

1. Answers

- (a) I would see which key  $K_1$  matched the plaintext/ciphertext pair for  $X = E_{K_1}(P)$  and the two keys  $K_2, K_3$  that matched the plaintext/ciphertext pair for  $X = E_{K_2}(D_{K_3}(C))$ . The found keys would be the answer since they both align to the same result.
- (b)  $2^{112} + 2^{56}$  keys (2 keys + 1 key)
- (c)  $2^{56}$  space (each key is only 56 bits)

2. Answers

- (a) I am assuming that there is a private key to complement the public key. I am also assuming that the public key cipher with the public and private keys can be used for digital signatures. I also assume A and B have synchronized clocks.
- (b)  $E_{K_{RA}}(timestamp)$
- (c) B will decrypt the message with  $KU_A$  to get the timestamp and then check if the timestamp is recent enough to be the current virtual circuit.
- (d) BG will not have the private key of A, so when A signs the timestamp B can be assured that A is the sender when decrypting with  $KU_A$ . The timestamp functions as a way to ensure that the message is recent enough to be used and can't be replayed by BG.

3. Answers

- (a)

$$\begin{aligned}
 8^1 \pmod{11} &\equiv 8 \pmod{11} \\
 8^2 \pmod{11} &\equiv 8^1 \pmod{11} \times 8^1 \pmod{11} \equiv 9 \pmod{11} \\
 8^3 \pmod{11} &\equiv 8^2 \pmod{11} \times 8^1 \pmod{11} \equiv 6 \pmod{11} \\
 8^4 \pmod{11} &\equiv 8^3 \pmod{11} \times 8^1 \pmod{11} \equiv 4 \pmod{11} \\
 8^5 \pmod{11} &\equiv 8^4 \pmod{11} \times 8^1 \pmod{11} \equiv 10 \pmod{11} \\
 8^6 \pmod{11} &\equiv 8^5 \pmod{11} \times 8^1 \pmod{11} \equiv 3 \pmod{11} \\
 8^7 \pmod{11} &\equiv 8^6 \pmod{11} \times 8^1 \pmod{11} \equiv 2 \pmod{11} \\
 8^8 \pmod{11} &\equiv 8^7 \pmod{11} \times 8^1 \pmod{11} \equiv 5 \pmod{11} \\
 8^9 \pmod{11} &\equiv 8^8 \pmod{11} \times 8^1 \pmod{11} \equiv 7 \pmod{11} \\
 8^{10} \pmod{11} &\equiv 8^9 \pmod{11} \times 8^1 \pmod{11} \equiv 1 \pmod{11}
 \end{aligned}$$

There are no repeat modular values

$\therefore 8$  is a primitive root of 11.

- (b) Solve for  $X_A$  given  $Y_A = 6, q = 11, \alpha = 8$

$$6 = 8^{X_A} \pmod{11}$$

$$\begin{aligned}
 8^1 \pmod{11} &\equiv 8 \pmod{11} \\
 8^2 \pmod{11} &\equiv 8^1 \pmod{11} \times 8^1 \pmod{11} \equiv 9 \pmod{11} \\
 8^3 \pmod{11} &\equiv 8^2 \pmod{11} \times 8^1 \pmod{11} \equiv 6 \pmod{11}
 \end{aligned}$$

$$8^3 \pmod{11} \equiv 6 = Y_A$$

$$\therefore X_A = 3$$

- (c) Solve for  $X_B$  given  $Y_B = 3, q = 11, \alpha = 8$

$$3 = 8^{X_B} \pmod{11}$$

$$\begin{aligned}
 8^1 \pmod{11} &\equiv 8 \pmod{11} \\
 8^2 \pmod{11} &\equiv 8^1 \pmod{11} \times 8^1 \pmod{11} \equiv 9 \pmod{11} \\
 8^3 \pmod{11} &\equiv 8^2 \pmod{11} \times 8^1 \pmod{11} \equiv 6 \pmod{11} \\
 8^4 \pmod{11} &\equiv 8^3 \pmod{11} \times 8^1 \pmod{11} \equiv 4 \pmod{11} \\
 8^5 \pmod{11} &\equiv 8^4 \pmod{11} \times 8^1 \pmod{11} \equiv 10 \pmod{11} \\
 8^6 \pmod{11} &\equiv 8^5 \pmod{11} \times 8^1 \pmod{11} \equiv 3 \pmod{11}
 \end{aligned}$$

