Computing Elliptic Curve Discrete Logarithms with Improved Baby-step Giant-step Algorithm

Steven D. Galbraith[1], Ping Wang[2] and Fangguo Zhang[3]

[1] Mathematics Department,
University of Auckland, New Zealand
S.Galbraith@math.auckland.ac.nz
[2] College of Information Engineering,
Shenzhen University, Shenzhen 518060, China
wangping@szu.edu.cn
[3] School of Information Science and Technology,
Sun Yat-sen University, Guangzhou 510006, China
isszhfg@mail.sysu.edu.cn

Abstract. The negation map can be used to speed up the computation of elliptic curve discrete logarithms using either the baby-step giant-step algorithm (BSGS) or Pollard rho. Montgomery's simultaneous modular inversion can also be used to speed up Pollard rho when running many walks in parallel. We generalize these ideas and exploit the fact that for any two elliptic curve points X and Y, we can efficiently get $X - Y$ when we compute $X + Y$. We apply these ideas to speed up the baby-step giant-step algorithm. Compared to the previous methods, the new methods can achieve a significant speedup for computing elliptic curve discrete logarithms in small groups or small intervals.
Another contribution of our paper is to give an analysis of the average-case running time of Bernstein and Lange's "grumpy giants and a baby" algorithm, and also to consider this algorithm in the case of groups with efficient inversion.
Our conclusion is that, in the fully-optimised context, both the interleaved BSGS and grumpy-giants algorithms have superior average-case running time compared with Pollard rho. Furthermore, for the discrete logarithm problem in an interval, the interleaved BSGS algorithm is considerably faster than the Pollard kangaroo or Gaudry-Schost methods.

Keywords: baby-step giant-step algorithm, elliptic curve discrete logarithm, negation map.

1 Introduction

The discrete logarithm problem (DLP) in finite groups is an important computational problem in modern cryptography. Its presumed hardness provides the basis for security for a number of cryptographic systems. The DLP was first proposed in the multiplicative groups of finite fields. Koblitz [13] and Miller [14]

were the first to suggest that elliptic curves over finite fields would have some advantages.

Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$. Let $P \in E(\mathbb{F}_q)$ be a point of prime order N, and let $\langle P \rangle$ be the prime order subgroup of E generated by P. If $Q \in \langle P \rangle$, then $Q = nP$ for some integer n, $0 \leq n < N$. The problem of finding n, given P, Q, and the parameters of E, is known as the *elliptic curve discrete logarithm problem* (ECDLP).

The baby-step giant-step algorithm (BSGS), attributed to Shanks [19] and Gel'fond (see [16]), allows one to compute discrete logarithms in a cyclic group G of order N in deterministic time $O(\sqrt{N})$ group operations. The algorithm also requires $\sqrt{N}$ group elements storage. Standard textbook descriptions of the algorithm state a worst-case running time of $2\sqrt{N}$ group operations, and Pollard showed that one can reduce the average case running time to $1.333\sqrt{N}$ group operations for general groups [18]. Bernstein and Lange [2] suggested a new variant of the BSGS algorithm, called the "two grumpy giants and a baby" algorithm, and they conjecture it runs faster in the average-case than Pollard's variant.

The Pollard rho algorithm [17] is a probabilistic algorithm that has low storage but retains the $O(\sqrt{N})$ expected running time. If G is an elliptic curve group chosen according to standard criteria then the best discrete logarithm algorithms available are variants of the baby-step giant-step method (for deterministic algorithms) and the Pollard rho method (for probabilistic algorithms).

Note that the BSGS algorithm is useful for a number of variants of the discrete logarithm problem. We define the "DLP in an interval" as: Given P and Q such that $Q = nP$ and $0 \leq n < N$, to find n. This problem makes sense even when the order of P is much larger than N. One can also consider the "multi-dimensional DLP", which can be used in point-counting algorithms [10]. Computations of discrete logarithms in small intervals are used as subroutines in several cryptographic protocols in the literature. For example: the BGN degree-2-homomorphic public-key encryption [4] system uses a generic discrete logarithm method for decryption; the Henry-Henry-Goldberg privacy-preserving protocol [12] uses discrete logarithms in small groups. Baby-step giant-step algorithms immediately apply to these applications as well.

The Pollard rho algorithm is not applicable to the DLP in an interval. Instead one should use the Pollard kangaroo algorithm [17] or the Gaudry-Schost algorithm [10]. The heuristic average-case running time of the Pollard kangaroo algorithm in an interval of size N is $(2 + o(1))\sqrt{N}$ group operations, and this was improved to a heuristic $(1.661 + o(1))\sqrt{N}$ operations by [8].

Gallant, Lambert and Vanstone [9], and Wiener and Zuccherato [23] pointed out that the Pollard rho method in an elliptic curve group of size N can be sped up by defining the random walk on equivalence classes. For a point P on an elliptic curve over a finite field $\mathbb{F}_q$, it is trivial to determine its inverse $-P$. Therefore, we can consider the equivalence relation $\sim$ induced by the negation map (i.e., $P \sim Q$ if and only if $Q \in \{P, -P\}$). One can then define a random walk on the set of equivalence classes $\{\pm P\}$ to halve the search space in the

Pollard rho algorithm. Theoretically, one can achieve a speedup by a factor of $\sqrt{2}$ when solving the ECDLP, for details see [1, 21]. This idea can be applied in any algebraic group for which inversion is efficient (for example in torus-based cryptography). For the DLP in an interval of size N, Galbraith and Ruprai [7] showed a variant of the Gaudry-Schost algorithm with heuristic average-case running time of $(1.36 + o(1))\sqrt{N}$ group operations.

The most expensive operation in the baby-step giant-step algorithm is point addition. To speed up the algorithm we can aim to either reduce the total number of group operations performed, or else to reduce the cost of each individual group operation. The main observation in this paper allows us to take the second approach. Precisely, we exploit the fact that, given any two points X and Y, we can compute $X + Y$ and $X - Y$ (in affine coordinates) in less than two times the cost of an elliptic curve addition, by recycling the field inversion. Hence, we propose improved algorithms to compute a list of consecutive elliptic curve points. We also discuss improved variants of the "grumpy giants" algorithm of Bernstein and Lange. We give a new approach to determine the average-case behaviour of the grumpy-giants algorithm, and also describe and analyse a variant of this algorithm for groups with efficient inversion. All our results are summarised in Table 3.

The paper is organized as follows. We recall the baby-step giant-step algorithm and its minor modifications in Section 2. Section 2.1 gives our new heuristic method for analysing interleaved BSGS algorithms. In Section 3, we describe a negation map variant of the baby-step giant-step algorithm and grumpy-giants algorithm. Section 4 discusses methods for efficiently computing lists of consecutive points on elliptic curve. Section 5 brings the ideas together to present faster variants of the BSGS algorithm. Details of our experiments are given in the Appendices.

2 The Baby-Step Giant-Step Algorithm (BSGS)

In this section, we recall the baby-step giant-step algorithm for DLP computations and present some standard variants of it. The baby-step giant-step algorithm makes use of a time-space tradeoff to solve the discrete logarithm problem in arbitrary groups. We use the notation of the ECDLP as defined as in the introduction: Given P and Q to find n such that $Q = nP$ and $0 \leq n < N$.

