Botan Reference Manual

Release 2.4.0-RSCS1

Jack Lloyd Daniel Neus René Korthaus
Juraj Somorovsky Tobias Niemann

2018-08-29

CONTENTS

CHAPTER

ONE

GETTING STARTED

If you need to build the library first, start with *Building The Library*. Some Linux distributions include packages for Botan, so building from source may not be required on your system.

1.1 Books and other references

You should have some knowledge of cryptography *before* trying to use the library. This is an area where it is very easy to make mistakes, and where things are often subtle and/or counterintuitive. Obviously the library tries to provide things at a high level precisely to minimize the number of ways things can go wrong, but naive use will almost certainly not result in a secure system.

Especially recommended are:

- *Cryptography Engineering* by Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno
- Security Engineering – A Guide to Building Dependable Distributed Systems (https://www.cl.cam.ac.uk/~rja14/book.html) by Ross Anderson
- Handbook of Applied Cryptography (http://www.cacr.math.uwaterloo.ca/hac/) by Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone

If you're doing something non-trivial or unique, you might want to at the very least ask for review/input on a mailing list such as the metzdowd (http://www.metzdowd.com/mailman/listinfo/cryptography) or randombit (https://lists.randombit.net/mailman/listinfo/cryptography) crypto lists. And (if possible) pay a professional cryptographer or security company to review your design and code.

CHAPTER TWO

PROJECT GOALS

Botan seeks to be a broadly applicable library that can be used to implement a range of secure distributed systems.

The library has the following project goals guiding changes. It does not succeed in all of these areas in every way just yet, but it describes the system that is the desired end result. Over time further progress is made in each.

- Secure and reliable. The implementations must of course be correct and well tested, and attacks such as side channels and fault attacks should be accounted for where necessary. The library should never crash, or invoke undefined behavior, regardless of circumstances.
- Implement schemes important in practice. It should be practical to implement any real-world crypto protocol using just what the library provides. It is worth some (limited) additional complexity in the library, in order to expand the set of applications which can easily adopt Botan.
- Ease of use. It should be straightforward for an application programmer to do whatever it is they need to do. There should be one obvious way to perform any operation. The API should be predicable, and follow the "principle of least astonishment" in its design. This is not just a nicety; confusing APIs often result in errors that end up compromising security.
- Simplicity of design, clarity of code, ease of review. The code should be easy to read and understand by other library developers, users seeking to better understand the behavior of the code, and by professional reviewers looking for bugs. This is important because bugs in convoluted code can easily escape multiple expert reviews, and end up living on for years.
- Well tested. The code should be correct against the spec, with as close to 100% test coverage as possible. All available static and dynamic analysis tools at our disposal should be used, including fuzzers, symbolic execution, and protocol specific tools. Within reason, all warnings from compilers and static analyzers should be addressed, even if they seem like false positives, because that maximizes the signal value of new warnings from the tool.
- Safe defaults. Policies should aim to be highly restrictive by default, and if they must be made less restrictive by certain applications, it should be obvious to the developer that they are doing something unsafe.
- Post quantum security. Possibly a practical quantum computer that can break RSA and ECC will never be built, but the future is notoriously hard to predict. It seems prudent to begin designing and deploying systems now which have at least the option of using a post-quantum scheme. Botan provides a conservative selection of algorithms thought to be post-quantum secure.
- Performance. Botan does not in every case strive to be faster than every other software implementation, but performance should be competitive and over time new optimizations are identified and applied.
- Support whatever I/O mechanism the application wants. Allow the application to control all aspects of how the network is contacted, and ensure the API makes asynchronous operations easy to handle. This both insulates Botan from system-specific details and allows the application to use whatever networking style they please.
- Portability to modern systems. Botan does not run everywhere, and we actually do not want it to (see non-goals below). But we do want it to run on anything that someone is deploying new applications on. That includes both major OSes like Windows, Linux, and BSD and also relatively new OSes such as IncludeOS.

- Well documented. Ideally every public API would have some place in the manual describing its usage.
- Useful command line utility. The botan command line tool should be flexible and featured enough to replace similar tools such as openssl for everyday users.

2.1 Non-Goals

There are goals some crypto libraries have, but which Botan actively does not seek to address.

- Deep embedded support. Botan requires a heap, C++ exceptions, and RTTI, and at least in terms of performance optimizations effectively assumes a 32 or 64 bit processor. It is not suitable for deploying on, say FreeRTOS running on a MSP430, or smartcard with an 8 bit CPU and 256 bytes RAM. A larger SoC, such as a Cortex-A7 running Linux, is entirely within scope.
- Implementing every crypto scheme in existence. The focus is on algorithms which are in practical use in systems deployed now, as well as promising algorithms for future deployment. Many algorithms which were of interest 5-15 years ago but which never saw widespread deployment and have no compelling benefit over other designs were originally implemented in the library but have since been removed to simplify the codebase.
- Portable to obsolete systems. There is no reason for crypto software to support ancient OS platforms like SunOS or Windows 2000, since these unpatched systems are completely unsafe anyway. The additional complexity supporting such platforms just creates more room for bugs.
- Portable to every C++ compiler ever made. Over time Botan moves forward to both take advantage of new language/compiler features, and to shed workarounds for dealing with bugs in ancient compilers. The set of supported compilers is fixed for each new release branch, for example Botan 2.x will always support GCC 4.8. But a future 3.x release version will likely increase the required versions for all compilers.
- FIPS 140 validation. The primary developer was (long ago) a consultant with a NIST approved testing lab. He does not have a positive view of the process or results, at least when it comes to Level 1 software validations (a Level 4 validation is however the real deal). The only benefit of a Level 1 validation is to allow for government sales, and the cost of validation includes enormous amounts of time and money, adding 'checks' that are useless or actively harmful, then freezing the software version so security updates cannot be applied in the future. It does force a certain minimum standard (ie, FIPS Level 1 does assure AES and RSA are probably implemented correctly) but this is an issue of interop not security since Level 1 does not seriously consider attacks of any kind. Any security budget would be far better spent on a review from a specialized crypto consultancy, who would look for actual flaws.

  That said it would be easy to add a "FIPS 140" build mode to Botan, which just disabled all the builtin crypto and wrapped whatever the most recent OpenSSL FIPS module exports.
- Educational purposes. The library code is intended to be easy to read and review, and so might be useful in an educational context. However it does not contain any toy ciphers (unless you count DES and RC4) nor any tools for simple cryptanalysis. Generally the manual and source comments assume previous knowledge on the basic concepts involved.
- User proof. Some libraries provide a very high level API in an attempt to save the user from themselves. Occasionally they succeed. It would be appropriate and useful to build such an API on top of Botan, but Botan itself wants to cover a broad set of uses cases and some of these involve having pointy things within reach.

CHAPTER

THREE

SUPPORT INFORMATION

3.1 Supported Platforms

For Botan 2, the tier-1 supported platforms are

- Linux x86-64, GCC 4.8 or higher
- Linux x86-64, Clang 3.5 or higher
- Linux aarch64, GCC 4.8+
- Linux ppc64le, GCC 4.8+
- Windows x86-64, Visual C++ 2013 and 2015

These platforms are all tested by continuous integration, and the developers have access to hardware in order to test patches. Problems affecting these platforms are considered release blockers.

For Botan 2, the tier-2 supported platforms are

- Linux x86-32, GCC 4.8+
- Linux arm32, GCC 4.8+
- Windows x86-64, MinGW GCC
- Apple OS X x86-64, XCode Clang
- iOS arm32/arm64, XCode Clang
- Android arm32, NDK Clang
- FreeBSD x86-64, Clang 3.8+
- IncludeOS x86-32, Clang 3.8+

Some (but not all) of the tier-2 platforms are tested by CI. Things should mostly work, and if problems are encountered, the Botan devs will probably be able to help. But they are not as well tested as tier-1.

Of course many other modern OSes such as OpenBSD, NetBSD, AIX, Solaris or QNX are also probably fine (Botan has been tested on all of them successfully in the past), but none of the core developers run these OSes and may not be able to help so much in debugging problems. Patches to improve the build for these platforms are welcome. Note that as a policy Botan does not support any OS which is not supported by its original vendor; any such EOLed systems that are still running are unpatched and insecure.

In theory any working C++11 compiler is fine but in practice, we only test with GCC, Clang, and Visual C++. There is support in the build system for several commercial C++ compilers (Intel, PGI, Sun Studio, Ekopath, etc) all of which worked with older (C++98) versions of both the code and the compilers, but it is not known if the latest versions of these compilers can compile the library properly.

3.2 Branch Support Status

Following table provides the support status for Botan branches. Any branch not listed here (including 1.11) is no longer supported. Dates in the future are approximate.

Branch	First Release	End of Active Development	End of Life
1.8	2008-12-08	2010-08-31	2016-02-13
1.10	2011-06-20	2012-07-10	2018-12-31
2	2017-01-06	2019-01-01	2021-12-31
3 (planned)	2019-04-01	2022-01-01	2023-12-31

"Active development" refers to adding new features and optimizations. At the conclusion of the active development phase, only bugfixes are applied.

3.3 Getting Help

To get help with Botan, open an issue on GitHub (https://github.com/randombit/botan/issues)

3.4 Custom Development or Support

Jack Lloyd, the primary developer, is available for projects including custom development, extended support, developer training, and reviewing code or protocol specifications for security flaws. Email him for more information.

CHAPTER

FOUR

BUILDING THE LIBRARY

This document describes how to build Botan on Unix/POSIX and Windows systems. The POSIX oriented descriptions should apply to most common Unix systems (including OS X), along with POSIX-ish systems like BeOS, QNX, and Plan 9. Currently, systems other than Windows and POSIX (such as VMS, MacOS 9, OS/390, OS/400, . . .) are not supported by the build system, primarily due to lack of access. Please contact the maintainer if you would like to build Botan on such a system.

Botan's build is controlled by configure.py, which is a Python (https://www.python.org) script. Python 2.6 or later is required.

For the impatient, this works for most systems:

```
$ ./configure.py [--prefix=/some/directory]
$ make
$ make install
```

Or using `nmake`, if you're compiling on Windows with Visual C++. On platforms that do not understand the '#!' convention for beginning script files, or that have Python installed in an unusual spot, you might need to prefix the `configure.py` command with `python` or `/path/to/python`:

```
$ python ./configure.py [arguments]
```

4.1 Configuring the Build

The first step is to run `configure.py`, which is a Python script that creates various directories, config files, and a Makefile for building everything. This script should run under a vanilla install of Python 2.6, 2.7, or 3.x.

The script will attempt to guess what kind of system you are trying to compile for (and will print messages telling you what it guessed). You can override this process by passing the options `--cc`, `--os`, and `--cpu`.

You can pass basically anything reasonable with `--cpu`: the script knows about a large number of different architectures, their sub-models, and common aliases for them. You should only select the 64-bit version of a CPU (such as "sparc64" or "mips64") if your operating system knows how to handle 64-bit object code - a 32-bit kernel on a 64-bit CPU will generally not like 64-bit code.

By default the script tries to figure out what will work on your system, and use that. It will print a display at the end showing which algorithms have and have not been enabled. For instance on one system we might see lines like:

```
INFO: Skipping, dependency failure - sessions_sqlite3
INFO: Skipping, incompatible CPU - mp_x86_32 simd_altivec
INFO: Skipping, incompatible OS - beos_stats cryptoapi_rng darwin_secrandom win32_
↪stats
INFO: Skipping, incompatible compiler - mp_x86_32_msvc
```

(continues on next page)

(continued from previous page)

```
INFO: Skipping, loaded only if needed by dependency - dyn_load mp_generic simd_scalar
INFO: Skipping, requires external dependency - boost bzip2 lzma sqlite3 tpm
```

The ones that are skipped because they are require an external dependency have to be explicitly asked for, because they rely on third party libraries which your system might not have or that you might not want the resulting binary to depend on. For instance to enable zlib support, add `--with-zlib` to your invocation of `configure.py`. All available modules can be listed with `--list-modules`.

You can control which algorithms and modules are built using the options `--enable-modules=MODS` and `--disable-modules=MODS`, for instance `--enable-modules=zlib` and `--disable-modules=rc5, idea`. Modules not listed on the command line will simply be loaded if needed or if configured to load by default. If you use `--minimized-build`, only the most core modules will be included; you can then explicitly enable things that you want to use with `--enable-modules`. This is useful for creating a minimal build targeting to a specific application, especially in conjunction with the amalgamation option; see *The Amalgamation Build*.

For instance:

```
$ ./configure.py --minimized-build --enable-modules=rsa,eme_oaep,emsa_pssr
```

will set up a build that only includes RSA, OAEP, PSS along with any required dependencies. A small subset of core features, including AES, SHA-2, HMAC, and the multiple precision integer library, are always loaded. Note that a minimized build does not include any random number generator, which is needed for example to generate keys, nonces and IVs. See *Random Number Generators* on which random number generators are available.

The option `--module-policy=POL` enables modules required by and disables modules prohibited by a text policy in `src/build-data/policy`. Additional modules can be enabled if not prohibited by the policy. Currently available policies include `bsi`, `nist` and `modern`:

```
$ ./configure.py --module-policy=bsi --enable-modules=tls,xts
```

4.1.1 Cross Compiling

Cross compiling refers to building software on one type of host (say Linux x86-64) but creating a binary for some other type (say MinGW x86-32). This is completely supported by the build system. To extend the example, we must tell *configure.py* to use the MinGW tools:

> $ ./configure.py –os=mingw –cpu=x86_32 –cc-bin=i686-w64-mingw32-g++ –ar=i686-w64-mingw32-ar … $ make … $ file botan.exe botan.exe: PE32 executable (console) Intel 80386, for MS Windows

You can also specify the alternate tools by setting the *CXX* and *AR* environment variables (instead of the *–cc-bin* and *–ar-command* options), as is commonly done with autoconf builds.

4.1.2 On Unix

The basic build procedure on Unix and Unix-like systems is:

```
$ ./configure.py [--enable-modules=<list>] [--cc=CC]
$ make
$ ./botan-test
```

If that fails with an error about not being able to find libbotan.so, you may need to set `LD_LIBRARY_PATH`:

```
$ LD_LIBRARY_PATH=. ./botan-test
```

If the tests look OK, install:

```
$ make install
```

On Unix systems the script will default to using GCC; use `--cc` if you want something else. For instance use `--cc=icc` for Intel C++ and `--cc=clang` for Clang.

The `make install` target has a default directory in which it will install Botan (typically `/usr/local`). You can override this by using the `--prefix` argument to `configure.py`, like so:

```
$ ./configure.py --prefix=/opt <other arguments>
```

On some systems shared libraries might not be immediately visible to the runtime linker. For example, on Linux you may have to edit `/etc/ld.so.conf` and run `ldconfig` (as root) in order for new shared libraries to be picked up by the linker. An alternative is to set your `LD_LIBRARY_PATH` shell variable to include the directory that the Botan libraries were installed into.

4.1.3 On macOS

A build on macOS works much like that on any other Unix-like system.

To build a universal binary for macOS, you need to set some additional build flags. Do this with the *configure.py* flag *–cc-abi-flags*:

```
--cc-abi-flags="-force_cpusubtype_ALL -mmacosx-version-min=10.4 -arch i386 -arch ppc"
```

4.1.4 On Windows

Note: The earliest versions of Windows supported are Windows 7 and Windows 2008 R2

You need to have a copy of Python installed, and have both Python and your chosen compiler in your path. Open a command shell (or the SDK shell), and run:

```
$ python configure.py --cc=msvc --os=windows
$ nmake
$ botan-test.exe
$ nmake install
```

Botan supports the nmake replacement Jom (https://wiki.qt.io/Jom) which enables you to run multiple build jobs in parallel.

For MinGW, use:

```
$ python configure.py --cc=gcc --os=mingw
$ make
```

By default the install target will be `C:\botan`; you can modify this with the `--prefix` option.

When building your applications, all you have to do is tell the compiler to look for both include files and library files in `C:\botan`, and it will find both. Or you can move them to a place where they will be in the default compiler search paths (consult your documentation and/or local expert for details).

4.1.5 For iOS using XCode

For iOS, you typically build for 3 architectures: armv7 (32 bit, older iOS devices), armv8-a (64 bit, recent iOS devices) and x86_64 for the iPhone simulator. You can build for these 3 architectures and then create a universal binary containing code for all of these architectures, so you can link to Botan for the simulator as well as for an iOS device.

To cross compile for armv7, configure and make with:

```
$ ./configure.py --os=ios --prefix="iphone-32" --cpu=armv7 --cc=clang \
                 --cc-abi-flags="-arch armv7"
xcrun --sdk iphoneos make install
```

To cross compile for armv8-a, configure and make with:

```
$ ./configure.py --os=ios --prefix="iphone-64" --cpu=armv8-a --cc=clang \
                 --cc-abi-flags="-arch arm64"
xcrun --sdk iphoneos make install
```

To compile for the iPhone Simulator, configure and make with:

```
$ ./configure.py --os=ios --prefix="iphone-simulator" --cpu=x86_64 --cc=clang \
                 --cc-abi-flags="-arch x86_64"
xcrun --sdk iphonesimulator make install
```

Now create the universal binary and confirm the library is compiled for all three architectures:

```
$ xcrun --sdk iphoneos lipo -create -output libbotan-2.a \
               iphone-32/lib/libbotan-2.a \
               iphone-64/lib/libbotan-2.a \
               iphone-simulator/lib/libbotan-2.a
$ xcrun --sdk iphoneos lipo -info libbotan-2.a
Architectures in the fat file: libbotan-2.a are: armv7 x86_64 armv64
```

The resulting static library can be linked to your app in Xcode.

4.1.6 For Android

Instructions for building the library on Android can be found here (https://www.danielseither.de/blog/2013/03/building-the-botan-library-for-android/).

4.2 Other Build-Related Tasks

4.2.1 Building The Documentation

There are two documentation options available, Sphinx and Doxygen. Sphinx will be used if `sphinx-build` is detected in the PATH, or if `--with-sphinx` is used at configure time. Doxygen is only enabled if `--with-doxygen` is used. Both are generated by the makefile target `docs`.

4.2.2 The Amalgamation Build

You can also configure Botan to be built using only a single source file; this is quite convenient if you plan to embed the library into another application.

To generate the amalgamation, run configure.py with whatever options you would ordinarily use, along with the option --amalgamation. This will create two (rather large) files, botan_all.h and botan_all.cpp, plus (unless the option --single-amalgamation-file is used) also some number of files like botan_all_aesni.cpp and botan_all_sse2.cpp which need to be compiled with the appropriate compiler flags to enable that instruction set. The ISA specific files are only generated if there is code that requires them, so you can simplify your build. The --minimized-build option (described elsewhere in this documentation) is also quite useful with the amalgamation.

Whenever you would have included a botan header, you can then include botan_all.h, and include botan_all.cpp along with the rest of the source files in your build. If you want to be able to easily switch between amalgamated and non-amalgamated versions (for instance to take advantage of prepackaged versions of botan on operating systems that support it), you can instead ignore botan_all.h and use the headers from build/include as normal.

You can also build the library using Botan's build system (as normal) but utilizing the amalgamation instead of the individual source files by running something like ./configure.py --amalgamation && make. This is essentially a very simple form of link time optimization; because the entire library source is visible to the compiler, it has more opportunities for interprocedural optimizations. Additionally, amalgamation builds usually have significantly shorter compile times for full rebuilds.

4.2.3 Modules Relying on Third Party Libraries

Currently configure.py cannot detect if external libraries are available, so using them is controlled explicitly at build time by the user using

- --with-bzip2 enables the filters providing bzip2 compression and decompression. Requires the bzip2 development libraries to be installed.
- --with-zlib enables the filters providing zlib compression and decompression. Requires the zlib development libraries to be installed.
- --with-lzma enables the filters providing lzma compression and decompression. Requires the lzma development libraries to be installed.
- --with-sqlite3 enables storing TLS session information to an encrypted SQLite database.
- --with-openssl adds an engine that uses OpenSSL for some public key operations and ciphers/hashes. OpenSSL 1.0.1 or later is supported. LibreSSL is API compatible with OpenSSL 1.0 and can be used instead.

4.2.4 Multiple Builds

It may be useful to run multiple builds with different configurations. Specify --with-build-dir=<dir> to set up a build environment in a different directory.

4.2.5 Setting Distribution Info

The build allows you to set some information about what distribution this build of the library comes from. It is particularly relevant to people packaging the library for wider distribution, to signify what distribution this build is from. Applications can test this value by checking the string value of the macro BOTAN_DISTRIBUTION_INFO. It can be set using the --distribution-info flag to configure.py, and otherwise defaults to "unspecified". For instance, a Gentoo (https://www.gentoo.org) ebuild might set it with --distribution-info="Gentoo ${PVR}" where ${PVR} is an ebuild variable automatically set to a combination of the library and ebuild versions.

4.2.6 Local Configuration Settings

You may want to do something peculiar with the configuration; to support this there is a flag to `configure.py` called `--with-local-config=<file>`. The contents of the file are inserted into `build/build.h` which is (indirectly) included into every Botan header and source file.

4.2.7 Configuration Parameters

There are some configuration parameters which you may want to tweak before building the library. These can be found in `build.h`. This file is overwritten every time the configure script is run (and does not exist until after you run the script for the first time).

Also included in `build/build.h` are macros which let applications check which features are included in the current version of the library. All of them begin with `BOTAN_HAS_`. For example, if `BOTAN_HAS_BLOWFISH` is defined, then an application can include `<botan/blowfish.h>` and use the Blowfish class.

`BOTAN_MP_WORD_BITS`: This macro controls the size of the words used for calculations with the MPI implementation in Botan. You can choose 8, 16, 32, or 64. Normally this defaults to either 32 or 64, depending on the processor. Unless you are building for a 8 or 16-bit CPU, this isn't worth messing with.

`BOTAN_DEFAULT_BUFFER_SIZE`: This constant is used as the size of buffers throughout Botan. The default should be fine for most purposes, reduce if you are very concerned about runtime memory usage.

4.3 Building Applications

4.3.1 Unix

Botan usually links in several different system libraries (such as `librt` or `libz`), depending on which modules are configured at compile time. In many environments, particularly ones using static libraries, an application has to link against the same libraries as Botan for the linking step to succeed. But how does it figure out what libraries it *is* linked against?

The answer is to ask the `botan` command line tool using the `config` and `version` commands.

`botan version`: Print the Botan version number.

`botan config prefix`: If no argument, print the prefix where Botan is installed (such as `/opt` or `/usr/local`).

`botan config cflags`: Print options that should be passed to the compiler whenever a C++ file is compiled. Typically this is used for setting include paths.

`botan config libs`: Print options for which libraries to link to (this will include a reference to the botan library iself).

Your `Makefile` can run `botan config` and get the options necessary for getting your application to compile and link, regardless of whatever crazy libraries Botan might be linked against.

4.3.2 Windows

No special help exists for building applications on Windows. However, given that typically Windows software is distributed as binaries, this is less of a problem - only the developer needs to worry about it. As long as they can remember where they installed Botan, they just have to set the appropriate flags in their Makefile/project file.

4.4 Language Wrappers

4.4.1 Building the Python wrappers

The Python wrappers for Botan use ctypes and the C89 API so no special build step is required, just import botan2.py

See *Python Bindings* for more information about the Python bindings.

4.4.2 Building the Perl XS wrappers

To build the Perl XS wrappers, after building the main library change your directory to `src/contrib/perl-xs` and run `perl Makefile.PL`, then run `make` to build the module and `make test` to run the test suite:

```
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Botan
$ make
cp Botan.pm blib/lib/Botan.pm
AutoSplitting blib/lib/Botan.pm (blib/lib/auto/Botan)
/usr/bin/perl5.8.8 /usr/lib64/perl5/5.8.8/ExtUtils/xsubpp  [...]
g++ -c   -Wno-write-strings -fexceptions  -g   [...]
Running Mkbootstrap for Botan ()
chmod 644 Botan.bs
rm -f blib/arch/auto/Botan/Botan.so
g++  -shared Botan.o  -o blib/arch/auto/Botan/Botan.so  \
           -lbotan -lbz2 -lpthread -lrt -lz     \

chmod 755 blib/arch/auto/Botan/Botan.so
cp Botan.bs blib/arch/auto/Botan/Botan.bs
chmod 644 blib/arch/auto/Botan/Botan.bs
Manifying blib/man3/Botan.3pm
$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl5.8.8 [...]
t/base64......ok
t/filt........ok
t/hex.........ok
t/oid.........ok
t/pipe........ok
t/x509cert....ok
All tests successful.
Files=6, Tests=83,  0 wallclock secs ( 0.08 cusr +  0.02 csys =  0.10 CPU)
```

CHAPTER FIVE

VERSIONING

All versions are of the tuple (major,minor,patch).

As of Botan 2.0.0, Botan uses semantic versioning. The minor number increases if any feature addition is made. The patch version is used to indicate a release where only bug fixes were applied. If an incompatible API change is required, the major version will be increased.

The library has functions for checking compile-time and runtime versions.

The build-time version information is defined in *botan/build.h*

`BOTAN_VERSION_MAJOR`

The major version of the release.

`BOTAN_VERSION_MINOR`

The minor version of the release.

`BOTAN_VERSION_PATCH`

The patch version of the release.

`BOTAN_VERSION_DATESTAMP`

Expands to an integer of the form YYYYMMDD if this is an official release, or 0 otherwise. For instance, 1.10.1, which was released on July 11, 2011, has a *BOTAN_VERSION_DATESTAMP* of 20110711.

`BOTAN_DISTRIBUTION_INFO`

New in version 1.9.3.

A macro expanding to a string that is set at build time using the `--distribution-info` option. It allows a packager of the library to specify any distribution-specific patches. If no value is given at build time, the value is the string "unspecified".

`BOTAN_VERSION_VC_REVISION`

New in version 1.10.1.

A macro expanding to a string that is set to a revision identifier cooresponding to the source, or "unknown" if this could not be determined. It is set for all official releases, and for builds that originated from within a git checkout.

The runtime version information, and some helpers for compile time version checks, are included in *botan/version.h*

std::string **`version_string`**`()`

Returns a single-line string containing relevant information about this build and version of the library in an unspecified format.

uint32_t **`version_major`**`()`

Returns the major part of the version.

uint32_t **`version_minor`**`()`

Returns the minor part of the version.

uint32_t **version_patch**()

Returns the patch part of the version.

uint32_t **version_datestamp**()

Return the datestamp of the release (or 0 if the current version is not an official release).

std::string **runtime_version_check**(uint32_t *major*, uint32_t *minor*, uint32_t *patch*)

Call this function with the compile-time version being built against, eg:

```
Botan::runtime_version_check(BOTAN_VERSION_MAJOR, BOTAN_VERSION_MINOR, BOTAN_
↪VERSION_PATCH)
```

It will return an empty string if the versions match, or otherwise an error message indicating the discrepency. This only is useful in dynamic libraries, where it is possible to compile and run against different versions.

BOTAN_VERSION_CODE_FOR(maj, min, patch)

Return a value that can be used to compare versions. The current (compile-time) version is available as the macro *BOTAN_VERSION_CODE*. For instance, to choose one code path for version 2.1.0 and later, and another code path for older releases:

```
#if BOTAN_VERSION_CODE >= BOTAN_VERSION_CODE_FOR(2,1,0)
   // 2.1+ code path
#else
   // code path for older versions
#endif
```

CHAPTER

SIX

MEMORY CONTAINER

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, or left floating around in memory for later retreval.

For this reason the library uses a `std::vector` with a custom allocator that will zero memory before deallocation, named via typedef as `secure_vector`. Because it is simply a STL vector with a custom allocator, it has an identical API to the `std::vector` you know and love.

Some operating systems offer the ability to lock memory into RAM, preventing swapping from occurring. Typically this operation is restricted to privledged users (root or admin), however some OSes including Linux and FreeBSD allow normal users to lock a small amount of memory. On these systems, allocations first attempt to allocate out of this small locked pool, and then if that fails will fall back to normal heap allocations.

The `secure_vector` template is only meant for primitive data types (bytes or ints): 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 `Pipe` objects 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).

CHAPTER

SEVEN

RANDOM NUMBER GENERATORS

The base class `RandomNumberGenerator` is in the header `botan/rng.h`.

The major interfaces are

void `RandomNumberGenerator::`**`randomize`**(uint8_t **output_array*, size_t *length*)

Places *length* random bytes into the provided buffer.

void `RandomNumberGenerator::`**`add_entropy`**(**`const`** uint8_t **data*, size_t *length*)

Incorporates provided data into the state of the PRNG, if at all possible. This works for most RNG types, including the system and TPM RNGs. But if the RNG doesn't support this operation, the data is dropped, no error is indicated.

void `RandomNumberGenerator::`**`randomize_with_input`**(uint8_t **data*, size_t *length*, **`const`** uint8_t **ad*, size_t *ad_len*)

Like randomize, but first incorporates the additional input field into the state of the RNG. The additional input could be anything which parameterizes this request. Not all RNG types accept additional inputs.

void `RandomNumberGenerator::`**`randomize_with_ts_input`**(uint8_t **data*, size_t *length*)

Creates a buffer with some timestamp values and calls `randomize_with_input`

uint8_t `RandomNumberGenerator::`**`next_byte`**()

Generates a single random byte and returns it. Note that calling this function several times is much slower than calling `randomize` once to produce multiple bytes at a time.

7.1 RNG Types

The following RNG types are included

7.1.1 HMAC_DRBG

HMAC DRBG is a random number generator designed by NIST and specified in SP 800-90A. It seems to be the most conservative generator of the NIST approved options.

It can be instantiated with any HMAC but is typically used with SHA-256, SHA-384, or SHA-512, as these are the hash functions approved for this use by NIST.

7.1.2 System_RNG

In `system_rng.h`, objects of `System_RNG` reference a single (process global) reference to the system PRNG (such as `/dev/urandom` or `CryptGenRandom`).

You can also use the function `system_rng()` which returns a reference to the global handle to the system RNG.

7.1.3 AutoSeeded_RNG

AutoSeeded_RNG is type naming a 'best available' userspace PRNG. The exact definition of this has changed over time and may change in the future, fortunately there is no compatability concerns when changing any RNG since the only expectation is it produces bits indistinguishable from random.

Note well: like most other classes in Botan, it is not safe to share an instance of `AutoSeeded_RNG` among multiple threads without serialization.

The current version uses the HMAC_DRBG with SHA-384 or SHA-256. The initial seed is generated either by the system PRNG (if available) or a default set of entropy sources. These are also used for periodic reseeding of the RNG state.

7.1.4 ChaCha_RNG

This is a very fast userspace PRNG based on ChaCha20 and HMAC(SHA-256). The key for ChaCha is derived by hashing entropy inputs with HMAC. Then the ChaCha keystream generator is run, first to generate the new HMAC key (used for any future entropy additions), then the desired RNG outputs.

This RNG composes two primitives thought to be secure (ChaCha and HMAC) in a simple and well studied way (the extract-then-expand paradigm), but is still an ad-hoc and non-standard construction. It is included because it is roughly 20x faster then HMAC_DRBG, and certain applications need access to a very fast RNG.

7.1.5 RDRAND_RNG

This RNG type directly calls the x86 `rdrand` instruction. If the instruction is not available it will throw at runtime, you can check beforehand by calling `Botan::CPUID::has_rdrand()`.

7.1.6 TPM_RNG

This RNG type allows using the RNG exported from a TPM chip.

7.1.7 PKCS11_RNG

This RNG type allows using the RNG exported from a hardware token accessed via PKCS11.

7.2 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 gathered.

Note for writers of `EntropySource` subclasses: 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 entropy.

7.3 Fork Safety

On Unix platforms, the `fork()` and `clone()` system calls can be used to spawn a new child process. Fork safety ensures that the child process doesn't see the same output of random bytes as the parent process. Botan tries to ensure fork safety by feeding the process ID into the internal state of the random generator and by automatically reseeding the random generator if the process ID changed between two requests of random bytes. However, this does not protect against PID wrap around. The process ID is usually implemented as a 16 bit integer. In this scenario, a process will spawn a new child process, which exits the parent process and spawns a new child process himself. If the PID wrapped around, the second child process may get assigned the process ID of it's grandparent and the fork safety can not be ensured.

Therefore, it is strongly recommended to explicitly reseed any userspace random generators after forking a new process. If this is not possible in your application, prefer using the system PRNG instead.

CHAPTER

EIGHT

HASH FUNCTIONS AND CHECKSUMS

Hash functions are one-way functions, which map data of arbitrary size to a fixed output length. Most of the hash functions in Botan are designed to be cryptographically secure, which means that it is computationally infeasible to create a collision (finding two inputs with the same hash) or preimages (given a hash output, generating an arbitrary input with the same hash). But note that not all such hash functions meet their goals, in particular MD4 and MD5 are trivially broken. However they are still included due to their wide adoption in various protocols.

The class `HashFunction` is defined in *botan/hash.h*.

Using a hash function is typically split into three stages: initialization, update, and finalization (often referred to as a IUF interface). The initialization stage is implicit: after creating a hash function object, it is ready to process data. Then update is called one or more times. Calling update several times is equivalent to calling it once with all of the arguments concatenated. After completing a hash computation (eg using `final`), the internal state is reset to begin hashing a new message.

class HashFunction

size_t **output_length**()
: Return the size (in *bytes*) of the output of this function.

void **update**(**const** uint8_t **input*, size_t *length*)
: Updates the computation with *input*.

void **update**(uint8_t *input*)
: Updates the computation with *input*.

void **update**(**const** std::vector<uint8_t> &*input*)
: Updates the computation with *input*.

void **update**(**const** std::string &*input*)
: Updates the computation with *input*.

void **final**(uint8_t **out*)
: Finalize the calculation and place the result into `out`. For the argument taking an array, exactly `output_length` bytes will be written. After you call `final`, the algorithm is reset to its initial state, so it may be reused immediately.

secure_vector<uint8_t> **final**()
: Similar to the other function of the same name, except it returns the result in a newly allocated vector.

secure_vector<uint8_t> **process**(**const** uint8_t *in*[], size_t *length*)
: Equivalent to calling `update` followed by `final`.

secure_vector<uint8_t> **process**(**const** std::string &*in*)
: Equivalent to calling `update` followed by `final`.

8.1 Code Example

Assume we want to calculate the SHA-1, Whirlpool and SHA-3 hash digests of the STDIN stream using the Botan library.

```
#include <botan/hash.h>
#include <botan/hex.h>
#include <iostream>
int main ()
   {
   std::unique_ptr<Botan::HashFunction> hash1(Botan::HashFunction::create("SHA-1"));
   std::unique_ptr<Botan::HashFunction> hash2(Botan::HashFunction::create("Whirlpool
↪"));
   std::unique_ptr<Botan::HashFunction> hash3(Botan::HashFunction::create("SHA-3"));
   std::vector<uint8_t> buf(2048);

   while(std::cin.good())
      {
      //read STDIN to buffer
      std::cin.read(reinterpret_cast<char*>(buf.data()), buf.size());
      size_t readcount = std::cin.gcount();
      //update hash computations with read data
      hash1->update(buf.data(),readcount);
      hash2->update(buf.data(),readcount);
      hash3->update(buf.data(),readcount);
      }
   std::cout << "SHA-1: " << Botan::hex_encode(hash1->final()) << std::endl;
   std::cout << "Whirlpool: " << Botan::hex_encode(hash2->final()) << std::endl;
   std::cout << "SHA-3: " << Botan::hex_encode(hash3->final()) << std::endl;
   return 0;
   }
```

8.2 Available Hash Functions

The following cryptographic hash functions are implemented.

8.2.1 BLAKE2b

Available if BOTAN_HAS_BLAKE2B is defined.

A recently designed hash function. Very fast on 64-bit processors. Can output a hash of any length between 1 and 64 bytes, this is specified by passing a value to the constructor with the desired length.

8.2.2 GOST-34.11

Available if BOTAN_HAS_GOST_34_11 is defined.

Russian national standard hash. It is old, slow, and has some weaknesses. Avoid it unless you must.

8.2.3 Keccak-1600

Available if BOTAN_HAS_KECCAK is defined.

An older (and incompatible) variant of SHA-3, but sometime used. Prefer SHA-3 in new code.

8.2.4 MD4

Available if `BOTAN_HAS_MD4` is defined.

An old hash function that is now known to be trivially breakable. It is very fast, and may still be suitable as a (non-cryptographic) checksum.

8.2.5 MD5

Available if `BOTAN_HAS_MD5` is defined.

Widely used, now known to be broken.

8.2.6 RIPEMD-160

Available if `BOTAN_HAS_RIPEMD160` is defined.

A 160 bit hash function, quite old but still thought to be secure. Somewhat deprecated these days.

8.2.7 SHA-1

Available if `BOTAN_HAS_SHA1` is defined.

Widely adopted NSA designed hash function. Starting to show significant signs of weakness, and collisions can now be generated. Avoid in new designs.

8.2.8 SHA-256

Available if `BOTAN_HAS_SHA2_32` is defined.

Relatively fast 256 bit hash function, thought to be secure.

Also includes the variant SHA-224. There is no real reason to use SHA-224.

8.2.9 SHA-512

Available if `BOTAN_HAS_SHA2_64` is defined.

SHA-512 is faster than SHA-256 on 64-bit processors. Also includes the truncated variants SHA-384 and SHA-512/256.

8.2.10 SHA-3

Available if `BOTAN_HAS_SHA3` is defined.

The new NIST standard hash. Fairly slow.

8.2.11 SHAKE (SHAKE-128, SHAKE-256)

Available if `BOTAN_HAS_SHAKE` is defined.

These are actually XOFs (extensible output functions) based on SHA-3, which can output a value of any length.

8.2.12 SM3

Available if `BOTAN_HAS_SM3` is defined.

Chinese national hash function, 256 bit output. Widely used in industry there. Fast and seemingly secure.

