Spring 2002

The RSA Encryption System and the Factorization of Large Numbers

Matthew Zanto
Carroll College, Helena, MT

Follow this and additional works at: https://scholars.carroll.edu/mathengcompsci_theses
Part of the Analysis Commons, and the Information Security Commons

Recommended Citation
https://scholars.carroll.edu/mathengcompsci_theses/83

This Thesis is brought to you for free and open access by the Mathematics, Engineering and Computer Science at Carroll Scholars. It has been accepted for inclusion in Mathematics, Engineering and Computer Science Undergraduate Theses by an authorized administrator of Carroll Scholars. For more information, please contact tkratz@carroll.edu.
The RSA Encryption System
and the Factorization of Large Numbers

Submitted in partial fulfillment of the requirements for graduation with honors to the

Department of Mathematics, Computer Science, Engineering

Carroll College

Written by:
Matthew J. Zanto

Helena, Montana
May, 2002
This thesis for honors recognition has been approved for the Department of Mathematics, Engineering, and Computer Science.

[Signatures and dates]

Director

Reader

Reader
Acknowledgements

I would like to take this opportunity and thank my charismatic and energetic Honors Thesis director, Phil Rose, for without his help, guidance, and vast knowledge, this paper would not have been possible. Thanks, Phil! I would also like to thank my two readers, Dr. Mark Parker and Dr. Jeff Morris, and Carroll College and the Department of Mathematics and Computer Science.
# Table of Contents

Introduction........................................................................................................... 1-5
Primes....................................................................................................................... 6-17  
   Pseudoprimes................................................................................................. 7-8  
   Carmichael Numbers................................................................................... 9  
   Strong Pseudoprimes............................................................................... 10-13  
   Euler’s Criterion.................................................................................. 13-17  
Explanation of the RSA Algorithm................................................................. 18-25  
   Encoding................................................................................................. 21-22  
   Decoding................................................................................................. 23-25  
Factoring............................................................................................................... 26-45  
   Trial Division.......................................................................................... 26-28  
   Fermat’s Algorithm............................................................................... 28-30  
   Pollard’s Rho..................................................................................... 30-33  
   Quadratic Reciprocity......................................................................... 33-40  
   The Quadratic Sieve.......................................................................... 40-45  
Conclusion........................................................................................................... 46  
Works Cited.......................................................................................................... 47

Appendices
   Miscellaneous Algorithms: .................................................................................. A1-A2  
      Fast Exponentiation............................................................................... A1  
      Greatest Common Divisor.................................................................... A1  
      Jacobi Symbol..................................................................................... A2  
   RSA Encoding and Decoding Algorithms: ..................................................... A2-A5  
      Encrypt................................................................................................... A2-A3  
      Decrypt................................................................................................... A3-A4  
      Functions................................................................................................. A4  
      Example................................................................................................. A4-A5  
   Prime Number Algorithms: ............................................................................. A5-A8  
      Generate Sequential Primes................................................................ A6  
      Generate Random Primes................................................................ A6-A7  
      Strong Pseudoprime Test................................................................ A7-A8  
   Factoring Algorithms: ....................................................................................... A8-A12  
      Trial Division..................................................................................... A8  
      Fermat’s Algorithm.......................................................................... A8-A9  
      Pollard’s Rho.................................................................................. A9  
      The Quadratic Sieve......................................................................... A9-A12
The RSA Encryption System and the Factorization of Large Numbers

Introduction

For thousands of years, codes and ciphers have been a key resource for kings and queens, armies and navies, diplomats and ambassadors. Julius Caesar conceived one of the most famous ciphers, called the Caesar Cipher. It was a monoalphabetic cipher in which each letter is simply shifted the same number of positions to the left or right. Yet, before the study of codes and ciphers developed, it was considered secure enough to use. As recently as World War II, the famous German Enigma military cipher and the equally famous Japanese Purple diplomatic cipher played important roles in the war, as did the people who broke them. These two machine ciphers were incredibly complex, and yet both were still broken in one of the most outstanding intellectual feats of the twentieth-century. Until the 1970s, no code, save a One-Time Pad cipher, was provably unbreakable. A One-Time Pad cipher is based on using a sheet of random letters, which the receiver and the sender possess. Each letter is mathematically combined with a letter in the message to come up with the encoded letter. Once a letter is used, it is scratched off the pad. Since the letters are supposed to be random, no cryptographer could ever hope to guess them, thus ensuring the security of the encoded message. Even today, with exception of the One-Time Pad, no code or cipher is completely secure. However, in 1977, three men, Ron Rivest, Adi Shamir, and Leonard Adleman, developed an encoding scheme that revolutionized the science of Cryptography: the RSA System.
The RSA System is built upon ideas that were previously published by other cryptographic experts. In 1976, a paper entitled "New Directions in Cryptography" was written and published by Martin Hellman and Whitfield Diffie. This paper, in turn, used ideas that were previously published in Ralph Merkel's paper, "Secure Communications over Insecure Channels". The idea behind both of these papers is the use of a one-way trapdoor function, $y = f(x)$, that has the property that if the $x$ value is known, then $y$ can be easily calculated. However, if $y$ is provided, then $x$ is either impossible or extremely difficult to calculate. This means that if a plaintext message $x$ was encoded into the ciphertext message $y$, and the ciphertext message $y$ was intercepted, it would be very difficult to get back the plaintext message $x$.

Although the idea of a one-way function seems to apply only to mathematics, it has analogies in the real world, too. An example of a one-way function is that it is easier to destroy a small island in the Pacific with a nuclear bomb than it is to put the island back together. Another example is that it is easier to burn gasoline than it is to make gasoline out of its byproducts, carbon dioxide and water. Also, it is simpler to format a hard drive than it is to get the deleted information back. Finally, it is easier to multiply two large prime numbers together than it is to factor the resulting composite number. This last example is the basis for a one-way function used in the RSA encoding scheme, and is the reason why RSA is so secure.
The RSA Encryption System and the Factorization of Large Numbers

Although a few real world analogies of a one-way function have been illustrated, it has not yet been proven that these mathematical functions exist. However, assume that they do exist. What use might they have? The following example was taken from Sarah Flannery’s book *In Code*:

Bob and Alice are both employees in a large corporation. Alice’s office is located in Los Angeles, and Bob resides in London. There is going to be a very important conference held in New Zealand, but the large corporation that Bob and Alice work for is trying to cut back on its expenses, so it will allow only Bob or Alice to go, but not both. If they were both in Los Angeles or London, then they would flip a coin, but since they are half a world away from each other, they settle upon a different method of deciding. By using a one-way function, they can simulate a coin being tossed, and both can be assured that no duplicity has taken place. Here is how they go about using a one-way function:

1. Bob and Alice create and agree upon a one-way function, \( y = f(x) \).
2. Alice picks a random integer \( x \) and computes \( y \) using the one-way function.
3. Alice sends the resulting \( y \) to Bob.
4. Bob receives the \( y \), and has to guess whether the \( x \) that generated \( y \) is an odd or an even integer. Despite the fact that Bob knows the function \( y = f(x) \), Bob cannot determine what \( x \) was because the function \( f \) is a one-way function. Therefore, he guesses odd or even and sends his guess to Alice.
5. Alice informs Bob whether he gets to go to New Zealand or not, and sends Bob the original \( x \) value.
6. Bob computes \( y \) using the one-way function and confirms what Alice told him.

By using this example of a one-way function, Bob and Alice can be assured that the other person has not been treacherous.

An important variant of a one-way function is a trapdoor function. Trapdoor functions are different from one-way functions in a significant way: a trapdoor
function appears to be a one-way function, or not invertible, when in reality it can be easily inverted. This means that a trapdoor function is not a one-way function, but only the person who originally designed the function knows this fact.

To see the usefulness of such functions, suppose that one really exists. Then, three people, Anne, Billy, and Caleb, each design a one-way trapdoor function for their own personal use, called $F_a$, $F_b$, and $F_c$. Each of these one-way functions is then placed in a public location where anyone in the world can access them. To Billy and Caleb, Anne’s function appears to be a one-way function. Similarly, Billy’s function appears to be one-way to everyone, and so does Caleb’s. Therefore, Anne’s, Billy’s, and Caleb’s functions do not seem to be quickly invertible. However, Anne knows the trapdoor of her function, $F_a$, Billy knows the trapdoor to his function, and Caleb knows the trapdoor to his function. Therefore, if anyone tries to use one of the one-way functions created by Anne, Billy, or Caleb, the owner of the respective function will be able to read the message that is sent to them, as long as the inverting functions, $F^{-1}_a$, $F^{-1}_b$, and $F^{-1}_c$, are kept completely secret.

Now, back to the RSA System. Ron Rivest, Adi Shamir, and Leonard Adleman invented what is known today as the RSA encryption algorithm, the name of which was derived from the first letters of their last names. This enciphering technique uses an algorithm that encodes the data in such a way that it would take every computer on the planet, running simultaneously, years and years to decipher.
The RSA Encryption System and the Factorization of Large Numbers

Generally, only the most top-secret messages are encoded with this much protection, and even then the message is encoded with only enough security so that the message can remain secret for a set amount of time. Also, this encoding procedure can take time to encode a message, so normally only small messages are encoded with it.

RSA, like many other ciphers, uses a key to encipher its information. A key is a value needed to encode or decode information. This key is made public, which is why this scheme is also called a public key algorithm. However, it does not use any ordinary key. For maximum security, it uses a massive composite number that is derived from multiplying two large prime numbers together. The two prime numbers often contain anywhere from 150 to 200 digits or more. The composite number, therefore, is often from 300 to 400 digits. The security of this system depends on the impossibility to factor a large composite number formed this way with any amount of speed by using known methods. Several algorithms, like Fermat's Algorithm and the Quadratic Sieve, have been developed to factor large composite numbers. However, there are no known algorithms that can factor a number this large (300 or 400 digits) efficiently and in a reasonable amount of time.
The RSA Encryption System and the Factorization of Large Numbers

Primes

Since this paper will make heavy use of modular arithmetic, this section will start out with a definition of what modular arithmetic is. Let three numbers, $a$, $b$, and $m$, be integers (Rosen 118). The definition of a congruence, written $a \equiv b \pmod{m}$, which means that $m$ divides $a - b$ (Rosen). Ancient Chinese mathematicians believed the following two conjectures were both true (Rosen 192).

(1) If $n$ is prime, then $n$ divides $2^{n-1} - 1$, or that $2^{n-1} \equiv 1 \pmod{n}$.
(2) If $2^{n-1} \equiv 1 \pmod{n}$, then $n$ must be prime.

However, if an integer $n$ divides $2^{n-1} - 1$, then $n$ is not necessarily prime (Rosen). If the last conjecture had been true, then this would have been a fast way to determine primality. By just finding $2^{n-1} \mod{n}$ and by using the fast exponentiation algorithm, to be discussed shortly, this congruence can be found quickly and efficiently, even for a large $n$. This would provide a very efficient primality test because only a simple congruence needs to be solved for the number being tested. This saves time and computer cycles, and would give an instantaneous answer that is one hundred percent reliable, instead of only an answer that may be 99.99999% reliable, as probabilistic algorithms can. Even though this accuracy is high, there is always the possibility that a composite number may be tested as being prime.

The first conjecture mentioned above, that for a prime number, $n$, $2^{n-1} \equiv 1 \pmod{n}$, is true because the conjecture is simply Fermat's Little Theorem, where $p = n$ and
The RSA Encryption System and the Factorization of Large Numbers

\( a = 2 \) (Rosen 192). Fermat's Little Theorem states that for any \( p \) that is prime, there exists the relationship \( a^{p-1} \equiv 1 \pmod{p} \) for every number \( a \) such that \( a \) and \( p \) are relatively prime to each other (Rosen). The main difference between Fermat's Little Theorem and the Chinese conjecture is that \( a \) can be any number that is relatively prime to \( p \), where as the Chinese conjecture requires that \( a \) be the number two (Rosen).

