Pedro Avila

Computer Science
California Polytechnic State University
San Luis Obispo

Senior Project Paper
Advisor Dr. T.J. Kearns
2004.06.01

qpeN™
RSA Cryptosystem for ASCII files

For my parents, André and Kátia Ávila, and my brother, Paulo Ávila,
without whom I never would have made it this far.

Table of Contents

Abstract

This senior project, qpeN™, is an implementation of the RSA algorithm using basic Java elements. The cryptosystem consists of 3 command line programs (source code) run by any user with a JRE installed (JREv.1.4.2).

The objectives for the project include generating a pair of secure public and private keys (enciphering and deciphering keys) of any size for any user (In the examples and tests, 1024 bit keys are generated for two users). The sender can use these keys to encipher and sign an ASCII text message into a file using the receiver's public key and the sender's own private key. The receiver can verify and decipher a signed file using the sender's public key, and the receiver's own private key. The application then translates the authenticated deciphered message to text such that the ASCII text of the deciphered message is the same as the original message. Although the program's objectives do not include being used as a secure means of communication, it meets most of today's standards for what communication is 'safe'.

This project exists because the intrigue of the equations used in the RSA algorithm tickles my thoughts and curiosity. The objectives of qpeN™ do not include being used as a secure means of communication, nor is it its purpose ever to be distributed for general use. It exists because I am curious and want to learn its implementation and its features efficiently and effectively, even if not securely. I have already begun to learn what I want to learn, and if anyone else learns anything from this project, it has more than served its purpose and exceeded my expectations.

Introduction to Cryptography and RSA

Cryptography is one of the most artistic results of combining computer science and mathematics. It dates back beyond any concept of a computer, to an era when mathematics was still young. It has evolved over centuries of simple substitutions, beyond manual and mechanical ciphers into the age of electronic ciphers dealing exclusively with 1's and 0's. We stand on the edge of technological and mathematical accomplishment; in our hands, we hold the late 70's advent of public-key cryptography and the RSA cipher algorithm, which has proven strong in the face of 20th century hackers and fears only the 21st century possibility of quantum computing. It is an algorithm whose power and beauty lies in its terrifying simplicity. This particular implementation of it is done in the hope of learning by doing.

Notes on this paper

Please note that words in this paper that are **bold** are glossary words and their definitions can be found in **Appendix F: Glossary**. *Italicized* words are equations and definitions and are usually indented in a different font if they are not part of a section. Enjoy.

Cryptography before RSA

The RSA public-key cryptosystems in use today have come a long way since the **transposition ciphers** used by Spartan Generals in Greece over 2000 years ago. In fact, cryptosystems nowadays are primarily classified as substitution ciphers, and can be broken into two groups, symmetric and asymmetric ciphers, the latter having been invented only during the last century.

Prior to asymmetric ciphers and the advent of Diffie-Hellman-Merkle key exchange, it was necessary for two people who wanted to communicate securely to physically exchange secrete **keys**, either by meeting in person or via a trusted courier. The key is applied to a **cipher** and the **ciphertext** is sent to the intended recipient. Upon receipt, the key is applied to the same cipher but with the algorithm reversed, revealing the original message. This symmetry is what gives **symmetric ciphers** their name.

Several ciphers existed in the 18th, 19th and 20th centuries that proved very secure in their time. The Vigenere Cipher, the Beale Cipher, The Great Cipher of the Rossignols, Enigma and Purple, among others, either gave cryptanalysts a very hard time or still remain unbroken (Singh, 55). Indeed, the Onetime Pad Cipher, although one of the more inefficient symmetric ciphers because of its key-distribution problem, is mathematically proven to be unbreakable (Singh, 122). But all the aforementioned ciphers possess the problem of key-exchange, the classic catch-22 situation where in order for the ciphers to work, keys must be securely communicated between two parties wishing to communicate securely. Whitfield Diffie, Martin Hellman and Ralph Merkle shattered that paradigm in the 1970's.

Diffie-Hellman-Merkle key exchange and the asymmetric cipher

When thinking about exchanging messages and particularly about the problem of key distribution, it is helpful to consider 3 parties: Alice (the sender A), Bob (the recipient B) and Eve (the eavesdropper E). The problem of key exchange is that if Alice wants to send a secret message to Bob without Eve being able to read it, she must first send Bob a secret key. So how do two parties exchange a secret without already sharing a secret (the key)?

The idea is easy to describe using a thought experiment involving boxes and padlocks. If Alice wants to send a secret message to Bob, she can place the letter in a box and put a padlock on it. She does not, however, have to mail her key to Bob in order for him to be able to open the box. He can, upon receiving the box, put his own lock on the box and return it to Alice, who then takes her padlock off of the box with her key. When she sends the box back to Bob, he can use his key to remove his padlock and read the message inside the box. All the while, Eve has not been able to intercept the box and read the message because she does not have either of the keys to either of the padlocks on the box. The only problem with this experiment is that applying a series of ones and zeros to a mathematical equation is just a little bit more complicated than putting padlocks on a box. Padlocks can be removed in any order, whereas symmetric ciphers must be deciphered in the same order in which they were enciphered (Singh, 258).

A problem concerning the Diffie-Hellman-Merkle key exchange is the impracticality of communicating back and forth to exchange the key in a symmetric cipher. Applied to the real world it could take days for two people across the globe to deal with the differences in time zones and communicate their keys before being able to send a secret message that needed to have been sent yesterday. It was Whitfield Diffie in 1976 who conjured the idea that a cipher need not be symmetric (Singh, 268).

In an **asymmetric cipher**, enciphering and deciphering are not mirror functions. If Alice enciphers a message with an enciphering key, she cannot necessarily decipher it, unless she has the deciphering key. Diffie, Hellman and Merkle knew that in order for their equations to work like padlocks and be easy to calculate but difficult to reverse, they would have to have qualities of what are known as **one-way functions**, but they were not able to come up with the necessary equations. They knew that if they could find an asymmetric cipher equation, the implications for Alice and Bob would be enormous. Alice could create an enciphering key and a deciphering key. In an electronic implementation, the keys are just numbers and since the enciphering key is given to anyone who wants it, it is generally called the *public key*. The deciphering equation, which is kept secret, is generally called the *private key*. By making her public key available to Bob, Charlie, David and Frank, she can receive messages from any of them and feel secure that only she has the private key to decipher the messages. In fact, after Bob enciphers the message he wants to send, even he cannot decipher it since he doesn't have the deciphering key. Diffie, Hellman and Merkle were able to provide a foundation of ideas for cryptography revolution. They were not, however, able to provide a concrete example of the implementation of their ideas. Academic credit of for the implementation is given to another trio (Singh, 271).

Rivest Shamir and Adleman

On the eighth floor of the MIT Laboratory for Computer Science, Leonard Adleman was entirely unaware of the history of cryptography and distinctly uninterested in the paper Ron Rivest tried to share with him about asymmetric ciphers by Diffie and Hellman (Singh, 272). Eventually, Rivest convinced Adleman to help him in his search for a one-way function that would satisfy the requirements of Diffie and Hellman's equations, and the pair was joined by Adi Shamir.

For a year the trio pondered and speculated about functions that can be reversed only if the receiver has some kind of special information. The solution, which they created independently, is centered around a one-way function based on modular arithmetic, known today as RSA in honor of these three men.

How RSA works

The mathematics of the RSA cipher is based on several principles from various branches of mathematics including number theory and modular arithmetic. Number theory relates to choosing the keys and modular arithmetic is necessary to understand the enciphering and deciphering equations. During the course of this project it has become necessary to familiarize myself with both bodies of knowledge, as they are not particularly intuitive. A brief overview of the necessary topics follows.

Excerpts from Number Theory Applicable to RSA

In order to understand the RSA cipher at the most basic level one must understand a few things about prime numbers, as they have some properties that make them invaluable in the study of cryptography.

A **prime number** is *a positive integer that only has as positive integer divisors the number 1 and itself* (Coutinho, 53). It is also important to know that two numbers are **coprime**, or **relatively prime** if *they have no common factors other than 1*. Another way to define coprimality is to say that the *greatest common divisor of the numbers (or the gcd ( ) function)* is equal to 1, and it follows that if *gcd (a, b) = 1*, then there exists two integers x and y such that *ax + by = 1* ("RSA Theory").

Euler's **totient** or **φ function** of n *counts the number of positive integers less than and coprime with n*. It should follow that if n is prime, then the totient function returns *n - 1*, since, by definition all numbers less than a prime number n have no common factors with n, including 1 ("RSA Theory").

These theorems only help in attaining the public and private keys needed for RSA to work. In order for the enciphering and deciphering equations to work it is necessary to have a function that is easy to calculate, but very difficult to reverse (a **one-way function**).

