COMP 360

Final Project

RSA Moduli Pitfalls

Authors:
Matt Adelman
Evan Carmi
Adam Forbes

December 14, 2012

Contents

Abstract

Hello world, this is the abstract. Hello world, this is the abstract. Hello world, this is the abstract. Hello world, this is the abstract. Hello world, this is the abstract. Hello world, this is the abstract. Hello world, this is the abstract. Hello world, this is the abstract.

1. History:

RSA is a public key encryption algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT in 1977. Rivest and Shamir are both computer scientists that were working on an "unbreakable" public key encryption method. Rivest and Shamir worked on many different codes, and would pose them as a challenge to Adleman. Forty two of these codes were presented, and Adleman broke them all. Finally on attempty number 43, they created what is now known as the RSA scheme. Incidentally, English Professor CLifford Cocks developed the exact same encryption system in 1973, but it was classified as top-secret, so it was not released until 1997. The RSA algorithm was released for public domain in 1997.

The algorithm operates by using two distinct, large prime numbers to generate public and private keys. Anyone can use the public key to encrypt a message, but only someone with the private key can decrypt it. The algorithm is hard to break, because if the prime numbers are large enough, the factorization is exponential in time.

2. Original Ideas:

When we first started out with this project, our ideas were fairly straightforward. We were going to implement the RSA algorithm in JavaScript, put it on a website, and address some of the common implementation issues that cause security flaws. A few that we were going to address were timing attacks, public exponent flaws and bad prime generation. This is where things got interesting. We were given the paper by Lenstra et al "Ron Was Wrong, Whit Is Right." While more on this paper will be addressed in the next section, the experiments performed by Lenstra exposed how big of an issue bad prime generation was. Our goal then shifted away from rigorous implementation, although we still did create a sufficient implementation, to recreating the experiments from the paper and attempting to find some bad moduli of our own.

3. Lit Review:

As stated earlier, "Ron was Wrong, Whit is Right" was not the initial focus of our project. However, upon further reading and the study of Lenstra et al. proved to be instrumental in the direction of our project. The paper was published on February 12, 2012 and created a splash in the field of cryptography. Its most influential discovery was that two out of every thousand RSA moduli offer no security at all. Leading to the conclusion that RSA offers a disconcertingly inadequate 99.8% security. This was discovered by utilizing the Euclidean Algorithm and running it against their huge dataset of certificates in order to find moduli with shared primes.

Their findings were shocking to us. We were inspired to recreate their experiments on the state of the web nine months later in the hopes of finding one of two possible conclusions. Either we find broken moduli or we find none. The former would imply a lack of real-world implementation despite the influence of their paper. The latter would indicate an improvement of online security since the

data collection of Lenstra et al. However, in order to begin our recreation we needed to first review their work which will be summarized below.

They gathered their data from a collection of public-key databases. Of these the two most significant databases being the SSL Observatory (a project run by the Electronic Frontier Foundation) and one from Massachusetts Institute of Technology. Their method of collection differed greatly from ours as they "did not engage in web crawling, extensive ssh-session monitoring, or other data activities that may be perceived as intrusive or aggressive", a philosophy we did not share. Their research led them to spend a long time discussing the differences between the public keys they collected and the security differences between them. They did extensive clustering of moduli to describe them as a graph. However, many of their findings are only tangentially related to the work we have done and therefore will be left out of this section.

The most pertinent information we are concerned with lies in the size of their final dataset and the number of compromised RSA moduli agnostic of the types of certificates they came from. They collected 11666704 public keys of which 6 386984 had distinct RSA moduli with the rest being split evenly between ElGamal and DSA. Of these 6 386 984 moduli they found 12934 different ones which could be factored by 14901 distinct primes and were therefore compromised.

