2008

A distributed public key creation system for ad-hoc groups

Brian Padalino

A Distributed Public Key Creation System For Ad-Hoc Groups

Brian Padalino
(bcp5143@cs.rit.edu)

September 12, 2007

Contents

List of Figures

List of Tables

Abstract

Ad-hoc networks are on the forefront of technological advances as more embedded devices allow for wireless communications without necessarily requiring a network infrastructure to connect to. One of the larger problems associated with such ad-hoc networks is the lack of being able to access a PKI to create individual secure sessions for these groups being created. For this project, an implementation generating the public and private keys for an RSA public-key protocol has been created on top of the M2MI middleware developed at RIT. In this implementation, as originally described by Dan Boneh, all parties help contribute to the generation of the RSA public modulus, N, without explicitly knowing the factorization of it. It has been shown that this implementation requires, on average, 32689 rounds of the protocol to create a 1024-bit RSA modulus for the group, and has an approximate growth of $\frac{log_2(N)}{2^5}$ rounds per bit.

1 Introduction

Recent publications have outlined a method for creating an RSA modulus, comprised of two prime numbers, within a distributed group to be possible. This project uses the M2MI layer developed at the Rochester Institute of Technology to create a group RSA modulus with separate decryption keys for each player in the group.

A few assumptions were made for the sake of simplicity. One such assumption is that each of the players will follow an honest-but-curious scenario and will not stray from the protocol. Secondly, a secure layer between the communicating objects was not added. Instead, it was assumed that all group public communications would be handled over an `M2MI.Omnihandle` whereas all private communications were over `M2MI.Unihandles`. This decision was made to limit the scope of the project to the creation of the RSA group. This is not an issue since the M2MI layer can be modified for secure `Unihandle` communications without affecting the results from this project.

2 Classical RSA

The classical RSA algorithm was developed by Ron Rivest, Adi Shamir, and Len Adleman at MIT in 1977. It is a public-key cryptography system that works between two parties over a public channel with the intention on making the channel private without requiring any previous communications.

2.1 RSA Algorithm Overview

The original algorithm as presented by Rivest, Shamir, and Adleman uses the properties of modular exponentiation along with prime numbers to "secretly" share information over a public channel. The algorithm uses the product of

RSA Algorithm
Input: Nothing
Output: Public-key(e, N), Private-key(p, q, d)

1. Choose two large prime numbers: p, q
2. Compute $N = p \cdot q$
3. Compute $\phi(N) = (p-1) \cdot (q-1)$
4. Choose an integer e such that $gcd(e, \phi(N)) = 1$
5. Compute d such that $de = 1 \ mod(\phi(N))$

Figure 1: Original RSA Algorithm

two *very* large prime numbers to create a public modulus to perform numeric operations on which allows for information to be obscured in a way believed to be very difficult to reverse without knowing the factorization of the modulus.

Though it is not proven that factorization is a problem that is extremely difficult, it is believed to be within that problem class. For that reason, RSA is the standard used in electronic commerce websites throughout the Internet.

2.2 RSA Example

A simple example is shown so that the steps of the RSA protocol can be followed and correlated to the group setting as presented later in the paper. It should be noted that even using relatively small values for p and q yields a relatively large value for N.

Using the values $e = 31$, $p = 887$ and $q = 991$, calculate d and test encryption and decryption.

$$\begin{aligned} N &= 887 \cdot 991 \\ N &= 879017 \\ \phi(N) &= 886 \cdot 990 \\ \phi(N) &= 877140 \\ e &= 31 \\ d &= e^{-1} \ mod \ \phi(N) \\ d &= 707371 \end{aligned}$$

After calculating d, we can test the algorithm by encrypting a test message. For this, we will use $m = 192$.

$$
\begin{aligned}
c &= m^e \ mod \ N \\
c &= 192^{31} \ mod \ 879017 \\
c &= 118121
\end{aligned}
$$

For this particular example, the ciphertext being sent over the public channel is equal to the number 118121 *mod* 879017. At this point, we should be able to apply the decryption exponent as calculated to reveal the original message.

$$
\begin{aligned}
p &= c^d \ mod \ N \\
p &= 118121^{707371} \ mod \ 879017 \\
p &= 192
\end{aligned}
$$

Thus, the RSA public-key crypto system example shows that given the correct public and private-key pairs, a message can be sent over public channels safely and securely.

The values for p and q used in the example are very small numbers - on the order of 10-bits. For standard security, a 1024-bit modulus (512-bit prime numbers) should be considered. If higher security is desired, a 2048-bit modulus should be used with other precautions when choosing the values of p and q. Further information can be referenced in Bruce Schneier's book, *Applied Cryptography*.

3 k-Group RSA

The idea behind RSA is very important to understand as it allows two parties to share information over a public channel in such a manner that only those members will know exactly what is being communicated. In a group setting, doing such a thing would require pairs between everyone being formed and the same message being encrypted k times, sent k times, and decrypted k times. The goal for a group setting should be one message being encrypted, broadcast once to everyone, decrypted and shared k times.

3.1 RSA Group Transition

The RSA algorithm has been incorporated into many electronic commerce protocols and is widely regarded as the de-facto standard for public-key cryptography. Unfortunately, the original RSA algorithm only works when dealing with a two-party system and wanting to securely share information between the two parties over a public channel.

k-Group RSA Algorithm
Input: Nothing
Output: Public-key(e, N), k Private-keys$(p_i, q_i, r_i, \alpha, \beta)$

1. Members choose random values for p and q
2. Members share and combine p and q candidates to calculate N
3. Members collectively test biprimality of N based on Fermat test
4. Members collectively test biprimality of N based on Boneh protocol
5. Use Catalano protocol to calculate individual private keys

Figure 2: k-Group RSA Algorithm

Luckily, due to the homomorphic properties of the original RSA concept, the algorithm needs only a few collaborative tweaks to extend from a two-party system to an k-party algorithm. Dan Boneh described an algorithm in which the homomorphic properties of the algorithm are exploited while using the scrambling and secure properties of distributed polynomial computation to create a group RSA protocol.

3.2 k-Group RSA Algorithm Overview

The algorithm leverages Shamir's "How To Share A Secret" paper wherein he proposes using randomly generated polynomials to perform such tasks as addition and multiplication. The method, as proposed by Shamir, uses the free term, or $f(0)$, of the polynomial as the value to be shared with every other coefficient randomly chosen. Overall, this is a very effective way of sharing the information with the group of semi-trusted players.

Once members are chosen, the algorithm is followed with points restarting with failures for the tests. For example, the biprimality tests may fail numerous times before the Catalano protocol is actually started.

All the data that was shared on an individual basis with each of the players was done so that a small modification to the M2MI layer can handle the level of point-to-point secure channels required.

4 Creating the Group

There are two different roles when forming a group: initiator and member. The initiator is, as the name suggests, the member which suggests forming the group. There can only be one initiator of a group, but many members. Moreover, the initiator has the special task of maintaining the state of the different members of the group and ensuring synchronization in the protocol. Members, on the other hand, are simply players in the creation process. Their main purpose is

to perform tasks and report back to the initiator.

Since the entire group creation process starts with the initiator, it's up to the initiator to create the member threshold and keep track of members joining and requesting to join the group. As members are being added to the group, a member list is constructed with an associated member ID. The initiator always has a member ID of 1 while every other member can have any other number, but for simplicity an incrementing member ID is used.

After the initiator has the required number of members in the member list, the list is distributed to all members. From this point, the algorithm starts the distributed process of calculating all necessary information for the formation of the secure group.

5 Choosing and Sharing Candidates

Once the group is created, it is up to the initiator to keep track of the state of each of the members. Moreover, it is up to the member to keep sending updated states to the initiator after completing a task.

5.1 Choosing Candidates

After the initial formation of the group and the member list is shared, the first step of the process is to select candidates and the sharing polynomials for the secret values of p and q.

For biprimality testing purposes later in the protocol, the secret values for p and q have to be Blum integers, or integers that are congruent to 3 mod 4. To ensure this, the initiator chooses their candidate to be congruent to 3 mod 4 while each member chooses a candidate that is congruent to 0 mod 4.

For choosing the polynomials for candidate sharing, each group member (including the initiator) chooses two random $\lfloor \frac{k}{2} \rfloor$ degree polynomials, $f(x)$ and $g(x)$ where k is equal to the number of members in the entire group and the free term is equal to the candidate value. This will allow that for every $f(x)$ chosen by every member, the evaluation $f(0)$ is equal to their candidate, p_i. The same logic can be applied to $g(0)$ and the respective candidate q_i.

Lastly, a randomizing polynomial, $h(x)$ with a degree of $k-1$ and a free term of 0 is chosen. This is strictly used as an extra measure for randomization.

5.2 Sharing Through Polynomials

As Shamir pointed out in his essay "How To Share A Secret", the use of polynomials can be used to create a system where an unknown secret can be shared between multiple parties. The interesting note in regard to this polynomial sharing is that each member has the ability to contribute to the secret without explicitly knowing the value of the secret in question. It is through this property that the candidates for p and q are shared and combined.

Player	$f(x)$	$g(x)$	$h(x)$
1	$143x+299$	$-240x+311$	$178x^2-582x+0$
2	$-180x+424$	$50x+400$	$24x^2-210x+0$
3	$100x+164$	$-64x+280$	$93x^2+ 40x+0$

Table 1: Players' choices of $f(x)$, $g(x)$, and $h(x)$

For an example, a small $k = 3$ group is being used, though the size of the group can be arbitrarily large. Each of the players chooses $f(x)$, $g(x)$, and $h(x)$ polynomials and sets their respective free terms to p_i, q_i and 0. In the example, p_1 corresponds to the value of 299, q_1 to 311, p_2 to 424, q_2 to 400, p_3 to 164, and finally q_3 to 280.

Note here that, in the example, the sum of the $f(x)$ free terms is 887 and the sum of the $g(x)$ free terms is 991 - two distinct prime numbers. Moreover, note the product of the two values is 879017. Lastly, note that the values for p_1 and q_1 correlates to the proper Blum values that should be chosen for the initiator of the group.

