The Discrete Log Problem

Chris Studholme

June 21, 2002

Contents

1 Introduction

The discrete logarithm is the inverse of discrete exponentiation in a finite cyclic group. Given a cyclic group G with group operation $\times$ and a generator g, exponentiation in G is defined by

$$g^x = \overbrace{g \times g \times \cdots \times g}^{x \text{ terms}}.$$

Suppose $y = g^x$, then the discrete logarithm of y is x and is written as

$$\log_g y = x.$$

Actually, the discrete logarithm of y is not unique as it can only be found modulo the order of g in G. If g is a generator as specified above, then the logarithm is found modulo the order of the group. If $n = |g|$, then

$$\log_g y \equiv x \pmod{n}.$$

Discrete exponentiation within a group can be performed quickly, doing only $O(\log x)$ group operations, by using the method of fast exponentiation; however, discrete logarithms appear to be much harder to compute. All methods for computing discrete logarithms designed to work in all cyclic groups require exponential time, with the fastest requiring $O(\sqrt{n})$ time. It has been proven in [17] that if the group operation and the calculation of inverses are the only computations that can be performed on group elements, then the so called "square root methods" are the best that can be expected. However, if more structure in known about the group, then it may be possible to do better. In particular, if a concept of smoothness and smooth elements exists for the group in question, then sub-exponential methods for computing discrete logarithms in that group can be used. The most notable of these groups is the multiplicitive group of units in a finite field. The most notable of the groups for which a concept of smoothness is not known to exist is the additive group of points on an eliptic curve.

The purpose of this paper is to discuss the various methods for computing discrete logarithms. Both the square root methods and the subexponential methods based on the index calculus approach and including the number field sieve will be described and analyzed. The discrete logarithm problem is important because of its wide spread use in the field of cryptography. The first such use of discrete logarithms in cryptography and perhaps the most common use today is the Diffie-Hellman key exchange protocol.

The problem of computing discrete logarithms was just a mathematical curiosity until, in 1976, Diffie and Hellman discribed a method of exchanging cryptographic keys [5] which relies on the hardness of the discrete logarithm problem for its security. The key exchange between two parties, A and B, works as follows:

1. A and B agree on group G and generator g. These choices can be public.

2. A chooses an exponent a at random, computes g^a, and sends this value to B. The exponent a must be kept private.

3. B chooses an exponent b at random, computes g^b, and sends this value to A. The exponent b must be kept private. B then computes, using the value g^a received from A, $K_b = (g^a)^b$.

4. When A receives g^b from B, A computes $K_a = (g^b)^a$.

If all goes well, it should be the case that $K_a = K_b$ and that this key is only known by A and B. These two parties can now use this private key to communicate using some cryptographically secure communication protocol.

During this key exchange protocol, the values g^a and g^b are publically visible. The Diffie-Hellman conjecture states that it is computationally difficult to compute g^{ab} from the known values g^a and g^b. Clearly, if discrete logarithms were easy to compute, then one could simply compute $b = \log_g(g^b)$ and then compute $g^{ab} = (g^a)^b$. It is not known if there is an easy way to compute g^{ab} from knowledge of g^a and g^b without computing the discrete logarithms a and b.

Since Diffie and Hellman's novel new key exchange protocol was introduced, many new key exchange, public key encryption, and digital signature protocols have been introduced which reply on the hardness of the discrete logarithm problem for their security. Also, since an easy way to compute discrete logarithms would immediately lead to an easy way to factor large integers, all of the public key cryptography that depends on the hardness of factoring integers for their security (for example, the RSA public key encryption algorithm) also depend on the hardness of the discrete logarithm problem.

The remainder of this paper is organized as follows. First the square root methods for computing discrete logarithms are outlined. Then the original index calculus idea is described. This method can be extended to arbitrary groups possessing a smoothness property. Various methods for solving large sparse linear systems are outlined as they are needed by all of the subexponential methods for computing discrete logarithms (and those for factoring integers too). Then, a detailed analysis of the complexity of the index calculus method is presented. The number field sieve extention to the index calculus method is described next and an analysis of its complexity is also presented. Finally, a description of the implementation of these various algorithms along and some empirical results are given. The paper concludes by noting some possible directions for future research.

2 Square-Root Attacks

There exists several alogorithms for computing discrete logarithms in an arbitrary cyclic group that run in exponential time. The best of these requires $O\left(\sqrt{n_g}\right)$ group operations, where n_g is the order of the base of the logarithm,

g, in the group. Note that if g generates G, $n_g = n$, the order of the group.

The most naive method one can use to solve the discrete logarithm problem is to start with the exponent zero and try each successive positive exponent until the correct one is found. This method will require (on average) $n_g/2$ group operations; however, it may be useful when n_g is very small (say, less than 100). This situation is most likely to occur when using the Pohlig-Hellman reduction.

In 1978, Pohlig and Hellman [12] observed that if the order of the group is known, along with its complete factorization, and if all of the prime factors of the group order are relatively small, then discrete logarithms can be quickly computed.

More generally, they observed that to compute a discrete logarithm, it suffices to compute the discrete logarithm modulo each of the prime powers dividing the group order, and then to combine these results using the Chinese Remainder theorem to find the discrete logarithm being sought.

Their method makes use the property of the generator, g, that says that if p^k is a prime power dividing the order of the group, n, then the order of the group element g^{n/p^k} is p^k. Furthermore, if y is an arbitrary group element, then y^{n/p^k} has order at most p^k. As long as p is small enough, the discrete logarithm

$$\log_{g^{n/p^k}}(y^{n/p^k}) \pmod{p^k}$$

can be quickly computed. If the prime, p, is very small, then the naive method described above can be used to find the discrete logarithm. If the the prime is not quite small enough to use the naive method, one of the square root methods described below can be used. Note also that large k is not a concern. The discrete logarithm modulo p^k is found using k applications of the algorithm used find the logarithm modulo p (see [12] for details). Using this approach, the discrete logarithm, $\log_g y \bmod n$, can be computed in time $O\left(k\sqrt{p}\right)$ where p is the largest prime dividing n and k is it multiplicity.

Because of this Pohlig-Hellman reduction, groups that are to be used for cryptographic purposes are chosen such that either their order is difficult to compute and is likely to have at least one large prime factor, or the order of the group is known to have at least one large prime factor. In the case where the multiplicitive group of a finite field is used, the group is typically chosen such that either the order of the group is a prime (in the case of $\mathbb{F}_{p^k}$, $k > 1$), or the order is twice a prime (in the case of $\mathbb{F}_p$).

2.1 Shanks' Baby-Step-Giant-Step

In 1973, Shanks [16] described an algorithm for computing discrete logarithms that runs in time $O(\sqrt{n})$ and requires space $O(\sqrt{n})$. This deterministic algorithm requires the construction of two arrays of group elements. The first (giant-steps) is described by

$$S = \{(i, g^{i\lceil\sqrt{n}\rceil}) \mid i = 0, \ldots, \lceil\sqrt{n}\rceil\}$$

and the second (baby-steps) by

$$T = \{(j, y \times g^{j}) \mid j = 0, \ldots, \lceil\sqrt{n}\rceil\},$$

where y is the group element who's discrete logarithm is sought. To compute the discrete logarithm, find a group element that appears in both lists. The logarithm is then

$$\log_g y \equiv i\lceil\sqrt{n}\rceil - j \pmod{n}.$$

To use this method in practice, one would typically only store the giant-steps array in some sort of hash table, and then lookup each successive group element from the baby-steps array until a match is found. Because of the enormous space required by this algorithm, it is rarely used in practice. Instead, when an exponential algorithm must be used, most people use the Pollard Rho method described below as it has the same time complexity as Shanks' method but requires negligable space.

2.2 Pollard Rho Method

Pollard's Rho method [13] for computing discrete logarithms was first introduced in 1978. This method makes use of the so-called "birthday problem" from statistics. The birthday problem asks how many people need to be assembled together to have a 50% chance that two of them share the same birthday. If the people are chosen at random from a uniform distribution, the answer is 23, which is approximately $\sqrt{366}$. More generally, anytime elements are selected at random from a set, say of size n, one only needs to select $O(\sqrt{n})$ of them to have a 50% chance of selecting the same element twice.

The Pollard Rho method works by first defining a pseudo-random sequence of elements from a group, and then looking for a cycle to appear in the sequence. Since the sequence is defined deterministically and each successive element is a function of only the previous element, once a single group element appears a second time, every element of the sequence after that will be a repeat of elements earlier in the sequence. The birthday problem dictates that a cycle should appear after $O\left(\sqrt{n}\right)$ elements of the sequence have been computed.

For the standard Pollard Rho algorithm, a sequence of group elements is defined by

$$a_{i+1} = \begin{cases} y \times a_i & \text{for } a_i \in S_1 \\ a_i^2 & \text{for } a_i \in S_2 \\ g \times a_i & \text{for } a_i \in S_3 \end{cases}$$

where S_1, S_2, and S_3 are an arbitrary partition of the group into roughly equal sized sets (not subgroups). Observing a group element repeat in this sequence will lead to the discrete logarithm of y with a very high probability. Furthermore, it is not necessary to store and search this sequence to find a repeated group element. Instead, one simply computes elements of the sequences a_i and a_{2i} until it is found that $a_i = a_{2i}$.

Improvements to this method can be made by optimizing the choice of iterator function, the size of the sets S_j, or changing the cycle detection criteria. For a discussion of these optimizations, see [18].

2.3 Lambda (Kangaroo) Method

Pollard, in the same paper where he introduced his rho method [13], also introduced a lambda method. The lambda method is useful when the unknown logarithm is known to lie within a relatively small interval. This method is also referred to as a kangaroo method since it can be visualized as following the paths of two kangaroos, one tame and one wild.

The method works by first defining a hash function $H : G \to \mathbb{Z}$ such that the group is divided into subsets $S_1, S_2, \ldots, S_r$, where each subset is given by

$$S_j = \{g \mid H(g) = j\}.$$

Then, a distance and a corresponding jump is associated with each of these subsets. Let $d_1, d_2, \ldots, d_r$ be the distances and $e_1, e_2, \ldots, e_r$ be the jumps. The jumps are defined by $e_j = g^{d_j}$. Finally, the path of a "kangaroo" is given as a sequence defined by

$$c_{i+1} = c_i \times e_{H(c_i)}.$$

The lambda method makes use of two such sequences. One, the path of the tame kangaroo, starts off at a known position. If the unknown logarithm is thought to lie in the interval $[a, b]$, then this tame kangaroo should be started at a some distance within this interval, say the group element $t_0 = g^{\frac{a+b}{2}}$. The tame kangaroo follows a path defined by $t_{i+1} = t_i \times e_{H(t_i)}$ and this path can be tracked by keeping a tally of the distances, d_j, that the kangaroo has jumped.

On the other hand, the wild kangaroo starts off at the group element who's logarithm is sought; namely $w_0 = y$. The kangaroo then follows the path defined by $w_{i+1} = w_i \times e_{H(w_i)}$. Although the starting position of the kangaroo is not known, the total distance it travels can be tracked by keeping a tally of the distances, d_j, just as with the tame kangaroo.

To find the unknown logarithm, one simply needs to observe a crossing of the kangaroos' paths. If the tame kangaroo started at the position $\frac{a+b}{2}$ and travelled a distance of d_t to get to the point where their paths crossed, and the wild kangaroo started at the position x and travelled a distance of d_w to get to the crossing, then the logarithm $x = \log_g y = \frac{a+b}{2} + d_t - d_w$.

Detecting a crossing of the kangaroos' paths is made easy by the fact that once their paths intersect, the kangaroos will forever follow the same path. It is, therefore, not necessary to observe the moment when their paths first intersect. Instead, any observation of a common group element is sufficient and there are guaranteed to be many such group elements once the paths have intersected. One method that can be used to detect group elements common to each path is to store the group elements and their associated distances for each power of two in either a sorted array or a hash table. The stored group elements are $t_1, t_2, t_4, t_8, \ldots$ and $w_1, w_2, w_4, w_8, \ldots$. After each update to the kangaroos' sequences, their positions are compared against the opposite array to find a match. As soon as a match as found, the logarithm can be computed.

The lambda method can find logarithms in an average case running time of $O\left(\sqrt{b-a}\right)$ and requires space $O\left(\log(b-a)\log n\right)$, where n is the order of the group. Several important details have been omitted here (such as how to choose the distances, d_j). These details can either be found in Pollard's original paper, or, for more recent work, a paper by Teske [19].

The Teske paper just mentioned also has a nice description of how the rho and lambda methods were named. To summarize here, their names were chosen because of the shape of the greek letters, ρ and λ, respectively. The rho method involves a sequence of group elements that evenually enter a cycle. This sequence will have an initial segment of elements that do not repeat; followed by a sequence that does. The sequence can be described pictorially by a curve in the shape of ρ. In the lambda method, there are two sequences of group elements defined with a goal of eventually seeing these two sequences have an element in common. The sequences are defined such that once they have an element in common, all subsequent elements in each sequence will be common to both. Pictorially, the two sequences eventually meeting, and then following the same path, is represented by the shape of λ. This etymology was complicated, however, by the introduction of a parallelized rho method, in which several sequences, each following a path depicted by ρ, are defined with the hope of two of these paths intesecting to form a path that looks like λ. This complication has lead to the parallelized rho method sometimes being called a lambda method.

3 The Index Calculus Method

Discrete logarithms can be computed in sub-exponential time if there is more structure to the group beyond just the set of elements and the group operation. Specifically, if certain group elements can be labeled as smooth and factored into a product of group elements from some relatively small factor base, then techniques from linear algebra can be employed to help solve the discrete logarithm problem. Although these sub-exponential time algorithms were first discovered before 1986 (Pollard mentions the posibility of sub-exponential time algorithms in his 1978 paper [13]), for this work the methods of Coppersmith, *et. al.* [3] were implemented.

