Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Beardsmore	Student ID:	15504319
Other name(s):	Connor		
Unit name:	Fundamental Concepts of Cryptography	Unit ID:	ISEC2000
Lecturer / unit coordinator:	Wan-Quan Liu	Tutor:	Antoni Liang
Date of submission:	19/05/17	Which assignment?	2 (Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: ______________________ Date of signature: 19/05/17

(By submitting this form, you indicate that you agree with all the above text.)

FCC200 Report
RSA Cryptosystem Implementation

Connor Beardsmore - 15504319

Curtin University
Science and Engineering
Perth, Australia
May 2017

RSA Implementation

Modular Exponentiation

Modular exponentiation is used to calculate the remainder when a base b is raised to an exponent e and reduced by some modulus m. The simple right-to-left method provided by Schneier 1996 utilizes exponentiation by squaring. The full Java code for the implementation of this method is illustrated below. The running time of this algorithm is $O(log\ e)$ which is a huge improvement over more simplistic methods of time $O(e)$ (Stallings 2011).

```
        n = p * q;
        totN = ( p - 1) * ( q - 1 );

        //use EEA to select e,n satisfying gcd(e, varphi(n)) == 1
        e = 1;

        //use EEA to solve private key d
        d = NumberTheory.extendEuclid( e, totN );

        //convert keyboard symbol to ASCII code for encrypt + decrypt

        //implement RSA encryption and decryption using Q1 of this assignment

    }

//-----------------------------------------------------------------------------
}
```

The code above was utilized to calculate the following example:

$$236^{239721}\ mod\ 2491 = 236$$

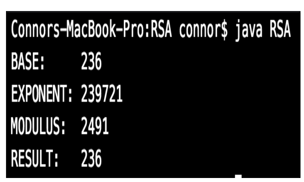

Figure 1: Modular Exponentiation Example

RSA Testing

```
\subsection{AFS Algebras}
###&&&
The Iris dataset is used as an illustrative example for AFS algebras through
this paper. It has 150 samples which are evenly distributed in three
classes and 4 features of sepal length($f_1$), sepal
width($f_2$), petal length($f_3$), and petal width($f_4$). Let a
pattern $x=(x_{1},x_{2},x_{3},x_{4})$, where $x_{i}$ is the $i$th
feature value of $x$. The following three linguist fuzzy rules have been obtained for Class 1 to build the

\subsection{Shannon��s Entropy}
Let $X$ be a discrete random variable with a finite set containing $N$ symbols
$x_{0}, x_{1}, \ldots, x_{N}$. If an output $x_{j}$ occurs with probability $p(x_{j})$, then the
amount of information associated with the known occurrence of the output $x_{j}$ is defined as
\begin{equation}
I(x_{j}) = -log_{2} p(x_{j})
\end{equation}
Based on this, the concept of Shannon��s entropy is defined as follows:
)))))~~~~~
```

Figure 2: RSA Plaintext

```
�E�3Ehx��+�k�^�ą{9h3'
E�����������r�hī'�E�
�
Eh���E��Eh�
E�
���{{�E�'
�Âh�h0
f{h⌶+'ą^��
{9h3'
EĞ�'+�9�������E�
h'\ī���
EĚ��E
f{hE���x��
f+��+���+'f
��+��
EE+x�
�h�Þ�Ğ�h�N�+��+xx�''h�xh�+�Ğ�h�+⌐⌐ā0�k�����E�h���h�
E���3h9��kh��
��+�������0�k����q�{+9�ks���0�k������h�kh��
��+�����
Eh�+�Ğ��E�Ğ�h�x+�xh��+����
��+�_E�ḩ'+���E�h���h�
E⌶+{{+E
������������
```

