PBC Library Manual 0.5.11

Ben Lynn

PBC Library Manual 0.5.11

by Ben Lynn

Revision History

2006 Revised by: BL

Table of Contents

Preface

The PBC library is a free portable C library allowing the rapid prototyping of pairing-based cryptosystems. It provides an abstract interface to a cyclic group with a bilinear pairing, insulating the programmer from mathematical details. Knowledge of elliptic curves is optional.

The PBC library is built on top of the GMP library, and the PBC API is strongly influenced by the GMP API. Accordingly, this manual tries to imitate the look and feel of the GMP manual.

The PBC library homepage: http://crypto.stanford.edu/pbc/

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

Chapter 1. Installing PBC

The PBC library needs the GMP library (http://www.swox.com/gmp/). Multiple ways to install PBC are provided.

1.1. GNU Build System (autotools)

This build system has been tested and works on Linux and Mac OS X with a fink installation.

```
$ ./configure
$ make
$ make install
```

On Windows, the configure command requires a couple of options:

```
$ ./configure -disable-static -enable-shared
```

By default the library is installed in `/usr/local/lib`. On some systems, this may not be in the library path. One way to fix this is to edit `/etc/ld.so.conf` and run `ldconfig`.

1.2. Simple Makefile

For speed and simplicity, I use `simple.make` during development. Naturally it is less portable.

```
$ make -f simple.make
```

PBC uses some GNU C extensions such as nested functions.

1.3. Quick start

We shall use the following notation. For our purposes, the pairing is a bilinear map from two cyclic groups, G1 and G2 to a third group GT, where each group has prime order r.

Run `pbc/pbc` and type:

```
g := rnd(G1);
g;
```

The first line generates a random element g of the group G1, while the second prints out the value of g. (The syntax was influenced by `bc`, an arbitrary precision calculator.) Next, enter:

```
h := rnd(G2);
h;
```

This assigns h to a random element of the group G2. Actually, the default pairing `pbc` uses is symmetric so G1 and G2 are in fact the same group, but in general they are distinct. To compute the pairing applied to g and h, type:

```
pairing(g,h);
```

The order of both g and h is r. Let's generate two random numbers between 1 and r:

```
a := rnd(Zr);
b := rnd(Zr);
```

By bilinearity, the resulting output of both of these lines should be identical:

```
pairing(g^a,h^b);
pairing(g,h)^(a*b);
```

This program has other features but the commands shown here should be enough to quickly and interactively experiment with many pairing-based cryptosystems using real numbers.

1.4. Basics

Programs using the PBC library should include the file `pbc.h`:

```
#include <pbc.h>
```

and linked against the PBC library and the GMP library, e.g.

```
$ gcc program.c -L. -lpbc -lgmp
```

The file `pbc.h` already includes `gmp.h`.

PBC follows GMP in several respects:

- Output arguments generally precede input arguments.
- The same variable can be used as input and output in one call.
- Before a variable may be used it must be initialized exactly once. When no longer needed it must be cleared. For efficiency, unnecessary initializating and clearing should be avoided.

- PBC variables ending with `_t` behave the same as GMP variables in function calls: effectively as call-by references. In other words, as in GMP, if a function that modifies an input variable, that variable remains modified when control return is returned to the caller.
- Like GMP, variables automatically allocate memory when needed. By default, `malloc()` and friends are called but this can be changed.
- PBC functions are mostly reentrant.

Since the PBC library is built on top of GMP, the GMP types are available. PBC types are similar to GMP types. The following example is paraphrased from an example in the GMP manual, and shows how to declare the PBC data type `element_t`.

```
element_t sum;
struct foo { element_t x, y; };
element_t vec[20];
```

GMP has the `mpz_t` type for integers, `mpq_t` for rationals and so on. In contrast, PBC uses the `element_t` data type for elements of different algebraic structures, such as elliptic curve groups, polynomial rings and finite fields. Functions assume their inputs come from appropriate algebraic structures.

PBC data types and functions can be categorized as follows. The first two alone suffice for a range of applications.

- `element_t`: elements of an algebraic structure.
- `pairing_t`: pairings where elements belong; can initialize from sample pairing parameters bundled with PBC in the `param` subdirectory.
- `pbc_param_t`: used to generate pairing parameters.
- `pbc_cm_t`: parameters for constructing curves via the CM method; sometimes required by `pbc_param_t`.
- `field_t`: algebraic structures: groups, rings and fields; used internally by `pairing_t`.
- a few miscellaneous functions, such as ones controlling how random bits are generated.

Functions operating on a given data type usually have the same prefix, e.g. those involving `element_t` objects begin with `element_`.

Chapter 2. Tutorial

This chapter walks through how one might implement the Boneh-Lynn-Shacham (BLS) signature scheme using the PBC library. It is based on the file `example/bls.c`.

We have three groups *G1*, *G2*, *GT* of prime order *r*, and a bilinear map *e* that takes an element from *G1* and an element from *G2*, and outputs an element of *GT*. We publish these along with the system parameter *g*, which is a randomly chosen element of *G2*.

Alice wishes to sign a message. She generates her public and private keys as follows. Her private key is a random element *x* of *Zr*, and her corresponding public key is g^x.

To sign a message, Alice hashes the message to some element *h* of *G1*, and then outputs the signature h^x.

To verify a signature σ, Bob checks that $e(h,g^x) = e(\sigma, g)$.

We now translate the above to C code using the PBC library.

2.1. BLS signatures

First we include `pbc/pbc.h`:

```
#include <pbc.h>
```

Next we initialize a pairing:

```
pairing_t pairing;
char param[1024];
size_t count = fread(param, 1, 1024, stdin);
if (!count) pbc_die("input error");
pairing_init_set_buf(pairing, param, count);
```

Later we give pairing parameters to our program on standard input. Any file in the `param` subdirectory will suffice, for example:

```
$ bls < param/a.param
```

We shall need several `element_t` variables to hold the system parameters, keys and other quantities. We declare them and initialize them,

```
element_t g, h;
element_t public_key, secret_key;
```

```
element_t sig;
element_t temp1, temp2;

element_init_G2(g, pairing);
element_init_G2(public_key, pairing);
element_init_G1(h, pairing);
element_init_G1(sig, pairing);
element_init_GT(temp1, pairing);
element_init_GT(temp2, pairing);
element_init_Zr(secret_key, pairing);
```

generate system parameters,

```
element_random(g);
```

generate a private key,

```
element_random(secret_key);
```

and the corresponding public key.

```
element_pow_zn(public_key, g, secret_key);
```

When given a message to sign, we first compute its hash, using some standard hash algorithm. Many libraries can do this, and this operation does not involve pairings, so PBC does not provide functions for this step. For this example, and our message has already been hashed, possibly using another library.

Say the message hash is "ABCDEF" (a 48-bit hash). We map these bytes to an element h of G1,

```
element_from_hash(h, "ABCDEF", 6);
```

then sign it:

```
element_pow_zn(sig, h, secret_key);
```

To verify this signature, we compare the outputs of the pairing applied to the signature and system parameter, and the pairing applied to the message hash and public key. If the pairing outputs match then the signature is valid.

```
pairing_apply(temp1, sig, g, pairing);
pairing_apply(temp2, h, public_key, pairing);
if (!element_cmp(temp1, temp2)) {
    printf("signature verifies\n");
} else {
    printf("signature does not verify\n");
}
```

2.2. Import/export

To be useful, at some stage the signature must be converted to bytes for storage or transmission:

```
int n = pairing_length_in_bytes_compressed_G1(pairing);
// Alternatively:
// int n = element_length_in_bytes_compressed(sig);
unsigned char *data = malloc(n);
element_to_bytes_compressed(data, sig);
```

On the other end, the signature must be decompressed:

```
element_from_bytes_compressed(sig, data);
```

Eliding `_compressed` in the above code will also work but the buffer *data* will be roughly twice as large.