Excerpts from Modular Arithmetic Applicable to RSA

Modular arithmetic is an area of mathematics that is rich in one-way functions, and is a very simple but very counter-intuitive science. Because of its cyclical (rather than linear) nature, normal, non-genius human beings have a hard time thinking about it off the top of their head.

A clock represents an application of modular arithmetic, which is sometimes called clock arithmetic (Singh, 261). In fact, time in general is cyclic in nature, and therefore can be represented as an interaction of numbers with reference to a modulus. When one person asks another in the middle of the afternoon what time it is, chances are that one of those people will inadvertently calculate a number modulo 12 (unless they are from another country, or in the military in which case they will calculate a number modulo 24). If it is 5 o'clock pm, one individual could tell the other that it is 5:00 pm, or he may say that it is 17:00 hours. As it just so happens, 17 in modulo 12 (we say *17 mod 12*) is 5 because that is the remainder in dividing 17 by 12. In fact, any number which has 5 as the remainder when divided by 12 is *congruent* to 5 in modulus 12.

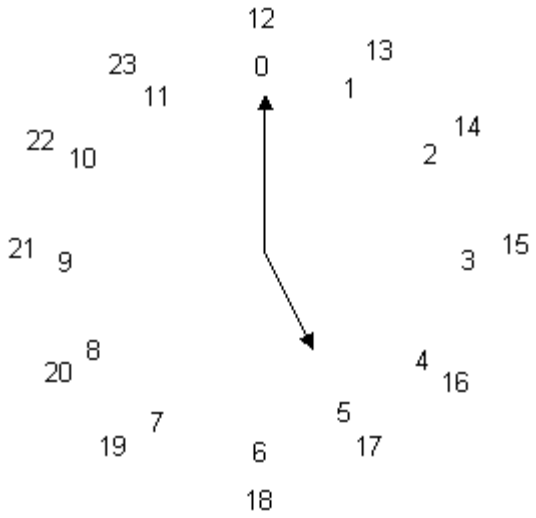


Fig X – A clock is a visual example how modular arithmetic works.

The notation $a \equiv b \pmod{n}$ means that a and b have the same remainder when divided by n; in other words, for some integer k, $a = nk + b$. In this case we say that *a is congruent to b modulo n*, where n is the **modulus** of the congruence ("RSA Theory").

The **Linear Congruence Theorem** is important in understanding how to derive certain RSA variables. In modular arithmetic, $d \times e \pmod{N} \equiv [d \pmod{N}] \times [e \pmod{N}]$. So if we say that $\alpha x \equiv \beta \pmod{n}$ and that α is coprime with n, then we can isolate x using properties of linear congruence. This calculation gives us the equation $x \equiv \beta/\alpha \pmod{n}$, or $x \equiv \beta \times \alpha^{-1} \pmod{n}$, which is useful for isolating d or e in the congruence equation $de \equiv 1 \pmod{N}$, which is used in the RSA algorithm.

Applying the Math to RSA

Now that we've discussed some of the basic principles governing the central equations of RSA, we can discuss the functions themselves. We go back to Alice and Bob and their confidential communication so desired by Eve.

Alice writes an important message intended for Bob and only Bob. Eve already can't wait for Alice to send the email containing the message so that she can read a copy of it. But Alice is aware of the dangers of email communication, so she creates her own public and private keys of a secure size. At this point in time, secure is defined as 1024 bits (Wagner, Law RSA1). Each one of the keys consists of two numbers. The **public key**, which is published in a place where people have public access, contains a number **e** and a number **N**. The **private key**, which Alice keeps secret, contains a number **d** and the same number **N**. The relationship between e, N and d are special and relate to two non-equal random prime integers **p** and **q**. The relationships Alice uses to create her public and private keys have the following properties:

$$N = p \times q$$
$$\gcd(d, \varphi(N)) = 1$$
$$e = d^{-1} \bmod \varphi(N)$$

Once Alice looks up Bob's public key, she is ready to encipher the message.

Alice converts the message written in ASCII text to a string of numbers (which is all that a piece of ASCII text represents anyway) and breaks it up into appropriately sized blocks. Alice enciphers each block of numbers as a block of ciphertext applying Bob's public key to the following enciphering equation:

$$E(M) = \text{ciphertext block} = \text{plaintext block}^{\,e} \bmod N$$

It should be noted that Rivest, Shamir and Adleman do not address the issue that in order for the algorithm to work, the value of the plaintext block must be less than the modulus. They may have considered it an implementation problem (I address such a problem in the implementation section), and so did not bother to mention it in their original paper.

As Alice computes the **ciphertext** blocks she can store them in a file and send that file to Bob. At this point (while the message is in transit), Eve gets a hold of a copy of the message, perhaps off of the mail server, or maybe she snags it right out of cyberspace. Eve tries to read it, but all she sees is a text file consisting of what appear to be blocks of randomly scattered digits and cannot make heads or tails out of it. Eve may know that the message has been enciphered using RSA and may even have the public key, but she does not have the private key and thus cannot decipher the message. In fact, Eve has no hope of obtaining the private key either by brute force or by applying algorithmic or cryptanalytic methods to the ciphertext (such as frequency analysis, or crib-to-key calculations) for reasons discussed in the next section.

Bob receives the message and realizes it is enciphered but that he is the intended recipient, meaning it was enciphered using his public key. Bob whips out his private key and applies it to the following deciphering equation with the ciphertext received:

$$D(C) = \text{deciphered ciphertext} = \text{ciphertext}^{\,d} \bmod N$$

Because of some wonderful implications of Fermat's Little Theorem (Kim, 3) and Euler's Extended Algorithm (Coutinho, 148), the enciphering and deciphering equations themselves are symmetric:

$$D(E(M)) = M \text{ and } E(D(C)) = C$$

This symmetry, applied with the asymmetric keys ensures that the deciphered message will be read only by the intended recipient and that it will be identical to the original message (Rivest, 5).

The significance of the equations described by Rivest, Shamir and Adleman in their 1978 paper is enormous for cryptographers. With the advent of Whitfield Diffie's public key cryptography and then the RSA algorithm, Alice can send and receive messages that exhibit **confidentiality**, **data integrity**, **authentication**, **authorization**, and **non-repudiation**. These 5 characteristics are still used today by RSASecurity® Inc. as a measure of what makes a communication channel secure ("RSA Services").

Why RSA is secure for now

What makes RSA secure is the size of the numbers in the key. There is no easy way to calculate or guess the private key, even with knowledge of the public key. The value of **d** can only be calculated based on values of **p** and **q**, which, ideally, are disposed of upon successful key generation. When dealing with numbers on the order of 10^{200}, people rarely grasp the actual vastness of the integer. But because **N** is such a huge number, attempting to factor it is a spectacularly unreasonable task. There is no known algorithm that works in logarithmic time for factoring a number, let alone a number as large as **N**. In order to give the reader an idea of the immensity of **N**, using all the computers on Earth to find the factors by brute force is estimated to

take many times the age of the universe (Singh, 279). I don't know any cryptanalyst determined (or foolish) enough to attempt such an intimidating task.

A conceived possibility for factoring numbers as large as those dealt with in RSA is to make use of a technology still in its infancy known as quantum computing. Without going too far into the details, quantum computing makes use of **qubits**, which unlike regular bits, can exist in all possible states at the same time (they don't necessarily have to be ON or OFF, they can be both ON and OFF at the same time). This dizzying property of **qubits** gives them the ability to calculate all possible combinations of digits in a 10^{200} sized number all at the same time, potentially reducing the run time of such a factoring calculation as previously discussed from universal age to microseconds (Singh, 329).

The implications could be astronomical.

Implementation of RSA: qpeN™

Being that RSA is a concept, a mathematical algorithm, it has many different implementations. **PGP™** (pretty good privacy) uses principles of RSA, as do most of the RSASecurity® Inc. products. Such products are replete with shortcuts to maximize efficiency and minimize runtime. For example, PGP™ does not encipher an entire message using the public key; it enciphers the key for a symmetric cipher and sends both the enciphered message, as well as the RSA enciphered key (Singh, 298). This method is advantageous in that enciphering a key (which is typically much shorter than a message) takes less time, and could have the added advantage of using a cipher that is mathematically unbreakable such as a one-time pad cipher.

My implementation of the RSA cipher (qpeN™) is but another distribution. There are very few clever cipher tricks in qpeN™ because the idea of the project is to implement as pure an RSA system as possible. By mimicking the mathematical steps of the algorithm within the code, I can learn an algorithm that drives today's security needs more efficiently.

In this next section I discuss the more interesting aspects of the implementation details and how I structure the qpeN™ code to be as intuitive as possible to a user who, after reading the mathematics section of this paper, is better versed in the structure and workings of the algorithm.

