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 provides a significant improvement over more simplistic methods of time complexity $O(e)$ (Stallings 2011).

```
//---------------------------------------------------------------------------
//NAME: modularExpo()
//IMPORT: base (int64_t), exponent (int64_t), modulus (int64_t)
//EXPORT: result (int64_t)
//PURPOSE: Calculae the value base^exponent mod modulus efficiently

int64_t modularExpo(int64_t base, int64_t exponent, int64_t modulus)
{
    int64_t result = 1;
    base = base % modulus;

    //check upper limit
    if ( ( base > LIMIT ) || ( exponent > LIMIT ) || ( modulus > LIMIT ) )
        return -1;

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

    //loop until all exponents reviewed
    while( exponent > 0 )
    {
      //check least significant bit
      if( exponent & 1 )
        result = ( result * base ) % modulus;

      //shift exponent to consider next bit
      exponent >>= 1;
      base = ( base * base ) % modulus;
    }

    return result;
}

//---------------------------------------------------------------------------
```

The code above was utilized to calculate the following example:

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

The running of this code provided the following output:

```
Connors-MBP:rsa connor$ ./rsa
MODULAR EXPONENTIATION:
        BASE:     236
        EXPONENT: 239721
        MODULUS:  2491
        RESULT: 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�h̬'+���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

If a receiving person were to try to verify message m, they would input H(m) along with the previously mentioned variables into the verification function and find that the output is equal to r. This would lead the receiver to believe that m originated from Alice. This situation, however is not as worrying as it would appear as it would be very unlikely that m contained any meaningful information. This is very similar to the Birthday Problem mentioned in Lecture 8, while it is quite probable to find two people in a group of thirty or more who share a birthday, but it is far less probable to find somebody whose birthday matches a specific birthday. It may be trivial for Bob to craft an m with a hash matching H(m), it is far less likely for Bob to craft a meaningful message with a hash matching H(m).

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 sharing the same birthday is greater than 50%.

RSA Source Code

makefile

```
# Makefile For RSA Implementation
# FCC200 Assignment
# Last Modified: 02/05/17
# Connor Beardsmore - 15504319

# MAKE VARIABLES
CC=gcc
CFLAGS=-std=c99 -Wall -pedantic -g
EXEC=rsa
OBJ=main.o numberTheory.o
TESTS=output.txt original.txt

# RULES + DEPENDENCIES

$(EXEC): $(OBJ)
	$(CC) -o $(EXEC) $(OBJ)

main.o: main.c main.h
	$(CC) $(CFLAGS) -c -o main.o main.c

numberTheory.o: numberTheory.c numberTheory.h
	$(CC) $(CFLAGS) -c -o numberTheory.o numberTheory.c

clean:
	rm -r $(OBJ) $(EXEC) $(TESTS)
```

numberTheory.h

```c
/***************************************************************************
* FILE: numberTheory.h
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Header file for number theory functionality
*   LAST MOD: 02/05/17
*   REQUIRES: stdio.h, stdlib.h
***************************************************************************/
#include <stdio.h>
#include <stdlib.h>

//CONSTANTS
#define PRIME_TESTS 25
#define LOWER_PRIME 1000
#define UPPER_PRIME 10000
#define LIMIT 10000000000

//BOOLEANS
#define TRUE 1
#define FALSE 0

//PROTOTYPES
int primalityTest(int64_t,int);
int64_t generatePrime(int,int);
int64_t modularExpo(int64_t, int64_t, int64_t);
int64_t extendedEuclid(int64_t,int64_t);
int64_t findGCD(int64_t, int64_t);

//---------------------------------------------------------------------------
```

numberTheory.c

```c
/***************************************************************************
* FILE: numberTheory.c
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Functionality for basic number theory techniques
*   LAST MOD: 02/05/17
*   REQUIRES: numberTheory.h
***************************************************************************/

#include "numberTheory.h"

//---------------------------------------------------------------------------
//NAME: primalityTest
//IMPORT: prime (int64_t), tests (int)
//EXPORT: isPrime (int)
//PURPOSE: Check if a number is prime or not to some confidence level

int primalityTest( int64_t prime, int tests )
{
  int64_t a;
  int64_t r;
    int64_t exponent;
  int isPrime = TRUE;

  for( int ii = 0; ii < tests; ii++ )
  {
        //calculate r result
    a = ( rand() % prime ) + 1;
    exponent = ( prime - 1 ) >> 1;
    r = modularExpo( a, exponent, prime );

        // if r not 1 or -1 it is 100% not prime
    if( ! ( (r == 1) || (r == (prime-1))) )
      return FALSE;

