5-1-2018

The Key to Cryptography: The RSA Algorithm

Clifton Paul Robinson

The Key to Cryptography: The RSA Algorithm

Clifton Paul Robinson

Submitted in Partial Completion of the
Requirements for Commonwealth Interdisciplinary Honors
in Computer Science and Mathematics

Bridgewater State University

May 1, 2018

Dr. Jacqueline Anderson Thesis Co-Advisor
Dr. Michael Black, Thesis Co-Advisor
Dr. Ward Heilman, Committee Member
Dr. Haleh Khojasteh, Committee Member

BRIDGEWATER STATE UNIVERSITY

UNDERGRADUATE THESIS

The Key To Cryptography: The RSA Algorithm

Author:

Clifton Paul ROBINSON

Advisors:

Dr. Jackie ANDERSON

Dr. Michael BLACK

Submitted in Partial Completion of the Requirements

for Commonwealth Honors in Computer Science and Mathematics

Dr. Ward Heilman, *Reading Committee*

Dr. Haleh Khojasteh, *Reading Committee*

Dedicated to

Mom, Dad, James, and Mimi

Contents

BRIDGEWATER STATE UNIVERSITY

Abstract

The Key To Cryptography: The RSA Algorithm

Cryptography is the study of codes, as well as the art of writing and solving them. It has been a growing area of study for the past 40 years. Now that most information is sent and received through the internet, people need ways to protect what they send. Some of the most commonly used cryptosystems today include a public key. Some public keys are based around using two large, random prime numbers combined together to help encrypt messages.

The purpose of this project was to test the strength of the RSA cryptosystem public key. This public key is created by taking the product of two large prime numbers. We needed to find a way to factor this number and see how long it would take to factor it. So we coded several factoring algorithms to test this. The algorithms that were implemented to factor are Trial Division, Pollard's Rho, and the Quadratic Sieve. Using these algorithms we were able to find the threshold for decrypting large prime numbers used in Cryptography.

Introduction

1.1 The Project Overview

In public-key encryption systems, if someone is able to break the public key they can fully decrypt the message. The main goal of this project was to test the threshold of the RSA public key. We wanted to test the semiprime number public-key against different factoring algorithms to find the threshold of RSA on a regular computer.

The way we tested this was by implementing the different algorithms into several different computer languages. Then we would collect data and compare the algorithms against each other and from that find the threshold of RSA compared to the specific algorithms.

Section two contains all of the theorems and definitions needed for this research. This gives a general overview of what one will see throughout the paper as well as some mathematic background that is based on prime numbers and factoring.

Section three is about the history of Cryptography. It covers several main topics that helped define this subject, such as how it started and how it evolved over the years. We will also show the creation and use of the RSA cryptosystem because of the importance to this research.

All of the math is explained in section four, which covers prime numbers and general factoring, the two main parts of this research. It also shows the math behind the algorithms and how they work when they are factoring numbers.

Section five quickly outlines all of the programming languages used throughout the research. Sections six and seven cover the implementation of the algorithms as well as the observations taken from the data. Section seven displays all of the graphs that were created from the time measurements.

Lastly, in sections eight and nine we talk about improvements that can be made to the code in the future as well as draw a conclusion from the research. In addition, we discuss potential risks to public-key encryption in the future, especially with respect to quantum computing.

Theorems and Definitions

2.1 Definitions

- **Algorithm**
  - An algorithm is a well-defined procedure that allows a computer to solve a problem.
- **Carmichael Number**
  - A Carmichael number is an odd composite number which passes the Fermat Primality Test for every base, b, that is relatively prime to that number. These numbers can often be confused with prime numbers in factoring.
- **Cipher**
  - Also known as a cryptographic algorithm, is a mathematical function which uses plaintext as the input and produces CIPHERTEXT as the output and vice versa.
- **CIPHERTEXT**
  - The encrypted text or message.
- **Continued Fraction**
  - A continued fraction is a fraction created by an iterative process that can possibly be infinite of the form:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots}}}$$

    Where a_0 is any integer and a_i is a positive integer for $i \geq 1$.

- **Cryptography**
  - The art of writing or solving codes.
- **Cryptology**
  - The science and art of making and breaking codes and ciphers.
- **Cryptosystem**
  - A suite of cryptographic algorithms needed to implement a particular security service, most commonly for achieving confidentiality (encryption). Typically, a cryptosystem consists of three algorithms: one for key generation, one for encryption, and one for decryption.
- **Decryption**
  - The process of taking encoded or encrypted text or other data and converting it back into text that can be read and understood. (*Terms and Definitions*)
- **Encryption**
  - The process of converting information or data into a code, especially to prevent unauthorized access. (*Terms and Definitions*)
- **Fermat Primality Test**
  - This is a test created by the mathematician Fermat to determine whether a number is a probable prime. By Fermat's Little Theorem, if p is a probable prime and a is not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$. With this test, we will pick all a's not divisible by p to see if it is a prime number.
- **Modular Arithmetic**
  - A system of arithmetic for integers, where numbers "wrap around" upon reaching a certain value called the modulus.
- **plaintext**
  - The original text or message.
- **Prime Number**
  - A whole number greater than 1 whose only positive integer factors are 1 and itself.

- **Public-Key Cryptography**
  - Public key cryptography uses an encryption algorithm in which two keys are produced. One key is made public while the other is kept private. The public key and the private key are cryptographic inverses; what one key encrypts only the other key can decrypt. Public key cryptography is also called asymmetric cryptography.
- **Quantum Computing**
  - This is an area in computer technology that is based off of Quantum Theory. It works differently than regular computers, instead of bits it uses something called qubits that can take on the value 0, or 1, or both simultaneously, which makes these computers far superior than others. (*Google's Quantum Computer Is 100 Million Times Faster Than Your Laptop*)
- **RSA**
  - The acronym stands for Rivest, Shamir, and Adelman, the inventors of the technique. RSA is one of the first public-key cryptosystems and is widely used for secure data transmission. The basic security in RSA comes from the fact that, while it is relatively easy to multiply two huge prime numbers together to obtain their product, it is computationally difficult to go the reverse direction: to find the two prime factors of a given composite number. It is this one-way nature of RSA that allows an encryption key to be generated and disclosed to the world, and yet not allow a message to be decrypted. (*Glossary of Cryptographic Terms*)
- **Semiprime Number**
  - These are numbers that are the product of two prime numbers. This means the only divisors are 1, itself, prime 1, prime 2.
- **Time Complexity**
  - Time complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input. (*Time Complexity*)

2.2 Theorems

Theorem 1 (The Fundamental Theorem of Arithmetic)**.**
Every integer greater than 1 can be written as a product of prime numbers, perhaps with just one prime in the product, and this product is unique when the primes are written in non-descending order.

$$p_1^{e_1} \times p_2^{e_2} \times ... \times p_k^{e_k} = \prod_{i=1}^{k} p_i^{e_i}$$

Theorem 2 (Infinite Prime Numbers)**.**
The number of prime numbers is infinite.

