ECC2K-130 on NVIDIA GPUs

Computing small discrete logarithms faster

Daniel J. Bernstein[1,2] and Tanja Lange[2]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7053, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, the Netherlands
tanja@hyperelliptic.org

Abstract. Computations of small discrete logarithms are feasible even in "secure" groups, and are used as subroutines in several cryptographic protocols in the literature. For example, the Boneh–Goh–Nissim degree-2-homomorphic public-key encryption system uses generic square-root discrete-logarithm methods for decryption. This paper shows how to use a small group-specific table to accelerate these subroutines. The cost of setting up the table grows with the table size, but the acceleration also grows with the table size. This paper shows experimentally that computing a discrete logarithm in an interval of order ℓ takes only $1.93 \cdot \ell^{1/3}$ multiplications on average using a table of size $\ell^{1/3}$ precomputed with $1.21 \cdot \ell^{2/3}$ multiplications, and computing a discrete logarithm in a group of order ℓ takes only $1.77 \cdot \ell^{1/3}$ multiplications on average using a table of size $\ell^{1/3}$ precomputed with $1.24 \cdot \ell^{2/3}$ multiplications.

Keywords: Discrete logarithms, random walks, precomputation.

1 Introduction

Fully homomorphic encryption is still prohibitively slow, but there are much more efficient schemes achieving more limited forms of homomorphic encryption. We highlight Freeman's variant [11] of the scheme by Boneh, Goh, and Nissim [7]. The Boneh–Goh–Nissim (BGN) scheme can handle adding arbitrary subsets of encrypted data, multiplying the sums, and adding any number of the products. Freeman's variant works in groups typically encountered in pairing-based protocols. The scheme is vastly more efficient than schemes handling unlimited numbers of additions and multiplications. Encryption takes only one exponentiation, as does addition of encrypted messages; multiplication takes a pairing computation.

The limitation to one level of multiplication means that polynomial expressions of degree at most 2 can be evaluated over the encrypted messages, but

* This work was supported by the National Science Foundation under grant 1018836, by the Netherlands Organisation for Scientific Research under grant 639.073.005, and by the European Commission under Contract ICT-2007-216676 ECRYPT II. Permanent ID of this document: b83446575069e4e1d5517415fa8a2421. Date: 2012.08.12.

this is sufficient for a variety of protocols. For example, [7] presented protocols for private information retrieval, elections, and generally universally verifiable computation. There are 377 citations of [7] so far, according to Google Scholar.

The BGN protocol does not have any built-in limit on the number of ciphertexts added, but it does take more time to decrypt as this number grows. The problem is that decryption requires computing a discrete logarithm, where the message is the unknown exponent. If this message is a sum of B products of sums of A input messages from the space $\{0, \ldots, M\}$, then the final message can be essentially anywhere in the interval $[0, (AM)^2B]$. This means that even if the space for the input messages is limited to bits $\{0, 1\}$, the discrete-logarithm computation needs to be able to handle the interval $[0, A^2B]$. For "random" messages the result is almost certainly in a much shorter interval, but most applications need to be able to handle non-random messages.

Boneh, Goh, and Nissim suggested using Pollard's kangaroo method for the discrete-logarithm computation. This method runs in time $\Theta(\ell^{1/2})$ for an interval of size ℓ. This bottleneck becomes quite troublesome as A and B grow.

For larger message spaces, Hu, Martin, and Sunar in [18] sped up the discrete-logarithm computation at the cost of expanding the ciphertext length and slowing down encryption and operations on encrypted messages. They suggested representing the initial messages by their residues modulo small coprime numbers $d_1, \ldots, d_j$ with $\prod d_i > (AM)^2B$, and encrypt these j residues separately. This means that the ciphertexts are j times as long and that each operation on the encrypted messages is replaced by j operations of the same type on the components. The benefit is that each discrete logarithm is limited to $[0, (Ad_i)^2B]$, which is a somewhat smaller interval. The original messages are reconstructed using the Chinese remainder theorem.

Contributions to BGN. This paper explains (Section 3) how to speed up computations of small discrete logarithms, i.e., discrete logarithms in small intervals. The speedup requires a one-time computation of a small group-specific table. The speedup grows as the table grows; an interesting special case is a table of size $\Theta(\ell^{1/3})$, speeding up the discrete logarithm to $\Theta(\ell^{1/3})$ group operations. The space for the table (and the one-time cost for computing the table) is not a problem for the sizes of ℓ used in these applications.

Our experiments (Section 4) show discrete logarithms in an interval of order ℓ taking only $1.93 \cdot \ell^{1/3}$ multiplications on average using a table of size $\ell^{1/3}$. Precomputation of the table used $1.21 \cdot \ell^{2/3}$ multiplications. This paper also explains (Section 5) how to compress each table entry below $\lg \ell$ bits with negligible overhead.

This algorithm directly benefits the BGN scheme for any message size M. As an illustration, consider the common binary case $M = 1$, and assume $A = B$. The cost of decryption then drops from $\Theta(A^{3/2})$ (superlinear in the number of additions carried out) to just $\Theta(A)$, using a table of size $\Theta(A)$. The same speedup means that [18] can afford to use fewer moduli, saving both space and time.

Further applications of discrete logarithms in small intervals. Many protocols use only degree-1-homomorphic encryption: i.e., addition without any

multiplications. The pairing used in the BGN protocol is then unnecessary: one can use a faster elliptic curve that does not support pairings. Decryption still requires a discrete-logarithm computation, this time on the elliptic curve rather than in a multiplicative group. These protocols can also use Paillier's homomorphic cryptosystem, but elliptic curves provide faster encryption and smaller ciphertexts.

As an example we mention the basic aggregation protocol proposed by Kursawe, Danezis, and Kohlweiss in [21] to enable privacy for smart-meter power-consumption readings. The power company obtains the aggregated consumptions $\sum c_j$ in the exponent as $g^{\sum c_j}$, and compares this to its own measurement of the total consumption c by checking whether $\log_g(g^{\sum c_j}/g^c)$ lies within a tolerance interval. This is another example of a discrete-logarithm computation in a small fixed interval within a large, secure group; we use a small group-specific table to speed up this computation, allowing larger intervals, more aggregation, and better privacy. In cases of sufficiently severe cheating, the discrete logarithm will be too large, causing any discrete-logarithm computation to fail; one recognizes this case by seeing that the computation is running several times longer than expected.

Applications of discrete logarithms in small groups. Another interesting category of applications uses "trapdoor discrete-logarithm groups": groups in which computations of discrete logarithms are feasible with some trapdoor information and hard otherwise. These applications include the Maurer–Yacobi ID-based encryption system in [24], for example, and the Henry–Henry–Goldberg privacy-preserving protocol in [16].

Maurer and Yacobi in [24, Section 4] introduced a construction of a trapdoor discrete-logarithm group, with a quadratic gap between the user's cost and the attacker's cost. It is generally regarded as preferable to have constructive applications be polynomial time and cryptanalytic computations exponential time, but this quadratic gap is adequate for practical applications. A different construction uses Weil descent with isogenies as trapdoor; see [26] for credits and further discussion of both constructions.

The Maurer–Yacobi construction works as follows. Choose an RSA modulus $n = pq$, where $p-1$ and $q-1$ have many medium-size factors — distinct primes ℓ_i chosen small enough that a user knowing the factors of $p-1$ and $q-1$ can solve discrete logarithms in each of these subgroups, using $\Theta(\ell_i^{1/2})$ multiplications modulo p or q, but large enough that the $p-1$ method for factoring n, using $\Theta(\ell_i)$ multiplications modulo n, is out of reach. The group $(\mathbf{Z}/n)^*$ is then a trapdoor discrete-logarithm group. The trapdoor information consists of p, q, and the primes ℓ_i. Note that the trapdoor computation here consists of computations of discrete logarithms in small groups, not small intervals inside larger groups; this turns out to make our techniques slightly more efficient.

Henry and Goldberg in [15] presented a fast GPU implementation of the trapdoor computation of discrete logarithms, using Pollard's rho method. A simple GMP-based implementation of our algorithm on a single core of a low-cost AMD CPU takes an order of magnitude less wall-clock time than the optimized GPU

implementation described in [15], for the same DLP sizes considered in [15], even though the GPU is more than 30 times faster than the CPU core at modular multiplications. Specifically, for a group of prime order almost exactly 2^{48}, our experiments show a discrete-logarithm computation taking just $115729 \approx 1.77 \cdot 2^{16}$ multiplications on average, using a table of size $65536 = 2^{16}$. Precomputation of the table used $5333245354 \approx 1.24 \cdot 2^{32}$ multiplications.

Previous work. Escott, Sager, Selkirk, and Tsapakidis in [9, Section 4.4] showed experimentally that attacking a total of $2, 3, 4, 5$ DLPs with the parallel rho method took, respectively, $1.52, 1.90, 2.22, 2.49$ times longer than solving just one DLP. The basic idea, which [9] said "has also been suggested by Silverman and Stapleton" in 1997, is to compute $\log_g h_1$ with the rho method; compute $\log_g h_2$ with the rho method, reusing the distinguished points produced by h_1; compute $\log_g h_3$ with the rho method, reusing the distinguished points produced by h_1 and h_2; etc.

Kuhn and Struik in [20] analyzed this method and concluded that solving a batch of L discrete logarithms in a group of prime order ℓ reduces the cost of an average discrete logarithm to $\Theta(L^{-1/2}\ell^{1/2})$ multiplications — but only for $L \ll \ell^{1/4}$; see [20, Theorem 1]. Each discrete logarithm here costs at least $\Theta(\ell^{3/8})$; see [20, footnote 5].

