What is the RSA cryptosystem, and what can it be used for? Many programs such as Internet Explorer, Netscape, and anything requiring the transfer of information. So if your program communicates across a network with sensitive information, RSA may be important to your programs.

RSA Means "Really Stupid Algorithm" Right?

Actually, no. RSA is a cryptosystem or a way of encrypting messages between two parties. The RSA cryptosystem is a way of transporting information in a secure, encrypted way. It does this through the use of keys, which lock or unlock a message. These keys are the private key and the public key. For someone to send you an encrypted message, you send them your public key. They take this public key, encrypt the message, and send the message to you. Unless someone is able to factor 128-bit numbers in less than 100 years, your message is relatively safe. After receiving the message, you decrypt the message using your private key. Someone else holding your public key will not be able to decrypt your message.

A Prime Time for Primes

As I noted above, factoring large numbers takes a very long time. This time is greatly extended when a 256-bit number has only two factors other than 1 and itself. Our goal is to take two large prime numbers and multiply them together to get a number that satisfies our requirement. The only known certain way to be absolutely sure that a number is prime is to check every number up to the square root of the number. As I have stated before, this is going to take an incredible amount of time. Therefore, we need an algorithm that will give primes with an insignificant error that we can control. The most prominent algorithm is called the Miller-Rabin Algorithm. Let's examine this algorithm.

Miller-Rabin is based off of Fermat's Little Theorem. It states: Let n be an odd integer to be tested for primality. Then

- n - 1 = 2^s * d

where:

1. s >= 1 since (n - 1) is even
2. d is odd

Then, n is NOT prime if

1. a^d (mod n) != 1 AND
2. a^((2^k) * d) (mod n) != 1 for all k such that 0 <= k <= (s - 1)

Special Thanks to Tan Chew Keong for pointing out an error in my writing above and helping me fix it.

If you don't know what this mod operator is, don't dispair. Just look below at the Modular Arithmetic section and that will help get you straightened out. If we combine these, for a randomly selected 'a', the probability of being incorrect is less than 1/2. If we try 'x' times, the probability of return ing a 'pseudoprime' or a fake prime is 1/2^x. For all these words, you will probably never figure out what I'm saying, so I'll give you an example in C#.

For this we will want two functions. One that will help us to randomize our 'a', and another tha t will test our alleged prime with the 'a'. We'll call our main function Miller-Rabin and our secondary function Witness.

NOTE: For the purpose that we will be using very large numbers, I will use the BigInteger class found at the address mentioned in the Reader's Comments to hold the numbers. For all my methods, you will have to decide how best to implement them. Many of these methods are already implemented in the BigInteger, but reiterated here for clarity. The BigInteger source code was not used for production of these algorithms, excepting the random bit generator.

```
enum Primality { PRIME, COMPOSITE };
const int NUMBER_BITS = 512;

public bool Witness(BigInteger n, BigInteger a) {
   BigInteger t, u, x, y;
   for(t = 0; ((n - 1) & (2^(t+1))) == 0; t++);        //My hack way of finding t and u
   u = (n - 1) / (2^t);                                //such that 2^t * u = n - 1 where u is odd
   y = ModularExponentiation(a, u, n);        //This is described in the next section
   for(BigInteger i = 1; i <= t; i++) {
      x = (y * y) % n;
      if(x == 1 && y != 1 && y != (n - 1))
         return true;
   }
   if(x != 1)
      return true;
   return false;
}

public BigInteger Random(BigInteger low, BigInteger high) {
   BigInteger RandInt;
   System.Random rand = new Random();
   do {
      RandInt.genRandomBits(logn(2, high), rand);
   } while(RandInt > high || RandInt < low);
   return RandInt;
}

public Primality Miller-Rabin(BigInteger n, BigInteger s) {
   if(n == 3)
      return PRIME;
   BigInteger a;
   for(BigInteger j = 1; j <= s; j++) {
      a = Random(2, n-2);
   if(Witness(a, n))
         return COMPOSITE;
   }
   return PRIME;
}
```

Remember that Miller-Rabin is definately correct if it returns COMPOSTIE, but 'n' may be a pseudoprime if Miller-Rabin returns PRIME.

The probability of Miller-Rabin returning a pseudoprime depends on the 's' you supply, but is not greater than (1/2)^s. Therefore, if we use an 's' of 8, Miller-Rabin will return a pseudoprime not more than one in eight times. For a good probability, I recommend an s between 100 and 1000, or higher if you so desire.

Okay, so now we have a prime number. What does this do for us? Until we learn some modular arithmetic, not much, so we will cover that subject next.

Modular Arithmetic

The basis of the RSA cryptosystem is the modulus operator. What is the modulus operator, you ask? The modulus operator gives a result based on the Division Theorem which, simplified, states that for an 'a' and a 'b', both integers with b > 0, there exists a 'q' and an 'r' such that:

- a = q * b + r

where r > 0. 'q' ends up being equal to the floor of (a / b), and 'r' is equal to (a mod b). This is where our modulus operator comes from. In case I lost you, here are some examples:
a = 11, b = 3
11 = 3q + r
q = 3 and r = 2, therefore (11 mod 3) = 2
In simpler terms, you can also think of it as being the remainder of division.