The "textbook" baby-step giant-step algorithm is as follows: Let $M = \lceil \sqrt{N} \rceil$. Then $n = n_0 + Mn_1$, where $0 \leq n_0 < M$ and $0 \leq n_1 < M$. Precompute $P' = MP$. Now compute the baby steps $n_0 P$ and store the pairs $(n_0 P, n_0)$ in an easily searched structure (searchable on the first component) such as a sorted list or binary tree. Then compute the giant steps $Q - n_1 P'$ and check whether each value lies in the list of baby steps. When a match $n_0 P = Q - n_1 P'$ is found then the ECDLP is solved. The baby-step giant-step algorithm is deterministic. The algorithm requires $2\sqrt{N}$ group operations in the worst case, and $\frac{3}{2}\sqrt{N}$ group operations on average over uniformly random choices for Q.

To improve the average-case performance one can instead choose $M = \lceil \sqrt{N/2} \rceil$. Then $n = n_0 + Mn_1$, where $0 \leq n_0 < M$ and $0 \leq n_1 < 2M$. We precompute $P' = MP$ and compute baby steps $n_0 P$ for $0 \leq n_0 < M$ and store them appropriately. Then we compute the giant steps $Q - n_1 P'$. On average, the algorithm finds a match after half the giant steps have been performed. Hence, this variant solves the DLP in $\sqrt{2N}$ group operations on average. The worst-case complexity is $(M + 2M) = (\frac{1}{\sqrt{2}} + \sqrt{2})\sqrt{N} = \frac{3}{\sqrt{2}}\sqrt{N}$ group operations.

A variant of the baby-step giant-step algorithm due to Pollard [18] is to compute the baby steps and giant steps in parallel, storing the points in two sorted lists/binary trees. One can show that the average-case running time of this variant of the baby-step giant-step algorithm is $\frac{4}{3}\sqrt{N}$ group operations (Pollard justifies this with an argument that uses the fact that the expected value of $\max\{x, y\}$, over uniformly distributed $x, y \in [0, 1]$, is $2/3$; at the end of Section 2.1 we give a new derivation of this result). We call this the "interleaving" variant of BSGS. The downside is that the storage requirement slightly increases (in both the average and worst case). We remark that the average-case running time of Pollard's method does not seem to be very sensitive in practice to changes in M; however to minimise both the average-case and worst-case running time one should choose the smallest value for M with the property that when both lists contain exactly M elements then the algorithm must terminate.

All the above analysis applies both for the discrete logarithm problem in a group of size N and the discrete logarithm problem in an interval of length N.

2.1 Grumpy Giants

Bernstein and Lange [2] suggested a new variant of the BSGS algorithm, called the "two grumpy giants and a baby" algorithm. The baby steps are of the form $n_0 P$ for small values n_0. One grumpy giant starts at Q and takes steps of size $P' = MP$ for $M \approx 0.5\sqrt{N}$. The other grumpy giant starts at $2Q$ and takes steps of size $-P'' = -(M + 1)P$. The algorithm is an interleaving algorithm in the sense that all three walks are done in parallel and stored in lists. At each step one checks for a match among the lists (a match between any two lists allows to solve the DLP; the case of a match $2Q - j(M + 1)P = iP$ implies the DLP satisfies $2n \equiv i + j(M + 1) \pmod{N}$, and this equation has a unique solution when $N > 2$ is prime). The exact performance of the grumpy giants method is not known, but Bernstein and Lange conjecture that their method can be faster than Pollard rho. However [2] does not contain very much evidence to support their claim.

The crux of the analysis in [2], building on the work of Chateauneuf, Ling and Stinson [5], is to count "slopes". We re-phrase this idea as follows: After L steps we have computed three lists $\{iP : 0 \leq i < L\}$, $\{Q + jMP : 0 \leq j < L\}$ and $\{2Q - k(M + 1)P : 0 \leq k < L\}$ and the DLP is solved as long as either $Q = (i - jM)P$ or $2Q = (i + k(M + 1))P$ or $Q = (jM + k(M + 1))P$. Hence, the number of points Q whose DLP is solved after L steps is exactly the size of

the union

$$\begin{aligned}\mathcal{L}_L = &\{i - jM \pmod N : 0 \le i, j < L\} \\ &\cup \{2^{-1}(i + k(M+1)) \pmod N : 0 \le i, k < L\} \\ &\cup \{jM + k(M+1) \pmod N : 0 \le j, k < L\}.\end{aligned}$$

The algorithm succeeds after L steps with probability $\#\mathcal{L}_L/N$, over random choices for Q. The term "slopes" is just another way to express the number of elements in $\mathcal{L}_L$.

Note that the grumpy giants algorithm is designed for solving the DLP in a group of size N. The algorithm is not appropriate for the DLP in an interval, as the lists $\{iP\}$ and $\{Q + jMP\}$ might not collide at all for $0 \le i, j \le M$, and the point $2Q$ may be outside the interval. Hence, for the rest of this section we assume P has order equal to N.

We have developed a new approach for computing approximations of the average-case runtimes of interleaved BSGS algorithms. We use this approach to give an approximation to the average-case running time of the grumpy giants algorithm. Such an analysis does not appear in [2].

We let α be such that the algorithm halts after at most $\alpha\sqrt{N}$ steps. When $M = \sqrt{N}$ the DLP is solved using the first two sets $\{iP\}$, $\{Q + jMP\}$ after $\sqrt{N}$ steps. Hence it seems safe to always assume $\alpha \le 1$. Note that α seems to be the main quantity that is influenced by the choice of M.

Now, for $1 \le L \le \alpha\sqrt{N}$ one can consider the size of $\mathcal{L}_L$ on average (the precise size depends slightly on the value of N). Initially we expect $\#\mathcal{L}_L$ to grow like $3L^2$, but as L becomes larger then the number decreases until finally at $L \approx \sqrt{N}$ we have $\#\mathcal{L}_L = N \approx L^2$. Bernstein and Lange suggest that $\#\mathcal{L}_L \approx \frac{23}{8}L^2$ when L becomes close to $\sqrt{N}$. The performance of the algorithm depends on the value $\#\mathcal{L}_L/L^2$ so we need to study this in detail. It will be more convenient to rescale the problem to be independent of N. So for integers $0 \le L \le \alpha\sqrt{N}$ we write $c(L/\sqrt{N}) = \#\mathcal{L}_L/L^2$. We then extend this function to $c(t)$ for a real variable $0 \le t \le \alpha$ by linearly interpolating those points. Note that $\#\mathcal{L}_L/N = c(L/\sqrt{N})L^2/N = c(t)t^2$ when $t = L/\sqrt{N}$.

We have performed simulations, for relatively small values of N, that enumerate the set $\mathcal{L}_L$ for a range of values of L. From these simulations we can get approximate values for α and get some data points for $c(t)$. A typical example (this is when $N \approx 2^{28}$ and $M = \sqrt{N/2}$) is $0.94 < \alpha < 0.97$ and some data points for $c(t)$ are listed in Table 1.

t	0	0.12	0.24	0.37	0.49	0.61	0.73	0.85	0.97
$c(t)$	3.00	3.00	3.00	2.99	2.79	2.30	1.77	1.35	1.06

Table 1. Size of set $\mathcal{L}_L$ written as $c(t)L^2$ where $t = L/\sqrt{N}$.