$$8^6 \pmod{11} \equiv 3 = Y_B$$

$$\therefore X_B = 6$$

(d) Given the equation  $K = Y_B^{X_A} \pmod{q}$  where  $Y_B = 3$ ,  $X_A = 3$ , and  $q = 11$

$$\therefore K = 3^3 \pmod{11} = 5$$

(e) Given the equation  $K = Y_A^{X_B} \pmod{q}$  where  $Y_A = 6$ ,  $X_B = 6$ , and  $q = 11$

$$\therefore K = 6^6 \pmod{11} = 5$$

#### 4. Answers

(a) Given the equation  $Y_A = \alpha^{X_A} \pmod{q}$ ,  $q = 11$ ,  $\alpha = 7$ , and  $X_A = 6$ :

$$Y_A = 7^6 \pmod{11} \equiv 4$$

$$\therefore Y_A = 4$$

(b) Given the equation  $K = Y_A^k \pmod{q}$ ,  $\alpha = 7$ ,  $q = 11$ ,  $k = 2$ ,  $M = 3$ , and  $Y_A = 4$ :

$$K = 4^2 \pmod{11} \equiv 5$$

$$\therefore K = 5$$

Using the equations  $C_1 = \alpha^k \pmod{q}$  and  $C_2 = KM \pmod{q}$ , it follows:

$$C_1 = 7^2 \pmod{11} \equiv 5$$

$$C_2 = 5 \times 3 \pmod{11} \equiv 4$$

$$\therefore C_1 = 5, C_2 = 4$$

(c) Given the equations  $K = C_1^{X_A} \pmod{q}$  and  $M = C_2 K^{-1} \pmod{q}$ ,  $C_1 = 5$ ,  $C_2 = 4$ , and  $X_A = 6$ , it follows:

$$K = 5^6 \pmod{11} \equiv 5$$

$$\therefore K = 5$$

$K^{-1}$  is calculated by testing all possible values:

$$5 \times 0 \equiv 0 \pmod{11}$$

$$5 \times 1 \equiv 5 \pmod{11}$$

$$5 \times 2 \equiv 10 \pmod{11}$$

$$5 \times 3 \equiv 4 \pmod{11}$$

$$5 \times 4 \equiv 9 \pmod{11}$$

$$5 \times 5 \equiv 3 \pmod{11}$$

$$5 \times 6 \equiv 8 \pmod{11}$$

$$5 \times 7 \equiv 2 \pmod{11}$$

$$5 \times 8 \equiv 7 \pmod{11}$$

$$5 \times 9 \equiv 1 \pmod{11}$$

$$\therefore K^{-1} = 9$$

From this,  $M$  can be calculated as following:

$$M = 4 \times 9 \pmod{11} \equiv 3$$

$\therefore M$  is calculated to be 3, and A has successfully recovered the message.

#### 5. Answers

- (a) The BG could take the public key of B and replace it with their public key and send that to A.
- (b) BG could take an old transmission from round 2 and send that to A instead of the legitimate transmission. The old transmission might have an old public key for B which the BG could use as their new public key.
- (c) In round 6, BG could send a nonce  $N_2$  to A and A would have no way of confirming it came from B. The nonce in round 3 is only known to A and B, so without it A cannot confirm B's identity.
- (d) BG could request their public key from the authority and send it to A in round 2. A would be forced to assume that this is the response they wanted when in reality it would be the BG's public key.

# Programming Problem

## Self-critique

The program works as expected and its answers are verifiably correct. The only issue is the chosen plaintext for the third set of attacks. Since the plaintext is only 67 and not something higher, the first brute force attack is significantly faster than the second factoring attack. I do not believe that the first attack is faster than the second attack unless the best case occurs where  $M$  is close to 1, as seen in the program.

