Principles Of Cryptography Programming Assignment

Group-11

Hemakshi Janyani – 106117031

Riddhi Gupta – 106117077

Vidhyalakshmi B -106117107

CSE-A

**11a)**

Alice uses Bob’s RSA public key (e=17, n=19519) to send a four-character message to Bob using the (A-0, B-1, …., Z-25) encoding scheme and encrypting each character separately. Eve intercepts the ciphertext (6625 0 2968 17863) and decrypts the message without factoring the modulus. Find the plaintext and explain why Eve could easily break the cipher text.

**Solution:**

**About RSA Algorithm**

### **Generate the RSA modulus**

The initial procedure begins with selection of two prime numbers namely p and q, and then calculating their product N, as shown −

N=p\*q

Here, let N be the specified large number.

### **Derived Number (e)**

Consider number e as a derived number which should be greater than 1 and less than (p-1) and (q-1). The primary condition will be that there should be no common factor of (p-1) and (q-1) except 1

### **Public key**

The specified pair of numbers **n** and **e** forms the RSA public key and it is made public.

### **Private Key**

Private Key **d** is calculated from the numbers p, q and e. The mathematical relationship between the numbers is as follows −

ed = 1 mod (p-1) (q-1)

The above formula is the basic formula for Extended Euclidean Algorithm, which takes p and q as the input parameters.

## Encryption Formula

C = Pe mod n

## Decryption Formula

Plaintext = Cd mod n

**Attacking the RSA Algorithm**

We are not using factorization since it is a costly operation. The time complexity is high.

Since the public exponent(e) is small (e=17). We use a low public exponent attack given below.

**Franklin-Reiter Related messages Attack**

This attack works in a scenario where two messages differ only by a fixed known difference and are encrypted using public key e and same modulus N. The attacker can then recover the two messages in the above scenario using Franklin Reiter's Attack.

**Theorem**

Suppose there are two messages M1 and M2 where M1!= M2, both less than N and related to each other as [equation](https://github.com/ashutosh1206/Crypton/blob/master/RSA-encryption/Attack-Franklin-Reiter/Pictures/1.gif) for some linear polynomial [equation](https://github.com/ashutosh1206/Crypton/blob/master/RSA-encryption/Attack-Franklin-Reiter/Pictures/2.gif) where b!=0. These two messages are to be sent by encrypting using the public key (N, e), thus giving ciphertexts C1 and C2 respectively. Then, given (N, e, C1, C2, f), the attacker can recover messages M1 and M2.

**Proof**

We can write C1 and C2 as:  
[equation](https://github.com/ashutosh1206/Crypton/blob/master/RSA-encryption/Attack-Franklin-Reiter/Pictures/3.gif)  
We can also write,  
[equation](https://github.com/ashutosh1206/Crypton/blob/master/RSA-encryption/Attack-Franklin-Reiter/Pictures/4.gif)  
[equation](https://github.com/ashutosh1206/Crypton/blob/master/RSA-encryption/Attack-Franklin-Reiter/Pictures/5.gif)  
We can then write the polynomials g1(x) and g2(x) as:  
[equation](https://github.com/ashutosh1206/Crypton/blob/master/RSA-encryption/Attack-Franklin-Reiter/Pictures/6.gif)  
[equation](https://github.com/ashutosh1206/Crypton/blob/master/RSA-encryption/Attack-Franklin-Reiter/Pictures/7.gif)  
So clearly M2 is a root of both the polynomials above and hence they have a common factor **x-M2** (Since, g1(M2) = 0 and g2(M2) = 0) Therefore, we can simply calculate GCD of g1 and g2 and if the resultant polynomial is linear, then we get out M2 and hence M1!