8.2.13 Skein-512

Available if `BOTAN_HAS_SKEIN_512` is defined.

A contender for the NIST SHA-3 competition. Very fast on 64-bit systems. Can output a hash of any length between 1 and 64 bytes. It also accepts a "personalization string" which can create variants of the hash. This is useful for domain separation.

8.2.14 Streebog (Streebog-256, Streebog-512)

Available if `BOTAN_HAS_STREEBOG` is defined.

Newly designed Russian national hash function. Seemingly secure, but there is no real reason to use it unless compatibility is needed.

8.2.15 Tiger

Available if `BOTAN_HAS_TIGER` is defined.

An older 192-bit hash function, optimized for 64-bit systems. Seemingly secure but not widely used. Prefer Skein-512 or BLAKE2b in new code.

8.2.16 Whirlpool

Available if `BOTAN_HAS_WHIRLPOOL` is defined.

A 512-bit hash function standarized by ISO and NESSIE. Relatively slow. Prefer Skein-512 or BLAKE2b in new code.

8.3 Hash Function Combiners

These are functions which combine multiple hash functions to create a new hash function. They are typically only used in specialized applications.

8.3.1 Parallel

Available if `BOTAN_HAS_PARALLEL_HASH` is defined.

Parallel simply concatenated multiple hash functions. For example “Parallel(SHA-256,SHA-512)” outputs a 256+512 bit hash created by hashing the input with both SHA-256 and SHA-512 and concatenating the outputs.

Note that due to the “multicollision attack” it turns out that generating a collision for multiple parallel hash functions is no harder than generating a collision for the strongest hash function.

8.3.2 Comp4P

Available if `BOTAN_HAS_COMB4P` is defined.

This combines two cryptographic hashes in such a way that preimage and collision attacks are provably at least as hard as a preimage or collision attack on the strongest hash.

8.4 Checksums

Note: Checksums are not suitable for cryptographic use, but can be used for error checking purposes.

8.4.1 Adler32

Available if `BOTAN_HAS_ADLER32` is defined.

The Adler32 checksum is used in the zlib format. 32 bit output.

8.4.2 CRC24

Available if `BOTAN_HAS_CRC24` is defined.

This is the CRC function used in OpenPGP. 24 bit output.

8.4.3 CRC32

Available if `BOTAN_HAS_CRC32` is defined.

This is the 32-bit CRC used in protocols such as Ethernet, gzip, PNG, etc.

CHAPTER

NINE

BLOCK CIPHERS

Block ciphers are a n-bit permutation for some small n, typically 64 or 128 bits. They are a cryptographic primitive used to generate higher level operations such as authenticated encryption.

Note: In general a bare block cipher is not what you should be using. You probably want a cipher mode instead (see *Cipher Modes*)

`class BlockCipher`

`static` std::unique_ptr<*BlockCipher*> **`create`**(**`const`** std::string *&algo_spec*, **`const`** std::string *&provider* = "")

Create a new block cipher object, or else return null.

`static` std::unique_ptr<*BlockCipher*> **`create_or_throw`**(**`const`** std::string *&algo_spec*, **`const`** std::string *&provider* = "")

Like `create`, except instead of returning null an exception is thrown if the cipher is not known.

void **`set_key`**(**`const`** uint8_t **key*, size_t *length*)

This sets the key to the value specified. 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.

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, an exception will be thrown. thrown.

bool **`valid_keylength`**(size_t *length*) **`const`**

This function returns true if and only if *length* is a valid keylength for this algorithm.

size_t **`minimum_keylength`**() **`const`**

Return the smallest key length (in bytes) that is acceptible for the algorithm.

size_t **`maximum_keylength`**() **`const`**

Return the largest key length (in bytes) that is acceptible for the algorithm.

std::string **`name`**() **`const`**

Return a human readable name for this algorithm. This is guaranteed to round-trip with `create` and `create_or_throw` calls, ie create("Foo")->name() == "Foo"

void **`clear`**()

Zero out the key. The key must be reset before the cipher object can be used.

BlockCipher ***`clone`**() **`const`**

Return a newly allocated BlockCipher object of the same type as this one.

size_t **`block_size`**() **`const`**

Return the size (in *bytes*) of the cipher.

size_t **parallelism**() **const**

Return the parallelism underlying this implementation of the cipher. This value can vary across versions and machines. A return value of N means that encrypting or decrypting with N blocks can operate in parallel.

size_t **parallel_bytes**() **const**

Returns `parallelism` multiplied by the block size as well as a small fudge factor. That's because even ciphers that have no implicit parallism typically see a small speedup for being called with several blocks due to caching effects.

std::string **provider**() **const**

Return the provider type. Default value is "base" but can be any arbitrary string. Other example values are "sse2", "avx2", "openssl".

void **encrypt_n**(**const** uint8_t *in*[], uint8_t *out*[], size_t *blocks*) **const**

Encrypt *blocks* blocks of data, taking the input from the array *in* and placing the ciphertext into *out*. The two pointers may be identical, but should not overlap ranges.

void **decrypt_n**(**const** uint8_t *in*[], uint8_t *out*[], size_t *blocks*) **const**

Decrypt *blocks* blocks of data, taking the input from the array *in* and placing the plaintext into *out*. The two pointers may be identical, but should not overlap ranges.

void **encrypt**(**const** uint8_t *in*[], uint8_t *out*[]) **const**

Encrypt a single block. Equivalent to *encrypt_n*(in, out, 1).

void **encrypt**(uint8_t *block*[]) **const**

Encrypt a single block. Equivalent to *encrypt_n*(block, block, 1)

void **decrypt**(**const** uint8_t *in*[], uint8_t *out*[]) **const**

Decrypt a single block. Equivalent to *decrypt_n*(in, out, 1)

void **decrypt**(uint8_t *block*[]) **const**

Decrypt a single block. Equivalent to *decrypt_n*(block, block, 1)

template<typename **Alloc**>
void **encrypt**(std::vector<uint8_t, Alloc> &*block*) **const**

Assumes `block` is of a multiple of the block size.

template<typename **Alloc**>
void **decrypt**(std::vector<uint8_t, Alloc> &*block*) **const**

Assumes `block` is of a multiple of the block size.

9.1 Code Example

For sheer demonstrative purposes, the following code encrypts a provided single block of plaintext with AES-256 using two different keys.

```
#include <botan/block_cipher.h>
#include <botan/hex.h>
#include <iostream>
int main ()
   {
   std::vector<uint8_t> key = Botan::hex_decode(
↪"000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F");
   std::vector<uint8_t> block = Botan::hex_decode("00112233445566778899AABBCCDDEEFF");
   std::unique_ptr<Botan::BlockCipher> cipher(Botan::BlockCipher::create("AES-256"));
   cipher->set_key(key);
   cipher->encrypt(block);
```

(continues on next page)

(continued from previous page)

```
   std::cout << std::endl <<cipher->name() << "single block encrypt: " << Botan::hex_
↪encode(block);

   //clear cipher for 2nd encryption with other key
   cipher->clear();
   key = Botan::hex_decode(
↪"1337133713371337133713371337133713371337133713371337133713371337");
   cipher->set_key(key);
   cipher->encrypt(block);

   std::cout << std::endl << cipher->name() << "single block encrypt: " << Botan::hex_
↪encode(block);
   return 0;
   }
```

9.2 Available Ciphers

Botan includes a number of block ciphers that are specific to particular countries, as well as a few that are included mostly due to their use in specific protocols such as PGP but not widely used elsewhere. The ciphers that seem best for new code are AES, Serpent, and Threefish-512.

Avoid any 64-bit cipher in new code. There are combinatoric issues that affect any 64-bit cipher that render it insecure when large amounts of data are processed.

9.2.1 AES

Comes in three variants, AES-128, AES-192, and AES-256.

The standard 128-bit block cipher. Many modern platforms offer hardware acceleration. However, on platforms without hardware support, AES implementations typically are vulnerable to side channel attacks.

If you are developing new code and have no particular opinion, pick AES.

Available if `BOTAN_HAS_AES` is defined.

9.2.2 ARIA

South Korean cipher used in industry there. No reason to use it otherwise.

Available if `BOTAN_HAS_ARIA` is defined.

9.2.3 Blowfish

A 64-bit cipher popular in the pre-AES era. Very slow key setup. Also used (with bcrypt) for password hashing.

Available if `BOTAN_HAS_BLOWFISH` is defined.

9.2.4 CAST-128

A 64-bit cipher, commonly used in OpenPGP.

Available if `BOTAN_HAS_CAST128` is defined.

9.2.5 CAST-256

A 128-bit cipher that was a contestent in the NIST AES competition. Rarely used, and now deprecated in Botan. Use AES or Serpent instead.

Available if `BOTAN_HAS_CAST256` is defined.

9.2.6 Camellia

Comes in three variants, Camellia-128, Camellia-192, and Camellia-256.

A Japanese design standardized by ISO, NESSIE and CRYPTREC. Somewhat common. Prefer AES or Serpent in new designs.

Available if `BOTAN_HAS_CAMELLIA` is defined.

9.2.7 Cascade

Creates a block cipher cascade, where each block is encrypted by two ciphers with independent keys. Useful if you're very paranoid. In practice any single good cipher (such as Serpent, SHACAL2, or AES-256) is more than sufficient.

Available if `BOTAN_HAS_CASCADE` is defined.

9.2.8 DES, 3DES, DESX

Originally designed by IBM and NSA in the 1970s. Very slow, but still common in some industries such as finance. Avoid in new code.

Available if `BOTAN_HAS_DES` is defined.

9.2.9 GOST-28147-89

A old 64-bit Russian cipher. Possible security issues. Avoid unless compatability is needed.

Available if `BOTAN_HAS_GOST_28147_89` is defined.

9.2.10 IDEA

An older but still unbroken 64-bit cipher with a 128-bit key. Somewhat common due to its use in PGP. Avoid in new designs.

Available if `BOTAN_HAS_IDEA` is defined.

9.2.11 Kasumi

A 64-bit cipher used in 3GPP mobile phone protocols. There is no reason to use it outside of this context.

Available if `BOTAN_HAS_KASUMI` is defined.

9.2.12 Lion

A "block cipher construction" which can encrypt blocks of nearly arbitrary length. Built from a stream cipher and a hash function. Useful in certain protocols where being able to encrypt large or arbitrary length blocks is necessary.

Available if `BOTAN_HAS_LION` is defined.

9.2.13 MISTY1

A 64-bit Japanese cipher standardized by NESSIE and ISO. Seemingly secure, but quite slow and saw little adoption. No reason to use it in new code. The implementation in Botan is deprecated, and it is likely to be removed in a future release.

Available if `BOTAN_HAS_MISTY1` is defined.

9.2.14 Noekeon

A fast 128-bit cipher by the designers of AES. Easily secured against side channels.

Available if `BOTAN_HAS_NOEKEON` is defined.

9.2.15 SEED

A older South Korean cipher, widely used in industry there.

Available if `BOTAN_HAS_SEED` is defined.

9.2.16 SHACAL2

The 256-bit block cipher used inside SHA-256. Accepts up to a 512-bit key. Fast and seemingly very secure, but obscure. Standardized by NESSIE.

Available if `BOTAN_HAS_SHACAL2` is defined.

9.2.17 SM4

A 128-bit Chinese national cipher, required for use in certain commercial applications in China. Quite slow. Probably no reason to use it outside of legal requirements.

Available if `BOTAN_HAS_SM4` is defined.

9.2.18 Serpent

An AES contender. Widely considered the most conservative design. Fairly slow, especially if no SIMD instruction set is available.

Available if `BOTAN_HAS_SERPENT` is defined.

9.2.19 Threefish-512

A 512-bit tweakable block cipher that was used in the Skein hash function. Very fast on 64-bit processors.

Available if `BOTAN_HAS_THREEFISH_512` is defined.

9.2.20 Twofish

An AES contender. Somewhat complicated key setup and a “kitchen sink” design.

Available if `BOTAN_HAS_TWOFISH` is defined.

9.2.21 XTEA

A 64-bit cipher popular for its simple implementation. Avoid in new code.

Available if `BOTAN_HAS_XTEA` is defined.

CHAPTER

TEN

STREAM CIPHERS

In contrast to block ciphers, stream ciphers operate on a plaintext stream instead of blocks. Thus encrypting data results in changing the internal state of the cipher and encryption of plaintext with arbitrary length is possible in one go (in byte amounts). All implemented stream ciphers derive from the base class `StreamCipher` (*botan/stream_cipher.h*).

Note that some of the implemented stream ciphers require a fresh initialisation vector.

`class StreamCipher`

std::string **`name`**`()` **`const`**
: Returns a human-readable string of the name of this algorithm.

void **`clear`**`()`
: Clear the key.

StreamCipher ***`clone`**`()` **`const`**
: Return a newly allocated object of the same type as this one.

void **`set_key`**`(`**`const`** uint8_t **key*, size_t *length*`)`
: Set the stream cipher key. If the length is not accepted, an `Invalid_Key_Length` exception is thrown.

bool **`valid_keylength`**`(`size_t *length*`)` **`const`**
: This function returns true if and only if *length* is a valid keylength for the algorithm.

size_t **`minimum_keylength`**`()` **`const`**
: Return the smallest key length (in bytes) that is acceptible for the algorithm.

size_t **`maximum_keylength`**`()` **`const`**
: Return the largest key length (in bytes) that is acceptible for the algorithm.

bool **`valid_iv_length`**`(`size_t *iv_len*`)` **`const`**
: This function returns true if and only if *length* is a valid IV length for the stream cipher. Some ciphers do not support IVs at all, and will return false for any value except zero.

void **`set_iv`**`(`**`const`** uint8_t *, size_t *len*`)`
: Load IV into the stream cipher state. This should happen after the key is set and before any operation (encrypt/decrypt/seek) is called.

void **`seek`**`(`uint64_t *offset*`)`
: Sets the state of the stream cipher and keystream according to the passed *offset*. Therefore the key and the IV (if required) have to be set beforehand.

void **`cipher`**`(`**`const`** uint8_t **in*, uint8_t **out*, size_t *n*`)`
: Processes *n* bytes plain/ciphertext from *in* and writes the result to *out*.

void **`cipher1`**`(`uint8_t **inout*, size_t *n*`)`
: Processes *n* bytes plain/ciphertext in place. Acts like `cipher`(inout, inout, n).

void **encipher** (std::vector<uint8_t> *inout*)

void **encrypt** (std::vector<uint8_t> *inout*)

void **decrypt** (std::vector<uint8_t> *inout*)
Processes plain/ciphertext *inout* in place. Acts like `cipher`(inout.data(), inout.data(), inout.size()).

10.1 Code Example

The following code encrypts a provided plaintext using ChaCha20.

```
#include <botan/stream_cipher.h>
#include <botan/auto_rng.h>
#include <botan/hex.h>
#include <iostream>

int main()
   {
   std::string plaintext("This is a tasty burger!");
   std::vector<uint8_t> pt(plaintext.data(),plaintext.data()+plaintext.length());
   const std::vector<uint8_t> key = Botan::hex_decode(
↪"000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F");
   std::unique_ptr<Botan::StreamCipher> cipher(Botan::StreamCipher::create("ChaCha(20)
↪"));

   //generate fresh nonce (IV)
   std::unique_ptr<Botan::RandomNumberGenerator> rng(new Botan::AutoSeeded_RNG);
   std::vector<uint8_t> iv(8);
   rng->randomize(iv.data(),iv.size());

   //set key and IV
   cipher->set_key(key);
   cipher->set_iv(iv.data(),iv.size());
   cipher->encipher(pt);

   std::cout << cipher->name() << " with iv " << Botan::hex_encode(iv) << ": "
             << Botan::hex_encode(pt) << "\n";
   return 0;
   }
```

10.2 Available Stream Ciphers

Botan provides the following stream ciphers. If in doubt use CTR or ChaCha.

10.2.1 CTR-BE

A cipher mode that converts a block cipher into a stream cipher. It offers parallel execution and can seek within the output stream.

(The `-BE` suffix refers to big-endian convention for the counter. This is the most common case.)

10.2.2 OFB

Another stream cipher based on a block cipher. Unlike CTR mode, it does not allow parallel execution or seeking within the output stream. Prefer CTR.

Available if `BOTAN_HAS_OFB` is defined.

10.2.3 ChaCha

A very fast cipher, now widely deployed in TLS as part of the ChaCha20Poly1305 AEAD. Can be used with 8 (fast but dangerous), 12 (balance), or 20 rounds (conservative). Even with 20 rounds, ChaCha is very fast. Use 20 rounds.

Available if `BOTAN_HAS_CHACHA` is defined.

10.2.4 Salsa20

An earlier iteration of the ChaCha design, this cipher is popular due to its use in the libsodium library. Prefer ChaCha.

Note: The 'XSalsa20' variant of Salsa20 is also supported by the same class; this is selected by using a 192-bit nonce instead of Salsa20's 64-bit nonce.

Available if `BOTAN_HAS_SALSA20` is defined.

10.2.5 SHAKE-128

This is the SHAKE-128 XOF exposed as a stream cipher. It is slower than ChaCha and somewhat obscure.

Available if `BOTAN_HAS_SHAKE_CIPHER` is defined.

10.2.6 RC4

An old and very widely deployed stream cipher notable for its simplicity. Now broken. **Avoid in new code**

Available if `BOTAN_HAS_RC4` is defined.

CHAPTER ELEVEN

MESSAGE AUTHENTICATION CODES (MAC)

A Message Authentication Code algorithm computes a tag over a message utilizing a shared secret key. Thus a valid tag confirms the authenticity and integrity of the message. Only entities in possession of the shared secret key are able to verify the tag.

Note: When combining a MAC with unauthenticated encryption mode, prefer to first encrypt the message and then MAC the ciphertext. The alternative is to MAC the plaintext, which depending on exact usage can suffer serious security issues. For a detailed discussion of this issue see the paper "The Order of Encryption and Authentication for Protecting Communications" by Hugo Krawczyk

The Botan MAC computation is split into five stages.

1. Instantiate the MAC algorithm.
2. Set the secret key.
3. Process IV.
4. Process data.
5. Finalize the MAC computation.

class MessageAuthenticationCode

std::string **name**() **const**
Returns a human-readable string of the name of this algorithm.

void **clear**()
Clear the key.

MessageAuthenticationCode ***clone**() **const**
Return a newly allocated object of the same type as this one.

void **set_key**(**const** uint8_t **key*, size_t *length*)
Set the shared MAC key for the calculation. This function has to be called before the data is processed.

bool **valid_keylength**(size_t *length*) **const**
This function returns true if and only if *length* is a valid keylength for the algorithm.

size_t **minimum_keylength**() **const**
Return the smallest key length (in bytes) that is acceptible for the algorithm.

size_t **maximum_keylength**() **const**
Return the largest key length (in bytes) that is acceptible for the algorithm.

void **start**(**const** uint8_t **nonce*, size_t *nonce_len*)
Set the IV for the MAC calculation. Note that not all MAC algorithms require an IV. If an IV is required,

the function has to be called before the data is processed. For algorithms that don't require it, the call can be omitted, or else called with `nonce_len` of zero.

void **update** (**const** uint8_t **input*, size_t *length*)
Process the passed data.

void **update** (**const** secure_vector<uint8_t> *&in*)
Process the passed data.

void **update** (uint8_t *in*)
Process a single byte.

void **final** (uint8_t **out*)
Complete the MAC computation and write the calculated tag to the passed byte array.

secure_vector<uint8_t> **final** ()
Complete the MAC computation and return the calculated tag.

bool **verify_mac** (**const** uint8_t **mac*, size_t *length*)
Finalize the current MAC computation and compare the result to the passed `mac`. Returns `true`, if the verification is successful and false otherwise.

11.1 Code Example

The following example code computes a AES-256 GMAC and subsequently verifies the tag.

```
#include <botan/mac.h>
#include <botan/hex.h>
#include <iostream>

int main()
   {
   const std::vector<uint8_t> key = Botan::hex_decode(
↪"1337133713371337133713371337133713371337133713371337133713371337");
   const std::vector<uint8_t> iv = Botan::hex_decode("FFFFFFFFFFFFFFFFFFFFFFFF");
   const std::vector<uint8_t> data = Botan::hex_decode(
↪"6BC1BEE22E409F96E93D7E117393172A");
   std::unique_ptr<Botan::MessageAuthenticationCode>␣
↪mac(Botan::MessageAuthenticationCode::create("GMAC(AES-256)"));
   if(!mac)
      return 1;
   mac->set_key(key);
   mac->start(iv);
   mac->update(data);
   Botan::secure_vector<uint8_t> tag = mac->final();
   std::cout << mac->name() << ": " << Botan::hex_encode(tag) << std::endl;

   //Verify created MAC
   mac->start(iv);
   mac->update(data);
   std::cout << "Verification: " << (mac->verify_mac(tag) ? "success" : "failure");
   return 0;
   }
```

The following example code computes a valid AES-128 CMAC tag and modifies the data to demonstrate a MAC verification failure.

```
#include <botan/mac.h>
#include <botan/hex.h>
#include <iostream>

  int main()
     {
     const std::vector<uint8_t> key = Botan::hex_decode(
↪"2B7E151628AED2A6ABF7158809CF4F3C");
     std::vector<uint8_t> data = Botan::hex_decode("6BC1BEE22E409F96E93D7E117393172A
↪");
     std::unique_ptr<Botan::MessageAuthenticationCode>␣
↪mac(Botan::MessageAuthenticationCode::create("CMAC(AES-128)"));
     if(!mac)
        return 1;
     mac->set_key(key);
     mac->update(data);
     Botan::secure_vector<uint8_t> tag = mac->final();
     //Corrupting data
     data.back()++;
     //Verify with corrupted data
     mac->update(data);
     std::cout << "Verification with malformed data: " << (mac->verify_mac(tag) ?
↪"success" : "failure");
     return 0;
     }
```

11.2 Available MACs

Currently the following MAC algorithms are available in Botan. In new code, default to HMAC or CMAC.

11.2.1 CBC-MAC

An older authentication code based on a block cipher. Serious security problems, in particular **insecure** if messages of several different lengths are authenticated. Avoid unless required for compatability.

Available if `BOTAN_HAS_CBC_MAC` is defined.

11.2.2 CMAC

A modern CBC-MAC variant that avoids the security problems of plain CBC-MAC. Approved by NIST. Also sometimes called OMAC.

Available if `BOTAN_HAS_CMAC` is defined.

11.2.3 GMAC

GMAC is related to the GCM authenticated cipher mode. It is quite slow unless hardware support for carryless multiplications is available. A new nonce must be used with each message authenticated, or otherwise all security is lost.

Available if `BOTAN_HAS_GMAC` is defined.

11.2.4 HMAC

A message authentication code based on a hash function. Very commonly used.

Available if `BOTAN_HAS_HMAC` is defined.

11.2.5 Poly1305

A polynomial mac (similar to GMAC). Very fast, but tricky to use safely. Forms part of the ChaCha20Poly1305 AEAD mode. A new key must be used for *each* message, or all security is lost.

Available if `BOTAN_HAS_POLY1305` is defined.

11.2.6 SipHash

A modern and very fast PRF. Produces only a 64-bit output.

Available if `BOTAN_HAS_SIPHASH` is defined.

11.2.7 X9.19-MAC

A CBC-MAC variant sometimes used in finance. Always uses DES. Avoid unless required.

Available if `BOTAN_HAS_X919_MAC` is defined.

CHAPTER

TWELVE

CIPHER MODES

A block cipher by itself, is only able to securely encrypt a single data block. To be able to securely encrypt data of arbitrary length, a mode of operation applies the block cipher's single block operation repeatedly to encrypt an entire message.

All cipher mode implementations are are derived from the base class *`Cipher_Mode`*, which is declared in `botan/cipher_mode.h`.

> **Warning:** Using an unauthenticted cipher mode without combining it with a *Message Authentication Codes (MAC)* is insecure. Prefer using an *AEAD Mode*.

`class Cipher_Mode`

void **`set_key`**(**`const`** uint8_t **key*, size_t *length*)

Set the symmetric key to be used.

bool **`valid_keylength`**(size_t *length*) **`const`**

This function returns true if and only if *length* is a valid keylength for the algorithm.

size_t **`minimum_keylength`**() **`const`**

Return the smallest key length (in bytes) that is acceptible for the algorithm.

size_t **`maximum_keylength`**() **`const`**

Return the largest key length (in bytes) that is acceptible for the algorithm.

void **`start_msg`**(**`const`** uint8_t **nonce*, size_t *nonce_len*)

Set the IV (unique per-message nonce) of the mode of operation and prepare for message processing.

void **`start`**(**`const`** std::vector<uint8_t> *nonce*)

Acts like *`start_msg`*(nonce.data(), nonce.size()).

void **`start`**(**`const`** uint8_t **nonce*, size_t *nonce_len*)

Acts like *`start_msg`*(nonce, nonce_len).

`virtual` size_t **`update_granularity`**() **`const`**

The *`Cipher_Mode`* interface requires message processing in multiples of the block size. Returns size of required blocks to update and 1, if the mode can process messages of any length.

`virtual` size_t **`process`**(uint8_t **msg*, size_t *msg_len*)

Process msg in place and returns bytes written. msg must be a multiple of *`update_granularity`*.

void **`update`**(secure_vector<uint8_t> &*buffer*, size_t *offset* = 0)

Continue processing a message in the buffer in place. The passed buffer's size must be a multiple of *`update_granularity`*. The first *offset* bytes of the buffer will be ignored.

size_t **minimum_final_size**() **const**

Returns the minimum size needed for `finish`.

void **finish**(secure_vector<uint8_t> &*final_block*, size_t *offset* = 0)

Finalize the message processing with a final block of at least `minimum_final_size` size. The first *offset* bytes of the passed final block will be ignored.

12.1 Code Example

The following code encrypts the specified plaintext using AES-128/CBC with PKCS#7 padding.

Warning: This example ignores the requirement to authenticate the ciphertext

```
#include <botan/rng.h>
#include <botan/auto_rng.h>
#include <botan/cipher_mode.h>
#include <botan/hex.h>
#include <iostream>

int main()
   {
   Botan::AutoSeeded_RNG rng;

   const std::string plaintext("Your great-grandfather gave this watch to your␣
↪granddad for good luck. Unfortunately, Dane's luck wasn't as good as his old man's.
↪");
   const std::vector<uint8_t> key = Botan::hex_decode(
↪"2B7E151628AED2A6ABF7158809CF4F3C");

   std::unique_ptr<Botan::Cipher_Mode> enc(Botan::get_cipher_mode("AES-128/CBC/PKCS7",
↪ Botan::ENCRYPTION));
   enc->set_key(key);

   Botan::secure_vector<uint8_t> pt(plaintext.data(), plaintext.data()+plaintext.
↪length());

   //generate fresh nonce (IV)
   enc->start(rng.random_vec(enc->default_nonce_length()));
   enc->finish(pt);

   std::cout << enc->name() << " with iv " << Botan::hex_encode(iv) << " " <<␣
↪Botan::hex_encode(pt) << "\n";
   return 0;
   }
```

12.2 Available Unauthenticated Cipher Modes

Note: CTR and OFB modes are also implemented, but these are treated as `Stream_Ciphers` instead.

12.2.1 CBC

Available if `BOTAN_HAS_MODE_CBC` is defined.

CBC requires the plaintext be padded using a reversible rule. The following padding schemes are implemented

PKCS#7 (RFC5652) The last byte in the padded block defines the padding length p, the remaining padding bytes are set to p as well.

ANSI X9.23 The last byte in the padded block defines the padding length, the remaining padding is filled with 0x00.

OneAndZeros (ISO/IEC 7816-4) The first padding byte is set to 0x80, the remaining padding bytes are set to 0x00.

12.2.2 CFB

Available if `BOTAN_HAS_MODE_CFB` is defined.

CFB uses a block cipher to create a self-syncronizing stream cipher. It is used for example in the OpenPGP protocol. There is no reason to prefer it.

12.2.3 XTS

Available if `BOTAN_HAS_MODE_XTS` is defined.

XTS is a mode specialized for encrypting disk storage. XTS requires all inputs be at least 1 byte longer than the native block size of the cipher.

12.3 AEAD Mode

AEAD (Authenticated Encryption with Associated Data) modes provide message encryption, message authentication, and the ability to authenticate additional data that is not included in the ciphertext (such as a sequence number or header). It is a subclass of *`Cipher_Mode`*.

The AEAD interface can be used directly, or as part of the filter system by using `AEAD_Filter` (a subclass of `Keyed_Filter` which will be returned by *`get_cipher`* if the named cipher is an AEAD mode).

`class AEAD_Mode`

void **`set_key`**(**`const`** SymmetricKey &*key*)

Set the key

Key_Length_Specification **`key_spec`**() **`const`**

Return the key length specification

void **`set_associated_data`**(**`const`** uint8_t *ad*[], size_t *ad_len*)

Set any associated data for this message. For maximum portability between different modes, this must be called after *`set_key`* and before *`start`*.

If the associated data does not change, it is not necessary to call this function more than once, even across multiple calls to *`start`* and *`finish`*.

void **`start`**(**`const`** uint8_t *nonce*[], size_t *nonce_len*)

Start processing a message, using *nonce* as the unique per-message value.

> **Warning:** With most AEADs, if the same nonce is ever used to encrypt two different messages under the same key, all security is lost.

void **update** (secure_vector<uint8_t> &*buffer*, size_t *offset* = 0)

Continue processing a message. The *buffer* is an in/out parameter and may be resized. In particular, some modes require that all input be consumed before any output is produced; with these modes, *buffer* will be returned empty.

On input, the buffer must be sized in blocks of size `update_granularity`. For instance if the update granularity was 64, then *buffer* could be 64, 128, 192, . . . bytes.

The first *offset* bytes of *buffer* will be ignored (this allows in place processing of a buffer that contains an initial plaintext header)

void **finish** (secure_vector<uint8_t> &*buffer*, size_t *offset* = 0)

Complete processing a message with a final input of *buffer*, which is treated the same as with `update`. It must contain at least `final_minimum_size` bytes.

Note that if you have the entire message in hand, calling finish without ever calling update is both efficient and convenient.

> **Note:** During decryption, finish will throw an instance of Integrity_Failure if the MAC does not validate. If this occurs, all plaintext previously output via calls to update must be destroyed and not used in any way that an attacker could observe the effects of.
>
> One simply way to assure this could never happen is to never call update, and instead always marshall the entire message into a single buffer and call finish on it when decrypting.

size_t **update_granularity** () **const**

The AEAD interface requires `update` be called with blocks of this size. This will be 1, if the mode can process any length inputs.

size_t **final_minimum_size** () **const**

The AEAD interface requires `finish` be called with at least this many bytes (which may be zero, or greater than `update_granularity`)

bool **valid_nonce_length** (size_t *nonce_len*) **const**

Returns true if *nonce_len* is a valid nonce length for this scheme. For EAX and GCM, any length nonces are allowed. OCB allows any value between 8 and 15 bytes.

size_t **default_nonce_length** () **const**

Returns a reasonable length for the nonce, typically either 96 bits, or the only supported length for modes which don't support 96 bit nonces.

12.4 Available AEAD Modes

12.4.1 ChaCha20Poly1305

Available if `BOTAN_HAS_AEAD_CHACHA20_POLY1305` is defined.

Unlike the other AEADs which are based on block ciphers, this mode is based on the ChaCha stream cipher and the Poly1305 authentication code. It is very fast on all modern platforms.

12.4.2 GCM

Available if `BOTAN_HAS_AEAD_GCM` is defined.

NIST standard, commonly used. Requires a 128-bit block cipher. Fairly slow, unless hardware support for carryless multiplies is available.

12.4.3 OCB

Available if `BOTAN_HAS_AEAD_OCB` is defined.

A block cipher based AEAD. Supports 128-bit, 256-bit and 512-bit block ciphers. This mode is very fast and easily secured against side channels. Adoption has been poor because it is patented in the United States, though a license is available allowing it to be freely used by open source software.

12.4.4 EAX

Available if `BOTAN_HAS_AEAD_EAX` is defined.

A secure composition of CTR mode and CMAC. Supports 128-bit, 256-bit and 512-bit block ciphers.

12.4.5 SIV

Available if `BOTAN_HAS_AEAD_SIV` is defined.

Requires a 128-bit block cipher. Unlike other AEADs, SIV is “misuse resistent”; if a nonce is repeated, SIV retains security, with the exception that if the same nonce is used to encrypt the same message multiple times, an attacker can detect the duplicated message (this is because for identical plaintexts SIV will output the same ciphertext each time, in the case the nonce is repeated.)

12.4.6 CCM

Available if `BOTAN_HAS_AEAD_CCM` is defined.

Requires a 128-bit block cipher. This is a NIST standard mode but that is about all to recommenmd it. Prefer EAX.

CHAPTER

THIRTEEN

PUBLIC KEY CRYPTOGRAPHY

Public key cryptography (also called asymmetric cryptography) is a collection of techniques allowing for encryption, signatures, and key agreement.

13.1 Key Objects

Public and private keys are represented by classes `Public_Key` and it's subclass `Private_Key`. The use of inheritence here means that a `Private_Key` can be converted into a reference to a public key.

None of the functions on `Public_Key` and `Private_Key` itself are particularly useful for users of the library, because 'bare' public key operations are *very insecure*. The only purpose of these functions is to provide a clean interface that higher level operations can be built on. So really the only thing you need to know is that when a function takes a reference to a `Public_Key`, it can take any public key or private key, and similiarly for `Private_Key`.

Types of `Public_Key` include `RSA_PublicKey`, `DSA_PublicKey`, `ECDSA_PublicKey`, `ECKCDSA_PublicKey`, `ECGDSA_PublicKey`, `DH_PublicKey`, `ECDH_PublicKey`, `Curve25519_PublicKey`, `ElGamal_PublicKey`, `McEliece_PublicKey`, `XMSS_PublicKey` and `GOST_3410_PublicKey`. There are cooresponding `Private_Key` classes for each of these algorithms.

13.2 Creating New Private Keys

Creating a new private key requires two things: a source of random numbers (see *Random Number Generators*) and some algorithm specific parameters that define the *security level* of the resulting key. For instance, the security level of an RSA key is (at least in part) defined by the length of the public key modulus in bits. So to create a new RSA private key, you would call

`RSA_PrivateKey``::`**`RSA_PrivateKey`**(RandomNumberGenerator &*rng*, size_t *bits*)

A constructor that creates a new random RSA private key with a modulus of length *bits*.

Algorithms based on the discrete-logarithm problem use what is called a *group*; a group can safely be used with many keys, and for some operations, like key agreement, the two keys *must* use the same group. There are currently two kinds of discrete logarithm groups supported in botan: the integers modulo a prime, represented by *DL_Group*, and elliptic curves in GF(p), represented by *EC_Group*. A rough generalization is that the larger the group is, the more secure the algorithm is, but correspondingly the slower the operations will be.

Given a `DL_Group`, you can create new DSA, Diffie-Hellman and ElGamal key pairs with

`DSA_PrivateKey``::`**`DSA_PrivateKey`**(RandomNumberGenerator &*rng*, **`const`** DL_Group &*group*, **`const`** BigInt &*x* = 0)

`DH_PrivateKey``::`**`DH_PrivateKey`**(RandomNumberGenerator &*rng*, **`const`** DL_Group &*group*, **`const`** BigInt &*x* = 0)

ElGamal_PrivateKey::**ElGamal_PrivateKey**(RandomNumberGenerator &*rng*, **const** DL_Group &*group*, **const** BigInt &*x* = 0)

The optional *x* parameter to each of these constructors is a private key value. This allows you to create keys where the private key is formed by some special technique; for instance you can use the hash of a password (see *PBKDF Algorithms* for how to do that) as a private key value. Normally, you would leave the value as zero, letting the class generate a new random key.

Finally, given an `EC_Group` object, you can create a new ECDSA, ECKCDSA, ECGDSA, ECDH, or GOST 34.10-2001 private key with

ECDSA_PrivateKey::**ECDSA_PrivateKey**(RandomNumberGenerator &*rng*, **const** EC_Group &*domain*, **const** BigInt &*x* = 0)

ECKCDSA_PrivateKey::**ECKCDSA_PrivateKey**(RandomNumberGenerator &*rng*, **const** EC_Group &*domain*, **const** BigInt &*x* = 0)

ECGDSA_PrivateKey::**ECGDSA_PrivateKey**(RandomNumberGenerator &*rng*, **const** EC_Group &*domain*, **const** BigInt &*x* = 0)

ECDH_PrivateKey::**ECDH_PrivateKey**(RandomNumberGenerator &*rng*, **const** EC_Group &*domain*, **const** BigInt &*x* = 0)

GOST_3410_PrivateKey::**GOST_3410_PrivateKey**(RandomNumberGenerator &*rng*, **const** EC_Group &*domain*, **const** BigInt &*x* = 0)

13.3 Serializing Private Keys Using PKCS #8

The standard format for serializing a private key is PKCS #8, the operations for which are defined in `pkcs8.h`. It supports both unencrypted and encrypted storage.

secure_vector<uint8_t> PKCS8::**BER_encode**(**const** Private_Key &*key*, RandomNumberGenerator &*rng*, **const** std::string &*password*, **const** std::string &*pbe_algo* = "")

Takes any private key object, serializes it, encrypts it using *password*, and returns a binary structure representing the private key.

The final (optional) argument, *pbe_algo*, specifies a particular password based encryption (or PBE) algorithm. If you don't specify a PBE, a sensible default will be used.

std::string PKCS8::**PEM_encode**(**const** Private_Key &*key*, RandomNumberGenerator &*rng*, **const** std::string &*pass*, **const** std::string &*pbe_algo* = "")

This formats the key in the same manner as `BER_encode`, but additionally encodes it into a text format with identifying headers. Using PEM encoding 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.