Hitchcock, Montague, Carter, and Dawson in [17, Section 3] viewed the computation of many preliminary discrete logarithms $\log_g h_1, \log_g h_2, \ldots$ as a precomputation for the main computation of $\log_g h_k$, and analyzed some tradeoffs between the main computation time and the precomputation time. Two much more recent papers, independent of each other, have instead emphasized tradeoffs between the main computation time and the *space* for a table of precomputed distinguished points. The earlier paper, [22] by Lee, Cheon, and Hong, pointed out that these algorithms are tools not just for the cryptanalyst but for the cryptographer, specifically for trapdoor discrete-logarithm computations. The later paper, our paper [6], pointed out that these algorithms illustrate the gap between the time and space taken by an attack and the difficulty of finding the attack, causing trouble for security definitions in the provable-security literature. Both [22] and [6] clearly break the $\Theta(\ell^{3/8})$-time-per-discrete-logarithm barrier from [20].

In this paper we point out that the same idea, suitably adapted, works not only for discrete logarithms in small groups ("rho") but also for discrete logarithms in small intervals ("kangaroos"). This is critical for BGN-type protocols. We also point out three improvements applicable to both the rho setting and the kangaroo setting: we reduce the number of multiplications by a constant factor by choosing the table entries more carefully; we further reduce the number of multiplications by choosing the iteration function more carefully; and we reduce the space consumed by each table entry. This paper includes several illustrative experiments.

2 Review of generic discrete-logarithm algorithms

This section reviews several standard "square-root" methods to compute discrete logarithms in a group of prime order ℓ. Throughout this paper we write the group operation multiplicatively, write g for the standard generator of the group, and write h for the DLP input; our objective is thus to compute $\log_g h$, i.e., the unique integer k modulo ℓ such that $h = g^k$.

All of these methods are "generic": they work for any order-ℓ group, given an oracle for multiplication (and assuming sufficient hash randomness, for the methods using a hash function). "Square-root" means that the algorithms take $\Theta(\ell^{1/2})$ multiplications on average over all group elements h.

Shanks's baby-step-giant-step method. The baby-step-giant-step method [30] computes $\lceil \ell/W \rceil$ "giant steps" $g^0, g^W, g^{2W}, g^{3W}, \ldots$ and then computes a series of W "baby steps" $h, hg, hg^2, \ldots, hg^{W-1}$. Here W is an algorithm parameter. It is easy to see that there will be a collision $g^{iW} = hg^j$, revealing $\log_g h = iW - j$.

Normally W is chosen as $\Theta(\ell^{1/2})$, so that there are $O(\ell^{1/2})$ multiplications in total; more precisely, as $(1+o(1))\ell^{1/2}$ so that there are $\leq (2+o(1))\ell^{1/2}$ multiplications in total. Interleaving baby steps with giant steps, as suggested by Pollard in [28, page 439, top], obtains a collision after $(4/3+o(1))\ell^{1/2}$ multiplications on average. We have recently introduced a "two grumpy giants and a baby" variant that reduces the constant 4/3; see [5].

The standard criticism of these methods is that they use a large amount of memory, around $\ell^{1/2}$ group elements. One can reduce the giant-step storage to, e.g., $\ell^{1/3}$ by taking W as $\ell^{2/3}$, but this also increases the average number of baby steps to $\ell^{2/3}$. This criticism is addressed by the rho and kangaroo methods discussed below, which drastically reduce space usage while still using just $\Theta(\ell^{1/2})$ multiplications.

Pollard's rho method. Pollard's original rho method [27, Section 1] computes a pseudorandom walk $1, F(1), F(F(1)), \ldots$. Here $F(u)$ is defined as gu or u^2 or hu, depending on whether a hash of u is 0 or 1 or 2. Each iterate $F^n(1)$ then has the form $g^y h^x$ for some easily computed pair $(x, y) \in (\mathbf{Z}/\ell)^2$, and any collision $g^y h^x = g^{y'} h^{x'}$ with $(x, y) \neq (x', y')$ immediately reveals $\log_g h$. One expects a sufficiently random-looking walk on ℓ group elements to collide with itself within $O(\ell^{1/2})$ steps. There are several standard methods to find the collision with negligible memory consumption.

Van Oorschot and Wiener in [33] proposed running many walks in parallel, starting from different points $g^y h^x$ and stopping each walk when it reaches a "distinguished point". Here a fraction $1/W$ of the points are defined (through another hash function) as "distinguished", where W is an algorithm parameter; each walk reaches W points on average. One checks for collisions only among the occasional distinguished points, not among all of the group elements produced. The critical observation is that if two walks reach the same group element then they will eventually reach the same distinguished point — or will enter cycles, but cycles have negligible chance of appearing if W is below the scale of $\ell^{1/2}$.

There are many other reasonable choices of F. One popular choice — when there are many walks as in [33], not when there is a single walk as in [27, Section 1] — is a "base-g r-adding walk": this means that the hash function has r different values, and $F(u)$ is defined as s_1u or s_2u or ... or s_ru respectively, where $s_1, s_2, \ldots, s_r$ are precomputed as random powers of g. One then starts each walk at a different power h^x. This approach has several minor advantages (for example, x is constant in each walk and need not be updated) and the major advantage of simulating a random walk quite well as r increases. See, e.g., [29], [34], and [5] for further discussion of the impact of r. The bottom line is that this method finds a discrete logarithm within $(\sqrt{\pi/2}+o(1))\ell^{1/2}$ multiplications on average.

The terminology "r-adding walk" is standard in the literature but the terminology "base-g r-adding walk" is not. We use this terminology to distinguish a base-g r-adding walk from a "base-(g,h) r-adding walk", in which $s_1, s_2, \ldots, s_r$ are precomputed as products of random powers of g and h. This distinction is critical in Section 3.

Pollard's kangaroo method. An advantage of baby-step-giant-step, already exploited by Shanks in the paper [30] introducing the method, is that it immediately generalizes from computing discrete logarithms in any *group* of prime order ℓ to computing discrete logarithms in any *interval* of length ℓ inside any group of prime order $p \geq \ell$. The rho method uses $\Theta(p^{1/2})$ group operations, often far beyond $\Theta(\ell^{1/2})$ group operations.

Pollard in [27, Section 3] introduced a "kangaroo" method that combines the advantages of the baby-step-giant-step method and the rho method: it takes only $\Theta(\ell^{1/2})$ group operations to compute discrete logarithms in an interval of length ℓ, while still using negligible memory. This method

- chooses a base-g r-adding iteration function whose steps have average exponents $\Theta(\ell^{1/2})$, instead of exponents chosen uniformly modulo ℓ;
- runs a walk starting from g^y (the "tame kangaroo"), where y is at the right end of the interval;
- records the Wth step in this walk (the "trap"), where W is $\Theta(\ell^{1/2})$; and
- runs a walk (the "wild kangaroo") starting from h, checking at each step whether this walk has fallen into the trap.

van Oorschot and Wiener in [33] proposed a parallel kangaroo method in which tame kangaroos start from g^y for many values of y, all close to the middle of the interval, and a similar number of wild kangaroos start from hg^y for many small values of y. Collisions are detected by distinguished points as in the parallel rho method, but the distinguished-point property is chosen to have probability considerably higher than $1/W$; walks continue past distinguished points. The walks are adjusted to avoid collisions between tame kangaroos and to avoid collisions between wild kangaroos. Several subsequent papers have proposed refinements of the kangaroo method, obtaining constant-factor speedups.

The Nechaev–Shoup bound. Shoup proved in [31] that all generic discrete-logarithm algorithms have success probability $O(m^2/\ell)$ after m multiplications.

(The same bound had been proven by Nechaev in [25] for a more limited class of algorithms, which one might call "representation oblivious" generic discrete-logarithm algorithms.) All generic discrete-logarithm algorithms therefore need $\Omega(\ell^{1/2})$ multiplications on average; i.e., the usual square-root discrete-logarithm algorithms are optimal up to a constant factor. A closer look shows that the lower bound is $(\sqrt{1/2}+o(1))\ell^{1/2}$, so both the baby-step-giant-step method and the rho method are within a factor $2+o(1)$ of optimal.

There are much faster discrete-logarithm algorithms (e.g., index-calculus algorithms) for specific classes of groups. However, the conventional wisdom is that these square-root algorithms are the fastest discrete-logarithm algorithms for "secure" groups: a sensibly chosen elliptic-curve group, for example, or the order-ℓ subgroup of $\mathbf{F}_p^*$ for sufficiently large p.

In the rest of this paper we discuss algorithms that improve upon these square-root algorithms by a non-constant factor. Evidently these improved algorithms do not fit Shoup's model of "generic" algorithms — but these improved algorithms *do* apply to "secure" groups. The algorithms deviate from the "generic" model by requiring an extra input, a small table that depends on the group but not on the particular discrete logarithm being computed. The table is set up by a generic algorithm, and if one views the setup and use of the table as a single unified algorithm then Shoup's bound applies to that algorithm; but if the table is set up once and used enough times to amortize the setup costs then each use of the table evades Shoup's bound.

3 Using a small table to accelerate generic discrete-logarithm algorithms

This section explains how to use a small table to accelerate Pollard's rho and kangaroo methods. The table depends on the group, and on the base point g, but not on the target h. For intervals the table depends on the length of the interval but not on the position of the interval: dividing h by g^A reduces a discrete logarithm in the interval $\{A, A+1, \ldots, A+\ell-1\}$ to a discrete logarithm in the interval $\{0, 1, \ldots, \ell-1\}$, eliminating the influence of A.

The speedup factor grows as the square root of the table size T. As T grows, the average number of multiplications needed to compute a discrete logarithm drops far below the $\approx\ell^{1/2}$ multiplications used in the previous section.