Proof. (*The Prime Pages*)
Suppose that $p_1, ..., p_k$ were all of the prime numbers.
Let $n = p_1 \times p_2 \times ... \times p_k + 1$.
Then n is not divisible by any p_i because n mod $p_i = 1$ for every i.
By **Theorem 1**, n can be written as the product of one or more prime numbers.
So, n is divisible by some prime, which cannot be one of the p_i.
Therefore, the assumption that $p_1, ..., p_k$ are all of the prime numbers is false, and the number of all primes is infinite. □

Theorem 3.
If n is composite, then n has a prime factor $p \leq \sqrt{n}$.

Proof.
If n is composite, then it has at least two prime factors.
Let p and q be two of them.
Assume $p \leq q$, then $n \geq pq \geq p^2$
So $p \leq \sqrt{n}$. □

Theorem 4 (Fermat's Little Theorem)**.**
If p is a prime number and n is an integer not divisible by p, then p divides $n^{p-1} - 1$, *that is,* $n^{p-1} \equiv 1 \pmod{p}$.

Corollary 4.1. *If p is a prime number and n is an integer, then* $n^p \equiv n \pmod{p}$.

Theorem 5 (Euclid's First Theorem)**.**
If p is a prime number and $p|ab$ *then* $p|a$ *or* $p|b$.

The History of Cryptography

3.1 Origins

Throughout history, people have needed to protect their secrets. For thousands of years, people have been using codes and ciphers to protect those secrets. Back then, cryptography started off as an art; it was only studied by writers and artists. It was used as early as 1900 BCE in Ancient Egypt. During these times the Egyptians would create a code using hieroglyphics by switching the order of them and only people who knew the order could translate the message. (*Cryptology*)

As the years went on these methods become more clever and involved. The Greeks contributed a lot to cryptography, including two ciphers, the Spartan Scytale and the Polybius Square. The scytale was used by the Spartan army to send messages without being detected. Two people in the army would have two pieces of wood that were equal in diameter. The messages would be written on strips of leather wrapped around the wood. These messages could only be read if the strip was wrapped around a wood of the same size. The Polybius Square was another unique technique. The Greeks used a 5 by 5 square, with sides labeled 1 through 5 on the top and the side, while the squares would be filled with the alphabet (*Cryptology*). Each letter is associated with a two-digit number, one digit each coming from the side and top.

3.2 A Transition

Up until the Greeks, Cryptography was considered to be a form of literature because it would just be manipulating writing. Once the Romans came along the focus shifted from literature to mathematics. The Caesar Cipher, named after Julius Caesar, started to implement tiny amounts of math, mainly addition. This cipher, more commonly referred to as the shift cipher, shifts the original letters in the message a certain amount of spaces and returns a random group of letters. For example, using our

alphabet, if we shifted the letters 5 spaces then the letter 'A' would be changed to 'F' and the person receiving this message would be able to convert it back.

3.3 Cryptography at War

Cryptography has fully transitioned to mathematics and computer science now. Governments use different encryptions to secure their secrets and private messages. Banks use cryptography to secure accounts and transactions. Credit card companies will ensure that their cards get encrypted when used to keep customers' information safe. The list goes on.

An important part of recent history during World War II involves the Enigma Machine. The Nazis were using a type of rotor machine that was referred to as Enigma. This machine has multiple stages, so every step changes the letters. These messages would be sent over the radio and would be intercepted by the allies but they could never decrypt them. Luckily, the Polish Cipher Bureau was able to obtain the detailed structure of an enigma machine so they could recreate it. (*Cryptology*)

Shortly before the invasion of Poland, the agents were able to meet up with British cryptographers in France to decrypt the Enigma. The British code group at Bletchley Park was able to find the decryption and because of this, helped end the war early saving millions of lives. One of the members of the British group was Alan Turing, the conceptual founder of modern computing. Recently, Hollywood turned this story into a movie called *The Imitation Game* starring Benedict Cumberbatch as Alan Turing.

3.4 The Creation and Uses of RSA

In 1977, the RSA Algorithm was created by Ron Rivest, Adi Shamir, and Leonard Adleman. This is an algorithm that is used in public-key encryptions (Hoffstein, Pipher, and Silverman, 2008). Public-key encryption is when a user publishes a public key so that other users can encrypt and send messages to them. However, each user has their own private key for decryption. The reason that the decryption key is so important is that it is extremely hard to find the decryption key from the public key. This is because the public key is created by using two large prime numbers multiplied together.

RSA is now considered an older algorithm, but it is still being used by some companies as well as the government to encrypt messages. On the next page, we will see a table that walks you through how sending messages in RSA works. With this table, you will be able to also send messages with RSA.

Bob	Alice
Key Creation	
• Pick your secret primes, p and q. Compute $N = pq$ • Choose an encryption exponent, e, and make sure $gcd(e, (p-1)(q-1)) = 1$. • Publish N and e.	This is the part where Bob will create a public key so people can send him messages. (For back and forth communication Alice will create her own public key)
Encryption	
During this step Alice will encrypt her message and send it back to Bob for him to decrypt it.	• Choose a plaintext message to encrypt, m. • Use Bob's public key (N, e) to compute $c \equiv m^e (mod N)$. • Send the ciphertext message, c, to Bob.
Decryption	
• Compute d satisfying: $ed \equiv 1 (mod (p-1)(q-1))$ • Compute $m' \equiv c^d (mod N)$. • Then, m' =plaintext m	If everything is done correctly, Bob will receive the message that Alice sent.