The second conjecture mentioned above, that if \( 2^{n-1} \equiv 1 \pmod{n} \) for a number \( n \), then \( n \) must be prime, is unfortunately false (Rosen 192). But the fact that this conjecture is false is not readily apparent. In fact, for the first 271 composite numbers, the relationship \( 2^{n-1} \equiv 1 \pmod{n} \), is false (Rosen 193). For example, \( 2^7 \equiv 0 \pmod{8} \) and \( 2^{339} \equiv 8 \pmod{340} \). However, the first counter example to this conjecture appears with the number 341 (Rosen 192). \( 2^{340} \equiv 1 \pmod{341} \), and since \( 341 = 11 \times 31 \), 341 is not prime, and therefore the second conjecture is proven wrong (Rosen 193). If this had been a correct idea, then testing for primes would be simple and fast.

**Pseudoprimes**

If \( 2^{340} \equiv a \pmod{341} \), where \( a \neq 1 \), then it would be immediately obvious that 341 is a composite number because of the first conjecture shown above (Rosen 193). This conjecture is known to be true because Fermat's Little Theorem states that if \( n \) is prime, then \( 2^{n-1} \equiv 1 \pmod{n} \) (Rosen 192). The contrapositive to this conjecture, which
The RSA Encryption System and the Factorization of Large Numbers

is logically equivalent to it, states that if $2^{n-1}$ is not congruent to 1 (mod $n$), then $n$ is
not prime (Rosen). However, $a$ is equal to one because as seen previously, $2^{340} \equiv 1$
(mod 341). So this congruence alone does not show that the number 341 is prime
(Rosen 193). Fermat's Little Theorem states that for a prime number $p$, $a^{p-1} \equiv 1$ (mod
$p$) for all $a$'s such that $a$ and $p$ are relatively prime (Rosen 192). It is clear that 3 is
relatively prime to 341, yet $3^{340} \equiv 56$ (mod 341), and therefore 341 is a composite
number, and 3 is called a witness to the compositeness of 341 (Rosen 193).

Here is a definition that will be used as part of an explanation:

(3) If $n$ is an odd composite number which is relatively prime to $a$ and
if $a^{n-1} \equiv 1$ (mod $n$), then $n$ is called a pseudoprime for the base $a$ (Rosen
193).

By using the definition (3), 341 can be called a pseudoprime with the base of 2. But,
it is not a pseudoprime for the base 3 since $3^{340}$ is not congruent to 1 (mod 341)
(Rosen). There is an important aspect to this definition: it can be shown that a
number is composite without actually finding the factors of that number. This may
not seem valuable, but the following example shows why this definition is
important. $2^{7926900} \equiv 7660171$ (mod 7926901), so therefore 7926901 is a composite
number, but none of the number's factors needed to be found to establish this fact.

So, in summary, if the base $a$ and the number $n$ are relatively prime, and if $a^{n-1}$ is not
congruent to 1 (mod $n$) for a base $a$, then $n$ is a composite number (Rosen 194). By
using this method, there are very large numbers that can be shown to be composite,
despite the fact that no factors may be known for these numbers.
Carmichael Numbers

There may appear to be a way to get a primality test out of the above method, but this is not true. Yet another special kind of number comes into play. Before investigating these numbers, a definition of what they are is important to state carefully. Start with an integer \( n \) and pick the smallest base value \( a \) so that \( a \) and \( n \) are relatively prime to each other (Rosen 194). Then, compute \( a^{n-1} \pmod{n} \). If \( a^{n-1} \equiv 1 \pmod{n} \), then pick the next base \( a \) value where \( a \) and \( n \) are relatively prime and re-compute the above congruence (Rosen). Keep repeating this procedure until the base value \( a \) reaches the integer value \( n \) (Rosen). If \( a \) reaches \( n \), and for every \( a \) where \( a \) and \( n \) are relatively prime, \( a^{n-1} \equiv 1 \pmod{n} \), then \( n \) would appear to be a prime number (Rosen). However, this is not the case. There exists such rare integers, called *Carmichael Numbers*, that test as being pseudoprimes for every base to which they are relatively prime (Rosen). The first of these numbers is 561, because \( 2^{560} \equiv 1 \pmod{561} \), \( 4^{560} \equiv 1 \pmod{561} \), \( 5^{560} \equiv 1 \pmod{561} \), and so on up until the last integer that is relatively prime to 561 (Rosen 195). These numbers are named after R.D. Carmichael and occur extremely rarely. For the first 25,000,000,000 numbers, there are only 2163 Carmichael Numbers (Bressoud 33). Despite their rarity, the existence of these numbers shows that the pseudoprime test that was outlined earlier cannot be used to prove or disprove the primality of a number.
Strong Pseudoprimes

A stronger test for determining if a number is prime was created by Pomerance, Selfridge, and Waystaff in 1980 (Bressoud 75). Suppose that there exists an integer, \( n \), and that this integer \( n \) passes the pseudoprime test for some base \( b \) such that \( b \) is relatively prime to \( n \) (Bressoud). Therefore, \( n \) divides \( b^{n-1}-1 \), or \( b^{n-1} \equiv 1 \pmod{n} \) (Bressoud). Assuming that \( n \) is an odd integer, \( n \) can be written as the relation \( n = 2^m + 1 \) for some integer \( m \) (Bressoud 76). So, \( n \) divides \( b^{n-1}-1 \), or \( n \) divides \( b^{2^m+1}-1 \), or \( n \) divides \( b^{2^m}-1 \), or since \( b^{2^m}-1 = (b^m-1) \times (b^{m+1}) \), \( n \) divides \( (b^m-1) \times (b^{m+1}) \) (Bressoud). Thus, if \( n \) really is a prime number, then \( n \) divides \( b^m-1 \) or \( b^{m+1} \) (Bressoud). Notice that the integer \( n \) cannot divide both of these quantities because if that were true, then \( n \) would also divide the difference of these two quantities, which is 2, and this is impossible because the number is assumed to be odd. In conclusion, if the integer \( n \) really is prime, then \( b^m \equiv 1 \pmod{n} \) or \( b^m \equiv -1 \pmod{n} \), but not both (Bressoud).

Suppose that the integer value \( n \) is actually a composite number. If this is true, then some factors of \( n \) could divide \( b^m-1 \) and some other factors of \( n \) may divide \( b^m-1 \) (Bressoud 76). However, if this were true, then that also means that the integer \( n \) would have satisfied the pseudoprime test, but it would not have satisfied the above two tests. For example, the first pseudoprime number, 341, is equal to 11 times 31. 341 is equal to \( 2 \times 170 + 1 \) (Bressoud). Therefore, the \( m \) value in this case is 170 (Bressoud). The congruence \( 2^{170} \equiv 1 \pmod{341} \) is easy to show, so the integer
341 is still looking prime. But, by using the two quantities outlined earlier, 341 would have to divide $2^{85}-1$ times $2^{85}+1$, which means that $2^{85} \equiv 1 \pmod{341}$ or $2^{85} \equiv -1 \pmod{341}$ (Bressoud). However, $2^{85} \equiv 32 \pmod{341}$, so the previous conjecture is false. This integer tested false because 341 equals 11 times 31 and $2^{85} \equiv -1 \pmod{11}$ and $2^{85} \equiv 1 \pmod{31}$ (Bressoud).

To generalize this test, suppose that an integer $n$ needs to be tested for primality. Suppose there is also another integer, $b$, such that $n$ is relatively prime to $b$ and that the congruence $b^{n-1} \equiv 1 \pmod{n}$ is true. This means that $n$ has passed the pseudoprime test for base $b$. Since $n$ has passed the pseudoprime test, $n$ can be written as $(2^a)^t+1$, where the integer $t$ is an odd integer and $a$ is at least 1 (Bressoud 76). Then, the relation $b^{n-1}-1$ is equal to $b^{2^{a_1}t}$, which equals $(b^t-1) \cdot (b^t+1) \cdot (b^{2t}+1) \cdot (b^{4t}+1) \cdot \ldots \cdot (b^{2^{a-1}t}+1)$ (Bressoud). If the integer $n$ really is prime, then $n$ will divide exactly one of the factors. For example, suppose that $a = 4$ and $t = 3$. Then, $b^{48} = (b^3-1) \cdot (b^3+1) \cdot (b^6+1) \cdot (b^{12}+1) \cdot (b^{24}+1)$. Therefore, if an integer $n$ is really prime, then that integer must divide only one of the above factors (Bressoud). However, if an integer passes this test, and is an odd composite integer, then the integer is called a strong pseudoprime (Bressoud). A strong pseudoprime for the base $b$ is defined to be an odd integer, $n$, that is composite, relatively prime to a base $b$, and divides one of the quantities of the right hand side of the equation (Bressoud). By using this
definition, a new, stronger prime test can be derived. This test is very efficient, and it will effectively run as fast as the original pseudoprime test.

Strong pseudoprimes are known to exist, but they are very rare. The number of strong pseudoprimes less than 25,000,000,000 for the base 2 is 4,842, compared with 21,853 pseudoprimes for the base 2 that are less than 25,000,000,000 (Bressoud 78). Since strong pseudoprimes are rarer than pseudoprimes, a test using strong pseudoprimes will be more effective than one that uses pseudoprimes. The integer 1,729 is the first pseudoprime that occurs for the bases 2, 3, and 5 (Bressoud). For all integers less than 25 billion, there are 2,522 pseudoprimes with these bases (Bressoud). The integer 25,326,001 is the first strong pseudoprime that occurs for the bases 2, 3, and 5, with there being 13 strong pseudoprimes less than 25 billion (Bressoud). Therefore, by adding another base, 7, to the bases 2, 3, and 5, the pseudoprime test becomes even better because only 1,770 pseudoprimes exists for these bases that are less than 25 billion, and there exists only one strong pseudoprime, 3,215,031,751 (Bressoud). By using more bases, a more stable primality test for numbers below 25 billion can be utilized: if an integer $n$ passes the strong pseudoprime test for bases 2, 3, 5, and 7, and $n$ is less than 25 billion, and $n$ does not equal the only strong pseudoprime less than 25 billion, then $n$ is prime (Bressoud). However, there are much simpler ways of checking to see if such a small number is prime or not.
The RSA Encryption System and the Factorization of Large Numbers

One good attribute of strong pseudoprimes is that there does not exist any Carmichael type numbers in the set of strong pseudoprimes (Bressoud 78). In other words, if an integer passes the strong pseudoprime test for all bases less than the integer that are relatively prime to the integer, then that number is prime. There are no composite numbers that could pass the test for more than half of the bases that are less than the number. It can even be shown that all composite numbers will fail the strong pseudoprime test for at least one fourth of the bases less than itself (Bressoud). This, in theory, gives an idea for a primality test: if an integer, \( n \), fails the strong pseudoprime test for more than half of the bases less than \( n \), then the integer is composite (Bressoud). Even though this test sounds good, it is too impractical, in that it would take much too long to test all the bases that are less than or equal to \( n \) for large \( n \) (Bressoud).

**Euler’s Criterion**

In order to show that a number is composite when it fails a strong pseudoprime test, some more information is needed. It was shown earlier that if an integer \( n \) is an odd prime, then \( n = 2^m + 1 \), and \( n \) divides either \( b^{m-1} \) or \( b^{m+1} \), but not both. However, which one does it divide? This question will be partially answered in the following section. This answer was discovered by the great mathematician Euler.
The RSA Encryption System and the Factorization of Large Numbers

The theorem that determines how to tell which quantity is divided by \( n \) is called Euler's Criterion, which can be stated as follows (Bressoud 79). Let the integer \( p \) be prime and odd (Bressoud). Since \( p \) is odd, then \( p = 2 \cdot m + 1 \) (Bressoud). Also, let the variable \( b \) be another integer that is positive, so that \( b \) and \( p \) are relatively prime (Bressoud). Then \( b^m \equiv 1 \pmod{p} \) if and only if there is another integer \( t \) that exists so that \( b \equiv t^2 \pmod{p} \) (Bressoud).

