The Genus 2 Crypto C++ Library v1.0.0

Ning Shang

Table of Contents

1 Overview

1.1 What is G2HEC?

The G2HEC (Genus 2 HyperElliptic Curve) library is a free portable C++ library providing divisor group operations in the Jacobian of genus 2 (imaginary) hyperelliptic curve. Such curves can be used for discrete-logarithm-based cryptosystems with advantages over elliptic curves (genus 1 curves). Divisor group operations are essential to using genus 2 curves for cryptography.

It is built on top of V. Shoup's NTL library, "a high performance, portable C++ library providing data structures and algorithms for arbitrary length integers; for vectors, matrices, and polynomials over the integers and over finite fields; and for arbitrary precision floating point arithmetic."

It is recommended to build NTL using GMP (the GNU Multi-Precision package) for best performance. However, the G2HEC library can be built and used with and without the existence of GMP.

This library does not assume users' familiarity with any non-trivial math background except basic concepts of finite fields and polynomials. No prior knowledge of genus 2 curve is needed to used the library.

The G2HEC library is released under the GNU General Public License. See the file COPYING for details.

The G2HEC library homepage: `https://github.com/syncom/libg2hec/`.

The NTL library homepage: `http://www.shoup.net/ntl/`.

The GMP library homepage: `http://www.swox.com/gmp/`.

1.2 Genus 2 curve basics

Our object of interest, a genus 2 curve, is a nonsingular algebraic curve

$$C : y^2 + h(x)y = f(x)$$

over a finite field $GF(q)$, where $f(x)$ is monic (i.e., has leading coefficient 1) of degree 5, $h(x)$ has degree not greater than 2, q is a prime or a power of a prime.

Currently (version 1.0.0) G2HEC only supports odd prime q by default. You can "hack" the header file in `include/g2hec_nsfieldtype.h` to switch its support to q as a power of an odd prime. However, we do not recommend to do so and it should be avoided except in extreme cases. So far the library is tested only for odd prime numbers q.

There is currently no "hack" to have field types `GF2` and `GF2E` supported (curve singularity test does not work for characteristic 2 yet). Support for characteristic 2 maybe added in the future.

A pair (x_0, y_0) with $x_0, y_0 \in GF(q)$ such that $x = x_0$ and $y = y_0$ give a solution to the curve equation of C is called a $GF(q)$**-rational point of the genus 2 curve** C. Obviously, C has only finitely many $GF(q)$-rational points. Unlike the case of elliptic curves, the set of $GF(q)$-rational points of the curve C does not naturally hold a group structure, therefore they cannot be used directly to do discrete-logarithm-based cryptography. Instead, we choose to use the $GF(q)$**-rational points of the Jacobian** of the curve C, which is a finite group, to achieve our (cryptographic) goal. It is not necessary for the users of the G2HEC library know how the Jacobian of a genus 2 curve is defined, given the following two facts:

1. The $GF(q)$-rational points of the Jacobian of a genus 2 curve C, $Jac(C, GF(q))$, is a finite group of size approximately q^2.
2. An element of $Jac(C, GF(q))$ is called a **divisor class** (**divisor** in short) over $GF(q)$. A divisor (class) D is represented by a pair of polynomials $[u(x), v(x)]$ over $GF(q)$. Several conditions need to be satisfied for a pair $[u(x), v(x)]$ to be a divisor of C.

The unit element of $Jac(C, GF(q))$ is $[1, 0]$.

By convention, the group operation in the Jacobian of the curve C is written additively. We shall write

$$D = D_1 + D_2, D = D_1 - D_2, D = [n]D_1$$

for **divisor addition, divisor subtraction** and **divisor scalar multiplication**, respectively, where $D = [n]D_1$ means $D = D_1 + D_1 + ... + D_1$ for n times.

More information about the math background of the Jacobian of genus 2 curves can be found in the book "Handbook of Elliptic And Hyperelliptic Curve Cryptography" by Cohen et al.

In addition to the high-level functions that operates on curves and divisors, the G2HEC library also exports low-level functions that allow knowledgeable users to take control of mathematical factors of curves and divisors. Our intent is to make the library useful for both mathematicians and cryptographers (or the ones who play both roles).

2 Installation

The G2HEC library is dependent of V. Shoup's NTL library, which can be found at NTL: A Library for doing Number Theory (`http://www.shoup.net/ntl/`).

The following procedure should work on Unix and Unix-like platforms. The installation has been tested on Linux and Solaris, and in Cygwin on Windows.

