

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).

```

1 //-----
2 //NAME: modularExpo()
3 //IMPORT: base (int64_t), exponent (int64_t), modulus (int64_t)
4 //EXPORT: result (int64_t)
5 //PURPOSE: Calculate the value base^exponent mod modulus efficiently
6
7 int64_t modularExpo(int64_t base, int64_t exponent, int64_t modulus)
8 {
9     int64_t result = 1;
10    base = base % modulus;
11
12    //check upper limit
13    if ( ( base > LIMIT ) || ( exponent > LIMIT ) || ( modulus > LIMIT ) )
14        return -1;
15
16    //anything mod 1 results in 0
17    if ( modulus == 1 )
18        return 0;
19
20    //loop until all exponents reviewed
21    while( exponent > 0 )
22    {
23        //check least significant bit
24        if( exponent & 1 )
25            result = ( result * base ) % modulus;
26
27        //shift exponent to consider next bit
28        exponent >>= 1;
29        base = ( base * base ) % modulus;
30    }
31
32    return result;
33 }
34
35 //-----

```

The code above was utilized to calculate the following example:

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

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


```

1  \subsection{AFS Algebras}
2  ###&&&
3  The Iris dataset is used as an illustrative example for AFS algebras through
4  this paper. It has 150 samples which are evenly distributed in three
5  classes and 4 features of sepal length( $f_1$ ), sepal
6  width( $f_2$ ), petal length( $f_3$ ), and petal width( $f_4$ ). Let a
7  pattern  $x=(x_1,x_2,x_3,x_4)$ , where  $x_i$  is the  $i$ th
8  feature value of  $x$ . The following three linguist fuzzy rules have been obtained for Class 1 to build the
9
257
258  \subsection{Shannon's Entropy}
259  Let  $X$  be a discrete random variable with a finite set containing  $N$  symbols
260   $x_0, x_1, \dots, x_N$ . If an output  $x_j$  occurs with probability  $p(x_j)$ , then the
261  amount of information associated with the known occurrence of the output  $x_j$  is defined as
262  \begin{equation}
263  I(x_j) = -\log_2 p(x_j)
264  \end{equation}
265  Based on this, the concept of Shannon's entropy is defined as follows:
266  ))))~~~~~

```

Figure 4: RSA Recovered Plaintext

Additional Questions

Signature Forgery

RSA can be utilized as a message signature scheme to provide authentication to messages. If Alice wants to send a signed message to Bob, she first produces a *hash value* of the message $H(m)$, then raises this to the power $d(modulo n)$ and attaches it to the message as a signature. When Bob receives the message, he utilizes the same hashing function, raises the result to the power of $e(modulo n)$ and compares to the message signature ($H(m) = H(m')$). If they are the same, Bob can be assured that the message was signed by Alice or someone with knowledge of Alice's private key.

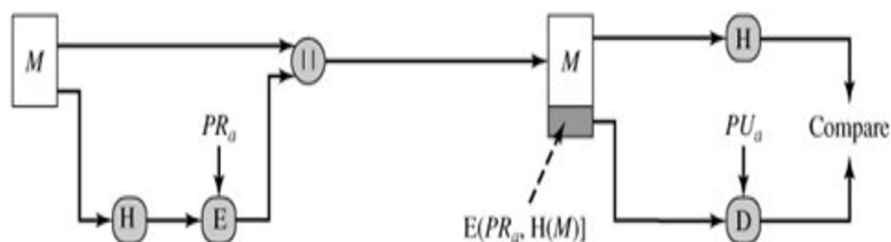


Figure 5: RSA Signature Scheme (Schneier 1996)

In this scheme, it is not possible to completely ensure the message was sent by Alice. If Bob has some alternate message m'' with a hash value $H(m'')$ matching that of Alice's $H(m)$ as discussed in Rivest, Shamir, and Adleman 1978, Bob can pretend to be Alice. He can simply resend Alice's signature he received onwards and the receiver of this message will believe that the message was sent by Alice. This can only occur if Bob finds a hash collision where $H(m) = H(m')$. It is unlikely for Bob to create a *meaningful* message with a hash value matching Alice's. This is analogous to the idea of Birthday attacks mentioned in Liu 2017b.

