# The Implementation and Optimization of the RSA Encryption Method

New Mexico Supercomputing Challenge

Final Report

April 1, 2009

Challenge Team #35

Desert Academy

**Team Members:**

Cristian Cavalli

Colton McDonald

C. Rose Morris-Wright

Avery Rowlison

Bjørn Swenson

**Mentor:**

S. Thomas Christie

## TABLE OF CONTENTS

## Abstract

RSA is one of the most secure encryption methods available to date. Our goal was to gain an intimate knowledge of the workings of RSA by constructing a fully-functional implementation of the algorithm that would encode and decode both text messages and images of arbitrary-length. We recognized that careful optimization was necessary to generate keys, and we collected and analyzed data from different configurations to optimize key-generation efficiency for use in a web-based implementation.

We first implemented the required scripts using GP, a programmable calculator which uses a C library called PARI and is capable of handling very large numbers. We created scripts to test the primality of numbers using three different methods: Rabin Miller, Fermat's Little Theorem, and the Sieve of Eratosthenes, and re-edited these scripts to generate prime numbers of specified lengths. Then we tested the efficiency of these three different functions, optimized them[1], and tested their efficiency again. After our implementation was fully functional in GP, we implemented our scripts in PHP and tested their efficiency in that language. The PHP implementation was significantly slower, but it could be hosted on a web site. Then, we used the same encryption method to encrypt images by encoding each pixel.

Our implementations have several benefits: our image encoding modification would be useful for the government and other agencies to securely transmit images. Our text-based implementation would provide users with an ultra-secure method of transmitting data via the internet, and with some more work, it would probably be fast enough to be incorporated into an instant messaging program on websites or on mobile devices.

In the future, we hope to improve efficiency even further by incorporating a GMP Library into our PHP implementation. We are also going to try to implement AJAX on our website so it will update itself and encrypt in real time.

[1] This is a function that finds the next prime after the inputted number.

## Introduction

The RSA encryption method is a widely used method for encrypting sensitive information. Banks and banking websites use RSA to secure funds. Sites like Amazon and PayPal use RSA to encrypt our credit card information to prevent outside access of these numbers. RSA is also significant because it is, as far as we know, practically unbreakable when using sufficiently large keys.

The RSA encryption method, named for its three discoverers Ron Rivest, Adi Shamir, and Leonard Adleman, was discovered in 1977 and patented by MIT. New technology and faster factoring algorithms have made older implementations with shorter keys less secure, but with larger keys, RSA is still one of the most secure methods available today.

To encrypt using RSA, the user must first convert their message into an array of numbers, and then take each entry to a large power, *k*, modulo *m* [2]. *k* and *m* taken together are the public key and published, in order to be available to everyone. *m* is the product of the two private keys, *p* and *q*. The receiver first generates the key and then publishes the public key. Anyone can encrypt a message and send it to the receiver, but only the receiver can decrypt these messages, because only the receiver knows what *p* and *q* are. One benefit of RSA is that one public key can be used by multiple users. While each user can encrypt their data using the same public key, they are unable to decrypt other users' data. This makes RSA ideal for implementation on the internet for sending sensitive data to websites.

The only straightforward way to break a properly implemented RSA algorithm is to factor the public key. The National Security Agency recruits the world's finest number theorists in an

[2] To take the mod of a number is to divide the mod into the number until only the remainder is left.

attempt to factor large public keys. Today, keys of 300-bits[3] or shorter can be factored using only your personal computer in just a few hours. Some keys, up to 663-bit, have also been factored and it is believed that researchers may be close to factoring 1024 bit keys, so they will not be secure much longer. Right now most people use between 512 and 4096 bit keys, depending on the desired level of security[4].

### Problem

The goal of our project is to investigate the RSA encryption method, create a working implementation of the algorithm, and optimize our algorithm for maximum speed and security. We compared several different prime-finding methods in the hopes of optimizing our key generation. We hoped to get an encryption system that would be fast enough to implement on a mobile device, such as the iPhone, but would still maintain a high level of security. We wanted to create a user-friendly program that would allow our user to encrypt and decrypt text and images without worry or hassle.

### Method

In order to optimize the RSA algorithm, we first had to create a working model. We learned all about how the algorithm works and implemented it in PARI/GP without using any of GP's built-in mathematical functions so it would be easy to translate to other languages and so we ensured that we thoroughly understood the mathematics behind the encryption system. We then tested the time it took for the computer to perform each different part of the algorithm, namely encoding, decoding, and generating a key of desired length.

Once we had isolated the step in the algorithm that was taking the most time, we worked to optimize that particular step. One step that we found was taking a long time was the decoding

[3] This means the mod *m* is about the size of $2^{300}$
[4] http://en.wikipedia.org/wiki/Cryptographic_key_length

and specifically the step in which we had to factor the φ($m$) (see Decoding with Euler's Theorem, page 15). In order to make this part faster we had our key generator generate keys such that the φ($m$) would be easily factorable.

After we had done all we could to make this part of the algorithm faster, we tested to make sure we had not compromised the security. The security was hard to test, as there was no way to quantitatively analyze it. RSA is already nearly impossible to break, and we do not have the distributed computer resources to see if our algorithm is as secure as those created by professional-grade programs. However, we did try to factor the public key using several advanced factoring methods, such as Pollard's P-1 algorithm. In the case of the more easily factorable φ($m$)s, we found that we had compromised the security and could factor the public key. We had to find another way of decoding that took less time and did not compromise security (see Decoding page 13-17 for more detail into each algorithm and the changes we made).

After altering our decoding process, we studied which part of our algorithm was taking the most time. We found that the key generation was the part of the algorithm that took the longest because it took a long time to find the huge prime numbers to serve as the private key. To generate 300 digit prime numbers, which is approximately the number needed for 2048-bit encryption, was taking about 5 seconds in GP (see graph #4). Generating huge keys, in the 4096-bit range, was taking an unacceptable amount of time, and we wanted to get it fast enough to be able to implement it on mobile devices which have smaller processors.

Once we realized that the problem was in finding large prime numbers, we considered making a database of prime numbers for the program to draw on, but discarded this idea when we realized how large such a database would have to be in order for a message to remain secure. We did make a database of smaller primes to facilitate in our study of prime numbers, but so far have only found those primes up to 11.3 billion. This is only about 11 digits and already text files containing the primes take up 4 gigabytes worth of space. Clearly, keeping a database of prime keys in our program would be a hindrance rather than a help.

Instead we worked on optimizing different probabilistic ways of finding prime numbers, testing them for both accuracy and speed. We researched different primality testing techniques and decided that we would compare the Sieve of Eratosthenes, Fermat’s Little Theorem, and the Rabin Miller algorithm. The Sieve is very fast for small numbers but slows down considerably when the numbers we test get large enough and requires gigabytes of stored primes as data. Fermat's method is very simple and based on simple number theory but requires modular exponentiation involving very large numbers. The Rabin-Miller method is a recent discovery and has been proven by others to be very quick and very accurate, but it requires a greater number of elementary operations than the others, meaning the efficiency of the algorithm relies heavily on the big-number library used in its implementation. (See the following section, GP vs. PHP, for more information). Our goal in testing these methods was to find the strengths and weaknesses of each algorithm and combine them in order to generate very large primes as fast as possible.

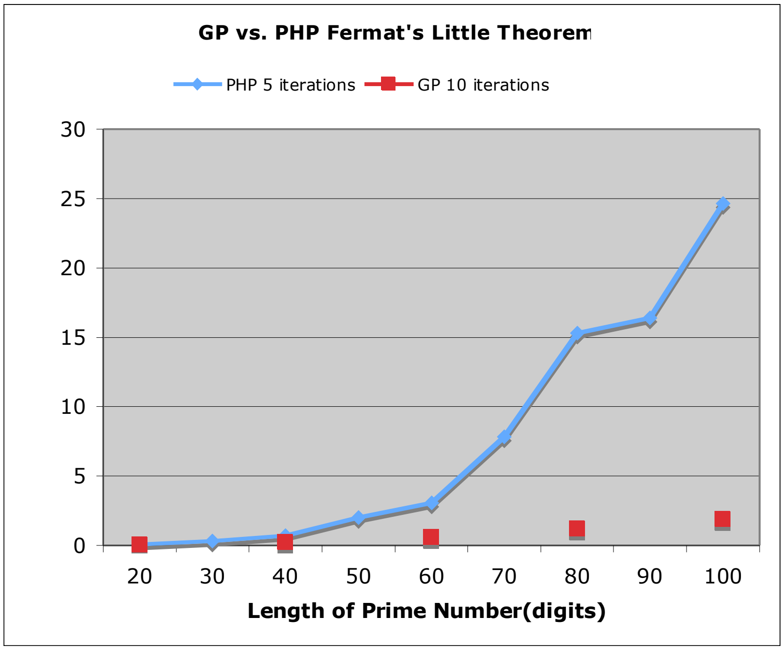
When we felt that our algorithm, including our key generation, was sufficiently fast enough, we implemented it in a user-friendly fashion on a website. (See GUI page 32 for details).

### GP vs. PHP

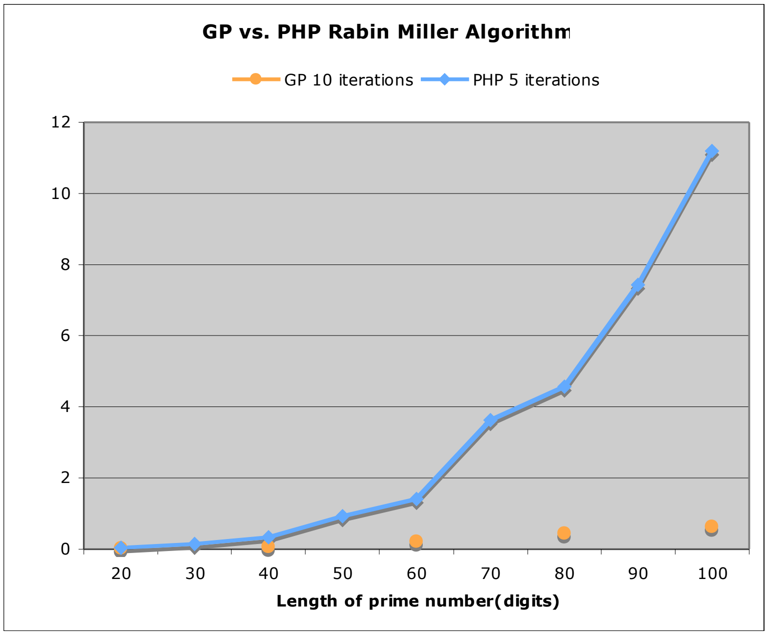
We wrote all our initial scripts in GP and used the PARI library. This helped us because GP is often used for number theory programs and has many functions built in. While we tried not to use these function, so we could prove that we understood the nuts and bolts of the mathematics, these functions were useful in checking to make sure our programs returned the correct answer. For example, GP has a built in function that tests primality. We built all our primality testing function from scratch and then tested them against the built in version to make sure that they worked properly.