(Wagstaff, 2013)

This table shows how you can send messages using RSA.

The Mathematics

4.1 What is a Prime Number?

Prime numbers are an integral part of mathematics, especially when it comes to factoring. We know that a prime number is a number that is only divisible by 1 and itself. One amazing thing about prime numbers is that any whole number is comprised of different primes. For example, let's look at the number 100:
When you start to factor 100 you get:

$$2 \times 50$$

2 is a prime number so it cannot be factored anymore but 50 can:

$$50 = 2 \times 25$$

Again, 2 is prime so then:

$$25 = 5 \times 5$$

So finally, 100 can be written as:

$$100 = 2 \times 2 \times 5 \times 5 \text{ or } 100 = 2^2 \times 5^2$$

This is helpful when factoring because every number can be expressed as different prime numbers multiplied together. Prime numbers are the building blocks of whole numbers. Most importantly, every positive integer greater than 2 can be factored into primes *in a unique way*. (Hardy et al., 2008)

In Cryptography, prime numbers are important, especially today. A decent amount of the modern computer encryptions use prime numbers to create large numbers. The reason people still use that technique is that there has been no efficient way to factor large numbers that are a product of two primes.

Most of the encryption techniques that use products of prime numbers are called "Public-Key Encryptions," because the number is known to everyone. However, this is hard to factor as the number gets larger (Hardy et al., 2008). Therefore, you can make the encryptions stronger by using larger prime numbers. So, as long as there is no efficient way to factor these numbers, prime numbers will always play a role in current Cryptography.

4.2 Factoring Numbers

Every number has factors, they are the numbers that are multiplied together to create that number. Some numbers have multiple factors and some only have two (other than 1 and itself). We want to focus on factoring the numbers that only have two prime factors, these are called semiprime numbers. The reason we do not focus on the numbers with multiple prime factors is that they can be associated with multiple prime numbers.

As we saw in Section 4.1, numbers can be written only as products of prime numbers. When you get into prime factorization, certain numbers can be comprised of many different prime numbers. However, with semiprime numbers, there will only be two prime numbers in the prime factorization form. So the question becomes, how can we factor these numbers easily?

4.3 Factoring Products of Two Prime Numbers

One problem with semiprime numbers is that there is no easy way to factor them once they get big enough. There are some incredibly fast factoring algorithms that work great; the problem is that they can only work for so long before the numbers get too big. Going back to the RSA algorithm, a semiprime is one of the safest public keys one can have because once it gets big enough it will just take too long to factor (Wagstaff, 2013). There is a belief that once quantum computing becomes real, we will be able to factor any number instantly. However, we are probably still years away from fully obtaining this technology.

4.4 The Use of Fermat's Little Theorem

The work of famous French mathematician, Pierre de Fermat, is extremely helpful when it comes to prime numbers in general. Fermat's Little Theorem is useful for primality testing as well as for proving that the RSA Algorithm is correct. If this theorem is implemented into code it can check to see if larger numbers are prime or not.

The problem is that this theorem can take longer to run as the numbers get larger. People may cut corners, not check every number, and a number will seem to be a prime number, that is not. This theorem is mostly used to help determine what *isn't* a prime number. Sometimes there are numbers that pass the test, like Fermat pseudoprimes and Carmichael numbers, but are not

actually prime numbers. These will mess up public keys because you are not actually using prime numbers. When this theorem is used correctly then it is a really powerful theorem that makes prime factorization and public-key cryptography easier.

4.5 The Factoring Algorithms

4.5.1 Trial Division

Trial Division is the factoring algorithm that we used as the base model for comparison. This is because trial division is the easiest to understand out of all the algorithms. There are downsides to this though; it is considered the most laborious of the factoring algorithms.

The way this algorithm works is by using the trial division test. We take a number, n, and check every number less than n (in Theorem 3) to see if it can be factored. When all of the factors are found you will see every possible factor for n and if you go a step further with this algorithm you can put n in its prime factorization form.

4.5.2 Pollard Rho

The Pollard's Rho method is a factoring algorithm that was created by John Pollard in 1975. This algorithm works by using modular arithmetic to iterate a polynomial until a cycle is detected and then the number can be factored (Wagstaff, 2013). On a basic level, this is how this method works:
The number we are trying to factor is N, where $N = pq$ and p and q are both unique prime numbers. So we need to find either p or q.

- Pollard's Rho uses the function $f(x) = x^2 + b$ in $\mathbb{Z}_n$
- From this we follow the orbit, which looks like this:
  - $S \to f(S) \to f(f(S)) \to ...$
- It will keep continuing until it finds a number that is a factor of N. What we do is we compare S to $f^n(S)$ as we iterate. We compute GCD(N, $f^n(S) - S$). If this value is not 1, then we have potentially found a factor of N.

4.5.3 Continued Fraction Factoring

A continued fraction is just another way of writing fractions. It is a way to compute the square root of a number extremely accurately. The form of continued fractions is shown below:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots}}}$$

This form is called a continued fraction. The value of a_n must be an integer. A continued fraction can also be rational or irrational, it depends on the number. Continued fraction representation of numbers such as π or e are infinitely long. Given a number, α, a continued fraction can be created by using the recursive algorithm:

$$a_i = [\alpha_i]$$

$$\alpha_{i+1} = \frac{1}{\alpha_i - a_i}$$

To understand continued fractions better, we shall look at an example:

