

# Botan API Reference

2010/06/14

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Targets . . . . .	4
1.2	Why Botan? . . . . .	4
<b>2</b>	<b>Getting Started</b>	<b>6</b>
2.1	Basic Conventions . . . . .	6
2.2	Initializing the Library . . . . .	6
2.3	Pitfalls . . . . .	6
2.4	Information Flow: Pipes and Filters . . . . .	7
2.5	Fork . . . . .	9
2.5.1	Chain . . . . .	10
2.6	The Pipe API . . . . .	10
2.6.1	Initializing Pipe . . . . .	10
2.6.2	Giving Data to a Pipe . . . . .	11
2.6.3	Getting Output from a Pipe . . . . .	11
2.7	A Filter Example . . . . .	12
2.8	Filter Catalog . . . . .	13
2.8.1	Keyed Filters . . . . .	13
2.8.2	Cipher Filters . . . . .	14
2.8.3	Hashes and MACs . . . . .	14
2.8.4	PK Filters . . . . .	15
2.8.5	Encoders . . . . .	15
2.9	Rolling Your Own . . . . .	16
<b>3</b>	<b>Public Key Cryptography</b>	<b>17</b>
3.1	Creating PK Algorithm Key Objects . . . . .	17
3.1.1	Creating a DL_Group . . . . .	17
3.2	Key Checking . . . . .	18
3.3	Getting a PK algorithm object . . . . .	18
3.4	Encryption . . . . .	19
3.5	Signatures . . . . .	19
3.6	Key Agreement . . . . .	20
3.7	Importing and Exporting PK Keys . . . . .	20
3.7.1	Public Keys . . . . .	21
3.7.2	Private Keys . . . . .	21
<b>4</b>	<b>Certificate Handling</b>	<b>24</b>
4.1	So what's in an X.509 certificate? . . . . .	24
4.1.1	X.509v3 Extensions . . . . .	25
4.1.2	Revocation Lists . . . . .	25
4.2	Reading Certificates . . . . .	25

4.3	Storing and Using Certificates . . . . .	26
4.3.1	Adding Certificates . . . . .	26
4.3.2	Adding CRLs . . . . .	26
4.3.3	Storing Certificates . . . . .	26
4.3.4	Searching for Certificates . . . . .	26
4.3.5	Certificate Stores . . . . .	27
4.3.6	Verifying Certificates . . . . .	27
4.4	Certificate Authorities . . . . .	29
4.4.1	Generating CRLs . . . . .	29
4.4.2	Self-Signed Certificates . . . . .	29
4.4.3	Creating PKCS #10 Requests . . . . .	30
4.4.4	Certificate Options . . . . .	30
<b>5</b>	<b>The Low-Level Interface</b>	<b>32</b>
5.1	Basic Algorithm Abilities . . . . .	32
5.2	Keys and IVs . . . . .	32
5.3	Symmetrically Keyed Algorithms . . . . .	32
5.4	Block Ciphers . . . . .	33
5.5	Stream Ciphers . . . . .	33
5.6	Hash Functions / Message Authentication Codes . . . . .	34
<b>6</b>	<b>Random Number Generators</b>	<b>35</b>
6.1	Randpool . . . . .	35
6.2	ANSI X9.31 . . . . .	36
6.3	Entropy Sources . . . . .	36
<b>7</b>	<b>User Interfaces</b>	<b>37</b>
<b>8</b>	<b>Botan's Modules</b>	<b>38</b>
8.1	Pipe I/O for Unix File Descriptors . . . . .	38
8.2	Entropy Sources . . . . .	38
8.3	Compressors . . . . .	39
8.3.1	Bzip2 . . . . .	39
8.3.2	Zlib . . . . .	39
8.3.3	Data Sources . . . . .	39
8.3.4	Data Sinks . . . . .	40
8.4	Writing Modules . . . . .	40
<b>9</b>	<b>Miscellaneous</b>	<b>42</b>
9.1	S2K Algorithms . . . . .	42
9.1.1	OpenPGP S2K . . . . .	42
9.2	Checksums . . . . .	42
9.3	Exceptions . . . . .	43
9.4	Threads and Mutexes . . . . .	43
9.5	Secure Memory . . . . .	43
9.6	Allocators . . . . .	44
9.7	BigInt . . . . .	44
9.7.1	Efficiency Hints . . . . .	45
<b>10</b>	<b>Algorithms</b>	<b>46</b>
10.1	Recommended Algorithms . . . . .	46
10.2	Compliance with Standards . . . . .	46
10.3	Algorithms Listing . . . . .	46
10.4	Compatibility . . . . .	47

<b>11 Support and Further Information</b>	<b>48</b>
11.1 Patents . . . . .	48
11.2 Recommended Reading . . . . .	48
11.3 Support . . . . .	48
11.4 Contact Information . . . . .	48
11.5 License . . . . .	48

# 1 Introduction

Botan is a C++ library that attempts to provide the most common cryptographic algorithms and operations in an easy to use, efficient, and portable way. It runs on a wide variety of systems, and can be used with a number of different compilers.

The base library is written in ISO C++, so it can be ported with minimal fuss, but Botan also supports a modules system. This system exposes system dependent code to the library through portable interfaces, extending the set of services available to users.

## 1.1 Targets

Botan's primary targets (system-wise) are 32 and 64-bit CPUs, with a flat memory address space of at least 32 bits. Generally, given the choice between optimizing for 32-bit systems and 64-bit systems, Botan is written to prefer 64-bit, simply on the theory that where performance is a real concern, modern 64-bit processors are the obvious choice. However in most cases this is not an issue, as many algorithms are specified in terms of 32-bit operations precisely to target commodity processors.

Smaller handhelds, set-top boxes, and the bigger smart phones and smart cards, are also capable of using Botan. However, Botan uses a fairly large amount of code space (up to several megabytes, depending upon the compiler and options used), which could be prohibitive in some systems. Usage of RAM is fairly modest, usually under 64K.

Botan's design makes it quite easy to remove unused algorithms in such a way that applications do not need to be recompiled to work, even applications that use the algorithms in question. They can simply ask Botan if the algorithm exists, and if Botan says yes, ask the library to give them such an object for that algorithm.

## 1.2 Why Botan?

Botan may be the perfect choice for your application. Or it might be a terribly bad idea. This section will make clear what Botan is and is not.

First, let's cover the major strengths:

- Support is (usually) quickly available on the project mailing lists. Commercial support licenses are available for those that desire them.
- Is written in a (fairly) clean object-oriented style, and the usual API works in terms of reasonably high-level abstractions.
- Supports a huge variety of algorithms, including most of the major public key algorithms and standards (such as IEEE 1363, PKCS, and X.509v3).
- Supports a name-based lookup scheme, so you can get a hold of any algorithm on the fly.
- You can easily extend much of the system at application compile time or at run time.
- Works well with a wide variety of compilers, operating systems, and CPUs, and more all the time.
- Is the only open source crypto library (that I know of) that has support for memory allocation techniques that prevent an attacker from reading swap in an attempt to gain access to keys or other secrets. In fact several different such methods are supported, depending on the system (two methods for Unix, another for Windows).
- Has (optional) support for Zlib and Bzip2 compression/decompression integrated completely into the system – it only takes a line or two of code to add compression to your application.

And the major downsides and deficiencies are:

- It's written in C++. If your application isn't, Botan is probably going to be more pain than it's worth.
- Botan doesn't directly support higher-level protocols and formats like SSL or OpenPGP. SSH support is available from a third-party, and there is an alpha-level SSL/TLS library currently available.
- Doesn't currently support any very high level 'envelope' style processing - support for this will probably be added once support for CMS is available, so code using the high level interface will produce data readable by many other libraries.

## 2 Getting Started

### 2.1 Basic Conventions

With a very small number of exceptions, declarations in the library are contained within the namespace `Botan`. Botan declares several typedef'ed types to help buffer it against changes in machine architecture. These types are used extensively in the interface, thus it would be often be convenient to use them without the `Botan` prefix. You can do so by using the namespace `Botan::types` (this way you can use the type names without the namespace prefix, but the remainder of the library stays out of the global namespace). The included types are `byte` and `u32bit`, which are unsigned integer types.

The headers for Botan are usually available in the form `botan/headername.h`. For brevity in this documentation, headers are always just called `headername.h`, but they should be used with the `botan/` prefix in your actual code.

### 2.2 Initializing the Library

There is a set of core services that the library needs access to while it is performing requests. To ensure these are set up, you must create a `LibraryInitializer` object (usually called 'init' in Botan example code; 'botan.library' or 'botan\_init' may make more sense in real applications) prior to making any calls to Botan. This object's lifetime must exceed that of all other Botan objects your application creates; for this reason the best place to create the `LibraryInitializer` is at the start of your `main` function, since this guarantees that it will be created first and destroyed last (via standard C++ RAII rules). The initializer does things like setting up the memory allocation system and algorithm lookup tables, finding out if there is a high resolution timer available to use, and similar such matters. With no arguments, the library is initialized with various default settings. So most of the time (unless you are writing threaded code; see below), all you need is:

```
Botan::LibraryInitializer init;
```

at the start of your `main`.

The constructor takes an optional string that specifies arguments. Currently the only possible argument is "thread\_safe", which must have a Boolean argument (for instance "thread\_safe=false" or "thread\_safe=true"). If "thread\_safe" is specified as true the library will attempt to register a mutex type to properly guard access to shared resources. However these locks do not protect individual Botan objects: explicit locking must be used in this case.

If you do not create a `LibraryInitializer` object, pretty much any Botan operation will fail, because it will be unable to do basic things like allocate memory or get random bits. Note too, that you should be careful to only create one such object.

It is not strictly necessary to create a `LibraryInitializer`; the actual code performing the initialization and shutdown are in static member functions of `LibraryInitializer`, called `initialize` and `deinitialize`. A `LibraryInitializer` merely provides a convenient RAII wrapper for the operations (thus for the internal library state as well).

### 2.3 Pitfalls

There are a few things to watch out for to prevent problems when using Botan.

Never allocate any kind of Botan object globally. The problem with doing this is that the constructor for such an object will be called before the library is initialized. Many Botan objects will, in their constructor, make one or more calls into the library global state object. Access to this object is checked, so an exception should be thrown (rather than a memory access violation or undetected uninitialized object access). A rough equivalent that will work is to keep a global pointer to the object, initializing it after creating your

**LibraryInitializer**. Merely making the **LibraryInitializer** also global will probably not help, because C++ does not make very strong guarantees about the order that such objects will be created.

The same rule applies for making sure the destructors of all your Botan objects are called before the **LibraryInitializer** is destroyed. This implies you can't have static variables that are Botan objects inside functions or classes (since in most C++ runtimes, these objects will be destroyed after `main` has returned). This is inelegant, but seems to not cause many problems in practice.

Botan's memory object classes (**MemoryVector**, **SecureVector**, **SecureBuffer**) are extremely primitive, and do not (currently) meet the requirements for an STL container object. After Botan starts adopting C++0x features, they will be replaced by typedefs of `std::vector` with a custom allocator.

Use a **try/catch** block inside your **main** function, and catch any `std::exception` throws (remember to catch by reference, as `std::exception`'s **what** method is polymorphic). This is not strictly required, but if you don't, and Botan throws an exception, the runtime will call `std::terminate`, which usually calls **abort** or something like it, leaving you (or worse, a user of your application) wondering what went wrong.

## 2.4 Information Flow: Pipes and Filters

Many common uses of cryptography involve processing one or more streams of data (be it from sockets, files, or a hardware device). Botan provides services that make setting up data flows through various operations, such as compression, encryption, and base64 encoding. Each of these operations is implemented in what are called *filters* in Botan. A set of filters are created and placed into a *pipe*, and information "flows" through the pipe until it reaches the end, where the output is collected for retrieval. If you're familiar with the Unix shell environment, this design will sound quite familiar.

Here is an example that uses a pipe to base64 encode some strings:

```
Pipe pipe(new Base64_Encoder); // pipe owns the pointer
pipe.start_msg();
pipe.write('message 1');
pipe.end_msg(); // flushes buffers, increments message number

// process_msg(x) is start_msg() && write(x) && end_msg()
pipe.process_msg('message2');

std::string m1 = pipe.read_all_as_string(0); // 'message1'
std::string m2 = pipe.read_all_as_string(1); // 'message2'
```

Bytestreams in the pipe are grouped into messages; blocks of data that are processed in an identical fashion (*i.e.*, with the same sequence of **Filters**). Messages are delimited by calls to **start\_msg** and **end\_msg**. Each message in a pipe has its own identifier, which currently is an integer that increments up from zero.

As you can see, the **Base64\_Encoder** was allocated using **new**; but where was it deallocated? When a filter object is passed to a **Pipe**, the pipe takes ownership of the object, and will deallocate it when it is no longer needed.

There are two different ways to make use of messages. One is to send several messages through a **Pipe** without changing the **Pipe**'s configuration, so you end up with a sequence of messages; one use of this would be to send a sequence of identically encrypted UDP packets, for example (note that the *data* need not be identical; it is just that each is encrypted, encoded, signed, etc in an identical fashion). Another is to change the filters that are used in the **Pipe** between each message, by adding or removing **Filters**; functions that let you do this are documented in the Pipe API section.

Most operations in Botan have a corresponding filter for use in Pipe. Here's code that encrypts a string with AES-128 in CBC mode:

```

AutoSeeded_RNG rng,
SymmetricKey key(rng, 16); // a random 128-bit key
InitializationVector iv(rng, 16); // a random 128-bit IV

// Notice the algorithm we want is specified by a string
Pipe pipe(get_cipher('AES-128/CBC', key, iv, ENCRYPTION));

pipe.process_msg('secrets');
pipe.process_msg('more secrets');

MemoryVector<byte> c1 = pipe.read_all(0);

byte c2[4096] = { 0 };
u32bit got_out = pipe.read(c2, sizeof(c2), 1);
// use c2[0...got_out]

```

Note the use of `AutoSeeded_RNG`, which is a random number generator. If you want to, you can explicitly set up the random number generators and entropy sources you want to, however for 99% of cases `AutoSeeded_RNG` is preferable.