A note about code structure

qpeN™ was originally designed to be coded in C, to make the algorithm as true to the original RSA paper as possible. The lack of object orientation in C makes the code more linear, or procedural in design, which aids in making the product more closely relate to the algorithm. However, Java 1.4.2 offers many shortcuts and allows more focus on the algorithm and mathematical structure without worrying about how to implement the syntax of the more trivial mathematical functions which are tried and trusted. Thus Java is chosen as the implementation language; however the procedural structure of the program remains for my own design clarity. The program is as far from object oriented as can be, but it does work and the linear structure has helped me tremendously during the design phase.

Generating Keys - generating random prime numbers p and q

Since generating keys is a multi-step process from a mathematical point of view, the source code for qpeN™ follows the same steps. The first step in this process is to choose 2 non-equal random prime numbers of a determined number of bits.

Before we produce a random number, we must first define what it means for a number to be random. For the purposes of this project, random means that *given a sequence of numbers, there is no deterministic method to predict the next number in the sequence* (Deley, Introduction). In order to produce numbers that are random by this definition, the inherent **entropy** of other more natural systems must be harnessed.

The simplest, most obvious way to build chaos into a number is to take user input, such as random keystrokes, or mouse movement over the screen. However, this randomness is sometimes not enough. In terms of true chaos, even users can sometimes be predictable if a cryptanalyst takes into account whether or not the user is left-handed, whether the last input came from the right or left side of the keyboard, or on what circle or diagonal the last mouse movement took place. For true, unquestionable entropy, we turn to the sometimes strange (although very real) principles of quantum mechanics. Sparing the reader the dizzying details of quantum mechanics, there exist principles in the science (such as the **Heisenberg uncertainty**

principle), which prevent knowledge of all information associated with a particle at the same time. In short, one of the only ways to generate truly random numbers is to detect the presence of particles, such as those emitted by radioactively decaying material. An idea (albeit a complicated one) for building random numbers is to detect particles with a Geiger counter and at every detection interpret a 1 instead of a 0 to generate very chaotic numbers.

It would seem then, that by definition, a **deterministic machine** such as a computer is not only unfit for building entropy into a number, it is mathematically incapable of doing so…is this a problem for cryptographers? Not necessarily for the creative ones.

Since it has not been proven that the RSA cipher is NOT susceptible to cryptanalysis (unlike the **one-time-pad cipher**), all that is necessary is a degree of certainty that it will be extremely difficult to **cryptanalyze** the private key by any known methods such as large number factorization. True entropy, then, is not a concern, so long as p and q are "random enough".

Luckily, deterministic machines are very capable of producing **pseudorandom numbers** and Java 1.4.2 comes particularly well equipped with a Random class that generates a stream (or sequence) of random numbers. The beauty of this class is that the number generator can be seeded during instantiation with any number to produce a different sequence of pseudorandom numbers. It should be noted that two generators seeded with the same number will produce identical sequences of pseudorandom numbers (Java 1.4.2 API); such is the nature of deterministic pseudorandom number generators, and a problem faced by anyone implementing a public key cryptosystem.

But for those who are a little more optimistic, this deterministic nature of pseudorandom number generators is a godsend, for if 2 generators seeded with the same number must be identical, then 2 generators seeded with different numbers must be different. By seeding the random number generator used to find p and q with the current system time in milliseconds at different times, qpeN™ can almost guarantee that the only way to crypt-analyze the factors of N is to set up a system that is cyclical in nature, performing whatever task it performs on every multiple of 1000 milliseconds (1s).

Choosing p and q from their respective "random-enough" pseudorandom number sequences such that they are non-equal is as simple as calling the BigInteger class to do so and give different bit sizes to p and q. This differentiation has the added advantage that if p and q differ in length by between 10 and 20 bits, it makes cryptanalysis much more difficult. To choose p and q such that they are prime with an industry standard certainty of $1:2^{100}$ chance that they are not prime is a task trivial for the purposes of this project and left to Java's BigInteger class (Han, Description).

Generating Keys - choosing d such that gcd (d, phiN) == 1

After choosing non-equal "random-enough" pseudorandom prime numbers p and q, obtaining their product and calculating the totient function of N (*φ (N)*), we are ready to choose the private key, d. The totient function of N is the variable phiN in the source code, and defined as *φ (N) = (p-1)*(q-1)* since $N = p \ q$ and the totient function of a prime number returns the number minus one (Coutinho, 148).

In this section of the code I veer slightly off the path of the mathematical algorithm, though I do not deviate from the instructions set out by Rivest, Shamir and Adleman in their 1978 paper. Algorithmically, we should choose a random number d such that d and φ (N) are co-prime. Strictly speaking, this process translates into pseudo-code as "choose a random number d *until gcd (d, φ (N)) == 1)*." This code is an obvious loop structure that has the potential to be infinite. Fortunately,

Rivest, Shamir and Adleman suggest an alternative way to choose d only once, and guarantee that d and φ (N) will be co-prime.

Although they state in their 1978 paper in which RSA was first publicly announced that any prime number *d > max (p, q)* will satisfy the requirements, they did not explain why. Being that number theory and modular arithmetic are not intuitive for most non-genius human beings, I wanted proof that such a number *d* does indeed assert that *gcd (d, φ (N)) == 1*. I did some research, snooped around between the Math and Computer Science Department at Cal Poly and even consulted overseas Professor of Mathematics, Dr. Geraldo Avila, retired from the University of Brasilia in Brazil. I came up with the following proof, verified by Dr. Rawlings of the Cal Poly Math Department, Dr. Brady and Dr. Gharybian of the Cal Poly Computer Science Department, and Dr. Ávila of the University of Brasília:

Given - *p & q are prime numbers*
- *p != q*
- *d is a prime number*
- *d > max (p, q)*
- *N = p*q*
- *φ (N) = (p-1)*(q-1)* (φ (N), also denoted by phi (N) is the totient function of N)

Prove that *gcd (d, φ (N)) = 1*.

Proof:
Suppose to the contrary that *gcd (d, φ (N)) != 1*.
Since d is a prime number, this means that d is a factor of phiN.
Since *φ (N) = (p - 1) × (q - 1)* and d is prime, we can insist that d is either a factor of (p - 1) or a factor of (q - 1), i.e. either *(p - 1) / d = k*, or *(q - 1) / d = k*, where k is a positive integer.
This means that either *d = (p - 1) / k* or *d = (q - 1) / k*.
But this cannot be true since it is given that d > p and d > q, and therefore *d > (p - 1) / k* and *d > (q - 1) / k*.
We arrived to a contradiction, which means that the supposition that *gcd (d, φ (N)) != 1* is false.

Conclusion: *gcd (d, φ (N)) == 1*

By forcing d to be a “random-enough” pseudorandom prime number and ensuring that it is, for example, twice as long (by bit count) as the larger number of p and q, we can assert that the result meets the required conditions. Having p, q, N and d, we move on to finding e, the public key. Finding e is, for this project, a trivial task left to Java’s BigInteger class, which comes readily equipped with a modular inverse function as defined by Euler’s Extended Algorithm, and is laid out as:

$e = d^{-1} \bmod (\varphi(N))$

OR

```
e = d.modInverse (phiN);
```

Enciphering M - Ensuring the numerical value of the message

When a message is converted from ASCII to a numeric value to be thrust into the enciphering equation $C = M^e \bmod N$, the numerical value of the exponentiated message must be less than the modulus. In other words, $M^e < N$, or else deciphering the message will be impossible since it has

an infinite number of possible results. While enciphering one character of the message at a time will be easy to implement, the resulting ciphertext will be susceptible to frequency analysis. To design a cryptosystem in this day and age that can fall victim to the same cryptanalysis used to break a **monoalphabetic substitution** ciphers of 400 years ago would be a waste of technology. So in designing qpeN™, I considered two possible solutions as to how to ensure that $M^e < N$.

The ideal solution to the problem is suggested by Dr. P. Nico from the Cal Poly Computer Science Department. He suggests that I convert M to a base N number, ensuring that it will always be less than the modulus. This system allows me to encipher only one number per message, which is a fantastic way to reduce worst case run time by ensuring that at the very most, the program will only execute one large calculation each time it runs. However, as I geared up to implement the suggested solution I began to realize that although an algorithm exists for such a task, its implementation is impractical for the following reason:

The algorithm for converting a base 10 number to another base is simple:

- Let N be the decimal number to have its base changed.
- Let B be the desired base of N.
- Divide N by B, store the quotient and the remainder as Q_x and R_x respectively.
- Divide Q_x by B until $Q_x = 0$, storing R_x as this task is done.
- Reverse the string of characters $R_0R_1…R_x-1R_x$ to be $R_xR_{x-1}…R_1R_0$.
- Convert that string to its integer representation and that is the number base N.