We can save more space by using the *x*-coordinate of the signature only

```
int n = pairing_length_in_bytes_x_only_G1(pairing);
// Alternative:
//   int n = element_length_in_bytes_x_only(sig);
unsigned char *data = malloc(n);
element_to_bytes_compressed(data, sig);
```

but then there is a complication during verification since two different points have the same *x*-coordinate. One way to solve this problem is to guess one point and try to verify. If that fails, we try the other. It can be shown that the pairing outputs of the two points are inverses of each other, avoiding the need to compute a pairing the second time. (In fact, there are even better ways to handle this.)

```
int n = pairing_length_in_bytes_x_only_G1(pairing);
//int n = element_length_in_bytes_x_only(sig);
unsigned char *data = malloc(n);

element_to_bytes_x_only(data, sig);

element_from_bytes_x_only(sig, data)

pairing_apply(temp1, sig, g, pairing);
pairing_apply(temp2, h, public_key, pairing);

if (!element_cmp(temp1, temp2)) {
    printf("signature verifies on first guess\n");
} else {
    element_invert(temp1, temp1);
    if (!element_cmp(temp1, temp2)) {
        printf("signature verifies on second guess\n");
    } else {
        printf("signature does not verify\n");
    }
```

```
}
```

Chapter 3. Pairing functions

An application should first initialize a pairing object. This causes PBC to setup curves, groups and other mathematical miscellany. After that, elements can be initialized and manipulated for cryptographic operations.

Parameters for various pairings are included with the PBC library distribution in the `param` subdirectory, and some are suitable for cryptographic use. Some programs in the `gen` subdirectory may be used to generate parameters (see Chapter 7). Also, see the PBC website for many more pairing parameters.

Pairings involve three groups of prime order. The PBC library calls them G1, G2, and GT, and calls the order r. The pairing is a bilinear map that takes two elements as input, one from G1 and one from G2, and outputs an element of GT.

The elements of G2 are at least as long as G1; G1 is guaranteed to be the shorter of the two. Sometimes G1 and G2 are the same group (i.e. the pairing is symmetric) so their elements can be mixed freely. In this case the `pairing_is_symmetric` function returns 1.

Bilinear pairings are stored in the data type `pairing_t`. Functions that operate on them start with `pairing_`.

3.1. Initializing pairings

To initialize a pairing from an ASCIIZ string:

```
pairing_t pairing;
pairing_init_set_str(pairing, s);  // Where s is a char *.
```

The string *s* holds *pairing parameters* in a text format. The `param` subdirectory contains several examples.

Alternatively, call:

```
pairing_t pairing;
pairing_init_pbc_param(pairing, param);
```

where *param* is an initialized `pbc_param_t` (see Chapter 5).

int **pairing_init_set_str**(*pairing_t pairing*, *const char *s*)

Initialize pairing from parameters in a ASCIIZ string *str* Returns 0 on success, 1 on failure.

int **pairing_init_set_buf**(*pairing_t pairing*, *const char *s*, *size_t len*)

Same, but read at most *len* bytes. If *len* is 0, it behaves as the previous function. Returns 0 on success, 1 on failure.

void **pairing_init_pbc_param**(*struct pairing_s *pairing*, *pbc_param_t p*)

Initialize a pairing with pairing parameters *p*.

void **pairing_clear**(*pairing_t pairing*)

Free the space occupied by *pairing*. Call whenever a `pairing_t` variable is no longer needed. Only call this after all elements associated with *pairing* have been cleared, as they need information stored in the *pairing* structure.

3.2. Applying pairings

The function `pairing_apply` can be called to apply a bilinear map. The order of the inputs is important. The first, which holds the output, must be from the group GT. The second must be from G1, the third from G2, and the fourth must be the `pairing_t` variable that relates them.

In some applications, the programmer may know that many pairings with the same G1 input will be computed. If so, preprocessing should be used to avoid repeating many calculations saving time in the long run. A variable of type `pairing_pp_t` should be declared, initialized with the fixed G1 element, and then used to compute pairings:

```
pairing_pp_t pp;
pairing_pp_init(pp, x, pairing); // x is some element of G1
pairing_pp_apply(r1, y1, pp); // r1 = e(x, y1)
pairing_pp_apply(r2, y2, pp); // r2 = e(x, y2)
pairing_pp_clear(pp); // don't need pp anymore
```

Never mix and match G1, G2, and GT groups from different pairings.

void **pairing_pp_init**(*pairing_pp_t p*, *element_t in1*, *pairing_t pairing*)

Get ready to perform a pairing whose first input is *in1*, and store the results of time-saving precomputation in *p*.

void **pairing_pp_clear**(*pairing_pp_t p*)

Clear *p*. This should be called after *p* is no longer needed.

void **pairing_pp_apply**(*element_t out*, *element_t in2*, *pairing_pp_t p*)

Compute a pairing using *in2* and the preprocessed information stored in *p* and store the output in *out*. The inputs to the pairing are the element previously used to initialize *p* and the element *in2*.

void **element_pairing**(*element_t out*, *element_t in1*, *element_t in2*)

Computes a pairing: *out* = *e*(*in1*, *in2*), where *in1*, *in2*, *out* must be in the groups G1, G2, GT.

void **element_prod_pairing**(*element_t out*, *element_t in1[]*, *element_t in2[]*, *int n*)

Computes the product of pairings, that is *out* = *e*(*in1*[0], *in2*[0]) ... *e*(*in1*[n-1], *in2*[n-1]). The arrays *in1*, *in2* must have at least *n* elements belonging to the groups G1, G2 respectively, and *out* must belong to the group GT.

3.3. Other pairing functions

int **pairing_is_symmetric**(*pairing_t pairing*)

Returns true if G1 and G2 are the same group.

int **pairing_length_in_bytes_G1**(*pairing_t pairing*)

Returns the length in bytes needed to represent an element of G1.

int **pairing_length_in_bytes_x_only_G1**(*pairing_t pairing*)

Returns the length in bytes needed to represent the x-coordinate of an element of G1.

int **pairing_length_in_bytes_compressed_G1**(*pairing_t pairing*)

Returns the length in bytes needed to represent a compressed form of an element of G1. There is some overhead in decompressing.

int **pairing_length_in_bytes_G2**(*pairing_t pairing*)

Returns the length in bytes needed to represent an element of G2.

int **pairing_length_in_bytes_compressed_G2**(*pairing_t pairing*)

Returns the length in bytes needed to represent a compressed form of an element of G2. There is some overhead in decompressing.

int **pairing_length_in_bytes_x_only_G2**(*pairing_t pairing*)

Returns the length in bytes needed to represent the x-coordinate of an element of G2.

int **pairing_length_in_bytes_GT**(*pairing_t pairing*)

Returns the length in bytes needed to represent an element of GT.

int **pairing_length_in_bytes_Zr**(*pairing_t pairing*)

Returns the length in bytes needed to represent an element of Zr.