With the polynomials chosen, each player can now securely and individually share the evaluation of each of the functions with each other player. Each of the k players evaluates the values of $f(x)$, $g(x)$, and $h(x)$ for every other members member ID and shares those values.

$Player_i$	$f_j(i)$ **values**	$g_j(i)$ **values**	$h_j(i)$ **values**
1	442+244+264	71+450+216	-404-186+133
2	585+ 64+364	-169+500+152	-452-324+452
3	728 - 116+464	-409+550+ 88	-144-414+957

Table 2: Calculating individual components for each player

For example, $Player_1$ will receive k different shared values for each of the functions: $f_j(1)$, $g_j(1)$, and $h_j(1)$. $Player_2$ has their functions evaluated at the value of 2, and so forth up until $Player_k$ is evaluated at k.

At this point, each player has shared their partial information with each other without revealing anything other than a single point on a secret polyno-

mial. To get useful information from this, those secret polynomials are combined to create a single polynomial with the free term equal to the product of the sum of the free terms of all $f_j(x)$ and $g_j(x)$.

6 Finding the product N from Shared Candidates

Now, each player uses the data they have collected to create their individual N-share polynomial point. Each of these shares will then be viewed as a function evaluation of $N(i)$, where i is the member ID associated with the player.

$$N(i) \quad = \quad \sum_{j=0}^{k} f_j(i) \cdot \sum_{j=0}^{k} g_j(i) + \sum_{j=0}^{k} h_j(i)$$

Note the manipulation reveals nothing about the free-term of the polynomial generated in $N(x)$. In fact, all this function does is find a single point, i, on the function $N(x)$ which is a $k-1$ degree polynomial.

$Player_i$	$f_j(i)$ **sum**	$g_j(i)$ **sum**	$h_j(i)$ **sum**	$N(i)$
1	950	737	-457	699693
2	1013	483	-324	488955
3	1076	229	399	246803

Table 3: Calculation of individual $N(i)$ components

6.1 Adding and Multiplying Candidates

In this step, an individual and secure communication is sent between each of the members of the group. The data being shared uses each member's ID to evaluate the functions $f(x)$, $g(x)$, and $h(x)$. Each member sends this triple to each other member using their respective member ID for evaluation.

$$
\begin{aligned}
\sum f(i) &= \sum_{x=1}^{k} f_x(i) \\
\sum g(i) &= \sum_{x=1}^{k} g_x(i) \\
\sum h(i) &= \sum_{x=1}^{k} h_x(i) \\
N(i) &= \sum f(i) \cdot \sum g(i) + \sum h(i)
\end{aligned}
$$

After each member, i, receives k triples, each member then calculates the value of $N(i)$ which are points on a $k-1$ degree polynomial with the free term equal to the product of the sum of the shares p_i and q_i.

6.2 Lagrange Interpolation

At this point, all the members have secretly shared their private picks, p_i and q_i, mixed together with their randomizing function. On top of that, each member then multiplied out their share of the sharing. The result is a set of at least k points on a polynomial of degree $k-1$. Since there are k known points on the polynomial, a technique of Lagrange interpolation can be used to calculate the free term.

$$
\begin{aligned}
P(x) &= \sum_{j=1}^{n} P_j(x) \\
P_j(x) &= y_j \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k}
\end{aligned}
$$

The Lagrange interpolation takes the k points on the $k-1$ polynomial as (x_k, y_k) pairs and can interpolate the exact value of any arbitrary value, $P(x)$. In the case of all of the sharing done, the member ID always correlates to the x_k value with the y_k value being calculated at the steps. Calculating the interpolated $P(x)$ value ends up being a relatively trivial calculation.

6.2.1 Lagrange Example

Given the k points from the previous exchange of values, points on the $k-1$ degree polynomial are used to calculate the free-term of the polynomial, $P(0)$. The Lagrange interpolated value is the sum of the products for a value of $x = 0$.

After finding the values for $P_j(0)$, the free term is calculated by using the corresponding y_j values, multiplying them and using their sum. The free term

$Player_j$	y_j	$P_j(x)$	$y_j \cdot P_j(x)$
1	699693	$\frac{0-2}{1-2} \cdot \frac{0-3}{1-3}$	2099079
2	488955	$\frac{0-1}{2-1} \cdot \frac{0-3}{2-3}$	-1466865
3	246803	$\frac{0-1}{3-1} \cdot \frac{0-2}{3-2}$	246803

Table 4: Lagrange interpolation of players values

calculated is equivalent to the value of N in the group creation scheme. In this case, the value of N is calculated to be 879017.

After calculating N, the validity that N is truly a composite of two and only two primes must be tested. Therefore, probabilistic biprimality tests are used to try to determine, with a high amount of certainty, that the modulus created by the group truly is a composite of nothing more than two prime numbers.

7 Testing Biprimality of N

Simply creating the product of the sum of some candidates does not necessarily make a good RSA modulus. Remembering the requirements for an RSA modulus, the product should strictly be a product of two prime numbers. Other than factoring the number, there are no ways of testing strict biprimality with full confidence, but there are a few different probabilistic tests that can be done. These tests provide a high enough probability to be reliable, though it should be noted that a test message should be encrypted and decrypted to be completely positive.

7.1 Trial Division

The simplest of all the testing, trial division only requires the initiator to perform some division upon finding the public modulus, N. The idea stems from the fact that most composite numbers are composed of many small factors, and not necessarily a few large ones. Compiling a table of the first several thousand prime numbers requires very little effort and only a small amount of memory. Moreover, the test requires absolutely no communication.

Implementing this test can occur in two different ways. The first utilizes the initiator as the performer of the test. A problem with this model is the fact that one player is performing all the computations while all other players are sitting idle. This isn't a very efficient use of the computational power presented by the group, but it is very efficient with regards to communications.

The second implementation utilizes a somewhat distributed method of division. Given that every player has a list of the prime numbers, once the value of N has been computed, each player can then automatically start with

the trial division testing. Instead of testing every number, given a group of size k, each member will test every $k^{th} + ID$ number. For example, the initiator of the group divides by elements in the prime number array with indices of $1, k+1, 2k+1, 3k+1, ...$ while a player with an ID of 2 checks $2, k+2, 2k+2, 3k+2, ...$ etc. Therefore, the trial division task is split up into k tasks, all with similar computational complexity. If any of the players report failure, the initiator can then restart the protocol from the beginning with very minimal communication.

The particular implementation used should depend completely on the application along with the requirements of the protocol. One uses more time due to computing complexity but uses no communications, while the other uses less processor power but requires more communications.

7.2 Fermat Test

The first test performed is a modified Fermat primality test. In the normal Fermat primality test, a generator of the group, g, is calculated and raised to the value $p-1$ all taken mod p. If the number is prime, then the value should be congruent to 1, otherwise the number is considered composite.

$$\begin{aligned} \phi(p) &= p-1 \\ \phi(p \cdot q) &= (p-1) \cdot (q-1) \\ \phi(N) &= p \cdot q - p - q + 1 \end{aligned}$$

This test can be extended such that any generator, g, raised to the value $\phi(N)$, where N is the number being checked, is congruent to 1 mod N. In the case of biprimality where N is a composite of p and q primes, $\phi(N)$ is equal to the value of $N - p - q + 1$.

$$\begin{aligned} g^{\phi(p)} &\equiv 1 \ mod \ p \\ g^{\phi(N)} &\equiv 1 \ mod \ N \\ g^{N-p-q+1} &\equiv 1 \ mod \ N \end{aligned}$$

For security purposes, the values of p and q must be kept secret, as well as $\phi(p)$, $\phi(q)$ and $\phi(N)$. To keep all these numbers secret in the case of a distributed p and q, the initiator chooses a random group generator, g, and calculates the value $g^{N-p_i-q_i+1}$ while all other members calculate values $g^{-(p_i+q_i)}$ all taken mod N. All the values are shared in a protocol which is similar to the previous polynomial sharing method, but changes the free term of the polynomial to allow for multiplicative sharing instead of the additive sharing as used previously.

7.2.1 Fermat Example

To safely perform the Fermat test over the group of shared candidates, a polynomial sharing scheme is used similar to the candidate sharing and combining, except the function performed on the received points of the polynomial is only multiplication.

To perform the test, the initiator finds a generator number, g, and evaluates the Jacobi symbol $\left(\frac{g}{N}\right) = -1$. This is done to show the generator is a quadratic nonresidue of the modulus N. Once chosen, this generator of the group is distributed among the players and is used for the Fermat test.

Next, the initiator calculates the value $g^{N-p_1-q_1+1} \ mod \ N$ while all other players calculate $g^{-(p_i+q_i)} \ mod \ N$. Continuing, each player then chooses a random polynomial for each of the other players, setting the product of the free terms of each polynomial to be the value calculated for the Fermat test.

$$\prod_{n=1}^{k} f_n(0) = \begin{cases} g^{N-p_i-q_i} & i = 1 \\ g^{-(p_i+q_i)} & i \neq 1 \end{cases}$$

$$\prod_{i=1}^{k}\prod_{j=1}^{k} f_{i,j}(0) \equiv 1 \ mod \ N$$

For the current example, the Jacobi symbol for $\left(\frac{27}{879017}\right) = -1$. Using this as the generator, g, each player then calculates the Fermat test value to be shared.

Player	Fermat Test	Fermat Value
1	$g^{N-p_1-q_1+1}$	193382
2	$g^{-(p_2+q_2)}$	565943
3	$g^{-(p_3+q_3)}$	716811

Table 5: Calculating Fermat Values for players

This section extends the example throughout the paper by showing the Fermat Testing phase. The Benaloh protocol can be referenced from the Boneh paper and will not be shown in detail for simplicity and clearness of the example.

7.3 Step(4) Test

The second test that is performed tests a specific property of composite numbers which fall into the category of having $gcd(N, p+q-1) > 1$. Testing this condition is done by calculating a multiple of $p+q-1$ and testing the GCD against the already known value of N.

$$z \quad = \quad \left(\sum_{i=1}^{k} r_i\right) \cdot \left(-1 + \sum_{i=1}^{k} (p_i + q_i)\right) \ mod\ N$$

To create this calculation, every party chooses a random value, r_i, and also calculates the sum $p_i + q_i$. Using the method that was described for sharing and combining the candidates chosen to create N, these same values are shared. Once the values are shared, the initiator can check to see if the test passes.

