A Parallel Implementation of Pollard's Rho Algorithm for Discrete Logarithms on Elliptic Curves

Christopher Pardy

April 2015

Table of Contents

Abstract **1**

1 Introduction **2**

1.1 Certicom ECC Challenge 2

1.2 Elliptic Curves Over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ 3

1.2.1 The $\mathbb{F}_p$ Case 3

1.2.2 The $\mathbb{F}_{2^m}$ Case 5

2 The Discrete Logarithm Problem **9**

2.1 Methods for solving Discrete Logarithm Problems 10

2.1.1 Exhaustive Search 11

2.1.2 Baby-Step Giant-Step Algorithm 11

2.1.3 Pollard's Rho Algorithm 13

2.2 Parallel Pollard's Rho Algorithm 15

3 Implementation and Results **20**

3.1 Implementation Details 20

3.2 Results of Implementation 22

3.3 Time Estimates 28

Appendices **33**

Appendix A **34**

Appendix B **65**

Abstract

The use of elliptic curves in cryptography was first suggested by Koblitz and Miller in 1985 [5,7], where existing cryptosystems were adapted to work with points on elliptic curves. The elliptic curve discrete logarithm problem (ECDLP) for points P and Q on an elliptic curve E is to find an integer l such that $Q = lP$. In 1997 Certicom announced their elliptic curve cryptography challenge consisting of a collection of discrete logarithm problems, meant to spur interest in the ECDLP. In this thesis we take a look at some of the methods that can be used to solve this problem, in particular an implementation of the parallelized Pollard's rho algorithm is developed and examined.

Chapter 1

Introduction

1.1 Certicom ECC Challenge

In 1997 Certicom presented their Elliptic Curve Cryptography challenge in order to stimulate interest and research into the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP), and the security of elliptic curve cryptosystems. The ultimate goal was to confirm the security comparisons of the security levels of ECC with such systems as RSA, and DSA.

The challenge presents two challenge levels. Level I, which has 109 and 131 bit challenges, and Level II which contains 163, 191, 239, 359 bit challenges. The 109-bit challenges were solved by 2004 [12]. The 131-bit challenges however prove more

difficult and have not yet been solved, though they may be within reach. The two challenge sets contain challenges in the finite fields $\mathbb{F}_p$ (the integers modulo an odd prime p), and $\mathbb{F}_{2^m}$ (the field of 2^m elements).

1.2 Elliptic Curves Over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$

1.2.1 The $\mathbb{F}_p$ Case

An elliptic curve, over the field $\mathbb{F}_p$, defined by the parameters $a, b \in \mathbb{F}_p$, where $4a^3 + 27b^2 \neq 0$ for an odd prime $p > 3$ is the set

$$E = \{(x, y) \in \mathbb{F}_p^2 | y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\},$$

where $\mathcal{O}$ is called the *the point at infinity.* E forms an abelian group under the addition rules [12] defined below:

1. $\mathcal{O} + \mathcal{O} = \mathcal{O}$
2. $(x, y) + \mathcal{O} = \mathcal{O} + (x, y) = (x, y)$, $\forall (x, y) \in E$
3. $(x, y) + (x, -y) = \mathcal{O}$, $\forall (x, y) \in E$
4. (Rule to add distinct points that are not inverses of each other)

Let $P = (x_1, y_1) \in E$, $Q = (x_2, y_2) \in E$, where $x_1 \neq x_2$ then $P + Q = (x_3, y_3)$ is given by

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

$$x_3 = \lambda^2 - x_1 - x_2$$

$$y_3 = \lambda(x_1 - x_3) - y_1.$$

5. (Doubling a point) Let $P = (x_1, y_1) \in E$ where $y_1 \neq 0$ (in this case $P = -P$), then $2P = (x_3, y_3)$, given by

$$\lambda = \frac{3x_1^2 + a}{2y_1}$$

$$x_3 = \lambda^2 - 2x_1$$

$$y_3 = \lambda(x_1 - x_3) - y_1.$$

An example

For example consider the elliptic curve, E, over $\mathbb{Z}_7$ defined by the parameters $a = 1$, and $b = 1$, then $E = \{\mathcal{O}, (0,1), (0,6), (2,2), (2,5)\}$.. Letting $P = (2,5)$, and $Q = (0,6)$ computing $P + Q$ gives us

$$\lambda = \frac{6-5}{0-2} = \frac{1}{-2} = -4 = 3$$

$$x_3 = (3)^2 - 2 - 0 = 9 - 2 = 0$$

$$y_3 = 3(2-0) - 5 = 6 - 5 = 1$$

so we see $P + Q = (0, 1)$. Computing $Q + P$ we have

$$\lambda = \frac{5-6}{2-0} = \frac{-1}{2} = -4 = 3$$

$$x_3 = (3)^2 - 2 - 0 = 9 - 2 = 0$$

$$y_3 = 3(0-0) - 6 = -6 = 1$$

So we see $Q + P = (0, 1) = P + Q$.

1.2.2 The $\mathbb{F}_{2^m}$ Case

The Finite Field of 2^m Elements

The finite field $\mathbb{F}_{2^m}$ is the ring $\mathbb{Z}_2[x]/(f(x))$ for some irreducible polynomial $f(x)$, of degree m. So $\mathbb{F}_{2^m} = \{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0 | a_i \in \mathbb{Z}_2\}$. Addition and multiplication are preformed modulo $f(x)$, so for $a = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0$ and $b = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_1x + b_0$ we have:

- $a + b = c_{m-1}x^{m-1} + c_{m-2}x^{m-2} + \cdots + c_1x + c_0$, where $c_i = a_i + b_i (\mod 2)$
- $a \cdot b = d_{m-1}x^{m-1} + d_{m-2}x^{m-2} + \cdots + d_1x + d_0$, where $d_{m-1}x^{m-1} + d_{m-2}x^{m-2} + \cdots + d_1x + d_0$ is remainder of $(a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0)(b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \cdots + b_1x + b_0)$ after division by $f(x)$ over $\mathbb{Z}_2[x]$.

Note that elements of $\mathbb{F}_{2^m}$ can easily be represented by m-bit strings, where $(a_{m-1}a_{m-2}\cdots a_0)$ represents the element $a_{m-1}x^{m-1} + \cdots + a_0$.

Elliptic Curves Over $\mathbb{F}_{2^m}$

Over a field of characteristic two a general elliptic curve is written as

$$E = \{(x, y)|y^2 + ay + by = x^3 + cx^2 + dxy + ex + f\} \cup \{\mathcal{O}\}$$

however for the purposes of the challenge all curves are of the form

$$E = \{(x, y)|y^2 + xy = x^3 + ax^2 + b\} \cup \{\mathcal{O}\},$$

where $\mathcal{O}$ is the point at infinity. E forms an abelian group under the following addition rules [12]:

1. $\mathcal{O} + \mathcal{O} = \mathcal{O}$
2. $(x, y) + \mathcal{O} = \mathcal{O} + (x, y) = (x, y)$, $\forall (x, y) \in E$
3. $(x, y) + (x, x + y) = \mathcal{O}$, $\forall (x, y) \in E$
4. (Rule to add distinct points that are not inverses of each other)

   Let $P = (x_1, y_1) \in E$, $Q = (x_2, y_2) \in E$, where $x_1 \neq x_2$ then $P + Q = (x_3, y_3)$ is given by

   $$\lambda = \frac{y_2 + y_1}{x_2 + x_1}$$

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$$

$$y_3 = \lambda(x_1 + x_3) + y_1 + x_3.$$

5. (Doubling a point) Let $P = (x_1, y_1) \in E$ where $x_1 \neq 0$ (in this case $P = -P$), then $2P = (x_3, y_3)$, given by

$$\lambda = x_1 + \frac{y_1}{x1}$$

$$x_3 = \lambda^2 + \lambda + a$$

$$y_3 = x_1^2 + (\lambda + 1)x_3.$$

An example

Consider the elliptic curve,E, defined over $\mathbb{F}_{2^m} = \mathbb{Z}_2[x]/(x^2 + x + 1)$, defined by the parameters $a = b = (10)$. Then we have

$$E = \{\mathcal{O}, ((0), (11)), ((1), (10)), ((1), (11)), ((11), (0)), ((11), (11))\}.$$

Letting $P = ((11), (11)) = (x + 1, x + 1)$, and $Q = ((01), (10)) = (1, x)$, computing $P + Q$ gives us

$$\lambda = \frac{x + 1 + 1}{x + 1 + x} = \frac{1}{x} = x + 1$$

$$x_3 = (x+1)^2 + (x+1) + (x+1) + 1 + x = x^2 + 2x + 1 + 1 + x = x^2 + x = (x+1) + x = 1$$

$$y_3 = (x + 1)(x + 1 + 1) + 1 + (1 + x) = (x + 1)x + x = x + 1$$

so we have $P + Q = (1, x + 1) = ((01), (11))$. Computing $Q + P$ we get

$$\lambda = \frac{x + x + 1}{1 + x + 1} = \frac{1}{x} = x + 1$$

$$x_3 = (x+1)^2 + (x+1) + 1 + (x+1) + x = x^2 + 2x + 1 + 1 + x = x^2 + x = (x+1) + x = 1$$

$$y_3 = (x + 1)(1 + 1) + 1 + x = (x + 1)(0) + 1 + x = x + 1$$

so we see that $Q + P = (1, x + 1) = ((01), (11)) = P + Q$.

Chapter 2

The Discrete Logarithm Problem

Informally the discrete logarithm problem is the problem of inverting exponentiation. There are different settings in which problem can be posed. Two commonly studied versions of the problem are *the discrete logarithm problem over* $\mathbb{F}_p$ (DLP) and *the elliptic curve discrete logarithm problem* (ECDLP), as defined below:

- The discrete logarithm problem over $\mathbb{F}_p$: Given $\mathbb{F}_p$ a finite field, and $x, y \in \mathbb{F}_p$, determine, if possible, the least $l \in \mathbb{N}$ such that $x^l = y$. Here l is denoted as $\log_x y$.