  }
  return isPrime;
}

//---------------------------------------------------------------------------
//NAME: generatePrime()
//EXPORT: newPrime (int64_t)
//PURPOSE: Generate a random prime number between the two bounds given

int64_t generatePrime( int lower, int upper)
{
  int64_t newPrime;

  do
  {
    newPrime = ( rand() % ( upper - lower) ) + lower;
  }
  while( !primalityTest( newPrime, PRIME_TESTS ) );

  return newPrime;
}

//---------------------------------------------------------------------------
//NAME: modularExpo()
//IMPORT: base (int64_t), exponent (int64_t), modulus (int64_t)
//EXPORT: result (int64_t)
//PURPOSE: Calculae the value base^exponent mod modulus efficiently

```

```
int64_t modularExpo(int64_t base, int64_t exponent, int64_t modulus)
{
    int64_t result = 1;
    base = base % modulus;

    //check upper limit
    if ( ( base > LIMIT ) || ( exponent > LIMIT ) || ( modulus > LIMIT ) )
        return -1;

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

    //loop until all exponents reviewed
    while( exponent > 0 )
    {
      //check least significant bit
      if( exponent & 1 )
        result = ( result * base ) % modulus;

      //shift exponent to consider next bit
      exponent >>= 1;
      base = ( base * base ) % modulus;
    }

    return result;
}

//---------------------------------------------------------------------------------
//NAME: extendedEuclid()
//IMPORT: a (int64_t), n (int64_t)
//EXPORT: t (int64_t)
//PURPOSE: Find the inverse modular a number via the extended euclidean algorithm

int64_t extendedEuclid( int64_t a, int64_t n )
{
    int64_t t = 0, newt = 1;
    int64_t r = n, newr = a;
  int64_t q = 0, temp = 0;

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

    //perform the actual 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 += n;

  return t;
}

//---------------------------------------------------------------------------------
//NAME: findGCD()
```

```c
//IMPORT: a (int64_t), b (int64_t)
//EXPORT: gcd (int64_t)
//PURPOSE: Find greatest common denominator of 2 numbers

int64_t findGCD( int64_t a, int64_t b )
{
  int64_t gcd, quotient, residue;

    //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 = findGCD( b, residue );

  return gcd;
}

//---------------------------------------------------------------------------
```

main.h

```c
/***************************************************************************
* FILE: main.h
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Header file for rsa
*   LAST MOD: 02/05/17
*   REQUIRES: numberTheory.h
***************************************************************************/

#include <time.h>
#include <string.h>
#include "numberTheory.h"

//FUNCTION POINTER
typedef int(*FuncPtr)(FILE*,FILE*);

//CONSTANTS
#define PLAIN_BYTES 2
#define CIPHER_BYTES 4

//PROTOTYPES
int64_t generateE(int64_t);
void generateKeys(void);
void printvals(void);
FuncPtr readArgs(int,char**);
char* readLine(FILE*);
int encrypt(FILE*,FILE*);
int decrypt(FILE*,FILE*);
void printKeys(void);

//GLOBALS
int64_t p, q, n, totN, e, d;
char *inFile, *outFile;

//---------------------------------------------------------------------------
```

main.c

```c
/***************************************************************************
* FILE: main.c
* AUTHOR: Connor Beardsmore - 15504319
* UNIT: FCC200
* PURPOSE: Main RSA implementation
*   LAST MOD: 02/05/17
*   REQUIRES: main.h
***************************************************************************/

#include "main.h"

//---------------------------------------------------------------------------

int main( int argc, char **argv )
{
    if ( argc > 7 )
    {
        printf( "USAGE: ./rsa <infile> <outfile> <mode> <keys>\n" );
        printf( "\tMODE: -e = encryption, -d = decryption\n" );
        printf( "\tKEYS: if mode = -d, supply values for d and n\n" );
        exit(1);
    }

    FuncPtr modeFunc = NULL;
    FILE* input = NULL;
    FILE* output = NULL;

  //seed random
    srand( time(NULL) );

    //generate keys on default
    generateKeys();

    //read command line arguments, ignoring the first
  modeFunc = readArgs( argc, argv );

    //open files
  input = fopen(inFile, "rb");
  if( input == NULL )
  {
    printf("CANNOT OPEN %s FOR FILE READING\n\n", inFile);
        exit(1);
  }
  output = fopen(outFile, "wbcat");
  if( output == NULL )
  {
    printf("Problem opening %s for writing\n\n", outFile);
        exit(1);
  }

    //perform actual encryption or decryption
  while( (*modeFunc)(input,output) != EOF );

  return 0;
}

//---------------------------------------------------------------------------
//NAME: encrypt()
//IMPORT: input (FILE*), output (FILE*)
//EXPORT: retVal (int)
//PURPOSE: Reads in two bytes, encrypts, and write back out 4 bytes

int encrypt(FILE* input, FILE* output)
```

```
{
  int c;
    int retVal = 0;
  int64_t plaintext = 0;
  int64_t ciphertext;