This will be proved in only one direction: if there is an integer value \( t \) such that \( b \equiv t^2 \pmod{p} \), and \( b \) and \( p \) are relatively prime, then \( b^m \equiv 1 \pmod{p} \) (Bressoud 79). The proof starts by assuming that such a \( t \) value does exist (Bressoud). Therefore, \( b^m \equiv (t^2)^m \equiv t^{2m} \), and since the odd prime integer \( p \) equals \( 2 \cdot m + 1 \), \( 2 \cdot m = p - 1 \) (Bressoud). This yields the relation \( b^m \equiv t^{p-1} \pmod{p} \), which means that \( p \) does not divide \( t \) (Bressoud). If \( p \) did divide \( t \), then it would also have to divide \( t^2 \). But, \( b \equiv t^2 \pmod{p} \), so that \( p \) divides \( b \), and this produces a contradiction, since it was assumed that \( b \) and \( p \) were relatively prime (Bressoud). In conclusion, \( p \) does not divide \( b \), so \( p \) and \( b \) are relatively prime and therefore, by utilizing Fermat's Little Theorem, \( b^m \equiv t^{p-1} \equiv 1 \pmod{p} \) (Bressoud).

To continue on with Euler's Criterion, a new definition needs to be stated. This new definition is for quadratic residues modulo \( p \). An integer \( n \) is said to be a quadratic residue modulo \( p \) if \( p \) is a prime modulus and \( n \) is any integer that is relatively prime to \( p \), and there exists an integer \( t \) such that \( n \equiv t^2 \pmod{p} \) (Bressoud).
Quadratic residues are fairly simple to find. All that needs to be done is to square all the positive integers \( n \) less than the integer \( p \) and to check the residues of the results (Bressoud). For example, let \( p = 11 \). Therefore, \( 1^2 \equiv 1 \pmod{11} \), \( 2^2 \equiv 4 \pmod{11} \), \( 3^2 \equiv 9 \pmod{11} \), \( 4^2 \equiv 5 \pmod{11} \), \( 5^2 \equiv 3 \pmod{11} \), \( 6^2 \equiv 3 \pmod{11} \), \( 7^2 \equiv 5 \pmod{11} \), \( 8^2 \equiv 9 \pmod{11} \), \( 9^2 \equiv 4 \pmod{11} \), and \( 10^2 \equiv 1 \pmod{11} \). Therefore, the quadratic residues modulo 11 are 1, 3, 4, 5, and 9. It is possible to know how many residues a modulus \( p \) will have. Since \( i^2 = (-i)^2 \), the number of quadratic residues must be less than or equal to \((p - 1) / 2\) (Bressoud 80). Because \( i^2 \equiv j^2 \pmod{p} \), \( p \) must divide \( i - j \) or \( i + j \) (Bressoud). This means that each quadratic residue exists twice, and therefore the number of positive quadratic residues must be less than or equal to \((p - 1) / 2\) (Bressoud).

This next theorem, called Wilson's Theorem, is used to show the second half of Euler's Criterion. This theorem says that an integer \( n \) divides the quantity \((n - 1)! + 1\) if and only if \( n \) is a prime integer (Bressoud 80). The proof for this theorem is omitted. Now, assume that \( n \) is prime. If an integer \( i \) is less than \( n \), then \( i \) and \( n \) are relatively prime, and it is known that \( i \) must have a multiplicative inverse (Bressoud). This means that there exists an integer \( j \) such that \( i \cdot j \equiv 1 \pmod{n} \) (Bressoud). By assuming that \( i \neq 1 \) and that \( i \neq n - 1 \), then \( i \) and \( j \) must be two different integers because if they were the same number, then \( i^2 \equiv 1 \pmod{n} \) (Bressoud). This means that \( n \) would divide \((i - 1) \cdot (i + 1)\), or that \( n \) would divide \( i - 1 \) or \( i + 1 \), but not both (Bressoud). However, this is impossible because \( n \) is a prime
The RSA Encryption System and the Factorization of Large Numbers

integer, and the value \( i \) is less than \( n - 1 \). Therefore, when the product \( (n - 1)! \) is taken, the integers from 2 to \( n - 2 \) can be paired with their multiplicative inverses to get \( (n - 1)! = 1 \times \text{(product of 1s)} \times (n - 1) \equiv n - 1 \equiv -1 \) (mod \( n \)) (Bressoud). So, the factorial \( (n - 1) \equiv -1 \) (mod \( n \)) or \( n \) divides the quantity \( (n - 1)! + 1 \) (Bressoud). This result can be used to finish the Euler's Criterion proof.

Euler's Criterion, as stated above, is that if \( b^m \equiv 1 \) (mod \( p \)), then an integer \( t \) exists such that \( b \equiv t^2 \) (mod \( p \)) (Bressoud 79). This is equivalent to saying that if \( b \) is not a quadratic residue of modulo \( p \), then \( b^m \) is not congruent to 1 (mod \( p \)), or that \( b^m \equiv -1 \) (mod \( p \)) (Bressoud). The proof begins by noting that for every positive integer that is less than the modulus \( p \), there exists a unique positive integer \( j \), which is also less than \( p \), such that \( i \times j \equiv b \) (mod \( p \)) (Bressoud 81). This is true because another integer \( i' \) can be found such that \( i \times i' \equiv 1 \) (mod \( p \)) (Bressoud). Therefore, \( i \times i' \times b \equiv b \) (mod \( p \)) and \( i \times j \equiv b \) (mod \( p \)), where \( j = i' \times b \). If \( i \times j \equiv i \times k \equiv b \) (mod \( p \)), then \( (i \times j) \times j \equiv (i \times k) \times j \equiv b \times j \) (mod \( p \)), and therefore \( j \equiv k \equiv b \times j \) (mod \( p \)) (Bressoud). This means that the integer \( j \) is unique to the modulus \( p \). Because \( b \) is assumed to not be a quadratic residue, then \( i \) can never be equal to \( j \) (Bressoud). So, by taking all the positive integers that are less than the modulus \( p \) and pairing them with their multiplicative inverses whose products are \( b \) (mod \( p \)), this equation is created: \( (p - 1)! \equiv b \times b \times b \times \ldots \times b \) (mod \( p \)) \( \equiv b^m \) (mod \( p \)) (Bressoud). Therefore, \( (p - 1)! \) is congruent to \( b^m \) (mod \( p \)) (Bressoud). However, as stated previously, Wilson's Theorem puts forth the relation \( (p - 1)! \equiv -1 \) (mod \( p \)), and therefore \( b^m \equiv -1 \) (mod \( p \)), or that \( p \) divides
The RSA Encryption System and the Factorization of Large Numbers

$b^m + 1$, the result of which is that $p$ does not divide $b^m - 1$ (Bressoud). The results from this proof can be used to show that there are no strong Carmichael Numbers, but the proof is omitted.
The RSA Encryption System and the Factorization of Large Numbers

Explanation of the RSA Algorithm

RSA is one of the most powerful encryption algorithms available today. It is based on a method called modular exponentiation. The result of an exponentiation operation is taken modulo the product of two large primes. The person who is encoding the message has a key, one part of which consists of a composite number that is the product of two large primes. These two large primes might be 150 to 200 or more digits in length. These primes are relatively easy to produce if probabilistic methods are used to test their primality.

The primary security of this algorithm is based on the combination of these two large prime numbers (Bressoud 44). If a person attempting to compromise the security of a message tries to decode it, they would be unable to. Although they know the key used to encode the message, there is no quick method for finding the prime factors of the large composite number (Bressoud 45). However, there is a method open to a person trying to decode an RSA encoded message. This method is the repeated encoding of messages that are similar to the suspected contents of the encoded message. If the person has a reasonable idea of what the message could contain, then they could encode message after message, using the encryption key $e$ and the large composite number $n$. If one of the attempted encodings matches the original encoded message, then the person will know what the original message was. This method will only work well if the person trying to decode the message
The RSA Encryption System and the Factorization of Large Numbers

generally knows what the message is. Otherwise, the chances of accidentally decoding the message are as good as factoring the composite number. Also, this method will not yield the two large prime numbers, so it will work for only the message being compromised. Of course, things change, and a message that is secure today, may not be so tomorrow. This is why the prime numbers must be so large. When the algorithm was first conceived, two 50-digit prime numbers would have worked fine. Since computing power has increased steadily in the last 30 years, the number of digits that the prime numbers must be has also had to increase. A 100-digit composite number can be factored much faster than a 400-digit composite number.

An overall description of how the RSA algorithm works will help to make some aspects more clear. To begin with, two prime numbers, each generated randomly and each 150 digits or more need to be chosen (Bressoud 46). This can be done quickly and efficiently using a program like Mathematica. Second, calculate the value \( n \), which is the product of the two primes found earlier (Bressoud). The result is a large composite integer that contains between 300 and 400 digits, depending on how large the original prime numbers were. After calculating \( n \), the Euler \( \phi \) function of \( n \) needs to be calculated (Bressoud). This function is defined to be the number of numbers less than or equal to \( n \) which are relatively prime to \( n \) (Bressoud 35). It also has the multiplicative property. If \( p \) and \( q \) are relatively prime, then \( \phi(p^*q)=\phi(p)\phi(q) \).

Since \( p \) and \( q \) are prime, \( \phi(p)=p-1 \), and \( \phi(q)=q-1 \) and thus \( \phi(p^*q) \) must equal \( \phi(p)\phi(q) \),
The RSA Encryption System and the Factorization of Large Numbers

which in turn is equal to \((p-1)(q-1)\). Therefore, \(\phi(n) = \phi(p^*q) = \phi(p) \cdot \phi(q) = (p-1)(q-1) = p^*q - p - q + 1 = n - p - q + 1\). Then, using the \(\phi(n)\) value, find a value, \(e\), such that \(e\) and \(\phi(n)\) are relatively prime, or \(\text{GCD}(e, \phi(n)) = 1\) (Bressoud 46). Also, \(e\) needs to be less than \(\phi(n)\) (Bressoud). After finding \(e\), the inverse of \(e\) modulo \(\phi(n)\) needs to be found. This value can be obtained by using the Euclidean Algorithm on the \(e\) value and \(\phi(n)\) (Bressoud). This is known to exist because \(e\) was chosen so that \(e\) was relatively prime to \(\phi(n)\) (Bressoud). Since this algorithm is a public key algorithm, the composite number \(n\) and \(e\) can be publicly displayed without the security being compromised (Bressoud 43).

When these numbers are calculated, a message can be sent using them. First, the person who desires to send a secret message obtains the public key, consisting of the composite number, \(n\), and the value, \(e\) (Bressoud 43). Then, assuming that \(P\) stands for "plaintext" and \(C\) stands for "ciphertext", the sender encodes their message by using this equation: \(C = P^e \pmod{n}\) (Bressoud 44). After encoding the message, the ciphertext is sent via any communication channel desired. The sender is secure knowing that even if the message is intercepted, the message is still unreadable because \(n\), which is \(p^*q\), is unfactorable, using today's methods (Bressoud, 46). The encoded message, \(C\), is decoded by using the inverse of the encoding equation: \(P = (P^e \pmod{n})^{-1} = C^d \pmod{n}\) (Bressoud 44). The inverse equation works because only the receiver knows what \(p\), \(q\), and \(d\) are, and these
The RSA Encryption System and the Factorization of Large Numbers

numbers are not obtainable by any person who only knows the public keys $n$ and $e$ and the encoded message (Bressoud 44).

**Encoding**

Before using the RSA algorithm, the message must be converted to numbers (Rosen 260). Two methods for converting a message to a number are generally used. They are essentially the same, except for one part: where the counting starts. In one method, the counting starts at 0, so that A=0, B=1, C=2, etc. In the other method, the counting starts at 1, so that A=1, B=2, C=3, etc. For the example used in this paper, the first counting method is used.

Prior to the conversion, a couple of decisions must be made. First of all, two prime numbers must be chosen (Rosen 260). For this example, 43 and 59 are the two primes that will be used because this will allow the RSA system to be conveniently displayed (Rosen). However, in the appendix D and F, there is code for doing this with arbitrarily large primes. Also, the special number often referred to in RSA literature as $e$ must also be chosen (Rosen). This number must be selected such that the greatest common divisor (GCD) of $e$ and $\phi(n)$ equals 1, where $n$ is the product of the two primes, $p$ and $q$, and the $\phi(n)$ is the Euler $\phi$ function (Rosen). Therefore, as explained earlier, $e$ is chosen so that the GCD of $e$ and $(p-1)*(q-1)$ must equal 1 (Rosen). In this example, $n$ equals 2537, because $n = p * q = 43 * 59$, and $\phi(n)$ equals 2436 (Rosen). A value of 13 will be assigned to $e$, since $(e, \phi(n)) = 1$ is 13 (Rosen).
To begin the example, the simple message "PUBLIC KEY CRYPTOGRAPHY" will be used (Rosen). Converting these letters into numbers yield this result: 15 20 01 11 08 02 10 04 24 02 17 24 15 19 14 06 17 00 15 07 24 (Rosen). If a letter converts to only a single digit number, then a 0 is added to the front to make it consistent with the other letters that are greater then 9. To simplify this chunk of numbers, the message is broken into blocks of 4 digits, which represent 2 letters total (Rosen). In case of an odd number of letters, a dummy letter of X (23) is added to the end of the message (Rosen). This is what the separation method yields: 1520 0111 0802 1004 2402 1724 1519 1406 1700 1507 2423 (Rosen).