Denote by $\Pr(L)$ the probability that the algorithm has solved the DLP after L steps, and $\Pr(> L) = 1 - \Pr(L)$ the probability that the algorithm has not succeeded after L steps. As noted, after $L = t\sqrt{N}$ steps, for $0 \leq t \leq \alpha$, the algorithm succeeds with probability $\Pr(L) = \#\mathcal{L}_L/N = c(L/\sqrt{N})L^2/N = c(t)t^2$. Hence, the probability that the algorithm has not succeeded after $L = t\sqrt{N}$ steps is $\Pr(> L) = (1 - c(t)t^2)$. Now, the expected number of steps before the algorithm halts is (using the fact that $\Pr(\lfloor \alpha\sqrt{N} \rfloor) = 1$)

$$\sum_{L=1}^{\lfloor \alpha\sqrt{N} \rfloor} L(\Pr(L) - \Pr(L-1)) = \lfloor \alpha\sqrt{N} \rfloor \; - \sum_{L=0}^{\lfloor \alpha\sqrt{N} \rfloor} \Pr(L)$$
$$\approx \int_0^{\alpha\sqrt{N}} (1 - c(L/\sqrt{N})L^2/N)dL.$$

Substituting $t = L/\sqrt{N}$ and noting $dL = \sqrt{N}dt$ allows us to approximate the sum as

$$\left(\alpha - \int_0^{\alpha} c(t)t^2 dt\right)\sqrt{N}.$$

To determine the running time it remains to estimate $\int_0^{\alpha} c(t)t^2 dt$ and to multiply by 3 (since there are three group operations performed for each step of the algorithm). For example, using numerical integration (we simply use the trapezoidal rule) based on the data in Table 1, we estimate $\alpha \approx 0.97$ and $\int_0^{\alpha} c(t)t^2 dt \approx 0.55$. This estimate would give running time (number of group operations) roughly

$$3(0.97 - 0.55)\sqrt{N} = 1.26\sqrt{N}.$$

We now give slightly more careful, but still experimental, analysis. Figure 1 reports the results of some of our simulations, for three different choices of M. The corresponding values for α are $0.95 < \alpha < 1$ for the first set of simulations ($M = \sqrt{N/2}$), and $0.96 < \alpha < 0.99$ respectively $0.98 < \alpha < 1.02$ for $M = \sqrt{N}$ and $M = \sqrt{N}/2$. For the three cases, we compute areas under the graphs in Figure 1 using very simple numerical methods (summing the areas of rectangles coming from the data points). The areas lie in the intervals, respectively, $[0.577, 0.581]$, $[0.552, 0.569]$ and $[0.595, 0.617]$; the lower limit is the sum of rectangles under the curve and the upper limit is the sum of rectangles above the curve. For these values, the corresponding running times of the algorithm is $c\sqrt{N}$ where the constant c lies in, respectively, $[1.11, 1.27]$, $[1.17, 1.31]$ and $[1.09, 1.28]$. These results suggest $M = \sqrt{N/2}$ is a good choice, and that the constant should be close to 1.2. It is an open problem to give a complete and rigorous analysis of the grumpy-giants algorithm.

The above analysis does not allow us to give a very precise conjecture on the running time of the grumpy-giants algorithm. Instead we performed some computational experiments to get an estimate of its average-case performance. Details of the experiments are given in Appendix A. The results are listed in Table 2 (these results are for $M = \sqrt{N/2}$). In Table 3 we write the value 1.25, which seems to be a reasonable conjecture. We write 3α for the worst-case cost,

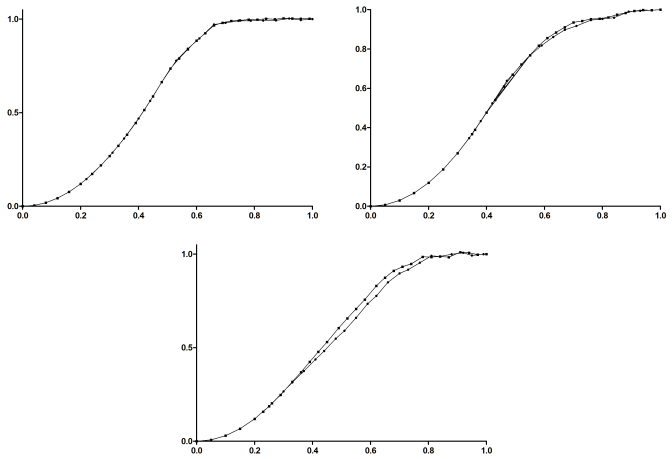

Fig. 1. Graph of $c(t)t^2$ for $t \in [0,1]$ obtained by simulations for the three values $M = \sqrt{N/2}, M = \sqrt{N}$ and $M = \frac{1}{2}\sqrt{N}$. The multiple lines denote the results for different experiments coming from different choices of N.

which seems a safe upper bound. Overall, our analysis and experiments support the claim by Bernstein and Lange [2] that the grumpy-giant algorithm has better complexity than Pollard rho. Our work also confirms that their suggested value $M = \sqrt{N}/2$ is a good choice.

Bits	#Elliptic Curves	#DLPs per Curve	average value for c	standard deviation
28	100	10000	1.2579	0.0083
29	100	10000	1.2533	0.0064
30	100	10000	1.2484	0.0062
31	100	10000	1.2517	0.0067
32	100	10000	1.2736	0.0054

Table 2. Results of experiments with the grumpy-giants algorithm without negation.

We remark that Pollard's result on interleaved BSGS can also be obtained using this theoretical argument: We have $\alpha = 1$, $c(t) = 1$, and the constant in the running time is $2(1 - \int_0^1 t^2 dt) = 2(1 - 1/3) = 4/3$.

2.2 Summary

The known results, and new results of our paper, are summarised in Table 3. Out of interest we include the values $\sqrt{\pi/2}$ (and $\sqrt{\pi/4}$ when using negation) for the average-case number of group operations of Pollard rho using distin-

guished points. We also include entries for versions of the Gaudry-Schost algorithm (which performs better than the Pollard kangaroo method). The $+o(1)$ term accounts for many issues (e.g., initial set-up costs, "un-needed" group elements computed in the final block, effects from the number of partitions used to define pseudorandom walks, expected time to get a distinguished point, dealing with cycles in random walks on equivalence classes, etc).

Algorithm	Average-case	Worst-case
Textbook BSGS [19]	1.5	2.0
Textbook BSGS optimised for average-case [18]	1.414	2.121
Pollard interleaving BSGS [17]	1.333	2.0
Grumpy giants [2]	1.25^*	≤ 3
Pollard rho using distinguished points [20]	1.253	∞
Gaudry-Schost [8]	1.661	∞
BSGS with negation	1.0	1.5
Pollard interleaving BSGS with negation	0.943	1.414
Grumpy giants with negation	0.9^*	≤ 2.7
Pollard rho using negation [1, 21]	$0.886(1 + o(1))$	∞
Gaudry-Schost using negation [7]	1.36	∞
Interleaved BSGS with block computation	0.38	0.57
Grumpy giants with block computation	0.36^*	≤ 1.08
Pollard rho with Montgomery trick	0.47	∞
Gaudry-Schost with Montgomery trick	0.72	∞

Table 3. The table lists constants c such that the named algorithm requires $(c + o(1))\sqrt{N}$ group operations for large enough groups of size N. The first block lists algorithms for general groups, and all these results are known (see Section 2). The values for the grumpy-giant algorithm (marked by an asterisk) are conjectural and the values for the rho and Gaudry-Schost algorithm are heuristic. The second block lists algorithms for groups having an efficiently computable inversion (see Section 3). Some of these results are new (the first one appears as an exercise in the first author's textbook). The third block lists algorithms that exploit efficient inversion as well as our main observation, and these results are all new (see Section 5).