- The elliptic curve discrete logarithm problem: Given an elliptic curve E, and two points $P, Q \in E$, find, if possible, an $l \in \mathbb{N}$ such that $lP = Q$. Here l is denoted $\log_P(Q)$ and P is called the base of the logarithm.

For example in $\mathbb{Z}_{11}$ $5^4 = 9(\mod 11)$ so $\log_5(9) = 4$.

The discrete logarithm problem is important in several cryptographic schemes such as the ElGamal cryptosystem [3] and Diffie-Hellman key exchange protocol [2], these same schemes can be adapted to work with elliptic curves [5, 7].

2.1 Methods for solving Discrete Logarithm Problems

There are a number of ways to solve DLP and ECDLP, however not all algorithms that solve the DLP can be adapted to solve the ECDLP as shown by the index calculus algorithm which solves DLP in sub-exponential time [6], but it is not believed to be applicable to the ECDLP [14]. This section will cover a few important algorithms used in solving the ECDLP.

The following notation is used:

- $\mathbb{F}_q$ is a finite field of order q
- E is an elliptic curve over $\mathbb{F}_q$
- P is an element of E
- n is the order of P, i.e. n is the least positive integer such that $nP = \mathcal{O}$

- Q is another element of E

The goal is to find an integer l such that $lP = Q$, provided such an l exists. For the remainder of this section we shall assume that such an l does exist for the given P and Q. Also note that we can write $l = qn + r$ where $0 \leq r < n$, so $lP = (qn + r)P = q(nP) + rP = q\mathcal{O} + rP = \mathcal{O} + rP = rP$. We may therefore assume without loss of generality that $0 \leq l < n$.

2.1.1 Exhaustive Search

The easiest to understand, and the least efficient method is to compute multiples of P in succession until Q is obtained. In the worst case($Q = (n-1)P$) this will take $n - 2$ elliptic curve operations.

2.1.2 Baby-Step Giant-Step Algorithm

The baby-step giant-step algorithm uses a time-memory trade-off to improve upon exhaustive search. The algorithm is based on the fact that is if $Q = lP$ and $m = \lceil \sqrt{n} \rceil$ then we can write $l = im + r$ for $0 \leq i, r < m$. The algorithm is as follows:

1. Set $m = \lceil \sqrt{n} \rceil$.

2. Construct a table of the pairs (r, rP) for $0 \leq r < m$.

3. Set $\alpha = -mP$

4. Set $\beta = Q$

5. For i from 0 to $m - 1$ do:

   (a) Check is β is the second component of a pair in the table.

   (b) If it is then the pair is (r, rP) for some r so return $l = im + r$ as the solution.

   (c) Set $\beta = \beta + \alpha$

Of course this algorithm requires a fast way to search for a point in the table, one such way is to sort the table by second component of the pairs (in $O(n \log n)$ comparisons) and then using a binary search. We see that this algorithm requires memory for $O(\sqrt{n})$ points in the table. Constructing the table takes $O(\sqrt{n})$ elliptic curve additions, finding the logarithm in step 5 takes $O(\sqrt{n})$ additions and $O(\sqrt{n})$ table lookups(each using $O(\log(\sqrt{n}))comparaisons$. The memory requirement for this algorithm limits it's usability in practice, for example for n an 80-bit integer we would need to store a table of about $\sqrt{n} \approx 2^{40}$ points which would require more than a terrabyte of memory.

2.1.3 Pollard's Rho Algorithm

Pollard's rho algorithm [11] is a probabilistic algorithm with the same expected time complexity as the baby-step giant-step algorithm, but with negligible memory requirement. Thus it is preferred for practical instances of the ECDLP. We may assume that n is prime (as is the case with the ECC challenges) due to the *Pollhig-Hellman algorithm* [10] which reduces the problem to determining the logarithm modulo each each prime factor of n and then using the Chinese remainder theorem to construct l.

We first partition the set $S = \{\mathcal{O}, P, 2P, 3P, ..., (n-1)P\}$ into three sets $S1$, $S2$, and $S3$ of roughly equal size, where membership of each set is easily testable. Then we pick two integers $a_0, b_0 \in [1, n-1]$. Set $x_0 = a_0Q + b_0P$ and define the sequence $x_0, x_1, x_2, \ldots$ by the rule $x_{i+1} = f(x_i)$ where the function f is defined by

$$f(x) = \begin{cases} Q + x \text{ if } x \in S_1 \\ 2x \text{ if } x \in S_2 \\ P + x \text{ if } x \in S_3 \end{cases}$$

for $i \geq 0$. This sequence then in turn defines two integer sequences $a_0, a_1, a_2, \ldots$, and $b_0, b_1, b_2, \ldots$ where

$$a_{i+1} = \begin{cases} a_i + 1 \mod n \text{ if } x_i \in S_1 \\ 2a_i \mod n \text{ if } x_i \in S_2 \\ a_i \text{ if } x_i \in S_3 \end{cases}$$

and

$$b_{i+1} = \begin{cases} b_i \text{ if } x_i \in S_1 \\ 2b_i \mod n \text{ if } x_i \in S_2 \\ b_i + 1 \mod n \text{ if } x_i \in S_3 \end{cases}$$

so we have $x_i = a_iQ + b_iP$. Due to the fact that S is finite the sequence $x_0, x_1, \ldots$ must eventually begin to cycle, thus there must be distinct integers i, j such that $i \neq j$ and $x_i = x_j$ (called a collision) so

$$a_iQ + b_iP = a_jQ + b_jP$$

$$(a_i - a_j)Q = (b_j - b_i)P$$

$$(a_i - a_j)lP = (b_j - b_i)P$$

$$(a_i - a_j)l = b_j - b_i \mod n$$

so if $a_i \neq a_j \mod n$ (called a useful collision) we see $l = (b_j - b_i)(a_i - a_j)^{-1}$, otherwise we restart with new new integers a_0 and b_0. Thus finding l reduces to finding i, j. If we were to run through the sequence storing each point generated, then letting X represent the number of points stored before a collision is found and assuming f behaves pseudo-randomly we have

$$P(X > k) = \prod_{i=1}^{k}(1 - \frac{i}{n}) \approx e^{\frac{-k^2}{2n}}$$

for large n and $k \in O(\sqrt{n})$ [9]. We then see that

$$E(X) = \sum_{i=1}^{\infty} iP(x = i) = \sum_{i=1}^{\infty} i(P(X > i-1) - P(X > i)) = \sum_{i=0}^{\infty} P(X > i)$$

so

$$E(X) \approx \sum_{i=0}^{\infty} e^{\frac{-i^2}{2n}} \approx \int_0^{\infty} e^{\frac{-x^2}{2n}} = \sqrt{\frac{\pi n}{2}}.$$

However this method is also expected to require space to store $\sqrt{\frac{\pi n}{2}}$ points. One way to cut down the storage requirement is to use *Floyd's cycle finding algorithm* where at each step x_i and x_{2i} are computed until an i such that $x_i = x_{2i}$ is found, this is guaranteed to happen since the sequence must enter a cycle with first element x_μ and cycle length λ, so for $i \geq \mu$ and $k \geq 0$ we have $x_i = x_{i+k\lambda}$ hence if $k = \left\lceil \frac{\mu}{\lambda} \right\rceil$ then $i = k\lambda$ will suffice. Using Floyd's algorithm in tandem with Pollard's rho is known to have expected running time in $O(\sqrt{n})$ if f acts pseudo-randomly [1].

2.2 Parallel Pollard's Rho Algorithm

Unfortunately due to the serial nature of computing the sequence $x_1, x_2, \ldots$ Pollard's rho algorithm is not directly parallelizable. If there are m processors each generating it's own sequence of points then the expected number of iterations by each processor before a collision is found is $\sqrt{\frac{\pi n}{2m}}$ [15]. In fact this approach would be expected to

use $m\sqrt{\frac{\pi n}{2m}} = \sqrt{\frac{\pi nm}{2}}$, iterations in total before reaching a collision, a factor of $\sqrt{m}$ more than the serial case.

In order to more efficiently parallelize the algorithm the method described in [15] has each processor generating its own sequence, with random starting points, using the same recursive rule as the serial version of the algorithm, until a *distinguished point*, $x_d = a_dQ + b_dP$, is found, where a point is distinguished if it satisfies an easily testable property such as having a certain number of leading zero bits in it's first coordinate. Once a distinguished point is found the processor begins working on a new sequence with a random starting point. The point x_d and the integers a_d and b_d are then saved to a list of distinguished points, this is repeated until a collision of distinguished points is found, a collision of distinguished points indicates that two of the sequences produced must have had a collision at or before the distinguished point, as seen in figure 2.1. A collision, $x_2 = x_1$ is useful if $a_1 \neq a_2$ as we can then determine l.

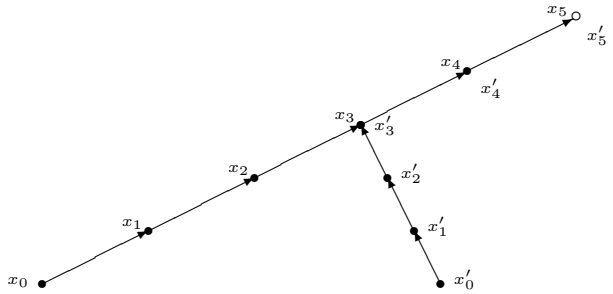

Figure 2.1: Image demonstrating a collision of the two sequences $x_1, x_2, \ldots$ and $x'_1, x'_2, \ldots$, the hollow dot represents a distinguished point.