Let $\alpha = 437$

$$\sqrt{437} = 20 + (\sqrt{437} - 20)$$

$$= 20 + \frac{1}{\frac{1}{\sqrt{437}-20}\frac{\sqrt{437}+20}{\sqrt{437}+20}}$$

$$= 20 + \frac{1}{\frac{20+\sqrt{437}}{37}}$$

$$\frac{20+\sqrt{437}}{37} = 1 + \frac{-17+\sqrt{437}}{37} = 1 + \frac{1}{\frac{37}{-17+\sqrt{437}}\frac{-17-\sqrt{437}}{-17-\sqrt{437}}}$$

$$= 1 + \frac{1}{\frac{-629-37\sqrt{437}}{-148}} = 1 + \frac{1}{\frac{17+\sqrt{437}}{4}}$$

$$\frac{17+\sqrt{437}}{4} = 9 + \frac{-19+\sqrt{437}}{4} = 9 + \frac{1}{\frac{4}{-19+\sqrt{437}}\frac{-19-\sqrt{437}}{-19-\sqrt{437}}}$$

$$= 9 + \frac{-76-4\sqrt{437}}{-76} = 9 + \frac{1}{\frac{19+\sqrt{437}}{19}}$$

$$\frac{19+\sqrt{437}}{19} = 2 + \frac{-19+\sqrt{437}}{19} = 2 + \frac{1}{\frac{19}{-19+\sqrt{437}}\frac{-19-\sqrt{437}}{-19-\sqrt{437}}}$$

$$= 2 + \frac{1}{\frac{361+19\sqrt{437}}{76}} = 2 + \frac{1}{\frac{1}{\frac{19+\sqrt{437}}{4}}}$$

$$\frac{19+\sqrt{437}}{4} = 9 + \frac{-17+\sqrt{437}}{4} = 9 + \frac{1}{\frac{4}{-17+\sqrt{437}}\frac{-17-\sqrt{437}}{-17-\sqrt{437}}}$$

$$= 9 + \frac{1}{\frac{68+4\sqrt{437}}{148}} = 9 + \frac{1}{\frac{1}{\frac{17+\sqrt{437}}{37}}}$$

$$\frac{17+\sqrt{437}}{37} = 1 + \frac{-20+\sqrt{437}}{37} = 1 + \frac{1}{\frac{37}{-20+\sqrt{437}}\frac{-20-\sqrt{437}}{-20-\sqrt{437}}}$$

$$= 1 + \frac{1}{\frac{(37*20)+37\sqrt{437}}{37}} = 1 + \frac{1}{\frac{1}{\frac{20+\sqrt{437}}{1}}}$$

$$20+\sqrt{437} = 40$$

This will now continue to repeat

We know to stop there because for square roots every continued fraction repeats eventually. So in this example it went:

$$20, 1, 9, 2, 9, 1, 40$$

When we finally see the repetition, the final number in the sequence will be double the first number (i.e. $20 * 2 = 40$).

In Cryptography, continued fractions can actually be used to factor. Continued fractions can also be used for $\sqrt{n}$ and it will continue to use the form we saw above:

$$\sqrt{n} = 1 + \frac{x-1}{1+\sqrt{n}}$$

Using different numbers as integers from every step of the continued fraction we can create a strong algorithm that factors numbers well. There are five integers in this algorithm: i, q_i, p_i, Q_i, A_i (Wagstaff, 2013). i is just the step you are on, so it will start at 0 and work its way up until the fraction ends. A_i is computed outside of the method, but it is the numerator of the i-th convergent. In the continued fraction method, once it is computed then every part of it is used to help factor. Each integer has a specific location in the continued fraction and they are used to create a table to compute different integers. These are the positions:

$$q_i + \frac{1}{\frac{p_i+\sqrt{\alpha}}{Q_i}}$$

We are able to use continued fractions to factor because they help us find perfect square solutions. The key fact is the relationship between all of these variables when we compute the continued fraction. We have $(-1)^i Q_i = A_{i-1}^2 - B_{i-1}^2 N$, which implies $A_{i-1}^2 \equiv (-1)^i Q_i \pmod{N}$. Thus, if we can find a Q_i that is a perfect square (or we can build a perfect square by multiplying together different Q_i), then we have a solution to the congruence below. The solution is to $x^2 \equiv y^2 \pmod{n}$. Then that means $(x+y)(x-y) \equiv 0 \pmod{n}$ and continuing going until we are able to find the factors of our original number (Wagstaff, 2013). However, the continued fraction factoring algorithm was overshadowed by the Quadratic Sieve. In 4.5.4 you will learn about the second step of both algorithms. Both use the second step, but the first steps are different, with the Quadratic Sieve being more efficient.

4.5.4 Quadratic Sieve

The Quadratic Sieve Algorithm was invented by Carl Pomerance in 1981. It is a method that is built off of previous ideas by two mathematicians, Kraitchik and Dixon. (Wagstaff, 2013)

This algorithm is similar to the Continued Fraction Factoring Algorithm. The difference comes in the initial step. Unlike the Continued Fraction, the sieve uses outputs from a quadratic polynomial to construct perfect squares. In the second step, we use linear algebra to find and create different perfect squares if there are none created originally, this allows us to factor the initial number.

Year ago, there was an RSA challenge number that was created. It was a 129-digit semiprime that needed to be factored (Wagstaff, 2013). The Quadratic Sieve was used to factor this number. This algorithm is much more advanced than the trial division method. This is actually one of the top factoring algorithms that you can use today.

Computing Languages

One of the main goals of this project was to implement the factoring algorithms into code. From the code, we would take our data. We ended up using three programming languages: Python, Sage, and Java.

5.1 Python

Python was created in the late 1980s and it is one of the most commonly used languages today in programming. Throughout this research, it was also the main language that was used. Python is a language that is able to handle numbers no matter how large they get, however, the runtime is slower than other languages. Python is also good when you need to explain the code; it is usually straightforward and can be explained easily. We were also lucky that Python had an easy to implement timer so we could gather extremely accurate data.

5.2 SageMath

SageMath was brought into this research late, but it ended up being the most effective language for what we needed. SageMath is actually a form of the language Python, but it adds in tons of new math functions. So this gave us a lot of extra information that we could use, including certain factoring algorithms. This is still a fairly new language and it also runs on a server. So because of this gathering time measurements can be different because of how busy the server is. However, you can download your own copy of Sage so it doesn't depend on server traffic.

5.3 Java

Initially, we expected Java to be useful for this research, however, it was the opposite. This language was faster at factoring, but it is good with large numbers. We could have added different classes to improve larger numbers, but it was not needed when researching this topic.

Implementation of the Algorithms

6.1 Overview of the Program

The original program that was created did much more than factor numbers. We created a program that was like a sandbox, where we could just test and create anything based on prime numbers. We implemented different factoring algorithms, prime number generators, and many things that surround prime numbers. This was to get a better understanding of everything based on this topic, rather than just factoring semiprimes.

