Practical Assignment-5
Network Security

Chinmay Somani

18074004

CSE IDD Part 4

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 logarithms** in a cyclic group that is even if we know ga and gk, it is extremely difficult to compute gak.

Idea of ElGamal cryptosystem

Suppose Alice wants to communicate with Bob.

1. Bob generates public and private keys:
   - Bob chooses a very large number **q** and a cyclic group **F**q.
   - From the cyclic group **F**q, he choose any element **g** and

     an element **a** such that gcd(a, q) = 1.
   - Then he computes h = ga.
   - Bob publishes **F**, **h = ga**, **q**, and **g** as his public key and retains **a** as private key.
2. Alice encrypts data using Bob's public key :
   - Alice selects an element **k** from cyclic group **F**

     such that gcd(k, q) = 1.
   - Then she computes p = gk and s = hk = gak.
   - She multiples s with M.
   - Then she sends (p, M*s) = (gk, M*s).
3. Bob decrypts the message :
   - Bob calculates s′ = pa = gak.
   - He divides M*s by s′ to obtain M as s = s′.

Public Key generation

alice = Elgamal()

alice_public_key = alice.publishPublicKey()

Encryption

message = "Hello World !"

bob = Elgamal()

bob_cipher = bob.cipher(alice_public_key, message)

Decryption

alice_decrypted = alice.unCipher(bob_cipher)