There are two well known properties of all logarithms that are exploited by the index calculus method:

$$\begin{aligned} \log_g(a \times b) &= \log_g a + \log_g b \quad \text{and} \\ \log_g(a^e) &= e \cdot \log_g a. \end{aligned}$$

These equivalences are used to express the logarithm of a smooth group element as a linear combination of the logarithms of its factors. For example, suppose it is known that $\log_g r = u$ (perhaps $r = g^u$ was assigned explicit) and that the factorization of r is known, say $r = {p_1}^{e_1} \times \cdots \times {p_k}^{e_k}$, then the linear relation

$$u = \log_g(r) = e_1 \cdot \log_g p_1 + \cdots + e_k \cdot \log_g p_k$$

can be written. From linear algebra it is known that with enough relations like this one, the linear system of equations can be solved for the unknown logarithms $\log_g p_i$.

The index calculus method for computing discrete logarithms consists of three phases. The first phase is to find linear relations relating the logarithms of the primes in the factor base. The second phase is to solve for these logarithms using techniques from linear algebra. The final phase is to find the individual logarithm that is being sought by making use of the logarithms of the primes in the factor base. Before describing these three phases in greater detail, the concepts of smoothness, a factor base, and testing integers for smoothness will be explained.

3.1 Smooth Integers, Factor Bases, and Sieving

A concept of smoothness is easily defined in the ring of integers. Since the integers form a unique factorization domain, all non-zero integers r can be uniquely factored (except for order) into a product of primes. Suppose

$$r = {p_1}^{e_1} \cdots {p_k}^{e_k},$$

and then, if each of the p_i with a non-zero e_i are less than some smoothness bound, say B, the integer, r, is said to be B-smooth.

To make use of this smoothness property when calculating discrete logarithms, a factor base needs to be defined. Assuming that the group elements can be ordered based on some definition of size (or norm), the factor base is simply the set of all group elements (factors) that are less than the smoothness bound. For the integers, the factor base consists of the all prime numbers less than B, and they can be quickly found using the sieve of Eratosthenes.

Once a smoothness bound, B, has been chosen and a factor base has been created, it will be necessary to be able to efficiently test a large number of integers for smoothness. There are several methods available for doing these tests.

3.1.1 Trial Division

Trial division is a very simple method for testing a single integer for smoothness to a bound B. Simply attempt to divide the integer by each of the factors in the factor base, one at a time, and in each case where a divisor is found, keep the quotient. If a quotient of 1 is eventually found, then the number is smooth and the exponents of the factorization are known. This method has complexity $O(\pi(B))$, where $\pi(B)$ is the number of primes less than B. Trial division is too slow to be useful as a smoothness test for the index calculus method; however, it is a good method for determining the factorization of an integer that has been determined to be smooth by the sieve method described below. It should be noticed that when factoring an integer that is known to be smooth, the trial division method always terminates early and is therefore reasonably efficient.

3.1.2 Pollard Rho Factoring Method

The Pollard rho factoring method can determine whether a single integer is smooth and find its complete factorization in time $O\left(\sqrt{B}\right)$. This method works in a manner similar to the Pollard rho method for computing discrete logarithms described earlier in this paper. The idea is to define a sequence of integers

$$a_{i+1} = a_i^2 + 1 \pmod{n},$$

where n is the composite to be factored. This sequence is assumed to be pseudo-random and because of the "birthday problem" from statistics it is expected to repeat after $O(\sqrt{n})$ elements have been computed. Furthermore, the sequence $a_i \bmod q$, for some (unknown) prime factor, q, of n, is expected to repeat after only $O\left(\sqrt{q}\right)$ elements have been computed. This implies that a non-trivial factor of n can be found in time $O\left(\sqrt{q}\right)$, where q is the smallest prime factor of n.

As a smoothness test, the rho method only needs to be run for time $O\left(\sqrt{B}\right)$ before giving up and assuming that the integer is not smooth. Although the rho method is asymptotically factor than the trial division method as a smoothness test, in practice it is found to be slower for small B. As a method for finding the factorization of an integer that is known to be smooth, the rho method does not appear to offer any advantage over trial division. Despite this, it can be useful as a smoothness test in the "finding an individual logarithm" phase of the index calculus method; although, ECM (described next) is faster.

3.1.3 Elliptic Curve Method

The elliptic curve method (ECM) for factoring integers is expected to find a non-trivial factor near B in time

$$O\left(e^{\sqrt{(2+o(1))\log B \log\log B}}(\log p)^2\right),$$

where p is an upper bound on the integer being factored. A description of this factoring method is beyond the scope of this paper and can be found in the paper by Lenstra where the method was first was introduced [10].

As a smoothness test for individual integers, the ECM method is the fastest known and is therefore useful during the "finding an individual logarithm" phase of the index calculus method; however, when a large number of integers (say, the image of a polynomial) need to be tested for smoothness, a sieve method is far more efficient.

3.1.4 Polynomial Sieve

A sieve is a method of testing a large number of integers for smoothness very efficiently. The most famous sieve is the sieve of Eratosthenes, mentioned above, which is useful for finding the primes in an interval of integers. The sieve that will be described here is capable of testing the set of values assumed by polynomial (the image of the polynomial) with arguments chosen from a particular domain (an interval). For the index calculus algorithm, it is only necessary to test integers that are the image of single variable degree one polynomials. When the number field sieve is discussed in a later section, it will be necessary there to sieve the image of a two variable homogeneous polynomial. In this section, an algorithm for sieving the image of a single variable polynomial of arbitrary degree will be discussed.

Let $f(X) = a_n \cdot X^n + \ldots a_1 \cdot X + a_0$ and suppose that the values assumed by this polynomial are to be tested for all x in the domain $c \leq x < d$. The key observation to be made here is that if $f(b)$, for some b, is divisible by some prime, say p, then $f(b+p)$ is also divisible by p. To find all b such that $f(b)$ is divisible by a particular p, $f(X)$ can be factored in the field $\mathbb{F}_p[X]$ to find its zeros. There will be at most n such zeros, and from each, all x in the domain such that $f(x)$ is divisible by p can be found.

To determine which of the values assumed by the polynomial are smooth, it is necessary to keep track of which primes divide each image. One efficient way to do this is to tally up the logarithms of each of the primes dividing the image. In addition to finding the primes dividing an image, divisibility by each of the powers of the primes dividing the image also needs to be checked. In pseudo-code, the algorithm is:

for all $c \leq x < d$ **do**
  $l[x] \leftarrow 0$
end for
for all $p \in$ factor base **do**
  // reduce the coefficients of $f(x)$ modulo p and factor
  $f_p(X) \leftarrow f(X) \bmod p$
  $z_p \leftarrow \{z \mid 0 \leq z < p$ and z is a zero of $f_p(X)\}$
  if $f_p \equiv 0 \bmod p$ **then**
    // $f(X)$ is divisible by p
    for all $c \leq x < d$ **do**
      $l[x] \leftarrow l[x] + \log p$
    end for
  else
    for all $z \in z_p$ **do**
      $x_p \leftarrow$ least non-negative residue of $-c + z \bmod p$
      $x \leftarrow c + x_p$
      while $x < d$ **do**
        $l[x] \leftarrow l[x] + \log p$
        $x \leftarrow x + p$
      end while
    end for
  end if
  for all $z_0 \in z_p$ **do**
    // check higher powers of p
    // this is a recursive algorithm, of which, only one step is described
    // suppose $f(z_0)$ is divisible by p^{e-1} for $e > 1$
    $f_{p^e}(W) \leftarrow f(z_0 + Wp^{e-1})/p^{e-1} \pmod p$
    $w_{p^e} \leftarrow \{w \mid 0 \leq w < p$ and w is a zero of $f_{p^e}(W)\}$
    $z_{p^e} \leftarrow \{z \mid 0 \leq z < p$ and $z \equiv z_0 + wp^{e-1} \pmod p$ for $w \in w_{p^e}\}$
    for all $z \in z_{p^e}$ **do**
      $x_p \leftarrow$ least non-negative residue of $-c + z \bmod p$
      $x \leftarrow c + x_p$
      while $x < d$ **do**
        // note that lower powers of p have already been added

$l[x] \leftarrow l[x] + \log p$
$x \leftarrow x + p^e$
end while
recurse: check for divisibility by p^{e+1}
end for
end for
end for
for all $c \leq x < d$ **do**
if $l[x] \sim \log f(x)$ **then**
output: $f(x)$ is smooth
end if
end for

The logarithms of the primes in the factor base are used because addition is generally faster than multiplication or division. Also, instead of using floating point approximations of these logarithms, fixed point approximations of the logarithms can be used to make the algorithm more efficient. The maximum value of the image of $f(X)$ can be found by finding the extrema of $f(X)$, of which there are at most $n+1$, so the base of the logarithm can be chosen such that the logarithms make maximum use of a 32-bit or 64-bit integer, as desired for the particular implementation. Since approximations are being used, the algorithm isn't guaranteed to output all the smooth images and only the smooth images; however, if the logarithms of the primes in the factor base, $\log p$, are all rounded up, and the logarithms of the image of $f(X)$ are all rounded down (and the test for smoothness is $l[x] \geq \log f(x)$), the algorithm is guaranteed not to miss any smooth images (but some non-smooth images may be erroneously output). Non-smooth images that are output should be very rare and will be discovered when the exponents of their factorizations are found using the trial division method (or some other factoring method), so they need not be a concern.

Finally, if it is found that computing the logarithms of the image of $f(X)$ is taking too much time, one can avoid many of these calculations by partitioning the domain at the extrema and working within each partition from the lowest value of the image to the largest. Suppose that $f(c_0)$ and $f(c_1)$ are two extrema, and that $c_0 < c_1$ and $f(c_0) < f(c_1)$, then $\log f(x-1)$ can be used as a lower bound for $\log f(x)$. If $l[x]$ fails to exceed this lower bound, then $f(x)$ is not smooth and $\log f(x-1)$ can be used as a lower bound for $\log f(x+1)$. If $l[x]$ does exceed the lower bound, then the real logarithm, $\log f(x)$, will need to be computed and can be used as a lower bound for $\log f(x+1)$. Each approximation can be used several times as the array elements are checked from smallest image to largest, and the approximation only needs to be updated each time a logarithm is found to exceed the approximation. In practice, this use of approximations has a significant effect on the performance of the code.

Schirokauer, in [14], gives the time complexity of this sieve as

$$\pi(B)(n+\log B)^{O(1)} + O\left((d-c)\log\log B\right).$$

3.2 Random Relations

The index calculus method relies on being able to find linear relations involving the (unknown) logarithms of the primes in the factor base. For instance, when computing discrete logarithms in the field of integers modulo some prime, say p, relations of the form

$$u = e_1 \cdot \log_g p_1 + \cdots + e_k \cdot \log_g p_k \pmod{p-1},$$

where $p_1, \ldots, p_k$ are the primes in the factor base, might be used. The easiest way to find such relations is to choose an integer u at random satisfying $1 \le u < p-1$, compute the least non-negative residue $r \equiv g^u \bmod p$, and then test r for smoothness. If r factors as

$$r = {p_1}^{e_1} \cdots {p_k}^{e_k},$$

then a linear relation involving the logarithms of the elements in the factor base has been found. Unfortunately, this method is a little bit too simplistic and does not yield an efficient algorithm. The problem here is that the smoothness test is too costly for two reasons: r is $O(p)$ and a sieve cannot be easily employed.

A more efficient way to find random relations is to set $H = \lceil \sqrt{p} \rceil$, and then for c_1 and c_2, small and non-negative, compute the least non-negative residue of $(H+c_1)(H+c_2)$. If $H^2 = p + J$, where J is $O\left(\sqrt{p}\right)$, then

$$\begin{aligned}(H+c_1)(H+c_2) &= H^2 + (c_1+c_2)H + c_1c_2 \\ &= p + J + (c_1+c_2)H + c_1c_2 \\ &\equiv J + (c_1+c_2)H + c_1c_2 \pmod{p}.\end{aligned}$$

This residue is $O\left(\sqrt{p}\right)$ and for fixed c_1, many values of c_2 can be tested for smoothness using the polynomial sieve described above. If, for a given c_1 and c_2, the residue is smooth, it can be factored as

$$(H + c_1)(H + c_2) \equiv {p_1}^{e_1} \cdots {p_k}^{e_k} \pmod{p},$$

and the linear relation

$$\log_g(H + c_1) + \log_g(H + c_2) = e_1 \cdot \log_g p_1 + \cdots + e_k \cdot \log_g p_k$$

will have been found.

3.3 The Three Phases

The index calculus method for computing discrete logarithms has three phases. The first two phases are a precomputation (or initialization) stage that does not depend on any particular logarithm that is to be computed. Once these first two phases are complete for a particular group and generator, many discrete logarithms can be quickly computed (relative to the precomputation). This happens in the last phase: computation of an individual logarithm.

The first phase of the algorithm is to find many random relations as described above. The second phase is to consider these random relations as a linear system and solve the system using linear algebra techniques. The final phase involves finding random relations in a manner similar to the first, except that only one relation needs to be found for each individual logarithm to compute.

The precomputation stage consumes the vast majority of the time and space required to compute a logarithm, and once it is complete, the logarithms of a large number of group elements can be efficiently computed.

3.3.1 Precomputation

The purpose of the precomputation stage (the first two phases) is to find the logarithms of all of the primes in the factor base. To simplify this description of the algorithm, assume that discrete logarithms in the field of integers modulo some prime, say p, are to be computed. The following steps are required:

1. choose a smoothness bound, B, and find all of the primes less than B (the factor base);

2. for each $c_1 > 0$, use the polynomial sieve to sieve $f(c_2) = (c_1 + H)c_2 + (c_1 H + J)$ over the domain $c_2 \geq c_1$;

3. using the pairs c_1 and c_2 which yield smooth residues, create a linear system for the unknown logarithms;

4. when enough relations have been found to allow a solution to the linear system to be found (that is, more relations than unknowns) the solution to the linear system will yield the logarithms of the primes in the factor base.

It is important to notice here that the linear system that needs to be solved does not only have the logarithms of the primes in the factor base as unknowns. The logarithms of the $H + c_1$ and $H + c_2$ factors are also unknowns in the linear system. As more random relations are found, the number of unknowns increases, and thus the number of relations required also increases. Luckily, for each value of c_1, many values of c_2 are found which lead to a relation, and thus, in practice, the number of relations increases faster than the number of unknowns. Provided that the length of the sieve used when sieving over c_2 is chosen appropriately, at most four times as many relations as there are primes in the factor base are required to solve the system (tested for moduli up to 10^{40}). The length of the sieve is a tuning parameter that is known to lie on a particular curve and will be discussed later.

