

# Blinded Distributed Computing

## 6.857 Final Project

Cristian Cadar  
cristic@mit.edu

Cătălin Frâncu  
cata@mit.edu

Ovidiu Gheorghioiu  
ovy@mit.edu

December 4, 2001

### **Abstract**

With over one billion Internet users, harnessing the idle time of computers around the world has become a powerful idea. Recently, many distributed computing systems have been launched, covering various research topics: searching for intelligent life in space, developing drugs to fight AIDS and other diseases, finding prime numbers, searching for craters on Mars etc. In certain cases, however, volunteers involved in distributed computing systems should not be able to read the information they process; we call this blinded distributed computing and we try to find general techniques for blinding basic cryptographic problems. We focus our attention on the discrete logarithm problem, for which we also build a toy version system.



# Contents

<b>1</b>	<b>Overview</b>	<b>5</b>
<b>2</b>	<b>Blinding and distributing the DLP</b>	<b>6</b>
2.1	Exhaustive Search . . . . .	7
2.2	Index-Calculus Algorithm . . . . .	8
2.3	Optimizations and Caveats . . . . .	9
2.4	The Distributed Index-Calculus Algorithm . . . . .	10
2.5	Blinding the Index-Calculus Algorithm . . . . .	11
2.6	Benchmarks . . . . .	13
<b>3</b>	<b>Extensions and Limitations of BDC</b>	<b>13</b>
<b>4</b>	<b>Appendix: Source Code</b>	<b>16</b>
4.1	Client.java . . . . .	16
4.2	Server.java . . . . .	18
4.3	ServerLogic.java . . . . .	20
4.4	ServerIOWorker.java . . . . .	25
4.5	GenPrimePairs.java . . . . .	27



# 1 Overview

The history of distributed computing is closely related to the history of networks. At this moment in time, there are approximately one billion computers connected to the Internet, with an average speed of 500 Mhz each. This impressive computation power lead to the obvious question of how to make use of the unused CPU cycles.

One of the first distributed computing programs available to the general public was launched in 1996 and aimed at finding Mersenne prime numbers. Another successful distributed computing system is **distributed.net** [1], launched in January 1997. Distributed.net has currently thousands of users and is trying to break different encryption schemes and hash functions; its current challenge is to break RC-64. Distributed.net is also concerned with finding optimal Golomb rulers.

The most successful distributed computing system is **SETI@home** [2]. The project, launched in May 1999, searches for radio signal fluctuations that may indicate the presence of intelligent life in space. The project has overcome all expectations by gathering over two millions volunteers. Experts estimate SETI to be twice as fast as the world's fastest supercomputer (The U.S. Government's *ASCI White*), which cost \$110 million dollars to build. Comparatively, SETI@home cost only \$500,000, excluding the cost of the home PCs.

There are many distributed computing projects launched over the last couple of years: NASA's project to search for craters on Mars, Stanford's project to understand the process of protein folding, Entropia and Olson labs project to develop drugs to fight AIDS etc.

All of these projects are purely research projects initiated by research laboratories. The information involved in these projects is not confidential and thus any data sent to project volunteers does not need to be encrypted.

However, imagine that the Department of Defense wants to use a distributed system to crack what they believe to be an encryption of a terrorist message. In this case, the DoD wants to make sure that whoever decrypts the message cannot actually read it. We call this **blinded distributed computing**.

In this paper, we focus our attention on some basic cryptographic problems, which commonly arise in encryption/decryption schemes: taking the discrete logarithm, computing square roots modulo a large composite number, and factoring large numbers

There are several possible requirements for a blinded distributed computing system, depending on the desired level of security:

## A. Blinding

1. The solution to the original problem should not be visible to the volunteers doing the computation, if one of them finds a blinded solution. Only the server should be able to recover the original solution from the blinded one.

2. The volunteers should be unable to correlate the blinded problems they are given with the problem in the clear, or with other blinded instances of the same problem.
3. The volunteers should not be able to tell whether or not they found a solution to the problem, except with negligible probability. This could be accomplished by knowingly introducing a large number of false positives, with only the server being able to distinguish between false positives and the actual solution. Of course, the number of large positives should still be significantly smaller than the search space.

## B. Distributing

- 1 The complexity of the blinded problem should be the same as the complexity of the original problem, independently of the chosen algorithm.
- 2 The algorithm using  $c$  clients should be approximately  $c$  times faster than the algorithm using only 1 client.

In this paper, we present a powerful scheme that solves the discrete logarithm problem (DLP) in a blinded, distributed manner and has all the properties mentioned above. We also provide a toy implementation of the algorithm. Finally, we show the extent to which our concept can be generalized by presenting our results for the integer factorization problem and for the problem of finding square roots modulo a large composite number.