Unencrypted serialization is also supported.

> **Warning:** In most situations, using unecrypted private key storage 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, described above.

secure_vector<uint8_t> PKCS8::**BER_encode**(**const** Private_Key &*key*)

Serializes the private key and returns the result.

std::string PKCS8::**PEM_encode**(**const** Private_Key &*key*)

Serializes the private key, base64 encodes it, and returns the result.

Last but not least, there are some functions that will load (and decrypt, if necessary) a PKCS #8 private key:

Private_Key *`PKCS8::`**`load_key`**(DataSource *&in*, RandomNumberGenerator *&rng*, **`const`** User_Interface *&ui*)

Private_Key *`PKCS8::`**`load_key`**(DataSource *&in*, RandomNumberGenerator *&rng*, std::string *passphrase* = "")

Private_Key *`PKCS8::`**`load_key`**(**`const`** std::string *&filename*, RandomNumberGenerator *&rng*, **`const`** User_Interface *&ui*)

Private_Key *`PKCS8::`**`load_key`**(**`const`** std::string *&filename*, RandomNumberGenerator *&rng*, **`const`** std::string *&passphrase* = "")

These functions will return an object allocated key object based on the data from whatever source it is using (assuming, of course, the source is in fact storing a representation of a private key, and the decryption was successful). 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 create - the other is a simple wrapper functions that take either a filename or a memory buffer and create the appropriate `DataSource`.

The versions taking a `std::string` attempt to decrypt using the password given (if the key is encrypted; if it is not, the passphase value will be ignored). If the passphrase does not decrypt the key, an exception will be thrown.

The ones taking a `User_Interface` provide a simple callback interface which makes handling incorrect passphrases and such a bit simpler. 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.

Note: In a future version, it is likely that `User_Interface` will be replaced by a simple callback using `std::function`.

To use `User_Interface`, derive a subclass and implement:

std::string `User_Interface::`**`get_passphrase`**(**`const`** std::string *&what*, **`const`** 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.

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).

13.3.1 Serializing Public Keys

To import and export public keys, use:

std::vector<uint8_t> `X509::`**`BER_encode`**(**`const`** Public_Key *&key*)

std::string `X509::`**`PEM_encode`**(**`const`** Public_Key *&key*)

Public_Key *`X509::`**`load_key`**(DataSource *&in*)

Public_Key *`X509::`**`load_key`**(**`const`** secure_vector<uint8_t> *&buffer*)

Public_Key *X509::**load_key**(**const** std::string &*filename*)

These functions operate in the same way as the ones described in *Serializing Private Keys Using PKCS #8*, except that no encryption option is availabe.

13.3.2 DL_Group

As described in *Creating New Private Keys*, a discrete logarithm group can be shared among many keys, even keys created by users who do not trust each other. However, it is necessary to trust the entity who created the group; that is why organization like NIST use algorithms which generate groups in a deterministic way such that creating a bogus group would require breaking some trusted cryptographic primitive like SHA-2.

Instantiating a `DL_Group` simply requires calling

DL_Group::**DL_Group**(**const** std::string &*name*)

The *name* parameter is a specially formatted string that consists of three things, the type of the group ("modp" or "dsa"), the creator of the group, and the size of the group in bits, all delimited by '/' characters.

Currently all "modp" groups included in botan are ones defined by the Internet Engineering Task Force, so the provider is "ietf", and the strings look like "modp/ietf/N" where N can be any of 1024, 1536, 2048, 3072, 4096, 6144, or 8192. This group type is used for Diffie-Hellman and ElGamal algorithms.

The other type, "dsa" is used for DSA keys. They can also be used with Diffie-Hellman and ElGamal, but this is less common. The currently available groups are "dsa/jce/1024" and "dsa/botan/N" with N being 2048 or 3072. The "jce" groups are the standard DSA groups used in the Java Cryptography Extensions, while the "botan" groups were randomly generated using the FIPS 186-3 algorithm by the library maintainers.

You can generate a new random group using

DL_Group::**DL_Group**(RandomNumberGenerator &*rng*, PrimeType *type*, size_t *pbits*, size_t *qbits* = 0)

The *type* can be either `Strong`, `Prime_Subgroup`, or `DSA_Kosherizer`. *pbits* specifies the size of the prime in bits. If the *type* is `Prime_Subgroup` or `DSA_Kosherizer`, then *qbits* specifies the size of the subgroup.

You can serialize a `DL_Group` using

secure_vector<uint8_t> DL_Group::**DER_Encode**(Format *format*)

or

std::string DL_Group::**PEM_encode**(Format *format*)

where *format* is any of

- `ANSI_X9_42` (or `DH_PARAMETERS`) for modp groups
- `ANSI_X9_57` (or `DSA_PARAMETERS`) for DSA-style groups
- `PKCS_3` is an older format for modp groups; it should only be used for backwards compatibility.

You can reload a serialized group using

void DL_Group::**BER_decode**(DataSource &*source*, Format *format*)

void DL_Group::**PEM_decode**(DataSource &*source*)

Code Example

The example below creates a new 2048 bit `DL_Group`, prints the generated parameters and ANSI_X9_42 encodes the created group for further usage with DH.

```
#include <botan/dl_group.h>
#include <botan/auto_rng.h>
#include <botan/rng.h>
#include <iostream>

int main()
   {
      std::unique_ptr<Botan::RandomNumberGenerator> rng(new Botan::AutoSeeded_RNG);
      std::unique_ptr<Botan::DL_Group> group(new Botan::DL_Group(*rng.get(),␣
↪Botan::DL_Group::Strong, 2048));
      std::cout << std::endl << "p: " << group->get_p();
      std::cout << std::endl << "q: " << group->get_q();
      std::cout << std::endl << "g: " << group->get_q();
      std::cout << std::endl << "ANSI_X9_42: " << std::endl << group->PEM_
↪encode(Botan::DL_Group::ANSI_X9_42);

    return 0;
   }
```

13.3.3 EC_Group

An EC_Group is initialized by passing the name of the group to be used to the constructor. These groups have semi-standardized names like "secp256r1" and "brainpool512r1".

13.4 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 public key type has a function

bool Public_Key::**check_key**(RandomNumberGenerator &*rng*, bool *strong*)

This function performs a number of algorithm-specific tests that the key seems to be mathematically valid and consistent, and returns true if all of the tests pass.

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 *strong* is true, then it does "strong" checking, which includes expensive operations like primality checking.

As key checks are not automatically performed they must be called manually after loading keys from untrusted sources. If a key from an untrusted source is not checked, the implementation might be vulnerable to algorithm specific attacks.

The following example loads the Subject Public Key from the x509 certificate cert.pem and checks the loaded key. If the key check fails a respective error is thrown.

```
#include <botan/x509cert.h>
#include <botan/auto_rng.h>
#include <botan/rng.h>

int main()
   {
   Botan::X509_Certificate cert("cert.pem");
   std::unique_ptr<Botan::RandomNumberGenerator> rng(new Botan::AutoSeeded_RNG);
   std::unique_ptr<Botan::Public_Key> key(cert.subject_public_key());
   if(!key->check_key(*rng.get(), false))
```

(continues on next page)

(continued from previous page)

```
      {
      throw std::invalid_argument("Loaded key is invalid");
      }
   }
```

13.5 Encryption

Safe public key encryption requires the use of a padding scheme which hides the underlying mathematical properties of the algorithm. Additionally, they will add randomness, so encrypting the same plaintext twice produces two different ciphertexts.

The primary interface for encryption is

class PK_Encryptor

secure_vector<uint8_t> **encrypt**(**const** uint8_t *in, size_t *length*, RandomNumberGenerator &*rng*) **const**

secure_vector<uint8_t> **encrypt**(**const** std::vector<uint8_t> &*in*, RandomNumberGenerator &*rng*) **const**

These encrypt a message, returning the ciphertext.

size_t **maximum_input_size**() **const**

Returns the maximum size of the message that can be processed, in bytes. If you call *PK_Encryptor::encrypt* with a value larger than this the operation will fail with an exception.

PK_Encryptor is only an interface - to actually encrypt you have to create an implementation, of which there are currently three available in the library, *PK_Encryptor_EME*, *DLIES_Encryptor* and *ECIES_Encryptor*. DLIES is a hybrid encryption scheme (from IEEE 1363) that uses the DH key agreement technique in combination with a KDF, a MAC and a symmetric encryption algorithm to perform message encryption. ECIES is similar to DLIES, but uses ECDH for the key agreement. Normally, public key encryption is done using algorithms which support it directly, such as RSA or ElGamal; these use the EME class:

class PK_Encryptor_EME

PK_Encryptor_EME(**const** Public_Key &*key*, std::string *eme*)

With *key* being the key you want to encrypt messages to. The padding method to use is specified in *eme*.

The recommended values for *eme* is “EME1(SHA-1)” or “EME1(SHA-256)”. If you need compatibility with protocols using the PKCS #1 v1.5 standard, you can also use “EME-PKCS1-v1_5”.

class DLIES_Encryptor

Available in the header dlies.h

DLIES_Encryptor(**const** DH_PrivateKey &*own_priv_key*, RandomNumberGenerator &*rng*, KDF *kdf, MessageAuthenticationCode *mac, size_t *mac_key_len* = 20)

Where *kdf* is a key derivation function (see *Key Derivation Functions*) and *mac* is a MessageAuthenticationCode. The encryption is performed by XORing the message with a stream of bytes provided by the KDF.

DLIES_Encryptor(**const** DH_PrivateKey &*own_priv_key*, RandomNumberGenerator &*rng*, KDF *kdf, Cipher_Mode *cipher, size_t *cipher_key_len*, MessageAuthenticationCode *mac, size_t *mac_key_len* = 20)

Instead of XORing the message a block cipher can be specified.

class ECIES_Encryptor

Available in the header `ecies.h`.

Parameters for encryption and decryption are set by the `ECIES_System_Params` class which stores the EC domain parameters, the KDF (see *Key Derivation Functions*), the cipher (see *Cipher Modes*) and the MAC.

ECIES_Encryptor(**const** PK_Key_Agreement_Key &*private_key*, **const** ECIES_System_Params &*ecies_params*, RandomNumberGenerator &*rng*)

Where *private_key* is the key to use for the key agreement. The system paramters are specified in *ecies_params* and the RNG to use is passed in *rng*.

ECIES_Encryptor(RandomNumberGenerator &*rng*, **const** ECIES_System_Params &*ecies_params*)

Creates an ephemeral private key which is used for the key agreement.

The decryption classes are named `PK_Decryptor`, `PK_Decryptor_EME`, `DLIES_Decryptor` and `ECIES_Decryptor`. They are created in the exact same way, except they take the private key, and the processing function is named `decrypt`.

Botan implements the following encryption algorithms and padding schemes:

1. **RSA**
   - "PKCS1v15" || "EME-PKCS1-v1_5"
   - "OAEP" || "EME-OAEP" || "EME1" || "EME1(SHA-1)" || "EME1(SHA-256)"
2. DLIES
3. ECIES
4. SM2

13.5.1 Code Example

The following Code sample reads a PKCS #8 keypair from the passed location and subsequently encrypts a fixed plaintext with the included public key, using EME1 with SHA-256. For the sake of completeness, the ciphertext is then decrypted using the private key.

```cpp
#include <botan/pkcs8.h>
#include <botan/hex.h>
#include <botan/pk_keys.h>
#include <botan/pubkey.h>
#include <botan/auto_rng.h>
#include <botan/rng.h>
#include <iostream>
int main (int argc, char* argv[])
  {
  if(argc!=2)
     return 1;
  std::string plaintext("Your great-grandfather gave this watch to your granddad for good luck. Unfortunately, Dane's luck wasn't as good as his old man's.");
  std::vector<uint8_t> pt(plaintext.data(),plaintext.data()+plaintext.length());
  std::unique_ptr<Botan::RandomNumberGenerator> rng(new Botan::AutoSeeded_RNG);

  //load keypair
  std::unique_ptr<Botan::Private_Key> kp(Botan::PKCS8::load_key(argv[1],*rng.get()));

  //encrypt with pk
  Botan::PK_Encryptor_EME enc(*kp,*rng.get(), "EME1(SHA-256)");
```

(continues on next page)

(continued from previous page)

```
std::vector<uint8_t> ct = enc.encrypt(pt,*rng.get());

//decrypt with sk
Botan::PK_Decryptor_EME dec(*kp,*rng.get(), "EME1(SHA-256)");
std::cout << std::endl << "enc: " << Botan::hex_encode(ct) << std::endl << "dec: "<
↪< Botan::hex_encode(dec.decrypt(ct));

return 0;
}
```

13.6 Signatures

Signature generation is performed using

class PK_Signer

PK_Signer(**const** Private_Key *&key*, **const** std::string *&emsa*, Signature_Format *format* = IEEE_1363)

Constructs a new signer object for the private key *key* using the signature format *emsa*. The key must support signature operations. In the current version of the library, this includes RSA, DSA, ECDSA, ECKCDSA, ECGDSA, GOST 34.10-2001. Other signature schemes may be supported in the future.

Note: Botan both supports non-deterministic and deterministic (as per RFC 6979) DSA and ECDSA signatures. Deterministic signatures are compatible in the way that they can be verified with a non-deterministic implementation. If the `rfc6979` module is enabled, deterministic DSA and ECDSA signatures will be generated.

Currently available values for *emsa* 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. The Raw encoding signs the input directly; if the message is too big, the signing operation will fail. Raw is not useful except in very specialized applications. Examples are “EMSA1(SHA-1)” and “EMSA4(SHA-256)”.

For RSA, use EMSA4 (also called PSS) unless you need compatibility with software that uses the older PKCS #1 v1.5 standard, in which case use EMSA3 (also called “EMSA-PKCS1-v1_5”). For DSA, ECDSA, ECKCDSA, ECGDSA and GOST 34.10-2001 you should use EMSA1.

The *format* defaults to `IEEE_1363` which is the only available format for RSA. For DSA, ECDSA, ECGDSA and ECKCDSA you can also use `DER_SEQUENCE`, which will format the signature as an ASN.1 SEQUENCE value.

void **update**(**const** uint8_t **in*, size_t *length*)

void **update**(**const** std::vector<uint8_t> *&in*)

void **update**(uint8_t *in*)

These add more data to be included in the signature computation. Typically, the input will be provided directly to a hash function.

secure_vector<uint8_t> **signature**(RandomNumberGenerator *&rng*)

Creates the signature and returns it

secure_vector<uint8_t> **sign_message**(**const** uint8_t **in*, size_t *length*, RandomNumberGenerator *&rng*)

secure_vector<uint8_t> **sign_message**(**const** std::vector<uint8_t> *&in*, RandomNumberGenerator *&rng*)

These functions are equivalent to calling *PK_Signer::update* and then *PK_Signer::signature*. Any data previously provided using `update` will be included.

Signatures are verified using

class PK_Verifier

PK_Verifier(**const** Public_Key *&pub_key*, **const** std::string *&emsa*, Signature_Format *format* = IEEE_1363)

Construct a new verifier for signatures assicated with public key *pub_key*. The *emsa* and *format* should be the same as that used by the signer.

void **update**(**const** uint8_t **in*, size_t *length*)

void **update**(**const** std::vector<uint8_t> *&in*)

void **update**(uint8_t *in*)

Add further message data that is purportedly assocated with the signature that will be checked.

bool **check_signature**(**const** uint8_t **sig*, size_t *length*)

bool **check_signature**(**const** std::vector<uint8_t> *&sig*)

Check to see if *sig* is a valid signature for the message data that was written in. Return true if so. This function clears the internal message state, so after this call you can call *PK_Verifier::update* to start verifying another message.

bool **verify_message**(**const** uint8_t **msg*, size_t *msg_length*, **const** uint8_t **sig*, size_t *sig_length*)

bool **verify_message**(**const** std::vector<uint8_t> *&msg*, **const** std::vector<uint8_t> *&sig*)

These are equivalent to calling *PK_Verifier::update* on *msg* and then calling *PK_Verifier::check_signature* on *sig*.

Botan implements the following signature algorithms:

1. RSA
2. DSA
3. ECDSA
4. ECGDSA
5. ECKDSA
6. GOST 34.10-2001
7. Ed25519
8. SM2

13.6.1 Code Example

The following sample program below demonstrates the generation of a new ECDSA keypair over the curve secp512r1 and a ECDSA signature using EMSA1 with SHA-256. Subsequently the computed signature is validated.

```
#include <botan/auto_rng.h>
#include <botan/ecdsa.h>
#include <botan/ec_group.h>
#include <botan/pubkey.h>
```

(continues on next page)

(continued from previous page)

```
#include <botan/hex.h>
#include <iostream>

int main()
  {
  Botan::AutoSeeded_RNG rng;
  // Generate ECDSA keypair
  Botan::ECDSA_PrivateKey key(rng, Botan::EC_Group("secp521r1"));

  std::string text("This is a tasty burger!");
  std::vector<uint8_t> data(text.data(),text.data()+text.length());
  // sign data
  Botan::PK_Signer signer(key, rng, "EMSA1(SHA-256)");
  signer.update(data);
  std::vector<uint8_t> signature = signer.signature(rng);
  std::cout << "Signature:" << std::endl << Botan::hex_encode(signature);
  // verify signature
  Botan::PK_Verifier verifier(key, "EMSA1(SHA-256)");
  verifier.update(data);
  std::cout << std::endl << "is " << (verifier.check_signature(signature)? "valid" :
↪"invalid");
  return 0;
  }
```

13.7 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 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 secure_vector<uint8_t> 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 KDF you're using, you *might not* get back a key of the size you requested. 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 secure_vector<uint8_t>.

"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.

Botan implements the following key agreement methods:

1. ECDH over GF(p) Weierstrass curves
2. ECDH over x25519

3. DH over prime fields
4. McEliece
5. NewHope

13.7.1 Code Example

The code below performs an unauthenticated ECDH key agreement using the secp521r elliptic curve and applies the key derivation function KDF2(SHA-256) with 256 bit output length to the computed shared secret.

```
#include <botan/auto_rng.h>
#include <botan/ecdh.h>
#include <botan/ec_group.h>
#include <botan/pubkey.h>
#include <botan/hex.h>
#include <iostream>

int main()
   {
   Botan::AutoSeeded_RNG rng
   // ec domain and
   Botan::EC_Group domain("secp521r1");
   std::string kdf = "KDF2(SHA-256)";
   // generate ECDH keys
   Botan::ECDH_PrivateKey keyA(rng, domain);
   Botan::ECDH_PrivateKey keyB(rng, domain);
   // Construct key agreements
   Botan::PK_Key_Agreement ecdhA(keyA,rng,kdf);
   Botan::PK_Key_Agreement ecdhB(keyB,rng,kdf);
   // Agree on shared secret and derive symmetric key of 256 bit length
   Botan::secure_vector<uint8_t> sA = ecdhA.derive_key(32,keyB.public_value()).bits_
↪of();
   Botan::secure_vector<uint8_t> sB = ecdhB.derive_key(32,keyA.public_value()).bits_
↪of();

   if(sA != sB)
      return 1;

   std::cout << "agreed key: " << std::endl << Botan::hex_encode(sA);
   return 0;
   }
```

13.8 McEliece

McEliece is a cryptographic scheme based on error correcting codes which is thought to be resistent to quantum computers. First proposed in 1978, it is fast and patent-free. Variants have been proposed and broken, but with suitable parameters the original scheme remains secure. However the public keys are quite large, which has hindered deployment in the past.

The implementation of McEliece in Botan was contributed by cryptosource GmbH. It is based on the implementation HyMES, with the kind permission of Nicolas Sendrier and INRIA to release a C++ adaption of their original C code under the Botan license. It was then modified by Falko Strenzke to add side channel and fault attack countermeasures. You can read more about the implementation at http://www.cryptosource.de/docs/mceliece_in_botan.pdf

Encryption in the McEliece scheme consists of choosing a message block of size *n*, encoding it in the error correcting code which is the public key, then adding *t* bit errors. The code is created such that knowing only the public key, decoding *t* errors is intractible, but with the additional knowledge of the secret structure of the code a fast decoding technique exists.

The McEliece implementation in HyMES, and also in Botan, uses an optimization to reduce the public key size, by converting the public key into a systemic code. This means a portion of the public key is a identity matrix, and can be excluded from the published public key. However it also means that in McEliece the plaintext is represented directly in the ciphertext, with only a small number of bit errors. Thus it is absolutely essential to only use McEliece with a CCA2 secure scheme.

One such scheme, KEM, is provided in Botan currently. It it a somewhat unusual scheme in that it outputs two values, a symmetric key for use with an AEAD, and an encrypted key. It does this by choosing a random plaintext (n - log2(n)*t bits) using `McEliece_PublicKey::random_plaintext_element`. Then a random error mask is chosen and the message is coded and masked. The symmetric key is SHA-512(plaintext || error_mask). As long as the resulting key is used with a secure AEAD scheme (which can be used for transporting arbitrary amounts of data), CCA2 security is provided.

In `mcies.h` there are functions for this combination:

secure_vector<uint8_t> **mceies_encrypt**(**const** McEliece_PublicKey &*pubkey*, **const** secure_vector<uint8_t> &*pt*, uint8_t *ad*[], size_t *ad_len*, RandomNumberGenerator &*rng*, **const** std::string &*aead* = "AES-256/OCB")

secure_vector<uint8_t> **mceies_decrypt**(**const** McEliece_PrivateKey &*privkey*, **const** secure_vector<uint8_t> &*ct*, uint8_t *ad*[], size_t *ad_len*, **const** std::string &*aead* = "AES-256/OCB")

For a given security level (SL) a McEliece key would use parameters n and t, and have the cooresponding key sizes listed:

SL	n	t	public key KB	private key KB
80	1632	33	59	140
107	2280	45	128	300
128	2960	57	195	459
147	3408	67	265	622
191	4624	95	516	1234
256	6624	115	942	2184

You can check the speed of McEliece with the suggested parameters above using `botan speed McEliece`

13.9 eXtended Merkle Signature Scheme (XMSS)

Botan implements the single tree version of the eXtended Merkle Signature Scheme (XMSS) using Winternitz One Time Signatures+ (WOTS+). The implementation is based on IETF Internet-Draft “XMSS: Extended Hash-Based Signatures”.

XMSS uses the Botan interfaces for public key cryptography. The following algorithms are implemented:

1. XMSS_SHA2-256_W16_H10
2. XMSS_SHA2-256_W16_H16
3. XMSS_SHA2-256_W16_H20
4. XMSS_SHA2-512_W16_H10

5. XMSS_SHA2-512_W16_H16
6. XMSS_SHA2-512_W16_H20
7. XMSS_SHAKE128_W16_H10
8. XMSS_SHAKE128_W16_H10
9. XMSS_SHAKE128_W16_H10
10. XMSS_SHAKE256_W16_H10
11. XMSS_SHAKE256_W16_H10
12. XMSS_SHAKE256_W16_H10

13.9.1 Code Example

The following code snippet shows a minimum example on how to create an XMSS public/private key pair and how to use these keys to create and verify a signature:

```
#include <botan/botan.h>
#include <botan/auto_rng.h>
#include <botan/xmss.h>

int main()
   {
   // Create a random number generator used for key generation.
   Botan::AutoSeeded_RNG rng;

   // create a new public/private key pair using SHA2 256 as hash
   // function and a tree height of 10.
   Botan::XMSS_PrivateKey private_key(
      Botan::XMSS_Parameters::xmss_algorithm_t::XMSS_SHA2_256_W16_H10,
      rng);
   Botan::XMSS_PublicKey public_key(private_key);

   // create signature operation using the private key.
   std::unique_ptr<Botan::PK_Ops::Signature> sig_op =
      private_key.create_signature_op(rng, "", "");

   // create and sign a message using the signature operation.
   Botan::secure_vector<uint8_t> msg { 0x01, 0x02, 0x03, 0x04 };
   sig_op->update(msg.data(), msg.size());
   Botan::secure_vector<uint8_t> sig = sig_op->sign(rng);

   // create verification operation using the public key
   std::unique_ptr<Botan::PK_Ops::Verification> ver_op =
      public_key.create_verification_op("", "");

   // verify the signature for the previously generated message.
   ver_op->update(msg.data(), msg.size());
   if(ver_op->is_valid_signature(sig.data(), sig.size()))
      {
      std::cout << "Success." << std::endl;
      }
   else
      {
      std::cout << "Error." << std::endl;
```

(continues on next page)

(continued from previous page)

```
        }
    }
```

CHAPTER

FOURTEEN

X.509 CERTIFICATES AND CRLS

A certificate is a binding between some identifying information (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, used for instance in the *Transport Layer Security (TLS)* protocol. A X.509 certificate is represented by

class X509_Certificate

X509_Certificate(**const** std::string &*filename*)
Load a certificate from a file. PEM or DER is accepted.

X509_Certificate(**const** std::vector<uint8_t> &*in*)
Load a certificate from a byte string.

X509_Certificate(DataSource &*source*)
Load a certificate from an abstract `DataSource`.

std::unique_ptr<Public_Key> **load_subject_public_key**() **const**
Deserialize the stored public key and return a new object. This might throw, if it happens that the public key object stored in the certificate is malformed in some way, or in the case that the public key algorithm used is not supported by the library.

See *Serializing Public Keys* for more information about what to do with the returned object. It may be any type of key, in principle, though RSA and ECDSA are most common.

std::vector<uint8_t> **serial_number**() **const**
Return the certificates serial number. The tuple of issuer DN and serial number should be unique.

X509_DN **subject_dn**() **const**
Returns the distinguished name (DN) of the certificate's subject.

X509_DN **issuer_dn**() **const**
Returns the distinguished name (DN) of the certificate's issuer

X509_Time **not_before**() **const**
Returns the point in time the certificate becomes valid

X509_Time **not_after**() **const**
Returns the point in time the certificate expires

const *Extensions* &**v3_extensions**() **const**

Returns all extensions of this certificate. You can use this to examine any extension data associated with the certificate, including custom extensions the library doesn't know about.

const AlternativeName &**subject_alt_name**() **const**

Return the subjects alternative name. This is used to store values like associated URIs, DNS addresses, and email addresses.

const AlternativeName &**issuer_alt_name**() **const**

Return alternative names for the issuer.

std::string **fingerprint**(**const** std::string &*hash_name* = "SHA-1") **const**

Return a fingerprint for the certificate.

Key_Constraints **constraints**() **const**

Returns either an enumeration listing key constraints (what the associated key can be used for) or `NO_CONSTRAINTS` if the relevent extension was not included. Example values are `DIGITAL_SIGNATURE` and `KEY_CERT_SIGN`. More than one value might be specified.

bool **allowed_extended_usage**(**const** OID &*usage*) **const**

Returns true if the OID is included in the extended usage extension.

bool **matches_dns_name**(**const** std::string &*name*) **const**

Check if the certificate's subject alternative name DNS fields match `name`. This function also handles wildcard certificates.

std::string **to_string**() **const**

Returns a free-form human readable string describing the certificate.

The `X509_Certificate` class has several other functions not described here. See the header `x509cert.h` for details.

The data of an X.509 certificate is stored as a `shared_ptr` to a structure containing the decoded information. So copying `X509_Certificate` objects is quite cheap.

14.1 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 *Serializing Public 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, DSA, and ECDSA 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, an organizational sub-unit name, 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 a locality is, nobody knows, but it's usually given as a city name.

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

So, how to get the information? 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`.

14.1.1 X.509v3 Extensions

X.509v3 specifies a large number of possible extensions. Botan supports some, but by no means all of them. The following listing lists which X.509v3 extensions are supported and notes areas where there may be problems with the handling.

- Key Usage and Extended Key Usage: No problems known.
- Basic Constraints: No problems known. A self-signed v1 certificate is assumed to be a CA, while 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" and raw IPv4 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.
- Name Constraints: No problems known (though encoding is not supported).

Any unknown critical extension in a certificate will lead to an exception during path validation.

Extensions are handled by a special class taking care of encoding and decoding. It also supports encoding and decoding of custom extensions. To do this, it internally keeps two lists of extensions. Different lookup functions are provided to search them.

Note: Validation of custom extensions during path validation is currently not supported.

`class Extensions`

void **`add`**(Certificate_Extension *extn, bool *critical* = false)
: Adds a new extension to the list. `critical` specifies whether the extension should be marked as critical.

`replace`(Certificate_Extension *extn, bool *critical* = false)
: Adds an extension to the list or replaces it, if the same extension was already added

std::unique_ptr<Certificate_Extension> **`get`**(**`const`** OID &*oid*) **`const`**
: Searches for an extension by OID and returns the result

template<typename **`T`**>
std::unique_ptr<T> **`get_raw`**(**`const`** OID &*oid*)
: Searches for an extension by OID and returns the result. Only the unknown extensions, that is, extensions types that are not listed above, are searched for by this function.

std::vector<std::pair<std::unique_ptr<Certificate_Extension>, bool>> **`extensions`**() **`const`**
: Returns the list of extensions together with the corresponding criticality flag. Only contains the supported extension types listed above.

std::map<OID, std::pair<std::vector<uint8_t>, bool>> **`extensions_raw`**() **`const`**
: Returns the list of extensions as raw, encoded bytes together with the corresponding criticality flag. Contains all extensions, known as well as unknown extensions.

14.1.2 Certificate 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). Every once in a while the CA will release a new 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 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 `Certificate_Store` object with

void `Certificate_Store::`**`add_crl`**`(`**const** X509_CRL &*crl*`)`

and all future verifications will take into account the provided CRL.

14.2 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 return an empty `std::shared_ptr` 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 certificate lookup methods are `find_cert` (by Subject Distinguished Name and optional Subject Key Identifier) and `find_cert_by_pubkey_sha1` (by SHA-1 hash of the certificate's public key). The Subject Distinguished Name is given as a `X509_DN`, while the SKID parameter takes a `std::vector<uint8_t>` containing the subject key identifier in raw binary. Both lookup methods are mandatory to implement.

Finally, there is a method for finding a CRL, called `find_crl_for`, that takes an `X509_Certificate` object, and returns a `std::shared_ptr` of `X509_CRL`. The `std::shared_ptr` return type makes it easy to return no CRLs by returning `nullptr` (eg, if the certificate store doesn't support retrieving CRLs). Implementing the function is optional, and by default will return `nullptr`.

Certificate stores are used in the *Transport Layer Security (TLS)* module to store a list of trusted certificate authorities.

14.2.1 In Memory Certificate Store

The in memory certificate store keeps all objects in memory only. Certificates can be loaded from disk initially, but also added later.

`class Certificate_Store_In_Memory`

Certificate_Store_In_Memory (**const** std::string *&dir*)
Attempt to parse all files in `dir` (including subdirectories) as certificates. Ignores errors.

Certificate_Store_In_Memory (**const** *X509_Certificate &cert*)
Adds given certificate to the store

Certificate_Store_In_Memory ()
Create an empty store

void **add_certificate** (**const** *X509_Certificate &cert*)
Add a certificate to the store

void **add_certificate** (std::shared_ptr<**const** *X509_Certificate*> *cert*)
Add a certificate already in a shared_ptr to the store

void **add_crl** (**const** X509_CRL *&crl*)
Add a certificate revocation list (CRL) to the store.

void **add_crl** (std::shared_ptr<**const** X509_CRL> *crl*)
Add a certificate revocation list (CRL) to the store as a shared_ptr

14.2.2 SQL-backed Certificate Stores

The SQL-backed certificate stores store all objects in an SQL database. They also additionally provide private key storage and revocation of individual certificates.

class Certificate_Store_In_SQL

Certificate_Store_In_SQL (**const** std::shared_ptr<SQL_Database> *db*, **const** std::string *&passwd*, RandomNumberGenerator *&rng*, **const** std::string *&table_prefix* = "")
Create or open an existing certificate store from an SQL database. The password in `passwd` will be used to encrypt private keys.

bool **insert_cert** (**const** *X509_Certificate &cert*)
Inserts `cert` into the store. Returns *false* if the certificate is already known and *true* if insertion was successful.

remove_cert (**const** *X509_Certificate &cert*)
Removes `cert` from the store. Returns *false* if the certificate could not be found and *true* if removal was successful.

std::shared_ptr<**const** Private_Key> **find_key** (**const** *X509_Certificate&*) **const**
Returns the private key for "cert" or an empty shared_ptr if none was found

std::vector<std::shared_ptr<**const** *X509_Certificate*>> **find_certs_for_key** (**const** Private_Key *&key*) **const**
Returns all certificates for private key `key`

bool **insert_key** (**const** *X509_Certificate &cert*, **const** Private_Key *&key*)
Inserts `key` for `cert` into the store, returns *false* if the key is already known and *true* if insertion was successful.

void **remove_key** (**const** Private_Key *&key*)
Removes `key` from the store

void **revoke_cert** (**const** *X509_Certificate&*, CRL_Code, **const** X509_Time *&time* = X509_Time())
Marks `cert` as revoked starting from `time`

void **affirm_cert**(**const** *X509_Certificate&*)

Reverses the revokation for `cert`

std::vector<X509_CRL> **generate_crls**() **const**

Generates CRLs for all certificates marked as revoked. A CRL is returned for each unique issuer DN.

The `Certificate_Store_In_SQL` class operates on an abstract `SQL_Database` object. If support for sqlite3 was enabled at build time, Botan includes an implementation of this interface for sqlite3, and a subclass of `Certificate_Store_In_SQL` which creates or opens a sqlite3 database.

class Certificate_Store_In_SQLite

Certificate_Store_In_SQLite(**const** std::string *&db_path*, **const** std::string *&passwd*, RandomNumberGenerator *&rng*, **const** std::string *&table_prefix* = "")

Create or open an existing certificate store from an sqlite database file. The password in `passwd` will be used to encrypt private keys.

14.3 Path Validation

The process of validating a certficate chain up to a trusted root is called *path validation*, and in botan that operation is handled by a set of functions in `x509path.h` named `x509_path_validate`:

Path_Validation_Result **x509_path_validate**(**const** *X509_Certificate &end_cert*, **const** *Path_Validation_Restrictions &restrictions*, **const** Certificate_Store *&store*, **const** std::string *&hostname* = "", Usage_Type *usage* = Usage_Type::UNSPECIFIED, std::chrono::system_clock::time_point *validation_time* = std::chrono::system_clock::now(), std::chrono::milliseconds *ocsp_timeout* = std::chrono::milliseconds(0), **const** std::vector<std::shared_ptr<**const** OCSP::*Response*>> *&ocsp_resp* = {})

The last five parameters are optional. `hostname` specifies a hostname which is matched against the subject DN in `end_cert` according to RFC 6125. An empty hostname disables hostname validation. `usage` specifies key usage restrictions that are compared to the key usage fields in *end_cert* according to RFC 5280, if not set to `UNSPECIFIED`. `validation_time` allows setting the time point at which all certificates are validated. This is really only useful for testing. The default is the current system clock's current time. `ocsp_timeout` sets the timeout for OCSP requests. The default of 0 disables OCSP checks alltogether. `ocsp_resp` allows adding additional OCSP responses retrieved from outside of the path validation. Note that OCSP online checks are done only as long as the http_util module was compiled in. Availability of online OCSP checks can be checked using the macro BOTAN_HAS_ONLINE_REVOCATION_CHECKS.

For the different flavours of `x509_path_validate`, check `x509path.h`.

The result of the validation is returned as a class:

class Path_Validation_Result

Specifies the result of the validation

bool **successful_validation**() **const**

Returns true if a certificate path from *end_cert* to a trusted root was found and all path validation checks passed.

std::string **result_string**() **const**

Returns a descriptive string of the validation status (for instance "Verified", "Certificate is not yet valid", or "Signature error"). This is the string value of the *result* function below.

const *X509_Certificate* &**trust_root**() **const**

If the validation was successful, returns the certificate which is acting as the trust root for *end_cert*.

const std::vector<*X509_Certificate*> &**cert_path**() **const**

Returns the full certificate path starting with the end entity certificate and ending in the trust root.

Certificate_Status_Code **result**() **const**

Returns the 'worst' error that occurred during validation. For instance, we do not want an expired certificate with an invalid signature to be reported to the user as being simply expired (a relativly innocuous and common error) when the signature isn't even valid.

const std::vector<std::set<Certificate_Status_Code>> &**all_statuses**() **const**

For each certificate in the chain, returns a set of status which indicate all errors which occurred during validation. This is primarily useful for diagnostic purposes.

std::set<std::string> **trusted_hashes**() **const**

Returns the set of all cryptographic hash functions which are implicitly trusted for this validation to be correct.

A `Path_Validation_Restrictions` is passed to the path validator and specifies restrictions and options for the validation step. The two constructors are:

Path_Validation_Restrictions(bool *require_rev*, size_t *minimum_key_strength*, bool *ocsp_all_intermediates*, **const** std::set<std::string> &*trusted_hashes*)

If *require_rev* is true, then any path without revocation information (CRL or OCSP check) is rejected with the code *NO_REVOCATION_DATA*. The *minimum_key_strength* parameter specifies the minimum strength of public key signature we will accept is. The set of hash names *trusted_hashes* indicates which hash functions we'll accept for cryptographic signatures. Any untrusted hash will cause the error case *UNTRUSTED_HASH*.

Path_Validation_Restrictions(bool *require_rev* = false, size_t *minimum_key_strength* = 80, bool *ocsp_all_intermediates* = false)

A variant of the above with some convenient defaults. The current default *minimum_key_strength* of 80 roughly cooresponds to 1024 bit RSA. The set of trusted hashes is set to all SHA-2 variants, and, if *minimum_key_strength* is less than or equal to 80, then SHA-1 signatures will also be accepted.

14.4 Creating New Certificates

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::**X509_CA**(**const** *X509_Certificate* &*cert*, **const** Private_Key &*key*, **const** std::string &*hash_fn*, RandomNumberGenerator &*rng*)

The private `key` is the private key corresponding to the public key in the CA's certificate. `hash_fn` is the name of the hash function to use for signing, e.g., *SHA-256*. `rng` is queried for random during signing.