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 2017a) 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:

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

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

```
1 # Makefile For RSA Implementation
2 # FCC200 Assignment
3 # Last Modified: 02/05/17
4 # Connor Beardsmore - 15504319
5
6 # MAKE VARIABLES
7 CC=gcc
8 CFLAGS=-std=c99 -Wall -pedantic -g
9 EXEC=rsa
10 OBJ=main.o numberTheory.o
11 TESTS=output.txt original.txt
12
13 # RULES + DEPENDENCIES
14
15 $(EXEC): $(OBJ)
16     $(CC) -o $(EXEC) $(OBJ)
17
18 main.o: main.c main.h
19     $(CC) $(CFLAGS) -c -o main.o main.c
20
21 numberTheory.o: numberTheory.c numberTheory.h
22     $(CC) $(CFLAGS) -c -o numberTheory.o numberTheory.c
23
24 clean:
25     rm -r $(OBJ) $(EXEC) $(TESTS)
```


numberTheory.h

```
1  /*****
2  * FILE: numberTheory.h
3  * AUTHOR: Connor Beardsmore - 15504319
4  * UNIT: FCC200
5  * PURPOSE: Header file for number theory functionality
6  *   LAST MOD: 02/05/17
7  *   REQUIRES: stdio.h, stdlib.h
8  *****/
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 //CONSTANTS
13 #define PRIME_TESTS 25
14 #define LOWER_PRIME 1000
15 #define UPPER_PRIME 10000
16 #define LIMIT 10000000000
17
18 //BOOLEANS
19 #define TRUE 1
20 #define FALSE 0
21
22 //PROTOTYPES
23 int primalityTest(int64_t, int);
24 int64_t generatePrime(int, int);
25 int64_t modularExpo(int64_t, int64_t, int64_t);
26 int64_t extendedEuclid(int64_t, int64_t);
27 int64_t findGCD(int64_t, int64_t);
28
29 //
```

numberTheory.c

```

1  /*****
2  * FILE: numberTheory.c
3  * AUTHOR: Connor Beardsmore - 15504319
4  * UNIT: FCC200
5  * PURPOSE: Functionality for basic number theory techniques
6  *   LAST MOD: 02/05/17
7  *   REQUIRES: numberTheory.h
8  *****/
9
10 #include "numberTheory.h"
11
12 //-----
13 //NAME: primalityTest
14 //IMPORT: prime (int64_t), tests (int)
15 //EXPORT: isPrime (int)
16 //PURPOSE: Check if a number is prime or not to some confidence level
17
18 int primalityTest( int64_t prime, int tests )
19 {
20     int64_t a;
21     int64_t r;
22     int64_t exponent;
23     int isPrime = TRUE;
24
25     for( int ii = 0; ii < tests; ii++ )
26     {
27         //calculate r result
28         a = ( rand() % prime ) + 1;
29         exponent = ( prime - 1 ) >> 1;
30         r = modularExpo( a, exponent, prime );
31
32         // if r not 1 or -1 it is 100% not prime
33         if( ! ( (r == 1) || (r == (prime-1))) )
34             return FALSE;
35     }
36     return isPrime;
37 }
38
39 //-----
40 //NAME: generatePrime()
41 //EXPORT: newPrime (int64_t)
42 //PURPOSE: Generate a random prime number between the two bounds given
43
44 int64_t generatePrime( int lower, int upper)
45 {
46     int64_t newPrime;
47
48     do
49     {
50         newPrime = ( rand() % ( upper - lower ) ) + lower;
51     }
52     while( !primalityTest( newPrime, PRIME_TESTS ) );
53
54     return newPrime;
55 }
56
57 //-----
58 //NAME: modularExpo()
59 //IMPORT: base (int64_t), exponent (int64_t), modulus (int64_t)
60 //EXPORT: result (int64_t)
61 //PURPOSE: Calculate the value base^exponent mod modulus efficiently
62
63

```

```

64 int64_t modularExpo(int64_t base, int64_t exponent, int64_t modulus)
65 {
66     int64_t result = 1;
67     base = base % modulus;
68
69     //check upper limit
70     if ( ( base > LIMIT ) || ( exponent > LIMIT ) || ( modulus > LIMIT ) )
71         return -1;
72
73     //anything mod 1 results in 0
74     if ( modulus == 1 )
75         return 0;
76
77     //loop until all exponents reviewed
78     while( exponent > 0 )
79     {
80         //check least significant bit
81         if( exponent & 1 )
82             result = ( result * base ) % modulus;
83
84         //shift exponent to consider next bit
85         exponent >>= 1;
86         base = ( base * base ) % modulus;
87     }
88
89     return result;
90 }
91
92 //-----
93 //NAME: extendedEuclid()
94 //IMPORT: a (int64_t), n (int64_t)
95 //EXPORT: t (int64_t)
96 //PURPOSE: Find the inverse modular a number via the extended euclidean algorithm
97
98 int64_t extendedEuclid( int64_t a, int64_t n )
99 {
100     int64_t t = 0, newt = 1;
101     int64_t r = n, newr = a;
102     int64_t q = 0, temp = 0;
103
104     //only applicable if gcd is 1
105     if ( findGCD( a, n ) != 1 )
106         return -1;
107
108     //perform the actual eea
109     while( newr != 0 )
110     {
111         q = r / newr;
112         temp = t;
113         t = newt;
114         newt = temp - ( q * newt );
115         temp = r;
116         r = newr;
117         newr = temp - ( q * newr );
118     }
119
120     //ensure t is positive
121     if( t < 0 )
122         t += n;
123
124     return t;
125 }
126
127 //-----
128 //NAME: findGCD()