## 2 Blinding and distributing the DLP

We start by defining the facts and terminology involved in the discrete logarithm problem:

**Fermat's Theorem:** If  $p$  is prime, then  $a^{p-1} = 1 \pmod{p}$ , for all  $a \in Z_p^*$ .

**Definition:** Given a prime  $p$  and a number  $a$  modulo  $p$ , the least positive  $x$  such that  $a^x = 1 \pmod{p}$  is called the **order** of  $a$ , modulo  $p$ .

**Definition:** Given a prime  $p$ , then a number  $g$  modulo  $p$  is called a **generator** modulo  $p$  if the order of  $g \pmod{p}$  is equal to  $p - 1$ .

**Theorem:** Given a prime  $p$ , a generator  $g \pmod{p}$  and a value  $a \pmod{p}$ , then there exists a unique  $z \pmod{p}$  such that  $g^z = a \pmod{p}$ . This value  $z$  is called the **discrete logarithm** of  $a$ , modulo  $p$ , to the base  $g$ .

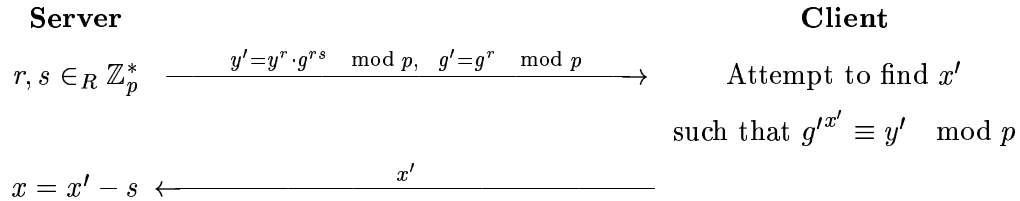
**The Discrete Logarithm Problem:** Given a prime  $p$ , a generator  $g \pmod{p}$  and a value  $y \pmod{p}$ , find  $x$  such that  $y = g^x \pmod{p}$ .

By notation,  $n = p - 1$ , and all the logarithms are in base  $g$ , unless otherwise noted.

## 2.1 Exhaustive Search

Let's try a first approach that searches for  $x$  exhaustively, by computing  $g^1, g^2, g^3, \dots$  until the desired value  $y$  is obtained. A scheme that obeys the required properties follows:

**Public:**  $p$             **Server secret:**  $y, g$             **Unknown:**  $x$  such that  $g^x \equiv y \pmod{p}$ .



**Description:** The Server sends to the Client a pair of indirect values  $y' = y^r \cdot g^{rs}$  and  $g' = g^r$ , thereby hiding  $y$  and  $g$  from the Client. The Client then uses brute force to find  $x'$ , the discrete log of  $y'$  in base  $g'$ . The two logarithms are related by the equation:

$$y' = y^r \cdot g^{rs} \Rightarrow g'^{x'} = g^{rx} g^{rs} \Rightarrow x = x' - s$$

The Server can easily recover the solution for  $(y, g)$  given a solution for  $(y', g')$ . Moreover, the Server can limit the search space for the Client. If the Server wants the Client to search for solutions such that  $x_1 \leq x \leq x_2$ , it will instruct the Client to search for an  $x'$  such that  $x_1 + s \leq x' \leq x_2 + s$ .

**Completeness:** The description above shows that the protocol succeeds if provided with sufficient computation power. The Server is capable of sweeping across the entire interval  $[1, p - 1]$  and distributing sub-intervals to the clients.

**Soundness:** A malicious Client will not be able to send an incorrect solution. The server can always verify that  $x = x' - s$  is indeed the discrete logarithm of  $y$ .

**Zero knowledge:** The triplets  $(g', x', y')$  can easily be generated by the Client. Thus, by cracking  $y'$ , the Client learns nothing about  $g, x, y, r$  or  $s$ .

**Weakness:** This scheme does not satisfy property A3. A malicious Client can tell when it finds a solution, and may even hide the solution (i.e. the Client always pretends the assigned interval does not contain any solution). We can however ignore this problem, since the probability of any particular client finding the solution is negligible.

The soundness and zero-knowledge properties demonstrate that this system meets conditions A1 and A2. The distributing requirements are also satisfied. Obviously, the algorithm works  $c$  times faster if there are  $c$  times more users. Also, the complexity of the blinded problem is the same as the complexity for the original problem. Indeed, note

that we reduce the problem of finding  $\log_g y$  modulo  $p$  to the problem of finding a  $\log_{g'} y'$  mod  $p$ . But the complexity of an exhaustive search is still  $O(p)$ , which is independent of the choice for  $g$  and  $y$ . The required encryption of  $g$  and  $y$  does not slow down the algorithm.

While this scheme is clearly impractical, we consider it illustrative for how blinding works.