Pipe also has convenience methods for dealing with `std::iostreams`. Here is an example of those, using the `Bzip_Compression` filter (included as a module; if you have `bzlib` available, check [building.pdf](#) for how to enable it) to compress a file:

```

std::ifstream in('data.bin', std::ios::binary)
std::ofstream out('data.bin.bz2', std::ios::binary)

Pipe pipe(new Bzip_Compression);

pipe.start_msg();
in >> pipe;
pipe.end_msg();
out << pipe;

```

However there is a hitch to the code above; the complete contents of the compressed data will be held in memory until the entire message has been compressed, at which time the statement `out << pipe` is executed, and the data is freed as it is read from the pipe and written to the file. But if the file is very large, we might not have enough physical memory (or even enough virtual memory!) for that to be practical. So instead of storing the compressed data in the pipe for reading it out later, we divert it directly to the file:

```

std::ifstream in('data.bin', std::ios::binary)
std::ofstream out('data.bin.bz2', std::ios::binary)

Pipe pipe(new Bzip_Compression, new DataSink_Stream(out));

pipe.start_msg();
in >> pipe;
pipe.end_msg();

```

This is the first code we've seen so far that uses more than one filter in a pipe. The output of the compressor is sent to the `DataSink_Stream`. Anything written to a `DataSink_Stream` is written to a file; the filter produces no output. As soon as the compression algorithm finishes up a block of data, it will send it along, at which point it will immediately be written to disk; if you were to call `pipe.read_all()` after `pipe.end_msg()`, you'd get an empty vector out.



Here's an example using two computational filters:

```
AutoSeeded_RNG rng,
SymmetricKey key(rng, 32);
InitializationVector iv(rng, 16);

Pipe encryptor(get_cipher("AES/CBC/PKCS7", key, iv, ENCRYPTION),
               new Base64_Encoder);

encryptor.start_msg();
file >> encryptor;
encryptor.end_msg(); // flush buffers, complete computations
std::cout << encryptor;
```

## 2.5 Fork

It is fairly common that you might receive some data and want to perform more than one operation on it (*i.e.*, encrypt it with Serpent and calculate the SHA-256 hash of the plaintext at the same time). That's where **Fork** comes in. **Fork** is a filter that takes input and passes it on to *one or more* **Filters** that are attached to it. **Fork** changes the nature of the pipe system completely. Instead of being a linked list, it becomes a tree.

Each **Filter** in the fork is given its own output buffer, and thus its own message. For example, if you had previously written two messages into a **Pipe**, then you start a new one with a **Fork** that has three paths of **Filter**'s inside it, you add three new messages to the **Pipe**. The data you put into the **Pipe** is duplicated and sent into each set of **Filters**, and the eventual output is placed into a dedicated message slot in the **Pipe**.

Messages in the **Pipe** are allocated in a depth-first manner. This is only interesting if you are using more than one **Fork** in a single **Pipe**. As an example, consider the following:

```
Pipe pipe(new Fork(
    new Fork(
        new Base64_Encoder,
        new Fork(
            NULL,
            new Base64_Encoder
        )
    ),
    new Hex_Encoder
);
```

In this case, message 0 will be the output of the first **Base64\_Encoder**, message 1 will be a copy of the input (see below for how **Fork** interprets **NULL** pointers), message 2 will be the output of the second **Base64\_Encoder**, and message 3 will be the output of the **Hex\_Encoder**. As you can see, this results in message numbers being allocated in a top to bottom fashion, when looked at on the screen. However, note that there could be potential for bugs if this is not anticipated. For example, if your code is passed a **Filter**, and you assume it is a "normal" one that only uses one message, your message offsets would be wrong, leading to some confusion during output.

If **Fork**'s first argument is a null pointer, but a later argument is not, then **Fork** will feed a copy of its input directly through. Here's a case where that is useful:

```
// have std::string ciphertext, auth_code, key, iv, mac_key;
```

```

Pipe pipe(new Base64_Decoder,
          get_cipher('AES-128', key, iv, DECRYPTION),
          new Fork(
              0
              new MAC_Filter('HMAC(SHA-1)', mac_key)
          )
);

pipe.process_msg(ciphertext);
std::string plaintext = pipe.read_all_as_string(0);
SecureVector<byte> mac = pipe.read_all(1);

if(mac != auth_code)
    error();

```

Here we wanted to not only decrypt the message, but send the decrypted text through an additional computation, in order to compute the authentication code.

Any **Filters** that are attached to the **Pipe** after the **Fork** are implicitly attached onto the first branch created by the fork. For example, let's say you created this **Pipe**:

```

Pipe pipe(new Fork(new Hash_Filter("MD5"), new Hash_Filter("SHA-1")),
          new Hex_Encoder);

```

And then called **start\_msg**, inserted some data, then **end\_msg**. Then *pipe* would contain two messages. The first one (message number 0) would contain the MD5 sum of the input in hex encoded form, and the other would contain the SHA-1 sum of the input in raw binary. However, it's much better to use a **Chain** instead.

### 2.5.1 Chain

A **Chain** filter creates a chain of **Filters** and encapsulates them inside a single filter (itself). This allows a sequence of filters to become a single filter, to be passed into or out of a function, or to a **Fork** constructor.

You can call **Chain**'s constructor with up to 4 **Filter\*s** (they will be added in order), or with an array of **Filter\*s** and a **u32bit** that tells **Chain** how many **Filter\*s** are in the array (again, they will be attached in order). Here's the example from the last section, using chain instead of relying on the obscure rule that version used.

```

Pipe pipe(new Fork(
    new Chain(new Hash_Filter("MD5"), new Hex_Encoder),
    new Hash_Filter("SHA-1")
));

```

## 2.6 The Pipe API

### 2.6.1 Initializing Pipe

By default, **Pipe** will do nothing at all; any input placed into the **Pipe** will be read back unchanged. Obviously, this has limited utility, and presumably you want to use one or more **Filters** to somehow process the data. First, you can choose a set of **Filters** to initialize the **Pipe** via the constructor. You can pass it either a set of up to 4 **Filter\*s**, or a pre-defined array and a length:

```

Pipe pipe1(new Filter1(*args/), new Filter2(*args/),
           new Filter3(*args/), new Filter4(*args/));
Pipe pipe2(new Filter1(*args/), new Filter2(*args/));

Filter* filters[5] = {
    new Filter1(*args/), new Filter2(*args/), new Filter3(*args/),
    new Filter4(*args/), new Filter5(*args/) /* more if desired... */
};
Pipe pipe3(filters, 5);

```

This is by far the most common way to initialize a **Pipe**. However, occasionally a more flexible initialization strategy is necessary; this is supported by 4 member functions: **prepend(Filter\*)**, **append(Filter\*)**, **pop()**, and **reset()**. These functions may only be used while the **Pipe** in question is not in use; that is, either before calling **start\_msg**, or after **end\_msg** has been called (and no new calls to **start\_msg** have been made yet).

The function **reset()** simply removes all the **Filters** that the **Pipe** is currently using – it is reset to an initialize, “empty” state. Any data that is being retained by the **Pipe** is retained after a **reset()**, and **reset()** does not affect the message numbers (discussed later).

Calling **prepend** and **append** will either prepend or append the passed **Filter** object to the list of transformations. For example, if you **prepend** a **Filter** implementing encryption, and the **Pipe** already had a **Filter** that hex encoded the input, then the next set of input would be first encrypted, then hex encoded. Alternately, if you called **append**, then the input would be first be hex encoded, and then encrypted (which is not terribly useful in this particular example).

Finally, calling **pop()** will remove the first transformation of the **Pipe**. Say we had called **prepend** to put an encryption **Filter** into a **Pipe**; calling **pop()** would remove this **Filter** and return the **Pipe** to its state before we called **prepend**.

### 2.6.2 Giving Data to a Pipe

Input to a **Pipe** is delimited into messages, which can be read from independently (*i.e.*, you can read 5 bytes from one message, and then all of another message, without either read affecting any other messages). The messages are delimited by calls to **start\_msg** and **end\_msg**. In between these two calls, you can write data into a **Pipe**, and it will be processed by the **Filter(s)** that it contains. Writes at any other time are invalid, and will result in an exception.

As to writing, you can call any of the functions called **write()**, that can take any of: a `byte[]/u32bit` pair, a `SecureVector<byte>`, a `std::string`, a `DataSource&`, or a single `byte`.

Sometimes, you may want to do only a single write per message. In this case, you can use the **process\_msg** series of functions, which start a message, write their argument into the **Pipe**, and then end the message. In this case you would not make any explicit calls to **start\_msg/end\_msg**. The version of **write** that takes a single `byte` is not supported by **process\_msg**, but all the other variants are.

**Pipe** can also be used with the `>>` operator, and will accept a `std::istream`, (or on Unix systems with the `fd_unix` module), a Unix file descriptor. In either case, the entire contents of the file will be read into the **Pipe**.

### 2.6.3 Getting Output from a Pipe

Retrieving the processed data from a **Pipe** is a bit more complicated, for various reasons. In particular, because **Pipe** will separate each message into a separate buffer, you have to be able to retrieve data from each message independently. Each of **Pipe**’s read functions has a final parameter that specifies what message to read from (as a 32-bit integer). If this parameter is set to `Pipe::DEFAULT_MESSAGE`, it will read the current

default message (`DEFAULT_MESSAGE` is also the default value of this parameter). The parameter will not be mentioned in further discussion of the reading API, but it is always there (unless otherwise noted).

Reading is done with a variety of functions. The most basic are `u32bit read(byte out[], u32bit len)` and `u32bit read(byte& out)`. Each reads into *out* (either up to *len* bytes, or a single byte for the one taking a `byte&`), and returns the total number of bytes read. There is a variant of these functions, all named **peek**, which performs the same operations, but does not remove the bytes from the message (reading is a destructive operation with a `Pipe`).

There are also the functions `SecureVector<byte> read_all()`, and `std::string read_all_as_string()`, which return the entire contents of the message, either as a memory buffer, or a `std::string` (which is generally only useful if the `Pipe` has encoded the message into a text string, such as when a `Base64_Encoder` is used).

To determine how many bytes are left in a message, call `u32bit remaining()` (which can also take an optional message number). Finally, there are some functions for managing the default message number: `u32bit default_msg()` will return the current default message, `u32bit message_count()` will return the total number of messages (`0...message_count()-1`), and `set_default_msg(u32bit msgno)` will set a new default message number (which must be a valid message number for that `Pipe`). The ability to set the default message number is particularly important in the case of using the file output operations (`<<` with a `std::ostream` or Unix file descriptor), because there is no way to specify it explicitly when using the output operator.

## 2.7 A Filter Example

Here is some code that takes one or more filenames in *argv* and calculates the result of several hash functions for each file. The complete program can be found as `hasher.cpp` in the Botan distribution. For brevity, most error checking has been removed.

```
string name[3] = { "MD5", "SHA-1", "RIPEMD-160" };
Botan::Filter* hash[3] = {
    new Botan::Chain(new Botan::Hash_Filter(name[0]),
                    new Botan::Hex_Encoder),
    new Botan::Chain(new Botan::Hash_Filter(name[1]),
                    new Botan::Hex_Encoder),
    new Botan::Chain(new Botan::Hash_Filter(name[2]),
                    new Botan::Hex_Encoder) };

Botan::Pipe pipe(new Botan::Fork(hash, COUNT));

for(u32bit j = 1; argv[j] != 0; j++)
{
    ifstream file(argv[j]);
    pipe.start_msg();
    file >> pipe;
    pipe.end_msg();
    file.close();
    for(u32bit k = 0; k != 3; k++)
    {
        pipe.set_default_msg(3*(j-1)+k);
        cout << name[k] << "(" << argv[j] << ") = " << pipe << endl;
    }
}
```

## 2.8 Filter Catalog

This section contains descriptions of every **Filter** included in the portable sections of Botan. **Filters** provided by modules are documented elsewhere.

### 2.8.1 Keyed Filters

A few sections ago, it was mentioned that **Pipe** can process multiple messages, treating each of them exactly the same. Well, that was a bit of a lie. There are some algorithms (in particular, block ciphers not in ECB mode, and all stream ciphers) that change their state as data is put through them.

Naturally, you might well want to reset the keys or (in the case of block cipher modes) IVs used by such filters, so multiple messages can be processed using completely different keys, or new IVs, or new keys and IVs, or whatever. And in fact, even for a MAC or an ECB block cipher, you might well want to change the key used from message to message.

Enter **Keyed\_Filter**, which acts as an abstract interface for any filter that uses keys: block cipher modes, stream ciphers, MACs, and so on. It has two functions, **set\_key** and **set\_iv**. Calling **set\_key** will, naturally, set (or reset) the key used by the algorithm. Setting the IV only makes sense in certain algorithms – a call to **set\_iv** on an object that doesn't support IVs will be ignored. You *must* call **set\_key** before calling **set\_iv**: while not all **Keyed\_Filter** objects require this, you should assume it is required anytime you are using a **Keyed\_Filter**.

Here's an example:

```
Keyed_Filter *cast, *hmac;
Pipe pipe(new Base64_Decoder,
    // Note the assignments to the cast and hmac variables
    cast = new CBC_Decryption("CAST-128", "PKCS7", cast_key, iv),
    new Fork(
        0, // Read the section 'Fork' to understand this
        new Chain(
            hmac = new MAC_Filter("HMAC(SHA-1)", mac_key, 12),
            new Base64_Encoder
        )
    )
);
pipe.start_msg();
[use pipe for a while, decrypt some stuff, derive new keys and IVs]
pipe.end_msg();

cast->set_key(cast_key2);
cast->set_iv(iv2);
hmac->set_key(mac_key2);

pipe.start_msg();
[use pipe for some other things]
pipe.end_msg();
```

There are some requirements to using **Keyed\_Filter** that you must follow. If you call **set\_key** or **set\_iv** on a filter that is owned by a **Pipe**, you must do so while the **Pipe** is “unlocked”. This refers to the times when no messages are being processed by **Pipe** – either before **Pipe**'s **start\_msg** is called, or after **end\_msg** is called (and no new call to **start\_msg** has happened yet). Doing otherwise will result in undefined behavior, probably silently getting invalid output.

And remember: if you're resetting both values, reset the key *first*.

## 2.8.2 Cipher Filters

Getting a hold of a **Filter** implementing a cipher is very easy. Simply make sure you’re including the header `lookup.h`, and call **get\_cipher**. Generally you will pass the return value directly into a **Pipe**. There are actually a couple different functions, which do pretty much the same thing:

```
get_cipher(std::string cipher_spec, SymmetricKey key, InitializationVector iv, Cipher_Dir dir);  
get_cipher(std::string cipher_spec, SymmetricKey key, Cipher_Dir dir);
```

The version that doesn’t take an IV is useful for things that don’t use them, like block ciphers in ECB mode, or most stream ciphers. If you specify a *cipher\_spec* that does want a IV, and you use the version that doesn’t take one, an exception will be thrown. The *dir* argument can be either **ENCRYPTION** or **DECRYPTION**. In a few cases, like most (but not all) stream ciphers, these are equivalent, but even then it provides a way of showing the “intent” of the operation to readers of your code.

