Elliptic Curve Factorization

Tolga Gölbaşı

120400002006

Department of Computer Engineering

Faculty of Engineering

Advisor: Hüseyin Hışıl

September 2014

Keywords

Factorization, Elliptic Curves, GMP

Declaration

The work contained in this thesis has not been previously submitted for a degree or diploma at any higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed: ..... **Date:** .....

Öz

General Number Field Sieve'deki sieving kısmında hızlandırmak için eliptik eğriler kullanılmaktadır. Bu tezde iki ayrı methoda bakacağız. İlk methot Lenstra'nın 1988'de geliştirdiği eliptik eğri methodu. İkincisi ise Montgomery'nin geliştirmiş olduğu eliptik eğri methodu. Bu methodların nasıl çalıştığı ve GMP üzerinde nasıl implemente edilebileceğini göstereceğiz.

Abstract

Hardness of solving factorization problems are still the main power of our cryptographic protocols we use in everyday life. The problem won't be getting easier even with Quantum Computers developed since it is out of their scope. Elliptic Curves are used in sieving part of Factorization using General Number Field Sieve. In this thesis we look at two methods for Elliptic Curve Factorization. First method is Lenstra's Elliptic Curve Method which is created by Hendrik Lenstra in 1988. Second method is Montgomery 's Elliptic Curve Method created by Peter Montgomery. We will be showing how they work and how they can be implemented with GMP.

Acknowledgements

The acknowledgements goes here.

Contents

List of Figures

List of Codes

Chapter 1

Project Statement

1.1 Motivation

In a world where information getting more precious day by day vital part of the security of the information is cryptographic systems. As a student of Cyber Security Master's Program i wanted to learn how i can implement such systems myself using world-wide accepted tools. I have choosen Elliptic Curve Method because of its parallelizable first stage and further improvements coming from better implementation of it. Also using GMP library and C programming with using a software versioning and revision control system such as SVN gave me a real world idea of working on projects other than loose schoolwork. Furthermore working with Elliptic Curve Cryptography gave me the understanding of working with coding tools that are reliable and supported well by the community and companies. At start of my work i started working with OpenCL graphics programming language. But over time i realized that OpenCL wasn't well supported as i thought it was and it was hard to get help online on my mistakes or compilers shortcomings.

1.2 Outline

This thesis is divided into 7 chapters. Chapter 1 gives an introduction and my motivation for working for this thesis and also understanding about why cryptographic systems are important and vital to our communities. Chapter 2 presents the foundations of well understood Elliptic Curve Method. Chapter 3 is an introduction to using GMP library which used heavily in this thesis. Chapter 4 is focused on implementation of Elliptic Curve Method with regular curves and improved version using Montgomery Curves. Chapter 5 is the section where methodology and results of our implementation are given. In Chapter 6 we present our results for testing the implementation. Chapter 7 summarizes the main conclusions of the thesis and presents future work.

1.3 Introduction / Background

Factorization problem is the problem of finding factors of numbers. If you are not familiar with factoring it might sound pretty easy for the numbers we work with in our daily lives. But factorization of mostly the big numbers are really hard problem to deal with. And these hardness today makes us much more secure since a lot of cryptographic protocols depends on the premise that no one can break this protocol by finding factors of our primes. Earliest factoring aids were tables of prime factors published back of math texts in 1600s. For numbers out of scope of the tables new techniques had to be developed. Simplest factorization method is trial division. Also trial division works very fast for huge percentage of numbers. Because most of the numbers has small prime factors. 50 percentage of the numbers divided by 2! Of course trial division method can be extremely slow for numbers with very high primes. Fermat's Algorithm and Euler's Algorithm did further improvements on factorization but it was obvious that factorization were still extremely hard problem without the aid of machines. D.H. Lehmer built two machines. First one is called bicycle-chain sieve. The machine consisted of bicycle chains and electrical switches when all the electrical switches close at the same time it created a electrical circuit and a solution was found. Second one was Photoelectric Number Sieve consisted of 16 mm film with holes in them replacing the bicycle chains in the old model. Also in those days it was a common practice to use computers for factorization in their idle

time. And factorization problems intensity on computers were useful for IBM engineers as they found hardware problems on computers that standard tests could not. With the emergence of the public key cryptography research on factorization intensified. Pollard's rho method Pollard's p-1 method Sub exponential methods Qua

1.4 Aim and objectives

Aim and objectives go here. Mandatory section.

1.5 Deliverables

Deliverables go here. Mandatory section.

1.6 Roadmap

Roadmap goes here. Explain how the thesis is structured. Optional section.