## 2.2 Index-Calculus Algorithm

Index-calculus is one of the most powerful known algorithms for solving the discrete logarithm problem. In this section, we describe the original algorithm, we show how to distribute it and blind it, and we briefly discuss the toy implementation we wrote.

The index-calculus algorithm, presented in [1], §3.6.5, works as follows:

1. Generate the first  $t$  prime numbers,  $\{p_1, p_2, \dots, p_t\}$ .
2. Repeat:
  - (a) Pick  $k \in_R \mathbb{Z}_n$ , and compute  $g^k \pmod p$ .
  - (b) Try to factor  $g^k$  using only the first  $t$  prime numbers:

$$g^k = \prod_{i=1}^t p_i^{e_i}$$

- (c) If successful, take logarithms in base  $g$  of both sides and express  $k$  as a linear combination of the logarithms of the first  $t$  prime numbers:

$$k = \sum_{i=1}^t e_i \log p_i$$

3. Find the logarithms of  $p_1, p_2, \dots, p_t$ .
  - (a) Collect  $t + c$  equations using step 2 above, where  $c$  is a constant chosen such that the system of  $t + c$  equations has a unique solution with high probability.
  - (b) Solve the system of  $t + c$  equations modulo  $n$  to find values for the logarithms of the first  $t$  prime numbers.
4. Crack the log.
  - (a) Pick  $k \in_R \mathbb{Z}_n$ , and compute  $y \cdot g^k \pmod p$ .
  - (b) Try to factor  $y \cdot g^k$  using only the first  $t$  prime numbers:

$$y \cdot g^k = \prod_{i=1}^t p_i^{e_i}$$

- (c) If unsuccessful, repeat from 4(a).
- (d) If successful, take logarithms in base  $g$  of both sides:

$$\log y = x = \sum_{i=1}^t e_i \log p_i - k$$

## 2.3 Optimizations and Caveats

**Incremental system solving.** In practice, it turned out that  $c$  is far from a small constant. The systems contain a large number of redundant equations, so  $c$  is even larger than  $t$  in most cases. Therefore, we chose a different approach. We merged steps 2 and 3, and we solved the system using Gaussian elimination. As soon as we generate an equation, we reduce it with all the existing equations. If what is left is an identity of the form  $0 = 0$ , the equation is redundant and we can discard it.

This is saving memory, because now we only need to store  $t$  equations. The necessary space is  $O(t^2)$  as opposed to  $O(t(t+c))$  in the original algorithm. However, this optimization alone is not saving time, because we will still generate all the redundant equations. The redundancy stems from a very simple cause. If, for example,  $t = 2000$ , then  $p_t = 17389$ . To find the log of  $p_t$ , we need to find an equation in  $\log p_t$ , which means we need to find a multiple of  $p_t$ . On the average, we will have to generate 17389 random numbers until one of them will be a multiple of 17389.

**Solving for a subsystem.** To fix this problem, we keep track of the subset of prime numbers that have appeared in the factorizations. If this subset is large enough and completely determined (i.e. there are  $m$  factors that have appeared in the factorizations and we have  $m$  independent equations), we stop and attempt to perform step 4 (cracking the logarithm) in this subset. For large enough  $m$ , finding one more number that factors in this subset is easy (as a proof, we have already found  $m$  such numbers by pure chance). We could, in theory, have chosen a smaller factor base initially, but, since we had no way of guessing what factors we will get before starting to factorize, we are reducing the factor base dynamically. In effect, this avoids the problem pointed above.

**Safe primes.** Because the system is defined modulo  $n$ , which is composite, not all the numbers have inverses modulo  $n$ . Therefore we need to postpone taking inverses until the end of the Gaussian elimination. Then we are left with  $t$  independent equations of the form

$$ax \equiv b \pmod{n}$$

where  $x = \log p_i$  for some small prime  $p_i$ . If  $g$  is a generator, the equation has several solutions, but we still cannot take the inverse of  $a$  modulo  $n$ . The way to solve the equation is to define  $d = \gcd(a, n)$  and to divide  $a$ ,  $b$  and  $n$  by  $d$ :

$$a' = \frac{a}{d} \quad b' = \frac{b}{d} \quad n' = \frac{n}{d}$$

$$a'x' \equiv b' \pmod{n'}$$

Now, because  $\gcd(a', n') = 1$ , we can take the inverse of  $a'$  modulo  $n'$  and write

$$x' = a'^{-1}b' \pmod{n'}$$

The original equation has  $d$  solutions of the form  $x = x' + kn' \pmod{n}$ , for  $k = 0, 1, \dots, d - 1$ . We identify the correct value for  $x$  by checking that  $g^x = p_i$ . A proof of why this works is given in [8].