The *cipher\_spec* is a string that specifies what cipher is to be used. The general syntax for *cipher\_spec* is “STREAM\_CIPHER”, “BLOCK\_CIPHER/MODE”, or “BLOCK\_CIPHER/MODE/PADDING”. In the case of stream ciphers, no mode is necessary, so just the name is sufficient. A block cipher requires a mode of some sort, which can be “ECB”, “CBC”, “CFB(n)”, “OFB”, “CTR-BE”, or “EAX(n)”. The argument to CFB mode is how many bits of feedback should be used. If you just use “CFB” with no argument, it will default to using a feedback equal to the block size of the cipher. EAX mode also takes an optional bit argument, which tells EAX how large a tag size to use – generally this is the size of the block size of the cipher, which is the default if you don’t specify any argument.

In the case of the ECB and CBC modes, a padding method can also be specified. If it is not supplied, ECB defaults to not padding, and CBC defaults to using PKCS #5/#7 compatible padding. The padding methods currently available are “NoPadding”, “PKCS7”, “OneAndZeros”, and “CTS”. CTS padding is currently only available for CBC mode, but the others can also be used in ECB mode.

Some example *cipher\_spec* arguments are: “DES/CFB(32)”, “TripleDES/OFB”, “Blowfish/CBC/CTS”, “SAFER-SK(10)/CBC/OneAndZeros”, “AES/EAX”, “ARC4”

“CTR-BE” refers to counter mode where the counter is incremented as if it were a big-endian encoded integer. This is compatible with most other implementations, but it is possible some will use the incompatible little endian convention. This version would be denoted as “CTR-LE” if it were supported.

“EAX” is a new cipher mode designed by Wagner, Rogaway, and Bellare. It is an authenticated cipher mode (that is, no separate authentication is needed), has provable security, and is free from patent entanglements. It runs about half as fast as most of the other cipher modes (like CBC, OFB, or CTR), which is not bad considering you don’t need to use an authentication code.

## 2.8.3 Hashes and MACs

Hash functions and MACs don’t need anything special when it comes to filters. Both just take their input and produce no output until **end\_msg()** is called, at which time they complete the hash or MAC and send that as output.

These **Filters** take a string naming the type to be used. If for some reason you name something that doesn’t exist, an exception will be thrown.

```
Hash_Filter(std::string hash, u32bit outlength):
```

This type hashes its input with *hash*. When **end\_msg** is called on the owning **Pipe**, the hash is completed and the digest is sent on to the next thing in the pipe. The argument *outlength* specifies how much of the output of the hash will be passed along to the next filter when **end\_msg** is called. By default, it will pass the entire hash.

Examples of names for **Hash\_Filter** are “SHA-1” and “Whirlpool”.

```
MAC_Filter(std::string mac, const SymmetricKey& key, u32bit outlength):
```

The constructor for a `MAC_Filter` takes a key, used in calculating the MAC, and a length parameter, which has semantics exactly the same as the one passed to `Hash_Filters` constructor.

Examples for *mac* are “HMAC(SHA-1)”, “CMAC(AES-128)”, and the exceptionally long, strange, and probably useless name “CMAC(Lion(Tiger(20,3),MARK-4,1024))”.

## 2.8.4 PK Filters

There are four classes in this category, `PK_Encoder_Filter`, `PK_Decryptor_Filter`, `PK_Signer_Filter`, and `PK_Verifier_Filter`. Each takes a pointer to an object of the appropriate type (`PK_Encoder`, `PK_Decryptor`, etc) that is deleted by the destructor. These classes are found in `pk_felts.h`.

Three of these, for encryption, decryption, and signing are pretty much identical conceptually. Each of them buffers its input until the end of the message is marked with a call to the `end_msg` function. Then they encrypt, decrypt, or sign their input and send the output (the ciphertext, the plaintext, or the signature) into the next filter.

Signature verification works a little differently, because it needs to know what the signature is in order to check it. You can either pass this in along with the constructor, or call the function `set_signature` – with this second method, you need to keep a pointer to the filter around so you can send it this command. In either case, after `end_msg` is called, it will try to verify the signature (if the signature has not been set by either method, an exception will be thrown here). It will then send a single byte onto the next filter – a 1 or a 0, which specifies whether the signature verified or not (respectively).

For more information about PK algorithms (including creating the appropriate objects to pass to the constructors), read the section “Public Key Cryptography” in this manual.

## 2.8.5 Encoders

Often you want your data to be in some form of text (for sending over channels that aren’t 8-bit clean, printing it, etc). The filters `Hex_Encoder` and `Base64_Encoder` will convert arbitrary binary data into hex or base64 formats. Not surprisingly, you can use `Hex_Decoder` and `Base64_Decoder` to convert it back into its original form.

Both of the encoders can take a few options about how the data should be formatted (all of which have defaults). The first is a `bool` which simply says if the encoder should insert line breaks. This defaults to false. Line breaks don’t matter either way to the decoder, but it makes the output a bit more appealing to the human eye, and a few transport mechanisms (notably some email systems) limit the maximum line length.

The second encoder option is an integer specifying how long such lines will be (obviously this will be ignored if line-breaking isn’t being used). The default tends to be in the range of 60-80 characters, but is not specified exactly. If you want a specific value, set it. Otherwise the default should be fine.

Lastly, `Hex_Encoder` takes an argument of type `Case`, which can be `Uppercase` or `Lowercase` (default is `Uppercase`). This specifies what case the characters A-F should be output as. The base64 encoder has no such option, because it uses both upper and lower case letters for its output.

The decoders both take a single option, which tells it how the object should behave in the case of invalid input. The enum (called `Decoder_Checking`) can take on any of three values: `NONE`, `IGNORE_WS`, and `FULL_CHECK`. With `NONE` (the default, for compatibility with previous releases), invalid input (for example, a “z” character in supposedly hex input) will simply be ignored. With `IGNORE_WS`, whitespace will be ignored by the decoder, but receiving other non-valid data will raise an exception. Finally, `FULL_CHECK` will raise an exception for *any* characters not in the encoded character set, including whitespace.

You can find the declarations for these types in `hex.h` and `base64.h`.

## 2.9 Rolling Your Own

The system of filters and pipes was designed in an attempt to make it as simple as possible to write new **Filter** objects. There are essentially four functions that need to be implemented by an object deriving from **Filter**:

**void write**(byte *input*[], u32bit *length*):

The **write** function is what is called when a filter receives input for it to process. The filter is *not* required to process it right away; many filters buffer their input before producing any output. A filter will usually have **write** called many times during its lifetime.

**void send**(byte *output*[], u32bit *length*):

Eventually, a filter will want to produce some output to send along to the next filter in the pipeline. It does so by calling **send** with whatever it wants to send along to the next filter. There is also a version of **send** taking a single byte argument, as a convenience.

**void start\_msg**():

This function is optional. Implement it if your **Filter** would like to do some processing or setup at the start of each message (for an example, see the Zlib compression module).

**void end\_msg**():

Implementing the **end\_msg** function is optional. It is called when it has been requested that filters finish up their computations. Note that they must *not* deallocate their resources; this should be done by their destructor. They should simply finish up with whatever computation they have been working on (for example, a compressing filter would flush the compressor and **send** the final block), and empty any buffers in preparation for processing a fresh new set of input. It is essentially the inverse of **start\_msg**.

Additionally, if necessary, filters can define a constructor that takes any needed arguments, and a destructor to deal with deallocating memory, closing files, etc.

There is also a **BufferingFilter** class (in **buf\_filt.h**) that will take a message and split it up into an initial block that can be of any size (including zero), a sequence of fixed sized blocks of any non-zero size, and last (possibly zero-sized) final block. This might make a useful base class for your filters, depending on what you have in mind.



## 3 Public Key Cryptography

Let's create a 1024-bit RSA private key, encode the public key as a PKCS #1 file with PEM encoding (which can be understood by many other cryptographic programs)

```
// everyone does:
AutoSeeded_RNG rng;

// Alice
RSA_PrivateKey priv_rsa(rng, 1024 /* bits */);

std::string alice_pem = X509::PEM_encode(priv_rsa);

// send alice_pem to Bob, who does

// Bob
std::auto_ptr<Public_Key> alice(load_key(alice_pem));

RSA_PublicKey* alice_rsa = dynamic_cast<RSA_PublicKey>(alice);
if(alice_rsa)
{
    /* ... */
}
```

### 3.1 Creating PK Algorithm Key Objects

The library has interfaces for encryption, signatures, etc that do not require knowing the exact algorithm in use (for example RSA and Rabin-Williams signatures are handled by the exact same code path).

One place where we *do* need to know exactly what kind of algorithm is in use is when we are creating a key (*But*: read the section “Importing and Exporting PK Keys”, later in this manual).

There are (currently) two kinds of public key algorithms in Botan: ones based on integer factorization (RSA and Rabin-Williams), and ones based on the discrete logarithm problem (DSA, Diffie-Hellman, Nyberg-Rueppel, and ElGamal). Since discrete logarithm parameters (primes and generators) can be shared among many keys, there is the notion of these being a combined type (called **DL\_Group**).

There are two ways to create a DL private key (such as **DSA\_PrivateKey**). One is to pass in just a **DL\_Group** object – a new key will automatically be generated. The other involves passing in a group to use, along with both the public and private values (private value first).

Since in integer factorization algorithms, the modulus used isn't shared by other keys, we don't use this notion. You can create a new key by passing in a **u32bit** telling how long (in bits) the key should be, or you can copy an pre-existing key by passing in the appropriate parameters (primes, exponents, etc). For RSA and Rabin-Williams (the two IF schemes in Botan), the parameters are all **BigInts**: prime 1, prime 2, encryption exponent, decryption exponent, modulus. The last two are optional, since they can easily be derived from the first three.

#### 3.1.1 Creating a DL\_Group

There are quite a few ways to get a **DL\_Group** object. The best is to use the function **get\_dl\_group**, which takes a string naming a group; it will either return that group, if it knows about it, or throw an exception. Names it knows about include “IETF-n” where n is 768, 1024, 1536, 2048, 3072, or 4096, and “DSA-n”, where n is 512, 768, or 1024. The IETF groups are the ones specified for use with IPSec, and the DSA ones

are the default DSA parameters specified by Java's JCE. For DSA and Nyberg-Rueppel, you should only use the "DSA-n" groups, while Diffie-Hellman and ElGamal can use either type (keep in mind that some applications/standards require DH/ELG to use DSA-style primes, while others require strong prime groups).

You can also generate a new random group. This is not recommend, because it is quite slow, especially for safe primes.

## 3.2 Key Checking

Most public key algorithms have limitations or restrictions on their parameters. For example RSA requires an odd exponent, and algorithms based on the discrete logarithm problem need a generator  $> 1$ .

Each low-level public key type has a function named **check\_key** that takes a **bool**. This function returns a Boolean value that declares whether or not the key is valid (from an algorithmic standpoint). For example, it will check to make sure that the prime parameters of a DSA key are, in fact, prime. It does not have anything to do with the validity of the key for any particular use, nor does it have anything to do with certificates that link a key (which, after all, is just some numbers) with a user or other entity. If **check\_key**'s argument is **true**, then it does "strong" checking, which includes fairly expensive operations like primality checking.

Keys are always checked when they are loaded or generated, so typically there is no reason to use this function directly. However, you can disable or reduce the checks for particular cases (public keys, loaded private keys, generated private keys) by setting the right config toggle (see the section on the configuration subsystem for details).

## 3.3 Getting a PK algorithm object

The key types, like `RSA_PrivateKey`, do not implement any kind of padding or encoding (which is generally necessary for security). To get an object like this, the easiest thing to do is call the functions found in `look.pk.h`. Generally these take a key, followed by a string that specified what hashing and encoding method(s) to use. Examples of such strings are "EME1(SHA-256)" for OAEP encryption and "EMSA4(SHA-256)" for PSS signatures (where the message is hashed using SHA-256).

Here are some basic examples (using an RSA key) to give you a feel for the possibilities. These examples assume `rsa_key` is an `RSA_PrivateKey`, since otherwise we would not be able to create a decryption or signature object with it (you can create encryption or signature verification objects with public keys, naturally). Remember to delete these objects when you're done with them.

```
// PKCS #1 v2.0 / IEEE 1363 compatible encryption
PK_Ecryptor* rsa_enc1 = get_pk_encryptor(rsa_key, "EME1(RIPEMD-160)");
// PKCS #1 v1.5 compatible encryption
PK_Ecryptor* rsa_enc2 = get_pk_encryptor(rsa_key, "PKCS1v15");

// Raw encryption: no padding, input is directly encrypted by the key
// Don't use this unless you know what you're doing
PK_Ecryptor* rsa_enc3 = get_pk_encryptor(rsa_key, "Raw");

// This object can decrypt things encrypted by rsa_enc1
PK_Decryptor* rsa_dec1 = get_pk_decryptor(rsa_key, "EME1(RIPEMD-160)");

// PKCS #1 v1.5 compatible signatures
PK_Signer* rsa_sig = get_pk_signer(rsa_key, "EMSA3(MD5)");
PK_Verifier* rsa_verify = get_pk_verifier(rsa_key, "EMSA3(MD5)");

// PKCS #1 v2.1 compatible signatures
```

```

PK_Signer* rsa_sig2 = get_pk_signer(rsakey, "EMSA4(SHA-1)");
PK_Verifier* rsa_verify2 = get_pk_verifier(rsakey, "EMSA4(SHA-1)");

// Hash input with SHA-1, but don't pad the input in any way; usually
// used with DSA/NR, not RSA
PK_Signer* rsa_sig = get_pk_signer(rsakey, "EMSA1(SHA-1)");

```

### 3.4 Encryption

The `PK_Encoder` and `PK_Decoder` classes are the interface for encryption and decryption, respectively.

Calling **encrypt** with a `byte` array, a length parameter, and an RNG object will return the input encrypted with whatever scheme is being used. Calling the similar **decrypt** will perform the inverse operation. You can also do these operations with `SecureVector<byte>`s. In all cases, the output is returned via a `SecureVector<byte>`.

If you attempt an operation with a larger size than the key can support (this limit varies based on the algorithm, the key size, and the padding method used (if any)), an exception will be thrown. Alternately, you can call **maximum\_input\_size**, that will return the maximum size you can safely encrypt. In fact, you can often encrypt an object that is one byte longer, but only if enough of the high bits of the leading byte are set to zero. Since this is pretty dicey, it's best to stick with the advertised maximum.

Available public key encryption algorithms in Botan are RSA and ElGamal. The encoding methods are EME1, denoted by "EME1(HASHNAME)", PKCS #1 v1.5, called "PKCS1v15" or "EME-PKCS1-v1.5", and raw encoding ("Raw").

For compatibility reasons, PKCS #1 v1.5 is recommended for use with ElGamal (most other implementations of ElGamal do not support any other encoding format). RSA can also be used with PKCS #1 encoding, but because of various possible attacks, EME1 is the preferred encoding. EME1 requires the use of a hash function: unless a competent applied cryptographer tells you otherwise, you should use SHA-256 or SHA-512.