7.3.1 Step(4) Example

After testing to see if the initial Fermat test passes, the last condition is to check if $gcd(N, p + q - 1) > 1$. To do this, a simplified algorithm is explained by Boneh in which the condition is secretly calculated and shared amongst the players then checked by the initiator.

Player	r_i	$p_i + q_i$
1	839859	299 + 311
2	371743	424 + 400
3	120430	164 + 280

Table 6: Players choice of r_i, and $p_i + q_i$

Evaluating the function, it is found that $gcd(879017, 299716)$ is calculated and is 1 which confirms the suspicion that N is a composite of two primes (at least probabilistically).

8 Finding $e^{-1}\ mod\ \phi(N)$

After it is fairly certain that N is a composite of two primes, the last step is to calculate the multiplicative inverse of the encryption exponent. As an industry standard, the number 65537 is often used as a standard encryption exponent since it only has 2 set bits in the binary representation, thus simplifying the square-and-multiply method of modular exponentiation, although the algorithm used can find the inverse using any encryption exponent.

The Catalano method of computing inverses over a shared secret modulus creates some large numbers and trades complexity for network transmissions. In this case, the ease of the algorithm coupled with the decrease in network traffic was a formidable tradeoff.

In this method, we have the k-players all participating in the calculation. Each player starts with their input of the share of the secret modulus, ϕ_i (as described and used in the Fermat test), multiplied by a factor of $L = k!$. This is accomplished through a $\lfloor \frac{k}{2} \rfloor$-degree polynomial, $f(z)$, with the free term equal

to $L\phi$.

In the first round of the protocol, the players jointly generate two random $\lfloor \frac{k}{2} \rfloor$-degree polynomials, $g(z)$ and $h(z)$ with free terms $L\lambda$ and LR, respectively, along with a random k-degree polynomial, $\rho(z)$, with a free term of 0.

In the second round, they reconstruct the k-degree polynomial $F(z) = f(z) \cdot g(z) + e \cdot h(z) + \rho(z)$ and recover the free term $\gamma = F(0) = L^2\lambda\phi + LRe$.

Finally, the extended Euclidean algorithm is used to compute a and b such that $a\gamma + be = 1$. Each player, P_i, computes their share of d by setting $d_i = a \cdot h(i) + b$.

$$\begin{aligned} \gamma &= L^2\lambda\phi + LRe \\ a\gamma + be &= 1 \\ d &= (aLR + b) \ mod \ \phi \end{aligned}$$

This works because of the following observation.

$$\begin{aligned} a\gamma + be &= 1 \\ a(L^2\gamma\phi + LRe) + be &= 1 \\ aLRe + be &= 1 \ mod \ \phi \\ e(aLR + b) &= 1 \ mod \ \phi \\ d &= (aLR + b) \ mod \ \phi \end{aligned}$$

At the end of the protocol, every player then has the same values for a and b leaving their individual $h(i)$ values unique. Decryption of the ciphertext then becomes as simple as each player raising the original ciphertext to their respective $h(i)$ values and sharing them directly with each of their partners. Since the values of a and b are known throughout the group, each player can then combine the messages appropriately and yield the plaintext.

$$\begin{aligned} c_{LR} &= \prod_{i=1}^{k} c^{h(i)} \\ p &= c_{LR}^{a} \cdot c^{b} \ mod \ N \\ p &= c^{(aLR+b)} \ mod \ N \end{aligned}$$

8.1 Calculating d_i Example

In keeping with the example throughout the paper, consider the encryption exponent $e = 31$. First, each player must choose their values for λ_i and r_i where $R = \sum r_i$.

$Player_i$	ϕ_i	λ_i	r_i
1	878408	5823927	3829593
2	-824	983817	49298481
3	-444	57372759	2727639

Table 7: Players value of ϕ_i and choice of λ_i and r_i

Once combined through the BGW protocol using polynomials and solved, we can then calculate the value of $F(0) = \lambda = L^2\lambda\phi + LRe$.

$$\begin{aligned} \lambda &= \sum_{i=1}^{k} \lambda_i \\ \lambda &= 64180503 \end{aligned}$$

$$\begin{aligned} R &= \sum_{i=1}^{k} r_i \\ R &= 55855713 \end{aligned}$$

$$\begin{aligned} \gamma &= \lambda\phi + Re \\ \gamma &= (64180503 \cdot 877140) + (55855713 \cdot 31) \\ \gamma &= 56297017928523 \end{aligned}$$

At this point, the value of γ, which everyone has, is calculated. Finally, the values of a and b can be computed using the Extended Euclidean Algorithm to finish the secret key information.

$$\begin{aligned} a\gamma + be &= 1 \\ a \cdot 56297017928523 + b \cdot 31 &= 1 \end{aligned}$$

$$\begin{aligned} a &= -1 \\ b &= 1816032836404 \end{aligned}$$

At this point, the group creation is complete and a secure RSA group has been created. The only thing left to do, at this point, is to test a message.

9 Testing The Group

In testing the group, any member can take a message and encrypt it using the known encryption value of e. Once the plaintext message, p, is encrypted to ciphertext c, it can then be broadcast to every player. To decrypt, each player raises c to their secret r_i value chosen when finding the inverse of the modulus. Each player shares this information secretly between each other player. Once every secret has been applied to the c message, decryption is completed by applying the values of a and b also found in the last stage of the protocol.

For the example, using encryption exponent $e = 31$ and plaintext $p = 192$, the message is encrypted to be a value of 118121 *mod* 879017. Each player applies their own r_i value to ciphertext c as shown in the table.

Player	r_i	c^{r_i} *mod* N
1	3829593	630442
2	49298481	354970
3	2727639	586512

Table 8: Determining plaintext p from ciphertext c

$$\begin{aligned} c_R &= \prod_{i=1}^{k} c^{r_i} \ mod\ N \\ c_R &= 173591 \end{aligned}$$

The values of a and b are then applied in a way which satisfies the decryption exponent definition as stated before. It should be noted where numbers may be taken *mod* N and where they cannot since the real number definition is in terms of *mod* $\phi(N)$. The value of c_R may be taken in terms of *mod* N, but the values of a and b cannot since the values are raised to that power.

$$\begin{aligned} p &= c_R^a \cdot c^b \ mod\ N \\ p &= 173591^{-1} \cdot 118121^{1816032836404} \ mod\ 879017 \\ p &= 276292 \cdot 805843 \ mod\ 879017 \\ p &= 192 \end{aligned}$$

The group is tested and it is shown that the RSA group is secure without ever revealing the values of prime numbers p or q.

10 Software Architecture

The implementation of the k-Group RSA protocol leverages off of the `M2MI` middleware layer developed at the Rochester Institute of Technology Computer

Science Department. This layer allows for broadcast messages to efficiently be implemented along with direct one-to-one communication within the same programming interface.

Moreover, some extra classes were created for helping with the rather large numbers used within the protocol. Specifically, a `BigRational` class is used to express exact ratios as used within the protocol. One particular place where this is required is the Lagrange polynomial interpolation used to find the free term of random polynomials chosen for secret sharing.

10.1 Interface Programming

The Many-to-Many Invocation (`M2MI`) layer created at the Rochester Institute of Technology abstracts any remote method calls to appear as simple method calls on simple objects. Any and all marhsalling of data objects as well as communication channel considerations are handled by the `M2MI` layer.

For this to be successful, `M2MI` requires that interfaces be predefined for method execution. This is advantageous for a few different reasons, the first being that any implementation can be used as long as it adheres to the interface rules. For example, one implementation for a desktop computer might not be suitable for a mobile or embedded device, yet they are still able to talk to each other because the interface is well defined. Second, it allows for the software architecture to explore the different use cases of the design without requiring a strict implementation. Lastly, and especially for the case of this project, it allows for the programmer to abstract the channel down a layer and not have to necessarily worry about it. For example, this project requires that all individual communication that occurs between objects has to be in a secure channel. Since this is outside of the scope of the project, this was not implemented, but extending this capability is easily done by creating a `SecureUnihandle` object for the `M2MI` layer.

10.2 BigRational Class

Many of the operations that occur during the protocol require accurate representations of very large numbers. Moreover, manipulations of those numbers often require them to be represented as a ratio of two extremely large numbers. Due to this fact, a class was specifically made to aide in the manipulation of these ratios while maintaining exact accuracy.

10.3 Polynomial Class

When Shamir explained his secret sharing scheme, it involved the usage of polynomials of varying degree being evaluated at multiple values while using

```
class BigRational implements Serializable {
  BigRational( )
  BigRational( BigInteger num )
  BigRational( BigInteger num, BigInteger denom )
  BigRational( int num, int denom )
  BigRational( BigRational x )

  BigRational add( BigRational x )
  BigRational multiply( BigRational x )
  BigRational multiply( long x )
  BigRational pow( int power )
  BigRational divide( BigRational x )
  BigRational[] divideEvenly( int n )
  int compareTo( BigRational x )
  BigRational[] divideRandomly( Random rnd, int n )
  static BigRational next( Random rnd, int numBits )
  BigRational abs( )
  String toString( )
  static BigRational valueOf( int x )
  BigInteger intPart( )
  BigRational negate( )
}
```

Figure 3: BigRational Class Public Methods

Lagrange interpolation to calculate the free term. Boneh leverages off these random polynomials with known free terms as well. The `Polynomial` class is used for both creation and evaluation of the required polynomials.

```
class Polynomial {
  Polynomial( Random rnd, int degree )
  Polynomial( Random rnd, int degree, BigRational c )
  Polynomial( Random rnd, int degree, BigInteger c )

  BigRational f( int x )
  BigRational f( BigRational x )
  BigRational f( BigInteger x )
  String toString( )
}
```

Figure 4: Polynomial Class Public Methods

10.4 Lagrange Class

The use of polynomials within the protocol helps with sharing secrets securely between different members of the group. To do this effectively, an interpolation technique must be used to solve for a specific term of a k-degree polynomial using at least $k + 1$ points, where k is the size of the group used within the protocol. Lagrange interpolation is a method of calculating arbitrary points on an unknown polynomial when only a mapping of x, y values are known.