If  $n$  has many large prime factors, it may happen for one of the  $t$  equations that  $d$  is large, in which case searching for  $x$  may take a long time. Therefore, the implementation requires  $p$  to be a **safe prime**, i.e. a number of the form  $2q + 1$ , where  $q$  is also prime. This ensures that for any  $a$ ,  $\gcd(a, n) \leq 2$  with high probability.

**Efficiency.** Note that our method is only a substitute, dictated by time constraints, for much more efficient methods of collecting equations (e.g., the number field sieve, described in [6]; details on an implementation can be found in [7]). Similarly, we did not attempt to solve the equation system by a complex, efficient algorithm. We used Gaussian elimination, which is known to run in time  $O(t^3)$ . Ideally, we would seek an efficient, and easy to parallelize method to solve the system, which is outside the scope of this paper. However, we believe the methods we used are sufficient for the proof-of-concept implementation we tried to provide.

## 2.4 The Distributed Index-Calculus Algorithm

**Server:**

1. Generate the first  $t$  prime numbers.
2. Distribute  $p$ ,  $g$  and the prime numbers to each incoming client.
3. Start with an empty collection of equations,  $S = \emptyset$ .
4. Repeat until  $|S| = t$ :
  - (a) Wait for an equation  $q$  from the clients;
  - (b) Reduce  $q$  with all the equations in  $S$ ;
  - (c) If  $q$  is the identity  $0 = 0$ , discard  $q$ ;
  - (d) Otherwise  $S = S \cup \{q\}$ , and use  $q$  to reduce all the other equations in  $S$ .
5. Crack the log.
  - (a) Pick  $k \in_R \mathbb{Z}_n$ , and compute  $y \cdot g^k \pmod{p}$ .

(b) Try to factor  $y \cdot g^k$  using only the first  $t$  prime numbers:

$$y \cdot g^k = \prod_{i=1}^t p_i^{e_i}$$

(c) If unsuccessful, repeat from 4(a).

(d) If successful, take logarithms in base  $g$  of both sides:

$$\log y = x = \sum_{i=1}^t e_i \log p_i - k$$

### Clients:

1. Receive  $p$ ,  $g$  and the factor base from server (alternatively, clients can only receive the size of the factor base and generate the base themselves).

2. Send equations to the server until the server closes the connection:

(a) Pick  $k \in_R \mathbb{Z}_n$ , and compute  $g^k \pmod p$ .

(b) Try to factor  $g^k$  using only the first  $t$  prime numbers:

$$g^k = \prod_{i=1}^t p_i^{e_i}$$

(c) If successful, take logarithms in base  $g$  of both sides and express  $k$  as a linear combination of the logarithms of the first  $t$  prime numbers:

$$k = \sum_{i=1}^t e_i \log p_i$$

(d) Send the values  $\{e_1, e_2, \dots, e_t, k\}$  to the server.

## 2.5 Blinding the Index-Calculus Algorithm

The server selects a random  $r$  for each client, such that  $g^r$  is also a generator modulo  $p$ . If  $p$  is a safe prime, than any odd value of  $r$  is likely to work. Instead of sending the generator  $g$  to all the clients, the servers sends  $g^r$  to each client and memorizes the value of  $r$  associated with each client.

Clients then attempt to factor  $g^{rk}$  for random values of  $k$ , and they produce equations of the form:

$$k = \sum_{i=1}^t e_i \log_{g^r} p_i$$

The server must convert these logarithms to base  $g$  so that the equation is compatible with the system. Based on the fact that

$$\log_{g^r} p_i = \frac{\log p_i}{\log g^r} = \frac{\log p_i}{r}$$

the equivalent equation in base  $g$  is

$$kr = \sum_{i=1}^t e_i \log p_i$$

In other words, unblinding is as easy as multiplying  $k$  in each equation received from a client by  $r$ , the random number associated with that client.

Notice there is no need to blind  $y$ , as the server is the only one using  $y$ . Factoring  $y \cdot g^k$  is as easy as producing an equation, so there is no point in distributed this final part of the algorithm.

Following are the proofs that the desired properties hold for this distributed scheme.

**A1:** The client never learns anything about  $y$ . So, because the client has no idea what the problem is, he cannot solve it.

**A2:** The client cannot recover  $g$  because the only information the client holds is  $g^r$ . As long as the database of values of  $r$  remains secret, so does  $g$ . An interesting point to make here is that a client may store the equations it generates and find the logarithms of the numbers in the factor base, assuming it sets up its own network of mischievous clients. However, they can only find the logarithms in base  $g^r$ , but not the logarithms in base  $g$ .

**A3:** This property holds trivially. When the server closes the connection, the clients know that their combined equations have found the logarithms of the number in the factor base. But none of the clients can claim that it has made the vital contribution to the algorithm. The contribution of each client is directly proportional to the CPU power it has donated. In contrast, consider the exhaustive search, where exactly one of the clients must be the “lucky one” to find the actual logarithm (and claim the prize).