Exploit in a nutshell:

1. Calculate g1 and g2 as given above

2. Calculate GCD(g1, g2) and check if the resultant polynomial is linear or not

3. If the resultant polynomial is linear, then return GCD

**Code:**

#Franklin-Reiter Algorithm

from sage.all import \*

# All the variable names mean the same as mentioned in the explanation

# For eg, a,b are the values in the function f = ax + b

def gcd(a, b):

    while b:

        a, b = b, a % b

    return a.monic()

def franklinreiter(C1, C2, e, N, a, b):

    P.<X> = PolynomialRing(Zmod(N))

    g1 = (a\*X + b)^e - C1

    g2 = X^e - C2

    #print ("Result")

    result = -gcd(g1, g2).coefficients()[0]

    return hex(int(result))[2:].replace("L","") #.decode("hex")

#Factorisation for verification

import string

def p\_and\_q(n):

   data = []

   for i in range(2, n):

      if n % i == 0:

         data.append(i)

   return tuple(data)

def euler(p, q):

   return (p - 1) \* (q - 1)

def private\_index(e, euler\_v):

   for i in range(2, euler\_v):

      if i \* e % euler\_v == 1:

         return i

def decipher(d, n, c):

   return c \*\* d % n

def main():

      #creating hash-map

      keys=dict(zip(range(0,26),string.ascii\_lowercase))

      e = int(input("input e: "))

      n = int(input("input n: "))

      c1 = int(input("input c1: "))

      c2 = int(input("input c2: "))

      c3 = int(input("input c3: "))

      c4 = int(input("input c4: "))

      a=1

      b1=14

      b2=-6

      #franklin-reiter

      res=franklinreiter(c3,c1,e,n,a,b1)

      print("Franklin-Reiter Solution")

      res=int(res)

      print(keys[res])

      print("a")  #since we can directly say by O=O^e(modn)and 'a'=0

      print(keys[res+14])

      res=franklinreiter(c3,c4,e,n,a,b2)

      res=int(res)

      print(keys[res+6])

      #factorisation

      p\_and\_q\_v = p\_and\_q(n)

      # print("[p\_and\_q]: ", p\_and\_q\_v)

      euler\_v = euler(p\_and\_q\_v[0], p\_and\_q\_v[1])

      print("Verification by Factorisation")

      # print("[euler]: ", euler\_v)

      d = private\_index(e, euler\_v)

      plain1 = decipher(d, n, c1)

      plain2 = decipher(d, n, c2)

      plain3 = decipher(d, n, c3)

      plain4 = decipher(d, n, c4)

      print("plain1: ",keys[plain1])

      print("plain2: ",keys[plain2])

      print("plain3: ",keys[plain3])

      print("plain4: ",keys[plain4])

if \_\_name\_\_ == "\_\_main\_\_":

   main()

**Output:**

input e: 

input n: 

input c1: 

input c2: 

input c3: 

input c4: 

Franklin-Reiter Solution e a s y

Verification by Factorisation

plain1: e

plain2: a

plain3: s

plain4: y

**11b)** Using the Rabin cryptosystem with p=47 and q=11:

1. Encrypt P=17 to find the cipher text.
2. Use the Chinese Remainder theorem to find four possible plaintexts

**Solution:**

p=47 (47 ≅ 3mod4)

q=11(11≅3mod4)

public key = n=p\*q

=47\*11

=517

Plain Text = P =17

Cipher Text = C

In Rabin Cryptosystem,

**Encryption**

e=2

C ≅Pemodn

≅P2modn

Therefore,

C≅172mod517

C≅289mod517

**Cipher Text = 289**

**Decryption**

P≅modn

* There are four square roots of c modulo n.
* The message m is equal to one of these four messages

When p≅3mod4 there is a simple formula to compute the square root of C in mod p

(C(p+1)/4)2≅Cmodp

Hence the two square roots of C mod p are

C(p+1)/4modp

In a similar fashion, the two square roots of c mod q are

C(q+1)/4modq

Then we can obtain the four-square roots of c mod n using the Chinese Remainder Theorem