Once we had a working model in GP we wanted to make it more available to everyone who wanted to use it, so we implemented it in PHP. This worked out very well for us in creating a user interface, as well as in collecting our data, as PHP could be programmed to time itself and collect data automatically and deal with external files in much more easily.

There was one draw back to using PHP, however. In its default configuration, PHP does not have a fast big-number library like Pari to draw on. As a result, it takes longer to do simple mathematical functions, like multiplying and taking the modulus. This meant that certain scripts which were very fast in GP were slower in PHP because there were a lot of small calculations to do. We are currently considering incorporating a library in PHP called GMP. This would take care of the problem and speed up our algorithms in PHP considerably. We are attempting to have this done by April 20.



*Graph 1-GP vs. PHP Fermat's Little Theorem*



*Graph 2- GP vs. PHP Rabin Miller Algorithm*

These two graphs show different algorithms in both PHP and GP. As you can see, GP is much faster, even through the GP algorithm is doing each algorithm for ten iterations while the PHP algorithm is only doing each one for five iterations. So GP is doing twice as much work and is still faster. (For more information on what Fermat's Little Theorem and Rabin-Miller do, see Prime Finding: Fermat page 23 and Prime Finding: Rabin Miller page 24).

## Mathematical model

RSA, like many encryption systems, uses a trapdoor function. It is easy to fall down a trap door, but almost impossible to climb out unless you have a ladder. In the same way, it is fast and easy to encode using a trapdoor function, but very hard to decode unless you have the key.

RSA uses a very simple trapdoor function: multiplication of large numbers. It is very easy to multiply two numbers but surprisingly hard to factor the result. Because of this, RSA can encode data very efficiently, which is then very difficult to decode without a key.

### Encoding

To encode in RSA, a number is taken to a very large power (*k*) modulus (*m*)[5]

(encoded message)=(message)$^k$ mod($m$) **equation 1**

The numbers *k* and *m* are known as the public key and are available to everyone, so anyone can fall down the trapdoor and encode a message.

In order for the encrypted message to be difficult to decode, *m* and *k* have to be very large (in the order of several hundred digits), so large that encoding by multiplying the message by itself *k* times can take a very long time, even for a computer. In order to avoid this, encoders use an algorithm called *successive squaring* to take powers. A number is squared many times, and each time, the power is divided by two and the answer is reduced by the modulus so that numbers never become too large. For example

| | | |
|---|---|---|
| $23^{166}$mod(63) | | |
| $23^{2}$mod(63) | power=83 | 166/2=83 |
| 529mod(63) | | multiply out t, i.e. $23^{2}$ |
| 25mod(63) | | 529mod(63)=25 |
| 25*23mod(63) | power=82 | 83-1=82 |
| 575mod(63) | | multiply out, i.e. 23*25 |

[5] Modulus refers to a way of reducing numbers. The mod function makes numbers cyclical. Instead of having numbers from negative infinity to infinity, the only numbers that exist are the numbers between 1 and the mod. In mod(4) for example, all numbers are equated to a number between 1 and 4. 5=1,6=2,7=3 ect. This allows vary large numbers to be dealt with as if they were small numbers because they will never be greater than the mod. Another way to view moduli is that they represent the remainder after division. 13 mod(4)=1 because 13/4=3 remainder 1

8mod(63)    575mod(63)=8

$8*23^2$mod(63)    power=41    82/2=41 Etc.

Repeat until the power is reduced to zero. This seems very tedious on paper, but it is much faster than doing all that multiplication because you never have to deal with numbers larger than the key itself. A computer can do this very quickly.

## Decoding

In order to decode an encoded message, we need the private key, which is the ladder that will allow us to climb out of the trapdoor. In RSA the private key is made up of two prime numbers[6], *p* and *q*. The public key, *m,* is merely the product of *p* and *q*. As long as *p* and *q* are large enough, say several hundred digits, it will take years for anyone to factor *m* and this keeps the private key secure (see Introduction page 4). With modern algorithms and computing resources, a 512 bit key takes approximately two weeks to break using the most advanced factoring algorithms available.[7] It is estimated that a 1024-bit key would take nearly a year to break with the technology available now[8].

The process of decoding utilizes Euler's formula which tells us that if *a* and *m* are relatively prime[9], then:

$$a^{\varphi(m)}=1 \ \mathrm{mod}(m). \qquad \textbf{equation 2}$$

*p* and *q* are used to find the φ(*m*), or Euler's Totient function[10]. To decode, we must find the multiplicative inverse[11] of *k* mod(φ(*m*)), which will act as *k*'s opposite. If we took the message to

[6] This is a number that has no factors except itself and 1.

[7] http://en.wikipedia.org/wiki/RSA

[8] http://en.wikipedia.org/wiki/Cryptographic_key_length

[9] Two numbers are relatively prime when they have no common factors.

[10] The Euler Totient function or Euler φ is the number of relatively prime numbers. For example, φ(9)=6 because 1,2,4,5,7 and 8, are all relatively prime to 9.

[11] When a number is multiplied by its multiplicative inverse the product is 1 (i.e. ½ * 2 =1)

the power of $k$ mod($m$) to encode it, we must take it the opposite power to decode it. This opposite power is the multiplicative inverse.

$a^{k*j}=a^1=a$ **equation 3**

So to prove that it works:

1. $a^{k}$ mod($m$) encoded message-to decode we have to do something to this equation to get it equal to $a$
2. *a*= (*a*^*k*)^*j* mod(*m*) take to *j*th power, where *j* is the inverse of *k*
3. $=a^{(k*j)}$ mod($m$)
4. $k*j-x*\varphi(m)=1$[12] definition of $j$ as the inverse of $k$ mod($\varphi(m)$)
5. $k*j=1+x*\varphi(m)$ rearranged
6. $a=a^{1+x*\varphi(m)}$mod($m$) substitute line 5 into line 3
7. $a=a^1*a^{\varphi(m)\wedge x}$mod($m$)
8. $a=a^1*1^x$ mod($m$) $a^{\varphi(m)}=1$mod($m$) is Euler's formula (equation 2)
9. $a=a^1$mod($m$) original message!!

So to decode we must find the inverse of $k$ mod($\varphi(m)$), which we called $j$ above.

(message)=(encoded message)$^{j}$ mod $\varphi(m)$ **equation 4**

We tried several different methods of finding $j$, which we describe below, but before we find $j$, it is important to find $\varphi(m)$.

### Finding φ(*m*)

Recall that $\varphi(m)$ is defined as the number of integers less than $m$ which are relatively prime to $m$. Finding the Euler φ, for a general number $m$, may at first seem complicated. However, because we know the factors of $m$ we can find it fairly easily. We know that for any

[12] For example, the equation $k$=4 mod(m) can be rewritten as $k + x$*mod($m$) = 4, where x is some integer.

prime number $p$, the $\varphi(p)$ is one less than $p$ because all the numbers less than a prime number are relatively prime to it.

$\varphi(p)=p-1$ **equation 5**

We also know the φ of a number is equal to the φ of its factor multiplied together.

$\varphi(yz)=\varphi(y)*\varphi(z)$ **equation 6**

As $m$ can be factored into the two prime numbers, $p$ and $q$, we put equations 5 and 6 together and we get

$\varphi(m)=(p-1)*(q-1)$ **equation 7**

### Decoding with Euler's Formula

Finding $j$, the inverse of $k$, is harder, and it took us several tries to come up with a way to do it that was fast but did not compromise security. In our first attempt, we used Euler's formula again (equation 2).

$a^{\varphi(n)}=1 \mod(n)$ if a, n are relatively prime.

$\varphi(n)=g$

$a^{g}=1\mathrm{mod}(n)$ substitute in $g$

$a*a^{g-1}=1\mathrm{mod}(n)$

but, $a*a^{-1}=1\mathrm{mod(m)}$ definition of an inverse

therefore, $a^{-1}=a^{g-1}\mathrm{mod(m)}$

so the inverse is $g$-1

If we apply this to our encryption system, with $a=k$ and $n=\varphi(m)$, the inverse of $k$ mod $\varphi(m)$ is $\varphi(\varphi(m))-1$. This works because when we pick our key, we make sure that $k$ and $\varphi(m)$ are relatively prime so the Euler’s Formula applies.

To use this method, we must find $\varphi(\varphi(m))$. In order to do this we must factor $\varphi(m)$.

Factoring numbers of this magnitude, several hundred digits, is time consuming. Because of this our decoding script was taking a long time, even for relatively short keys

### Smooth Key Generation

Because our decoding script was taking a long time, we decided to custom-create keys that would be easy to factor and so speed up decoding. By multiplying many small numbers together we generated large smooth numbers, numbers that are easily factorable because their factors are all fairly small. We generated these numbers until we found one, such that the number one greater than the original smooth number was prime. These prime numbers would make up our private key, our $p$ and $q$. They were prime so that when multiplied together they would make a secure m, but the $\varphi(m)$ was easily factorable, because $(p-1)$ and $(q-1)$ are our smooth numbers.

Unfortunately this compromised our security. We were able to factor our public key using Pollard's P-1 factoring algorithm. This is a factoring algorithm that is designed to find factors of numbers just like our keys, smooth numbers that one less than prime numbers. It was able to factor our keys very quickly. It took less time to break the code than to generate new keys. Clearly we could not use this system to generate keys or to decode.

Once we had cracked our code, we decided that we would have to find a new way to decode that did not involve factoring the $\varphi(m)$.

### Decoding with Euclid's Extended Algorithm

Recall that to decode we must find a number $j$ such that:

$(k*j+x*\varphi(m))=1$ **Equation 8**

Euclid's Extended Algorithm provides an efficient system to find $x$ and $y$ such that $a*x+b*y=1$ where $a,b$ are relatively prime. We know $k$ and $\varphi(m)$, and they are relatively prime, because that is how the public key is chosen initially. Therefore, we can set $a=k$ and $b=\varphi(m)$ and use Euler's Extended Algorithm to solve for *j* and *x* (see equation 8). The *x* is unimportant for the purpose of decoding, but solving for it is a by-product of the algorithm.

The algorithm works like this:

| | | |
|---|---|---|
| 1. | 11x+30y=1 | As an example |
| 2. | Start with a series of equations found by dividing and finding the remainder at each step | |
| i. | 30=2*11+8 | |
| ii. | 11=1*8+3 | |
| iii. | 8=2*3+2 | |
| iv. | 3=1*2+1 | Until the remainder equals 1 |
| 3. | $a$=30, $b$=11 | |
| 4. | 30=2*11+8 | Equation i. |
| *5.* | $8=a-2b$ | Rearrange and substitute $a$ and $b$ |
| 6. | 11=1*8+3 | Equation ii. |
| 7. | $b=1*(a-2b)+3$ | Substitute equation form step 5 |
| 8. | 3=b-(a-2b) | Solve for remainder |
| 9. | $3=-a+3b$ | Simplify |
| 10. | 8=2*3+2 | Equation iii. |
| 11. | $a-2b=2*(-a+3b)+2$ | substitute equations from step 5 & 9 |
| 12. | $2=a-2b-2*(-a+3b)$ | solve for remainder |
| 13. | $2=3a-8b$ | Simplify |
| 14. | 3=1*2+1 | Equation iv. |
| 15. | $(-a+3b)=1*(3a-8b)+1$ | substitute from steps 9 & 13 |

16. $1=(-a+3b)-(3a-8b)$ solve for remainder

17. $1=-4a+11b$ Solution!!!