The cost of setting up the table is larger than $\ell^{1/2}$, also growing with the square root of T. However, this cost is amortized across all of the targets h handled with the same table. Comparing the table-setup cost $(\ell T)^{1/2}$ to the discrete-logarithm cost $(\ell/T)^{1/2}$ shows that the table-setup cost becomes negligible as the number of targets handled grows past T.

The main parameters in this algorithm are the table size T and the walk length W. Sensible parameter choices will satisfy $W \approx \alpha(\ell/T)^{1/2}$, where α is a small constant discussed below. Auxiliary parameters are various decisions used in building the table; these decisions are analyzed below.

The basic algorithm. To build the table, simply start some walks at g^y for random choices of y. The table entries are the distinct distinguished points produced by these walks, together with their discrete logarithms.

It is critical here for the iteration function used in the walks to be independent of h. A standard base-g r-adding walk satisfies this condition, and for simplicity we focus on the case of a base-g r-adding walk, although we recommend that implementors also try "mixed walks" with some squarings. Sometimes walks collide (this happens frequently when parameters are chosen sensibly), so setting up the table requires more than T walks; see below for quantification of this effect.

To find the discrete logarithm of h using this table, start walks at h^x for random choices of x, producing various distinguished points $h^x g^y$, exactly as in the usual rho method. Check for two of these new distinguished points colliding, but also check for one of these new distinguished points colliding with one of the distinguished points in the precomputed table. Any such collision immediately reveals $\log_g h$.

In effect, the table serves as a free foundation for the list of distinguished points naturally accumulated by the algorithm. If the number of h-dependent walks is small compared to T (this happens when parameters are chosen sensibly) then one can reasonably skip the check for two of the new distinguished points colliding; the algorithm almost always succeeds from collisions with distinguished points in the precomputed table.

Special cases. The extreme case $T = 0$ of this algorithm is the usual rho method with a base-g r-adding walk (or, more generally, the rho method with any h-independent iteration function). However, our main interest is in the speedups provided by larger values of T.

We also draw attention to the extreme case $r = 1$ with exponent 1, simply stepping from u to gu. In this case the main "rho" computation consists of taking, on average, W baby steps $h^x, h^x g, h^x g^2, \ldots$ and then looking up the resulting distinguished point in a table. What is interesting about this case is its evident similarity to the baby-step-giant-step method, but with the advantage of carrying out a table access only after W baby steps; the usual baby-step-giant-step method checks the table after every baby step. What is bad about this case is that the walk is highly nonrandom, requiring $\Theta(\ell)$ steps to collide with another such walk; larger values of r create collisions within $\Theta(\ell^{1/2})$ steps.

Recall from Section 1 the classic algorithm to solve multiple discrete logarithms: for each k in turn, compute $\log_g h_k$ with the rho method, reusing the distinguished points produced by $h_1, \ldots, h_{k-1}$. The $\log_g h_k$ part of this computation obviously fits the algorithm discussed here, with T implicitly defined as the number of distinguished points produced by $h_1, \ldots, h_{k-1}$. We emphasize, however, that this is a special choice of T, and that the parameter curve (T, W) used implicitly in this algorithm as k varies does not obey the relationship $W \approx \alpha(\ell/T)^{1/2}$ mentioned above. Treating T and W as explicit parameters allows several optimizations that we discuss below.

Optimizing the walk length. Assume that $W \approx \alpha(\ell/T)^{1/2}$, and consider the chance that a *single* walk already encounters one of the T distinguished points in the table, thereby solving the DLP. The T table entries were obtained from walks that, presumably, each covered about W points, for a total of TW points. The new walk also covers about W points and thus has $TW^2 \approx \alpha^2\ell$ collision opportunities. If these collision opportunities were independent then the chance of escaping all of these collisions would be $(1 - 1/\ell)^{\alpha^2\ell} \approx \exp(-\alpha^2)$.

This heuristic analysis suggests that a single walk succeeds with, e.g., probability $1 - \exp(-1/16) \approx 6\%$ for $\alpha = 1/4$, or probability $1 - \exp(-1/4) \approx 22\%$ for $\alpha = 1/2$, or probability $1 - \exp(-1) \approx 63\%$ for $\alpha = 1$, or probability $1 - \exp(-4) \approx 98\%$ for $\alpha = 2$.

The same analysis also suggests that the end of the precomputation, finding the Tth point in the table, will require trying $\exp(1/16) \approx 1.06$ length-W walks for $\alpha = 1/4$, $\exp(1/4) \approx 1.28$ length-W walks for $\alpha = 1/2$, $\exp(1) \approx 2.72$ length-W walks for $\alpha = 1$, or $\exp(4) \approx 54.6$ length-W walks for $\alpha = 2$.

The obvious advantage of taking very small α is that one can reasonably carry out several walks in parallel; for example, taking $\alpha = 1/8$ will require 64 walks on average. The most common argument for parallelization is that it allows the computation to exploit multiple cores, decreasing latency. Parallelization is helpful even when latency is not a concern: for example, it allows merging inversions in affine elliptic-curve computations (Montgomery's trick), and it often allows effective use of vector units in a single core. Solving many independent discrete-logarithm problems produces the same benefits, but requires the application to have many independent problems ready at the same time.

The obvious disadvantage of taking very small α is that the success probability per walk drops quadratically with α, while the walk length drops only linearly with α. In other words, chopping a small α in half makes each step half as effective, doubling the number of steps expected in the computation. Sometimes this is outweighed by the increase in parallelization (there are now four times as many walks), but clearly there is a limit to how small α can reasonably be taken.

Clearly there is also a limit to how large α can reasonably be taken. Doubling α beyond 1 does not make each step twice as effective: an $\alpha = 1$ walk already succeeds with chance 63%; an $\alpha = 2$ walk succeeds with chance 98% but is twice as expensive.

We actually recommend optimizing α experimentally (and not limiting it to powers of 2), rather than trusting the exact details of the heuristic analysis shown above. A small issue with the heuristic analysis is that the new walk sometimes takes only, say, $W/2$ steps, obtaining collisions with much lower probability than indicated above, and sometimes $2W$ steps; the success probability of a walk is not the same as the success probability of a length-W walk. A larger issue is that TW is only a crude approximation to the table coverage. Discarding previously discovered distinguished points when building the table creates a bias towards short walks, especially for large α; on the other hand, a walk finding a distinguished point will rarely see all of the ancestors of that point, and in a

moment we will see that this is a controllable effect, allowing the table coverage to be significantly increased.

Lee, Cheon, and Hong in [22, Lemma 1 and Theorem 1] give a detailed heuristic argument that starting M walks in the precomputation will produce $T \approx M(\sqrt{1+2a}-1)/a$ distinct distinguished points, where $a = MW^2/\ell$ (so our α is $(\sqrt{1+2a}-1)^{1/2}$), and that each walk in the main computation then succeeds with probability $1-1/\sqrt{1+2a}$ (i.e., $1-1/(\alpha^2+1)$). In [22, page 13] they recommend taking $a = (1+\sqrt{5})/4 \approx 0.809$ (equivalently, $\alpha \approx 0.786$); the heuristics then state that $T \approx 0.764M$ and that each walk in the main computation succeeds with probability $1-1/\sqrt{1+2a} \approx 0.382$, so the main computation uses $W/0.382 \approx 2.058(\ell/T)^{1/2}$ multiplications on average. We issue three cautions regarding this recommendation. First, assuming the same heuristics, it is actually better to take $a = 1.5$ (equivalently, $\alpha = 1$); then the main computation uses just $2(\ell/T)^{1/2}$ multiplications on average. Second, our improvements to the table coverage (see below) reduce the number of multiplications, and this reduction is different for different choices of a (see our experimental results in Section 4), rendering the detailed optimization in [22] obsolete. Third, even though we emphasize number of multiplications as a simple algorithm metric, the real goal is to minimize time; the parallelization issues discussed above seem to favor considerably smaller choices of α, depending on the platform.

Choosing the most useful distinguished points. Instead of randomly generating T distinguished points, we propose generating more distinguished points, say $2T$ or $10T$ or $1000T$, and then keeping the T most useful distinguished points.

The natural definition of "most useful" is "having the largest number of ancestors". By definition the ancestors of a distinguished point are the group elements that walk to this point; the chance of a uniform random group element walking to this point is exactly the number of ancestors divided by ℓ.

Unfortunately, without taking the time to survey all ℓ group elements, one does not know the number of ancestors of a distinguished point. Fortunately, one has a statistical estimate of this number: a distinguished point found by many walks is very likely to be more useful than a distinguished point found by fewer walks. This estimate is unreliable for a distinguished point found by very few walks, especially for distinguished points found by just one walk; we thus propose using the walk length as a secondary estimate. (In our experiments we computed a weight for each distinguished point as the total length of all walks reaching the point, plus $4W$ per walk; we have not yet experimented with modifications to this weighting.) This issue disappears as the number of random walks increases towards larger multiples of T.

This table-generation strategy reduces the number of walks required for the main discrete-logarithm computation. The table still has size T, and each walk still has average length W, but the success probability of each walk increases. The only disadvantage is an increase in the time spent setting up the table.

Interlude: the penalty for iteration functions that depend on h. Escott, Sager, Selkirk, and Tsapakidis in [9, Section 4.4] chose an iteration function "that is independent of all the Q_is" (the targets h_i): namely, a base-g r-adding walk,

optionally mixed with squarings. Kuhn and Struik in [20] said nothing about this independence condition; instead they chose a base-(g, h_k) r-adding walk. See [20, Section 2.2] ("$g^{a_i}h^{b_i}$") and [20, Section 4] ("all distinguished points $g^{a_j}h_i^{b_j}$ that were calculated in order to find x_i"). No experiments were reported in [20], except for a brief comment in [20, Remark 2] that the running-time estimate in [20, Theorem 1] was "a good approximation of practically observed values".