6.1.1 Pseudo-code of the Algorithms

Trial Division

```
def trial_division(n):
    a = []
    f = 2
        while n > 1:
            if (n % f == 0):
                a.append(f)
                n /= f
            else:
                f += 1
        return a
```

Pollard Rho

```
 x = 2; y = 2; d = 1
    while d = 1:
        x = g(x), where g(x)=(x2+1) mod n
        y = g(g(y))
        d = gcd(|x - y|, n)
    if d = n:
        return failure
    else:
        return d
```

6.2 Implementing Algorithms

Coding these algorithms was half of the battle in this research. We had to find the pseudo-code, understand the algorithms, and then write code for them to work. In the end, we coded two of the algorithms and used a built-in function for the other.

6.2.1 Trial Division

Implementing Trial Division was the easiest out of all the factoring algorithms. As you can see from the pseudo-code it is only several lines long. The actual code for this algorithm is shown in the appendix. This algorithm is the most trivial, so it was the easiest to create. This is because there is not much mathematics involved other than just division.

6.2.2 Pollard's Rho

Pollard's Rho was more difficult to implement. The code is only slightly longer, however, the algorithm only works *most* of the time. There are certain times where the algorithm wouldn't be able to factor a number or it just returned 1. The way we fixed this problem was by creating a loop around the algorithm. The loop cycles through the algorithm multiple times until a real answer is shown. The actual Python code for this algorithm is shown in the appendix.

6.2.3 Quadratic Sieve

The Quadratic Sieve algorithm is the most complex out of all the ones we chose. SageMath has a built-in factoring algorithm that uses the Quadratic Sieve. From this algorithm, we were able to factor numbers just by calling the function from SageMath. This saved us tons of time and gave us the perfectly created code for this project.

6.2.4 Continued Fraction Factoring Algorithm

When it came to the Continued Fraction Algorithm (CFRAC) we actually decided not to implement it. After researching both the Quadratic Sieve and the CFRAC we saw that the algorithms were very similar to the second step. So because of this, and how close both algorithms were in design, the CFRAC was not really used and implemented.

Data and Observations

7.1 Comparison of Algorithms

We want to also look at time complexity for each algorithm. This is looking at the efficiency of the algorithm, or how long it takes for the function to run.

Trial Division	Pollard's Rho	Continued Fraction	Quadratic Sieve
$O(\sqrt{n})$	$O(\sqrt{p}) \leq O(n^{1/4})$	$O(e^{\sqrt{2ln(n)ln(ln(n))}})$	$e^{(1+o(1))\sqrt{ln(n)ln(ln(n))}}$

Initially, we believed that the more advanced algorithms would be faster at factoring the semiprime numbers. However, it was on a much larger scale than we expected. When we look at the algorithms this is how the strength and speed are:

$$\text{Quadratic Sieve} \geq \text{Continued Fraction} > \text{Pollard's Rho} > \text{Trial Division}$$

When we look at the different time complexities Pollard's Rho shows us right away that it is better than Trial Division. Actually, Trial Division's time complexity shows us that we should not be using it because it is a ineffective algorithm for factoring. Pollard's Rho is decent, compared to Trial Division, but it is overshadowed by the Continued Fraction and the Quadratic Sieve. We can see that those two algorithms are similar in complexity, but the Quadratic Sieve is just slightly better. This time complexity is a good example as to why it was chosen over the Continued Fraction factoring algorithm.

If you are the person creating an algorithm you will want the time complexities of the factoring algorithm to be bad. This is because the slower the algorithm is the better the factoring algorithm is. In the case of RSA, people who are using this method to send and receive messages will hope people use the trial division algorithm to break the public key.

The important part to know is that if you are attempting to break this public key, an algorithm with a strong time complexity is better, but if you want your information secure when you send messages you want it to be a bad time complexity.

7.1.1 Graphs

For every algorithm that was implemented, we collected timed data to compare the algorithms against each other. Each algorithm has a graph that shows the data collected from the tests we ran. These tests were testing the speed of the algorithm in seconds compared to how many digits there were to factor. These graphs are able to show the algorithms against each other better than just explaining it.

There are several types of line graphs here. There are three individual line graphs for the three implemented algorithms. The comparison of the three algorithms against each other depending on the number of digits (The length of a Semiprime number). We will also see the comparison of some algorithms in different languages.

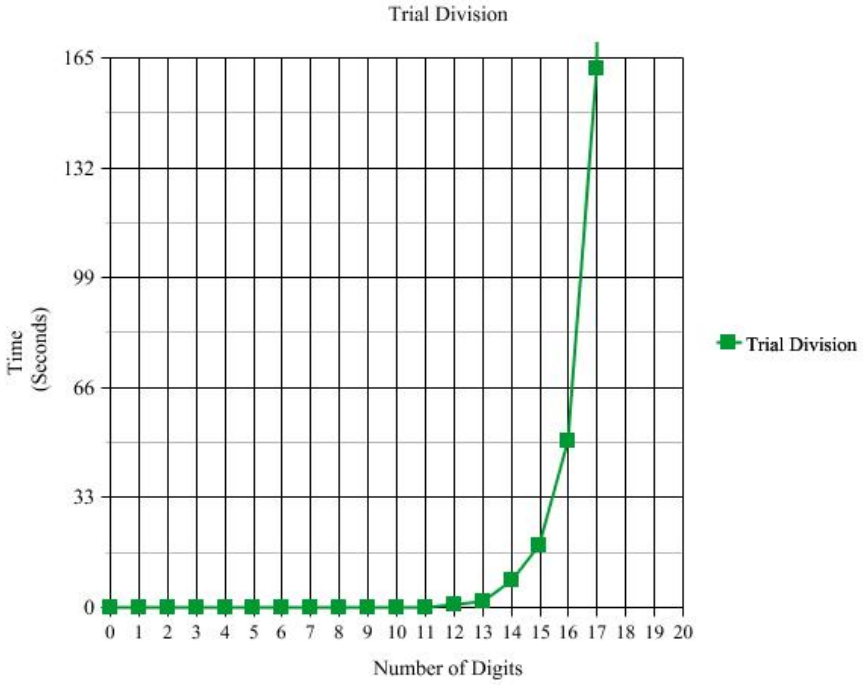

Figure 7-1: *Trial Division Timing Graph*

When we look at the Trial Division graph, we see it doing well at factoring numbers up to 16 digits. Once we get past 16 digits then the algorithm becomes slow and the program will exit out before it can finish factoring. Sadly, this algorithm was not able to factor up to 20 digits, but it still was good enough to compare against other algorithms. We used the Trial Division Algorithm as a constant for our research because of how trivial it was.