**B1:** The only additional operations are simple modular operations:

- Compute  $g^r$  to blind  $g$ . This is done once per client.
- Multiply  $k$  by  $r$  for each incoming equation. This is done once per equation.

Therefore, the time complexity does not increase. The space complexity on the server is still  $O(t^2)$ , while the space complexity on each client is  $O(t)$ .

**B2:** There exists some overhead due to several reasons, but it is negligible:

- The time it takes to manage clients (threads and network flow);
- The time to generate an  $r$  for each client;
- The time to blind  $g$  for each client;
- The time to unblind each incoming equation;

Aside from that, and assuming that all clients have equal CPU power, a system with  $c$  clients works  $c$  times faster than a stand-alone computer.

## 2.6 Benchmarks

- Working modulo an 80-bit number, and with a factor base of 5,000 primes, we computed a discrete log in 16 hours using 3 computers.
- We ran the program on a 128-bit number, with a factor base of 100,000 primes. While the running time and the memory requirements would have prevented the program from completing within the given time-frame, we noticed that our clients were generating roughly one equation in 15 minutes. Therefore, we estimate it would take approximately 30 hours on 1,000 clients to compute logarithms modulo a 128-bit number.

## 3 Extensions and Limitations of BDC

The blinding scheme for the discrete logarithm illustrates a general approach for blinding other computations as well. This scheme transforms the problem of taking the logarithm of  $y$  to the base  $g$  into another instance of the discrete logarithm problem, namely taking the discrete logarithm of  $y'$  to the base  $g'$ . In general, to compute the function  $F(x, y, z, \dots)$  in a blinded manner, one has to find a pair functions: the blinding function  $b$  and the unblinding function  $u$ , such that

$$u(F(b(x), b(y), b(z), \dots)) = F(x, y, z, \dots)$$

For example, taking the square root modulo a composite number  $n$  (whose factorization is unknown) is another computationally hard problem. In this case,

$$F(y) = \sqrt{y} \pmod{n}$$

The server can blind this problem using the functions

$$b(y) = y \cdot r^2, \quad u(x) = x/r$$

We can easily check that

$$u(F(b(y))) = \frac{\sqrt{y \cdot r^2}}{r} = \sqrt{y} = F(y)$$

The most important achievement is that we reduced the problem to another square root problem, and we can apply any existing algorithm, without the clients being able to recover  $y$  from  $b(y)$  or  $x$  from  $F(b(y))$ .

We failed to particularize this scheme for the problem of factoring large integers. That is, we failed to provide  $b$  such that  $b(n)$  is a product of two primes and  $b(n)$  has roughly the size of  $n$ . We believe that such functions may exist, however. Our approach differs from privacy homomorphisms in that privacy homomorphisms work in general and encrypt a small set of simple operations, whereas blinded distributed computing is tailor-made for a specific problem: it encrypts one (possibly complex) operation.

## References

- [1] distributed.net: Node Zero, <http://distributed.net>.
- [2] SETI@home: Search for Extraterrestrial Intelligence, <http://setiathome.ssl.berkeley.edu>.
- [3] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, *Handbook of Applied Cryptography*, 5th ed., CRC Press, 2001. Available online at <http://www.cacr.math.uwaterloo.ca/hac/>.
- [4] Shafi Goldwasser, Mihir Bellare, *Lecture Notes on Cryptography*, <http://www-cse.ucsd.edu/~mihir/papers/gb.ps>.
- [5] Distributed Computing - United Devices, <http://members.ud.com/about/dc/index.htm>.
- [6] A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse, J.M.Pollard, *The Number Field Sieve*, <http://www.std.org/~msm/common/nfspaper.ps>.
- [7] Damian Weber, *Computing Discrete Logarithms with the General Number Field Sieve*, [ftp://ftp.informatik.tu-darmstadt.de/pub/TI/reports/impl\\_nfs\\_dl.ps.gz](ftp://ftp.informatik.tu-darmstadt.de/pub/TI/reports/impl_nfs_dl.ps.gz).
- [8] John A. Beachy, *Abstract Algebra Online Study Guide*, [http://www.math.niu.edu/~beachy/abstract\\_algebra/study\\_guide/13.html](http://www.math.niu.edu/~beachy/abstract_algebra/study_guide/13.html).