3 Using Efficient Inversion in the Group

It is known [9, 23, 1, 22] that the negation map can be used to speed up the computation of elliptic curve discrete logarithms. Recall that if $P = (x_P, y_P)$ is a point on a Weierstrass model of an elliptic curve then $-P$ has the same x-coordinate. As mentioned, one can speed up the Pollard rho algorithm for the ECDLP by doing a pseudorandom walk on the set of equivalence classes under the equivalence relation $P \sim \pm P$. More generally, the idea applies to any group for which an inversion can be computed more efficiently than a general group operation (e.g., in certain algebraic tori).

One can also speed up the baby-step giant-step algorithm. We present the details in terms of elliptic curves. Let $M = \lceil \sqrt{N} \rceil$. Then $n = \pm n_0 + Mn_1$, where $-\frac{M}{2} \leq n_0 < \frac{M}{2}$ and $0 \leq n_1 < M$. Compute $M/2$ baby steps $n_0 P$ for $0 \leq n_0 \leq M/2$. Store the values $(x(n_0 P), n_0)$ in a sorted structure. Next, compute $P' = MP$ and the giant steps $Q - n_1 P'$ for $n_1 = 0, 1, \ldots$. For each point computed we check if its x-coordinate lies in the sorted structure. If we have a match then

$$Q - n_1 P' = \pm n_0 P$$

and so $Q = (\pm n_0 + Mn_1)P$ and the ECDLP is solved.

Lemma 1. *The average-case running time of the baby-step giant-step using efficient inversion is $\sqrt{N}$ group operations. The worst-case running time is $1.5\sqrt{N}$ group operations.*

Proof. Computing the baby steps requires $M/2$ group operations and computing the giant steps requires, on average $M/2$ group operations. Hence the total number of group operations is, on average, $M = \sqrt{N}$. In the worst case one performs all M giant steps.

Compared with the original "textbook" BSGS algorithm optimised for the average case, we have reduced the running time by a factor of $\sqrt{2}$. This is exactly what one would expect.

Readers may be confused about whether we are fully exploiting the $\pm$ sign. To eliminate confusion, note that a match $x(Q - n_1 P') = x(n_0 P)$ is the same as $\pm(Q - n_1 P') = \pm n_0 P$, and this reduces to the equation $Q - n_1 P' = \pm n_0 P$.

One can of course combine this trick with Pollard's interleaving idea. Take now $M = \sqrt{2N}$ so that $n = \pm n_0 + Mn_1$ with $0 \leq n_0, n_1 \leq M/2$. The algorithm computes the lists of baby-steps $\{x(n_0 P)\}$ and giant steps $\{x(Q - n_1 P')\}$ in parallel until the first match.

Lemma 2. *The average-case running time of the interleaved baby-step giant-step using efficient inversion is $(2\sqrt{2}/3)\sqrt{N}$ group operations. The worst-case running time is $\sqrt{2N}$ group operations.*

Proof. The worst-case number of steps (this is an interleaved algorithm so a "step" now means computing one baby step and one giant step) is $M/2$, giving a total cost of $2(M/2) = M = \sqrt{2N}$ group operations. By Pollard's analysis [18], generating both walks in parallel leads to a match in $4(M/2)/3$ group operations on average. The leading constant in the running time is therefore $2\sqrt{2}/3 \approx 0.9428$.

We can also prove this result using the method from Section 2.1. The number of points Q whose DLP can be solved after L steps is $2L^2$ (since we have $Q = (\pm i + jM)P$) and so $c(t) = 2$ for $0 \leq t < \alpha$. When $M = \sqrt{2N}$ then $\alpha = \sqrt{1/2}$ and so the constant in the running-time is

$$2\left(\alpha - \int_0^{\alpha} 2t^2 dt\right) = 2(\alpha - 2\alpha^3/3) = 0.9428.$$

One might wonder if there is a better way to organise the interleaving. Since one gets two baby-steps for each group operation it would be tempting to take more giant steps on average than baby steps. However, the goal is to maximise the number $2L_1L_2$ of points Q solved after $L_1 + L_2$ steps (where L_1 and L_2 denote the number of group operations spent computing baby-steps and giant-steps respectively). This boils down to maximising $f(x, y) = 2xy$ subject to $x + y = 1$, which is easily seen to have the solution $x = y = 1/2$. Hence, the optimal way to organise the interleaving is to use the same number of group operations for baby-steps and giant-steps for each time L.

3.1 Grumpy Giants with Negation

One can consider the grumpy giants method where matches are detected using $x(iP)$, $x(Q+jMP)$, $x(2Q-k(M+1)P)$. This algorithm is not mentioned in [2]. Hence, one of the contributions of our paper is to develop the algorithm in this case and analyse it.

The first task is to count "slopes". After L steps we have computed three lists $\{x(iP) : 0 \leq i < L\}$, $\{x(Q + jMP) : 0 \leq j < L\}$ and $\{x(2Q - k(M + 1)P) : 0 \leq k < L\}$. A collision between the first two lists implies $Q + jMP = \pm iP$ and so $Q = (\pm i - jM)P$. A collision between the first and third lists implies $2Q - k(M + 1)P = \pm iP$ and so $Q = 2^{-1}(\pm i + k(M + 1))P$. A collision between the second and third lists implies either $Q + jMP = 2Q - k(M + 1)P$ or $Q + jMP = -2Q + k(M + 1)P$. Hence we have either $Q = (jM + k(M + 1))P$ or $Q = 3^{-1}(k(M + 1) - jM)P$. The relevant quantity to consider is the size of the union

$$\begin{aligned}\mathcal{L}_L = &\{\pm i - jM \pmod N : 0 \leq i, j < L\} \\ &\cup \{2^{-1}(\pm i + k(M + 1)) \pmod N : 0 \leq i, k < L\} \\ &\cup \{jM + k(M + 1) \pmod N : 0 \leq j, k < L\} \\ &\cup \{3^{-1}(k(M + 1) - jM) \pmod N : 0 \leq j, k < L\}.\end{aligned}$$

We follow the heuristic analysis given in Section 2.1. Let α be such that the algorithm halts after at most $\alpha\sqrt{N}$ steps. We also define $c(t)$ to be such that the number of elements in $\mathcal{L}_L$ is equal to $c(L/\sqrt{N})L^2$. We have conducted simulations, for various values of N (again, the values of α and $c(t)$ depend on N and on the choice of M). A typical example (this is for $N \approx 2^{28}$ and $M = \sqrt{N/2}$) has $0.87 < \alpha < 0.92$ and the values for $c(t)$ as in Table 4.

t	0	0.15	0.30	0.46	0.61	0.76	0.91
$c(t)$	6.00	5.76	5.47	4.10	2.56	1.72	1.20

Table 4. Size of set $\mathcal{L}_L$ written as $c(t)L^2$ where $t = L/\sqrt{N}$.

Figure 2 reports on our simulations, again by computing $\mathcal{L}_L$ exactly for various choices of N. Our experiments suggest that $M = \sqrt{N/2}$ is the best choice, for which $0.9 \leq \alpha \leq 0.96$. The integral under the curve, computed using numerical methods, is in the interval $[0.58, 0.60]$, giving a running time $c\sqrt{N}$ for $3(0.9 - 0.6) = 0.9 \leq c \leq 1.14 = 3(0.96 - 0.58)$.