4. Algorithm Implementation:

For our implementation, we chose JavaScript as our environment because of its ease for web development. We really wanted to end up with something that we could show off and display in a public setting. The code to the implementation of our RSA algorithm can be found in Appendix A. For our implementation, we relied heavily on the BigInt library found here: http://leemon.com/crypto/BigInt.js. While there were other BigInt libraries, out there but this is the one that had all of the functions we needed, and was implemented in an easy format for us to deal with. The main functions we used from this library were `str2bigInt` which takes a string and a base that represents the big integer and converts it for us. We also used `inverseMod` which takes a big integer and a modulus as a big integer, and returns the big integer that is the inverse if one exists, otherwise `null`. We also used `mult` which has the obvious function of multiplying two big integers. We also have `greater` which returns a boolean depending on whether or not the first argument is greater than the second. We also used `modPow(a,b,c)` which takes three big integers and returns $a^b \mod c$. The last function we used from Leemon was `addInt` which takes a big integer and an integer and adds the integer to the big integer. This was used in the computation of $\phi(n)$. The functions we created are as follows: `generate_key(p:String, q:String, e:String) = [[n:BigInt, e:BigInt], [n:BigInt, d:BigInt]]` where p and q are our distinct primes and e is the public exponent all represented as strings. We then return n which is the modulus, with $n = pq$. Together with e the public exponent and d the private exponent. These together make up our keys. The second function we created is `crypt(key:[BigInt,BigInt], message:BigInt) = message′:BigInt` where the *key* is a modulus with an exponent, and the *message* is something to be encrypted or decrypted depending on what *key* with which we call `crypt`.

The next step for our implementation was getting everything to play nice with our web implementation (found here: http://carmi.github.com/rsa/code/). The

method was to create some helper methods that would access the data from our form, check for errors on input and even randomly generate pseudo-random primes based on a bit length. The code for these helper functions can be found in Appendix A. Some of the big ones are the `gen` function which takes the information given in the form, tests for errors such as bad public exponents, or if the public exponent is not relatively prime to $\phi(n)$. We also have functions to encrypt and decrypt messages given that we have public and private keys entered into our form.

To ensure that future changes do not introduce bugs into our already working codebase we use tests. For our JavaScript implementation of RSA we used QUnit tests, a testing framework in JavaScript used by jQuery among others. This allows our tests to be run easily in the browser along side our implementation. If future changes break a function, then the associated test will fail alerting us to the portion of code that is broken. This included minor tests for generation, encryption, decryption, and even a few number theory functions which were in our original implementation of the RSA algorithm that only worked on integers. This is actually when we realized we needed to switch to arbitrary precision integers, because the exponents get very large, very quickly.

5. Data Collection:

Any analysis of RSA certificates requires a set of data with which to test. For our project we needed a large set of data so as to best mimic the data that Lenstra et al used in their analysis of public keys. Lenstra et al stated in their paper that they used the Electronic Frontier Foundation's SSL Observatory - https://www.eff.org/observatory. We initially looked at the SSL Observatory as a possibly source of data for our project. However, unfortunately the SSL Observatory has not been updated in a few years. Furthermore, it includes no complete guide on how to use its source, and is simply a collection of python scripts that interact with a `MySQL` database in distinct ways. Thus, we were unable to successfully use the SSL Observatory source code as a means to gather RSA public keys. However, there exists data dumps from the SSL Observatory from 2010. We gathered this data, available through `bittorrent`, and used it as a starting point.

In order to see the effectiveness and change since the Lenstra paper was published in early 2012 we needed to scrap for fresh data. (The paper briefly mentions that the latest data they looked at was from February 2012.) The SSL Observatory scans all `IPv4` space for servers that respond to https requests. We did not have the computational time/power to do this, so we decided on the Alexa top 1 million domains as a source of hosts. Then for each one of these domains we attempted to connect via HTTPS and collect the public key used for the connection. The specific method of collecting these moduli was the This was done by `curl`'ing the domain at port 443 with a `HTTP` request. If we received a response, we used `openssl s_client` to get the decoded certificate and `openssl x509` to decode the certificate into a easily passable format. However, we noticed that less than 10% of domains responded to our requests. We added some further error handling for these situations. We followed redirects and tried a few common subdomains to improve our rate of responses to over 12%. Out of the 1000000 domains we received 125723 X.509 certificates each containing a moduli. This was the source of our data for our experiments.