So, if $k$ were 30 and $\varphi(m)$ were 11 then the multiplicative inverse of $k$ $\bmod(\varphi(m))$ would be -4.

Once we have found this inverse, we decode the message by taking it to the power of the inverse, and find the original message. We use successive squaring to do this the same way we did to encode.

$$\text{message}=\text{encoded message}^{j} \bmod \varphi(m) \qquad \textbf{equation 4}$$

One problem we ran into when we decoded with the Extended Euclidean Algorithm is that about 50% of the time, it gives a negative number as the inverse of $k$. Our original successive squaring code did not account for this so we had to add another clause. If the inverse of $k$ is negative the program reduces it by the modulus of $\varphi(m)$, so it becomes its equivalent positive number.

## Key generators

With our new method for decoding, we no longer needed our smooth keys and our code was once again secure. We made a new key generator that picks a random number of a specified length and then finds the next prime number after this[13]. To do this, it tests to see if the number is prime, is it is not, it goes on to the next number. It skips all even numbers for efficiency, and then tests each number against probabilistic prime testing algorithms (see Prime Finding page 19). Once we have two of these numbers, they become the private key, $p$ and $q$. We just set $k$ as 2^16+1, which is prime, so its relatively prime to any key we might pick, unless that key is a multiple of 2^16+1, which is very unlikely. This is a commonly used value for $k$ that is used because it is small enough to allow easy encoding, but large enough to be secure. If the power

[13] This is called a next prime function

and the message are too small, message$^k$ may be smaller than the very large modulus *m* and this would allow for easy decryption.

### Letters to Numbers

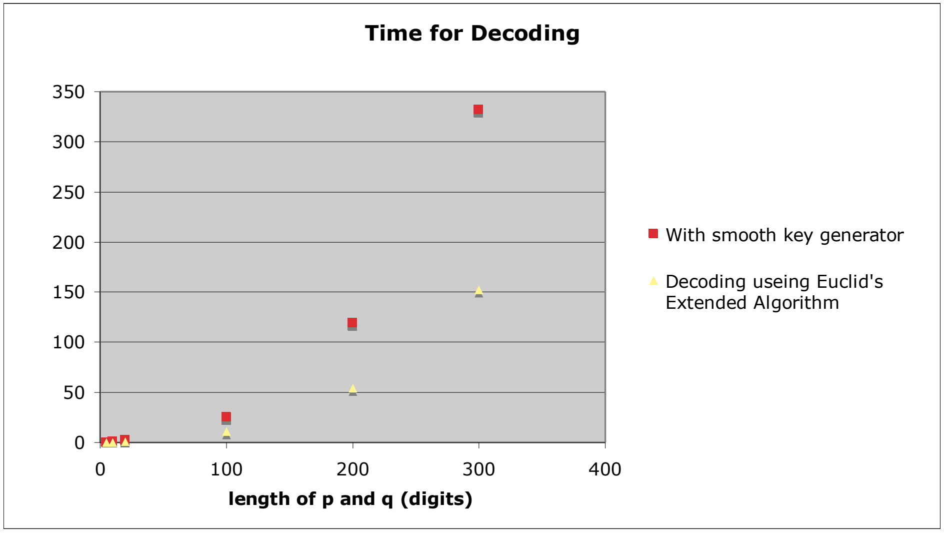
We wanted to encode actual text instead of just strings of numbers so we made a script which converted letters to numbers. Using GP's built-in function, called Vec-Small, we converted each letter into its ASCII equivalent, and then added 100 to all of them so that each letter would always have a three-digit equivalent. Then we concatenated these numbers together to create a large number and encoded the result, using the method described above.

The RSA algorithm requires that the message be shorter than the public key *m*, which acts as the modulus during successive squaring. If it is not, the message will be reduced by the modulus before it is encrypted and information will be lost or distorted. Because of this, we broke our message up into chunks 2 digits smaller than *m*. If, for example, *m* was 300 digits long, and the message was 500, we would split it into two messages, one that was 298 digits, and one that was 202 digits long. We stored each "chunk" in an array, and then encrypted each part of the message separately.

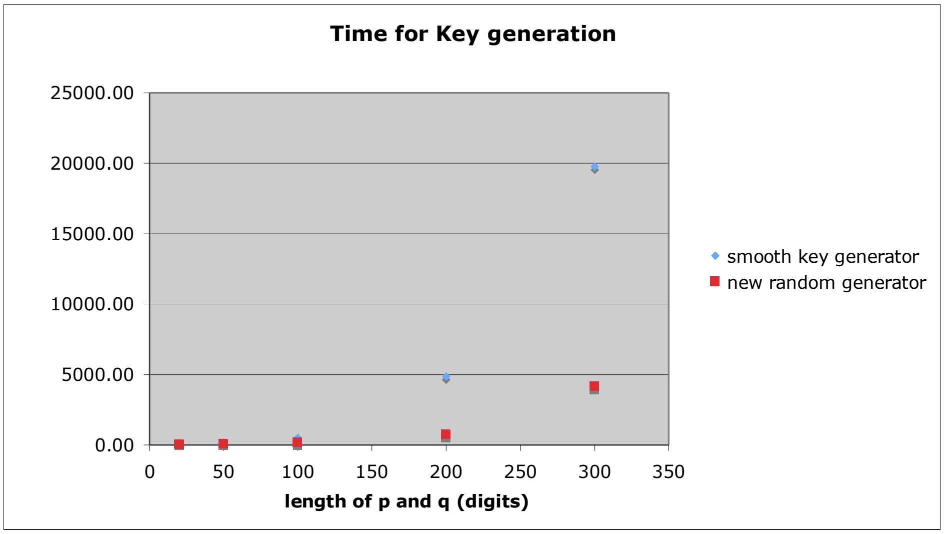
Incidentally, this is another reason that longer *m*'s make for more secure encryption. . Messages can be longer and there is much less of a chance of a person who is trying to brake the code finding a pattern between two different messages by analyzing the frequency of identical chunks and thereby guessing what the message means.

### Prime Finding

When decoding with the Extended Euclidean Algorithm, both the decoding process and the key generation happen much faster (see graphs 3 and 4). However the key generation still is not fast enough to implement on a mobile device. It is taking a long time because it has to test numbers to see if they are prime. It is basically a next prime function. It picks a number and then systematically tests numbers greater than this number to determine their priamlity. When it finds a prime number it stops. We decided to test different ways of checking primality and see which one, or which combination of them, was the fastest.



*Graph 3- Decoding Speed*



*Graph 4- Key Generation Speed*

### Eratosthenes' Sieve

Eratosthenes’ Sieve is the deterministic method for testing primality. It is based upon the idea that each factor of a number less than the number's square root has a partner greater than the square root. In order to determine whether or not a number is prime, we need to divide by every prime number below the square root of the number we are testing. If we reach the square root and have not found a divisor, we can be certain that the number in question is prime. For example, say we have the number 447 and we wish to determine whether or not it is prime. Using this method, we would divide by 2, 3, 5, 7, etc. until we get to the square root of 447 (or we find a prime factor of 447, in which case we immediately determine its compositeness). 447 is composite: first we divide by two and return a remainder of one, then divide by three and return a remainder 0. This indicates that 447 is indeed divisible by three. We now know that 447 is a composite number, and therefore ignore it and move on to the next number.

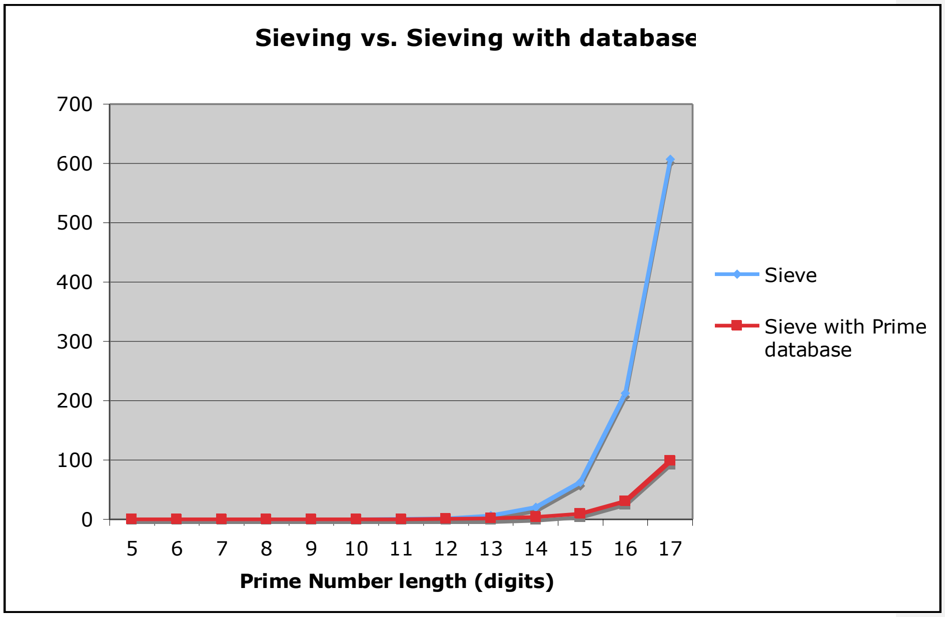
For instance, there is no reason to check even numbers above 2 since they are never prime. The same goes for multiples of 3. As a result of these two facts, we know that all prime numbers will be in the form of $6k\pm1$, where $k$ is a positive integer. Therefore, we only need to check numbers that are in this form. This greatly improved performance, as we are only checking 1/3 of all numbers. If none of them divide it evenly, then the number is prime.

While this method has 100% accuracy, it takes a very long time to generate prime numbers of even 20 digits, much less than is required for use as a RSA key.

Due to both its initial speed and very rapid decline in testing time, the Sieve's best use is in combination with other primality tests. While using the tests below, we can begin by dividing by the first several hundred primes, thereby weeding out some composites quickly and using the probabilistic algorithms for numbers with no small factors.