To determine the runtime of this algorithm we follow the details given in [15]. Let p be the proportion of collisions that are useful and let X denote the number of iterations across all processes before a useful collision is occurs then for large n and $k \in O(\sqrt{\theta n})$

$$P(X > k) = \prod_{i=1}^{k} \left(1 - \frac{pi}{n}\right) \approx e^{\frac{-k^2 p}{2n}}.$$

Hence the expected number of steps required in total to find a useful collision is about $\sqrt{\frac{\pi n}{2p}}$ so the expected number of steps per processor is $\frac{\sqrt{\frac{\pi n}{2p}}}{m}$ after this useful collision occurs the processor involved is expected to produce an additional $\frac{1}{\theta}$ points before reaching a distinguished point, where θ is the proportion of all points which are distinguished. So we see the expected number of steps taken is

$$\frac{\sqrt{\frac{\pi n}{2p}}}{m} + \frac{1}{\theta}.$$

If n is large then we would expected p to be very close to one, if θ is large enough for $\frac{1}{\theta}$ to be overshadowed by $\frac{\sqrt{\frac{\pi n}{2p}}}{m}$, then we could expect an overall running time of about $\frac{\sqrt{\frac{\pi n}{2p}}}{m}T$ where T is the amount of time that it takes to complete one step in the algorithm. So an overall CPU time of $\sqrt{\frac{\pi n}{2}}$ should be expected.

Here we have assumed that all sequences contain distinguished points, this is not necessarily the case so it is possible for a processor to get stuck in a loop without ever finding a distinguished point. This can be remedied by placing a limit on the number of points in a sequence to be generated without finding a distinguished point and having the processor restart with a new starting point if the limit is reached. Assuming the function f acts pseudo-randomly and each point generated is equally likely to be distinguished with probability θ then the number of points generated before reaching a distinguished point is geometrically distributed with mean $\frac{1}{\theta}$. So setting the limit to be $\frac{k}{\theta}$ implies the proportion of sequences abandoned is $(1-\theta)^{\frac{k}{\theta}} \approx e^{-k}$ for small θ, of course θ should be small in order to keep memory requirements low. Now each sequence abandoned will have length k times that of the average so the proportion of work lost is ke^{-k} which can be made as small as we like.

Since the expected number of steps taken by each processor is $\frac{\sqrt{\frac{\pi n}{2p}}}{m} + \frac{1}{\theta}$ in order to keep running time in $O(\sqrt{n})$ we need $\frac{1}{\theta} \in O(\sqrt{n})$ so that the $\frac{1}{\theta}$ dos not dominate the running time. It looks like setting θ to be as large as possible would be

optimal however that would involve storing many points and increasing the memory requirement.

Chapter 3

Implementation and Results

3.1 Implementation Details

Two separate programs implementing the parallel Pollard's rho algorithm were written, one for elliptic curves over $\mathbb{Z}_p$ for $p > 3$ a prime, and another for curves over $\mathbb{F}_{2^m}$. Both are written in C++, and were able to solve the 79-bit instances of Certicom's exercises. In order to deal with large integers that arise the GMP [4] software library was used, Shoup's NTL package [13] was used to handle operations in $\mathbb{F}_{2^m}$. To handle message passing the MPI message passing interface [8] was used.

The structure of the programs message passing is as follows, one process is chosen to be the master process which oversees work done by the other processes (called

slave processes). The master first hands out randomly generated starting points for the sequences that the slaves will create, this is handled by the master in order to avoid seeding multiple random number generators with the same seed as if two processes were generating their own points using identical seeds then they would produce identical sequences, wasting work. When a slave finds a distinguished point it sends it to the master who then places the point in a sorted binary tree, and checks for a useful collision, if no collision is found the master sends a new starting point to the slave. If a useful collision is detected then the master process solves the discrete logarithm. Upon solving the problem the master will then begin to kill the slave processes whenever they send a request for a new starting point, until all processes are finished.

In the context of the programs we call a point distinguished if it has a certain number, say t, of leading zero bits in its binary representation. This property is mainly chosen as it can be checked very easily by the computers. In order to prevent a processor from getting stuck forever in a cycle which has no distinguished points an upper limit of 20θ steps is imposed, after which if no distinguished point is found a new starting point will be requested. Since finding a distinguished point or hitting the limit for number of steps may take a long time, after every few million steps each slave process checks in with the master to determine whether or not it may finish.

This ensures that the processes do not take too long to finalize after a solution is found.

3.2 Results of Implementation

The programs were ran to to examine their capabilities for solving discrete logarithms on elliptic curves. The 79-bit exercises in the ECC challenge were solved correctly giving 9222150721970534568535 0 as the solution to the ECCp-79 exercise in roughly two days running time on 40 cores, and 276856274258963891889538 as the solution to the ECC2-79 exercise in roughly 5 days running on 84 cores. A potential reason that the ECC2-79 exercise took longer despite running on more cores is that the 84 cores it ran on are 32 bit processors as opposed to the 64 bit cores that the ECCp-79 ran on, also operations in a field of characteristic 2 (particularly multiplication) require more work than the integer operations used for ECCp-79.

In order to examine the complexity of the parallelized rho algorithm 3000 elliptic curves along with pairs of points (P, Q) were randomly generated as input for both programs. We will first look at the output from the program dealing with curves over fields of prime order. We see from figures 3.1, 3.2, and 3.3 that the approximate running time of $\sqrt{\frac{\pi n}{2}}$ is pretty good, as we see that the average number of elliptic

curve operations used is close to what we expected.

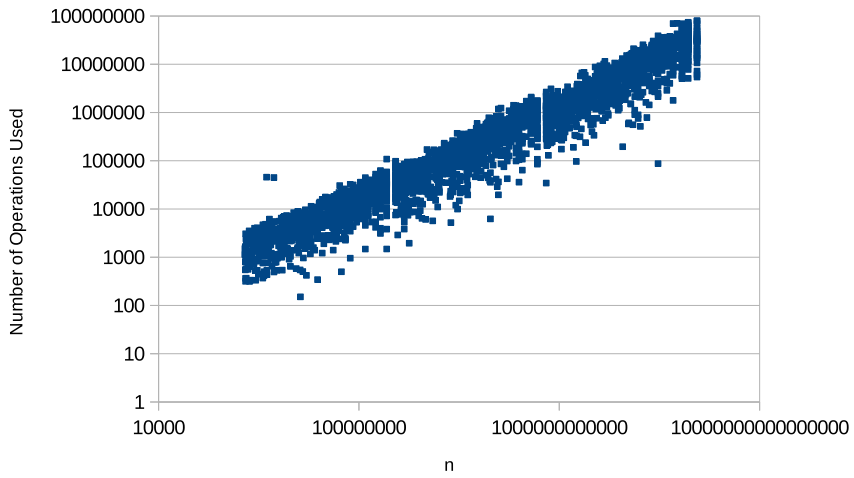

Figure 3.1: A graph of the number of operations to solve a discrete logarithm where n is the order of P.

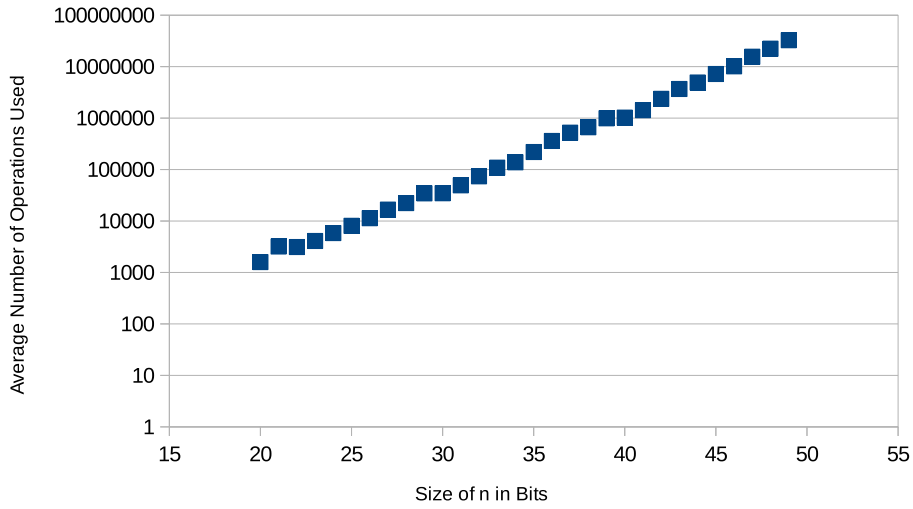

Figure 3.2: A graph of the average number of operations used to solve a discrete logarithm by the bit size of n.

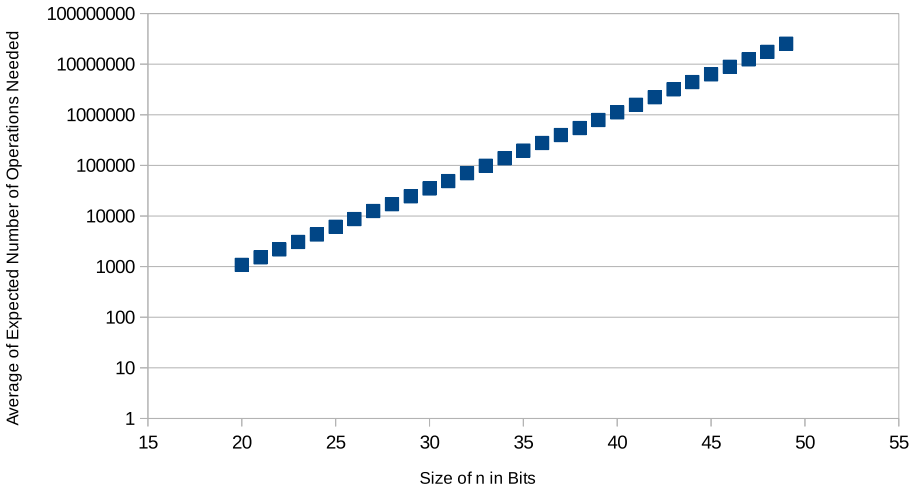

Figure 3.3: A graph of the average expected number of operations needed to solve a discrete logarithm for the n that were used grouped by number of bits.

Examining the output of the program in figures 3.4, 3.5, and 3.6 for curves over $\mathbb{F}_{2^m}$ we see similar results. Although on the lower end of the range there is significant deviation from the expectation, for the larger cases the estimation is pretty good.

