The El Gamal Cryptosystem

Lab Assignment 4 - Public Key Cryptography, UBB-CS Year 3

Popa Alex Ovidiu, group 936

Problem Statement: Given an alphabet consisting of the 26 letters of the English alphabet and the blank space, generate a public and a private key and use them to encrypt/decrypt a given text using the El Gamal cipher, in its basic version.

Proposed Solution:

1. Implement a key generator to create the necessary variables.
2. Implement a main function which encrypts and decrypts a text.
3. Implement a tester function to test the encryption algorithm against randomly generated text sequences.

The implementation follows the steps presented in prof. Crivei's seventh lecture, namely:

1. Generate a public and private key
2. Encrypt using the public key
3. Decrypt using the private key

Before starting the encryption process, the program checks the input message to make sure it only contains valid characters, and by valid we mean those which are in the alphabet.

The public key consists of a large prime number **p**, a generator **g** of the cyclic group $\mathbb{Z}_p^*$ and **g^a mod p**, where **a** is a random integer from the interval [1, p-2] and is equal to the private key.
For the purpose of the exercise, I've implemented a modular exponentiation function even though the python pow is significantly faster as far as I've read. Because a super-computer is not that easily obtainable, the prime numbers I generate go up to 10^7, as for 10^8 it already takes a good minute to run, and that is unfortunately not enough to encrypt larger messages without splitting them into chunks. However, in theory, very large prime numbers are relatively easy to get and that's why El Gamal's algorithm does not involve message splitting.

Before encrypting and before decrypting, we need to convert the message into a number and then a number into a text, because the algorithm works with numbers, so we need to make the first conversion, and at the end we expect another message, so we need to make the second one.
When we encrypt, we select another random number k (from the same interval as above), and create a pair (α, β) which will be the cipher text.
When we decrypt, we apply the formula from the lecture, and in it lies the proof of correctness of the algorithm: if we replace alpha and beta in the formula, that -a at the power of alpha causes the gˆ(a*k) given by beta to cancel out with the other one given by alpha which will have a minus, and we are left with the message m.
This is why it is very hard to hack the message without knowing the private key (a), and why El Gamal is (for now) safe and widely used.

Having said all of this, I will now start showing each function in its corresponding code chunk.
We declare the alphabet as a constant string consisting of all the characters in the requirement, namely the English alphabet and the blank space. We also keep track of our desired chunk length, meaning the length of character blocks, and the length of the alphabet.

```
<<Constants>>=
import random

ALPHABET = ' abcdefghijklmnopqrstuvwxyz'
CHUNK_LENGTH = 3
ALPHABET_LENGTH = len(ALPHABET)

@
```

Plaintext validation-check that all the characters belong to the alphabet.

```
<<Validate>>=
<<Constants>>

def validate(message):
    for char in message:
        if char not in ALPHABET:
            return False
    return True
@
```

Function which generates the prime numbers up to a bound using the sieve of Eratosthenes, to reduce the computational time as much as we can.
The first 10000 primes are discarded to ensure that the chunks don't overflow, and we get a relatively large prime (relative to our computational power, that is).

```
<<Sieve>>=
```

```
def sieve(bound):
    ans = []
    marked = [False for _ in range(bound + 1)]
    for i in range(2, bound + 1):
        if not marked[i]:
            ans.append(i)
            for j in range(i + i, bound + 1, i):
                marked[j] = True
    return ans[10000:]
```

@

Function which returns a random prime number up to a bound.

```
<<GetPrime>>=
<<Sieve>>
<<Constants>>

def get_random_prime(bound):
    primes = sieve(bound)
    return random.choice(primes)
```

@

Basic gcd implementation needed for the generators of a cyclic group.

```
<<Gcd>>=

def gcd(a, b):
    while b:
        r = a % b
        a = b
        b = r
    return a
```

@

Function which returns a list consisting of all the generators of a cyclic group Zn.
By definition, the generators of a cyclic group Zn are all the numbers in Zn which are coprime with n, i.e. gcd(n, x) = 1.

```
<<Generators>>=
<<Gcd>>
def generators(n):
    gens = []
    for g in range(2, n):
        if gcd(g, n) == 1:
```

```
            gens.append(g)
    return gens
```

@

Function which chooses a random generator from the list.

```
<<GetGenerator>>=
<<Constants>>
<<Generators>>

def get_random_generator(n):
    return random.choice(generators(n))
```

@

Function which chooses a random number in the interval [1, p-2].

```
<<GetInt>>=
<<Constants>>

def get_int_less_than(p):
    return random.randint(1, p - 2)
```