After converting the letters to numbers, the numbers are plugged into a formula. The formula used here is \( C = P^{13} \) (mod 2537), where \( C \) is the encoded ciphertext, \( P \) is the original plaintext, 13 is the \( e \) value, and 2537 is the \( n \) value, which is the product of the two primes (Rosen). The two values, \( e \) and \( n \), are the public keys and can be looked up by anyone. By taking each plaintext block to the \( e \) value and finding its remainder when dividing it by \( n \), a number is obtained that is the encoded cipher block (Rosen 261). By running all the plaintext blocks through this equation, the following numbers are obtained: 0095 1648 1410 1299 0811 2333 2132 0370 1185 1457 1084 (Rosen). This is the message that would be transmitted to the receiving end. To see an example of this algorithm, refer to Appendix D, F, and G in the Appendices section.
Decoding

Decoding the message, which means recovering the original plaintext message, can be accomplished quite rapidly, assuming that the decryption key, \( K_D(n,d) \), is known. As noted, the decryption key, \( d \), is the inverse of \( e \) modulo \((p-1)*(q-1)\) (Rosen 261). The \( d \) value is known to exist because the value of \( e \) was chosen so that the GCD of \( e \) and \((p-1)*(q-1)\) equals 1 (Rosen). Also, it is known that \( a*x \equiv b \pmod{m} \) has a solution if \( a \) and \( m \) are relatively prime (Rosen).

To find the value of \( d \) that is required to decrypt the message, the receiver carries out the following procedure. First, recalling that \( e \) was chosen at random, with the restrictions that \( e \) must be less than \( \phi(n) \) and that \( (e, \phi(n)) = 1 \), the multiplicative inverse of \( e \) (mod \( \phi(n) \)) can then be calculated (Rosen). As stated earlier, this value is known to exist, and it was called \( d \), and therefore \( d*e \equiv 1 \pmod{(p-1)*(q-1)} \) (Rosen). Another way of writing this equation is by stating that an integer, \( k \), exists such that \( d*e = 1+k(p-1)*(q-1) \) (Rosen).

Looking at the equation \( C = P^e \pmod{n} \), a similar equation can be derived from it by using the results outlined above (Rosen 261). Because the greatest common divisor of \( e \) and \((p-1)*(q-1)\) is 1, then there exists an integer \( k \) such that \( d*e = 1+k(p-1)*(q-1) \). This means that \( C^d = (P^e)^d = P^{ed} = P^{I+k(p-1)*(q-1)} = P*P^{k(p-1)*(q-1)} \) (Rosen). But, note that the GCD(P,p) and the GCD(P,q) nearly always equal one because the
only time that they would not equal one is when the value \( P \) is a multiple of either \( p \) or \( q \), and since \( p \) and \( q \) are large primes, it would be highly unlikely that this would happen (Rosen). Therefore, by using Fermat’s Little Theorem, the following two relations can be derived: \( P^{p-1} \equiv 1 \pmod{p} \) and \( P^{q-1} \equiv 1 \pmod{q} \), and hence \( C^d \equiv P^*(P^{p-1})^k \equiv P^*1 \equiv P \pmod{p} \) and \( C^d \equiv P^*(P^{q-1})^k \equiv P^*1 \equiv P \pmod{q} \) (Rosen). Finally, by using the Chinese Remainder Theorem, the equations used to decode a message is found: \( C^d \equiv P \pmod{p \times q} \) (Rosen).

Using the Euclidean algorithm, the value of \( d \) in this example is calculated as 937 (Rosen 261). Since the value \( d \) is now known, decoding the message is similar to encoding it. First, the ciphertext blocks are run through the following equation: \( P \equiv C^{927} \pmod{2537} \), where \( P \) will be the plaintext block, \( C \) is a ciphertext block, 937 is \( d \), the inverse value found earlier, and 2537 is the product of the two primes, also called \( n \) (Rosen). After running the ciphertext blocks through this equation, the following numbers are obtained: 1520 0111 0802 1004 2402 1724 1519 1406 1700 1507 2423.

Converting the numbers back to letters, using the counting method discussed above (A=0, B=1, C=2, etc.), the message “PUBLICKEYCRYPTOGRAPHYX” is found. Since spaces were not encoded, they were dropped during the encoding process. Also, the “X” on the end indicates a dummy character that was used to make an even number of characters and can be discarded.
The RSA Encryption System and the Factorization of Large Numbers

The security of this system lies in the fact that the value $n$ cannot be factored quickly. However, all of the security does not just rest on the unfactorability of the composite number (Rosen 262). By looking one step above factoring the composite number, it can be seen that to find $d$, the inverse value of $e$, any person attempting to compromise a message would only have to find the $\phi(n)$ (Rosen). But, $n = p^*q$ and $p$ and $q$ are relatively prime, so therefore $\phi(n) = \phi(p^*q) = \phi(p)\phi(q) = (p-1)(q-1) = p^*q - (p+q) + 1$ (Rosen). This means that finding $\phi(n)$ is just as difficult as factoring $n$, and since it would take a long time to factor $n$ using current algorithms, the security of the RSA algorithm is safe (Rosen). However, if a person did find a quick way to find $\phi(n)$, a method that did not depend on $p$ and $q$, then the RSA algorithm’s security would be broken, and the system would be rendered useless (Rosen). To view a demonstration of this algorithm with larger numbers, and to see it in code, refer to sections E, F, and G in the Appendices section.
The RSA Encryption System and the Factorization of Large Numbers

Factoring

The problem of quickly factoring a large, composite number is what gives RSA its security. Most current factoring techniques require weeks of computer time to factor a one hundred digit number. To factor a three to four hundred digit composite number could take years even if the fastest existing computer were working on it. Although there are no known fast algorithms for factoring large composite numbers, there are several techniques for finding the factors of smaller composite numbers. In the next section, several of the known factoring algorithm will be looked at.

Trial Division

This method for finding factors of a number is the simplest (Bressoud 20). However, the composite numbers it works on cannot exceed about one million (Bressoud). The basic idea of the algorithm is that if the number being factored is not a prime number, then one of the number’s factors must be less than the ceiling of the square root of the number (Bressoud 21). This is true because if a number had more than one factor that was greater than its square root, then these factors multiplied together would be greater than the original number (Bressoud). In a simple version of this algorithm, every number between two and the square root of the composite number are divided into the number (Bressoud). If the trial number divides the composite number, then it must be a factor (Bressoud).
This method does work, but there are faster ways of achieving the same results. One way is to see if the numbers two and three can divide the composite number (Bressoud 21). If it can be, then all even numbers and numbers that are multiples of three between one and the square root of the trial number can be eliminated (Bressoud). Another way of streamlining this process is to keep a list of all primes less than or equal to one thousand (Bressoud). Since only prime factors are generally wanted, trying only prime numbers can reduce the number of trial divisions (Bressoud). There are one hundred sixty-eight primes less than one-thousand, so doing this would make the algorithm more efficient (Bressoud 21). However, this would have no effect on large $n$ values.

Programming Trial Division into nearly any language is possible because the algorithm is relatively simple and the algorithm does not involve large numbers. In this paper, the Trial Division algorithm was programmed into two languages: Mathematica, which is a software package that is specifically designed for math, and Microsoft Visual Basic, which is a language that is often not too math friendly. Two different languages were used in order to compare the different programming techniques and to show that this algorithm could be used in any number of languages. Only the Mathematica version is included in this paper.
The RSA Encryption System and the Factorization of Large Numbers

This algorithm starts off by inputting a composite number to be factored and the highest number to be tried as a factor (Bressoud 21). After getting this information, the algorithm first tests the number to see if it is divisible by two, and then by three (Bressoud). If it is, then the original composite number is divided by two and three over and over again until the number can no longer be divided by two and three (Bressoud). The number of times two and three goes into the composite number is stored into an array, and the scaled down number is continued on into the algorithm (Bressoud 22). After trying these two factors, the algorithm sets the next potential factor as five and starts a “for loop” from five to the maximum number specified at the beginning (Bressoud). When the loop reaches the end, the factors and the number of times they divided the original composite number are outputted (Bressoud). To see this algorithm in code, refer to Appendix K.

Fermat’s Algorithm

Fermat’s Algorithm is better than trial division when the composite number that will be factored is greater then eleven digits long (Bressoud 58). This algorithm works most efficiently when the composite number has two factors near the square root of the composite number (Bressoud). This algorithm does not completely break a number into its factors. Instead, it will display the composite number as a product of two of its factors (Bressoud 59). Then, a primality test can be conducted on these numbers (Bressoud). If one or both of the numbers fails the test, then those factors
The RSA Encryption System and the Factorization of Large Numbers

must be composite and can also be run through Fermat’s Algorithm (Bressoud). This can continue until all the prime factors are found (Bressoud). If a visual representation were constructed of this process, then this visual would resemble a tree, with the interior leaves being the composite factors, and the childless nodes being the prime factors.

Unlike Trial Division, Fermat’s Algorithm starts by looking for factors near the square root of the composite number, instead of starting at one and working up to the square root of the composite number (Bressoud 58). The basic equation used in this factoring technique is \( x^2 - y^2 = n \), where \( n \) is the number being factored and \( x \) and \( y \) are two trial factors (Bressoud). To begin with, a value of \( x \) is chosen that is relatively near the square root of the number to be factored (Bressoud 59). Then, \( y \) is increased until the quantity is either equal to or less than the value \( n \), which is the number being factored (Bressoud). If the case is that \( y \) equals \( n \), then the factoring is over, and \( x \) and \( y \) are returned as the factors of \( n \) (Bressoud). However, if the quantity is less than \( n \), then the \( x \) value chosen is incremented by one, and the process is repeated (Bressoud). These iterations are continued until two factors are returned (Bressoud).

This algorithm is nice in that no multiplication or division is used in the looping of the \( x \) and \( y \) values (Bressoud 60). However, the main disadvantage of this algorithm is that it can take many loops for finding the factors of a large
The RSA Encryption System and the Factorization of Large Numbers

composite number, sometimes climbing into the tens of thousand (Bressoud). This many loops will slow down the algorithm (Bressoud). Also, since the algorithm only returns two factors of the number, which may or may not be prime, additional tests must be conducted on the two factors to determine if they need to be factored, too (Bressoud 59). If they do, then the algorithm must be run on them, also (Bressoud 59). These checks and reiterating of the algorithm are not actually part of the original algorithm, so they must be programmed separately (Bressoud 60).

It is possible to also program Fermat’s algorithm, just like Trial Division, but if the language being used does not support numbers over two billion, then two billion is the maximum number that can be factored. Mathematica and Visual Basic were also used to program Fermat’s Algorithm. Since Visual Basic has a maximum integer size of about two billion, then its scope of possible numbers to factor is limited. However, Mathematica can go far higher than two billion, so Mathematica can factor any number up to the maximum size that Fermat’s algorithm can do in a reasonable amount of time. To see the Mathematica coded version of this algorithm, see Appendix L in the Appendices section.

Pollard’s Rho

John Pollard, in 1974, came up with a method for efficiently find moderately sized factors, about 20 or 21 digits, of large composite numbers (Burton 338). The first step is to choose a simple polynomial, whose degree is no more than two, with
The RSA Encryption System and the Factorization of Large Numbers

integer coefficients (Burton). For example, \( f(x) = x^2 + a \), where \( a \neq 0 \) or -2 (Burton). The second step involves picking a random starting value, \( x_0 \), and creating a sequence \( x_1, x_2, x_3, \ldots \), from the congruence \( x_{k+1} \equiv f(x_k) \mod n \) for values of \( k = 0, 1, 2, \ldots \) (Burton).

