COMP26120
Academic Session: 2018-19

Lab Exercise 13: Public-key cryptosystems

Duration: 2 lab sessions.

For this lab exercise you should do all your work in your COMP26120/ex13 directory.

Introduction

In the lectures, you have learned/will learn about modular arithmetic and fast modular exponentiation. You also learned about the El Gamal public-key cryptosystem. In this lab, you will implement a proof-of-concept system. The system as it stands is not very secure; nor do we tackle the issue of deployment in practical situations. But these restrictions are, from a conceptual point of view, inessential. If you wish, you can always build a usable system of your own for fun.

For tasks 1, 2, 4 and 5, you should write a program in the file *elGamal.c*. For task 3, you should put your program demonstrating discrete logarithms in the file *dl.c* For task 6, you should write a program in the file *elGamalGMP.c*.

Description

Task 1: highest common factor

Let a and b be positive integers. Implement a function

```
unsigned long hcf(unsigned long a,unsigned long b)
```

to compute the highest common factor of a and b. Your program must run in time linear in the total size of the inputs, a and b. (Remember: the size of a positive integer a is the number of bits required to write it in standard notation, namely, $\lfloor \log a \rfloor + 1$.)

Task 2: fast modular exponentiation

Let p be a prime, and g a primitive root modulo p. Implement a function

```
unsigned long fme(unsigned long g,unsigned long x, unsigned long p)
```

to compute the function $x \to g^x \bmod p$. What is the running time of your program as a function of the sizes of g, x and p? Write your answer, with justification, as a comment to the function.

Task 3: discrete logarithm

Let p be a prime, and g a primitive root modulo p. Implement a function

```
unsigned long dl(unsigned long y,unsigned long g,unsigned long p)
```

to compute, for a given y $(1 \leq y < p)$ the unique integer x $(1 \leq x < p)$ such that $y = g^x \bmod p$. What is the running time of your program as a function of the sizes of y, g and p? Write your answer, with justification, as a comment to the function. As an additional comment in the code, explain why modular exponentiation can be regarded as a *one-way* function.

Task 4: inverse modulo prime

Let p be a prime. Implement a function

```
unsigned long imp(unsigned long y, unsigned long p)
```

which returns, for a given y $(1 \leq y < p)$ the unique integer x $(1 \leq x < p)$ such that $x \cdot y \equiv 1 \bmod p$. There are actually two sensible ways to do this: either using exercise 1 (if you implemented it in a certain way) or using exercise 2. Either way, you have to think a little bit about what you are doing, but I recommend using exercise 2. What is the running time of your program as a function of the sizes of y and p? Write your answer, with justification, as a comment to the function.

Task 5: El Gamal system

Now for the business end. Write a program that prints out the following message plus list of user-options:

```
Prime modulus is 65537
Primitive root wrt 65537 is 3
Choose: e (encrypt) | d (decrypt) | k (get public key) | x (exit)?
```

You will see that p and g have been set at fixed values. A simple check will confirm that $g = 3$ is indeed a primitive root of $p = 65537$. If the user selects k, he is prompted for a private key x in the range $1 \leq x < p$. The program then computes the El Gamal public key $y = g^x \bmod p$. The program goes back to the prompt. Here is the an example of the expected behaviour.

```
Choose: e (encrypt) | d (decrypt) |k (get public key) | x (exit)? k
Type private key: 40000
Public key is: 64675
```

If the user selects e, he is prompted for a message M in the range $1 \leq M < p$ and the public key y (as generated by the k-option). The program then selects a random number k in the range $1 \leq x < p$ and computes the El Gamal encryption of M as the integer pair (a, b), where

$$a = g^k \bmod p \; b = M \cdot y^k \bmod p,$$

where y is the public key. The program goes back to the prompt. Here is an example of the expected behaviour.

```
Choose: e (encrypt) | d (decrypt) |k (get public key) | x (exit)? e


Type secret number to send: 11121
Type recipientâĂŹs public key: 64675


The encrypted secret is: (53509, 46390)
```

If the user selects d, he is prompted for a cypher-text C of the form (a, b), with $1 \leq a, b < p$, and the *private* key x entered above. (Remember that x was used to generate the public key by means of which C was computed from some plain-text.) The program then decodes C so as to recover the plain-text. The program goes back to the prompt. Here is the an example of the expected behaviour.

```
Choose: e (encrypt) | d (decrypt) |k (get public key) | x (exit)? d


Type in received message in form (a,b): (53509,46390)
Type in your private key: 40000


The decrypted secret is: 11121
```

Task 6: working with arbitrary integers

Some people find that data-types like long or unsigned long long cramp their style. They would prefer to work with arbitrary integers. No problem: they can use the GNU Multiple Precision Arithmetic library. It makes available data-types such as mpz t which can store integers of *any* size.

You will need to have the gmp library in a directory that is on your LD_LIBRARY_PATH, and your program must contain the line `#include <gmp.h>`. You will then be able to declare, initialize and manipulate arbitrary precision integers, for example:

```c
mpz_t a, b, ans; // Declarations of variables
mpz_init(a); // Initialization analogous to malloc
mpz_init(b);
mpz_init(ans);
mpz_set_str(p,"2",10); // Set p=2 and g=3 (base 10)
mpz_set_str(g,"3",10);
mpz_mul(ans,a,b); // Let ans be a * b
gmp_printf("The answer to this hard problem is: %Zd\n",ans);
```

Figure 1: Example code output

The program is compiled with a line such as

```
gcc -lm -lgmp myProgram.c
```

so that the gmp library is linked.

You will need to use the documentation to work out how to use the library. (It is actually exceptionally well-written.) Demonstrate that the program works when p is set to

170141183460469231731687303715884105727

whose smallest primitive root is 43.

This part of the exercise is not for the faint-hearted and can be viewed as an extension activity with few marks attached.

Hints

For parts 1-5, you may have problems with number overflows. The data type unsigned long long may (or may not) solve them. However, your programs are not required to work for primes larger than 4093082899. For testing, I recommend using very small primes. Here is a table of a few primes *p* with primitive roots modulo *p*. (Don't try the last one unless you are using the GMP library in part 6.)

p	primitive root modulo p
17	3
257	3
443	2
65537	3
4093082899	2
170141183460469231731687303715884105727	43

For part 6, your program is required to work with integers of arbitrary sizes. Get parts 1-5 working first. Then make a copy and re-factor to use the GMP library.