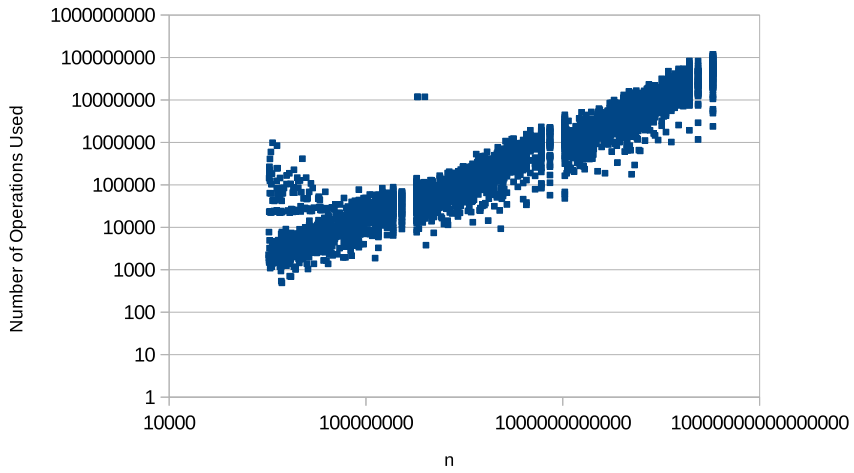

Figure 3.4: A graph of the number of operations to solve a discrete logarithm where n is the order of P.

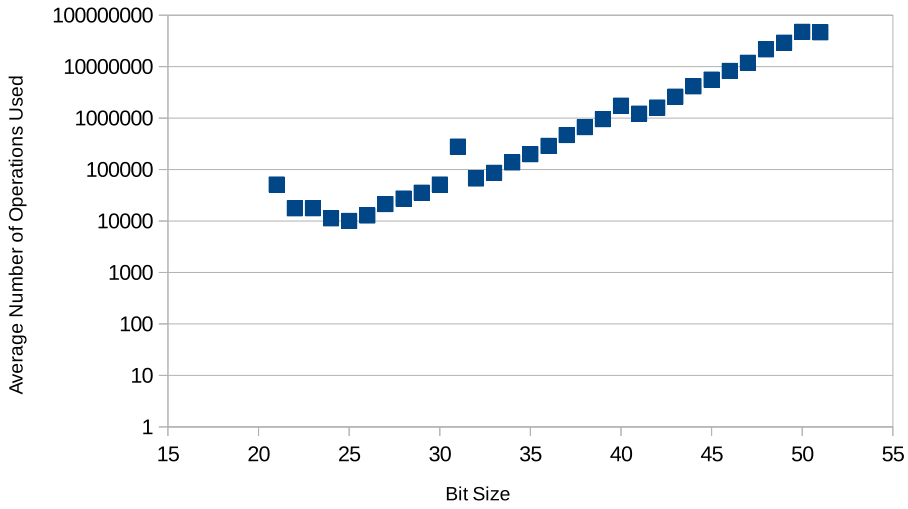

Figure 3.5: A graph of the average number of operations used to solve a discrete logarithm by the bit size of n.

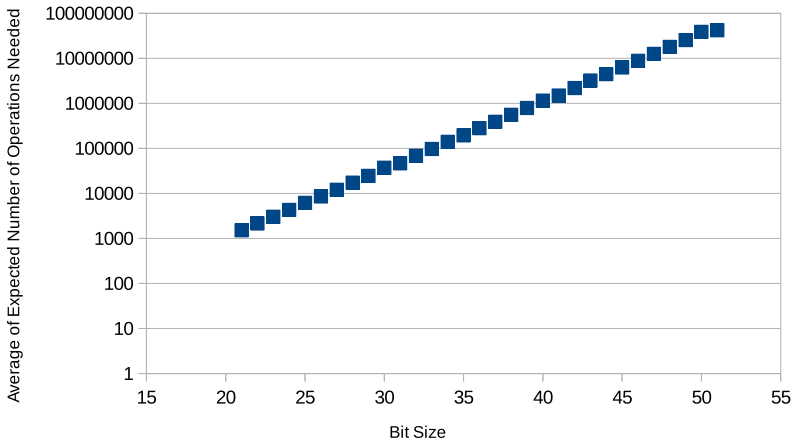

Figure 3.6: A graph of the average expected number of operations needed to solve a discrete logarithm for the n that were used grouped by number of bits.

We will also look at the effect of changing the conditions for a point to qualify as distinguished. To see the effect we vary t which is the number of leading zeros that a distinguished point must have in it's binary representation. Figures 3.2 and 3.2 show us that changing t up to a certain point does not have a large effect. However once a certain threshold is passed the effect is quite pronounced, causing the number of operations used to shoot up by multiple orders of magnitude.

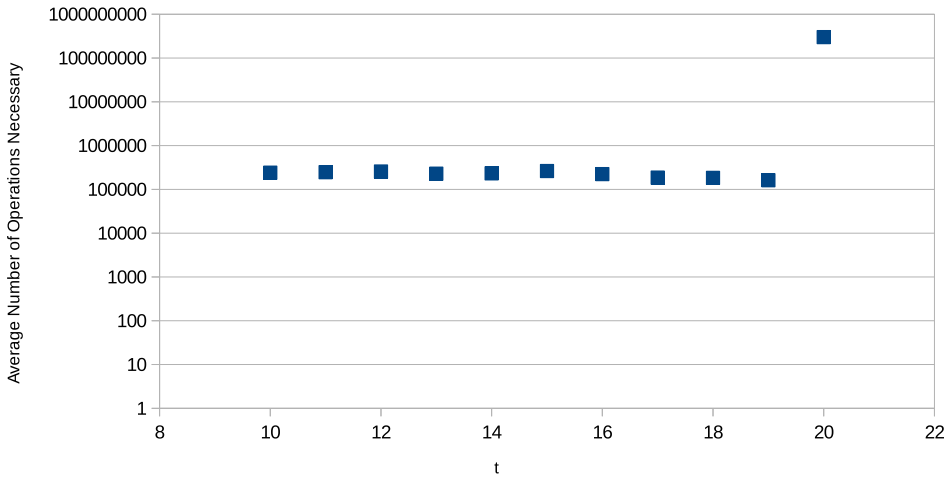

Figure 3.7: The average number of operations, across 50 trials used for each t when solving a discrete logarithm with n a 35-bit number.

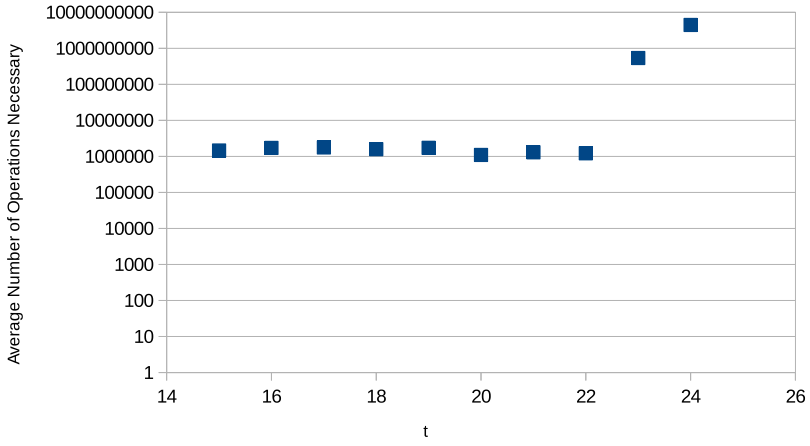

Figure 3.8: The average number of operations, across 25 trials used for each t when solving a discrete logarithm with n a 45-bit number.

3.3 Time Estimates

We can produce estimates of the time that it would take to solve the next unsolved challenge that Certicom has issued using these programs. Using the estimate of $\sqrt{\frac{\pi n}{2}}$ steps required to solve a discrete logarithm problem, we can determine roughly how many operations are needed. The next two unsolved challenges are ECC2-131 and ECCp-131 with respective orders $1361129467683753853898082827025389846147 \approx 1.36 \cdot 10^{39}$ and $1550031797834347859219047037805205710577 \approx 1.55 \cdot 10^{39}$. Looking at table 3.1 we see that the next challenge is rather far away and may be infeasible

for the near future.

Challenge	Number of Operations	Operations per Second	Runtime
ECC2-131	$\approx 4.623 \cdot 10^{19}$	54824.56	$\approx 9.76 \cdot 10^{9}$ days
ECCp-131	$\approx 4.934 \cdot 10^{19}$	132450.33	$\approx 4.31 \cdot 10^{9}$ days

Table 3.1: Estimations for the 131-bit challenges.

Bibliography

[1] Shi Bai and Richard P. Brent. On the efficiency of pollard's rho method for discrete logarithms. In James Harland and Prabhu Manyem, editors, *Fourteenth Computing: The Australasian Theory Symposium (CATS 2008)*, volume 77 of *CRPIT*, pages 125–131, Wollongong, NSW, Australia, 2008. ACS.

[2] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. In *Secure communications and asymmetric cryptosystems*, volume 69 of *AAAS Sel. Sympos. Ser.*, pages 143–180. Westview, Boulder, CO, 1982.

[3] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory*, 31(4):469–472, 1985.

[4] Torbjrn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.0.0 edition, 2014. http://gmplib.org/.

[5] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[6] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[7] VictorS. Miller. Use of elliptic curves in cryptography. In HughC. Williams, editor, *Advances in Cryptology CRYPTO 85 Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer Berlin Heidelberg, 1986.

[8] The Open MPI Development Team. *Open MPI*, 1.4.3 edition, 2012. https://www.open-mpi.org/.

[9] Kazuo Nishimura and Masaaki Sibuya. Probability to meet in the middle. *J. Cryptology*, 2(1):13–22, 1990.

[10] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. Information Theory*, IT-24(1):106–110, 1978.

[11] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Math. Comp.*, 32(143):918–924, 1978.

[12] Certicom Research. Certicom ecc challenge, 2009.

[13] Victor Shoup. *A Tour of NTL*, 9.0.2 edition, 2015. http://www.shoup.net/ntl/doc/tour.html.

[14] Joseph H. Silverman and Joe Suzuki. Elliptic curve discrete logarithms and the index calculus.