### Prime Database

After discussing sieving, key generation and the possible use of distributed computing to generate keys, we decided that a large database of prime numbers might be beneficial to our project. We constructed our database using only the sieve method because we wanted our database to be absolutely accurate. We were concerned about accuracy because some primality tests produce pseudoprimes, which are composites that will act as primes. Running a single laptop for several weeks, we accumulated 4 gigabytes worth of prime numbers stored in text files, which contain all prime numbers up to about 11.3 billion. We proceeded to use these numbers to test the efficiency of sieving by every odd number versus only prime



*Graph 5- Sieving with a database*

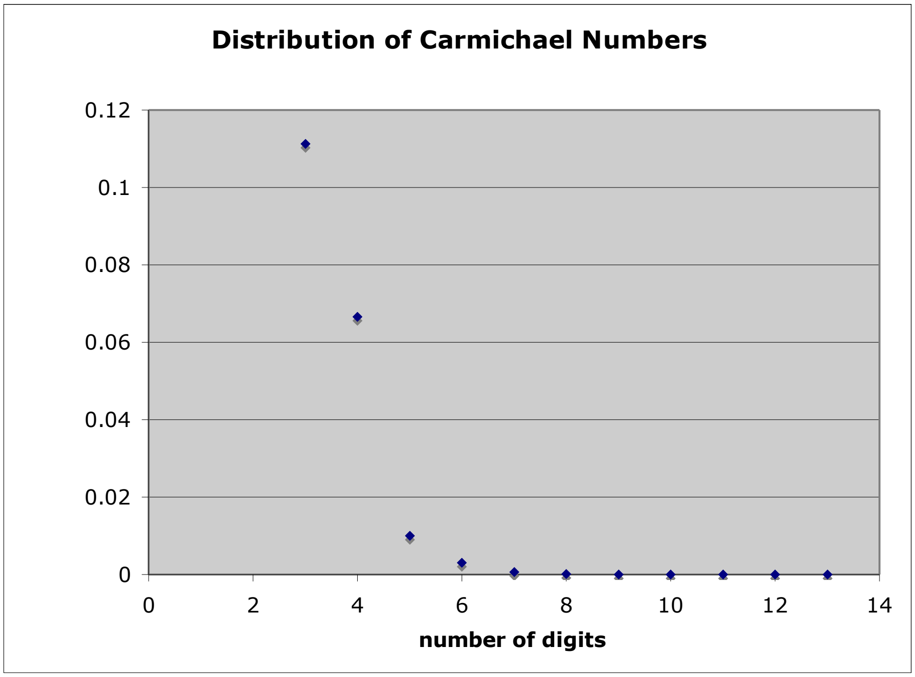
As can be seen above, if there are more primes in a database, for the Sieve to divide by, the Sieve will have to divide by fewer numbers, as it can then skip all multiples of these primes. This increases the speed tremendously.

### Fermat's Little Theorem

Fermat's Little Theorem states that if a number, *p,* is prime then

$a^{p-1} = 1 \bmod(p)$ $a$= any integer **equation 9**

So to test if a number is prime, select a random number *a*, take it the power of the number to be tested, *p*, and then see if it equals 1 mod(*p*). However there are a few composites that will "survive" this test and come up as prime even though they are not. We call these strong pseudoprimes. To counter this, we repeat the test several times, changing our base with each test. The more times it is tested, with different *a*'s, the more accurate the test is. While this greatly decreases the chance of identifying a composite as prime, there is always the small chance that the test will report false primes. However there are some numbers which behave like prime numbers, and return 1mod($p$) for every integer $a$, but are really composite numbers. These numbers are called Carmichael numbers. A Carmichael number will *always* act as a prime under Fermat's little theorem, no matter what base we use. They are pretty rare and they get even rarer as they get bigger, (see graph) but we still have to be aware of them. Most Carmichael numbers also have fairly small factors and so can be eliminated by combining this method with a very basic sieve.



*Graph 6 Distribution of Carmichael Numbers*

As can be seen in Graph 6, there are very few Carmichael numbers and a number is less likely to be a Carmichael number if it is as large as the numbers we deal with in RSA. Because of this, we were able to use the Fermat primality test, with little fear of its inaccuracy.

When we implemented this algorithm in PHP, we only repeated Fermat 5 times, each time with a different base. While this might not be the most accurate, we wanted to use the same number of iterations as we used for Rabin Miller so we could compare them. For implementation in the market, we would likely increase the number of trials to around 10, but for our purposes we found 5 trials to provide a good balance of speed and certainty.

### The Rabin-Miller Method

Let $p$ be the number we are testing. $q$ is some number such that $p$-1=$2^k q$. In other words, if we are testing a large number $p$ for primality, we start with $p$ being odd. We subtract 1 to get an even number, and then divide the result by 2 until we end up with an odd number. $q$ is that odd number, and $k$ is the number of times we divided by 2. If either of the following equations is true, then $p$ has a 75% chance of being prime[14].

$a^q$=1 mod($p$) $a$= any integer **equation 10**

$a^{2\wedge i * q}$=-1mod($p$) for i=0,1,2…$k$-1 **equation 11**

Like Fermat's Theorem, the more times this test is done, with different integers $a$, the more certain we can be that it has correctly identified a prime number. However, we can never be perfectly sure that the number we are testing is prime. The chance that it will report a composite as prime is $1/4^k$, where k is the number of times we run the test. Unlike Fermat's Theorem, the Rabin Miller test has no numbers like the Carmichael numbers, which will "act" prime no matter how many times you do the test and are still composite. This makes Rabin Miller a very trustworthy test.

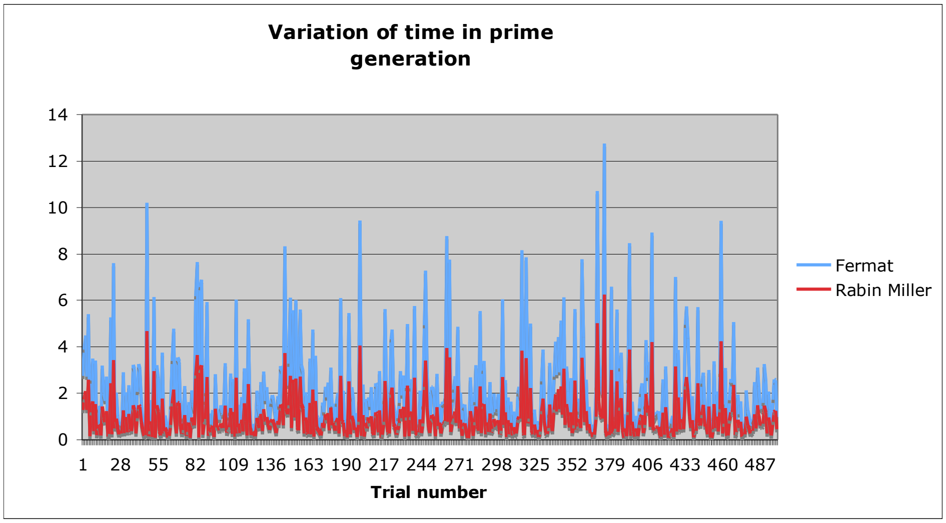
In our trials, we repeated the Rabin Miller test 5 times to verify primality. This means that for a given number, Rabin Miller tells us that it is composite with 100% confidence, or alternatively tells us that it's prime with a probability of error of 1/4^5 =0.00098, or less than

[14] Silverman, Joseph H. *A Friendly Introduction to Number Theory*.

0.01% chance of error.  For implementation in the market, we would likely increase the number of trials to around 10, but for our purposes we found 5 trials to provide a good balance of speed and certainty.

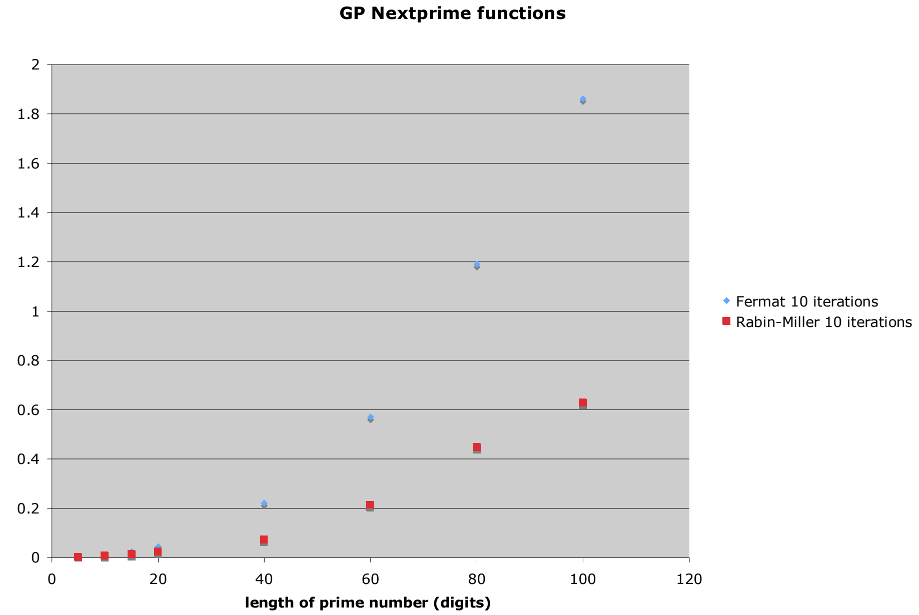
## Results:

First we tested all three primality testing methods separately in both GP and PHP. We found a great variety of times for all digits for every test. We believe that this occurs because sometimes the next prime after the random number chosen is very close and other times it is not. In order to compare each type of test, we timed each one 500 times at different digit lengths and took the average to those times.



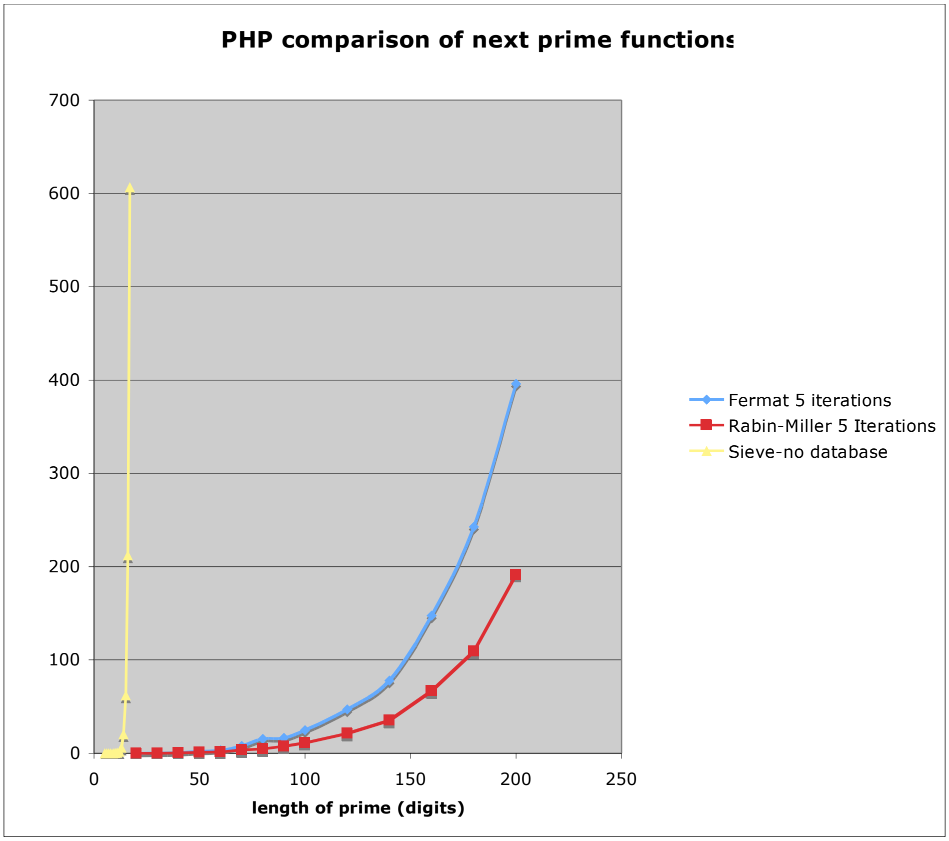
*Graph 7 Variation of time in Next Prime Functions (PHP).*

As can be seen in this graph, there is a great variation in the time taken for each next prime function, because of the random nature of the way it chooses numbers, and how far these numbers might be from prime numbers.



