ElGamal Encryption Algorithm

Rad Rares

This paper's subject is an implementation of the ElGamal public key cryptosystem. It is based on the Discrete Logarithm Problem and the Diffie-Hellman key exchange. It useses the fact that is very dificult to find a discrete logarithm in a cyclic group.

Let's denote g as a generator of the (Z*_p,.) cyclic group, with p a large random prime. a is a random integer belonging to (1,p-2), the public key will be a triple consisting of (p,g,gˆa modulo p) and the private one will be a.

The encryption consists in selecting another random integer, let's say k in the bounds (1,p-2). Denote m as the message we want to encrypt, represented as a number between 0 and p-1, in this case we write is in base 27 with coefficients the position in the alphabet of each letter. We will encrypt this message with two parts, alpha and beta. alpha will be gˆk modulo p and beta will be m * (g^a)k modulo p The touple (alpha, beta) will be send as the cipher.

To get the message in the decryption part, the user applies the private key a as such: raises alphaˆ-a * beta modulo p. By doing this, the gˆ-ak cancels with gˆak and only the number representing the message remains.

The whole idea of how this is dificult to crack is that the original message m is masked by gˆak, the common ground of both parts if gˆk and if someone doesn't know a, m cannot be unmasked.

This is the main function that runs the program, we generate the keys, read the message, encrypt it, decrypt it and print the result. If the result was the same as the original message, we print success.

```
<<*>>=
<<Keys>>
<<Encrypt>>
<<Decrypt>>
<<Test>>
def main():

    keys = generateKeys()
    publicKey = keys[0]
    privateKey = keys[1]
    decryptedMessage = ""
```

```
    message = input("INPUT MESSAGE PLS (uppercase letters from the english alphabet\n")
    print("message " + message)
    chunks = [message[i:i+2] for i in range(0, len(message), 2)]
    for chunk in chunks:
        encrypted = encrypt(publicKey,chunk)
        if(encrypted == False):
            print("Character not in alphabet, please input only eng uppercase letters")
            break
        decrypted = decrypt(privateKey,encrypted,publicKey[0])
        decryptedMessage += decrypted

    print("decrypted message " + decryptedMessage)

    if(decryptedMessage == message):
        print("ENCRYPTION-DECRYPTION SUCCEDED")

test()
main()
@
```

This is the test function. I just took some good and bad cases and run the algorithm over them. The ones that are bad should stop the program and tell the user to use valid messages.

```
<<Test>>=
<<Keys>>
<<Encrypt>>
<<Decrypt>>
def test():
    print("TESTS STARTING\n")
    keys = generateKeys()
    publicKey = keys[0]
    privateKey = keys[1]
    messeges = [" ","abc", "NU MI E OK", "ABC9", "ABCD", "ABCD EFG"]
    for message in messeges:
        decryptedMessage = ""
        print("message " + message)
        chunks = [message[i:i + 2] for i in range(0, len(message), 2)]
        for chunk in chunks:
            encrypted = encrypt(publicKey, chunk)
            if (encrypted == False):
                print("ERROR: Character not in alphabet, please input only eng uppercase let
                break
            decrypted = decrypt(privateKey, encrypted, publicKey[0])
            decryptedMessage += decrypted

        print("decrypted message " + decryptedMessage)
```

```
        if (decryptedMessage == message):
            print("ENCRYPTION-DECRYPTION SUCCEDED\n")
    print("TESTS FINISHED\n")
    return
@
```

This is the key generating function. We generate a big prime, a generator of Z*_p and a random integer a belonging (1,largePrime-2). The public key will be (largePrime, generator, generator^a). The private key will be a.

```
<<Keys>>=
<<BigPrime>>
<<GenerateGenerator>>
def generateKeys():
    import random
    largeRandomPrime = generateBigPrimeNumber()
    print("prime " + str(largeRandomPrime))
    generator = generateGenerator(largeRandomPrime)
    print("generator " + str(generator))
    privateKey = random.randrange(1,largeRandomPrime-2)
    print("private key " + str(privateKey))
    publicPart = pow(generator,privateKey,largeRandomPrime)
    print("g^a " + str(publicPart))
    publicKey = (largeRandomPrime,generator,publicPart)
    return (publicKey,privateKey)
@
```