1. Item 1
2. Item 2
3. Item 3

Chapter 2

Elliptic Curve Method

Elliptic Curve Method goes here.

2.1 Elliptic Curve

Text goes here.

2.1.1 The group law

Text goes here.

2.1.2 Elliptic Curve Over Finite Fields

Text goes here.

2.2 Elliptic Curve Method

Text goes here.

2.2.1 Stage One

Text goes here.

2.2.2 Stage Two

Text goes here.

Chapter 3

Introduction to GMP

3.1 Introduction to GMP

GNU MP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types.

Many applications use just a few hundred bits of precision; but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum.

The speed of GMP is achieved by using fullwords as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs, and by a general emphasis on speed (as opposed to simplicity or elegance).

There is assembly code for these CPUs: ARM Cortex-A9, Cortex-A15, and generic ARM, DEC Alpha 21064, 21164, and 21264, AMD K8 and K10 (sold under many brands, e.g. Athlon64, Phenom, Opteron) Bulldozer, and Bobcat, Intel Pentium, Pentium Pro/II/III, Pentium 4, Core2, Nehalem, Sandy bridge, Haswell, generic x86, Intel IA-64, Motorola/IBM

PowerPC 32 and 64 such as POWER970, POWER5, POWER6, and POWER7, MIPS 32-bit and 64-bit, SPARC 32-bit ad 64-bit with special support for all UltraSPARC models. There is also assembly code for many obsolete CPUs.

For up-to-date information on GMP, please see the GMP web pages at

https://gmplib.org/ The latest version of the library is available at

https://ftp.gnu.org/gnu/gmp/

3.2 GMP Basics

3.1 Headers and Libraries

All declarations needed to use GMP are collected in the include file gmp.h. It is designed to work with both C and C++ compilers.

```
1    #include <gmp.h>
```

Code 3.1:

Note however that prototypes for GMP functions with FILE * parameters are only provided if ¡stdio.h¿ is included too.

```
1    #include <stdio.h>
2    #include <gmp.h>
```

Code 3.2:

All programs using GMP must link against the libgmp library. On a typical Unix-like system this can be done with '-lgmp', for example gcc myprogram.c -lgmp

3.2 Nomenclature and Types

Integer usually means a multiple precision integer, as defined by the GMP library. The C data type for such integers is mpz_t. Here are some examples of how to declare such integers:

```
1    mpz_t sum;
2
3    struct foo { mpz_t x, y; };
4
5    mpz_t vec[20];
```

Code 3.3:

A limb means the part of a multi-precision number that fits in a single machine word. Normally a limb is 32 or 64 bits. The C data type for a limb is mp_limb_t.

Counts of limbs of a multi-precision number represented in the C type mp_size_t.

Also, in general mp_bitcnt_t is used for bit counts and ranges, and size_t is used for byte or character counts.

3.3 Function Classes

These are some of classes of functions in the GMP library:

Functions for signed integer arithmetic, with names beginning with mpz_. The associated type is mpz_t. There are about 150 functions in this class. Functions for rational number arithmetic, with names beginning with mpq_. The associated type is mpq_t. There are about 35 functions in this class, but the integer functions can be used for arithmetic on the numerator and denominator separately. Functions for floating-point arithmetic, with names beginning with mpf_. The associated type is mpf_t. There are about 70 functions is this class. Fast low-level functions that operate on natural numbers. These functions' names begin with mpn_. The associated type is array of mp_limb_t. There are about 60 (hard-to-use) functions in this class. Miscellaneous functions. Functions for setting up custom allocation and functions for generating random numbers.

3.4 Variable Conventions

GMP functions generally have output arguments before input arguments. This notation is by analogy with the assignment operator.

GMP lets you use the same variable for both input and output in one call. For example, the main function for integer multiplication, mpz_mul, can be used to square x and put the result back in x with

mpz_mul (x, x, x); Before you can assign to a GMP variable, you need to initialize it by calling one of the special initialization functions. When you're done with a variable, you need to clear it out, using one of the functions for that purpose. Which function to use depends on the type of variable.

A variable should only be initialized once, or at least cleared between each initialization. After a variable has been initialized, it may be assigned to any number of times.

For efficiency reasons, avoid excessive initializing and clearing. In general, initialize near the start of a function and clear near the end. For example,

```
1     void
2     foo (void)
3     {
4       mpz_t n;
5       int   i;
```

```
6       mpz_init (n);
7       for (i = 1; i < 100; i++)
8         {
9           mpz_mul (n, ...);
10          mpz_fdiv_q (n, ...);
11          ...
12        }
13      mpz_clear (n);
14    }
```

Code 3.4:

3.5 Parameter Conventions

When a GMP variable is used as a function parameter, it's effectively a call-by-reference, meaning if the function stores a value there it will change the original in the caller.

