Rivest-Shamir-Adleman Encryption (RSA)
CS485: Final Project

Lawrence Tyler Rush

Friday, May 8, 2009

Introduction

The RSA algorithm was developed at MIT (they hold a patent) in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman. And, as you may have guessed, RSA is an acronym for the last names of the inventors [11]. It is a deterministic reversible public-key encryption algorithm that, at its core, hinges on the fact that finding the prime factorization of large numbers (say 100 to 200 digits) is hard. More specifically, determining the integer factorization of a large number is an hard problem, and that therefore the "cracking" of the encryption is inherently hard.

Because RSA is a reversible public-key encryption algorithm, it can be used for authentication purposes. Anyone can identify herself simply by encrypting something with her secret key that anyone else can then decrypt with that same person's public key. Hence RSA is used frequently today with the internet and its need for authentication operations. A simple example is secure-sockets-layer (SSL) which uses RSA for authentication of the two users, from which point it uses symetric keys to encrypt any remaining information that is exchanged. This is due to that fact that encrypting with symmetric keys can be more efficient than RSA.

Mathematical Tools

The RSA algorithm relies on a couple of mathematical properties in Number Theory, which are described below.

Definition 1 *Two positive integers a and b are* ***relatively prime (or coprime)*** *if their greatest common divisor is equal to one.*

So saying that two numbers are coprime is also akin to saying that they have no common proper divisors. This may raise the question of just how many integers are less than and coprime to a given integer. Well Euler, like in many other cases, came up with a solution to this problem.

Definition 2 ***Euler's totient function (or Euler's phi function)*** *is the function φ defined for some non-zero natural number n and prime p as*

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

which gives the total number of integers from 1 to n that are relatively prime to n.

Thus for a prime integer p the number of integers from 1 to p that are relatively prime to p is $\varphi(p) = p(1-\frac{1}{p}) = p-1$, which is a slightly intuitive result since p has no proper divisors and that the only common divisor that p and any number less than p can have is one. Thereby making the greatest common divisor 1. Therefore all non-zero integers less than p are relatively prime to p. In the same manner we can see that for any two prime numbers p and q the following is true.

$$\varphi(pq) = pq\left(1 - \frac{1}{p}\right)\left(1 - \frac{1}{q}\right) = (p-1)(q-1) \tag{1}$$

Again this is slightly intuitive and is a result that the RSA algorithm makes use of when generating public and private keys. A more general result is that if m and n are relatively prime, then $\varphi(mn) = \varphi(m)\varphi(n)$, which is also a nice result and implies that one can break down the totient of a given number, say r, into the product of the totient of each prime number that divides r, raised to the quotient of both r and that prime number. A good exercise would be to confirm this property. (hint: What are the proper divisors of mn?)

The RSA algorithm also makes use of some simple modular arithmetic. As such, a blip about the additive and multiplicative properties of integers modulo and number n and a slightly more in-depth talk about modular inverses is necessary.

Theorem 1 ([2]) *Additivity and Multiplicity of the integers modulo some number n hold for all integers. That is to say, for two integers a and b, the following two properties are true in modulo n.*

(i). Additivity: $(a \bmod n) + (b \bmod n) = (a + b) \bmod n$

(ii). Mutiplicativity: $(a \bmod n)(b \bmod n) = ab \bmod n$

Basically, addition and multiplication as we know them, also reside in the world of modulo n. Further extensions include summations, products, and powers. Roughly the same is true for multiplicative inverses with modulo n.

Definition 3 *The* ***modular inverse*** *of a non-zero integer a modulo n is another integer x such that the following statement holds.*

$$ax \equiv 1 \bmod n$$

Furthermore, the multiplicative inverse of an integer a modulo n exists if and only if a and n are relatively prime.

This is also equivalent to saying that the remainder of dividing the product ax by n results in a remainder of 1, which implies that $n \mid ax - 1$. This in turn gives us the equation $ax - 1 = ny$ for some integer y, or equivalently we get the Diophantine equation, $ax + ny = 1$, since y is arbitrary and can easily "absorb" the negation of ny. However, how do we even know that such integers exist?

Theorem 2 (Bézout's Identity [1]) *For all non-zero integers a and b there exist integers x and y such that*

$$ax + by = \gcd(a, b)$$

and more commonly than not, either x or y is negative.

But, is this what we want? Well, remember that with x being the modular inverse of a mod n, a and n are relatively prime, which gives us what we need, namely $ax + ny = 1 = \gcd(a, n)$. However, the problem of finding these values now arises.

Being that the readers of this paper most probably posses a background in computer science or the like, a general understanding of the Euclidean Algorithm is assumed. However, an algorithm that is less known is the extended Euclidean Algorithm.

Theorem 3 *For all non-zero integers a and b, the extended Euclidean algorithm determines two integers x and y that solve the following Diophantine equation.*

$$ax + by = \gcd(a, b)$$

There are multiple ways in which the extended Euclidean algorithm can be implemented including but not limited to both iterative and recursive methods [3]. In the implementation of the RSA algorithm completed for this project (see below), the recursive method was used. Hence, that will be described. When attempting to find the greatest common divisor of two numbers a and b with arbitrarily assuming $a > b$, we must notice that (via the Euclidean algorithm) if a does not divide b, then we have that $\gcd(a, b) = gcd(b, a \bmod b)$. As a result, we also have that if a does not divide b, then for some x and y

$$ax + by = bx + (a \bmod b)y \tag{2}$$

which begins to look like something we could use, and indeed, such is the case, but how so? If we fiddle around with the right-hand side of equation 2 we can obtain the following sequence of equations.

$$\begin{aligned} bx + (a \bmod b)y &= bx + \left(a - \left\lfloor \frac{a}{b} \right\rfloor b\right) y \\ &= bx + ay - \left\lfloor \frac{a}{b} \right\rfloor by \\ &= ay + b\left(x - \left\lfloor \frac{a}{b} \right\rfloor y\right) \end{aligned}$$

This gives to us the recursive step for our algorithm that we desire, since we can solve for the solution to the equation $ax + by = \gcd(a, b)$ by solving for a solution to the equation $bx + (a \bmod b)y = \gcd(b, (a \bmod b))$, thereby decreasing the starting values. More specifically, the solution to $ax + by = \gcd(a, b)$ is

$$x = y', y = x' - y'\left(\left\lfloor \frac{a}{b} \right\rfloor\right)$$

where x' and y' is the solution to $bx' + (a \bmod b)y' = \gcd(b, a \bmod b)$. We will stop in the trivial case (our base case) when b divides a (i.e. b is the greatest common divisor of a and b), which yields $a(0) + b(1) = \gcd(a, b)$. Hence our recursive algorithm is as follows.