Another important tuning parameter is the size of the factor base. In practice, B is chosen such that each of these first two phases (finding the relations and solving the linear system) take roughly the same amount of time. The exact choice of B is implementation dependent as these two phases are quite different problems from a computational point of view. Finding the necessary relations is trivially parallizable and requires a relatively small amount of storage. This stage can be completed easily using a large number of personal computers attached to a local area network or the Internet. On the other hand, solving the linear system that is created is much more difficult to parallelize and typically requires a single fast machine or a tight cluster of machines with enough memory to hold the entire linear system. Despite these implementation issues, the optimal choice of B is also known to lie on a particular curve and will be discussed later in the section on the complexity of the index calculus method.

3.3.2 Computation of an Individual Logarithm

Once the logarithms of all of the primes in the factor base are known, the computation of an arbitrary logarithm is relatively easy. Note that logarithms with a base that differs from g can also be computed with almost as much ease using the standard change of base formula

$$\log_b a \equiv \frac{\log_g a}{\log_g b} \pmod{p-1}.$$

To compute an individual discrete logarithm, say $\log_g y$, first choose an upper smoothness bound (larger than B), say U, and then random exponents, say w, until one is found such that $y \cdot g^w \bmod p$ is U-smooth. For this step, it is not feasible to use a sieve to detect U-smooth residues, so either the Pollard rho factoring method or the elliptic curve factoring method needs to be used instead. Both of these methods is sufficiently fast for this step. The optimal choice of U will be discussed later, but for now, suffice it to say, $U < \sqrt{p}$.

Once a suitable w has been found, the problem becomes one of finding the discrete logarithms of several "medium-sized" primes (that is, primes no larger than U). For the primes that are less than B, their discrete logarithms are known from the precomputation stage. To find the discrete logarithm of a medium-sized prime, say m, the following steps are required:

1. starting at $u = \lceil \sqrt{p}/m \rceil$ and checking increasing values of u, find a u that is B-smooth; the discrete logarithm of u is known from its factorization;

2. then, starting at $v = H = \lceil \sqrt{p} \rceil$ and checking increasing values of v, find a B-smooth residue $n \equiv uvm \bmod p$; the logarithm of n is known from its factorization and the logarithm of v is also known from the precomputation;

3. clearly, $\log_g m \equiv \log_g n - \log_g u - \log_g v \pmod{p-1}$.

Note that appropriate u and v can both be found using the polynomial sieve method.

This discrete logarithm algorithm illustrates the fact that no discrete logarithm is significantly easier or harder to compute than any other. If a particular logarithm, say $\log_g y$, were relatively hard to compute, all that would have to be done to make it easier is to choose integers w at random until one is found such that the logarithm $\log_g(y \cdot g^w)$ is easy to compute.

4 Matrix Reduction Techniques

All of the current low complexity methods for factoring and the solving of discrete logarithms require solutions to large sparse linear systems of equations to be found. These linear systems can be as large as 100,000 equations with 100,000 variables (or larger); however, they tend to be very sparse, which makes finding solutions to them considerably easier. At the time the quadratic sieve was first conceived for factoring integers, the only known method for doing the linear algebra was the gaussian elimination method. This method requires $O(n^3)$ time and $O(n^2)$ space to find a solution and was considered to be the bottleneck for factoring. Now, however, there are several methods known for solving large sparse systems, all of which are considerably faster than gaussian elimination ($O(n^2)$ time) and require very little space beyond what is required to represent the non-zero coefficients of the system.

There are a couple of different ways in which these linear systems are "solved". In the quadratic sieve method for factoring, linear relations over the field $\mathbb{F}_2$ are generated and a linear dependency needs to be found. On the other hand, the index calculus method results in a system of equations with many unknowns being generated. The goal of the linear algebra in this case is to find values for the unknowns. The number field sieve method for factoring and the discrete logarithm problem results in a system of relations from which a linear dependency needs to be generated; however, in contrast to the quadratic sieve method, the relations are over the field $\mathbb{F}_p$ for some prime p.

For the purposes of this paper, only linear algebra techniques for working over fields $\mathbb{F}_p$, $p > 2$ prime, will be discussed. These techniques can also be used when working over $\mathbb{F}_2$; however, greater efficiency can be obtained using special purpose methods that will not be discussed here.

In both the index calculus method and the number field sieve, the linear algebra may need to be done over the ring $\mathbb{Z}/n\mathbb{Z}$ for some composite n. The methods described here for doing the linear algebra only work over fields $\mathbb{F}_p$; however, this need not be a concern. If the prime factorization of n is known, then the linear algebra can be done for each prime and prime power in the factorization, and then the results can be combined using the Chinese Remainder Theorem. If the prime factorization of n is not known, then, in most cases, the algebra can be done assuming that n is prime and the method will either succeed, or it will discover a factor of n.

For a nice survey of these linear algebra techniques, see [11]. The implementations that were done as a part of this work make use of the Lanczos method and a variant of the structured gaussian elimination method. Two other methods are also described briefly.

4.1 Structured Gaussian Elimination

The linear systems that need to be solved as part of modern integer factoring and discrete logarithm algorithms are very sparse, but they are not uniformly sparse. The columns of the matrix that are associated with the smallest primes in the factor base tend to be very dense while the columns associated with the largest primes are extremely sparse (perhaps having no non-zero coefficients). If standard gaussian elimination is attempted and the dense columns are eliminated first, one finds that the matrix immediately becomes non-sparse; however, if the sparse columns are eliminated first, many more columns can be eliminated before fill-in causes the matrix to become non-sparse. It is this observation that structured gaussian elimination is based on.

It is difficult to give an exact algorithm for structured gaussian elimination as it depends on factors such as the distribution of non-zero coefficients in the matrix and the desired output. Structured gaussian elimination is not normally used to completely solve a linear system. Instead, it is used to reduce a matrix to a significantly smaller matrix that is still considered sparse. Then, one of the other techniques for solving sparse systems (described below) is used to solve the system. Deciding when to cease the gaussian elimination and proceed with one of the other solution techniques is an implementation dependent parameter.

LaMacchia and Odlyzko [11] describe the important steps involved in structured gaussian elimination:

1. Delete all columns that have one or fewer non-zero coefficients and the rows in which those columns have non-zero coefficients.

2. Label each column either light or heavy, depending on the number of non-zero coefficients in each.

3. Delete some excess rows, selecting those which have the largest number of non-zero coefficients in the light columns.

4. For any row which has only a single non-zero coefficient equal to ± 1 in a light column, subtract appropriate multiples of that row from all other rows that have non-zero coefficients on that column so as to make those coefficients zero.

These steps are typically repeated until the matrix has been reduced by the desired amount. It can be seen that these steps will never result in an increase in the number of non-zero coefficients in the "light" part of the matrix. Also, as implied by step 3, the system should be overstated. LaMacchia and Odlyzko state that having more extra rows can lead to a smaller final matrix. Of course, those extra rows take time to generate so there is a trade-off here.

The implementation created as a part of this work does not make use of all of the steps described above. It was found that a relatively large reduction in matrix size can be achieved just by implementing steps 1 to 3. The number field sieve implementation created as a part of this work has the additional requirement that some columns must contain non-zero coefficients (preferably, more than one) to ensure that an appropriate linear dependency can be found. The gaussian elimination algorithm used there takes this into account by only deleting rows (step 3) that do not have non-zero coefficients in these critical columns.

4.2 Lanczos

The Lanczos algorithm was originally developed for solving systems over the real numbers, but it has since been found that it can be used over a finite field [11]. The algorithm for solving systems over a finite field is exactly the same as the algorithm used over $\mathbb{R}$. The only difficulty is that over a finite field, it is possible for a non-zero vector to be conjugate to itself. This difficulty, however, does not appear to be much of a problem in practice (it rarely happens).

In its standard form, the Lanczos algorithm is deterministic and can solve symmetric systems; however, it may be the case that a non-symmetric system needs to be solved. This case can be handled by a probabilistic transform. Suppose the system to be solved is

$$Bx = u$$

where B is $m \times n, m \geq n$, x is the unknown n-vector, and u is a given m-vector. This system can be transformed by selecting a random $m \times m$

diagonal matrix, D, where the diagonal elements are chosen from among the non-zero field elements. The transformed system is

$$Ax = w$$

where

$$A = B^T D^2 B \quad \text{and} \quad w = B^T D^2 u.$$

All of the details and assumptions can be found in [11].

A nice feature of the Lanczos algorithm is that very little storage space beyond that which is required to represent the matrix is required. Furthermore, the representation of the matrix itself need not be changed at all by the Lanczos method (in contrast to gaussian elimination).

The algorithm usually terminates within n iterations when used to solve a system of dimension n (in practice, it appears to always terminate within n iterations). The heuristic time bound for Lanczos is $O(n^2)$.

4.3 Conjugate Gradient

The conjugate gradient algorithm is almost identical to the Lanczos algorithm; only the iterations are slightly different. The conjugate gradient method is useful for finding several linear dependencies among a set of vectors. This is typically what needs to be done when factoring integers. In particular, the quadratic sieve requires several dependencies among a set of vectors to be found over the field $\mathbb{F}_2$. For this work, the conjugate gradient method was not implemented. The details of the algorithm can be found in [11], including the heuristic time bound, which is the same as that for the Lanczos algorithm, $O(n^2)$.

4.4 Wiedemann

The Wiedemann algorithm is an alternative to using the Lanczos algorithm. It is a Krylov subspace method, like Lanczos, but its main innovation is the use of the Berlekamp-Massey algorithm, which allows one to determine linear recurrence coefficients over finite fields very quickly. From a theory point of view, the advantage of the Wiedemann algorithm is that is comes with a rigorously proven time bound of $O(n^2)$; however, in practice, it does not appear to offer any improvement over the Lanczos algorithm. On the contrary, unless additional storage space, $O\left(n^2\right)$, is used, the Wiedemann algorithm is significantly slower than the Lanczos algorithm.

For the purposes of this work, the Wiedemann algorithm was not implemented. Complete details can be found in [21].

5 Analysis of the Index Calculus Method

The time required to compute a logarithm using the index calculus method is highly dependent on the choice of the smoothness bound B. On the one hand, if B is small then the test for smoothness of a single integer is fast and the time required to solve the linear system generated will be small. On the other hand, a small value for B makes smooth integers difficult (perhaps nearly impossible) to find. Therefore, B must be chosen to balance these opposing factors.

Furthermore, the choice of B also influences the ratio of the time required to find the relations (the sieving phase) verses the time required to solve the linear system. As already mentioned, these two problems are very different from a complexity point of view. Because of this dichotomy between these two phases, in practice, the choice of B may depend on what hardware is available. When attempting to compute discrete logarithms in a very large group, it is often necessary to choose B smaller than the theoretically optimal value because it is easier to obtain the hardware necessary to complete the sieving phase than it is to find sufficient hardware to solve the linear system.

5.1 Smooth Number Estimate

To find an optimal choice for B, it is necessary to have an estimate of the probability that a particular integer, r, is B-smooth. Define

$$\Psi(x, B) = \#\{r \mid 1 \leq r \leq x \text{ and } r \text{ is } B\text{-smooth}\}$$

as the number of B-smooth integers less than x.

This function, describing the number of smooth integers in a particular interval, has been extensively studied for at least the past 70 years. In addition to having an application in the study of discrete logarithm and factoring algorithms, the function has many other applications in number theory, and it is interesting in its own right. There have been many estimates given for $\Psi(x, B)$ over the years, each applicable in some subset of the choices of B. For a recent estimate that claims to apply uniformly over the domain $2 \leq B \leq x$, see [8]; however, for this analysis, it suffices to use an estimate by Dickman [4], first introduced in 1930. This estimate is

$$\Psi(x, x^{1/u}) \sim x\rho(u),$$

where $u = \log x / \log B$ and $\rho(u)$ is the solution to the differential equation

$$u\rho'(u) + \rho(u-1) = 0$$

with the initial condition

$$\rho(u) = 1 \quad (0 \leq u \leq 1).$$

The solution to this differential equation is estimated as

$$\rho(u) \sim u^{-u+o(u)}.$$

For analyzing the index calculus method, it is convenient to choose B to be of the form $B = e^{c\sqrt{\log p \log\log p}}$; however, when analyzing the number field sieve method (later in this paper), is will be necessary to choose all of the algorithm parameters to be of the form

$$L_p[s; c] = e^{c(\log p)^s(\log\log p)^{1-s}}.$$

Therefore, to be able to apply the same techniques to analyzing each algorithm, the index calculus parameters will be put in this form as well.

Suppose that x is of the form $x = L_p[s; c]$ and that $B = L_p[s_B; c_B]$, then the probability that a random integer chosen from the domain $1 \leq r \leq x$ is B-smooth is given by

$$\begin{aligned}
\frac{\Psi(x,B)}{x} &= u^{-u+o(u)} \\
&= \left(\frac{c(\log p)^s(\log\log p)^{1-s}}{c_B(\log p)^s_B(\log\log p)^{1-s_B}}\right)^{-\frac{c(\log p)^s(\log\log p)^{1-s}}{c_B(\log p)^s_B(\log\log p)^{1-s_B}}+o(u)} \\
&= e^{-(s-s_B)\frac{c}{c_B}(\log p)^{s-s_B}(\log\log p)^{-s+s_B}(\log\log p+O(\log\log\log p))} \\
&= e^{(-(s-s_B)\frac{c}{c_B}+o(1))(\log p)^{s-s_B}(\log\log p)^{1-s+s_B}} \\
&= L_p\left[s-s_B; -(s-s_B)\frac{c}{c_B}+o(1)\right].
\end{aligned}$$

This equation gives the probability that a random integer, chosen from the domain, is smooth. The following analysis for the index calculus method (and the later analysis of the number field sieve) is going to assume that integers that were not chosen randomly from the domain have this same probability of being smooth. It is this assumption that makes the results of this analysis a heuristic complexity estimate and not a proven complexity bound. Stated formally, the conjecture is:

Conjecture: Let f be a polynomial in n variables over $\mathbb{Z}$ and assume that $|f(x_1,\ldots,x_n)| < A$ whenever all x_i lie in the interval $[-\frac{1}{2}C, \frac{1}{2}C]$. Then the probability that $f(a_1,\ldots,a_n)$ is B-smooth for a_i chosen randomly from $[-\frac{1}{2}C, \frac{1}{2}C]$ is $\Psi(A,B)/A$. If B is chosen as above, and $A = L_p[s_A; c_A + o(1)]$, then this probability is

$$\frac{\Psi(A,B)}{A} = L_p\left[s_A - s_B; -(s_A - s_B)\frac{c_A}{c_B} + o(1)\right].$$

5.2 The $L_p[s;c]$ Function

The function $L_p[s;c]$ is a very useful function for studying an algorithm that has a time or space complexity that is somewhere between polynomial and exponential. It should be noted that if $s = 0$, then $L_p[s;c] = (\log p)^c$ is a polynomial (in the size of p), and that if $s = 1$, then $L_p[s;c] = e^{c\log p}$ is exponential in the size of p. In this section, a derivation will be presented describing where this function comes from and hinting to the fact that $s_B = 1/2$ for the index calculus method. In the next section it will be shown that $s_B = 1/2$ is the optimal solution for the index calculus method.