*Graph 8 Next Prime function in GP*

Rabin Miller was much faster in GP. This difference became more pronounced as the length of the desired prime number grew.

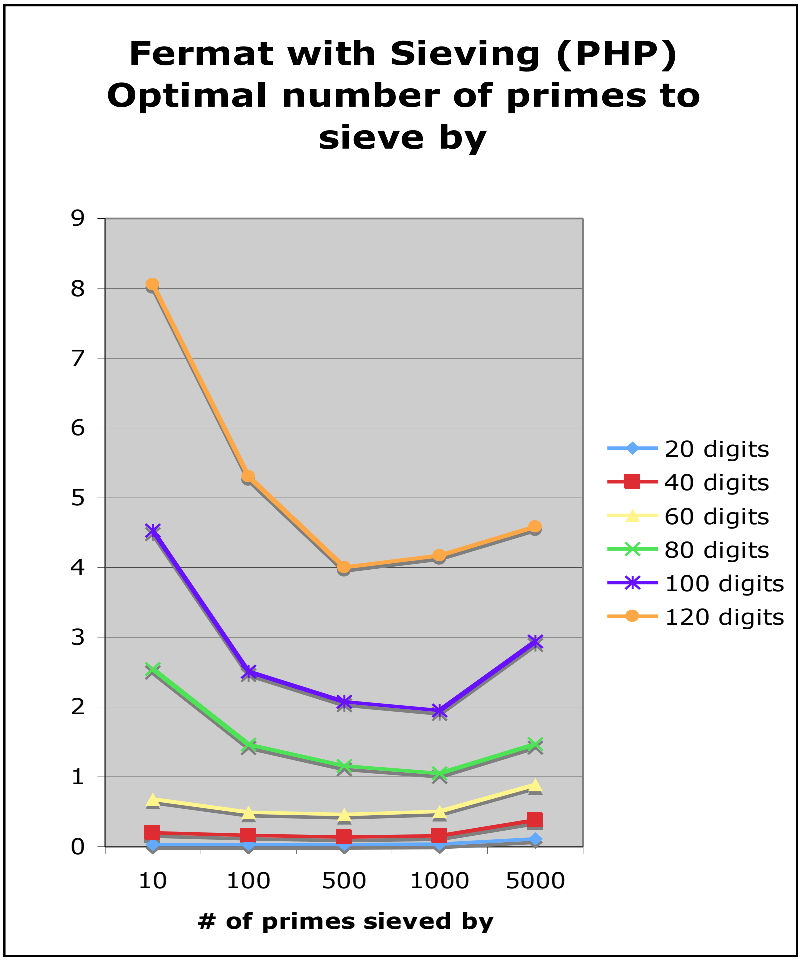


*Graph 9 Next Prime functions in PHP*

Rabin Miller was much faster in PHP. This difference became more pronounced as the length of the desired prime number grew. We can also see from this graph, that Sieve is much too slow to be of any use to us on its own for the large numbers we use in RSA.

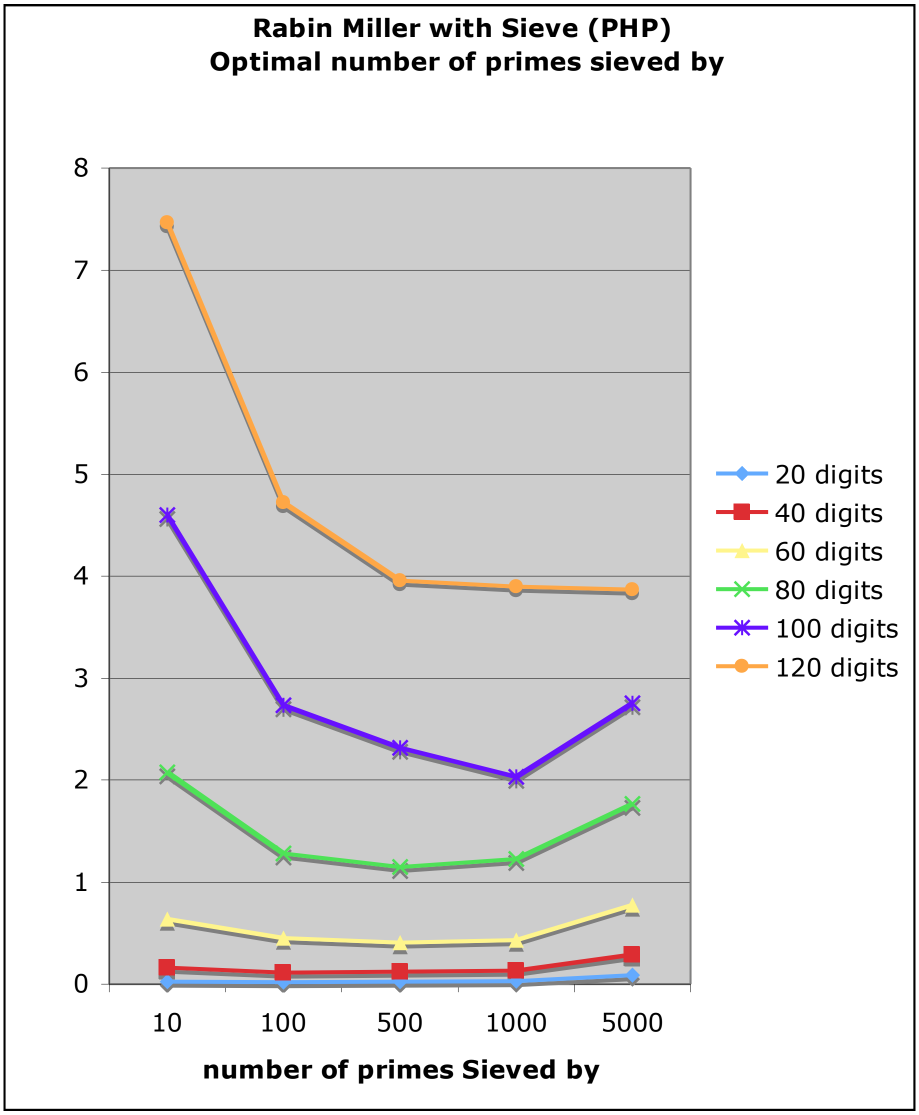
While it was clear that Rabin-Miller was the fastest of the three, in both PHP and GP, we wanted to see if combining Rabin Miller or Fermat with the Sieve would improve efficiency. The theory was that if we could eliminate some numbers quickly by dividing by smaller primes, there would be fewer numbers to test and the whole algorithm would be faster.  We also thought this might improve accuracy as Sieve is a 100% accurate test. Many of the Carmichael numbers have smaller factors and so would be eliminated when we used this method.

First we had to figure out how many primes was the ideal number to sieve by before starting Fermat or Rabin-Miller. We compared different numbers of primes at different length digits.



*Graph 10 Optimal number of Primes for Fermat with Sieve*

The optimal number of primes to sieve by before using Fermat increases as the length of the desired digit increases. See Appendix D for more detailed graphs.

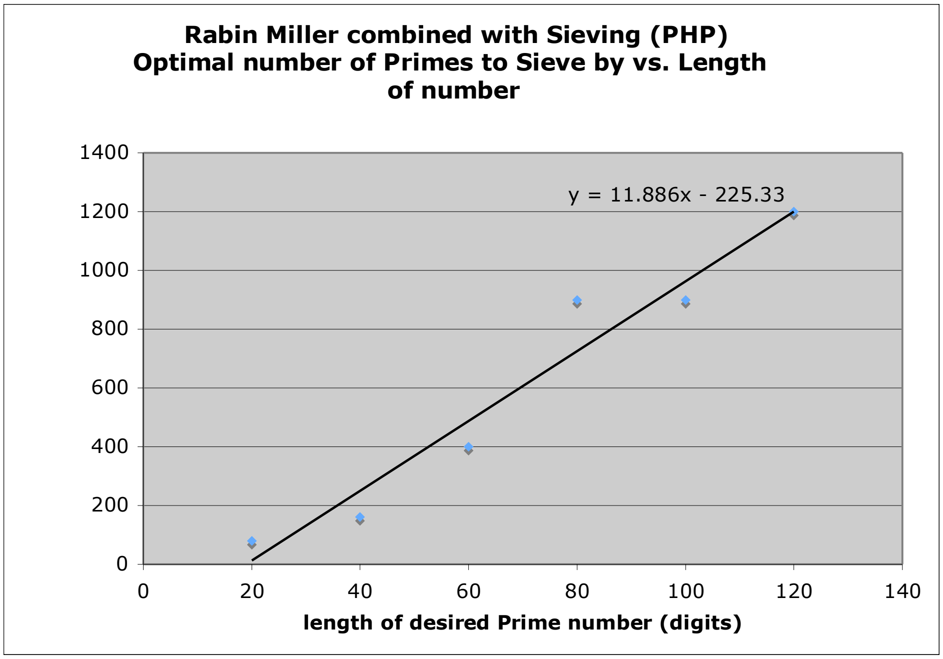


*Graph 11 Optimal number of primes for Rabin Miller with sieving*

The optimal number of primes to sieve by before using Rabin-Miller increases as the length of the desired digit increases. For more detailed graphs of each digit length, see Appendix D

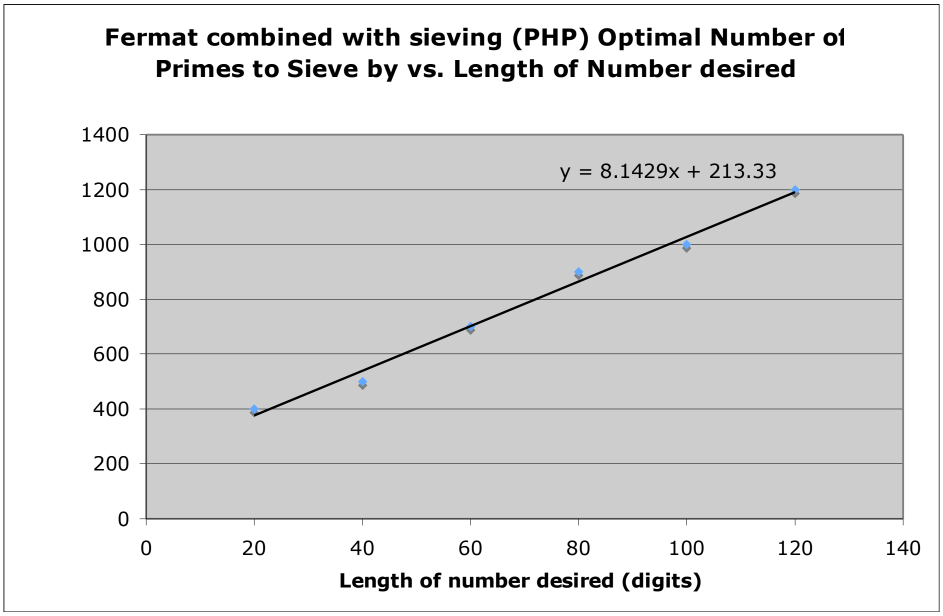
As can be seen in Graphs 10 and 11 above, the optimal number of primes to sieve by depends on the length of the prime desired. The larger the prime, the more prime numbers are needed. This makes sense as the longer the number, the more numbers there are with larger factors, and the more primes are needed to eliminate these numbers.