In order to deal with our large directory of certificates we needed combine them into a single csv that-for the sake of efficiency-discards all the impertinent data. We wrote a parsing program in Java that reads in ev-

ery file and parses it into a combined csv file. The biggest challenge with dealing with the certificates was handling the exceptions that inevitably arise when dealing with large numbers of only-mostly standardized data.

Once we had all of the certs in a csv file. The next step was to parse out the information we needed. Since the modulus was in a different column for each cert file, therefore it was in a different column for each line in the csv file. However, there was the unique character pattern ")Modulus (" on each line of the file. Therefore we could match for that using the `substring` method of the `String` class in Java. Once we had that, we stripped the modulus of all extraneous characters such as white space and colons. Then since all of the other information we needed in the csv file was before the modulus, we just used a `split(‘‘,’’)` on the line to split around commas and easily obtained the domain name, and a unique identifier for each modulus.

Finally we needed code to create a random sample of moduli to more swiftly run an analysis on. We also wanted to make sure that these moduli were unique. We did this by looping through and adding the moduli we chose to a set to make sure we never chose two that were the same. The parsing and random sample getting code is seen in Appendix A.

6. Experiments:

The main idea of our experiments is as follows: if we have two moduli m, n we can find if these two moduli share a prime p simply by taking the gcd of the two moduli. This is very fast and is linear in the number of bits. To do around 100000 gcd operations on the machines we were using took around 5 seconds each. Therefore our code, seen in Appendix A, does the following: It takes a file that contains the list of moduli, along with other identifying information. It also takes a smaller file which only contains a small section of the moduli. This was done so the processing could be broken up onto multiple machines. Once the two files are read into memory, the program stores the moduli in an array, and a unique identifier corresponding to the moduli in another array. We then loop through the data comparing every modulus in the smaller file with every modulus in the bigger file.

We used Java for our experiments due to speed, and the ease with which we can allocate memory to the virtual machine. The default JVM heap space is not large enough to read in the files we were working with (some of which were over 100MB) so we allocated extra memory with the `java -Xmx2g` option which allocates an extra 2GB of data to the virtual machine. We probably did not need this much more allocated, but it was just as a precaution.

The algorithm was designed to overlook all of moduli that are the same, because while that is interesting information, it gave us no additional data. If the algorithm came across two moduli n, m such that $n \neq m$ and $\gcd(n, m) \neq 1$ then we know we have a prime in common and have factored the modulus, rendering it insecure. Once it finds one, it prints out the gcd, the unique identifier for each modulus and the result of calling `isProbablePrime(80)` which returns true if there is less than a 2^{-80} chance that the gcd is composite. We then can use this data to try and factor the modulus.

7. Analysis of Data:

Using the results of Lenstra et al. we shall attempt to analyze the significance of our analysis. Here is a summary of both of our analysis:

Lenstra et al.

- 11666704 public keys

- 6386984 distinct RSA moduli
- 12934 distinct broken moduli
- 14901 distinct primes

Us

- 105984 distinct RSA moduli
- $\sim$60000 moduli analyzed
- 0 broken moduli
- 0 primes

These results are reassuring. They show that out of the 105984 distinct RSA moduli observed by our scraping we could not find any that were compromised. Unfortunately, due to a lack of time and resources (we ran analysis on six computers for approximately two weeks) we were only able to verify around 60 000 of the moduli against the set of 105984. Our analysis is ongoing and therefore 60000 is constantly increasing.

The set of 60000 were spread over six clusters arranged alphabetically on their url. We will assume that these were chosen randomly as the original list of the top 1 million domain names was outputted alphabetically and not by ranking. In addition to this prefixes to domains and subdomains further splits up the set of certificates collected.

Our first attempt at calculating the statistical significance of our results was to ask the following question: if there is a .002 chance of finding a broken moduli in the set of Lenstra et al. how many moduli should we expect to find?

- Let n be the size of our sample set of distinct moduli
- Let p be the probability of finding a broken moduli
- Let m be the expected number of broken moduli
- $np = m$
- $105984 \cdot .002 = 212$

We would expect to find 212, and we found 0. Is this significant? Yes it is, because the chance of finding no broken moduli is $(.998)^{50000} = 3.37 \cdot 10^{-44}$.