Suppose that the number of relations that need to be found is bounded by some constant multiple of the size of the factor base. The factor base consists of approximately $B/\log B$ primes. If the probability that an arbitrary integer, say r, is B-smooth is u^{-u}, then the expected number of random integers that need to be tested to find a smooth one is u^u. Thus, if we need $DB/\log B$ relations, the expected number of smoothness tests needed to find these relations is

$$\frac{DBu^u}{\log B}.$$

The time required to test an integer for smoothness is dependent on how the integer is tested:

Trial Division requires $O(\pi(B)) = O(B/\log B)$ divisions, each requiring $O((\log x)^2)$ time, thus the total time is $O(B(\log x)^2/\log B)$;

Pollard-Rho Method requires $O\left(\sqrt{B}\right)$ multiplications, each of which requires $O((\log x)^2)$ time, thus the total time is $O\left(\sqrt{B}(\log x)^2\right)$;

Elliptic Curve Method requires time $O\left(e^{\sqrt{(2+o(1))\log B \log\log B}}(\log x)^2\right)$; and

Linear Sieve requires time $O\left(\pi(B)(1+\log B)^{o(1)} + L\log\log B\right)$ for a sieve of length L, thus the time per integer is $\log\log B$.

In the above equations x is an upper bound on the size of the integers being tested.

All of these methods will result in a sub-exponential time discrete logarithm algorithm; however, clearly, the linear sieve method will yield the lowest complexity. Assuming the linear sieve is being used, the time required to generate the linear system is given by

$$T(B) = \frac{DBu^u}{\log B}\log\log B.$$

Set

$$S(B) = \log T(B) = \log D + \log B + u\log u - \log\log B + \log\log\log B,$$

and dispose of the $\log\log B$ and $\log\log\log B$ terms as they are dominated by the $\log B$ term. Differentiating yields

$$\begin{aligned}
\frac{dS}{dB} &= \frac{1}{B} + \frac{du}{dB}\log u + \frac{du}{dB} \\
&= \frac{1}{B} - \frac{\log x \log u}{B(\log B)^2} - \frac{\log x}{B(\log B)^2} \\
&= \frac{1}{B} - \frac{\log x}{B(\log B)^2}(1+\log u) \\
&= \frac{1}{B} - \frac{\log x}{B(\log B)^2}(1+\log\log x - \log\log B),
\end{aligned}$$

and setting this derivative to zero to find the optimal choice of B gives

$$(\log B)^2 = \log x(1+\log\log x - \log\log B).$$

Now, $1 < \log\log B < \log\log x$, so the inequalities

$$\log x < \log x(1 + \log\log x - \log\log B) < \log x \log\log x$$

imply that the optimal choice of B satisfies

$$e^{\sqrt{\log x}} < B < e^{\sqrt{\log x \log\log x}}.$$

With x of the from $x = p^d$, the optimal choice of B is of the form

$$B = L_p\left[\frac{1}{2}; c_B + o(1)\right],$$

for some constant c_B.

5.3 The Precomputation

Let $B = L_p[s_B; c_B + o(1)]$ for some constants s_B and c_B. Evidence was presented above suggesting that $s_B = 1/2$; however, in this section, s_B will be derived again and the value of $1/2$ will be shown to be the optimal choice for s_B. During the sieving stage of the algorithm, pairs of non-negative integers c_1 and c_2 are found such that the residue $(H+c_1)(H+c_2)$ is smooth. Let these integers be chosen such that $0 \leq c_1 < c_2 \leq C$, where $C = L_p[s_C; c_C + o(1)]$. The total number of residues that will be tested for smoothness is

$$\frac{1}{2}C^2 = L_p[s_C; 2c_C + o(1)].$$

Note that this is actually $O(L_p[\ldots])$ as the factor of $1/2$ has been dropped. In this section, all assertions of equality will involve an unspecified scalar multiple. Since the final complexity estimate will be specified as $O(\ldots)$, this need not be a concern.

The number of smooth residues that need to be found is

$$\begin{aligned} B + C &= L_p[s_B; c_B + o(1)] + L_p[s_C; c_C + o(1)] \\ &= L_p[\max\{s_B, s_C\}; \max\{c_B, c_C\} + o(1)]. \end{aligned}$$

Note here that the use of $\max\{c_B, c_C\}$ assumes that $s_B = s_C$ (which, it will be shown, is the case), but in the case where this assumption does not hold, the above equation is an upper bound on the size of the factor base.

Let $P_{\mathbb{Q}}$ be the probability that a random residue is smooth and assume that it is of the form $P_{\mathbb{Q}} = L_p\left[s_{\mathbb{Q}}; c_{\mathbb{Q}} + o(1)\right]$. To be able to solve the linear system that is produced, it must be the case that

$$\frac{1}{2}C^2 P_{\mathbb{Q}} \geq B + C.$$

Let x be the least non-negative residue such that $x \equiv (H + c_1)(H + c_2) \bmod p$. If $H = \left\lceil L_p\left[1; \frac{1}{2}\right]\right\rceil$ and $J = H^2 - p \leq 2H$, then x is bounded by

$$\begin{aligned}
x &= J + (c_1 + c_2)H + c_1 c_2 \\
&\leq (2 + c_1 + c_2)H + c_1 c_2 \\
&\leq 2L_p\left[s_C; c_C + o(1)\right] \cdot L_p\left[1; \frac{1}{2}\right] + L_p\left[s_C; 2c_C + o(1)\right] \\
&= L_p\left[1; \frac{1}{2} + o(1)\right].
\end{aligned}$$

Assuming that the conjecture stated above holds, the probability of a random residue being smooth is thus

$$P_{\mathbb{Q}} = \frac{\Psi(x, B)}{x} = L_p\left[1 - s_B; -\frac{1 - s_B}{2c_B} + o(1)\right].$$

The condition that enough smooth residues be found then becomes

$$\begin{aligned}
L_p\left[s_C; 2c_C + o(1)\right] \;\geq\;& L_p\left[\max\{s_B, s_C\}; \max\{c_B, c_C\} + o(1)\right] \\
& \cdot L_p\left[1 - s_B; \frac{1 - s_B}{2c_B} + o(1)\right],
\end{aligned}$$

which implies that at the very least

$$s_C \geq \max\{s_B, s_C, 1 - s_B\}.$$

The sieving phase takes time

$$\begin{aligned}
& C \cdot \left(\pi(B)(1 + \log B)^{o(1)} + C \log\log B\right) \\
&= L_p\left[s_C; c_C\right] \cdot \left(L_p\left[s_B; c_B\right] + L_p\left[s_C; c_C\right]\right) \\
&= L_p\left[\max\{s_B, s_C\}; c_C + \max\{c_B, c_C\} + o(1)\right],
\end{aligned}$$

and the linear algebra phase takes time

$$(B + C)^2 = L_p\left[\max\{s_B, s_C\}; \max\{2c_B, 2c_C\} + o(1)\right].$$

Thus the total time required for the precomputation is

$$L_p\left[\max\{s_B, s_C\}; \max\{2c_B, 2c_C\} + o(1)\right].$$

To minimize the running time of the algorithm, it is necessary to choose s_B and s_C such that $\max\{s_B, s_C\}$ is a minimum, and yet

$$s_C \geq \max\{s_B, 1 - s_B\}$$

is satisfied. This minimum occurs when $s_B = s_C = \frac{1}{2}$.

This choice gives

$$\begin{aligned} B &= L_p\left[\frac{1}{2}; c_B + o(1)\right], \\ C &= L_p\left[\frac{1}{2}; c_C + o(1)\right], \text{ and} \\ P_{\mathbb{Q}} &= L_p\left[\frac{1}{2}; \frac{-1}{4c_B} + o(1)\right]. \end{aligned}$$

The condition that enough smooth residues are found then becomes

$$L_p\left[\frac{1}{2}; 2c_C + o(1)\right] \geq L_p\left[\frac{1}{2}; \max\{c_B, c_C\} + o(1)\right] \cdot L_p\left[\frac{1}{2}; \frac{1}{4c_B} + o(1)\right]$$

or

$$2c_C \geq \max\{c_B, c_C\} + \frac{1}{4c_B},$$

and the total running time of the algorithm becomes

$$L_p\left[\frac{1}{2}; \max\{2c_B, 2c_C\} + o(1)\right].$$

Minimizing $\max\{2c_B, 2c_C\}$ subject to the sufficient smooth residue condition yields $c_B = c_C = \frac{1}{2}$. Thus the optimal choices for B and C are

$$B = L_p\left[\frac{1}{2}; \frac{1}{2} + o(1)\right] \text{ and } C = L_p\left[\frac{1}{2}; \frac{1}{2} + o(1)\right].$$

The total time required to complete the precomputation is

$$O\left(L_p\left[\frac{1}{2}; 1 + o(1)\right]\right).$$

5.4 Time to find an Individual Logarithm

Up to this point it has been assumed that the final phase of the index calculus method, finding an individual logarithm, is much faster than the first two phases and could therefore be ignored when doing the complexity analysis. This assumption now has to be validated, and actually, it only holds provided that the elliptic curve factoring method is used for a smoothness test in the one place where a sieve is not suitable.

The first step in finding an individual logarithm is selecting an upper smoothness bound, $U \geq B$, and finding a random integer w such that the least non-negative residue of $y \cdot g^w \bmod p$ is U-smooth. Let $U = L_p\left[s_U; c_U + o(1)\right]$ and the probability of finding a suitable w be $P_w = L_p\left[s_w; c_w + o(1)\right]$. Since the residue is bounded by $p = L_p\left[1; 1\right]$, the smooth number estimate gives this probability as

$$P_w = \frac{\Psi(p, U)}{p} = L_p\left[1 - s_U; -\frac{1 - s_U}{c_U} + o(1)\right].$$

Using the elliptic curve factoring method as a smoothness test, the time required to check each residue for smoothness is $e^{\sqrt{(2+o(1))\log U \log\log U}}(\log p)^2$, which, using the $L_p[\ldots]$ notation, is

$$L_p\left[\frac{s_U}{2}; \sqrt{2 s_U c_U} + o(1)\right] \cdot L_p\left[0; 2\right] = L_p\left[\frac{s_U}{2}; \sqrt{2 s_U c_U} + o(1)\right].$$

Since it is expected that $1/P_w$ smoothness tests will be required to find a suitable w, the total time to find w is

$$L_p\left[1 - s_U; \frac{1 - s_U}{c_U} + o(1)\right] \cdot L_p\left[\frac{s_U}{2}; \sqrt{2 s_U c_U} + o(1)\right].$$

Minimizing $\max\{1 - s_U, s_U/2\}$ gives $s_U = \frac{2}{3}$. Minimizing $1/(3c_U) + 2\sqrt{c_U/3}$ gives $c_U = \left(\frac{1}{3}\right)^{1/3}$. Therefore, U should be chosen as

$$U = L_p\left[\frac{2}{3}; \left(\frac{1}{3}\right)^{\frac{1}{3}} + o(1)\right],$$

and the total time required for this step is

$$L_p\left[\frac{1}{3}; 3^{\frac{1}{3}} + o(1)\right],$$

which is much faster than $L_p\left[\frac{1}{2}; 1+o(1)\right]$. This step is only faster than the first two phases if the elliptic curve factoring method is used as the smoothness test. If the Pollard Rho factoring method is used as a smoothness test, then U should be chosen as $U = L_p\left[\frac{1}{2}; 1+o(1)\right]$, and this choice makes the time required to complete this step $L_p\left[\frac{1}{2}; 1+o(1)\right]$, which is the same as the first two phases. If the trial division method is used as a smoothness test, then U should be chosen as $L_p\left[\frac{1}{2}; \sqrt{\frac{1}{2}}+o(1)\right]$, and the time required to complete this step is longer, at $L_p\left[\frac{1}{2}; \sqrt{2}+o(1)\right]$, than the time required to complete the precomputation.

Coppersmith suggests that U should be chosen as $U = L_p\left[\frac{1}{2}; 2\right]$ for what he describes as a non-optimal solution. With this choice of U, the number of smoothness tests required to find w is $L_p\left[\frac{1}{2}; \frac{1}{4}\right]$. Coppersmith gives the generous upper bound of $L_p\left[\frac{1}{2}; \frac{1}{4}\right]$ for the time needed to test for smoothness, where a time of $L_p\left[\frac{1}{4}; \sqrt{2}\right]$ would be more accurate. This is the reason why he then states that the time to find w is $L_p\left[\frac{1}{2}; \frac{1}{2}\right]$.

The next step in finding an individual logarithm is, for each medium prime m, finding a $u > \sqrt{p}/m$ that is B-smooth. Since m may be as small as B, u is of order $L_p\left[1; \frac{1}{2}\right]$ and the probability of random choice of u being smooth is $L_p\left[\frac{1}{2}; -\frac{1}{2}\right]$. To find one, it is necessary to test $L_p\left[\frac{1}{2}; \frac{1}{2}+o(1)\right]$ possibilities using the polynomial sieve. This takes time $L_p\left[\frac{1}{2}; \frac{1}{2}+o(1)\right]$, which is much faster than the precomputation.