In our case

± C(p+1)/4modp = 28912mod47

=±17mod47

±C(q+1)/4modq=2893mod11 = ±5mod11

By applying **Chinese Remainder Theorem** on the above

We get,

**m1=17**

**m2=500**

**m3=171**

**m4=346**

**CODE-**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <math.h>

#include <limits.h>

int e = 2;

int p=11, q=47;

int n=517;

int gcd(int a, int b)

{

int q, r1, r2, r;

if (a > b) {

        r1 = a;

        r2 = b;

    }

    else {

        r1 = b;

        r2 = a;

    }

    while (r2 > 0) {

        q = r1 / r2;

        r = r1 - q \* r2;

        r1 = r2;

        r2 = r;

    }

    return r1;

}

int PrimarityTest(int a, int i)

{

    int n = i - 1;

    int k = 0;

    int j, m, T;

    while (n % 2 == 0) {

        k++;

        n = n / 2;

    }

    m = n;

    T = FindT(a, m, i);

    if (T == 1 || T == i - 1)

        return 1;

    for (j = 0; j < k; j++)

    {

        T = FindT(T, 2, i);

        if (T == 1)

            return 0;

        if (T == i - 1)

            return 1;

    }

    return 0;

}

int FindT(int a, int m, int n)

{

    int r;

    int y = 1;

    while (m > 0) {

        r = m % 2;

        FastExponention(r, n, &y, &a);

        m = m / 2;

    }

    return y;

}

int FastExponention(int bit, int n, int\* y, int\* a)

{

    if (bit == 1)

        \*y = (\*y \* (\*a)) % n;

    \*a = (\*a) \* (\*a) % n;

}

int Encryption(int PlainText)

{

    printf("PlainText is %d\n", PlainText);

    int cipher = FindT(PlainText, e, n);

    fprintf(stdout, "Cipher Text is %d\n", cipher);

    return cipher;

}

int inverse(int a, int b)

{

    int inv;

    int q, r, r1 = a, r2 = b, t, t1 = 0, t2 = 1;

    while (r2 > 0) {

        q = r1 / r2;

        r = r1 - q \* r2;

        r1 = r2;

        r2 = r;

        t = t1 - q \* t2;

        t1 = t2;

        t2 = t;

    }

    if (r1 == 1)

        inv = t1;

    if (inv < 0)

        inv = inv + a;

    return inv;

}

int Decryption(int Cipher)

{

    int P1, P2, P3, P4;

    int a1, a2, b1, b2;

    a1 = FindT(Cipher, (p + 1) / 4, p);

    a2 = p - a1;

    b1 = FindT(Cipher, (q + 1) / 4, q);

    b2 = q - b1;

    P1 = ChineseRemainderTheorem(a1, b1, p, q);

    P2 = ChineseRemainderTheorem(a1, b2, p, q);

    P3 = ChineseRemainderTheorem(a2, b1, p, q);

    P4 = ChineseRemainderTheorem(a2, b2, p, q);

    printf("\nP1 = %d\nP2 = %d\nP3 = %d\nP4 = %d\n", P1, P2, P3, P4);

}

int ChineseRemainderTheorem(int a, int b, int m1, int m2)

{

    int M, M1, M2, M1\_inv, M2\_inv;

    int result;

    M = m1 \* m2;

    M1 = M / m1;

    M2 = M / m2;

    M1\_inv = inverse(m1, M1);

    M2\_inv = inverse(m2, M2);

    result = (a \* M1 \* M1\_inv + b \* M2 \* M2\_inv) % M;

    return result;

}

int main(void)

{

    int PlainText =17;

    int Cipher = Encryption(PlainText);

    Decryption(Cipher);

    return 0;

}

**OUTPUT-**

PlainText is 17                                                                Cipher Text is 289

P1 = 346                                                                       P2 = 500

P3 = 17

P4 = 171

**...Program finished with exit code 0**

**Press ENTER to exit console.**