Chapter 4. Element functions

Elements of groups, rings and fields are stored in the `element_t` data type. Variables of this type must be initialized before use, and should be cleared after they are no longer needed.

The `element_` functions must be used with caution. Just as division by zero does not make sense for integers, some operations may not make sense for particular elements. For example, in a ring, one cannot in general invert elements.

Another caveat is that many of these functions assume their arguments come from the same ring, group or field. No implicit type casting is performed.

For debug builds, turn on run-time checks by defining `PBC_DEBUG` before including `pbc.h`:

```
#define PBC_DEBUG
#include <pbc.h>
```

Also, when `PBC_DEBUG` is defined, the following macros are active. Normally they are replaced with empty statements.

PBC_ASSERT(expr, msg)

Macro: if `expr` evaluates to 0, print `msg` and exit.

PBC_ASSERT_MATCH2(a, b)

Macro: if elements `a` and `b` are from different fields then exit.

PBC_ASSERT_MATCH3(a, b, c)

Macro: if elements `a`, `b` and `c` are from different fields then exit.

4.1. Initializing elements

When an element is initialized it is associated with an algebraic structure, such as a particular finite field or elliptic curve group.

We use G1 and G2 to denote the input groups to the pairing, and GT for the output group. All have order r, and Zr means the ring of integers modulo r. G1 is the smaller group (the group of points over the base field). With symmetric pairings, G1 = G2.

void **element_init_G1**(*element_t e*, *pairing_t pairing*)

void **element_init_G2**(*element_t e*, *pairing_t pairing*)

void **element_init_GT**(*element_t e*, *pairing_t pairing*)

Initialize *e* to be an element of the group G1, G2 or GT of *pairing*.

void **element_init_Zr**(*element_t e*, *pairing_t pairing*)

Initialize *e* to be an element of the ring Z_r of *pairing*. r is the order of the groups G1, G2 and GT that are involved in the pairing.

void **element_init_same_as**(*element_t e*, *element_t e2*)

Initialize *e* to be an element of the algebraic structure that *e2* lies in.

void **element_clear**(*element_t e*)

Free the space occupied by *e*. Call this when the variable *e* is no longer needed.

4.2. Assigning elements

These functions assign values to elements. When integers are assigned, they are mapped to algebraic structures canonically if it makes sense (e.g. rings and fields).

void **element_set0**(*element_t e*)

Set *e* to zero.

void **element_set1**(*element_t e*)

Set *e* to one.

void **element_set_si**(*element_t e*, *signed long int i*)

Set *e* to *i*.

void **element_set_mpz**(*element_t e*, *mpz_t z*)

Set *e* to *z*.

void **element_set**(*element_t e*, *element_t a*)

Set *e* to *a*.

4.3. Converting elements

void **element_to_mpz**(*mpz_t z*, *element_t e*)

Converts *e* to a GMP integer *z* if such an operation makes sense

void **element_from_hash**(*element_t e*, *void *data*, *int len*)

Generate an element *e* deterministically from the *len* bytes stored in the buffer *data*.

4.4. Element arithmetic

Unless otherwise stated, all `element_t` arguments to these functions must have been initialized to be from the same algebraic structure. When one of these functions expects its arguments to be from particular algebraic structures, this is reflected in the name of the function.

The addition and multiplication functions perform addition and multiplication operations in rings and fields. For groups of points on an ellitpic curve, such as the G1 and G2 groups associated with pairings, both addition and multiplication represent the group operation (and similarly both 0 and 1 represent the identity element). It is recommended that programs choose and one convention and stick with it to avoid confusion.

In contrast, the GT group is currently implemented as a subgroup of a finite field, so only multiplicative operations should be used for GT.

void **element_add**(*element_t n*, *element_t a*, *element_t b*)

Set *n* to *a* + *b*.

void **element_sub**(*element_t n*, *element_t a*, *element_t b*)

Set *n* to *a* - *b*.

void **element_mul**(*element_t n*, *element_t a*, *element_t b*)

Set *n* = *a b*.

void **element_mul_mpz**(*element_t n*, *element_t a*, *mpz_t z*)

void **element_mul_si**(*element_t n*, *element_t a*, *signed long int z*)

Set *n* = *a z*, that is *a* + *a* + ... + *a* where there are *z* *a*'s.

void **element_mul_zn**(*element_t c*, *element_t a*, *element_t z*)

z must be an element of a integer mod ring (i.e. $\mathbf{Z}_n$ for some n). Set $c = a\ z$, that is $a + a + \ldots + a$ where there are *z* *a*'s.

void **element_div**(*element_t n*, *element_t a*, *element_t b*)

Set $n = a / b$.

void **element_double**(*element_t n*, *element_t a*)

Set $n = a + a$.

void **element_halve**(*element_t n*, *element_t a*)

Set $n = a/2$

void **element_square**(*element_t n*, *element_t a*)

Set $n = a^2$

void **element_neg**(*element_t n*, *element_t a*)

Set $n = -a$.

void **element_invert**(*element_t n*, *element_t a*)

Set *n* to the inverse of *a*.

4.5. Exponentiating elements

Exponentiation and multiexponentiation functions. If it is known in advance that a particular element will be exponentiated several times in the future, time can be saved in the long run by first calling the preprocessing function:

```
element_pp_t g_pp;
element_pp_init(g_pp, g);
element_pp_pow(h, pow1, g_pp); // h = g^pow1
element_pp_pow(h, pow2, g_pp); // h = g^pow2
element_pp_pow(h, pow3, g_pp); // h = g^pow3
element_pp_clear(g_pp);
```

void **element_pow_mpz**(*element_t x*, *element_t a*, *mpz_t n*)

Set $x = a^n$, that is a times a times … times a where there are n a's.

void **element_pow_zn**(*element_t x*, *element_t a*, *element_t n*)

Set $x = a^n$, where n is an element of a ring $\mathbf{Z}_N$ for some N (typically the order of the algebraic structure x lies in).

void **element_pow2_mpz**(*element_t x*, *element_t a1*, *mpz_t n1*, *element_t a2*, *mpz_t n2*)

Sets $x = a1^{n1}\ a2^{n2}$, and is generally faster than performing two separate exponentiations.

void **element_pow2_zn**(*element_t x*, *element_t a1*, *element_t n1*, *element_t a2*, *element_t n2*)

Also sets $x = a1^{n1}\ a2^{n2}$, but $n1$, $n2$ must be elements of a ring $\mathbf{Z}_n$ for some integer n.

void **element_pow3_mpz**(*element_t x*, *element_t a1*, *mpz_t n1*, *element_t a2*, *mpz_t n2*, *element_t a3*, *mpz_t n3*)

Sets $x = a1^{n1}\ a2^{n2}\ a3^{n3}$, generally faster than performing three separate exponentiations.

void **element_pow3_zn**(*element_t x*, *element_t a1*, *element_t n1*, *element_t a2*, *element_t n2*, *element_t a3*, *element_t n3*)

Also sets $x = a1^{n1}\ a2^{n2}\ a3^{n3}$, but *n1*, *n2*, *n3* must be elements of a ring $\mathbf{Z}_n$ for some integer n.

void **element_pp_init**(*element_pp_t p*, *element_t in*)

Prepare to exponentiate an element *in*, and store preprocessing information in *p*.

void **element_pp_clear**(*element_pp_t p*)

Clear *p*. Should be called after *p* is no longer needed.

void **element_pp_pow**(*element_t out*, *mpz_t power*, *element_pp_t p*)

Raise *in* to *power* and store the result in *out*, where *in* is a previously preprocessed element, that is, the second argument passed to a previous **element_pp_init** call.

void **element_pp_pow_zn**(*element_t out*, *element_t power*, *element_pp_t p*)

Same except *power* is an element of $\mathbf{Z}_n$ for some integer n.

void **element_dlog_brute_force**(*element_t x*, *element_t g*, *element_t h*)

Computes *x* such that $g^x = h$ by brute force, where *x* lies in a field where `element_set_mpz()` makes sense.

void **element_dlog_pollard_rho**(*element_t x*, *element_t g*, *element_t h*)

Computes *x* such that $g^x = h$ using Pollard rho method, where *x* lies in a field where `element_set_mpz()` makes sense.