Don't use "Raw" encoding unless you need it for backward compatibility with old protocols. There are many possible attacks against both ElGamal and RSA when they are used in this way.

### 3.5 Signatures

The signature algorithms look quite a bit like the hash functions. You can repeatedly call **update**, giving more and more of a message you wish to sign, and then call **signature**, which will return a signature for that message. If you want to do it all in one shot, call **sign\_message**, which will just call **update** with its argument and then return whatever **signature** returns. Generating a signature requires random numbers with some schemes, so **signature** and **sign\_message** both take a `RandomNumberGenerator&`.

You can validate a signature by updating the verifier class, and finally seeing if the value returned from **check\_signature** is true (you pass the supposed signature to the **check\_signature** function as a byte array and a length or as a `MemoryRegion<byte>`). There is another function, **verify\_message**, which takes a pair of byte array/length pairs (or a pair of `MemoryRegion<byte>` objects), the first of which is the message, the second being the (supposed) signature. It returns true if the signature is valid and false otherwise.

Available public key signature algorithms in Botan are RSA, DSA, Nyberg-Rueppel, and Rabin-Williams. Signature encoding methods include EMSA1, EMSA2, EMSA3, EMSA4, and Raw. All of them, except Raw, take a parameter naming a message digest function to hash the message with. Raw actually signs the input directly; if the message is too big, the signing operation will fail. Raw is not useful except in very specialized applications.

There are various interactions that make certain encoding schemes and signing algorithms more or less useful.

EMSA2 is the usual method for encoding Rabin-William signatures, so for compatibility with other implementations you may have to use that. EMSA4 (also called PSS), also works with Rabin-Williams. EMSA1 and EMSA3 do *not* work with Rabin-Williams.

RSA can be used with any of the available encoding methods. EMSA4 is by far the most secure, but is not (as of now) widely implemented. EMSA3 (also called “EMSA-PKCS1-v1.5”) is commonly used with RSA (for example in SSL). EMSA1 signs the message digest directly, without any extra padding or encoding. This may be useful, but is not as secure as either EMSA3 or EMSA4. EMSA2 may be used but is not recommended.

For DSA and Nyberg-Rueppel, you should use EMSA1. None of the other encoding methods are particularly useful for these algorithms.

### 3.6 Key Agreement

You can get a hold of a `PK_Key_Agreement_Scheme` object by calling `get_pk_kas` with a key that is of a type that supports key agreement (such as a Diffie-Hellman key stored in a `DH_PrivateKey` object), and the name of a key derivation function. This can be “Raw”, meaning the output of the primitive itself is returned as the key, or “KDF1(hash)” or “KDF2(hash)” where “hash” is any string you happen to like (hopefully you like strings like “SHA-256” or “RIPEMD-160”), or “X9.42-PRF(keywrap)”, which uses the PRF specified in ANSI X9.42. It takes the name or OID of the key wrap algorithm that will be used to encrypt a content encryption key.

How key agreement generally works is that you trade public values with some other party, and then each of you runs a computation with the other’s value and your key (this should return the same result to both parties). This computation can be called by using `derive_key` with either a byte array/length pair, or a `SecureVector<byte>` than holds the public value of the other party. The last argument to either call is a number that specifies how long a key you want.

Depending on the key derivation function you’re using, you may not *actually* get back a key of that size. In particular, “Raw” will return a number about the size of the Diffie-Hellman modulus, and KDF1 can only return a key that is the same size as the output of the hash. KDF2, on the other hand, will always give you a key exactly as long as you request, regardless of the underlying hash used with it. The key returned is a `SymmetricKey`, ready to pass to a block cipher, MAC, or other symmetric algorithm.

The public value that should be used can be obtained by calling `public_data`, which exists for any key that is associated with a key agreement algorithm. It returns a `SecureVector<byte>`.

“KDF2(SHA-256)” is by far the preferred algorithm for key derivation in new applications. The X9.42 algorithm may be useful in some circumstances, but unless you need X9.42 compatibility, KDF2 is easier to use.

There is a Diffie-Hellman example included in the distribution, which you may want to examine.

### 3.7 Importing and Exporting PK Keys

[This section mentions `Pipe` and `DataSource`, which is not covered until later in the manual. Please read those sections for more about `Pipe` and `DataSource` and their uses.]

There are many, many different (often conflicting) standards surrounding public key cryptography. There is, thankfully, only two major standards surrounding the representation of a public or private key: X.509 (for public keys), and PKCS #8 (for private keys). Other crypto libraries, like OpenSSL and B-SAFE, also support these formats, so you can easily exchange keys with software that doesn’t use Botan.

In addition to “plain” public keys, Botan also supports X.509 certificates. These are documented in the section “Certificate Handling”, later in this manual.

### 3.7.1 Public Keys

The interfaces for doing either of these are quite similar. Let's look at the X.509 stuff first:

```
namespace X509 {
    MemoryVector<byte> BER_encode(const Public_Key& key);
    std::string PEM_encode(const Public_Key& out);

    Public_Key* load_key(DataSource& in);
    Public_Key* load_key(const std::string& file);
    Public_Key* load_key(const SecureVector<byte>& buffer);
}
```

The function **X509::BER\_encode** will take any **Public\_Key** and return a standard binary structure representing the key which can be read by many other crypto libraries.

The function **X509::PEM\_encode** does the same, but additionally formats it into a text format with headers and base64 encoding. Using PEM is *highly* recommended for many reasons, including compatibility with other software, for transmission over 8-bit unclean channels, because it can be identified by a human without special tools, and because it sometimes allows more sane behavior of tools that process the data.

For loading a public key, the preferred method is one of the variants of **load\_key**. This function will return a newly allocated key based on the data from whatever source it is using (assuming, of course, the source is in fact storing a representation of a public key). The encoding used (PEM or BER) need not be specified; the format will be detected automatically. The key is allocated with **new**, and should be released with **delete** when you are done with it. The first takes a generic **DataSource** that you have to allocate – the others are simple wrapper functions that take either a filename or a memory buffer.

So what can you do with the return value of **load\_key**? On its own, a **Public\_Key** isn't particularly useful; you can't encrypt messages or verify signatures, or much else. But, using **dynamic\_cast**, you can figure out what kind of operations the key supports. Then, you can cast the key to the appropriate type and pass it to a higher-level class. For example:

```
/* Might be RSA, might be ElGamal, might be ... */
Public_Key* key = X509::load_key("pubkey.asc");
/* You MUST use dynamic_cast to convert, because of virtual bases */
PK_Encrypting_Key* enc_key = dynamic_cast<PK_Encrypting_Key*>(key);
if(!enc_key)
    throw Some_Exception();
PK_Ecryptor* enc = get_pk_encryptor(*enc_key, "EME1(SHA-256)");
SecureVector<byte> cipher = enc->encrypt(some_message, size_of_message);
```

### 3.7.2 Private Keys

There are two different options for private key import/export. The first is a plaintext version of the private key. This is supported by the following functions:

```
namespace PKCS8 {
    SecureVector<byte> BER_encode(const Private_Key& key);
    std::string PEM_encode(const Private_Key& key);
}
```

These functions are basically the same as the X.509 functions described previously. The only difference is that they take a **Private\_Key** type (which, again, can be either RSA, DSA, or Diffie-Hellman, but this time the key must be a private key). In most situations, using these is a bad idea, because anyone can come

along and grab the private key without having to know any passwords or other secrets. Unless you have very particular security requirements, always use the versions that encrypt the key based on a passphrase. For importing, the same functions can be used for encrypted and unencrypted keys.

The other way to export a PKCS #8 key is to first encode it in the same manner as done above, then encrypt it (using a passphrase and the techniques of PKCS #5), and store the whole thing into another structure. This method is definitely preferred, since otherwise the private key is unprotected. The following functions support this technique:

```
namespace PKCS8 {

    SecureVector<byte> BER_encode(const Private_Key& key,
                                  RandomNumberGenerator& rng,
                                  const std::string& pass,
                                  const std::string& pbe_algo = "");

    std::string PEM_encode(const Private_Key& key,
                           RandomNumberGenerator& rng,
                           const std::string& pass,
                           const std::string& pbe_algo = "");

}
```

There are three new arguments needed here to support the encryption process in addition to the private key itself. The first is a `RandomNumberGenerator`, which is needed for various purposes internally. The *pass* argument is the passphrase that will be used to encrypt the key. Both of these are required. The final (optional) argument is *pbe*; this specifies a particular password based encryption (or PBE) algorithm. If you don't specify a PBE, a compiled in default will be used; this should be fine.

Last but not least, there are some functions that are basically identical to `X509::load_key` that will load, and possibly decrypt, a PKCS #8 private key:

```
namespace PKCS8 {

    Private_Key* load_key(DataSource& in,
                          RandomNumberGenerator& rng,
                          const User_Interface& ui);

    Private_Key* load_key(DataSource& in,
                          RandomNumberGenerator& rng,
                          std::string passphrase = "");

    Private_Key* load_key(const std::string& filename,
                          RandomNumberGenerator& rng,
                          const User_Interface& ui);

    Private_Key* load_key(const std::string& filename,
                          RandomNumberGenerator& rng,
                          const std::string& passphrase = "");

}
```

The versions that take `std::string` *passphrases* are primarily for compatibility, but they are useful in limited circumstances. The `User_Interface` versions are how `load_key` is actually implemented, and provides for much more flexibility. Essentially, if the passphrase given to the function is not correct, then an exception is thrown and that is that. However, if you pass in an UI object instead, then the UI object can keep asking the user for the passphrase until they get it right (or until they cancel the action, though the UI interface). A `User_Interface` has very little to do with talking to users; it's just a way to glue together Botan and whatever user interface you happen to be using. You can think of it as a user interface interface.

The default `User_Interface` is actually very dumb, and effectively acts just like the versions taking the `std::string`.

All versions need access to a `RandomNumberGenerator` in order to perform probabilistic tests on the loaded key material.

After loading a key, you can use `dynamic_cast` to find out what operations it supports, and use it appropriately. Remember to `delete` it once you are done with it.

## 4 Certificate Handling

A certificate is essentially a binding between some identifying information of a person or other entity (called a *subject*) and a public key. This binding is asserted by a signature on the certificate, which is placed there by some authority (the *issuer*) that at least claims that it knows the subject named in the certificate really “owns” the private key corresponding to the public key in the certificate.

The major certificate format in use today is X.509v3, designed by ISO and further hacked on by dozens (hundreds?) of other organizations.

When working with certificates, the main class to remember is `X509_Certificate`. You can read an object of this type, but you can’t create one on the fly; a CA object is necessary for actually making a new certificate. So for the most part, you only have to worry about reading them in, verifying the signatures, and getting the bits of data in them (most commonly the public key, and the information about the user of that key). An X.509v3 certificate can contain a literally infinite number of items related to all kinds of things. Botan doesn’t support a lot of them, simply because nobody uses them and they’re an impossible mess to work with. This section only documents the most commonly used ones of the ones that are supported; for the rest, read `x509cert.h` and `asn1_obj.h` (which has the definitions of various common ASN.1 constructs used in X.509).

### 4.1 So what’s in an X.509 certificate?

Obviously, you want to be able to get the public key. This is achieved by calling the member function `subject_public_key`, which will return a `Public_Key*`. As to what to do with this, read about `load_key` in the section “Importing and Exporting PK Keys”. In the general case, this could be any kind of public key, though 99% of the time it will be an RSA key. However, Diffie-Hellman and DSA keys are also supported, so be careful about how you treat this. It is also a wise idea to examine the value returned by `constraints`, to see what uses the public key is approved for.

The second major piece of information you’ll want is the name/email/etc of the person to whom this certificate is assigned. Here is where things get a little nasty. X.509v3 has two (well, mostly just two ...) different places where you can stick information about the user: the *subject* field, and in an extension called *subjectAlternativeName*. The *subject* field is supposed to only included the following information: country, organization (possibly), an organizational sub-unit name (possibly), and a so-called common name. The common name is usually the name of the person, or it could be a title associated with a position of some sort in the organization. It may also include fields for state/province and locality. What exactly a locality is, nobody knows, but it’s usually given as a city name.

Botan doesn’t currently support any of the Unicode variants used in ASN.1 (UTF-8, UCS-2, and UCS-4), any of which could be used for the fields in the DN. This could be problematic, particularly in Asia and other areas where non-ASCII characters are needed for most names. The UTF-8 and UCS-2 string types *are* accepted (in fact, UTF-8 is used when encoding much of the time), but if any of the characters included in the string are not in ISO 8859-1 (*i.e.* 0 ... 255), an exception will get thrown. Currently the `ASN1_String` type holds its data as ISO 8859-1 internally (regardless of local character set); this would have to be changed to hold UCS-2 or UCS-4 in order to support Unicode (also, many interfaces in the X.509 code would have to accept or return a `std::wstring` instead of a `std::string`).

Like the distinguished names, subject alternative names can contain a lot of things that Botan will flat out ignore (most of which you would never actually want to use). However, there are three very useful pieces of information that this extension might hold: an email address (“person@site1.com”), a DNS name (“somehost.site2.com”), or a URI (“http://www.site3.com”).

So, how to get the information? Simply call `subject_info` with the name of the piece of information you want, and it will return a `std::string` that is either empty (signifying that the certificate doesn’t have this information), or has the information requested. There are several names for each possible item, but the most easily readable ones are: “Name”, “Country”, “Organization”, “Organizational Unit”, “Locality”, “State”,



“RFC822”, “URI”, and “DNS”. These values are returned as a `std::string`.

You can also get information about the issuer of the certificate in the same way, using `issuer_info`.

#### 4.1.1 X.509v3 Extensions

X.509v3 specifies a large number of possible extensions. Botan supports some, but by no means all of them. This section lists which ones are supported, and notes areas where there may be problems with the handling. You have to be pretty familiar with X.509 in order to understand what this is talking about.

- Key Usage and Extended Key Usage: No problems known.
- Basic Constraints: No problems known. The default for a v1/v2 certificate is assume it’s a CA if and only if the option “x509/default\_to\_ca” is set. A v3 certificate is marked as a CA if (and only if) the basic constraints extension is present and set for a CA cert.
- Subject Alternative Names: Only the “rfc822Name”, “dNSName”, and “uniformResourceIdentifier” fields will be stored; all others are ignored.
- Issuer Alternative Names: Same restrictions as the Subject Alternative Names extension. New certificates generated by Botan never include the issuer alternative name.
- Authority Key Identifier: Only the version using KeyIdentifier is supported. If the GeneralNames version is used and the extension is critical, an exception is thrown. If both the KeyIdentifier and GeneralNames versions are present, then the KeyIdentifier will be used, and the GeneralNames ignored.
- Subject Key Identifier: No problems known.