Figure 3: RSA Ciphertext

```
\subsection{AFS Algebras}
###&&&
The Iris dataset is used as an illustrative example for AFS algebras through
this paper. It has 150 samples which are evenly distributed in three
classes and 4 features of sepal length($f_1$), sepal
width($f_2$), petal length($f_3$), and petal width($f_4$). Let a
pattern $x=(x_{1},x_{2},x_{3},x_{4})$, where $x_{i}$ is the $i$th
feature value of $x$. The following three linguist fuzzy rules have been obtained for Class 1 to build the


\subsection{Shannon��s Entropy}
Let $X$ be a discrete random variable with a finite set containing $N$ symbols
$x_{0}, x_{1}, \ldots, x_{N}$. If an output $x_{j}$ occurs with probability $p(x_{j})$, then the
amount of information associated with the known occurrence of the output $x_{j}$ is defined as
\begin{equation}
I(x_{j}) = -log_{2} p(x_{j})
\end{equation}
Based on this, the concept of Shannon��s entropy is defined as follows:
)))))~~~~~
```

Figure 4: RSA Recovered Plaintext

Additional Questions

Signature Forgery

The RSA signature structure can be described as follows. justify the forgery etc etc

Birthday Attack

In a group of 23 randomly selected people, the probability that two of them share the same birthday is larger than 50%

Firstly, the probability that two people have different birthdays is found:

$$1 - \frac{1}{365} = \frac{364}{365} = 0.99726$$

This can be extended to determine if three people have different birthdays:

$$1 - \frac{2}{365} = \frac{363}{365} = 0.99452$$

Utilizing conditional probability (Liu 2017) we can construct the probability that all 23 people have different birthdays. This is simply represented as a series of fractions with their product producing the resultant probability:

$$1 \times (1 - \frac{1}{365})(1 - \frac{2}{365})...(1 - \frac{22}{365}) = 0.493$$

To find the probability that two of the people have the same birthday, we inverse this number by subtracting from the total probability (1):

$$1 - 0.493 = 0.507 = 50.7\%$$

It is thus evident that the probability of two people in a set of 23 random selected shared the same birthday is greater than 50%.

RSA Source Code

RSA.java

```java
/****************************************************************************
* FILE: RSA.java
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Performs RSA public-key encryption or decryption on a given file
*   LAST MOD: 01/05/17
*   REQUIRES: NONE
****************************************************************************/

import java.util.*;
import java.io.*;

public class RSA
{
//---------------------------------------------------------------------------

    public static void main( String[] args )
    {
        int p, q;
        int e, n;
        int totN;
        int d;

        //select two primes p/q using Q3 of lab 2. between 1000 and 10000
        p = NumberTheory.generatePrime();
        do
        {
            q = NumberTheory.generatePrime();
        } while ( p == q );

        //calculate n=pq
        n = p * q;
        totN = ( p - 1) * ( q - 1 );

        //use EEA to select e,n satisfying gcd(e, varphi(n)) == 1
        e = 1;

        //use EEA to solve private key d
        d = NumberTheory.extendEuclid( e, totN );

        //convert keyboard symbol to ASCII code for encrypt + decrypt

        //implement RSA encryption and decryption using Q1 of this assignment

    }

//---------------------------------------------------------------------------
}
```