## Program Output

```
hoodz@hoodz-laptop:/mnt/c/Users/hoodi/Desktop/Coding/CSCI-4534-Cryptography/Homework
-2$ ./main
-----
~ RSA Brute Force Attacker Program ~
-----
Running Problem 1...
-----
=====
Ciphertext: 10
Public Key: 7
Number: 15
=====
Running Attack One...
Plaintext: 10
Time to complete AttackOne: 500 nanoseconds

Running Attack Two...
Prime P: 5
Prime Q: 3
Private Key: 7
Plaintext: 10
Time to complete AttackTwo: 300 nanoseconds

-----
Running Problem 2...
-----
=====
Ciphertext: 356
Public Key: 13
Number: 527
=====
Running Attack One...
Plaintext: 271
Time to complete AttackOne: 8700 nanoseconds

Running Attack Two...
Prime P: 31
Prime Q: 17
Private Key: 37
Plaintext: 271
Time to complete AttackTwo: 1600 nanoseconds

-----
Running Problem 3...
-----
=====
Ciphertext: 8567
Public Key: 53
Number: 21583
=====
Running Attack One...
Plaintext: 67
Time to complete AttackOne: 3400 nanoseconds

Running Attack Two...
Prime P: 191
Prime Q: 113
Private Key: 20477
```

Plaintext: 67

Time to complete AttackTwo: 53000 nanoseconds

-----  
~ End of Program ~

hoodz@hoodz-laptop:/mnt/c/Users/hoodi/Desktop/Coding/CSCI-4534-Cryptography/Homework-2\$

## Human-readable code

### main.cpp

```
1  #include <chrono>
2  #include <iostream>
3  #include <iomanip>
4
5  #include "RSAAttacker.hpp"
6
7  using namespace std;
8
9  /// function for testing homework problems
10 void Problem(RSAAttacker& attack, Members info);
11
12 int main(){
13     cout << "-----" << endl;
14     cout << "~ RSA Brute Force Attacker Program ~" << endl;
15
16     // Blank initialization
17     RSAAttacker attack(0, 0, 0);
18
19     // formatted as public key, number, ciphertext
20     // private key = 7, plaintext = 10, primes = 3 & 5, totient = 8
21     Members q1 { 7, 15, 10 };
22
23     // private key = 37, plaintext = 271, primes = 17 & 31, totient = 480
24     Members q2 { 13, 527, 356 };
25
26     // private key = 20477 11143, plaintext = 67, primes = 113 & 191, totient = 21280
27     Members q3 { 53, 21583, 8567 };
28
29     cout << "-----" << endl;
30     cout << "Running Problem 1..." << endl;
31     cout << "-----" << endl;
32
33     Problem(attack, q1);
34
35     cout << "-----" << endl;
36     cout << "Running Problem 2..." << endl;
37     cout << "-----" << endl;
38
39     Problem(attack, q2);
40
41     cout << "-----" << endl;
42     cout << "Running Problem 3..." << endl;
43     cout << "-----" << endl;
44
45     Problem(attack, q3);
46
47     cout << "-----" << endl;
48     cout << "~ End of Program ~" << endl;
49
50     return 0;
51 }
52
53 /// @brief      a helper function to run both RSAAttacker functions on one problem set
54 /// @param attack (RSAAttacker) any object that will get its values changed with info param
55 /// @param info   (struct Members) the specified public key, number, and ciphertext xstruct
56 void Problem(RSAAttacker& attack, Members info){
57     // read in the information into the attacker
58     attack.SetInfo(info);
59
60     cout << "=====" << endl;
61     cout << "Ciphertext: " << info.ciphertext << endl;
62     cout << "Public Key: " << info.encryptionKey << endl;
63     cout << "Number: " << info.number << endl;
64     cout << "=====" << endl;
65
66
67     cout << "Running Attack One..." << endl;
68     const auto start1 = chrono::high_resolution_clock::now();
69     const int plaintext = attack.AttackOne();
70     const auto end1 = chrono::high_resolution_clock::now();
71     cout << "Plaintext: " << plaintext << endl;
```

```

72
73     const std::chrono::duration<double> time1 = end1 - start1;
74     const auto timeCast1 = chrono::duration_cast<chrono::nanoseconds>(end1 - start1);
75
76     cout << "Time to complete AttackOne: " << timeCast1.count() << " nanoseconds\n" << endl;
77
78     cout << "Running Attack Two..." << endl;
79     const auto start2 = chrono::high_resolution_clock::now();
80     const attackTwoReturn vals = attack.AttackTwo();
81     const auto end2 = chrono::high_resolution_clock::now();
82
83     cout << "Prime P:      " << vals.primeP << endl;
84     cout << "Prime Q:      " << vals.primeQ << endl;
85     cout << "Private Key: " << vals.privKey << endl;
86     cout << "Plaintext:   " << vals.plaintext << endl;
87
88     const auto timeCast2 = chrono::duration_cast<chrono::nanoseconds>(end2 - start2);
89
90     cout << "Time to complete AttackTwo: " << timeCast2.count() << " nanoseconds\n" << endl;
91 }