    //read in two characters
  for( int ii = 0; ii < PLAIN_BYTES; ii++ )
  {
    c = fgetc(input);
    if(c == EOF)
        {
            retVal = EOF;
            break;
        }
    else
      plaintext += c << ( 1 - ii ) * 8;
  }

    //skip over if there nothing read in
  if(plaintext != 0)
  {
        //calculate the actual ciphertext
    ciphertext = modularExpo(plaintext, e, n);

        //write back out 4 characters
    for( int ii = 0; ii < CIPHER_BYTES; ii++)
    {
      c = ciphertext >> ( 3 - ii ) * 8;
      fputc(c, output);
    }
  }

  //close files when done
  if(retVal == EOF)
  {
    fclose(input);
    fclose(output);
  }

  return retVal;
}

//---------------------------------------------------------------------------
//NAME: decrypt()
//IMPORT: input (FILE*), output (FILE*)
//EXPORT: retVal (int)
//PURPOSE: Reads in 4 bytes, decrypts, and write back out 2 bytes

int decrypt(FILE* input, FILE* output)
{
  int c;
  int64_t plaintext;
  int64_t ciphertext = 0;
  int retVal = 0;

  for(int ii = 0; ii < CIPHER_BYTES; ii++ )
  {
    c = fgetc(input);
    if( c == EOF )
    {
      retVal = EOF;
      break;
    }
    else
```

```
      ciphertext += c << ( 3 - ii ) * 8;
  }

    //skip over if nothing read in
  if(ciphertext != 0)
  {
        //calculate the actual plaintext
    plaintext = modularExpo(ciphertext, d, n);

        //write back out 2 bytes
    for( int ii = 0; ii < PLAIN_BYTES; ii++)
    {
      c = plaintext >> ( 1 - ii ) * 8;
      if( c != 0 )
        fputc( c, output );
    }
  }

  //close files when done
  if(retVal == EOF)
  {
    fclose(input);
    fclose(output);
  }

  return retVal;

}

//---------------------------------------------------------------------------
//NAME: readArgs()
//IMPORT: argc (int), argv (char**)
//PURPOSE: Read the command line arguments into global variables

FuncPtr readArgs( int argc, char **argv )
{
    FuncPtr modeFunc = NULL;
    //rename for readability
    inFile = argv[1];
    outFile = argv[2];
    char* mode = argv[3];

    //only if decryption is set
    if ( argc > 4 )
    {
        d = atoi( argv[4] );
        n = atoi( argv[5] );

        //check validity of keys
        if ( ( d == 0 ) || ( n == 0 ) )
        {
            printf("INVALID KEYS FOR DECRYPTION");
            exit(1);
        }
    }

    //set the correct mode
    if ( strcmp( mode, "-e" ) == 0 )
        modeFunc = &encrypt;
    else if ( strcmp( mode, "-d"  ) == 0 )
        modeFunc = &decrypt;
    else
    {
        printf( "INVALID MODE ARGUMENT\n" );
        exit(1);
```

```
    }

    return modeFunc;
}

//--------------------------------------------------------------------------
//NAME: generateKeys()
//PURPOSE: Generate key values for RSA, p, q, n, totN, e and d

void generateKeys(void)
{
    //generate two different prime numbers
  p = generatePrime( LOWER_PRIME, UPPER_PRIME );
  do
  {
    q = generatePrime( LOWER_PRIME, UPPER_PRIME );
  }
  while( p == q );

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

    //choose suitable e value
  e = generateE( totN );
    //determine modular inverse of e and totN, the d value
  d = extendedEuclid( e, totN );

    printKeys();
}

//--------------------------------------------------------------------------
//NAME: generateE()
//IMPORT: totN (int64_t)
//EXPORT: e (int64_t)
//PURPOSE: Determine suitable e value so that e and totN are coprime

int64_t generateE( int64_t totN )
{
  int64_t e;

    //repeat until the values of coprime
  do
  {
    e = rand() % totN;
  }
  while( findGCD( e, totN ) != 1 );

  return e;
}

//--------------------------------------------------------------------------
//NAME: printKeys()
//PURPOSE: Print all keys and variables required in RSA

void printKeys(void)
{
    printf("\tp = %lld\n\tq = %lld\n", p, q );
    printf("\tn = %lld\n\ttotN = %lld\n", n, totN );
    printf("\te = %lld\n\td = %lld\n", e, d );
}
//--------------------------------------------------------------------------
```

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.