So now let's do some modular arithmetic.
5 + 4 (mod 7) = 9 mod 7 = 2
9 - 17 (mod 5) = -8 mod 5 = 2 (-8 = 5 * -2 + 2)
5 * 4 (mod 9) = 20 mod 9 = 2
beware that division with integers is not good, and that it often leads to non integers, avoid it if you can!
5^3 (mod 3) = 125 mod 3 = 2
and so on...
These operations can be combined and split as in:
5^3 (mod 3) = (5 (mod 3)) * (5^2 (mod 3)) = 2 * 1 = 2

Ack!! For our project, we may end up doing an exponent 2^512 times. That is just not any fun at all, so we need to simplify it, and that is what this next algorithm is here for:

```
public BigInteger ModularExponentiation(BigInteger a, BigInteger b, BigInteger n) {
   BigInteger c = 0, d = 1;
   for(BigInteger i = logn(2, b); i >= 0; i--) {        //logn(base, exponent)
      c = 2 * c;
      d = (d * d) % n;
      if(((b) & (2^i)) != 0) {
         c++;
         d = (d * a) % n;
      }
   }
   return d;
}
```

So now that we have the basics of modular arithmetic, let's look deeper. One of the most important aspects of modular arithmetic is that if 'e' is relatively prime to 'n' then there is a 'd' such that

- e * d (mod n) = 1

In other words, there is another number also relatively prime to 'n' that is its reciprocal.
Now what in the word is relatively prime??? Relatively prime means that the greatest common divisor (gcd) between two numbers is 1, or that there is no number other than 1 that divides both 'e' and 'n' without a remainder.

Let's look at an example:
gcd(91, 28) = 7 = 91 * 1 + 28 * -3
gcd(4, 25) = 1 = 25 * 1 + 4 * -6

What are all those extra numbers tacked on to the end? We will be using those for finding our modular inverse. And for that, we will need a gcd function that can return those extra numbers. The Extended-Euclid Algorithm fits that bill perfectly, so that's what we will be using:

```
public BigInteger Extended-Euclid(BigInteger a, BigInteger b, out BigInteger x, out BigInteger y) {
   if(b == 0)
      if(a < 0) {
         x = -1;
         y = 0;
         return -a;
      } else {
         x = 1;
         y = 0;
         return a;
      }
   BigInteger xprime, d = Extended-Euclid(b, a % b, xprime, x);
   y = xprime - (a / b) * x;
   return d;
}
```

This function is designed to work for both positive and negative integers, although for our project it is unlikely that we will have any negative numbers for a and b. What this function gives us is information in the form:

- d = (a * x) + (b * y)

Now we will see this function's great ability: Finding Modular Inverses!
So we want our numbers 'e' and 'd' that multiply together and equal 1. Okay, good. Now lets mess with the above equation.
d = (a * x) + (b * y)
(1 = (e * d) + (n * y)) mod n --By plugging in values aimed at getting what we want
1 = (e * d) mod n + (n * y) mod n --Becuase (n * y) mod n will always equal zero, we get:
1 = (e * d) mod n
Therefore, by plugging in 'e' for 'a' and 'n' for 'b', the returned 'x' will be the multiplicative inverse of 'e'. Whew!!

A Phine Time for Phi

Phi is going to be very important to our cryptosystem. What is Phi you ask. For one, it is a greek letter looking

like an 'O' with an 'I' down the center, and it is also the function representing the number of integers less than or equal to 'n'. In Example:
Phi(5) = 4, {1, 2, 3, 4}
Phi(12) = 4, {1, 5, 7, 11}
Phi, rather interestingly can be determined by prime factorization. Phi(n) is the multiplication of one copy of each factor minus one, multiplied by all the other factors. Another Example:
Prime Factors of 5 are 5
Phi(5) = (5 - 1) = 4
Prime Factors of 12 are 2^2 * 3
Phi(5) = (2 - 1) * 2^1 * (3 - 1) = 4
So, what does Phi do?? We will use Phi in the construction of our cryptosystem which just happens to be our next section!

Construction of the RSA Cryptosystem

Now that we have all the tools we need, we can construct our cryptosystem using the functions we used above. The steps are:

1. Select two large prime numbers which we will call 'p' and 'q' such that p != q, and make them an arbitrary number of bits, say 512.
2. Compute our 'n' to be n = p*q.
3. Select a small, odd integer that is relatively prime to Phi(n) and not 1 (For this case, Phi(n) = (p - 1) * (q - 1))
4. Compute 'd' to be the multiplicative inverse of 'e' modulo 'Phi(n)'
5. The ordered pair (e, n) is your RSA public-key
6. The ordered pair (d, n) is your RSA private-key
7. Using whatever method you prefer, make sure that 'p' and 'q' are completely annihilated. Not doing so could compromise the security of your cryptosystem!

Deciphering the Tutorial

Now for a demonstration of how RSA works in practice:
Take a certain message 'M', then its encrypted form is the binary 'Z'. Then, we obtain Z by

- (M)^e (mod n) = Z

Inversely, we can decrypt our message we preform our inverse:

- (Z)^d (mod n) = M

All messages should be encrypted with you public-key (e, n) to ensure that you are the only one capable of reading the message.