## 4 Appendix: Source Code

### 4.1 Client.java

```
import java.math.BigInteger;
import java.util.Random;
import java.io.;
import java.net.;

public class Client {

    static int t;

    static BigInteger prime[], alpha, p, n;

    static final BigInteger ZERO = new BigInteger("0");
    static final BigInteger ONE = new BigInteger("1");
    static final BigInteger TWO = new BigInteger("2");

    static double eav[];

    public static void main(String args[]) throws Exception {
        if (args.length != 2) {
            System.out.println("Usage: java Client <server_address> <server_port>");
            return;
        }

        InetAddress serverAddress = InetAddress.getByAddress(args[0]);

        Socket sock = new Socket(serverAddress, Integer.parseInt(args[1]));

        System.out.println("Socket opened successfully");

        ObjectOutputStream toServer = new ObjectOutputStream(sock.getOutputStream());
        ObjectInputStream fromServer = new ObjectInputStream(sock.getInputStream());

        alpha = (BigInteger) fromServer.readObject();
        p = (BigInteger) fromServer.readObject();
        n = p.subtract(ONE);
        prime = (BigInteger[]) fromServer.readObject();

        t = prime.length - 1;

        System.out.println("Got a factor base of " + t + " numbers");
    }
}
```

```

eav = new double[t+2];

for (int i = t; i>=1; i--) {
    eav[i] = eav[i+1] + 1.0/(Integer.parseInt(prime[i].toString()));
}

System.out.println("largest factor: " + prime[t]);

int eqn = 0;
System.out.println("Sending equations: ");
while (true) {
    BigInteger[] v = collectEquation();
    eqn ++;
    System.out.print(eqn + " ");
    try {
        toServer.writeObject(v);
    } catch (IOException e) {
        System.out.println("\n Got exception: " + e +"; server has probably closed connection.");
        break;
    }
}

private static BigInteger[] collectEquation() {
    Random rnd = new Random();
    BigInteger k, alphak, salphak, div[] = new BigInteger[2];
    BigInteger v[] = new BigInteger[t+2];

    do {
        k = new BigInteger(p.bitLength(), rnd);
    } while (k.compareTo(p) < 0);

    salphak = alpha.modPow(k, p);

    while(true) {
        // Generate a number between 0 and p-1
        k = v[t+1] = k.multiply(TWO).add(ONE).mod(n);
        salphak = salphak.multiply(salphak).multiply(alpha).mod(p);
        if (k.compareTo(p) != -1) continue;

        // Try to factor alpha^k
        alphak = salphak;
        for (int i = 1; i <= t; i++) {
            v[i] = ZERO;

```

50

70

80



```

        System.out.println("Usage: java IndexCalculus alpha beta p");
        return;
    }
    alpha = new BigInteger(args[0]);
    beta = new BigInteger(args[1]);
    p = new BigInteger(args[2]);
    20

    slogic = new ServerLogic(alpha, beta, p);
    slogic.start();

    System.out.println("slogic: " + slogic);
    new ConnectorThread().start();

    slogic.join();
}
30

static class ConnectorThread extends Thread {
    ServerSocket ssock;

    public ConnectorThread() throws Exception {
        ssock = new ServerSocket(SERVER_PORT);
        setDaemon(true);
    }

    public void run() {
        while (true) {
            40
            try {
                Socket sock = ssock.accept();
                System.out.println("Opened socket...");
                System.out.println("slogic: " + slogic);
                new ServerIOWorker(alpha, p,
                    slogic.getFactorBase(),
                    slogic.getBuffer(),
                    sock)
                    .start();
            } catch (Exception e) { System.out.println("Exception in client accept: " + e); } 50
        }
    }
}
}

```

### 4.3 ServerLogic.java

```
import java.math.BigInteger;
import java.util.Random;

public final class ServerLogic extends Thread {

    final static int t = 10000;

    BigInteger alpha, beta, p, n;

    BigInteger[] v = new BigInteger[t+2];
    BigInteger[] prime = new BigInteger[t+1];

    // The matrix that stores the system of equations. The coefficients are
    // stored in columns 1 thru t, and the free terms in column t+1.
    // The matrix is 1-based.
    BigInteger[][] eqn = new BigInteger[t+1][];

    // The discrete logs of the prime elements base alpha (found by solving the
    // system)
    BigInteger[] log = new BigInteger[t+1];

    // The position of the pivot on each line in the system
    int pivot[] = new int[t+1], nnonzero;

    static final BigInteger ZERO = new BigInteger("0");
    static final BigInteger ONE = new BigInteger("1");

    public ServerLogic(BigInteger alpha, BigInteger beta, BigInteger p) {

        this.alpha = alpha;
        this.beta = beta;
        this.p = p;

        n = p.subtract(ONE);

        if (alpha.modPow(n.divide(new BigInteger("2")), p).
            equals(ONE)) {

            throw new RuntimeException(alpha + " is not a generator");

        }

        System.out.println("Generating the first " + t + " prime numbers...");
    }
}
```

```

    createFactorBase();
}

public BigInteger[] getBuffer() {
    return v;
}
50

public BigInteger[] getFactorBase() {
    return prime;
}

public void run() {
    synchronized (v) {
        System.out.println("Collecting / solving " + t + "equations ...");
        solveSystem();
        System.out.println("Cracking the log...");
        BigInteger l = crackLog();
        System.out.println("Verifying: " + alpha + "^" + l + " mod " + p
            + " = " + alpha.modPow(l, p));
    }
}

private void createFactorBase() {
    prime[1] = new BigInteger("2");
    for (int i=2; i<=t; i++) {
        prime[i] = prime[i-1];
        do prime[i] = prime[i].add(ONE);
        while (!prime[i].isProbablePrime(100));
    }
}
70

private void solveSystem() {
    int i = 1; // The current line we're collecting
    int redundant = 0; // How many redundant equations have we found?

    boolean[] pivoted = new boolean[t+2];
    boolean[] nonzero = new boolean[t+2];
80

    while (i-1 < nnonzero || i == 1) {

        // get a new line from a client
        try {
            v.wait();
        } catch (InterruptedException e) {};
    }
}