The third step is the main part of the method. Let an integer, \( d \), be a non-trivial divisor of \( n \), where \( d \neq 1 \) and \( d \neq n \), and let \( d \) be small in comparison to \( n \) (Burton). Since \( d \) is smaller than \( n \), then there may exist two values, \( x_j \) and \( x_k \), from the sequence generated in step two such that \( x_k \equiv x_j \mod d \), but \( x_k \) is not congruent to \( x_j \mod n \) (Burton). This holds true because there are many more congruence classes \( \mod n \) then there are \( \mod d \). Because \( d \) divides \( x_k - x_j \), and \( n \) does not divide \( x_k - x_j \), then the greatest common divisor of \( x_k - x_j \) and \( n \) is also a non-trivial divisor of \( n \) (Burton).

For this method to work, it is not necessary to know what \( d \) is in advance (Burton 338). Instead, by continuously computing the greatest common divisor of \( x_k - x_j \) and \( n \) until a non-trivial divisor is obtained, \( d \) can be found (Burton 339). However, this \( d \) may not be the smallest factor of \( n \), and it is sometimes not prime (Burton). Also, it can happen that \( d \) could turn out to be \( n \), so that \( x_k \equiv x_j \mod n \) (Burton). This only happens on rare occasions, and changing the \( x_0 \) value or changing the polynomial are two ways to attempt to remedy this problem (Burton).
The RSA Encryption System and the Factorization of Large Numbers

One problem that is associated with this method is that as \( k \) increases, calculating the greatest common divisor of \( x_k - x_j \) and \( n \) for each \( j \) that is less than \( k \) can take increasingly larger amounts of time (Burton 339). By looking at only the cases where \( k = 2^j \), the number of checks can be reduced by about half (Burton). This is not an unreasonable expectation, for it can be shown that in integer \( k \) exists such that 1 is less than the greatest common divisor of \( x_{2^k} - x_k \) and \( n \), which is in turn less than \( n \) (Burton). The only problem with using the \( k=2^j \) cases is that it may not find the first time the greatest common divisor of \( x_i - x_j \) and \( n \) is a non-trivial divisor of \( n \) (Burton).

To demonstrate this method, an example will be shown. Let \( n = 30623 \), \( x_0 = 3 \), and the polynomial be \( x^2 - 1 \) (Burton 339). The following sequence is generated by using the relation discussed earlier: \{8,63,3968,4801,21106,28526,18319,18926,...\} (Burton). By taking the \( k = 2^j \) cases from this sequence, it can be seen that \( x_2 - x_1 = 63 - 8 = 55 \), \( x_4 - x_2 = 4801 - 63 = 4738 \), \( x_6 - x_3 = 28526 - 3968 = 24558 \), and \( x_8 - x_4 = 18926 - 4801 = 14125 \) (Burton). By taking the greatest common divisor of these four final values and \( n \), three 1's and 113 are calculated, respectively (Burton). The 113 represents a factor of \( n \), since \( 113 \times 271 = 30623 \) (Burton). By reducing the numbers in the original sequence by the modulus 113, a new sequence is derived: \{8, 63, 13, 55, 86, 50, 13, 55, 86, 50,...\} (Burton). The four integers 13, 55, 86, and 50 are periodic in this sequence. Therefore, by making a graphical representation, where the non-periodic numbers represent a tail, and the periodic numbers represent a circle, the Greek letter, \( \rho \),
The RSA Encryption System and the Factorization of Large Numbers

called rho, is created (Burton 340). This is where the method received it name from (Burton).

This factoring method was the simplest to code, next to the trial division method. The ideas behind it are easy and straightforward, and are simple to program into Mathematica. To see a coded program of Pollard's Rho factoring algorithm, refer to Appendix X.

Quadratic Reciprocity

A definition will be used to start off this section. If an integer $n$ is a non-zero positive integer, and an integer $a$ is relatively prime to $n$, then $a$ is a quadratic residue of $n$ if the congruence $x^2 \equiv a \pmod{n}$ has a solution (Burton 181). If the congruence does not have a solution, then $a$ is a quadratic nonresidue of $m$ (Burton). This leads up to the fact that if an integer $p$ is an odd prime integer, then for $p$ there exists as many quadratic residues as quadratic nonresidues for $a$'s equal to 1, 2, 3, ..., $p-1$ (Burton).

The following two theorems state some attributes of quadratic residues. The first theorem states, if an integer $p$, which is odd and prime, and an integer $a$ exist such that $p$ does not divide $a$, then the congruence $x^2 \equiv a \pmod{p}$ has exactly two solutions or no solutions that are not congruent to the modulus $p$ (Burton 180). The proof is straightforward. If an integer $x_0$ is a solution to the congruence $x^2 \equiv a \pmod{p}$, then clearly $-x_0$, is another solution (Burton). Also, $x_0$ is not congruent to $-x_0$.
(mod p) because if it were congruent, then that would mean that p would divide 2 * x₀, but p is an odd integer, and a and p are relatively prime, so this cannot be true (Burton). This shows that there exists at least two non-congruent solutions to a modulus p. Could more exist? Suppose that two integers x₀ and x₁ exists and that they both solve the congruence \( x^2 \equiv a \pmod{p} \) (Burton). Therefore, \( x₀^2 \equiv x₁^2 \equiv a \pmod{p} \), and so \((x₀ + x₁) \cdot (x₀ - x₁) \equiv 0 \pmod{p}\) (Burton). As a result of this, p divides \((x₀ + x₁)\) or p divides \((x₀ - x₁)\), but not both (Burton). Also, this means that either \(x₀ = -x₁\) or that \(x₀ = x₁\), but not both (Burton). By using this result, the following theorem can be proven: if p is an integer that is prime and odd, then p has exactly \((p - 1) / 2\) quadratic residues and exactly \((p - 1) / 2\) quadratic nonresidues among the integers less than or equal to \(p - 1\) (Bressoud 80). This has already been proven earlier, so the proof is omitted here.

Now the Legendre Symbol will be defined. Let an integer p be odd and prime, and let another integer a be not divisible by p (Burton 185). Then the Legendre Symbol, denoted as \(\left(\frac{a}{p}\right)\) and also as \(a/p\), is defined to be 1 if a is a quadratic residue of p, and −1 if a is a quadratic nonresidue of p (Burton). For example, \(\left(\frac{1}{11}\right) = \left(\frac{3}{11}\right) = \left(\frac{4}{11}\right) = \left(\frac{5}{11}\right) = \left(\frac{9}{11}\right) = 1\), and \(\left(\frac{2}{11}\right) = \left(\frac{6}{11}\right) = \left(\frac{7}{11}\right) = \left(\frac{8}{11}\right) = \left(\frac{10}{11}\right) = -1\).

Euler came up with a criterion that gives an efficient way to find the quadratic residues of prime numbers. Euler's Criterion states that if an integer p is prime and odd and that if an integer a exists that is positive, and p does not divide a,
The RSA Encryption System and the Factorization of Large Numbers

then \((a/p) \equiv a^{\frac{p-1}{2}} \pmod{p}\) (Burton 181). To prove Euler's Criterion, assume that \((a/p) = 1\) and that \(x^2 \equiv a \pmod{p}\) has a solution \(x_0\) (Burton). Therefore \(x_0^2 \equiv a \pmod{p}\), and so \(a \equiv x_0^2 \pmod{p}\) (Burton). Now, Fermat's Little Theorem says \(a^{p-1} \equiv 1 \pmod{p}\), and so \(a^2 \equiv (x_0^2)^{\frac{p-1}{2}} = [(x_0^2)^{\frac{1}{2}}]^{p-1} = x_0^{p-1} \equiv 1 \pmod{p}\), so if \((a/p) = 1\), then \((a/p) \equiv a^{\frac{p-1}{2}} \pmod{p}\) (Burton). Now, suppose the opposite is true and that \((a/p) = -1\) and that \(x^2 \equiv a \pmod{p}\) has no solutions (Burton). As known from before, for each positive integer \(i\) that is greater than 1 but less than \(p - 1\), there exists a unique integer \(j\) that is also positive and less than \(p - 1\) so that \(i \cdot j \equiv a \pmod{p}\) (Bressoud 80). The integers \(i\) and \(j\) cannot be equal because of the fact that \(x^2 \equiv a \pmod{p}\) has no solution, therefore the positive integers that are less than \(p - 1\) can be paired off into \(\frac{p-1}{2}\) pairs whose product will be \(a\) (Burton). So, \((p - 1)! \equiv a^{\frac{p-1}{2}} \pmod{p}\). Wilson's theorem, as stated earlier, says that \((p - 1)! \equiv -1 \pmod{p}\), and so now the equation becomes \(-1 \equiv a^{\frac{p-1}{2}} \pmod{p}\), which means that \((a/p) \equiv a^{\frac{p-1}{2}} \pmod{p}\) (Burton 182). As an example of this last part, let an integer \(p = 23\), and \(a = 5\). As shown above, \(5^{\frac{23-1}{2}} = 5^{11} \equiv -1 \pmod{23}\). This means that 5 is a quadratic residue of the modulus 23. If \(p = 23\) and \(a = 6\) instead, then \(6^{11} \equiv 1 \pmod{23}\), and so this means that 6 is also a quadratic residue of the modulus 23. The fact that \(11^2 \equiv 6 \pmod{23}\) shows that 6 is a quadratic residue of 23.
The RSA Encryption System and the Factorization of Large Numbers

The following three statements are properties of the Legendre Symbol with the constraints of \( p \) being an odd prime integer, and \( a \) and \( b \) being integers that are not divisible by \( p \).

1. If \( a \equiv b \pmod{p} \), then \( (a/p) = (b/p) \) (Burton 186)
2. \( (a/p) \times (b/p) = ((a \times b)/p) \) (Burton)
3. \( (a^2/p) = 1 \) (Burton)

The proof of part (1) is straightforward. If \( a \equiv b \pmod{p} \), then \( x^2 \equiv a \pmod{p} \) has a solution if and only if \( x^2 \equiv b \pmod{p} \) has a solution (Burton). Therefore, \( (a/p) = (b/p) \) (Burton). For part (2), Euler’s Criterion states that \( (a/p) \equiv a^{(p-1)/2} \pmod{p} \) (Burton 181). It also states that \( (b/p) \equiv b^{(p-1)/2} \pmod{p} \), \( (a^2b^2/p) \equiv (a \times b)^{(p-1)/2} \pmod{p} \) (Burton 186). So \( (a/p) \times (b/p) \equiv a^{(p-1)/2} \times b^{(p-1)/2} = (a \times b)^{(p-1)/2} \equiv (a^2b^2/p) \pmod{p} \), and because the Legendre Symbol can only be 1 or -1, this proof is finished (Burton). State (3) follows by definition.

Multiplying quadratic residues and quadratic nonresidues are a lot like multiplying positive and negative integers, where quadratic residues are the positive numbers and quadratic nonresidues are the negative numbers. Two quadratic residues multiplied together produce a quadratic residue, and two quadratic nonresidues multiplied together produce a quadratic residue, just like how multiplying two positive numbers produce a positive number and two negative numbers produce a positive number (Burton 186). However, if a quadratic
residue and a quadratic nonresidue are multiplied together, then a quadratic nonresidue is the result (Burton).

By using Euler's Criterion, the following theorem can be stated: if \( p \) is an integer that is odd and prime, then \((-1/p)\) is 1 if \( p \equiv 1 \pmod{4} \) and \(-1\) if \( p \equiv -1 \pmod{4} \) (Burton 187). The proof of this theorem uses Euler’s Criterion. By using this criterion, \((-1/p) \equiv (-1)^{p-1} \pmod{p}\) (Burton). If \( p \equiv 1 \pmod{4} \), then \( p = 4k + 1 \) for some integer \( k \) (Burton). Therefore \((-1)^{p-1} = (-1)^{4k+1} = (-1)^{2k} = [(-1)^2]^k = 1\), and this means that \((-1/p) = 1\) (Burton). If \( p \equiv 3 \pmod{4} \), then \( p = 4k + 3 \) for some integer \( k \) (Burton). Therefore, like before, \((-1)^{p-1} = (-1)^{4k+1} = (-1)^{2k+1} = -1\), and so \((-1/p) = -1\) (Burton). This concludes the proof.