#### 4.1.2 Revocation Lists

It will occasionally happen that a certificate must be revoked before its expiration date. Examples of this happening include the private key being compromised, or the user to which it has been assigned leaving an organization. Certificate revocation lists are an answer to this problem (though online certificate validation techniques are starting to become somewhat more popular). Essentially, every once in a while the CA will release a CRL, listing all certificates that have been revoked. Also included is various pieces of information like what time a particular certificate was revoked, and for what reason. In most systems, it is wise to support some form of certificate revocation, and CRLs handle this fairly easily.

For most users, processing a CRL is quite easy. All you have to do is call the constructor, which will take a filename (or a `DataSource&`). The CRLs can either be in raw BER/DER, or in PEM format; the constructor will figure out which format without any extra information. For example:

```
X509_CRL crl1("crl1.der");

DataSource_Stream in("crl2.pem");
X509_CRL crl2(in);
```

After that, pass the `X509_CRL` object to a `X509_Store` object with `X509_Code add_crl(X509_CRL)`, and all future verifications will take into account the certificates listed, assuming `add_crl` returns `VERIFIED`. If it doesn’t return `VERIFIED`, then the return value is an error code signifying that the CRL could not be processed due to some problem (which could range from the issuing certificate not being found, to the CRL having some format problem). For more about the `X509_Store` API, read the section later in this chapter.

## 4.2 Reading Certificates

`X509_Certificate` has two constructors, each of which takes a source of data; a filename to read, and a `DataSource&`.

## 4.3 Storing and Using Certificates

If you read a certificate, you probably want to verify the signature on it. However, consider that to do so, we may have to verify the signature on the certificate that we used to verify the first certificate, and on and on until we hit the top of the certificate tree somewhere. It would be a might huge pain to have to handle all of that manually in every application, so there is something that does it for you: **X509.Store**.

This is a pretty easy thing to use. The basic operations are: put certificates and CRLs into it, search for certificates, and attempt to verify certificates. That's about it. In the future, there will be support for online retrieval of certificates and CRLs (*e.g.* with the HTTP cert-store interface currently under consideration by PKIX).

### 4.3.1 Adding Certificates

You can add new certificates to a certificate store using any of these functions:

```
add_cert(const X509_Certificate& cert, bool trusted = false)
add_certs(DataSource& source)
add_trusted_certs(DataSource& source)
```

The versions that take a **DataSource&** will add all the certificates that it can find in that source.

All of them add the cert(s) to the store. The 'trusted' certificates are the ones that you have some reason to trust are genuine. For example, say your application is working with certificates that are owned by employees of some company, and all of their certificates are signed by the company CA, whose certificate is in turned signed by a commercial root CA. What you would then do is include the certificate of the commercial CA with your application, and read it in as a trusted certificate. From there, you could verify the company CA's certificate, and then use that to verify the end user's certificates. Only self-signed certificates may be considered trusted.

### 4.3.2 Adding CRLs

```
X509_Code add_crl(const X509_CRL& crl);
```

This will process the CRL and mark the revoked certificates. This will also work if a revoked certificate is added to the store sometime after the CRL is processed. The function can return an error code (listed later), or will return **VERIFIED** if everything completed successfully.

### 4.3.3 Storing Certificates

You can output a set of certificates by calling **PEM.encode**, which will return a **std::string** containing each of the certificates in the store, PEM encoded and concatenated. This simple format can easily be read by both Botan and other libraries/applications.

### 4.3.4 Searching for Certificates

You can find certificates in the store with a series of functions contained in the **X509.Store.Search** namespace:

```
namespace X509_Store_Search {
std::vector<X509_Certificate> by_email(const X509_Store& store,
                                     const std::string& email_addr);
std::vector<X509_Certificate> by_name(const X509_Store& store,
                                     const std::string& name);
}
```

```
std::vector<X509_Certificate> by_dns(const X509_Store&,
                                     const std::string& dns_name);
}
```

These functions will return a (possibly empty) vector of certificates from *store* matching your search criteria. The email address and DNS name searches are case-insensitive but are sensitive to extra whitespace and so on. The name search will do case-insensitive substring matching, so, for example, calling **X509\_Store\_Search::by\_name**(*your\_store*, “dob”) will return certificates for “J.R. ‘Bob’ Dobbs” and “H. Dobbertin”, assuming both of those certificates are in *your\_store*.

You could then display the results to a user, and allow them to select the appropriate one. Searching using an email address as the key is usually more effective than the name, since email addresses are rarely shared.

#### 4.3.5 Certificate Stores

An object of type **Certificate\_Store** is a generalized interface to an external source for certificates (and CRLs). Examples of such a store would be one that looked up the certificates in a SQL database, or by contacting a CGI script running on a HTTP server. There are currently three mechanisms for looking up a certificate, and one for retrieving CRLs. By default, most of these mechanisms will simply return an empty **std::vector** of **X509\_Certificate**. This storage mechanism is *only* queried when doing certificate validation: it allows you to distribute only the root key with an application, and let some online method handle getting all the other certificates that are needed to validate an end entity certificate. In particular, the search routines will not attempt to access the external database.

The three certificate lookup methods are **by\_SKID** (Subject Key Identifier), **by\_name** (the Common-Name DN entry), and **by\_email** (stored in either the distinguished name, or in a subjectAlternative-Name extension). The name and email versions take a **std::string**, while the SKID version takes a **SecureVector<byte>** containing the subject key identifier in raw binary. You can choose not to implement **by\_name** or **by\_email**, but **by\_SKID** is mandatory to implement, and, currently, is the only version that is used by **X509\_Store**.

Finally, there is a method for finding CRLs, called **get\_crls\_for**, that takes an **X509\_Certificate** object, and returns a **std::vector** of **X509\_CRL**. While generally there will be only one CRL, the use of the vector makes it easy to return no CRLs (*e.g.*, if the certificate store doesn’t support retrieving them), or return multiple ones (for example, if the certificate store can’t determine precisely which key was used to sign the certificate). Implementing the function is optional, and by default will return no CRLs. If it is available, it will be used by **X509\_CRL**.

As for actually using such a store, you have to tell **X509\_Store** about it, by calling the **X509\_Store** member function

```
add_new_certstore(Certificate_Store* new_store)
```

The argument, *new\_store*, will be deleted by **X509\_Store**’s destructor, so make sure to allocate it with **new**.

#### 4.3.6 Verifying Certificates

There is a single function in **X509\_Store** related to verifying a certificate:

```
X509_Code validate_cert(const X509_Certificate& cert, Cert_Usage usage = ANY)
```

To sum things up simply, it returns **VERIFIED** if the certificate can safely be considered valid for the usage(s) described by *usage*, and an error code if it is not. Naturally, things are a bit more complicated than that. The enum **Cert\_Usage** is defined inside the **X509\_Store** class, it (currently) can take on any of the values **ANY** (any usage is OK), **TLS\_SERVER** (for SSL/TLS server authentication), **TLS\_CLIENT** (for SSL/TLS client authentication), **CODE\_SIGNING**, **EMAIL\_PROTECTION** (email encryption, usually this means S/MIME),

**TIME.STAMPING** (in theory any time stamp application, usually IETF PKIX's Time Stamp Protocol), or **CRL.SIGNING**. Note that Microsoft's code signing system, certainly the most widely used, uses a completely different (and basically undocumented) method for marking certificates for code signing.

First, how does it know if a certificate is valid? Basically, a certificate is valid if both of the following hold: a) the signature in the certificate can be verified using the public key in the issuer's certificate, and b) the issuer's certificate is a valid CA certificate. Note that this definition is recursive. We get out of this by "bottoming out" when we reach a certificate that we consider trusted. In general this will either be a commercial root CA, or an organization or application specific CA.

There are actually a few other restrictions (validity periods, key usage restrictions, etc), but the above summarizes the major points of the validation algorithm. In theory, Botan implements the certificate path validation algorithm given in RFC 2459, but in practice it does not (yet), because we don't support the X.509v3 policy or name constraint extensions.

Possible values for *usage* are **TLS\_SERVER**, **TLS\_CLIENT**, **CODE\_SIGNING**, **EMAIL\_PROTECTION**, **CRL\_SIGNING**, and **TIME\_STAMPING**, and **ANY**. The default **ANY** does not mean valid for any use, it means "is valid for some usage". This is generally fine, and in fact requiring that a random certificate support a particular usage will likely result in a lot of failures, unless your application is very careful to always issue certificates with the proper extensions, and you never use certificates generated by other apps.

Return values for **validate\_cert** (and **add\_crl**) include:

- **VERIFIED**: The certificate is valid for the specified use.
- **INVALID\_USAGE**: The certificate cannot be used for the specified use.
- **CANNOT\_ESTABLISH\_TRUST**: The root certificate was not marked as trusted.
- **CERT\_CHAIN\_TOO\_LONG**: The certificate chain exceeded the length allowed by a basicConstraints extension.
- **SIGNATURE\_ERROR**: An invalid signature was found
- **POLICY\_ERROR**: Some problem with the certificate policies was found.
- **CERT\_FORMAT\_ERROR**: Some format problem was found in a certificate.
- **CERT\_ISSUER\_NOT\_FOUND**: The issuer of a certificate could not be found.
- **CERT\_NOT\_YET\_VALID**: The certificate is not yet valid.
- **CERT\_HAS\_EXPIRED**: The certificate has expired.
- **CERT\_IS\_REVOKED**: The certificate has been revoked.
- **CRL\_FORMAT\_ERROR**: Some format problem was found in a CRL.
- **CRL\_ISSUER\_NOT\_FOUND**: The issuer of a CRL could not be found.
- **CRL\_NOT\_YET\_VALID**: The CRL is not yet valid.
- **CRL\_HAS\_EXPIRED**: The CRL has expired.
- **CA\_CERT\_CANNOT\_SIGN**: The CA certificate found does not have an contain a public key that allows signature verification.
- **CA\_CERT\_NOT\_FOR\_CERT\_ISSUER**: The CA cert found is not allowed to issue certificates.
- **CA\_CERT\_NOT\_FOR\_CRL\_ISSUER**: The CA cert found is not allowed to issue CRLs.
- **UNKNOWN\_X509\_ERROR**: Some other error occurred.

## 4.4 Certificate Authorities

Setting up a CA for X.509 certificates is actually probably the easiest thing to do related to X.509. A CA is represented by the type `X509_CA`, which can be found in `x509_ca.h`. A CA always needs its own certificate, which can either be a self-signed certificate (see below on how to create one) or one issued by another CA (see the section on PKCS #10 requests). Creating a CA object is done by the following constructor:

```
X509_CA(const X509_Certificate& cert, const Private_Key& key);
```

The private key is the private key corresponding to the public key in the CA's certificate.

Generally, requests for new certificates are supplied to a CA in the form of PKCS #10 certificate requests (called a `PKCS10_Request` object in Botan). These are decoded in a similar manner to certificates/CRLs/etc. Generally, a request is vetted by humans (who somehow verify that the name in the request corresponds to the name of the person who requested it), and then signed by a CA key, generating a new certificate.

```
X509_Certificate sign_request(const PKCS10_Request&) const;
```

### 4.4.1 Generating CRLs

As mentioned previously, the ability to process CRLs is highly important in many PKI systems. In fact, according to strict X.509 rules, you must not validate any certificate if the appropriate CRLs are not available (though hardly any systems are that strict). In any case, a CA should have a valid CRL available at all times.

Of course, you might be wondering what to do if no certificates have been revoked. In fact, CRLs can be issued without any actually revoked certificates - the list of certs will simply be empty. To generate a new, empty CRL, just call `X509_CRL X509_CA::new_crl(u32bit seconds = 0)` - it will create a new, empty, CRL. If `seconds` is the default 0, then the normal default CRL next update time (the value of the "x509/crl/next\_update") will be used. If not, then `seconds` specifies how long (in seconds) it will be until the CRL's next update time (after this time, most clients will reject the CRL as too old).

On the other hand, you may have issued a CRL before. In that case, you will want to issue a new CRL that contains all previously revoked certificates, along with any new ones. This is done by calling the `X509_CA` member function `update_crl(X509_CRL old_crl, std::vector<CRL_Entry> new_revoked, u32bit seconds = 0)`, where `X509_CRL` is the last CRL this CA issued, and `new_revoked` is a list of any newly revoked certificates. The function returns a new `X509_CRL` to make available for clients. The semantics for the `seconds` argument is the same as `new_crl`.

The `CRL_Entry` type is a structure that contains, at a minimum, the serial number of the revoked certificate. As serial numbers are never repeated, the pairing of an issuer and a serial number (should) distinctly identify any certificate. In this case, we represent the serial number as a `SecureVector<byte>` called `serial`. There are two additional (optional) values, an enumeration called `CRL_Code` that specifies the reason for revocation (`reason`), and an object that represents the time that the certificate became invalid (if this information is known).

If you wish to remove an old entry from the CRL, insert a new entry for the same cert, with a `reason` code of `DELETE_CRL_ENTRY`. For example, if a revoked certificate has expired 'normally', there is no reason to continue to explicitly revoke it, since clients will reject the cert as expired in any case.

### 4.4.2 Self-Signed Certificates

Generating a new self-signed certificate can often be useful, for example when setting up a new root CA, or for use in email applications. In this case, the solution is summed up simply as:

```
namespace X509 {
```

```

X509_Certificate create_self_signed_cert(const X509_Cert_Options& opts,
                                         const Private_Key& key);
}

```

Where *key* is obviously the private key you wish to use (the public key, used in the certificate itself, is extracted from the private key), and *opts* is an structure that has various bits of information that will be used in creating the certificate (this structure, and its use, is discussed below). This function is found in the header `x509self.h`. There is an example of using this function in the `self_sig` example.

#### 4.4.3 Creating PKCS #10 Requests

Also in `x509self.h`, there is a function for generating new PKCS #10 certificate requests.

```

namespace X509 {
    PKCS10_Request create_cert_req(const X509_Cert_Options&,
                                   const Private_Key&);
}

```

This function acts quite similarly to `create_self_signed_cert`, except it instead returns a PKCS #10 certificate request. After creating it, one would typically transmit it to a CA, who signs it and returns a freshly minted X.509 certificate. There is an example of using this function in the `pkcs10` example.

#### 4.4.4 Certificate Options

So what is this `X509_Cert_Options` thing we've been passing around? Basically, it's a bunch of information that will end up being stored into the certificate. This information comes in 3 major flavors: information about the subject (CA or end-user), the validity period of the certificate, and restrictions on the usage of the certificate.

First and foremost is a number of `std::string` members, which contains various bits of information about the user: *common\_name*, *serial\_number*, *country*, *organization*, *org\_unit*, *locality*, *state*, *email*, *dns\_name*, and *uri*. As many of these as possible should be filled it (especially an email address), though the only required ones are *common\_name* and *country*.

