Cryptography Lab4 Homework - ElGamal

Bardas Alexandru-Cristian, Group 321

In this paper, I will demonstrate the principles of the ElGamal cryptosystem, alongside with proofs of correctness for the algorithm and code examples.

Introduction

In cryptography, the ElGamal encryption system is an asymmetric key encryption algorithm for public-key cryptography which is based on the Diffie–Hellman key exchange. It was described by Taher Elgamal in 1985. ElGamal encryption is used in the free GNU Privacy Guard software, recent versions of PGP and other cryptosystems. The Digital Signature Algorithm (DSA) is a variant of the ElGamal signature scheme, which should not be confused with ElGamal encryption. 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.

Mathematical principles used in ElGamal

Definition of a group

A group is a set G together with a binary operation on G, here denoted $\cdot$, that combines any two elements a and b to form an element of G, denoted $a \cdot b$, in a way such that the following three requirements, known as group axioms, are satisfied:

1. Associativity: For all a, b, c in G, one has $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
2. Identity element: There exists an element e in G such that, for every a in G, one has $e \cdot a = a$ and $a \cdot e = a$. Such an element is unique and it is called the identity element of the group.
3. Inverse element: For each a in G, there exists an element b in G such that $a \cdot b = e$ and $b \cdot a = e$, where e is the identity element. For each a, the element b is unique; it is called the inverse of a and is commonly denoted a^{-1}.

Definition of a cyclic group and generators

A group $(G, \cdot)$ is called cyclic if there exists $x \in G$ such that $G =< x >$, that is, $G = \{x^k \mid k \in \mathbb{Z}\}$. Here x is called a generator of G.

The order of an element in a group

Let $(G, \cdot)$ be a group and $x \in G$. We say that x has finite order if $\exists m \in \mathbb{N} \ / \ \{0\}$: $x^m = 1$. In this case, ord $x = min\ \{k \in \mathbb{N} \ / \ \{0\} \mid x^k = 1\}$ is called the order of x.

Let $(G, \cdot)$ be a finite cyclic group with n elements generated by an element x. Then ord $x = n$ and $G =< x >= \{1, x, x^2, ..., x^{n-1}\}$.

Lagrange's Theorem

If H is a subgroup of a group G, then $|G| = [G : H] \cdot |H|$.

Corollary of Lagrange's Theorem

Let $(G, \cdot)$ be a finite group. Then $\forall x \in G$, ord x divides $|G|$.

Proof

Since $< a >$ is a subgroup of G then $| < a > | \ | \ |G|$. By the above statement about the order of an element in a group, ord $a = | < a > |$. Hence, by Lagrange's Theorem, ord $a \mid |G|$.

Congruences

Given an integer $n > 1$, called a modulus, two integers are said to be congruent modulo n, if n is a divisor of their difference (i.e., if there is an integer k such that $a - b = k \cdot n$). Congruence modulo n is a congruence relation, meaning that it is an equivalence relation that is compatible with the operations of addition, subtraction, and multiplication. Congruence modulo n is denoted:

$a \equiv b \pmod{n}$

Multiplicative group of integers modulo n

In modular arithmetic, the integers coprime (relatively prime) to n from the set $\{0, 1, ..., n-1\}$ of n non-negative integers form a group under multiplication modulo n, called the multiplicative group of integers modulo n. Equivalently, the elements of this group can be thought of as the congruence classes, also known as residues modulo n, that are coprime to n. Hence another name is the group of primitive residue classes modulo n. This group, usually denoted $(\mathbb{Z}/n\mathbb{Z})^\times$, is fundamental in number theory. It has found applications in cryptography, integer factorization and primality testing. It is an abelian, finite group whose order is given by Euler's totient function: $|(\mathbb{Z}/n\mathbb{Z})^\times| = \varphi(n)$. For prime n the

group is cyclic and in general the structure is easy to describe, though even for prime n no general formula for finding generators is known.

ElGamal

Underlying problems

Discrete Logarithm Problem

Let $(G, \cdot)$ be a finite cyclic group with n elements, having a generator g and let $y \in G$. Determine a power x $(0 \leq x \leq n-1)$ such that $y = g^x$ (we formally write $x = log_g y$).