There is an alternative constructor that lets you set additional options, namely the padding scheme that will be used by the X509_CA object to sign certificates and certificate revocation lists. If the padding is not set explicitly, the CA will use the padding scheme that was used when signing the CA certificate.

X509_CA::**X509_CA**(**const** *X509_Certificate &cert*, **const** Private_Key *&key*, **const** std::map<std::string, std::string> *&opts*, **const** std::string *&hash_fn*, RandomNumberGenerator *&rng*)

The only option valid at this moment is “padding”. The supported padding schemes can be found in src/lib/pubkey/padding.cpp. Some alternative names for the padding schemes are understood, as well.

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. A request is vetted by humans (who somehow verify that the name in the request corresponds to the name of the entity who requested it), and then signed by a CA key, generating a new certificate:

X509_Certificate X509_CA::**sign_request**(**const** PKCS10_Request *&req*, RandomNumberGenerator *&rng*, **const** X509_Time *&not_before*, **const** X509_Time *&not_after*)

14.4.1 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. Never fear; empty CRLs, which revoke nothing at all, can be issued. To generate a new, empty CRL, just call

X509_CRL X509_CA::**new_crl**(RandomNumberGenerator *&rng*, uint32_t *next_update* = 0)

This function will return a new, empty CRL. The `next_update` parameter is the number of seconds before the CRL expires. If it is set to the (default) value of zero, then a reasonable default (currently 7 days) will be used.

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

X509_CRL X509_CA::**update_crl**(**const** X509_CRL *&last_crl*, std::vector<CRL_Entry> *new_entries*, RandomNumberGenerator *&rng*, size_t *next_update* = 0)

Where `last_crl` is the last CRL this CA issued, and `new_entries` is a list of any newly revoked certificates. The function returns a new `X509_CRL` to make available for clients.

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 `secure_vector<uint8_t>` 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 `REMOVE_FROM_CRL`. 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.

14.4.2 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 specialized protocols. The library provides a utility function for this:

X509_Certificate **create_self_signed_cert**(**const** X509_Cert_Options *&opts*, **const** Private_Key *&key*, **const** std::string *&hash_fn*, RandomNumberGenerator *&rng*)

Where `key` is 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).

14.4.3 Creating PKCS #10 Requests

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

PKCS10_Request **`create_cert_req`**(**`const`** X509_Cert_Options *&opts*, **`const`** Private_Key *&key*, **`const`** std::string *&hash_fn*, RandomNumberGenerator *&rng*)

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.

14.4.4 Certificate Options

What is this `X509_Cert_Options` thing we've been passing around? It's a class representing 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. For special cases, you can also add custom X.509v3 extensions.

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".

Third, 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.

Moreover, you can specify the padding scheme to be used when digital signatures are computed by calling function `set_padding_scheme` with a string representing the padding scheme. This way, you can control the padding scheme for self-signed certificates and PKCS #10 requests. The padding scheme used by a CA when building a certificate or a certificate revocation list can be set in the `X509_CA` constructor. The supported padding schemes can be found in src/lib/pubkey/padding.cpp. Some alternative names for the padding schemes are understood, as well.

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 5280), or just 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.

Lastly, you can add any X.509v3 extensions in the *extensions* member. This is really only useful if you want to encode custom extensions in the certificate. Most users probably won’t need this. Note that extensions added this way will be overwritten by an `X509_CA` if also added by the `X509_CA` itself. This currently includes the Basic Constraints, Key Usage, Authority Key ID, Subject Key ID, Subject Alternative Name and Extended Key Usage extension.

14.5 OCSP Requests

A client makes an OCSP request to what is termed an ‘OCSP responder’. This responder returns a signed response attesting that the certificate in question has not been revoked. The most recent OCSP specification is as of this writing **RFC 6960** (https://tools.ietf.org/html/rfc6960.html).

Normally OCSP validation happens automatically as part of X.509 certificate validation, as long as OCSP is enabled (by setting a non-zero `ocsp_timeout` in the call to `x509_path_validate`, or for TLS by implementing the related `tls_verify_cert_chain_ocsp_timeout` callback and returning a non-zero value from that). So most applications should not need to directly manipulate OCSP request and response objects.

For those that do, the primary ocsp interface is in `ocsp.h`. First a request must be formed, using information contained in the subject certificate and in the subject’s issuing certificate.

class `OCSP::`**Request**

`OCSP::`**Request**(**const** *X509_Certificate &issuer_cert*, **const** BigInt &*subject_serial*)
Create a new OCSP request

`OCSP::`**Request**(**const** *X509_Certificate &issuer_cert*, **const** *X509_Certificate &subject_cert*)
Variant of the above, using serial number from `subject_cert`.

std::vector<uint8_t> **BER_encode**() **const**
Encode the current OCSP request as a binary string.

std::string **base64_encode**() **const**
Encode the current OCSP request as a base64 string.

Then the response is parsed and validated, and if valid, can be consulted for certificate status information.

class `OCSP::`**Response**

`OCSP::`**Response**(**const** uint8_t *response_bits*[], size_t *response_bits_len*)
Attempts to parse `response_bits` as an OCSP response. Throws an exception if parsing fails. Note that this does not verify that the OCSP response is valid (ie that the signature is correct), merely that the ASN.1 structure matches an OCSP response.

Certificate_Status_Code **check_signature**(**const** std::vector<Certificate_Store *> &*trust_roots*, **const** std::vector<std::shared_ptr<**const** *X509_Certificate*>> &*cert_path* = {}) **const**
Find the issuing certificate of the OCSP response, and check the signature.

If possible, pass the full certificate path being validated in the optional `cert_path` argument: this additional information helps locate the OCSP signer’s certificate in some cases. If this does not return `Certificate_Status_Code::OCSP_SIGNATURE_OK`, then the request must not be be used further.

Certificate_Status_Code **verify_signature**(**const** *X509_Certificate &issuing_cert*) **const**

If the certificate that issued the OCSP response is already known (eg, because in some specific application all the OCSP responses will always be signed by a single trusted issuer whose cert is baked into the code) this provides an alternate version of *check_signature*.

Certificate_Status_Code **status_for**(**const** *X509_Certificate &issuer*, **const** *X509_Certificate &subject*, std::chrono::system_clock::time_point *ref_time* = std::chrono::system_clock::now()) **const**

Assuming the signature is valid, returns the status for the subject certificate. Make sure to get the ordering of the issuer and subject certificates correct.

The `ref_time` is normally just the system clock, but can be used if validation against some other reference time is desired (such as for testing, to verify an old previously valid OCSP response, or to use an alternate time source such as the Roughtime protocol instead of the local client system clock).

const X509_Time &**produced_at**() **const**

Return the time this OCSP response was (claimed to be) produced at.

const X509_DN &**signer_name**() **const**

Return the distinguished name of the signer. This is used to help find the issuing certificate.

This field is optional in OCSP responses, and may not be set.

const std::vector<uint8_t> &**signer_key_hash**() **const**

Return the SHA-1 hash of the public key of the signer. This is used to help find the issuing certificate. The `Certificate_Store` API `find_cert_by_pubkey_sha1` can search on this value.

This field is optional in OCSP responses, and may not be set.

const std::vector<uint8_t> &**raw_bits**() **const**

Return the entire raw ASN.1 blob (for debugging or specialized decoding needs)

One common way of making OCSP requests is via HTTP, see **RFC 2560** (https://tools.ietf.org/html/rfc2560.html) Appendix A for details. A basic implementation of this is the function `online_check`, which is available as long as the `http_util` module was compiled in; check by testing for the macro `BOTAN_HAS_HTTP_UTIL`.

OCSP::*Response* **online_check**(**const** *X509_Certificate &issuer*, **const** BigInt *&subject_serial*, **const** std::string *&ocsp_responder*, **const** Certificate_Store **trusted_roots*)

Assemble a OCSP request for serial number `subject_serial` and attempt to request it to responder at URI `ocsp_responder` over a new HTTP socket, parses and returns the response. If trusted_roots is not null, then the response is additionally validated using OCSP response API `check_signature`. Otherwise, this call must be performed later by the application.

OCSP::*Response* **online_check**(**const** *X509_Certificate &issuer*, **const** *X509_Certificate &subject*, **const** Certificate_Store **trusted_roots*)

Variant of the above but uses serial number and OCSP responder URI from `subject`.

CHAPTER

FIFTEEN

TRANSPORT LAYER SECURITY (TLS)

New in version 1.11.0.

Botan has client and server implementations of various versions of the TLS protocol, including TLS v1.0, TLS v1.1, and TLS v1.2. As of version 1.11.13, support for the insecure SSLv3 protocol has been removed.

There is also support for DTLS (v1.0 and v1.2), a variant of TLS adapted for operation on datagram transports such as UDP and SCTP. DTLS support should be considered as beta quality and further testing is invited.

The TLS implementation does not know anything about sockets or the network layer. Instead, it calls a user provided callback (hereafter `output_fn`) whenever it has data that it would want to send to the other party (for instance, by writing it to a network socket), and whenever the application receives some data from the counterparty (for instance, by reading from a network socket) it passes that information to TLS using *`TLS::Channel::received_data`*. If the data passed in results in some change in the state, such as a handshake completing, or some data or an alert being received from the other side, then the appropriate user provided callback will be invoked.

If the reader is familiar with OpenSSL's BIO layer, it might be analagous to saying the only way of interacting with Botan's TLS is via a *BIO_mem* I/O abstraction. This makes the library completely agnostic to how you write your network layer, be it blocking sockets, libevent, asio, a message queue, lwIP on RTOS, some carrier pidgeons, etc.

Starting in 1.11.31, the application callbacks are encapsulated as the class `TLS::Callbacks` with the following members. The first four (`tls_emit_data`, `tls_record_received`, `tls_alert`, and `tls_session_established`) are mandatory for using TLS, all others are optional and provide additional information about the connection.

void **`tls_emit_data`**(**`const`** uint8_t *data*[], size_t *data_len*)

> Mandatory. The TLS stack requests that all bytes of *data* be queued up to send to the counterparty. After this function returns, the buffer containing *data* will be overwritten, so a copy of the input must be made if the callback cannot send the data immediately.
>
> As an example you could `send` to perform a blocking write on a socket, or append the data to a queue managed by your application, and initiate an asyncronous write.
>
> For TLS all writes must occur *in the order requested*. For DTLS this ordering is not strictly required, but is still recommended.

void **`tls_record_received`**(uint64_t *rec_no*, **`const`** uint8_t *data*[], size_t *data_len*)

> Mandatory. Called once for each application_data record which is received, with the matching (TLS level) record sequence number.
>
> Currently empty records are ignored and do not instigate a callback, but this may change in a future release.
>
> > As with `tls_emit_data`, the array will be overwritten sometime after the callback returns, so a copy should be made if needed.
> >
> > For TLS the record number will always increase.

For DTLS, it is possible to receive records with the *rec_no* field out of order, or with gaps, cooresponding to reordered or lost datagrams.

void **tls_alert**(Alert *alert*)

Mandatory. Called when an alert is received from the peer. Note that alerts received before the handshake is complete are not authenticated and could have been inserted by a MITM attacker.

bool **tls_session_established**(**const** TLS::*Session &session*)

Mandatory. Called whenever a negotiation completes. This can happen more than once on any connection, if renegotiation occurs. The *session* parameter provides information about the session which was just established.

If this function returns false, the session will not be cached for later resumption.

If this function wishes to cancel the handshake, it can throw an exception which will send a close message to the counterparty and reset the connection state.

std::string **tls_server_choose_app_protocol**(**const** std::vector<std::string> *&client_protos*)

Optional. Called by the server when a client includes a list of protocols in the ALPN extension. The server then choose which protocol to use, or "" to disable sending any ALPN response. The default implementation returns the empty string all of the time, effectively disabling ALPN responses.

void **tls_inspect_handshake_msg**(**const** Handshake_Message&)

This callback is optional, and can be used to inspect all handshake messages while the session establishment occurs.

void **tls_log_error**(**const** char **msg*)

Optional logging for an error message. (Not currently used)

void **tls_log_debug**(**const** char **msg*)

Optional logging for an debug message. (Not currently used)

void **tls_log_debug_bin**(**const** char **descr*, **const** uint8_t *val*[], size_t *len*)

Optional logging for an debug value. (Not currently used)

Versions from 1.11.0 to 1.11.30 did not have `TLS::Callbacks` and instead used independent std::functions to pass the various callback functions. This interface is currently still included but is deprecated and will be removed in a future release. For the documentation for this interface, please check the docs for 1.11.30. This version of the manual only documents the new interface added in 1.11.31.

15.1 TLS Channels

TLS servers and clients share an interface called *TLS::Channel*. A TLS channel (either client or server object) has these methods available:

class TLS::**Channel**

size_t **received_data**(**const** uint8_t *buf*[], size_t *buf_size*)

size_t **received_data**(**const** std::vector<uint8_t> *&buf*)

This function is used to provide data sent by the counterparty (eg data that you read off the socket layer). Depending on the current protocol state and the amount of data provided this may result in one or more callback functions that were provided to the constructor being called.

The return value of `received_data` specifies how many more bytes of input are needed to make any progress, unless the end of the data fell exactly on a message boundary, in which case it will return 0 instead.

void **send**(**const** uint8_t *buf*[], size_t *buf_size*)

void **send**(**const** std::string &*str*)

void **send**(**const** std::vector<uint8_t> &*vec*)

Create one or more new TLS application records containing the provided data and send them. This will eventually result in at least one call to the `output_fn` callback before `send` returns.

If the current TLS connection state is unable to transmit new application records (for example because a handshake has not yet completed or the connnection has already ended due to an error) an exception will be thrown.

void **close**()

A close notification is sent to the counterparty, and the internal state is cleared.

void **send_alert**(**const** *Alert* &*alert*)

Some other alert is sent to the counterparty. If the alert is fatal, the internal state is cleared.

bool **is_active**()

Returns true if and only if a handshake has been completed on this connection and the connection has not been subsequently closed.

bool **is_closed**()

Returns true if and only if either a close notification or a fatal alert message have been either sent or received.

bool **timeout_check**()

This function does nothing unless the channel represents a DTLS connection and a handshake is actively in progress. In this case it will check the current timeout state and potentially initiate retransmission of handshake packets. Returns true if a timeout condition occurred.

void **renegotiate**(bool *force_full_renegotiation* = false)

Initiates a renegotiation. The counterparty is allowed by the protocol to ignore this request. If a successful renegotiation occurs, the *handshake_cb* callback will be called again.

If *force_full_renegotiation* is false, then the client will attempt to simply renew the current session - this will refresh the symmetric keys but will not change the session master secret. Otherwise it will initiate a completely new session.

For a server, if *force_full_renegotiation* is false, then a session resumption will be allowed if the client attempts it. Otherwise the server will prevent resumption and force the creation of a new session.

std::vector<*X509_Certificate*> **peer_cert_chain**()

Returns the certificate chain of the counterparty. When acting as a client, this value will be non-empty unless the client's policy allowed anonymous connections and the server then chose an anonymous ciphersuite. Acting as a server, this value will ordinarily be empty, unless the server requested a certificate and the client responded with one.

SymmetricKey **key_material_export**(**const** std::string &*label*, **const** std::string &*context*, size_t *length*)

Returns an exported key of *length* bytes derived from *label*, *context*, and the session's master secret and client and server random values. This key will be unique to this connection, and as long as the session master secret remains secure an attacker should not be able to guess the key.

Per **RFC 5705** (https://tools.ietf.org/html/rfc5705.html), *label* should begin with "EXPERIMENTAL" unless the label has been standardized in an RFC.

15.2 TLS Clients

class TLS::**Client**

Client (Callbacks &*callbacks*, Session_Manager &*session_manager*, *Credentials_Manager &creds*, **const** *Policy &policy*, RandomNumberGenerator &*rng*, **const** Server_Information &*server_info* = Server_Information(), **const** *Protocol_Version offer_version* = *Protocol_Version*::*latest_tls_version*(), **const** std::vector<std::string> &*next_protocols* = {}, size_t *reserved_io_buffer_size* = 16 * 1024)

Initialize a new TLS client. The constructor will immediately initiate a new session.

The *callbacks* parameter specifies the various application callbacks which pertain to this particular client connection.

The *session_manager* is an interface for storing TLS sessions, which allows for session resumption upon reconnecting to a server. In the absence of a need for persistent sessions, use `TLS::Session_Manager_In_Memory` which caches connections for the lifetime of a single process. See *TLS Session Managers* for more about session managers.

The *credentials_manager* is an interface that will be called to retrieve any certificates, secret keys, pre-shared keys, or SRP information; see *Credentials Manager* for more information.

Use the optional *server_info* to specify the DNS name of the server you are attempting to connect to, if you know it. This helps the server select what certificate to use and helps the client validate the connection.

Note that the server name indicator name must be a FQDN. IP addresses are not allowed by RFC 6066 and may lead to interoperability problems.

Use the optional *offer_version* to control the version of TLS you wish the client to offer. Normally, you'll want to offer the most recent version of (D)TLS that is available, however some broken servers are intolerant of certain versions being offered, and for classes of applications that have to deal with such servers (typically web browsers) it may be necessary to implement a version backdown strategy if the initial attempt fails.

> **Warning:** Implementing such a backdown strategy allows an attacker to downgrade your connection to the weakest protocol that both you and the server support.

Setting *offer_version* is also used to offer DTLS instead of TLS; use *`TLS::Protocol_Version::latest_dtls_version`*.

Optionally, the client will advertise *app_protocols* to the server using the ALPN extension.

The optional *reserved_io_buffer_size* specifies how many bytes to pre-allocate in the I/O buffers. Use this if you want to control how much memory the channel uses initially (the buffers will be resized as needed to process inputs). Otherwise some reasonable default is used.

15.2.1 Code Example

A minimal example of a TLS client is provided below. The full code for a TLS client using BSD sockets is in *src/cli/tls_client.cpp*

```
#include <botan/tls_client.h>
#include <botan/tls_callbacks.h>
#include <botan/tls_session_manager.h>
#include <botan/tls_policy.h>
```

(continues on next page)

(continued from previous page)

```
#include <botan/auto_rng.h>
#include <botan/certstor.h>

/**
 * @brief Callbacks invoked by TLS::Channel.
 *
 * Botan::TLS::Callbacks is an abstract class.
 * For improved readability, only the functions that are mandatory
 * to implement are listed here. See src/lib/tls/tls_callbacks.h.
 */
class Callbacks : public Botan::TLS::Callbacks
{
   public:
      void tls_emit_data(const uint8_t data[], size_t size) override
         {
         // send data to tls server, e.g., using BSD sockets or boost asio
         }

      void tls_record_received(uint64_t seq_no, const uint8_t data[], size_t size)␣
↪override
         {
         // process full TLS record received by tls server, e.g.,
         // by passing it to the application
         }

      void tls_alert(Botan::TLS::Alert alert) override
         {
         // handle a tls alert received from the tls server
         }

      bool tls_session_established(const Botan::TLS::Session& session) override
         {
         // the session with the tls server was established
         // return false to prevent the session from being cached, true to
         // cache the session in the configured session manager
         return false;
         }
};

/**
 * @brief Credentials storage for the tls client.
 *
 * It returns a list of trusted CA certificates from a local directory.
 * TLS client authentication is disabled. See src/lib/tls/credentials_manager.h.
 */
class Client_Credentials : public Botan::Credentials_Manager
{
   public:
      std::vector<Certificate_Store*> trusted_certificate_authorities(
         const std::string& type,
         const std::string& context) override
         {
         // return a list of certificates of CAs we trust for tls server certificates,
         // e.g., all the certificates in the local directory "cas"
         return { new Botan::Certificate_Store_In_Memory("cas") };
         }
```

(continues on next page)

(continued from previous page)

```
      std::vector<X509_Certificate> cert_chain(
         const std::vector<std::string>& cert_key_types,
         const std::string& type,
         const std::string& context) override
         {
         // when using tls client authentication (optional), return
         // a certificate chain being sent to the tls server,
         // else an empty list
         return std::vector<Botan::X509_Certificate>();
         }

      Botan::Private_Key* private_key_for(const Botan::X509_Certificate& cert,
         const std::string& type,
         const std::string& context) override
         {
         // when returning a chain in cert_chain(), return the private key
         // associated with the leaf certificate here
         return nullptr;
         }
};

int main()
   {
   // prepare all the parameters
   Callbacks callbacks;
   Botan::AutoSeeded_RNG rng;
   Botan::TLS::Session_Manager_In_Memory session_mgr(rng);
   Botan::Client_Credentials creds;
   Botan::TLS::Strict_Policy policy;

   // open the tls connection
   Botan::TLS::Client client(callbacks,
                             session_mgr,
                             creds,
                             policy,
                             rng,
                             Botan::TLS::Server_Information("botan.randombit.net",␣
↪443),
                             Botan::TLS::Protocol_Version::TLS_V12);

   while(!client.is_closed())
      {
      // read data received from the tls server, e.g., using BSD sockets or boost asio
      // ...

      // send data to the tls server using client.send_data()
      }
   }
```

15.3 TLS Servers

class TLS::**Server**

Server (Callbacks *&callbacks*, Session_Manager *&session_manager*, *Credentials_Manager &creds*, **const** *Policy &policy*, RandomNumberGenerator *&rng*, bool *is_datagram* = false, size_t *reserved_io_buffer_size* = 16 * 1024)

The first 5 arguments as well as the final argument *reserved_io_buffer_size*, are treated similiarly to the *client*.

If a client sends the ALPN extension, the `callbacks` function `tls_server_choose_app_protocol` will be called and the result sent back to the client. If the empty string is returned, the server will not send an ALPN response. The function can also throw an exception to abort the handshake entirely, the ALPN specification says that if this occurs the alert should be of type *NO_APPLICATION_PROTOCOL*.

The optional argument *is_datagram* specifies if this is a TLS or DTLS server; unlike clients, which know what type of protocol (TLS vs DTLS) they are negotiating from the start via the *offer_version*, servers would not until they actually received a client hello.

15.3.1 Code Example

A minimal example of a TLS server is provided below. The full code for a TLS server using asio is in *src/cli/tls_proxy.cpp*.

```
#include <botan/tls_client.h>
#include <botan/tls_callbacks.h>
#include <botan/tls_session_manager.h>
#include <botan/tls_policy.h>
#include <botan/auto_rng.h>
#include <botan/certstor.h>
#include <botan/pk_keys.h>

#include <memory>

/**
 * @brief Callbacks invoked by TLS::Channel.
 *
 * Botan::TLS::Callbacks is an abstract class.
 * For improved readability, only the functions that are mandatory
 * to implement are listed here. See src/lib/tls/tls_callbacks.h.
 */
class Callbacks : public Botan::TLS::Callbacks
{
   public:
      void tls_emit_data(const uint8_t data[], size_t size) override
         {
         // send data to tls client, e.g., using BSD sockets or boost asio
         }

      void tls_record_received(uint64_t seq_no, const uint8_t data[], size_t size)␣
↪override
         {
         // process full TLS record received by tls client, e.g.,
         // by passing it to the application
         }

      void tls_alert(Botan::TLS::Alert alert) override
         {
         // handle a tls alert received from the tls server
         }
```

(continues on next page)

(continued from previous page)

```
      bool tls_session_established(const Botan::TLS::Session& session) override
         {
         // the session with the tls client was established
         // return false to prevent the session from being cached, true to
         // cache the session in the configured session manager
         return false;
         }
};

/**
 * @brief Credentials storage for the tls server.
 *
 * It returns a certificate and the associated private key to
 * authenticate the tls server to the client.
 * TLS client authentication is not requested.
 * See src/lib/tls/credentials_manager.h.
 */
class Server_Credentials : public Botan::Credentials_Manager
{
   public:
      Server_Credentials() : m_key(Botan::X509::load_key("botan.randombit.net.key"))
         {
         }

      std::vector<Certificate_Store*> trusted_certificate_authorities(
         const std::string& type,
         const std::string& context) override
         {
         // if client authentication is required, this function
         // shall return a list of certificates of CAs we trust
         // for tls client certificates, otherwise return an empty list
         return std::vector<Certificate_Store*>();
         }

      std::vector<X509_Certificate> cert_chain(
         const std::vector<std::string>& cert_key_types,
         const std::string& type,
         const std::string& context) override
         {
         // return the certificate chain being sent to the tls client
         // e.g., the certificate file "botan.randombit.net.crt"
         return { Botan::X509_Certificate("botan.randombit.net.crt") };
         }

      Botan::Private_Key* private_key_for(const Botan::X509_Certificate& cert,
         const std::string& type,
         const std::string& context) override
         {
         // return the private key associated with the leaf certificate,
         // in this case the one associated with "botan.randombit.net.crt"
         return &m_key;
         }

      private:
         std::unique_ptr<Botan::Private_Key> m_key;
};
```

(continues on next page)

(continued from previous page)

```
int main()
   {
   // prepare all the parameters
   Callbacks callbacks;
   Botan::AutoSeeded_RNG rng;
   Botan::TLS::Session_Manager_In_Memory session_mgr(rng);
   Botan::Client_Credentials creds;
   Botan::TLS::Strict_Policy policy;

   // accept tls connection from client
   Botan::TLS::Server server(callbacks,
                             session_mgr,
                             creds,
                             policy,
                             rng);

   // read data received from the tls client, e.g., using BSD sockets or boost asio
   // and pass it to server.received_data().
   // ...

   // send data to the tls client using server.send_data()
   // ...
   }
```

15.4 TLS Sessions

TLS allows clients and servers to support *session resumption*, where the end point retains some information about an established session and then reuse that information to bootstrap a new session in way that is much cheaper computationally than a full handshake.

Every time your handshake callback is called, a new session has been established, and a `TLS::Session` is included that provides information about that session:

class `TLS::`**`Session`**

Protocol_Version **`version`**`()` **`const`**

Returns the *`protocol version`* that was negotiated

Ciphersuite **`ciphersite`**`()` **`const`**

Returns the *`ciphersuite`* that was negotiated.

Server_Information **`server_info`**`()` **`const`**

Returns information that identifies the server side of the connection. This is useful for the client in that it identifies what was originally passed to the constructor. For the server, it includes the name the client specified in the server name indicator extension.

std::vector<*X509_Certificate*> **`peer_certs`**`()` **`const`**

Returns the certificate chain of the peer

std::string **`srp_identifier`**`()` **`const`**

If an SRP ciphersuite was used, then this is the identifier that was used for authentication.

bool **`secure_renegotiation`**`()` **`const`**

Returns `true` if the connection was negotiated with the correct extensions to prevent the renegotiation attack.

std::vector<uint8_t> **encrypt**(**const** SymmetricKey &*key*, RandomNumberGenerator &*rng*)

Encrypts a session using a symmetric key *key* and returns a raw binary value that can later be passed to `decrypt`. The key may be of any length.

Currently the implementation encrypts the session using AES-256 in GCM mode with a random nonce.

static *Session* **decrypt**(**const** uint8_t *ciphertext*[], size_t *length*, **const** SymmetricKey &*key*)

Decrypts a session that was encrypted previously with `encrypt` and `key`, or throws an exception if decryption fails.

secure_vector<uint8_t> **DER_encode**() **const**

Returns a serialized version of the session.

> **Warning:** The return value of `DER_encode` contains the master secret for the session, and an attacker who recovers it could recover plaintext of previous sessions or impersonate one side to the other.

15.5 TLS Session Managers

You may want sessions stored in a specific format or storage type. To do so, implement the `TLS::Session_Manager` interface and pass your implementation to the `TLS::Client` or `TLS::Server` constructor.

class TLS::**Session_Mananger**

void **save**(**const** *Session* &*session*)

Save a new *session*. It is possible that this sessions session ID will replicate a session ID already stored, in which case the new session information should overwrite the previous information.

void **remove_entry**(**const** std::vector<uint8_t> &*session_id*)

Remove the session identified by *session_id*. Future attempts at resumption should fail for this session.

bool **load_from_session_id**(**const** std::vector<uint8_t> &*session_id*, *Session* &*session*)

Attempt to resume a session identified by *session_id*. If located, *session* is set to the session data previously passed to *save*, and `true` is returned. Otherwise *session* is not modified and `false` is returned.

bool **load_from_server_info**(**const** Server_Information &*server*, *Session* &*session*)

Attempt to resume a session with a known server.

std::chrono::seconds **session_lifetime**() **const**

Returns the expected maximum lifetime of a session when using this session manager. Will return 0 if the lifetime is unknown or has no explicit expiration policy.

15.5.1 In Memory Session Manager

The `TLS::Session_Manager_In_Memory` implementation saves sessions in memory, with an upper bound on the maximum number of sessions and the lifetime of a session.

It is safe to share a single object across many threads as it uses a lock internally.

class TLS::**Session_Managers_In_Memory**

Session_Manager_In_Memory(RandomNumberGenerator *&rng*, size_t *max_sessions* = 1000, std::chrono::seconds *session_lifetime* = 7200)

Limits the maximum number of saved sessions to *max_sessions*, and expires all sessions older than *session_lifetime*.

15.5.2 Noop Session Mananger

The `TLS::Session_Manager_Noop` implementation does not save sessions at all, and thus session resumption always fails. Its constructor has no arguments.

15.5.3 SQLite3 Session Manager

This session manager is only available if support for SQLite3 was enabled at build time. If the macro `BOTAN_HAS_TLS_SQLITE3_SESSION_MANAGER` is defined, then `botan/tls_session_manager_sqlite.h` contains `TLS::Session_Manager_SQLite` which stores sessions persistently to a sqlite3 database. The session data is encrypted using a passphrase, and stored in two tables, named `tls_sessions` (which holds the actual session information) and `tls_sessions_metadata` (which holds the PBKDF information).

Warning: The hostnames associated with the saved sessions are stored in the database in plaintext. This may be a serious privacy risk in some applications.

class TLS::**Session_Manager_SQLite**

Session_Manager_SQLite(**const** std::string *&passphrase*, RandomNumberGenerator *&rng*, **const** std::string *&db_filename*, size_t *max_sessions* = 1000, std::chrono::seconds *session_lifetime* = 7200)

Uses the sqlite3 database named by *db_filename*.

15.6 TLS Policies

`TLS::Policy` is how an application can control details of what will be negotiated during a handshake. The base class acts as the default policy. There is also a `Strict_Policy` (which forces only secure options, reducing compatibility) and `Text_Policy` which reads policy settings from a file.

class TLS::**Policy**

std::vector<std::string> **allowed_ciphers**() **const**

Returns the list of ciphers we are willing to negotiate, in order of preference.

Clients send a list of ciphersuites in order of preference, servers are free to choose any of them. Some servers will use the clients preferences, others choose from the clients list prioritizing based on its preferences.

No export key exchange mechanisms or ciphersuites are supported by botan. The null encryption ciphersuites (which provide only authentication, sending data in cleartext) are also not supported by the implementation and cannot be negotiated.

Cipher names without an explicit mode refers to CBC+HMAC ciphersuites.

Default value: "ChaCha20Poly1305", "AES-256/GCM", "AES-128/GCM", "AES-256/CCM", "AES-128/CCM", "AES-256", "AES-128"

Also allowed: "AES-256/CCM(8)", "AES-128/CCM(8)", "Camellia-256/GCM", "Camellia-128/GCM", "ARIA-256/GCM", "ARIA-128/GCM", "Camellia-256", "Camellia-128"

Also allowed (though currently experimental): "AES-128/OCB(12)", "AES-256/OCB(12)"

Also allowed (although **not recommended**): "SEED", "3DES"

Note: Before 1.11.30 only the non-standard ChaCha20Poly1305 ciphersuite was implemented. The RFC 7905 ciphersuites are supported in 1.11.30 onwards.

Note: Support for the broken RC4 cipher was removed in 1.11.17

Note: SEED and 3DES are deprecated and will be removed in a future release.

std::vector<std::string> **allowed_macs**() **const**

Returns the list of algorithms we are willing to use for message authentication, in order of preference.

Default: "AEAD", "SHA-256", "SHA-384", "SHA-1"

A plain hash function indicates HMAC

Note: SHA-256 is preferred over SHA-384 in CBC mode because the protections against the Lucky13 attack are somewhat more effective for SHA-256 than SHA-384.

std::vector<std::string> **allowed_key_exchange_methods**() **const**

Returns the list of key exchange methods we are willing to use, in order of preference.

Default: "CECPQ1", "ECDH", "DH"

Note: CECPQ1 key exchange provides post-quantum security to the key exchange by combining NewHope with a standard x25519 ECDH exchange. This prevents an attacker, even one with a quantum computer, from later decrypting the contents of a recorded TLS transcript. The NewHope algorithm is very fast, but adds roughly 4 KiB of additional data transfer to every TLS handshake. And even if NewHope ends up completely broken, the 'extra' x25519 exchange secures the handshake.

For applications where the additional data transfer size is unacceptable, simply allow only ECDH key exchange in the application policy. DH exchange also often involves transferring several additional Kb (without the benefit of post quantum security) so if CECPQ1 is being disabled for traffic overhread reasons, DH should also be avoid.

Also allowed: "RSA", "SRP_SHA", "ECDHE_PSK", "DHE_PSK", "PSK"

Note: Static RSA ciphersuites are disabled by default since 1.11.34. In addition to not providing forward security, any server which is willing to negotiate these ciphersuites exposes themselves to a variety of chosen ciphertext oracle attacks which are all easily avoided by signing (as in PFS) instead of decrypting.

std::vector<std::string> **allowed_signature_hashes**() **const**

Returns the list of hash algorithms we are willing to use for public key signatures, in order of preference.

Default: "SHA-512", "SHA-384", "SHA-256"

Also allowed (although **not recommended**): "SHA-1"

Note: This is only used with TLS v1.2. In earlier versions of the protocol, signatures are fixed to using only SHA-1 (for DSA/ECDSA) or a MD5/SHA-1 pair (for RSA).

std::vector<std::string> **allowed_signature_methods**() **const**

Default: "ECDSA", "RSA"

Also allowed (disabled by default): "DSA", "" (empty string meaning anonymous)

Note: DSA authentication is deprecated and will be removed in a future release.

std::vector<std::string> **allowed_ecc_curves**() **const**

Return a list of ECC curves we are willing to use, in order of preference. The default ordering puts the best performing ECC first.

Default: "x25519", "secp256r1", "secp521r1", "secp384r1", "brainpool256r1", "brainpool384r1", "brainpool512r1"

No other values are currently defined.

bool **use_ecc_point_compression**() **const**

Prefer ECC point compression.

Signals that we prefer ECC points to be compressed when transmitted to us. The other party may not support ECC point compression and therefore may still send points uncompressed.

Note that the certificate used during authentication must also follow the other party's preference.

Default: false

std::vector<uint8_t> **compression**() **const**

Return the list of compression methods we are willing to use, in order of preference. Default is null compression only.

Note: TLS data compression is not currently supported.

bool **acceptable_protocol_version**(Protocol_Version *version*)

Return true if this version of the protocol is one that we are willing to negotiate.

Default: Accepts TLS v1.0 or higher and DTLS v1.2 or higher.

bool **server_uses_own_ciphersuite_preferences**() **const**

If this returns true, a server will pick the cipher it prefers the most out of the client's list. Otherwise, it will negotiate the first cipher in the client's ciphersuite list that it supports.

bool **negotiate_heartbeat_support**() **const**

If this function returns true, clients will offer the heartbeat support extension, and servers will respond to clients offering the extension. Otherwise, clients will not offer heartbeat support and servers will ignore clients offering heartbeat support.

If this returns true, callers should expect to handle heartbeat data in their `alert_cb`.

Default: false

bool **allow_client_initiated_renegotiation**() **const**

If this function returns true, a server will accept a client-initiated renegotiation attempt. Otherwise it will send the client a non-fatal `no_renegotiation` alert.

Default: true

bool **allow_server_initiated_renegotiation**() **const**

If this function returns true, a client will accept a server-initiated renegotiation attempt. Otherwise it will send the server a non-fatal `no_renegotiation` alert.

Default: false

bool **allow_insecure_renegotiation**() **const**

If this function returns true, we will allow renegotiation attempts even if the counterparty does not support the RFC 5746 extensions.

> **Warning:** Returning true here could expose you to attacks

Default: false

size_t **minimum_signature_strength**() **const**

Return the minimum strength (as `n`, representing `2**n` work) we will accept for a signature algorithm on any certificate.

Use 80 to enable RSA-1024 (*not recommended*), or 128 to require either ECC or large (~3000 bit) RSA keys.

Default: 110 (allowing 2048 bit RSA)

bool **require_cert_revocation_info**() **const**

If this function returns true, and a ciphersuite using certificates was negotiated, then we must have access to a valid CRL or OCSP response in order to trust the certificate.

> **Warning:** Returning false here could expose you to attacks

Default: true

std::string **dh_group**() **const**

For ephemeral Diffie-Hellman key exchange, the server sends a group parameter. Return a string specifying the group parameter a server should use.

Default: 2048 bit IETF IPsec group (“modp/ietf/2048”)

size_t **minimum_dh_group_size**() **const**

Return the minimum size in bits for a Diffie-Hellman group that a client will accept. Due to the design of the protocol the client has only two options - accept the group, or reject it with a fatal alert then attempt to reconnect after disabling ephemeral Diffie-Hellman.