Algorithm: Extended Euclidean

Input: Two non-zero integers a and b
Output: A tuple of integers, (x, y) such that $ax + by = \gcd(a, b)$ is satisfied

if $b \mid a$
    return (0,1)
else
    $(x, y) = \text{extendedEuclidean}(b, a \bmod b)$
    return $(y, x - y\left(\left\lfloor \frac{a}{b} \right\rfloor\right))$

There is one last item that must be mentioned before the mathematical background for the RSA algorithm is complete, and that is the Chinese Remainder Theorem.

Theorem 4 (Chinese Remainder Theorem [6, 15]) *For any set of non-zero positive integers $z_1, z_2, \ldots, z_k$ that are pair-wise relatively prime, there exists an x for all sets of integers $a_1, a_2, \ldots, a_k$ such that the following holds.*

$$\begin{aligned} x &\equiv a_1 \bmod z_1 \\ x &\equiv a_2 \bmod z_2 \\ &\vdots \\ x &\equiv a_k \bmod z_k \end{aligned}$$

Futhermore, for $b_1, b_2, \ldots, b_k$ such that

$$b_i \frac{Z}{z_i} \equiv 1 \bmod z_i$$

where $Z = z_1 z_2 \cdots z_k$ then the following is true.

$$x \equiv (a_1 b_1 \frac{Z}{z_1} + a_2 b_2 \frac{Z}{z_2} + \cdots + a_k b_k \frac{Z}{z_k}) \bmod Z$$

Note that while the RSA algorithm does not use the Chinese Remainder Theorem directly, knowledge of this fact can give one the ability to launch an attack on a system using the RSA algorithm as we will see in the discussion on RSA vulnerabilities.

The RSA Algorithm

The RSA algorithm consists of three main parts which are (1) generating the public and private keys, (2) encryption of plain-text into ciphertext, and decryption of ciphertext into plain-text. Remember that it is because of the the difficulty of factoring large integers which provides RSA encryption with its "strength".

Key Generation

The key generation scheme results in the creation of both the public and private keys, of which both are in the form of a tuple of integers,

$$(n, e)$$

in which n is called the modulus and e is called the exponent. The method to generate the two keys is as follows.

1. Choose two prime numbers p and q with $p \neq q$.
2. Compute the value n such that $n = pq$.
3. Compute the totient of n. That is, compute $\varphi(n)$, the number of positive integers less than and relatively prime to n.
4. Choose a non-zero natural number e that is less than n and relatively prime to $\varphi(n)$.

5. Compute a number d such that $ed \equiv 1 \bmod n$, that is, compute the modular inverse of e modulo $\varphi(n)$.

The RSA key generation method produces, from p and q, three new values, which are n, e, and d. These three new values are what make up the public and private keys of the RSA encryption scheme. The public key is the tuple (n, e) and the private key is the tuple (n, d). Keep in mind that not only does the value of d have to be kept secret, but also the values of p and q since knowing p and q along with knowing the public key provides one with the ability to compute the private key. These keys each can be used for decrypting a ciphertext that the opposite key has encrypted, hence the ability of RSA to be used as a signing mechanism.

For some of the steps in the generation of the RSA public and private keys there a few things that one needs to keep in mind when making choices during the process.

Choosing Prime Numbers: The prime numbers p and q should be choosen based on the fact that the larger the two numbers the more secure the algorithm/encryption is, but choosing large prime numbers comes with the price of less efficient encryption and decryption. Obviously, there is a delicate balance that needs to be found between the need for security at the cost for efficiency. See the later section on RSA vulnerabilities that mentions additional caveats about choosing p and q.

Computing the Totient of n: When computing the totient of n, rather than using the formula described in the definition of the totient function (Definition 2), the known property expressed in Equation 1 should be put to use, since we know that $n = pq$. Doing so will be more efficient since a general function that computes the totient of a number will most likely attempt to find all of the prime factors of n, which is more complex than necessary for our needs during the key generation process. When p and q are large, computing the totient of n with a general totient function is essentially trying to break the encryption of RSA.

Choosing e: It is common practice to choose a value for e that is prime, which is always garenteed to be relatively prime to n. Even more so, it is common to choose 3, 17, or 65537 to be the value of e. See the section on RSA vulnerabilities for possible weakening of security by choosing a value of e that is small.

Encryption and Decryption

The encryption and decryption methods of the RSA algorithm are functionally identical. The difference is that encryption takes plain-text as input and outputs the corresponding ciphertext, whereas decryption takes ciphertext as input and outputs the corresponding plain-text. Figure 1 depicts the way in which the encryption and decryption methods are virtually the same.

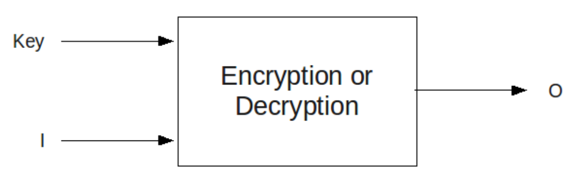

Figure 1: A depiction of the dual nature of the encryption and decryption of the RSA algorithm. The I represents the input plain-text or ciphertext and the O represents the corresponding output ciphertext or plain-text, depending on whether encryption or decryption is being performed.

In the above firgure, it is purposely not specified which key should be used for encryption or decryption. This is, as mentioned briefly earlier, because both the public and private keys can be used to encrypt plain-text into ciphertext and to decrypt ciphertext into text. This is why RSA can be used for signing and authentication. For instance, if Alice wants to authenticate herself to Bob, she simply encrypts some text or phrase using her secret key and sends the resulting ciphertext to Bob who can decrypt the ciphertext into the plain-text using Alice's public key. This authentication method is, of course, contingent upon the belief that Alice's private key has not been compromised. Nevertheless, disregarding any (hopefully small) chance that private keys are compromised, Bob and Alice can perform the analogous task of verifying Bob to Alice, thereby completing the authentication of each other to one another.

The Nitty-Gritty: Moving away from abstraction and towards the details, the encryption and decryption methods use simple modular arithmetic for computing their respective values from their respective inputs. However, the idea of a padding scheme must first be mentioned.