However, there are many problems with this analysis. The largest being that we were only able to scrape a dataset with a size that's 1.66% of that analyzed by Lenstra et al. This wouldn't be a large problem if you could identify the broken moduli by themselves, but by the very nature of our analysis you need to run the Euclidean Algorithm on every single moduli against every other moduli, hence the unsurpassable amount of time and computing power required to factor the moduli. This means that the chances of finding a broken moduli scale very badly with a smaller sample size.

Lenstra et al. found a large range of compromised moduli clusters. However, for this analysis let's assume the worst case: every modulus has a pair that shares a prime. In the case of the research done by Lenstra et al. we have 14901 distinct primes and therefore a total of 29802 pairs. This is no the actual case because their research shows clusters of moduli that share the same prime. If we were to include the statistical probabilities of every group size the model would become too cumbersome and therefore analyzing the worst case scenario provides a good estimate.

Due to a lack of probability experience we have written a program that simulates our statistical problem and determines the expected number of pairs. The program runs as follows:

- Populate an array of size 6386984 with 0s
- Loop 14901 times. Each time randomly assigning two unassigned elements to be a and negative a. These will act as the pairs of moduli that share a prime.
- Create another array of size 105984 and populate it with a random selection from the first array.
- Count the number of pairs in the new array

This program was run 100000 times and found that, on average, there are 8.188 pairs in the second array. Meaning that we should

expect to find at 8 pairs in our dataset âĂŞ a fact we did not. Now that we know we should expect 8 pairs in our dataset we need to calculate the probability of our findings. This code is seen in Appendix A.

First we need to calculate the probability any given moduli should be broken. $p = 8/105984 = 7.73 \cdot 105$ But if we only completed analyzing 50000 moduli, how many should we find to be broken? #broken = $(7.73 \cdot 105) \cdot 50000 = 3.86$ Let's round this up to a probable 4 broken moduli.

Therefore we can say with confidence that 8 out of every 100000 should be broken in our smaller set and we should expect to find a little under 4 broken moduli in our analysis. Notice that this chance of pairs is based on the research of Lenstra et al. and yet gives us a substantially worse chance of finding broken moduli. We have a 0.0000755 chance of finding a compromised moduli out of 105984 whereas their research found a 0.002 chance of finding a compromised moduli out of 6386984 moduli.

8. Problems:

Computation Power - The largest problem we encountered was computational power. Simply put, we did not have the computing power to analyze all the data we gathered. There primary function we wanted to do was run a gcd across every moduli we encountered. And we couldn't do this in time. There were a few methods we took to try and solve this. We split our data into multiple sections and ran it across a few different machines. Nonetheless, we were unable to process enough data fast enough. We simply needed a compute cluster, or more time. At some points we had data running on five different machines.

SSL Observatory - We attempted to contact the EFF SSL Observatory maintainer for help getting the Observatory running, and to ask if there was a newer data dump than the 2010 data that was on bittorrent. We never received a response. It's likely that the not-for-profit EFF didn't have anyone working on the project who wanted to help us.

Same Moduli - Something we did not expect but ran into quite frequently were the existence of many moduli that were the same. For instance out of the 125723 moduli that we collected, just over 104000 were unique. Therefore around 18% of the moduli we collected were not unique. In addition, around 10% of that was all from the same domain $*$.googleusercontent.com. The implication of this is that we could not run as many comparisons as we would have liked, because we had to throw out a lot of same data. The fact that Google uses the same moduli for so many of its domains really speaks to the security of this algorithm

9. Conclusion:

We did not find any moduli that did not offer security. What are the chances of this happening? $p = (1 - 0.000075)^{50000} = 0.0235$ There is a 2% chance of not finding any moduli in our set, and therefore we can say with 98% confidence that our findings show an improvement in RSA security because we were unable to find any broken moduli.

One possible explanation for this can be found in the bias of our domain names. We scraped from the top million most trafficked websites online, and therefore one can hypothesize that there is a lot of overlap between this list and the list of most maintained websites online. The highest probability of finding a bad modulus may lie in looking at the top million least visited websites.