The final step is finding a $v > \sqrt{p}$ such that the least non-negative residue of $uvm \bmod p$ is B-smooth. Such a v is also expected to be of order $L_p\left[1; \frac{1}{2}\right]$; thus the probability of a random choice being smooth is $L_p\left[\frac{1}{2}; -\frac{1}{2}\right]$ and the time needed to find one using a linear sieve is $L_p\left[\frac{1}{2}; \frac{1}{2}+o(1)\right]$. Again this is much faster than the precomputation. These last two steps for finding an individual logarithm need to be done once for each medium sized prime found in the first step; however, there will be at most $\log_B U = \frac{\log U}{\log B}$ of these. If $s_U = \frac{2}{3}$ then this is $O\left((\log p)^{\frac{1}{6}}\right)$ and if $s_U = \frac{1}{2}$ then this number is a constant, so either way, it is of no concern.

6 The Number Field Sieve

The general discrete logarithm problem in $\mathbb{F}_p$ is to find an integer x such that

$$b^x \equiv y \pmod{p}$$

where b and y are given. This general problem can be reduced to the special case where b is a "small" prime primitive element (a generator) of the field, say g, and the power y is a "medium sized" prime, say v, such that $v < p^{1/k}$ for some integer $k > 1$. This reduction is described by Weber in [20]. To summarize, the reduction involves making use of the standard change of base formula for logarithms and the finding of a w for which $y \cdot g^w \bmod p$ is $p^{1/k}$-smooth. Given a small prime generator, g, and a medium sized prime, v, the logarithm is denoted

$$x \equiv \log_g v \pmod{p-1}.$$

Also, it will be assumed that p is a Germain prime. Such a choice for p is generally considered to make the discrete logarithm problem as difficult as possible. Let $q = (p-1)/2$, a prime. Solving the problem using the number field sieve in cases where p is not of this form involves only a slight complication in the linear algebra phase which will not be described here.

Unlike the index calculus method, the version of the number field sieve that is presented here does not first calculate the discrete logarithm of all of the primes in the factor base, and then find the discrete logarithm of an individual logarithm. Instead, the discrete logarithm of medium sized primes are found one at a time. There are still two phases, however, which are similar to the first two phases of the index calculus method. The first phase is to sieve the image of a polynomial to find a number of relations, and the second phase is to use linear algebra techniques to find a dependency among the relations. These two phases are repeated to find the discrete logarithm of each of the medium sized prime factors of $y \cdot g^w \bmod p$.

The algorithm presented here is almost identical to the algorithm described by Schirokauer in [14], but it was also inspired by [7] and [20].

6.1 Choosing a Number Field

To choose a number field over which to work, first choose its degree, k, no less than 2. The choice of k is dependent on the size of p and will be discussed further in the analysis section below. In the empirical results section, data for choices of k in the range $2 \leq k \leq 4$ are presented.

Next, it is necessary to find an appropriate minimal polynomial for the number field. First, find the unique integer h such that $m = 2^h \cdot v$ is in the range $p^{1/k} < m < 2p^{1/k}$. Next, find the least integer c such that $cp > m^k$ and write cp in base m as

$$cp = m^k + \cdots + b_2' m^2 + b_1' m + b_0'.$$

Clearly, this polynomial in m is monic and of degree k.

This polynomial, however, may not be sufficient as it is required that the norm $|N(\alpha)|$, for some root α, be B-smooth, for some rational smoothness bound B. The general equation for calculating the norm of a principal ideal in the number field will be given below, but for now,

$$|N(\alpha)| = |\text{constant term in minimal polynomial}|.$$

Therefore, to ensure that this norm is B-smooth, it is necessary to find a minimal polynomial with a constant term that is B-smooth. The polynomial described above can be modified to meet this requirement by subtracting multiples of m from b_0' while incrementing b_1' until the constant term is B-smooth. To be more precise, find the least non-negative D such that $|b_0' - Dm|$ is B-smooth, and then write $f(X)$ as

$$\begin{aligned} f(X) &= X^k + \cdots + b_2' X^2 + (b_1' + D)X + (b_0' - Dm) \\ &= X^k + \cdots + b_2 X^2 + b_1 X + b_0. \end{aligned}$$

This $f(X)$ is suitable if

1. f is irreducible,
2. f is monic,
3. the degree of f is k,
4. the coefficients of f have absolute value less than $(D+1)m$,
5. $f(m) \equiv 0 \pmod{p}$,
6. the constant term, b_0, is B-smooth, and
7. q does not divide the discriminant Δf.

Using common polynomial factoring methods, requirement 1 can be checked, requirements 2 to 6 are obviously satisfied given the construction of $f(X)$, and requirement 7 can be checked by computing the discriminant of the polynomial. Polynomials generated in this manner are very unlikely to fail to meet these requirements.

Finally, use $f(X)$ to define a number field. Let α be a root of $f(X)$. Then $K = \mathbb{Q}(\alpha)$ is a number field, and let $\mathcal{O}_K$ be its ring of integers.

6.2 Norms and Smooth Integers

As mentioned above, B is a smoothness bound. It must be chosen such that $g \leq B$. This smoothness bound is used to test the smoothness of both rational integers and principal ideals. To test the smoothness of a principal ideal, compute its norm and then check the absolute value of the norm for smoothness as a rational integer.

The norm of a principal ideal, say $(c + d\alpha)$, is computed as

$$\begin{aligned} N(c + d\alpha) &= (-d)^k f(-c/d) \\ &= c^k - b_{k-1}c^{k-1}d + \cdots + b_1 c(-d)^{k-1} + b_0(-d)^k \end{aligned}$$

where the coefficients $b_{k-1}, \ldots, b_0$ are the same as they appear in the minimal polynomial of α, $f(X)$. Also mentioned above was the norm of (α). From this equation, it is clear that

$$|N(\alpha)| = |b_0|.$$

To completely factor a principal ideal, an algebraic factor base needs to be generated. This factor base will consist of all first degree prime ideals with norm less than B. All such prime ideals will have as their norm a rational prime; therefore, to find these prime ideals, it is necessary to factor the principal ideals, (r), for each rational prime, r, less than B.

Let r be a rational prime less than B. One of the following will be the case:

1. (r) does not split (it remains prime),
2. (r) splits, but none of its factors have degree 1,
3. (r) is the k^{th} power of a single prime ideal (this prime ideal is said to be totally ramified),

4. (r) splits into more than 1 but fewer than k distinct factors, some with degree 1, maybe some with higher degree, or

5. (r) splits into exactly k distinct degree 1 factors (it splits completely).

Note that the degree of a prime factor determines (or is determined by) its norm. If $\mathfrak{r} \mid (r)$ and has degree l, then $N\mathfrak{r} = r^l$. To determine which of the above possibilities is the case for a particular r, first reduce the coefficients of f modulo r. This gives

$$f_r(X) \equiv X^k + \cdots + a_2X^2 + a_1X + a_0 \pmod r$$

where each $a_i \equiv b_i \bmod r$. Then attempt to factor f_r in the field $\mathbb{F}_r[X]$ using standard polynomial factoring techniques. Factoring will result in one of the following:

1. f_r is irreducible,

2. f_r factors, but all of its factors are irreducible and have degree larger than 1,

3. f_r consists of a single degree 1 factor raised to the k^{th} power,

4. f_r has more than 1 but fewer than k factors, some of degree 1, or

5. f_r has exactly k distinct degree 1 factors.

These possibilities correspond one-for-one with the possible ways in which (r) may split (or not) described in the previous list. If 1 or 2 is the case, then no prime ideals are added to the algebraic factor base. In case 1, leaving the prime ideal, (r), out of the factor base is not a concern since the ideal $(c + d\alpha)$ will only have (r) as a factor if both c and d are divisible by r. During the sieving stage of the algorithm (see below), c and d will be chosen such that they are coprime so this cannot happen. In case 2, (r)'s prime factors cannot be added to the factor base either. In case 3, the single prime ideal is added to the factor base. In cases 4 and 5, each distinct degree 1 factor corresponds to a prime ideal with r as its norm and needs to be added to the factor base.

It will also be necessary to determine the exact factorization of an arbitrary ideal $(c + d\alpha)$. As mentioned above, first determine if the norm $|N(c + d\alpha)|$ is B-smooth. Assuming that it is, go through each of the norm's prime factors, and for each, determine which prime ideal they correspond to. In the cases where the prime ideal corresponding to a prime factor of the norm is totally ramified, case 3 above, there is only the single prime ideal that could be a factor and thus, nothing more to do; however, in the cases where the ideal generated by the prime dividing the norm splits in to two or more distinct degree 1 prime factors, it will be necessary to determine which of these prime factors divides the ideal $(c + d\alpha)$. First, recall the definition of the norm

$$N(c + d\alpha) = (-d)^k f(-c/d).$$

Then, since r divides $N(c + d\alpha)$, the equation for the norm reduces to

$$(-d)^k f(-c/d) \equiv 0 \pmod{r}.$$

Also, recall the reduction of f modulo r

$$\begin{aligned} f_r(X) &\equiv X^k + \cdots + a_2X^2 + a_1X + a_0 \pmod{r} \\ &\equiv (X - w_1)^{e_1}(X - w_2)^{e_2} \cdots (X - w_h)^{e_h} \cdot G_r(X) \pmod{r} \end{aligned}$$

where h is the number of distinct degree 1 factors of f_r, the w_i's are the roots of f_r, and $G_r(X)$ is the product of any irreducible factors with degree greater than 1. Now, combining the equation for the norm with this reduced equation for f gives

$$(c/d - w_1)^{e_1}(c/d - w_2)^{e_2} \cdots (c/d - w_h)^{e_h} \equiv 0 \pmod{r}.$$

Clearly, the value $c/d \bmod r$ must equal one of the roots. This value also uniquely determines which prime ideal divides $(c + d\alpha)$.

6.3 Sieving

The sieving phase for the number field sieve involves finding pairs of integers, c and d, such that both $c + dm$ and $(c + d\alpha)$ are B-smooth. The former is tested for smoothness as a rational integer while the later is tested as a principal ideal. Formally, suitable pairs of integers must satisfy:

1. $d \geq 1$,

2. $-m < c < m$,

3. c and d are coprime,

4. the rational integer $c + dm$ is B-smooth, and
5. the principal ideal $(c + d\alpha)$ is also B-smooth.

These pairs of integers can be efficiently found using a sieve over a homogeneous polynomial in two variables, and such a sieve is very similar to the polynomial sieve described in the index calculus section earlier in this paper; just extended to two variables.

Assume $f(X, Y)$ is a homogeneous polynomial of degree k and write

$$\tilde{f}(Z) = \frac{f(X, Y)}{Y^k},$$

where $Z = X/Y$. Clearly, $\tilde{f}(Z)$ is a one-variable polynomial, and the smooth values it assumes can be found using the sieve technique described earlier. Then, the values of X and Y for which $f(X, Y)$ is smooth are found using the equivalence $Z \equiv X/Y \bmod p$.

The total running time of this sieve algorithm is given in [14] as

$$\pi(B)(k + \log B)^{O(1)} + C^2 \log\log B,$$

where X and Y are both assumed to be restricted to the domain

$$-\frac{1}{2}C \leq X, Y \leq \frac{1}{2}C.$$

6.4 Character Maps

The goal of the linear algebra phase of the number field sieve is to construct a q^{th} power in the ring of integers, $\mathcal{O}_K$. To ensure that a linear dependency found during the algebra phase leads to a q^{th} power, a set of maps

$$\lambda : \mathcal{O}_K \longrightarrow q\mathcal{O}_K / q^2\mathcal{O}_K$$

are used. These maps are referred to as character maps, and they have the property that if γ is a q^{th} power in $\mathcal{O}_K$, then $\lambda(\gamma) = 0$.

For a given principal ideal $(c + d\alpha)$, the character maps produce a k-tuple, $(\lambda_1, \ldots, \lambda_k)$, of integers, each from $\mathbb{Z}/q\mathbb{Z}$. To compute the character map, it is necessary to calculate a power in the ring of polynomials with coefficients from $\mathbb{Z}/q^2\mathbb{Z}$ represented by polynomials in $(\mathbb{Z}/q^2\mathbb{Z})[X] \bmod f(X)$. In this ring, let $c + dX$ represent $(c + d\alpha)$ and then compute the power

$$r(X) = (c + dX)^e - 1.$$

Provided the exponent here has been chosen appropriately, all of the coefficients of this polynomial will be multiples of q. Therefore, let the character maps, $(\lambda_1, \ldots, \lambda_k)$, be the k coefficients of $r(X)$, each divided by q. More precisely,

$$r(X) = \lambda_1 q + \lambda_2 qX + \cdots + \lambda_k qX^{k-1} \pmod{q^2}.$$

The necessary exponent, e, is calculated by factoring the ideal (q). This is done in exactly the same manner described for generating the algebraic factor base. If any of the prime factors of (q) have a multiplicity greater than 1 then the character map calculation fails and a new number field must be chosen (this rarely happens in practice). Otherwise, for each prime factor of (q), if the factor has degree d, then e must be divisible by $q^d - 1$. The optimal choice for e is the least common multiple of these factors. See [14] for all the details regarding this calculation.

6.5 Linear System

It is necessary to collect at least $\pi(B) + \mu(B) + k - 1$ pairs c_i, d_i to create a linear system. Each pair consists of $\pi(B)$ exponents corresponding to the primes in the rational factor base, $\mu(B)$ exponents corresponding to the primes in the algebraic factor base, and k values associated with the character maps for $(c_i + d_i\alpha)$. One additional relation corresponding to the rational integer g and the principal ideal (1) will also be added. This additional relation will consist of a 1 as the exponent of g in the rational factor base, and 0 for all other exponents and the character maps.