Hitchcock, Montague, Carter, and Dawson in [17, page 89] pointed out that "the particular random walk recommended by Kuhn and Struik", with the iteration function used for h_k different from the iteration functions used for $h_1, \ldots, h_{k-1}$, fails to detect collisions "from different random walks". They reported experiments showing that a base-(g, h_k) r-adding walk was much less effective for multiple discrete logarithms than a base-g r-adding walk.

To understand this penalty, consider the probability that the main computation succeeds with one walk, i.e., that the resulting distinguished point appears in the table. There are $\approx \ell/W$ distinguished points, and the table contains T of those points, so the obvious first approximation is that the main computation succeeds with probability TW/ℓ. If the table is generated by a random walk independent of the walk used in the main computation then this approximation is quite reasonable. If the table was generated by the *same* walk used in the main computation then the independence argument no longer applies and the approximation turns out to be a severe underestimate.

In effect, the table-generation process in [20] selects the table entries uniformly at random from the set of distinguished points. The table-generation process in [9], [17], and [22] instead starts from random group elements and walks to distinguished points; this produces a highly non-uniform distribution of distinguished points covered by the table, biasing the table entries towards more useful distinguished points. We go further, biasing the table entries even more by selecting them carefully from a larger pool of distinguished points.

Choosing the most useful iteration function. Another useful way to spend more time on table setup is to try different iteration functions, i.e., different choices of exponents for the r-adding walk.

The following examples are a small illustration of the impact of varying the iteration function. Figure 3.1 is a directed graph on 1000 nodes obtained as follows. Each node marked itself as distinguished with probability $1/W$ where $W = 10$. (We did not enforce exactly 100 distinguished points; each node made its decision independently.) Each non-distinguished node created an outgoing edge to a uniform random node. We then used the `neato` program, part of the standard `graphviz` package [14], to draw the digraph with short edges. The $T = 10$ most useful distinguished points are black squares; the 593 nontrivial ancestors of those points are black circles; the other 99 distinguished points are white squares; the remaining points are white circles.

Figure 3.2 is another directed graph obtained in the same way, with the same values of W and T but different distinguished points and a different random walk. For this graph the 10 most useful distinguished points have 687 nontrivial ancestors, for an overall success probability of $697/1000 \approx 70\%$, significantly

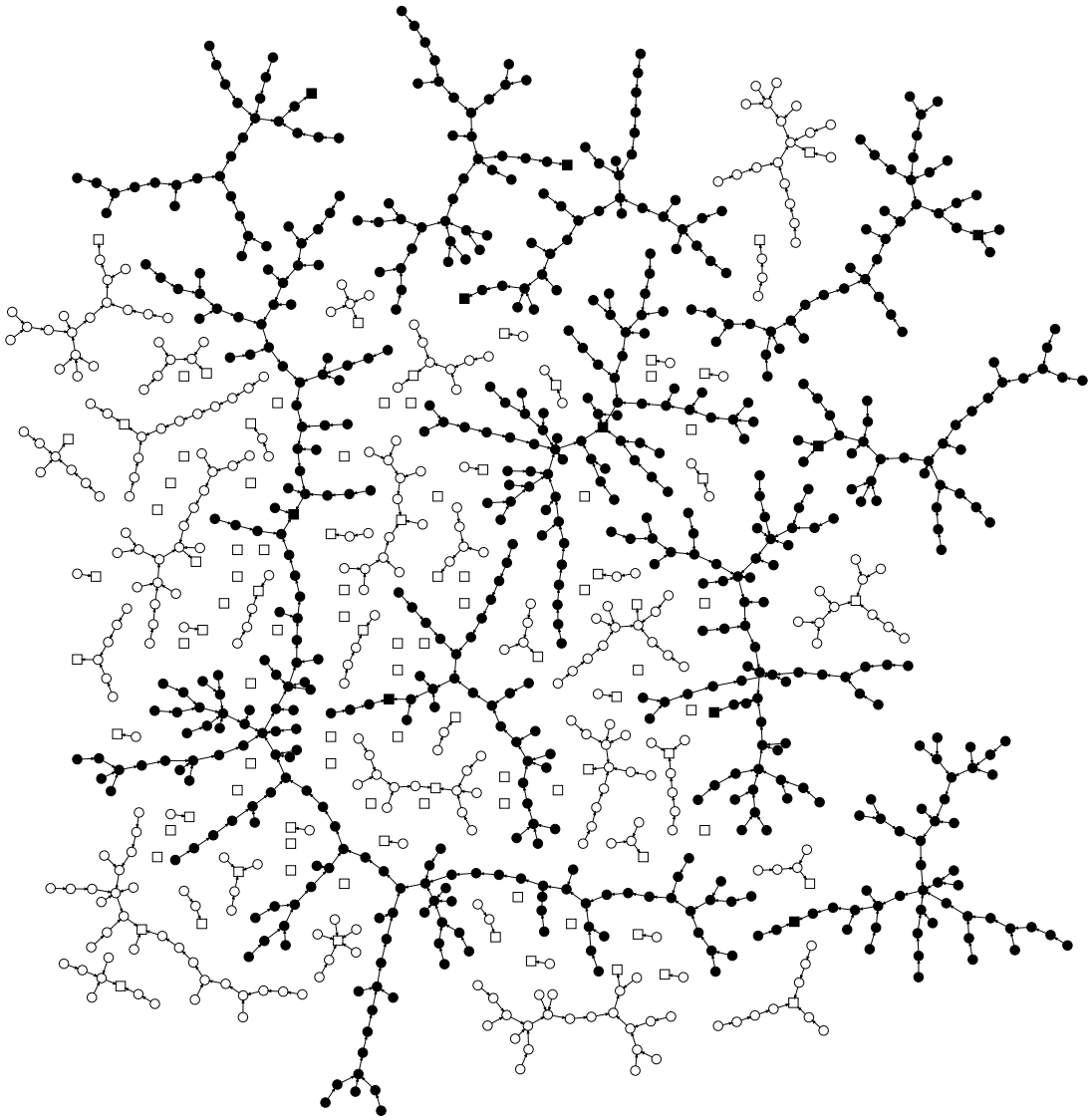

Fig. 3.1. Directed graph on 1000 nodes. Each node chose to be a distinguished point (square) with probability 10%. Each non-distinguished point created one outgoing edge to a uniform random node, and was then placed to keep edges short. The 603 ancestors of the top 10 distinguished points are marked in black.

better than Figure 3.1 and also significantly above the 63% heuristic mentioned earlier.

These graphs were not selected as outliers; they were the first two graphs we generated. Evidently the table coverage has rather high variance.

Of course, a larger table coverage by itself does not imply better performance: graphs with larger coverage tend to have longer walks. We thus use the actual performance of the resulting discrete-logarithm computations as a figure of merit for the graph.

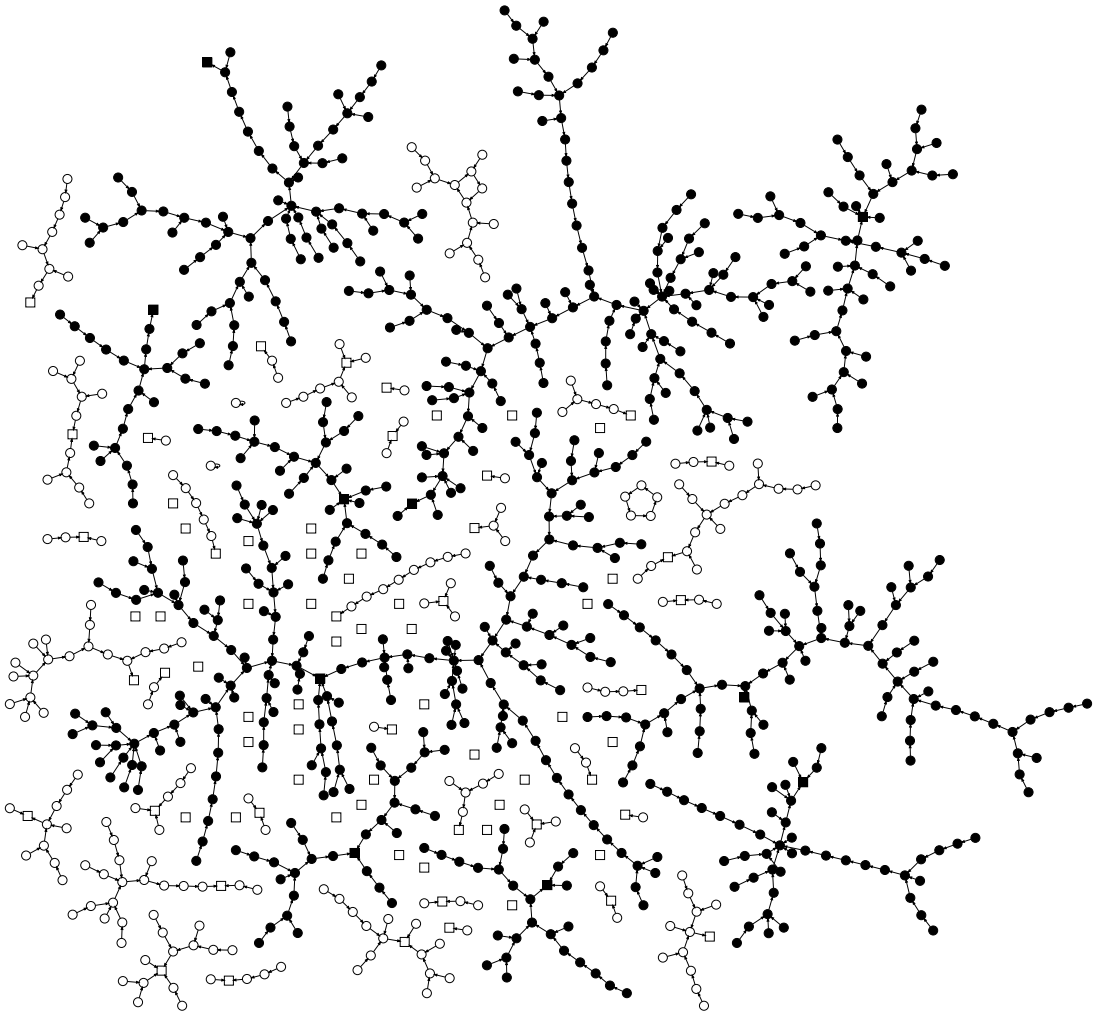

Fig. 3.2. Directed graph on 1000 nodes, generated in the same way as Figure 3.1. The 697 ancestors of the top 10 distinguished points are marked in black.