Default: 2048 bits

size_t **minimum_rsa_bits**() **const**

Minimum accepted RSA key size. Default 2048 bits.

size_t **minimum_dsa_group_size**() **const**
Minimum accepted DSA key size. Default 2048 bits.

size_t **minimum_ecdsa_group_size**() **const**
Minimum size for ECDSA keys (256 bits).

size_t **minimum_ecdh_group_size**() **const**
Minimum size for ECDH keys (255 bits).

void **check_peer_key_acceptable**(**const** Public_Key &*public_key*) **const**

Allows the policy to examine peer public keys. Throw an exception if the key should be rejected. Default implementation checks against policy values *minimum_dh_group_size*, *minimum_rsa_bits*, *minimum_ecdsa_group_size*, and *minimum_ecdh_group_size*.

bool **hide_unknown_users**() **const**
The SRP and PSK suites work using an identifier along with a shared secret. If this function returns true, when an identifier that the server does not recognize is provided by a client, a random shared secret will be generated in such a way that a client should not be able to tell the difference between the identifier not being known and the secret being wrong. This can help protect against some username probing attacks. If it returns false, the server will instead send an `unknown_psk_identity` alert when an unknown identifier is used.

Default: false

u32bit **session_ticket_lifetime**() **const**
Return the lifetime of session tickets. Each session includes the start time. Sessions resumptions using tickets older than `session_ticket_lifetime` seconds will fail, forcing a full renegotiation.

Default: 86400 seconds (1 day)

15.7 TLS Ciphersuites

class `TLS::`**Ciphersuite**

uint16_t **ciphersuite_code**() **const**
Return the numerical code for this ciphersuite

std::string **to_string**() **const**
Return the ful name of ciphersuite (for example "RSA_WITH_RC4_128_SHA" or "ECDHE_RSA_WITH_AES_128_GCM_SHA256")

std::string **kex_algo**() **const**
Return the key exchange algorithm of this ciphersuite

std::string **sig_algo**() **const**
Return the signature algorithm of this ciphersuite

std::string **cipher_algo**() **const**
Return the cipher algorithm of this ciphersuite

std::string **mac_algo**() **const**
Return the authentication algorithm of this ciphersuite

15.8 TLS Alerts

A `TLS::Alert` is passed to every invocation of a channel's *alert_cb*.

class `TLS::`**`Alert`**

`is_valid``()` **`const`**
Return true if this alert is not a null alert

`is_fatal``()` **`const`**
Return true if this alert is fatal. A fatal alert causes the connection to be immediately disconnected. Otherwise, the alert is a warning and the connection remains valid.

Type **`type`**`()` **`const`**
Returns the type of the alert as an enum

std::string **`type_string`**`()`
Returns the type of the alert as a string

15.9 TLS Protocol Version

TLS has several different versions with slightly different behaviors. The `TLS::Protocol_Version` class represents a specific version:

class `TLS::`**`Protocol_Version`**

`enum Version_Code`
`TLS_V10`, `TLS_V11`, `TLS_V12`, `DTLS_V10`, `DTLS_V12`

`Protocol_Version``(`*Version_Code named_version*`)`
Create a specific version

uint8_t **`major_version`**`()` **`const`**
Returns major number of the protocol version

uint8_t **`minor_version`**`()` **`const`**
Returns minor number of the protocol version

std::string **`to_string`**`()` **`const`**
Returns string description of the version, for instance “TLS v1.1” or “DTLS v1.0”.

`static` *Protocol_Version* **`latest_tls_version`**`()`
Returns the latest version of the TLS protocol known to the library (currently TLS v1.2)

`static` *Protocol_Version* **`latest_dtls_version`**`()`
Returns the latest version of the DTLS protocol known to the library (currently DTLS v1.2)

CHAPTER

SIXTEEN

CREDENTIALS MANAGER

A `Credentials_Manager` is a way to abstract how the application stores credentials in a way that is usable by protocol implementations. Currently the main user is the *Transport Layer Security (TLS)* implementation.

class Credentials_Manager

std::vector<*X509_Certificate*> **trusted_certificate_authorities** (**const** std::string *&type*, **const** std::string *&context*)

Return the list of trusted certificate authorities.

When *type* is "tls-client", *context* will be the hostname of the server, or empty if the hostname is not known.

When *type* is "tls-server", the *context* will again be the hostname of the server, or empty if the client did not send a server name indicator. For TLS servers, these CAs are the ones trusted for signing of client certificates. If you do not want the TLS server to ask for a client cert, `trusted_certificate_authorities` should return an empty list for *type* "tls-server".

The default implementation returns an empty list.

std::vector<*X509_Certificate*> **cert_chain** (**const** std::vector<std::string> *&cert_key_types*, **const** std::string *&type*, **const** std::string *&context*)

Return the certificate chain to use to identify ourselves

std::vector<*X509_Certificate*> **cert_chain_single_type** (**const** std::string *&cert_key_type*, **const** std::string *&type*, **const** std::string *&context*)

Return the certificate chain to use to identifier ourselves, if we have one of type *cert_key_tye* and we would like to use a certificate in this *type/context*.

Private_Key ***private_key_for** (**const** *X509_Certificate &cert*, **const** std::string *&type*, **const** std::string *&context*)

Return the private key for this certificate. The *cert* will be the leaf cert of a chain returned previously by `cert_chain` or `cert_chain_single_type`.

In versions before 1.11.34, there was an additional function on *Credentials_Manager*

This function has been replaced by *TLS::Callbacks::tls_verify_cert_chain*.

16.1 SRP Authentication

`Credentials_Manager` contains the hooks used by TLS clients and servers for SRP authentication.

bool **attempt_srp** (**const** std::string &*type*, **const** std::string &*context*)

Returns if we should consider using SRP for authentication

std::string **srp_identifier** (**const** std::string &*type*, **const** std::string &*context*)

Returns the SRP identifier we'd like to use (used by client)

std::string **srp_password** (**const** std::string &*type*, **const** std::string &*context*, **const** std::string &*identifier*)

Returns the password for *identifier* (used by client)

bool **srp_verifier** (**const** std::string &*type*, **const** std::string &*context*, **const** std::string &*identifier*, std::string &*group_name*, BigInt &*verifier*, std::vector<uint8_t> &*salt*, bool *generate_fake_on_unknown*)

Returns the SRP verifier information for *identifier* (used by server)

16.2 Preshared Keys

TLS and some other protocols support the use of pre shared keys for authentication.

SymmetricKey **psk** (**const** std::string &*type*, **const** std::string &*context*, **const** std::string &*identity*)

Return a symmetric key for use with *identity*

One important special case for `psk` is where *type* is "tls-server", *context* is "session-ticket" and *identity* is an empty string. If a key is returned for this case, a TLS server will offer session tickets to clients who can use them, and the returned key will be used to encrypt the ticket. The server is allowed to change the key at any time (though changing the key means old session tickets can no longer be used for resumption, forcing a full re-handshake when the client next connects). One simple approach to add support for session tickets in your server is to generate a random key the first time `psk` is called to retrieve the session ticket key, cache it for later use in the `Credentials_Manager`, and simply let it be thrown away when the process terminates.

See **RFC 4507** (https://tools.ietf.org/html/rfc4507.html) for more information about TLS session tickets.

std::string **psk_identity_hint** (**const** std::string &*type*, **const** std::string &*context*)

Returns an identity hint which may be provided to the client. This can help a client understand what PSK to use.

std::string **psk_identity** (**const** std::string &*type*, **const** std::string &*context*, **const** std::string &*identity_hint*)

Returns the identity we would like to use given this *type* and *context* and the optional *identity_hint*. Not all servers or protocols will provide a hint.

CHAPTER

SEVENTEEN

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 botan/bigint.h and botan/numthry.h.

17.1 Encoding Functions

These transform the normal representation of a BigInt into some other form, such as a decimal string:

secure_vector<uint8_t> BigInt::**encode**(**const** BigInt &*n*, Encoding *enc* = Binary)

This function encodes the BigInt n into a memory vector. Encoding is an enum that has values Binary, Decimal, and Hexadecimal.

BigInt BigInt::**decode**(**const** std::vector<uint8_t> &*vec*, Encoding *enc*)

Decode the integer from vec using the encoding specified.

These functions are static member functions, so they would be called like this:

```
BigInt n1 = ...; // some number
secure_vector<uint8_t> n1_encoded = BigInt::encode(n1);
BigInt n2 = BigInt::decode(n1_encoded);
assert(n1 == n2);
```

There are also C++-style I/O operators defined for use with BigInt. The input operator understands negative numbers and hexadecimal numbers (marked with a leading "0x"). The '-' must come before the "0x" 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 an array (a byte pointer plus a length) into a BigInt using a constructor.

17.2 Number Theory

Number theoretic functions available include:

BigInt **gcd**(BigInt *x*, BigInt *y*)

Returns the greatest common divisor of x and y

BigInt **lcm**(BigInt *x*, BigInt *y*)

Returns an integer z which is the smallest integer such that z % x == 0 and z % y == 0

BigInt **inverse_mod** (BigInt *x*, BigInt *m*)

Returns the modular inverse of x modulo m, that is, an integer y such that (x*y) % m == 1. If no such y exists, returns zero.

BigInt **power_mod** (BigInt *b*, BigInt *x*, BigInt *m*)

Returns b to the xth power modulo m. If you are doing many exponentiations with a single fixed modulus, it is faster to use a `Power_Mod` implementation.

BigInt **ressol** (BigInt *x*, BigInt *p*)

Returns the square root modulo a prime, that is, returns a number y such that (y*y) % p == x. Returns -1 if no such integer exists.

bool **is_prime** (BigInt *n*, RandomNumberGenerator &*rng*, size_t *prob* = 56, double *is_random* = false)

Test *n* for primality using a probablistic algorithm (Miller-Rabin). With this algorithm, there is some non-zero probability that true will be returned even if *n* is actually composite. Modifying *prob* allows you to decrease the chance of such a false positive, at the cost of increased runtime. Sufficient tests will be run such that the chance *n* is composite is no more than 1 in 2^{prob}. Set *is_random* to true if (and only if) *n* was randomly chosen (ie, there is no danger it was chosen maliciously) as far fewer tests are needed in that case.

bool **quick_check_prime** (BigInt *n*, RandomNumberGenerator &*rng*)

bool **check_prime** (BigInt *n*, RandomNumberGenerator &*rng*)

bool **verify_prime** (BigInt *n*, RandomNumberGenerator &*rng*)

Three variations on *is_prime*, with probabilities set to 32, 56, and 80 respectively.

BigInt **random_prime** (RandomNumberGenerator &*rng*, size_t *bits*, BigInt *coprime* = 1, size_t *equiv* = 1, size_t *equiv_mod* = 2)

Return a random prime number of `bits` bits long that is relatively prime to `coprime`, and equivalent to `equiv` modulo `equiv_mod`.

CHAPTER EIGHTEEN

KEY DERIVATION FUNCTIONS

Key derivation functions are used to turn some amount of shared secret material into uniform random keys suitable for use with symmetric algorithms. An example of an input which is useful for a KDF is a shared secret created using Diffie-Hellman key agreement.

class KDF

secure_vector<uint8_t> **derive_key**(size_t *key_len*, **const** std::vector<uint8_t> &*secret*, **const** std::string &*salt* = "") **const**

secure_vector<uint8_t> **derive_key**(size_t *key_len*, **const** std::vector<uint8_t> &*secret*, **const** std::vector<uint8_t> &*salt*) **const**

secure_vector<uint8_t> **derive_key**(size_t *key_len*, **const** std::vector<uint8_t> &*secret*, **const** uint8_t **salt*, size_t *salt_len*) **const**

secure_vector<uint8_t> **derive_key**(size_t *key_len*, **const** uint8_t **secret*, size_t *secret_len*, **const** std::string &*salt*) **const**

All variations on the same theme. Deterministically creates a uniform random value from *secret* and *salt*. Typically *salt* is a lable or identifier, such as a session id.

You can create a *KDF* using

KDF ***get_kdf**(**const** std::string &*algo_spec*)

CHAPTER

NINETEEN

PBKDF ALGORITHMS

There are various procedures for turning a passphrase into a arbitrary length key for use with a symmetric cipher. A general interface for such algorithms is presented in `pbkdf.h`. The main function is `derive_key`, which takes a passphrase, a salt, an iteration count, and the desired length of the output key, and returns a key of that length, deterministically produced from the passphrase and salt. 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).

The purpose of the iteration count is to make the algorithm take longer to compute the final key (reducing the speed of brute-force attacks of various kinds). Most standards recommend an iteration count of at least 10000. Currently defined PBKDF algorithms are "PBKDF1(digest)", "PBKDF2(digest)"; you can retrieve any of these using the `get_pbkdf`, found in `lookup.h`. As of this writing, "PBKDF2(SHA-256)" with at least 100000 iterations and a 16 byte salt is recommend for new applications.

OctetString `PBKDF::`**`derive_key`**(size_t *output_len*, **`const`** std::string &*passphrase*, **`const`** uint8_t **salt*, size_t *salt_len*, size_t *iterations*) **`const`**

Computes a key from *passphrase* and the *salt* (of length *salt_len* bytes) using an algorithm-specific interpretation of *iterations*, producing a key of length *output_len*.

Use an iteration count of at least 10000. The salt should be randomly chosen by a good random number generator (see *Random Number Generators* for how), or at the very least unique to this usage of the passphrase.

If you call this function again with the same parameters, you will get the same key.

```
PBKDF* pbkdf = get_pbkdf("PBKDF2(SHA-256)");
AutoSeeded_RNG rng;

secure_vector<uint8_t> salt = rng.random_vec(16);
OctetString aes256_key = pbkdf->derive_key(32, "password",
                                           &salt[0], salt.size(),
                                           10000);
```

19.1 PBKDF1

PBKDF1 is an old scheme that can only produce an output length at most as long as the hash function. It is deprecated and will be removed in a future release.

19.2 PBKDF2

PBKDF2 is a the "standard" password derivation scheme, widely implemented in many different libraries.

19.3 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, 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.

CHAPTER

TWENTY

AES KEY WRAPPING

NIST specifies two mechanisms for wrapping (encrypting) symmetric keys using another key. The first (and older, more widely supported) methd requres the input be a multiple of 8 bytes long. The other allows any length input, though only up to 2**32 bytes.

These algorithms are described in NIST SP 800-38F, and RFCs 3394 and 5694.

This API, defined in `nist_keywrap.h`, first became available in version 2.4.0

These functions take an arbitrary 128-bit block cipher object, which must already have been keyed with the key encryption key. NIST only allows these functions with AES, but any 128-bit cipher will do and some other implementations (such as in OpenSSL) do also allow other ciphers. Use AES for best interop.

std::vector<uint8_t> **nist_key_wrap**(**const** uint8_t *input*[], size_t *input_len*, **const** *BlockCipher &bc*)

This performs KW (key wrap) mode. The input must be a multiple of 8 bytes long.

secure_vector<uint8_t> **nist_key_unwrap**(**const** uint8_t *input*[], size_t *input_len*, **const** *BlockCipher &bc*)

This unwraps the result of nist_key_wrap, or throw Integrity_Failure on error.

std::vector<uint8_t> **nist_key_wrap_padded**(**const** uint8_t *input*[], size_t *input_len*, **const** *BlockCipher &bc*)

This performs KWP (key wrap with padding) mode. The input can be any length.

secure_vector<uint8_t> **nist_key_unwrap_padded**(**const** uint8_t *input*[], size_t *input_len*, **const** *BlockCipher &bc*)

This unwraps the result of nist_key_wrap_padded, or throws Integrity_Failure on error.

20.1 RFC 3394 Interface

This is an older interface that was first available (with slight changes) in 1.10, and available in its current form since 2.0 release. It uses a 128-bit, 192-bit, or 256-bit key to encrypt an input key. AES is always used. The input must be a multiple of 8 bytes; if not an exception is thrown.

This interface is defined in `rfc3394.h`.

secure_vector<uint8_t> **rfc3394_keywrap**(**const** secure_vector<uint8_t> *&key*, **const** SymmetricKey *&kek*)

Wrap the input key using kek (the key encryption key), and return the result. It will be 8 bytes longer than the input key.

secure_vector<uint8_t> **rfc3394_keyunwrap**(**const** secure_vector<uint8_t> *&key*, **const** SymmetricKey *&kek*)

Unwrap a key wrapped wtih rfc3394_keywrap.

CHAPTER

TWENTYONE

PASSWORD HASHING

Storing passwords for user authentication purposes in plaintext is the simplest but least secure method; when an attacker compromises the database in which the passwords are stored, they immediately gain access to all of them. Often passwords are reused among multiple services or machines, meaning once a password to a single service is known an attacker has a substantial head start on attacking other machines.

The general approach is to store, instead of the password, the output of a one way function of the password. Upon receiving an authentication request, the authenticator can recompute the one way function and compare the value just computed with the one that was stored. If they match, then the authentication request succeeds. But when an attacker gains access to the database, they only have the output of the one way function, not the original password.

Common hash functions such as SHA-256 are one way, but used alone they have problems for this purpose. What an attacker can do, upon gaining access to such a stored password database, is hash common dictionary words and other possible passwords, storing them in a list. Then he can search through his list; if a stored hash and an entry in his list match, then he has found the password. Even worse, this can happen *offline*: an attacker can begin hashing common passwords days, months, or years before ever gaining access to the database. In addition, if two users choose the same password, the one way function output will be the same for both of them, which will be visible upon inspection of the database.

There are two solutions to these problems: salting and iteration. Salting refers to including, along with the password, a randomly chosen value which perturbs the one way function. Salting can reduce the effectivness of offline dictionary generation, because for each potential password, an attacker would have to compute the one way function output for all possible salts. It also prevents the same password from producing the same output, as long as the salts do not collide. Choosing n-bit salts randomly, salt collisions become likely only after about 2:sup:*(n/2)* salts have been generated. Choosing a large salt (say 80 to 128 bits) ensures this is very unlikely. Note that in password hashing salt collisions are unfortunate, but not fatal - it simply allows the attacker to attack those two passwords in parallel easier than they would otherwise be able to.

The other approach, iteration, refers to the general technique of forcing multiple one way function evaluations when computing the output, to slow down the operation. For instance if hashing a single password requires running SHA-256 100,000 times instead of just once, that will slow down user authentication by a factor of 100,000, but user authentication happens quite rarely, and usually there are more expensive operations that need to occur anyway (network and database I/O, etc). On the other hand, an attacker who is attempting to break a database full of stolen password hashes will be seriously inconvenienced by a factor of 100,000 slowdown; they will be able to only test at a rate of .0001% of what they would without iterations (or, equivalently, will require 100,000 times as many zombie botnet hosts).

Memory usage while checking a password is also a consideration; if the computation requires using a certain minimum amount of memory, then an attacker can become memory-bound, which may in particular make customized cracking hardware more expensive. Some password hashing designs, such as scrypt, explicitly attempt to provide this. The bcrypt approach requires over 4 KiB of RAM (for the Blowfish key schedule) and may also make some hardware attacks more expensive.

Botan provides two techniques for password hashing, bcrypt and passhash9.

21.1 Bcrypt

Bcrypt (https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf) is a password hashing scheme originally designed for use in OpenBSD, but numerous other implementations exist. It is made available by including `bcrypt.h`.

It has the advantage that it requires a small amount (4K) of fast RAM to compute, which can make hardware password cracking somewhat more expensive.

Bcrypt provides outputs that look like this:

```
"$2a$12$7KIYdyv8Bp32WAvc.7YvI.wvRlyVn0HP/EhPmmOyMQA4YKxINO0p2"
```

Currently only the *2a* bcrypt format is supported.

std::string **`generate_bcrypt`**(**`const`** std::string *&password*, RandomNumberGenerator *&rng*, uint16_t *work_factor* = 10)

Takes the password to hash, a rng, and a work factor. Higher work factors increase the amount of time the algorithm runs, increasing the cost of cracking attempts. The increase is exponential, so a work factor of 10 takes roughly twice as long as work factor 9.

The resulting password hash is returned as a string.

Work factor must be at least 4. The bcrypt format allows up to 31, but Botan currently rejects all work factors greater than 18 since even that work factor requires roughly 30 seconds of computation on a fast machine.

bool **`check_bcrypt`**(**`const`** std::string *&password*, **`const`** std::string *&hash*)

Takes a password and a bcrypt output and returns true if the password is the same as the one that was used to generate the bcrypt hash.

21.2 Passhash9

Botan also provides a password hashing technique called passhash9, in `passhash9.h`, which is based on PBKDF2.

Passhash9 hashes look like:

```
"$9$AAAKxwMGNPSdPkOKJS07Xutm3+1Cr3ytmbnkjO6LjHzCMcMQXvcT"
```

This function should be secure with the proper parameters, and will remain in the library for the forseeable future, but it is specific to Botan rather than being a widely used password hash. Prefer bcrypt.

> **Warning:** This password format string ("$9$") conflicts with the format used for scrypt password hashes on Cisco systems.

std::string **`generate_passhash9`**(**`const`** std::string *&password*, RandomNumberGenerator *&rng*, uint16_t *work_factor* = 10, uint8_t *alg_id* = 1)

Functions much like `generate_bcrypt`. The last parameter, `alg_id`, specifies which PRF to use. Currently defined values are 0: HMAC(SHA-1), 1: HMAC(SHA-256), 2: CMAC(Blowfish), 3: HMAC(SHA-384), 4: HMAC(SHA-512)

Currently, this performs 10000 * `work_factor` PBKDF2 iterations, using 96 bits of salt taken from `rng`. The iteration count is encoded as a 16-bit integer and is multiplied by 10000.

bool **`check_passhash9`**(**`const`** std::string *&password*, **`const`** std::string *&hash*)

Functions much like `check_bcrypt`

CHAPTER

TWENTYTWO

CRYPTOBOX

22.1 Encryption using a passphrase

New in version 1.8.6.

This is a set of simple routines that encrypt some data using a passphrase. There are defined in the header *cryptobox.h*, inside namespace *Botan::CryptoBox*.

It generates cipher and MAC keys using 8192 iterations of PBKDF2 with HMAC(SHA-512), then encrypts using Serpent in CTR mode and authenticates using a HMAC(SHA-512) mac of the ciphertext, truncated to 160 bits.

std::string **encrypt** (**const** uint8_t *input*[], size_t *input_len*, **const** std::string &*passphrase*, RandomNumberGenerator &*rng*)

Encrypt the contents using *passphrase*.

std::string **decrypt** (**const** uint8_t *input*[], size_t *input_len*, **const** std::string &*passphrase*)

Decrypts something encrypted with encrypt.

std::string **decrypt** (**const** std::string &*input*, **const** std::string &*passphrase*)

Decrypts something encrypted with encrypt.

CHAPTER

TWENTYTHREE

SECURE REMOTE PASSWORD

The library contains an implementation of the SRP6-a (http://srp.stanford.edu/design.html) password authenticated key exchange protocol in `srp6.h`.

A SRP client provides what is called a SRP *verifier* to the server. This verifier is based on a password, but the password cannot be easily derived from the verifier. Later, the client and server can perform an SRP exchange, which results in a shared key.

SRP works in a discrete logarithm group. Special parameter sets for SRP6 are defined, denoted in the library as "modp/srp/<size>", for example "modp/srp/2048".

> **Warning:** While knowledge of the verifier does not easily allow an attacker to get the raw password, they could still use the verifier to impersonate the server to the client, so verifiers should be carefully protected.

BigInt **generate_srp6_verifier**(**const** std::string &*identifier*, **const** std::string &*password*, **const** std::vector<uint8_t> &*salt*, **const** std::string &*group_id*, **const** std::string &*hash_id*)

Generates a new verifier using the specified password and salt. This is stored by the server. The salt must also be stored.

std::string **srp6_group_identifier**(**const** BigInt &*N*, **const** BigInt &*g*)

class SRP6_Server_Session

BigInt **step1**(**const** BigInt &*v*, **const** std::string &*group_id*, **const** std::string &*hash_id*, RandomNumberGenerator &*rng*)

Takes a verifier (generated by generate_srp6_verifier) along with the group_id (which must match

SymmetricKey **step2**(**const** BigInt &*A*)

Takes the parameter A generated by srp6_client_agree, and return the shared secret key.

std::pair<BigInt, SymmetricKey> **srp6_client_agree**(**const** std::string &*username*, **const** std::string &*password*, **const** std::string &*group_id*, **const** std::string &*hash_id*, **const** std::vector<uint8_t> &*salt*, **const** BigInt &*B*, RandomNumberGenerator &*rng*)

The client receives these parameters from the server, except for the username and password which are provided by the user. The parameter B is the output of *step1*.

The client agreement step outputs a shared symmetric key along with the parameter A which is returned to the server (and allows it the compute the shared key).

CHAPTER

TWENTYFOUR

PSK DATABASE

New in version 2.4.0.

Many applications need to store pre-shared keys (hereafter PSKs) for authentication purposes.

An abstract interface to PSK stores is provided in `psk_db.h`

class PSK_Database

bool **is_encrypted**() **const**
: Returns true if (at least) the PSKs themselves are encrypted. Returns false if PSKs are stored in plaintext.

std::set<std::string> **list_names**() **const**
: Return the set of valid names stored in the database, ie values for which `get` will return a value.

void **set**(**const** std::string &*name*, **const** uint8_t *psk*[], size_t *psk_len*)
: Save a PSK. If `name` already exists, the current value will be overwritten.

secure_vector<uint8_t> **get**(**const** std::string &*name*) **const**
: Return a value saved with `set`. Throws an exception if `name` doesn't exist.

void **remove**(**const** std::string &*name*)
: Remove `name` from the database. If `name` doesn't exist, ignores the request.

void **set_str**(**const** std::string &*name*, **const** std::string &*psk*)
: Like `set` but accepts the psk as a string (eg for a password).

template<typename **Alloc**>
void **set_vec**(**const** std::string &*name*, **const** std::vector<uint8_t, Alloc> &*psk*)
: Like `set` but accepting a vector.

The same header also provides a specific instantiation of `PSK_Database` which encrypts both names and PSKs. It must be subclassed to provide the storage.

class Encrypted_PSK_Database : **public** *PSK_Database*

Encrypted_PSK_Database(**const** secure_vector<uint8_t> &*master_key*)
: Initializes or opens a PSK database. The master key is used the secure the contents. It may be of any length. If encrypting PSKs under a passphrase, use a suitable key derivation scheme (such as PBKDF2) to derive the secret key. If the master key is lost, all PSKs stored are unrecoverable.

  Both names and values are encrypted using NIST key wrapping (see NIST SP800-38F) with AES-256. First the master key is used with HMAC(SHA-256) to derive two 256-bit keys, one for encrypting all names and the other to key an instance of HMAC(SHA-256). Values are each encrypted under an individual key created by hashing the encrypted name with HMAC. This associates the encrypted key with the name, and prevents an attacker with write access to the data store from taking an encrypted key associated with one entity and copying it to another entity.

Names and PSKs are both padded to the next multiple of 8 bytes, providing some obfuscation of the length.

One artifact of the names being encrypted is that is is possible to use multiple different master keys with the same underlying storage. Each master key will be responsible for a subset of the keys. An attacker who knows one of the keys will be able to tell there are other values encrypted under another key.

virtual void **kv_set**(**const** std::string *&index*, **const** std::string *&value*) = 0
Save an encrypted value. Both `index` and `value` will be non-empty base64 encoded strings.

virtual std::string **kv_get**(**const** std::string *&index*) **const** = 0
Return a value saved with `kv_set`, or return the empty string.

virtual void **kv_del**(**const** std::string *&index*) = 0
Remove a value saved with `kv_set`.

virtual std::set<std::string> **kv_get_all**() **const** = 0
Return all active names (ie values for which `kv_get` will return a non-empty string).

A subclass of `Encrypted_PSK_Database` which stores data in a SQL database is also available. This class is declared in `psk_db_sql.h`:

class Encrypted_PSK_Database_SQL : **public** *Encrypted_PSK_Database*

Encrypted_PSK_Database_SQL(**const** secure_vector<uint8_t> *&master_key*, std::shared_ptr<SQL_Database> *db*, **const** std::string *&table_name*)
Creates or uses the named table in `db`. The SQL schema of the table is `(psk_name TEXT PRIMARY KEY, psk_value TEXT)`.

CHAPTER

TWENTYFIVE

PIPE/FILTER MESSAGE PROCESSING

Note: The system described below provides a message processing system with a straightforward API. However it makes many extra memory copies and allocations than would otherwise be required, and also tends to make applications using it somewhat opaque because it is not obvious what this or that Pipe& object actually does (type of operation, number of messages output (if any!), and so on), whereas using say a HashFunction or AEAD_Mode provides a much better idea in the code of what operation is occuring.

This filter interface is no longer used within the library itself (outside a few dusty corners) and will likely not see any further major development. However it will remain included because the API is often convenient and many applications use it.

Many common uses of cryptography involve processing one or more streams of data. 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 (ie, with the same sequence of filter operations). 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.

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` 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.

Botan has about 40 filters that perform different operations on data. Here's code that uses one of them to encrypt a string with AES:

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

// 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");

secure_vector<uint8_t> c1 = pipe.read_all(0);

uint8_t c2[4096] = { 0 };
size_t 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::iostream`. Here is an example of those, using the `Bzip_Compression` filter (included as a module; if you have bzlib available, check the build instructions 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 to the sink filter, which will immediately write it to the stream; if you were to call `pipe.read_all()` after `pipe.end_msg()`, you'd get an empty vector out. This is particularly useful for cases where you are processing a large amount of data, as it means you don't have to store everything in memory at once.

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;
```

You can read from a pipe while you are still writing to it, which allows you to bound the amount of memory that is in use at any one time. A common idiom for this is:

```
pipe.start_msg();
std::vector<uint8_t> buffer(4096); // arbitrary size
while(infile.good())
   {
   infile.read((char*)&buffer[0], buffer.size());
   const size_t got_from_infile = infile.gcount();
   pipe.write(buffer, got_from_infile);

   if(infile.eof())
      pipe.end_msg();

   while(pipe.remaining() > 0)
      {
      const size_t buffered = pipe.read(buffer, buffer.size());
      outfile.write((const char*)&buffer[0], buffered);
      }
   }
if(infile.bad() || (infile.fail() && !infile.eof()))
   throw Some_Exception();
```

25.1 Fork

It is common that you might receive some data and want to perform more than one operation on it (ie, 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 or acyclic graph.

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 filter 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,
```

(continues on next page)

(continued from previous page)

```
                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`. 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, // this message gets plaintext
             new MAC_Filter("HMAC(SHA-1)", mac_key)
          )
   );

pipe.process_msg(ciphertext);
std::string plaintext = pipe.read_all_as_string(0);
secure_vector<uint8_t> 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("SHA-256"),
                   new Hash_Filter("SHA-512")),
          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 SHA-256 sum of the input in hex encoded form, and the other would contain the SHA-512 sum of the input in raw binary. In many situations you'll want to perform a sequence of operations on multiple branches of the fork; in which case, use the filter described in *Chain*.

25.2 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 four Filter pointers (they will be added in order), or with an array of filter pointers and a size_t that tells Chain how many filters 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 implicit passthrough the other version used:

```
Pipe pipe(new Fork(
              new Chain(new Hash_Filter("SHA-256"), new Hex_Encoder),
              new Hash_Filter("SHA-512")
              )
         );
```

25.3 Sources and Sinks

25.3.1 Data Sources

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

25.3.2 Data Sinks

A DataSink (in data_snk.h) is a Filter that takes arbitrary amounts of input, and produces no output. 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 DataSink``s 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(get_cipher("AES-128/CTR-BE", 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 be used.

25.4 The Pipe API

25.4.1 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 four filter pointers, 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. 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).

void Pipe::**prepend**(Filter **filter*)

Calling prepend will put the passed filter first in 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 message processed would be first encrypted, and *then* hex encoded.

void Pipe::**append**(Filter **filter*)

Like prepend, but places the filter at the end of the message flow. This doesn't always do what you expect if there is a fork.

void Pipe::**pop**()

Removes the first filter in the flow.

void Pipe::**reset**()

Removes all the filters that the pipe currently holds - it is reset to an empty/no-op state. Any data that is being retained by the pipe is retained after a reset, and reset does not affect message numbers (discussed later).

25.4.2 Giving Data to a Pipe

Input to a Pipe is delimited into messages, which can be read from independently (ie, you can read 5 bytes from one message, and then all of another message, without either read affecting any other messages).

void Pipe::**start_msg**()

Starts a new message; if a message was already running, an exception is thrown. After this function returns, you can call write.

void Pipe::**write**(**const** uint8_t **input*, size_t *length*)

void Pipe::**write**(**const** std::vector<uint8_t> &*input*)

void Pipe::**write**(**const** std::string &*input*)

void Pipe::**write**(DataSource &*input*)

void Pipe::**write**(uint8_t *input*)

All versions of write write the input into the filter sequence. If a message is not currently active, an exception is thrown.

void Pipe::**end_msg**()

End the currently active message

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`.

Pipes 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.

25.4.3 Getting Output from a Pipe

Retrieving the processed data from a pipe is a bit more complicated, for various reasons. The pipe will separate each message into a separate buffer, and you have to retrieve data from each message independently. Each of the reader functions has a final parameter that specifies what message to read from. 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).

Functions in `Pipe` related to reading include:

size_t `Pipe::`**`read`**(uint8_t **out*, size_t *len*)

Reads up to `len` bytes into `out`, and returns the number of bytes actually read.

size_t `Pipe::`**`peek`**(uint8_t **out*, size_t *len*)

Acts exactly like *read*, except the data is not actually read; the next read will return the same data.

secure_vector<uint8_t> `Pipe::`**`read_all`**()

Reads the entire message into a buffer and returns it

std::string `Pipe::`**`read_all_as_string`**()

Like `read_all`, but it returns the data as a `std::string`. No encoding is done; if the message contains raw binary, so will the string.

size_t `Pipe::`**`remaining`**()

Returns how many bytes are left in the message

Pipe::message_id `Pipe::`**`default_msg`**()

Returns the current default message number

Pipe::message_id `Pipe::`**`message_count`**()

Returns the total number of messages currently in the pipe

`Pipe::`**`set_default_msg`**(Pipe::message_id *msgno*)

Sets the 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 the message explicitly when using the output operator.

25.4.4 Pipe I/O for Unix File Descriptors

This is a 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.

25.5 Filter Catalog

This section documents most of the useful filters included in the library.

25.5.1 Keyed Filters

A few sections ago, it was mentioned that Pipe can process multiple messages, treating each of them 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 is uses keys: block cipher modes, stream ciphers, MACs, and so on. It has two functions, set_key and set_iv. Calling set_key will 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 cause an exception. You must call set_key *before* calling set_iv.

Here's a example:

```
Keyed_Filter *aes, *hmac;
Pipe pipe(new Base64_Decoder,
          // Note the assignments to the cast and hmac variables
          aes = get_cipher("AES-128/CBC", aes_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();

aes->set_key(aes_key2);
aes->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*.

25.5.2 Cipher Filters

Getting a hold of a `Filter` implementing a cipher is very easy. Make sure you're including the header `lookup.h`, and then call `get_cipher`. You will pass the return value directly into a `Pipe`. There are a couple different functions which do varying levels of initialization:

Keyed_Filter ***get_cipher**(std::string *cipher_spec*, SymmetricKey *key*, InitializationVector *iv*, Cipher_Dir *dir*)

Keyed_Filter ***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`.

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: "AES-128/CBC", "Blowfish/CTR-BE", "Serpent/XTS", and "AES-256/EAX".

"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.

25.5.3 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::**Hash_Filter**(std::string *hash*, size_t *outlen* = 0)

> This constructor creates a filter that 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 filter in the pipeline. The parameter `outlen` specifies how many bytes of the hash output 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::**MAC_Filter**(std::string *mac*, SymmetricKey *key*, size_t *outlen* = 0)

> This constructor takes a name for a mac, such as "HMAC(SHA-1)" or "CMAC(AES-128)", along with a key to use. The optional `outlen` works the same as in `Hash_Filter`.

25.5.4 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 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. 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.

You can find the declarations for these types in `hex_filt.h` and `b64_filt.h`.

25.6 Writing New Filters

The system of filters and pipes was designed in an attempt to make it as simple as possible to write new filter types. There are four functions that need to be implemented by a class deriving from `Filter`:

void `Filter::`**`write`**`(`**`const`** uint8_t **input*, size_t *length*`)`

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

void `Filter::`**`send`**`(`uint8_t **output*, size_t *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 `Filter::`**`start_msg`**`()`

Implementing this function is optional. Implement it if your filter would like to do some processing or setup at the start of each message, such as allocating a data structure.

void `Filter::`**`end_msg`**`()`

Implementing this function is optional. It is called when it has been requested that filters finish up their computations. The filter should finish up with whatever computation it is 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.

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

CHAPTER TWENTYSIX

FORMAT PRESERVING ENCRYPTION

New in version 1.9.17.

Format preserving encryption (FPE) refers to a set of techniques for encrypting data such that the ciphertext has the same format as the plaintext. For instance, you can use FPE to encrypt credit card numbers with valid checksums such that the ciphertext is also an credit card number with a valid checksum, or similiarly for bank account numbers, US Social Security numbers, or even more general mappings like English words onto other English words.

The scheme currently implemented in botan is called FE1, and described in the paper Format Preserving Encryption (https://eprint.iacr.org/2009/251) by Mihir Bellare, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. FPE is an area of ongoing standardization and it is likely that other schemes will be included in the future.

To use FE1, use these functions, from `fpe_fe1.h`:

BigInt `FPE::`**`fe1_encrypt`**(**`const`** BigInt &*n*, **`const`** BigInt &*X*, **`const`** SymmetricKey &*key*, **`const`** std::vector<uint8_t> &*tweak*)

> Encrypts the value *X* modulo the value *n* using the *key* and *tweak* specified. Returns an integer less than *n*. The *tweak* is a value that does not need to be secret that parameterizes the encryption function. For instance, if you were encrypting a database column with a single key, you could use a per-row-unique integer index value as the tweak.
>
> To encrypt an arbitrary value using FE1, you need to use a ranking method. Basically, the idea is to assign an integer to every value you might encrypt. For instance, a 16 digit credit card number consists of a 15 digit code plus a 1 digit checksum. So to encrypt a credit card number, you first remove the checksum, encrypt the 15 digit value modulo 10^{15}, and then calculate what the checksum is for the new (ciphertext) number.