4.6. Comparing elements

These functions compare elements from the same algebraic structure.

int **element_is1**(*element_t n*)

Returns true if *n* is 1.

int **element_is0**(*element_t n*)

Returns true if *n* is 0.

int **element_cmp**(*element_t a*, *element_t b*)

Returns 0 if *a* and *b* are the same, nonzero otherwise.

int **element_is_sqr**(*element_t a*)

Returns nonzero if *a* is a perfect square (quadratic residue), zero otherwise.

int **element_sgn**(*element_t a*)

int **element_sign**(*element_t a*)

If *a* is zero, returns 0. For nozero *a* the behaviour depends on the algebraic structure, but has the property that element_sgn(*a*) = -element_sgn(-*a*) and element_sgn(*a*) = 0 implies *a* = 0 with overwhelming probability.

4.7. Element I/O

Functions for producing human-readable outputs for elements. Converting elements to and from bytes are discussed later.

size_t **element_out_str**(*FILE * stream*, *int base*, *element_t e*)

Output *e* on *stream* in base *base*. The base must be between 2 and 36.

int **element_printf**(*const char *format*, . . .)

int **element_fprintf**(*FILE * stream*, *const char *format*, . . .)

int **element_snprintf**(*char *buf*, *size_t size*, *const char *fmt*, . . .)

int **element_vsnprintf**(*char *buf*, *size_t size*, *const char *fmt*, *va_list ap*)

Same as printf family except also has the *B* conversion specifier for types of **element_t**, and *Y*, *Z* conversion specifiers for `mpz_t`. For example if *e* is of type `element_t` then

```
element_printf("%B\n", e);
```

will print the value of *e* in a human-readable form on standard output.

int **element_snprint**(*char *s*, *size_t n*, *element_t e*)

Convert an element to a human-friendly string. Behaves as **snprintf** but only on one element at a time.

int **element_set_str**(*element_t e*, *const char *s*, *int base*)

Set the element *e* from *s*, a null-terminated C string in base *base*. Whitespace is ignored. Points have the form "[*x,y*]" or "*O*", while polynomials have the form "[*a0,. . . ,an*]". Returns number of characters read (unlike GMP's mpz_set_str). A return code of zero means PBC could not find a well-formed string describing an element.

4.8. Random elements

Only works for finite algebraic structures. Effect on polynomial rings, fields of characteristic zero, etc. undefined.

See Section 6.1 for how PBC gets random bits.

void **element_random**(*element_t e*)

If the *e* lies in a finite algebraic structure, assigns a uniformly random element to *e*.

4.9. Element import/export

Functions for serializing and deserializing elements.

int **element_length_in_bytes**(*element_t e*)

Returns the length in bytes the element *e* will take to represent

int **element_to_bytes**(*unsigned char *data*, *element_t e*)

Converts *e* to byte, writing the result in the buffer *data*. The number of bytes it will write can be determined from calling **element_length_in_bytes()**. Returns number of bytes written.

int **element_from_bytes**(*element_t e*, *unsigned char *data*)

Reads *e* from the buffer *data*, and returns the number of bytes read.

int **element_to_bytes_x_only**(*unsigned char *data*, *element_t e*)

Assumes *e* is a point on an elliptic curve. Writes the x-coordinate of *e* to the buffer *data*

int **element_from_bytes_x_only**(*element_t e*, *unsigned char *data*)

Assumes *e* is a point on an elliptic curve. Sets *e* to a point with x-coordinate represented by the buffer *data*. This is not unique. For each *x*-coordinate, there exist two different points, at least for the elliptic curves in PBC. (They are inverses of each other.)

int **element_length_in_bytes_x_only**(*element_t e*)

Assumes *e* is a point on an elliptic curve. Returns the length in bytes needed to hold the x-coordinate of *e*.

int **element_to_bytes_compressed**(*unsigned char *data*, *element_t e*)

If possible, outputs a compressed form of the element *e* to the buffer of bytes *data*. Currently only implemented for points on an elliptic curve.

int **element_from_bytes_compressed**(*element_t e*, *unsigned char *data*)

Sets element *e* to the element in compressed form in the buffer of bytes *data*. Currently only implemented for points on an elliptic curve.

int **element_length_in_bytes_compressed**(*element_t e*)

Returns the number of bytes needed to hold *e* in compressed form. Currently only implemented for points on an elliptic curve.

int **element_item_count**(*element_t e*)

For points, returns the number of coordinates. For polynomials, returns the number of coefficients. Otherwise returns zero.

element_t **element_item**(*element_t e*, *int i*)

For points, returns *n*th coordinate. For polynomials, returns coefficient of x^n. Otherwise returns NULL. The element the return value points to may be modified.

element_t **element_x**(*element_t a*)

Equivalent to `element_item(a, 0)`.

element_t **element_y**(*element_t a*)

Equivalent to `element_item(a, 1)`.

Chapter 5. Param functions

Pairings are initialized from *pairing parameters*, which are objects of type `pbc_param_t`. Some applications can ignore this data type because `pairing_init_set_str()` handles it behind the scenes: it reads a string as a `pbc_param_t`, then initializes a pairing with these parameters.

int **pbc_param_init_set_str**(*pbc_param_t par*, *const char *s*)

Initializes pairing parameters from the string *s*. Returns 0 if successful, 1 otherwise.

int **pbc_param_init_set_buf**(*pbc_param_t par*, *const char *s*, *size_t len*)

Same, but read at most *len* bytes. If *len* is 0, it behaves as the previous function. Returns 0 if successful, 1 otherwise.

void **pbc_param_out_str**(*FILE *stream*, *pbc_param_t p*)

Write pairing parameters to '*stream*' in a text format.

void **pbc_param_clear**(*pbc_param_t p*)

Clear *p*. Call after *p* is no longer needed.

5.1. Param generation

These were used to prepare the sample parameters in the `param` subdirectory.

We label the pairing families with capital letters roughly in the order of discovery, so we can refer to them easily. Type A is fastest. Type D is a good choice when elements should be short but is slower. Type F has even shorter elements but is slower still. The speed differences are hardware-dependent, and also change when preprocessing is used. Type B and C are unimplemented.