Diffie-Hellman Problem

Let $(G, \cdot)$ be a finite cyclic group with n elements, having a generator g and let g^a, $g^b \in G$ for some $a, b \in \{0, \ldots, n-1\}$. Determine $g^{a \cdot b}$.

These two problems are conjectured to be computationally equivalent, both being solvable only by exponential-time algorithms.

1. Key generation

1.1. We generate a random prime p and a generator g of $(\mathbb{Z}_p$ - $\{\hat{0}\}, \cdot)$.

In the algorithm I have created, I have used the notion of a Sophie-Germain prime and it's associated safe prime. In number theory, a prime number p is a Sophie-Germain prime if $2 \cdot p + 1$ is also prime. The number $2 \cdot p + 1$ associated with a Sophie-Germain prime is called a safe prime. Choosing a Sophie-Germain prime eases our job of finding a generator. By Lagrange's corollary, we know that the order of an element must divide the order of the group. The order of our group is $p - 1$, but by choosing a safe prime, we know that $q = (p-1)/2$ is also prime, hence the only divisors of $p - 1$ are 2 and q. We also know that the order of a generator g is always the order of the group, hence ord $g = p - 1$. So, the only checking that we ought to do is that our chosen generator g is not of order 2 or q.

Our prime generation algorithm consists of multiple steps :

- generating a number of the desired bit-length (I used 256 bits in this example)
- ruling out some of the possible prime factors

```
<<genLowLevelPrime>>=
# Pre generated primes
first_primes_list = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
                     31, 37, 41, 43, 47, 53, 59, 61, 67,
```

```
                    71, 73, 79, 83, 89, 97, 101, 103,
                    107, 109, 113, 127, 131, 137, 139,
                    149, 151, 157, 163, 167, 173, 179,
                    181, 191, 193, 197, 199, 211, 223,
                    227, 229, 233, 239, 241, 251, 257,
                    263, 269, 271, 277, 281, 283, 293,
                    307, 311, 313, 317, 331, 337, 347, 349]

def getLowLevelPrime(n):
    # Generate a prime candidate not divisible
    # by first primes
    while True:
        # Obtain a random number of the desired bit length
        # n bits
        pc = rand.randrange(2 ** (n - 1) + 1, 2 ** n - 1)

        # Test divisibility by pre-generated
        # primes
        for divisor in first_primes_list:
            if pc % divisor == 0 and divisor ** 2 <= pc:
                break
        else:
            return pc

@
```

- testing if that number x is prime, and also if $(x-1)/2$ is prime, hence making x be a safe prime, but this function returns the Sophie-Germain prime associated

```
<<genRandPrime>>=
def genRandPrime(n):
    # creates a n-bit length number that isnt divisible by the primes
    # in the pregenerated list, then puts it to the Miller Rabin test
    # we test to see if we can get Sophie-Germain primes,and the safe prime associated
    # with that Sophie-Germain prime
    while True:
        prime_candidate = getLowLevelPrime(n)
        if not millerRabin(prime_candidate):
            continue
        else:
            if not millerRabin((prime_candidate - 1) // 2):
                continue
            else:
                return (prime_candidate - 1) // 2

@
```

The primality test used is the Miller-Rabin test, which will be found in the Annexe.

1.2. Select a random integer a $(1 \leq a \leq p-2)$.

1.3. Compute g^a mod p.

1.4. The public key is (p, g, g^a); the private key is a.

```
<<genKeys>>=
def gen_keys():
    # generates a public and private key
    p = genRandPrime(256)
    # generates a Sophie-Germain prime of 256 bits
    g = 0
    q = p
    # the associated safe prime
    p = 2 * q + 1
    # using our safe prime,we test for a generator
    found = False
    while not found:
        # we keep generating numbers until one satisfies the condition we have mentioned ab
        g = rand.randint(2, p - 1)
        if pow(g, 2) % p != 1 and power(g, q, p) != 1:
            found = True
    # generates random a
    a = rand.randint(2, p - 2)
    # computes g^a using repeated squaring modular exponentiation
    ga = power(g, a, p)
    publ = (p, g, ga)
    priv = a
    return publ, priv
@
```

The algorithm for repeated squaring modular exponentiation will be found in the Annexe.