For small examples it is easy to calculate the exact average-case performance, rather than just estimate it statistically. Figure 3.2 uses, on average, 10.8506 steps to compute a discrete logarithm if walks are limited to 27 steps. (Here 27 is optimal for that graph. The graph has cycles, so some limit or other cycle-detection mechanism is required. One can also take this limit into account in deciding which distinguished points are best.) Figure 3.1 uses, on average, 11.2007 steps.

Adapting the method to a small interval. We now explain a small set of tweaks that adapt the basic algorithm stated above to the problem of computing discrete logarithms in an interval of length ℓ. These tweaks trivially combine with the refinements stated above, such as choosing the most useful distinguished points.

As in the standard kangaroo method, choose the steps $s_1, s_2, \ldots, s_r$ as powers of g where the exponents are $\beta\ell/W$ on average. We recommend numerical optimization of the constant β.

Start walks at g^y for random choices of y in the interval. As in the basic algorithm, stop each walk when it reaches a distinguished point, and build a table of discrete logarithms of the resulting distinguished points.

To find the discrete logarithm of h, start a walk at hg^y for a random small integer y; stop at the first distinguished point; and check whether the resulting distinguished point is in the table. (As in the basic algorithm, there appears to be little value in checking for collisions between these h-dependent walks.) In our experiments we defined "small" as "bounded by $\ell/256$", but it would also have been reasonable to start the first walk at h, the second at hg, the third at hg^2, etc.

We are deviating in several ways here from the typical kangaroo methods stated in the literature. Our walks starting from g^y can be viewed as tame kangaroos, but our tame kangaroos are spread through the interval rather than being clustered together. We do not continue walks past distinguished points. We select the most useful distinguished points experimentally, rather than through preconceived notions of how far kangaroos should be allowed to jump.

We do not claim that the details of this approach are optimal. However, this approach has the virtue of being very close to the basic algorithm, and our experiments so far have found discrete logarithms in intervals of length ℓ almost as quickly as discrete logarithms in groups of order ℓ.

4 Experiments

This section reports several experiments with the algorithm described in Section 3, both for small groups and for small intervals inside larger groups. To allow verification of our results we have included, as Appendix A of this paper, our software for a typical experiment.

Case study: a small-group experiment. We began with several experiments targeting the discrete-logarithm problem modulo pq described in [15, Table 2, first line]. Here p and q are "768-bit primes" generated so that $p-1$ and $q-1$ are "2^{48}-smooth"; presumably this means that $(p-1)/2$ is a product of 16 primes slightly below 2^{48}, and similarly for $(q-1)/2$. The original discrete-logarithm problem then splits into 16 separate 48-bit DLPs modulo p and 16 separate 48-bit DLPs modulo q.

What [15] reports is that a 448-ALU NVIDIA Tesla M2050 graphics card takes an average of 23 seconds for these 32 discrete-logarithm computations, i.e., 0.72 seconds for each 48-bit discrete-logarithm computation. The discrete-logarithm computations in [15] use standard techniques, using more than 2^{24} modular multiplications; the main accomplishment of [15] is at a lower level, using the graphics card to compute 52 million 768-bit modular multiplications per second.

We do not own a Tesla M2050. The card is currently advertised for $1300 but we have decided not to buy it. Instead we are using a single core of a 6-core 3.3GHz AMD Phenom II X6 1100T CPU. This CPU is no longer available but it cost only $190 when we purchased it last year.

We generated an integer p as $1+2\ell_1\ell_2\cdots\ell_{16}$, where $\ell_1,\ell_2,\ldots,\ell_{16}$ are random primes between $2^{48}-2^{20}$ and 2^{48}. We repeated this process until p was prime, and then took $\ell=\ell_1$. This ℓ turned out to be $2^{48}-313487$. We do not claim that this narrow range of 48-bit primes is cryptographically secure in the context of [15]; we stayed very close to 2^{48} to avoid any possibility of our order-ℓ DLP being noticeably smaller than the DLP in [15]. We chose g as $2^{(p-1)/\ell}$ in $\mathbf{F}_p^*$.

For modular multiplications we used the standard C++ interface to the well-known GMP library (version 5.0.2). This interface allows writing readable code such as

```
x = (a * b) % p
```

which turns out to run slightly faster than 1.4 million modular multiplications per second on our single CPU core for our 769-bit p. This understates GMP's internal speeds — it is clear from other benchmarks that we could gain at least a factor of 2 by precomputing an approximate reciprocal of p — but for our experiments we decided to use GMP in the most straightforward way.

We selected $T=64$ and $W=1048576$; here $\alpha=1/2$, i.e., $W\approx(1/2)(\ell/T)^{1/2}$. Precomputing T table entries used a total of $80289876\approx 1.20TW\approx 0.60(\ell T)^{1/2}$ multiplications; evidently some distinguished points were found more than once. We then carried out a series of 1024 discrete-logarithm experiments, all targeting the same h. Each experiment chose a random y and started a walk from hg^y, hoping that (1) the walk would reach a distinguished point within $8W$ steps and (2) the distinguished point would be in the table. If both conditions were satisfied, the experiment double-checked that it had correctly computed the discrete logarithm of h, and finally declared success.

These experiments used a total of $1040325443\approx 0.97\cdot 1024W$ multiplications (not counting the occasional multiplications for the randomization of hg^y and for the double-checks) and succeeded 192 times, on average using $5418361\approx 2.58(\ell/T)^{1/2}$ multiplications per discrete-logarithm computation. Note that the randomization of hg^y made these speeds independent of h.

More useful distinguished points. We then changed the precomputation, preserving $T=64$ and $W=1048576$ but selecting the T table entries as the most useful 64 table entries from a pool of $N=128$ distinguished points. This increased the precomputation cost to $167040079\approx 1.24NW\approx 1.24(\ell T)^{1/2}$ multiplications. We ran 4096 new discrete-logarithm experiments, using a total of $3980431381\approx 0.93\cdot 4096W$ multiplications and succeeding 1060 times, on average using $3755123\approx 1.79(\ell/T)^{1/2}$ multiplications per discrete-logarithm computation.

The $T^{1/2}$ scaling. We then reduced W to 262144, increased T to 1024, and increased N to 2048. This increased the precomputation cost to $626755730\approx 1.17NW\approx 1.17(\ell T)^{1/2}$ multiplications. We then ran 8192 new experiments,

T	512	640	768	896	1024
α	0.70711	0.79057	0.86603	0.93541	1.00000
precomputation, $N = T$	0.84506	0.94916	1.11884	1.23070	1.34187
precomputation, $N = 2T$	1.89769	2.33819	2.74627	3.27589	3.66113
precomputation, $N = 8T$	15.7167	20.7087	26.1621	31.2112	36.9350
main computation, $N = T$	2.13856	2.03391	2.01172	1.98725	2.01289
main computation, $N = 2T$	1.62474	1.59358	1.58893	1.59218	1.61922
main computation, $N = 8T$	1.38323	1.40706	1.42941	1.46610	1.49688

Table 4.1. Observed cost for 15 types of discrete-logarithm computations in a group of order $\ell \approx 2^{48}$. Each discrete-logarithm experiment used T table entries selected from N distinguished points, and used $W = 524288 \approx \alpha(\ell/T)^{1/2}$. Each "main computation" table entry reports, for a series of 2^{20} discrete-logarithm experiments, the average number of multiplications per successful discrete-logarithm computation, scaled by $(\ell/T)^{1/2}$. Each "precomputation" table entry reports the total number of multiplications to build the table, scaled by $(\ell T)^{1/2}$.

using a total of $2123483139 \approx 0.99 \cdot 8192W$ multiplications and succeeding 2265 times, on average using just $937520 \approx 1.79(\ell/T)^{1/2}$ multiplications per discrete-logarithm computation.

We also checked that these computations were running at more than 1.4 million multiplications per second, i.e., under 0.67 seconds per discrete-logarithm computation — less real time on a single CPU core than [15] needed on an entire GPU. There was no noticeable overhead beyond GMP's modular multiplications. The precomputation for $T = 1024$ took several minutes, but this is not a serious problem for a cryptographic protocol that is going to be run many times.

We then reduced W to 32768, increased T to 65536, and increased N to 131072. This increased the precomputation cost to $5333245354 \approx 1.24NW \approx 1.24(\ell T)^{1/2}$ multiplications, roughly an hour. We then ran 4194304 experiments, using a total of $137426510228 \approx 1.00 \cdot 4194304W$ multiplications and succeeding 1187484 times, on average using just $115729 \approx 1.77(\ell/T)^{1/2}$ multiplications per discrete-logarithm computation — under 0.1 seconds.