There is another value that is only useful when creating a PKCS #10 request, which is called *challenge*. This is a challenge password, which you can later use to request certificate revocation (if the CA supports doing revocations in this manner).

Then there is the validity period; these are set with **not\_before** and **not\_after**. Both of these functions also take a `std::string`, which specifies when the certificate should start being valid, and when it should stop being valid. If you don't set the starting validity period, it will automatically choose the current time. If you don't set the ending time, it will choose the starting time plus a default time period. The arguments to these functions specify the time in the following format: "2002/11/27 1:50:14". The time is in 24-hour format, and the date is encoded as year/month/day. The date must be specified, but you can omit the time or trailing parts of it, for example "2002/11/27 1:50" or "2002/11/27".

Lastly, you can set constraints on a key. The one you're mostly likely to want to use is to create (or request) a CA certificate, which can be done by calling the member function **CA\_key**. This should only be used when needed.

Other constraints can be set by calling the member functions **add\_constraints** and **add\_ex\_constraints**. The first takes a `Key_Constraints` value, and replaces any previously set value. If no value is set, then the certificate key is marked as being valid for any usage. You can set it to any of the following (for more than one usage, OR them together): `DIGITAL_SIGNATURE`, `NON_REPUDIATION`, `KEY_ENCIPHERMENT`, `DATA_ENCIPHERMENT`,

KEY\_AGREEMENT, KEY\_CERT\_SIGN, CRL\_SIGN, ENCIPHER\_ONLY, DECIPHER\_ONLY. Many of these have quite special semantics, so you should either consult the appropriate standards document (such as RFC 3280), or simply not call **add\_constraints**, in which case the appropriate values will be chosen for you.

The second function, **add\_ex\_constraints**, allows you to specify an OID that has some meaning with regards to restricting the key to particular usages. You can, if you wish, specify any OID you like, but there is a set of standard ones that other applications will be able to understand. These are the ones specified by the PKIX standard, and are named “PKIX.ServerAuth” (for TLS server authentication), “PKIX.ClientAuth” (for TLS client authentication), “PKIX.CodeSigning”, “PKIX.EmailProtection” (most likely for use with S/MIME), “PKIX.IPsecUser”, “PKIX.IPsecTunnel”, “PKIX.IPsecEndSystem”, and “PKIX.TimeStamping”. You can call **add\_ex\_constraints** any number of times – each new OID will be added to the list to include in the certificate.

## 5 The Low-Level Interface

Botan has two different interfaces. The one documented in this section is meant more for implementing higher-level types (see the section on filters, earlier in this manual) than for use by applications. Using it safely requires a solid knowledge of encryption techniques and best practices, so unless you know, for example, what CBC mode and nonces are, and why PKCS #1 padding is important, you should avoid this interface in favor of something working at a higher level (such as the CMS interface).

### 5.1 Basic Algorithm Abilities

There are a small handful of functions implemented by most of Botan’s algorithm objects. Among these are:

**std::string name():**

Returns a human-readable string of the name of this algorithm. Examples of names returned are “Blowfish” and “HMAC(MD5)”. You can turn names back into algorithm objects using the functions in `lookup.h`.

**void clear():**

Clear out the algorithm’s internal state. A block cipher object will “forget” its key, a hash function will “forget” any data put into it, etc. Basically, the object will look exactly as it did when you initially allocated it.

**clone():**

This function is central to Botan’s name-based interface. The **clone** has many different return types, such as `BlockCipher*` and `HashFunction*`, depending on what kind of object it is called on. Note that unlike Java’s clone, this returns a new object in a “pristine” state; that is, operations done on the initial object before calling **clone** do not affect the initial state of the new clone.

Cloned objects can (and should) be deallocated with the C++ `delete` operator.

### 5.2 Keys and IVs

Both symmetric keys and initialization values can simply be considered byte (or octet) strings. These are represented by the classes `SymmetricKey` and `InitializationVector`, which are subclasses of `OctetString`.

Since often it’s hard to distinguish between a key and IV, many things (such as key derivation mechanisms) return `OctetString` instead of `SymmetricKey` to allow its use as a key or an IV.

**OctetString(u32bit length):**

This constructor creates a new random key of size *length*.

**OctetString(std::string str):**

The argument *str* is assumed to be a hex string; it is converted to binary and stored. Whitespace is ignored.

**OctetString(const byte input[], u32bit length):**

This constructor simply copies its input.

### 5.3 Symmetrically Keyed Algorithms

Block ciphers, stream ciphers, and MACs all handle keys in pretty much the same way. To make this similarity explicit, all algorithms of those types are derived from the `SymmetricAlgorithm` base class. This type has three functions:

**void set\_key(const byte key[], u32bit length):**



Most algorithms only accept keys of certain lengths. If you attempt to call **set\_key** with a key length that is not supported, the exception **Invalid\_Key\_Length** will be thrown. There is also another version of **set\_key** that takes a **SymmetricKey** as an argument.

```
bool valid_keylength(u32bit length) const:
```

This function returns true if a key of the given length will be accepted by the cipher.

There are also three constant data members of every **SymmetricAlgorithm** object, which specify exactly what limits there are on keys which that object can accept:

**MAXIMUM\_KEYLENGTH**: The maximum length of a key. Usually, this is at most 32 (256 bits), even if the algorithm actually supports more. In a few rare cases larger keys will be supported.

**MINIMUM\_KEYLENGTH**: The minimum length of a key. This is at least 1.

**KEYLENGTH\_MULTIPLE**: The length of the key must be a multiple of this value.

In all cases, **set\_key** must be called on an object before any data processing (encryption, decryption, etc) is done by that object. If this is not done, the results are undefined – that is to say, Botan reserves the right in this situation to do anything from printing a nasty, insulting message on the screen to dumping core.

## 5.4 Block Ciphers

Block ciphers implement the interface **BlockCipher**, found in **base.h**, as well as the **SymmetricAlgorithm** interface.

```
void encrypt(const byte in[BLOCK_SIZE], byte out[BLOCK_SIZE]) const
```

```
void encrypt(byte block[BLOCK_SIZE]) const
```

These functions apply the block cipher transformation to *in* and place the result in *out*, or encrypts *block* in place (*in* may be the same as *out*). **BLOCK\_SIZE** is a constant member of each class, which specifies how much data a block cipher can process at one time. Note that **BLOCK\_SIZE** is not a static class member, meaning you can (given a **BlockCipher\*** named *cipher*), call *cipher*→**BLOCK\_SIZE** to get the block size of that particular object. **BlockCiphers** have similar functions **decrypt**, which perform the inverse operation.

```
AES_128 cipher;
SymmetricKey key(cipher.MAXIMUM_KEYLENGTH); // randomly created
cipher.set_key(key);
```

```
byte in[16] = { /* secrets */ };
byte out[16];
cipher.encrypt(in, out);
```

## 5.5 Stream Ciphers

Stream ciphers are somewhat different from block ciphers, in that encrypting data results in changing the internal state of the cipher. Also, you may encrypt any length of data in one go (in byte amounts).

```
void encrypt(const byte in[], byte out[], u32bit length)
```

```
void encrypt(byte data[], u32bit length):
```

These functions encrypt the arbitrary length (well, less than 4 gigabyte long) string *in* and place it into *out*, or encrypts it in place in *data*. The **decrypt** functions look just like **encrypt**.

Stream ciphers implement the **SymmetricAlgorithm** interface.

Some stream ciphers support random access to any point in their cipher stream. For such ciphers, calling **void seek(u32bit byte)** will change the cipher's state so that it is as if the cipher had been keyed as normal, then encrypted *byte* – 1 bytes of data (so the next byte in the cipher stream is byte number *byte*).

## 5.6 Hash Functions / Message Authentication Codes

Hash functions take their input without producing any output, only producing anything when all input has already taken place. MACs are very similar, but are additionally keyed. Both of these are derived from the base class `BufferedComputation`, which has the following functions.

```
void update(const byte input[], u32bit length)
```

```
void update(byte input)
```

```
void update(const std::string & input)
```

Updates the hash/mac calculation with *input*.

```
void final(byte out[OUTPUT_LENGTH])
```

```
SecureVector<byte> final():
```

Complete the hash/MAC calculation and place the result into *out*. `OUTPUT_LENGTH` is a public constant in each object that gives the length of the hash in bytes. After you call **final**, the hash function is reset to its initial state, so it may be reused immediately.

The second method of using **final** is to call it with no arguments at all, as shown in the second prototype. It will return the hash/mac value in a memory buffer, which will have size `OUTPUT_LENGTH`.

There is also a pair of functions called **process**. They are essentially a combination of a single **update**, and **final**. Both versions return the final value, rather than placing it in an array. Calling **process** with a single byte value isn't available, mostly because it would rarely be useful.

A MAC can be viewed (in most cases) as simply a keyed hash function, so classes that are derived from `MessageAuthenticationCode` have **update** and **final** classes just like a `HashFunction` (and like a `HashFunction`, after **final** is called, it can be used to make a new MAC right away; the key is kept around).

A MAC has the `SymmetricAlgorithm` interface in addition to the `BufferedComputation` interface.

## 6 Random Number Generators

The random number generators provided in Botan are meant for creating keys, IVs, padding, nonces, and anything else that requires ‘random’ data. It is important to remember that the output of these classes will vary, even if they are supplied with exactly the same seed (*i.e.*, two `Randpool` objects with similar initial states will not produce the same output, because the value of high resolution timers is added to the state at various points).

To ensure good quality output, a PRNG needs to be seeded with truly random data (such as that produced by a hardware RNG). Typically, you will use an `EntropySource` (see below). To add entropy to a PRNG, you can use `void add_entropy(const byte data[], u32bit length)` or (better), use the `EntropySource` interface.

Once a PRNG has been initialized, you can get a single byte of random data by calling `byte random()`, or get a large block by calling `void randomize(byte data[], u32bit length)`, which will put random bytes into each member of the array from indexes 0 ... `length - 1`.

You can avoid all the problems inherent in seeding the PRNG by using the globally shared PRNG, described later in this section.

### 6.1 Randpool

`Randpool` is the primary PRNG within Botan. In recent versions all uses of it have been wrapped by an implementation of the X9.31 PRNG (see below). If for some reason you should have cause to create a PRNG instead of using the “global” one owned by the library, it would be wise to consider the same on the grounds of general caution; while `Randpool` is designed with known attacks and PRNG weaknesses in mind, it is not an standard/official PRNG. The remainder of this section is a (fairly technical, though high-level) description of the algorithms used in this PRNG. Unless you have a specific interest in this subject, the rest of this section might prove somewhat uninteresting.

`Randpool` has an internal state called pool, which is 512 bytes long. This is where entropy is mixed into and extracted from. There is also a small output buffer (called buffer), which holds the data which has already been generated but has just not been output yet.

It is based around a MAC and a block cipher (which are currently HMAC(SHA-256) and AES-256). Where a specific size is mentioned, it should be taken as a multiple of the cipher’s block size. For example, if a 256-bit block cipher were used instead of AES, all the sizes internally would double. Every time some new output is needed, we compute the MAC of a counter and a high resolution timer. The resulting MAC is XORed into the output buffer (wrapping as needed), and the output buffer is then encrypted with AES, producing 16 bytes of output.

After 8 blocks (or 128 bytes) have been produced, we mix the pool. To do this, we first rekey both the MAC and the cipher; the new MAC key is the MAC of the current pool under the old MAC key, while the new cipher key is the MAC of the current pool under the just-chosen MAC key. We then encrypt the entire pool in CBC mode, using the current (unused) output buffer as the IV. We then generate a new output buffer, using the mechanism described in the previous paragraph.

To add randomness to the PRNG, we compute the MAC of the input and XOR the output into the start of the pool. Then we remix the pool and produce a new output buffer. The initial MAC operation should make it very hard for chosen inputs to harm the security of `Randpool`, and as HMAC should be able to hold roughly 256 bits of state, it is unlikely that we are wasting much input entropy (or, if we are, it doesn’t matter, because we have a very abundant supply).

## 6.2 ANSI X9.31

`ANSI_X931_PRNG` is the standard issue X9.31 Appendix A.2.4 PRNG, though using AES-256 instead of 3DES as the block cipher. This PRNG implementation has been checked against official X9.31 test vectors.

Internally, the PRNG holds a pointer to another PRNG (typically `Randpool`). This internal PRNG generates the key and seed used by the X9.31 algorithm, as well as the date/time vectors. Each time an X9.31 PRNG object receives entropy, it simply passes it along to the PRNG it is holding, and then pulls out some random bits to generate a new key and seed. This PRNG considers itself seeded as soon as the internal PRNG is seeded.

As of version 1.4.7, the X9.31 PRNG is by default used for all random number generation.

## 6.3 Entropy Sources

An `EntropySource` is an abstract representation of some method of gather “real” entropy. This tends to be very system dependent. The *only* way you should use an `EntropySource` is to pass it to a PRNG that will extract entropy from it – never use the output directly for any kind of key or nonce generation!

`EntropySource` has a pair of functions for getting entropy from some external source, called **`fast_poll`** and **`slow_poll`**. These pass a buffer of bytes to be written; the functions then return how many bytes of entropy were actually gathered. `EntropySources` are usually used to seed the global PRNG using the functions found in the `Global_RNG` namespace.

Note for writers of `EntropySources`: it isn’t necessary to use any kind of cryptographic hash on your output. The data produced by an `EntropySource` is only used by an application after it has been hashed by the `RandomNumberGenerator` that asked for the entropy, thus any hashing you do will be wasteful of both CPU cycles and possibly entropy.

## 7 User Interfaces

Botan has recently changed some infrastructure to better accommodate more complex user interfaces, in particular ones that are based on event loops. Primary among these was the fact that when doing something like loading a PKCS #8 encoded private key, a passphrase might be needed, but then again it might not (a PKCS #8 key doesn't have to be encrypted). Asking for a passphrase to decrypt an unencrypted key is rather pointless. Not only that, but the way to handle the user typing the wrong passphrase was complicated, undocumented, and inefficient.

So now Botan has an object called `UI`, which provides a simple interface for the aspects of user interaction the library has to be concerned with. Currently, this means getting a passphrase from the user, and that's it (`UI` will probably be extended in the future to support other operations as they are needed). The base `UI` class is very stupid, because the library can't directly assume anything about the environment that it's running under (for example, if there will be someone sitting at the terminal, if the application is even *attached* to a terminal, and so on). But since you can subclass `UI` to use whatever method happens to be appropriate for your application, this isn't a big deal.