The `pbc_cm_t` data type holds CM parameters that are used to generate type D and G curves.

void **pbc_cm_init**(*pbc_cm_t cm*)

Initializes *cm*.

void **pbc_cm_clear**(*pbc_cm_t cm*)

Clears *cm*.

int **pbc_cm_search_d**(*int (*callback)(pbc_cm_t, void *)*, *void *data*, *unsigned int D*, *unsigned int bitlimit*)

For a given discriminant D, searches for type D pairings suitable for cryptography (MNT curves of embedding degree 6). The group order is at most *bitlimit* bits. For each set of CM parameters found, call *callback* with `pbc_cm_t` and given `void *`. If the callback returns nonzero, stops search and returns that value. Otherwise returns 0.

int **pbc_cm_search_g**(*int (*callback)(pbc_cm_t, void *)*, *void *data*, *unsigned int D*, *unsigned int bitlimit*)

For a given discriminant D, searches for type G pairings suitable for cryptography (Freeman curve). The group order is at most *bitlimit* bits. For each set of CM parameters found, call *callback* with `pbc_cm_t` and given `void *`. If the callback returns nonzero, stops search and returns that value. Otherwise returns 0.

void **pbc_param_init_a_gen**(*pbc_param_t par*, *int rbits*, *int qbits*)

Generate type A pairing parameters and store them in *p*, where the group order r is *rbits* long, and the order of the base field q is *qbits* long. Elements take *qbits* to represent.

To be secure, generic discrete log algorithms must be infeasible in groups of order r, and finite field discrete log algorithms must be infeasible in finite fields of order q^2, e.g. *rbits* = 160, *qbits* = 512.

The file `param/a.param` contains parameters for a type A pairing suitable for cryptographic use.

void **pbc_param_init_a1_gen**(*pbc_param_t param*, *mpz_t n*)

Generate type A1 pairing parameters and store them in *p*. The group order will be *n*. The order of the base field is a few bits longer. To be secure, generic discrete log algorithms must be infeasible in groups of order *n*, and finite field discrete log algorithms must be infeasible in finite fields of order roughly n^2. Additionally, *n* should be hard to factorize.

For example: *n* a product of two primes, each at least 512 bits.

The file `param/a1.param` contains sample parameters for a type A1 pairing, but it is only for benchmarking: it is useless without the factorization of `n`, the order of the group.

void **pbc_param_init_d_gen**(*pbc_param_t p*, *pbc_cm_t cm*)

Type D curves are generated using the complex multiplication (CM) method. This function sets *p* to a type D pairing parameters from CM parameters *cm*. Other library calls search for appropriate CM parameters and the results can be passed to this function.

To be secure, generic discrete log algorithms must be infeasible in groups of order r, and finite field discrete log algorithms must be infeasible in finite fields of order q^6. For usual CM parameters, r is a few bits smaller than q.

Using type D pairings allows elements of group G1 to be quite short, typically 170-bits. Because of a certain trick, elements of group G2 need only be 3 times longer, that is, about 510 bits rather than 6 times long. They are not quite as short as type F pairings, but much faster.

I sometimes refer to a type D curve as a triplet of numbers: the discriminant, the number of bits in the prime q, and the number of bits in the prime r. The `gen/listmnt` program prints these numbers.

Among the bundled type D curve parameters are the curves 9563-201-181, 62003-159-158 and 496659-224-224 which have shortened names `param/d201.param`, `param/d159.param` and `param/d225.param` respectively.

See `gen/listmnt.c` and `gen/gendparam.c` for how to generate type D pairing parameters.

void **pbc_param_init_e_gen**(*pbc_param_t p*, *int rbits*, *int qbits*)

Generate type E pairing parameters and store them in *p*, where the group order r is *rbits* long, and the order of the base field q is *qbits* long. To be secure, generic discrete log algorithms must be infeasible in groups of order r, and finite field discrete log algorithms must be infeasible in finite fields of order q, e.g. *rbits* = 160, *qbits* = 1024.

This pairing is just a curiosity: it can be implemented entirely in a field of prime order, that is, only arithmetic modulo a prime is needed and there is never a need to extend a field.

If discrete log in field extensions are found to be substantially easier to solve than previously thought, or discrete log can be solved in elliptic curves as easily as they can be in finite fields, this pairing type may become useful.

void **pbc_param_init_f_gen**(*pbc_param_t p*, *int bits*)

Generate type F pairing parameters and store them in *p*. Both the group order r and the order of the base field q will be roughly *bits*-bit numbers. To be secure, generic discrete log algorithms must be infeasible in groups of order r, and finite field discrete log algorithms must be infeasible in finite fields of order q^12, e.g. *bits* = 160.

Type F should be used when the top priority is to minimize bandwidth (e.g. short signatures). The current implementation makes them slow.

If finite field discrete log algorithms improve further, type D pairings will have to use larger fields, but type F can still remain short, up to a point.

void **pbc_param_init_g_gen**(*pbc_param_t p*, *pbc_cm_t cm*)

Type G curves are generated using the complex multiplication (CM) method. This function sets *p* to a type G pairing parameters from CM parameters *cm*. They have embedding degree 10.

To be secure, generic discrete log algorithms must be infeasible in groups of order r, and finite field discrete log algorithms must be infeasible in finite fields of order q^6. For usual CM parameters, r is a few bits smaller than q.

They are quite slow at the moment so for now type F is a better choice.

The file `param/g149.param` contains parameters for a type G pairing with 149-bit group and field sizes.

Chapter 6. Other functions

Random number generation, memory allocation, logging.

6.1. Random bits

The first time PBC is asked to generate a random number, the library will try to open the file `/dev/urandom` as a source of random bits. If this fails, PBC falls back to a deterministic random number generator (which is of course completely useless for cryptography).

It is possible to change the file used for random bits. Also, explicitly selecting the deterministic random number generator will suppress the warning.

On Windows, by default, PBC uses the Microsoft Crypto API to generate random bits.

void **pbc_random_set_file**(*char *filename*)

Sets *filename* as a source of random bytes. For example, on Linux one might use `/dev/random`.

void **pbc_random_set_deterministic**(*unsigned int seed*)

Uses a determinstic random number generator, seeded with *seed*.

void **pbc_random_set_function**(*void (*fun)(mpz_t*, *mpz_t*, *void *)*, *void *data*)

Uses given function as a random number generator.

void **pbc_mpz_random**(*mpz_t z*, *mpz_t limit*)

Selects a random *z* that is less than *limit*.

void **pbc_mpz_randomb**(*mpz_t z*, *unsigned int bits*)

Selects a random *bits*-bit integer *z*.

6.2. Custom allocation

Like GMP, PBC can be instructed to use custom memory allocation functions. This must be done before any memory allocation is performed, usually at the beginning of a program before any other PBC functions have been called.

Also like GMP, the PBC wrappers around `malloc` and `realloc` will print a message on standard error and terminate program execution if the calls fail. Replacements for these functions should act similarly.

However, unlike GMP, PBC does not pass the number of bytes previously allocated along with the pointer in calls to `realloc` and `free`.

void **pbc_set_memory_functions**(*void *(*malloc_fn)(size_t)*, *void *(*realloc_fn)(void *, size_t)*, *void (*free_fn)(void *)*)

Set custom allocation functions. The parameters must be function pointers to drop-in replacements for malloc, realloc and free, except that malloc and realloc should terminate the program on failure: they must not return in this case.

6.3. Logging

int **pbc_set_msg_to_stderr**(*int i*)

By default error messages are printed to standard error. Call `pbc_set_msg_to_stderr(0)` to suppress messages.

void **pbc_die**(*const char *err*, ...)