2. Encryption

2.1. Get a public key (p, g, g^a).

2.2. Represent the message as a number m between 0 and $p-1$.

Here, the variant I've chosen to represent the message is quite basic. We take our message m, parse it character by character and create a long string which concatenates the binary representation of each character, in the order they appear in the message.

```
<<StringToBin>>=
def StringToBin(msg):
    # returns the binary representation of a given string
    # by iterating through the string and saving the binary representation
    # for each character
    m = ""
    for letter in msg:
        aux = str(bin(ord(letter)))[2:]
        # adds 0's in front so that the byte is complete
        while len(aux) != 8:
            aux = "0" + aux
        m = m + aux
    return m
@
```

That binary string is then represented into decimal, and if $m > p$, we have to split the message into chunks and encrypt them individually.

```
<<breakIntoChunks>>=
def breakIntoChunks(msg, size):
    # breaks a string into chunks, with a given bound for the chunk size
    # it splits the message into chunks s.t. each chunk contains full bytes
    chunks = []
    i = 0
    diff = math.floor(size // 8)
    diff = diff*8
    while i < len(msg):
        chunks.append(msg[i : i + diff])
        i = i + diff
    return chunks

@
```

Each chunk's value will be less than p, and it will contain full bytes, so we can avoid errors by letting chunks contain partial bytes and then losing information in the process.

2.3. Select a random integer k $(1 \leq k \leq p-2)$.

2.4. Compute $\alpha = g^k \bmod p$ and $\beta = m \cdot (g^a)^k \bmod p$.

2.5. Obtain the ciphertext $c = (\alpha, \beta)$.

```
<<encrypt>>=
def encrypt(msg, p, g, ga):
    k = rand.randint(2, p - 2) # generates k
    # represent the message m in binary
```

```
    # takes the binary ascii code of each character in the messages
    # and concatenates them into a big long string
    m = StringToBin(msg)
    binMsg = m
    m = int(m, 2) # gets the decimal value of that
    alfa = power(g, k, p)
    beta = [] # beta is made a list in case there's a need to split the
    # message into chunks and encrypt them individually
    # since beta carries the message into the ciphertext
    aux = power(ga, k, p)
    if m > p:
        chunks = breakIntoChunks(binMsg, p.bit_length())
        # encrypt each chunk individually
        for chunk in chunks:
            beta.append(int(chunk, 2) * aux % p)
        return alfa, beta
    beta.append(m * aux % p)
    return alfa, beta
@
```

3. Decryption

3.1. Using the private key a, compute the original message by calculating $m = \alpha^{-a} \cdot \beta \bmod p$.

From the way we have represented our message as a number, we need to have a way to reproduce it with characters. Hence, I have designed an inverse function, which takes in a binary string, splits it into bytes(chunks of 8) and represents their associated character value.

```
<<BinToString>>=
def BinToString(msg):
    # gets a binary string and splits it into chunks of 8
    # and then transforms each byte into the corresponding letter
    m = ""
    # adds 0 in front if the given length is not divisible by 8
    while len(msg) % 8 != 0:
        msg = "0" + msg
    for i in range(0, len(msg), 8):
        aux = msg[i:i + 8]
        aux = int(aux, 2)
        aux = chr(aux)
        m = m + aux
    return m
@
```

The full decryption algorithm is below:

```
<<decrypt>>=
def decrypt(alfa, beta, p, a):
    # gets the inverse mod p of alfa
    inv = inverseModN(alfa, p)
    #calculates alfa ^ -a mod p
    newAlfa = power(inv, a, p)
    # if there was no need for chunking
    if len(beta) == 1:
        m = newAlfa * beta[0]
        # gets the original message and reduce it mod p
        m = m % p
        m = str(bin(m))[2:]
        # converts m to binary and splits it in chunks of 8 bits
        # each chunk represents a character, hence the original message is decoded and repr
        msg = BinToString(m)
        return msg
    else:
        # if the message was split into chunks
        # decrypt each chunk individually, turn it into binary
        # and concatenate it to get the original message
        # in binary representation
        msg = ""
        for i in range(len(beta)):
            m = newAlfa * beta[i]
            m = m % p
            aux = str(bin(m))[2:]
            # if the binary representation isnt a multiple of 8, add 0's in front
            while len(aux) % 8 != 0:
                aux = "0" + aux
            msg = msg + aux
        return BinToString(msg)
@
```

The algorithm for the modular multiplicative inverse can be found in the Annexe.