Basically, a padding scheme is a method for taking a string and creating a sequence of integers (or bits) less than a certain value [11]. RSA encryption/decryption demands a padding scheme that takes a message and transforms it such that the sequence of integers produced are non-zero and less than $n = pq$ (or equivalently transforms it into an number of bits that are less than the size of n in bits). Furthermore, it demands that this transformation be known to both the sender and receiver of the message/ciphertext and also that the transformation is reversible, by which we mean a bijection. This is, of course, so that the reciever can "unpad" the number resulting from the decryption of the ciphertext. Note that, from here on out, we will speak of the text and ciphertext without mention of padding with the understanding that there is always some sort of inherent padding scheme. The aim here is for the clarity of the reader.

As was mentioned earlier, the encryption and decryption methods use simple modular arithmetic. Lets say that Alice would like to send a message m to Bob. Using Bob's public key, (n, e), Alice performs the encryption of m by creating the ciphertext c via the following calculation.

$$c = m^e \bmod n \tag{3}$$

Now that Alice made the ciphertext, she ships it off to Bob for decryption and reading. When Bob recieves the ciphertext from Alice he decrypts c into m via peforming the following computation using his private key (n, d).

$$m = c^d \bmod n \tag{4}$$

But Why? What is the reason that this encryption and decryption process returns the correct result? By making use of Theorem 1 and a result from number theory that states that if n is prime or a product of two distinct primes, then we have that for all integers x, $x^y \bmod n = x^{y \bmod \varphi(n)} \bmod n$ where φ is Euler's totient function [6], we can obtain the the reason why RSA encryption works. Assuming Equations 3 and 4 hold, we have the following sequence of equations.

$$\begin{aligned} m &= c^d \bmod n \\ &= (m^e \bmod n)^d \\ &= m^{ed} \bmod n \\ &= m^{ed \bmod \varphi(n)} \bmod n \\ &= m \bmod n \\ &= m \end{aligned}$$

Notice that the simplification from line 4 to line 5 above is posible because we chose e and d such that $ed \equiv 1 \bmod \varphi(n)$ which means that the remainder when dividing ed by $\varphi(n)$ is 1, that is, $ed \bmod \varphi(n) = 1$. The simplification from line 5 to line 6 is a result of the demand that m is less than n. Without this property RSA would not work, which is why we needed to demand that encyption only use integers that are less than n.

Note that any uses of p, q, n, e, m, and c beyond this point are taken to be defined as they are in the descriptions above of the key generation, encryption, and decryption methods of the RSA algorithm.

Vulnerabilities of the RSA Algorithm

While RSA does seem like a rather strong method for encryption and decryption, it unfortunately has some weaknesses. Most vulnerabilies have easy fixes, but some could prove to be a problem in the future (i.e. the abilities of Quantum Computers... ooooh, aaaah).

Note that here we discuss the problems strictly inherent to RSA encryption and leave out the vulnerabilities that, while should be accounted for when using RSA encryption and are of some conern, are, however, generic by nature to all public-private key methods (i.e. man-in-the-middle attacks, private keys being compromised,...).

Choosing p and q Wisely

Two people using identical primes for n: It at first doesn't seem all that bad as long as they are not both the same, however if there is someone with keys generated using p and q (call this person the attacker) and another person that generated keys using at least one of either p or q, then the attacker has a method for factoring the other person's n (which of course is publicly known). Hence if the attacker begins finding remainders of division by p and q of n in public key after public key, stumbling across someone who used the same p or q for her value of n will result in a remainder of zero. This gives the attacker the prime factorization of n in that person's RSA encryption system.

So how can this be avoided? Well, in all practicality, it does not need to be avoided as much as it simply has a low probability of happening. Sure it can happen in theory, but it most likely will not. This is because of the prime number theorem, conjectured in 1791 by Gauss and proved by Hadamard and de la Vallée Poussin in 1896 [13, p. 124].

Theorem 5 (Prime Number Theorem [13]) *Given a large enough N, the function π defined as*

$$\pi(N) \approx \frac{N}{\ln N}$$

returns the number of prime numbers less than N.

Because $\pi(N) \approx \frac{N}{\ln N}$ is the number of primes less than N and we have that

$$\lim_{N \to \infty} \frac{N}{\ln N} = \infty$$

then we can see that for significantly large N there are plenty of primes from which to choose. Furthermore, because of the Prime Number Theorem, we have that the probability of two people choosing the same prime number in the range of all primes from 2 to N is $\frac{\ln N}{N}$. This probability for, say an eleven digit decimal number (i.e. at least 10^{10}) is

$$\frac{\ln 10^{10}}{10^{10}} \approx 2.30 \times 10^{-9}$$

which when knowing that current RSA techniques use on the order of 300 digit numbers renders the probability of two people choosing the same prime to be virtually unimaginable [16, p. 176].

Choosing p and q Too Close Together [4]: The Fermat Method is a method of factorization based on the fact that an odd integer can be represented as the difference of squares of two integers. Moreover, if the number n to be factored by the Fermat Method is such that $n = cd$ then n can be factored quickly if either c or d is within $\sqrt[4]{4n}$ of $\sqrt{n}$, actually within one step. Thus the primes p and q should be chosen such that their difference is not too small. Namely, if either p or q is within $\sqrt[4]{4n}$ of $\sqrt{n}$ then the Fermat Factorization Method has the ability to break the RSA encryption in one step of its algorithm.

Choosing a Small Value for e

Smaller Roots for m: Because of the method of encryption that RSA employs (Equation 3) if m is less than $\sqrt[e]{n}$ then $m^e < n$ which implies that $m^e \bmod n = m^e$. Hence, an attacker can take the e^{th} root of the encrypted message (seeing as e is part of the public key) and will automatically have the original message. Taking the e^{th} root of a number for small values of e is trivial.

A Single Message Encrypted e Times [6]: Another problem that arises when using a small value of e would be the problem of the situation that occurs when a single message m is encrypted by at least e people, say with public keys $(n_1, e), (n_2, e), \ldots, (n_e, e)$ where $n_1, n_2, \ldots, n_e$ are pair-wise relatively prime, is more likely to occur. Attackers can then derive, via the Chinese Remainder Theorem (Theorem 4), the value of $m^e \bmod n_1 n_2 \cdots n_e$ as demanded by the RSA key generation method. Also, since an attacker would know that $m < n_i$ for all $i \in \{1, \ldots, e\}$, then she also knows that $m^3 < n_1 n_2 \cdots n_e$ which means simply that $m^e \bmod n_1 n_2 \cdots n_e = m^e$. Therefore finding the message m can be accomplished by taking the e^{th} root. Hence, for small values of e finding m is simple.

Choosing an e large enough (but not so large as to decrease encryption/decryption efficiency), can side-step the above problems with small values of e. It is suggested that a value of e of 65537 should be used due to the striking of a nice balance between the magnitude of e and the efficiency of the encryption and decryption process using such an e [6].