When a function is going to return a GMP result, it should designate a parameter that it sets, like the library functions do. More than one value can be returned by having more than one output parameter, again like the library functions. A return of an mpz_t etc doesn't return the object, only a pointer, and this is almost certainly not what's wanted.

Here's an example accepting an mpz_t parameter, doing a calculation, and storing the result to the indicated parameter.

```
1     void
2     foo (mpz_t result, const mpz_t param, unsigned long n)
3     {
4       unsigned long i;
5       mpz_mul_ui (result, param, n);
6       for (i = 1; i < n; i++)
7         mpz_add_ui (result, result, i*7);
8     }
9
10    int
11    main (void)
12    {
13      mpz_t r, n;
14      mpz_init (r);
15      mpz_init_set_str (n, "123456", 0);
16      foo (r, n, 20L);
17      gmp_printf ("%Zd\n", r);
18      return 0;
19    }
```

Code 3.5:

3.6 Memory Management

The GMP types like mpz_t are small, containing only a couple of sizes, and pointers to allocated data. Once a variable is initialized, GMP takes care of all space allocation. Additional space is allocated whenever a variable doesn't have enough.

mpz_t and mpq_t variables never reduce their allocated space. Normally this is the best policy, since it avoids frequent reallocation. Applications that need to return memory to the heap at some particular point can use mpz_realloc2, or clear variables no longer needed.

All memory is allocated using malloc and friends by default, but this can be changed, see Custom Allocation. Temporary memory on the stack is also used (via alloca), but this can be changed at build-time if desired, see Build Options.

3.11 Efficiency

Small Operands On small operands, the time for function call overheads and memory allocation can be significant in comparison to actual calculation. This is unavoidable in a general purpose variable precision library, although GMP attempts to be as efficient as it can on both large and small operands.

Initializing and Clearing Avoid excessive initializing and clearing of variables, since this can be quite time consuming, especially in comparison to otherwise fast operations like addition.

Reallocations An mpz_t or mpq_t variable used to hold successively increasing values will have its memory repeatedly realloced, which could be quite slow or could fragment memory, depending on the C library. If an application can estimate the final size then mpz_init2 or mpz_realloc2 can be called to allocate the necessary space from the beginning (see Initializing Integers). It doesn't matter if a size set with mpz_init2 or mpz_realloc2 is too small, since all functions will do a further reallocation if necessary. Badly overestimating memory required will waste space though.

mpz_mul is currently the opposite, a separate destination is slightly better. A call like mpz_mul(x,x,y) will, unless y is only one limb, make a temporary copy of x before forming the result. Normally that copying will only be a tiny fraction of the time for the multiply, so this is not a particularly important consideration.

Divisibility Testing (Small Integers) mpz_divisible_ui_p and mpz_congruent_ui_p are the best functions for testing whether an mpz_t is divisible by an individual small integer. They use an algorithm which is faster than mpz_tdiv_ui, but which gives no useful information about the actual remainder, only whether it's zero (or a particular value). However when testing divisibility by several small integers, it's best to take a remainder modulo their product, to save multi-precision operations.

The division functions like mpz_tdiv_q_ui which give a quotient as well as a remainder are generally a little slower than the remainder-only functions like mpz_tdiv_ui. If the quotient is only rarely wanted then it's probably best to just take a remainder and then go back and

calculate the quotient if and when it's wanted (mpz_divexact_ui can be used if the remainder is zero).

3.3 Installing GMP

GMP has an autoconf/automake/libtool based configuration system. On a Unix-like system a basic build can be done with

./configure make Some self-tests can be run with

make check And you can install (under /usr/local by default) with

make install

CPU types In general, if you want a library that runs as fast as possible, you should configure GMP for the exact CPU type your system uses. However, this may mean the binaries won't run on older members of the family, and might run slower on other members, older or newer. The best idea is always to build GMP for the exact machine type you intend to run it on.

Generic C Build If some of the assembly code causes problems, or if otherwise desired, the generic C code can be selected with the configure –disable-assembly. Note that this will run quite slowly, but it should be portable and should at least make it possible to get something running if all else fails.

Fat binary, –enable-fat Using –enable-fat selects a fat binary build on x86, where optimized low level subroutines are chosen at runtime according to the CPU detected. This means more code, but gives good performance on all x86 chips.

