CMSC 222 – Discrete Structures
Project 1: RSA - Overview

Joseph George

Overview:

The RSA cryptography algorithm, developed by Rivest, Shamir, and Adleman, is based off of a public key algorithm that is capable of both encryption and decryption. The algorithm is based off of the idea of a public key and a private key. An individual has a public key, which anyone can use to encrypt a message and send it to the individual. However, the message encrypted by an individual's public key can only be decrypted by that individual's private key, so in the event of an interception, the message cannot be deciphered. Methods of encryption and decryption using the RSA algorithm are based off of large prime numbers. The aim of this project is to implement a version of the RSA public key crytographic algorithm, and then perform experiments using the algorithm.

General Background to Methodology:

A public key is generated by choosing two large prime numbers, p and q. A number, e, is chosen that is relatively prime to $(p-1)(q-1)$. The public key can then be represented as $< e, n >$. A plaintext message represented an an integer M is then encrypted using $C = M^d mod n$. Consequently, a private key is generated using a number d that is the multiplicative inverse of $e mod (p-1)(q-1)$. After finding a private key, $< d, n >$, the original plaintext message can be deciphered using $M = C^d mod n$. This project implements this methodology through three files containing various RSA encryption type algorithms which will now be discussed in detail: rsaEncrypt.java, rsaDecrypt.java, and gcd.java.

Methodology and Overview of Encryption Process:

The RSA encryption process can be implemented in java code. The program can perform the encryption algorithm on a provided string of ASCII characters using a public key, $< e, n >$. The exact algorithmic implementation can be broken down as such:

1. **Breaking down the string of characters to translate into ASCII**

   The first step to encrypting a plaintext message is to break down the characters so that each individual character can be translated to the ASCII numerical representation. Taking the length of the string, one can run through the string, taking each individual character and putting all of the characters into a separate array for later manipulation. Breaking down the plaintext into individual characters ensures a smooth translation into ASCII numerical representation in the program.

2. **Creation of the ASCII values using the characters of the plaintext message**

   Taking the individual characters of the plaintext, one can produce a long string of digits with the ASCII numerical translation equivalent. Going through each individual character in your array that contains the individual characters, each character can be translated to its ASCII equivalent, and then concatenated with the previous character, until there are no more characters left to translate into ASCII. The end result of this algorithm is a long string that is constructed through the continuous concatenation of characters being translated to their ASCII equivalent, using a while control statement.

3. **Separate the ASCII string into workable chunks for encryption**

   Now that one has a string of integers, the ideal thing to do is separate these into chunks, and turn them into integers in the process (as opposed to remaining strings, which cannot be mathematically implemented during encryption). This can be efficiently done by using various flow control methods, and indexing sections of 8 strings which can then be turned into integers. The end result of this implementation should be an array containing at each index a chunk of 8 digits.

4. **Encryption using modular exponentiation**

   This is the last stretch of the implementation of the RSA encryption algorithm into the program. The modular exponentiation algorithm can be described by the following pseudo-code:

```
c = 1;
power = m mod n;
for ( i = 0; i < # bits in e; i++ )
{
   if ( ith least significant bit is 1 )
   {
      c = (c * power) mod n;
   }

   power = (power * power) mod n;
}

return c;   // c = M^e mod n
```

   What is the doing exactly? Before going into discussion it is important to note that this section of the program will utilize three parameters: m - the chunk of ASCII digits to encrypt with the algorithm, e - the public key encryption exponent (note that a detailed discussion of what certain values like e stand for exactly will follow), and a large number n such that $n = pq$, where p and q are your prime numbers given.

   The algorithm above is a smart way to compute $M^e mod n$ without overflow issues in the program. It uses a process called modular exponentiation, which is a method of exponentiation over a modulus (ignoring the precise mathematical definition concerning congruence, a mod operator in computer languages is simply a remainder). By taking the binary representation of e, the for loop goes through the binary representation starting at the end, where the least significant bit is, and performs modular exponentiation. The result is the ciphertext - a long string of digits, and effectively, your encrypted text using RSA algorithmic methods.

Methodology and Overview of Decryption Process: The RSA decryption process can also be implemented in Java code. The program can perform the RSA decryption algorithm on a provided string of encrypted digits using the given private key $< d, n >$, where d is the private key decryption exponent and $n = pq$. The exact algorithmic implementation can be broken down as such:

1. **Break the input string of digits into blocks of 9**

   The first step in the decryption process is taking the long string of digits and breaking them up into smaller chunks of 9. A good implementation idea at this point would be to create a string array whose size is the length of the string of digits, divided by 9, so there is enough room to store them. Using carefully set indices, one can repeatedly go through the string of digits, and store each chunk of 9 inside of the string array.