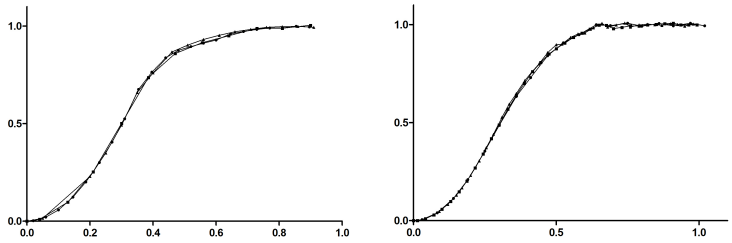

Fig. 2. Graph of $c(t)t^2$ for $t \in [0, 1]$ obtained by simulations, for grumpy giants method using x-coordinates, with $M = \sqrt{N/2}$ and $M = \sqrt{N}$.

Table 5 gives our experimental results. Details about the experiment are given in Appendix A. The average running time seems to be around $0.9\sqrt{N}$, so that is the value we put in Table 3. Note that $1.25/0.9 \approx 1.39 < \sqrt{2}$ so we have not obtained a speed-up by a factor of approximately $\sqrt{2}$. Hence, this algorithm seems to be *not* faster than Pollard rho on equivalence classes $[1, 21]$. However, it does still seem to be an improvement over Pollard's interleaved version of the standard BSGS algorithm.

Bits	#Elliptic Curves	#DLPs per Curve	average value for c	standard deviation
28	100	10000	0.8926	0.0077
29	100	10000	0.9053	0.0061
30	100	10000	0.8961	0.0073
31	100	10000	0.9048	0.0068
32	100	10000	0.9207	0.0065

Table 5. Results of experiments with the grumpy-giants algorithm exploiting efficient inversion.

4 Computing Lists of Elliptic Curve Points Efficiently

The BSGS and Pollard rho algorithms require computing elliptic curve operations in affine coordinates rather than projective coordinates (otherwise collisions

are not detected). Affine arithmetic requires inversion in the field, and this operation is considerably more expensive than multiplications in the field or other operations. A standard technique is to use the Montgomery trick [15] to offset the cost of inversions, but this technique in the context of baby-step giant-step algorithms has not previously been fully explored.

4.1 Elliptic Curve Group Law for Affine Weierstrass Equations

Let

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

be a Weierstrass equation for an elliptic curve and let $X = (x_1, y_1)$ and $Y = (x_2, y_2)$ be two general points on E (in other words, we assume $Y \neq \pm X$). Recall that $-X = (x_1, -y_1 - a_1x_1 - a_3)$. Let $(x_3, y_3) = X + Y$. (We are interested in affine coordinates rather than projective coordinates as the BSGS algorithm cannot be used with projective coordinates.) Then

$$\begin{aligned}
\lambda &= \frac{y_2 - y_1}{x_2 - x_1}, \\
x_3 &= \lambda^2 + a_1\lambda - x_1 - x_2 - a_2, \\
y_3 &= -\lambda(x_3 - x_1) - y_1 - a_1x_3 - a_3.
\end{aligned}$$

Let $\mathcal{M}$, $\mathcal{S}$ and $\mathcal{I}$ denote the cost of a multiplication, squaring and inversion in $\mathbb{F}_q$ respectively. We ignore the cost of multiplications by fixed constants such as a_1, since these are often chosen to be 0 or 1. Then, the cost of point addition over $E(\mathbb{F}_q)$ is close to $\mathcal{I} + 2\mathcal{M} + \mathcal{S}$. One can check that point doubling requires $\mathcal{I} + 2\mathcal{M} + 2\mathcal{S}$. Therefore the main cost of point addition is the computation of field inversion: $(x_2 - x_1)^{-1}$. Extensive experiments from [6] suggest that a realistic estimate of the ratio $\mathcal{I}/\mathcal{M}$ of inversion to multiplication cost is 8 (or higher).

Now, note that $(x_4, y_4) = X - Y = X + (-Y) = (x_1, y_1) + (x_2, -y_2 - a_1x_2 - a_3)$ as follows:

$$\begin{aligned}
\lambda &= \frac{-y_2 - a_1x_2 - a_3 - y_1}{x_2 - x_1}, \\
x_4 &= \lambda^2 + a_1\lambda - x_1 - x_2 - a_2, \\
y_4 &= -\lambda(x_3 - x_1) - y_1 - a_1x_3 - a_3.
\end{aligned}$$

We see that we can re-use the inversion $(x_2 - x_1)^{-1}$ and hence compute $X - Y$ with merely the additional costs $2\mathcal{M} + \mathcal{S}$.

Taking $\mathcal{I} = 8\mathcal{M}$ and $\mathcal{S} = 0.8\mathcal{M}$ the cost of point "combined $\pm$ addition" is $\mathcal{I} + 4\mathcal{M} + 2\mathcal{S} = 13.6\mathcal{M}$. Compared with the cost $2(\mathcal{I} + 2\mathcal{M} + \mathcal{S})$ of two additions, we have a speedup by a factor of

$$\frac{\mathcal{I} + 4\mathcal{M} + 2\mathcal{S}}{2(\mathcal{I} + 2\mathcal{M} + \mathcal{S})} \approx \frac{(8 + 4 + 1.6)\mathcal{M}}{2(8 + 2 + 0.8)\mathcal{M}} \approx 0.63.$$

If the cost of inversion is much higher (e.g., $\mathcal{I} \approx 20\mathcal{M}$) then the speedup is by a factor close to 2.

Elliptic curves over non-binary finite fields can be transformed to Edwards form $x^2+y^2 = c^2(1+x^2y^2)$, with $(0, c)$ as identity element. Edwards curves can be efficient when using projective coordinates, but we need to use affine coordinates for BSGS algorithms. Let $X = (x_1, y_1)$ and $Y = (x_2, y_2)$, the addition law is

$$X + Y = \left(\frac{x_1y_2 + y_1x_2}{c(1 + x_1x_2y_1y_2)}, \frac{y_1y_2 - x_1x_2}{c(1 - x_1x_2y_1y_2)} \right).$$

The cost of addition is therefore $\mathcal{I} + 6\mathcal{M}$. Since $-(x, y) = (-x, y)$, we have

$$X - Y = \left(\frac{x_1y_2 - y_1x_2}{c(1 - x_1x_2y_1y_2)}, \frac{y_1y_2 + x_1x_2}{c(1 + x_1x_2y_1y_2)} \right).$$

We can re-use the inversions and compute $X - Y$ with merely the additional costs $2\mathcal{M}$. So the speedup is

$$\frac{\mathcal{I} + 8\mathcal{M}}{2(\mathcal{I} + 6\mathcal{M})} \approx \frac{8 + 8}{2(8 + 6)} \approx 0.57.$$

The speedup looks good, but note that the cost of combined $\pm$ addition is $\mathcal{I}+8\mathcal{M}$ which is slower than the $\mathcal{I} + 4\mathcal{M} + 2\mathcal{S}$ when using Weierstrass curves.

The observation that one can compute $X + Y$ and $X - Y$ efficiently simultaneously was first introduced by Wang and Zhang [22], and they tried to apply it to the Pollard rho algorithm. This seems not to be effective, since the Pollard rho algorithm uses random walks that go in a "single direction", whereas the combined operation $X+Y$ and $X-Y$ gives us steps in "two different directions".

4.2 Computing Lists of Points Using the Montgomery Inversion Trick

The naive method to compute a list $\mathcal{L} = \{S + [i]T : 0 \leq i < M\}$ of elliptic curve points is to perform M point additions in serial, taking time roughly $M(\mathcal{I} + 2\mathcal{M} + \mathcal{S})$.