To obtain the source code and documentation for G2HEC, download `https://github.com/syncom/libg2hec/tarball/vX.Y.Z`. Here "X.Y.Z" denotes the current version number, i.e., 1.0.0.

2.1 General installation instructions

If the NTL library is installed in a standard system directory, then do the following:

```
% git clone https://github.com/syncom/libg2hec.git
% cd ligb2hec
% autoreconf -vi
% ./configure --prefix=$HOME/nssw
% make
% make check
% make install
```

This will build, test and install G2HEC in `$HOME/nssw`. You can change to a different location by specifying it with the `--prefix=` option (the default is `/usr/local`). After installation, you will find the G2HEC header files in `$HOME/nssw/include` and the compiled static library `libg2hec.a` in `$HOME/nssw/lib/`.

2.2 NTL not in default location

If you have installed the NTL/GMP libraries into locations which require special include or library paths, you can pass them to `LDFLAGS` and `CXXFLAGS` variables for the configure script. For example (for bash)

```
% env LDFLAGS=-L$HOME/nssw/lib CXXFLAGS=-I$HOME/nssw/include ./configure
--prefix=$HOME/nssw
```

Depending on whether GMP is found in the search path or not, the G2HEC library is configured to build with or without GMP.

2.3 After G2HEC is built

After the library is built, executing `make check` runs some test programs.

It is highly recommended to do this to make sure things went well.

Executing `make install` copies a number of files to a directory `<prefix>` that you specify by passing `--prefix=<prefix>` as an argument to `configure` at configuration time, or as an argument to `make install` at installation time. Recall that the default is `/usr/local`.

To uninstall the library, execute `make uninstall`.

To remove object files, execute `make clean`.

Note that this installation process is almost standard.

Assuming you have installed G2HEC as above, to compile program `myprog.C` that uses G2HEc, do

```
% g++ -I<prefix>/include -I<ntl_prefix>/include -L<prefix>/lib \
-L<ntl_prefix>/lib myprog.c -o myprog -lg2hec -lntl -lm
```

The binary file `myprog` is created.

If you built NTL using GMP, do

```
% g++ -I<prefix>/include -I<ntl_prefix>/include -L<prefix>/lib \
-L<ntl_prefix>/lib -L<gmp_prefix>/lib myprog.c -o myprog -lg2hec \
-lntl -lgmp -lm
```

Of course, there is no need to duplicate the flags if some of the locations are the same, and you may leave out the corresponding flags if the locations are standard directories.

3 Quick start

3.1 A web demo of divisor scalar multiplication

You can play around with divisor scalar multiplication on the following demo page:
`http://www.math.purdue.edu/~nshang/g2hec/demo.html`.

This demo page shows an application built on top of the G2HEC library. Although you may smell a bit of mathematics from this demo, in general programmers can build cryptographic algorithms/protocols without knowing any college-level math.

4 Tutorial

We shall walk you through several cryptographic applications of G2HEC.

4.1 HEC-Diffie-Hellman key exchange

The basic Diffie-Hellman key exchange protocol works as follows:

Alice and Bob wish to agree on a secret random element over an insecure channel without having to exchange any information previously.

They agree on an element P in the Jacobian of the genus 2 curve C with a large (known or unknown) order G.

1. Alice generates a random integer $a \in \{1, \ldots, \#G - 1\}$, then sends to Bob the element (divisor)
$$Q_1 = [a]P.$$

2. Bob generates a random integer $b \in \{1, \ldots, \#G - 1\}$, then sends to Alice the element (divisor)
$$Q_2 = [b]P.$$

3. Alice then computes
$$[ab]P = [a]Q_2.$$

4. Similarly, Bob computes
$$[ab]P = [b]Q_1.$$

This element $[ab]P$ now serves as the secret that only Alice and Bob know.

We illustrate a local version of this protocol using a genus 2 curve over a prime field $GF(p)$, for an educational purpose.