10. Further Research:

The first priority would be to collect a larger dataset of moduli closer to that of Lenstra et

al. as it would greatly improve the statistical significance of our research. With the data we collected, including the older 2010 data dump from the SSL Observatory, we have a large set of moduli. We discussed the possibility of creating a service that would check any new moduli against the entire every seen set of moduli to ensure there are never duplicates. This would, in effect, allow us verify that certificates are safe prior to them being used by websites. In fact, we hope that some certificate authorities are already doing this to ensure that within their own network of certificates there are no duplicates. The fact that gcd is so fast makes a database, and a test like this incredibly feasible.

A. Appendix of code:

Listing 1: A ruby script that interfaces with unix command line utilities to collect X.509 certificates given a list of hosts.

```ruby
# A ruby shell script to scrape for https certificates and decode them.
# @author - Evan Carmi

# a csv file with a number and host on each line. Ex: 1,google.com.
# Often useful: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

HOST_FILE = 'unseen.txt'
PORT = 443

# Directory for downloaded certificates
CERT_DIR = 'certs'

# Directory for decoded certificates
DECODED_DIR = 'decoded'

`mkdir -p #{CERT_DIR}/`
`mkdir -p #{DECODED_DIR}/`

def cmd_runner(cmd)
  out = `#{cmd}`
  unless $?.success?
    puts "\n\n\nERROR\nFailed while running #{cmd} with error: #{out}\n\n\n"
  end
  $?.success?
end

File.open(HOST_FILE).each_line do |line|
  domain = line.strip

  # first check if server responds to https request
  cmd = "curl -sI https://#{domain} --connect-timeout 2 -m 2"
  worked = cmd_runner(cmd)
  unless worked
    domain.prepend("www.")
    cmd = "curl -sI https://#{domain} --connect-timeout 2 -m 2"
  end
  next unless cmd_runner(cmd)

  # download certificate
  cmd1 = "echo -n | openssl s_client -connect #{domain}:#{PORT} | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > #{
      CERT_DIR}/#{domain}.cert"
  next unless cmd_runner(cmd1)

  # unpack certificate
  cmd2 = "openssl x509 -in #{CERT_DIR}/#{domain}.cert -text -noout > #{DECODED_DIR}/#{domain}.cert.decoded"
  next unless cmd_runner(cmd2)

  cmd3 = "echo '#{domain}' >> downloaded_hosts.txt"
  next unless cmd_runner(cmd3)

end
```

Listing 2: A Java class that takes the csv file of certs and extracts all the useful information.

```java
/*
 * Matt Adelman
 * RSA Final Project
 */

```

```
import java.math.BigInteger;
import java.util.Scanner;
import java.io.FileNotFoundException;
import java.io.FileReader;

public class FormatNewData {

    public static void main(String[] args) {

        String[] lines = new String[125723];

        try {
            // Getting the file
            String fileName = "formatted-certs-all.csv";
            Scanner file = new Scanner(new FileReader(fileName));
            // Looping through the file
            for (int i = 0; i < lines.length; i++) {
                String line = file.nextLine();
                String add = "";
                int start = line.indexOf("Modulus");
                String rest = line.substring(start);
                start = rest.indexOf(":");
                rest = rest.substring(start);
                String modulus = (rest.split(",")[0]).replaceAll(":","");
                String[] splits = line.split(",");
                add = add + splits[0] + ",";
                add = add + splits[2] + ",";
                add = add + modulus + "\n";
                lines[i] = add;
            }

            for (int i = 0; i < lines.length; i++) {
                System.out.print(lines[i]);
            }

        }



    // Catching a FileNotFoundException exception
        catch (FileNotFoundException fnfe) {
            System.out.println("File was not found");
            System.exit(0);
        }

    }

}
```