Optimizing α. We then carried out a series of experiments with $W = 524288$, varying both T and N/T as shown in Table 4.1. Each table entry is rounded to 6 digits. The smallest "main computation" table entry, 1.38314 for $T = 512$ and $N/T = 8$, means (modulo this rounding) that a series of discrete-logarithm experiments used $1.38314(\ell/T)^{1/2}$ multiplications per successful discrete-logarithm computation. Each table entry involved 2^{20} discrete-logarithm experiments, of which more than 2^{18} were successful, so each table entry is very likely to have an experimental error below 0.02.

This table shows that the optimal choice of α depends on the ratio N/T, but also that rather large variations in α around the optimum make a relatively small difference in performance. Performance is much more heavily affected by increased N/T, i.e., by extra precomputation.

T	512	640	768	896	1024
α	0.70711	0.79057	0.86603	0.93541	1.00000
precomputation, $N = T$	0.85702	1.00463	1.14077	1.28112	1.41167
precomputation, $N = 2T$	1.99640	2.38469	2.81441	3.17253	3.61816
precomputation, $N = 8T$	15.5307	20.2547	25.2022	30.7112	36.7452
main computation, $N = T$	2.32320	2.21685	2.14892	2.10155	2.09915
main computation, $N = 2T$	1.66106	1.64183	1.63488	1.65603	1.66895
main computation, $N = 8T$	1.44377	1.44808	1.46581	1.49548	1.52502

Table 4.2. Observed cost for 15 types of discrete-logarithm computations in an interval of length $\ell = 2^{48}$ inside a much larger group. Table entries have the same meaning as in Table 4.1.

To better understand the tradeoffs between precomputation time and main-computation time, we plotted the 15 pairs of numbers in Table 4.1, obtaining Figure 4.3. For example, Table 4.1 indicates for $T = 512$ and $N = 2T$ that each successful discrete-logarithm computation took $1.62474(\ell/T)^{1/2}$ multiplications on average after $1.89769(\ell T)^{1/2}$ multiplications in the precomputation, so $(1.89769, 1.62474)$ is one of the points plotted in Figure 4.3. Figure 4.3 suggests that optimizing α to minimize main-computation time for fixed N/T does not produce the best tradeoff between main-computation time and precomputation time; one should instead decrease α somewhat and increase N/T. To verify this theory we are performing more computations to fill in more points in Figure 4.3.

Small-interval experiments. Starting from the same software, we then made the following tweaks to compute discrete logarithms in a short interval inside a much larger prime-order group:

- We replaced p by a "strong" 256-bit prime, i.e., a prime for which $(p-1)/2$ is also prime. Of course, 256 bits is not adequate for cryptographic security for groups of the form $\mathbf{F}_p^*$, but it is adequate for these experiments.
- We replaced g by a large square modulo p.
- We replaced ℓ by exactly 2^{48}, and removed the reductions of discrete logarithms modulo ℓ.
- We increased r, the number of precomputed steps, from 32 to 128.
- We generated each step as g^y with y chosen uniformly at random between 0 and $\ell/(4W)$, rather than between 0 and ℓ.
- We started each walk from hg^y with y chosen uniformly at random between 0 and $\ell/2^8$, rather than between 0 and ℓ.
- After each successful experiment, we generated a new target h for the following experiments.

For $W = 131072$, $T = 4096$, and $N = 8192$ the precomputation cost was $1337520628 \approx 1.25NW \approx 1.25(\ell T)^{1/2}$ multiplications. We ran 4194304 experiments, using a total of $550245759582 \approx 1.00 \cdot 4194304W$ multiplications and succeeding 1097822 times, on average using $501117 \approx 1.91(\ell/T)^{1/2}$ multiplications per discrete-logarithm computation.

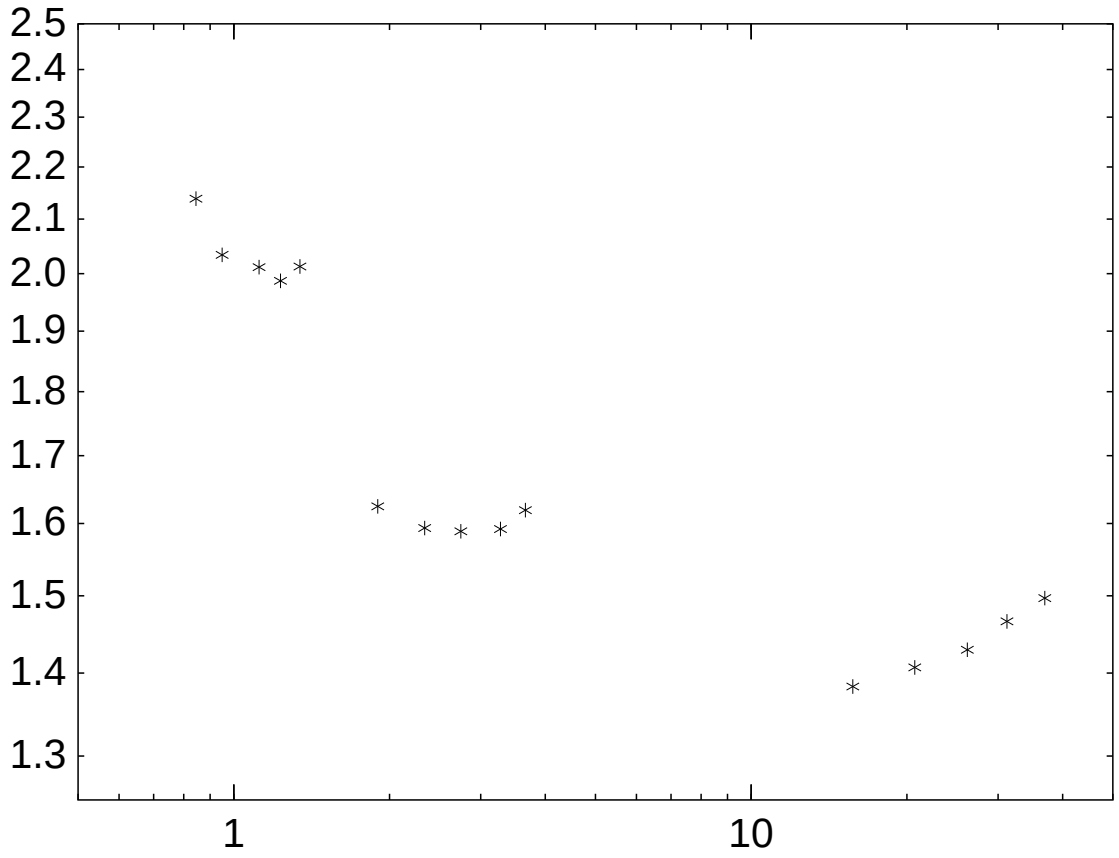

Fig. 4.3. Observed tradeoffs between precomputation time and main-computation time in Table 4.1. Horizontal axis is observed average precomputation time, scaled by $(\ell T)^{1/2}$. Vertical axis is observed average main-computation time, scaled by $(\ell/T)^{1/2}$.

For $W = 32768$, $T = 65536$, and $N = 131072$ the precomputation cost was $5214755468 \approx 1.21NW \approx 1.21(\ell T)^{1/2}$ multiplications. We ran 16777216 experiments, using a total of $548997610777 \approx 1.00 \cdot 16777216W$ multiplications and succeeding 4331750 times, on average using just $126738 \approx 1.93(\ell/T)^{1/2}$ multiplications per discrete-logarithm computation.

Table 4.2 reports the results of experiments for $W = 524288$ with various choices of T and N/T, all using the same bounds $\ell/(4W)$ and $\ell/2^8$ stated above. Comparing Table 4.2 to Table 4.1 shows that this approach to computing discrete logarithms in an interval of length ℓ uses — for the same table size, and essentially the same amount of precomputation — only slightly more multiplications than computing discrete logarithms in a group of order ℓ.

5 Space optimization

Each table entry described in Section 3 consists of a group element, at least $\lg \ell$ bits, and a discrete logarithm, also $\lg \ell$ bits, for a total of at least $2T \lg \ell$

bits. This section explains several ways to compress the table to a much smaller number of bits.

Many of these compression mechanisms slightly increase the number of multiplications used to compute $\log_g h$. This produces a slightly worse tradeoff between the number of multiplications and the number of table *entries*, but produces a much better tradeoff between the number of multiplications and the number of table *bits*.

For comparison, [22, Table 2] took $T = 586463$ and $W = 2^{11}$ for a group of size $\ell \approx 2^{42}$, and reported about $2(\ell/T)^{1/2}$ multiplications per discrete-logarithm computation, using 150 megabytes for the table. Previous sections of this paper explain how to use significantly fewer multiplications for the same T; this section reduces the space consumption by two orders of magnitude for the same T, with only a small increase in the number of multiplications. Equivalently, for the same number of table bits, we use an order of magnitude fewer multiplications.

Lossless compression of each distinguished point. There are several standard techniques to reversibly compress elements of commonly used groups. For example, nonzero elements of the "Curve25519" elliptic-curve group are pairs $(x, y) \in \mathbf{F}_q \times \mathbf{F}_q$ satisfying $y^2 = x^3 + 486662x^2 + x$; here $q = 2^{255} - 19$ and $\ell \approx 2^{252}$. This pair is trivially compressed to x and a single bit of y, for a total of $256 \approx \lg \ell$ bits.

A typical distinguished-point definition states that a point is distinguished if and only if its bottom $\lg W$ bits are 0. These $\lg W$ bits need not be stored. This reduces the space for a distinguished elliptic-curve point to approximately $\lg(\ell/W)$ bits; e.g., $(2/3) \lg \ell$ bits for $W \approx \ell^{1/3}$.