First we include the header file g2hec_Genus2_ops.h:

```
#include <g2hec_Genus2_ops.h>
```

Then we define the maximum length (in decimal digits) of the prime number p to be 300:

```
#undef MAX_STRING_LEN
#define MAX_STRING_LEN 300
```

The following statement invokes the namespace used by the G2HEC library, and must be present for every client program.

```
NS_G2_CLIENT
```

In the main function, we first set the pseudo-random number generator seed, and allocate a string buffer for receiving a prime number input by the user:

```
SetSeed(to_ZZ(1234567890));
char p[MAX_STRING_LEN];

cin.getline(p, MAX_STRING_LEN);
```

We declare and initialize an NTL big integer object `pZZ` to store this prime number:

```
ZZ pZZ = to_ZZ(p);
```

Now we initialize the underline prime field $GF(p)$:

```
field_t::init(pZZ);
```

We shall declare and initialize several elements to hold the genus 2 curve, system parameters, keys and so on:

```
ZZ a, b;
g2hcurve curve;
divisor P, Q1, Q2;
```

Generate a random valid genus 2 curve:

```
curve.random();
```

Set curve for the divisors. A curve needs only to be set once in a program to work for all divisors.

```
P.set_curve(curve);
```

To perform Diffie-Hellman key exchange, it is not necessary to know the order of P. A fact is that this order is close to p^2. So we choose a and b to be two random number bounded by p^2.

```
RandomBnd(a, pZZ*pZZ);
RandomBnd(b, pZZ*pZZ);
```

We generate a random base point P:

```
P.random();
```

Alice computes

```
Q1 = a * P;
```

Bob computes

```
Q2 = b * P;
```

A successful key exchange should yield the shared secret

$$[ab]P = [a]Q_2 = [b]Q_1:$$

```
if ( b * Q1 == a * Q2)
  cout << "DH key exchange succeeded!" << endl;
else
  cout << "DH key exchange failed!" << endl;
```

A complete source file for this example can be found in `example/dh.C`.

4.2 HEC-ElGamal encryption

This example is similar to the Diffie-Hellman key exchange example. Please refer to the source file in `examples/elgamal_enc.C` for details.

4.3 HEC-ElGamal digital signature

We shall use G2HEC to build a digital signature scheme: the ElGamal digital signature.

Recall how this signature scheme works:

Bob chooses a genus 2 curve C and a prime number p so that the number of $GF(p)$-rational points of the Jacobian of C has a large prime factor N – preferably the number itself is prime. He then chooses a divisor g of order N, a secret private key $x \in \{1, \ldots, N-1\}$, then computes a divisor $h = [x]g$. Bob publishes g and h as his public key.

To sign a message $m \in Z/(N)$, Bob first generates a random integer $i \in \{1, \ldots, N-1\}$, and computes

$$a = [k]g.$$

Bob then computes a solution, $b \in Z/(N)$ to the congruence

$$m \equiv xf(a) + bk(modN).$$

Here f is a map from the Jacobian of C to $Z/(N)$ which is almost injective.

Bob sends the signature (a, b) together with the message m to Alice.

Upon receiving the message and signature from Bob, Alice verifies the signature by checking that

$$[f(a)]h + [b]a = [m]g$$

holds.

It needs to be point out that a technical aspect of this algorithm involves the choice of the curve C and the prime number p so that its Jacobian has an order divisible by a large prime number. This is not a trivial task – it involves quite amount of mathematics. Fortunately, there are published data that users can use directly to avoid this difficulty.

According to the paper "Construction of Secure Random Curves of Genus 2 over Prime Fields" by Gaudry and Schost, we choose to use in our example the curve

$$C : y^2 = x^5 + 268281082283935564490073 6x^3 + 226591355295993102902116x^2$$

$$+254767471595292971789991 8x + 479730995970848967305935 0$$

with $p = 5 \cdot 10^{24} + 850349$. The order of the Jacobian of the curve is a prime number

$$N = 24999999999994130438600999402209463966197516075 69,$$

which is to our best interest. In this case, we can pick any random non-unit divisor g as our base point.

At this point, we are able present a local version of this signature scheme.

As usual we include the header file `g2hec_Genus2_ops.h`:

```
#include <g2hec_Genus2_ops.h>
```

Then we define some macros that we are going to use, including the values of coefficients of the polynomial f, the prime number p, the group order of the Jacobian, and a function that maps a suitable value to an NTL `ZZ_p` object.

```
#define f3 "26828108228393556449007 36"
#define f2 "22659135529599310290211 6"
#define f1 "25476747159529297178999 18"
#define f0 "47973099597084896730593 50"
#define ps "50000000000000000008503491"
#define N "24999999999994130438600999402209463966197516075699"

#define str_to_ZZ_p(x) to_ZZ_p(to_ZZ(x))
```

Also we issue the statement

```
NS_G2_CLIENT
```

Recall that we need an almost bijection that maps elements in the Jacobian to an element in $\{1, \ldots, N-1\}$. This map can be chosen as follows:

```
static ZZ from_divisor_to_ZZ(const divisor& div, const ZZ& n)
{
  poly_t u = div.get_upoly();
  ZZ temp = AddMod(sqr(rep(u.rep[0])), sqr(rep(u.rep[1])), n);
  return ( IsZero(temp) ? to_ZZ(1) : temp );
}
```

We just mention that we choose this function to take a divisor

$$[u(x), v(x)]$$

to the value defined by the sum of the squares of the degree 0 and degree 1 coefficients modulo the group order N of the Jacobian. Users can define their own such function to use. Security might be affected by bad choice of this almost injective function.