As demonstrated in the following example, converting a number from decimal (base 10) to base 13, a place holder problem arises:

Convert 487_{10} to base 13:

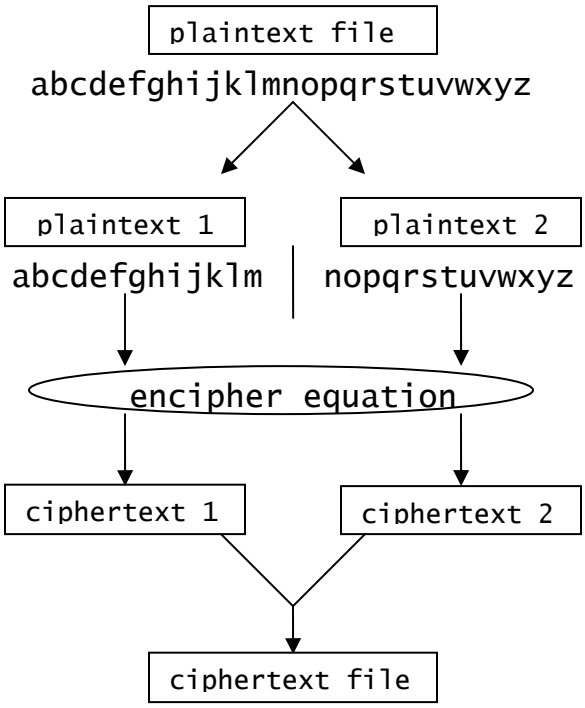
487 ÷ 13 = 37 r 6
37 ÷ 13 = 2 r 11 ← **Problem**
2 ÷ 13 = 0 r 2

Reversing the string 6-11-2 gives us 2-11-6. But how does one represent 11 as a single digit? In fact, how de we represent any number N greater than 9 and less than the base as a digit? In the hexadecimal system (base 16), we use A, B, C, D, E, F to represent 10, 11, 12, 13, 14, 15, respectively. But if the base is greater than 36 (as it will almost certainly be in the qpeN™ implementation), we will run out of decimal digits and letters in the alphabet, to say nothing of the size of the table that the program will need in order to know all the remainder values.

Because of this placeholder problem, this idea is abandoned in lieu of a less elegant but more effective solution.

Enciphering M - Ensuring the length of the numerical value of the message

Since ensuring the numerical value of the message is inefficient, why not try to ensure the length of the numerical value of the message by counting the number of digits? This solution is somewhat of a “divide and conquer” approach to the problem and much less intuitive to think about procedurally, but it gets the job done efficiently and effectively. The algorithm is shown and described below.



- As the message is read in character by character, append the character to a buffer until the buffer reaches a critical size (defined by the size of the modulus), or until the message is expired.
- At such a point, convert the contents of the buffer to a numerical value and plug that value of M into the enciphering equation.
- Output the result to a cipher-text buffer.

This solution has the advantage of facilitating the decipherment and translation of the message in the final step by providing a measured unit of the message to the deciphering equation.

Digital Signature and Authentication

As stated in the outline, the digital signature & authentication feature are optional and time-permitting. Since time permits, I opt to add the feature into qpeN™. It adds more security into the application since the receiver can positively authenticate not only that the message came from the author of the message, but also that it originated from said author. This added security comes at a cost of only about two lines of added Java code per module.

Public key exchange has been described as distributing locks to which only the intended recipient has the key (Singh, 258). Following this paradigm, signing a message would be tantamount to locking a box with a lock that only the sender possesses, but to which everyone has a key. By locking a box with two locks, one to which only the intended recipient has the key, and another which only the sender possesses but to which everyone has a key, the sender has essentially enciphered the message (put it in a state that no one but the intended recipient can read), and signed it (authenticating to anyone reading the message that the sender is the author).

With this concept in mind, signing a message is, in fact, as simple as applying the sender's private key to a message enciphered with the receiver's public key. Note that this action does not

reveal the sender's private key, but merely uses it in an unorthodox way. Authenticating the enciphered message then, is as simple as applying the sender's public key, and then deciphering the authenticated message using the recipient's private key. If it is still not clear that signing and verification are features intrinsic to the Diffie-Hellman system of public key exchange, look to the code. Proof of this quality is found in the sheer simplicity of the additions to the version of qpeN™ without digital signing and authentication in order to add the feature.

In designing the system, I had no active plans to add digital signing and authentication features. Although the actual code to add the desired functionality turns out to be simple in concept, the thought process to get there is not. The variables in the version without digital signing are intuitive and self-explanatory from the mathematical equations of the system:

```
//encipher
C = M.modPow (e, N_public);

//decipher
M = C.modPow (d, N_private);
```

In order to add the signing/authentication functionality, variable names need to be changed and reorganized. I use the following table to help me think more clearly about a situation in which Alice wants to send a signed message to Bob:

User	Eqn. Type	Resulting Obj.	Resulting Vars.	Equation	Input Param.	Input Key
Alice	encipher	$M_{enciphered}$	unsignedCT	$M_{plaintext}^{e\ Bob} \bmod N_{Bob}$	$M_{plaintext}$	Bob_{public}
Bob	decipher	$M_{deciphered}$	decipheredPT	$M_{enciphered}^{d\ Bob} \bmod N_{Bob}$	$M_{enciphered}$	$Bob_{private}$
Bob	authenticate	$M_{enciphered}$	verifiedCT	$M_{enciphered\ \&\ signed}^{e\ Alice} \bmod N_{Alice}$	$M_{enciphered\ \&\ signed}$	$Alice_{public}$
Alice	sign	$M_{enciphered\ \&\ signed}$	signedCT	$M_{enciphered}^{d\ Alice} \bmod N_{Alice}$	$M_{enciphered}$	$Alice_{private}$

After studying this table, it is apparent what lines of code must be replaced in the program to add the digital signing and authentication features:

```
//encipher
unsignedCT = plaintext.modPow (e Bob, N Bob public);

//sign
signedCT = unsignedCT.modPow (d Alice, N Alice private);

//authenticate
verifiedCT = signedCT.modPow (e Alice, N Alice public);

//decipher
decipheredPT = verifiedCT.modPow (d Bob, N Bob private);
```

It is important to note from the mathematical section that the enciphering and deciphering equations are congruent inverses of each other with respect to the modulus. This symmetric property is what makes digital signatures and authentication so easy to implement without compromising the security of the cryptosystem – chronologically, it makes one-way functions operate like padlocks.

Conclusion

As mentioned earlier, qpeN™ is not meant for distribution or deployment. Its purpose is for me to learn about cryptography and RSA in particular via its implementation (learning by doing in the Cal Poly fashion, as it were). However, qpeN™ could potentially be tweaked to be used commercially. The following changes could be made to the software:

- Redesign the encipher/decipher process to read in 8 bits at a time so that any information can be read, not just ASCII text. A random digit would have to be appended before enciphering so as no to facilitate frequency analysis.
- Restructure the code to be object-oriented. This change makes the code more portable and re-pluggable into another API, facilitating the implementation of the feature below.

Furthermore, if a new implementation is designed, it will be just as ineffective for individual users as qpeN™ if the fundamental user-level problem is not addressed: the lazy user.

A problem of cryptography today is fundamentally different at the user-level today than ever before – people need to be convinced to use it. The Vigenére Cipher of the 1500's, more advanced and secure beyond its time, was not used for 200 years after its invention (Singh, 51). At that time, it was too labor intensive and offered an unnecessarily high level of security. Users are not willing to pay the financial or temporal cost of security if it is too high – in other words, security is a commodity (Kabay, 1). According to Cooley Godward® IT and Security Manager, André Ávila, "users in general simply won't take the time to encrypt a document if it takes any time or effort at all. The only hope of their willingness to do so is by a true understanding of the reasons why."

A solution to the lazy user problem is to make the encryption transparent to the user. For example, by enciphering the message at the outgoing channel of the SMTP server, the message remains secure though the user is not necessarily aware of the encryption. Decryption can happen on the same level at the receiving end. Alice and Bob can each still have their own keys; they just don't have to manually encipher/decipher individual messages.

Such transparency can make qpeN™ usable with an industry standard degree of certainty and security, which ought to be good enough for home users who desire security in their private communications.

Acknowledgements

This project has been a tremendous learning opportunity for me. Not only has it given me the self-confidence of knowing that I am capable of doing original research and of using my own initiative to solve problems in creative ways, it has also expanded my knowledge and ability to learn new things in the fields of Mathematics and Computer Science. Through the work achieved in this project I have learned about number theory, modular arithmetic, and the history or cryptography. I believe I am also a better software engineer as a result of applying myself to the difficulties this project presented in terms of design, implementation and testing. In short, the culminating work of my experience at Cal Poly is a tremendous success and a great pleasure.