Summation

I hope that you find this tutorial to be an invaluable tool in the future. Thank you for taking the time to read this introduction and happy programming.

Marcus Griep is a high school student currently taking Computer Science and Mathematics courses at CU. He has been an avid programmer for many years and is currently working on a physics engine to more accurately model physics for controlled bodies.

RSA Encryption Tutorial

Disclaimer

This tutorial only explains the basics of RSA Encryption. Do not expect any advanced RSA proofs here. If you see any mistakes please email me.

Two common uses of RSA Encryption algorithms are secure data transfer and digital signatures. Communication between two computers sometimes needs to be secure so that a possible eavesdropper cannot decipher the exact messages that are being passed back and forth between the two computers. A common way to implement this is to use an encryption algorithm. RSA utilizes a public key cryptosystem to encrypt data.

Assume that one computer is called Tux and the other is called Daemon. Tux and Daemon both need a set of keys (public and private). The public keys can be exchanged with anyone but the private keys must be kept secret! Only the creator of the private key should have access to that key. Now we will go through the steps of creating a pair of keys.

We need two large primes (randomly selected): p & q
Several books suggest that p and q be at least 512 bits each. Higher levels of encryption will require even bigger numbers.

*Calculate n as n = p*q*

We need a small odd integer e such that e is relatively prime to Φ(n).
Φ(n) = (p-1)(q-1) is known as the Euler phi function*
Determining if e is relatively prime to Φ(n) is very simple if we utilize the Extended Euclid Algorithm.
Extended Euclid computes 6 integers: a, b, floor(a/b), d, x, y such that d = gcd(a, b) = ax + by
The following is pseudocode for Extended Euclid
Note: "%" means "modulo"

```
extended-euclid(a, b) {
  if(b == 0)
     return(a, 1, 0)
  (d', x', y') = extended-euclid(b, a % b)

  return(d, x, y)
}
```

The following table demonstrates a run of extended euclid on the numbers 99 and 78.

a	b	floor(a/b)	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	--	3	1	0

Having trouble understanding in what order that table was completed?

Simply follow the algorithm. Remember that Extended Euclid is recursive so you simply start off by filling in a and b (the arguments given to extended-euclid). Then compute the floor of a/b. Note that if b is greater than a then the algorithm will take one run to reverse the numbers. Once you fill in a, b, floor(a/b) then proceed to the next row. The next row represents another call to extended-euclid. The arguments to a and b can be found by observing the following line in the algorithm: (d', x', y') = extended-euclid(b, a % b)
The argument for a will now be b from the previous row. The argument to b will be a%b using the values of a and b from the previous row. You continue in this fashion until you reach a row where the value of b is equal to 0. The algorithm states that if b is equal to 0 then return(a, 1, 0).
The return statement now allows you to start filling in the values for the second half of the table (values: d, x, y).
The return statement gives you the first set of values for d, x, y so simply write in the current value of a for d, 1 for x, and 0 for y.
*Now proceed one row up and fill in the values for d, x, y according to the the following statement in the algorithm: (d, x, y) = (d', y', x' - floor(a/b)*y')*
*What are the d', x', and y'? The variables "d prime", "x prime", and "y prime" represent those same values from the previous (lower) row. "Prime" means "before" in our case. So... the value d for the current row gets the value of d from the previous row, the value x for the current row gets the value of y from the previous row, the value y for the current row gets the result of (x' - floor(a/b)*y')*
Continue in this fashion until you reach the first row. Then you should have the value of d which gives the greatest common divisor of a and b or in your case e and Φ(n). If d is equal to 1 then the a is relatively prime to b or e is relatively prime to Φ(n).

Now we need to compute a value D which is the multiplicative inverse of e, % Φ(n).

We now set P = (e, n) as the public key and S = (D, n) as the secret key.

Now we can encrypt data with the following equations:
let M be the data you want to encrypt.
P(M) = C where C is known as the ciphertext or encrypted text.
S(C) = M where M is the original data.
Note that P(S(C)) = C and S(P(M)) = M and P(S(C)) = P(M) and S(P(M)) = S(C)
The previous equations might come in useful but are not necessary since you can figure them out on your own.

What is all this saying?
Let's go back to our two computers Tux and Daemon that wish to exchange information securely by first encrypting the data. Tux would use Daemon's public key to encrypt the data it wishes to send to Daemon. Daemon would receive the ciphertext C and would use its secret key and apply it to C so that it can retrieve the original data M.

So Tux would use: $P_{Daemon}(M) = C$ and send C to Daemon.
Daemon would use: $S_{Daemon}(C) = M$ and read the data M.

Similarly if Daemon wanted to send data secure to tux then:
Daemon would use: $P_{Tux}(M) = C$ and send C to Tux.
Tux would use: $S_{Tux}(C) = M$ and read the data M.

What does P(M) actually do mathematically?
$P(M) = (M^e \% n) = C$

What does S(C) actually do mathematically?
$S(C) = (C^D \% n) = M$