We start working on the `main` function by initializing the PRNG seed and declaring and define several values:

```
SetSeed(to_ZZ(1234567890));

ZZ p = to_ZZ(ps);

field_t::init(p);

ZZ order = to_ZZ(N);

ZZ x, k, b, m;
// Private key x, random number k, parameter b, message m

ZZ f_a;

g2hcurve curve;

divisor g, h, a;

poly_t f;
```

Then we manually assign values of polynomial f and define the corresponding genus 2 curve:

```
SetCoeff(f, 5, 1);
SetCoeff(f, 4, 0);
SetCoeff(f, 3, str_to_ZZ_p(f3));
SetCoeff(f, 2, str_to_ZZ_p(f2));
SetCoeff(f, 1, str_to_ZZ_p(f1));
SetCoeff(f, 0, str_to_ZZ_p(f0));
curve.set_f(f);
```

Then you update the curve information:

```
curve.update();
```

This will update some flags related to properties of the curve, and needs to be done immediately after manual assignments to the curve's defining elements (e.g., the polynomial f).

Now we are ready to set the curve as the underlying curve of the divisors.

```
g.set_curve(curve);
```

We randomly choose the base point g, message m to be signed, the private key x, and the public key h:

```
do {
  g.random();
} while (g.is_unit());
```

```
RandomBnd(m, order);

do {
  RandomBnd(x, order);
} while (IsZero(x));

h = x * g;
```

Note that we want the base point g to be any divisor except the unit divisor, and we do not allow the private key x to be 0. This is what the *do...while* statements do.

Now we shall generate the ElGamal signature (a, b):

```
do {
  RandomBnd(k, order);
} while (IsZero(k));

a = k * g;

f_a = from_divisor_to_ZZ(a, order);

/* b = (m - x*f(a))/k mod N */
b = MulMod(m - x * f_a, InvMod(k, order), order);
```

The element `f_a` holds the value $f(a)$ given by the almost injection applied to the divisor a.

Alright! Now the most exciting moment has arrived – signature verification:

```
if ( f_a * h + b * a == m * g )
  cout << "ElGamal signature verification succeeded!" << endl;
else
  cout << "ElGamal signature verification failed!" << endl;
```

The complete source file can be found in `examples/elgamal_sig.C`.

****

In the following chapters, we will give a general overview of the G2HEC's programming interface. This includes:

- Functions related to genus 2 curve generation and manipulation
- Functions related to divisor generation and manipulation
- Divisor arithmetic functions
- Input and output functions
- Randomness functions

All these interfaces are exported by `include/g2hec_Genus2_ops.h` unless otherwise specified. **********

5 Genus 2 curve functions

We only consider the so-called "imaginary" genus 2 curves, i.e., curves given by the equation

$$C : y^2 + h(x)y = f(x),$$

with $\deg h \leq 2$ and $\deg f = 5$. We also require that C be nonsingular and the leading coefficient of $f(x)$ be 1.

Types `field_t` and `poly_t` are for the field element type and the corresponding polynomial type, respectively. They are defined in the header file `include/g2hec_nsfieldtype.h`. Type `poly_tc` is the type `const poly_t`.

A genus 2 curve is stored in the `g2hcurve` class.

Below are some of the class's public member functions. They are exported in the file `include/g2hec_Genus2_ops.h`.

`g2hcurve()`:

default constructor; set curve to $y^2 = 0$.

`g2hcurve(const poly_tc& poly1, const poly_tc& poly2)`:

constructor; sets curve to $y^2 + \texttt{poly2} * y = \texttt{poly1}$.

`g2hcurve(const g2hcurve& hcurve)`:

copy constructor.

`void set_f(const poly_tc& poly)`:

sets $f(x)$ to `poly`.

`void set_h(const poly_tc& poly)`:

sets $h(x)$ to `poly`.

`void update()`:

checks and set internal flags to determine whether the curve is nonsingular and of genus 2. This function MUST be called immediately after the curve has been changed by `set_f()`, `set_h()`, or/and assignment, and before any further operations related to the curve. This is very important to remember, otherwise error may occur.