I would like to thank all the professors that put up with my incessant questions and curiosity, as well as all the people that may or may not know that they inadvertently gave me ideas, intrigue and excitement throughout my experience at Cal Poly and during this project. In particular, I would like to thank from the Cal Poly Computer Science Department:

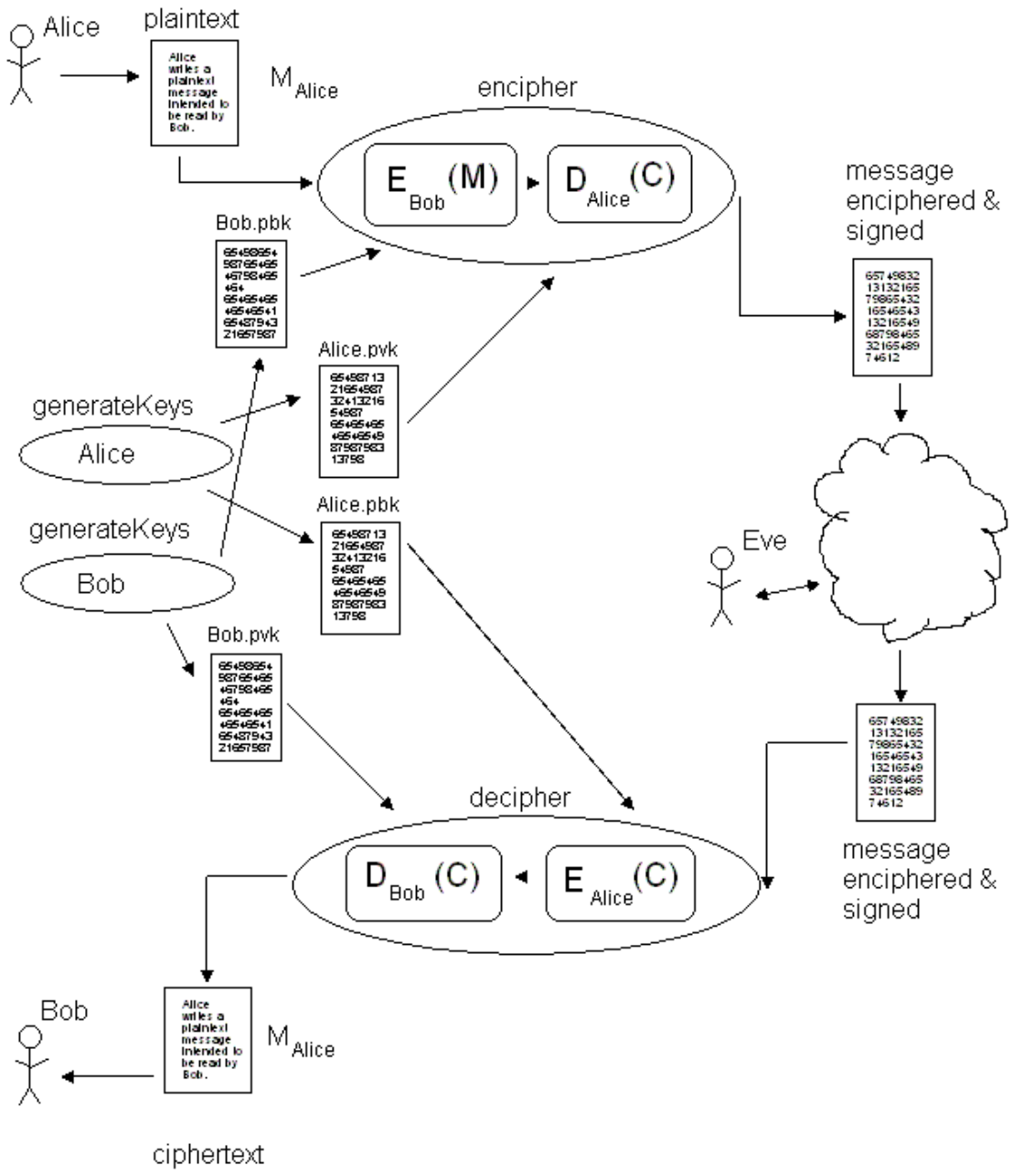
Dr. Clark Turner, Dr. Michael Haungs, Dr. Phil Nico, Dr. Aaron Keen, Dr. Hasmik Gharybian, Dr. Dan Stearns, Dr. Mei Ling Liu, Dr. Lois Brady, and Dr. Tim Kearns.

From the Cal Poly Math Department, I would like to thank Dr. Don Rawlings.

And for the inspiration for mathematics in general since my first steps, I would like to thank my grandfather, Dr. Geraldo Avila, retired from the University of Brasilia in Brazil.

The rest of my thanks go to my parents, André and Kátia Ávila, and my brother, Paulo Ávila for inspiration, humor, purpose and love beyond all reason.

Appendix A - qpeN™ Data Flow Diagram



User	Eqn. Type	Resulting Obj.	Resulting Vars.	Equation	Input Param.	Input Key
Alice	encipher	$M_{enciphered}$	unsignedCT	$M_{plaintext}^{e\ Bob} \bmod N_{Bob}$	$M_{plaintext}$	Bob_{public}
Alice	sign	$M_{enciphered\ \&\ signed}$	signedCT	$M_{enciphered}^{d\ Alice} \bmod N_{Alice}$	$M_{enciphered}$	$Alice_{private}$
Bob	authenticate	$M_{enciphered}$	verifiedCT	$M_{enciphered\ \&\ signed}^{e\ Alice} \bmod N_{Alice}$	$M_{enciphered\ \&\ signed}$	$Alice_{public}$
Bob	decipher	$M_{deciphered}$	decipheredPT	$M_{enciphered}^{d\ Bob} \bmod N_{Bob}$	$M_{enciphered}$	$Bob_{private}$

Note: Rows are in order of use throughout the application.

Appendix B - qpeN™ User Walk-through

For a demonstration of the application of this project, download the 3 .class files available at the project website at http://www.csc.calpoly.edu/~pavila/application.html.

Java 1.4.2 is necessary to run this Java application. The Java Run Time Environment version 1.4.2 can be obtained by following the link on the above project page as well.

To run the application, first make sure that any text files to be enciphered with qpeN™ are in the same directory as the .class files obtained earlier, and that any such text files contain nothing but ASCII text (Any other kind of text – special characters from MS Word, symbols, etc. – will not be properly processed by qpeN™). For the walk-through, we will call one of these files "demo.txt".

Use of qpeN™ can now begin. For this walk-through we will assume the existence of 2 simulation users in remote locations communicating via electronic means and call them A and B respectively.

Generating Keys

Each user begins by generating keys for themselves.

A types at the command line:

```
java generateKeys A 1024
```

This command produces the following key files:

```
A.pbk
A.pvk
```

B does virtually the same thing, remembering that in order to exchange messages with A, his keys must be the same size as A's keys:

```
java generateKeys B 1024
```

This command produces the following key files:

```
B.pbk
B.pvk
```

Each user makes the "*.pbk" file available in a public place (website, public directory, email, etc.) The 2 users are now ready to communicate securely.

Enciphering

A wants to send demo.txt to B securely, so A enters the following into the command line prompt:

```
java encipher demo.txt B.pbk A.pvk
```

This command outputs the file:

```
demo.txt.cph
```

The file produced contains the ciphertext of *demo.txt* enciphered using *B.pbk* and digitally signed using *A.pvk*. *At this point, only B can decipher the file and there is no way that anybody other than A enciphered the file*. A sends the signed ciphertext to B.

Deciphering

B receives the signed ciphertext and is told that it has been sent from A. B enters the following at the command line prompt:

```
java decipher demo.txt.cph A.pbk B.pvk
```

This command produces the file:

```
demo.txt.cph.txt
```

If the file has been sent by A and is actually intended for B, then the file produced contains ASCII characters identical to the original message. If the file has not been sent by A, or if the file is not intended for B, or if the file has been tampered with by some unknown third party, then running it through qpeN™'s decipher application will output a file by the correct name, but containing unreadable garbage.