10.5 BonehRSA Interface

The `BonehRSA` interface is the collection of methods which allow for the execution of the protocol as explained by Boneh. The structure of the interface generally has a method to start a particular part of the protocol which is broadcast to every object participating within the group accompanied by a method which is

```
final class Lagrange {
  static BigRational rationalPointsInterpolation(
                          BigRational x,
                          Set<Map.Entry<BigRational,BigRational>> xys )
  static BigRational integerPointsInterpolation(
                          BigRational x,
                          Set<Map.Entry<BigInteger,BigRational>> xys )
}
```

Figure 5: Lagrange Class Public Methods

used to specifically share data between pairs of objects.

```
interface BonehRSA {
  void announceGroup( BonehRSA initiator, String groupName ) ;
  void joinGroup( String groupName, BonehRSA client ) ;
  void assignMemberID( String groupName, int ID ) ;
  void rejectJoinGroup( String groupName ) ;
  void distributeMemberList( String groupName, ConcurrentHashMap<BonehRSA, BigInteger> groupHandle) ;
  void chooseCandidates( ) ;
  void operationComplete( BonehRSA client, BonehRSAState state ) ;
  void shareCandidates( ) ;
  void shareCandidate( BonehRSA client, BigRational pEval, BigRational qEval, BigRational hEval ) ;
  void ensureCandidatesShared( ) ;
  void shareShare( BonehRSA client, BigRational nPart ) ;
  void shareShares( ) ;
  void ensureSharesShared( ) ;
  void calculateN( ) ;
  void calculateFermatValues( BigInteger g ) ;
  void shareFermatValues( ) ;
  void shareFermatValue( BonehRSA client, BigRational x ) ;
  void ensureFermatValuesShared( ) ;
  void shareFermatProducts( ) ;
  void shareFermatProduct( BonehRSA client, BigRational x ) ;
  void chooseZValues( ) ;
  void shareZValues( ) ;
  void shareZValue( BonehRSA client, BigRational rPart, BigRational pqPart ) ;
  void ensureZValuesShared( ) ;
  void reportZi( ) ;
  void sendZi( BonehRSA client, BigRational zi ) ;
  void reinitialize( BonehRSA client ) ;
  void chooseCatalanValues( ) ;
  void shareCatalanValues( ) ;
  void shareCatalanValue( BonehRSA client, BigRational fzValue, BigRational gzValue, BigRational hzValue, BigRational rhozValue ) ;
  void ensureCatalanValuesShared( ) ;
  void calculateCatalanProducts( ) ;
  void shareCatalanProducts( ) ;
  void shareCatalanProduct( BonehRSA client, BigRational cValue ) ;
  void ensureCatalanProductsShared( ) ;
  void calculateD( ) ;
  void decrypt( BigInteger ct ) ;
  void shareMessage( BonehRSA client, BigInteger encryptedMsg, BigInteger decryptedMsg ) ;
}
```

Figure 6: BonehRSA Public Interface

10.6 BonehRSAState Enumeration

Every object that implements the `BonehRSA` interface also follows a basic state enumeration which is defined by the `BonehRSAState` enumeration. The state reported by each object is the current state within the protocol that the object is ready to perform. Each state corresponds to a method which is defined within the `BonehRSA` interface. The purpose of this enumeration is to keep track of where in the protocol the particular object is and be able to report it back to the initiator object. Lastly, upon each state update for each object, the current state is reported back to the initiator. Once an object has reached the `D_CALCULATED` state, the object has been initialized and is ready for encrypting and decrypting messages passed to it.

```
public enum BonehRSAState {
  UNINITIALIZED,
  MEMBERLIST_RECEIVED,
  CANDIDATES_CHOSEN,
  CANDIDATES_SHARED,
  CANDIDATES_ENSURED,
  SHARES_SHARED,
  SHARES_ENSURED,
  N_CALCULATED,
  FERMAT_CALCULATED,
  FERMAT_VALUES_SHARED,
  FERMAT_VALUES_ENSURED,
  FERMAT_PRODUCTS_SHARED,
  Z_VALUES_CHOSEN,
  Z_VALUES_SHARED,
  Z_VALUES_ENSURED,
  ZI_REPORTED,
  CATALAN_VALUES_CHOSEN,
  CATALAN_VALUES_SHARED,
  CATALAN_VALUES_ENSURED,
  CATALAN_PRODUCTS_CALCULATED,
  CATALAN_PRODUCTS_SHARED,
  CATALAN_PRODUCTS_ENSURED,
  D_CALCULATED
} ;
```

Figure 7: BonehRSAState Enumeration

10.7 BonehObject

The `BonehObject` implements the `BonehRSA` interface and is the main object of interest throughout the implmentation. Most of the methods which belong to the `BonehObject` are that of the `BonehRSA` interface. To configure the object as an initiator, the `setName()` method must be used to set the group name as well as `setSize()` method to specify how many members the group needs. Lastly, the `runGroup()` method is called to run the protocol. This sends out the request for other members to join the group. The only configuration required to be considered for group membership is the creation of a new `BonehObject`. All `BonehObject`s automatically listen for all incoming group requests and try to join if they are not already a member of a group.

10.8 Setting Up The Groups

When setting up a number of participants, only an instance of the `BonehObject` is required to be created before a group can start to be formed. Objects inherently want to join a group as long as there is a request being sent.

Since the Boneh protocol depends on one object being the initiator of the group and all the other objects are simply orchestrated by the initiator, setting up a group requires $k-1$ listening nodes and 1 initiator. To setup a listening node, a new instantiation of the `BonehObject` is required. A name for the node is required to be given in the constructor, while a seed for all random number generation is optional. Currently, this listening object will respond and join the first group that it hears from.

```
BonehObject bo = new BonehObject( "Listener0", 12345678L ) ;
```

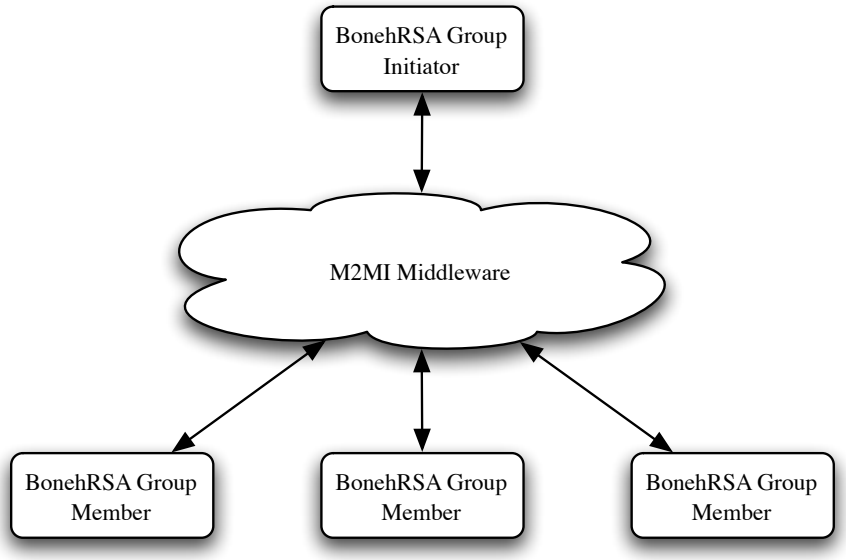

Figure 8: Initiator Using M2MI for Group Communication

To create an initiator, a few extra steps are required. First, a group name is required to keep track of different groups. This is setup using the `setGroupName()` method. Next, the minimum number of bits for the modulus N is required. The only guarantee given is that the modulus will be *at least* the number specified. This is specified using the `setNumBits()` method. Next, the size of the group needs to be specified. This also corresponds to the number of different shares that are produced for the group and is set using the `setNumberOfShares()` method.

```
BonehObject bo = new BonehObject( "Initiator" ) ;
bo.setGroupName( "SecureGroup" ) ;
bo.setNumBits( 1024 ) ;
bo.setNumberOfShares( 5 ) ;
bo.runGroup( ) ;
```

Figure 9: Example Setting Up The Initiator

Listeners don't necessarily have to be on a different machine as the initiator is, and in fact a small program can be created which runs on a single computer to run trial tests.

```
private static int performTest( int k, int numBits ) {
    BonehObject[] bos = new ObjectObject[k] ;
    for( int i = 0 ; i < k ; i++ )
        bos[i] = new BonehObject( "" + i ) ;
    bos[0].setGroupName( "TestGroup" ) ;
    bos[0].setNumBits( numBits ) ;
    bos[0].setNumberOfShares( k ) ;
    bos[0].runGroup( ) ;
}
```

Figure 10: Test method for Group Creation

10.9 Running The Protocol

The main protocol operates in a command relationship between the initiator of the group and the members of the group. The initiator sends out method requests for the different portions of the algorithm all while the members of the group report back their current state to the initiator.

Once all the required steps are performed for the initiator, the last thing to do is run the group. A blocking call to the `runGroup()` method can be performed which, once returned from, will guarantee the object has been setup and the group created can be called, or a new `Thread` can be created on the object and polled until the current status of the object is `BonehRSAState.D_CALCULATED`.

10.10 Creating the Group

During the protocol, a group initiator first announces a group of size k. Each `BonehObject` which chooses to join sends a request to join the group. If accepted into the group, the initiator sends a response with an associated ID. If rejected, the initiator sends an appropriate rejection response.

Once k is reached, the initiator sends out a memberlist mapping of {`M2MI.Unihandle`,ID}. This sets up the polynomial values to be evaluated for each group member. It should be noted here that the values of the ID do not matter except: no ID of 0 is ever assigned, the ID of 1 designates the initiator of the group, and no ID values can be duplicated within the map.

The initiator, at this point, uses the methods as defined within the `BonehRSA` interface to go through the different algorithm steps. For synchronization, a mapping of {`M2MI.Unihandle`,`BonehRSAState`} is kept by the initiator and updated by the group objects after each algorithm step is performed. The initiator

continues once all the group objects report their updated status.