Theorem

The ElGamal Algorithm is correct.

Proof

We have $m = \alpha^{-a} \cdot \beta = g^{-a \cdot k} \cdot m \cdot (g^a)^k = m$.

Annexe

1. Miller-Rabin primality test

The Miller–Rabin primality test is a probabilistic primality test: an algorithm which determines whether a given number is likely to be prime or is definitely composite. The Miller-Rabin test relies on the following result:

Theorem

Let p be a prime. Then the equation

$a^2 \equiv 1 \bmod p$

has only the solutions $a \equiv 1 \bmod p$ and $a \equiv -1 \bmod p$.

Proof

We may assume that $a \in \{0, \dots, p-1\}$.

We have $a^2 \equiv 1 \bmod p \leftrightarrow p|(a-1) \cdot (a+1)$.

It follows that $p|a-1$ or $p|a+1$. If $p|a-1$, then $a-1=0$, because $a-1<p$. Hence $a=1$.

If $p|a+1$, then $a+1=0$ or $a+1=p$, because $a+1<p+1$. Hence $a=p-1=-1$.

The algorithm:

Step 0. Write $n-1=2^s \cdot t$, where t is odd.

Step 1. Choose (randomly) $1<a<n$.

Step 2. Compute (by the repeated squaring modular exponentiation) the following sequence (modulo n):

$a^t, a^{2 \cdot t}, a^{2^2 \cdot t}, \dots, a^{2^s \cdot t}$.

Explanation:

By Fermat's Little Theorem, $a^{2^s \cdot t} \equiv 1 \bmod n$.

Earch term of the sequence $a^{2^s \cdot t}, \dots, a^{2 \cdot t}, a^t$ is a square root of the previous term. Since the first term is congruent to 1, the second term is a square root modulo n of 1. By the previous theorem, it is congruent to either 1 or -1. If it is congruent to -1, we are done. Otherwise, it is congruent to 1 and we can iterate the reasoning. At the end, either one of the terms is congruent to -1, or all of them are congruent to 1, and in particular the last term, a^t, is.

Step 3. If either the first number in the sequence is 1 or if one gets the value 1 and its previous number -1, then n is possible to be prime and one repeats Steps 1-3 at most k times. If one does not get to Step 4, then the algorithm stops and n is probable prime.

Step 4. The algorithm stops and n is composite.

```
<<millerRabin>>=
def millerRabin(n):
    # Implementation of the Miller - Rabin test
    # with k = 20 steps, thus producing an error of 1/4^20
    s = 0
    t = n - 1
    while t & 1 == 0:
        # trying to write n - 1 = 2^s * t
        # building s
        t = t // 2
        s = s + 1
    k = 20
    for i in range(k):
        a = rand.randint(2, n - 1)
        x = power(a, t, n)
        # checking if the first element is 1 or -1
        if x == 1 or x == n - 1:
            continue
        for j in range(s):
            x = pow(x, 2) % n
            # if we get a -1, the next value will be a 1, so the algorithm ends
            # and tries with another base
            if x == n - 1:
                continue
        return False
    return True

@
```

Remark: If the algorithm gives the answer PRIME, then the probability of correct answer is $1 - 1/4^k$, where k is the number of repetitions.

2. Modular multiplicative inverse

In mathematics, particularly in the area of number theory, a modular multiplicative inverse of an integer a is an integer x such that the product $a \cdot x$ is congruent to 1 with respect to the modulus m. In the standard notation of modular arithmetic this congruence is written as

$a \cdot x \equiv 1 \mod m$.

```
<<inverseModN>>=
def inverseModN(a, b):
    # calculates the inverse of a mod b
    return 0 if a == 0 else 1 if b % a == 0 else b - inverseModN(b % a, a) * b // a
@
```

This algorithm returns the Bezout coefficient such that $a \cdot x \equiv gcd(a, m)$, but our moduli being a prime number, that $gcd(a, m) = 1$, hence it will return the modular multiplicative inverse of a mod m.