Appendix C - qpeN™ generateKeys.java Source Code

```
/**
 *  Generates 2 RSA keys as text files named after the input username. The
 *          'username.pbk' file is the public key consisting of 'e' followed by a
 *          space, followed by 'N'. The 'username.pvk' file is the private key
 *          consisting of 'd' followed by a space, followed by 'N'. This arrangement
 *          is comprable to saying that the keys are:
 *<ul>
 *<li>public key: [e,N]</li>
 *<li>private key: [d, N]</li>
 *</ul>
 *<p>
 *          The two files are created (or replace exisiting files by the same name)
 *          in the current working directory (CWD).
 *
 *  @author Pedro Resende Avila - Cal Poly CSc 492 Senior Project qpeN&#8482;
 *  @param  username  a string to be used in naming the generated key files.
 *  @see           {@link}encipher, {@link}decipher
 ********************************************************************************/

import java.io.*;
import java.lang.*;
import java.math.*;
import java.util.*;

public class generateKeys
{
    /* ******************************
     * Private variable declarations
     * ******************************/
    private static BigInteger ZERO = BigInteger.ZERO;
    private static BigInteger ONE = BigInteger.ONE;
    private static boolean toAppend = false;

    public static void main(String args[])
    {
            try
            {
                    /* ************************************
                     * generateKeys variable declarations
                     * ************************************/
                    String pbkFileName = new String(args[0] + ".pbk");
                    String pvkFileName = new String(args[0] + ".pvk");

                    FileWriter pbkFile = new FileWriter(pbkFileName, toAppend);
                    FileWriter pvkFile = new FileWriter(pvkFileName, toAppend);

                    String NBitSizeStr = args[1];
                    Integer NBS = new Integer(NBitSizeStr);

                    int KeyBitSize = NBS.intValue();
                    int dBitSize = KeyBitSize;
                    int pBitSize = (dBitSize / 2) + 10;
                    int qBitSize = (dBitSize / 2) - 10;

                    BigInteger p = ZERO;
                    BigInteger q = ZERO;
                    BigInteger d = ZERO;
                    BigInteger N = ZERO;
                    BigInteger e = ZERO;
                    BigInteger phiN = ZERO;
```

```
                /************************** Key Generation ********************
                 *      - choose large random prime numbers p & q (at present,
                 *            large is defined as ~512 bits. For more, see "Laws
                 *            of Cryptography link".
                 *      - calculate N from p & q
                 *      - calculate phiN, Euler's totient function
                 *      - choose d such that is > max(p,q) AND gcd(d,phiN) == 1
                 *      - calculate e from d
                 ************************************************************/

                System.out.println("\nGenerating " + KeyBitSize + " bit keys for " +
                                        args[0] + "...\n");
                /* P code */
                Random pRnd = new Random(System.currentTimeMillis());
                p = BigInteger.probablePrime(pBitSize, pRnd);

                /* Q code */
                Random diffRnd = new Random(System.currentTimeMillis());
                long qRndSeed = diffRnd.nextLong();
                Random qRnd = new Random(qRndSeed);
                q = BigInteger.probablePrime(qBitSize, qRnd);

                N = p.multiply(q);

                phiN = (p.subtract(ONE)).multiply(q.subtract(ONE));

                /* D code */
                Random dRndGen = new Random(System.currentTimeMillis());

                /* Choose d until it is coprime with phiN AND > max between p
                 *      and q. */
                while(d.compareTo(p.max(q)) <= 0)
                        d = BigInteger.probablePrime(dBitSize, dRndGen);
                /*      Here we can assert that (d.gcd(phiN)compareTo(ONE) == 0)
                 *      because since d is prime and > max(p,q), d must be > (p-1)
                 *      and (q-1), and since phiN = (p-1)(q-1), d is greater than
                 *      the factors of phiN,and cannot, therefore be a factor of
                 *      phiN. The proof for this assertion made in section VII.C of
                 *      the Rivest Shamir Adleman paper is available in the
                 *      implementation section under Generating Keys. */

                e = d.modInverse(phiN);

                /* Output public key to file 'pbkFileName'.pbk */
                pbkFile.write(e.toString() + " " + N.toString());
                pvkFile.write(d.toString() + " " + N.toString());

                pbkFile.close();
                pvkFile.close();

                System.out.println("...Done.");
        }
        catch(Exception e)
        {
                System.err.println("Main Error");
                e.printStackTrace();
        }
    }
}
```

Appendix D - qpeN™ encipher.java Source Code

```
/**
 * Generates an enciphered file as a text file named after the input plaintext
 *          file. Takes as input the name of the plaintext file in the CWD to be
 *          enciphered, the name of the public key file with which to encipher the
 *          message, and the name of the private key file with which to sign the
 *          message.
 *<p>
 *          The 'plaintextFile.txt.cph' file is the file containing the result of
 *          enciphering the plaintext in the input file ('plaintext.txt').
 *<p>
 *          The file is created (or replaces an exisiting file by the same name) in
 *          the current working directory (CWD).
 *
 *  @author Pedro Resende Avila - Cal Poly CSc 492 Senior Project qpeN&#8482;
 *  @param         plaintextFileName             a string representing the name of the
plaintext file to be enciphered.
 *  @param         publicKeyFileName     a string representing the name of the public key
file to be used in
 *                                       enciphering the plaintext.
 *  @param         privateKeyFileName  a string representing the name of the private key file
to be used in signing the
 *                                       enciphered plaintext.
 *  @see           {@link}generateKeys, {@link}decipher
 ******************************************************************************/

import java.io.*;
import java.lang.*;
import java.math.*;
import java.util.*;

public class encipher
{
    /* ******************************
     * Private variable declarations
     * ******************************/
    private static BigInteger ZERO = BigInteger.ZERO;
    private static BigInteger ONE = BigInteger.ONE;
    private static int ASCIICharLength = 3;
    private static int criticalNMDifference = 3;
    private static boolean toAppend = false;

    public static void main(String args[])
    {
            try
            {
                    /* ***************************
                     * Main variable declarations
                     * ***************************/
                    String ptFileName = new String(args[0]);
                    FileInputStream ptFileStream = new FileInputStream(ptFileName);
                    BufferedReader message = new BufferedReader(
                            new InputStreamReader(ptFileStream));

                    String pbkFileName = new String(args[1]);
                    FileInputStream pbkFileStream = new FileInputStream(pbkFileName);
                    BufferedReader publicKey = new BufferedReader(
                            new InputStreamReader(pbkFileStream));

                    String pvkFileName = new String(args[2]);
                    FileInputStream pvkFileStream = new FileInputStream(pvkFileName);
                    BufferedReader privateKey = new BufferedReader(
                            new InputStreamReader(pvkFileStream));

                    FileWriter signedCTFile = new FileWriter(ptFileName + ".cph", toAppend);
```

```
BigInteger e = ZERO;
BigInteger Npbk = ZERO;
BigInteger d = ZERO;
BigInteger Npvk = ZERO;

String pbkStr = "";
String pvkStr = "";
Integer iKey = new Integer(0);
StringBuffer eBuf = new StringBuffer();
StringBuffer NpbkBuf = new StringBuffer();
StringBuffer dBuf = new StringBuffer();
StringBuffer NpvkBuf = new StringBuffer();


/* Build public key from the file */
while(publicKey.ready())
{
        //iPBK = new Integer(publicKey.read());
        pbkStr = Integer.toString(publicKey.read());

        /* convert pbkStr to ASCII char */
        iKey = new Integer(pbkStr);
        pbkStr = String.valueOf((char)iKey.intValue());

        if(!pbkStr.equals(" "))
        {
                eBuf.append(pbkStr);
        }
        else
        {
                while(publicKey.ready())
                {
                        pbkStr = Integer.toString(publicKey.read());

                        /* convert pbkStr to ASCII char */
                        iKey = new Integer(pbkStr);
                        pbkStr = String.valueOf((char)iKey.intValue());

                        NpbkBuf.append(pbkStr);
                }
        }
}
publicKey.close();

e = new BigInteger(eBuf.toString());
Npbk = new BigInteger(NpbkBuf.toString());

/* Build private key from the file */
while(privateKey.ready())
{
        pvkStr = Integer.toString(privateKey.read());

        /* convert pbkStr to ASCII char */
        iKey = new Integer(pvkStr);
        pvkStr = String.valueOf((char)iKey.intValue());

        if(!pvkStr.equals(" "))
        {
                dBuf.append(pvkStr);
        }
        else
        {
                while(privateKey.ready())
                {
                        pvkStr = Integer.toString(privateKey.read());
```

```
                        /* convert pbkStr to ASCII char */
                        iKey = new Integer(pvkStr);
                        pvkStr = String.valueOf((char)iKey.intValue());

                        NpvkBuf.append(pvkStr);
                }
        }
}
privateKey.close();

d = new BigInteger(dBuf.toString());
Npvk = new BigInteger(NpvkBuf.toString());

/* ************************ Encipher ******************************
 *      - Convert the message to message strings that fit into the
 *              enciphering equation.
 *      - Encipher the plaintext.
 *      - Digitally sign the unsigned ciphertext and append to the
 *              ciphertext buffer (aka output the text that is to be sent).
 * ****************************************************************/

/* ********************************
 * Encipher variable declarations
 * ********************************/
int nLength = 0;
double rndmNZdigit = 0;
String messageStr = new String("");
StringBuffer messageBuf = new StringBuffer();
BigInteger plainText = BigInteger.ZERO;
BigInteger unsignedCT = BigInteger.ZERO;
BigInteger cipherText = BigInteger.ZERO;
StringBuffer cipherTextBuf = new StringBuffer();

                /* ******************** Message Conversion ****************
                 * - If the length of the message buffer is critical
                 *      (defined as: length >= length of N - 3:
                 *              - Set plaintext = contents of the message
                 *                      buffer
                 *              -  Encipher the plaintext and sign it.
                 *                      Append the result to the ciphertext
                 *                      buffer (aka, output to the ciphertext
                 *                      file.
                 *              - Empty the message buffer to get another
                 *                      message string.
                 * ****************************************************/

System.out.println("\nConverting text from " + ptFileName + " to numerical
                        plaintext...");

nLength = Math.min(Npvk.toString().length(), Npbk.toString().length());

/* Read all the characters from the file into a string buffer. */
while(message.ready())
{
        /* If the M buffer is empty, append a random non-zero digit. */
        if(messageBuf.length() == 0)
        {
                rndmNZdigit = Math.ceil(9 * Math.random());
                messageBuf.append((int)rndmNZdigit);
        }

        messageStr = Integer.toString(message.read());

        /* Loop to add "0" to the front of the string while the length is
         *              less than 3 bits for proper delimitation. */
        while(messageStr.length() < ASCIICharLength)
                messageStr = "0" + messageStr;
```

```
                        messageBuf.append(messageStr);

                                /* ******************* Encipher & Sign ******************
                                 *      - If the length of the M buffer is critical (defined
                                 *              as >= 3 less than the length of N:
                                 *              - Set M equal to the contents of the M
                                 *                      buffer.
                                 *              - Encrypt M and append the result to the C
                                 *                      buffer.
                                 *              - Empty the M buffer to get another M.
                                 ******************************************************/
                        if(messageBuf.length() >= nLength - criticalNMDifference ||
                                                                !message.ready())
                        {
                                plainText = new BigInteger(messageBuf.toString());

                                unsignedCT = plainText.modPow(e, Npbk);// enciphers message
                                cipherText = unsignedCT.modPow(d, Npvk);// signs message

                                if(!message.ready())
                                        cipherTextBuf.append(cipherText.toString());
                                else
                                        cipherTextBuf.append(cipherText.toString() + " ");
                                messageBuf = new StringBuffer();
                        }
                }

                message.close();

                System.out.println("Done.");

                /* Output the enciphered and signed message to a file. */
                signedCTFile.write(cipherTextBuf.toString());

                signedCTFile.close();

                System.out.println("\n" + ptFileName + " has been enciphered with " +
                        pbkFileName + " and signed with " + pvkFileName + ".\n");

        }
        catch(Exception e)
        {
                System.err.println("Main Error");
                e.printStackTrace();
        }
    }
}
```

