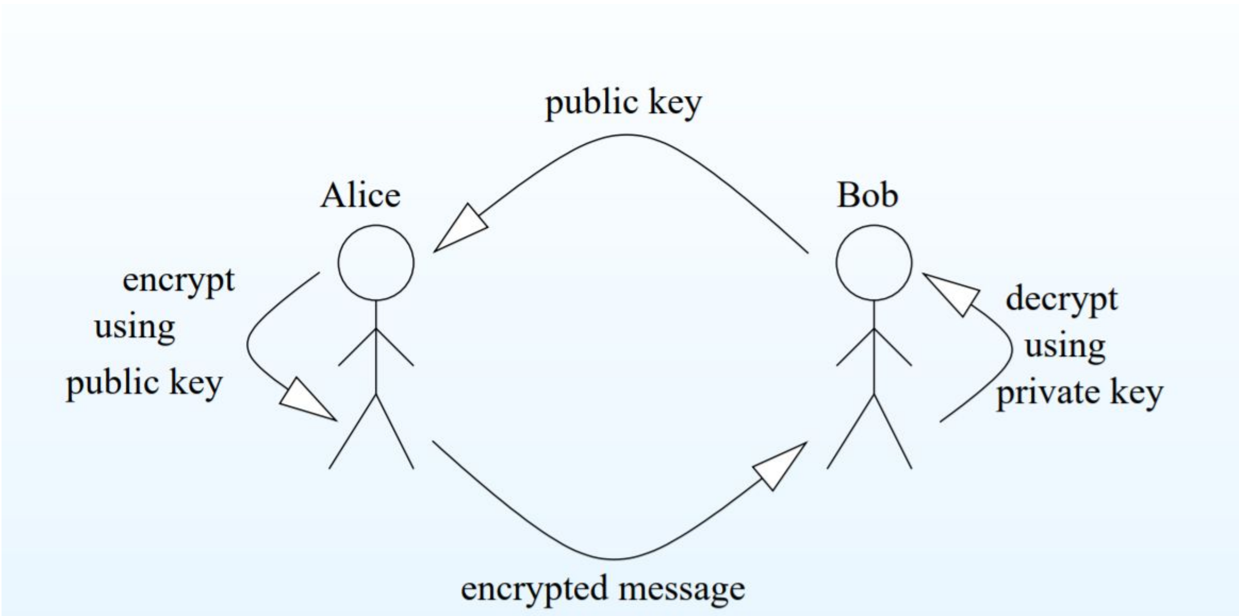
B. Tech – CE | Semester: VI | Subject: NIS
Lab 5

Overview Of Elgamal cryptosystem :

→ In 1984, T. Elgamal announced a public-key scheme based on discrete logarithms, closely related to the Diffie-Hellman technique.The Elgamal cryptosystem is one of the most widely used public-key cryptosystems that depends on the difficulty of computing the discrete logarithms over finite fields. Over the years, the original system has been modified and altered in order to achieve a higher security and efficiency. In this paper, a generalization for the original ElGamal system is proposed which also relies on the discrete logarithm problem.

→ The encryption process of the scheme is improved such that it depends on the prime factorization of the plaintext. Modular exponentiation is taken twice during the encryption; once with the number of distinct prime factors of the plaintext and then with the secret encryption key. If the plaintext consists of only one distinct prime factor, then the new method is similar to that of the basic ElGamal algorithm. The proposed system preserves immunity against the Chosen Plaintext Attack (CPA) .The ElGamal cryptosystem is used in some form in a number of standards including the digital signature standard (DSS).



Two public parameters:

• p: prime number

• g: generator such that ∀n ∈ [1; p − 1] : ∃k;

n = gk mod p

Procedure: 1.

• Alice generates a private random integer a

• Bob generates a private random integer b

Procedure 2.

• Alice generates her public value g^a mod p

• Bob generates his public value g^bb mod p

Procedure3.

• Alice computes gab = (g^a)*(1^b) mod p

• Bob computes gba = (g^b)*1^a mod p

Procedure 4.