Deterministic Nature of RSA

Because the RSA algorithm is deterministic, attackers can use what is called a plain-text attack to try to determine the private key of someone. A plain-text attack can be performed by encrypting likely plain texts, "Hello World" for instance, with multiple different keys and comparing them to ciphertexts that are obtained one way or another. If the two ciphertexts turn out to be the same, the attacker then knows the key that was used to encrypt the text since the attacker knows the original text. To safeguard against this weakness, the employment of a good padding scheme is necessary, preferably one that utilizes the power of randomness in order to counter-act the deterministic nature of the RSA encryption.

Authentication Backfire: A Sneaky Attacker uses Modular Mutiplicativity [16]

Given the multiplicative property of modular arithmetic (Theorem 1) two messages, m_1 and m_2, upon encryption have the property that

$$(m_1^e \bmod n)(m_2^e \bmod n) = (m_1 m_2)^e \bmod n$$

meaning that the product of two messages encrypted with the same key is identical to the encryption of the product of those two messages. Hence if an attacker Eve has a ciphertext encrypted via Alice's public key (n, e), $c = m^e \bmod n$, where m is say a session key that someone is sending to Alice for a symmetric encryption scheme, then Eve can now choose an r that is less than and relatively prime to n with which she can produce a new innocent-looking message $y = (c)(r^e \bmod n) = cr^e \bmod n$ using Alice's public key (n, e). Eve then asks Alice to verify herself by decrypting y. Alice uses herown private key, (n, d) to do so and sends the result back to Eve. Eve knows how the decrypting process works and thereby knows that she is recieving from Alice $y^d = c^d r^{ed} \bmod n$. Because Eve knows the following two facts,

$$r^{ed} = r \bmod n \qquad \text{and} \qquad m = c^d \bmod n$$

given by the facts that r and n are relatively prime, $ed \equiv 1 \bmod \varphi(n)$, and Equation 3, then Eve also knows $y^d = c^d r^{ed} \bmod n = mr \bmod n$. This information gives to Eve the knowledge of the remainder of dividing mr by n, from which she can use the Extended Euclidean Algorithm to compute m.

Quantum Power

Remembering that the strength of the security of the RSA encryption relies on the inability of *some* computers to factor large integers. It is the current belief of the majority of the computer science community that while Integer Factorization is in NP, it is not in P. This of course also implies the underlying belief that NP≠P. However, in 1994 Peter Shor discovered a way of using quantum mathematics and more specifically the Quantum Fourier Transform to solve the integer factorization problem (and, as point of interest, Shor was also able to solve the "discrete logarithm" around the same time) [9, p. 6]. Figure 2 depicts the location of integer factorization in the current system of beliefs about the arrangement of complexity classes, where Bounded error, Quantum Polynomial time (BQP) is the set of problems solvable in polynomial time via the use of quantum computers.

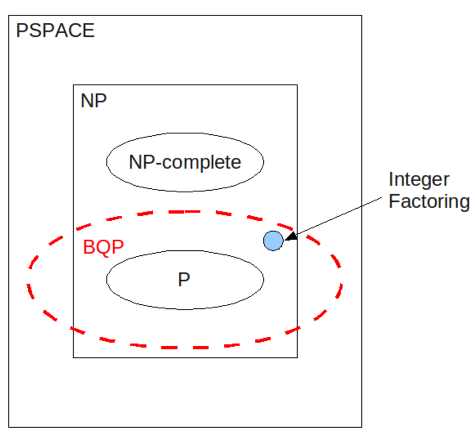

Figure 2: The leading belief of the complexity classes in PSPACE. More specifically, the belief of the location of integer factorization as being in NP and in BQP, but not in P.

This makes for great concern about the use of the RSA algorithm, since quantum computers will be able to crack an RSA scheme in a polynomial amount of time. This is even more so since, in 2001, IBM researchers were able to build a quantum computer and put Shor's algorithm to use. However, you may be relieved to know that the maximum number that they were able to factor was 15. No, NOT a 15 digit number, but 15, 3(5). Hence we still have *at least* a few years of quality research to go before quantum computers threaten the livelihood of our modern security infrastructure that is RSA.

Implementation

The **`Main`** module here is a shell for accessing the modules used for RSA key generation, encryption and decryption. There are different compilers/interpreters for Haskell. I prefer the Glasgow Haskell Compiler (GHC), and will discuss the building of the program from the usage of such a compiler. GHC is freely available online (www.haskell.org) for Windows, Linux, Mac (all three of which are fully supported) and also Solaris (which is community supported). Haskell was the language chosen for implementation of the RSA algorithm because of the arbitrary-precision `Integer` type which has no problems with handling extremely large numbers.

The building of the `Main` module (i.e. Crypt.lhs) requires the following modules:

- Encryption.lhs
- FractionalInteger.lhs
- ModularArithmetic.lhs
- Prime.lhs
- RSA.lhs

Once Crypt.lhs and all of the above files are in the same directory (or in another if the -i option for ghc is specified), then creation of the executable can be completed by executing the following command:

```
ghc --make Crypt -o desired_executable_name
```

which will produce a binary file with the name "desired_executable_name".

In order to run the program simply make the following call.

```
desired_executable_name -rsa
```

```
-- FILE: Crypt.lhs
-- DATE CREATED: May 6, 2009
-- LAST UPDATED: May 11, 2009
-- AUTHOR: Lawrence Tyler Rush
--         me@tylerlogic.com
--         www.tylerlogic.com
-- VERSION: 1.0.0
module Main ( main ) where
```

Imports of the Main Module

The `Data.Char` module is imported to use its `digitToInt()` function to perform the transformation from the string representation of an integer to its corresponding integer value that occurs in the `stringToInteger()` function of this module.

```
import Data.Char ( digitToInt )
```

The `RSA` module is imported for the obvious reasons of being able to generate keys, encryption of text, and decryption of ciphertext all by using the RSA algorithm.

```
import RSA ( rsaDecrypt , rsaEncrypt , rsaKeyGenerator )
```

The `System` module is imported so that the command-line arguments for the program can be obtained through use of the module's `getArgs()` function.

```
import System ( getArgs )
```