FFT Multiplication, –disable-fft By default multiplications are done using Karatsuba, 3-way Toom, higher degree Toom, and Fermat FFT. The FFT is only used on large to very large operands and can be disabled to save code size if desired.

In particular for long-running GMP applications, and applications demanding extremely large numbers, building and running the tuneup program in the tune subdirectory, can be important. For example,

cd tune make tuneup ./tuneup will generate better contents for the gmp-mparam.h parameter file.

3.4 Integer Functions

The functions for integer arithmetic assume that all integer objects are initialized. You do that by calling the function mpz_init. For example,

```
{
  mpz_t integ;
  mpz_init (integ);
  ...
  mpz_add (integ, ...);
  ...
  mpz_sub (integ, ...);

  /* Unless the program is about to exit, do ... */
  mpz_clear (integ);
}
```

Code 3.6:

As you can see, you can store new values any number of times, once an object is initialized.

Function: void mpz_init (mpz_t x) Initialize x, and set its value to 0.

Function: void mpz_inits (mpz_t x, ...) Initialize a NULL-terminated list of mpz_t variables, and set their values to 0.

Function: void mpz_clear (mpz_t x) Free the space occupied by x. Call this function for all mpz_t variables when you are done with them.

Function: void mpz_clears (mpz_t x, ...) Free the space occupied by a NULL-terminated list of mpz_t variables.

Function: void mpz_realloc2 (mpz_t x, mp_bitcnt_t n) Change the space allocated for x to n bits. The value in x is preserved if it fits, or is set to 0 if not.

These functions assign new values to already initialized integers (see Initializing Integers).

Function: void mpz_set (mpz_t rop, const mpz_t op) Set the value of rop from op.

Function: int mpz_set_str (mpz_t rop, const char *str, int base) Set the value of rop from str, a null-terminated C string in base base.

Function: void mpz_swap (mpz_t rop1, mpz_t rop2) Swap the values rop1 and rop2 efficiently.

Here is an example of using one:

```
{
  mpz_t pie;
  mpz_init_set_str (pie, "3141592653589793238462643383279502884", 10);
  ...
  mpz_sub (pie, ...);
  ...
  mpz_clear (pie);
}
```

Code 3.7:

Once the integer has been initialized by any of the mpz_init_set... functions, it can be used as the source or destination operand for the ordinary integer functions.

Function: void mpz_init_set (mpz_t rop, const mpz_t op) Initialize rop with limb space and set the initial numeric value from op.

Function: int mpz_init_set_str (mpz_t rop, const char *str, int base) Initialize rop and set its value like mpz_set_str

Function: unsigned long int mpz_get_ui (const mpz_t op) Return the value of op as an unsigned long.

Function: signed long int mpz_get_si (const mpz_t op) If op fits into a signed long int return the value of op. Otherwise return the least significant part of op, with the same sign as op.

Function: char * mpz_get_str (char *str, int base, const mpz_t op) Convert op to a string of digits in base base.

5.5 Arithmetic Functions

Function: void mpz_add (mpz_t rop, const mpz_t op1, const mpz_t op2) Set rop to op1 + op2.

Function: void mpz_sub (mpz_t rop, const mpz_t op1, const mpz_t op2) Set rop to op1 - op2.

Function: void mpz_mul (mpz_t rop, const mpz_t op1, const mpz_t op2) Set rop to op1 times op2.

Function: void mpz_addmul (mpz_t rop, const mpz_t op1, const mpz_t op2) Set rop to rop + op1 times op2.

Function: void mpz_submul (mpz_t rop, const mpz_t op1, const mpz_t op2) Set rop to rop - op1 times op2.

Function: void mpz_mul_2exp (mpz_t rop, const mpz_t op1, mp_bitcnt_t op2) Set rop to op1 times 2 raised to op2. This operation can also be defined as a left shift by op2 bits.

Function: void mpz_neg (mpz_t rop, const mpz_t op) Set rop to -op.

Function: void mpz_abs (mpz_t rop, const mpz_t op) Set rop to the absolute value of op.

Function: void mpz_mod (mpz_t r, const mpz_t n, const mpz_t d) Set r to n mod d. The sign of the divisor is ignored; the result is always non-negative.

These routines are much faster than the other division functions, and are the best choice when exact division is known to occur, for example reducing a rational to lowest terms.

Function: int mpz_divisible_p (const mpz_t n, const mpz_t d) Return non-zero if n is exactly divisible by d, or in the case of mpz_divisible_2exp_p by $2\hat{b}$.

n is divisible by d if there exists an integer q satisfying n = q*d. Unlike the other division functions, d=0 is accepted and following the rule it can be seen that only 0 is considered divisible by 0.