Appendix E - qpeN™ decipher.java Source Code

```
/**
 *  Generates a deciphered file as a text file named after the input ciphertext
 *          file. Takes as input the name of the ciphertext file in the CWD to be
 *          deciphered, the name of the public key file with which to verify
 *          (authenticate) the message, and the name of the private key file with
 *          which to decipher the message.
 *<p>
 *          The 'plaintextFile.txt.cph.txt' file is the file containing the result of
 *          deciphering the ciphertext in the input file ('plaintext.txt.cph').
 *<p>
 *          The file is created (or replaces an exisiting file by the same name) in
 *          the current working directory (CWD).
 *
 *  @author Pedro Resende Avila - Cal Poly CSc 492 Senior Project qpeN&#8482;
 *  @param  ciphertextFileName  a string representing the name of the plaintext
 *                              file to be enciphered.
 *  @param  publicKeyFileName a string representing the name of the public
 *                              key file to be used in verifying the ciphertext.
 *  @param  privateKeyFileName  a string representing the name of the private
 *                              key file to be used in deciphering the verified ciphertext.
 *  @see           {@link}generateKeys, {@link}encipher
 ********************************************************************************/

import java.io.*;
import java.lang.*;
import java.math.*;
import java.util.*;

public class decipher
{
    /* ******************************
     * Private variable declarations
     * ******************************/
    private static BigInteger ZERO = BigInteger.ZERO;
    private static BigInteger ONE = BigInteger.ONE;
    private static int ASCIICharLength = 3;
    private static boolean toAppend = false;

    public static void main(String args[])
    {
            try
            {
                    /* ****************************
                     * Main variable declarations
                     * ****************************/
                    String ctFileName = new String(args[0]);
                    FileInputStream ctFileStream = new FileInputStream(ctFileName);
                    BufferedReader cipherText = new BufferedReader(
                            new InputStreamReader(ctFileStream));

                    String pbkFileName = new String(args[1]);
                    FileInputStream pbkFileStream = new FileInputStream(pbkFileName);
                    BufferedReader publicKey = new BufferedReader(
                            new InputStreamReader(pbkFileStream));

                    String pvkFileName = new String(args[2]);
                    FileInputStream pvkFileStream = new FileInputStream(pvkFileName);
                    BufferedReader privateKey = new BufferedReader(
                            new InputStreamReader(pvkFileStream));

                    FileWriter ptTranslationFile = new FileWriter(ctFileName + ".txt",
toAppend);

                    BigInteger d = ZERO;
                    BigInteger Npvk = ZERO;
```

```
BigInteger e = ZERO;
BigInteger Npbk = ZERO;

String pvkStr = "";
String pbkStr = "";
String ctStr = new String("");
Integer iKey = new Integer(0);
Integer iCT = new Integer (0);
StringBuffer signedCTBuf = new StringBuffer();
StringBuffer dBuf = new StringBuffer();
StringBuffer NpvkBuf = new StringBuffer();
StringBuffer eBuf = new StringBuffer();
StringBuffer NpbkBuf = new StringBuffer();


/* Build public key from the file */
while(publicKey.ready())
{
        pbkStr = Integer.toString(publicKey.read());

        /* convert pbkStr to ASCII char */
        iKey = new Integer(pbkStr);
        pbkStr = String.valueOf((char)iKey.intValue());

        if(!pbkStr.equals(" "))
        {
                eBuf.append(pbkStr);
        }
        else
        {
                while(publicKey.ready())
                {
                        pbkStr = Integer.toString(publicKey.read());

                        /* convert pbkStr to ASCII char */
                        iKey = new Integer(pbkStr);
                        pbkStr = String.valueOf((char)iKey.intValue());

                        NpbkBuf.append(pbkStr);
                }
        }
}
publicKey.close();

e = new BigInteger(eBuf.toString());
Npbk = new BigInteger(NpbkBuf.toString());

/* Build private key from the file */
while(privateKey.ready())
{
        pvkStr = Integer.toString(privateKey.read());

        /* convert pbkStr to ASCII char */
        iKey = new Integer(pvkStr);
        pvkStr = String.valueOf((char)iKey.intValue());

        if(!pvkStr.equals(" "))
        {
                dBuf.append(pvkStr);
        }
        else
        {
                while(privateKey.ready())
                {
                        pvkStr = Integer.toString(privateKey.read());
```

```
                        /* convert pbkStr to ASCII char */
                        iKey = new Integer(pvkStr);
                        pvkStr = String.valueOf((char)iKey.intValue());

                        NpvkBuf.append(pvkStr);
                }
        }
}
privateKey.close();

d = new BigInteger(dBuf.toString());
Npvk = new BigInteger(NpvkBuf.toString());

/* Build cipherText from file */
while(cipherText.ready())
{
        ctStr = Integer.toString(cipherText.read());

        /* convert ciphertext to ASCII char */
        iCT = new Integer(ctStr);
        ctStr = String.valueOf((char)iCT.intValue());

        signedCTBuf.append(ctStr);
}

/* ************************* Decipher ********************************
 * - While there are characters to read from the signedCT buffer:
 *              - Read in a signedCT character from the buffer until a
 *                      space or the end of the buffer is reached.
 *      - Verifiy the digital signature and decipher the message.
 *              - Translate the deciphered plaintext.
 * ******************************************************************/

/* ******************************
 * Decipher variable declarations
 * ******************************/
int i = 0;
int j = 0;
boolean newSetFlag = true;
char signedCTChar = '0';
String signedCTStr = new String("");
BigInteger signedCT = BigInteger.ZERO;
BigInteger verifiedCT = BigInteger.ZERO;
BigInteger decipheredPT = BigInteger.ZERO;
StringBuffer decipheredPTBuf = new StringBuffer();
char decipheredPTChar = '0';
String ptTranslationStr = new String("");
Integer iT = new Integer(0);
StringBuffer ptTranslationBuf = new StringBuffer();



                /* ******************* Verification & Deciphering ********
                 *      - While there are characters in the signed
                 *              ciphertext buffer to be read, read each one
                 *              into a signedCT string.
                 *      - If the signed character is a space or there is no
                 *              more of the buffer to read,
                 *              - verify the signature of the message.
                 *              - decipher the message.
                 *              - append the deciphered message to the
                 *                      deciphered plaintext buffer to be
                 *                      translated.
                 * ****************************************************/
```

```
                System.out.println("\nDeciphering ciphertext from " + ctFileName + "...");

                while(i < signedCTBuf.length())
                {
                        signedCTChar = signedCTBuf.charAt(i);
                        i++;

                        if(signedCTChar != ' ')
                        {
                                signedCTStr += signedCTChar;
                        }

                        if(signedCTChar == ' ' || i >= signedCTBuf.length())
                        {
                                signedCT = new BigInteger(signedCTStr);
                                signedCTStr = "";

                                verifiedCT = signedCT.modPow(e, Npbk);
    // verifiy signature
                                decipheredPT = verifiedCT.modPow(d, Npvk);   // decipher
message

                                decipheredPTBuf.append(decipheredPT.toString() + " ");
                        }
                }

                System.out.println("Done.");



                        /* ******************** Translate ************************
                         *      - While there are digits in the deciphered
                         *              ciphertext buffer:
                         *              - If the newSetFlag is true, skip the first
                         *                      digit and change the flag.
                         *              - Read in three deciphered plaintext
                         *                      characters into the ptTranslationStr
                         *                      until the character read is a space
                         *                      or the end of the deciphered
                         *                      plaintext buffer has been reached. In
                         *                      this case, set the flag, increment
                         *                      the index and break.
                         *              - Cast the int values to char, and output.
                         * ******************************************************/


                System.out.println("\nTranslating the deciphered text...");

                while(j < decipheredPTBuf.length())
                {
                        if(newSetFlag == true)
                        {
                                j++;
                                newSetFlag = false;
                        }

                        while(ptTranslationStr.length() < ASCIICharLength)
                        {
                                decipheredPTChar = (char)decipheredPTBuf.charAt(j);

                                if(decipheredPTChar == ' ' || j >=
                                                        decipheredPTBuf.length())
                                {
                                        newSetFlag = true;
                                        j++;
                                        break;
```

```
                                }
                                else
                                {
                                        ptTranslationStr += decipheredPTChar;
                                        j++;
                                }
                        }

                        /* If not a new set, read the decimal ASCII stored in
                         *      ptTranslationStr and convert it to a character, appending
                         *      it to the ptTranlation buffer. */
                        if(newSetFlag == false)
                        {
                                iT = new Integer(ptTranslationStr);
                                ptTranslationBuf.append((char)iT.intValue());
                                ptTranslationStr = "";
                        }
                }

                System.out.println("Done.");

                ptTranslationFile.write(ptTranslationBuf.toString());
                ptTranslationFile.close();

                System.out.println("\n" + ctFileName + " has been verified with " +
                        pbkFileName + " and deciphered with " + pvkFileName + ".\n");

        }
        catch(Exception e)
        {
                System.err.println("Main Error");
                e.printStackTrace();
        }
    }
}
```

