Date : 23/04/2022

CSPC63: **Principles of Cryptography**

Assignment - 3

Roll no. : **106119100**

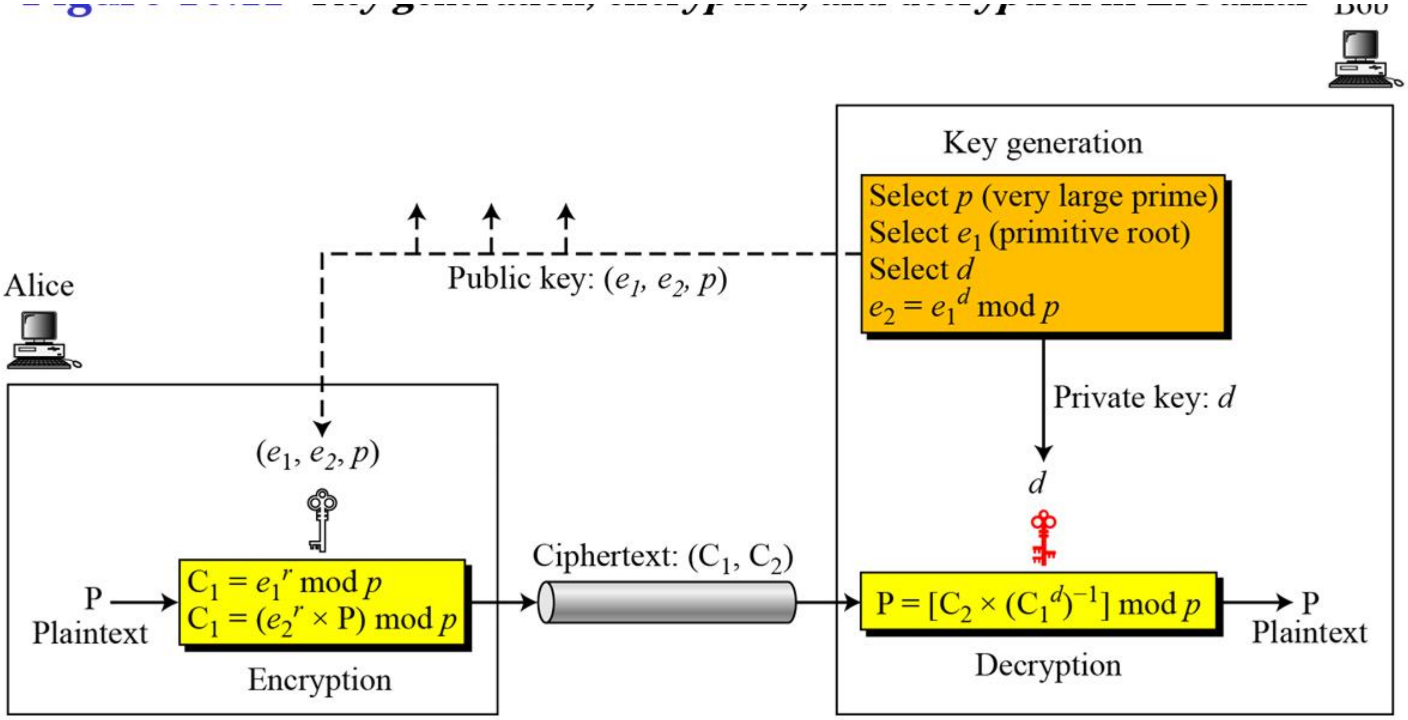
Name : **Rajneesh Pandey**

Section : **CSE-B**

Write a program to implement Elgamal encryption.

Explanation:

ElGamal encryption is a public-key cryptosystem. It uses asymmetric key encryption for communicating between two parties and encrypting the message. This cryptosystem is based on the difficulty of finding **discrete logarithm** in a cyclic group that is even if we know g^a and g^k, it is extremely difficult to compute g^{ak}.



ElGamal encryption can be defined over any cyclic group G, like multiplicative group of integers modulo n. Its security depends upon the difficulty of a certain problem in G related to computing discrete logarithms.

Implementation

Both encrypting and decrypting a message is implemented.

The program will generate a pair of keys (K1, K2) used for encryption and decryption.