Reports error message and exits with code 128.

void **pbc_info**(*const char *err*, ...)

Reports informational message.

void **pbc_warn**(*const char *err*, . . .)

Reports warning message.

void **pbc_error**(*const char *err*, . . .)

Reports error message.

Chapter 7. Bundled programs

Several binaries and curve parameters are bundled with the PBC library, such as the `pbc` program.

The `param` subdirectory contains pairing parameters one might use in a real cryptosystem. Many of the test programs read the parameters from files such as these on standard input, for example:

```
$ benchmark/benchmark < param/c159.param
$ example/bls < param/e.param
```

7.1. Pairing-based calculator

The `pbc` subdirectory contains the pairing-based calculator, `pbc`, which is loosely based on `bc`, a well-known arbitrary precision calculator.

See `pairing_test.pbc` for an example script. Some differences: the assignment operator is `:=`, and newlines are ordinary whitespace and not statement terminators.

If started with the `-y` option, the syntax is compatible with `bc`: newlines are treated as statement terminators and = is assignment. Additionally, `pbc` displays a prompt. This mode may be easier for beginners.

Initially, the variables G1, G2, GT and Zr are represent groups associated with a particular A pairing.

An element is represented with a tree of integers, such as `[[1,2], 3]`, or `4`.

Assignments such as `variable := expression;` return the value of the variable.

The arithmetic operators `+, -, /, *, ^` have the standard precedence. The C comparison operators and ternary operator are available.

Each statement should be terminated by a semicolon.

Comments are the same as in (original) C, or begin with "#" and end at a newline.

Some of the pbc functions:

`init_pairing_A()`

Set the variables G1, G2, GT and Zr to the groups in a particular A pairing:

```
init_pairing_A();
```

Other sample pairings can be used by replacing `A` with one of `D, E, F, G`.

`rnd(`*G*`)`

Returns a random element of an algebraic structure *G*, e.g:

```
g := rnd(Zr);
```

Synonym: `random`.

`pairing(`*g*, *h*`)`

Returns the pairing applied to *g* and *h*. The element *g* must be an element of G1 and *h* of G2, e.g:

```
pairing(rnd(G1), rnd(G2));
```

G`(`*g*`)`

Maps an element *g* to element of the field *G*, e.g:

```
Zr(123);
GT([456, 789]);
```

7.2. Parameter generation

Programs that generate pairing parameters are located in the `gen` subdirectory. Some of the programs are already functional enough to be used to find parameters for real applications. I need to write more documentation first; for now, read the source!

listmnt

Searches for discriminants D that lead to MNT curves with subgroups of prime order.

genaparam, **gena1param**, **gendparam**, **geneparam**, **genfparam**, **gengparam**

Prints parameters for a curve suitable for computing pairings of a given type. The output can be fed to some of the other test programs. The programs `gendparam` and `gengparam` should be given a discriminant as the first argument.

hilbertpoly

Prints the Hilbert polynomial for a given range of discriminants. Computing the Hilbert polynomial is an intermediate step when generating type D parameters.

7.3. Example cryptosystems

In the `example` subdirectory there are various programs that read curve parameters on standard input and perform computations that would be required in a typical implementation of a pairing-based cryptosystem. Sample schemes include:

- Boneh-Lynn-Shacham short signatures
- Hess identity-based signatures
- Joux tripartite Diffie-Hellman
- Paterson identity-based signatures
- Yuan-Li identity-based authenticated key agreement
- Zhang-Kim identity-based blind/ring signatures
- Zhang-Safavi-Naini-Susilo signatures

More work would be required to turn these programs into real applications.

7.4. Benchmarks

I use the programs in the `benchmark` subdirectory to measure running times of pairings, and also RSA decryptions.

The `benchmark` program takes pairing parameters on standard input and reports the average running time of the pairing over 10 runs, while `timersa` estimates the time required to perform one 1024-bit RSA decryption.

Chapter 8. PBC internals

The source code is organized by subdirectories:

include: Headers describing the official API. Headers in other places are for internal use only.

arith: Finite fields: modular arithmetic, polynomial rings, and polynomial rings modulo a polynomial. Finite fields of low characteristic are unsupported.

ecc: Elliptic curve generation, elliptic curve groups and pairings. One source file is dedicated to each type of pairing, containing specialized optimizations. Some of the code requires arbitrary precision complex numbers, which also live here but should be moved elsewhere one day.

misc: Dynamic arrays, symbol tables, benchmarking, logging, debugging, other utilities.

gen: Programs that generate pairing parameters and list Hilbert polynomials. These were used to prepare the samples in the `param` directory.

example: Example programs showing how to use the library.

guru: Tests, experimental code.

8.1. Groups, rings, fields

Algebraic structures are represented in the `field_t` data type, which mostly contains pointers to functions written to perform operations such as addition and multiplication in that particular group, ring or field:

```
struct field_s {
  ...
  void (*init)(element_ptr);
  void (*clear)(element_ptr);
  ...
  void (*add)(element_ptr, element_ptr, element_ptr);
  void (*sub)(element_ptr, element_ptr, element_ptr);
  void (*mul)(element_ptr, element_ptr, element_ptr);
  ...
};
typedef struct field_s *field_ptr;
typedef struct field_s field_t[1];
```

The name `algebraic_structure_t` is arguably more accurate, but far too cumbersome. It may help if one views groups and rings as handicapped fields.

The last two lines of the above code excerpt show how GMP and PBC define data types: they are arrays of length one so that when a variable is declared, space is automatically allocated for it on the stack. Yet when used as a argument to a function, a pointer is passed, thus there is no need to explicitly allocate and deallocate memory, nor reference and dereference variables.

Each `element_t` contains a field named `field` to such a `field_t` variable. The only other field is `data`, which stores any data needed for the implementation of the particular algebraic structure the element resides in.

```
struct element_s {
  struct field_s *field;
  void *data;
};
```

When an `element_t` variable is initialized, `field` is set appropriately, and then the initialization specific to that field is called to complete the initialization. Here, a line of code is worth a thousand words:

```
void element_init(element_t e, field_ptr f) {
  e->field = f;
  f->init(e);
}
```

Thus during a call to one of the `element_` functions, the `field` pointer is followed then the appropriate routine is executed. For example, modular addition results when the input element is an element of a finite field, while polynomial addition is performed for elements of a polynomial ring and so on.

```
void element_add(element_t n, element_t a, element_t b) {
  n->field->add(n, a, b);
}
```

My design may seem dangerous because if a programmer inadvertently attempts to add a polynomial and a point on an elliptic curve, say, the code will compile without warnings since they have the same data type.

However I settled on having a catch-all "glorified `void *`" `element_t` because I wanted to

- extend a field an arbitrary number of times (though in practice, currently I only need to extend a field twice at most),
- switch fields easily, so for example a program that benchmarks addition in polynomial rings can be trivially modified to benchmark addition in a group, and
- interchange different implementations of the same algebraic structure, for example, compare Montgomery representation versus a naive implementation of integer modulo rings.

Additionally, defining `PBC_DEBUG` catches many type mismatches.

In mathematics, groups, rings and fields should be distinguished, but for implmentation, it is simplest lump them together under the same heading. In any event, distinct data types may lead to a false sense of security. Fields of prime order with different moduli would still fall under the same data type, with unpleasant results if their elements are mistakenly mixed.