Listing 3: A Java class to get the random sample of moduli.

```
/*
 * Matt Adelman
 * RSA Final Project
 */

import java.util.Scanner;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.HashSet;
import java.util.Random;

public class GetRandomSample {


    public static void main(String[] args) {

        int count = 0;
        int samples = 10000;

        String[] allLines = new String[125723];

        String[] sample = new String[10000];
        String[] modArr = new String[allLines.length];
        Random rand = new Random();
        HashSet mods = new HashSet(20000);

        try {
            // Getting the file
            String fileName = "new-form-data.csv";
            Scanner file = new Scanner(new FileReader(fileName));
            // Looping through the file
            for (int i = 0; i < allLines.length; i++) {
                String line = file.nextLine();
                allLines[i] = line + "\n";
                modArr[i] = line.split(",")[2];
```

```
            }
        }

        catch (FileNotFoundException fnfe) {
            System.out.println("File was not found");
            System.exit(0);
        }

        while (count < samples) {
            int index = rand.nextInt(allLines.length);
            if (modArr[index].length() > 10 && mods.add(modArr[index])) {
                sample[count] = allLines[index];
                count++;
            }
        }

        for (int i = 0; i < sample.length; i++) {
            System.out.print(sample[i]);
        }

    }

}
```

Listing 4: A Java class that reads in two files containing our moduli and other information and prints out a prime if we find one between two moduli.

```
/*
 * Matt Adelman
 * RSA Final Project
 */

import java.math.BigInteger;
import java.util.Scanner;
import java.io.FileNotFoundException;
import java.lang.NumberFormatException;
import java.lang.StringIndexOutOfBoundsException;
import java.io.FileReader;

public class RandomSampleCheck {

    public static void main(String[] args) {

        int count1 = 0;
        int count2 = 0;
        // Public moduli
        BigInteger[] keys1 = new BigInteger[2000];
        BigInteger[] keys2 = new BigInteger[125723];
        // associated data
        String[] data1 = new String[keys1.length];
        String[] data2 = new String[keys2.length];
        BigInteger big;

        try {
            // Getting the file
            // Change each time
            String fileName1 = "xaa";
            Scanner file1 = new Scanner(new FileReader(fileName1));
            String fileName2 = "new-form-data.csv";
            Scanner file2 = new Scanner(new FileReader(fileName2));
            // Looping through the file
            for (int i = 0; i < keys1.length; i++) {
                // Getting the line of the file with pertitant info
                String line = file1.nextLine();
                // Break the line, because it's a CSV
                //

                String[] lineScan = line.split(",");
                data1[i] = lineScan[1]; //id number
                String modulus = lineScan[2];
                big = new BigInteger(modulus, 16);
                keys1[count1] = big;
                count1++;
            }
            for (int i = 0; i < keys2.length; i++) {
                // Getting the line of the file with pertitant info
                String line = file2.nextLine();
                // Break the line, because it's a CSV
                //

                String[] lineScan = line.split(",");
                data2[i] = lineScan[1]; //id number
                String modulus = lineScan[2];
                big = new BigInteger(modulus, 16);
                keys2[count2] = big;
```

```
                count2++;
            }
            System.out.println("Done!");
            // Timing
            Long startTime = new Long(System.currentTimeMillis());
            // Pairwise GCD of moduli
            for (int i = 0; i < count1; i++) {
              if ((i % 1000) == 0) {
              System.out.println("Working on i = " + i);
              }
                for (int j = 0; j < count2; j++) {
                    BigInteger gcd = keys1[i].gcd(keys2[j]);
                    BigInteger one = BigInteger.ONE;
                    // If the Moduli are equal we get no useful info
                    if (!(one.equals(gcd)) && !(keys1[i].equals(keys2[j]))) {
                        System.out.println(data1[i] + ", " + data2[j] +
                        ", " + gcd + gcd.isProbablePrime(80));
                    }
                }
            }

        Long endTime = new Long(System.currentTimeMillis());

        //System.out.println(endTime.intValue() - startTime.intValue());
            //System.out.println(count);
            file1.close();
            file2.close();
        }

    // Catching a FileNotFoundException exception
        catch (FileNotFoundException fnfe) {
            System.out.println("File was not found");
            System.exit(0);
        }
        catch (NumberFormatException nfe) {
            System.out.println(count1 + 1);
            big = BigInteger.ONE;
        }
        catch (StringIndexOutOfBoundsException sioobe) {
            System.out.println(count1 +1);
            System.out.println("out of bounds");
        }



    }


}
```