K1 is the public key and contains three integers **(p, g, h).**

p is an n bit prime. The probability that p is prime is equal to $1-(2^{-t})$

g is the square of a primitive root mod p

$h = g^x \bmod p$; x is randomly chosen, $1 <= x < p$,

h is computed using fast modular exponentiation, implemented as modexp(base, exp, modulus)

K2 is the private key and contains three integers **(p, g, x)** that are described above.

Next the program encodes the bytes of the message into integers z[i] < p.

The module for this is named **encode()** and is described further where it is implemented.

After the message has been encoded into integers, the integers are encrypted and written. The encryption procedure is implemented in encrypt().

Algorithm

It works as follows:

Each corresponds to a pair (c, d) that is written to Ciphertext.

For each integer z[i]:

c[i] = g^y (mod p). d[i] = z[i] h^y (mod p)

where y is chosen randomly, 0 <= y < p

The decryption module decrypt() reads each pair of integers from Ciphertext and converts them back to encoded integers.

It is implemented as follows:

s = c[i]^x (mod p)

z[i] = d[i]*s^-1 (mod p)

The decode() module takes the integers produced from the decryption module and separates them into the bytes received in the initial message.

modular exponentiation

fast modular exponentiaton, **modexp()**

finding primitive roots

finding large prime numbers

finding prime numbers with confidence > 2

Code:

```
import random
import sys


class PrivateKey(object):
    def __init__(self, p=None, g=None, x=None, iNumBits=0):
        self.p = p
        self.g = g
        self.x = x
        self.iNumBits = iNumBits


class PublicKey(object):
    def __init__(self, p=None, g=None, h=None, iNumBits=0):
        self.p = p
        self.g = g
        self.h = h
        self.iNumBits = iNumBits

# computes the greatest common denominator of a and b.  assumes a > b


def gcd(a, b):
    while b != 0:
        c = a % b
        a = b
        b = c
    # a is returned if b == 0
    return a

# computes base^exp mod modulus


def modexp(base, exp, modulus):
    return pow(base, exp, modulus)



# solovay-strassen primality test.  tests if num is prime

def SS(num, iConfidence):
    # ensure confidence of t
    for i in range(iConfidence):
        # choose random a between 1 and n-2
        a = random.randint(1, num-1)
```

```
        # if a is not relatively prime to n, n is composite
        if gcd(a, num) > 1:
            return False

        # declares n prime if jacobi(a, n) is congruent to a^((n-1)/2) mod n
        if not jacobi(a, num) % num == modexp(a, (num-1)//2, num):
            return False

    # if there have been t iterations without failure, num is believed to be
prime
    return True

# computes the jacobi symbol of a, n


def jacobi(a, n):
    if a == 0:
        if n == 1:
            return 1
        else:
            return 0
    # property 1 of the jacobi symbol
    elif a == -1:
        if n % 2 == 0:
            return 1
        else:
            return -1
    # if a == 1, jacobi symbol is equal to 1
    elif a == 1:
        return 1
    # property 4 of the jacobi symbol
    elif a == 2:
        if n % 8 == 1 or n % 8 == 7:
            return 1
        elif n % 8 == 3 or n % 8 == 5:
            return -1
    # property of the jacobi symbol:
    # if a = b mod n, jacobi(a, n) = jacobi( b, n )
    elif a >= n:
        return jacobi(a % n, n)
    elif a % 2 == 0:
        return jacobi(2, n)*jacobi(a//2, n)
    # law of quadratic reciprocity
    # if a is odd and a is coprime to n
    else:
```

```python
        if a % 4 == 3 and n % 4 == 3:
            return -1 * jacobi(n, a)
        else:
            return jacobi(n, a)


# finds a primitive root for prime p
def find_primitive_root(p):
    if p == 2:
        return 1
    # the prime divisors of p-1 are 2 and (p-1)/2 because
    # p = 2x + 1 where x is a prime
    p1 = 2
    p2 = (p-1) // p1

    # test random g's until one is found that is a primitive root mod p
    while(1):
        g = random.randint(2, p-1)
        # g is a primitive root if for all prime factors of p-1, p[i]
        # g^((p-1)/p[i]) (mod p) is not congruent to 1
        if not (modexp(g, (p-1)//p1, p) == 1):