I have vague plans to add flags to `field_t` describing the capabilities of a particular `field_t`. These flags would be set during initialization, and would indicate for example whether one can invert every nonzero element, whether there are one or two operations (that is, group versus ring), whether the field is an integer mod ring, polynomial ring, or polynomial mod ring, and so on. Once in place, more runtime checks can be performed to avoid illegal inversion and similar problems.

Another option is to introduce data types for each of the four pairing-related algebraic structures, namely G1, G2, GT and Zr, as these are the only ones needed for implementing pairing-based cryptosystems.

An alternative was to simply use `void *` instead of `element_t` and require the programmer to pass the field as a parameter, e.g. `element_add(a, b, c, F_13)`, but I decided the added annoyance of having to type this extra variable every time negated any benefits, such as obviating the need for the `field` pointer in `struct element_s`, even if one ignores the more serious problem that runtime type checking is considerably harder, if not impossible.

I suppose one could write a preprocessor to convert one type of notation to the other, but I would like the code to be standard C. (On the other hand, as Hovav Shacham suggested, it may be nice to eventually have a converter that takes human-friendly infix operator expressions like `a = (b + c) * d` and outputs the assembly-like `element_` equivalents.)

8.2. Internal randomness

Some algorithms require a quadratic nonresidue in a given field. These are computed lazily: The first time a quadratic nonresidue is requested, one is generated at random, using the same source of random bits as other PBC random functions. [Which reminds me, should I get rid of the `nqr` field and instead have it as part of the `data` field in struct field_s?]

In `fieldquadratic.c`, a quadratic field extension is constructed with a square root of this randomly generated quadratic nonresidue in the base field. Thus for a nondeterminstic source of random bits, the same field may be constructed differently on different runs.

To construct the same field the same way every time, one must record the quadratic nonresidue generated from one run, and call `field_set_nqr()` every time this particular construction of a quadratic field extension is desired. Another use for this function is to save time by setting the quadratic nonresidue to some precomputed value.

Similarly, for higher degree extensions, a random irreducible polynomial may be chosen to construct it, but this must be recorded if the same construction is later required.

This happens behind the scenes in PBC.

8.3. Type A internals

Type A pairings are constructed on the curve $y^2 = x^3 + x$ over the field F_q for some prime q = 3 mod 4. Both G1 and G2 are the group of points E(F_q), so this pairing is symmetric. It turns out #E(F_q) = q + 1 and #E(F_q^2) = $(q + 1)^2$. Thus the embedding degree k is 2, and hence GT is a subgroup of F_q^2. The order r is some prime factor of q + 1.

Write q + 1 = r * h. For efficiency, r is picked to be a Solinas prime, that is, r has the form 2^a +- 2^b +- 1 for some integers 0 < b < a.

Also, we choose q = -1 mod 12 so F_q^2 can be implemented as F_q[i] (where i = sqrt(-1)) and since q = -1 mod 3, cube roots in F_q are easy to compute. This latter feature may be removed because I have not found a use for it yet (in which case we only need q = -1 mod 4).

`a_param` struct fields:

```
exp2, exp1, sign1, sign0, r:
  r = 2^exp2 + sign1 * 2^exp1 + sign0 * 1 (Solinas prime)
q, h:
  r * h = q + 1
  q is a prime, h is a multiple of 12 (thus q = -1 mod 12)
```

Type A1 uses the same equation, but have different fields since the library is given r and cannot choose it.

`a1_param` struct fields:

```
p, n, l:
  p + 1 = n * l
  p is prime, same as the q in a_param, n is the order of the group.
```

8.4. Type B internals

Unimplemented. Similar to type A. The curve $y^2 = x^3 + 1$ over the field F_q for some prime q = 2 mod 3, which implies cube roots in F_q are easy to compute, though we can achieve this for type A pairings by constraining q appropriately. I recommend requiring q = 3 mod 4 as well, so that -1 is a quadratic nonresidue.

The lack of an x term simplifies some routines such as point doubling.

It turns out we must choose between symmetry or efficiency due to the nature of a certain optimization.

8.5. Type C internals

Unimplemented. The supersingular curves $y^2 = x^3 + 2x + 1$ and $y^2 = x^3 + 2x - 1$ over a field of characteristic 3. Discussed at length by Boneh, Lynn, and Shacham, "Short signatures from the Weil pairing". Many optimizations can be applied to speed up these pairings; see Barreto et al., "Efficient algorithms for pairing-based cryptosystems", but sadly, an attack due to Coppersmith makes these curves less attractive.

8.6. Type D internals

These are ordinary curves of with embedding degree 6, whose orders are prime or a prime multiplied by a small constant.

A type D curve is defined over some field F_q and has order h * r where r is a prime and h is a small constant. Over the field F_q^6 its order is a multiple of r^2.

Typically the order of the curve E is around 170 bits, as is F_q, the base field, thus q^k is around the 1024-bit mark which is commonly considered good enough.

`d_param` struct fields:

```
q   F_q is the base field
n   # of points in E(F_q)
r   large prime dividing n
h   n = h * r
a   E: y^2 = x^3 + ax + b
b
nk  # of points in E(F_q^k)
hk  nk = hk * r * r
coeff0 coefficients of a monic cubic irreducible over F_q
coeff1
coeff2
nqr quadratic nonresidue in F_q
```

These were discovered by Miyaji, Nakabayashi and Takano, "New explicit conditions of elliptic curve traces for FR-reduction".

8.7. Type E Internals

The CM (Complex Multiplication) method of constructing elliptic curves starts with the Diophantine equation

```
DV^2 = 4q - t^2
```

If t = 2 and q = D r^2 h^2 + 1 for some prime r (which we choose to be a Solinas prime) and some integer h, we find that this equation is easily solved with V = 2rh.

Thus it is easy to find a curve (over the field F_q) with order q - 1. Note r^2 divides q - 1, thus we have an embedding degree of 1.

Hence all computations necessary for the pairing can be done in F_q alone. There is never any need to extend F_q.

As q is typically 1024 bits, group elements take a lot of space to represent. Moreover, many optimizations do not apply to this type, resulting in a slower pairing.

`e_param` struct fields:

```
exp2, exp1, sign1, sign0, r:
  r = 2^exp2 + sign1 * 2^exp1 + sign0 * 1 (Solinas prime)
q, h
  q = h r^2 + 1 where r is prime, and h is 28 times a perfect square
a, b
  E: y^2 = x^3 + ax + b
```

8.8. Type F internals

Using carefully crafted polynomials, k = 12 pairings can be constructed. Only 160 bits are needed to represent elements of one group, and 320 bits for the other.

Also, embedding degree k = 12 allows higher security short signatures. (k = 6 curves cannot be used to scale security from 160-bits to say 256-bits because finite field attacks are subexponential.)

`f_param` struct fields:

```
q:
  The curve is defined over Fq
r:
  The order of the curve.
```

```
b:
  E: y^2= x^3 + b
beta:
  A quadratic nonresidue in Fq: used in quadratic extension.
alpha0, alpha1:
  x^6 + alpha0 + alpha1 sqrt(beta) is irreducible: used in sextic extension.
```

Discovered by Barreto and Naehrig, "Pairing-friendly elliptic curves of prime order".

8.9. Type G Internals

Another construction based on the CM method.