Exported Functions of the Main Module

The **main()** function simply asks the user for two prime numbers from which RSA public and private keys will be generated. The user will then be asked for a message to encrypt, and encryption will take place along with the subsequent decryption back into the original text. In order to not take too much time (due to current implementation inefficiencies of RSA module 1.0.0) prime numbers below about 500 may be a good idea to test and not have to wait forever, like 487 and 397. Order of entry of the p and q matters not. Also note that as of version 1.0.0 of the RSA module, the padding scheme to convert from a string to an integer is to take the integer ASCII value of each character and perform the RSA encryption on each such integer value. Thus p and q should be chosen (for RSA module versions less than 1.0.0) such that n is greater than the highest ASCII integer value of 127. Likewise p and q should be choosen such that $n > 65537$ since the RSA module of version 1.0.0 or earlier employs a value of $e = 65537$. Playing around with the program, as in entering composite numbers for p and q obtains wacky results. Trying for example using 350 and 400 to encrypt

```
This is a trial of not using primes
```

results in the jibberish that follows.

```
�@[0011]i�[0011]i�A� �[0011]A`�[0001]��@[0001] ��i[0011]@ω �[0011]I9i
```

Hence choose prime numbers. A similar situation occurs when selecting some values of p and q such that $n < 65537$.

```
main :: IO ()
main = do
    args <- getArgs
    parseArgs args
    -- Get the two prime numbers
    putStrLn "Enter the first prime number."
    pString <- getLine
    putStrLn "Enter the second prime number."
    qString <- getLine
    -- Convert the strings to actual numbers and compute the RSA keys
    let p = stringToInteger pString
    let q = stringToInteger qString
    let keys = rsaKeyGenerator p q
    putStrLn $ "Your public key: " ++ (show.fst) keys ++ "\n" ++
               "Your private key: " ++ (show.snd) keys ++ "\n" ++
               "Press enter to continue..."
    nothing <- getLine
    putStrLn "\nEnter the phrase that you would like to encrypt."
    plainText <- getLine
    putStrLn $ "\nThe following phrase will be encrypted: " ++
               "\"" ++ plainText ++ "\"" ++ "\n" ++
               "Press enter to continue..."
    nothing <- getLine
    -- Encrypt the plain text and show it
    putStrLn "Encryption Ciphertext: "
    let encryption = rsaEncrypt (fst keys) plainText
    putStrLn $ show encryption ++ "\n" ++
               "Press enter key to decrypt..."
    nothing <- getLine
    -- Decrypt the ciphertext and show it
    putStrLn "Decrypting..."
    let decryption = rsaDecrypt (snd keys) encryption
    putStrLn (decryption ++ "\n")
```

Non-Exported Functions of the Main Module

The **rsaOpt()** function simply provides the string that is used as the option in the command-line arguments to indicate that RSA encryption is to be used.

```
rsaOpt :: String
rsaOpt = "-rsa"
```

The **parseArgs()** function takes in the command-line arguments and determines whether or not there are any errors, causing error messages to be displayed if the command-line syntax is incorrect.

```
parseArgs :: [String] -> IO()
parseArgs args
    | length args /= 1 = error "ERROR: Incorrect number of arguments."
    | (args !! 0) == rsaOpt = putStr ""
    | otherwise = error "ERROR: Unrecognized arguments."
```

The **stringToInteger()** function takes a string representation of an integer (i.e. from a command line text) and converts it to the corresponding value in the Haskell `Integer` type.

```
stringToInteger :: String -> Integer
stringToInteger str = thisRecurse str 0
    where thisRecurse str num
              | str == [] = num
              | otherwise = thisRecurse t (nextNum*(10^(length str - 1)) + num)
                where nextNum = (toInteger.digitToInt.head) str
                      t = tail str
```

MODULE: ModularArithmetic

The **ModularArithmetic** module provides functionality for common modular arithmetic, which, at this point, minimally contains the `divides()` function.

```
-- FILE: ModularArithmetic.lhs
-- DATE CREATED: May 4, 2009
-- LAST UPDATED: May 8, 2009
-- AUTHOR: Lawrence Tyler Rush
--         me@tylerlogic.com
--         www.tylerlogic.com
-- VERSION: 1.0.0
module ModularArithmetic ( divides ) where
```

Exported Functions of the ModularArithmetic Module

The **divides()** function indicates whether a specified integer divides another via testing if the modulo is equal to zero.

```
divides :: Integer -> Integer -> Bool
divides n = ( (== 0) . (mod n) )
```

MODULE: FractionalInteger

The **FractionalInteger** module provides the ability to represent a rational number using arbitrary precision through the use of the Haskell `Integer` type.

```
-- FILE: FractionalInteger.lhs
-- DATE CREATED: May 4, 2009
-- LAST UPDATED: May 8, 2009
-- AUTHOR: Lawrence Tyler Rush
```

```
--         me@tylerlogic.com
--         www.tylerlogic.com
-- VERSION: 1.0.0
module FractionalInteger
( FractionalInteger,
  denom,
  nume,
  toFractionalInteger,
  toIntegerFromFractionalInteger )
where
```

Imports of the FractionalInteger Module

This module uses the `divides()` function of the `ModularArithmetic` module in determining the equivalence of two `FractionalInteger`s.

```
import ModularArithmetic ( divides )
```

This module needs the `Ratio` module in order to instantiate the `Rational` class for the `FractionalInteger` abstract data type and functions that correlate to the `Rational` class.

```
import Ratio ( Rational , denominator , numerator )
```

Abstract Data Types of the FractionalInteger Module

This module implements a fraction version of the `Integer` type via the **`FractionalInteger`** abstract data type which represents the fraction version of a `Integer` via a tuple of two `Integer` types. Notice that this representation is the reason that a `FractionalInteger` can only represent rational numbers.

```
data FractionalInteger = FI (Integer,Integer)
```

Class Instantiations of the FractionalInteger Module

The `Eq` class instantiation determines the equality of two `FractionalInteger`s via determining the equivalence of the numerator and denominator of each of the two `FractionalIntegers`.

```
instance Eq FractionalInteger where
    ( FI (a,b) ) == ( FI (c,d) )
        | divides b d = a == (quot b d)*c
        | divides d b = c == (quot d b)*a
        | otherwise   = False
```

This module instantiates the `Fractional` class by defining the division, reciprical, and fromRational functions in the obvious way.

```
instance Fractional FractionalInteger where
    (/) ( FI (a,b) ) ( FI (c,d) ) = simplify ( FI (a*d,b*c) )
    recip ( FI (a,b) )            = simplify ( FI (b,a) )
    fromRational rat              = FI (numerator rat,denominator rat)
```