            if not modexp(g, (p-1)//p2, p) == 1:
                return g

# find n bit prime


def find_prime(iNumBits, iConfidence):
    # keep testing until one is found
    while(1):
        # generate potential prime randomly
        p = random.randint(2**(iNumBits-2), 2**(iNumBits-1))
        # make sure it is odd
        while(p % 2 == 0):
            p = random.randint(2**(iNumBits-2), 2**(iNumBits-1))

        # keep doing this if the solovay-strassen test fails
        while(not SS(p, iConfidence)):
            p = random.randint(2**(iNumBits-2), 2**(iNumBits-1))
            while(p % 2 == 0):
                p = random.randint(2**(iNumBits-2), 2**(iNumBits-1))

        # if p is prime compute p = 2*p + 1
        # if p is prime, we have succeeded; else, start over
        p = p * 2 + 1
        if SS(p, iConfidence):
```

```
            return p

# encodes bytes to integers mod p.  reads bytes from file


def encode(sPlaintext, iNumBits):
    byte_array = bytearray(sPlaintext, 'utf-16')

    # z is the array of integers mod p
    z = []

    # each encoded integer will be a linear combination of k message bytes
    # k must be the number of bits in the prime divided by 8 because each
    # message byte is 8 bits long
    k = iNumBits//8

    # j marks the jth encoded integer
    # j will start at 0 but make it -k because j will be incremented during first
iteration
    j = -1 * k
    # num is the summation of the message bytes
    num = 0
    # i iterates through byte array
    for i in range(len(byte_array)):
        # if i is divisible by k, start a new encoded integer
        if i % k == 0:
            j += k
            num = 0
            z.append(0)
        # add the byte multiplied by 2 raised to a multiple of 8
        z[j//k] += byte_array[i]*(2**(8*(i % k)))

    # example
        # if n = 24, k = n / 8 = 3
        # z[0] = (summation from i = 0 to i = k)m[i]*(2^(8*i))
        # where m[i] is the ith message byte

    # return array of encoded integers
    return z

# decodes integers to the original message bytes


def decode(aiPlaintext, iNumBits):
    # bytes array will hold the decoded original message bytes
    bytes_array = []
```

```
    # same as in the encode function.
    # each encoded integer is a linear combination of k message bytes
    # k must be the number of bits in the prime divided by 8 because each
    # message byte is 8 bits long
    k = iNumBits//8

    # num is an integer in list aiPlaintext
    for num in aiPlaintext:
        # get the k message bytes from the integer, i counts from 0 to k-1
        for i in range(k):
            # temporary integer
            temp = num
            # j goes from i+1 to k-1
            for j in range(i+1, k):
                # get remainder from dividing integer by 2^(8*j)
                temp = temp % (2**(8*j))
            # message byte representing a letter is equal to temp divided by
2^(8*i)
            letter = temp // (2**(8*i))
            # add the message byte letter to the byte array
            bytes_array.append(letter)
            # subtract the letter multiplied by the power of two from num so
            # so the next message byte can be found
            num = num - (letter*(2**(8*i)))

    # example
    # if "You" were encoded.
    # Letter        #ASCII
    # Y              89
    # o              111
    # u              117
    # if the encoded integer is 7696217 and k = 3
    # m[0] = 7696217 % 256 % 65536 / (2^(8*0)) = 89 = 'Y'
    # 7696217 - (89 * (2^(8*0))) = 7696128
    # m[1] = 7696128 % 65536 / (2^(8*1)) = 111 = 'o'
    # 7696128 - (111 * (2^(8*1))) = 7667712
    # m[2] = 7667712 / (2^(8*2)) = 117 = 'u'

    decodedText = bytearray(b for b in bytes_array).decode('utf-16')

    return decodedText

# generates public key K1 (p, g, h) and private key K2 (p, g, x)
```

```python
def generate_keys(iNumBits=256, iConfidence=32):
    # p is the prime
    # g is the primitve root
    # x is random in (0, p-1) inclusive
    # h = g ^ x mod p
    print("number of bits n : " + str(iNumBits))
    print("----------------------------------------")
    print("t is for probability that the key is prime is 1-(2^-t) : " +
str(iConfidence))

    p = find_prime(iNumBits, iConfidence)
    g = find_primitive_root(p)