The well-known Montgomery simultaneous modular inversion trick [15] allows to invert k field elements using one inversion and $3(k-1)$ multiplications in the field. Such ideas have been used in the context of Pollard rho: one performs k pseudorandom walks in parallel, and the group operations are computed in parallel by using simultaneous modular inversion. However the BSGS algorithm as usually described is inherently serial, so this approach does not seem to have been proposed in the context of BSGS.

The aim of this section is to describe efficient ways to compute a list of points of the form $\mathcal{L} = \{S + [i]T : 0 \leq i < M\}$. The serial method is to compute M point additions, each involving a single inversion.

A way to make this computation somewhat parallel is the following. Fix an integer k and set $M' = \lceil M/k \rceil$. The list will be computed as $S + (jM' + i)T$ for $0 \leq j < k$, $0 \leq i < M'$. One can then compute $\mathcal{L}$ using Algorithm 1.

Algorithm 1 Compute list of points using parallel computation

Input: Points S and T, integer M
Output: $\mathcal{L} = \{S + [i]T : 0 \leq i < M\}$

1: Choose k
2: Set $M' = \lceil M/k \rceil$
3: Compute $T' = (M')T$
4: Compute (serial) $T_0 = S,\ T_j = T_{j-1} + T'$ with $1 \leq j < k$
5: $\mathcal{L} = \{T_0, T_1, \ldots, T_{k-1}\}$
6: **for** $i = 0$ to $M' - 1$ **do**
7: Compute in parallel using Montgomery trick $T_j = T_j + T$ with $0 \leq j < k$
8: $\mathcal{L} = \mathcal{L} \cup \{T_0, T_1, \ldots, T_{k-1}\}$
9: **end for**

Lemma 3. *Algorithm 1 runs in time proportional to*

$$(2\log_2(M/k) + k - 1)(\mathcal{I} + 2\mathcal{M} + 2\mathcal{S}) \ + \ (M/k)(\mathcal{I} + (5k-3)\mathcal{M} + k\mathcal{S}).$$

Ignoring precomputation, this is roughly $(M/k)\mathcal{I} + 5M\mathcal{M} + M\mathcal{S}$.

Proof. Step 3 takes up to $2\log_2(M/k)$ group operations (including both doubles and adds, so we use the cost $\mathcal{I} + 2\mathcal{M} + 2\mathcal{S}$), while step 4 is k group operations (in serial). The loop contains k additions performed in parallel, and so benefits from the simultaneous modular inversion trick. The cost of each iteration is $\mathcal{I} + 3(k-1)\mathcal{M}$ for the modular inversion, followed by $k(2\mathcal{M} + \mathcal{S})$ for the rest of the elliptic curve addition operations. □

Taking $\mathcal{I} = 8\mathcal{M}$ and $\mathcal{S} = 0.8\mathcal{M}$, the speedup from the naive algorithm to Algorithm 1 is roughly

$$\frac{(M/k)\mathcal{I} + 5M\mathcal{M} + M\mathcal{S}}{M(\mathcal{I} + 2\mathcal{M} + \mathcal{S})} = \frac{M(8/k + 5 + 0.8)\mathcal{M}}{M(8 + 2 + 0.8)\mathcal{M}} = \frac{8/k + 5.8}{10.8}$$

which is about 0.63 when $k = 8$ and tends to 0.53 as $k \to \infty$. (Of course we cannot take very large k, as then the precomputation dominates.)

In other words, we can significantly reduce the overall cost of performing inversions, by taking a few extra multiplications. Next we explain how to do even better: we can essentially halve the number of inversions with only a few more additional multiplications.

4.3 Improved Algorithm for Computing Lists of Points

We now combine the $X \pm Y$ idea with simultaneous modular inversion. Both tricks replace an inversion with a couple of multiplications. Recall from Algorithm 1 that we are already computing the list $\mathcal{L}$ as $\{S + (jM' + i)T : 0 \leq j < k, 0 \leq i < M'\}$ where the points $T_j = S + jM'$ are precomputed and the additions $+T$ are parallelised with simultaneous modular inversion.

The idea is to also precompute $T'' = 3T$. For $0 \le i < M'/3$ we compute the sums $T_j \pm T$ and then $T_j + T''$. In other words, we compute three consecutive points $T_j + (3i-1)T, T_j + (3i)T, T_j + (3i+1)T$ using fewer than three full group operations.

To get a greater speedup we use more of the combined operations. Hence instead of using $T'' = [3]T$ we can take $T'' = [5]T$. Then in the i-th iteration we compute $T_j = T_j + T'' = T_j + 5T$ and then also $T_j \pm T$ and $T_j \pm 2T$ (where $2T$ is precomputed). In other words, we compute a block of 5 consecutive points $T_j + (5i-2)T$, $T_j + (5i-1)T$, $T_j + (5i)T$, $T_j + (5i+1)T$ and $T_j + (5i+2)T$ with one addition and two "combined $\pm$ additions". Extending this idea one can use $T'' = (2\ell+1)T$, and then precompute $2T, \ldots, \ell T$. We give the details as Algorithm 2.

Algorithm 2 Compute list of points using parallel computation

Input: Points S and T, integer M
Output: $\mathcal{L} = \{S + [i]T : 0 \le i < M\}$

1: Choose k and ℓ
2: Set $M' = \lceil M/k \rceil$ and $M'' = \lceil M/(k(2\ell+1)) \rceil$
3: Compute (serial) $2T, 3T, \ldots, \ell T$, $T'' = (2\ell+1)T$ and $T' = (M')T$
4: Compute (serial) $T_0 = S + \ell T,\ T_1 = T_0 + T',\ T_2 = T_1 + T', \ldots,\ T_{k-1} = T_{k-2} + T'$
5: $\mathcal{L} = \{T_0, T_1, \ldots, T_{k-1}\}$
6: **for** $i = 0$ to $M'' - 1$ **do**
7: **for** $u = 1$ to ℓ **do**
8: Compute in parallel $T_0 \pm uT,\ T_1 \pm uT,\ \ldots,\ T_{k-1} \pm uT$
9: Add the $2k$ new points to $\mathcal{L}$
10: **end for**
11: Compute in parallel $T_0 = T_0 + T'',\ T_1 = T_1 + T'',\ \ldots,\ T_{k-1} = T_{k-1} + T''$
12: $\mathcal{L} = \mathcal{L} \cup \{T_0, T_1, \ldots, T_{k-1}\}$
13: **end for**

Lemma 4. *Algorithm 2 runs in time proportional to*

$$(\ell + k + 2 + 2\log_2(M/k))(\mathcal{I} + 2\mathcal{M} + \mathcal{S})$$
$$+\ M/(k(2\ell+1))((\ell+1)\mathcal{I} + (7k\ell + 5k - 3(\ell+1))\mathcal{M} + (2k\ell + k)\mathcal{S}).$$

Ignoring precomputation, this is roughly $(M/(k(2\ell+1)))\mathcal{I} + \frac{7}{2}M\mathcal{M} + M\mathcal{S}$.

Proof. The precomputation in step 3 is at most $\ell + 2 + 2\log_2(M/k)$ elliptic curve operations. Similarly, step 4 is k group operations (in serial).