This module instantiates the `Num` class by defining the addition, subtraction, multiplication, negation, fromInteger, abs, and signum functions in the obvious way.

```
instance Num FractionalInteger where
    (+) ( FI (a,b) ) ( FI (c,d) ) = simplify ( FI (a*d+b*c,b*d) )
    (*) ( FI (a,b) ) ( FI (c,d) ) = simplify ( FI (a*c,b*d) )
    negate ( FI (a,b) )           = FI (-a,b)
    (-) one two                   = (+) one (negate two)
    fromInteger a                 = FI (a,1)
    abs ( FI (a,b) )              = FI (abs a,abs b)
```

```
    signum ( FI (a,b) )
        | b == 0 = error "ERROR: Division by zero!!"
        | a == 0 = 0
        | (a < 0 && b < 0) || (a > 0 && b > 0) = 1
        | (a < 0 && b > 0) || (a > 0 && b < 0) = -1
```

This module instantiates the `Show` class by using the `show` function on the numerator and denominator of the `FractionalInteger`.

```
instance Show FractionalInteger where
    show ( FI (q,r) ) = (show q) ++ " / " ++ (show r)
```

Exported Functions of the FractionalInteger Module

The **`denom()`** function gets the denominator of the `FractionalInteger`.

```
denom :: FractionalInteger -> Integer
denom ( FI (q,r) ) = r
```

The **`nume()`** function gets the numerator of the `FractionalInteger`.

```
nume :: FractionalInteger -> Integer
nume ( FI (q,r) ) = q
```

The **`toFractionalInteger()`** function converts two integers (one for each of the numerator and denominator) to a `FractionalInteger`. This function causes an error if a `FractionalInteger` is attempted to be created using a 0 as the denominator, which of course is undefined in mathematics.

```
toFractionalInteger :: Integer -> Integer -> FractionalInteger
toFractionalInteger q r
    | r == 0    = error err_TO_FRACTIONAL_INTEGER_DIVISION_BY_ZERO
    | otherwise = simplify ( FI (q,r) )
err_TO_FRACTIONAL_INTEGER_DIVISION_BY_ZERO = "ERROR: (FractionalInteger." ++
                                             "toFractionalInteger) Cannot create a " ++
                                             "FractionalInteger where a denominator " ++
                                             "of zero. Division by zero is undefined."
```

The **`toIntegerFromFractionalInteger()`** function converts a `FractionalInteger` to a normal `Integer`, but only, as expected, if the denominator of the fraction is one. This function causes an error when the denominator of the `FractionalInteger` is not equal to 1.

```
toIntegerFromFractionalInteger :: FractionalInteger -> Integer
toIntegerFromFractionalInteger ( FI (q,r) )
     | denom simple == 1 = nume simple
     | otherwise         = error err_TO_INTEGER_FROM_FRACTIONAL_INTEGER
       where simple = simplify ( FI (q,r) )
err_TO_INTEGER_FROM_FRACTIONAL_INTEGER = "ERROR: (FractionalInteger." ++
                                         "toIntegerFromFractionalInteger) The " ++
                                         "denominator of a FractionalInteger must " ++
                                         "be 1 in order to convert to Integer."
```

Non-Exported Functions of the FractionalInteger Module

The **`simplify()`** function performs the normal operation of simplifying the `FractionalInteger` just as happens with any normal fraction. Note that no functions outside of this module should need this function since all currently exported functionality incorporates the simplification when needed, and any future modifications *should* do the same.

```
simplify :: FractionalInteger -> FractionalInteger
simplify ( FI (a,b) )
    | gcdAB /= 1 = simplify ( FI (quot a gcdAB,quot b gcdAB) )
    | otherwise  = FI (a,b)
      where gcdAB = gcd a b
```

MODULE: Prime

The **Prime** module houses multiple functions that have to do with primes in some way. Some of the more common functions are functions that determine all the divisors of a number, the proper divisors of a number, and the prime factorization of a number. Other functions include Euler's totient function and a function that produces a list of all primes via a method called "Prime Wheels"

```
-- FILE: Prime.lhs
-- DATE CREATED: April 29, 2009
-- LAST UPDATED: May 8, 2009
-- AUTHOR: Lawrence Tyler Rush
--         me@tylerlogic.com
--         www.tylerlogic.com
-- VERSION: 1.0.0
module Prime
( divisors,
  prime,
  primeFactorization,
  primeFactorizationToInteger,
  primeFactors,
  primes,
  properDivisors,
  totient )
where
```

Imports of the Prime Module

This module imports the `FractionalInteger` module in order to perform division of the Haskell `Integer` type. This functionality is used mainly in the totient function.

```
import FractionalInteger ( toIntegerFromFractionalInteger )
```

Some of the functions in this module need to know whether or not a particular number is a divisor of another. Hence the importing of the `ModularArithmetic` module.

```
import ModularArithmetic ( divides )
```

Exported Functions of the Prime Module

The **divisors()** function determines every divisor of a specified number returning the divisors in a list in increasing order.

```
divisors :: Integer -> [Integer]
divisors n = 1:(properDivisors n) ++ [n]
```

The **prime()** function indicates whether or not a specified integer is prime. This is done simply by making sure that the given integer has no proper divisors.

```
prime :: Integer -> Bool'
prime n'
    | n <= 1 = False
    | n == 2 = True
    | even n = False'
    | otherwise = (length.properDivisors) n == 0
```

The **primeFactorization()** function determines the prime factorization of a given integer, returning the prime factorization as a list of tuples in which the first element of each tuple is a prime divisor of the integer, and the second element is the quotient resulting from the division of the given integer by the prime divisor. That is, each tuple (p, k) is such that p is a prime number and k is the largest integer such that p^k divides the integer passed to this function.

```
primeFactorization :: Integer -> [(Integer,Integer)]
primeFactorization n = thisRecurse n primes []
    where thisRecurse n primesRemain currList
              | n == 1      = reverse currList
              | divides n h = thisRecurse (quot n subProduct) t (newFactor:currList)
              | otherwise   = thisRecurse n t currList
                where h = head primesRemain
                      t = tail primesRemain
                      subProduct = h^(snd newFactor)
                      newFactor = (h,numberOfDivisions h n)
```

The **primeFactorizationToInteger()** function computes the product of a prime factorization, resulting in the unique intger represented by the specified prime factorization.

```
primeFactorizationToInteger :: [(Integer,Integer)] -> Integer
primeFactorizationToInteger [] = 1
primeFactorizationToInteger (x:xs) = ((fst x)^(snd x))*(primeFactorizationToInteger xs)
```