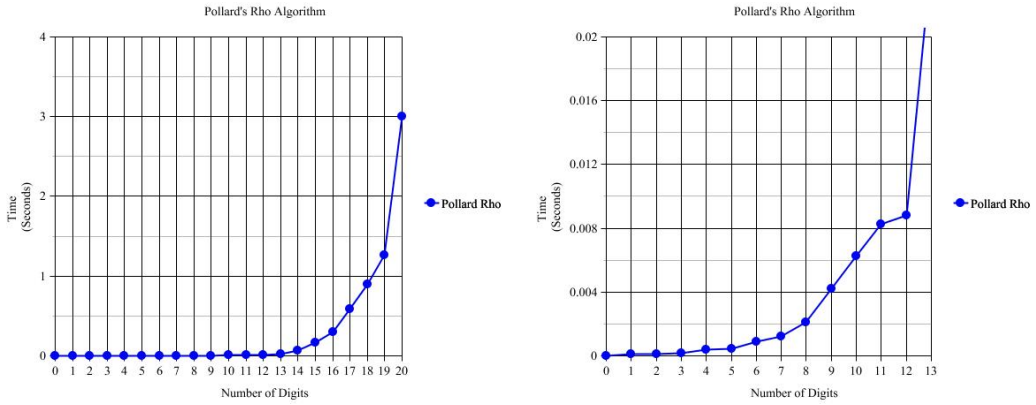

Figure 7-2: *Pollard's Rho Timing Graph*

Unlike the Trial Division algorithm, Pollard's Rho was able to factor up to 20 digits. It was fairly quick up to 20 digits, only taking about 3 seconds. However, once we got past 20 digits the algorithm was not able to factor anymore because of how long it took. Similar to the Trial Division, Pollard's Rho did not perform as well as expected. For factoring digits 1 to 20 it performed extremely well, however it dropped off so quickly once it got larger that it made it slow.

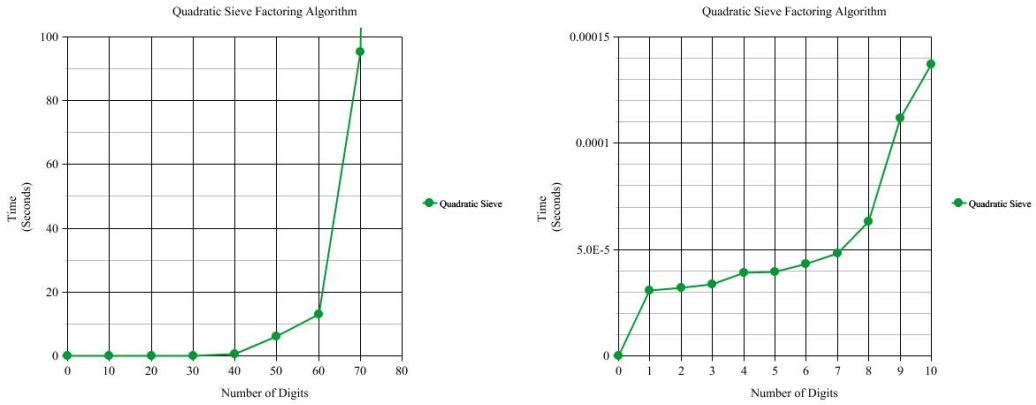

Figure 7-3: *Quadratic Sieve Timing Graph*

The Quadratic Sieve graph looks much different than the other two algorithms. This is because the sieve works best with large numbers. We were able to factor semiprime numbers up to 70 digits long. This is a huge improvement from the first two algorithms. This was also expected because of how strong it is. I believe that if we allowed more time to gather data we would have been able to factor the 129-digit number associated with RSA.

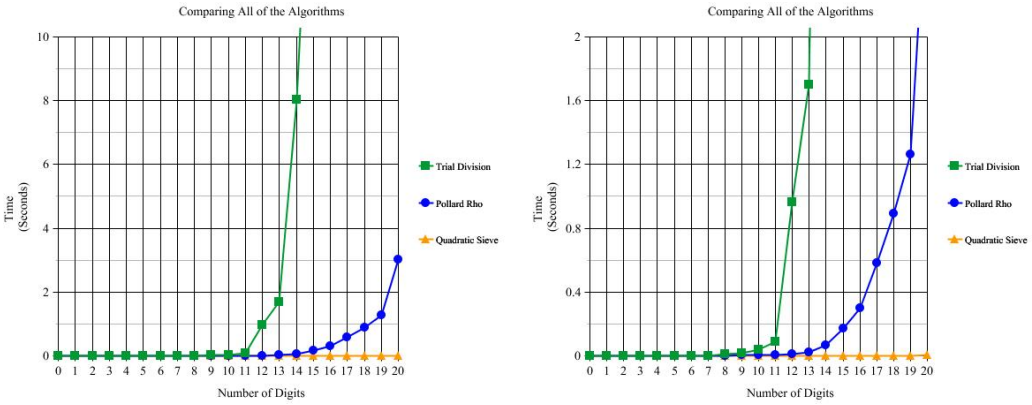

Figure 7-4: *Algorithm Comparison Timing Graph*

Looking at all three algorithms together shows a lot about their different rates. Trial Division ends up going off the graph fairly quick. This shows the strength of the other two algorithms and also how weak Trial Division is. Pollard's Rho looks good compared to Trial Division, but when you see the Quadratic Sieve it beats all of them. The sieve almost looks like a completely flat line because it factors 20-digit semiprimes in under one second. This is great proof of how powerful the Quadratic Sieve is compared to the others.