Appendix F - qpeN™ Glossary

Authentication - To positively verify the identity of a user, device, or other entity in a computer system, often as a prerequisite to allowing access to resources in a system.

Authorization - The process of giving someone permission to do or have something.

Cipher – Any general system for hiding the meaning of a message by replacing each letter in the original message with another letter. The cipher has built-in flexibility known as the key.

Ciphertext – The resulting series of characters after encrypting plaintext.

Code – A system for hiding the meaning of a message by replacing each word or phrase in the original message with another character or set of characters. The code has no built-in flexibility as the only key is the codebook.

Confidentiality – The property of assuring that information will be kept secret, with access limited to appropriate persons.

Coprime (Relatively prime) – Two numbers that have no common factors other than 1.

Cryptosystem - A package of all processes, formulae, and instructions for encrypting and decrypting messages.

d – A "random-enough" number generated with the property that $gcd(d, \varphi(N)) == 1$.

Data integrity - A condition existing when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed in an unauthorized manner.

Decipher – The process of converting a ciphertext message into an alternate plaintext message using a cipher.

Decode - The process of converting a codetext message into an alternate plaintext message using a code.

Decrypt – To decipher or decode.

Deterministic machine - Attribute of systems whose behavior is specified without probabilities (other than zero or one) and predictable without uncertainty once the relevant conditions are known.

e – The modular inverse of **d** with respect to $\varphi(N)$.

Encipher - The process of converting a plaintext message into an alternate ciphertext message using a cipher.

Encode - The process of converting a plaintext message into an alternate codetext message using a code.

Encrypt – To encipher or encode.

Entropy – The measure of disorder in a system.

Heisenberg Uncertainty principle - states that it is not possible to know exactly both the position *x* and the momentum *p* of an object at the same time.

Linear Congruence Theorem – If a and b are any integers and n is a positive integer, then the congruence $ax \equiv b \pmod{n}$ has a solution *x* if and only if the greatest common divisor(*a, n*) divides *b*.

Modular arithmetic - A form of arithmetic where integers are considered equal (or congruent) if they leave the same remainder when divided by a modulus.

Monoalphabetic substitution – A substitution cipher in which the cipher alphabet is fixed throughout encryption.

N – The product of pseudorandom integers **p** and **q**.

Non-repudiation - Method by which the sender of data is provided with proof of delivery and the recipient is assured of the sender's identity, so that neither can later deny having processed the data.

Number theory - A branch of mathematics that investigates the relationships and properties of numbers.

One-time pad cipher – A cipher that relies on a random key that is the same length as the message. If used correctly, it is proven mathematically unbreakable.

One-way function – A function that is easy to calculate, but difficult to reverse or undo.

p – One of two “random-enough” numbers generated as a factor of **N** in the RSA algorithm.

PGP™ (Pretty Good Privacy) – A computer encryption algorithm developed by Phil Zimmerman based on RSA.

Plaintext – The original message written by the sender and read by the receiver.

Private key – A key that is secret to the public and is used by the intended recipient to decrypt a message encrypted with the corresponding public key in a public-key cryptosystem.

Prime number - A positive integer that only has as positive integer divisors the number 1 and itself.

Public key – A key that is available to the public and is used by any sender to encrypt a message to the recipient bearing the corresponding private key in a public-key cryptosystem.

Pseudorandom numbers – Numbers that are deterministic in nature or origin.

q – One of two “random-enough” numbers generated as a factor of **N** in the RSA algorithm.

Qubits – Quantum bits. Binary digits used in quantum superposition states to perform any number of computations at the same time.

Totient function (φ (N)) – Returns the number of positive integers less than and coprime with the input.

Transposition – A system of encryption in which letters in the plaintext change their position but retain their identity.

References

1. Coutinho, S.C. Numeros Inteiros e Criptografia RSA. Rio de Janeiro: SBM, © 1997.

2. Crow, Ian. Mathematical Computing. University of Aberdeen, UK 14 December 2003. 2 April 2004. <http://www.maths.abdn.ac.uk/~igc/tch/mx3015/notes/node8.html>.

3. Deley, David W. Computer Generated Random Numbers. © 1991. 10 April 2004. <http://world.std.com/~franl/crypto/random-numbers.html>.

4. Diffie, Whitfield, Martin E. Hellman. New Directions in Cryptography. 24 September 1976. IEEE. 18 April 2004. <http://www.cs.jhu.edu/~rubin/courses/sp03/papers/diffie.hellman.pdf>.

5. Dournaee, Blake, Dan Stearns. Encryption Technologies. 20 January 2004. Cal Poly SLO. 20 April 2004. <http://www.csc.calpoly.edu/~dstearns/SeniorProjectsWWW/Dournaee/dournaee.pdf>.

6. Han, Dong Wan. Generating Strong Prime Numbers for RSA using Probabilistic Rabin-Miller Algorithm. George Mason University. 3 March 2004. <http://mason.gmu.edu/~kgaj/ECE590/spec/dong.html>.

7. Kabay, M. E. “RSA Founders Give Perspective on Cryptography” Network World Newsletter. 05.04.04. Email. May 7, 2004.

8. Kim, Chinuk, Izidor Gertner. VHDL Implementation of Systolic Modular Multiplication on RSA Cryptosystem. © January 2002. City College of the City of New York. 23 March 2004. <http://wwwcs.engr.ccny.cuny.edu/~gertner/Students/Master/Chinuk/MS_Thesis_Chinuk_Kim.PDF>.

9. Litterio, Francis. Why are One-Time Pads Perfectly Secure? 15 April 2004. <http://world.std.com/~franl/crypto/one-time-pad.html>.

10. Sedgewick, Robert, Kevin Wayne. Cryptography. © 2004. 12 April 2004. <http://www.cs.princeton.edu/introcs/104crypto/>.

11. Singh, Simon. The Code Book. New York: Anchor Books, © 1999.

12. Rivest, R.L., A. Shamir, L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Ed. S.L. Graham, R.L. Rivest. © 1978. MIT Laboratory for Computer Science. 18 April 2004. <http://cne.gmu.edu/modules/acmpkp/security/texts/PUBKEY.PDF>.

13. “RSA Services.” RSA Security® Professional Services. © 2004. 25 April 2004. <http://www.rsasecurity.com/services/flash/flash/defaultdetect.htm.>

14. “RSA Theory”. © 2002. DI Management Services. 12 March 2004. <http://www.di-mgt.com.au/rsa_theory.pdf>.

15. Wagner, Neal R. The Laws of Cryptography: The RSA Cryptosystem. © 2001. 10 April 2004. <http://www.cs.utsa.edu/~wagner/laws/RSA.html>.