There is (currently) a single function that can be overridden by subclasses of `UI` (the `std::string` arguments are actually `const std::string&`, but shown as simply `std::string` to keep the line from wrapping):

```
std::string get_passphrase(std::string what, std::string source, UI_Result& result) const;
```

The *what* argument specifies what the passphrase is needed for (for example, PKCS #8 key loading passes *what* as "PKCS #8 private key"). This lets you provide the user with some indication of *why* your application is asking for a passphrase; feel free to pass the string through `gettext(3)` or moral equivalent for i18n purposes. Similarly, *source* specifies where the data in question came from, if available (for example, a file name). If the source is not available for whatever reason, then *source* will be an empty string; be sure to account for this possibility when writing a `UI` subclass.

The function returns the passphrase as the return value, and a status code in *result* (either `OK` or `CANCEL_ACTION`). If `CANCEL_ACTION` is returned in *result*, then the return value will be ignored, and the caller will take whatever action is necessary (typically, throwing an exception stating that the passphrase couldn't be determined). In the specific case of PKCS #8 key decryption, a `Decoding_Error` exception will be thrown; your `UI` should assume this can happen, and provide appropriate error handling (such as putting up a dialog box informing the user of the situation, and canceling the operation in progress).

There is an example `UI` that uses GTK+ available on the web site. The `GTK_UI` code is cleanly separated from the rest of the example, so if you happen to be using GTK+, you can copy (and/or adapt) that code for your application. If you write a `UI` object for another windowing system (Win32, Qt, wxWidgets, FOX, etc), and would like to make it available to users in general (ideally under a permissive license such as public domain or MIT/BSD), feel free to send in a copy.

## 8 Botan’s Modules

Botan comes with a variety of modules that can be compiled into the system. These will not be available on all installations of the library, but you can check for their availability based on whether or not certain macros are defined.

### 8.1 Pipe I/O for Unix File Descriptors

This is a fairly minor feature, but it comes in handy sometimes. In all installations of the library, Botan’s `Pipe` object overloads the `<<` and `>>` operators for C++ `iostream` objects, which is usually more than sufficient for doing I/O.

However, there are cases where the `iostream` hierarchy does not map well to local ‘file types’, so there is also the ability to do I/O directly with Unix file descriptors. This is most useful when you want to read from or write to something like a TCP or Unix-domain socket, or a pipe, since for simple file access it’s usually easier to just use C++’s file streams.

If `BOTAN_EXT_PIPE_UNIXFD_IO` is defined, then you can use the overloaded I/O operators with Unix file descriptors. For an example of this, check out the `hash_fd` example, included in the Botan distribution.

### 8.2 Entropy Sources

All of these are used by the `Global_RNG::seed` function if they are available. Since this function is called by the `LibraryInitializer` class when it is created, it is fairly rare that you will need to deal with any of these classes directly. Even in the case of a long-running server that needs to renew its entropy poll, it is easier to simply call `Global_RNG::seed` (see the section entitled “The Global PRNG” for more details).

**EGD\_EntropySource:** Query an EGD socket. If the macro `BOTAN_EXT_ENTROPY_SRC_EGD` is defined, it can be found in `es_egd.h`. The constructor takes a `std::vector<std::string>` that specifies the paths to look for an EGD socket.

**Unix\_EntropySource:** This entropy source executes programs common on Unix systems (such as `uptime`, `vmstat`, and `df`) and adds it to a buffer. It’s quite slow due to process overhead, and (roughly) 1 bit of real entropy is in each byte that is output. It is declared in `es_unix.h`, if `BOTAN_EXT_ENTROPY_SRC_UNIX` is defined. If you don’t have `/dev/urandom` or EGD, this is probably the thing to use. For a long-running process on Unix, keep on object of this type around and run fast polls ever few minutes.

**FTW\_EntropySource:** Walk through a filesystem (the root to start searching is passed as a string to the constructor), reading files. This tends to only be useful on things like `/proc` that have a great deal of variability over time, and even then there is only a small amount of entropy gathered: about 1 bit of entropy for every 16 bits of output (and many hundreds of bits are read in order to get that 16 bits). It is declared in `es_ftw.h`, if `BOTAN_EXT_ENTROPY_SRC_FTW` is defined. Only use this as a last resort. I don’t really trust it, and neither should you.

**Win32\_CAPI\_EntropySource:** This routines gathers entropy from a Win32 CAPI module. It takes an optional `std::string` that will specify what type of CAPI provider to use. Generally the CAPI RNG is always the same software-based PRNG, but there are a few that may use a hardware RNG. By default it will use the first provider listed in the option “`rng/ms_capi_prov_type`” that is available on the machine (currently the providers “`RSA_FULL`”, “`INTEL_SEC`”, “`FORTEZZA`”, and “`RNG`” are recognized).

**BeOS\_EntropySource:** Query system statistics using various BeOS-specific APIs.

**Pthread\_EntropySource:** Attempt to gather entropy based on jitter between a number of threads competing for a single mutex. This entropy source is *very* slow, and highly questionable in terms of security. However, it provides a worst-case fallback on systems that don’t have Unix-like features, but do support POSIX threads. This module is currently unavailable due to problems on some systems.

## 8.3 Compressors

There are two compression algorithms supported by Botan, Zlib and Bzip2 (Gzip and Zip encoding will be supported in future releases). Only lossless compression algorithms are currently supported by Botan, because they tend to be the most useful for cryptography. However, it is very reasonable to consider supporting something like GSM speech encoding (which is lossy), for use in encrypted voice applications.

You should always compress *before* you encrypt, because encryption seeks to hide the redundancy that compression is supposed to try to find and remove.

### 8.3.1 Bzip2

To test for Bzip2, check to see if `BOTAN_EXT_COMPRESSOR_BZIP2` is defined. If so, you can include `bzip2.h`, which will declare a pair of `Filter` objects: `Bzip2_Compression` and `Bzip2-Decompression`.

You should be prepared to take an exception when using the decompressing filter, for if the input is not valid Bzip2 data, that is what you will receive. You can specify the desired level of compression to `Bzip2_Compression`'s constructor as an integer between 1 and 9, 1 meaning worst compression, and 9 meaning the best. The default is to use 9, since small values take the same amount of time, just use a little less memory.

The Bzip2 module was contributed by Peter J. Jones.

### 8.3.2 Zlib

Zlib compression works pretty much like Bzip2 compression. The only differences in this case are that the macro is `BOTAN_EXT_COMPRESSOR_ZLIB`, the header you need to include is called `botan/zlib.h` (remember that you shouldn't just `#include <zlib.h>`, or you'll get the regular zlib API, which is not what you want). The Botan classes for Zlib compression/decompression are called `Zlib_Compression` and `Zlib-Decompression`.

Like Bzip2, a `Zlib-Decompression` object will throw an exception if invalid (in the sense of not being in the Zlib format) data is passed into it.

In the case of zlib's algorithm, a worse compression level will be faster than a very high compression ratio. For this reason, the Zlib compressor will default to using a compression level of 6. This tends to give a good trade off in terms of time spent to compression achieved. There are several factors you need to consider in order to decide if you should use a higher compression level:

- Better security: the less redundancy in the source text, the harder it is to attack your ciphertext. This is not too much of a concern, because with decent algorithms using sufficiently long keys, it doesn't really matter *that* much (but it certainly can't hurt).
- Decreasing returns. Some simple experiments by the author showed minimal decreases in the size between level 6 and level 9 compression with large (1 to 3 megabyte) files. There was some difference, but it wasn't that much.
- CPU time. Level 9 zlib compression is often two to four times as slow as level 6 compression. This can make a substantial difference in the overall runtime of a program.

While the zlib compression library uses the same compression algorithm as the gzip and zip programs, the format is different. The zlib format is defined in RFC 1950.

### 8.3.3 Data Sources

A `DataSource` is a simple abstraction for a thing that stores bytes. This type is used fairly heavily in the areas of the API related to ASN.1 encoding/decoding. The following types are `DataSources`: `Pipe`,

SecureQueue, and a couple of special purpose ones: `DataSource_Memory` and `DataSource_Stream`.

You can create a `DataSource_Memory` with an array of bytes and a length field. The object will make a copy of the data, so you don't have to worry about keeping that memory allocated. This is mostly for internal use, but if it comes in handy, feel free to use it.

A `DataSource_Stream` is probably more useful than the memory based one. Its constructors take either a `std::istream` or a `std::string`. If it's a stream, the data source will use the `istream` to satisfy read requests (this is particularly useful to use with `std::cin`). If the string version is used, it will attempt to open up a file with that name and read from it.

### 8.3.4 Data Sinks

A `DataSink` (in `data_snk.h`) is a `Filter` that takes arbitrary amounts of input, and produces no output. Generally, this means it's doing something with the data outside the realm of what `Filter`/`Pipe` can handle, for example, writing it to a file (which is what the `DataSink_Stream` does). There is no need for `DataSinks` that write to a `std::string` or memory buffer, because `Pipe` can handle that by itself.

Here's a quick example of using a `DataSink`, which encrypts `in.txt` and sends the output to `out.txt`. There is no explicit output operation; the writing of `out.txt` is implicit.

```
DataSource_Stream in("in.txt");
Pipe pipe(new CBC_Encryption("Blowfish", "PKCS7", key, iv),
          new DataSink_Stream("out.txt"));
pipe.process_msg(in);
```

A real advantage of this is that even if "in.txt" is large, only as much memory is needed for internal I/O buffers will actually be used.

## 8.4 Writing Modules

It's a lot simpler to write modules for Botan that it is to write code in the core library, for several reasons. First, a module can rely on external libraries and services beyond the base ISO C++ libraries, and also machine dependent features. Also, the code can be added at configuration time on the user's end with very little effort (*i.e.* the code can be distributed separately, and included by the user without needing to patch any existing source files).

Each module lives in a subdirectory of the `modules` directory, which exists at the top-level of the Botan source tree. The "short name" of the module is the same as the name of this directory. The only required file in this directory is `info.txt`, which contains directives that specify what a particular module does, what systems it runs on, and so on. Comments in `info.txt` start with a `#` character and continue to end of line.

Recognized directives include:

**realname** <name>: Specify that the 'real world' name of this module is <name>.

**note** <note>: Add a note that will be seen by the end-user at configure time if the module is included into the library.

**require\_version** <version>: Require at configure time that the version of Botan in use be at least <version>.

**define** <macro>[,<macro>[,...]]: Cause the macro `BOTAN_EXT_<macro>` (for each instance of <macro> in the directive) to be defined in `build.h`. This should only be used if the module creates user-visible changes. There is a set of conventions that should be followed in deciding what to call this macro (where xxx denotes some descriptive and distinguishing characteristic of the thing implemented, such as `ALLOC_MLOCK` or `MUTEX_PTHREAD`):



- Allocator: `ALLOC_XXX`
- Compressors: `COMPRESSOR_XXX`
- EntropySource: `ENTROPY_SRC_XXX`
- Engines: `ENGINE_XXX`
- Mutex: `MUTEX_XXX`
- Timer: `TIMER_XXX`

`<libs> / </libs>`: This specifies any extra libraries to be linked in. It is a mapping from OS to library name, for example `linux -> rt`, which means that on Linux `librt` should be linked in. You can also use “all” to force the library to be linked in on all systems.

`<add> / </add>`: Tell the configuration script to add the files named between these two tags into the source tree. All these files must exist in the current module directory.

`<ignore> / </ignore>`: Tell the configuration script to ignore the files named in the main source tree. This is useful, for example, when replacing a C++ implementation with a pure assembly version.

`<replace> / </replace>`: Tell the configuration script to ignore the file given in the main source tree, and instead use the one in the module’s directory.

Additionally, the module file can contain blocks, delimited by the following pairs:

`<os> / </os>`, `<arch> / </arch>`, `<cc> / </cc>`

For example, putting “alpha” and “ia64” in a `<arch>` block will make the configuration script only allow the module to be compiled on those architectures. Not having a block means any value is acceptable.

## 9 Miscellaneous

This section has documentation for anything that just didn't fit into any of the major categories. Many of them (Timers, Allocators) will rarely be used in actual application code, but others, like the S2K algorithms, have a wide degree of applicability.

### 9.1 S2K Algorithms

There are various procedures (usually fairly ad-hoc) for turning a passphrase into a (mostly) arbitrary length key for a symmetric cipher. A general interface for such algorithms is presented in `s2k.h`. The main function is `derive_key`, which takes a passphrase, and the desired length of the output key, and returns a key of that length, deterministically produced from the passphrase. If an algorithm can't produce a key of that size, it will throw an exception (most notably, PKCS #5's PBKDF1 can only produce strings between 1 and  $n$  bytes, where  $n$  is the output size of the underlying hash function).

Most such algorithms allow the use of a "salt", which provides some extra randomness and helps against dictionary attacks on the passphrase. Simply call `change_salt` (there are variations of it for most of the ways you might wish to specify a salt, check the header for details) with a block of random data. You can also have the class generate a new salt for you with `new_random_salt`; the salt that was generated can be retrieved with `current_salt`.

Additionally some algorithms allow you to set some sort of iteration count, which will make the algorithm take longer to compute the final key (reducing the speed of brute-force attacks of various kinds). This can be changed with the `set_iterations` function. Most standards recommend an iteration count of at least 1000. Currently defined S2K algorithms are "PBKDF1(digest)", "PBKDF2(digest)", and "OpenPGP-S2K(digest)"; you can retrieve any of these using the `get_s2k`, found in `lookup.h`. As of this writing, "PBKDF2(SHA-256)" with 10000 iterations and an 8 byte salt is recommend for new applications.

#### 9.1.1 OpenPGP S2K

There are some oddities about OpenPGP's S2K algorithms that are documented here. For one thing, it uses the iteration count in a strange manner; instead of specifying how many times to iterate the hash, it tells how many *bytes* should be hashed in total (including the salt). So the exact iteration count will depend on the size of the salt (which is fixed at 8 bytes by the OpenPGP standard, though the implementation will allow any salt size) and the size of the passphrase.

To get what OpenPGP calls "Simple S2K", set iterations to 0 (the default for OpenPGP S2K), and do not specify a salt. To get "Salted S2K", again leave the iteration count at 0, but give an 8-byte salt. "Salted and Iterated S2K" requires an 8-byte salt and some iteration count (this should be significantly larger than the size of the longest passphrase that might reasonably be used; somewhere from 1024 to 65536 would probably be about right). Using both a reasonably sized salt and a large iteration count is highly recommended to prevent password guessing attempts.

### 9.2 Checksums

Checksums are very similar to hash functions, and in fact share the same interface. But there are some significant differences, the major ones being that the output size is very small (usually in the range of 2 to 4 bytes), and is not cryptographically secure. But for their intended purpose (error checking), they perform very well. Some examples of checksums included in Botan are the Adler32 and CRC32 checksums.