Function: int mpz_congruent_p (const mpz_t n, const mpz_t c, const mpz_t d) Return non-zero if n is congruent to c modulo d, or in the case of mpz_congruent_2exp_p modulo $2\hat{b}$.

n is congruent to c mod d if there exists an integer q satisfying n = c + q*d. Unlike the other division functions, d=0 is accepted and following the rule it can be seen that n and c are considered congruent mod 0 only when exactly equal.

Function: void mpz_sqrt (mpz_t rop, const mpz_t op) Set rop to the truncated integer part of the square root of op.

Function: int mpz_cmp (const mpz_t op1, const mpz_t op2) Compare op1 and op2. Return a positive value if op1 ¿ op2, zero if op1 = op2, or a negative value if op1 ¡ op2.

Macro: int mpz_sgn (const mpz_t op) Return +1 if op ¿ 0, 0 if op = 0, and -1 if op ¡ 0.

5.12 Input and Output Functions

Function: size_t mpz_out_str (FILE *stream, int base, const mpz_t op) Output op on stdio stream stream, as a string of digits in base base. The base argument may vary from 2 to 62 or from -2 to -36.

Function: size_t mpz_inp_str (mpz_t rop, FILE *stream, int base) Input a possibly white-space preceded string in base base from stdio stream stream, and put the read integer in rop.

Function: size_t mpz_out_raw (FILE *stream, const mpz_t op) Output op on stdio stream stream, in raw binary format. The integer is written in a portable format, with 4 bytes of size information, and that many bytes of limbs. Both the size and the limbs are written in decreasing significance order (i.e., in big-endian).

Function: size_t mpz_inp_raw (mpz_t rop, FILE *stream) Input from stdio stream stream in the format written by mpz_out_raw, and put the result in rop.

5.14 Integer Import and Export

mpz_t variables can be converted to and from arbitrary words of binary data with the following functions.

Function: void mpz_import (mpz_t rop, size_t count, int order, size_t size, int endian, size_t nails, const void *op) Set rop from an array of word data at op.

The parameters specify the format of the data. count many words are read, each size bytes. order can be 1 for most significant word first or -1 for least significant first. Within each word endian can be 1 for most significant byte first, -1 for least significant first, or 0 for the native endianness of the host CPU. The most significant nails bits of each word are skipped, this can be 0 to use the full words.

Here's an example converting an array of unsigned long data, most significant element first, and host byte order within each value.

```
1         unsigned long a[20];
2      /* Initialize z and a */
3      mpz\_import (z, 20, 1, sizeof(a[0]), 0, 0, a);
```

Code 3.8:

Function: void * mpz_export (void *rop, size_t *countp, int order, size_t size, int endian, size_t nails, const mpz_t op) Fill rop with word data from op.

The parameters specify the format of the data produced. Each word will be size bytes and order can be 1 for most significant word first or -1 for least significant first. Within each word endian can be 1 for most significant byte first, -1 for least significant first, or 0 for the native endianness of the host CPU. The most significant nails bits of each word are unused and set to zero, this can be 0 to produce full words.

Chapter 4

Implementation

This is the main chapter of your thesis.

4.1 A section

Text goes here.

4.1.1 A subsection

Text goes here.

4.1.2 Another subsection

Text goes here.

4.2 Another section

Text goes here.

4.2.1 A subsection

Text goes here.

4.2.2 Another subsection

Text goes here.

Chapter 5

Methodology and Results

Methodology and Results go into this chapter.

5.1 A section

Text goes here.

5.1.1 A subsection

Text goes here.

5.1.2 Another subsection

Text goes here.

5.2 Another section

Text goes here.

5.2.1 A subsection

Text goes here.

5.2.2 Another subsection

Text goes here.

Chapter 6

Testing

Testing is explained in this chapter.

6.1 A section

Text goes here.

6.1.1 A subsection

Text goes here.

6.1.2 Another subsection

Text goes here.

6.2 Another section

Text goes here.

6.2.1 A subsection

Text goes here.

6.2.2 Another subsection

Text goes here.

