

# 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).

```

1  //-----
2  //NAME: modularExpo()
3  //IMPORT: base (int64_t), exponent (int64_t), modulus (int64_t)
4  //EXPORT: result (int64_t)
5  //PURPOSE: Calculae 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:

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

## RSA Testing

```

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 1: RSA Plaintext

```

1  0E03Ehx00+0k0^0q{9h3'
2  E00000000000r0hi'0E0
3  0
4  Eh000E00Eh0
5  E0
6  000-{0E0'
7  0Âh0h0
8  f{hIj+'a^00
9  {9h3'
10 EĜ0'+09000000E0
11 h'\i000
12 EĚ00E
13 f{hE000x00
525 f+00+000+'f
526 00+00
527 EE+x0
528 0h0p0Ĝ0h0N0+00+xx0' 'h0xh0+0Ĝ0h0+~ā00k00000E0h000h0
529 E0003h900kh00
530 00+00000000k0000q0{+90ks00000k000000h0kh00
531 00+00000
532 Eh0+0Ĝ00E0Ĝ0h0x+0xh00+0000
533 00+0_E0h'+000E0h000h0
534 EIj+{'+E
535 000000000000

```

Figure 2: RSA Ciphertext

```

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 3: 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:

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

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 shared 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
12 # RULES + DEPENDENCIES
13 $(EXEC): $(OBJ)
14     $(CC) -o $(EXEC) $(OBJ)
15
16 main.o: main.c main.h
17     $(CC) $(CFLAGS) -c -o main.o main.c
18
19 numberTheory.o: numberTheory.c numberTheory.h
20     $(CC) $(CFLAGS) -c -o numberTheory.o numberTheory.c
21
22 clean:
23     rm -r $(OBJ) $(EXEC)
```

## 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, stdint.h, limits.h, math.h
8  *****/
9
10 #include <stdlib.h>
11 #include <stdint.h>
12 #include <stdio.h>
13 #include <limits.h>
14 #include <math.h>
15
16 //CONSTANTS
17 #define PRIME_TESTS 25
18 #define LOWER_PRIME 1000
19 #define UPPER_PRIME 10000
20 #define LIMIT 10000000000
21
22 //BOOLEANS
23 #define TRUE 1
24 #define FALSE 0
25
26 //PROTOTYPES
27 int primalityTest(int64_t, int);
28 int64_t generatePrime(int, int);
29 int64_t modularExpo(int64_t, int64_t, int64_t);
30 int64_t extendedEuclid(int64_t, int64_t);
31 int64_t findGCD(int64_t, int64_t);
32
33 //
```



## 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: 02/03/17
7  *   REQUIRES: numberTheory.h
8  *****/
9
10 #include "numberTheory.h"
11 #include <time.h>
12 #include <string.h>
13
14 //PROTOTYPES
15 int64_t generateE(int64_t);
16 void generateKeys(void);
17 void printvals(void);
18 void readArgs(int, char**);
19 void readFilepath(char**, char*);
20 char* readLine(FILE*);
21 void populate();
22 void readKeysFromFile(char*);
23 void help(void);
24 int encrypt(void);
25 int decrypt(void);
26 void printKeys(void);
27
28 //GLOBALS
29 int64_t p, q, n, totN, e, d;
30 int (*modeFunc)(void);
31 char *inFile, *outFile;
32 FILE* input, *output;
33
34 //
```

**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
25     input = NULL;
26     output = NULL;
27     modeFunc = NULL;
28     n = 0;
29
30     //seed random
31     srand( time(NULL) );
32
33     //generate keys on default
34     generateKeys();
35
36     //read command line arguments, ignoring the first
37     readArgs( argc, argv );
38
39     //open files
40     input = fopen(inFile, "rb");
41     if( input == NULL )
42     {
43         printf("CANNOT OPEN %s FOR FILE READING\n\n", inFile);
44         exit(1);
45     }
46     output = fopen(outFile, "wbcat");
47     if( output == NULL )
48     {
49         printf("Problem opening %s for writing\n\n", outFile);
50         exit(1);
51     }
52
53     //perform actual encryption or decryption
54     while( modeFunc() != EOF );
55
56     return 0;
57 }
58
59 //-----
60 //NAME: encrypt()
61 //EXPORT: retVal (int)
62 //PURPOSE: Reads in two bytes, encrypts, and write back out 4 bytes
63

```

```

64 int encrypt(void)
65 {
66     int c;
67     int retVal = 0;
68     int64_t plaintext = 0;
69     int64_t ciphertext;
70
71     //read in two characters
72     for( int ii = 0; ii < 2; ii++ )
73     {
74         c = fgetc(input);
75         if(c == EOF)
76         {
77             retVal = EOF;
78             break;
79         }
80         else
81             plaintext += c << ( 1 - ii ) * 8;
82     }
83
84     //skip over if there nothing read in
85     if(plaintext != 0)
86     {
87         //calculate the actual ciphertext
88         ciphertext = modularExpo(plaintext, e, n);
89
90         //write back out 4 characters
91         for( int ii = 0; ii < 4; ii++ )
92         {
93             c = ciphertext >> ( 3 - ii ) * 8;
94             fputc(c, output);
95         }
96     }
97
98     //close files when done
99     if(retVal == EOF)
100     {
101         fclose(input);
102         fclose(output);
103     }
104
105     return retVal;
106 }
107
108 //-----
109 //NAME: decrypt()
110 //EXPORT: retVal (int)
111 //PURPOSE: Reads in 4 bytes, decrypts, and write back out 2 bytes
112
113 int decrypt(void)
114 {
115     int c;
116     int64_t plaintext;
117     int64_t ciphertext = 0;
118     int retVal = 0;
119
120     for(int ii = 0; ii < 4; 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 < 2; 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 void readArgs( int argc, char **argv )
164 {
165     //rename for readability
166     inFile = argv[1];
167     outFile = argv[2];
168     char* mode = argv[3];
169
170     //only if decryption is set
171     if ( argc > 4 )
172     {
173         d = atoi( argv[4] );
174         n = atoi( argv[5] );
175
176         //check validity of keys
177         if ( ( d == 0 ) || ( n == 0 ) )
178         {
179             printf("INVALID KEYS FOR DECRYPTION");
180             exit(1);
181         }
182     }
183
184     //set the correct mode
185     if ( strcmp( mode, "-e" ) == 0 )
186         modeFunc = &encrypt;
187     else if ( strcmp( mode, "-d" ) == 0 )
188         modeFunc = &decrypt;
189     else
190     {
191         printf( "INVALID MODE ARGUMENT\n" );
192         exit(1);
193     }

```

```

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

```



## 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.