With all of these pairs (relations), the following matrix is created:

$$A = \begin{pmatrix} \longleftarrow [g] \longrightarrow & \longleftarrow [(1)] \longrightarrow & 0 & \ldots & 0 \\ \longleftarrow [c_1 + d_1 m] \longrightarrow & \longleftarrow [(c_1 + d_1\alpha)] \longrightarrow & \lambda_1 & \ldots & \lambda_k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \longleftarrow [c_l + d_l m] \longrightarrow & \longleftarrow [(c_l + d_l\alpha)] \longrightarrow & \lambda_1 & \ldots & \lambda_k \end{pmatrix}$$

where $\leftarrow [\eta] \rightarrow$ denotes the exponents of the factorization of η within the appropriate factor base. This matrix has $\pi(B) + \mu(B) + k$ columns, and therefore, should have at least that many rows.

The final step in creating the linear system is to construct a row, say y, which consists of the value of h (found when the minimal polynomial was chosen) in the column that corresponds to the rational integer 2, the exponents of the factorization of (α) (recall that care was taken to ensure that $|N(\alpha)|$ is B-smooth), and the character maps associated with the principal ideal (α). This row is

$$y = \big(\longleftarrow [2^h] \longrightarrow \;\; \longleftarrow [(\alpha)] \longrightarrow \;\; \lambda_1 \;\; \ldots \;\; \lambda_k \; \big) .$$

Now, using standard linear algebra techniques, express $-y$ as a linear combination of the rows of A. That is, find a row x such that

$$x \cdot A \equiv -y \pmod q.$$

6.6 The Solution

The element of x corresponding to the row that just has the exponent for the rational integer g (the row of the matrix with a single 1 in it), say x_0, will be, with high probability, the desired logarithm, $\log_g v \bmod q$.

To see this, consider that the solution to the matrix equation yields two relations:

$$\begin{aligned} g^{x_0} \cdot \prod_i x_i(c_i + d_i m) &= 2^{-h} \cdot (q^{th} \text{ power}) \text{ and} \\ \prod_i x_i(c_i + d_i \alpha) &= (\alpha)^{-1} \cdot (q^{th} \text{ power}). \end{aligned}$$

In other words,

$$2^h \cdot g^{x_0} \cdot \prod_i x_i(c_i + d_i m) \text{ and } (\alpha) \cdot \prod_i x_i(c_i + d_i \alpha)$$

are both q^{th} powers.

There exists a homomorphism $\varphi : K \longrightarrow \mathbb{Z}$ such that

$$\varphi(\alpha) = m.$$

Applying this homomorphism to the second q^{th} power above gives

$$\begin{aligned} \varphi\left((\alpha) \cdot \prod_i x_i(c_i + d_i \alpha)\right) &= m \cdot \prod_i x_i(c_i + d_i m) \\ &= v \cdot 2^h \cdot \prod_i x_i(c_i + d_i m), \end{aligned}$$

since $m = 2^h v$. Now, multiply both sides of this equation by g^{x_0} to get

$$g^{x_0} \cdot \varphi\left((\alpha) \cdot \prod_i x_i(c_i + d_i\alpha)\right) = v \cdot 2^h \cdot g^{x_0} \cdot \prod_i x_i(c_i + d_i m),$$

which leads to

$$g^{x_0} \cdot (q^{th} \text{ power}) \equiv v \cdot (q^{th} \text{ power}) \pmod{p},$$

and then,

$$\log_g v \equiv x_0 \pmod{q}.$$

Finally, finding the logarithm modulo $p-1$ is simply a matter of testing both x_0 and $x_0 + q$.

7 Analysis of the Number Field Sieve

The time required to compute a discrete logarithm using the number field sieve is divided between its two phases, the sieving phase and the linear algebra phase. As was the case with the index calculus method, these two phases each require roughly the same time to complete and the division between them can be varied to suit the available hardware.

This analysis of the number field sieve will proceed in very much the same manner as the analysis of the index calculus method given earlier. To begin, the various algorithm parameters that need to be chosen will be expressed as functions of the form

$$L_p[s; c] = e^{c(\log p)^s(\log\log p)^{1-s}}.$$

In particular, let the smoothness bound, B, and the degree of the number field, k, be given by

$$B = L_p[s_B; c_B + o(1)] \text{ and } p^{1/k} = L_p[s_k; c_k + o(1)],$$

or to express k directly,

$$k = \frac{1}{c_k + o(1)}\left(\frac{\log p}{\log\log p}\right)^{1-s_k}.$$

During the sieving phase, pairs of integers, c and d are found such that $c + dm$ and $(c + d\alpha)$ are both B-smooth. Let c and d be such that $-\frac{1}{2}C < c < \frac{1}{2}C$ and $0 < d < C$, and define C by

$$C = L_p\left[s_C; c_C + o(1)\right].$$

With these restrictions on the domain of the sieve, the sieving phase is expected to take time

$$\pi(B)(k + \log B)^{O(1)} + C^2 \log\log B,$$

which, after substituting for B and C, is

$$k \cdot L_p\left[s_B; c_B + o(1)\right] + L_p\left[s_C; 2c_C + o(1)\right].$$

The linear algebra phase, using the Lanczos method, takes time

$$O\left(kB^2\right),$$

which, after substituting for B, is

$$k \cdot L_p\left[s_B; 2c_B + o(1)\right].$$

Let the total time required to complete the sieving and linear algebra phases be $L_p\left[s_T; c_T + o(1)\right]$, where

$$\begin{aligned} s_T &= \max\{s_B, s_C\} \text{ and} \\ c_T &= \max\{2c_B, 2c_C\}. \end{aligned}$$

At the completion of the sieving phase, it is necessary to have found sufficient smooth numbers to have a reasonable chance of success in the linear algebra phase. The number of smooth numbers required is $\pi(B)+\mu(B)+k-1$, which is bounded by $kB = k \cdot L_p\left[s_B; c_B + o(1)\right]$.

Suppose that $P_{\mathbb{Q}}$ is the probability that an integer $c+dm$ in the domain of the sieve is B-smooth, and that P_K is the probability that an ideal $(c + d\alpha)$, also in the domain of the sieve, is B-smooth. Let $P_{\mathbb{Q}} = L_p\left[s_Q; c_Q + o(1)\right]$ and $P_K = L_p\left[s_K; c_K + o(1)\right]$. For the sieving phase to be successful, the following inequality must hold

$$C^2 \cdot P_{\mathbb{Q}} \cdot P_K > k \cdot B,$$

which is to say

$$L_p\left[s_C; 2c_C + o(1)\right] \cdot P_{\mathbb{Q}} \cdot P_K > k \cdot L_p\left[s_B; c_B + o(1)\right].$$

During the analysis of the index calculus method above, an estimate for the probability that a value assumed by a polynomial is B-smooth was given. The conjecture stated that if $|f(x_1, \ldots, x_n)| < A$ for x_i in some bounded domain and $A = L_p\left[s_A; c_A + o(1)\right]$, then the probability that the value assumed by the polynomial, given randomly chosen x_i, is smooth is given by

$$\frac{\Psi(A, B)}{A} = L_p\left[s_A - s_B; -(s_A - s_B)\frac{c_A}{c_B} + o(1)\right].$$

The integers $c + dm$ all satisfy

$$\begin{aligned} c + dm &< L_p\left[s_k; c_k + o(1)\right] \cdot L_p\left[s_C; c_C + o(1)\right] \\ &< L_p\left[\max\{s_k, s_C\}; c_k + c_C + o(1)\right], \end{aligned}$$

since $m \sim p^{1/k}$. Furthermore, the norm of the ideal $(c + d\alpha)$ satisfies

$$\begin{aligned} N(c + d\alpha) &= (-d)^k f(-c/d) \\ &< (k+1) \cdot D \cdot L_p\left[s_k; c_k + o(1)\right] \cdot L_p\left[s_C; c_C + o(1)\right]^k \\ &= (k+1) \cdot D \cdot L_p\left[s_k; c_k + o(1)\right] \cdot L_p\left[s_C + 1 - s_k; \frac{c_C}{c_k} + o(1)\right] \\ &< (k+1) \cdot D \cdot L_p\left[\max\{s_k, s_C + 1 - s_k\}; c_k + \frac{c_C}{c_k} + o(1)\right]. \end{aligned}$$

By making use of the smooth number estimate, the required probabilities for finding smooth rational and algebraic integers are

$$\begin{aligned} P_{\mathbb{Q}} &\geq L_p\left[\max\{s_k, s_C\} - s_B; -(\max\{s_k, s_C\} - s_B)\frac{c_k + c_C}{c_B} + o(1)\right] \text{ and} \\ P_K &\geq L_p\left[\max\{s_k, s_C + 1 - s_k\} - s_B; -(\max\{s_k, s_C + 1 - s_k\} - s_B)\frac{c_k^2 + c_C}{c_k \cdot c_B} + o(1)\right], \end{aligned}$$

respectively, and these imply that

$$\begin{aligned} s_{\mathbb{Q}} &\geq \max\{s_k, s_C\} - s_B \text{ and} \\ s_K &\geq \max\{s_k, s_C + 1 - s_k\} - s_B. \end{aligned}$$

The requirement that enough smooth numbers are found to successfully solve the linear system implies that

$$\begin{aligned} s_C &\geq \max\{s_B, s_{\mathbb{Q}}, s_K\} \\ &\geq \max\{s_B, s_k - s_B, s_C + 1 - s_k - s_B, s_C - s_B\} \\ &\geq \max\{2s_B, s_k, s_C + 1 - s_k, s_C\} - s_B, \end{aligned}$$

and minimizing $s_T = \max\{s_B, s_C\}$ subject to this last inequality yields $s_B = s_C = \frac{1}{3}$ and $s_k = \frac{2}{3}$.

This analysis has ignored the value of D. Suppose $D = L_p\,[s_D; c_D + o(1)]$. For the above analysis to be correct, it must be the case that $s_D \leq \frac{2}{3}$; however, to get the overall time complexity desired, it is necessary that $s_D \leq \frac{1}{3}$. To ensure that this bound is satisfied, first note that the smooth number estimate gives

$$\begin{aligned} \frac{1}{D} &\leq \frac{\Psi(Dm, B)}{Dm} \\ &= L_p\left[\max\{s_D, s_k\} - s_B; -(\max\{s_D, s_k\} - s_B)\frac{c_D + c_k}{c_B} + o(1)\right], \end{aligned}$$

which implies that

$$\begin{aligned} s_D &\geq \max\{s_D, s_k\} - s_B \\ &= \max\{s_D, \frac{2}{3}\} - \frac{1}{3}, \end{aligned}$$

and which, in turn, gives the minimum value $s_D = \frac{1}{3}$.

Given the values of the s's found above, the following are known

$$\begin{aligned} c + dm &< L_p\left[\frac{2}{3}; c_k + o(1)\right], \\ N(c + d\alpha) &< L_p\left[\frac{2}{3}; c_k + \frac{c_C}{c_k} + o(1)\right], \\ P_{\mathbb{Q}} &= L_p\left[\frac{1}{3}; -\frac{1}{3}\frac{c_k}{c_B}(1 + o(1))\right], \text{ and} \\ P_K &= L_p\left[\frac{1}{3}; -\frac{1}{3}\frac{c_k^2 + c_C}{c_k c_B}(1 + o(1))\right]. \end{aligned}$$

Again, the requirement that enough smooth numbers be found to successfully solve the linear system implies that

$$2c_C > c_B + \frac{1}{3}\left(\frac{c_k}{c_B} + \frac{c_k^2 + c_C}{c_k c_B}\right).$$

Minimizing $c_T = \max\{2c_B, 2c_C\}$ subject to this inequality yields $c_B = c_C = (\frac{8}{9})^{1/3}$ and $c_k = (\frac{1}{3})^{1/3}$. The key parameters are thus

$$\begin{aligned} B &= L_p\left[\frac{1}{3}; \left(\frac{8}{9}\right)^{1/3} + o(1)\right], \\ k &= \left(\frac{(3 + o(1))\log p}{\log\log p}\right)^{1/3}, \text{ and} \\ C &= L_p\left[\frac{1}{3}; \left(\frac{8}{9}\right)^{1/3} + o(1)\right]. \end{aligned}$$

Finally, the total running time of the number field sieve algorithm is expected to be

$$L_p\left[\frac{1}{3}; \left(\frac{64}{9}\right)^{1/3} + o(1)\right].$$

8 Implementation and Empirical Results

To directly test the theory presented in this paper, implementations of both the index calculus method and the number field sieve were written. Both implementations were written in *C++* and make use of the *NTL* big integer library published by Shoup. The implementations were used to find optimal choices for the various algorithm parameters, and then the running time of each algorithm was compared to the asymptotic estimates given above. Finally, a comparison to Weber's implementation of the number field sieve [20] is given.

The *NTL* big integer library is an excellent base upon which to write implementations of algorithms such as these; however, it did need to be extended in various ways. The extensions that were implemented include: classes for sparse vectors and matrices, linear algebra techniques for solving sparse systems of equations, general purpose factoring methods including the elliptic curve factoring method, and classes for creating factor bases and sieving over them. Some of these extensions will be submitted to Shoup for possible inclusion in the *NTL* library.

The index calculus method requires that at least two parameters be chosen, namely, the smoothness bound, B, and the length of the sieve, C. The number field sieve requires that at least three parameters be chosen, B, C, and the degree of the number field, k. To find optimal values for these parameters empirically, the simplex method described in [1] was used. Many people know the simplex method as a "curve fitting" algorithm; however, it is actually a general purpose optimization algorithm. To find B and C for one of the discrete logarithm algorithms, a three vertex simplex is created using values of B and C that are thought to be optimal (from theory or guessing). The simplex method is then run to find values of B and C that result in the smallest running time. During each run of the discrete logarithm algorithm, the discrete logarithm of 2 is computed in cases where the generator is not 2 and the discrete logarithm of 3 is computed in the case that the generator is 2. For the index calculus method, the computation of an individual logarithm simply verifies that the precomputation was successful (the computation itself takes no time); however, for the number field sieve, computing the logarithm of smaller primes (especially 2) appears to require more time in practice, and therefore, these smaller primes for chosen for this analysis.