BigInt `FPE::`**`fe1_decrypt`**(**`const`** BigInt &*n*, **`const`** BigInt &*X*, **`const`** SymmetricKey &*key*, **`const`** std::vector<uint8_t> &*tweak*)

> Decrypts an FE1 ciphertext produced by *`fe1_encrypt`*; the *n*, *key* and *tweak* should be the same as that provided to the encryption function. Returns the plaintext.
>
> Note that there is not any implicit authentication or checking of data, so if you provide an incorrect key or tweak the result is simply a random integer.

This example encrypts a credit card number with a valid Luhn checksum (https://en.wikipedia.org/wiki/Luhn_algorithm) to another number with the same format, including a correct checksum.

```
/*
* (C) 2014,2015 Jack Lloyd
*
* Botan is released under the Simplified BSD License (see license.txt)
*/

#include "cli.h"
#include <botan/hex.h>
```

(continues on next page)

(continued from previous page)

```
#if defined(BOTAN_HAS_FPE_FE1) && defined(BOTAN_HAS_PBKDF)

#include <botan/fpe_fe1.h>
#include <botan/pbkdf.h>

namespace Botan_CLI {

namespace {

uint8_t luhn_checksum(uint64_t cc_number)
   {
   uint8_t sum = 0;

   bool alt = false;
   while(cc_number)
      {
      uint8_t digit = cc_number % 10;
      if(alt)
         {
         digit *= 2;
         if(digit > 9)
            {
            digit -= 9;
            }
         }

      sum += digit;

      cc_number /= 10;
      alt = !alt;
      }

   return (sum % 10);
   }

bool luhn_check(uint64_t cc_number)
   {
   return (luhn_checksum(cc_number) == 0);
   }

uint64_t cc_rank(uint64_t cc_number)
   {
   // Remove Luhn checksum
   return cc_number / 10;
   }

uint64_t cc_derank(uint64_t cc_number)
   {
   for(size_t i = 0; i != 10; ++i)
      {
      if(luhn_check(cc_number * 10 + i))
         {
         return (cc_number * 10 + i);
         }
      }

   return 0;
```

(continues on next page)

(continued from previous page)

```
   }

uint64_t encrypt_cc_number(uint64_t cc_number,
                           const Botan::secure_vector<uint8_t>& key,
                           const std::vector<uint8_t>& tweak)
   {
   const Botan::BigInt n = 1000000000000000;

   const uint64_t cc_ranked = cc_rank(cc_number);

   const Botan::BigInt c = Botan::FPE::fe1_encrypt(n, cc_ranked, key, tweak);

   if(c.bits() > 50)
      {
      throw Botan::Internal_Error("FPE produced a number too large");
      }

   uint64_t enc_cc = 0;
   for(size_t i = 0; i != 7; ++i)
      {
      enc_cc = (enc_cc << 8) | c.byte_at(6 - i);
      }
   return cc_derank(enc_cc);
   }

uint64_t decrypt_cc_number(uint64_t enc_cc,
                           const Botan::secure_vector<uint8_t>& key,
                           const std::vector<uint8_t>& tweak)
   {
   const Botan::BigInt n = 1000000000000000;

   const uint64_t cc_ranked = cc_rank(enc_cc);

   const Botan::BigInt c = Botan::FPE::fe1_decrypt(n, cc_ranked, key, tweak);

   if(c.bits() > 50)
      {
      throw CLI_Error("FPE produced a number too large");
      }

   uint64_t dec_cc = 0;
   for(size_t i = 0; i != 7; ++i)
      {
      dec_cc = (dec_cc << 8) | c.byte_at(6 - i);
      }
   return cc_derank(dec_cc);
   }

}

class CC_Encrypt final : public Command
   {
   public:
      CC_Encrypt() : Command("cc_encrypt CC passphrase --tweak=") {}

      void go() override
         {
```

(continues on next page)

(continued from previous page)

```
         const uint64_t cc_number = std::stoull(get_arg("CC"));
         const std::vector<uint8_t> tweak = Botan::hex_decode(get_arg("tweak"));
         const std::string pass = get_arg("passphrase");

         std::unique_ptr<Botan::PBKDF> pbkdf(Botan::PBKDF::create("PBKDF2(SHA-256)"));
         if(!pbkdf)
            {
            throw CLI_Error_Unsupported("PBKDF", "PBKDF2(SHA-256)");
            }

         Botan::secure_vector<uint8_t> key = pbkdf->pbkdf_iterations(32, pass, tweak.
↪data(), tweak.size(), 100000);

         output() << encrypt_cc_number(cc_number, key, tweak) << "\n";
         }
   };

BOTAN_REGISTER_COMMAND("cc_encrypt", CC_Encrypt);

class CC_Decrypt final : public Command
   {
   public:
      CC_Decrypt() : Command("cc_decrypt CC passphrase --tweak=") {}

      void go() override
         {
         const uint64_t cc_number = std::stoull(get_arg("CC"));
         const std::vector<uint8_t> tweak = Botan::hex_decode(get_arg("tweak"));
         const std::string pass = get_arg("passphrase");

         std::unique_ptr<Botan::PBKDF> pbkdf(Botan::PBKDF::create("PBKDF2(SHA-256)"));
         if(!pbkdf)
            {
            throw CLI_Error_Unsupported("PBKDF", "PBKDF2(SHA-256)");
            }

         Botan::secure_vector<uint8_t> key = pbkdf->pbkdf_iterations(32, pass, tweak.
↪data(), tweak.size(), 100000);

         output() << decrypt_cc_number(cc_number, key, tweak) << "\n";
         }
   };

BOTAN_REGISTER_COMMAND("cc_decrypt", CC_Decrypt);

}

#endif // FPE && PBKDF
```

CHAPTER TWENTYSEVEN

LOSSLESS DATA COMPRESSION

Some lossless data compression algorithms are available in botan, currently all via third party libraries - these include zlib (including deflate and gzip formats), bzip2, and lzma. Support for these must be enabled at build time; you can check for them using the macros `BOTAN_HAS_ZLIB`, `BOTAN_HAS_BZIP2`, and `BOTAN_HAS_LZMA`.

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

Compression is done through the `Compression_Algorithm` and `Decompression_Algorithm` classes, both defined in *compression.h*

Compression and decompression both work in three stages: starting a message (`start`), continuing to process it (`update`), and then finally completing processing the stream (`finish`).

class Compression_Algorithm

void **start** (size_t *level*)

Initialize the compression engine. This must be done before calling `update` or `finish`. The meaning of the *level* parameter varies by the algorithm but generally takes a value between 1 and 9, with higher values implying typically better compression from and more memory and/or CPU time consumed by the compression process. The decompressor can always handle input from any compressor.

void **update** (secure_vector<uint8_t> &*buf*, size_t *offset* = 0, bool *flush* = false)

Compress the material in the in/out parameter `buf`. The leading `offset` bytes of `buf` are ignored and remain untouched; this can be useful for ignoring packet headers. If `flush` is true, the compression state is flushed, allowing the decompressor to recover the entire message up to this point without having the see the rest of the compressed stream.

class Decompression_Algorithm

void **start** ()

Initialize the decompression engine. This must be done before calling `update` or `finish`. No level is provided here; the decompressor can accept input generated by any compression parameters.

void **update** (secure_vector<uint8_t> &*buf*, size_t *offset* = 0)

Decompress the material in the in/out parameter `buf`. The leading `offset` bytes of `buf` are ignored and remain untouched; this can be useful for ignoring packet headers.

This function may throw if the data seems to be invalid.

The easiest way to get a compressor is via the functions

Compression_Algorithm ***make_compressor*** (std::string *type*)

Decompression_Algorithm ***make_decompressor**(std::string *type*)

Supported values for *type* include *zlib* (raw zlib with no checksum), *deflate* (zlib's deflate format), *gzip*, *bz2*, and *lzma*. A null pointer will be returned if the algorithm is unavailable.

To use a compression algorithm in a *Pipe* use the adaptor types *Compression_Filter* and *Decompression_Filter* from *comp_filter.h*. The constructors of both filters take a *std::string* argument (passed to *make_compressor* or *make_decompressor*), the compression filter also takes a *level* parameter. Finally both constructors have a parameter *buf_sz* which specifies the size of the internal buffer that will be used - inputs will be broken into blocks of this size. The default is 4096.

CHAPTER

TWENTYEIGHT

PKCS#11

New in version 1.11.31.

PKCS#11 is a platform-independent interface for accessing smart cards and hardware security modules (HSM). Vendors of PKCS#11 compatible devices usually provide a so called middleware or "PKCS#11 module" which implements the PKCS#11 standard. This middleware translates calls from the platform-independent PKCS#11 API to device specific calls. So application developers don't have to write smart card or HSM specific code for each device they want to support.

> **Note:** The Botan PKCS#11 interface is implemented against version v2.40 of the standard.

Botan wraps the C PKCS#11 API to provide a C++ PKCS#11 interface. This is done in two levels of abstraction: a low level API (see *Low Level API*) and a high level API (see *High Level API*). The low level API provides access to all functions that are specified by the standard. The high level API represents an object oriented approach to use PKCS#11 compatible devices but only provides a subset of the functions described in the standard.

To use the PKCS#11 implementation the `pkcs11` module has to be enabled.

> **Note:** Both PKCS#11 APIs live in the namespace `Botan::PKCS11`

28.1 Low Level API

The PKCS#11 standards committee provides header files (`pkcs11.h`, `pkcs11f.h` and `pkcs11t.h`) which define the PKCS#11 API in the C programming language. These header files could be used directly to access PKCS#11 compatible smart cards or HSMs. The external header files are shipped with Botan in version v2.4 of the standard. The PKCS#11 low level API wraps the original PKCS#11 API, but still allows to access all functions described in the standard and has the advantage that it is a C++ interface with features like RAII, exceptions and automatic memory management.

The low level API is implemented by the *`LowLevel`* class and can be accessed by including the header `botan/p11.h`.

28.1.1 Preface

All constants that belong together in the PKCS#11 standard are grouped into C++ enum classes. For example the different user types are grouped in the *UserType* enumeration:

enum class UserType : CK_USER_TYPE

> **enumerator** UserType::**SO** = CKU_SO
>
> **enumerator** UserType::**User** = CKU_USER
>
> **enumerator** UserType::**ContextSpecific** = CKU_CONTEXT_SPECIFIC

Additionally, all types that are used by the low or high level API are mapped by type aliases to more C++ like names. For instance:

using FunctionListPtr = CK_FUNCTION_LIST_PTR

C-API Wrapping

There is at least one method in the *LowLevel* class that corresponds to a PKCS#11 function. For example the *C_GetSlotList* method in the *LowLevel* class is defined as follows:

class LowLevel

> bool **C_GetSlotList** (Bbool *token_present*, SlotId **slot_list_ptr*, Ulong **count_ptr*, *ReturnValue *return_value* = ThrowException) **const**

The *LowLevel* class calls the PKCS#11 function from the function list of the PKCS#11 module:

```
CK_DEFINE_FUNCTION(CK_RV, C_GetSlotList)( CK_BBOOL tokenPresent, CK_SLOT_ID_
↪PTR pSlotList,
                                          CK_ULONG_PTR pulCount )
```

Where it makes sense there is also an overload of the *LowLevel* method to make usage easier and safer:

> bool **C_GetSlotList** (bool *token_present*, std::vector<SlotId> &*slot_ids*, *ReturnValue *return_value* = ThrowException) **const**

With this overload the user of this API just has to pass a vector of `SlotId` instead of pointers to preallocated memory for the slot list and the number of elements. Additionally, there is no need to call the method twice in order to determine the number of elements first.

Another example is the *C_InitPIN* overload:

> template<typename **Talloc**>
> bool **C_InitPIN** (SessionHandle *session*, **const** std::vector<uint8_t, TAlloc> &*pin*, *ReturnValue *return_value* = ThrowException) **const**

The templated `pin` parameter allows to pass the PIN as a `std::vector<uint8_t>` or a `secure_vector<uint8_t>`. If used with a `secure_vector` it is assured that the memory is securely erased when the `pin` object is no longer needed.

Error Handling

All possible PKCS#11 return values are represented by the enum class:

enum class ReturnValue : CK_RV

All methods of the `LowLevel` class have a default parameter `ReturnValue* return_value = ThrowException`. This parameter controls the error handling of all `LowLevel` methods. The default behaviour `return_value = ThrowException` is to throw an exception if the method does not complete successfully. If a non-`NULL` pointer is passed, `return_value` receives the return value of the PKCS#11 function and no exception is thrown. In case `nullptr` is passed as `return_value`, the exact return value is ignored and the method just returns `true` if the function succeeds and `false` otherwise.

28.1.2 Getting started

An object of this class can be instantiated by providing a `FunctionListPtr` to the `LowLevel` constructor:

explicit LowLevel (*FunctionListPtr ptr*)

The `LowLevel` class provides a static method to retrieve a `FunctionListPtr` from a PKCS#11 module file:

static bool **C_GetFunctionList** (Dynamically_Loaded_Library *&pkcs11_module*, *FunctionListPtr *function_list_ptr_ptr*, *ReturnValue *return_value* = ThrowException)

Code Example: Object Instantiation

```
Botan::Dynamically_Loaded_Library pkcs11_module( "C:\\pkcs11-
↪middleware\\library.dll" );
Botan::PKCS11::FunctionListPtr func_list = nullptr;
Botan::PKCS11::LowLevel::C_GetFunctionList( pkcs11_module, &func_list );
Botan::PKCS11::LowLevel p11_low_level( func_list );
```

Code Example: PKCS#11 Module Initialization

```
Botan::PKCS11::LowLevel p11_low_level(func_list);

Botan::PKCS11::C_InitializeArgs init_args = { nullptr, nullptr, nullptr,␣
↪nullptr,
        static_cast<CK_FLAGS>(Botan::PKCS11::Flag::OsLockingOk), nullptr };

p11_low_level.C_Initialize(&init_args);

// work with the token

p11_low_level.C_Finalize(nullptr);
```

More code examples can be found in the test suite in the `test_pkcs11_low_level.cpp` file.

28.2 High Level API

The high level API provides access to the most commonly used PKCS#11 functionality in an object oriented manner. Functionality of the high level API includes:

- Loading/unloading of PKCS#11 modules
- Initialization of tokens
- Change of PIN/SO-PIN

- Session management
- Random number generation
- Enumeration of objects on the token (certificates, public keys, private keys)
- Import/export/deletion of certificates
- Generation/import/export/deletion of RSA and EC public and private keys
- Encryption/decryption using RSA with support for OAEP and PKCS1-v1_5 (and raw)
- Signature generation/verification using RSA with support for PSS and PKCS1-v1_5 (and raw)
- Signature generation/verification using ECDSA
- Key derivation using ECDH

28.2.1 Module

The *Module* class represents a PKCS#11 shared library (module) and is defined in `botan/p11_module.h`.

It is constructed from a a file path to a PKCS#11 module and optional `C_InitializeArgs`:

class Module

```
Module(const std::string& file_path, C_InitializeArgs init_args =
   { nullptr, nullptr, nullptr, nullptr, static_cast<CK_FLAGS>(Flag::OsLockingOk),
↪ nullptr })
```

It loads the shared library and calls `C_Initialize` with the provided `C_InitializeArgs`. On destruction of the object `C_Finalize` is called.

There are two more methods in this class. One is for reloading the shared library and reinitializing the PKCS#11 module:

```
void reload(C_InitializeArgs init_args =
   { nullptr, nullptr, nullptr, nullptr, static_cast< CK_FLAGS >
↪(Flag::OsLockingOk), nullptr });
```

The other one is for getting general information about the PKCS#11 module:

Info **get_info**() **const**

This function calls `C_GetInfo` internally.

Code example:

```
Botan::PKCS11::Module module( "C:\\pkcs11-middleware\\library.dll" );

// Sometimes useful if a newly connected token is not detected by the PKCS
↪#11 module
module.reload();

Botan::PKCS11::Info info = module.get_info();

// print library version
std::cout << std::to_string( info.libraryVersion.major ) << "."
   << std::to_string( info.libraryVersion.minor ) << std::endl;
```

28.2.2 Slot

The *Slot* class represents a PKCS#11 slot and is defined in `botan/p11_slot.h`.

A PKCS#11 slot is usually a smart card reader that potentially contains a token.

class Slot

> **Slot** (*Module &module*, SlotId *slot_id*)
>
> To instantiate this class a reference to a *Module* object and a `slot_id` have to be passed to the constructor.

To retrieve available slot ids this class has the following static method:

> **static** std::vector<SlotId> **get_available_slots** (*Module &module*, bool *token_present*)
>
> The parameter `token_present` controls whether all slots or only slots with a token attached are returned by this method. This method calls *C_GetSlotList*.

Further methods:

> SlotInfo **get_slot_info** () **const**
>
> Returns information about the slot. Calls `C_GetSlotInfo`.
>
> TokenInfo **get_token_info** () **const**
>
> Obtains information about a particular token in the system. Calls `C_GetTokenInfo`.
>
> std::vector<MechanismType> **get_mechanism_list** () **const**
>
> Obtains a list of mechanism types supported by the slot. Calls `C_GetMechanismList`.
>
> MechanismInfo **get_mechanism_info** (MechanismType *mechanism_type*) **const**
>
> Obtains information about a particular mechanism possibly supported by a slot. Calls `C_GetMechanismInfo`.
>
> void **initialize** (**const** std::string *&label*, **const** secure_string *&so_pin*) **const**
>
> Calls `C_InitToken` to initialize the token. The `label` must not exceed 32 bytes. The current PIN of the security officer must be passed in `so_pin` if the token is reinitialized or if it's a factory new token, the `so_pin` that is passed will initially be set.

Code example:

```
// only slots with connected token
std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_
↪available_slots( module, true );

// use first slot
Botan::PKCS11::Slot slot( module, slots.at( 0 ) );

// print firmware version of the slot
Botan::PKCS11::SlotInfo slot_info = slot.get_slot_info();
std::cout << std::to_string( slot_info.firmwareVersion.major ) << "."
   << std::to_string( slot_info.firmwareVersion.minor ) << std::endl;

// print firmware version of the token
Botan::PKCS11::TokenInfo token_info = slot.get_token_info();
std::cout << std::to_string( token_info.firmwareVersion.major ) << "."
   << std::to_string( token_info.firmwareVersion.minor ) << std::endl;

// retrieve all mechanisms supported by the token
```

(continues on next page)

(continued from previous page)

```
std::vector<Botan::PKCS11::MechanismType> mechanisms = slot.get_mechanism_
↪list();

// retrieve information about a particular mechanism
Botan::PKCS11::MechanismInfo mech_info =
   slot.get_mechanism_info( Botan::PKCS11::MechanismType::RsaPkcsOaep );

// maximum RSA key length supported:
std::cout << mech_info.ulMaxKeySize << std::endl;

// initialize the token
Botan::PKCS11::secure_string so_pin( 8, '0' );
slot.initialize( "Botan PKCS11 documentation test label", so_pin );
```

28.2.3 Session

The *Session* class represents a PKCS#11 session and is defined in `botan/p11_session.h`.

A session is a logical connection between an application and a token.

class Session

There are two constructors to create a new session and one constructor to take ownership of an existing session.

Session (*Slot &slot*, bool *read_only*)

To initialize a session object a *Slot* has to be specified on which the session should operate. `read_only` specifies whether the session should be read only or read write. Calls `C_OpenSession`.

Session (*Slot &slot*, Flags *flags*, VoidPtr *callback_data*, Notify *notify_callback*)

Creates a new session by passing a *Slot*, session `flags`, `callback_data` and a `notify_callback`. Calls `C_OpenSession`.

Session (*Slot &slot*, SessionHandle *handle*)

Takes ownership of an existing session by passing *Slot* and a session `handle`.

The destructor calls `C_Logout` if a user is logged in to this session and always `C_CloseSession`.

SessionHandle **release** ()

Returns the released `SessionHandle`

void **login** (*UserType userType*, **const** secure_string *&pin*)

Login to this session by passing *UserType* and `pin`. Calls `C_Login`.

void **logoff** ()

Logout from this session. Not mandatory because on destruction of the *Session* object this is done automatically.

SessionInfo **get_info** () **const**

Returns information about this session. Calls `C_GetSessionInfo`.

void **set_pin** (**const** secure_string *&old_pin*, **const** secure_string *&new_pin*) **const**

Calls `C_SetPIN` to change the PIN of the logged in user using the `old_pin`.

void **init_pin** (**const** secure_string *&new_pin*)

Calls *C_InitPIN* to change or initialize the PIN using the SO_PIN (requires a logged in session).

Code example:

```
// open read only session
{
Botan::PKCS11::Session read_only_session( slot, true );
}

// open read write session
{
Botan::PKCS11::Session read_write_session( slot, false );
}

// open read write session by passing flags
{
Botan::PKCS11::Flags flags =
   Botan::PKCS11::flags( Botan::PKCS11::Flag::SerialSession |␣
↪Botan::PKCS11::Flag::RwSession );

Botan::PKCS11::Session read_write_session( slot, flags, nullptr, nullptr );
}

// move ownership of a session
{
Botan::PKCS11::Session session( slot, false );
Botan::PKCS11::SessionHandle handle = session.release();

Botan::PKCS11::Session session2( slot, handle );
}

Botan::PKCS11::Session session( slot, false );

// get session info
Botan::PKCS11::SessionInfo info = session.get_info();
std::cout << info.slotID << std::endl;

// login
Botan::PKCS11::secure_string pin = { '1', '2', '3', '4', '5', '6' };
session.login( Botan::PKCS11::UserType::User, pin );

// set pin
Botan::PKCS11::secure_string new_pin = { '6', '5', '4', '3', '2', '1' };
session.set_pin( pin, new_pin );

// logoff
session.logoff();

// log in as security officer
Botan::PKCS11::secure_string so_pin = { '0', '0', '0', '0', '0', '0', '0', '0
↪' };
session.login( Botan::PKCS11::UserType::SO, so_pin );

// change pin to old pin
session.init_pin( pin );
```

28.2.4 Objects

PKCS#11 objects consist of various attributes (CK_ATTRIBUTE). For example CKA_TOKEN describes if a PKCS#11 object is a session object or a token object. The helper class *AttributeContainer* helps with storing these attributes. The class is defined in `botan/p11_object.h`.

class AttributeContainer

Attributes can be set in an *AttributeContainer* by various `add_` methods:

> void **add_class**(ObjectClass *object_class*)
>
> Add a class attribute (`CKA_CLASS` / `AttributeType::Class`)
>
> void **add_string**(AttributeType *attribute*, **const** std::string &*value*)
>
> Add a string attribute (e.g. `CKA_LABEL` / `AttributeType::Label`).
>
> void *AttributeContainer*::**add_binary**(AttributeType *attribute*, **const** uint8_t **value*, size_t *length*)
>
> Add a binary attribute (e.g. `CKA_ID` / `AttributeType::Id`).
>
> template<typename **TAlloc**>
> void *AttributeContainer*::**add_binary**(AttributeType *attribute*, **const** std::vector<uint8_t, TAlloc> &*binary*)
>
> Add a binary attribute by passing a `vector/secure_vector` (e.g. `CKA_ID` / `AttributeType::Id`).
>
> void `AttributeContainer`::**add_bool**(AttributeType *attribute*, bool *value*)
>
> Add a bool attribute (e.g. `CKA_SENSITIVE` / `AttributeType::Sensitive`).
>
> template<typename **T**>
> void `AttributeContainer`::**add_numeric**(AttributeType *attribute*, T *value*)
>
> Add a numeric attribute (e.g. `CKA_MODULUS_BITS` / `AttributeType::ModulusBits`).

Object Properties

The PKCS#11 standard defines the mandatory and optional attributes for each object class. The mandatory and optional attribute requirements are mapped in so called property classes. Mandatory attributes are set in the constructor, optional attributes can be set via `set_` methods.

In the top hierarchy is the *ObjectProperties* class which inherits from the *AttributeContainer*. This class represents the common attributes of all PKCS#11 objects.

class ObjectProperties : **public** *AttributeContainer*

The constructor is defined as follows:

> **ObjectProperties**(ObjectClass *object_class*)
>
> Every PKCS#11 object needs an object class attribute.

The next level defines the *StorageObjectProperties* class which inherits from *ObjectProperties*.

class StorageObjectProperties : **public** *ObjectProperties*

The only mandatory attribute is the object class, so the constructor is defined as follows:

> **StorageObjectProperties**(ObjectClass *object_class*)

But in contrast to the *ObjectProperties* class there are various setter methods. For example to set the `AttributeType::Label`:

> void **set_label**(**const** std::string &*label*)
>
> Sets the label description of the object (RFC2279 string).

The remaining hierarchy is defined as follows:

- `DataObjectProperties` inherits from *StorageObjectProperties*

- `CertificateProperties` inherits from *`StorageObjectProperties`*
- `DomainParameterProperties` inherits from *`StorageObjectProperties`*
- `KeyProperties` inherits from *`StorageObjectProperties`*
- `PublicKeyProperties` inherits from `KeyProperties`
- `PrivateKeyProperties` inherits from `KeyProperties`
- `SecretKeyProperties` inherits from `KeyProperties`

PKCS#11 objects themself are represented by the *`Object`* class.

`class Object`

Following constructors are defined:

> **`Object`** (*Session &session*, ObjectHandle *handle*)
> Takes ownership over an existing object.
>
> **`Object`** (*Session &session*, **`const`** *ObjectProperties &obj_props*)
> Creates a new object with the *`ObjectProperties`* provided in `obj_props`.

The other methods are:

> secure_vector<uint8_t> **`get_attribute_value`** (AttributeType *attribute*) **`const`**
> Returns the value of the given attribute (using `C_GetAttributeValue`)
>
> void **`set_attribute_value`** (AttributeType *attribute*, **`const`** secure_vector<uint8_t> *&value*) **`const`**
> Sets the given value for the attribute (using `C_SetAttributeValue`)
>
> void **`destroy`** () **`const`**
> Destroys the object.
>
> ObjectHandle **`copy`** (**`const`** *AttributeContainer &modified_attributes*) **`const`**
> Allows to copy the object with modified attributes.

And static methods to search for objects:

> template<typename **`T`**>
> **`static`** std::vector<T> **`search`** (*Session &session*, **`const`** std::vector<Attribute> *&search_template*)
> Searches for all objects of the given type that match `search_template`.
>
> template<typename **`T`**>
> **`static`** std::vector<T> **`search`** (*Session &session*, **`const`** std::string *&label*)
> Searches for all objects of the given type using the label (`CKA_LABEL`).
>
> template<typename **`T`**>
> **`static`** std::vector<T> **`search`** (*Session &session*, **`const`** std::vector<uint8_t> *&id*)
> Searches for all objects of the given type using the id (`CKA_ID`).
>
> template<typename **`T`**>
> **`static`** std::vector<T> **`search`** (*Session &session*, **`const`** std::string *&label*, **`const`** std::vector<uint8_t> *&id*)
> Searches for all objects of the given type using the label (`CKA_LABEL`) and id (`CKA_ID`).
>
> template<typename **`T`**>
> **`static`** std::vector<T> **`search`** (*Session &session*)
> Searches for all objects of the given type.

The ObjectFinder

Another way for searching objects is to use the *ObjectFinder* class. This class manages calls to the C_FindObjects* functions: C_FindObjectsInit, C_FindObjects and C_FindObjectsFinal.

class ObjectFinder

The constructor has the following signature:

ObjectFinder (*Session &session*, **const** std::vector<Attribute> *&search_template*)
A search can be prepared with an ObjectSearcher by passing a *Session* and a search_template.

The actual search operation is started by calling the *find* method:

std::vector<ObjectHandle> **find** (std::uint32_t *max_count* = 100) **const**
Starts or continues a search for token and session objects that match a template. max_count specifies the maximum number of search results (object handles) that are returned.

void **finish** ()
Finishes the search operation manually to allow a new *ObjectFinder* to exist. Otherwise the search is finished by the destructor.

Code example:

```
// create an simple data object
Botan::secure_vector<uint8_t> value = { 0x00, 0x01 ,0x02, 0x03 };
std::size_t id = 1337;
std::string label = "test data object";

// set properties of the new object
Botan::PKCS11::DataObjectProperties data_obj_props;
data_obj_props.set_label( label );
data_obj_props.set_value( value );
data_obj_props.set_token( true );
data_obj_props.set_modifiable( true );
data_obj_props.set_object_id( Botan::DER_Encoder().encode( id ).get_contents_
↪unlocked() );

// create the object
Botan::PKCS11::Object data_obj( session, data_obj_props );

// get label of this object
Botan::PKCS11::secure_string retrieved_label =
   data_obj.get_attribute_value( Botan::PKCS11::AttributeType::Label );

// set a new label
Botan::PKCS11::secure_string new_label = { 'B', 'o', 't', 'a', 'n' };
data_obj.set_attribute_value( Botan::PKCS11::AttributeType::Label, new_label␣
↪);

// copy the object
Botan::PKCS11::AttributeContainer copy_attributes;
copy_attributes.add_string( Botan::PKCS11::AttributeType::Label, "copied␣
↪object" );
Botan::PKCS11::ObjectHandle copied_obj_handle = data_obj.copy( copy_
↪attributes );
```

(continues on next page)

(continued from previous page)

```
// search for an object
Botan::PKCS11::AttributeContainer search_template;
search_template.add_string( Botan::PKCS11::AttributeType::Label, "Botan" );
auto found_objs =
   Botan::PKCS11::Object::search<Botan::PKCS11::Object>( session, search_
↪template.attributes() );

// destroy the object
data_obj.destroy();
```

28.2.5 RSA

PKCS#11 RSA support is implemented in `<botan/p11_rsa.h>`.

RSA Public Keys

PKCS#11 RSA public keys are provided by the class *`PKCS11_RSA_PublicKey`*. This class inherits from `RSA_PublicKey` and *`Object`*. Furthermore there are two property classes defined to generate and import RSA public keys analogous to the other property classes described before: `RSA_PublicKeyGenerationProperties` and `RSA_PublicKeyImportProperties`.

`class PKCS11_RSA_PublicKey` : **`public`** RSA_PublicKey, **`public`** *Object*

`PKCS11_RSA_PublicKey` (*Session &session*, ObjectHandle *handle*)
Existing PKCS#11 RSA public keys can be used by providing an `ObjectHandle` to the constructor.

`PKCS11_RSA_PublicKey` (*Session &session*, **`const`** RSA_PublicKeyImportProperties *&pubkey_props*)
This constructor can be used to import an existing RSA public key with the `RSA_PublicKeyImportProperties` passed in `pubkey_props` to the token.

RSA Private Keys

The support for PKCS#11 RSA private keys is implemented in a similar way. There are two property classes: `RSA_PrivateKeyGenerationProperties` and `RSA_PrivateKeyImportProperties`. The *`PKCS11_RSA_PrivateKey`* class implements the actual support for PKCS#11 RSA private keys. This class inherits from `Private_Key`, `RSA_PublicKey` and *`Object`*. In contrast to the public key class there is a third constructor to generate private keys directly on the token or in the session and one method to export private keys.

`class PKCS11_RSA_PrivateKey` : **`public`** Private_Key, **`public`** RSA_PublicKey, **`public`** *Object*

`PKCS11_RSA_PrivateKey` (*Session &session*, ObjectHandle *handle*)
Existing PKCS#11 RSA private keys can be used by providing an `ObjectHandle` to the constructor.

`PKCS11_RSA_PrivateKey` (*Session &session*, **`const`** RSA_PrivateKeyImportProperties *&priv_key_props*)
This constructor can be used to import an existing RSA private key with the `RSA_PrivateKeyImportProperties` passed in `priv_key_props` to the token.

PKCS11_RSA_PrivateKey(*Session &session*, uint32_t *bits*, **const** RSA_PrivateKeyGenerationProperties *&priv_key_props*)

Generates a new PKCS#11 RSA private key with bit length provided in `bits` and the `RSA_PrivateKeyGenerationProperties` passed in `priv_key_props`.

RSA_PrivateKey **export_key**() **const**

Returns the exported `RSA_PrivateKey`.

PKCS#11 RSA key pairs can be generated with the following free function:

PKCS11_RSA_KeyPair `PKCS11::`**generate_rsa_keypair**(*Session &session*, **const** RSA_PublicKeyGenerationProperties *&pub_props*, **const** RSA_PrivateKeyGenerationProperties *&priv_props*)

Code example:

```
Botan::PKCS11::secure_string pin = { '1', '2', '3', '4', '5', '6' };
session.login( Botan::PKCS11::UserType::User, pin );

/************ import RSA private key *************/

// create private key in software
Botan::AutoSeeded_RNG rng;
Botan::RSA_PrivateKey priv_key_sw( rng, 2048 );

// set the private key import properties
Botan::PKCS11::RSA_PrivateKeyImportProperties
   priv_import_props( priv_key_sw.get_n(), priv_key_sw.get_d() );

priv_import_props.set_pub_exponent( priv_key_sw.get_e() );
priv_import_props.set_prime_1( priv_key_sw.get_p() );
priv_import_props.set_prime_2( priv_key_sw.get_q() );
priv_import_props.set_coefficient( priv_key_sw.get_c() );
priv_import_props.set_exponent_1( priv_key_sw.get_d1() );
priv_import_props.set_exponent_2( priv_key_sw.get_d2() );

priv_import_props.set_token( true );
priv_import_props.set_private( true );
priv_import_props.set_decrypt( true );
priv_import_props.set_sign( true );

// import
Botan::PKCS11::PKCS11_RSA_PrivateKey priv_key( session, priv_import_props );

/************ export PKCS#11 RSA private key *************/
Botan::RSA_PrivateKey exported = priv_key.export_key();

/************ import RSA public key *************/

// set the public key import properties
Botan::PKCS11::RSA_PublicKeyImportProperties pub_import_props( priv_key.get_
↪n(), priv_key.get_e() );
pub_import_props.set_token( true );
pub_import_props.set_encrypt( true );
pub_import_props.set_private( false );
```

(continues on next page)

(continued from previous page)

```
// import
Botan::PKCS11::PKCS11_RSA_PublicKey public_key( session, pub_import_props );

/************ generate RSA private key *************/

Botan::PKCS11::RSA_PrivateKeyGenerationProperties priv_generate_props;
priv_generate_props.set_token( true );
priv_generate_props.set_private( true );
priv_generate_props.set_sign( true );
priv_generate_props.set_decrypt( true );
priv_generate_props.set_label( "BOTAN_TEST_RSA_PRIV_KEY" );

Botan::PKCS11::PKCS11_RSA_PrivateKey private_key2( session, 2048, priv_
↪generate_props );

/************ generate RSA key pair *************/

Botan::PKCS11::RSA_PublicKeyGenerationProperties pub_generate_props( 2048UL␣
↪);
pub_generate_props.set_pub_exponent();
pub_generate_props.set_label( "BOTAN_TEST_RSA_PUB_KEY" );
pub_generate_props.set_token( true );
pub_generate_props.set_encrypt( true );
pub_generate_props.set_verify( true );
pub_generate_props.set_private( false );

Botan::PKCS11::PKCS11_RSA_KeyPair rsa_keypair =
   Botan::PKCS11::generate_rsa_keypair( session, pub_generate_props, priv_
↪generate_props );

/************ RSA encrypt *************/

Botan::secure_vector<uint8_t> plaintext = { 0x00, 0x01, 0x02, 0x03 };
Botan::PK_Encryptor_EME encryptor( rsa_keypair.first, rng, "Raw" );
auto ciphertext = encryptor.encrypt( plaintext, rng );

/************ RSA decrypt *************/

Botan::PK_Decryptor_EME decryptor( rsa_keypair.second, rng, "Raw" );
plaintext = decryptor.decrypt( ciphertext );

/************ RSA sign *************/

Botan::PK_Signer signer( rsa_keypair.second, rng, "EMSA4(SHA-256)",␣
↪Botan::IEEE_1363 );
auto signature = signer.sign_message( plaintext, rng );

/************ RSA verify *************/

Botan::PK_Verifier verifier( rsa_keypair.first, "EMSA4(SHA-256)",␣
↪Botan::IEEE_1363 );
auto ok = verifier.verify_message( plaintext, signature );
```

28.2.6 ECDSA

PKCS#11 ECDSA support is implemented in `<botan/p11_ecdsa.h>`.

ECDSA Public Keys

PKCS#11 ECDSA public keys are provided by the class *PKCS11_ECDSA_PublicKey*. This class inherits from `PKCS11_EC_PublicKey` and `ECDSA_PublicKey`. The necessary property classes are defined in `<botan/p11_ecc_key.h>`. For public keys there are `EC_PublicKeyGenerationProperties` and `EC_PublicKeyImportProperties`.

class PKCS11_ECDSA_PublicKey : **public** PKCS11_EC_PublicKey, **public virtual** ECDSA_PublicKey

PKCS11_ECDSA_PublicKey(*Session &session*, ObjectHandle *handle*)
Existing PKCS#11 ECDSA private keys can be used by providing an `ObjectHandle` to the constructor.

PKCS11_ECDSA_PublicKey(*Session &session*, **const** EC_PublicKeyImportProperties *&props*)
This constructor can be used to import an existing ECDSA public key with the `EC_PublicKeyImportProperties` passed in `props` to the token.

ECDSA_PublicKey *PKCS11_ECDSA_PublicKey*::**export_key**() **const**
Returns the exported `ECDSA_PublicKey`.