Listing 5: A Java class that simulates the probable number of pairs of bad moduli we should have found.

```
/*
Adam Forbes
RSA Project
*/


import java.math.*;
import java.util.Arrays;

public class statTest {
    public static void main(String[] args) {
        //The following program is written to figure out the chances of getting
        //no pairs in our smaller set of moduli collected.
        //Lenstra et al. used 6 386 984 moduli and found 14901 distinct primes
        int numTests = 100000;
        int lenstraSetSize = 6386984;
        int numberOfPairs = 14901*2;
        int ourSetSize = 105984;
        int[] numPairs = new int[numTests];
        populateNegOne(numPairs);

        double sum = 0;
        for (int i = 0; i < numTests; i++) {
            int[] lenstraSet = new int[lenstraSetSize];
            populateZero(lenstraSet);
            populatePairs(lenstraSet, numberOfPairs);

            int[] ourSet = new int[ourSetSize];
            populateSubset(lenstraSet, ourSet);

            numPairs[i] = numPairs(ourSet);

            sum += numPairs[i];

```

```
35              if (i % 100 == 0) {
36                  System.out.println("Test number: " + i + " of " + numTests);
37                  System.out.println("Average number of pairs: " + sum/i);
38              }
39          }
40      }
41
42      public static void populatePairs(int[] arr, int numPairs) {
43          //We will use the following notation:
44          //0: an unbroken moduli
45          //k: a broken moduli p1
46          //-k: the pair of p1
47
48          //we don't want to use 0 here because our default slot is 0!
49          for (int i = 1; i <= numPairs; i++) {
50              int p1 = (int)(Math.random() * arr.length);
51              int p2 = (int)(Math.random() * arr.length);
52
53              while (arr[p1] != 0) {
54                  p1 = (int)(Math.random() * arr.length);
55              }
56              while (arr[p2] != 0 && p2 != p1) {
57                  p2 = (int)(Math.random() * arr.length);
58              }
59              //Now we have chosen two empty spots the pair
60
61              arr[p1] = i;
62              arr[p2] = -i;
63          }
64      }
65
66      public static void populateSubset(int[] arr, int[] subArr) {
67          for (int i = 0; i < subArr.length; i++) {
68              //Because the array is populated at random we can assume
69              //that by choosing the first subArr.length number of
70              //moduli we are getting a random sample
71              subArr[i] = arr[i];
72          }
73      }
74
75      public static void populateZero(int[] arr) {
76          for (int i = 0; i < arr.length; i++){
77              arr[i] = 0;
78          }
79      }
80
81      public static void populateNegOne(int[] arr) {
82          for (int i = 0; i < arr.length; i++){
83              arr[i] = -1;
84          }
85      }
86
87      public static int numPairs(int[] arr) {
88          //this method will check for the number of pairs in an array
89          int[] sortedArr = Arrays.copyOf(arr, arr.length);
90          Arrays.sort(sortedArr);
91          int numPairs = 0;
92
93          //in order to check for pairs we need only to itterate from the start
94          //of the sorted array to the beginning of the 0s, each time checking
95          //if there exists a match to the negative number in the positive side
96          //A custom built sorting algorithm would probably be the most efficient
97          //method to do this... but time constraints have always been a problem
98          int i = 0;
99          while (sortedArr[i] < 0) {
100             //if the search is < 0 then there is no pair!!
101             if (Arrays.binarySearch(sortedArr, sortedArr[i]*-1) > 0) {
102                 numPairs++;
103             }
104             i++;
105         }
106
107         return numPairs;
108     }
109
110     public static void printArr(int[] arr) {
111         for (int i = 0; i < arr.length; i++) {
112             if (i % 20 != 0) {
113                 System.out.print("|" + arr[i]);
114             }
115             else {
116                 System.out.println();
117             }
118         }
119         System.out.println();
120     }
121 }
```