[15] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

Appendices

Appendix A

The source code for the program used to solve the ECDLP over $\mathbb{Z}_p$, $p \geq 3$.

pElliptic.h

```
#include<iostream>
#include<gmp.h>
#include<gmpxx.h>
#include<stdlib.h>
#include<stdio.h>
#include<vector>
using namespace std;
mpz_class p;// prime
mpz_class one("1",10); //one as an mpz_class
vector<mpz_class> infty;
mpz_class a,b;// parameters defining the curve
```

```
//computes the inverse of n modulo m
mpz_class invm(mpz_class n,mpz_class m){
        mpz_t g,s,t;
        mpz_init(g);
        mpz_init(s);
        mpz_init(t);
        mpz_gcdext(g,s,t,n.get_mpz_t(),m.get_mpz_t());
        mpz_class ret(s);
        mpz_clear(g);
        mpz_clear(s);
        mpz_clear(t);
        //mpz_clears(g,s,t);
        return ret;
}
//computes the inverse of n modulo p
mpz_class inv(mpz_class n){
        mpz_t g,s,t;
        mpz_init(g);
        mpz_init(s);
        mpz_init(t);
        mpz_gcdext(g,s,t,n.get_mpz_t(),p.get_mpz_t());
        mpz_class ret(s);
```

```
        mpz_clear(g);
        mpz_clear(s);
        mpz_clear(t);
        return ret;
}

//Adds the two points p1 and p2
vector<mpz_class> Eadd(vector<mpz_class> p1, vector<mpz_class> p2){
        vector<mpz_class> p3;
        p3.reserve(2);
        mpz_class lambda,px,py,qx,qy,x3,y3;
        px = p1[0];
        py = p1[1];
        qx = p2[0];
        qy = p2[1];
        if(p1==infty) return p2;
        else if(p2==infty) return p1;
        //two different points
        else if((px-qx)%p!=0){
                lambda = (qy-py)*inv(qx-px) %p;
                x3 = ((lambda * lambda-px-qx)%p);
                y3 = ((lambda*(px-x3)-py)%p);
```

```
        }
        //Doubling a Point
        else if((px-qx)%p==0 && (py-qy)%p==0){
                lambda = (3*px*px+a)*inv(2*py%p)%p;
                x3 = ((lambda * lambda-2*px)%p);
                y3 = ((lambda*(px-x3)-py)%p);


        }
        //inverses
        else if((qx-px)%p == 0){
                return infty;
        }
        p3.push_back( (x3+p)%p );
        p3.push_back((y3+p)%p );

        return p3;
}


//multiplies the point p by l
vector<mpz_class> lp(mpz_class l, vector<mpz_class> p){
        vector<mpz_class> ret,p2;
        ret.reserve(2);
```

```
        p2.reserve(2);
        p2.push_back(p[0]);                                                    80
        p2.push_back(p[1]);
        mpz_class place,zero("0",10);
        place = 1;
        while((l&place) == zero){
                place = place<<1;


                p2 = Eadd(p2,p2);
        }
        ret = p2;
        while(place<=l){                                                       90
                place = place<<1;
                p2 = Eadd(p2,p2);
                if((place&l)!=zero){
                        ret = Eadd(ret,p2);
                }
        }
        return ret;
}
```

prho.h

```
#include"pElliptic.h"

vector<mpz_class> P,Q;

//P is the base point, Q is the point for which we compute the logarithm

mpz_class k=28;

// the order of the point P in the Elliptic curve group;


// iteration function for pollard's rho

vector<mpz_class> xi_step(vector<mpz_class> x){

        mpz_class xx =(x[0]+p)%p ,a = x[2],b = x[3];

        vector<mpz_class> temp;

        temp.reserve(2);

        temp.push_back(x[0]);

        temp.push_back(x[1]);


        if((xx<=p/3)){

                temp = Eadd(temp,Q);

                a= a+1;

        }

        else if((xx<=2*p/3)){

                temp = Eadd(temp,temp);

                a=2*a%k;

                b=2*b%k;
```

```
        }
        else{
                temp = Eadd(temp,P);
                b=b+1;
        }
        vector<mpz_class> ret;
        ret.reserve(4);
        ret.push_back(temp[0]);
        ret.push_back(temp[1]);
        ret.push_back(a%k);
        ret.push_back(b%k);
        return ret;

}
```

main C++ file

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"mpi.h"
#include"prho.h"
```

```
#include<gmpxx.h>
#include<gmp.h>
#include<time.h>
#include<fstream>

//structure for binary tree
struct tree_node {
        vector<mpz_class> point;
        tree_node *left;
        tree_node *right;

        tree_node(vector<mpz_class> pt){//Constructor
                point=pt;
                left =NULL;
                right = NULL;
        }
};

//base input is read in
int base = 10;
int t; // number of leading zeroes to be distinguished
int p_size;
```

```
tree_node *root; // root of the binary tree
int done = 0;//tracks whether or not solution is found
int killed = 0; // how many slaves have been finalized
mpz_class noPoints=0; // number of times a slave returns without a distinguished point
mpz_class num_points=0;// number of distinguished points
ofstream file;// output file
mpz_class counter; // counts the number of elliptic curve operations necessary;
mpz_class total_operations=0; // number of operations used in finding the logarithm


//sends the vector p to dest
void sendPoint(vector<mpz_class> p, int dest){
        int size;
        string x,y,a,b;
        x=p[0].get_str();
        y=p[1].get_str();
        a=p[2].get_str();
        b=p[3].get_str();


        size = x.length();
        char x_send[size+1];
        strcpy(x_send,x.c_str());
        MPI_Send(&size,1,MPI_INT,dest,0,MPI_COMM_WORLD);
```

```
        MPI_Send(x_send,size+1,MPI_CHAR,dest,0,MPI_COMM_WORLD);

        size = y.length();
        char y_send[size+1];
        strcpy(y_send,y.c_str());
        MPI_Send(&size,1,MPI_INT,dest,0,MPI_COMM_WORLD);
        MPI_Send(y_send,size+1,MPI_CHAR,dest,0,MPI_COMM_WORLD);

        size = a.length();
        char a_send[size+1];
        strcpy(a_send,a.c_str());
        MPI_Send(&size,1,MPI_INT,dest,0,MPI_COMM_WORLD);
        MPI_Send(a_send,size+1,MPI_CHAR,dest,0,MPI_COMM_WORLD);

        size = b.length();
        char b_send[size+1];
        strcpy(b_send,b.c_str());
        MPI_Send(&size,1,MPI_INT,dest,0,MPI_COMM_WORLD);
        MPI_Send(b_send,size+1,MPI_CHAR,dest,0,MPI_COMM_WORLD);


}
```

```
//prints the point x
void print_pt(vector<mpz_class> x){
        cout<< (x[0]) <<" "<< (x[1]) << "\n";
}

// uses a collision to solve the logarithm
mpz_class solve(vector<mpz_class> x,vector<mpz_class> y){
        cout <<x[0]<<" " << x[1]<< " "<<x[2]<<" "<<x[3]<<"\n";
        cout <<y[0]<<" " << y[1]<< " "<<y[2]<<" "<<y[3]<<"\n";
        mpz_class l = (y[3]-x[3])*invm(x[2]-y[2],k)%k;
        cout<<"collision found\n"<< "the solution is"<< (l+k)%k <<"\n";
        cout << "a total of " << total_operations << " operations were used.\n";
        file<< p<<" "<<k<<" "<<t<<" "<<total_operations<<" "<<noPoints<<" "<<num_points<<"\n"
        done = 1;
        return (l+p)%p;
}

int  comparePoints(vector<mpz_class> x, vector<mpz_class> y){
        //returns 0 if x and y are equal
        //        1 if x>y lexicographically
        //        -1 if x<y lexicographically
```

```
        if(x[0]>=y[0]){
                if(x[0]!=y[0]) return 1;
                else if(x[1] == y[1]) return 0; // x==y
                else if(x[1]>y[1]) return 1;//x<y
                else return -1; // must be the case that x<y
        }
        else return -1; // must be the case that x<y                    100
}

//inserts distinguished point into tree
int treeInsert( vector<mpz_class> pt){
    //returns 1 if collision if found
//      print_pt(pt);
        if(root == NULL){
        //tree is empty so creat first item
                root = new tree_node(pt);
                return 0;                                               110
        }
        else{
                tree_node *curNode = root, *parent;
                int cmp;
                while (curNode != NULL){
```

```
            //print_pt(pt);
            cmp = comparePoints(pt,curNode->point);
            if(cmp==0 && curNode->point[2] != pt[2]){
                //found useful collision
                solve(curNode->point,pt);
                return 1;
            }
            else{
                parent = curNode;
                if(cmp<0){
                    curNode = curNode->left;
                }
                if(cmp>0){
                    curNode = curNode->right;
                }
            }
        }
        if(cmp<0) parent->left = new tree_node(pt);
        else if(cmp>0) parent->right = new tree_node(pt);
        return 0;
    }
```

```
}

                                                                    140

//prints the tree in order
void print_tree(tree_node *node){
        if(node ==NULL) return;
        print_tree(node->left);
        print_pt(node->point);
        print_tree(node->right);


}


//determines if the point is distinguished                          150
bool isDistinguished(vector<mpz_class> x){
        if(x[0] < (one<<(p_size - t))){
            // ie if the x coordinate has t leading zeroes
            return 1;
        }
        return 0;
}


int main(int argc, char *argv[])
```

```
{                                                                          160
    int id,master,nprocs,i,j;
    int tag, dest;
    vector<mpz_class> xi;
    mpz_class qx,qy,px,py;
    //start MPI
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&id);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    master = 0;                                                            170
    if(argc!=5){
        cout << "usage: "<< argv[0]<<" [t] [p_size] [inbase] [outfile] \n";
        cout << argc <<"\n";
        cout << argv[0]<<" "<< argv[1]<< " "<<argv[2]<< " "<<argv[3]<<"\n";
        exit(0);
    }
    t=atoi(argv[1]);
    p_size = atoi(argv[2]);
    base = atoi(argv[3]);
    file.open(argv[4],ios::out | ios::app);                                180
```