The other techniques discussed in this section work for any group, not just an elliptic-curve group.

Replacing each distinguished point with a hash. To do better we simply suppress some additional bits: we hash each distinguished point to a smaller number of bits and store the hash instead of the distinguished point. This creates a risk of false alarms, but the cost of false alarms is merely the cost of checking a bad guess for $\log_g h$. Checking one guess takes only about $(1 + 1/\lg\lg \ell) \lg \ell$ multiplications, and standard multiexponentiation techniques check several guesses even more efficiently.

If each distinguished point is hashed to $\lg(T/\gamma)$ bits then one expects many false alarms as γ increases past 1. Specifically, a distinguished point outside the table has probability γ/T of colliding with any particular table entry (if the hash behaves randomly), so it is expected to collide with γ table entries overall, creating γ bad guesses for $\log_g h$. For a successful walk, the expected number of bad guesses drops approximately in half, or slightly below half if the discrete logarithms with each hash are sorted in decreasing order of utility.

If γ is far below $W/\lg \ell$ then the cost of checking γ bad guesses is far below W multiplications, the average cost of a walk. For example, if T is much smaller than W then one can afford to hash each distinguished point to 0 bits: the table then consists simply of T discrete logarithms, occupying $T \lg \ell$ bits, and one checks the end of each walk against each table entry.

Compressing a sorted sequence of hashes. It is well known that a sorted sequence of n d-bit integers contains far fewer than nd bits of information when n and d are not very small. "Delta compression" takes advantage of this by computing differences between successive integers and using a variable-length encoding of the differences. For random integers the average difference is close to $2^d/n$ and is encoded as slightly more than $d - \lg n$ bits if $d \geq \lg n$, saving nearly $n \lg n$ bits.

Delta compression does not allow fast random access: to search for an integer one must read the sequence from the beginning. This is not visible in this paper's multiplication counts, but it nevertheless becomes a bottleneck as T grows past W. We instead use a simpler approach that allows fast random access: namely, store the sorted sequence $x_1, x_2, \ldots, x_n$ of d-bit integers as

- the sorted sequence $x_1, x_2, \ldots, x_m$ of $(d-1)$-bit integers where m is the largest index such that $x_m < 2^{d-1}$; and
- the sorted sequence $x_{m+1} - 2^{d-1}, x_{m+2} - 2^{d-1}, \ldots, x_n - 2^{d-1}$ of $(d-1)$-bit integers.

To search for an integer s we search $x_1, \ldots, x_m$ for s if $s < 2^{d-1}$ and search $x_{m+1} - 2^{d-1}, \ldots, x_n - 2^{d-1}$ for $s - 2^{d-1}$ if $s \geq 2^{d-1}$. The second search requires a pointer to the second sorted sequence, i.e., a count of the number of bits used to encode $x_1, x_2, \ldots, x_m$.

This transformation saves 1 bit in each of the n table entries at the expense of a small amount of overhead. This is a sensible transformation if the overhead is below n bits. The transformation is inapplicable if $d = 0$; we encode a sequence of 0-bit integers as simply the number of integers.

Of course, we can and do apply the transformation recursively. The recursion continues for nearly $\lg n$ levels if $d \geq \lg n$, again saving nearly $n \lg n$ bits. For small d the compressed sequence drops to a fraction of n bits.

For example, if each distinguished point is hashed to $d \approx \lg(4T)$ bits, at the expense of $1/4$ bad guesses for each walk, then the hashes are compressed from $Td \approx T \lg(4T)$ bits to just a few bits per table entry. If each distinguished point is hashed to slightly fewer bits, at the expense of more bad guesses for each walk, then the T hashes are compressed to fewer than T bits; in this case one should concatenate the hashes with the discrete logarithms before applying this compression mechanism.

Compressing each discrete logarithm. We finish by considering two mechanisms for compressing discrete logarithms in the table. The first mechanism was introduced in the ongoing ECC2K-130 computation; see [3]. The second mechanism appears to be new.

The first mechanism is as follows. Instead of choosing a random y and starting a walk at g^y, choose a pseudorandom y determined by a short seed. The seed is about $\lg T$ bits, or slightly more if one tries more than T walks; for example, the seed is about 3 times shorter than the discrete logarithm if $T \approx \ell^{1/3}$. Store the seed as a proxy for the discrete logarithm of the resulting distinguished point. Reconstructing the discrete logarithm then takes about W multiplications

to recompute the walk starting from g^y. This reconstruction is a bottleneck if distinguished points are hashed to fewer than $\lg T$ bits (creating many bad guesses), and it slows down the main computation by a factor of almost 2 if α is large, but if distinguished points are hashed to more than $\lg T$ bits and α is small then the reconstruction cost is outweighed by the space savings.

The second mechanism is to simply suppress most of the bits of the discrete logarithm. Reconstructing those bits is then a discrete-logarithm problem in a smaller interval; solve these reconstruction problems with the same algorithm recursively, using a smaller table and a smaller number of multiplications. For example, communicating just 9 bits of an ℓ-bit discrete logarithm means reducing an ℓ-bit DLP to an $(\ell - 9)$-bit DLP, which takes 1/8th as many multiplications using a $T/8$-entry table (or 1/16th as many multiplications using a $T/2$-entry table); if the number of bad guesses is sufficiently small then this is a good tradeoff.

Note that this second mechanism relies on being able to quickly compute discrete logarithms in small intervals, even if the original goal is to compute discrete logarithms in small groups.

References

[1] — (no editor), *2nd ACM conference on computer and communication security, Fairfax, Virginia, November 1994*, Association for Computing Machinery, 1994. See [32].

[2] Mikhail J. Atallah, Nicholas J. Hopper (editors), *Privacy enhancing technologies, 10th interational symposium, PETS 2010, Berlin, Germany, July 21–23, 2010, proceedings*, Lecture Notes in Computer Science, 6205, Springer, 2010. ISBN 978-3-642-14526-1. See [16].

[3] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier Van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gürkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, Bo-Yin Yang, *Breaking ECC2K-130* (2010). URL: http://eprint.iacr.org/2009/541/. Citations in this document: §5.

[4] Feng Bao, Pierangela Samarati, Jianying Zhou (editors), *Applied cryptography and network security, 10th international conference, ACNS 2012, Singapore, June 26–29, 2012, proceedings (industrial track)*, 2012. URL: http://icsd.i2r.a-star.edu.sg/acns2012/proceedings-industry.pdf. See [18].

[5] Daniel J. Bernstein, Tanja Lange, *Two grumpy giants and a baby*, Proceedings of ANTS 2012, to appear (2012). URL: http://eprint.iacr.org/2012/294. Citations in this document: §2, §2.

[6] Daniel J. Bernstein, Tanja Lange, *Non-uniform cracks in the concrete: the power of free precomputation* (2012). URL: http://eprint.iacr.org/2012/318. Citations in this document: §1, §1.

[7] Dan Boneh, Eu-Jin Goh, Kobi Nissim, *Evaluating 2-DNF formulas on ciphertexts*, in TCC 2005 [19] (2005), 325–341. URL: http://crypto.stanford.edu/~dabo/abstracts/2dnf.html. Citations in this document: §1, §1, §1.

[8] Donald W. Davies (editor), *Advances in cryptology — EUROCRYPT '91, workshop on the theory and application of cryptographic techniques, Brighton, UK, April 8–11, 1991, proceedings*, Lecture Notes in Computer Science, 547, Springer, 1991. See [24].

[9] Adrian E. Escott, John C. Sager, Alexander P. L. Selkirk, Dimitrios Tsapakidis, *Attacking elliptic curve cryptosystems using the parallel Pollard rho method*, CryptoBytes **4** (1999). URL: ftp://ftp.rsa.com/pub/cryptobytes/crypto4n2.pdf. Citations in this document: §1, §1, §3, §3.

[10] Simone Fischer-Hübner, Nicholas Hopper (editors), *Privacy enhancing technologies — 11th international symposium, PETS 2011, Waterloo, ON, Canada, July 27–29, 2011, proceedings*, Lecture Notes in Computer Science, 6794, Springer, 2011. See [21].

[11] David Mandell Freeman, *Converting pairing-based cryptosystems from composite-order groups to prime-order groups*, in Eurocrypt 2010 [**12**] (2010), 44–61. URL: http://theory.stanford.edu/~dfreeman/papers/subgroups.pdf. Citations in this document: §1.

[12] Henri Gilbert (editor), *Advances in cryptology — EUROCRYPT 2010, 29th annual international conference on the theory and applications of cryptographic techniques, French Riviera, May 30–June 3, 2010, proceedings*, Lecture Notes in Computer Science, 6110, Springer, 2010. See [11].

[13] Walter Fumy (editor), *Advances in cryptology — EUROCRYPT '97, international conference on the theory and application of cryptographic techniques, Konstanz, Germany, May 11–15, 1997*, Lecture Notes in Computer Science, 1233, Springer, 1997. See [31].

[14] Emden R. Gansner, Stephen C. North, *An open graph visualization system and its applications to software engineering*, Software: Practice and Experience **30** (2000), 1203–1233. Citations in this document: §3.

[15] Ryan Henry, Ian Goldberg, *Solving discrete logarithms in smooth-order groups with CUDA*, in Workshop Record of SHARCS 2012: Special-purpose Hardware for Attacking Cryptographic Systems (2012), 101–118. URL: http://2012.sharcs.org/record.pdf. Citations in this document: §1, §1, §1, §4, §4, §4, §4, §4, §4, §4.