```

## RSAAttacker.hpp

```

1  #ifndef RSA_ATTACKER_HPP
2  #define RSA_ATTACKER_HPP
3
4  /// @brief internal data members of RSAAttacker class
5  struct Members {
6      int encryptionKey; // public key of RSA
7      int number;        // number N of RSA
8      int ciphertext;     // ciphertext from  $m^e \text{ mod } N$ 
9  };
10
11 /// @brief return value for AttackTwo to aggregate data and return one item
12 struct attackTwoReturn {
13     int primeP; // calculated primes from RSA
14     int primeQ;
15     int privKey; // private key/decryption key associated with RSA encryption key
16     int plaintext; // calculated message m where  $m = c^{dKey} \text{ mod } N$ 
17 };
18
19
20 class RSAAttacker{
21 private:
22     Members info; // aggregated data members, see Members struct
23
24 public:
25     /// -----
26     /// @brief      main constructor for RSAAttacker helper class
27     /// @param eKey   (int) encryption key/private key of RSA
28     /// @param number (int) calculated number N of RSA
29     /// @param ciphertext (int) valid ciphertext of RSA derived from  $m^e \text{ mod } N$  (m is unknown to
30     ///               class)
31     /// @note         this class does not check for validness of inputs
32     /// -----
33     RSAAttacker(int eKey, int number, int ciphertext);
34
35     /// -----
36     /// @brief      constructor for RSAAttacker helper class that allows assignment using Members struct
37     /// @param info  (struct Members) values needed for attacker, see RSAAttacker(int, int, int)
38     /// @note         this class does not check for validness of inputs
39     /// -----
40     RSAAttacker(Members info);
41
42     ~RSAAttacker() = default;
43
44     /// -----
45     /// @brief      setter to change out internal information
46     /// @param info  (struct Members) struct values to change to
47     /// -----
48     void SetInfo(Members info) { this->info = info; }
49
50     /// -----
51     /// @brief      calculates the Euler totient of two prime numbers

```

```

51  /// @param p    (int) a prime number
52  /// @param q    (int) a prime number
53  /// @return     (int) totient of primes p and q
54  /// -----
55  int eulerTotient(int p, int q) const { return (p-1)*(q-1); }
56
57  /// -----
58  /// @brief run through all possible values of m until [m ^ eKey mod number = ciphertext] is found
59  /// @return (int) plaintext associated with the inputted ciphertext from info
60  /// -----
61  int AttackOne();
62
63  /// -----
64  /// @brief factor inputted info.number into its two primes and get plaintext by calculating private key
65  /// @return (struct attackTwoReturn) following calculated values via RSA algorithm: { (int) prime P, (int)
66  /// prime Q, (int) private key, (int) plaintext }
67  /// -----
68  attackTwoReturn AttackTwo();
69
70  /// -----
71  /// @brief solve a ^ b mod n fast using less multiplications
72  /// @param a    (int) positive base of the equation
73  /// @param b    (int) positive exponent of the equation
74  /// @param n    (int) modulus value of the equation
75  /// @return     (int) result of a ^ b mod n
76  /// -----
77  int fastModExponentiation(int a, int b, int n) const;
78
79  /// -----
80  /// @brief helper function to calculate modular inverse, ax = 1 mod n
81  /// @param a    (int) positive known term in the above equation
82  /// @param n    (int) modulus value of the equation
83  /// @return     (int) modular inverse of a mod n
84  /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
85  /// -----
86  int modInverse(int a, int n) const;
87
88  /// -----
89  /// @brief uses the extended Euclidean algorithm to find the solution to ax = b mod n
90  /// @param a    (int) positive known term in the above equation
91  /// @param b    (int) positive congruent term in the above equation
92  /// @param n    (int) modulus value of the equation
93  /// @return     (int) the solution x to the equation ax = b mod n
94  /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
95  /// -----
96  int modLinearEquationSolver(int a, int b, int n) const;
97
98  /// -----
99  /// @brief the extended Euclidean algorithm to find the greatest common denominator
100  /// @note remainder = ax + by = GCD(a, b)
101  /// @param a    (int) the first term to find the GCD with
102  /// @param b    (int) the second term to find the GCD with
103  /// @param x    (int*) return parameter to return Bezout coefficient x
104  /// @param y    (int*) return parameter to return Bezout coefficient y
105  /// @return     (int) the GCD of a & b, as well as Bezout coefficients in x & y
106  /// -----
107  int gcdExtended(int a, int b, int* x, int* y) const;
108 };
109 #endif