NumberTheory.java

```java
/***************************************************************************
* FILE: NumberTheory.java
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Basic helper functions for performing RSA encryption
*   LAST MOD: 01/05/17
*   REQUIRES: NONE
***************************************************************************/

import java.util.Random;

public class NumberTheory
{
    //CONSTANTS
    public static final long LIMIT = 10000000000L;
    public static final int LOWER_PRIME = 1000;
    public static final int UPPER_PRIME = 10000;
    public static final int CONFIDENCE = 25;

//---------------------------------------------------------------------------
    //NAME: modularExpo()
    //IMPORT: base (int), exponent (int), modulus (int)
    //EXPORT: result (int)
    //PURPOSE: Calculate the value base^exponent mod modulus efficiently

    public static int modularExpo( int base, int exponent, int modulus )
    {
        int result = 1;

        //check upper limit
        if ( ( base > LIMIT ) || ( exponent > LIMIT ) || ( modulus > LIMIT) )
            throw new IllegalArgumentException("INVALID MODULAR EXPO NUMBER");

        //anything mod 1 results in 0
        if ( modulus == 1 )
            return 0;

        //reduce base to the lowest form
        base = base % modulus;

        //loop until all exponents reviewed
        while ( exponent > 0 )
        {
            //if the bit is set (from lowest to highest order bit)
            if ( ( exponent & 1 ) == 1 )
                //increase result by the base
                result = ( result * base ) % modulus;
            //shift exponent to consider the next higher order bit
            exponent = exponent >> 1;
            //increase the base
            base = ( base * base ) % modulus;
        }

        return result;
    }

//---------------------------------------------------------------------------
//FUNCTION: generatePrime()
//EXPORT: newPrime (int)
//PURPOSE: Generate a prime number between LOWER_PRIME and UPPER_PRIME

     public static int generatePrime()
     {
```

```
         Random rand = new Random();

         //loop until the random generated number is a prime, based on Lehmanns
         //number generated will be between 1000 and 10000
         int newPrime = rand.nextInt( UPPER_PRIME - LOWER_PRIME ) + LOWER_PRIME;
         while ( !primalityTest( newPrime, CONFIDENCE ) )
            newPrime = rand.nextInt( UPPER_PRIME - LOWER_PRIME ) + LOWER_PRIME;

         return newPrime;
     }

//---------------------------------------------------------------------------------
// FUNCTION: primalityTest
// IMPORT: p (int), tests (int)
// EXPORT: isPrime (boolean)
// PURPOSE: Determine if a number is prime or not to some confidence level

    private static boolean primalityTest( int prime, int tests )
    {
        long a, r;
        long expo;
        Random rand = new Random();

        //perform multiple tests
        for ( int ii = 0; ii < tests; ii++ )
        {
            //calculate r result
            a = rand.nextInt() % ( prime - 1 ) + 1;
            expo = ( prime - 1 ) >> 1;
            r = (long)Math.pow( a, expo ) % prime;

            //if r not 1 or -1 it is 100% not prime
            if ( ( r != 1 ) && ( r != -1 ) )
                if ( ( r != ( prime - 1 ) ) && ( r != ( prime - 1 ) ) )
                return false;
        }

        return true;
    }

//---------------------------------------------------------------------------------
// FUNCTION: gcd
// IMPORT: a (int), b (int)
// EXPORT: gcd (int)
// PURPOSE: Find greatest common denominator of 2 numbers

    public static int gcdFunction( int a, int b )
    {
        int gcd = 1;
        int quotient;
        int residue;

        //swap the elements to ensure b is smaller
        if ( a < b )
        {
            int temp = a;
            a = b;
            b = temp;
        }

        //check if either number is 0
        if ( a == 0 )   return b;
        if ( b == 0 )   return a;

        //satisfy the equation A = B * quotient + residue
```

```
        quotient = a / b;
        residue = a - ( b * quotient );

        //recursively call gcd
        gcd = gcdFunction( b, residue );

        return gcd;
    }

//---------------------------------------------------------------------------------
// FUNCTION: extendEuclid
// IMPORT: a (int), n (int)
// EXPORT: t (int)
// PURPOSE: Extended Euclidean algorithm to find inverse modular

    public static int extendEuclid( int a, int n )
    {
        int t = 0, newt = 1;
        int r = n, newr = a;
        int q = 0, temp = 0;

        //only applicable if the gcd is 1
        if ( gcdFunction( a, n ) != 1 )
            return -1;

        //perform the eea
        while ( newr != 0 )
        {
            q = r / newr;
            temp = t;
            t = newt;
            newt = temp - ( q * newt );
            temp = r;
            r = newr;
            newr = temp - ( q * newr );
        }

        //ensure t is positive
        if ( t < 0 )
            t = t + n;

        return t;
    }

//---------------------------------------------------------------------------------
}
```

References

Liu, Wan-Quan. 2017. *Lecture 6: Number Theory*. Curtin University.

Liu, Wan-Quan. 2017. *Lecture 5: Public Key Cryptosystem*. Curtin University.

Schneier, Bruce. 1996. *Applied Cryptography*. 5th ed. John Wiley & Sons Inc.

Stallings, William. 2011. *Cryptography and Network Security: Principles and Practice*. 5th ed. Prentice Hall.