3. Repeated squaring modular exponentiation

The idea

Let us compute b^k mod n, where $b, k \in \mathbb{N}$ are large.

Write k in binary, say $\sum_{i=1}^{t} k_i \cdot 2^i$, where $k_i \in \{0, 1\}$.

Then, we have $b^k = \prod_{i=0}^{t} b^{k_i \cdot 2^i} = (b^{2^0})^{k_i} \cdot \cdots \cdot (b^{2^t})^{k_t}$.

```
<<repeatedSqModExp>>=
def power(a, b, c):
    # Implementation of repeated squaring modular exponentiation
    x = 1
    y = a
    while b > 0:
        if b & 1 != 0:
            x = (x * y) % c
        y = (y * y) % c
        b = int(b >> 1)
    return x % c
@
```

4. Other functions for the user input and the main method of the program

```
<<main>>=
def menu():
    #Prints a interactive menu
    print("1. Generate keys")
    print("2. Show keys")
    print("3. Encrypt")
    print("4. Decrypt")
    print("5. Show available ciphers")
    print("6. Exit")

def main():
    # main function of the program
```

```
keys = [] # list of generated keys
ciphers = [] # list of generated ciphers
opt = ""
while opt != "6":
    menu()
    opt = input("Option: ")
    if opt == "1":
        ok = False
        name = input("Name : ")
        for key in keys:
            if key[0] == name:
                print("That name already exists!")
                ok = True
        if ok:
            continue
        print("Generating keys...This shouldn't take long.")
        publ, priv = gen_keys()
        keys.append((name, publ, priv))
        print("Public key : (" + str(publ[0]) + ", " + str(publ[1]) + ", " + str(publ[2]
        print("Private key : " + str(priv))
    elif opt == "2":
        for elem in keys:
            print("Name : " + str(elem[0]))
            print("Public key : (" + str(elem[1][0]) + ", " + str(elem[1][1]) + ", " + s
            print("Private key : " + str(elem[2]))
    elif opt == "3":
        name = input("Enter name from whom to take public key: ")
        p = 0
        g = 0
        ga = 0
        ok = False
        for key in keys:
            if key[0] == name:
                p = key[1][0]
                g = key[1][1]
                ga = key[1][2]
                ok = True
                break
        if not ok:
            print("That person doesnt have any keys generated!")
            continue
        msg = input("Enter message: ")
        c1, c2 = encrypt(msg, p, g, ga)
        ciphers.append((c1, c2, name))
        print("Public key used: (" + str(p) + ", " + str(g) + ", " + str(ga) + ")")
        print("Cipher-text : (" + str(c1) + ", " + str(c2) + ")")
```

```
        elif opt == "4":
            index = int(input("Give the index for the cipher you want to decrypt(the number
            if index >= len(ciphers) or index < 0:
                print("Incorrect index!")
                continue
            cipher = ciphers[index]
            name = cipher[2]
            p = 0
            a = 0
            for key in keys:
                if key[0] == name:
                    p = key[1][0]
                    a = key[2]
                    break
            print("Private key used : " + str(a))
            print("Key belongs to : " + name)
            msg = decrypt(cipher[0], cipher[1], p, a)
            print("Original message: " + msg)
        elif opt == "5":
            i = 0
            for cipher in ciphers:
                print("c" + str(i) + " = (" + str(cipher[0]) +", " + str(cipher[1]) + ") - e
                i = i + 1
        elif opt == "6":
            print("Bye bye!")
        elif opt != "6":
            print("Wrong option!")


if __name__ == "__main__":
    main()

@

<<*>>=
import random as rand
<<repeatedSqModExp>>
<<millerRabin>>
<<genLowLevelPrime>>
<<genRandPrime>>
<<StringToBin>>
<<BinToString>>
<<breakIntoChunks>>
<<encrypt>>
<<decrypt>>
<<main>>
```

@