```

## RSAAttacker.cpp

```

1  #include "RSAAttacker.hpp"
2
3  /// -----
4  /// @brief main constructor for RSAAttacker helper class
5  /// @param eKey    (int) encryption key/private key of RSA
6  /// @param number  (int) calculated number N of RSA
7  /// @param ciphertext (int) valid ciphertext of RSA derived from m ^ eKey mod number (m is unknown to class)
8  /// @note          this class does not check for validness of inputs
9  /// -----
10 RSAAttacker::

```



```

11 RSAAttacker(int eKey, int number, int ciphertext) {
12     info.encryptionKey = eKey;
13     info.number = number;
14     info.ciphertext = ciphertext;
15 }
16
17 /// -----
18 /// @brief      constructor for RSAAttacker helper class that allows assignment using Members struct
19 /// @param info  (struct Members) values needed for attacker, see RSAAttacker(int, int, int)
20 /// @note        this class does not check for validness of inputs
21 /// -----
22 RSAAttacker::
23 RSAAttacker(Members info) : info(info) {}
24
25 /// -----
26 /// @brief      run through all possible values of m until [m ^ eKey mod number = ciphertext] is found
27 /// @return (int) plaintext associated with the inputted ciphertext from info
28 /// -----
29 int RSAAttacker::
30 AttackOne(){
31     int guess = 0, m = 0;
32
33     while (guess != info.ciphertext)
34         guess = fastModExponentiation(++m, info.encryptionKey, info.number);
35
36     return m;
37 }
38
39 /// -----
40 /// @brief      factor inputted info.number into its two primes and get plaintext by calculating private key from
41 ///              them
42 /// @return (struct attackTwoReturn) following calculated values via RSA algorithm: { (int) prime P, (int)
43 ///              prime Q, (int) private key, (int) plaintext }
44 /// -----
45 attackTwoReturn RSAAttacker::
46 AttackTwo(){
47     // calculate the two prime factors (naively)
48     int p, q;
49     for (int i = 1; i < info.number; i++){
50         if (info.number % i == 0){
51             p = i; q = info.number / p;
52         }
53     }
54     // calculate the totient of the primes
55     int totient = eulerTotient(p, q);
56
57     // calculate the private key by getting the inverse of eKey mod totient
58     int privateKey = modInverse(info.encryptionKey, totient);
59
60     // calculate plaintext from c ^ dKey mod number
61     int m = fastModExponentiation(info.ciphertext, privateKey, info.number);
62
63     return attackTwoReturn { p, q, privateKey, m };
64 }
65
66 /// -----
67 /// @brief      solve a ^ b mod n fast using less multiplications
68 /// @param a     (int) positive base of the equation
69 /// @param b     (int) positive exponent of the equation
70 /// @param n     (int) modulus value of the equation
71 /// @return      (int) result of a ^ b mod n
72 /// @note        relies on the idea that modding smaller parts gets same result
73 /// @example     a^5 mod n = a^2 mod n * a^2 mod n * a mod n
74 /// -----
75 int RSAAttacker::
76 fastModExponentiation(int a, int b, int n) const {
77     int result = 1;
78     a = a % n;
79
80     while (b > 0){
81         if (b & 1) result = (result * a) % n; // if b is odd, set the new result and mod it by n (result
82         * [a mod n])
83         b >>= 1; // another a^2 mod n is added, divide exponent b by 2
84         a = (a * a) % n; // a = a^2 mod n
85     }
86 }