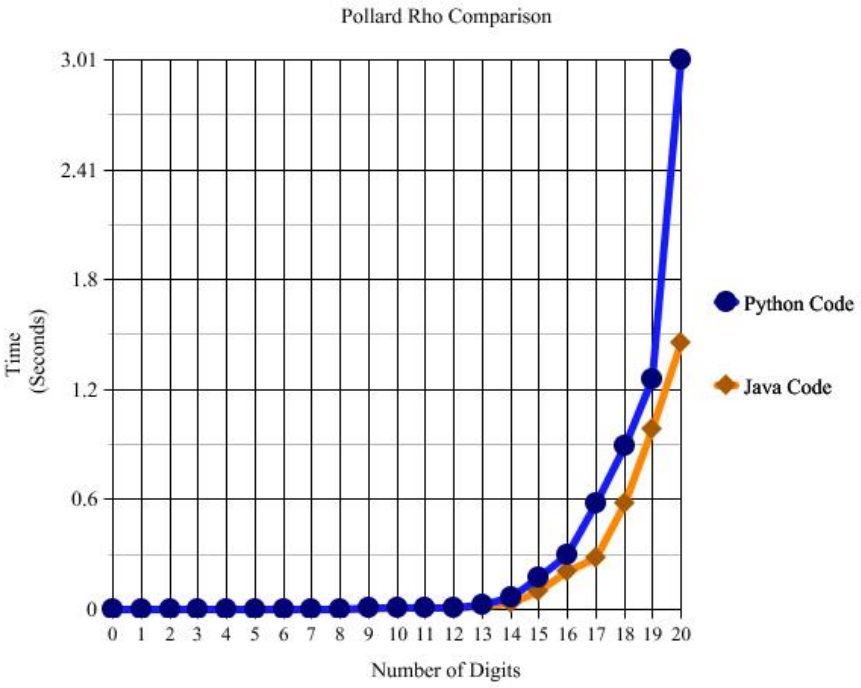

Figure 7-5: *Python Vs. Java Timing Graph*

Comparing the same algorithm in different languages is interesting to look at. Out of all the graphs, this was the most interesting one to me. This is because although we are applying the exact same algorithm; but it can perform differently depending on the language. Here we see Pollard Rho in both Python and Java. Up until 20 digits, Java performs slightly better, but although Python is known as a slower language, it did well in comparison. It was disappointing that we were not able to look at the algorithms more deeply in other languages but that could be looked at in the future when there is more time.

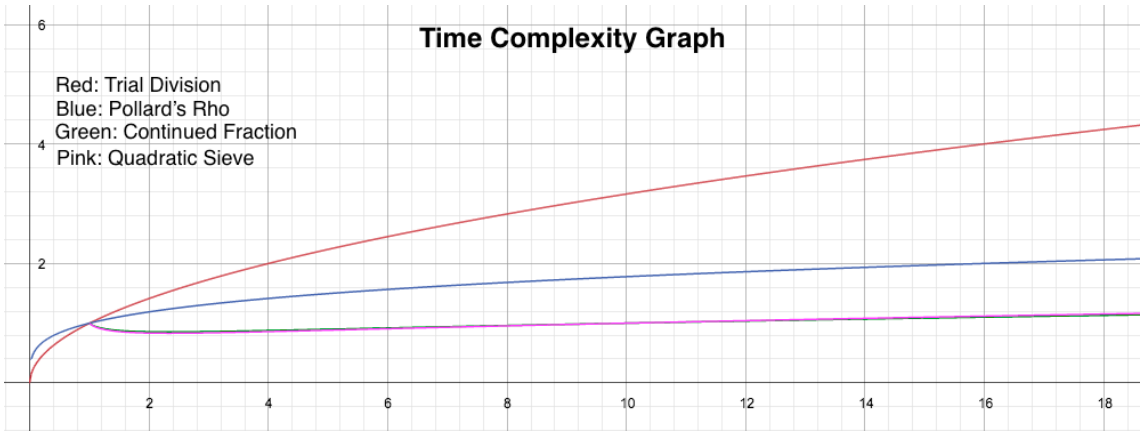

Figure 7-6: *Time Complexity Graph*

Time complexity is important when comparing algorithms. Figure 7-6 looks at the time complexity equations for the four algorithms. All four algorithms have decent time complexity graphs, but you can see which equations are better. This graph was created by a equation generator, it looks at all of the equations and graphs them to be compared. It is interesting to look at the Continued Fraction and the Quadratic Sieve because of how similar they are. The Sieve gains a small edge that you can barely see, but it is just fast enough to beat out the Continued Fraction.

There is actually another interesting part of the Quadratic Sieve and Continued Fraction time complexities. It comes at the beginning of the graph. It dips down right after the beginning. This shows both of these algorithms are better as it increases in size, which is expected. This graph also proves the strength of our algorithms compared against each other.

Improving The Code

8.1 Automated Programming

In research, there is always room for improvement. Over the course of this project, the code improved a lot by updating it, but there is still a lot that can be added to make it run more smoothly and more automated. A single program could be created to do everything at once. So there would be a program that would have the four factoring algorithms implemented correctly. The program would randomly select a semiprime number and have the algorithms attempt to factor it while it is collecting the data. Once the program finishes running it would create different graphs for users to compare the algorithms in real time. This type of program would be perfect because users would be able to start the program and it will run all day while the user can do other research. The main goal of improving the code in the future would be making an automated program.

8.2 More Programming Languages

Originally, the plan was to implement factoring algorithms in multiple languages and compare them. However, we ended up focusing mostly on Python because of what it offered. Implementing these factoring algorithms in other languages and comparing them against each other would be interesting to see. On a very small scale we saw Python vs Java, but to see it on a larger scale with more algorithms would help give us a better picture when comparing everything. Potential languages that would be used would be Matlab, Java, C++, SageMath, and C.

Lastly, if we were able to use a server to test algorithms we would possibly see better data when factoring larger numbers. This would probably be the most important improvement to the project because we have been using a regular laptop for everything.

Conclusion

9.1 Future Threats

The RSA public key is so simple, but it is so hard to break. Prime numbers are such an interesting topic because we know so much about them but at the same time, we barely know anything as well. It is extremely impressive that the RSA cryptosystem has lasted for over 40 years. It appears that public keys will keep being used until there is an easy way to factor the semiprime numbers.