A Python3 implementation of ElGamal encryption algorithm.

```
from random import randint
from math import ceil, sqrt
def generateBigOddNumber():
   '''
   A big odd number generator
   :return: an odd number between 2**8 and 2**11
   '''
   return (2 * randint(2 ** 8, 2 ** 10)) + 1

def generateBigSafePrimeNumber():
   '''
   A big safe prime number generator
   :return: a big safe prime number (2**8 < x < 2**16)
   '''
   big_odd_number = generateBigOddNumber()
   while not (isSafePrime(big_odd_number)):
       big_odd_number -= 2
```

```
    return big_odd_number


def is_prime(x):
    '''
    Test the primality of a number
    :param x: the number to test
    :return: if x is prime or not
    '''
    if x % 2 == 0:
        return False
    # by steps of 2, avoid to check even number
    # stop at square of x to avoid useless check, because k * i = n with i > sqrt(n)
is impossible
    # because it implies that k < sqrt(n) so it will have been already checked
    for i in range(3, ceil(sqrt(x)), 2):
        if x % i == 0:
            return False
    return True


def generateASmallerPrimeNumber(x, q):
    '''
    Generate a prime number smaller than x
    :param x: a prime number
    :return: a prime number smaller than x
    '''
    # generate a number between 2 and x-1
    startNumber = randint(2, x - 1)
    # if we haven't generate 2 and startNumber is even, we change startNumber to an
odd number
    if startNumber != 2 and startNumber % 2 == 0:
        startNumber -= 1
    while not (is_prime(startNumber)) and startNumber > 2:
        startNumber -= 2
    return startNumber
```

```python
def generateQuadtraticGenerator(p):
    '''
    Generate a quadratic residual generator of the cyclic group of order p (nammed Qp with p is safe prime) using the subgroup q
    When the order of group is prime, all element are generator
    x^2 mod order_of_groups is quadratic residual
    :param p: the order of the cyclic group
    :return: a generator
    '''
    q = int((p - 1) / 2)
    if not isSafePrime(p):
        raise Exception("p not safe prime")
    if not is_prime(q):
        raise Exception("q not prime, p not safe prime")
    generator = randint(2, min(2**4, q))
    if not isSafePrime(p):
        raise Exception("Safe prime needed")
    generator = (generator ** 2) % p
    if not quadraticResidual(generator, p):
        raise Exception("Generator need to be a quadratic residual")
    # test generator
    residual_generated = [lambda i: (0 if not quadraticResidual(i, p) else 1) for i in range(0, p)]
    for i in range(1, q):
        tmp = (generator ** i) % p
        if residual_generated[tmp] == 0:
            raise Exception("Not a quadratic generator")
        elif residual_generated[tmp] == 1:
            residual_generated[tmp] = 2
        elif residual_generated[tmp] == 2:
            raise Exception("Not a cyclic group")
    return generator
```

```
def quadraticResidual(a, q):
    '''
    Check if a number is a quadratic residual
    :param a: the number to test
    :param q: the order of the cyclic group
    :return: if a is a quadratic residual
    '''
    for i in range(1, q):
        if a % q == (i ** 2) % q:
            return True
    return False


def isSafePrime(q):
    return is_prime(q) and is_prime((q - 1) / 2)
```

```
from random import randint

class Elgamal:

    def __init__(self):
        '''
        The initialisation of ElGamal's protocol, initialise our public and private
key
        '''
        self.q = generateBigSafePrimeNumber()
        self.g = generateQuadtraticGenerator(self.q)
        # we choose a big secret key to prevent from the attack
        self.sk = randint(min(2**4, int(self.q/2)), self.q)
        self.h = (self.g ** self.sk) % self.q
        self.residual = []
        for i in range(0, self.q):
            if quadraticResidual(i, self.q):
                self.residual.append(i)
```

```
def publishPublicKey(self):
        '''
        :return: publish your public key
        '''
        return self.q, self.g, self.h

def cipher(self, pk, m):
        '''
        Cipher a message from a public key
        :param pk: Alice's public key (tuple q,g,h)
        :param m: the message to cipher
        :return: a tuple containing an information about the random x picked and
the cipher
        '''
        q = pk[0]
        g = pk[1]
        h = pk[2]
        r = randint(1, q)
        c1 = (g ** r) % q
        y = (h ** r) % q
        residual = []
        for i in range(0, q):
            if quadraticResidual(i, q):
                residual.append(i)
        if type(m) == str:
            if m > q / 2:
                raise Exception("Message too big for the ordrer of the group")
            else:
                m = residual[m]
                c2 = ""
                for character in m:
                    c2 = c2 + str(ord(character) * y) + ","
        else:
            if m > q / 2:
                raise Exception("Message too big for the ordrer of the group")
            else:
                m = residual[m]
                c2 = m * y
```

```
        return c1, c2

    def unCipher(self, cipher):
        '''
        Decrypt the cipher
        :param cipher: a tuple containing C1 (g**r mod q) from Bob and the cipher,
which could be a str or a number
        :return: the unencrypted message
        '''
        c1 = cipher[0]
        cipher = cipher[1]
        if cipher == -1:
            print("ERREUR : Message plus grand que l'ordre du groupe cyclique")
            result = "ERROR"
        else:
            if type(cipher) == str:
                result = ""
                for character in cipher.split(','):
                    if character != '':
                     result = result + chr(int(int(character) / ((c1 ** self.sk) %
self.q)))
            else:
                result = cipher / ((c1 ** self.sk) % self.q)
                result = self.residual.index(result)
        return result
```

```python
def attackElGamal(public_key):
    '''
    Print the private key if she's founded
    Prove that it's important to choose a big random x to prevent attacks
    :param public_key: the public key tuple
    '''
    q = public_key[0]
    g = public_key[1]
    h = public_key[2]
    for i in range(1, q):
        if (g ** i) % q == h:
            print("FOUND !")
            print("Private key = {}".format(i))
            return


if __name__== "__main__":
    alice = Elgamal()
    bob = Elgamal()

    # El Gamal for integer
    print("Integer")
    message = 5
    alice_public_key = alice.publishPublicKey()
    print("Mesage".format(message))
    print("THis is Alice Public keys : {}".format(alice_public_key))
    print("THis is Alice first Public keys : : {}".format(alice_public_key[0]))
    print(alice_public_key)
    bob_cipher = bob.cipher(alice_public_key, message)
    print("Bob cipher: {}".format(bob_cipher))
    alice_decrypted = alice.unCipher(bob_cipher)
    print("Alice decrypted {}".format(alice_decrypted))
```