`const poly_tc& get_f()`:

returns reference to polynomial $f(x)$.

`const poly_tc& get_h()`:

returns reference to polynomial $h(x)$.

`bool_t is_valid_curve()`:

returns TRUE is curve is nonsingular, genus 2, and $f(x)$ is monic.

You can also write `curve1 = curve2` naturally to set the value of `curve1` to be the same as `curve2`. It is not needed that `curve2` be valid for this assignment.

You can do comparisons like `curve1 == curve2` or `curve1 != curve2` to test if two curves are the same. Curves do not need to be valid for these comparisons.

6 Divisor functions

A divisor (class) is a pair of polynomials $[u(x), v(x)]$ over $GF(q)$ with $\deg v < \deg u \leq 2$ so that they satisfy certain conditions related to the curve C. Therefore, every divisor (class) has a curve to associate with. G2HEC use a C++ class `divisor` to hold a divisor.

For cryptographic purposes, G2HEC does not support the existence of divisors associated with different curves in the same running process. It implements the associated genus 2 curve as a static data member of the `divisor` class. The client program only needs to set the curve once for a divisor, and it works for all divisors. A curve change for one divisor in a program will cause all divisors in the same program to switch to the new curve. This is another important convention of G2HEC.

We list some of the `divisor` class's public member functions as follows.

`divisor()`:

default constructor; sets $u(x) = 1, v(x) = 0$, and associated curve as $y^2 = 0$. A divisor is not valid by default.

`divisor(const poly_tc& polyu, const poly_tc& polyv, const g2hcurve& curve)`:

constructor; set $u(x)$ =`polyu`, $v(x)$ =`polyv`, and associated curve to `curve`.

`divisor(const divisor& div)`:

copy constructor.

`void set_upoly(poly_tc& poly)`:

sets $u(x)$ to `poly` for the divisor.

`void set_vpoly(poly_tc& poly)`:

sets $v(x)$ to `poly` for the divisor.

`void set_curve(const g2hcurve& curve)`:

sets the associated curve to `curve` for divisor. The curve does not need to be a valid genus 2 curve.

`void update()`:

This function checks and updates information related to the divisor. It must be called by a client program after any `set_` routine is called. Otherwise error make occur when operations on divisor are performed.

`const poly_tc& get_upoly()`:

returns $u(x)$ of the divisor.

`const poly_tc& get_vpoly()`:

returns $v(x)$ of the divisor.

`const g2hcurve& get_curve()`:

returns the associated curve of the divisor.

`bool_t is_valid_divisor()`:

returns `TRUE` is the representation of the divisor is valid, `FALSE` otherwise.

`bool_t is_unit()`:

tests if the divisor is the unit divisor $[1, 0]$.

`set_unit()`:

sets the divisor to the unit divisor $[1, 0]$.

You can write `divisor1 = divisor2` to assign the value of `divisor2` to `divisor1`. `divisor2` can be either valid or invalid.

You can do comparisons like `divisor1 == divisor2` or `divisor1 != divisor2` to test if two divisors are the same. Divisors do not need to be valid for such comparisons.

There is also a non-member function

```
bool_t on_same_curve(const divisor& a, const divisor& b)
```

that tests if two divisors `a` and `b` have the same associated curve. According to the way G2HEC implements the `divisor` class, this function always returns `TRUE`. Therefore this test is redundant.

7 Divisor arithmetic

The divisor operations are **addition, negation, subtraction** and **scalar multiplication**.

The divisor arithmetic is implemented using the **explicit formulas**, which is supposed to be faster than the generic one via **Cantor's algorithm**, for the most common cases of the divisor operations. The Cantor's algorithm is also implemented in G2HEC and used to handle special cases that rarely happen in cryptographic applications.

The interfaces for divisor arithmetic are strait-forward. We list them in below in both their procedure and operator version (if both version exist).

`bool_t add_cantor(divisor& x, const divisor& a, const divisor& b)`:

This is the Cantor's algorithm which computes `x = a + b`.

`bool_t add((divisor& x, const divisor& a, const divisor& b)`,

`divisor operator+(const divisor& a, const divisor& b)`:

compute `x = a + b` via explicit formulas; special cases handled by calling `add_cantor()`.

`bool_t sub((divisor& x, const divisor& a, const divisor& b)`,

`divisor operator-(const divisor& a, const divisor& b)`:

compute `x = a - b`.