A threat to these types of public keys is currently being worked; it is called Quantum Computing. If Quantum Computing becomes more than just a theory and works on the scale it is expected to work, it would be able to factor any number almost instantly. This would make many of today's cryptosystems obsolete but also bring in a new age of encryption techniques. The reason this is so fast is that it can hold values of 0, 1, or both at the same time with "quantum bits". Regular bits can just hold one value at a time, not multiple, which is why Quantum Computing is so much faster. Currently, many companies has been working towards creating a quantum computer.

9.2 Final Thoughts

This research was a step in the right direction for learning more about Public-Key Encryption Systems. These have had a big impact on Cryptography in the modern era because of how strong their public keys are. Seeing all of the factoring algorithms and comparing them against each other showed an interesting fact: there are some *very* advanced algorithms, but there is still no perfect or effective way to factor these numbers. Sure, some algorithms are much more powerful than others, but it is just crazy to think about how there is no one algorithm to factor all numbers. Moving forward in Cryptography, it will be interesting to see where Quantum Computing goes and if Public-Key Encryptions will last to see 50 years. However, for now, we will continue to use what we have until we are faced with a problem that puts our information at risk.

Acknowledgements

This thesis was only possible because of the help of many individuals. Writing a thesis is no easy task and I would like to extend my sincerest thanks to everyone that helped.

First, I would like to thank my advisor, Dr. Jackie Anderson. Someone who spent over a year working with me on this project and without her amazing knowledge and guidance would not have been able to complete this research.

Next, I would like to thank my other advisor, Dr. Michael Black. He is someone who helped add an interdisciplinary look at this project and was always available to help when it was needed.

I would also like to thank the members of my reading committee, Dr. Ward Heilman, and Dr. Haleh Khojasteh. They took the time to help edit and make this thesis better than it was before and gave great feedback for this paper.

Also, a special thanks to Undergraduate Research at Bridgewater State University for supplying us with an Adrian Tinsley Program (ATP) Summer Grant for Undergraduate Research. This gave us the opportunity to put in so much time and effort and helped bring this research to a new level.

Lastly, I would like to thank Bridgewater State University's Computer Science and Mathematics departments for giving me the opportunity to conduct and complete an interdisciplinary honors thesis and being supportive in the process as well.

Bibliography

Caldwell, C. *The Prime Pages*. Available at https://primes.utm.edu/ (1994).

Engelfriet, Arnoud. *Glossary of Cryptographic Terms*. Available at http://www.pgp.net/pgpnet/pgp-faq/pgp-faq-glossary.html (2002).

Hardy, G. H. et al. (2008). *An Introduction to the Theory of Numbers*. 6th. Oxford: Oxford University Press.

Heilman, Ward. *Cryptology*. The Class "Introduction to Cryptology" at Bridgewater State University (Fall 2016).

Hoffstein, J., J. Pipher, and J. H. Silverman (2008). *Introduction to Mathematical Cryptography*. 1st. New York, NY: Springer Science Business Media.

IBM. *Terms and Definitions*. Available at https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/terms.html (2018).

Nield, David. *Google's Quantum Computer Is 100 Million Times Faster Than Your Laptop*. Available at https://www.sciencealert.com/google-s-quantum-computer-is-100-million-times-faster-than-your-laptop (2015).

Techopedia. *Time Complexity*. Available at https://www.techopedia.com/definition/22573/time-complexity (2018).

Wagstaff, S. S. (2013). *The Joy of Factoring*. Providence, RI: American Mathematical Society.

Code Appendix

A.1 The Main Code Used

```
These are the needed for a lot of the code:
import random
import math
import re
from fractions import gcd

def primeNumber(n):
    ## This program checks to see if a number is prime or not ##
    squareRoot = int(math.sqrt(n)+1)

    if all(n%i!=0 for i in range(2,squareRoot)):
        print(n, "is a prime number.")

    else:
        print(n, "is a composite number.")



def primeGenerator(n,outfileName):
    outfile = open(outfileName, "w")
    primeList = []

    for num in range(2,n):
        if all(num%i!=0 for i in range(2,int(math.sqrt(num))+1)):
            print(num, file=outfile)
            primeList.append(num)

    outfile.close()
    print("\nThe", len(primeList),"prime numbers have been written to",
            outfileName)
```

```
def primeGeneratorParameters(a,b,outfileName):
    outfile = open(outfileName, "w")
    primeList = []

    for num in range(a,b):
        if all(num%i!=0 for i in range(2,int(math.sqrt(num))+1)):
            print(num, file=outfile)
            primeList.append(num)

    outfile.close()
    print("\nThe", len(primeList),"prime numbers have been written to",
            outfileName)


def FLT(n):
    primeTest = 2**(n-1)
    if(n==2):
        print(n, "is a prime number.")
    elif(n%2==0):
        print(n, "is not a prime number.")
    elif(primeTest % n == 1):
        print("There is a possibility that", n, "is a prime number.")
    else: ## If nothing else is there then it is not a prime ##
        print(n, "is not a prime number.")


def normalFactor(number):
    factors = []
    squareRoot = int(math.sqrt(number)+1)

    for num in range(1, squareRoot):
        if(number%num == 0):
            factors.append(num)
            factors.append(int(number/num))
            print("     ", num, "x", int(number/num), "=", number)

    print("")
    print("There are", int(len(factors)/2), "ways to factor", number)
```

```
def pollardRho(N):
    b = random.randint(1, N-3)  ## finds a random b ##
    s = random.randint(0, N-1)  ## finds a random s ##
    A = s
    B = s

    def f(x):
        return ((x**2)+b)%N

    g = 1
    attempt = eval(input("How many times would you like to attempt this? "))
    print("")

    for i in range(attempt):
        while(g == 1):
            A = f(A)  ## sends A through the f(x)
            B = f(f(B))  ## sends B through f(x) twice
            g = gcd(A-B,N)  ## sets g to the gcd

        if(g < N):
            break  ## exit loop
        else:
            print("attempt", i+1)

    if(g < N):
        print(g, "is a proper factor of", N)
        print(int(N/g), "is a proper factor of", N)
    else:
        print("\nRestart and try again.")
```