The **primeFactors()** function determines all of the prime numbers that divide a specified integer, returning these numbers in a list.

```
primeFactors :: Integer -> [Integer]
primeFactors n = thisRecurse n primes []
    where thisRecurse n primesRemain currList
              | n == 1      = reverse currList
              | divides n h = thisRecurse (quot n subProduct) t (h:currList)
              | otherwise   = thisRecurse n t currList
                where h = head primesRemain
                      t = tail primesRemain
                      subProduct = h^(numberOfDivisions h n)
```

The **primes()** function [10] creates a list of every prime number, through the use of Haskell's functionality for infinite lists of course. Other algorithms such as only listing odds after two as possible candidates for primes work, but this is still much slower that the one implemented here since this implementation looks at fewer possible candidates for actual primes. The algorithm, called "Prime Wheels" by some, uses the fact that every prime number p has either the form

$$6k + 1 \quad \text{or} \quad 6k + 5$$

for all non-zero natural numbers k. This fact makes for quicker formation of all primes simply by only considering numbers of this form.

```
primes :: [Integer]
primes = 2:3:primes'
    where
        1:p:candidates = [ 6*k+r | k <- [0..], r <- [1,5] ]
        primes'        = p : filter isPrime candidates
        isPrime n      = all (not . divides n) (takeWhile (\p -> p*p <= n) primes')'
```

The **properDivisors()** function determines every proper divisor of a specified number, returning said divisors in a list.

```
properDivisors :: Integer -> [Integer]
properDivisors n = thisWithCutoff n 2
    where thisWithCutoff n start
              | start*start > n = []
              | divides n start = (start:(thisWithCutoff n start')) ++ [quotient]
              | otherwise       = thisWithCutoff n start'
                where quotient = quot n start
                      start' = start + 1
```

The **`totient()`** function computes the value of the Euler's Totient (phi) function for any integer $n > 1$, also known as $\varphi(n)$. This number directly returns the number of intgers that are relatively prime to n.

```
totient :: Integer -> Integer
totient n = toIntegerFromFractionalInteger ((fromInteger n) * theProduct)
    where one = fromInteger 1
          theProduct = product [ one - (one / (fromInteger p)) | p <- (primeFactors n)]
```

Non-Exported Functions of the Prime Module

The **`numberOfDivisions()`** function, for inputs b and x, determines the largest integer n such that $b^n < x$. That is, the number of times b divides x. It may be asked why is the function not simply taking the floor of the logarithm of x with the appropriate base? Such a computation would be ideal, however the function `log()` in Haskell requires a floating point as input, which makes it hard to somehow use such a function when the input should be an type that has arbitrary precision like that of the `Integer` type in Haskell.

```
numberOfDivisions :: Integer -> Integer -> Integer
numberOfDivisions base x
    | divides x base = 1 + numberOfDivisions base (quot x base)
    | otherwise      = 0
```

MODULE: Encryption

The **`Encryption`** module provides functions that are used by some encryption algorithms, in particular RSA. Currently, only functionality for the RSA algorithm is implemented, but future modifications may include providing functionality needed by multiple other encryption algorithms. It provides functions for determining the modular inverse of an integer modulo some other integer via the solving for x and y in the Diophantine Equation $ax + by = \gcd(a, b)$. Such a function is the extended Euclidean algorithm.

```
-- FILE: Encryption.lhs
-- DATE CREATED: May 4, 2009
-- LAST UPDATED: May 8, 2009
-- AUTHOR: Lawrence Tyler Rush
--         me@tylerlogic.com
--         www.tylerlogic.com
-- VERSION: 1.0.0
module Encryption
( extendedGcd,
  extendedGcdA,
  extendedGcdB )
where
```

Imports of the Encryption Module

This module imports the `FractionalInteger` in order to perform certain division of Haskell's `Integer` type. This can be done since it is known that all of the operations deal with rational numbers.

```
import FractionalInteger ( toIntegerFromFractionalInteger )
```

This module imports the `ModularArithmetic` simply to use its `divide` function in the extended Euclidean algorithm.

```
import ModularArithmetic ( divides )
```

Exported Functions of the Encryption Module

The **`extendedGcd()`** function [3] computes a solution to the equation

$$ax + by = \gcd(a, b)$$

given non-zero intgers a and b. This is done through the use of the extended Euclidean algorithm using the recursive method.

```
extendedGcd :: Integer -> Integer -> (Integer,Integer)
extendedGcd a b
    | divides high low = (0,1)
    | otherwise        = (y',x' - y'*(quot high low))
      where high = a
            low  = b
            (x',y') = extendedGcd low (mod high low)
```

The **extendedGcdA()** function [2, p. 278] computes all of the solutions x', y', given an a and b, for the equation

$$ax + by = \gcd(a, b)$$

such that x' is greater than the x in the solution (x, y) given by the extended Euclidean algorithm for a and b, which is taken to be the "starting point" (Note this difference from the extendedGcdB function). There are an infinite number of such solutions, and they are created based on the fact that if x_0, y_0 is a solution to the above equation, then so are all x, y such that

$$x = x_0 + m\frac{b}{\gcd(a,b)} \quad \text{and} \quad y = y_0 - m\frac{a}{\gcd(a,b)}$$

where m ranges over the integers.

```
extendedGcdA :: Integer -> Integer -> [(Integer,Integer)]
extendedGcdA a b = result:[ (x0 + m*bOverGCD,y0 - m*aOverGCD) | m <- [1..] ]
    where result = extendedGcd a b
          x0 = fst result
          y0 = snd result
          theGCD = gcd a b
          bOverGCD = toIntegerFromFractionalInteger (fromInteger b / fromInteger theGCD)
          aOverGCD = toIntegerFromFractionalInteger (fromInteger a / fromInteger theGCD)
```

The **extendedGcdB()** function [2, p. 278] computes all of the solutions x', y', given an a and b, for the equation

$$ax + by = \gcd(a, b)$$

such that y' is greater than the y in the solution (x, y) given by the extended Euclidean algorithm for a and b, which is taken to be the "starting point" (Note this difference from the extendedGcdA function). There are an infinite number of such solutions, and they are created based on the fact that if x_0, y_0 is a solution to the above equation, then so are all x, y such that

$$x = x_0 + m\frac{b}{\gcd(a,b)} \quad \text{and} \quad y = y_0 - m\frac{a}{\gcd(a,b)}$$

where m ranges over the integers.

```
extendedGcdB :: Integer -> Integer -> [(Integer,Integer)]
extendedGcdB a b = result:[ (x0 + (-m)*bOverGCD,y0 - (-m)*aOverGCD) | m <- [1..] ]
    where result = extendedGcd a b
          x0 = fst result
          y0 = snd result
          theGCD = gcd a b
          bOverGCD = toIntegerFromFractionalInteger (fromInteger b / fromInteger theGCD)
          aOverGCD = toIntegerFromFractionalInteger (fromInteger a / fromInteger theGCD)
```