The main loop has $M/(k(2\ell+1))$ iterations. Within that there is a loop of ℓ iterations that performs one simultaneous modular inversion (cost $\mathcal{I} + 3(k-1)\mathcal{M}$) followed by the remaining $2\mathcal{M} + \mathcal{S}$ for $2k$ point additions. After that loop there is a further simultaneous modular inversion and k additions. So the cost of each iteration of the main loop is

$$\ell(\mathcal{I} + 3(k-1)\mathcal{M} + 2k(2\mathcal{M} + \mathcal{S})) + (\mathcal{I} + 3(k-1)\mathcal{M} + k(2\mathcal{M} + \mathcal{S})).$$

The result follows. □

The important point is that we have reduced the number of inversions further and reduced the number of multiplications. Again, a naive argument suggests a speedup by up to $(3.5 + 0.8)/(10.8) = 0.39$ compared with the standard serial approach, or $4.3/5.8 = 0.74$ compared with using Algorithm 1.

5 Application to BSGS Algorithm

It is now straightforward to combine the ideas of Section 4 to the various baby-step giant-step algorithms for the DLP. All such algorithms involve computing two or three lists of points, and such operations can be sped-up by the general techniques in Section 4. It follows that we get a saving by a constant factor for all such algorithms (see Algorithm 3 for an example of the standard BSGS algorithm using efficient inversion; the details for the grumpy-giants algorithm are similar). This explains the last rows of Table 3.

However, note that there is again a tradeoff in terms of the size of blocks (and hence the values of (k, ℓ)) and the running time. The BSGS algorithm will compute a block of points in parallel and then test for matches between the lists. Hence, the algorithm will usually perform more work than necessary before detecting the first match and solving the DLP. It follows that the average number of group operations performed is increased by a small additive factor proportional to $k\ell$ (this contributes to the $o(1)$ term in the running time).

Algorithm 3 Interleaved baby-step giant-step algorithm for elliptic curves, exploiting efficient inversion (i.e., using x-coordinates), and computing points in blocks.

Input: Initial points P and Q
Output: (n_0, n_1) with $Q = nP$, $M = \lceil \sqrt{N} \rceil$ and $n = \pm n_0 - M n_1$

1: Precompute $P' = MP$ and other points needed for efficient computation of blocks of elliptic curve points using Algorithm 2
2: $S \leftarrow Q$
3: **while** True **do**
4: Use Algorithm 2 to compute block of baby steps $(x(n_0 P), n_0)$ for $0 \leq n_0 \leq M/2$ and store in easily searched structure $\mathcal{L}_1$
5: Use Algorithm 2 to compute block of giant steps $(x(Q - n_1 P'), n_1)$ for $0 \leq n_0 \leq M/2$ and store in easily searched strucutre $\mathcal{L}_2$
6: **if** $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$ **then**
7: Determine corresponding values of n_0 and n_1
8: Determine sign of $\pm n_0 P = Q - n_1 P'$
9: Return $\pm n_0 + M n_1$
10: **end if**
11: **end while**

Lemma 4 shows that one can speed-up computation of a list of M points by using blocks of length ℓ from $M(\mathcal{I} + 2\mathcal{M} + \mathcal{S})$ (using Weierstrass curves) to

$(M/(k(2\ell+1)))\mathcal{I} + \frac{7}{2}M\mathcal{M} + M\mathcal{S}$. Taking $\mathcal{I} = 8\mathcal{M}, \mathcal{S} = 0.8\mathcal{M}$ and $k = \ell = 8$ this is a speedup by

$$\frac{M(\mathcal{I}/136 + 7\mathcal{M}/2 + \mathcal{S})}{M(\mathcal{I} + 2\mathcal{M} + \mathcal{S})} \approx \frac{8/136 + 7/2 + 0.8}{8 + 2 + 0.8} \approx 0.404.$$

Hence, all the results in Table 3 can be multiplied by 0.4, and that is what we have done to get the first two rows in the final block (e.g., $0.9 \cdot 0.4 = 0.36$ is the claimed constant in the running time of the grumpy-giants algorithm).

For comparison, we consider the Pollard method implemented so that k walks are performed in parallel using Montgomery's simultaneous modular inversion method. As we explained, this method gives an asymptotic speed-up of 0.53, meaning the constant in this parallel Pollard rho algorithm is $0.53 \cdot 0.886(1 + o(1)) = 0.47(1 + o(1))$. This value is the final entry in Table 3. Our conclusion is that, in this optimised context, both the interleaved BSGS and grumpy-giants algorithms are superior to Pollard rho.

6 Other Settings

There are other groups that have efficient inversion. For example, consider the group $G_{q,2}$ of order $q + 1$ in $\mathbb{F}_{q^2}^*$. This can be viewed as the group of elements in $\mathbb{F}_{q^2}^*$ of norm 1. A standard fact is that one can compute a product $h * g$ of two elements in $\mathbb{F}_{q^2}$ using three multiplications in $\mathbb{F}_q$ (and some additions). Now, when g lies in the group $G_{q,2}$ then computing g^{-1} is easy (since $gg^q = \mathrm{Norm}(g) = 1$, so $g^{-1} = g^q$ is got by action of Frobenius. So if $g = u + v\theta$ then $g^{-1} = u - v\theta$. It follows that one can compute hg and hg^{-1} in four multiplications, only one multiplication more than computing hg. So our BSGS improvement can be applied in this setting too. The speedup is by a factor of $4/6 = 2/3$ since we need 4 multiplications to do the work previously done by 6 multiplications. This is less total benefit than in the elliptic curve case, but it still might be of practical use.

The group $G_{q,2}$ is relevant for XTR. One way to represent XTR is using elements of $G_{q^3,2}$. We refer to Section 3 of [11] for discussion. The cost of an "affine" group operation in the torus representation of $G_{q,2}$ is $\mathcal{I} + 2\mathcal{M}$, which is worse than the cost $3\mathcal{M}$ mentioned already. So when implementing a BSGS algorithm it is probably better to use standard finite field representations.

7 Conclusion

Our work is inspired by the idea that the negation map can be used to speed up the computation of elliptic curve discrete logarithms. We explain how to compute lists of consecutive elliptic curve points efficiently, by exploiting Montgomery's trick and also the fact that for any two points X and Y we can efficiently get $X - Y$ when we compute $X + Y$ by sharing a field inversion. We use these ideas to speed up the baby-step giant-step algorithm. Compared to the previous

approaches we achieve a significant speedup for computing elliptic curve discrete logarithms.

We also give a new method to analyse the grumpy-giants algorithm, and describe and analyse a variant of this algorithm for groups with efficient inversion.

The new algorithms, like the original, have low overhead but high memory. This means that our algorithms are useful only for discrete logarithm problems small enough for the lists to fit into fast memory. For applications such as [4] that require solving DLP instances in a short interval, the best method by far is Pollard's interleaved BSGS algorithm together with our idea of computing points in blocks; the grumpy-giants algorithm is not suitable for this problem and the Pollard kangaroo and Gaudry-Schost methods will take about twice the time.

Acknowledgements

The authors thank Siouxsie Wiles for assistance with the graphs, and an anonymous referee for helpful comments.