Function that generates a big prime. It generates a big odd number and while it is not prime, reduces it by 2.

```
<<BigPrime>>=
<<BigOdd>>
<<CheckPrime>>
def generateBigPrimeNumber():
    bigOdd = generateBigOddNumber()
    while not (isPrime(bigOdd)):
        bigOdd -= 2
    return bigOdd
@
```

Auxiliar function that generates a number between (2^9, 2^11) then adds 1 so it will be odd.

```
<<BigOdd>>=
def generateBigOddNumber():
    import random
    return (2 * random.randint(2 ** 8, 2 ** 10)) + 1
@
```

Simple function that checks if the number is prime by first checking if it is prime, then going from 2 to 2 until the square root if it if it is divided.

```
<<CheckPrime>>=
def isPrime(x):
    import math
    if x % 2 == 0:
        return False
    for i in range(3, math.ceil(math.sqrt(x)), 2):
        if x % i == 0:
            return False
    return True
@
```

This function generates the generator of Z*_p. This was a bit dificult to do, but I searched in some online books and sites. The order of Z*_p with p prime is p-1. We factorize p-1. We choose a random element alpha in Z*_p. For each factor we compute b = alphaˆ(p/factor) If b is 1 we try with another factor. Else alpha is a generator. Basically, in my humble understanding it has to do with the coprimes between the elements in Z*_p and p, my first year algebra is kinda dusty.

```
<<GenerateGenerator>>=
<<PrimeFactors>>
def generateGenerator(n):
    import random,math
    while(True):
        alpha = random.randrange(1,n)
        factors = list(primes(n-1))
        for factor in factors:
            if(pow(alpha,math.ceil(n-1/factor),n)==1):
                continue
            else:
                return alpha
@
```

Simple function that returns the set of the decomposition in primes of a number.

```
<<PrimeFactors>>=
def primes(n):
    primeFactors = set()
    d = 2
    while d*d <= n:
        while (n % d) == 0:
            primeFactors.add(d)
            n //= d
        d += 1
    if n > 1:
```

```
        primeFactors.add(n)
    return primeFactors


    return True
@
```

Encryption function: Receives the public key and the message. Checks the validity of the message, as in if the characters belong to the alphabet. Write the message as an integer < p-1 in base 27, with the coefficients as their position in the alphabet. Generates a random integer k < p-2. The first part of the cipher is alpha = gˆk. The second part of the cipher is beta = numberEquivalentToMessage * gˆa modulo p gˆa and g are known from the public key. Lastly we send the cypher.

```
<<Encrypt>>=
def encrypt(publicKey, message):
    import random
    alphabet = list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')

    inOrder = True
    for character in message:
        if character not in alphabet:
            inOrder = False

    if(inOrder):
        textToNumber = 0
        string = ""
        i=len(message)-1
        for character in message:
            textToNumber += alphabet.index(character)*pow(27,i)
            string += str(alphabet.index(character))+"*27^"+str(i) + "\n"
            i-=1

        #print("text2Number " + str(textToNumber))
        # print(publicKey)
        k = random.randrange(1,publicKey[0]-1)
        alpha = pow(publicKey[1],k,publicKey[0])
        beta = textToNumber*pow(publicKey[2],k,publicKey[0])
        beta = beta%publicKey[0]

        return (alpha,beta)

    else: return False
@
```

Decryption function: Receive the private key, the cipher. Raises the multiplied

parts of the cipher to the private key to nullify gˆak Returns a string composed of the letters from the alphabet with position the quotient of the division to 27 and the remainder to 27, 27 being the length of the alphabet(with the blank space added)

```
<<Decrypt>>=
def decrypt(private,cipher,p):
    import math
    alphabet = list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
    messageInNumbers = pow(cipher[0],p-1-private)*cipher[1]%p

    return str(alphabet[math.floor(messageInNumbers/27)]) + str(alphabet[math.floor(messagel
@
```