```

```

84     return result;
85 }
86
87 /// -----
88 /// @brief      helper function to calculate modular inverse,  $ax = 1 \pmod n$ 
89 /// @param a    (int) positive known term in the above equation
90 /// @param n    (int) modulus value of the equation
91 /// @return     (int) modular inverse of a mod n
92 /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
93 /// -----
94 int RSAAttacker::
95 modInverse(int a, int n) const {
96     return modLinearEquationSolver(a, 1, n);
97 }
98
99 /// -----
100 /// @brief      uses the extended Euclidean algorithm to find the solution to  $ax = b \pmod n$ 
101 /// @param a    (int) positive known term in the above equation
102 /// @param b    (int) positive congruent term in the above equation
103 /// @param n    (int) modulus value of the equation
104 /// @return     (int) the solution x to the equation  $ax = b \pmod n$ 
105 /// @throw      if the modular inverse cannot be found, a const char* message notification is thrown
106 /// -----
107 int RSAAttacker::
108 modLinearEquationSolver(int a, int b, int n) const {
109     int x,          // first Bezout coefficient from gcdExtended
110     y;             // second Bezout coefficient from gcdExtended
111
112     int gcd = gcdExtended(a, n, &x, &y);    // get the greatest common denominator and the Bezout coefficients
113     if (b % gcd == 0){
114         return ((x % n + n) % n);          // make sure solution x is within modulus range and return it
115     }
116
117     throw "Modular Linear Equation Solver cannot return a value";
118 }
119
120 /// -----
121 /// @brief      the extended Euclidean algorithm to find the greatest common denominator
122 /// @note       remainder =  $ax + by = \text{GCD}(a, b)$ 
123 /// @param a    (int) the first term to find the GCD with
124 /// @param b    (int) the second term to find the GCD with
125 /// @param x    (int*) return parameter to return Bezout coefficient x
126 /// @param y    (int*) return parameter to return Bezout coefficient y
127 /// @return     (int) the GCD of a & b, as well as Bezout coefficients in x & y
128 /// @cite       https://en.wikipedia.org/wiki/Extended\_Euclidean\_algorithm#Pseudocode
129 /// @date       4 Mar 2024
130 /// -----
131 int RSAAttacker::
132 gcdExtended(int a, int b, int* x, int* y) const{
133     int remainder    = a, newRemainder    = b;
134     int bezoutX      = 1, bezoutXCalc     = 0;
135     int bezoutY      = 0, bezoutYCalc     = 1;
136     int quotient     = 0, temp            = 0;
137
138     while (newRemainder != 0){                // while there is something to divide
139         quotient = remainder / newRemainder;    // get the quotient result
140         temp = newRemainder;
141         newRemainder = remainder - (quotient * newRemainder); // calculate next remainder
142         remainder = temp;
143
144         temp = bezoutXCalc;
145         bezoutXCalc = bezoutX - (quotient * bezoutXCalc); // calculate next x Bezout coefficient
146         bezoutX = temp;
147
148         temp = bezoutYCalc;
149         bezoutYCalc = bezoutY - (quotient * bezoutYCalc); // calculate next y Bezout coefficient
150         bezoutY = temp;
151     }
152
153     // cout << "Quotients of GCD: (" << bezoutYCalc << ", " << bezoutXCalc << ")" << endl;
154     // cout << "Final GCD: " << remainder << endl;
155     // cout << "Bezout coefficients: (" << bezoutX << ", " << bezoutY << ")" << endl;
156
157     *x = bezoutX; *y = bezoutY;                // assign return parameters and return GCD
158     return remainder;
159 }

```

---