## 9.3 Exceptions

Sooner or later, something is going to go wrong. Botan's behavior when something unusual occurs, like most C++ software, is to throw an exception. Exceptions in Botan are derived from the `Exception` class. You can see most of the major varieties of exceptions used in Botan by looking at `exceptn.h`. The only function you really need to concern yourself with is `const char* what()`. This will return an error message relevant to the error that occurred. For example:

```
try {
    // various Botan operations
}
catch(Botan::Exception& e)
{
    cout << "Botan exception caught: " << e.what() << endl;
    // error handling, or just abort
}
```

Botan's exceptions are derived from `std::exception`, so you don't need to explicitly check for Botan exceptions if you're already catching the ISO standard ones.

## 9.4 Threads and Mutexes

Botan includes a mutex system, which is used internally to lock some shared data structures that must be kept shared for efficiency reasons (mostly, these are in the allocation systems – handing out 1000 separate allocators hurts performance and makes caching memory blocks useless). This system is supported by the `mux_pthr` module, implementing the `Mutex` interface for systems that have POSIX threads.

If your application is using threads, you *must* add the option “thread\_safe” to the options string when you create the `LibraryInitializer` object. If you specify this option and no mutex type is available, an exception is thrown, since otherwise you would probably be facing a nasty crash.

## 9.5 Secure Memory

A major concern with mixing modern multiuser OSes and cryptographic code is that at any time the code (including secret keys) could be swapped to disk, where it can later be read by an attacker. Botan stores almost everything (and especially anything sensitive) in memory buffers that a) clear out their contents when their destructors are called, and b) have easy plugins for various memory locking functions, such as the `mlock(2)` call on many Unix systems.

Two of the allocation methods used (“malloc” and “mmap”) don't require any extra privileges on Unix, but locking memory does. At startup, each allocator type will attempt to allocate a few blocks (typically totaling 128k), so if you want, you can run your application `setuid root`, and then drop privileges immediately after creating your `LibraryInitializer`. If you end up using more than what's been allocated, some of your sensitive data might end up being swappable, but that beats running as `root` all the time. BTW, I would note that, at least on Linux, you can use a kernel module to give your process extra privileges (such as the ability to call `mlock`) without being root. For example, check out my Capability Override LSM ([http://www.randombit.net/projects/cap\\_over/](http://www.randombit.net/projects/cap_over/)), which makes this pretty easy to do.

These classes should also be used within your own code for storing sensitive data. They are only meant for primitive data types (int, long, etc): if you want a container of higher level Botan objects, you can just use a `std::vector`, since these objects know how to clear themselves when they are destroyed. You cannot, however, have a `std::vector` (or any other container) of `Pipes` or `Filters`, because these types have pointers to other `Filters`, and implementing copy constructors for these types would be both hard and quite expensive (vectors of pointers to such objects is fine, though).

These types are not described in any great detail: for more information, consult the definitive sources – the header files `secmem.h` and `allocate.h`.

`SecureBuffer` is a simple array type, whose size is specified at compile time. It will automatically convert to a pointer of the appropriate type, and has a number of useful functions, including `clear()`, and `u32bit size()`, which returns the length of the array. It is a template that takes as parameters a type, and a constant integer which is how long the array is (for example: `SecureBuffer<byte, 8> key;`).

`SecureVector` is a variable length array. Its size can be increased or decreased as need be, and it has a wide variety of functions useful for copying data into its buffer. Like `SecureBuffer`, it implements `clear` and `size`.

## 9.6 Allocators

The containers described above get their memory from allocators. As a user of the library, you can add new allocator methods at run time for containers, including the ones used internally by the library, to use. The interface to this is in `allocate.h`. Basically how it works is that code needing an allocator uses `get_allocator`, which returns a pointer to an allocator. This pointer should not be freed: the caller does not own the allocator (it is shared among multiple users, and locks itself as needed). It is possible to call `get_allocator` with a specific name to request a particular type of allocator, otherwise, a default allocator type is returned.

At start time, the only allocator known is a `DefaultAllocator`, which just allocates memory using `malloc`, and `memsets` it to 0 when the memory is released. It is known by the name “malloc”. If you ask for another type of allocator (“locking” and “mmap” are currently used), and it is not available, some other allocator will be returned.

You can add in a new allocator type using `add_allocator_type`. This function takes a string and a pointer to an allocator. The string gives this allocator type a name to which it can be referred when one is requesting it with `get_allocator`. If an error occurs (such as the name being already registered), this function returns false. It will return true if the allocator was successfully registered. If you ask it to, `LibraryInitializer` will do this for you.

Finally, you can set the default allocator type that will be returned using the policy setting “default\_alloc” to the name of any previously registered allocator.

## 9.7 BigInt

`BigInt` is Botan’s implementation of a multiple-precision integer. Thanks to C++’s operator overloading features, using `BigInt` is often quite similar to using a native integer type. The number of functions related to `BigInt` is quite large. You can find most of them in `bigint.h` and `numthry.h`.

Due to the sheer number of functions involved, only a few, which a regular user of the library might have to deal with, are mentioned here. Fully documenting the MPI library would take a significant while, so if you need to use it now, the best way to learn is to look at the headers.

Probably the most important are the encoding/decoding functions, which transform the normal representation of a `BigInt` into some other form, such as a decimal string. The most useful of these functions are

```
SecureVector<byte> BigInt::encode(BigInt, Encoding)
```

and

```
BigInt BigInt::decode(SecureVector<byte>, Encoding)
```

`Encoding` is an enum that has values `Binary`, `Octal`, `Decimal`, and `Hexadecimal`. The parameter will default to `Binary`. These functions are static member functions, so they would be called like this:

```

BigInt n1; // some number
SecureVector<byte> n1_encoded = BigInt::encode(n1);
BigInt n2 = BigInt::decode(n1_encoded);
// now n1 == n2

```

There are also C++-style I/O operators defined for use with `BigInt`. The input operator understands negative numbers, hexadecimal numbers (marked with a leading “0x”), and octal numbers (marked with a leading ‘0’). The ‘-’ must come before the “0x” or ‘0’ marker. The output operator will never adorn the output; for example, when printing a hexadecimal number, there will not be a leading “0x” (though a leading ‘-’ will be printed if the number is negative). If you want such things, you’ll have to do them yourself.

`BigInt` has constructors that can create a `BigInt` from an unsigned integer or a string. You can also decode a `byte[]` / length pair into a `BigInt`. There are several other `BigInt` constructors, which I would seriously recommend you avoid, as they are only intended for use internally by the library, and may arbitrarily change, or be removed, in a future release.

An essentially random sampling of `BigInt` related functions:

`u32bit BigInt::bytes()`: Return the size of this `BigInt` in bytes.

`BigInt random_prime(u32bit b)`: Return a prime number *b* bits long.

`BigInt gcd(BigInt x, BigInt y)`: Returns the greatest common divisor of *x* and *y*. Uses the binary GCD algorithm.

`bool is_prime(BigInt x)`: Returns true if *x* is a (possible) prime number. Uses the Miller-Rabin probabilistic primality test with fixed bases. For higher assurance, use `verify_prime`, which uses more rounds and randomized 48-bit bases.

### 9.7.1 Efficiency Hints

If you can, always use expressions of the form `a += b` over `a = a + b`. The difference can be *very* substantial, because the first form prevents at least one needless memory allocation, and possibly as many as three.

If you’re doing repeated modular exponentiations with the same modulus, create a `BarrettReducer` ahead of time. If the exponent or base is a constant, use the classes in `mod_exp.h`. This stuff is all handled for you by the normal high-level interfaces, of course.

Never use the low-level MPI functions (those that begin with `bigint_`). These are completely internal to the library, and may make arbitrarily strange and undocumented assumptions about their inputs, and don’t check to see if they are actually true, on the assumption that only the library itself calls them, and that the library knows what the assumptions are. The interfaces for these functions can change completely without notice.

## 10 Algorithms

### 10.1 Recommended Algorithms

This section is by no means the last word on selecting which algorithms to use. However, Botan includes a sometimes bewildering array of possible algorithms, and unless you're familiar with the latest developments in the field, it can be hard to know what is secure and what is not. The following attributes of the algorithms were evaluated when making this list: security, standardization, patent status, support by other implementations, and efficiency (in roughly that order).

It is intended as a set of simple guidelines for developers, and nothing more. It's entirely possible that there are algorithms in Botan that will turn out to be more secure than the ones listed, but the algorithms listed here are (currently) thought to be safe.

- Block ciphers: AES or Serpent in CBC or CTR mode
- Hash functions: SHA-256, SHA-512
- MACs: HMAC with any recommended hash function
- Public Key Encryption: RSA with “EME1(SHA-256)”
- Public Key Signatures: RSA with EMSA4 and any recommended hash, or DSA with “EMSA1(SHA-256)”
- Key Agreement: Diffie-Hellman, with “KDF2(SHA-256)”

### 10.2 Compliance with Standards

Botan is/should be at least roughly compatible with many cryptographic standards, including the following:

- \* RSA: **PKCS #1 v2.1**, **ANSI X9.31**
- \* DSA: **ANSI X9.30**, **FIPS 186-2**
- \* Diffie-Hellman: **ANSI X9.42**, **PKCS #3**
- \* Certificates: **ITU X.509**, **RFC 3280/3281 (PKIX)**, **PKCS #9 v2.0**, **PKCS #10**
- \* Private Key Formats: **PKCS #5 v2.0**, **PKCS #8**
- \* DES/DES-EDE: **FIPS 46-3**, **ANSI X3.92**, **ANSI X3.106**
- \* SHA-1: **FIPS 180-2**
- \* HMAC: **ANSI X9.71**, **FIPS 198**
- \* ANSI X9.19 MAC: **ANSI X9.9**, **ANSI X9.19**

There is also support for the very general standards of **IEEE 1363-2000** and **1363a**. Most of the contents of such are included in the standards mentioned above, in various forms (usually with extra restrictions that 1363 does not impose).

### 10.3 Algorithms Listing

Botan includes a very sizable number of cryptographic algorithms. In nearly all cases, you never need to know the header file or type name to use them. However, you do need to know what string (or strings) are used to

identify that algorithm. Generally, these names conform to those set out by SCAN (Standard Cryptographic Algorithm Naming), which is a document that specifies how strings are mapped onto algorithm objects, which is useful for a wide variety of crypto APIs (SCAN is oriented towards Java, but Botan and several other non-Java libraries also make at least some use of it). For full details, read the SCAN document, which can be found at <http://www.users.zetnet.co.uk/hopwood/crypto/scan/>

Many of these algorithms can take options (such as the number of rounds in a block cipher, the output size of a hash function, etc). These are shown in the following list; all of them default to reasonable values (unless otherwise marked). There are algorithm-specific limits on most of them. When you see something like “HASH” or “BLOCK”, that means you should insert the name of some algorithm of that type. There are no defaults for those options.

A few very obscure algorithms are skipped; if you need one of them, you’ll know it, and you can look in the appropriate header to see what that classes’ **name** function returns (the names tend to match that in SCAN, if it’s defined there).

- **ROUNDS**: The number of rounds in a block cipher.
- **OUTSZ**: The output size of a hash function or MAC
- **PASS**: The number of passes in a hash function (more passes generally means more security).

**Block Ciphers:** “AES”, “Blowfish”, “CAST-128”, “CAST-256”, “DES”, “DESX”, “TripleDES”, “GOST”, “IDEA”, “MARS”, “MISTY1(ROUNDS)”, “RC2”, “RC5(ROUNDS)”, “RC6”, “SAFER-SK(ROUNDS)”, “SEED”, “Serpent”, “Skipjack”, “Square”, “TEA”, “Twofish”, “XTEA”

**Stream Ciphers:** “ARC4”, “MARK4”, “Turing”, “WiderWake4+1-BE”

**Hash Functions:** “FORK-256”, “HAS-160”, “GOST-34.11”, “MD2”, “MD4”, “MD5”, “RIPEMD-128”, “RIPEMD-160”, “SHA-160”, “SHA-256”, “SHA-384”, “SHA-512”, “Skein-512”, “Tiger(OUTSZ,PASS)”, “Whirlpool”

**MACs:** “HMAC(HASH)”, “CMAC(BLOCK)”, “X9.19-MAC”

## 10.4 Compatibility

Generally, cryptographic algorithms are well standardized, thus compatibility between implementations is relatively simple (of course, not all algorithms are supported by all implementations). But there are a few algorithms that are poorly specified, and these should be avoided if you wish your data to be processed in the same way by another implementation (including future versions of Botan).

The block cipher GOST has a particularly poor specification: there are no standard Sboxes, and the specification does not give test vectors even for sample boxes, which leads to issues of endian conventions, etc.

If you wish maximum portability between different implementations of an algorithm, it’s best to stick to strongly defined and well standardized algorithms, TripleDES, AES, HMAC, and SHA-256 all being good examples.

## 11 Support and Further Information

### 11.1 Patents

Some of the algorithms implemented by Botan may be covered by patents in some locations. Algorithms known to have patent claims on them in the United States and that are not available in a license-free/royalty-free manner include: IDEA, MISTY1, RC5, RC6, and Nyberg-Rueppel.

You must not assume that, just because an algorithm is not listed here, it is not encumbered by patents. If you have any concerns about the patent status of any algorithm you are considering using in an application, please discuss it with your attorney.

### 11.2 Recommended Reading

It's a very good idea if you have some knowledge of cryptography prior to trying to use this stuff. You really should read one or more of these books before seriously using the library (note that the Handbook of Applied Cryptography is available for free online):

*Handbook of Applied Cryptography*, Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone; CRC Press

*Security Engineering – A Guide to Building Dependable Distributed Systems*, Ross Anderson; Wiley

*Cryptography: Theory and Practice*, Douglas R. Stinson; CRC Press

*Applied Cryptography, 2nd Ed.*, Bruce Schneier; Wiley

Once you've got the basics down, these are good things to at least take a look at: IEEE 1363 and 1363a, SCAN, NESSIE, PKCS #1 v2.1, the security related FIPS documents, and the CFRG RFCs.

### 11.3 Support

Questions or problems you have with Botan can be directed to the development mailing list. Joining this list is highly recommended if you're going to be using Botan, since often advance notice of upcoming changes is sent there. "Philosophical" bug reports, announcements of programs using Botan, and basically anything else having to do with Botan are also welcome.

The lists can be found at <http://lists.randombit.net/mailman/listinfo/>.

### 11.4 Contact Information

A PGP key with a fingerprint of 621D AF64 11E1 851C 4CF9 A2E1 6211 EBF1 EFBA DFBC is used to sign all Botan releases. This key can be found in the file `doc/pgpkeys.asc`; PGP keys for the developers are also stored there.

Web Site: <http://botan.randombit.net>

### 11.5 License

Copyright ©2000-2008, Jack Lloyd

Licensed under the same terms as the Botan source