Once we had found the optimal number at each specific length, we made a graph to show this relationship.



*Graph 12 Rabin Miller with Sieving (PHP): optimal primes vs. length of digit.*

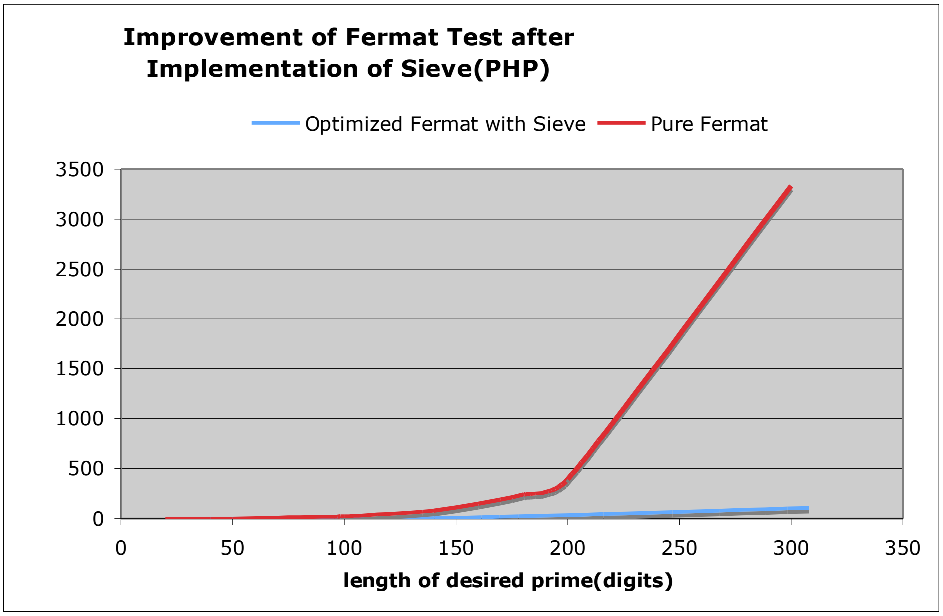
There appeared to be a linear relationship between the length of the digits desired and the optimal number of primes to sieve by before doing Rabin Miller. We used this relationship to create a script that would always sieve by the optimal number of primes.



*Graph 13 Fermat with Sieving (PHP) optimal primes vs. length of digit*

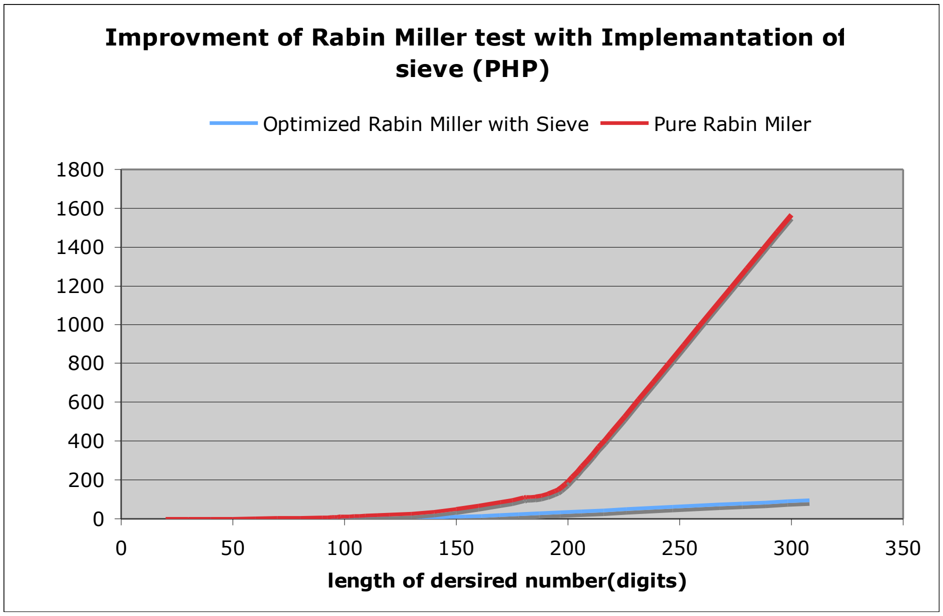
There appeared to be a linear relationship between the length of the digits desired and the optimal number of primes to sieve by before doing Fermat. We used this relationship to create a script that would always sieve by the optimal number of primes.

Once we had optimized the number of primes that we would sieve by we compared these hybrid algorithms to those with the Fermat or Rabin miller tests alone. We found the efficiency had improved greatly with the addition of the sieve. We were also pleased that the optimal number of primes was only about 40,000 for the largest keys we would generate (600 digits for 4096 bit RSA). This amount of primes would be easy to keep in a database and transport.



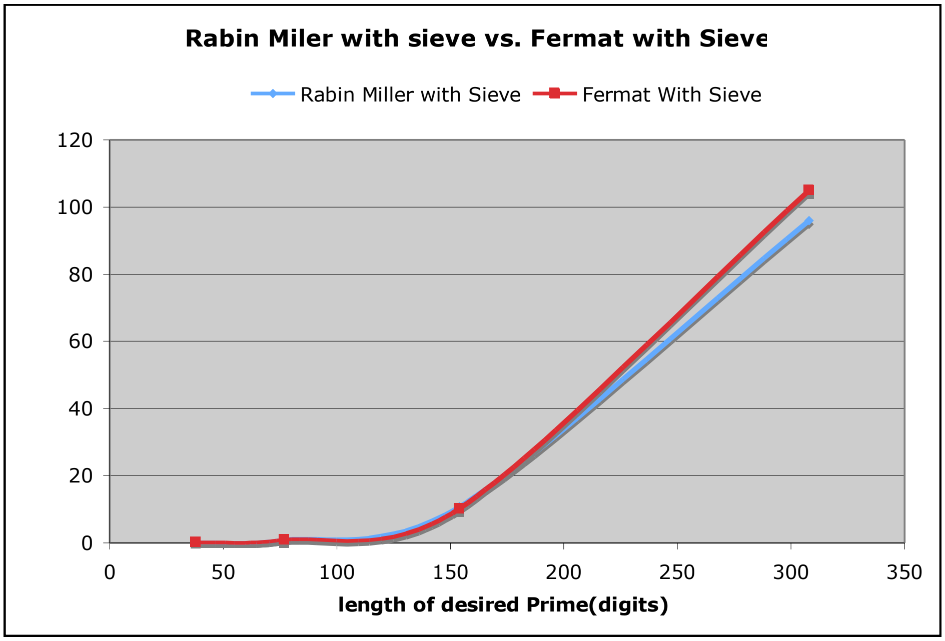
*Graph 14 Fermat with sieve vs. Fermat without*

It seems that Fermat's efficiency was much improved by the addition of sieving by optimal number of primes. It became much faster, especially for larger numbers, which are the ones we are most concerned about in implementing RSA.



*Graph 15 Rabin-Miller with sieve vs. Rabin-Miller without*

It seems that Rabin-Miller's efficiency was much improved by the addition of sieving by optimal number of primes. It became much faster, especially for larger numbers, which are the ones we are most concerned about in implementing RSA.

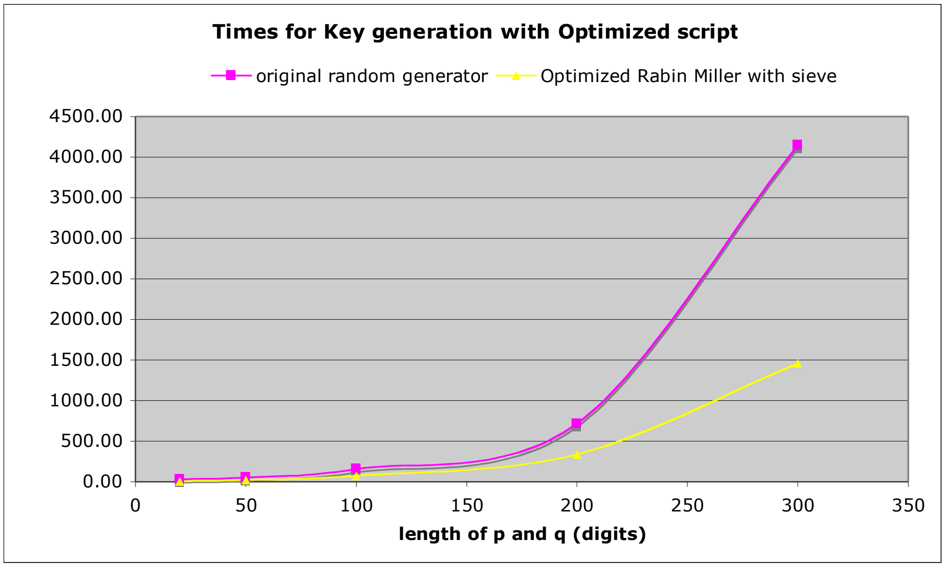


*Graph 16 Rabin Miller with sieve vs. Fermat with Sieve.*

Rabin-Miller and Fermat took almost the same amount of time once we had implemented them with the sieving. Because of this we decided to use the Rabin-Miller with sieving as our final key generation script because it was just as fast Fermat with sieving and more accurate due to the fact that Fermat misses the Carmichael numbers.

In our final key generator we used Rabin Miller with our optimized sieving. It appears to be slightly faster than Fermat with optimized sieving, and is also more accurate, as Rabin Miller has no Carmichael numbers or other equivalents.

We found that Rabin miller was the fastest probabilistic primality test. Its efficiency was improved significantly when implemented with sieve. Our final key generator is much faster and more accurate than our original key generator, and this will help us to generate keys quickly. Now all three parts of the encryption algorithm are optimized to the best of our ability, and are much faster than those we started with.