This next section begins with an important statement that was made by Gauss. It is used to determine if an integer \( a \), where \( a \) and the odd prime integer \( p \) are relatively prime, is a quadratic residue of \( p \) (Burton 189). This criterion is called Gauss’ Criterion. Let an integer \( p \) be odd and prime, and let another integer \( a \) be an integer such that \( a \) and \( p \) are relatively prime (Burton). If an integer, \( s \), is the number of least positive quadratic residues of the integer moduli \( a, 2a, 3a \) up to \(((p-1)/2)a\) that are less than \( p/2 \), then \((a/p) = (-1)^s\) (Burton). The proof for this is omitted, and instead an example will be shown. Suppose that a \( p \) value of 11 is used, and an integer 5 is used as the \( a \) value. Then, if a solution exists to the congruence \( x^2 \equiv 5 \(
The RSA Encryption System and the Factorization of Large Numbers

(mod 11), or if the Legendre Symbol of 5 and 11 is 1, then the least positive residues of 1*5, 2*5, 3*5, 4*5, and 5*5 need to be computed. 5 is chosen because (p-1)/2 = 5. All of these residues are taken modulo 11. Therefore, 5 ≡ 5 (mod 11), 10 ≡ 10 (mod 11), 15 ≡ 4 (mod 11), 20 ≡ 9 (mod 4), and 25 ≡ 3 (mod 11). The value of p / 2 is equal to 5.5, and there are exactly two values that are less than 5.5, and therefore Gausses' Criterion says that (5/11) = (-1)^2 = 1.

By using the previously defined Gauss' Criterion, several theorems can be derived. The first is if p is an integer that is prime and odd, then \((2/p) =(-1)^{(p-1)/2}\) (Burton 191). The second is if p is an integer that is prime and odd, then \((3/p) =1\) if \(p \equiv 1 \text{ or } -1 \pmod{12}\) and \((3/p) =-1\) if \(p \equiv 5 \text{ or } -5 \pmod{12}\) (Burton 199). These theorems could continue on for small values of a, but then the great Gauss, at age 19, came up with an efficient algorithm for finding the Legendre Symbol for values of a (Burton 196). This algorithm is called the Quadratic Reciprocity Theorem. This famous theorem says that if two integers p and q are odd and prime, and at least one of them is congruent to 1 (mod 4), then \((p/q) = (q/p)\) (Burton). If both p and q are both congruent to 3 (mod 4), then \((p/q) = -(q/p)\) (Burton). Another way of stating this theorem is to say that \((p/q)^*(q/p) = (-1)^{(p-1)(q-1)/4}\) (Burton). These two definitions are equivalencies of the famous Law of Quadratic Reciprocity.
The proof for the Quadratic Reciprocity Theorem is omitted, and instead an example will be shown. First of all, it is plain that \((p-1)/2\) is even when \(p \equiv 1 \pmod{4}\) and is odd when \(p \equiv 3 \pmod{4}\). Therefore, \(\frac{p-1}{2} \cdot \frac{q-1}{2}\) is even if \(p \equiv 1 \pmod{4}\) or when \(q \equiv 1 \pmod{4}\), however \(\frac{p-1}{2} \cdot \frac{q-1}{2}\) is odd if \(p\) and \(q\) are both congruent to \(3 \pmod{4}\) (Burton 198). As a result, \((p/q)\cdot(q/p)\) is 1 if \(p \equiv 1 \pmod{4}\) or if \(q \equiv 1 \pmod{4}\) or if both \(p\) and \(q\) are congruent to 1 \((\pmod{4})\) (Burton). Also, \((p/q)\cdot(q/p)\) if \(-1\) if \(p \equiv 3 \pmod{4}\) and if \(q \equiv 3 \pmod{4}\), or in other words, if \(p \equiv q \equiv 3 \pmod{4}\) (Burton). Because \((p/q)\) and \((q/p)\) can only be 1 or \(-1\), then \((p/q)\) equals \((q/p)\) if \(p \equiv 1 \pmod{4}\) or if \(q \equiv 1 \pmod{4}\) or perhaps both, and \((p/q)\) equals \(-(q/p)\) if \(p \equiv q \equiv 3 \pmod{4}\), or if \(p\) and \(q\) are odd primes, then \((p/q) = (q/p)\), unless both if \(p\) and \(q\) are congruent to 3 \((\pmod{4})\), in which case \((p/q)\) is equal to \(-\!(p/q)\) (Burton). To demonstrate this, let \(p = 7\) and \(q = 19\). \(7 = 1 \times 4 + 3\) and \(19 = 4 \times 4 + 3\), so both \(p\) and \(q\) are congruent to 3 \((\pmod{4})\). Therefore, \((7/19) = -(19/7) = -(5/7) = -(7/5) = -(2/5) = -(1) = 1\), since it is known that \((2/p) = (−1)^{\frac{p-1}{2}}\).

The previous discussion about the Legendre Symbol shows an effective and fast way to find \((n/p)\), if \(n\) is factorable. But, what if \(n\) is a large number, and factoring it is a problem? A man named Carl Gustav Jacobi came up with a way to deal with this problem. He defined what is known as the Jacobi Symbol. Let \(n\) be an integer and \(m\) be any other positive odd integer such that \(m = p_1 \times p_2 \times ... \times p_r\) where the \(p\)s are odd primes that can be repeated as much as needed (Burton 202). The
The RSA Encryption System and the Factorization of Large Numbers

Jacobi Symbol \((n/m)\) is defined as \((n/m) = (n/p_1) \ast (n/p_2) \ast \ldots \ast (n/p_r)\), where \((n/p_r)\) is the Legendre Symbol (Burton). The Jacobi Symbol’s power is derived from the fact that it uses the same computational rules as the Legendre Symbol (Burton).

The following theorem states seven properties of the Jacobi Symbol. They are stated without proof.

Let \(m, m'\) be positive integers that are odd, and let \(n\) be any positive integers (Burton 202). Then:
- (a) \((n/m)* (n/m') = (n/m^*m')\) (Burton)
- (b) \((n/m)* (n'/m) = (n^*n'/m)\) (Burton)
- (c) \((n^2/m) = 1 = (n/m^2)\) if \(n\) and \(m\) are relatively prime (Burton)
- (d) If \(n \equiv n' \pmod m\), then \((n/m) = (n'/m)\) (Burton)
- (e) \((-1/m) = 1\) if \(m \equiv 1 \pmod 4\) and \((-1/m) = -1\) if \(m \equiv -1 \pmod 4\) (Burton)
- (f) \((2/m) = 1\) if \(m \equiv 1\) or \(-1\) \((\pmod 8)\), \((2/m) = -1\) if \(m \equiv 3\) or \(-3\) \((\pmod 8)\) (Burton)
- (g) \((n/m) = (m/n)\) if \(n\) and/or \(m\) are congruent to 1 \((\pmod 4)\), and \((n/m) = - (m/n)\) if \(n\) and \(m\) are congruent to 3 \((\pmod 4)\) (Burton)

These seven statements say that it does not matter if the denominator of the Jacobi Symbol is prime or not. The only exception is when factors of 2 need to be pulled out of the denominator. This theorem can be implemented in a fast and efficient manner, and it is used for deciding if an integer \(n\) is a quadratic residue of a modulo \(p\). For example, suppose that \((1003/1151)\) needs to be found. Since \(1003 = 4 \ast 250 + 3\) and \(1151 = 4 \ast 287 + 1\), then 
\[-(1151/1003) = -(148/1003) = -(4/1003) \ast (37/1003) = -(22/1003) \ast (37/1003) = -(37/1003) = -(1003/37) = -(4/37) = -1.\]
The Quadratic Sieve

The results of the previous section will be put to work as the factoring method known as the Quadratic Sieve is looked at. The Quadratic Sieve is considered one of the "Big Gun" factoring algorithms, and is considered the best known algorithm for factoring large composite numbers. The secret lies in the "sieving" of factors out of the composite number (Burton 343). This is also where the algorithm can fall down: the sieving is the part that takes the longest time of the entire algorithm (Bressoud 110). Despite the fact that the sieving can take a long time, the Quadratic Sieve is still the best available factoring algorithm.

To begin the sieving process, a composite number $n$ is chosen (Burton 341). Then, the floor of the square root of $n$ is found (Burton). This number is used in a function that will be discussed later. After finding the floor of the square root of $n$, the factor base needs to be calculated (Burton). The factor base is a list of prime numbers in which the Legendre Symbol of the floor of the square root of $n$ and a prime number is equal to one (Burton). The Legendre Symbol, $(a/p)$, as seen previously, is defined to be an odd prime $p$ and an integer $a$, where $a$ and $p$ are relatively prime (Burton 185). If $a$ is a quadratic residue of $p$, then the Legendre Symbol is 1 (Burton). However, if $a$ is a quadratic nonresidue of $p$, then the Legendre Symbol is $-1$ (Burton). The factor base does not encompass all of the prime numbers. Instead, it is generally all the primes less than ten thousand for which the Legendre Symbol of $n$ and the prime is equal to one (Burton 341).
The RSA Encryption System and the Factorization of Large Numbers

The next part of the algorithm that needs to be calculated is a list of values that is centered about the floor of the square root of \( n \) (Burton 341). Each number in this list is then run through the equation \( f(x) = x^2 - n \), where \( x \) is the value in the list that was just created, and \( n \) is the value to be factored (Burton). This will produce a new list that contains values that are composite (Burton). These values are then factored themselves, using any desired factoring method (Burton). The resulting factors of each of the calculated numbers are then compared to the factor base, which was found earlier (Burton). If all of the factors from the calculated numbers are in the factor base, the number and its factors are noted and set aside (Burton). However, if the calculated number has at least one factor that is not in the factor base, then this number and its factors are discarded (Burton).

After the list of calculated numbers and their factors that are in the factor base is found, the factor matrix needs to be created (Burton 342). This matrix, which is composed of ones and zeroes, tells which factors of the calculated numbers were in the factor base (Burton). The size of the matrix is dependent on the size of the factor base and how many numbers had all their factors in the factor base (Burton). The number of columns is dependent on the number of factors in the factor base, and the number of rows is dependent on the number of calculated numbers that had all their factors in the factor base (Burton). For each row, each of the factors that were in the factor base is looked at. If the factor is to an odd power, like one, then a one is put in
The RSA Encryption System and the Factorization of Large Numbers

the column associated with that prime factor (Burton). If the factor is to an even power, or that calculated number did not have a particular factor, then a zero is put in that column (Burton). This matrix is then used to find linear dependencies (Burton).

There are several different ways to find the linear dependencies. The easiest way, which will be outlined here, is not the quickest way. A faster way would be to augment the factor matrix with an identity matrix and use Gaussian Elimination on it (Bressoud 114). However, for simplicity sake, a less complicated version of guess and check will be used instead. First of all, a linear dependency consists of three or more rows in the factor matrix that, when added together and taken modulo two, produce all zeros for each of the columns. These dependencies could have as few as three rows in them, or as much as all the rows in them. It should be noted that there could never be only two rows in a dependency, because in order for the two rows to produce an all zero row, the two rows would have to be identical, and this is quite impossible. Therefore, at least three rows must be used.

After finding a successful linear dependency, congruencies that relate to the vectors found need to be created (Burton 343). To do this, first get the original numbers that created the rows in the factor matrix (Burton). Then, raise these to the second power, and set them congruent to its factors modulo n (Burton). For example, if three vectors that were found to be linearly dependent had original x
values of 85, 89, and 98, then they would have the following congruencies: \( f(85) = 85^2 \equiv -2 \cdot 3 \cdot 13 \cdot 29 \pmod{9487} \), \( f(89) = 89^2 \equiv -2 \cdot 3^3 \cdot 29 \pmod{9587} \), and \( f(98) = 98^2 \equiv 3^2 \cdot 13 \pmod{9487} \) (Burton). After finding these \( x \) values and their factors, the two sides are multiplied together (Burton). For example, the left-hand side would become \((85 \cdot 89 \cdot 98)^2\) and the right hand side would become \((2 \cdot 3^3 \cdot 13 \cdot 29)^2 \pmod{9487}\) (Burton). Multiplying out the two sides yields \( 741370^2 \equiv 20358^2 \pmod{9487} \) (Burton). By removing the squares on the numbers, and testing the congruence, it can be seen if the linear dependency is valid or not (Burton). In this case, 741370 is congruent to 20358 (mod 9487), so the three lines from the factor matrix that were used were invalid, and other lines should be chosen instead (Burton).