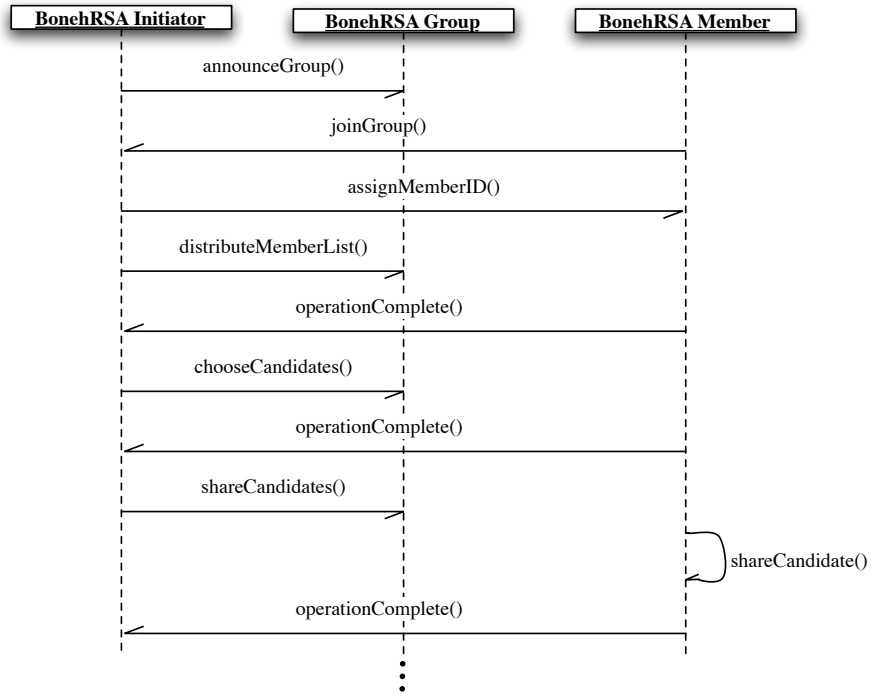

Figure 11: Showing Initiator Relationship with Other Objects

Information regarding the newly created group can be accessed by any of the objects within the group using any of the accessors such as `getCount()` for the number of rounds that were required to create the group or `getCalculatedN()` for the modulus N.

Sending a message once the group is created is accomplished by using the `sendMessage()` method. This method accepts a `BigInteger` as the message to be sent, so any type of padding or encoding has to be done before sending anything to this method. Once all the pieces are received from each of the other members, the original message is printed to the console.

11 Results

After implementing the aforementioned algorithm, a test was run for creating 50 different groups of size $k = 3$, $k = 5$, $k = 10$, and $k = 20$. Modulus lengths

of size 64, 128, 256, 512, 1024, 2048 and 4096-bits were all created and tested.

11.1 Expected

Before any analysis can be done, an expected curve has to be created to be a baseline for all the results. To do this, the probability of finding two prime numbers must first be defined. This is accomplished by first understanding that the number of prime numbers grows, as a generalization, at a rate of $\frac{x}{ln(x)}$. This function is defined as $\pi(x)$. Since the project requires all primes to be Blum integers, and assuming the distribution of prime numbers is equal between numbers that are 1 mod 4 and 3 mod 4, this leaves $\frac{\pi(x)}{2}$ prime numbers to choose from.

In the project software implementation, given a length of N, the length of p is chosen to be of size $\frac{log_2(N)}{2} - 3$ and q is chosen to be of size $\frac{log_2(N)}{2} + 3$. More explanation needed here.

Finding the probability of choosing a prime number is the ratio of prime numbers to composite numbers. Moreover, because the experiments are run as powers of 2, further reduction is possible.

$$\begin{aligned} P(x) &= \frac{\frac{\pi(x)}{2}}{\frac{x}{4}} \\ P(x) &= \frac{2}{ln(x)} \\ P(2^x) &= \frac{2}{x \cdot ln(2)} \end{aligned}$$

Next, an equality must be setup to find the average number of times that it can be expected the round does not produce a set of two primes.

$$\begin{aligned} P(2^{log_2(N)}) &= 1 - \frac{2}{(\frac{log_2(N)}{2} - 3) \cdot (\frac{log_2(N)}{2} - 3)} \\ P(2^{log_2(N)})^x &\leq 0.50 \\ x &> \frac{ln(0.50)}{ln(P(2^{log_2(N)}))} \end{aligned}$$

Solving for x gives the expected results for each of the different sizes of N.

11.2 Experimental

Once the expected numbers are calculated, test trials were run for group sizes of 3, 5, 10 and 20 all with different modulus sizes of 64, 128, 256, 512, 1024, 2048 and 4096-bits.

	Group Size (k)			
$log_2(N)$	3	5	10	20
64	147.74	111.64	128.08	173.58
128	544.44	533.04	532.40	623.36
256	1988.46	2155.46	1958.54	2200.10
512	8232.50	8265.52	7931.18	8211.50
1024	24080.70	21684.76	25097.68	23156.24
2048	57002.20	36138.90	41712.90	43813.92
4096	75041.76	64210.38	67786.30	58203.86

Table 9: Average Rounds To Create A k-sized RSA Group

It is seen that the expected line is slightly less than the experimental results. This suggests that some valid composite numbers which are valid for RSA are actually failing the probabilistic biprimality tests. Moreover, there is a curve that occurs within the data for large numbers, deviating from the expected values. If the total number of rounds are weighted by the number of *successful* group creations as opposed to the total number of group creations, the deviation is significantly less. This suggests that, for these larger numbers, the probability of a false positive on the biprimality tests is relatively high.

This deviation can be remedied by weighting the average number of group creations by the success rate of the group creation process.

The larger the N, the larger the deviation from the group simply due to the low group creation success rate. For the case of $log_2(N) = 4096$, the overall success rate was on the order of 15%.

$log_2(N)$	Average	Rounds/Bit	% Successful	Normalized
64	140.26	2.1916	100.0	2.1916
128	558.31	4.3618	99.5	4.3837
256	2075.64	8.1080	97.5	8.3159
512	8160.18	15.9378	94.5	16.8654
1024	23504.85	22.9540	72.0	31.8805
2048	44666.98	21.8100	45.0	48.4333
4096	66079.96	16.1328	15.0	106.2105

Table 10: Total Statistics

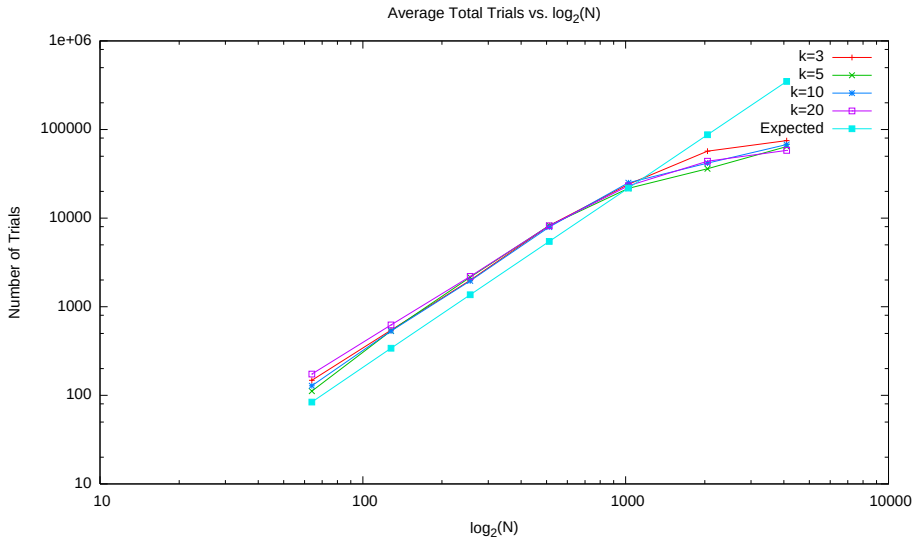

Figure 12: Average Number of Trails Total

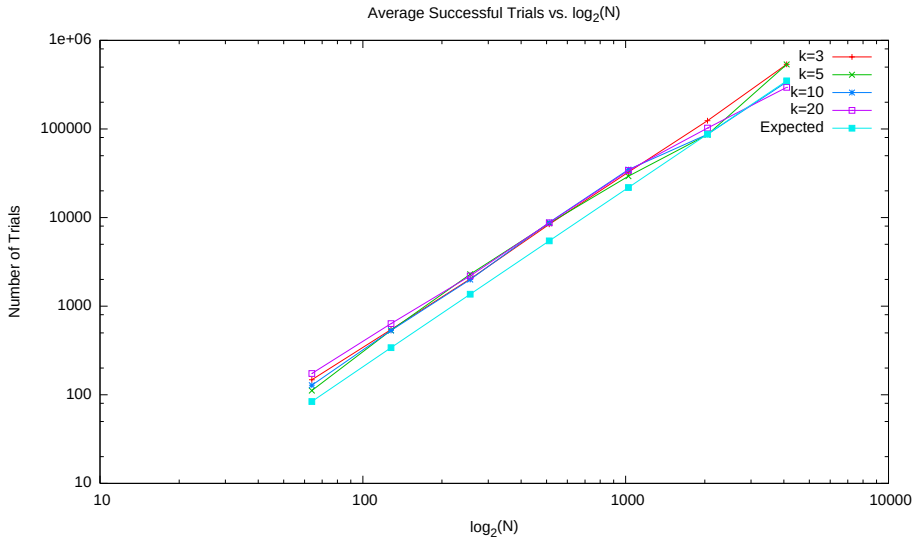

Figure 13: Average Number of Trails Successful

Though the group creation process may return a false positive, for very large numbers, and the average number of rounds required to successfully create a group may be significantly large, the histograms of all group creations (both

successful and unsuccessful) show that most groups are created well within the expected average for the larger numbers. These histograms and cumulative percentages are shown in the Appendices.

The results show a linearly increasing average number of rounds required to build groups with larger modulus numbers. This is on par with the exponential increase of prime numbers to composite numbers ratio as defined by the prime number function $\pi(x)$ and approximated by $\frac{x}{ln(x)}$.

Unfortunately, the results also show a large variance in the number of rounds required to create a group of a specific size. In a mobile ad-hoc network, this is simply unacceptable performance and cannot be considered feasible to perform the operations as described in this paper. Fortunately, in networks where reliability and network connectivity is relatively high, the setup time could almost be considered insignificant versus the security gained through parties that only have semi-trusting relationships.