MODULE: RSA

The **RSA** module provides functionality for generating public and private keys and for encrypting/decrypting messages using the RSA algorithm.

```
-- FILE: RSA.lhs
-- DATE CREATED: May 4, 2009
-- LAST UPDATED: May 8, 2009
-- AUTHOR: Lawrence Tyler Rush
--         me@tylerlogic.com
--         www.tylerlogic.com
-- VERSION: 1.0.0
module RSA
( RsaKey,
  rsaDecrypt,
  rsaEncrypt,
  rsaExponent,
  rsaKeyGenerator,
  rsaModulus )
where
```

Imports of the RSA Module

This module uses the extended Euclidean algorithm of the `Encryption` module and also the similar function from the `Encryption` module that produces, given a solution from the extended Euclidean algorithm, a second solution. These functions are used by the `RSA` module only during key generation.

```
import Encryption ( extendedGcd , extendedGcdA )
```

This module uses the functions of the `Data.Char` module that convert characters to their integer values and visa versa.

```
import Data.Char ( chr , ord )
```

Abstract Data Types of the RSA Module

To represent an RSA key, the `RsaKey` abstract data type is used and consists of a tuple of the Haskell type `Integer`. As expected, the first `Integer` is the modulus of the RSA key and the second is the exponent of the RSA key.

```
data RsaKey = RK (Integer,Integer)
```

Class Instantiations of the RSA Module

This `Show` class instantiation converts an `RsaKey` to a string via using the `show()` function of tuples to convert the tuple that represents the key to a string.

```
instance Show RsaKey where
    show ( RK key ) = show key
```

Two RSA keys can be compared for equivalence via this `Eq` class instanstiation, which determines whether two RSA keys are equal by determining if the tuples that are used to represent them are equal.

```
instance Eq RsaKey where
    ( RK key1 ) == ( RK key2 ) = key1 == key2
```

Exported Functions of the RSA Module

The **`rsaDecrypt()`** function decrypts a given ciphertext via the RSA decryption algorithm using the given RSA key to convert to plain text. This function uses the padding scheme set forth by the `pad()`/`unpad()` functions. Note that this implementation is quite slow with larger numbers, and that future implementations should implement the use of modular exponentiation.

```
rsaDecrypt :: RsaKey -> [Integer] -> String
rsaDecrypt ( RK (n,d) ) ciphertext = unpad (map ((`mod` n).(^d)) ciphertext)
```

The **rsaEncrypt()** function encrypts a given text via the RSA encryption algorithm using the given RSA key to generate the encrypted list of integers. This function uses the padding scheme set forth by the pad()/unpad() functions. Note that like the decryption method above, this function should make use of modular exponentiation in future implementations.

```
rsaEncrypt :: RsaKey -> String -> [Integer]
rsaEncrypt ( RK (n,e) ) text = map ((`mod` n).(^e)) (pad text)
```

The **rsaExponent()** function returns the exponent of a given key.

```
rsaExponent :: RsaKey -> Integer
rsaExponent ( RK (n,e) ) = e
```

The **rsaKeyGenerator()** function creates a public and private key to use for RSA encryption and decryption, returning both of them in a tuple where the first element is the public key and the second element is the private key.

```
rsaKeyGenerator :: Integer -> Integer -> (RsaKey,RsaKey)
rsaKeyGenerator p q = (RK (n,e),RK (n,d))
    where n = p*q
          tot = (p-1)*(q-1)
          e = 65537
          d = (head.dropWhile (< 0).map fst) gcdE
          gcdE = extendedGcdA e tot
```

The **rsaModulus()** function returns the modulus of a given key.

```
rsaModulus :: RsaKey -> Integer
rsaModulus ( RK (n,e) ) = n
```

Non-Exported Functions of the RSA Module

The **pad()** function is a padding scheme that simply takes a string of characters to their integer values. Yes, very simple, but reversible nonetheless. Furture implemenations should create a better padding scheme.

```
pad :: String -> [Integer]
pad = map (toInteger.ord)
```

The **unpad()** function takes a list of integers and converts them to characters, thereby making a string. Note the integers should be values that actually are the integer values of ACSII characters.

```
unpad :: [Integer] -> String
unpad = map (chr.fromInteger)
```

References

[1] "Bezout's Identity" http://en.wikipedia.org/wiki/B%C3%A9zout%27s_identity Accessed: April 28, 2009.

[2] Dummit, David; Richard Foote. "Abstract Algebra" 3rd Ed. John Wiley and Sons: Hoboken, 2004

[3] "Extended Euclidean Algorithm" http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm Accessed: April 28, 2009.

[4] "Fermat's factorization method" http://en.wikipedia.org/wiki/Fermat%27s_method Accessed: May 7, 2009.

[5] Hudak, P., Peterson J., Fasel J. "A Gentle Introduction to Haskell 98" Unpublished, 1999.

[6] Kaufman, C., Perlman, R., Spenciner, M. "Network Security" Prentice Hall: Englewood Cliffs, 1995.

[7] Kurose, James F. Keith W. Ross. "Computer Networking, A Top-Down Approach" 4th ed. Pearson Education: Boston, 2008.

[8] Moore, Cristopher; Steven Mertens. "The Nature of Computation" Oxford University Press: pre-print. (should be available early 2010)

[9] Nielson, Michael; Isaac Chuang. "Quantum Computation and Quantum Information" Cambridge University Press:Cambridge, 2000.

[10] "Prime Numbers" http://www.haskell.org/haskellwiki/Prime_numbers Accessed: May 4, 2009.

[11] "RSA" http://en.wikipedia.org/wiki/RSA Accessed: April 28, 2009.

[12] "Shor's Algorithm" http://en.wikipedia.org/wiki/Shor%27s_algorithm Accessed: May 5, 2009.

[13] Shoup, Victor. "A Computation Introduction to Number Theory and Algebra" Ver. 2, Cambridge University Press, 2008. Freely available under the Creative Commons license at http://www.shoup.net/ntb/

[14] Simonds, Fred. "Network Security" McGraw-Hill:New York, 1996.

[15] Weisstein, Eric W. "Chinese Remainder Theorem." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/ChineseRemainderTheorem.html Accessed: May 6, 2009.

[16] Wobst, Reinhard. "Cryptology Unlocked" trans. Shafir, A. John Wiley and Sons: Hoboken, 2007.