References

1. Daniel J. Bernstein, Tanja Lange and Peter Schwabe, "On the Correct Use of the Negation Map in the Pollard rho Method", in D. Catalano et al (eds.), PKC 2011, Springer LNCS 6571 (2011) 128–146.
2. D. J. Bernstein and T. Lange, "Two grumpy giants and a baby", in E. W. Howe and K. S. Kedlaya (eds.), Proceedings of the Tenth Algorithmic Number Theory Symposium, MSP, Vol. 1 (2013) 87–111.
3. D. J. Bernstein and T. Lange, "Computing small discrete logarithms faster", in S. D. Galbraith and M. Nandi (eds.), INDOCRYPT 2012, Springer Lecture Notes in Computer Science 7668 (2012) 317–338.
4. D. Boneh, E. Goh and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts", in J. Kilian (ed.), Theory of Cryptography-TCC 2005, Springer LNCS 3378 (2005) 325-341.
5. M. Chateauneuf, A. C. H. Ling and D. R. Stinson, "Slope packings and coverings, and generic algorithms for the discrete logarithm problem", Journal of Combinatorial Designs, Vol. 11, No. 1 (2003) 36–50.
6. K. Fong, D. Hankerson, J. Lopez, A. Menezes, "Field Inversion and Point Halving Revisited", IEEE Trans. Computers 53(8) (2004) 1047–1059.
7. S. D. Galbraith and R. S. Ruprai, "Using equivalence classes to speed up the discrete logarithm problem in a short interval", in P. Nguyen and D. Pointcheval (eds.), PKC 2010, Springer LNCS 6056 (2010) 368–383.
8. S. D. Galbraith, J. M. Pollard and R. S. Ruprai, "Computing discrete logarithms in an interval", Math. Comp., **82**, No. 282 (2013) 1181–1195.
9. R. Gallant, R. Lambert and S. Vanstone, "Improving the parallelized Pollard lambda search on binary anomalous curves", Mathematics of Computation, Volume 69 (1999) 1699–1705.
10. P. Gaudry and E. Schost, "A low-memory parallel version of Matsuo, Chao and Tsujii's algorithm, in D. A. Buell (ed.), ANTS VI, Springer LNCS 3076 (2004) 208–222.

11. R. Granger, D. Page and M. Stam, "On Small Characteristic Algebraic Tori in Pairing-Based Cryptography", LMS Journal of Computation and Mathematics, **9** (2006) 64–85.
12. R. Henry, K. Henry and I. Goldberg, "Making a nymbler Nymble using VERBS", in M. J. Atallah and N. J. Hopper (eds.), PETS 2010, Springer LNCS 6205 (2010) 111–129.
13. N. Koblitz, "Elliptic curve cryptosystems", Mathematics of Computation, 48 (1987) 203–209.
14. V. Miller, "Use of elliptic curves in cryptography", in H. C. Williams (ed.), Crypto '85, Springer LNCS 218 (1986) 417–426.
15. P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization", Mathematics of Computation 48 (1987) 243–264.
16. V. I. Nechaev, "Complexity of a determinate algorithm for the discrete logarithm", Mathematical Notes **55**, no. 2 (1994) 165–172.
17. J. M. Pollard, "Monte Carlo methods for index computation (mod p)", Math. Comp. **32**, no. 143 (1978) 918–924.
18. J. Pollard, "Kangaroos, Monopoly and discrete logarithms", Journal of Cryptology 13 (2000) 437–447.
19. D. Shanks, "Five number-theoretic algorithms", Proceedings of the Second Manitoba Conference on Numerical Mathematics, Congressus Numerantium, No. VII, Utilitas Math., Winnipeg, Man. (1973) 51–70.
20. P. van Oorschot and M. Wiener, Parallel collision search with cryptanalytic applications, Journal of Cryptology, vol. 12, no. 1 (1999) 1–28.
21. P. Wang and F. Zhang, "Computing Elliptic Curve Discrete Logarithms with the Negation Map", Information Sciences, Vol. 195 (2012) 277–286.
22. P. Wang and F. Zhang, "Improving the Parallelized Pollard Rho Method for Computing Elliptic Curve Discrete Logarithms", in The 4th International Conference on Emerging Intelligent Data and Web Technologies (EIDWT-2013), 2013.
23. M. Wiener and R. Zuccherato, "Faster attacks on elliptic curve cryptosystems", in S. E. Tavares and H. Meijer (eds.), Selected Areas in Cryptography '98, Springer LNCS 1556 (1998) 190–120.

A Implementation and Experimental Evidence

We implemented the grumpy-giants algorithm and the grumpy-giants algorithm with efficient inversion on elliptic curve groups over prime fields using SAGE. The purpose of our experiments is to evaluate the expected numbers of steps until a match is found with different elliptic curves and random DLP instances.

One can perform simulations in the group $(\mathbb{Z}_N, +)$, but to test any bias coming from elliptic curve group orders we performed simulations using elliptic curves. More precisely, for each integer $i \in [28, 32]$ we generated 100 elliptic curves, where each of them have a subgroup G of prime order N, such that $N \in [2^i, 2^{i+1}]$. To do this we randomly chose a prime number q in a certain range. Then we randomly chose parameters $a, b \in \mathbb{F}_q$, which determine the elliptic curve $E_{a,b}$ over $\mathbb{F}_q$. We then checked whether the order of group $E_{a,b}(\mathbb{F}_q)$ had large prime factor N in the range $[2^i, 2^{i+1}]$. If not, repeat the above procedures. This gave us a prime order subgroup G of $E_{a,b}(\mathbb{F}_q)$. Then we chose a generator P of G. We then chose 10000 random points $Q \in G$ and ran the grumpy-giants

algorithm (or its variant) on the DLP instance (P, Q) and counted the number of group operations performed until each DLP was solved. The average number of group operations over those 10000 trials was computed and represented in the form $c\sqrt{N}$. Repeating this for each of the groups G gives a list of 100 values c, each of which is an estimate of the average-case running time of the algorithm in one of the groups G. Then we computed the mean and standard deviation of the 100 estimates c, giving us a good estimate of the average case complexity $c_i\sqrt{N}$ of the algorithm for $(i+1)$-bit group orders. This is the value reported. Table 2 gives the mean values obtained for c_i over such experiments with the original grumpy-giants algorithm with $M = \sqrt{N/2}$. Table 5 gives the mean values obtained for the grumpy-giants algorithm using efficient inversion with $M = \sqrt{N/2}$.

B 4-giants Algorithm

In [2] a 4-giants algorithm is mentioned (two in one direction, two in the other, with computer-optimized shifts of the initial positions) as a variant of 2-giants algorithm. The paper did not describe the 4-giants algorithm in detail. However, we implemented a possible version of the 4-giants algorithm, and the experimental results do not seem to be as good as one may have expected. It seems that the 4-giants algorithm is not as efficient as the 2-giants algorithm.

More precisely, the baby steps are of the form n_0P for small values n_0. The first grumpy giant starts at Q and takes steps of size $P' = MP$ for $M \approx 0.5\sqrt{N}$. The second grumpy giant starts at $2Q$ and takes steps of size $P'' = -(M+1)P$. The third one starts at $3Q$ and takes steps of size $P''' = (M+2)P$. The fourth one starts at $4Q$ and takes steps of size $P'''' = -(M+3)P$. The algorithm is an interleaving algorithm in the sense that all five walks are done in parallel and stored in lists. At each step one checks for a match among the lists (a match between any two lists allows to solve the DLP). We performed experiments in the same framework as above and summarize the results in the following table. One sees that the constants are larger than the 1.25 of the 2-giants method.

Bits	#Elliptic Curves	#DLPs per Curve	average value for c
28	100	10000	1.2867
29	100	10000	1.3002
30	100	10000	1.2926
31	100	10000	1.2944
32	100	10000	1.3150

Table 6. Results of experiments with the 4-giants algorithm without negation.