```
//set the point at infinity.
infty.reserve(2);
infty.push_back(p);
infty.push_back(p);

if(id==master)
{
        int size;
        string p_str,px_str,py_str,qx_str,qy_str,a_str,b_str,k_str;

        //read necessary information and initialize variables.
        cout << "Enter p:\n";
        cin >> p_str;

        cout << "Enter a:\n";
        cin >> a_str;

        cout << "Enter b:\n";
        cin >> b_str;

        cout << "Enter k:\n";
        cin >> k_str;
```

```
p.set_str(p_str.c_str(),base);
a.set_str(a_str.c_str(),base);
b.set_str(b_str.c_str(),base);
k.set_str(k_str.c_str(),base);

cout << "Enter Px:\n";
cin >> px_str;
px.set_str(px_str.c_str(),base);

cout << "Enter Py:\n";
cin >> py_str;
py.set_str(py_str.c_str(),base);

P.reserve(2);
P.push_back(px);
P.push_back(py);

cout << "Enter Qx:\n";
cin >> qx_str;
qx.set_str(qx_str.c_str(),base);
```

```
cout << "Enter Qy:\n";
cin >> qy_str;
qy.set_str(qy_str.c_str(),base);

Q.reserve(2);                                                    230
Q.push_back(qx);
Q.push_back(qy);
print_pt(P);
print_pt(Q);

//broadcast information
size = p_str.length();
char p_send[size+1];
strcpy(p_send,p_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);                240
MPI_Bcast(p_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

size = a_str.length();
char a_send[size+1];
strcpy(a_send,a_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(a_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);
```

```
size = b_str.length();
char b_send[size+1];
strcpy(b_send,b_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(b_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


size = k_str.length();
char k_send[size+1];
strcpy(k_send,k_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(k_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


size = px_str.length();
char px_send[size+1];
strcpy(px_send,px_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(px_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


size = py_str.length();
char py_send[size+1];
strcpy(py_send,py_str.c_str());
```

```
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(py_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

size = qx_str.length();
char qx_send[size+1];
strcpy(qx_send,qx_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(qx_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

size = qy_str.length();
char qy_send[size+1];
strcpy(qy_send,qy_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(qy_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

/* Begin handing out work to slaves */
time_t seed = time(NULL);
srand(seed);
cout << "Using seed: " << seed <<"\n";

int r1,r2;
for(i=1;i<nprocs;i++){
```

```
        //give slaves their starting points
        r1 = rand();
        r2 = rand();
        MPI_Send(&r1,1,MPI_INT,i,i,MPI_COMM_WORLD);
        MPI_Send(&r2,1,MPI_INT,i,i,MPI_COMM_WORLD);
}
int isDist=0;
int source;
                                                                    300
vector<mpz_class> pRecv;
pRecv.resize(4);
mpz_class xr,yr,ar,br;//will recieved from slaves

while(!done || killed<nprocs-1){

        //wait for something to do
        MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);

        MPI_Recv(&isDist,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);    310
        if(isDist==2){
                //slave was checking if we are done
                MPI_Send(&done,1,MPI_INT,source,0,MPI_COMM_WORLD);
```

```
            if(done) killed++;
            continue;
        }

        //receive number of operations done by slave
        MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
        char countRecv[size+1];
        MPI_Recv(countRecv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);
        counter.set_str(countRecv,10);
        total_operations+=counter;

        if(isDist){
            //Source will send a distinguished point
            num_points++;

            MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
            char x_recv[size+1];
            MPI_Recv(x_recv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);

            MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
            char y_recv[size+1];
            MPI_Recv(y_recv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);
```

```
MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
char a_recv[size+1];
MPI_Recv(a_recv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);
                                                                    340
MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
char b_recv[size+1];
MPI_Recv(b_recv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);


xr.set_str(x_recv,10);
yr.set_str(y_recv,10);
ar.set_str(a_recv,10);
br.set_str(b_recv,10);


pRecv[0] = xr;                                                      350
pRecv[1] = yr;
pRecv[2] = ar;
pRecv[3] = br;



if(!done){
        cout <<pRecv[0]<<" " << pRecv[1]<< " "<<pRecv[2]<<" "<<pRecv[
```

```
                    treeInsert(pRecv);
                }
            }
            else{
                noPoints++;
            }

            // send slave more work
            r1 = rand();
            r2 = rand();

            MPI_Send(&r1,1,MPI_INT,source,0,MPI_COMM_WORLD);
            MPI_Send(&r2,1,MPI_INT,source,0,MPI_COMM_WORLD);
            MPI_Send(&done,1,MPI_INT,source,0,MPI_COMM_WORLD);

            if(done){
                killed++;
            }


        }
    }
```

```
                                                                    380
/* Slave part */
else
{
        int size;

        //receive information
        MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
        char p_str[size+1];
        MPI_Bcast(p_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);
                                                                    390
        MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
        char a_str[size+1];
        MPI_Bcast(a_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

        MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
        char b_str[size+1];
        MPI_Bcast(b_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

        MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
        char k_str[size+1];                                         400
        MPI_Bcast(k_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);
```

```
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char px_str[size+1];
MPI_Bcast(px_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char py_str[size+1];
MPI_Bcast(py_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char qx_str[size+1];
MPI_Bcast(qx_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char qy_str[size+1];
MPI_Bcast(qy_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

//make use of the recieved info
p.set_str(p_str,base);
a.set_str(a_str,base);
b.set_str(b_str,base);
k.set_str(k_str,base);
```

```
px.set_str(px_str,base);
py.set_str(py_str,base);
P.reserve(2);
P.push_back(px);
P.push_back(py);
qx.set_str(qx_str,base);
qy.set_str(qy_str,base);
Q.reserve(2);
Q.push_back(qx);
Q.push_back(qy);



//set the point at infinity.
infty.reserve(2);
infty.push_back(p);
infty.push_back(p);


//receive work
int a0,b0;
MPI_Recv(&a0,1,MPI_INT,0,id,MPI_COMM_WORLD,&status);
MPI_Recv(&b0,1,MPI_INT,0,id,MPI_COMM_WORLD,&status);
vector<mpz_class> x0 = Eadd(lp(a0,Q),lp(b0,P));
```

```
        xi.resize(4);
        xi[0] = x0[0];
        xi[1] = x0[1];
        xi[2] = a0%k;
        xi[3] = b0%k;                                                    450


//
        //Start work
        int isWork = 1;
        int isDist=0;
        int checkdone=2;
        int isMessage,message;
        mpz_class maxTrailLength = 20*one<<t; // limit to the length of the trail as suggested in one of
        while(isWork){
            counter =0;                                                  460
            mpz_class curTrailLength = 0;
            isDist = 0;
            for(curTrailLength = 0; !done && curTrailLength<maxTrailLength && !isDistinguished(x
                xi = xi_step(xi);
                counter++;
                if(curTrailLength%10000000==0){
                    //check if a solution has been found
```

```
                MPI_Send(&id,1,MPI_INT,master,0,MPI_COMM_WORLD);
                MPI_Send(&checkdone,1,MPI_INT,master,0,MPI_COMM_WORLD);
                MPI_Recv(&done,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        }
}
if(done){
        isWork=0;
        break;
}

if(isDistinguished(xi)) isDist = 1;

MPI_Send(&id,1,MPI_INT,master,0,MPI_COMM_WORLD);
MPI_Send(&isDist,1,MPI_INT,master,0,MPI_COMM_WORLD);

//send number of operations used to master
string x = counter.get_str();
size = x.length();
char x_send[size+1];
strcpy(x_send,x.c_str());
MPI_Send(&size,1,MPI_INT,master,0,MPI_COMM_WORLD);
MPI_Send(x_send,size+1,MPI_CHAR,master,0,MPI_COMM_WORLD);
```

```
                                                                    490
        if(isDist){
            //send distinguished point to master
            sendPoint(xi,master);
        }

        //receive new starting point
        MPI_Recv(&a0,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        MPI_Recv(&b0,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        MPI_Recv(&done,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        if(done){                                                   500
            isWork = 0;
            break;
        }

        x0 = Eadd(lp(a0,Q),lp(b0,P));
        xi[0] = x0[0];
        xi[1] = x0[1];
        xi[2] = a0%k;
        xi[3] = b0%k;
//      }                                                           510
    }
```

```
        }
        MPI_Finalize();
        return(0);
}
```

Appendix B

The source code for the program used to solve the ECDLP over $\mathbb{Z}_{2^m}$.

gmpconvgf2e.h

```
#include <NTL/GF2E.h>

#include <gmpxx.h>

#include <gmp.h>



/*Converts the polynomial p over GF2
 *to an integer representation
 */
mpz_class gf2e_to_mpz(GF2E f){
        GF2X g=conv<GF2X>(f);
        mpz_class ret_val = 0,place = 1;
        int lim = deg(g),i;
```

```
        for(i=0;i<=lim;i++){
                if(IsOne(coeff(g,i))){
                        ret_val+=place;
                }
                place = place<<1;
        }
        return ret_val;
}

//converts an integer to a field element
GF2E mpz_to_GF2E(mpz_class x){
        GF2X f;
        mpz_class place = 1;
        int i=0;
        while(place<=x){
                if((place&x) == place){
                        SetCoeff(f,i,1);

                }
                i++;
                place = place<<1;
        }
```

```
        return conv<GF2E>(f);


}


//converts integer to polynomial
GF2X mpz_to_GF2X(mpz_class x){
        GF2X f;
        mpz_class place = 1;
        int i=0;
        while(place<=x){
                if((place&x) == place){
                        SetCoeff(f,i,1);
                }
                i++;
                place = place<<1;
        }
        return f;


}
```

pElliptic.h

```
//#include<2mFieldv2.h>
```

```
#include<gmpxx.h>

#include<gmp.h>

#include<NTL/GF2E.h>

#include<vector>

using namespace std;

using namespace NTL;


GF2E a,b;// parameters a,b which define the curve

GF2X p;//modulus of field

int m;//degree of modulus

mpz_class one(1);//1 as an mpz number

vector<GF2E> infty;//will represent the group identity



//Adds two points

vector<GF2E> Eadd(vector<GF2E> p1,vector<GF2E> p2){

        vector<GF2E> retval;

        retval.reserve(2);