ECDSA Private Keys

The class *PKCS11_ECDSA_PrivateKey* inherits from `PKCS11_EC_PrivateKey` and implements support for PKCS#11 ECDSA private keys. There are two property classes for key generation and import: `EC_PrivateKeyGenerationProperties` and `EC_PrivateKeyImportProperties`.

class PKCS11_ECDSA_PrivateKey : **public** PKCS11_EC_PrivateKey

PKCS11_ECDSA_PrivateKey(*Session &session*, ObjectHandle *handle*)
Existing PKCS#11 ECDSA private keys can be used by providing an `ObjectHandle` to the constructor.

PKCS11_ECDSA_PrivateKey(*Session &session*, **const** EC_PrivateKeyImportProperties *&props*)
This constructor can be used to import an existing ECDSA private key with the `EC_PrivateKeyImportProperties` passed in `props` to the token.

PKCS11_ECDSA_PrivateKey(*Session &session*, **const** std::vector<uint8_t> *&ec_params*, **const** EC_PrivateKeyGenerationProperties *&props*)
This constructor can be used to generate a new ECDSA private key with the `EC_PrivateKeyGenerationProperties` passed in `props` on the token. The `ec_params` parameter is the DER-encoding of an ANSI X9.62 Parameters value.

ECDSA_PrivateKey **export_key**() **const**
Returns the exported `ECDSA_PrivateKey`.

PKCS#11 ECDSA key pairs can be generated with the following free function:

PKCS11_ECDSA_KeyPair `PKCS11`::**generate_ecdsa_keypair**(*Session &session*, **const** EC_PublicKeyGenerationProperties *&pub_props*, **const** EC_PrivateKeyGenerationProperties *&priv_props*)

Code example:

```
Botan::PKCS11::secure_string pin = { '1', '2', '3', '4', '5', '6' };
session.login( Botan::PKCS11::UserType::User, pin );

/************ import ECDSA private key *************/

// create private key in software
Botan::AutoSeeded_RNG rng;

Botan::ECDSA_PrivateKey priv_key_sw( rng, Botan::EC_Group( "secp256r1" ) );
priv_key_sw.set_parameter_encoding( Botan::EC_Group_Encoding::EC_DOMPAR_ENC_
↪OID );

// set the private key import properties
Botan::PKCS11::EC_PrivateKeyImportProperties priv_import_props(
   priv_key_sw.DER_domain(), priv_key_sw.private_value() );

priv_import_props.set_token( true );
priv_import_props.set_private( true );
priv_import_props.set_sign( true );
priv_import_props.set_extractable( true );

// label
std::string label = "test ECDSA key";
priv_import_props.set_label( label );

// import to card
Botan::PKCS11::PKCS11_ECDSA_PrivateKey priv_key( session, priv_import_props␣
↪);

/************ export PKCS#11 ECDSA private key *************/
Botan::ECDSA_PrivateKey priv_exported = priv_key.export_key();

/************ import ECDSA public key *************/

// import to card
Botan::PKCS11::EC_PublicKeyImportProperties pub_import_props( priv_key_sw.
↪DER_domain(),
   Botan::DER_Encoder().encode( EC2OSP( priv_key_sw.public_point(),␣
↪Botan::PointGFp::UNCOMPRESSED ),
   Botan::OCTET_STRING ).get_contents_unlocked() );

pub_import_props.set_token( true );
pub_import_props.set_verify( true );
pub_import_props.set_private( false );

// label
label = "test ECDSA pub key";
pub_import_props.set_label( label );

Botan::PKCS11::PKCS11_ECDSA_PublicKey public_key( session, pub_import_props␣
↪);

/************ export PKCS#11 ECDSA public key *************/
Botan::ECDSA_PublicKey pub_exported = public_key.export_key();
```

(continues on next page)

(continued from previous page)

```
/************ generate PKCS#11 ECDSA private key *************/
Botan::PKCS11::EC_PrivateKeyGenerationProperties priv_generate_props;
priv_generate_props.set_token( true );
priv_generate_props.set_private( true );
priv_generate_props.set_sign( true );

Botan::PKCS11::PKCS11_ECDSA_PrivateKey pk( session,
   Botan::EC_Group( "secp256r1" ).DER_encode( Botan::EC_Group_Encoding::EC_
↪DOMPAR_ENC_OID ),
   priv_generate_props );

/************ generate PKCS#11 ECDSA key pair *************/

Botan::PKCS11::EC_PublicKeyGenerationProperties pub_generate_props(
   Botan::EC_Group( "secp256r1" ).DER_encode(Botan::EC_Group_Encoding::EC_
↪DOMPAR_ENC_OID ) );

pub_generate_props.set_label( "BOTAN_TEST_ECDSA_PUB_KEY" );
pub_generate_props.set_token( true );
pub_generate_props.set_verify( true );
pub_generate_props.set_private( false );
pub_generate_props.set_modifiable( true );

Botan::PKCS11::PKCS11_ECDSA_KeyPair key_pair = Botan::PKCS11::generate_ecdsa_
↪keypair( session,
   pub_generate_props, priv_generate_props );

/************ PKCS#11 ECDSA sign and verify *************/

std::vector<uint8_t> plaintext( 20, 0x01 );

Botan::PK_Signer signer( key_pair.second, rng, "Raw", Botan::IEEE_1363,
↪"pkcs11" );
auto signature = signer.sign_message( plaintext, rng );

Botan::PK_Verifier token_verifier( key_pair.first, "Raw", Botan::IEEE_1363,
↪"pkcs11" );
bool ecdsa_ok = token_verifier.verify_message( plaintext, signature );
```

28.2.7 ECDH

PKCS#11 ECDH support is implemented in `<botan/p11_ecdh.h>`.

ECDH Public Keys

PKCS#11 ECDH public keys are provided by the class *PKCS11_ECDH_PublicKey*. This class inherits from `PKCS11_EC_PublicKey`. The necessary property classes are defined in `<botan/p11_ecc_key.h>`. For public keys there are `EC_PublicKeyGenerationProperties` and `EC_PublicKeyImportProperties`.

class PKCS11_ECDH_PublicKey : **public** PKCS11_EC_PublicKey

PKCS11_ECDH_PublicKey (*Session &session*, ObjectHandle *handle*)
Existing PKCS#11 ECDH private keys can be used by providing an `ObjectHandle` to the constructor.

PKCS11_ECDH_PublicKey(*Session &session*, **const** EC_PublicKeyImportProperties *&props*)
This constructor can be used to import an existing ECDH public key with the `EC_PublicKeyImportProperties` passed in `props` to the token.

ECDH_PublicKey **export_key**() **const**
Returns the exported `ECDH_PublicKey`.

ECDH Private Keys

The class *`PKCS11_ECDH_PrivateKey`* inherits from `PKCS11_EC_PrivateKey` and `PK_Key_Agreement_Key` and implements support for PKCS#11 ECDH private keys. There are two property classes. One for key generation and one for import: `EC_PrivateKeyGenerationProperties` and `EC_PrivateKeyImportProperties`.

class PKCS11_ECDH_PrivateKey : **public virtual** PKCS11_EC_PrivateKey, **public virtual** PK_Key_Agreement_K

PKCS11_ECDH_PrivateKey(*Session &session*, ObjectHandle *handle*)
Existing PKCS#11 ECDH private keys can be used by providing an `ObjectHandle` to the constructor.

PKCS11_ECDH_PrivateKey(*Session &session*, **const** EC_PrivateKeyImportProperties *&props*)
This constructor can be used to import an existing ECDH private key with the `EC_PrivateKeyImportProperties` passed in `props` to the token.

PKCS11_ECDH_PrivateKey(*Session &session*, **const** std::vector<uint8_t> *&ec_params*, **const** EC_PrivateKeyGenerationProperties *&props*)
This constructor can be used to generate a new ECDH private key with the `EC_PrivateKeyGenerationProperties` passed in `props` on the token. The `ec_params` parameter is the DER-encoding of an ANSI X9.62 Parameters value.

ECDH_PrivateKey **export_key**() **const**
Returns the exported `ECDH_PrivateKey`.

PKCS#11 ECDH key pairs can be generated with the following free function:

PKCS11_ECDH_KeyPair `PKCS11::`**generate_ecdh_keypair**(*Session &session*, **const** EC_PublicKeyGenerationProperties *&pub_props*, **const** EC_PrivateKeyGenerationProperties *&priv_props*)

Code example:

```
Botan::PKCS11::secure_string pin = { '1', '2', '3', '4', '5', '6' };
session.login( Botan::PKCS11::UserType::User, pin );

/************ import ECDH private key *************/

Botan::AutoSeeded_RNG rng;

// create private key in software
Botan::ECDH_PrivateKey priv_key_sw( rng, Botan::EC_Group( "secp256r1" ) );
priv_key_sw.set_parameter_encoding( Botan::EC_Group_Encoding::EC_DOMPAR_ENC_
↪OID );

// set import properties
```

(continues on next page)

(continued from previous page)

```
Botan::PKCS11::EC_PrivateKeyImportProperties priv_import_props(
   priv_key_sw.DER_domain(), priv_key_sw.private_value() );

priv_import_props.set_token( true );
priv_import_props.set_private( true );
priv_import_props.set_derive( true );
priv_import_props.set_extractable( true );

// label
std::string label = "test ECDH key";
priv_import_props.set_label( label );

// import to card
Botan::PKCS11::PKCS11_ECDH_PrivateKey priv_key( session, priv_import_props );

/************ export ECDH private key *************/
Botan::ECDH_PrivateKey exported = priv_key.export_key();

/************ import ECDH public key *************/

// set import properties
Botan::PKCS11::EC_PublicKeyImportProperties pub_import_props( priv_key_sw.
↪DER_domain(),
   Botan::DER_Encoder().encode( EC2OSP( priv_key_sw.public_point(),␣
↪Botan::PointGFp::UNCOMPRESSED ),
   Botan::OCTET_STRING ).get_contents_unlocked() );

pub_import_props.set_token( true );
pub_import_props.set_private( false );
pub_import_props.set_derive( true );

// label
label = "test ECDH pub key";
pub_import_props.set_label( label );

// import
Botan::PKCS11::PKCS11_ECDH_PublicKey pub_key( session, pub_import_props );

/************ export ECDH private key *************/
Botan::ECDH_PublicKey exported_pub = pub_key.export_key();

/************ generate ECDH private key *************/

Botan::PKCS11::EC_PrivateKeyGenerationProperties priv_generate_props;
priv_generate_props.set_token( true );
priv_generate_props.set_private( true );
priv_generate_props.set_derive( true );

Botan::PKCS11::PKCS11_ECDH_PrivateKey priv_key2( session,
   Botan::EC_Group( "secp256r1" ).DER_encode( Botan::EC_Group_Encoding::EC_
↪DOMPAR_ENC_OID ),
   priv_generate_props );

/************ generate ECDH key pair *************/

Botan::PKCS11::EC_PublicKeyGenerationProperties pub_generate_props(
   Botan::EC_Group( "secp256r1" ).DER_encode( Botan::EC_Group_Encoding::EC_
↪DOMPAR_ENC_OID ) );
```

(continues on next page)

(continued from previous page)

```
pub_generate_props.set_label( label + "_PUB_KEY" );
pub_generate_props.set_token( true );
pub_generate_props.set_derive( true );
pub_generate_props.set_private( false );
pub_generate_props.set_modifiable( true );

Botan::PKCS11::PKCS11_ECDH_KeyPair key_pair = Botan::PKCS11::generate_ecdh_keypair(
   session, pub_generate_props, priv_generate_props );

/************ ECDH derive *************/

Botan::PKCS11::PKCS11_ECDH_KeyPair key_pair_other = Botan::PKCS11::generate_ecdh_keypair(
   session, pub_generate_props, priv_generate_props );

Botan::PK_Key_Agreement ka( key_pair.second, rng, "Raw", "pkcs11" );
Botan::PK_Key_Agreement kb( key_pair_other.second, rng, "Raw", "pkcs11" );

Botan::SymmetricKey alice_key = ka.derive_key( 32,
   Botan::unlock( Botan::EC2OSP( key_pair_other.first.public_point(),
   Botan::PointGFp::UNCOMPRESSED ) ) );

Botan::SymmetricKey bob_key = kb.derive_key( 32,
   Botan::unlock( Botan::EC2OSP( key_pair.first.public_point(),
   Botan::PointGFp::UNCOMPRESSED ) ) );

bool eq = alice_key == bob_key;
```

28.2.8 RNG

The PKCS#11 RNG is defined in `<botan/p11_randomgenerator.h>`. The class *PKCS11_RNG* implements the `Hardware_RNG` interface.

class PKCS11_RNG : **public** Hardware_RNG

PKCS11_RNG (*Session &session*)

A PKCS#11 *Session* must be passed to instantiate a `PKCS11_RNG`.

void **randomize** (uint8_t *output*[], std::size_t *length*) **override**

Calls `C_GenerateRandom` to generate random data.

void **add_entropy** (**const** uint8_t *in*[], std::size_t *length*) **override**

Calls `C_SeedRandom` to add entropy to the random generation function of the token/middleware.

Code example:

```
Botan::PKCS11::PKCS11_RNG p11_rng( session );

/************ generate random data *************/
std::vector<uint8_t> random( 20 );
p11_rng.randomize( random.data(), random.size() );
```

(continues on next page)

(continued from previous page)

```
/************ add entropy *************/
Botan::AutoSeeded_RNG auto_rng;
auto auto_rng_random = auto_rng.random_vec( 20 );
p11_rng.add_entropy( auto_rng_random.data(), auto_rng_random.size() );

/************ use PKCS#11 RNG to seed HMAC_DRBG *************/
Botan::HMAC_DRBG drbg( Botan::MessageAuthenticationCode::create( "HMAC(SHA-
↪512)" ), p11_rng );
drbg.randomize( random.data(), random.size() );
```

28.2.9 Token Management Functions

The header file `<botan/p11.h>` also defines some free functions for token management:

void **initialize_token** (*Slot &slot*, **const** std::string *&label*, **const** secure_string *&so_pin*, **const** secure_string *&pin*)
    Initializes a token by passing a `Slot`, a `label` and the `so_pin` of the security officer.

void **change_pin** (*Slot &slot*, **const** secure_string *&old_pin*, **const** secure_string *&new_pin*)
    Change PIN with `old_pin` to `new_pin`.

void **change_so_pin** (*Slot &slot*, **const** secure_string *&old_so_pin*, **const** secure_string *&new_so_pin*)
    Change SO_PIN with `old_so_pin` to new `new_so_pin`.

void **set_pin** (*Slot &slot*, **const** secure_string *&so_pin*, **const** secure_string *&pin*)
    Sets user `pin` with `so_pin`.

Code example:

```
/************ set pin *************/

Botan::PKCS11::Module module( Middleware_path );

// only slots with connected token
std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_
↪available_slots( module, true );

// use first slot
Botan::PKCS11::Slot slot( module, slots.at( 0 ) );

Botan::PKCS11::secure_string so_pin = { '1', '2', '3', '4', '5', '6', '7', '8
↪' };
Botan::PKCS11::secure_string pin = { '1', '2', '3', '4', '5', '6' };
Botan::PKCS11::secure_string test_pin = { '6', '5', '4', '3', '2', '1' };

// set pin
Botan::PKCS11::set_pin( slot, so_pin, test_pin );

// change back
Botan::PKCS11::set_pin( slot, so_pin, pin );

/************ initialize *************/
Botan::PKCS11::initialize_token( slot, "Botan handbook example", so_pin, pin␣
↪);
```

(continues on next page)

(continued from previous page)

```
/************ change pin *************/
Botan::PKCS11::change_pin( slot, pin, test_pin );

// change back
Botan::PKCS11::change_pin( slot, test_pin, pin );

/************ change security officer pin *************/
Botan::PKCS11::change_so_pin( slot, so_pin, test_pin );

// change back
Botan::PKCS11::change_so_pin( slot, test_pin, so_pin );
```

28.2.10 X.509

The header file <botan/p11_x509.h> defines the property class X509_CertificateProperties and the class *PKCS11_X509_Certificate*.

class PKCS11_X509_Certificate : **public** *Object*, **public** *X509_Certificate*

PKCS11_X509_Certificate (*Session &session*, ObjectHandle *handle*)
Allows to use existing certificates on the token by passing a valid ObjectHandle.

PKCS11_X509_Certificate (*Session &session*, **const** X509_CertificateProperties &*props*)
Allows to import an existing X.509 certificate to the token with the X509_CertificateProperties passed in props.

Code example:

```
// load existing certificate
Botan::X509_Certificate root( "test.crt" );

// set props
Botan::PKCS11::X509_CertificateProperties props(
   Botan::DER_Encoder().encode( root.subject_dn() ).get_contents_unlocked(),␣
↪root.BER_encode() );

props.set_label( "Botan PKCS#11 test certificate" );
props.set_private( false );
props.set_token( true );

// import
Botan::PKCS11::PKCS11_X509_Certificate pkcs11_cert( session, props );

// load by handle
Botan::PKCS11::PKCS11_X509_Certificate pkcs11_cert2( session, pkcs11_cert.
↪handle() );
```

28.2.11 Tests

The PKCS#11 tests are not executed automatically because the depend on an external PKCS#11 module/middleware. The test tool has to be executed with --pkcs11-lib= followed with the path of the PKCS#11 module and a second argument which controls the PKCS#11 tests that are executed. Passing `pkcs11` will execute all PKCS#11 tests but it's also possible to execute only a subset with the following arguments:

- pkcs11-ecdh
- pkcs11-ecdsa
- pkcs11-lowlevel
- pkcs11-manage
- pkcs11-module
- pkcs11-object
- pkcs11-rng
- pkcs11-rsa
- pkcs11-session
- pkcs11-slot
- pkcs11-x509

The following PIN and SO-PIN/PUK values are used in tests:

- PIN 123456
- SO-PIN/PUK 12345678

!!! Warning !!!

Unlike the CardOS (4.4, 5.0, 5.3), the aforementioned SO-PIN/PUK is inappropriate for Gemalto (IDPrime MD 3840) cards, as it must be a byte array of length 24. For this reason some of the tests for Gemalto card involving SO-PIN will fail. You run into a risk of exceding login attempts and as a result locking your card! Currently, specifying pin via command-line option is not implemented, and therefore the desired PIN must be modified in the header src/tests/test_pkcs11.h:

Code example:

```
// SO PIN is expected to be set to "12345678" prior to running the tests
const std::string SO_PIN = "12345678";
const auto SO_PIN_SECVEC = Botan::PKCS11::secure_string(SO_PIN.begin(), SO_
↪PIN.end());
```

28.2.12 Tested/Supported Smartcards

You are very welcome to contribute your own test results for other testing environments or other cards.

Test results

Smartcard	Status	OS	Midleware	Botan	Errors
CardOS 4.4	mostly works	Windows 10, 64-bit, version 1709	API Version 5.4.9.77 (Cryptoki v2.11)	2.4.0, Cryptoki v2.40	[50]
CardOS 5.0	mostly works	Windows 10, 64-bit, version 1709	API Version 5.4.9.77 (Cryptoki v2.11)	2.4.0, Cryptoki v2.40	[51]
CardOS 5.3	mostly works	Windows 10, 64-bit, version 1709	API Version 5.4.9.77 (Cryptoki v2.11)	2.4.0, Cryptoki v2.40	[52]
Gemalto ID-Prime MD 3840	mostly works	Windows 10, 64-bit, version 1709	IDGo 800, v1.2.4 (Cryptoki v2.20)	2.4.0, Cryptoki v2.40	[53]
SoftHSM 2.3.0 (OpenSSL 1.0.2g)	works	Windows 10, 64-bit, version 1709	Cryptoki v2.40	2.4.0, Cryptoki v2.40	

[20] Test fails due to unsupported copy function (CKR_FUNCTION_NOT_SUPPORTED)
[21] Generating private key for extraction with property extractable fails (CKR_ARGUMENTS_BAD)
[22] Generate rsa private key operation fails (CKR_TEMPLATE_INCOMPLETE)
[23] Raw RSA sign-verify fails (CKR_MECHANISM_INVALID)
[3] CKR_MECHANISM_INVALID (0x70=112)
[2] CKR_ARGUMENTS_BAD (0x7=7)
[4] CKR_FUNCTION_NOT_SUPPORTED (0x54=84)
[5] CKR_RANDOM_SEED_NOT_SUPPORTED (0x120=288)
[50] Failing operations for CardOS 4.4:

- object_copy[20]
- rsa_privkey_export[21]
- rsa_generate_private_key[22]
- rsa_sign_verify[23]
- ecdh_privkey_import[3]
- ecdh_privkey_export[2]
- ecdh_pubkey_import[4]
- ecdh_pubkey_export[4]
- ecdh_generate_private_key[3]
- ecdh_generate_keypair[3]
- ecdh_derive[3]
- ecdsa_privkey_import[3]
- ecdsa_privkey_export[2]
- ecdsa_pubkey_import[4]
- ecdsa_pubkey_export[4]
- ecdsa_generate_private_key[3]
- ecdsa_generate_keypair[3]
- ecdsa_sign_verify[3]
- rng_add_entropy[5]

[32] Invalid argument OS2ECP: Unknown format type 155
[33] Invalid argument OS2ECP: Unknown format type 92
[30] Invalid argument Decoding error: BER: Value truncated
[51] Failing operations for CardOS 5.0

- object_copy[20]
- rsa_privkey_export[21]
- rsa_generate_private_key[22]

Error descriptions

- rsa_sign_verify[23]
- ecdh_privkey_export[2]
- ecdh_pubkey_import[4]
- ecdh_generate_private_key[32]
- ecdh_generate_keypair[3]
- ecdh_derive[33]
- ecdsa_privkey_export[2]
- ecdsa_generate_private_key[30]
- ecdsa_generate_keypair[30]
- ecdsa_sign_verify[30]
- rng_add_entropy[5]

[6] CKM_X9_42_DH_KEY_PAIR_GEN | CKR_DEVICE_ERROR (0x30=48)

[31] Invalid argument Decoding error: BER: Length field is to large

[34] Invalid argument OS2ECP: Unknown format type 57

[52] Failing operations for CardOS 5.3

- object_copy[20]
- rsa_privkey_export[21]
- rsa_generate_private_key[22]
- rsa_sign_verify[23]
- ecdh_privkey_export[2]
- ecdh_pubkey_import[6]
- ecdh_pubkey_export[6]
- ecdh_generate_private_key[30]
- ecdh_generate_keypair[31]
- ecdh_derive[30]
- ecdsa_privkey_export[2]
- ecdsa_pubkey_import[6]
- ecdsa_pubkey_export[6]
- ecdsa_generate_private_key[31]
- ecdsa_generate_keypair[31]
- ecdsa_sign_verify[34]
- rng_add_entropy[5]

[7] CKR_TEMPLATE_INCONSISTENT (0xD1=209)

[8] CKR_ENCRYPTED_DATA_INVALID | CKM_SHA256_RSA_PKCS (0x40=64)

[53] Failing operations for Gemalto IDPrime MD 3840

- session_login_logout[2]
- session_info[2]
- set_pin[2]
- initialize[2]
- change_so_pin[2]
- object_copy[20]
- rsa_generate_private_key[7]
- rsa_encrypt_decrypt[8]
- rsa_sign_verify[2]
- rng_add_entropy[5]

CHAPTER TWENTYNINE

TRUSTED PLATFORM MODULE (TPM)

New in version 1.11.26.

Some computers come with a TPM, which is a small side processor which can perform certain operations which include RSA key generation and signing, a random number generator, accessing a small amount of NVRAM, and a set of PCRs which can be used to measure software state (this is TPMs most famous use, for authenticating a boot sequence).

The TPM NVRAM and PCR APIs are not supported by Botan at this time, patches welcome.

Currently only v1.2 TPMs are supported, and the only TPM library supported is TrouSerS (http://trousers.sourceforge.net/). Hopefully both of these limitations will be removed in a future release, in order to support newer TPM v2.0 systems. The current code has been tested with an ST TPM running in a Lenovo laptop.

Test for TPM support with the macro `BOTAN_HAS_TPM`, include `<botan/tpm.h>`.

First, create a connection to the TPM with a `TPM_Context`. The context is passed to all other TPM operations, and should remain alive as long as any other TPM object which the context was passed to is still alive, otherwise errors or even an application crash are possible. In the future, the API may change to using `shared_ptr` to remove this problem.

class TPM_Context

TPM_Context (pin_cb *cb*, **const** char **srk_password*)

The (somewhat improperly named) pin_cb callback type takes a std::string as an argument, which is an informative message for the user. It should return a string containing the password entered by the user.

Normally the SRK password is null. Use nullptr to signal this.

The TPM contains a RNG of unknown design or quality. If that doesn't scare you off, you can use it with `TPM_RNG` which implements the standard `RandomNumberGenerator` interface.

class TPM_RNG

TPM_RNG (*TPM_Context &ctx*)

Initialize a TPM RNG object. After initialization, reading from this RNG reads from the hardware? RNG on the TPM.

The v1.2 TPM uses only RSA, but because this key is implemented completely in hardware it uses a different private key type, with a somewhat different API to match the TPM's behavior.

class TPM_PrivateKey

TPM_PrivateKey (*TPM_Context &ctx*, size_t *bits*, **const** char **key_password*)

Create a new RSA key stored on the TPM. The bits should be either 1024 or 2048; the TPM interface hypothetically allows larger keys but in practice no v1.2 TPM hardware supports them.

The TPM processor is not fast, be prepared for this to take a while.

The key_password is the password to the TPM key ?

std::string **register_key**(TPM_Storage_Type *storage_type*)

Registers a key with the TPM. The storage_type can be either *TPM_Storage_Type::User* or *TPM_Storage_Type::System*. If System, the key is stored on the TPM itself. If User, it is stored on the local hard drive in a database maintained by an intermediate piece of system software (which actual interacts with the physical TPM on behalf of any number of applications calling the TPM API).

The TPM has only some limited space to store private keys and may reject requests to store the key.

In either case the key is encrypted with an RSA key which was generated on the TPM and which it will not allow to be exported. Thus (so goes the theory) without physically attacking the TPM

Returns a UUID which can be passed back to constructor below.

TPM_PrivateKey(*TPM_Context &ctx*, **const** std::string *&uuid*, TPM_Storage_Type *storage_type*)

Load a registered key. The UUID was returned by the `register_key` function.

std::vector<uint8_t> **export_blob**() **const**

Export the key as an encrypted blob. This blob can later be presented back to the same TPM to load the key.

TPM_PrivateKey(*TPM_Context &ctx*, **const** std::vector<uint8_t> *&blob*)

Load a TPM key previously exported as a blob with `export_blob`.

std::unique_ptr<Public_Key> **public_key**() **const**

Return the public key associated with this TPM private key.

TPM does not store public keys, nor does it support signature verification.

TSS_HKEY **handle**() **const**

Returns the bare TSS key handle. Use if you need to call the raw TSS API.

A `TPM_PrivateKey` can be passed to a `PK_Signer` constructor and used to sign messages just like any other key. Only PKCS #1 v1.5 signatures are supported by the v1.2 TPM.

std::vector<std::string> *TPM_PrivateKey*::**registered_keys**(*TPM_Context &ctx*)

This static function returns the list of all keys (in URL format) registered with the system

CHAPTER

THIRTY

ONE TIME PASSWORDS

One time password schemes are a user authentication method that relies on a fixed secret key which is used to derive a sequence of short passwords, each of which is accepted only once. Commonly this is used to implement two-factor authentication (2FA), where the user authenticates using both a conventional password (or a public key signature) and an OTP generated by a small device such as a mobile phone.

Botan implements the HOTP and TOTP schemes from RFC 4226 and 6238.

Since the range of possible OTPs is quite small, applications must rate limit OTP authentication attempts to some small number per second. Otherwise an attacker could quickly try all 1000000 6-digit OTPs in a brief amount of time.

30.1 HOTP

HOTP generates OTPs that are a short numeric sequence, between 6 and 8 digits (most applications use 6 digits), created using the HMAC of a 64-bit counter value. If the counter ever repeats the OTP will also repeat, thus both parties must assure the counter only increments and is never repeated or decremented. Thus both client and server must keep track of the next counter expected.

Anyone with access to the client-specific secret key can authenticate as that client, so it should be treated with the same security consideration as would be given to any other symmetric key or plaintext password.

class HOTP

Implement counter-based OTP

HOTP (**const** SymmetricKey &*key*, **const** std::string &*hash_algo* = "SHA-1", size_t *digits* = 6)

Initialize an HOTP instance with a secret key (specific to each client), a hash algorithm (must be SHA-1, SHA-256, or SHA-512), and the number of digits with each OTP (must be 6, 7, or 8).

In RFC 4226, HOTP is only defined with SHA-1, but many HOTP implementations support SHA-256 as an extension. The collision attacks on SHA-1 do not have any known effect on HOTP's security.

uint32_t **generate_hotp** (uint64_t *counter*)

Return the OTP assosciated with a specific counter value.

std::pair<bool, uint64_t> **verify_hotp** (uint32_t *otp*, uint64_t *starting_counter*, size_t *resync_range* = 0)

Check if a provided OTP matches the one that should be generated for the specified counter.

The *starting_counter* should be the counter of the last successful authentication plus 1. If *resync_resync* is greater than 0, some number of counter values above *starting_counter* will also be checked if necessary. This is useful for instance when a client mistypes an OTP on entry; the authentication will fail so the server will not update its counter, but the client device will subsequently show the OTP for the next counter. Depending on the environment a *resync_range* of 3 to 10 might be appropriate.

Returns a pair of (is_valid,next_counter_to_use). If the OTP is invalid then always returns (false,starting_counter), since the last successful authentication counter has not changed.

CHAPTER

THIRTYONE

FFI (C89) INTERFACE

New in version 1.11.14.

Botan's ffi module provides a C89 API intended to be easily usable with other language's foreign function interface (FFI) libraries. For instance the included Python wrapper uses Python's `ctypes` module and the C89 API. This API is of course also useful for programs written directly in C.

Code examples can be found in *src/tests/test_ffi.cpp*.

31.1 Versioning

uint32_t **`botan_ffi_api_version`**()

Returns the version of the currently supported FFI API. This is expressed in the form YYYYMMDD of the release date of this version of the API.

int **`botan_ffi_supports_api`**(uint32_t *version*)

Returns 0 iff the FFI version specified is supported by this library. Otherwise returns -1. The expression botan_ffi_supports_api(botan_ffi_api_version()) will always evaluate to 0. A particular version of the library may also support other (older) versions of the FFI API.

`const` char ***`botan_version_string`**()

Returns a free-from version string, e.g., 2.0.0

uint32_t **`botan_version_major`**()

Returns the major version of the library

uint32_t **`botan_version_minor`**()

Returns the minor version of the library

uint32_t **`botan_version_patch`**()

Returns the patch version of the library

uint32_t **`botan_version_datestamp`**()

Returns the date this version was released as an integer, or 0 if an unreleased version

31.2 Utility Functions

int **`botan_same_mem`**(**`const`** uint8_t **x*, **`const`** uint8_t **y*, size_t *len*)

Returns 0 if *x[0..len]* == *y[0..len]*, -1 otherwise.

int **`botan_hex_encode`**(**`const`** uint8_t **x*, size_t *len*, char **out*, uint32_t *flags*)

Performs hex encoding of binary data in *x* of size *len* bytes. The output buffer *out* must be of at least *x*2* bytes in size. If *flags* contains `BOTAN_FFI_HEX_LOWER_CASE`, hex encoding will only contain lower-case letters, upper-case letters otherwise. Returns 0 on success, 1 otherwise.

31.3 Random Number Generators

typedef opaque ***botan_rng_t**

An opaque data type for a random number generator. Don't mess with it.

int **botan_rng_init** (*botan_rng_t *rng*, **const** char **rng_type*)

Initialize a random number generator object from the given *rng_type*: "system" or *nullptr*: *System_RNG*, "user": *AutoSeeded_RNG*.

int **botan_rng_get** (*botan_rng_t rng*, uint8_t **out*, size_t *out_len*)

Get random bytes from a random number generator.

int **botan_rng_reseed** (*botan_rng_t rng*, size_t *bits*)

Reseeds the random number generator with *bits* number of bits from the *System_RNG*.

int **botan_rng_destroy** (*botan_rng_t rng*)

Destroy the object created by `botan_rng_init`.

31.4 Block Ciphers

This is a 'raw' interface to ECB mode block ciphers. Most applications want the higher level cipher API which provides authenticated encryption. This API exists as an escape hatch for applications which need to implement custom primitives using a PRP.

typedef opaque ***botan_block_cipher_t**

An opaque data type for a block cipher. Don't mess with it.

int **botan_block_cipher_init** (*botan_block_cipher_t *bc*, **const** char **cipher_name*)

Create a new cipher mode object, *cipher_name* should be for example "AES-128" or "Threefish-512"

int **botan_block_cipher_block_size** (*botan_block_cipher_t bc*)

Return the block size of this cipher.

int **botan_block_cipher_clear** (*botan_block_cipher_t bc*)

Clear the internal state (such as keys) of this cipher object, but do not deallocate it.

int **botan_block_cipher_set_key** (*botan_block_cipher_t bc*, **const** uint8_t *key*[], size_t *key_len*)

Set the cipher key, which is required before encrypting or decrypting.

int **botan_block_cipher_encrypt_blocks** (*botan_block_cipher_t bc*, **const** uint8_t *in*[], uint8_t *out*[], size_t *blocks*)

The key must have been set first with `botan_block_cipher_set_key`. Encrypt *blocks* blocks of data stored in *in* and place the ciphertext into *out*. The two parameters may be the same buffer, but must not overlap.

int **botan_block_cipher_decrypt_blocks** (*botan_block_cipher_t bc*, **const** uint8_t *in*[], uint8_t *out*[], size_t *blocks*)

The key must have been set first with `botan_block_cipher_set_key`. Decrypt *blocks* blocks of data stored in *in* and place the ciphertext into *out*. The two parameters may be the same buffer, but must not overlap.

int **botan_block_cipher_destroy** (*botan_block_cipher_t rng*)

Destroy the object created by `botan_block_cipher_init`.

31.5 Hash Functions

typedef opaque ***botan_hash_t**

An opaque data type for a hash. Don't mess with it.

botan_hash_t **botan_hash_init** (**const** char **hash*, uint32_t *flags*)

Creates a hash of the given name, e.g., "SHA-384". Returns null on failure. Flags should always be zero in this version of the API.

int **botan_hash_destroy** (*botan_hash_t hash*)

Destroy the object created by `botan_hash_init`.

int **botan_hash_copy_state** (*botan_hash_t *dest*, **const** *botan_hash_t source*)

Copies the state of the hash object to a new hash object.

int **botan_hash_clear** (*botan_hash_t hash*)

Reset the state of this object back to clean, as if no input has been supplied.

size_t **botan_hash_output_length** (*botan_hash_t hash*)

Return the output length of the hash function.

int **botan_hash_update** (*botan_hash_t hash*, **const** uint8_t **input*, size_t *len*)

Add input to the hash computation.

int **botan_hash_final** (*botan_hash_t hash*, uint8_t *out*[])

Finalize the hash and place the output in out. Exactly `botan_hash_output_length` bytes will be written.

31.6 Message Authentication Codes

typedef opaque ***botan_mac_t**

An opaque data type for a MAC. Don't mess with it, but do remember to set a random key first.

botan_mac_t **botan_mac_init** (**const** char **mac*, uint32_t *flags*)

Creates a MAC of the given name, e.g., "HMAC(SHA-384)". Returns null on failure. Flags should always be zero in this version of the API.

int **botan_mac_destroy** (*botan_mac_t mac*)

Destroy the object created by `botan_mac_init`.

int **botan_mac_clear** (*botan_mac_t mac*)

Reset the state of this object back to clean, as if no key and input have been supplied.

size_t **botan_mac_output_length** (*botan_mac_t mac*)

Return the output length of the MAC.

int **botan_mac_set_key** (*botan_mac_t mac*, **const** uint8_t **key*, size_t *key_len*)

Set the random key.

int **botan_mac_update** (*botan_mac_t mac*, uint8_t *buf*[], size_t *len*)

Add input to the MAC computation.

int **botan_mac_final** (*botan_mac_t mac*, uint8_t *out*[], size_t **out_len*)

Finalize the MAC and place the output in out. Exactly `botan_mac_output_length` bytes will be written.

31.7 Ciphers

typedef opaque ***botan_cipher_t**

An opaque data type for a MAC. Don't mess with it, but do remember to set a random key first. And please use an AEAD.

botan_cipher_t **botan_cipher_init** (**const** char **cipher_name*, uint32_t *flags*)

Create a cipher object from a name such as "AES-256/GCM" or "Serpent/OCB".

Flags is a bitfield The low bit of flags specifies if encrypt or decrypt

int **botan_cipher_destroy** (*botan_cipher_t cipher*)

int **botan_cipher_clear** (*botan_cipher_t hash*)

int **botan_cipher_set_key** (*botan_cipher_t cipher*, **const** uint8_t **key*, size_t *key_len*)

int **botan_cipher_set_associated_data** (*botan_cipher_t cipher*, **const** uint8_t **ad*, size_t *ad_len*)

int **botan_cipher_start** (*botan_cipher_t cipher*, **const** uint8_t **nonce*, size_t *nonce_len*)

int **botan_cipher_is_authenticated** (*botan_cipher_t cipher*)

size_t **botan_cipher_tag_size** (*botan_cipher_t cipher*)

int **botan_cipher_valid_nonce_length** (*botan_cipher_t cipher*, size_t *nl*)

size_t **botan_cipher_default_nonce_length** (*botan_cipher_t cipher*)

31.8 PBKDF

int **botan_pbkdf** (**const** char **pbkdf_algo*, uint8_t *out*[], size_t *out_len*, **const** char **passphrase*, **const** uint8_t *salt*[], size_t *salt_len*, size_t *iterations*)

Derive a key from a passphrase for a number of iterations using the given PBKDF algorithm, e.g., "PBKDF2".

int **botan_pbkdf_timed** (**const** char **pbkdf_algo*, uint8_t *out*[], size_t *out_len*, **const** char **passphrase*, **const** uint8_t *salt*[], size_t *salt_len*, size_t *milliseconds_to_run*, size_t **out_iterations_used*)

Derive a key from a passphrase using the given PBKDF algorithm, e.g., "PBKDF2". If *out_iterations_used* is zero, instead the PBKDF is run until *milliseconds_to_run* milliseconds have passed. In this case, the number of iterations run will be written to *out_iterations_used*.