2. **Performing the decryption algorithm on the blocks of digits**

   This part of the algorithm essentially involves doing the reverse of the encryption using modular exponentiation. This section of the program utilizes three parameters: c - the blocks of encrypted digits (ciphertext), d - the private key decryption exponent, and n, private key long number from $n = pq$ where p and q are prime. Note that the "power" variable this time will be $Cmodn$. If the resulting decryption of any chunk of integers is less than 8 digits, one must pad to the left with sufficient zeros. After performing the decryption on each chunk of encrypted digits, it is important to concatenate all of the resulting integers for the next step of the algorithm

3. **Conversion of decrypted integers to ASCII equivalent characters**

   At this point in the algorithm, one must break the resulting decrypted integers into chunks of three such that these chunks of three can be converted to their ASCII equivalent representation. A "chunk holder" array can be created, each index containing a chunk of three integers. This array can then be put through another loop, which takes each chunk of three, and after converting the chunk to its ASCII equivalent, stores the appropriate ASCII character into a new array. The result of this process is an array which contains all of the individual characters that is the decrypted message. The final step of this decryption process is to simple concatenate the contents of this character array together to get your very own message.

Methodology and Overview of GCD/Euclid's Algorithm Process:

The gcd portion of the Java implementation of the RSA encryption algorithm is extremely important. It is responsible for performing the Euclidean Algorithm on two provided input integers, and producing not only the gcd and book-keeping parameters u and v (will be discussed later), but it is also responsible for producing your private key decryption exponent d.

1. **The Euclidean Algorithm**

   A general description of how the Euclidean algorithm works is essential to understanding its implementation. The Euclidean Algorithm, first described in Euclid's *Elements*, is a method for finding the greatest common divisor of two positive integers, a and b. The following theorem forms the backbone of the algorithm:

   Let x and y be two positive integers. Then $gcd(x, y) = gcd(ymodx, x)$.

   More precisely, the algorithm can be implemented through the following pseudo-code:

```
Input: Two positive integers, x and y.
Output: gcd(x, y).

If ( y < x )
     Swap x and y.
r = y mod x.
While ( r does not equal 0 )
     y := x
     x := r.
     r := y mod x.
End-while
Return(x)
```

The implementation of this pseudo code goes through the steps of the Euclidean Algorithm, each step replacing y with x, x with r, and computing a new r. When this r finally hits zero, the gcd of the two integers is returned. It is important to note, however, that gcd can also be represented as $gcd(a, b) = r = u(a) + v(b)$, where r is the final remainder after performing the Euclidean Algorithm and u and v are numerical parameters which can be calculated by adding in a few steps to the implementation of the Euclidean algorithm.

Generally, at each step of the Euclidean Algorithm, k, the following two equations can be used to calculate the u and v values.

$$u_k = u_{k-2} - q_k u_{k-1}$$

$$v_k = v_{k-2} - q_k v_{k-1}$$

where $u_{-2} = 1$ and $v_{-2} = 0$

and where $u_{-1} = 0$ and $v_{-1} = 1$

Since the u and v values can be incremented at each step, it is necessary to create a dynamic array list that will contain the calculated values of u and v at each step of the algorithm. This means that the equations for u_k and v_k can be implemented into the while loop of the pseudo-code above. When $r = 0$ and the loop stops, the array lists will contain your u and v parameters as discussed above.

The decryption exponent d will be equal to this u parameter, unless u is negative, in which case u and b (recall that b is in $gcd(a, b)$) are added together to get the equivalent positive value of u modulo b. Also recall that d is the inverse to $amod(p-1)(q-1)$. This concludes the algorithm for calculating the GCD and the associated parameters, along with the decryption exponent. The final program should return all of these values.

Testing/Experimentation:

To make sure the algorithms were implemented into the programs correctly, the decryption exponent d was determined for a given public key pair $< e, n >$. Assuming that the given public key exponent for encryption e is 65537 and the two prime values are $p = 22943$ and $q = 22961$, the GCD program was run using the parameters e and $(p-1)(q-1)$. The result of running the algorithm based off of these parameters returned a private key decryption exponent $d = 177514433$. Consequently, the decryption algorithm was run using a long string of encrypted digits as one parameter, d as the second parameter, and $n = pq$ as the final parameter. The result of the decryption on the ciphercode produced the following message:

”Society, how can you teach; I wanna know, if you don’t practice what you preach? – J. Tweedy”

In order to further test the program, the encrypted message of someone else was also decrypted using my decryption program, along with their own unique p and q values. Specifically, using the same public key e along with Tenbus’s two prime numbers $p = 23131$ and $q = 23143$, the decrpytion exponent for Tenbus was calculated to be 230234093. Putting this through the decryption algorithm, the resulting message turned out to be another quote by J. Tweedy:

”To conquer fear, that’s quite a quest; Until we do, never rest. – J. Tweedy”

Lastly, it is important to note that the code was put through a stress test - for example, the incorrect number of command line arguments were entered, and the decryption and encryption program was fed a variety of different sized messages.

Determining Primes p and q: A Side Note for the Audience

There do exist some methods of finding large prime numbers, but it is okay to state that for the purpose of this project, a large number was chosen and then its primality was tested probabilistically.

Conclusion

The RSA cryptography algorithm is an efficient means of keeping information secure because of its basis in the difficulty that comes from factoring large integers that are the product of two large prime numbers. The implementation of this algorithm into code shows a basic example that the mathematical concepts of modular exponentiation and Euclid’s Algorithm can be utilized to encrypt and decrypt messages.