Lastly, it can be seen in the calculation of $e^{-1}\ mod\ \phi(N)$ section of the algorithm that the values of a and b, found through use of the Extended Euclidean Algorithm, have the possibility of being very large numbers which cannot be reduced due to the fact that $\phi(N)$ is unknown. This can cause for a very large amount of computational overhead when decrypting any message sent though the technique described. Considering that mobile ad-hoc networks are usually of the embedded type of computer and extremely limited in the amount of computing power at the disposal of the machine, performing this operation many times is not very feasible.

12 Conclusion

Presented was a distributed public-key creation system for ad-hoc groups based on a traditional RSA system. The system presented was able to create these secure groups with a 1024-bit RSA modulus within an average of around 32689 rounds. Unfortunately, there is a very large variance and undetermined amount of time, due to the independent choice of random variables for the candidacy of p and q, associated with the algorithm which yields the algorithm impractical for real world use in mobile ad-hoc networks due to the volatile nature of such networks.

On the other hand, but the algorithm may show itself useful in other networks where the setup time for a 1024-bit of 2048-bit modulus is significantly less than the overall lifetime of the network. In these cases, though the setup time may be long, the generation of a secure group given a semi-trusting relationship with each of the players makes sense.

In the end, the algorithm described proved useful when dealing with networks that have a high connection reliability and are non-volatile where the computation and real time overhead required for creating the secure RSA group is insignificant when compared to the amount of time the group will be established. Moreover, the complexity and the number of rounds required for the creation of the secure RSA group is not feasible for the current set of embedded systems used in mobile ad-hoc networks today, but with an invention of a hardware accelerated RSA engine to perform the large modular exponentiation and improved battery life for longer radio communication, this solution can be revisited as a way to create secure groups between semi-trusted parties in less volatile mobile ad-hoc networks.

13 Future Work

Though the testing was successful in creating a secure RSA group using the protocol as defined by Boneh, there are still some outstanding questions as well as implementation details that were not within the scope of this project but still are required to fully implement the secure RSA group.

First, a secure M2MI `Unihandle` implementation is required to provide appropriate security during the protocol. Once implemented, adding the functionality to the current software functionality should be relatively trivial and should not change any of the underlying architecture.

Second, the peculiar deviation in the average number of rounds required to create a group is not understood. Since this deviation occurs at larger lengths of N, it is definitely worth investigating to see if there is an upper bound to the number of rounds required to find a valid modulus and a reason why, for these larger numbers, the failure rate of the probabilistic biprimality tests fail.

Along the same line of thought, because the distribution of the number of rounds required for group creation is an exponential, the significant majority of groups are created significantly less than the maximum outliers. It is worth investigating whether resetting the pseudo-random number generator after a specified count may be worth while in achieving a secure group quicker without having to fully create all groups.

Next, all objects are setup to run as quickly as possible without any user interaction. All messages being passed back and forth are automatically assumed correct and are never passed above the `BonehObject` object. This was done to facilitate a very fast group setup time and avoid any and all user interaction. Unfortunately, this is not desirable when deployed so a `MessageSink` will be necessary to act as a control channel allowing a supervisory object to control group creations, joining, message sending and message receiving.

Moreover, there is no way to store the group configuration or private key information to retrieve it later. Once any member of the group leaves, it is assumed the group is disbanded and a new group must then be created.

Lastly, measures need to be taken to tolerate deviant members within the group creation process. Boneh alluded to the fact that his method can be tolerant if certain measures are taken but might be susceptible in the trivial case.

14 References

1. Ben-Or, M., Goldwasser, S., and Wigderson, A. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing* (Chicago, Ill., May 2-4). ACM, New York, pp. 1-10.
2. Benaloh (Cohen), J. 1987. Secret sharing homomorphisms: keeping shares of a secret secret. *Advances in Cryptology - Crypto '86*. Lecure Notes in Computer Science, vol. 263. Springer-Verlag, New York, pp. 251-260.
3. Boneh, D., Franklin, M. 2001. Efficient generation of shared RSA keys. *Proceedings Crypto '97*. Lecture Notes in Computer Science, vol. 1233. Springer-Verlag, New York, pp. 425-439.
4. Catalano D, Gennaro R, Halevi S. Computing Inverses over A Shared Secret Modulus. *Advances in Cryptology - EUROCRYPT 2000*. Lecture Notes in Computer Science, vol. 1807. Springer-Verlag, New York, pp. 190-206.
5. Malkin M, Wu T, Boneh D. Experimenting with shared generation of RSA keys. *Internet society's symposium on network and distributed system security (SNDSS)*, 1999; pp. 43-56.
6. Shamir, A. 1979. How to share a secret. *Commun. ACM 22*, 11 (Nov.), 612-613.

Appendix A: GroupTest.java

```
import edu.rit.m2mi.M2MI ;

/**
 * Test program that will run a number of times through the group creation
 * process and print out an average of the number of rounds required
 * to create the group.
 *
 * @author    Brian Padalino
 */
public class GroupTest {

    /**
     * Perform the test for <i>n</i> iterations and return the average
     * number of iterations for group creation.
     *
     * @param   n   The number of times to perform the test.
     * @return      The average number of rounds required to run the test.
     */
    private static int performTest( int n, int k, int numbits ) {
        int average = 0 ;
        for( int i = 0 ; i < n ; i++ ) {
            System.out.println( "Starting test #" + i ) ;
            BonehObject[] objects = new BonehObject[k] ;
            for( int j = 0 ; j < objects.length ; j++ ) {
                objects[j] = new BonehObject( "" + j,
                                (long)((i+1)*(j+1)*n*k*numbits) ) ;
                objects[j].setNumBits( numbits ) ;
            }
            objects[0].setGroupName( "Group" ) ;
            objects[0].setNumberOfShares(k) ;
            objects[0].runGroup() ;
            System.out.println( "  Count: " + objects[0].getCount()) ;
            average += objects[0].getCount() ;
        }
        return average / n ;
    }

    /**
     * Run the tests for 512, 1024, 2048 and 4096 bits for a number of
     * tests and print out the average.
     *
     * @param   args    Arguments that actually don't do anything.
     */
    public static void main( String args[] ) {

        int average = 0 ;

        M2MI.initialize() ;

        for( int i = 64 ; i <= 4096 ; i=i*2 ) {
            average = 0 ;
            System.out.println( "Running tests for " + i + " bits") ;
            average = performTest( 25, 5, i ) ;
            System.out.println( "Average: " + average ) ;
        }
    }

}
```

Figure 14: GroupTest.java Testing Program

This program was used to generate the data used for the trial runs.

Appendix B: $log_2(N) = 64$ Trial Runs

Data taken using the `GroupTest` class as written in Appendix A.

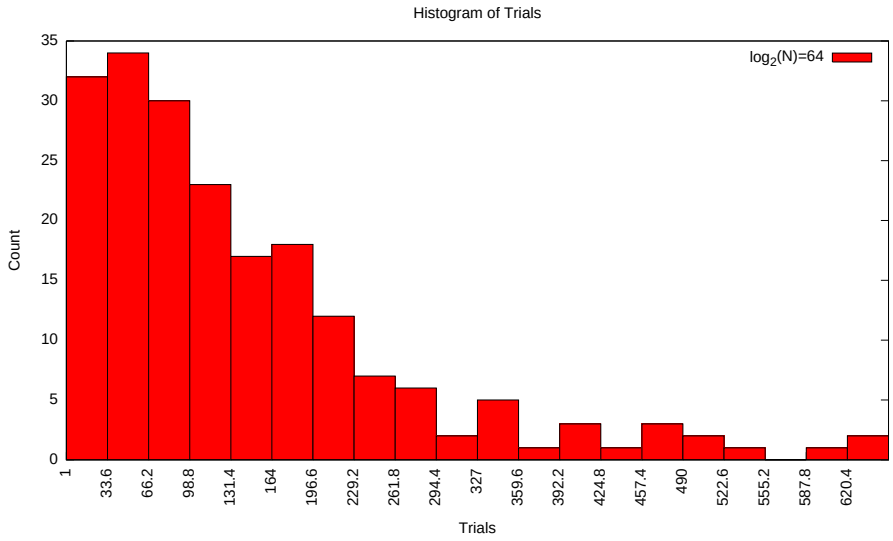

Figure 15: Histogram $log_2(N) = 64$

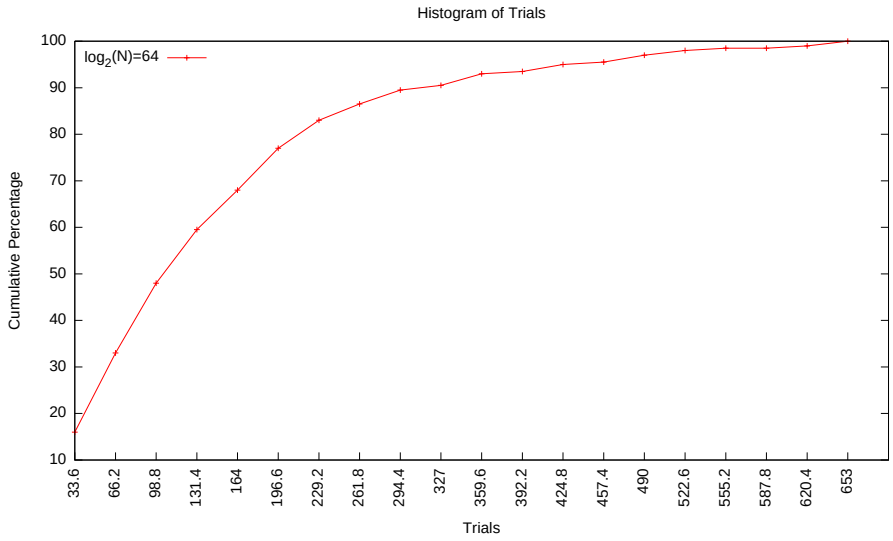

Figure 16: Cumulative Percentage $log_2(N) = 64$

Appendix C: $log_2(N) = 128$ Trial Runs

Data taken using the `GroupTest` class as written in Appendix A.