After a valid dependency has been found, and the congruency has been created, then the final part of the quadratic sieve factor method can be implemented (Burton). When the final congruency has been found, both sides of it need to be taken modulo \( n \) (Burton). For example, if each side of the dependency \( 769500^2 \equiv 26334^2 \pmod{9487} \), were taken modulo 9487, the resulting dependency would be \( 1053^2 \equiv 7360^2 \pmod{9487} \) (Burton). Since \( 1053^2 \) is not congruent to \( 7360^2 \pmod{9487} \), this dependency will work out (Burton). As the final step, the greatest common divisor of the left-hand side added to the non-modulus right-hand side and the modulus is taken (Burton). The result of this is a factor of the original \( n \) (Burton). Then, to find another factor, \( n \) is divided by the newfound factor (Burton). This
The RSA Encryption System and the Factorization of Large Numbers

factoring method, which is partially coded in *Mathematica*, can be viewed in the N Appendix in the Appendices section.
The RSA Encryption System and the Factorization of Large Numbers

Conclusion

This month, April 2002, marks the 25th anniversary of the RSA system. It continues to be an important cryptographic system, and is the object of on-going research. The security of RSA is dependent on the ability to factor a large composite number, and since both the government and the private sector use the RSA algorithm, a lot is riding on the ability to factor these large composite numbers. Since new technology and faster computers are being invented everyday, it is only a matter of time before computers become fast enough to allow the Quadratic Sieve to factor much larger composite number quickly, or before a mathematician comes up with a new, faster factoring algorithm than can factor 300 or more digit composite numbers in a matter of minutes, instead of years. Until this time is realized, though, RSA will continue to remain secure, and the problem of trying to find fast algorithms to factor large composite numbers will remain the subject of much research and study. It will be interesting to see the results that develop in the coming years as the importance of cryptographic systems, and their applications, continues to increase.¹ ²

¹ To read more information about Cryptography and its applications, please consult the following books:
The RSA Encryption System and the Factorization of Large Numbers

Works Cited


Appendices
Miscellaneous Algorithms

Appendix A: Fast Exponentiation

This algorithm is used to quickly find the power of the modular expression, \( a^b \mod m \). Three examples are shown: one using the old way with just powers and the mod function, the second one using Mathematica’s PowerMod function, and the last way using the new FastExpo function. Fast Exponentiation is used in the decoding of RSA messages, and it can be seen in Appendices E and F. The pseudocode for this algorithm came from Bressoud’s Factorization and Primality Testing, page 34.

\[
\text{FastExpo}[x_, y_, z_] := (a = x; b = y; m = z; n = 1;
\text{While}[b \neq 0, \text{If}[\text{OddQ}[b] == \text{True}, n = \text{Mod}[n \cdot a, m]];
\text{b} = \text{Floor}[	ext{b}/2]; a = \text{Mod}[a \cdot a, m]; \text{Print}[n])
\]

\[
a = 231;
b = 7777;
m = 78901;
\text{Mod}[a^b, m]
\]

34660

\[
\text{PowerMod}[a, b, m]
\]

34660

\[
\text{FastExpo}[a, b, m]
\]

34660

Appendix B: Greatest Common Divisor

This function is used to find the Greatest Common Divisor of two numbers. The first example demonstrates the new function, and the second example shows the same using Mathematica’s GCD function. The pseudocode for this algorithm came from Bressoud’s Factorization and Primality Testing, page 10.

\[
\text{NewGCD}[x_, y_] := (a = x; b = y; u = \{1, 0, a\}; v = \{0, 1, b\};
\text{While}[u[[3]] \neq 0, q = \text{Floor}[u[[3]]/v[[3]]];
\text{For}[i = 1, i \leq 3, i++, \text{temp} = v[[i]]];
\text{v}[[i]] = u[[i]] - q \cdot v[[i]]; u[[i]] = \text{temp};] \text{Print}[u[[3]]])
\]

\[
\text{NewGCD}[3463787886, 2342344243545]
\]

3

\[
\text{GCD}[3463787886, 2342344243545]
\]

3
Appendix C: Jacobi Symbol

The Jacobi Symbol is essentially the Legendre symbol where the value m is an odd prime. A Legendre symbol is a test used to find out if a number is a quadratic residue of a prime number. Two examples are shown: one using the new algorithm, and another using Mathematica's built in function. The pseudocode for this algorithm came from Bressoud's Factorization and Primality Testing, page 97.

```math
Jacobi[x_, y_] := (n = x; m = y;
    jacobi = 1;
    If[GCD[n, m] > 1,
        Return[0];
        n = Mod[n, m];
    While[n > 1,
        If[Mod[(n - 1) * (m - 1), 8] = 4,
            jacobi = -1 * jacobi;
        temp = n;
        n = Mod[m, n];
        m = temp;
        count = 0;
        While[EvenQ[n] = True,
            n = n/2;
            count = 1 - count;
            If[Mod[count * ((m^2) - 1), 16] = 8,
                jacobi = -1 * jacobi;]]];
    Return[jacobi];);

Jacobi[126, 509]
1

JacobiSymbol[126, 509]
1
```

RSA Encoding and Decoding Algorithms

Appendix D: Encrypt

This is the encrypting part of the RSA algorithm, shown line by line. A full explanation of it can be seen in the Explanation of the RSA Algorithm: Encoding portion of the paper. The code for this is the author's, but the example came from Rosen's Elementary Number Theory and its Applications, on pages 260-261.
ToEncode = "PUBLIC KEY CRYPTOGRAPHY"
Mess = ToCharacterCode[ToEncode]
For[i = 1, i ≤ Length[Mess], Mess[[i]] = Mess[[i]] - 65; i++]
For[i = 1, i ≤ Length[Mess],
  If[Mess[[i]] == -33, Mess = Delete[Mess, i]; i++]
If[Mod[Length[Mess], 2] != 0, Mess = Append[Mess, 23]]

PUBLIC KEY CRYPTOGRAPHY

{80, 85, 66, 76, 73, 67, 32, 75, 69, 89,
  32, 67, 82, 89, 80, 84, 79, 71, 82, 65, 80, 72, 89}

{15, 20, 1, 11, 8, 2, 10, 4, 24, 2, 17, 24, 15, 19, 14, 6, 17, 0, 15, 7, 24, 23}

NewMess = {};
For[i = 1, i ≤ Length[Mess], If[OddQ[i] == True,
  NewMess = Append[NewMess, Mess[[i]] * 100 + Mess[[i + 1]]]; i++]

NewMess

{1520, 111, 802, 1004, 2402, 1724, 1519, 1406, 1700, 1507, 2423}

p = 43; q = 59; e = 13; PQ = p * q; d = 937;

For[i = 1, i ≤ Length[NewMess],
  NewMess[[i]] = Mod[NewMess[[i]]^e, PQ]; i++

NewMess

{95, 1648, 1410, 1299, 811, 2333, 2132, 370, 1185, 1957, 1084}

Appendix E: Decrypt

This is the decrypting part of the RSA algorithm, shown line by line. A full explanation of it can be seen in the Explanation of the RSA Algorithm: Decoding portion of the paper. The code for this is the author's, but the example came from Rosen's *Elementary Number Theory and its Applications*, on page 261.

For[i = 1, i ≤ Length[NewMess],
  NewMess[[i]] = PowerMod[NewMess[[i]], d, PQ]; i++]

NewMess

{1520, 111, 802, 1004, 2402, 1724, 1519, 1406, 1700, 1507, 2423}

OrigMess = {};
For[i = 1, i ≤ Length[NewMess], pi = IntegerPart[NewMess[[i]] / 100];
  p2 = ((NewMess[[i]] / 100) - IntegerPart[NewMess[[i]] / 100]) * 100;
  OrigMess = Append[OrigMess, pi]; OrigMess = Append[OrigMess, p2]; i++]

OrigMess

{15, 20, 1, 11, 8, 2, 10, 4, 24, 2, 17, 24, 15, 19, 14, 6, 17, 0, 15, 7, 24, 23}
Appendix F: Functions

This appendix shows the encoding and decoding algorithms put into Mathematica functions for easy use. The code for this is the author's.

\[
\text{InvMod}[m_, z_] := \begin{cases} 
(a = \text{If}[z > m, z; b = m], b = z]) \\
\text{al} = a; bl = b; \\
xold = 1; yold = 0; \\
x = 0; y = 1; c = 1; \\
\text{While}[c != 0, \\
q = \text{Floor}[a/b]; \\
c = a - q*b; \\
\text{If}[c == 0, \text{While}[y < 0, y = y + m]] \\
xnew = xold - q*x; \\
ynew = yold - q*y; \\
xold = x; yold = y; \\
x = xnew; y = ynew; a = b; b = c;]
\end{cases}
\]

\[
\text{PKEDecode}[\text{DecMess}_-, p_, q_, e_, d] := (\text{ToDecode} = \text{DecMess}; \\
\text{PKEDecode}[\text{DecMess}_-, d_] := (\text{PQ} = p*q; \text{ToDecode} = \text{DecMess}; \\
\text{InvMod}[(p - 1)*q, e]; d = IM; \text{For}[i = 1, i \leq \text{Length}[\text{ToDecode}], \\
\text{ToDecode}[[i]] = \text{PowerMod}[\text{ToDecode}[[i]], d, \text{PQ}]; i++]; \text{OrigMess} = \{}; \\
\text{For}[i = 1, i \leq \text{Length}[\text{ToDecode}], p1 = \text{IntegerPart}[\text{ToDecode}[[i]]/100]; \\
p2 = ((\text{ToDecode}[[i]]/100) - \text{IntegerPart}[\text{ToDecode}[[i]]/100]) \times 100; \\
\text{OrigMess} = \text{Append}[\text{OrigMess}, p1]; \text{OrigMess} = \text{Append}[\text{OrigMess}, p2]; i++]; \\
\text{Mess} = \{}; \text{For}[i = 1, i \leq \text{Length}[\text{OrigMess}], \\
\text{Mess} = \text{Append}[\text{Mess}, \text{OrigMess}[[i]] + 65]; i++]; \\
\text{FromCharacterCode}[\text{Mess}])
\]

Appendix G: Example
This section shows an example of the encoding and decoding functions. The two primes are both 50 digits in length. The number, 331, is the e values and is relatively prime to the Euler $\phi$ of the two primes. The code for this is the author's.

```plaintext
PKEncode["PUBLIC KEY CRYPTOGRAPHY", 871355364796143366952212421643936766139686444799293, 881345847596963866866675674346268894933737781585497, 331]

{53395781665947722241917701029263354057763617339342626987396213995388016. 01367744067050548968381903994, 502077786860712114915856425219563745862547879050641914378879897120113. 69365470811659329778119471955, 1019483176757503043410920001484388230494229669196198907694771341797858. 704435620983501560367425664444, 47185401409811135192659846227597307037522060375689114837052276988561969. 55576250196129346192484127256, 395015755809531487328708423976955840678580418824916616108962947213914. 43496465675649109782742592947, 9891328221359173800971208185683775851088737395630414495537050548605344. 6454267187813268742037533723, 56665496615495315761242591744916379224307526894104219562920182019577331. 3660439756653661924129731748, 42228953510101394236858859808840385685106382377396959924329366507485681. 375426222827453792591707500123, 70769096498700296787676770811588233068313507327415858443281125977122934. 30471506685297979743620702287, 15211952600275103073468076118575756363027901599694444208974525570570349. 09546431199739610956853587953, 679833006089189928088553935298170949274824081024010534161521010923749783. 4837129050527636657519677210210}
PKDecode[NewMess, 871355364796143366952212421643936766139686444799293, 881345847596963866866675674346268894933737781585497, 331]
PUBLICKEYCRYPTOGRAPHY
p = 871355364796143366952212421643936766139686444799293; q = 881345847596963868866675674346268894933737781585497; PrimeQ[p] True
PrimeQ[q] True
GCD[(871355364796143366952212421643936766139686444799293 - 1) * (881345847596963868866675674346268894933737781585497 - 1), 331] 1
```