By using the simplex method to find optimal choices for B and C (k is chosen manually) for each of several problems of varying sizes, plots of B and C as a function of problem size can be produced. Since the theory implies that these parameters are described by functions of the form $d \cdot L_p[s;c]$, the simplex method can then be used again to fit a curve to the acquired data points. Although the simplex method can be used to find values for all three of the curve parameters (s, c, and d), theory predicts that s will be exactly $\frac{1}{2}$ for the index calculus method and $\frac{1}{3}$ for the number field sieve, and therefore, the curve fitting stage was done twice for each algorithm: once with s fixed and once with s as a parameter. With the limited domain over which the optimizations were done, only the curves fit with s fixed are considered meaningful.

All of the running time measurements presented in this section were performed on an Athlon XP 1700+ PC with 256MB of PC2100 DDR RAM and running the Debian GNU/Linux operating system. The time measurements should include only the CPU time used by the algorithm itself; however, to ensure the accuracy of the measurements, steps were taken to avoid other tasks running on the machine at the same time the measurements were being performed.

8.1 Index Calculus

The index calculus method has two algorithm parameters that need to be chosen appropriately to get a reasonable running time. The parameters are the smoothness bound, B, and the sieve length, C. Optimal values for these parameters were found using the simplex method on randomly chosen problems in the range of 30 bits to 100 bits (size of the modulus), and then the curve $d \cdot L_p[s; c]$ was fit to these data points. The curve was fit twice, once allowing s to vary and once with s fixed at 0.5. The results of this curve fitting are given in table 1. Note that the curves fit with s varying match the theoretic estimates reasonably well (with the curve for B being the largest discrepancy).

Table 1: Optimal parameters for the index calculus method.

Parameter	s varying	s fixed at 0.5
B	$13.8 \cdot L_p[0.779; 0.183]$	$3.33 \cdot L_p[0.5; 0.476]$
C	$2.72 \cdot L_p[0.499; 0.420]$	$2.78 \cdot L_p[0.5; 0.417]$
run time	$53.7 \cdot 10^{-6} \cdot L_p[0.449; 1.10]$	$109 \cdot 10^{-6} \cdot L_p[0.5; 0.916]$

Figures 1 and 2 present the data points and the curves for the optimal choices of the smoothness bound and the sieve length, respectively. Figure 3 displays the running time required for each problem along with the fitted curve. All of these figures display curves computed for s fixed at 0.5.

From this empirical run-time data, it is possible to predict what problems can be solved given enough time (and sufficient storage). These predictions are given in table 2.

8.2 Number Field Sieve

With the number field sieve, there is the added complication of the choice of the degree of the number field, k. Since this number is a small integer (typically between 2 and 5), it could not be fit for using the simplex method. Instead, a range of problems were optimized for each of several choices of k,

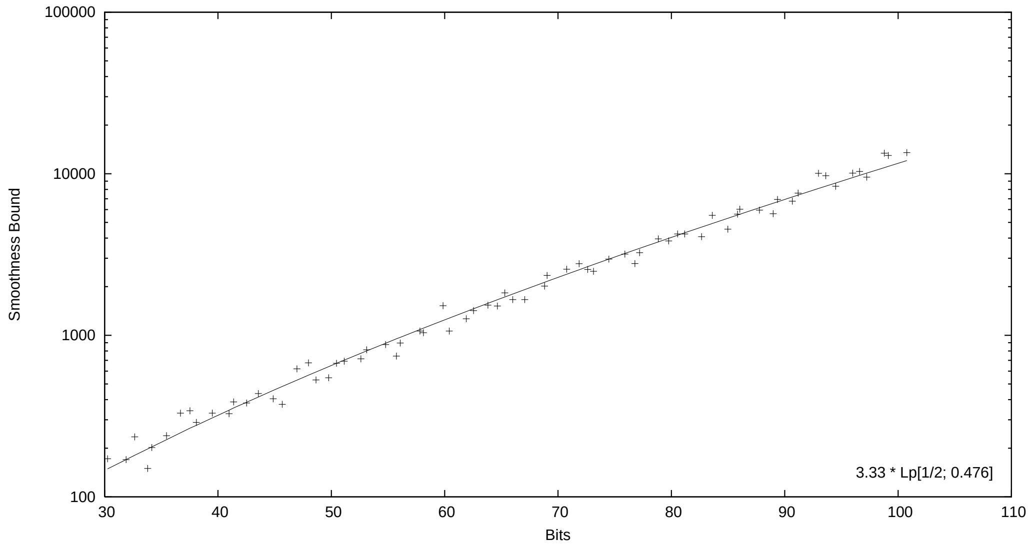

Figure 1: Optimal smoothness bound for the index calculus method as a function of problem size.

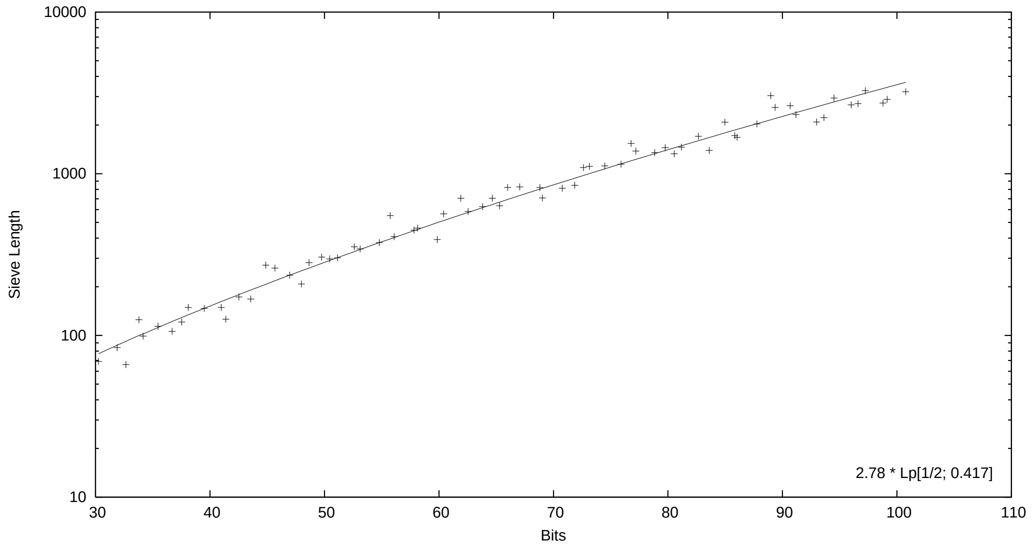

Figure 2: Optimal sieve length for the index calculus method as a function of problem size.

Table 2: Predicted size of problems that can be solved using the index calculus method given sufficient time.

Time Available	Problem Size
1 hour	117 bits (35 digits)
1 day	154 bits (46 digits)
1 week	179 bits (53 digits)
1 month	198 bits (59 digits)
1 year	234 bits (70 digits)

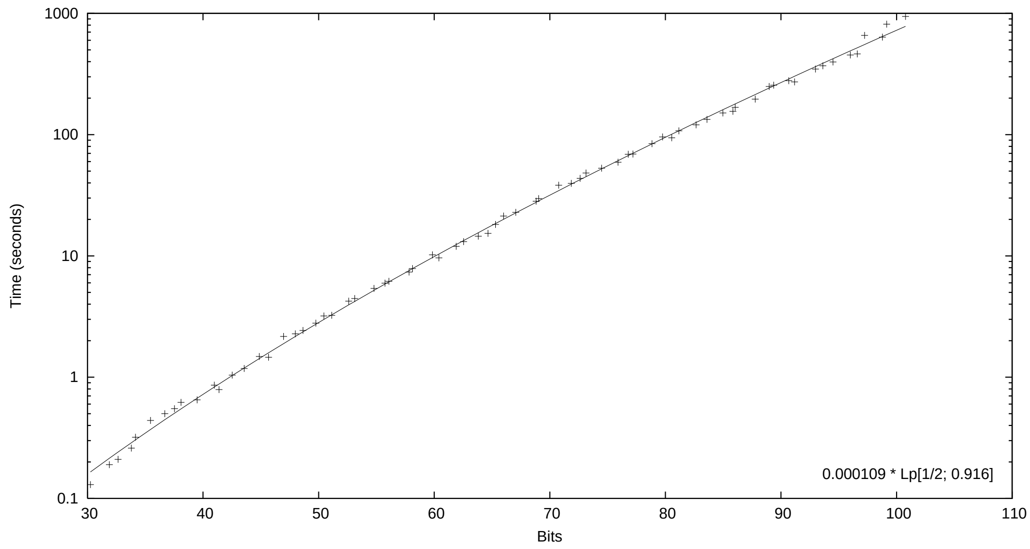

Figure 3: Running time of the index calculus method as a function of problem size.

and each of these ranges were chosen to overlap. In this work, value of 2, 3, and 4 where used for k.

For each choice of problem size and k, two parameters were optimized using the simplex method: the smoothness bound, B, and the sieve width, C. The sieve width is the range of values of c that were checked in the sieving phase of the algorithm. That is $-\frac{1}{2}C < c < \frac{1}{2}C$. The variable d is started at 1 and increases until a sufficient number of relations has been found. The total number of integers sieved is simply the maximum value d attained multiplied by C. This value was noted as part of each data point and is presented below.

Once optimal choices for the algorithm parameters have been found, the curve $d \cdot L_p[s; c]$ can be fit to the data. The results of this curve fitting can be found in table 3. The results for s varying do not match the estimates from theory very well (which is probably do to the limited range of problem sizes tested); however, the results for s fixed at 1/3 fit reasonable well. Note that theory predicts that the optimal choice for B is $O(L_p[1/3; 0.961])$, that the total number of integers that need to be sieved is $O(L_p[1/3; 1.92])$, and that the running time will be $O(L_p[1/3; 1.92])$. The sieve width parameter, C, was not predicted in the theory presented above.

Table 3: Optimal parameters for the number field sieve.

Parameter	s varying	s fixed at 1/3
B	$2.05 \cdot L_p[0.492; 0.585]$	$0.401 \cdot L_p[1/3; 1.06]$
C $(k = 2)$	$12.8 \cdot L_p[0.438; 1.14]$	$2.37 \cdot L_p[1/3; 1.67]$
C $(k = 3)$	$0.253 \cdot L_p[0.189; 2.21]$	$3.74 \cdot L_p[1/3; 1.24]$
C $(k = 4)$	$0.278 \cdot L_p[0.360; 1.31]$	$0.158 \cdot L_p[1/3; 1.46]$
total sieve	$8.99 \cdot L_p[0.503; 1.04]$	$0.332 \cdot L_p[1/3; 1.97]$
run time	$5.41 \cdot 10^{-6} \cdot L_p[0.369; 1.65]$	$0.779 \cdot 10^{-6} \cdot L_p[1/3; 2.03]$

Figure 4 gives the optimal smoothness bound as a function of the size of the modulus. Notice that the optimal choice for the smoothness bound is not dependent on the choice of the degree of the number field; however, in figure 5, it is seen that the optimal choice for the width of the sieve (domain of c) varies significantly as a function of the degree of the number field. The total number of pairs of integers, c and d, involved in the sieve is given in figure 6 and is strongly correlated with the total running time, given in figure 7.

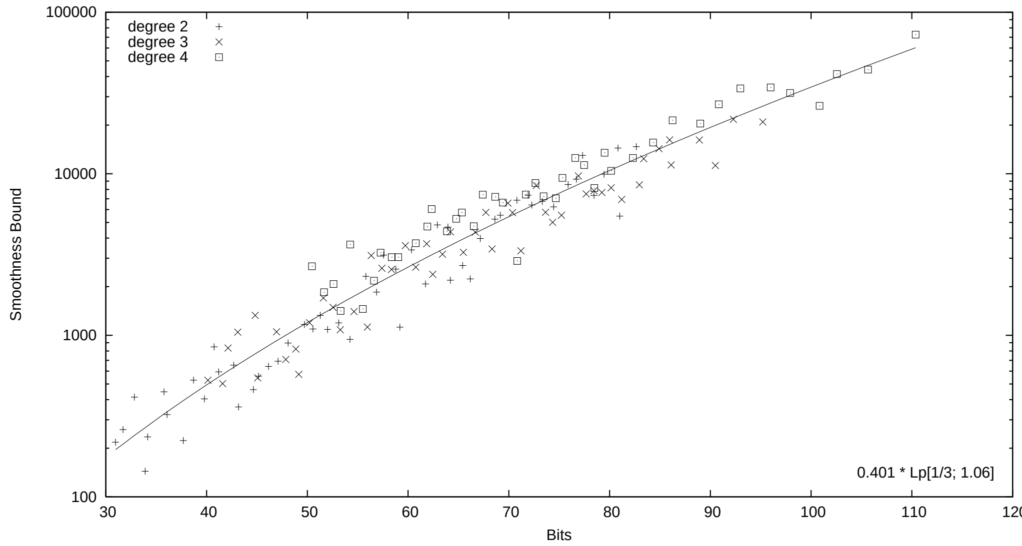

Figure 4: Optimal smoothness bound for the number field sieve as a function of problem size.

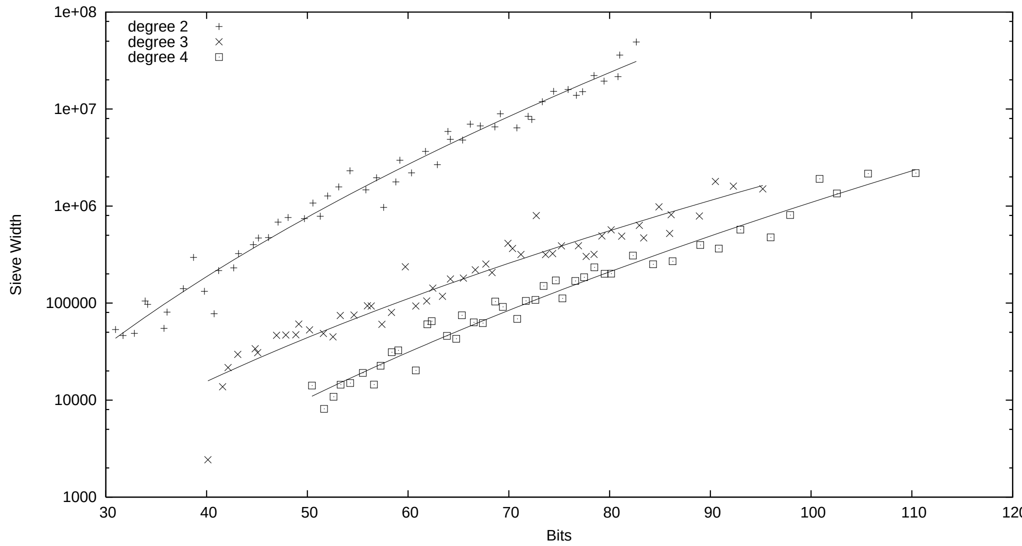

Figure 5: Optimal width of the sieve using in the number field sieve as a function of problem size.