*Graph 17 The Final results-much Faster than before (GP)*

Our new optimized key generation was much faster then the key generation we were using before (see key generation page 18 for information about our key generator before it was optimized.

We were able to optimize RSA to be much faster. In GP, at 2048-bit (308 digits), encoding now takes 3 ms, decoding takes about 150 ms, and key generation takes only 1.5 seconds. This is faster than we had ever hoped, and if we can get it to be this fast in PHP, using a GMP library then we will be able to implement it for mobile devices, without inconveniencing the user.

## Implementation:

### Graphical User Interface (GUI)

As we started to program our encryption and decryption processes, we started to think of what we would do for our final user interface. Due to the communicative nature of any encryption system, it made the most sense to create a program that would allow for message transmission, including the encryption and decryption of text and images. Ultimately, we wanted to create something where multiple users could log into the system, using a uniform program to encode and decode, and communicate using an instant-messenger like program that would be easy to use and would not impede the speed of message transmission by any noticeable amount. We are on the path to this result, but at present only have a PHP-powered website that users can utilize to encode and decode messages, as well as generate keys.

We decided that we would create a user interface that would be easy for both experienced programmers and novice computer users to utilize effectively. We brainstormed the use of several available programming languages in order to find the one that would make the interface as simple as possible while still maintaining the features we wanted to include in the final product. We decided to use Flash, as it would allow us to design and use a more complex graphical system to enhance what the user would be working with. Our goal was to make it look nice as well as to run smoothly and easily.

After looking over several Flash-based programs, we settled on using the latest Adobe Creative Suite 4's Flash program as it gave us the most options and the greatest freedom. We wanted our UI to look smooth and contemporary and to separate into an encoding, a decoding, and a key generation section to avoid any confusion for the user. We programmed a script to make the menus expand and contract depending on which one was selected. We built a series of simple menus that would enable the user to select if they wanted to encrypt, decrypt, or generate a key. Next, we programmed the text boxes to allow the user to generate his or her own key to use instead of one of the pre-programmed keys, encrypt his or her message and select how secure he or she wants his or her message to be, and to decrypt using either the user's own key or the pre-programmed key he or she used to encrypt.

At this point, the implementation requires the user to have a PHP server in order to run the UI, which is a huge drawback to our current implementation. Secure usage requires the user to generate the key and perform the encoding on his or her own computer, but in PHP this requires the user to install and configure a PHP server on their own computer. In the future we would like to create a Java applet that will run in the browser which will allow the user more freedom, and which would hopefully be much quicker than PHP due to Java's superior big-number libraries.

## Images

Writing the algorithms themselves may have been an arduous task, but encoding text is trivial and we decided to experiment with different mediums, since our encoding, decoding and key generation algorithms will work with any data that can be translated into an array of numbers.

Given that images and sound files are commonly transmitted forms of data in instant messaging applications, we considered modifying our algorithms to work with those inputs. To date, we have a functional image encoding algorithm up and running. Our original goal was to encode them in real time, however the program we were using for this could not handle our hundred digit long keys. We have decided to use PHP instead for the number crunching, so the image will be fully encoded before the user sees the output.

To encode an image, we must first break the image into pixels, each 6 digits long (these are in base 16 A-F, using two digits to represent the red, green, and blue values). First, we attempted to encode each pixel and then paint it onto a new canvas containing each encoded pixel. The problem here is that we would get an image with distorted colors, retaining the same shape, borders, and textures as our original image. This was unacceptable, as the some of the original information conveyed was still present. To remedy this we had to encode fractions of pixels. Implementing this required a buffer structure. Essentially each time we look at a pixel, we read the hex value in digit by digit. If, at any time during this reading in, the length of our buffer exceeds 9 digits (if we are encoding 1.5 pixels at a time, 6*1.5=9), we encode our buffer and paint it onto our encoded canvas. This fixed the worst of our problems, however there is one downside - if we have a large group of pixels in the same row with the same color, such as eight white pixels in a row (white is FFFFFF in hex), our buffer will contain 5 sets of 7 F's and these will encode as the same thing (giving us a group of purple), so we still are vulnerable to groups of pixels. Keep in mind though that this problem of border/texture preservation is only present when GIFs or PNGs are encoded. If an image with a lossy encryption method (namely JPGs) are encoded, there will be a subtle blur applied over the whole image. This subtle blur means that if a white pixel is encoded, and a pixel that is *almost* white (FFFFFE) is encoded, we might get a pink and a brown instead of two orange pixels. Our buffer structure increases the "messiness" of the image even further.

## Conclusion

We learned through this project that design and implementation are two very different things, and that the nuts and bolts of programming are more difficult than we originally thought. We also increased our knowledge of modular arithmetic, successive squaring, and taking the modular inverses of exponents. We discovered a lot about primality testing and the different means of discovering prime numbers.

We were able to implement RSA in a manner that is secure and very fast. What once took minutes is now happening in a matter of seconds. Once we have PHP connected to the GMP library, our algorithm should be fast enough to implement on a mobile device or website.

Our most significant achievement was implementing our scripts in PHP, particularly our

Rabin Miller function. We spent many hours debugging this program, and when it worked, we rejoiced. However, Rabin Miller was still slower in PHP than in GP, even when optimized.

In the future, we would like to optimize our image decoding to run faster, which would allow us to implement this in a user-friendly program for secure image transmission. We would like to write our user interface in Java next so that our program could run in a browser, making it accessible to more users.  Eventually we would like to implement our system on a mobile device like the iPhone in order to provide an ultra-secure and fast instant messaging service.

## Acknowledgements

We would like to thank our mentor, Thomas, for allowing us to work at his house and spending his spring break proofreading this final report. We would like to thank his wife, Emily, for her toleration of us five teenagers in her home. We would like to acknowledge Desert Academy and Chris Zappe for their support and attention to our project. Our thanks go to Nick Bennett and Joshua Thorpe for answering all of our stupid questions. And we would like to thank the New Mexico Supercomputing Challenge and its officials for giving us this opportunity.

# Appendices :

## Appendix A: Codes:

**GP-Encoding code-Sucessive Squaring**

This script takes a message to a power mod (m) using successive squaring and we used it to encode. It is very fast, less than 3 ms for 600 digits numbers.

```
encode(message,power,m)=
{local(b);
b=1;
until(power<1,if(power%2==1,b=(message*b)%m);
message=(message^2)%m;
power=floor(power/2));
return(b);
}
```

**GP-Final Decoding code**

This script decodes using Euclid's Extended Algorithm

```
decode(message,power,p,q)=
{
local(phim,solution);
local(x,g,a,v,w,b,y,s,t,c);
phim=(p-1)*(q-1);
a=power;
b=phim;
x=1;
g=a;
v=0;
w=b;
until(w==0,t=g%w;
c=(g-t)/w;
s=x-c*v;
x=v;
g=w;
v=s;
w=t;
);
y=(g-a*x)/b;
print("x="x);
print("g="g);

solution=sucsquare(message,x,p,q);
return(solution);

}
```

**GP Successive Squaring for decoding**

This was the Successive Squaring script we used for  decodeing that incorporated negative numbers, should Euclid’s Extended Algorithm give a negative number.

```
sucsquare(ssa,ssk,ssp,ssq)=
{local(ssb,ssm);
ssb=1;
ssm=ssp*ssq;
if(abs(ssk)!=ssk,ssk=ssk%((ssp-1)*(ssq-1)));
until(ssk<1,if(ssk%2==1,ssb=(ssa*ssb)%ssm);
ssa=(ssa^2)%ssm;
ssk=floor(ssk/2));
return(ssb);
}
```

**GP- Find a number of a particular length**

This script has a particular length inputed, and it spits our a random number of that length. i.e. rlength(300) gives a random 300 digits long number

```
rlength(x)=
{
local(b,e,n);
b=10^(x-1);
e=8;
for(i=1,x-1,e=10*e+9);
n=random(e)+b;
return(n);
}
```

**GP-Fermat’s Little Theorem Next Prime Function:**

This script finds a random number of an imputed length and finds the next prime after this number, determining primality from Fermat’s Little Theorem

```
fnxt(z)=
{
local(k,x,plist,found,failedsieve,failedfermat);
k=10;
x=rlength(z);
if(x%2==0,x++);
until(found==1,
failedfermat=0;
for(i=2,k,if(sucsquare(i,x-1,x)!=1,failedfermat=1));
if(failedfermat==0,return(x));
x+=2;
);}
```

**GP-Rabin Miller Next Prime function**

This script finds a random number of an imputed length and finds the next prime after this number, determining primality from Rabin Miller’s test

```
rmnxt(z)=
{
local(n,v,x);
v=10;
x=rlength(z);
n=x;
if(n%2==0,n++);
until(rm(n,v)==1,n+=2);
return(n);
}
/* Rabin Miller Primality Test */
/*Returns 1 if x is prime, returns 0 if it's composite*/
rm(p,f)=
{
local(k,q,p1,a,e,success);
k=0;
p1=p-1;
q=p1;
if(p==2,return(1));
if(p%2==0,return(0));
until(q%2==1,q=q/2;k++);
e=k-1;
for(j=2,f,
a=random(p-2)+2;
success=0;
if(sucsquare(a,q,p)==1,success=1,;
/*Test
#2*/for(i=0,e,if(sucsquare(a,(2^i)*q,p)==p1,success=1)));if(success==0,return(0)));
return(1);
}
```

**GP-Final Key Gen script**

This is the optimized key generator that sieves by the optimal number fo primes and them does Rabin Miller.

```
np1(z)=
{
local(x,s,plist,test,failedsieve,i,failedrabin,found);
x=rlength(z);
if(x%2==0,x++);
plist= [3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89,
97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181,
191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397,
401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,
509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619,
631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743,
751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863,
877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947];
until(found==1,;
i=1;
failedsieve=0;
failedrabin=0;
until(i==160||failedsieve==1,
if(plist[i]==x,return(x););
if(x%plist[i]==0,failedsieve=1);
i++;
    );
if(failedsieve==0,if(rmfast(x,10)==0,failedrabin=1,failedrabin==0));

if(failedsieve==0&&failedrabin==0,found=1,x+=2));
return(x);
}
rmfast(n,p)=
{

local(k,d,q,g,bt,probablycomposite);
k=0;
q=n-1;
if(n==2,return(1));
if(n%2==0,return(0));
until(q%2==1,q=q/2;k++);
/*
print("q is " q);
print("k is "k);*/

for(i=1,p,

    a=random(n-3)+2;
    g=gcd(a,n);
    if(g>1,return(0));
    bt=sucsquare(a,q,n);

        if(bt==1||bt==(n-1), ,
            probablycomposite=1;

            for(j=1,k-1,

                bt=sucsquare(bt,2,n);
                if(bt==(n-1),probablycomposite=0);
            );
            if(probablycomposite==1,return(0));
        );
);
return(1);

}
```

### Appendix B: Equations:

| | | |
|---|---|---|
| **1.** | encoded message=message$^{k}$ mod(m) | RSA Mathematical Model |
| **2.** | $a^{\varphi(m)}=1$ mod(m). | Euler's Totient Formula |
| **3.** | $a^{k*j}=a^{1}=a$ | Definition of Exponential Inverse |
| **4.** | message=encoded message$^{j}$ mod $\varphi(m)$ | Solution of RSA Encrypted Message |
| **5.** | $\varphi(p)=p-1$ | Definition of the φ of a prime number |
| **6.** | $\varphi(yz)=\varphi(y)*\varphi(z)$ | Multiplication of the φ of two numbers |
| **7.** | $\varphi(m)=(p-1)*(q-1)$ | Multiplication of φ of two prime numbers |
| **8.** | $(k*j+x*\varphi(m))=1$ | Euler's Extended Algorithm |
| **9.** | $a^{p-1} = 1$ mod(p) | Fermat's Little Theorem |
| **10.** | $a^{q}=1$ mod(p) | Equation 1 of the Rabin Miller algorithm |
| **11.** | $a^{(2\wedge k)*q} = -1$mod(p) for i=0,1,2…k-1 | Equation 2 of the Rabin Miller algorithm |

## Appendix C: Glossary of Important Terms:

**512 and 1024 bit-** the number of digits of the key used in RSA. 512 bit is equivalent to $2^{512}$, and 1024 bit is equivalent to $2^{1024}$. There are also 2048 and 4096 bit RSA keys as well.