[16] Ryan Henry, Kevin Henry, Ian Goldberg, *Making a nymbler Nymble using VERBS*, in PETS 2010 [**2**] (2010), 111–129. URL: http://www.cypherpunks.ca/~iang/pubs/nymbler-pets.pdf. Citations in this document: §1.

[17] Yvonne Hitchcock, Paul Montague, Gary Carter, Ed Dawson, *The efficiency of solving multiple discrete logarithm problems and the implications for the security of fixed elliptic curves*, International Journal of Information Security **3** (2004), 86–98. Citations in this document: §1, §3, §3.

[18] Yin Hu, William J. Martin, Berk Sunar, *Enhanced flexibility for homomorphic encryption schemes via CRT*, in ACNS 2012 industrial track [4] (2012), 93–110. Citations in this document: §1, §1.

[19] Joe Kilian (editor), *Theory of cryptography, second theory of cryptography conference, TCC 2005, Cambridge, MA, USA, February 10–12, 2005, proceedings*, Lecture Notes in Computer Science, 3378, Springer, 2005. ISBN 3-540-24573-1. See [7].

[20] Fabian Kuhn, Rene Struik, *Random walks revisited: extensions of Pollard's rho algorithm for computing multiple discrete logarithms*, in SAC 2001 [**35**] (2001), 212–229. URL: http://www.distcomp.ethz.ch/publications.html. Citations in this document: §1, §1, §1, §1, §3, §3, §3, §3, §3, §3, §3.

[21] Klaus Kursawe, George Danezis, Markulf Kohlweiss, *Privacy-friendly aggregation for the smart-grid*, in PETS 2011 [**10**] (2011), 175–191. URL: http://research.microsoft.com/pubs/146092/main.pdf. Citations in this document: §1.

[22] Hyung Tae Lee, Jung Hee Cheon, Jin Hong, *Accelerating ID-based encryption based on trapdoor DL using pre-computation*, 11 Jan 2012 (2012). URL: http://eprint.iacr.org/2011/187. Citations in this document: §1, §1, §3, §3, §3, §3, §5.

[23] Donald J. Lewis (editor), *1969 Number Theory Institute: proceedings of the 1969 summer institute on number theory: analytic number theory, Diophantine problems, and algebraic number theory; held at the State University of New York at Stony Brook, Stony Brook, Long Island, New York, July 7–August 1, 1969*, Proceedings of Symposia in Pure Mathematics, 20, American Mathematical Society, Providence, Rhode Island, 1971. ISBN 0-8218-1420-6. MR 47:3286. See [30].

[24] Ueli M. Maurer, Yacov Yacobi, *Non-interactive public-key cryptography*, in Eurocrypt 1991 [**8**] (1991), 498–507. Citations in this document: §1, §1.

[25] Vasiliĭ I. Nechaev, *Complexity of a determinate algorithm for the discrete logarithm*, Mathematical Notes **55** (1994), 165–172. Citations in this document: §2.

[26] Kenneth G. Paterson, Sriramkrishnan Srinivasan, *On the relations between non-interactive key distribution, identity-based encryption and trapdoor discrete log groups*, Designs, Codes and Cryptography **52** (2009), 219–241. URL: http://www.isg.rhul.ac.uk/~prai175/PatersonS09.pdf. Citations in this document: §1.

[27] John M. Pollard, *Monte Carlo methods for index computation (mod p)*, Mathematics of Computation **32** (1978), 918–924. URL: http://www.ams.org/mcom/1978-32-143/S0025-5718-1978-0491431-9/S0025-5718-1978-0491431-9.pdf. Citations in this document: §2, §2, §2.

[28] John M. Pollard, *Kangaroos, Monopoly and discrete logarithms*, Journal of Cryptology **13** (2000), 437–447. Citations in this document: §2.

[29] Jürgen Sattler, Claus-Peter Schnorr, *Generating random walks in groups*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica **6** (1989), 65–79. ISSN 0138-9491. MR 89a:68108. URL: http://ac.inf.elte.hu/Vol_006_1985/065.pdf. Citations in this document: §2.

[30] Daniel Shanks, *Class number, a theory of factorization, and genera*, in [**23**] (1971), 415–440. MR 47:4932. Citations in this document: §2, §2.

[31] Victor Shoup, *Lower bounds for discrete logarithms and related problems*, in Eurocrypt 1997 [**13**] (1997), 256–266. URL: http://www.shoup.net/papers/. Citations in this document: §2.

[32] Paul C. van Oorschot, Michael Wiener, *Parallel collision search with application to hash functions and discrete logarithms*, in [**1**] (1994), 210–218; see also newer version [**33**].

[33] Paul C. van Oorschot, Michael Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999), 1–28; see also older version [**32**]. ISSN 0933–2790. URL: http://members.rogers.com/paulv/papers/pubs.html. Citations in this document: §2, §2, §2.

[34] Edlyn Teske, *On random walks for Pollard's rho method*, Mathematics of Computation **70** (2001), 809–825. URL: http://www.ams.org/journals/mcom/2001-70-234/S0025-5718-00-01213-8/S0025-5718-00-01213-8.pdf. Citations in this document: §2.

[35] Serge Vaudenay, Amr M. Youssef (editors), *Selected areas in cryptography: 8th annual international workshop, SAC 2001, Toronto, Ontario, Canada, August 16–17, 2001, revised papers*, Lecture Notes in Computer Science, 2259, Springer, 2001. ISBN 3-540-43066-0. MR 2004k:94066. See [20].

A Appendix: Software

```
#include <algorithm>
#include <iostream>
#include <gmpxx.h>
#include <math.h>
using std::cout;
using std::flush;
using std::sort;
using std::lower_bound;

#define R 128    // must be a power of 2
#define W 524288 // must be a power of 2
#define T 896
#define N 7168

struct tableentry { mpz_class x; mpz_class log; long long weight; };
tableentry table[N]; // of which T will be used in main computation

bool tablesort(tableentry i,tableentry j)
{
  return i.weight > j.weight; // move higher weights earlier
}
bool tablesort2(tableentry i,tableentry j)
{
  return i.x < j.x; // move smaller points earlier
}

mpz_class l("281474976710656");
mpz_class p
("1090589793224317469591828120135173945200379588911931153368770671904302682037 59");

mpz_class power(const mpz_class &g,const mpz_class &e)
{
  mpz_class result;
  mpz_powm(result.get_mpz_t(),g.get_mpz_t(),e.get_mpz_t(),p.get_mpz_t());
  return result;
}

int hash(const mpz_class &w)
{
  return w.get_si() & (R-1);
}

int distinguished(const mpz_class &w)
{
  return !(w.get_si() & (W-1));
}

int main()
```

```
{
  gmp_randclass ra(gmp_randinit_default);

  mpz_class g = p / l; g = (g * g) % p;

  mpz_class s[R];
  mpz_class slog[R];
  for (int i = 0;i < R;++i) slog[i] = ra.get_z_bits(46) / W;
  for (int i = 0;i < R;++i) s[i] = power(g,slog[i]);

  long long totalnumsteps = 0;
  long long numsteps = 0;

  int tabledone = 0;
  while (tabledone < N) {
    mpz_class wlog = ra.get_z_bits(48);
    mpz_class w = power(g,wlog);
    for (int loop = 0;loop < 8*W;++loop) {
      if (distinguished(w)) {
        int i;
        for (i = 0;i < tabledone;++i) if (table[i].x == w) break;
        if (i < tabledone) {
          table[i].weight += 4*W + numsteps;
        } else {
          table[tabledone].x = w;
          table[tabledone].log = wlog;
          table[tabledone].weight = 4*W + numsteps;
          ++tabledone;
        }
        numsteps = 0;
        break;
      }
      int h = hash(w);
      wlog = wlog + slog[h];
      w = (w * s[h]) % p;
      ++numsteps;
      ++totalnumsteps;
    }
  }

  sort(table,table + N,tablesort);
  sort(table,table + T,tablesort2);

  cout << "alpha = " << W / sqrt(float(l.get_ui())/T) << "\n";
  cout << "r = " << R << "\n";
  cout << "W = " << W << "\n";
  cout << "T = " << T << "\n";
  cout << "N = " << N << "\n";
  cout << totalnumsteps << " precomputation steps; ";
  cout << totalnumsteps / sqrt(float(l.get_ui())*T) << "\n";
```

```
  totalnumsteps = 0;

  mpz_class hlog = ra.get_z_bits(48);
  mpz_class h = power(g,hlog);
  long long numsuccesses = 0;
  long long experiments = 0;

  for (;;) {
    numsteps = 0;

    mpz_class wdist = ra.get_z_bits(40);
    mpz_class w = (h * power(g,wdist)) % p;
    for (int loop = 0;loop < 8*W;++loop) {
      if (distinguished(w)) {
        tableentry desired;
        desired.x = w;
        tableentry *position = lower_bound(table,table + T,desired,tablesort2);
        if (position < table + T)
          if (position->x == w) {
            wdist = position->log - wdist;
          }
        break;
      }
      int h = hash(w);
      wdist = wdist + slog[h];
      w = (w * s[h]) % p;
      ++numsteps;
      ++totalnumsteps;
    }
    if (power(g,wdist) == h) {
      ++numsuccesses;
      hlog = ra.get_z_bits(48);
      h = power(g,hlog);
    }
    ++experiments;
    if (numsuccesses > 0 && !(experiments & (experiments - 1))) {
      cout << experiments << " experiments, "
           << totalnumsteps << " steps, "
           << numsuccesses << " successes, "
           << totalnumsteps/numsuccesses << " steps/success; "
           << (totalnumsteps/numsuccesses) / sqrt(float(l.get_ui())/T) << "\n";
      cout << flush;
    }
  }

  return 0;
}
```