Both now have a shared secret k since k = g^(ab) = g^(ba)

<u>Task : Write a Program to find the primitive roots for the Multiplicative Group with respect to Prime Modulus. Using that Implement Elgamal Cryptosystem.</u>

Source Code : Python

```python
from collections import defaultdict
import math
from random import randrange, getrandbits
import random

# primitive_root_list=list()

def DoMultiplictiveInverse(n , a):
    r1, r2, t1, t2= n, a, 0, 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:
        if t1 < 0:
            return t1 + n
        else:
            return t1

    else:
        return -1

def is_prime(n, k=128):
    """ Test if a number is prime
        Args:
            n -- int -- the number to test
            k -- int -- the number of tests to do
        return True if n is prime
    """
    # Test if n is not even.
    # But care, 2 is prime !
    if n == 2 or n == 3:
        return True
```

```
    if n <= 1 or n % 2 == 0:
        return False
    # find r and s
    s = 0
    r = n - 1
    while r & 1 == 0:
        s += 1
        r //= 2
    # do k tests
    for _ in range(k):
        a = randrange(2, n - 1)
        x = pow(a, r, n)
        if x != 1 and x != n - 1:
            j = 1
            while j < s and x != n - 1:
                x = pow(x, 2, n)
                if x == 1:
                    return False
                j += 1
            if x != n - 1:
                return False
    return True

def generate_prime_candidate(length):
    """ Generate an odd integer randomly
        Args:
            length -- int -- the length of the number to generate, in bits
        return a integer
    """
    # generate random bits
    p = getrandbits(length)
    # apply a mask to set MSB and LSB to 1
    p |= (1 << length - 1) | 1
    return p


def generate_prime_number(length=14):
    """ Generate a prime
        Args:
```

```
            length -- int -- length of the prime to generate, in
bits
        return a prime
    """
    p = 4
 # keep generating while the primality test fail
    while not is_prime(p, 128):
        p = generate_prime_candidate(length)
    return p

def fast_power(bas, exp, N):
    t = 1
    while(exp > 0):

        # for cases where exponent
        # is not an even value
        if (exp % 2 != 0):
            t = (t * bas) % N

        bas = (bas * bas) % N
        exp = int(exp / 2)
    return t % N


def find_roots(goto, MOD):
    root_list=list()
    for i in goto:
        for j in range(1 , len(goto)+1):
            temp = fast_power(i , j,MOD)
            if  temp == 1 and j != len(goto):
                break
            elif temp == 1 and j == len(goto):
                root_list.append(i)
    return root_list



def key_Generation():
    p = generate_prime_number()
    print("p -->", p)
```

```python
    goto=list()
    for i in range(1, p):
        if DoMultiplictiveInverse(p , i) != -1:
            goto.append(i)
    primitive_root_list = find_roots(goto, p)

    e1 = random.choice(primitive_root_list)
    d=random.randint(1 , p-2)
    e2 = fast_power(e1 , d , p)
    public_key = [e1, e2, p]
    private_key = d
    print("public key -->", public_key)
    print("Private key -->" , private_key)
    return public_key , private_key, p, primitive_root_list

def Elgamal_Encryption(m, e1, e2, p):
    r = random.randint(1, p-2)
    # r = 4
    print("r -->" , r)
    c1 = fast_power(e1 ,r, p )
    print("e2 -->", e2)
    c2 = (fast_power(e2, r, p) * (m%p)) % p

    return c1, c2

def Elgamal_Decryption(c1, c2, d, p):

    c1 = fast_power(c1 , d, p)
    c1_inv = DoMultiplictiveInverse(p , c1)
    print()
    print("c1_inv -->",c1_inv)
    ans = (c1_inv*c2)%p
    return ans

if __name__ == "__main__":
    M = 200
    print("m-->", M)
    public_key, private_key, p, primitive_root_list = key_Generation()
    c1, c2 = Elgamal_Encryption(M , public_key[0] , public_key[1], p)
    print("c1, c2 -->", c1, c2)
```

```python
    decryption = Elgamal_Decryption(c1, c2, private_key, p)
    print("decrypted Number is -->",decryption)
```

Output :

```
C:\Users\acer>python C:\Users\acer\Desktop\temper\elgamal.py
m--> 123
p --> 5653
public key --> [2294, 2927, 5653]
Privarte key --> 3689
r --> 915
e2 --> 2927
c1, c2 --> 2000 3280

c1_inv 636
decryption 123

C:\Users\acer>python C:\Users\acer\Desktop\temper\elgamal.py
m--> 200
p --> 12853
public key --> [7807, 3193, 12853]
Privarte key --> 10512
r --> 5081
e2 --> 3193
c1, c2 --> 1435 8302

c1_inv --> 9639
decrypted Number is --> 200
```