        GF2E lambda;

        GF2E x1 = p1[0];

        GF2E y1 = p1[1];

        GF2E x2 = p2[0];
```

```
        GF2E y2 = p2[1];
        if(p1 == infty) return p2;
        else if(p2 == infty) return p1;
        else if(p1[0]!=p2[0]){
                lambda = (y1+y2)/(x1+x2);
                retval.push_back(lambda*lambda + lambda + x1 + x2 + a);
                retval.push_back(lambda*(x1+retval[0])+retval[0]+y1);
        }
        else if(p1[0] == p2[0] && p1[1]==p2[1]&& p2[0]!=0){
                //doubling a point
                lambda = x1+y1/x1;
                retval.push_back(lambda*lambda + lambda + a);
                retval.push_back(x1*x1 + (lambda + 1)*retval[0]);
        }
        else if(p1[0] == p2[0] && p2[1] == (p1[1]+p1[0])){
                //inverses
                return infty;
        }

        return retval;
}
```

```
//multiplies a point, p, by l
vector<GF2E> lp(mpz_class l, vector<GF2E> p){
        vector<GF2E> ret,p2;
        ret.reserve(2);
        p2.reserve(2);
        p2.push_back(p[0]);
        p2.push_back(p[1]);
        mpz_class place,zero("0",10);
        place = 1;
        while((l&place)==0){
                place = place<<1;

                p2 = Eadd(p2,p2);
        }
        ret = p2;
        while(place<=l){
                place = place<<1;
                p2 = Eadd(p2,p2);
                if((place&l)!=zero){
                        ret = Eadd(ret,p2);
                }
        }
```

```
        return ret;
}
```

prho.h

```
#include"2mElliptic.h"
#include"gmpconvgf2e.h"
vector<GF2E> P,Q;//P is the base point, Q is the point for which we compute the logarithm
mpz_class k;// the order of the point P in the Elliptic curve group;


//computes the inverse of n modulo m
mpz_class invm(mpz_class n,mpz_class m){
        mpz_t g,s,t;
        mpz_init(g);
        mpz_init(s);
        mpz_init(t);
        mpz_gcdext(g,s,t,n.get_mpz_t(),m.get_mpz_t());
        mpz_class ret(s);
        mpz_clear(g);
        mpz_clear(s);
        mpz_clear(t);
        return ret;
```

```
}


//iteration function for pollard's rho
vector<GF2E> xi_step(vector<GF2E> x){
        mpz_class xx =gf2e_to_mpz(x[0]);
        vector<GF2E> temp;
        temp.reserve(2);
        temp.push_back(x[0]);
        temp.push_back(x[1]);

        if((xx<=(one<<m)/3)){
                temp = Eadd(temp,Q);
        }
        else if((xx<=2*(one<<m)/3)){
                temp = Eadd(temp,temp);
        }
        else{
                temp = Eadd(temp,P);
        }
        return temp;

}
```

```
                                                                                    40

//iteration function for ai and bi

vector<mpz_class> ab_update(vector<GF2E> x,vector<mpz_class> ab){
        mpz_class xx =gf2e_to_mpz(x[0]);
        vector<mpz_class> temp=ab;


        if((xx<=(one<<m)/3)){
                temp[0]=(temp[0]+1)%k;
        }
        else if((xx<=2*(one<<m)/3)){
                temp[0]=(temp[0]*2)%k;                                              50
                temp[1]=(temp[1]*2)%k;
        }
        else{
                temp[1]=(temp[1]+1)%k;
        }
        return temp;
}
```

main C++ file

```
#include<stdio.h>
```

```
#include<stdlib.h>
#include<string.h>
#include"mpi.h"
#include"prho.h"
#include<gmpxx.h>
#include<gmp.h>
#include<time.h>
#include<fstream>

//structure for binary tree
struct tree_node {
        vector<GF2E> point;
        vector<mpz_class> ab;
        tree_node *left;
        tree_node *right;

        tree_node(vector<GF2E> pt,vector<mpz_class> AB){//Constructor
                point=pt;
                ab=AB;
                left =NULL;
                right = NULL;
        }
```

```
};


int base = 10; //base input will be read in
int t=1; // number of leading zeroes to be distinguished
tree_node *root;// root of binary tree
GF2E zero(0);//zero as a field element
int done = 0;//has a solution been found
ofstream file;//output file
int killed = 0;//how many slaves have been finalized
mpz_class noPoints=0; // number of times a slave returns without a distinguished point
mpz_class num_points=0;//number of distinguished points
mpz_class counter; // counts the number of elliptic curve operations necessary;
mpz_class total_operations=0; // number of operations used in finding the logarithm


void print_pt(vector<GF2E> x){
        cout<< (x[0]) <<" "<< (x[1]) << "\n";
}

//sends p, ab to dest
void sendPoint(vector<GF2E> p,vector<mpz_class> ab, int dest)
```

```
{
        int size;
        mpz_class xmpz,ympz;
        string x,y,a,b;
        xmpz=gf2e_to_mpz(p[0]);
        ympz=gf2e_to_mpz(p[1]);
        x=xmpz.get_str();
        y=ympz.get_str();
        a=ab[0].get_str();
        b=ab[1].get_str();

        size = x.length();
        char x_send[size+1];
        strcpy(x_send,x.c_str());
        MPI_Send(&size,1,MPI_INT,dest,0,MPI_COMM_WORLD);
        MPI_Send(x_send,size+1,MPI_CHAR,dest,0,MPI_COMM_WORLD);

        size = y.length();
        char y_send[size+1];
        strcpy(y_send,y.c_str());
        MPI_Send(&size,1,MPI_INT,dest,0,MPI_COMM_WORLD);
        MPI_Send(y_send,size+1,MPI_CHAR,dest,0,MPI_COMM_WORLD);
```

```
        size = a.length();
        char a_send[size+1];                                                    70
        strcpy(a_send,a.c_str());
        MPI_Send(&size,1,MPI_INT,dest,0,MPI_COMM_WORLD);
        MPI_Send(a_send,size+1,MPI_CHAR,dest,0,MPI_COMM_WORLD);


        size = b.length();
        char b_send[size+1];
        strcpy(b_send,b.c_str());
        MPI_Send(&size,1,MPI_INT,dest,0,MPI_COMM_WORLD);
        MPI_Send(b_send,size+1,MPI_CHAR,dest,0,MPI_COMM_WORLD);
                                                                                80

}


// uses a collision to solve the logarithm
mpz_class solve(vector<mpz_class> x,vector<mpz_class> y){
        cout <<x[0]<<" " << x[1]<<"\n";
        cout <<y[0]<<" " << y[1]<<"\n";
        mpz_class l = (y[1]-x[1])*invm(x[0]-y[0],k)%k;
        cout <<k<<"\n";
```

```
        cout<<"collision found\n"<< "the solution is"<< (l+k)%k <<"\n";
        cout << "a total of " << total_operations << " operations were used.\n";
        file<< p<<" "<<k<<" "<<t<<" "<<total_operations<<" "<<noPoints<<" "<<num_points<<"\n"
        done = 1;
        return 1;
}

int comparePoints(vector<GF2E> x, vector<GF2E> y){
        //returns 0 if x and y are equal
        //        1 if x>y lexicographically
        //        -1 if x<y lexicographically
        mpz_class x1,x2,y1,y2;
        x1=gf2e_to_mpz(x[0]);
        x2=gf2e_to_mpz(x[1]);
        y1=gf2e_to_mpz(y[0]);
        y2=gf2e_to_mpz(y[1]);
        if(x1>=y1){
                if(x1!=y1) return 1;
                else if(x2 == y2) return 0; // x==y
                else if(x2>y2) return 1;//x<y
                else return -1; // must be the case that x<y
        }
```

```
        else return −1; // must be the case that x<y
}


int treeInsert( vector<GF2E> pt,vector<mpz_class> ab){
        //returns 1 if collision if found
        if(root == NULL){
        //tree is empty so creat first item
                root = new tree_node(pt,ab);                                    120
                return 0;
        }
        else{
                tree_node *curNode = root, *parent;
                int cmp;
                while (curNode != NULL){
                        cmp = comparePoints(pt,curNode->point);
                        if(cmp==0 && curNode->ab[0] != ab[0]){
                                //found useful collision
                                solve(curNode->ab,ab);                          130
                                return 1;
                        }
                        else{
```

```
                        parent = curNode;
                        if(cmp<0){
                                curNode = curNode->left;
                        }
                        if(cmp>0){
                                curNode = curNode->right;
                        }
                }
        }
        if(cmp<0) parent->left = new tree_node(pt,ab);
        else if(cmp>0) parent->right = new tree_node(pt,ab);
        return 0;
    }


}

void print_tree(tree_node *node){
    if(node ==NULL) return;
    print_tree(node->left);
    print_pt(node->point);
    print_tree(node->right);
```

```
}

bool isDistinguished(vector<GF2E> x){
        GF2X g;                                                          160
        g=conv<GF2X>(x[0]);

        if(deg(g) < (m - t)){
            // ie if the x coordinate has t leading zeroes
            return 1;
        }
        return 0;
}

int main(int argc, char *argv[])                                         170
{
        int id,master,nprocs,i,j;
        int tag, dest;
        vector<GF2E> xi;
        vector<mpz_class> ab;
        mpz_class qxmpz,qympz,pxmpz,pympz; // input will be in mpz form, later converted to GF2E
        GF2E qx,qy,px,py;
```