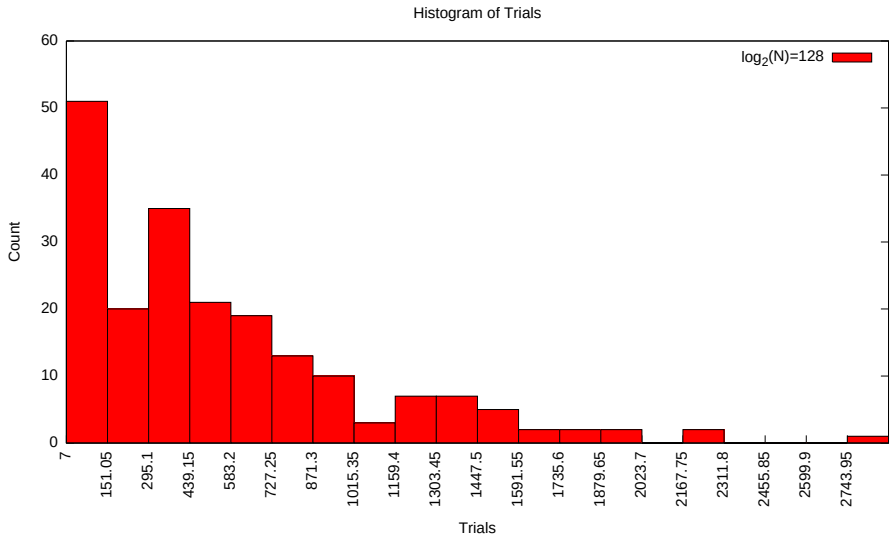

Figure 17: Histogram $log_2(N) = 128$

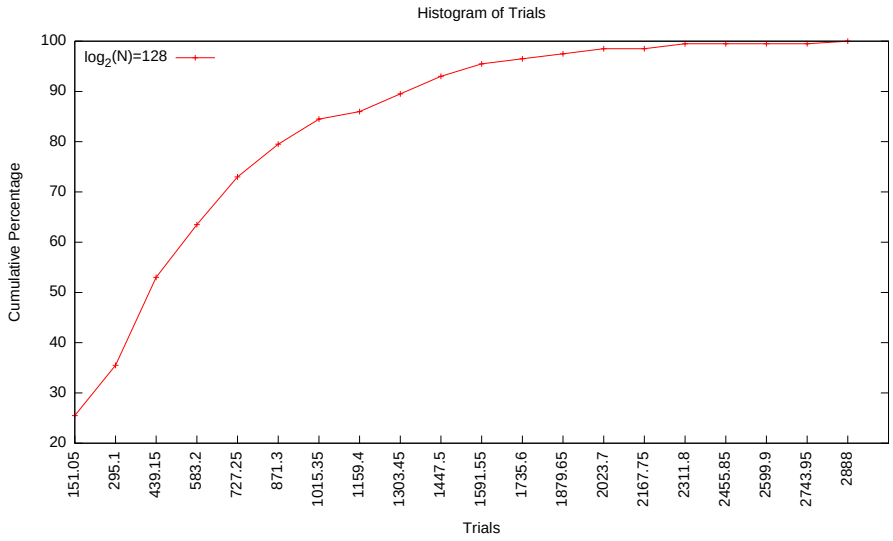

Figure 18: Cumulative Percentage $log_2(N) = 128$

Appendix D: $log_2(N) = 256$ Trial Runs

Data taken using the `GroupTest` class as written in Appendix A.

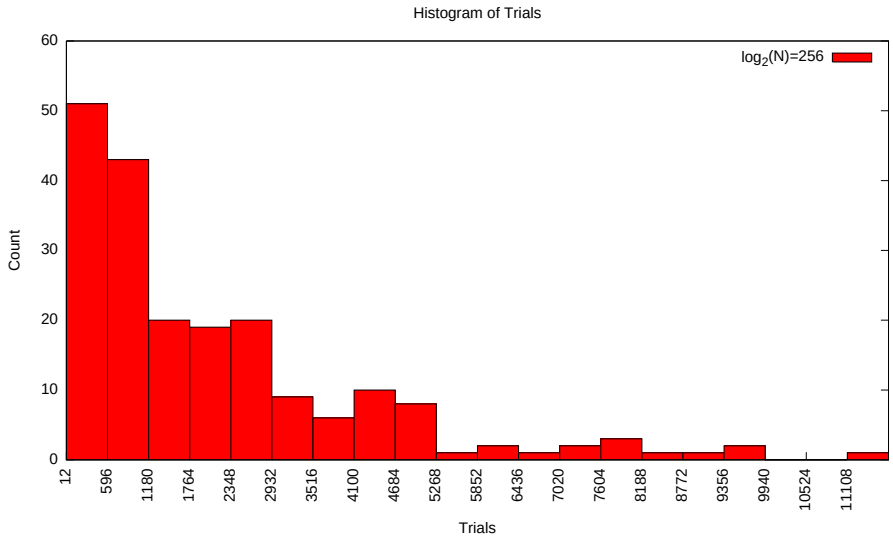

Figure 19: Histogram $log_2(N) = 256$

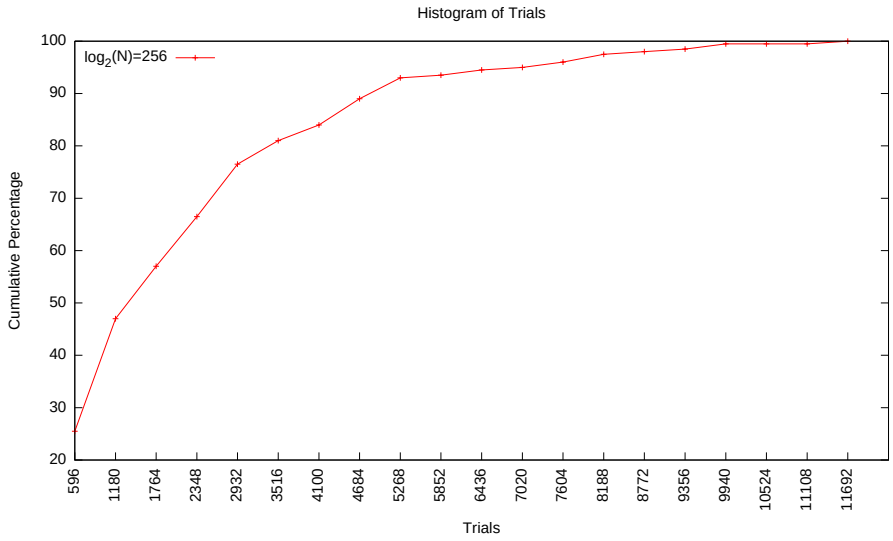

Figure 20: Cumulative Percentage $log_2(N) = 256$

Appendix E: $log_2(N) = 512$ Trial Runs

Data taken using the `GroupTest` class as written in Appendix A.

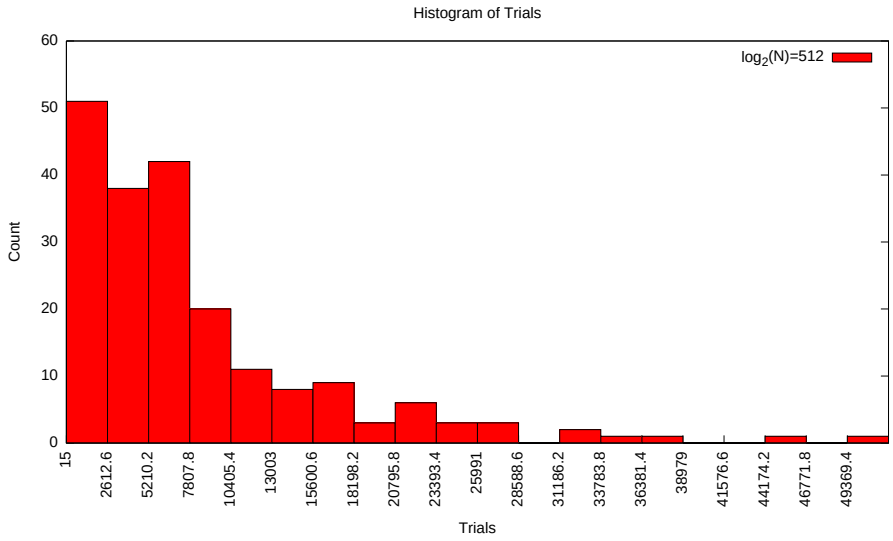

Figure 21: Histogram $log_2(N) = 512$

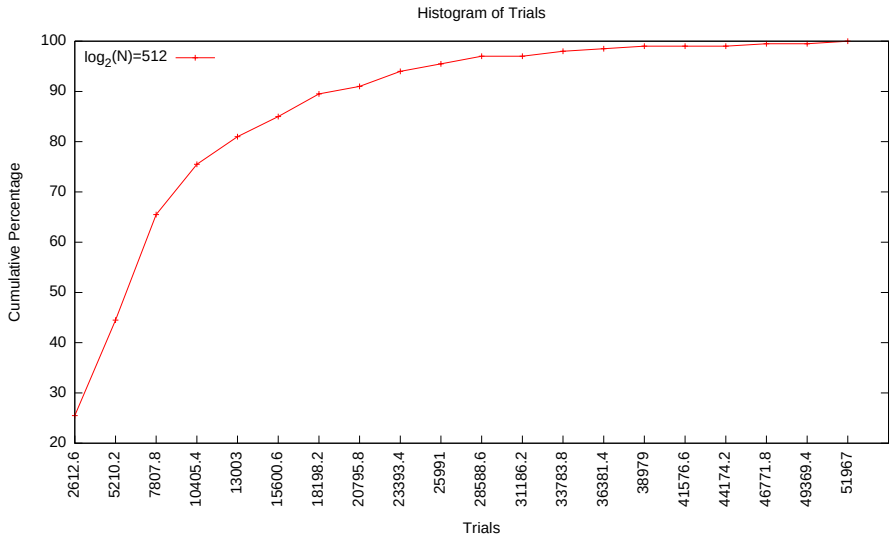

Figure 22: Cumulative Percentage $log_2(N) = 512$

Appendix F: $log_2(N) = 1024$ Trial Runs

Data taken using the `GroupTest` class as written in Appendix A.