```

```
129 //IMPORT: a (int64_t), b (int64_t)
130 //EXPORT: gcd (int64_t)
131 //PURPOSE: Find greatest common denominator of 2 numbers
132
133 int64_t findGCD( int64_t a, int64_t b )
134 {
135     int64_t gcd, quotient, residue;
136
137     //check if either number is 0
138     if ( a == 0 )    return b;
139     if ( b == 0 )    return a;
140
141     //satisfy the equation A = B * quotient + residue
142     quotient = a / b;
143     residue = a - ( b * quotient );
144
145     //recursively call gcd
146     gcd = findGCD( b, residue );
147
148     return gcd;
149 }
150
151 //-----
```

main.h

```
1  /*****
2  * FILE: main.h
3  * AUTHOR: Connor Beardsmore - 15504319
4  * UNIT: FCC200
5  * PURPOSE: Header file for rsa
6  *   LAST MOD: 02/05/17
7  *   REQUIRES: numberTheory.h
8  *****/
9
10 #include <time.h>
11 #include <string.h>
12 #include "numberTheory.h"
13
14 //FUNCTION POINTER
15 typedef int (*FuncPtr)(FILE*,FILE*);
16
17 //CONSTANTS
18 #define PLAIN_BYTES 2
19 #define CIPHER_BYTES 4
20
21 //PROTOTYPES
22 int64_t generateE(int64_t);
23 void generateKeys(void);
24 void printvals(void);
25 FuncPtr readArgs(int, char**);
26 char* readLine(FILE*);
27 int encrypt(FILE*,FILE*);
28 int decrypt(FILE*,FILE*);
29 void printKeys(void);
30
31 //GLOBALS
32 int64_t p, q, n, totN, e, d;
33 char *inFile, *outFile;
34
35 //
```

main.c

```

1  /*****
2  * FILE: main.c
3  * AUTHOR: Connor Beardsmore - 15504319
4  * UNIT: FCC200
5  * PURPOSE: Main RSA implementation
6  *   LAST MOD: 02/05/17
7  *   REQUIRES: main.h
8  *****/
9
10 #include "main.h"
11
12 //-----
13
14 int main( int argc, char **argv )
15 {
16     if ( argc > 7 )
17     {
18         printf( "USAGE: ./rsa <infile> <outfile> <mode> <keys>\n" );
19         printf( "\tMODE: -e = encryption, -d = decryption\n" );
20         printf( "\tKEYS: if mode = -d, supply values for d and n\n" );
21         exit(1);
22     }
23
24     FuncPtr modeFunc = NULL;
25     FILE* input = NULL;
26     FILE* output = NULL;
27
28     //seed random
29     srand( time(NULL) );
30
31     //generate keys on default
32     generateKeys();
33
34     //read command line arguments, ignoring the first
35     modeFunc = readArgs( argc, argv );
36
37     //open files
38     input = fopen(inFile, "rb");
39     if( input == NULL )
40     {
41         printf("CANNOT OPEN %s FOR FILE READING\n\n", inFile);
42         exit(1);
43     }
44     output = fopen(outFile, "wbc");
45     if( output == NULL )
46     {
47         printf("Problem opening %s for writing\n\n", outFile);
48         exit(1);
49     }
50
51     //perform actual encryption or decryption
52     while( (*modeFunc)(input, output) != EOF );
53
54     return 0;
55 }
56
57 //-----
58 //NAME: encrypt()
59 //IMPORT: input (FILE*), output (FILE*)
60 //EXPORT: retVal (int)
61 //PURPOSE: Reads in two bytes, encrypts, and write back out 4 bytes
62
63 int encrypt(FILE* input, FILE* output)