`g_param` struct fields:

```
q, n, h, r:
  h * r = n is the order of E(F_q)
a, b:
  E: y^2 = x^3 + ax + b
nk:
  #E(F_q^10)
hk:
  hk * r^2 = nk
coeff:
  array of coefficients of polynomial used for quintic extension.
nqr:
  a quadratic nonresidue
```

`g_param` struct fields:

Discovered by Freeman, "Constructing pairing-friendly elliptic curves with embedding degree 10."

8.10. Testing functions

For testing, debugging, demonstrations and benchmarks. Declared in `pbc_test.h`:

void **pbc_demo_pairing_init**(*pairing_t pairing*, *int argc*, *char **argv*)

Initializes pairing from file specified as first argument, or from standard input if there is no first argument.

double **pbc_get_time**(*void*)

Returns seconds elapsed since the first call to this function. Returns 0 the first time.

EXPECT(condition)

Macro: if `condition` evaluates to 0 then print an error.

int **pbc_err_count**

Total number of failed EXPECT checks.

8.11. Dynamic arrays

The `darray_t` data type manages an array of pointers of type `void \*`, allocating more memory when necessary. Declared in `pbc_darray.h`.

void **darray_init**(*darray_t a*)

Initialize a dynamic array *a*. Must be called before *a* is used.

void **darray_clear**(*darray_t a*)

Clears a dynamic array *a*. Should be called after *a* is no longer needed.

void **darray_append**(*darray_t a*, *void *p*)

Appends *p* to the dynamic array *a*.

void * **darray_at**(*darray_t a*, *int i*)

Returns the pointer at index *i* in the dynamic array *a*.

void **darray_remove_index**(*darray_t a*, *int n*)

Removes the pointer at index *i* in the dynamic array *a*.

int **darray_count**(*darray_t a*)

Returns the number of pointers held in *a*.

8.12. Symbol tables

The `symtab_t` data type manages symbol tables where the keys are strings of type `char *` and the values are pointers of type `void \*`.

At present, they are implemented inefficiently using dynamic arrays, but this will change if the need arises. They are only used when reading a `pbc_param_t` from a string. Declared in `pbc_symtab.h`.

void **symtab_init**(*symtab_t t*)

Initialize symbol table *t*. Must be called before *t* is used.

void **symtab_clear**(*symtab_t t*)

Clears symbol table *t*. Should be called after *t* is no longer needed.

void **symtab_put**(*symtab_t t*, *void *value*, *const char *key*)

Puts *value* at *key* in *t*.

int **symtab_has**(*symtab_t t*, *const char *key*)

Returns true if *t* contains key *key*.

void * **symtab_at**(*symtab_t t*, *const char *key*)

Returns pointer at key *key* in *t*.

8.13. Religious stances

I chose C because:

- GMP, which PBC requires and is also modeled on, is also written in C.
- PBC is intended to be a low-level portable cryptographic library. C is the least common denominator. It should not be difficult to wrap PBC for other languages.
- Despite its drawbacks (I would appreciate operator overloading and genericity, and to a lesser extent garbage collection), I've found few languages I like better. To quote Rob Pike, C is the desert island language. (I also agree with his statement that OO languages conceptually provide little extra over judicious use of function pointers in C.)

With respect to indentation, I'm migrating the code to follow Google C++ Style Guide (http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml) to avoid having to switch styles all the time. The code was originally written using my old style: 4-space indent with 1TBS (One True Brace Style).

I'd like to have no library dependencies (except standard C libraries), but then I'd have to write a large integer library. Furthermore, I'd have to write it in assembly, and then port it.

To avoid this, I use an existing library. I selected GMP because the library's focus is on multiprecision arithmetic and nothing else, and it aims to be as fast as possible on many platforms. Another important factor is that GMP is released under a free license.

On the other hand, GMP is written to deal with extremely large numbers, while I mostly only need integers that are roughly between 160 and 2048 bits. It is possible a library specializing in numbers of these sizes would be better for PBC.

I'm fond of GMP's method for eliminating the need for the `&` and `*` operators most of the time by declaring a typedef on arrays of size 1. I try to do the same with PBC for consistency, though this trick does have drawbacks.

I would like to have GMP as the only library dependency, though I do not mind using other libraries so long as they are optional. For example, one of the test programs is much easier to use if compiled with the GNU readline library, but by default compiles without it and is still functional.

I dislike the C preprocessor. I like to place platform-specific code in separate files and let the build system work out which one to use. Integer constants can be defined with enum instead. I intend to minimize the number of `#include` statements in header files for PBC's internal use as much as possible (they should be in the `.c` files instead), and later perhaps even remove those annoying `#ifndef` statements too. I grudgingly accept some macros for PBC's debugging features.

I liberally use nested functions, a GNU C extension. I find their expressiveness so indispensable that I'm willing to sacrifice portability for them.

The GNU libc manual (http://www.gnu.org/software/libc/manual/html_node/Reserved-Names.html) states that data types ending in `_t` should not be used because they are reserved for future additions to C or POSIX. On the other hand, I want to stay consistent with GMP, and ending data types with `_t` is common practice.

Chapter 9. Security issues

Potential problems for the paranoid.

Truncated hashes

For points on an elliptic curve over the base field, `element_from_hash()` will truncate the input hash until it can represent an x-coordinate in that field. (PBC then computes a corresponding y-coordinate.) Ideally the hash length should be smaller than size of the base field and also the size of the elliptic curve group.

Hashing to elements in field extensions does not take advantage of the fact that the extension has more elements than the base field. I intend to rewrite the code so that for a degree n extension code, PBC splits the hash into n parts and determine each polynomial coefficient from one ofthe pieces. At the moment every coefficient is the same and depends on the whole hash.

This is harmless for the base field, because all the pairing types implemented so far use an integer mod ring as the base field, rather than an extension of some low characteristic field.

Zeroed memory

Unlike OpenSSL, there are no functions to zero memory locations used in sensitive computations. To some extent, one can use `element_random()` to overwrite data.

PRNG determinism

On platforms without `/dev/urandom` PBC falls back on a deterministic pseudo-random number generator, except on Windows where it attempts to use the Microsoft Crypto API.

Also, `/dev/urandom` differs from `/dev/random`. A quote from its manpage:

> A read from the /dev/urandom device will not block waiting for more entropy. As a result, if there is not sufficient entropy in the entropy pool, the returned values are theoretically vulnerable to a cryptographic attack on the algorithms used by the driver. Knowledge of how to do this is not available in the current non-classified literature, but it is theoretically possible that such an attack may exist. If this is a concern in your application, use /dev/random instead.

Appendix A. Contributors

Ben Lynn wrote the original PBC library and documentation and is still maintaining and developing it.

Hovav Shacham wrote the multiexponentiation, sliding windows and preprocessed exponentiation routines, Makefile improvements, and other enhancements. He also helps administer the mailing list.

Joseph Cooley wrote the GNU build system files, tested the library on Mac OS X, and added miscellaneous improvements. Among other things, pairings can be read from memory buffer and most compile-time warnings were removed.

Rob Figueiredo and Roger Khazan wrote changes which allow the PBC library to be compiled on Windows (via mingw).

Dmitry Kosolapov sent in manual corrections, and wrote several cryptosystem demos.

John Bethencourt sent in many helpful patches, e.g. fixes that allow PBC to work on 64-bit platforms.

Paul Miller reported bugs, manual corrections and also wrote the Gentoo portage overlay for PBC.

If you're not mentioned here but should be, please let me know! (blynn at cs dot stanford dot edu).