**Carmichael Numbers:** Psudoprimes for Fermat's Little Theorem primality test. No matter how many times you do the test, these numbers will always come up prime, when they are really composite.

**Euler Totient function or Euler φ-** the number of relatively prime numbers. For example, φ(9)=6 because 1,2,4,5,7 and 8, are all relatively prime to 9.

**Integer**-A whole number. No fractions no decimals. RSA is a function that uses Number Theory, which uses only positive integer. This is why when we divide instead of fractions, or decimals we have remainders.

**Modulus**- a way of reducing very large numbers. The mod function makes numbers cyclical. Instead of having numbers from negative infinity to infinity, the only numbers that exist are the numbers between 1 and the mod. In mod(4) for example, all numbers are equated to a number between 1 and 4. 5=1,6=2,7=3 etc. This allows vary large numbers to be dealt with as if they were small numbers because they will never be greater than the mod. Another way to view moduli is that they represent the remainder after division. 13 mod(4)=1 because 13/4=3 remainder 1

**Multiplicative Inverse**-When a number is multiplied by its multiplicative inverse the product is 1

**Next Prime**-This is a function that finds the next prime after the inputted number.

**Pollard's P-1 algorithm**-Factoring Algorithm used to factor smooth numbers that are prime when thye have one added to them.

**Prime**-This is a number that has no factors except itself and 1.

**Private key**-the information kept by the sender of the public key and includes the factorization of *m* into p and q, which are two huge prime numbers.

**Psudoprime**-A number which tests positive in one of the probabilistic primality tests but is really composite.

**Public key**-two numbers, the power k and the modulus pq, that are published so that one is able to encrypt and send messages. The public key is usually generated by someone and sent to the person who wants to encrypt and the sender keeps what is known as the private key.

**Rabin Miller**- This is a primality test that runs the number through two tests, and if the number fails both, then the number is composite. However, if the number passes either of the tests, then it is prime. See page 16 for more information on the two tests.

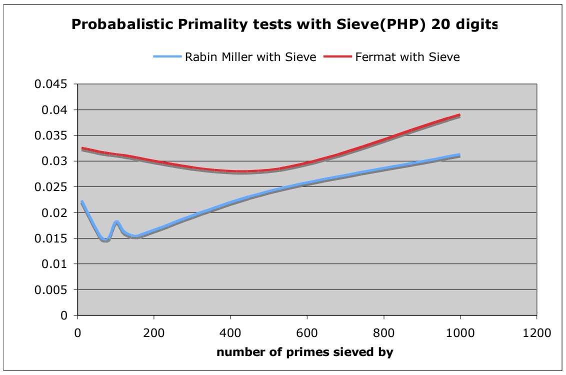
**Relatively Prime**-This means that they have no common factors between them.

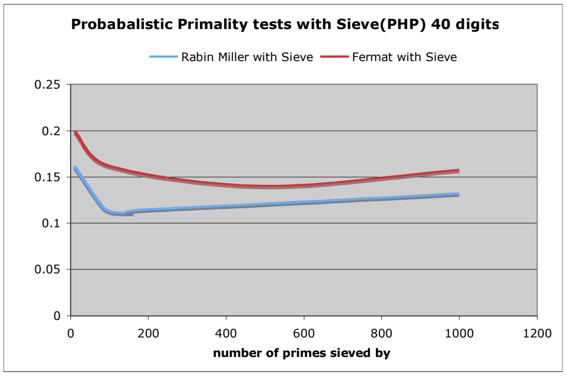
**Sieve**-Eratosthenes' Sieve is a function that checks primality based on the presumption that if a number has a factor, one factor will be under the square root of the number while the other will be over the square root and checks all the numbers under the square root for factors.

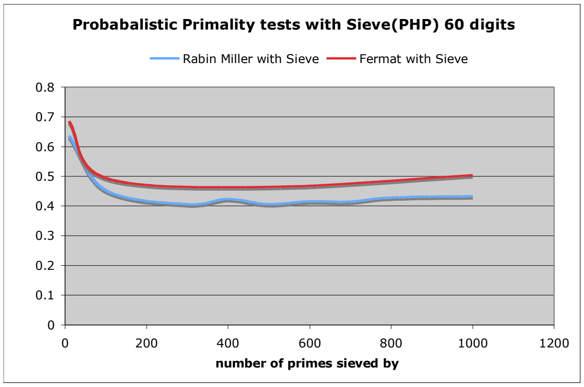
**Smooth number**-Number made of small prime numbers multiplied together. Easily factorable

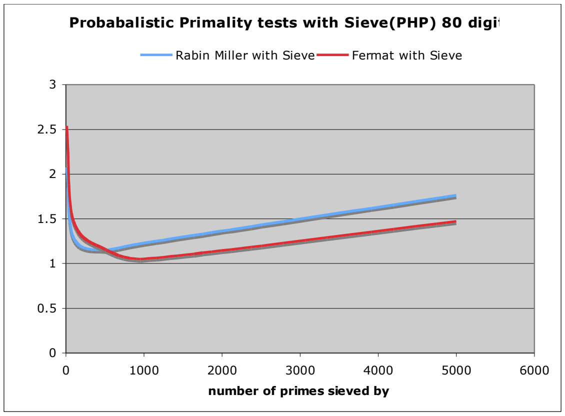
**Trapdoor function**-A function that is easy to do, but very hard to undo.

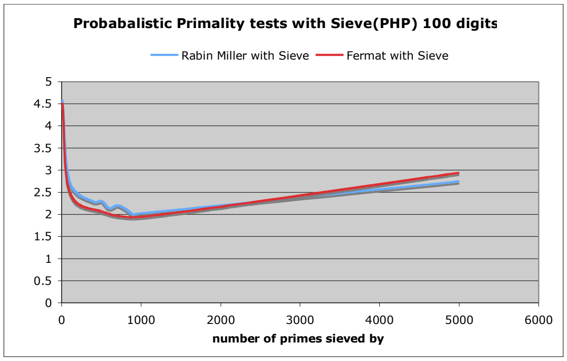
## Appendix D: Graphs:

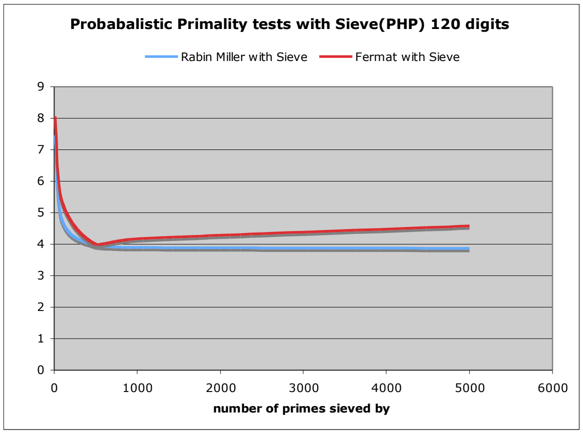







Probabalistic Primality tests with Sieve(PHP) 80 digi
Rabin Miller with Sieve
Fermat with Sieve
3
2.5
2
1.5
1
0.5
0
0
1000
2000
3000
4000
5000
6000
number of primes sieved by


Probabalistic Primality tests with Sieve(PHP) 100 digits
Rabin Miller with Sieve
Fermat with Sieve
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
1000
2000
3000
4000
5000
6000
number of primes sieved by


Probabalistic Primality tests with Sieve(PHP) 120 digits
Rabin Miller with Sieve
Fermat with Sieve
9
8
7
6
5
4
3
2
1
0
0
1000
2000
3000
4000
5000
6000
number of primes sieved by

## Bibliography

“Cryptographic key Length”. <<http://en.wikipedia.org/wiki/Cryptographic_key_length>>. (accessed March 24, 2009)

“*GP Pari*.” n.d. http://pari.math.u-bordeaux.fr/ (accessed Sept. 14, 2008).

“List of Carmichael Numbers” http://www.kobepharma-u.ac.jp/~math/notes/note02.html (accessed February 5, 2009)

*“PERT.”* n.d. http://www.netmba.com/operations/project/pert/ (accessed March 24, 2009)

“*Prime Number Hide and Seek: How the RSA Ciper Works*.” n.d. http://www.myppetlabs.com/~breadbox/txt/rsa.html (accessed Oct. 13, 2008).

“RSA Laboritories”  Security Division of EMC. <<http://www.rsa.com/rsalabs/node.asp?id=2152>> (accessed March 24, 2009)

“*RSA*.” n.d. http://en.wikipedia.org/wiki/RSA (accessed Sept. 29, 2008).

“*Smooth Numbers*.” n.d. http://en.wikipedia.org/wiki/Smooth_Numbers (accessed Oct. 30, 2008).

Doctor Anthony. *The RSA Encryption*. http://mathforum.org/library/drmath/view/60582.html. The Math Forum. (accessed February 2, 2009)

Raiter, Brian. *Prime Number Hide-and-Seek: How the RSA Cipher Works.* http://www.muppetlabs.com/~breadbox/txt/rsa.html (accessed January 12, 2009)

Riesel, Hans. *Prime Numbers and Computer Methods for Factoring*. 2nd ed. Royal Institute of Technology in Sweden: Birkhauser Boston, 1994.

Schneier, Bruce. *Applied Cryptography*. 2nd ed. Canada: John Wiley and Sons, Inc., 1996.

Silverman, Joseph H. *A Friendly Introduction to Number Theory*. 3rd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2006.

Stalings, William. *Cryptography and Network Security*. 1st ed. Upper Saddle River, NJ: Prentice Hall, 2005. Pages 242-243

**Comment [RM1]:** Fix this later