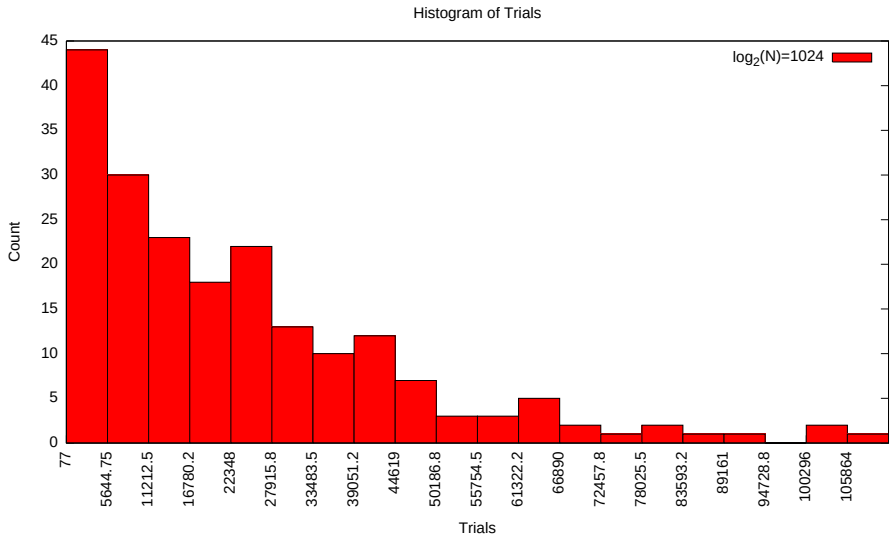

Figure 23: Histogram $log_2(N) = 1024$

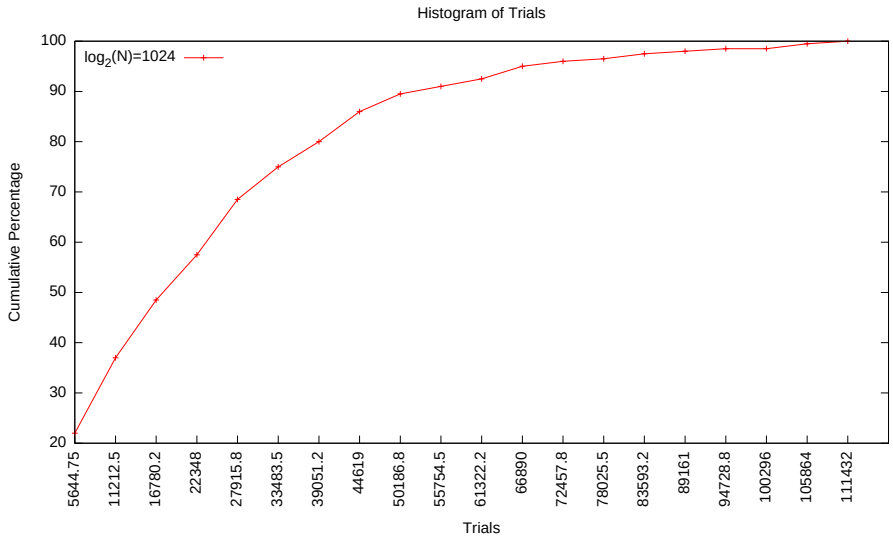

Figure 24: Cumulative Percentage $log_2(N) = 1024$

Appendix G: $log_2(N) = 2048$ Trial Runs

Data taken using the `GroupTest` class as written in Appendix A.

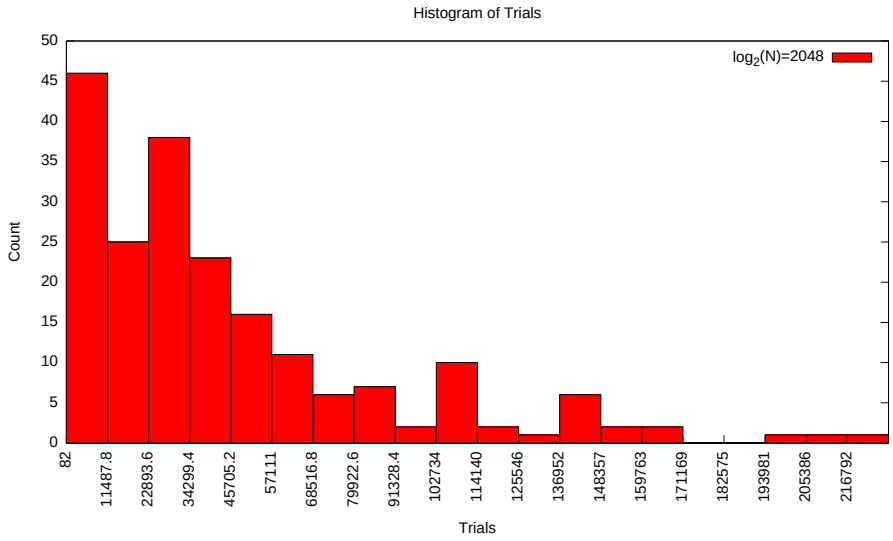

Figure 25: Histogram $log_2(N) = 2048$

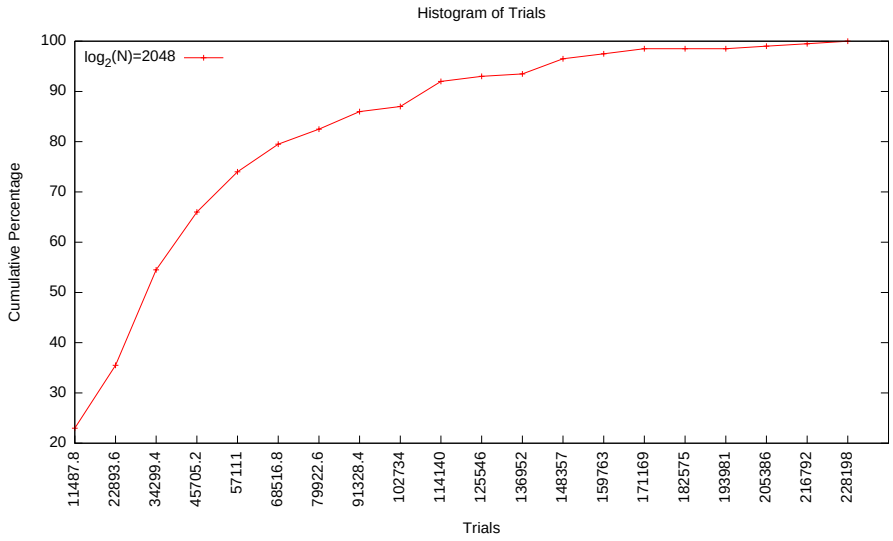

Figure 26: Cumulative Percentage $log_2(N) = 2048$

Appendix H: $log_2(N) = 4096$ Trial Runs

Data taken using the `GroupTest` class as written in Appendix A.

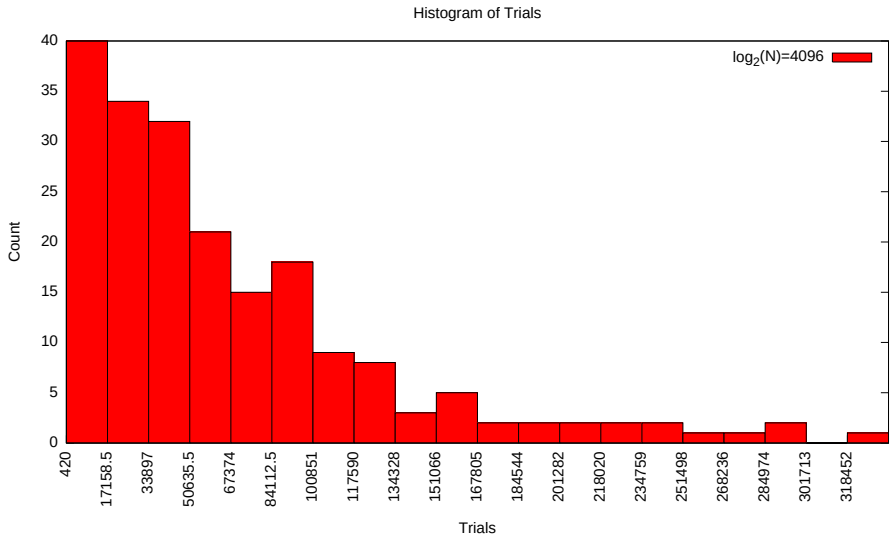

Figure 27: Histogram $log_2(N) = 4096$

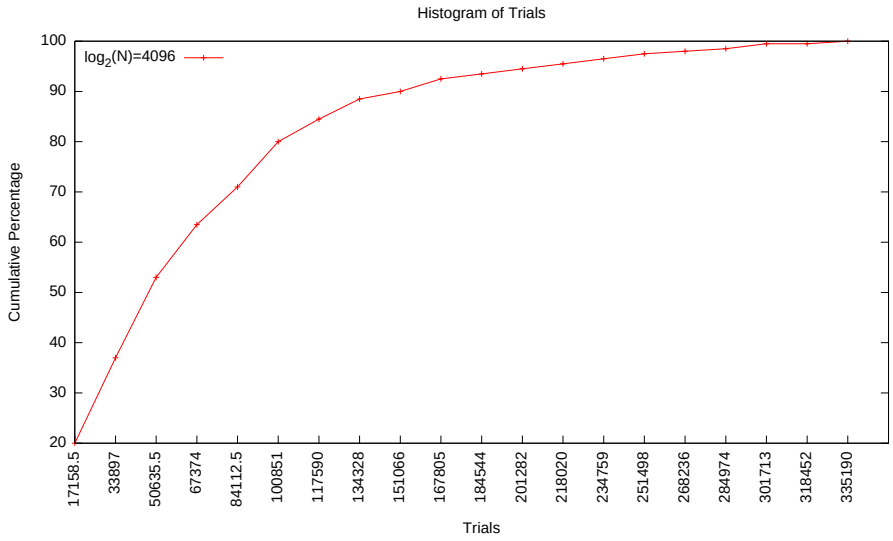

Figure 28: Cumulative Percentage $log_2(N) = 4096$