`bool_t dnegate(divisor& x, const divisor& a)`,

`divisor operator-(const divisor& a)`:

compute `x = -a`.

`scalar_mul(divisor& x, const divisor& a, const ZZ& n, bool_t (*method)(divisor&, const divisor&, const ZZ&))`,

`scalar_mul(divisor& x, const divisor& a, long n, bool_t (*method)(divisor&, const divisor&, const ZZ&))`:

compute `x = [n]a` using a method specified by `method`. Currently, three methods are provided by G2HEC:

1. `SAM`: sum-and-square method

2. `NAF`: method using non-adjacent forms
3. `ML`: method of the **Montgomery's ladder**; used to resist side-channel attacks.

Setting `method` to `NULL` performs scalar multiplication via `SAM`.

`divisor operator*(const ZZ& n, const divisor& a)`,

`divisor operator*(const divisor& a, const ZZ& n)`,

`divisor operator*(long n, const divisor& a)`,

`divisor operator*(const divisor& a, long n)`:

return `[n]a`. `SAM` is used for scalar multiplication.

8 I/O

G2HEC exports several functions to output of polynomials, curves and divisors. They might be more useful for mathematics research on genus 2 curves over finite fields than for implementation of cryptographic protocols. For sake of completeness, we briefly introduce them here.

`void print_poly(poly_t& poly, std::ostream *s)`:

directs a natural representation of polynomial `poly` to `s`.

For example, calling `print_poly(f, cout)` will display in the standard output

```
x^5 + 3*x^2 + x
```

if `f` is a `poly_t` object representing $f(x) = x^5 + 3x^2 + x$.

Calling `cout << curve;` will display in the standard output something like

```
Curve: y^2 + h(x)*y = f(x)
       h(x) = 0
       f(x) = x^5 + 1
       Genus 2 curve is nonsingular
```

if `curve` is a valid genus 2 curve, or

```
Curve: y^2 + h(x)*y = f(x)
       h(x) = 0
       f(x) = x^2
       Curve is singular, or not genus 2, or f(x) is not monic
```

if `curve` is not valid.

Calling `cout << divisor;` will display in the standard output something like

```
###
Divisor [u(x), v(x)] for Jacobian group of curve y^2 + h(x)*y = f(x).
Curve: y^2 + h(x)*y = f(x)
       h(x) = 0
       f(x) = x^5 + 1
       Genus 2 curve is nonsingular
[u(x), v(x)]:
       u(x) = 1
       v(x) = 0
       Divisor is valid
###
```

if `divisor` is a valid divisor, or

```
###
```

```
Divisor [u(x), v(x)] for Jacobian group of curve y^2 + h(x)*y = f(x).
Curve: y^2 + h(x)*y = f(x)
       h(x) = 0
       f(x) = x^5 + 1
       Genus 2 curve is nonsingular
[u(x), v(x)]:
       u(x) = 1
       v(x) = x
       Divisor is invalid
###
```

if `divisor` is not valid.

9 Randomness functions

The G2HEC library uses NTL's pseudo-random number generator for generating functions that output random values. It is recommended to override the default seed by calling

`NTL::SetSeed( seed )`

to set the PRNG seed using a `ZZ` object `seed` if you are to do serious cryptographic applications which use the G2HEC's randomness functions.

The G2HEC library exports in `include/g2hec_rand.h` a function

`ZZ g2hec_rand()`,

which obtains a random `ZZ` object of 128-bit integer by trying to use the file /dev/urandom as a source of random bits; if it fails, then 0 is returned.

User can set its own random seed, or call

`NTL::SetSeed(g2hec_rand())`

to generate one.

The class `g2hcurve` has a member function to generate a random valid genus 2 curve.

`g2hcurve& random()`:

generates a random valid genus 2 curve.

The class `divisor` has two member functions to generate random valid divisors, if the associated curve is valid.

`divisor& random()`:

If the associated curve is valid, sets the divisor to a random valid divisor of degree 2, i.e., $\deg u = 2$.

`divisor& random(divdeg_t dgr)`:

If the associated curve is valid, sets divisor to a random valid divisor of degree `dgr`, where `dgr` takes values 1 (DEGREE_1) or 2 (DEGREE_2).

Note that `divisor.random()` never returns a unit divisor $[1, 0]$.

10 Some issues

Most functions in the G2HEC are reentrant, with exception of functions that set the associate curve for a divisor, such as `divisor::set_curve` and a constructor of the `divisor` class. It is considered an unchecked programming error to change the associated curve when performing divisor operations.