Prime Number Algorithms
Appendix H: Generate Sequential Primes

This function will generate a series of sequential primes. It takes as input the number of digits the primes should be, and how many are wanted. This code for this is the author's.

```math
FindPrime[x_, y_] :=
  Digits = x;
  NumOfPrimes = y;
  Counter = 0;
  PrimeArray = Table[(i - (i + 1)), {i, 1, Digits}];
  PNumber = FromDigits[PrimeArray];
  For[i = 1, i ≤ 10000, i++;
    If[PrimeQ[PNumber],
      Print[PNumber]; PNumber = PNumber + 2; Counter = Counter + 1;
      If[Counter = y, i = 10001],
      PNumber = PNumber + 2];

  x = FindPrime[100, 10]
```

Appendix I: Generate Random Primes

This function will generate random primes, instead of sequential primes. It takes as input the length of the primes, and how many should be generated. The code for this is the author's.
FindPrime[x_, y_] := (  Digits = x;  NumOfPrimes = y;  Counter = 0;  For[k = 1, k ≤ NumOfPrimes, k++;  PrimeArray = Table[(i - i) + Random[Integer, {1, 9}], {i, 1, Digits}];  PNumber = FromDigits[PrimeArray];  For[i = 1, i ≤ 100000000, i++;  If[PrimeQ[PNumber], Print[PNumber];  i = 100000001, PNumber = PNumber + 1]]);)

x = FindPrime[50, 10]

36913369577164776314786787468737396614396161476023
9116729595899321767189155122735863912421242476383
24911557288131965841328254372224521547711211611351
76851994637735114697859656124337564377536816137521
52641488513618584631952789514178269837631982125401
1296667141189487385367652376298581948275476768437
96791837227182213219225195167953165491542997769423
8422245545737546651523151987682782169533157247267
42186724552217134366764814995722344687294697786627
17529329715871485491194373275976883744222796981673

Appendix J: Strong Pseudoprime Test

The strong pseudoprime test is primarily used in determining if a number is prime. A full description of it can be found in the part of the paper entitled Primes: Strong Pseudoprimes. The pseudocode for this algorithm came from Bressoud's Factorization and Primality Testing, page 77.
StrongPS[x_, y_] := (n = x; b = y;
   If[GCD[n, b] ≠ 1,
      Print["The GCD of n and b is not 1."];]
   t = n - 1;
   a = 0;
   While[EvenQ[t] = True,
      t = t/2;
      a = a + 1;
   ];
   test = PowerMod[b, t, n];
   If[test = 1 || test = n - 1,
      Return[True];]
   For[i = 1, i ≤ a - 1, i++,
      test = Mod[(test * test), n];
      If[test = n - 1,
         Return[True];]
   ];
   Return[False];);

StrongPS[3215031751, 2]
StrongPS[3215031751, 3]
StrongPS[3215031751, 5]
StrongPS[3215031751, 7]

True
True
True
True

Factoring Algorithms

Appendix K: Trial Division

This is the most basic of the factoring algorithms. A full description of it can be seen in the Factoring: Trial Division section of the paper. The code for this is the author's.

n = 102569;
For[i = 2, i ≤ Floor[Sqrt[n]], i++,
   If[IntegerQ[n/i] == True,
      Print[i, " ", n/i]; i = Floor[Sqrt[n]]];
109 941

Appendix L: Fermat's Algorithm
This algorithm is more powerful than Trial Factorization, but is still not really powerful. For a full description of its strengths and weaknesses, refer to the Factoring: Fermat's Algorithm in the paper. The pseudocode for this algorithm came from Bressoud's *Factorization and Primality Testing*, pages 59-60.

```
FermatFactor[x_] := (n = x;
  sqrt = Ceiling[Sqrt[n]];
  u = 2 * sqrt + 1;
  v = 1;
  r = sqrt * sqrt - n;
  While[r ≠ 0, If[r > 0, While[r > 0, r = r - v; v = v + 2]]; 
    If[r < 0, r = r + u; u = u + 2];
    a = (u + v - 2) / 2;
    b = (u - v) / 2;
    Print[{a, b}])
```

FermatFactor[100]
{10, 10}

FermatFactor[1783647329]
{84449, 21121}

Appendix M: Pollard's Rho

This factoring algorithm is more powerful than Fermat's Algorithm, but not as powerful as the Quadratic Sieve. To read more about it, see the Factoring: Pollard's Rho section of the paper. The code for this algorithm is the author's, but the idea came from Burton's *Elementary Number Theory*, page 339-340.

```
Clear[n, x0, f, x, SeqList]
n = 1391411183;
x0 = 3;
f[x_] = x^2 - 1;
SeqList = {x0};
For[i = 2, i ≤ 1000, i++,
  AppendTo[SeqList, Mod[f[SeqList[[i - 1]], n]]];
  SeqList = Delete[SeqList, 1];
  For[j = 1, j ≤ Floor[Length[SeqList] / 2], j++,
    If[GCD[SeqList[[2 * j]] - SeqList[[j]], n] ≠ 1,
      Print["One factor of ", n, " is ", GCD[SeqList[[2 * j]] - SeqList[[j]], n], ", and the other is ", n/GCD[SeqList[[2 * j]] - SeqList[[j]], n];
      j = Floor[Length[SeqList] / 2]];]
  ];

One factor of 1391411183 is 29243, and the other is 47581
```

Appendix N: The Quadratic Sieve
The Quadratic Sieve is considered to be the most powerful known algorithm for factoring large composite numbers. For a full write-up of it and the power behind it, see the Factoring: The Quadratic Sieve section of the paper. The code for this is the author's, but the idea came from Burton's Elementary Number Theory, pages 341-343.

```plaintext
Num = 9487
FloorN = Floor[Sqrt[Num]]
F[x_] := x^2 - Num;
FactorBase = {-1, 2};
For[i = 2, i ≤ 10, i++,
    If[JacobiSymbol[Num, Prime[i]] == 1, AppendTo[FactorBase, Prime[i]]];]

XList = {FloorN};
For[j = 1, j ≤ 16, j++,
    AppendTo[XList, FloorN + j];
    AppendTo[XList, FloorN - j];
XList = Sort[XList];

FXList = {};
For[k = 1, k ≤ Length[XList],
    k++, AppendTo[FXList, F[XList[[k]]]];]

ListOfFactors = {};
For[l = 1, l ≤ Length[FXList], l++,
    AppendTo[ListOfFactors, FactorInteger[FXList]];]

ListOfInFactors = {};
For[m = 1, m ≤ Length[ListOfFactors], m++,
    InFactorBase = True;
    For[o = 1, o ≤ Length[ListOfFactors[[1, m]]], o++,
        If[MemberQ[FactorBase, ListOfFactors[[1, m, o, 1]]] = False,
            InFactorBase = False];
    AppendTo[ListOfInFactors, InFactorBase];]

ListOfX = {};
For[v = 1, v ≤ Length[ListOfInFactors], v++,
    If[ListOfInFactors[[v]] = True,
        AppendTo[ListOfX, XList[[v]]];]

ListOfCompleteFactors = {};
For[n = 1, n ≤ Length[ListOfFactors], n++,
    If[ListOfInFactors[[n]] = True,
        AppendTo[ListOfCompleteFactors, ListOfFactors[[1, n]]];]

FactorMatrix = {};
For[r = 1, r ≤ Length[ListOfCompleteFactors], r++,
    FactorRow = Table[x - x, {x, 1, Length[FactorBase]}];
    For[p = 1, p ≤ Length[ListOfCompleteFactors[[r]]], p++,
        For[q = 1, q ≤ Length[FactorBase], q++,
            FactorRow[
```
If[FactorBase[[q]] == ListOfCompleteFactors[[r, p, 1]],
    FactorRow[[q]] = Mod[ListOfCompleteFactors[[r, p, 2]], 2];]
AppendTo[FactorMatrix, FactorRow];

Total = Table[1, {x, 1, Length[FactorBase]}];
GoodVector = {};
GoodNumber = {};
GoodFactor = {};
ChosenVectors = {};
ChosenNumbers = {};
ChosenFactors = {};
Counter = Table[x, {x, 1, Length[FactorMatrix] - 2}];
Size = 3;
For[g = 3, g ≤ Length[FactorMatrix] - 2, g++,
    While[Counter[[1]] ≠ Length[FactorMatrix] &&
        MemberQ[Counter, Length[FactorMatrix] + 1] == False,
        Total = 0;
        For[c = 1, c ≤ Size, c++,
            Total = Total + FactorMatrix[[Counter[[c]]]];
            Total = Mod[Total, 2];
        ];
    ];
If[Total == Table[0, {x, 1, Length[FactorBase]}],
    For[d = 1, d ≤ Size, d++,
        AppendTo[GoodVector, FactorMatrix[[Counter[[d]]]]];
        AppendTo[GoodNumber, ListOfX[[Counter[[d]]]]];
        AppendTo[GoodFactor, ListOfCompleteFactors[[Counter[[d]]]]];
        AppendTo[ChosenVectors, GoodVector];
        AppendTo[ChosenNumbers, GoodNumber];
        AppendTo[ChosenFactors, GoodFactor];
        GoodVector = {};
        GoodNumber = {};
        GoodFactor = {};
    ];
If[MemberQ[Counter, Length[FactorMatrix] - 1] == True &&
    Position[Counter, Length[FactorMatrix] - 1][[1, 1]] ≠ Size,
    For[f = Length[Counter], f ≥ 2, f--,
        If[Counter[[f]] - Counter[[f - 1]] > 1,
            ToIncrement = f - 1; f = 2];
        ];
    Counter[[ToIncrement]] = Counter[[ToIncrement]] + 1;
    For[e = ToIncrement + 1, e ≤ Size, e++,
        Counter[[e]] = Counter[[e - 1]] + 1];
    ];
If[MemberQ[Counter, Length[FactorMatrix]] == True,
    Counter[[Position[Counter, Length[FactorMatrix]][[1, 1]] - 1]] =
    Counter[[Position[Counter, Length[FactorMatrix]][[1, 1]] - 1]] + 1;
    Counter[[Position[Counter, Length[FactorMatrix]][[1, 1]]]] =
    Counter[[Position[Counter, Length[FactorMatrix]][[1, 1]] - 1]] + 1,
    Counter[[Size]] = Counter[[Size]] + 1];
];
Size = Size + 1;
Counter = Table[x, {x, 1, Length[FactorMatrix] - 2}];

Page All
FactorCounter = 1;
LHS = 1;
RHS = 1;
While[Mod[LHS, Num] == Mod[RHS, Num],
LHS = ChosenNumbers[[FactorCounter, 1]] * 
    ChosenNumbers[[FactorCounter, 2]] * ChosenNumbers[[FactorCounter, 3]];
Powers = {};
Factors = {};
FactorPowers = {};
NewChosenFactors = Flatten[ChosenFactors[[FactorCounter]], 1];
For[v = 1, v ≤ Length[FactorBase], v++,
    AppendTo[Powers, {FactorBase[[v]], 0}]];
For[w = 1, w ≤ Length[NewChosenFactors], w++,
    If[NewChosenFactors[[w, 2]] == 1,
        AppendTo[Factors, NewChosenFactors[[w, 1]]],
        Powers[[Flatten[
            Position[FactorBase, NewChosenFactors[[w, 1]]]][1], 2]] = 
            Powers[[Flatten[Position[FactorBase, NewChosenFactors[[w, 1]]]][1], 2]] + NewChosenFactors[[w, 2]]];
Factors = Sort[Factors];
For[a = 1, a ≤ Length[FactorBase], a++,
    Powers[[a, 2]] = Powers[[a, 2]] + Count[Factors, FactorBase[[a]]];
RHS = Powers[[2, 1]]^ (Powers[[2, 2]] / 2);
For[b = 3, b ≤ Length[Powers], b++,
    RHS = RHS * Powers[[b, 1]]^ (Powers[[b, 2]] / 2)];
FactorCounter = FactorCounter + 1;]
ModReductionLHS = Mod[LHS, Num];
ModRecuctionRHS = Mod[RHS, Num];

FactorOne = GCD[ModReductionLHS + ModRecuctionRHS, Num]
FactorTwo = Num / FactorOne

9487
97
179
53