@

Repeated squaring modular exponentiation algorithm, following the steps from prof. Crivei's second lecture.

```
<<Pow>>=

def modular_exp(x, y, p):
    res = 1
    x = x % p
    if x == 0:
        return 0
    # bit shifting is safer for large numbers, when possible
    while y > 0:
        if (y & 1) == 1:
            res = res * x % p
        y = y >> 1  # y = y//2
        x = x * x % p

    return res
```

@

Key generator function, returns a tuple consisting of the public and private keys.

```
<<KeyGen>>=
```

```
<<GetPrime>>
<<GetInt>>
<<GetGenerator>>
<<Pow>>

def keys_generator():
    p = get_random_prime(10 ** 7)
    print("generated prime is=" + str(p))
    g = get_random_generator(p)
    print("generator g is=" + str(g))
    a = get_int_less_than(p)  # private key
    ga = modular_exp(g, a, p)
    print("g^a is=" + str(ga))
    public_key = (p, g, ga)
    return public_key, a

@
```

Conversion functions, from message to digits and vice versa, which is nothing more than converting from/to basis 27, thanks to some easy arithmetic.

```
<<Conversions>>=
<<Constants>>

def convert_characters_to_number(characters):
    number = 0
    power = 1
    for char in reversed(characters):
        number += power * (ALPHABET.find(char) + 1)
        power *= (ALPHABET_LENGTH+1)
    return number


def convert_number_to_characters(number):
    characters = ""
    while number:
        characters = ALPHABET[number % (ALPHABET_LENGTH+1) - 1]+characters
        number//=(ALPHABET_LENGTH+1)
    return characters

@
```

Test suite, consisting of two functions, the first one which randomly generates test cases of length between 5 and 10, and the second one which performs the actual encryption/decryption process on them, and asserts whether the end result is equal to the test case.

```
<<Tests>>=
```

```
<<Constants>>
<<KeyGen>>
<<Conversions>>
<<GetInt>>
<<Pow>>

def generate_random_text():
    tests = []
    for i in range(25):
        length = random.randint(5,10)
        message = ""
        for _ in range(length):
            message += random.choice(ALPHABET)
        tests.append(message)
    return tests


def test():
    test_cases = generate_random_text()
    print(test_cases)
    public_key, private_key = keys_generator()
    print("public key=" + str(public_key))
    print("private key=" + str(private_key))
    p = public_key[0]
    g = public_key[1]
    ga = public_key[2]
    a = private_key
    for test_message in test_cases:

        decrypted_text = ""
        chunks = [test_message[i:i + CHUNK_LENGTH] for i in range(0, len(test_message), CHUN
        for c in chunks:
            nr = convert_characters_to_number(c)

            k = get_int_less_than(p)
            alpha = modular_exp(g, k, p)
            beta = nr * modular_exp(ga, k, p) % p

            decrypted_number = modular_exp(alpha, p - 1 - a, p) * beta % p
            decrypted_text += convert_number_to_characters(decrypted_number)

        print(test_message, decrypted_text)
        assert decrypted_text == test_message

@
```

Main function, here is where we interpret the command line arguments, generate the keys, encrypt/decrypt the given message and assert whether the two messages (initial one and resulted one) are the same.

```
<<*>>=
<<Validate>>
<<Tests>>

def main():
    import sys
    args = sys.argv[1:]
    initial_message = ""
    if not args:
        initial_message = "el gamal"
    elif args[0] == "-m":
        initial_message = " ".join(args[1:])
    elif args[0] == "-t":
        print("el gamal tests starting")
        test()
        print("el gamal tests passed")
        return

    print("message:" + initial_message)

    if not validate(initial_message):
        print("Invalid characters in input message. Please only use lowercase alphabet and s
        assert False

    public_key, private_key = keys_generator()
    print("public key=" + str(public_key))
    print("private key=" + str(private_key))
    p = public_key[0]
    g = public_key[1]
    ga = public_key[2]
    a = private_key

    decrypted_text = ""
    chunks = [initial_message[i:i + CHUNK_LENGTH] for i in range(0, len(initial_message), CH
    for c in chunks:
        nr = convert_characters_to_number(c)
        k = get_int_less_than(p)
        alpha = modular_exp(g, k, p)
        beta = nr * modular_exp(ga, k, p) % p

        decrypted_number = modular_exp(alpha, p - 1 - a, p) * beta % p
        decrypted_text += convert_number_to_characters(decrypted_number)
```

```
    print("decrypted message:" + decrypted_text)
    assert decrypted_text == initial_message

main()
```

@

Runner command example:

```
notangle elgamal.md > elgamal.py && python elgamal.py -m life is good
notangle elgamal.md > elgamal.py && python elgamal.py -t
```

Use -m to give a message separated by spaces, use -t to run the tests. Message defaults to “el gamal” if one omits both flags, and the tests aren’t run. Computing the prime p and the generator g takes a few seconds, so it will take a bit until the keys are generated and you’ll see the result.