```

```

64 {
65     int c;
66     int retVal = 0;
67     int64_t plaintext = 0;
68     int64_t ciphertext;
69
70     //read in two characters
71     for( int ii = 0; ii < PLAIN.BYTES; ii++ )
72     {
73         c = fgetc(input);
74         if(c == EOF)
75         {
76             retVal = EOF;
77             break;
78         }
79         else
80             plaintext += c << ( 1 - ii ) * 8;
81     }
82
83     //skip over if there nothing read in
84     if(plaintext != 0)
85     {
86         //calculate the actual ciphertext
87         ciphertext = modularExpo(plaintext, e, n);
88
89         //write back out 4 characters
90         for( int ii = 0; ii < CIPHER.BYTES; ii++ )
91         {
92             c = ciphertext >> ( 3 - ii ) * 8;
93             fputc(c, output);
94         }
95     }
96
97     //close files when done
98     if(retVal == EOF)
99     {
100         fclose(input);
101         fclose(output);
102     }
103
104     return retVal;
105 }
106
107 //-----
108 //NAME: decrypt()
109 //IMPORT: input (FILE*), output (FILE*)
110 //EXPORT: retVal (int)
111 //PURPOSE: Reads in 4 bytes, decrypts, and write back out 2 bytes
112
113 int decrypt(FILE* input, FILE* output)
114 {
115     int c;
116     int64_t plaintext;
117     int64_t ciphertext = 0;
118     int retVal = 0;
119
120     for(int ii = 0; ii < CIPHER.BYTES; ii++ )
121     {
122         c = fgetc(input);
123         if( c == EOF )
124         {
125             retVal = EOF;
126             break;
127         }
128         else

```

```

129     ciphertext += c << ( 3 - ii ) * 8;
130 }
131
132 //skip over if nothing read in
133 if(ciphertext != 0)
134 {
135     //calculate the actual plaintext
136     plaintext = modularExpo(ciphertext, d, n);
137
138     //write back out 2 bytes
139     for( int ii = 0; ii < PLAIN_BYTES; ii++)
140     {
141         c = plaintext >> ( 1 - ii ) * 8;
142         if( c != 0 )
143             fputc( c, output );
144     }
145 }
146
147 //close files when done
148 if(retVal == EOF)
149 {
150     fclose(input);
151     fclose(output);
152 }
153
154 return retVal;
155 }
156
157
158 //-----
159 //NAME: readArgs()
160 //IMPORT: argc (int), argv (char**)
161 //PURPOSE: Read the command line arguments into global variables
162
163 FuncPtr readArgs( int argc, char **argv )
164 {
165     FuncPtr modeFunc = NULL;
166     //rename for readability
167     inFile = argv[1];
168     outFile = argv[2];
169     char* mode = argv[3];
170
171     //only if decryption is set
172     if ( argc > 4 )
173     {
174         d = atoi( argv[4] );
175         n = atoi( argv[5] );
176
177         //check validity of keys
178         if ( ( d == 0 ) || ( n == 0 ) )
179         {
180             printf("INVALID KEYS FOR DECRYPTION");
181             exit(1);
182         }
183     }
184
185     //set the correct mode
186     if ( strcmp( mode, "-e" ) == 0 )
187         modeFunc = &encrypt;
188     else if ( strcmp( mode, "-d" ) == 0 )
189         modeFunc = &decrypt;
190     else
191     {
192         printf( "INVALID MODE ARGUMENT\n" );
193         exit(1);
194     }
195 }

```



```

194     }
195
196     return modeFunc;
197 }
198
199 //-----
200 //NAME: generateKeys()
201 //PURPOSE: Generate key values for RSA, p, q, n, totN, e and d
202
203 void generateKeys(void)
204 {
205     //generate two different prime numbers
206     p = generatePrime( LOWER_PRIME, UPPER_PRIME );
207     do
208     {
209         q = generatePrime( LOWER_PRIME, UPPER_PRIME );
210     }
211     while( p == q );
212
213     //calculate n and totN
214     n = p * q;
215     totN = ( p - 1 ) * ( q - 1 );
216
217     //choose suitable e value
218     e = generateE( totN );
219     //determine modular inverse of e and totN, the d value
220     d = extendedEuclid( e, totN );
221
222     printKeys();
223 }
224
225 //-----
226 //NAME: generateE()
227 //IMPORT: totN (int64_t)
228 //EXPORT: e (int64_t)
229 //PURPOSE: Determine suitable e value so that e and totN are coprime
230
231 int64_t generateE( int64_t totN )
232 {
233     int64_t e;
234
235     //repeat until the values of coprime
236     do
237     {
238         e = rand() % totN;
239     }
240     while( findGCD( e, totN ) != 1 );
241
242     return e;
243 }
244
245 //-----
246 //NAME: printKeys()
247 //PURPOSE: Print all keys and variables required in RSA
248
249 void printKeys(void)
250 {
251     printf("\tp = %lld\n\tq = %lld\n", p, q );
252     printf("\tn = %lld\n\ttotN = %lld\n", n, totN );
253     printf("\te = %lld\n\td = %lld\n", e, d );
254 }
255 //-----

```

References

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

— . 2017b. *Lecture 8: Birthday Attacks*. Curtin University.

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

Rivest, R. L., A. Shamir, and L. Adleman. 1978. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems”. *Commun. ACM* (New York, NY, USA) 21 (2): 120–126.

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.