Chapter 7

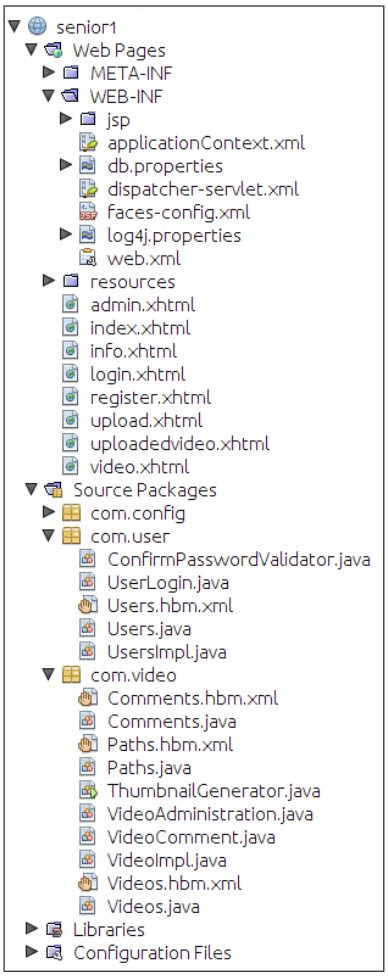
Evaluation, Conclusion and Future work

The final conclusions are derived in this chapter.

Appendix A

The Appendix goes here.