```
//start MPI
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&id);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
master = 0;
if(argc!=5){
        cout << "usage: "<< argv[0]<<" [t] [p_size] [inbase] [outfile] \n";
        cout << argc <<"\n";
        cout << argv[0]<<" "<< argv[1]<< " "<<argv[2]<< " "<<argv[3]<<"\n";
        exit(0);
}
t=atoi(argv[1]);
m = atoi(argv[2]);
base=atoi(argv[3]);
file.open(argv[4],ios::out | ios::app);
infty.reserve(2);
infty.push_back(zero);
infty.push_back(zero);

if(id==master)
```

```
{                                                                    200
    int size;
    string p_str,px_str,py_str,qx_str,qy_str,a_str,b_str,k_str;
    mpz_class p_mpz,a_mpz,b_mpz;
    //read necessary information and initialize variables.
    cout << "Enter p:\n";
    cin >> p_str;
    p_mpz.set_str(p_str.c_str(),base);
    p=mpz_to_GF2X(p_mpz);
    GF2E::init(p);
    cout<< p;                                                        210
    cout << "Enter a:\n";
    cin >> a_str;
    cout << "Enter b:\n";
    cin >> b_str;
    a_mpz.set_str(a_str.c_str(),base);
    a=mpz_to_GF2E(a_mpz);
    b_mpz.set_str(b_str.c_str(),base);
    b=mpz_to_GF2E(b_mpz);
    cout << "Enter k:\n";
    cin >> k_str;                                                    220
    k.set_str(k_str.c_str(),base);
```

```
cout << "Enter Px:\n";
cin >> px_str;
pxmpz.set_str(px_str.c_str(),base);
cout << "Enter Py:\n";
cin >> py_str;
pympz.set_str(py_str.c_str(),base);
cout << "before conv\n";
px=mpz_to_GF2E(pxmpz);
cout << "during conv\n";
py=mpz_to_GF2E(pympz);
cout << "after conv\n";
P.reserve(2);
P.push_back(px);
P.push_back(py);
cout << "Enter Qx:\n";
cin >> qx_str;
qxmpz.set_str(qx_str.c_str(),base);
cout << "Enter Qy:\n";
cin >> qy_str;
qympz.set_str(qy_str.c_str(),base);
qx=mpz_to_GF2E(qxmpz);
qy=mpz_to_GF2E(qympz);
```

```
Q.reserve(2);

Q.push_back(qx);

Q.push_back(qy);


//broadcast information

size = p_str.length();

char p_send[size+1];

strcpy(p_send,p_str.c_str());

MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);

MPI_Bcast(p_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


size = a_str.length();

char a_send[size+1];

strcpy(a_send,a_str.c_str());

MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);

MPI_Bcast(a_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


size = b_str.length();

char b_send[size+1];

strcpy(b_send,b_str.c_str());

MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);

MPI_Bcast(b_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);
```

```
size = k_str.length();
char k_send[size+1];
strcpy(k_send,k_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(k_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

size = px_str.length();
char px_send[size+1];
strcpy(px_send,px_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(px_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

size = py_str.length();
char py_send[size+1];
strcpy(py_send,py_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(py_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

size = qx_str.length();
char qx_send[size+1];
strcpy(qx_send,qx_str.c_str());
```

```
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(qx_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);

size = qy_str.length();
char qy_send[size+1];
strcpy(qy_send,qy_str.c_str());
MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
MPI_Bcast(qy_send,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


/* Begin handing out jobs */
time_t seed = time(NULL);
srand(seed);
cout << "Using seed: " << seed <<"\n";
int r1,r2;
for(i=1;i<nprocs;i++){
        r1 = rand();
        r2 = rand();
        MPI_Send(&r1,1,MPI_INT,i,i,MPI_COMM_WORLD);
        MPI_Send(&r2,1,MPI_INT,i,i,MPI_COMM_WORLD);
}
int isDist=0;
int source;
```

```
vector<GF2E> xRecv;
vector<mpz_class> abRecv;
xRecv.resize(2);
abRecv.resize(2);
GF2E xr,yr;
mpz_class xrmpz,yrmpz,ar,br;//will recieved from slaves
while(!done || killed<nprocs-1){
        MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);

        MPI_Recv(&isDist,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);

        if(isDist==2){
                MPI_Send(&done,1,MPI_INT,source,0,MPI_COMM_WORLD);
                if(done) killed++;
                continue;
        }

        MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
        char countRecv[size+1];
        MPI_Recv(countRecv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);
        counter.set_str(countRecv,10);
        total_operations+=counter;
```

```
if(isDist){//Source will send a distinguished pointi
        num_points++;
        MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
        char x_recv[size+1];
        MPI_Recv(x_recv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);

        MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
        char y_recv[size+1];
        MPI_Recv(y_recv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);

        MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
        char a_recv[size+1];
        MPI_Recv(a_recv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);

        MPI_Recv(&size,1,MPI_INT,source,0,MPI_COMM_WORLD,&status);
        char b_recv[size+1];
        MPI_Recv(b_recv,size+1,MPI_CHAR,source,0,MPI_COMM_WORLD,&status);

        xrmpz.set_str(x_recv,10);
        yrmpz.set_str(y_recv,10);
        ar.set_str(a_recv,10);
        br.set_str(b_recv,10);
```

```
            xr=mpz_to_GF2E(xrmpz);
            yr=mpz_to_GF2E(yrmpz);

            xRecv[0] = xr;
            xRecv[1] = yr;
            abRecv[0] = ar;
            abRecv[1] = br;                                                    360

            if(!done){
                cout <<xRecv[0]<<" " << xRecv[1]<< " "<<abRecv[0]<<" "<<abRec
                treeInsert(xRecv,abRecv);
            }
        }
        else noPoints++;
        //Give slave more work

        r1 = rand();                                                           370
        r2 = rand();
        MPI_Send(&r1,1,MPI_INT,source,0,MPI_COMM_WORLD);
        MPI_Send(&r2,1,MPI_INT,source,0,MPI_COMM_WORLD);
        MPI_Send(&done,1,MPI_INT,source,0,MPI_COMM_WORLD);
```

```
                if(done){
                        killed++;
                }


        }
}
/* Slave part */
else
{
        int size;
        /*Receive info*/
        MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
        char p_str[size+1];
        MPI_Bcast(p_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


        MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
        char a_str[size+1];
        MPI_Bcast(a_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


        MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
        char b_str[size+1];
```

```
MPI_Bcast(b_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char k_str[size+1];
MPI_Bcast(k_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char px_str[size+1];
MPI_Bcast(px_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char py_str[size+1];
MPI_Bcast(py_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char qx_str[size+1];
MPI_Bcast(qx_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);


MPI_Bcast(&size,1,MPI_INT,master,MPI_COMM_WORLD);
char qy_str[size+1];
MPI_Bcast(qy_str,size+1,MPI_CHAR,master,MPI_COMM_WORLD);
```

```
//make use of the recieved info
mpz_class p_mpz(p_str,base);
mpz_class a_mpz,b_mpz,px_mpz,py_mpz,qx_mpz,qy_mpz;
p=mpz_to_GF2X(p_mpz);
GF2E::init(p);
//set the point at infinity.
infty.reserve(2);
infty.push_back(zero);
infty.push_back(zero);
a_mpz.set_str(a_str,base);
b_mpz.set_str(b_str,base);
k.set_str(k_str,base);
px_mpz.set_str(px_str,base);
py_mpz.set_str(py_str,base);
px=mpz_to_GF2E(px_mpz);
py=mpz_to_GF2E(py_mpz);
P.reserve(2);
P.push_back(px);
P.push_back(py);
qx_mpz.set_str(qx_str,base);
qy_mpz.set_str(qy_str,base);
a=mpz_to_GF2E(a_mpz);
```

```
b=mpz_to_GF2E(b_mpz);

qx=mpz_to_GF2E(qx_mpz);

qy=mpz_to_GF2E(qy_mpz);

Q.reserve(2);

Q.push_back(qx);

Q.push_back(qy);


int a0,b0;

MPI_Recv(&a0,1,MPI_INT,0,id,MPI_COMM_WORLD,&status);                    450

MPI_Recv(&b0,1,MPI_INT,0,id,MPI_COMM_WORLD,&status);

vector<GF2E> x0 = Eadd(lp(a0,Q),lp(b0,P));

vector<mpz_class> ab;


ab.resize(2);

xi.resize(2);

xi[0] = x0[0];

xi[1] = x0[1];

ab[0] = a0%k;

ab[1] = b0%k;                                                           460


//Start work

int isWork = 1;
```

```
int isDist=0;
int checkdone=2;
int isMessage,message;
mpz_class maxTrailLength = 20*one<<t;


while(isWork){
        counter =0;                                                                    470
        mpz_class curTrailLength = 0;
        isDist = 0;
        for(curTrailLength = 1;!done&& curTrailLength<maxTrailLength && !isDistinguished(xi);
                ab=ab_update(xi,ab);
                xi = xi_step(xi);
                counter++;
                if(curTrailLength%1000000==0){
                        MPI_Send(&id,1,MPI_INT,master,0,MPI_COMM_WORLD);
                        MPI_Send(&checkdone,1,MPI_INT,master,0,MPI_COMM_WORLD);
                        fflush(stdout);                                                480
                        MPI_Recv(&done,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
                        fflush(stdout);
                }
        }
        if(done){
```

```
        isWork=0;
        break;
    }
    if(isDistinguished(xi)) isDist = 1;
                                                                    490
    MPI_Send(&id,1,MPI_INT,master,0,MPI_COMM_WORLD);
    MPI_Send(&isDist,1,MPI_INT,master,0,MPI_COMM_WORLD);

    //send number of operations used to master
    string x = counter.get_str();
    size = x.length();
    char x_send[size+1];
    strcpy(x_send,x.c_str());
    MPI_Send(&size,1,MPI_INT,master,0,MPI_COMM_WORLD);
    MPI_Send(x_send,size+1,MPI_CHAR,master,0,MPI_COMM_WORLD);       500
    if(isDist){
        sendPoint(xi,ab,master);
    }
    MPI_Recv(&a0,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    MPI_Recv(&b0,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    MPI_Recv(&done,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    if(done){
```

```
                    isWork = 0;
                    cout << "slave " << id << "dead\n";
                    break;
                }
                x0 = Eadd(lp(a0,Q),lp(b0,P));

                xi[0] = x0[0];
                xi[1] = x0[1];
                ab[0] = a0%k;
                ab[1] = b0%k;
            }
        }
        MPI_Finalize();
        return(0);
}
```