}) = 1.$$

The result now follows using induction. □

Lemma 3. *Let $N > 0$, let $t \in \mathbb{N}$. Then*

$$\frac{1}{\sqrt{N}}\sum_{k=0}^{\infty}\left(\frac{k}{\sqrt{N}}\right)^t e^{-k^2/2N} \to I_t \quad (N \to \infty).$$

Proof: The result follows from the observation that the expression is a Riemann sum that converges to I_t. □

Lemma 4. *Let $N > 0$, let $t \in \mathbb{N}$, and let $p_k := \prod_{i=0}^{k-1}(1 - i/N)$. Then*

$$\frac{1}{\sqrt{N}}\sum_{k=0}^{\infty}\left(\frac{k}{\sqrt{N}}\right)^t p_k \to I_t \quad (N \to \infty).$$

Proof: The result follows from Lemma 3, using the estimate $p_k \approx e^{-k^2/2N}$ while upper bounding the approximation error in the 'tail' of the summation. Details will be provided in the full paper. □

Remark 5. One can show that convergence is as follows:

$$\frac{1}{\sqrt{N}}\sum_{k=0}^{\infty}\left(\frac{k}{\sqrt{N}}\right)^t p_k = (1+\varepsilon)I_t, \quad \text{where } |\varepsilon| \in O(\log(N)/\sqrt{N}).$$

Hence, for big values of N (as in our applications) the approximation of the expression by I_t is almost precise.