Figure A.1: Project Structure




```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <iostream>
#include <math.h>
#include <stdint.h>
#include <gmp.h>
#include <stdlib.h>
using namespace std;
long Isqrt(long x){
  long  squaredbit, remainder, root;

  if (x < 1) return 0;
  squaredbit = (long)((((unsigned long)~0L) >> 1) &
    ~(((unsigned long)~0L) >> 2));
  remainder = x; root = 0;
  while (squaredbit > 0) {
    if (remainder >= (squaredbit | root)) {
      remainder -= (squaredbit | root);
      root >>= 1; root |= squaredbit;
    }
    else {
      root >>= 1;
    }
    squaredbit >>= 2;
```

```
  }

  return root;
}
bool IsSquare(long n)
{
  if (n < 0)
    return false;

  long tst = (long)(sqrt(n) + 0.5);
  return (tst*tst == n);
}

int gll(mpz_t* P, mpz_t* Q, int a, mpz_t n) {
  if (mpz_cmp_ui(P[2],0) == 0){
    for (int i = 0; i < 3; i++)
      mpz_set(P[i],Q[i]);
    return 1;
  }
  if (mpz_cmp_ui(Q[2],0) == 0){
    return 1;
  }
  if (mpz_cmp(P[0],Q[0])==0){
    if (mpz_cmp(P[1],Q[1])!=0){
      mpz_set_ui(P[1],1);
      return 1;
    }
    if (mpz_cmp_ui(P[0],0)==0){
      mpz_set_ui(P[1],1);
      return 1;
    }
    else{
      mpz_t t0,t9,t10,t11,t1,t2,t3,t4,t5,t6,t7,t8;
      mpz_init_set_ui(t0,2);
      mpz_init_set_ui(t9,3);
      mpz_init_set_ui(t10,a);
      mpz_init(t11);
      mpz_init(t1);
      mpz_init(t2);
      mpz_init(t3);
      mpz_init(t4);
      mpz_init(t5);
      mpz_init(t6);
      mpz_init(t7);
      mpz_init(t8);
      mpz_mul(t1,t0,P[0]);
      mpz_mod(t1,t1,n);
      mpz_mul(t2,P[0],P[0]);
      mpz_mod(t2,t2,n);
      mpz_mul(t3,t9,t2);
      mpz_mod(t3,t3,n);
      mpz_gcd(t11, n, t1);
      if (mpz_cmp_ui(t11,1)!=0){
        mpz_set(P[2],t1);
        return 1;
      }
      mpz_add(t4,t10,t3);
      mpz_invert(t5,t1,n);
      mpz_mul(t6,t4,t5);
      mpz_mod(t6,t6,n);
      mpz_mul(t7,t6,t6);
      mpz_mod(t7,t7,n);
      mpz_t x;
      mpz_init(x);
      mpz_sub(x,t7,t1);
      mpz_sub(t8,P[0],x);
      mpz_mul(t9,t8,t6);
      mpz_mod(t9,t9,n);
      mpz_t y;
      mpz_init(y);
      mpz_sub(y,t9,P[1]);
      mpz_set(P[0],x);
      mpz_set(P[1],y);
      mpz_set_ui(P[2],1);
      mpz_clear(x);
      mpz_clear(y);
      mpz_clear(t0);
      mpz_clear(t9);
      mpz_clear(t10);
      mpz_clear(t11);
      mpz_clear(t1);
      mpz_clear(t2);
      mpz_clear(t3);
```

```
109       mpz_clear(t4);
110       mpz_clear(t5);
111       mpz_clear(t6);
112       mpz_clear(t7);
113       mpz_clear(t8);
114       return 1;
115     }
116   }
117   else{
118     mpz_t t0,t9,t10,t11,t1,t2,t3,t4,t5,t6,t7,t8;
119     mpz_init(t0);
120     mpz_init(t10);
121     mpz_init(t11);
122     mpz_init(t1);
123     mpz_init(t2);
124     mpz_init(t3);
125     mpz_init(t4);
126     mpz_init(t5);
127     mpz_init(t6);
128     mpz_init(t7);
129     mpz_init(t8);
130     mpz_init(t9);
131     mpz_sub(t1,Q[0],P[0]);
132     mpz_gcd(t9, t1, n);
133     if (mpz_cmp_ui(t9,1)!=0){
134       mpz_set(P[2],t1);
135       return 1;
136     }
137     mpz_sub(t2,P[1],Q[1]);
138     mpz_sub(t3,P[0],Q[0]);
139     mpz_invert(t4,t3,n);
140     mpz_mul(t5,t2,t4);
141     mpz_mod(t5,t5,n);
142     mpz_mul(t6,t5,t5);
143     mpz_mod(t6,t6,n);
144     mpz_t x;
145     mpz_init(x);
146     mpz_sub(x, t6, t1);
147     mpz_mod(x,x,n);
148     mpz_sub(t7, P[0], x);
149     mpz_mod(t7,t7,n);
150     mpz_mul(t8,t5,t7);
151     mpz_mod(t8,t3,n);
152     mpz_t y;
153     mpz_init(y);
154     mpz_sub(y,t8,P[1]);
155     mpz_mod(y,y,n);
156     mpz_set(P[0],x);
157     mpz_set(P[1],y);
158     mpz_set_ui(P[2],1);
159     mpz_clear(x);
160     mpz_clear(y);
161     mpz_clear(t0);
162     mpz_clear(t9);
163     mpz_clear(t10);
164     mpz_clear(t11);
165     mpz_clear(t1);
166     mpz_clear(t2);
167     mpz_clear(t3);
168     mpz_clear(t4);
169     mpz_clear(t5);
170     mpz_clear(t6);
171     mpz_clear(t7);
172     mpz_clear(t8);
173     return 1;
174   }
175
176 }
177 void mult2l(int b1, mpz_t* P, int a, mpz_t n) {
178   mpz_t r[3];
179   mpz_init_set_ui(r[0],0);
180   mpz_init_set_ui(r[1],0);
181   mpz_init_set_ui(r[2],1);
182   int b2,b;
183   for (b2 = 2; b2 < b1; b2++){
184     b = b2;
185     while (b != 0){
186       if ((b % 2) == 1){
187         gll(P, r, a, n);
188       }
189       b = b / 2;
190       gll(P, P, a, n);
191       if (mpz_cmp_ui(P[2],1)>0)
```

```
192         return;
193       }
194     }
195     mpz_clear(r[0]);
196     mpz_clear(r[1]);
197     mpz_clear(r[2]);
198   }
199   int ecm(mpz_t z,mpz_t n, int bounda, int boundb) {
200     unsigned long y2;
201     unsigned long Bp = 100;
202     int a,b;
203     unsigned long x;
204     mpz_t fpoint[3];
205     for (a = 1; a <= bounda; a++){
206       if ((4*a*a*a + 27) == 0)
207         continue;
208       for (b = 2; b <= boundb; b++){
209         x = 0;
210         do{
211           x = x + 1;
212           y2 = x*x*x + (a*x) + b;
213         } while (!IsSquare(y2)&& x<100);
214         if (x>=100){ continue; }
215         mpz_init_set_ui(fpoint[0],x);
216         mpz_init_set_ui(fpoint[1],Isqrt(y2));
217         mpz_init_set_ui(fpoint[2],1);
218         mult2l(Bp, fpoint, a, n);
219         if (mpz_cmp_ui(fpoint[2],0)==0)
220           break;
221         mpz_init(z);
222         mpz_gcd(z, fpoint[2], n);
223         if (mpz_cmp_ui(z,1)>0){
224           printf("a:%d b:%d x:%d\n",a,b,x);
225           return 1;
226         }
227       }
228     }
229     return 0;
230   }
231   int main() {
232     srand((unsigned)time(NULL));
233     unsigned long n[2];
234     for (int i = 0; i < 10; i++){
235       n[0] = rand()/10000;
236       n[1] = rand()*rand();
237       mpz_t x;
238       mpz_t z1,z2,z;
239       mpz_init(x);
240       mpz_init(z);
241       mpz_init(z1);
242       mpz_init(z2);
243       mpz_import(x, 1, -1, sizeof(n[0]), 0, 0, n);
244       mpz_nextprime(z1, x);
245       mpz_import(x, 1, -1, sizeof(n[1]), 0, 0, n+1);
246       mpz_nextprime(z2, x);
247       mpz_mul(z,z1,z2);
248       mpz_t factor;
249       mpz_init(factor);
250   //    mpz_init_set_ui(z,8774208478331381);
251       ecm(factor,z, 100, 100);
252       mpz_out_str(NULL,10,z);
253       printf(" mod ");
254       mpz_out_str(NULL,10,factor);
255       printf(";\n");
256
257     }
258   }
```

Code A.1: project/GMPECM.cpp

Bibliography

[1] *LaTex*. WikiBooks, 2011.