```

```

        // the following optimization tries to catch nonrelevant lines
        // early (there should be many of them at the end...)
    boolean relevant = false;
    //     for (int j = t+1; j >= 1 && !relevant; j--) {
    //         if (!pivoted[j] && !v[j].equals(ZERO)) {
    //             relevant = true;
    //         }
    //     }
    relevant = true;

        // Reduce this line with all the lines above it
    if (relevant) {
        for (int k = 1; k < i; k++)
            if (!v[pivot[k]].equals(ZERO)) {
                BigInteger iPivot = new BigInteger(v[pivot[k]].toString());
                BigInteger kPivot = eqn[k][pivot[k]];
                for (int j = 1; j <= t + 1; j++)
                    if (eqn[k][j].equals(ZERO) && v[j].equals(ZERO)) {
                        // we know we'll get zero, simply set to 0
                        // this also saves memory (no new allocation)
                        v[j] = ZERO;
                    } else {
                        v[j] = v[j].multiply(kPivot)
                            .subtract(eqn[k][j].multiply(iPivot)).mod(n);
                    }
            }

        // identity?
        relevant = false;
        for (int j = t; j >= 1 && !relevant; j--) {
            if (!v[j].equals(ZERO)) {
                relevant = true;
                pivot[i] = j;
            }
        }
    }

    if (relevant) {
        // Now we can reduce the values in column pivot[i] in all the lines
        // above i
        pivoted[pivot[i]] = true;
    }

```

```

BigInteger iPivot = v[pivot[i]];
for (int k = 1; k < i; k++) {
    if (!eqn[k][pivot[i]].equals(ZERO)) {
        BigInteger kPivot = new BigInteger(eqn[k][pivot[i]].toString());
        for (int j = 1; j <= t + 1; j++) {
            if ((pivoted[j] && pivot[k] != j) || eqn[k][j].equals(ZERO) && v[j].equals(ZERO)) {
                eqn[k][j] = ZERO;
                140
            } else {
                eqn[k][j] = eqn[k][j].multiply(iPivot)
                    .subtract(v[j].multiply(kPivot)).mod(n);
            }
        }
    }
}

// copy
eqn[i] = new BigInteger[t+2];
150
for (int j = 1; j <= t+1; j++) {
    eqn[i][j] = v[j];
    if (j<=t && !v[j].equals(ZERO) && !nonzero[j]) {
        nonzero[j] = true;
        nnonzero++;
    }
}

System.out.print(i + "(" + nnonzero + ") ");
i++;
160
} else {
    System.out.print(' ');
    redundant++;
}

// for (int a = 1; a < i; a++) {
//     for (int b = 1; b <= t+1; b++)
//         System.out.print(eqn[a][b] + " ");
//     System.out.println("pivot: " + pivot[a]);
// }
}
170
System.out.println();
System.out.println("Found " + redundant + " redundant equations.");

// Now we have a bunch of equations of the form ax=b mod n, i.e.
// eqn[i][pivot[i] log[pivot[i]] = eqn[i][t+1]
// The way to solve for x is:
// - divide a, b and n by d=gcd(a,n). If the equation has a solution then
//     d | b, and there are d solutions to the equation.