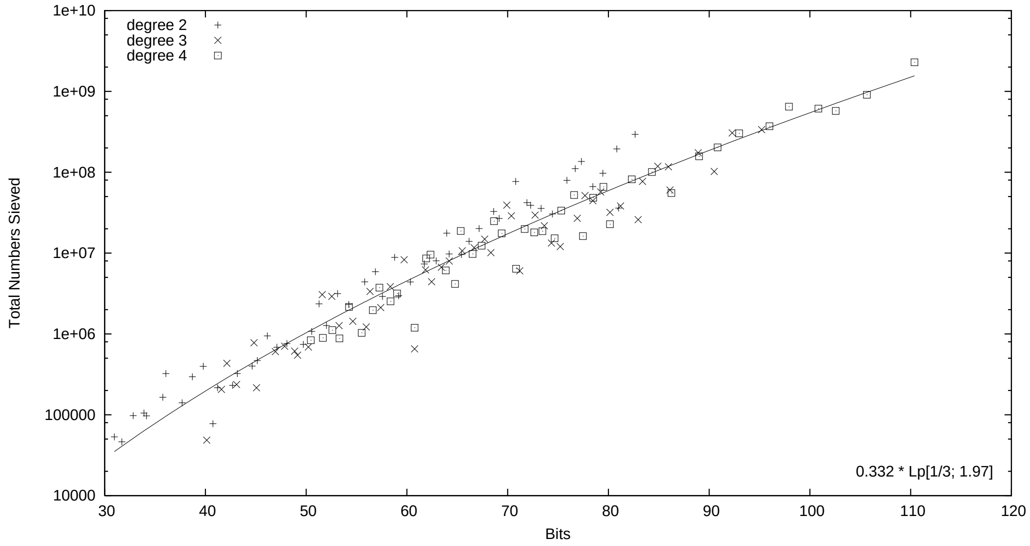

Figure 6: Total number of integers sieved for the number field sieve as a function of problem size.

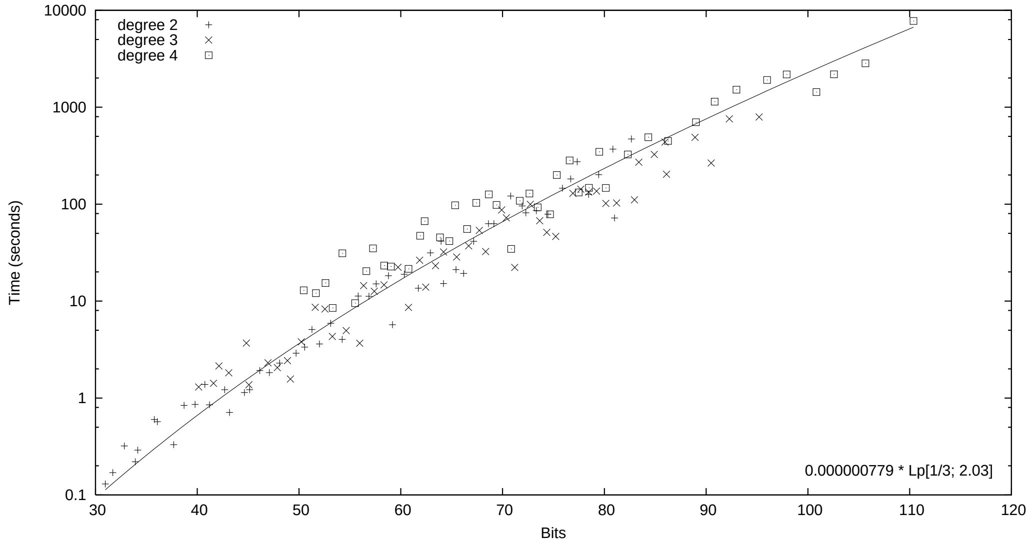

Figure 7: Running time of the number field sieve as a function of problem size.

As was the case with the index calculus data, this empirical run-time data for the number field sieve can also be used to predict what problems can be solved given enough time and storage. These predictions are presented in table 4.

Table 4: Predicted size of problems that can be solved using the number field sieve given sufficient time.

Time Available	Problem Size
1 hour	104 bits (31 digits)
1 day	137 bits (41 digits)
1 week	160 bits (48 digits)
1 month	178 bits (53 digits)
1 year	213 bits (64 digits)

8.3 Results from the Literature

In [20], Weber gave two examples: a 25-digit example and a 40-digit example. Those two problems have been solved using the implementation developed for this work.

The 25-digit problem to solve is

$$7^x \equiv 17 \pmod{1234567890123456789000421}.$$

Weber's solution to the problem involved the use of a degree 3 number field, a smoothness bound of 2400, and required that a 728×703 linear system be solved. Using hardware he had available at the time (in 1995), the sieving took 12 hours and the linear algebra required 8 hours. Using the more modern hardware described above, a number field of degree 3, a smoothness bound of 9000, and having to solve a 1706×1706 linear system took a mere 2.5 minutes (70 seconds to sieve and 75 seconds for the algebra). Of course, the answer was correct but is irrelevant here. For all the details regarding this problem, including the answer, see [20].

The 40-digit problem is

$$23^x \equiv 29 \pmod{3108193812051968080419611909199224122909}.$$

For this problem Weber also chose a degree 3 number field, but he chose to use two different smoothness bounds, one for the rational integers, 12503, and a different bound for the algebraic integers, 17321. He claims that the sieving required only 21 hours on the same hardware used for the 25-digit problem. This claim is hard to believe when one notes that he sieved through more than 10 times the numbers in the latter problem but required less than twice the time compared with the former problem. The linear system that needed to be solved was 3500×3477 and was completed in 40 minutes on a massively parallel Paragon system.

To solve this problem using the hardware described above, a degree 3 number field was used, the smoothness bound was 112000, and a 17018×17018 linear system had to be solved. The sieving phase took 2.5 hours and the linear algebra took 3.2 hours for a total time of 5.7 hours to find the discrete logarithm

$$x = log_{23}29 = 176114974145347413230457520171564394092 0.$$

The current "world record" for discrete logarithm computations of this type is held by Joux and Lercier. In early 2001 they succeeded at computing a discrete logarithm in a field with a 120-digit prime modulus. The problem required 10 weeks of computer time and made use of the methods described in [9]. The implementation developed for this work is not yet ready to challenge this record, but it may be soon.

9 Research Directions

It has been shown that the ability to solve the discrete logarithm problem depends greatly on the structure of the group in which the problem has been defined. If the group elements are opaque and no structure beyond the group operation and inverse elements is known, then Shoup has proven in [17] that the discrete logarithm problem requires time that is at least $O\left(\sqrt{N}\right)$, where N is the order of the group. In this paper, several methods for achieving this bound were presented. This is great news for people hoping the make use of the discrete logarithm problem in cryptographic applications; however, a cryptographically useful group lacking additional structure has yet to be found. The group of points on an elliptic curve appears to come close, but no proof that these groups possess no additional structure is known. It remains possible that some researcher might discover a new way of looking at these groups that will greatly help the discrete logarithm problem, and therefore, researchers continue to caution against the use of these groups for cryptographic applications.

On the other hand, Enge and Gaudry [6] have shown that if a concept of smoothness exists within the group, then a sub-exponential discrete logarithm algorithm can always be constructed. More specifically, they have shown that algorithms with an estimated running time of $L_p\,[1/2; c]$, for some constant c, are possible. Their approach is very similar to the index calculus method described above; however, they have generalized the method to arbitrary groups possessing a smoothness property.

For many of the groups possessing a smoothness property, it is possible to compute discrete logarithms in time $L_p\,[1/3; c]$, for some constant c. The number field sieve method for computing discrete logarithms in the field $\mathbb{F}_p$ was presented above. This method is a non-trivial extension of the index calculus method and required extensive use of algebraic number theory to design and implement. For other groups, such as the multiplicative group of the field $\mathbb{F}_{2^n}$, an algorithm to compute discrete logarithms in time $L_p\,[1/3; c]$ was much easier to create. Coppersmith [2] presented an algorithm which achieves this smaller time bound by using a clever method for finding relations that results is smaller polynomials to test for smoothness.

Since many of the groups to which the Enge and Gaudry method apply now have specialized algorithms to compute discrete logarithms in the smaller time of $L_p\,[1/3; c]$, it is thought that it should be possible to improve the Enge and Gaudry result to this smaller time bound. Such an improvement would be a significant breakthrough as it would likely require finding an algorithm to compute discrete logarithms in $\mathbb{F}_p$, which not only achieves this lower time bound, but does not require the use of algebraic number theory.

A completely different method for computing discrete logarithms is seen in an interesting method by Schnorr [15]. His method claims to factor integers as well as compute discrete logarithms by computing Diophantine approximations and making use of the LLL method to reduce lattices. Although the method is currently not practical, studying the approach may lead to an alternate algorithm for computing discrete logarithms in $\mathbb{F}_p$.

The problems of factoring large integers and computing discrete logarithms in finite fields (and $\mathbb{F}_p$ in particular) have always appeared to go hand in hand. Indeed, if one can compute discrete logarithms in the ring $\mathbb{Z}/N\mathbb{Z}$, then one can easily compute the order of elements within that ring, and from those computations, one can easily factor N. Therefore, the discrete logarithm problem is certainly no easier than the problem of factoring integers. Furthermore, each time an advance is made in the state of the art for factoring (for instance, the number field sieve for factoring), someone always seems to find a way to adapt that advance to the computation of discrete logarithms (as happened with the number field sieve). There is no known formal proof that the discrete logarithm problem in the ring $\mathbb{Z}/N\mathbb{Z}$, or the field $\mathbb{F}_p$, must be as easy as the problem of factoring; however, it appears as though it might be.

It has already been mentioned that the discrete logarithm problem has many applications in cryptography. To assess the security of cryptographic protocols which make use of the discrete logarithm problem, it is important to understand the difficulties associated with computing these logarithms. As part of the proof of security of such protocols, it is often desirable to "reduce" the security of the protocol to the problem of computing discrete logarithms. Doing this ensures that breaking the protocol is at least as difficult as finding an efficient algorithm for computing discrete logarithms. In some cases such a reduction is not possible, however. One example is the Diffie-Hellman conjecture described at the beginning of this paper. In this case it is not known if breaking the Diffie-Hellman protocol is as hard as computing discrete logarithms. Clearly, a strong understanding of the difficulties associated with computing discrete logarithms will aid in the development of protocols that make use of this problem for their security.

References

[1] Marco S. Caceci and William P. Cacheris, *Fitting curves to data*, Byte **May** (1984), 340–362.

[2] Don Coppersmith, *Fast evaluation of logarithms in fields of characteristic two*, IEEE Transactions on Information Theory **IT-30** (1984), no. 4, 587–594.

[3] Don Coppersmith, Andrew M. Odlzyko, and Richard Schroeppel, *Discrete logarithms in $GF(p)$*, Algorithmica **1** (1986), 1–15.

[4] K. Dickman, *On the frequency of numbers containing prime factors of a certain relative magnitude*, Arkiv för Matematik, Astronomi och Fysik **22** (1930), no. 10, 1–14.

[5] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **IT-22** (1976), no. 6, 644–654.

[6] A. Enge and P. Gaudry, *A general framework for subexponential discrete logarithm algorithms*, Polytechnique (2000).

[7] Daniel M. Gordon, *Discrete logarithms in $GF(p)$ using the number field sieve*, SIAM Journal on Discrete Mathematics **6** (1993), no. 1, 124–138.

[8] Adolf Hildebrand and Gerald Tenenbaum, *On integers free of large prime factors*, Transactions of the American Mathematical Society **296** (1986), no. 1, 265–290.

[9] Antoine Joux and Reynald Lercier, *Improvements on the general number field sieve for discrete logarithms in prime fields*, Mathematics of Computation (2000).

[10] H. W. Lenstra Jr., *Factoring integers with elliptic curves*, The Annals of Mathematics **126** (1987), 649–673.

[11] B. A. LaMacchia and A. M. Odlyzko, *Solving large sparse linear systems over finite fields*, Lecture Notes in Computer Science **537** (1991), 109–133.

[12] Stephen C. Pohlig and Martin E. Hellman, *An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance*, IEEE Transactions on Information Theory **IT-24** (1978), no. 1, 106–110.

[13] J. M. Pollard, *Monte carlo methods for index computation (mod p)*, Mathematics of Computation **32** (1978), no. 143, 918–924.

[14] Oliver Schirokauer, *Discrete logarithms and local units*, Philosophical Transactions: Physical Sciences and Engineering **345** (1993), 409–423.

[15] C. P. Schnorr, *Factoring integers and computing discrete logarithms via diophantine approximation*, Lecture Notes in Computer Science **547** (1991), 281–293.

[16] Daniel Shanks, *Class number, a theory of factorization, and genera*, Symposium Pure Mathematics, 1972.

[17] Victor Shoup, *Lower bounds for discrete logarithms and related problems*, Theory and Application of Cryptographic Techniques, 1997, pp. 256–266.

[18] Edlyn Teske, *Speeding up pollard's rho method for computing discrete logarithms*, Proceedings of ANTS III (1998), 541–553.

[19] ———, *Computing discrete logarithms with the parallelized kangaroo method*, Proceedings of the Com2MaC Workshop on Cryptography (2000), 129–145.

[20] Damain Weber, *An implementation of the general number field sieve to compute discrete logarithms mod p*, Theory and Application of Cryptographic Techniques, 1995, pp. 95–105.

[21] Douglas H. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory **IT-32** (1986), no. 1, 54–62.