[2] Latex on wikibooks @https://en.wikibooks.org/wiki/LaTeX, Dec. 2011.

[3] Youtube in wikipedia @http://en.wikipedia.org/wiki/YouTube, Dec. 2011.

[4] C. Adamson and J. Marinacci. *Swing Hacks: Tips and Tools for Killer GUIs*. O'Reilly Media, 1 edition, 2005.

[5] Apache. Manager app how-to. `https://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html`.

[6] D. Brown, C. M. Davis, and S. Stanlick. *Struts 2 in Action*. Manning Publications, 2008.

[7] CoreServlets. Hibernate and jpa: An introduction and tutorial. `http://courses.coreservlets.com/Course-Materials/hibernate.html`.

[8] CoreServlets. Jsf 2.0 tutorials. `http://www.coreservlets.com/JSF-Tutorial/jsf2/`.

[9] CoreServlets. Spring tutorial: An introduction and tutorial for the spring framework. `http://courses.coreservlets.com/Course-Materials/spring.html`.

[10] P. Deitel. *Internet & World Wide Web: How to Program*. Prentice Hall, 4 edition, 2007.

[11] P. Deitel and H. Deitel. *Java How to Program*. Prentice Hall, 9 edition, 2012.

[12] T. Downey. *Web Development with Java: Using Hibernate, JSPs and Servlets*. Springer, 1 edition, 2007.

[13] D. Geary and C. Horstmann. *Core JavaServer Faces*. Prentice Hall, 3 edition, 2010.

[14] IdleWorx. Setting up log4j for simple java web applications + tomcat 5/6. http://blog.idleworx.com/2010/01/setting-up-log4j-for-simple-java-web.html.

[15] Mkyong. Hibernate tutorial. http://www.mkyong.com/tutorials/hibernate-tutorials/.

[16] Mkyong. How to resize an image in java ? http://www.mkyong.com/java/how-to-resize-an-image-in-java/.

[17] Mkyong. Jsf 2.0 + spring + hibernate integration example. http://www.mkyong.com/jsf2/jsf-2-0-spring-hibernate-integration-example/.

[18] Mkyong. Jsf 2.0 tutorial. http://www.mkyong.com/tutorials/jsf-2-0-tutorials/.

[19] Mkyong. Spring tutorial. http://www.mkyong.com/tutorials/spring-tutorials/.

[20] S. Overflow. Datatable - incell editing how to get the value of the object edited in the managed bean. http://stackoverflow.com/questions/10398960/datatable-incell-editing-how-to-get-the-value-of-the-object-edited-in-the-mana.

[21] S. Overflow. Jsf 2.0: Validate equality of 2 inputsecret fields (confirm password) without writing code? http://stackoverflow.com/questions/2909021/jsf-2-0-validate-equality-of-2-inputsecret-fields-confirm-password-without-wr.

[22] V. Patel. Javaserver faces jsf validation tutorial: Error handling in jsf. http://viralpatel.net/blogs/2009/02/javaserver-faces-jsf-validation-tutorial-error-handling-jsf-validator.html.

[23] PrimeFaces. Primefaces - datatable - incell editing. http://www.primefaces.org/showcase/ui/datatableEditing.jsf.

[24] ShinePHP. Installing jdk 7 on ubuntu. http://www.shinephp.com/install-jdk-7-on-ubuntu/.

[25] J. Shirazi. *Java Performance Tuning*. O'Reilly Media, 2 edition, 2003.

[26] P. Teknoloji. *PrimeFaces's User's Guide*. Prime Teknoloji, 2011.

[27] Video.js. Video.js. http://videojs.com/.

[28] Xuggle. Decodeandcaptureframes. https://github.com/xuggle/xuggle-xuggler/blob/master/src/com/xuggle/mediatool/demos/DecodeAndCaptureFrames.java.

[29] Xuggle. Xuggler download page. http://xuggle.com/downloads.

[30] Xuggle. Xuggler main page. http://xuggle.com/.