```

```

        // - a is now invertible, hence  $x = b a^{-1}$  is the unique solution for
        // the simplified equation 180
        // This gives a base solution x. Other solutions have the form
        //  $x + kn$ , where  $k = 0, 1, \dots, d-1$ 
    for (i = 1; i <= nnonzero; i++) {
        BigInteger d = eqn[i][pivot[i]].gcd(n);
        BigInteger a = eqn[i][pivot[i]].divide(d);
        BigInteger b = eqn[i][t+1].divide(d);
        BigInteger m = n.divide(d);
        log[pivot[i]] = b.multiply(a.modInverse(m)).mod(m);
        for (long k = 0; k < d.longValue(); k++)
            if (alpha.modPow(log[pivot[i]], p).equals(prime[pivot[i]])) 190
                break;
            else
                log[pivot[i]] = log[pivot[i]].add(m).mod(n);
        if (!alpha.modPow(log[pivot[i]], p).equals(prime[pivot[i]]))
            System.out.println("Couldn't find the log of "
                + prime[pivot[i]]);
            //else
            //System.out.println("log of " + prime[pivot[i]]
            //                    + " is " + log[pivot[i]]);
    } 200
}

private BigInteger crackLog() {
    Random rnd = new Random();
    BigInteger k, z, result;
    while(true) {
        // Generate a number between 0 and p-1
        k = new BigInteger(p.bitLength(), rnd);
        if (k.compareTo(p) != -1) continue; 210

        // Try to factor beta x alpha^k
        result = ZERO.subtract(k).mod(n);
        z = alpha.modPow(k, p).multiply(beta).mod(p);
        for (int j = 1; j <= nnonzero; j++) {
            BigInteger exponent = ZERO;
            // System.out.println("prime: " + prime[pivot[j]]);
            while (z.mod(prime[pivot[j]]).equals(ZERO)) {
                z = z.divide(prime[pivot[j]]);
                exponent = exponent.add(ONE);
                // System.out.println("k = " + k); 220
            }
            result = result.add(exponent.multiply(log[pivot[j]]).mod(n));
        }
    }
}

```

```

        // Well, did it work?
        if (z.equals(ONE))
            return result;
    }
}
}

```

230

## 4.4 ServerIOWorker.java

```

import java.math.BigInteger;
import java.io.;
import java.net.;

public class ServerIOWorker extends Thread {

    private int t;

    private BigInteger alpha, p, r, n, ralpha, rin;
    private ObjectInputStream fromClient;
    private ObjectOutputStream toClient;

    private BigInteger[] buffer, recv, factorBase;

    static final BigInteger ZERO = new BigInteger("0");
    static final BigInteger ONE = new BigInteger("1");

    private InetAddress client;

    public ServerIOWorker(BigInteger alpha, BigInteger p,
        BigInteger[] factorBase, BigInteger[] buffer,
        Socket socket) throws Exception {

        this.factorBase = factorBase;
        this.t = factorBase.length - 1;
        this.alpha = alpha;
        this.p = p;
        this.buffer = buffer;
        n = p.subtract(ONE);

        setDaemon(true);
    }
}

```

```

client = socket.getInetAddress();
setName("IOWorker(" + client.getHostName() + ")");

    // init salt
do {
    r = new BigInteger(p.bitLength(), new java.util.Random());
}
while (r.compareTo(n) < 0 && r.gcd(n).equals(ONE));

ralpha = alpha.modPow(r, p);

try {
    fromClient = new ObjectInputStream(socket.getInputStream());
    toClient = new ObjectOutputStream(socket.getOutputStream());

        // send problem to client
toClient.writeObject(ralpha);
toClient.writeObject(p);
toClient.writeObject(factorBase);
} catch (Exception e) {
    System.out.println("Client " + client + ": initialization failed:");
    e.printStackTrace();

    throw e;
}

        // done init
System.out.println("Client " + client + ": initialization succeeded");
}

public void run() {

    while (true) {
        try {
            recv = (BigInteger []) fromClient.readObject();
        } catch (Exception e) {
            System.out.println("Client " + client + ": I/O operation failed, terminating connection");
            e.printStackTrace();
            return;
        }

        // insert verifications and decryption here
        // verification
        // BigInteger result = ONE;

```

```

//          for (int i = 1; i<=t; i++)
//              if (!recv.equals(ZERO)) {
//                  result = result.multiply(factorBase[i].modPow(recv[i], p));
//              };

//          if (result.equals(ralpha.modPow(recv[t+1], p))) {
//              System.out.print("XOK");
//          } else {
//              System.out.print("XBad!");
//          }

```

80

```

recv[t+1] = recv[t+1].multiply(r);

synchronized(buffer) {
    System.arraycopy(recv, 1, buffer, 1, t+1);
    buffer.notify();
}
}
}

```

90

## 4.5 GenPrimePairs.java

```

import java.io.;
import java.math.BigInteger;
import java.util.Random;

public class GenPrimePairs {

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java GenPrimePairs <bitlength>");
            return;
        }
        int bitLength = new Integer(args[0]).intValue();
        Random rnd = new Random();
        BigInteger p, q;
        do {
            p = new BigInteger(bitLength, 100, rnd);
            q = p.multiply(new BigInteger("2")).add(new BigInteger("1"));
        } while (!q.isProbablePrime(100));
        System.out.println(p + " " + q);
    }
}

```

10

20