    g = modexp(g, 2, p)
    x = random.randint(1, (p - 1) // 2)
    h = modexp(g, x, p)

    publicKey = PublicKey(p, g, h, iNumBits)
    privateKey = PrivateKey(p, g, x, iNumBits)

    return {'privateKey': privateKey, 'publicKey': publicKey}


# encrypts a string sPlaintext using the public key k
def encrypt(key, sPlaintext):
    z = encode(sPlaintext, key.iNumBits)

    # cipher_pairs list will hold pairs (c, d) corresponding to each integer in z
    cipher_pairs = []
    # i is an integer in z
    for i in z:
        # pick random y from (0, p-1) inclusive
        y = random.randint(0, key.p)
        # c = g^y mod p
        c = modexp(key.g, y, key.p)
        # d = ih^y mod p
        d = (i*modexp(key.h, y, key.p)) % key.p
        # add the pair to the cipher pairs list
        cipher_pairs.append([c, d])

    encryptedStr = ""
    for pair in cipher_pairs:
        encryptedStr += str(pair[0]) + ' ' + str(pair[1]) + ' '

    return encryptedStr

# performs decryption on the cipher pairs found in Cipher using
```

```
# prive key K2 and writes the decrypted values to file Plaintext


def decrypt(key, cipher):
    # decrpyts each pair and adds the decrypted integer to list of plaintext
integers
    plaintext = []

    cipherArray = cipher.split()
    if (not len(cipherArray) % 2 == 0):
        return "Malformed Cipher Text"
    for i in range(0, len(cipherArray), 2):
        # c = first number in pair
        c = int(cipherArray[i])
        # d = second number in pair
        d = int(cipherArray[i+1])

        # s = c^x mod p
        s = modexp(c, key.x, key.p)
        # plaintext integer = ds^-1 mod p
        plain = (d*modexp(s, key.p-2, key.p)) % key.p
        # add plain to list of plaintext integers
        plaintext.append(plain)

    decryptedText = decode(plaintext, key.iNumBits)

# remove trailing null bytes
    decryptedText = "".join([ch for ch in decryptedText if ch != '\x00'])

    return decryptedText


def test(message):
    assert (sys.version_info >= (3, 4))
    keys = generate_keys()
    priv = keys['privateKey']
    pub = keys['publicKey']
    cipher = encrypt(pub, str(message))
    plain = decrypt(priv, cipher)
    return {'privateKey': priv, 'publicKey': pub, 'cipher': cipher, 'plain':
plain}


# taking input of the message that have to be encrypted and decrypting it
with open('plainText.txt', 'r') as file:
    plainText = file.read().rstrip()
```

```python
print("----------------------------------------")
print("message: ", str(plainText))
print("----------------------------------------")
values = test(plainText)

print("----------------------------------------")
print("Cipher Text: ", str(values['cipher']))
print("----------------------------------------")
print("Decoded Text: ", str(values['plain']))
print("----------------------------------------")
```

Output :

```
TERMINAL    PROBLEMS    OUTPUT    DEBUG CONSOLE

rajne Code
$ python elgamal.py
----------------------------------------
message:  My name is Rajneesh-106119100 and this is assignment 3 of Cryptography.
----------------------------------------
number of bits n : 256
----------------------------------------
t is for probability that the key is prime is 1-(2^-t) : 32
----------------------------------------
Cipher Text:  59307842377098768603478130783963997862314762813771475801425439159273376954974 6503531846001217003912660109053980456701484336940339376033191621969
4107981360 26317352075133253809600973075112559067581548275237866833043116165552710710973 7421788070552084967818113741369046392793210495430207063638909492635379
0965799 29202830020050694970821645664465783608560107562937425959002340591693729687884 1445164864647221793668126025157826739989982774115727398596104615458616755
2402 42123008276373782620056385268074902647913053982614518627977616018534673128420 4310851226158315891168964470143275678272092945984785375535526044324942490348
8 25706984498268886609124682909996440970707905902577357613796854269787963007418 31888142430115771276221272646293739153808399014244446691854320195142097042380
----------------------------------------
Decoded Text:  My name is Rajneesh-106119100 and this is assignment 3 of Cryptography.
----------------------------------------
rajne Code
$ |
```