31.9 KDF

int **botan_kdf** (**const** char **kdf_algo*, uint8_t *out*[], size_t *out_len*, **const** uint8_t *secret*[], size_t *secret_len*, **const** uint8_t *salt*[], size_t *salt_len*, **const** uint8_t *label*[], size_t *label_len*)

Derive a key using the given KDF algorithm, e.g., "SP800-56C". The derived key of length *out_len* bytes will be placed in *out*.

31.10 Multiple Precision Integers

typedef opaque ***botan_mp_t**

An opaque data type for a multiple precision integer. Don't mess with it.

int **botan_mp_init** (botan_mp_t *mp)
: Initialize a botan_mp_t. Initial value is zero, use botan_mp_set_X to load a value.

int **botan_mp_destroy** (botan_mp_t mp)
: Free a botan_mp_t

int **botan_mp_to_hex** (botan_mp_t mp, char *out)
: Writes exactly botan_mp_num_bytes(mp)*2 + 1 bytes to out

int **botan_mp_to_str** (botan_mp_t mp, uint8_t base, char *out, size_t *out_len)
: Base can be either 10 or 16.

int **botan_mp_set_from_int** (botan_mp_t mp, int initial_value)
: Set botan_mp_t from an integer value.

int **botan_mp_set_from_mp** (botan_mp_t dest, botan_mp_t source)
: Set botan_mp_t from another MP.

int **botan_mp_set_from_str** (botan_mp_t dest, **const** char *str)
: Set botan_mp_t from a string. Leading prefix of “0x” is accepted.

int **botan_mp_num_bits** (botan_mp_t n, size_t *bits)
: Return the size of n in bits.

int **botan_mp_num_bytes** (botan_mp_t n, size_t *uint8_ts)
: Return the size of n in bytes.

int **botan_mp_to_bin** (botan_mp_t mp, uint8_t vec[])
: Writes exactly botan_mp_num_bytes(mp) to vec.

int **botan_mp_from_bin** (botan_mp_t mp, **const** uint8_t vec[], size_t vec_len)
: Loads botan_mp_t from a binary vector (as produced by botan_mp_to_bin).

int **botan_mp_is_negative** (botan_mp_t mp)
: Return 1 if mp is negative, otherwise 0.

int **botan_mp_flip_sign** (botan_mp_t mp)
: Flip the sign of mp.

int **botan_mp_add** (botan_mp_t result, botan_mp_t x, botan_mp_t y)
: Add two botan_mp_t``s and store the output in ``result.

int **botan_mp_sub** (botan_mp_t result, botan_mp_t x, botan_mp_t y)
: Subtract two botan_mp_t``s and store the output in ``result.

int **botan_mp_mul** (botan_mp_t result, botan_mp_t x, botan_mp_t y)
: Multiply two botan_mp_t``s and store the output in ``result.

int **botan_mp_div** (botan_mp_t quotient, botan_mp_t remainder, botan_mp_t x, botan_mp_t y)
: Divide x by y and store the output in quotient and remainder.

int **botan_mp_mod_mul** (botan_mp_t result, botan_mp_t x, botan_mp_t y, botan_mp_t mod)
: Set result to x times y modulo mod.

int **botan_mp_equal** (botan_mp_t x, botan_mp_t y)
: Return 1 if x is equal to y, 0 if x is not equal to y

int **botan_mp_is_zero** (**const** botan_mp_t x)
: Return 1 if x is equal to zero, otherwise 0.

int **botan_mp_is_odd** (**const** botan_mp_t x)
: Return 1 if x is odd, otherwise 0.

int **botan_mp_is_even** (**const** *botan_mp_t x*)
Return 1 if `x` is even, otherwise 0.

int **botan_mp_is_positive** (**const** *botan_mp_t x*)
Return 1 if `x` is greater than or equal to zero.

int **botan_mp_is_negative** (**const** *botan_mp_t x*)
Return 1 if `x` is less than zero.

int **botan_mp_to_uint32** (**const** *botan_mp_t x*, uint32_t **val*)
If x fits in a 32-bit integer, set val to it and return 0. If x is out of range an error is returned.

int **botan_mp_cmp** (int **result*, *botan_mp_t x*, *botan_mp_t y*)
Three way comparison: set result to -1 if `x` is less than `y`, 0 if `x` is equal to `y`, and 1 if `x` is greater than `y`.

int **botan_mp_swap** (*botan_mp_t x*, *botan_mp_t y*)
Swap two `botan_mp_t` values.

int **botan_mp_powmod** (*botan_mp_t out*, *botan_mp_t base*, *botan_mp_t exponent*, *botan_mp_t modulus*)
Modular exponentiation.

int **botan_mp_lshift** (*botan_mp_t out*, *botan_mp_t in*, size_t *shift*)
Left shift by specified bit count, place result in `out`.

int **botan_mp_rshift** (*botan_mp_t out*, *botan_mp_t in*, size_t *shift*)
Right shift by specified bit count, place result in `out`.

int **botan_mp_mod_inverse** (*botan_mp_t out*, *botan_mp_t in*, *botan_mp_t modulus*)
Compute modular inverse. If no modular inverse exists (for instance because `in` and `modulus` are not relatively prime), then sets `out` to -1.

int **botan_mp_rand_bits** (*botan_mp_t rand_out*, *botan_rng_t rng*, size_t *bits*)
Create a random `botan_mp_t` of the specified bit size.

int **botan_mp_rand_range** (*botan_mp_t rand_out*, *botan_rng_t rng*, *botan_mp_t lower_bound*, *botan_mp_t upper_bound*)
Create a random `botan_mp_t` within the provided range.

int **botan_mp_gcd** (*botan_mp_t out*, *botan_mp_t x*, *botan_mp_t y*)
Compute the greatest common divisor of `x` and `y`.

int **botan_mp_is_prime** (*botan_mp_t n*, *botan_rng_t rng*, size_t *test_prob*)
Test if `n` is prime. The algorithm used (Miller-Rabin) is probabilistic, set `test_prob` to the desired assurance level. For example if `test_prob` is 64, then sufficient Miller-Rabin iterations will run to assure there is at most a `1/2**64` chance that `n` is composite.

int **botan_mp_get_bit** (*botan_mp_t n*, size_t *bit*)
Returns 0 if the specified bit of `n` is not set, 1 if it is set.

int **botan_mp_set_bit** (*botan_mp_t n*, size_t *bit*)
Set the specified bit of `n`

int **botan_mp_clear_bit** (*botan_mp_t n*, size_t *bit*)
Clears the specified bit of `n`

31.11 Password Hashing

int **botan_bcrypt_generate** (uint8_t **out*, size_t **out_len*, **const** char **password*, *botan_rng_t rng*, size_t *work_factor*, uint32_t *flags*)
Create a password hash using Bcrypt. The output buffer *out* should be of length 64 bytes. The output is formatted bcrypt $2a$…

int **botan_bcrypt_is_valid**(**const** char *pass, **const** char *hash)

Check a previously created password hash. Returns 0 if if this password/hash combination is valid, 1 if the combination is not valid (but otherwise well formed), negative on error.

31.12 Public Key Creation, Import and Export

typedef opaque ***botan_privkey_t**

An opaque data type for a private key. Don't mess with it.

int **botan_privkey_create**(botan_privkey_t *key, **const** char *algo_name, **const** char *algo_params, botan_rng_t rng)

int **botan_privkey_create_rsa**(botan_privkey_t *key, botan_rng_t rng, size_t n_bits)

int **botan_privkey_create_ecdsa**(botan_privkey_t *key, botan_rng_t rng, **const** char *params)

int **botan_privkey_create_ecdh**(botan_privkey_t *key, botan_rng_t rng, **const** char *params)

int **botan_privkey_create_mceliece**(botan_privkey_t *key, botan_rng_t rng, size_t n, size_t t)

int **botan_privkey_create_dh**(botan_privkey_t *key, botan_rng_t rng, **const** char *params)

int **botan_privkey_load**(botan_privkey_t *key, botan_rng_t rng, **const** uint8_t bits[], size_t len, **const** char *password)

int **botan_privkey_destroy**(botan_privkey_t key)

int **botan_privkey_export**(botan_privkey_t key, uint8_t out[], size_t *out_len, uint32_t flags)

int **botan_privkey_export_encrypted**(botan_privkey_t key, uint8_t out[], size_t *out_len, botan_rng_t rng, **const** char *passphrase, **const** char *encryption_algo, uint32_t flags)

Deprecated, use `botan_privkey_export_encrypted_msec` or `botan_privkey_export_encrypted_iter`

typedef opaque ***botan_pubkey_t**

An opaque data type for a public key. Don't mess with it.

int **botan_pubkey_load**(botan_pubkey_t *key, **const** uint8_t bits[], size_t len)

int **botan_privkey_export_pubkey**(botan_pubkey_t *out, botan_privkey_t in)

int **botan_pubkey_export**(botan_pubkey_t key, uint8_t out[], size_t *out_len, uint32_t flags)

int **botan_pubkey_algo_name**(botan_pubkey_t key, char out[], size_t *out_len)

int **botan_pubkey_estimated_strength**(botan_pubkey_t key, size_t *estimate)

int **botan_pubkey_fingerprint**(botan_pubkey_t key, **const** char *hash, uint8_t out[], size_t *out_len)

int **botan_pubkey_destroy**(botan_pubkey_t key)

int **botan_pubkey_get_field**(botan_mp_t output, botan_pubkey_t key, **const** char *field_name)

Read an algorithm specific field from the public key object, placing it into output. For example "n" or "e" for RSA keys or "p", "q", "g", and "y" for DSA keys.

int **botan_privkey_get_field**(botan_mp_t output, botan_privkey_t key, **const** char *field_name)

Read an algorithm specific field from the private key object, placing it into output. For example "p" or "q" for RSA keys, or "x" for DSA keys or ECC keys.

31.13 RSA specific functions

int **botan_privkey_rsa_get_p** (*botan_mp_t p*, *botan_privkey_t rsa_key*)
Set p to the first RSA prime.

int **botan_privkey_rsa_get_q** (*botan_mp_t q*, *botan_privkey_t rsa_key*)
Set q to the second RSA prime.

int **botan_privkey_rsa_get_d** (*botan_mp_t d*, *botan_privkey_t rsa_key*)
Set d to the RSA private exponent.

int **botan_privkey_rsa_get_n** (*botan_mp_t n*, *botan_privkey_t rsa_key*)
Set n to the RSA modulus.

int **botan_privkey_rsa_get_e** (*botan_mp_t e*, *botan_privkey_t rsa_key*)
Set e to the RSA public exponent.

int **botan_pubkey_rsa_get_e** (*botan_mp_t e*, *botan_pubkey_t rsa_key*)
Set e to the RSA public exponent.

int **botan_pubkey_rsa_get_n** (*botan_mp_t n*, *botan_pubkey_t rsa_key*)
Set n to the RSA modulus.

int **botan_privkey_load_rsa** (*botan_privkey_t *key*, *botan_mp_t p*, *botan_mp_t q*, *botan_mp_t e*)
Initialize a private RSA key using parameters p, q, and e.

int **botan_pubkey_load_rsa** (*botan_pubkey_t *key*, *botan_mp_t n*, *botan_mp_t e*)
Initialize a public RSA key using parameters n and e.

31.14 DSA specific functions

int **botan_privkey_load_dsa** (*botan_privkey_t *key*, *botan_mp_t p*, *botan_mp_t q*, *botan_mp_t g*, *botan_mp_t x*)
Initialize a private DSA key using group parameters p, q, and g and private key x.

int **botan_pubkey_load_dsa** (*botan_pubkey_t *key*, *botan_mp_t p*, *botan_mp_t q*, *botan_mp_t g*, *botan_mp_t y*)
Initialize a private DSA key using group parameters p, q, and g and public key y.

31.15 ElGamal specific functions

int **botan_privkey_load_elgamal** (*botan_privkey_t *key*, *botan_mp_t p*, *botan_mp_t g*, *botan_mp_t x*)
Initialize a private ElGamal key using group parameters p and g and private key x.

int **botan_pubkey_load_elgamal** (*botan_pubkey_t *key*, *botan_mp_t p*, *botan_mp_t g*, *botan_mp_t y*)
Initialize a public ElGamal key using group parameters p and g and public key y.

31.16 Diffie-Hellman specific functions

int **botan_privkey_load_dh** (*botan_privkey_t *key*, *botan_mp_t p*, *botan_mp_t g*, *botan_mp_t x*)
Initialize a private Diffie-Hellman key using group parameters p and g and private key x.

int **botan_pubkey_load_dh** (*botan_pubkey_t *key*, *botan_mp_t p*, *botan_mp_t g*, *botan_mp_t y*)
Initialize a public Diffie-Hellman key using group parameters p and g and public key y.

31.17 Public Key Encryption/Decryption

typedef opaque *__botan_pk_op_encrypt_t__

An opaque data type for an encryption operation. Don't mess with it.

int **botan_pk_op_encrypt_create** (*botan_pk_op_encrypt_t *op*, *botan_pubkey_t key*, **const** char **padding*, uint32_t *flags*)

int **botan_pk_op_encrypt_destroy** (*botan_pk_op_encrypt_t op*)

int **botan_pk_op_encrypt** (*botan_pk_op_encrypt_t op*, *botan_rng_t rng*, uint8_t *out*[], size_t **out_len*, **const** uint8_t *plaintext*[], size_t *plaintext_len*)

typedef opaque *__botan_pk_op_decrypt_t__

An opaque data type for a decryption operation. Don't mess with it.

int **botan_pk_op_decrypt_create** (*botan_pk_op_decrypt_t *op*, *botan_privkey_t key*, **const** char **padding*, uint32_t *flags*)

int **botan_pk_op_decrypt_destroy** (*botan_pk_op_decrypt_t op*)

int **botan_pk_op_decrypt** (*botan_pk_op_decrypt_t op*, uint8_t *out*[], size_t **out_len*, uint8_t *ciphertext*[], size_t *ciphertext_len*)

31.18 Signatures

typedef opaque *__botan_pk_op_sign_t__

An opaque data type for a signature generation operation. Don't mess with it.

int **botan_pk_op_sign_create** (*botan_pk_op_sign_t *op*, *botan_privkey_t key*, **const** char **hash_and_padding*, uint32_t *flags*)

int **botan_pk_op_sign_destroy** (*botan_pk_op_sign_t op*)

int **botan_pk_op_sign_update** (*botan_pk_op_sign_t op*, **const** uint8_t *in*[], size_t *in_len*)

int **botan_pk_op_sign_finish** (*botan_pk_op_sign_t op*, *botan_rng_t rng*, uint8_t *sig*[], size_t **sig_len*)

typedef opaque *__botan_pk_op_verify_t__

An opaque data type for a signature verification operation. Don't mess with it.

int **botan_pk_op_verify_create** (*botan_pk_op_verify_t *op*, *botan_pubkey_t key*, **const** char **hash_and_padding*, uint32_t *flags*)

int **botan_pk_op_verify_destroy** (*botan_pk_op_verify_t op*)

int **botan_pk_op_verify_update** (*botan_pk_op_verify_t op*, **const** uint8_t *in*[], size_t *in_len*)

int **botan_pk_op_verify_finish** (*botan_pk_op_verify_t op*, **const** uint8_t *sig*[], size_t *sig_len*)

31.19 Key Agreement

typedef opaque *__botan_pk_op_ka_t__

An opaque data type for a key agreement operation. Don't mess with it.

int **botan_pk_op_key_agreement_create** (*botan_pk_op_ka_t *op*, *botan_privkey_t key*, **const** char **kdf*, uint32_t *flags*)

int **botan_pk_op_key_agreement_destroy** (*botan_pk_op_ka_t op*) int **botan_pk_op_key_agreement_export_public** (*botan_privkey_t key*, uint8_t *out*[], size_t **out_len*)

int **botan_pk_op_key_agreement** (*botan_pk_op_ka_t op*, uint8_t *out*[], size_t **out_len*, **const** uint8_t *other_key*[], size_t *other_key_len*, **const** uint8_t *salt*[], size_t *salt_len*)

int **botan_mceies_encrypt** (*botan_pubkey_t mce_key*, *botan_rng_t rng*, **const** char **aead*, **const** uint8_t *pt*[], size_t *pt_len*, **const** uint8_t *ad*[], size_t *ad_len*, uint8_t *ct*[], size_t **ct_len*)

int **botan_mceies_decrypt** (*botan_privkey_t mce_key*, **const** char **aead*, **const** uint8_t *ct*[], size_t *ct_len*, **const** uint8_t *ad*[], size_t *ad_len*, uint8_t *pt*[], size_t **pt_len*)

31.20 X.509 Certificates

typedef opaque ***botan_x509_cert_t**

An opaque data type for an X.509 certificate. Don't mess with it.

int **botan_x509_cert_load** (*botan_x509_cert_t *cert_obj*, **const** uint8_t *cert*[], size_t *cert_len*)

int **botan_x509_cert_load_file** (*botan_x509_cert_t *cert_obj*, **const** char **filename*)

int **botan_x509_cert_destroy** (*botan_x509_cert_t cert*)

int **botan_x509_cert_gen_selfsigned** (*botan_x509_cert_t *cert*, *botan_privkey_t key*, *botan_rng_t rng*, **const** char **common_name*, **const** char **org_name*)

int **botan_x509_cert_get_time_starts** (*botan_x509_cert_t cert*, char *out*[], size_t **out_len*)

int **botan_x509_cert_get_time_expires** (*botan_x509_cert_t cert*, char *out*[], size_t **out_len*)

int **botan_x509_cert_get_fingerprint** (*botan_x509_cert_t cert*, **const** char **hash*, uint8_t *out*[], size_t **out_len*)

int **botan_x509_cert_get_serial_number** (*botan_x509_cert_t cert*, uint8_t *out*[], size_t **out_len*)

int **botan_x509_cert_get_authority_key_id** (*botan_x509_cert_t cert*, uint8_t *out*[], size_t **out_len*)

int **botan_x509_cert_get_subject_key_id** (*botan_x509_cert_t cert*, uint8_t *out*[], size_t **out_len*)

int **botan_x509_cert_path_verify** (*botan_x509_cert_t cert*, **const** char **ca_dir*)

int **botan_x509_cert_get_public_key_bits** (*botan_x509_cert_t cert*, uint8_t *out*[], size_t **out_len*)

int **botan_x509_cert_get_public_key** (*botan_x509_cert_t cert*, *botan_pubkey_t *key*)

int **botan_x509_cert_get_issuer_dn** (*botan_x509_cert_t cert*, **const** char **key*, size_t *index*, uint8_t *out*[], size_t **out_len*)

int **botan_x509_cert_get_subject_dn** (*botan_x509_cert_t cert*, **const** char **key*, size_t *index*, uint8_t *out*[], size_t **out_len*)

int **botan_x509_cert_to_string** (*botan_x509_cert_t cert*, char *out*[], size_t **out_len*)

enum botan_x509_cert_key_constraints

Certificate key usage constraints. Allowed values: *NO_CONSTRAINTS*, *DIGITAL_SIGNATURE*, *NON_REPUDIATION*, *KEY_ENCIPHERMENT*, *DATA_ENCIPHERMENT*, *KEY_AGREEMENT*, *KEY_CERT_SIGN*, *CRL_SIGN*, *ENCIPHER_ONLY*, *DECIPHER_ONLY*.

int **botan_x509_cert_allowed_usage** (*botan_x509_cert_t cert*, unsigned int *key_usage*)

CHAPTER

THIRTYTWO

PYTHON BINDING

New in version 1.11.14. The Python binding is based on the *ffi* module of botan and the *ctypes* module of the Python standard library.

32.1 Versioning

botan.**version_major**()
Returns the major number of the library version (currently, 1)

botan.**version_minor**()
Returns the minor number of the library version (currently, 11)

botan.**version_patch**()
Returns the patch number of the library version (currently, 14)

botan.**version_string**()
Returns a free form version string for the library

32.2 Random Number Generators

class botan.**rng**(*rng_type = 'system'*)

Type 'user' also allowed (userspace HKDF RNG seeded from system rng). The system RNG is very cheap to create, as just a single file handle or CSP handle is kept open, from first use until shutdown, no matter how many 'system' rng instances are created. Thus it is easy to use the RNG in a one-off way, with *botan.rng().get(32)*.

get(*length*)
Return some bits

reseed(*bits = 256*)
Meaningless on system RNG, on userspace RNG causes a reseed/rekey

32.3 Hash Functions

class botan.**hash_function**(*algo*)
Algo is a string (eg 'SHA-1', 'SHA-384', 'Skein-512')

clear()
Clear state

output_length()

update(*x*)
Add some input

final()
Returns the hash of all input provided, resets for another message.

32.4 Message Authentication Codes

class botan.**message_authentication_code**(*algo*)
Algo is a string (eg 'HMAC(SHA-256)', 'Poly1305', 'CMAC(AES-256)')

clear()

output_length()

set_key(*key*)
Set the key

update(*x*)
Add some input

final()
Returns the MAC of all input provided, resets for another message with the same key.

32.5 Ciphers

class botan.**cipher**(*object*, *algo*, *encrypt = True*)

The algorithm is spcified as a string (eg 'AES-128/GCM', 'Serpent/OCB(12)', 'Threefish-512/EAX').

Set the second param to False for decryption

tag_length()
Returns the tag length (0 for unauthenticated modes)

default_nonce_length()
Returns default nonce length

update_granularity()
Returns update block size. Call to update() must provide input of exactly this many bytes

is_authenticated()
Returns True if this is an AEAD mode

valid_nonce_length(*nonce_len*)
Returns True if nonce_len is a valid nonce len for this mode

clear()
Resets all state

set_key(*key*)
Set the key

start(*nonce*)
Start processing a message using nonce

update (*txt*)

Consumes input text and returns output. Input text must be of update_granularity() length. Alternately, always call finish with the entire message, avoiding calls to update entirely

finish (*txt = None*)

Finish processing (with an optional final input). May throw if message authentication checks fail, in which case all plaintext previously processed must be discarded. You may call finish() with the entire message

32.6 Bcrypt

botan.**bcrypt** (*passwd*, *rng*, *work_factor = 10*)

Provided the password and an RNG object, returns a bcrypt string

botan.**check_bcrypt** (*passwd*, *bcrypt*)

Check a bcrypt hash against the provided password, returning True iff the password matches.

32.7 PBKDF

botan.**pbkdf** (*algo*, *password*, *out_len*, *iterations = 100000*, *salt = rng().get(12)*)

Runs a PBKDF2 algo specified as a string (eg 'PBKDF2(SHA-256)', 'PBKDF2(CMAC(Blowfish))'). Runs with n iterations with meaning depending on the algorithm. The salt can be provided or otherwise is randomly chosen. In any case it is returned from the call.

Returns out_len bytes of output (or potentially less depending on the algorithm and the size of the request).

Returns tuple of salt, iterations, and psk

botan.**pbkdf_timed** (*algo*, *password*, *out_len*, *ms_to_run = 300*, *salt = rng().get(12)*)

Runs for as many iterations as needed to consumed ms_to_run milliseconds on whatever we're running on. Returns tuple of salt, iterations, and psk

32.8 KDF

botan.**kdf** (*algo*, *secret*, *out_len*, *salt*)

32.9 Public Key

class botan.**public_key** (*object*)

fingerprint (*hash = 'SHA-256'*)

class botan.**private_key** (*algo*, *param*, *rng*)

Constructor creates a new private key. The parameter type/value depends on the algorithm. For "rsa" is is the size of the key in bits. For "ecdsa" and "ecdh" it is a group name (for instance "secp256r1"). For "ecdh" there is also a special case for group "curve25519" (which is actually a completely distinct key type with a non-standard encoding).

get_public_key ()

Return a public_key object

export()

32.10 Public Key Operations

class botan.**pk_op_encrypt**(*pubkey*, *padding*)

encrypt(*msg*, *rng*)

class botan.**pk_op_decrypt**(*privkey*, *padding*)

decrypt(*msg*)

class botan.**pk_op_sign**(*privkey*, *hash_w_padding*)

update(*msg*)

finish(*rng*)

class botan.**pk_op_verify**(*pubkey*, *hash_w_padding*)

update(*msg*)

check_signature(*signature*)

class botan.**pk_op_key_agreement**(*privkey*, *kdf*)

public_value()

Returns the public value to be passed to the other party

agree(*other*, *key_len*, *salt*)

Returns a key derived by the KDF.

CHAPTER

THIRTYTHREE

BOTAN

Subtitle Botan command line util

Manual section 1

33.1 Outline

The `botan` program is a command line tool for using a broad variety of functions of the Botan library in the shell.

All commands follow the syntax `botan <command> <command-options>`.

If `botan` is run with an unknown command, or without any command, or with the `--help` option, all available commands with their options will be printed.

33.2 Hash Function

`hash --algo=SHA-256 --buf-size=4096 files` Compute the *algo* digest over the data in any number of *files*. If no files are listed on the command line, the input source defaults to standard input.

33.3 Password Hash

`gen_bcrypt --work-factor=12 password` Calculate the bcrypt password digest of *password*. *work-factor* is an integer between 4 and 18. A higher *work-factor* value results in a more expensive hash calculation.

`check_bcrypt password hash` Checks if the bcrypt hash of the passed *password* equals the passed *hash* value.

33.4 HMAC

`hmac --hash=SHA-256 --buf-size=4096 key files` Compute the HMAC tag with the cryptographic hash function *hash* using the key in file *key* over the data in *files*. *files* defaults to STDIN.

33.5 Public Key Cryptography

`keygen --algo=RSA --params= --passphrase= --pbe= --pbe-millis=300 --der-out`
Generate a PKCS #8 *algo* private key. If *der-out* is passed, the pair is BER encoded. Otherwise, PEM encoding is used. To protect the PKCS #8 formatted key, it is recommended to encrypt it with a provided *passphrase*. *pbe* is the name of the desired encryption algorithm, which uses *pbe-millis* milliseconds to derive the encryption key from the passed *passphrase*. Algorithm specific parameters, as the desired bitlength of an RSA key, can be passed with *params*.

- For RSA *params* specifies the bit length of the RSA modulus. It defaults to 3072.
- For DH *params* specifies the DH parameters. It defaults to modp/ietf/2048.
- For DSA *params* specifies the DSA parameters. It defaults to dsa/botan/2048.
- For EC algorithms *params* specifies the elliptic curve. It defaults to secp256r1.

`pkcs8 --pass-in= --pub-out --der-out --pass-out= --pbe= --pbe-millis=300 key`
Open a PKCS #8 formatted key at *key*. If *key* is encrypted, the passphrase must be passed as *pass-in*. It is possible to (re)encrypt the read key with the passphrase passed as *pass-out*. The parameters *pbe-millis* and *pbe* work similarly to `keygen`.

`sign --passphrase= --hash=SHA-256 --emsa= key file` Sign the data in *file* using the PKCS #8 private key *key*. If *key* is encrypted, the used passphrase must be passed as *pass-in*. *emsa* specifies the signature scheme and *hash* the cryptographic hash function used in the scheme.

- For RSA signatures EMSA4 (RSA-PSS) is the default scheme.
- For ECDSA and DSA *emsa* defaults to EMSA1 (signing the hash directly)

`verify --hash=SHA-256 --emsa= pubkey file signature` Verify the authenticity of the data in *file* with the provided signature *signature* and the public key *pubkey*. Similarly to the signing process, *emsa* specifies the signature scheme and *hash* the cryptographic hash function used in the scheme.

`gen_dl_group --pbits=1024 --qbits=0 --type=subgroup` Generate ANSI X9.42 encoded Diffie-Hellman group parameters.

- If *type=subgroup* is passed, the size of the prime subgroup q is sampled as a prime of *qbits* length and p is *pbits* long. If *qbits* is not passed, its length is estimated from *pbits* as described in RFC 3766.
- If *type=strong* is passed, p is sampled as a safe prime with length *pbits* and the prime subgroup has size q with *pbits*-1 length.

`dl_group_info --pem name` Print raw Diffie-Hellman parameters (p,g) of the standarized DH group *name*. If *pem* is set, the X9.42 encoded group is printed.

`ec_group_info --pem name` Print raw elliptic curve domain parameters of the standarized curve *name*. If *pem* is set, the encoded domain is printed.

33.6 X.509

`gen_pkcs10 key CN --country= --organization= --email= --key-pass= --hash=SHA-256 --emsa=`
Generate a PKCS #10 certificate signing request (CSR) using the passed PKCS #8 private key *key*. If the private key is encrypted, the decryption passphrase *key-pass* has to be passed.*emsa* specifies the padding scheme to be used when calculating the signature.

- For RSA keys EMSA4 (RSA-PSS) is the default scheme.
- For ECDSA, DSA, ECGDSA, ECKCDSA and GOST-34.10 keys *emsa* defaults to EMSA1.

`gen_self_signed key CN --country= --dns= --organization= --email= --key-pass= --ca --hash=S`
Generate a self signed X.509 certificate using the PKCS #8 private key *key*. If the private key is encrypted, the decryption passphrase *key-pass* has to be passed. If *ca* is passed, the certificate is marked for certificate authority (CA) usage. *emsa* specifies the padding scheme to be used when calculating the signature.

- For RSA keys EMSA4 (RSA-PSS) is the default scheme.
- For ECDSA, DSA, ECGDSA, ECKCDSA and GOST-34.10 keys *emsa* defaults to EMSA1.

`sign_cert --ca-key-pass= --hash=SHA-256 --duration=365 --emsa= ca_cert ca_key pkcs10_req`
Create a CA signed X.509 certificate from the information contained in the PKCS #10 CSR *pkcs10_req*. The CA certificate is passed as *ca_cert* and the respective PKCS #8 private key as *ca_key*. If the private key is encrypted, the decryption passphrase *ca-key-pass* has to be passed. The created certificate has a validity period of *duration* days. *emsa* specifies the padding scheme to be used when calculating the signature. *emsa* defaults to the padding scheme used in the CA certificate.

`ocsp_check subject issuer` Verify an X.509 certificate against the issuers OCSP responder. Pass the certificate to validate as *subject* and the CA certificate as *issuer*.

`cert_info --fingerprint --ber file` Parse X.509 PEM certificate and display data fields. If `--fingerprint` is used, the certificate's fingerprint is also printed.

`cert_verify subject *ca_certs` Verify if the provided X.509 certificate *subject* can be sucessfully validated. The list of trusted CA certificates is passed with *ca_certs*, which is a list of one or more certificates.

33.7 TLS Server/Client

`tls_ciphers --policy=default --version=tls1.2` Prints the list of ciphersuites that will be offered under a particular policy/version. The policy can be any of the the strings "default", "suiteb", "strict", or "all" to denote built-in policies, or it can name a file from which a policy description will be read.

`tls_client host --port=443 --print-certs --policy= --tls1.0 --tls1.1 --tls1.2 --session-db=`
Implements a testing TLS client, which connects to *host* via TCP or UDP on port *port*. The TLS version can be set with the flags *tls1.0*, *tls1.1* and *tls1.2* of which the lowest specified version is automatically chosen. If none of the TLS version flags is set, the latest supported version is chosen. The client honors the TLS policy defined in the *policy* file and prints all certificates in the chain, if *print-certs* is passed. *next-protocols* is a comma seperated list and specifies the protocols to advertise with Application-Layer Protocol Negotiation (ALPN).

`tls_server cert key --port=443 --type=tcp --policy=` Implements a testing TLS server, which allows TLS clients to connect. Binds to either TCP or UDP on port *port*. The server uses the certificate *cert* and the respective PKCS #8 private key *key*. The server honors the TLS policy defined in the *policy* file.

`tls_http_server cert key --port=443 --policy= --session-db --session-db-pass=`
Only available if asio support was enabled. Provides a simple HTTP server which replies to all requests with an informational text output. The server honors the TLS policy defined in the *policy* file.

`tls_proxy listen_port target_host target_port server_cert server_key` Only available if asio support was enabled. Listens on a port and forwards all connects to a target server specified at `target_host` and `target_port`.

33.8 Number Theory

`is_prime --prob=56 n` Test if the integer *n* is composite or prime with a Miller-Rabin primality test with *(prob+2)/2* iterations.

`factor n` Factor the integer *n* using a combination of trial division by small primes, and Pollard's Rho algorithm.

`gen_prime --count=1 bits` Samples *count* primes with a length of *bits* bits.

33.9 PSK Database

Only available if sqlite3 support was compiled in.

`psk_set db db_key name psk` Using the PSK database named db and encrypting under the (hex) key `db_key`, save the provided psk (also hex) under `name`:

```
$ botan psk_set psk.db deadba55 bunny f00fee
```

`psk_get db db_key name` Get back a value saved with `psk_set`:

```
$ botan psk_get psk.db deadba55 bunny
f00fee
```

`psk_list db db_key` List all values saved to the database under the given key:

```
$ botan psk_list psk.db deadba55
bunny
```

33.10 Data Encoding/Decoding

`base64_dec file` Encode *file* to Base64.

`base64_enc file` Decode Base64 encoded *file*.

`hex_dec file` Encode *file* to Hex.

`hex_enc file` Decode Hex encoded *file*.

33.11 Miscellaneous Commands

`version --full` Print the version number. If option `-full` is provided additional details are printed.

`config info_type` Prints build information, useful for applications which want to build against the library. The `info_type` argument can be any of `prefix`, `cflags`, `ldflags`, or `libs`.

`cpuid` List available processor flags (aes_ni, SIMD extensions, . . .).

`asn1print file` Decode and print *file* with ASN.1 Basic Encoding Rules (BER).

`http_get url` Retrieve ressource from the passed http/https *url*.

`speed --msec=100 --provider= --buf-size=4096 algos` Measures the speed of the passed *algos*. If no *algos* are passed all available speed tests are executed. *msec* (in milliseconds) sets the period of measurement for each algorithm.

`rng --system --rdrand bytes` Sample *bytes* random bytes from the specified random number generator. If *system* is set, the Botan System_RNG is used. If *system* is unset and *rdrand* is set, the hardware rng RDRAND_RNG is used. If both are unset, the Botan AutoSeeded_RNG is used.

`cc_encrypt CC passphrase --tweak=` Encrypt the passed valid credit card number *CC* using FPE encryption and the passphrase *passphrase*. The key is derived from the passphrase using PBKDF2 with SHA256. Due to the nature of FPE, the ciphertext is also a credit card number with a valid checksum. *tweak* is public and parameterizes the encryption function.

`cc_decrypt CC passphrase --tweak=` Decrypt the passed valid ciphertext *CC* using FPE decryption with the passphrase *passphrase* and the tweak *tweak*.

CHAPTER

THIRTYFOUR

SIDE CHANNELS

Many cryptographic systems can be broken by side channels. This document notes side channel protections which are currently implemented, as well as areas of the code which are known to be vulnerable to side channels. The latter are obviously all open for future improvement.

The following text assumes the reader is already familiar with cryptographic implementations, side channel attacks, and common countermeasures.

34.1 RSA

Blinding is always used to protect private key operations (there is no way to turn it off). As an optimization, instead of choosing a new random mask and inverse with each decryption, both the mask and its inverse are simply squared to choose the next blinding factor. This is much faster than computing a fresh value each time, and the additional relation is thought to provide only minimal useful information for an attacker. Every BOTAN_BLINDING_REINIT_INTERVAL (default 32) operations, a new starting point is chosen.

RSA signing uses the CRT optimization, which is much faster but vulnerable to trivial fault attacks [RsaFault] which can result in the key being entirely compromised. To protect against this (or any other computational error which would have the same effect as a fault attack in this case), after every private key operation the result is checked for consistency with the public key. This introduces only slight additional overhead and blocks most fault attacks; it is possible to use a second fault attack to bypass this verification, but such a double fault attack requires significantly more control on the part of an attacker than a BellCore style attack, which is possible if any error at all occurs during either modular exponentiation involved in the RSA signature operation.

See blinding.cpp and rsa.cpp.

If the OpenSSL provider is enabled, then no explicit blinding is done; we assume OpenSSL handles this. See openssl_rsa.cpp.

34.2 Decryption of PKCS #1 v1.5 Ciphertexts

This padding scheme is used with RSA, and is very vulnerable to errors. In a scenario where an attacker can repeatedly present RSA ciphertexts, and a legitimate key holder will attempt to decrypt each ciphertext and simply indicates to the attacker if the PKCS padding was valid or not (without revealing any additional information), the attacker can use this behavior as an oracle to perform iterative decryption of arbitrary RSA ciphertexts encrypted under that key. This is the famous million message attack [MillionMsg]. A side channel such as a difference in time taken to handle valid and invalid RSA ciphertexts is enough to mount the attack [MillionMsgTiming].

Preventing this issue in full requires some application level changes. In protocols which know the expected length of the encrypted key, PK_Decryptor provides the function *decrypt_or_random* which first generates a random fake key, then decrypts the presented ciphertext, then in constant time either copies out the random key or the decrypted plaintext depending on if the ciphertext was valid or not (valid padding and expected plaintext length). Then in the case of an attack, the protocol will carry on with a randomly chosen key, which will presumably cause total failure in a way that does not allow an attacker to distinguish (via any timing or other side channel, nor any error messages specific to the one situation vs the other) if the RSA padding was valid or invalid.

One very important user of PKCS #1 v1.5 encryption is the TLS protocol. In TLS, some extra versioning information is embedded in the plaintext message, along with the key. It turns out that this version information must be treated in an identical (constant-time) way with the PKCS padding, or again the system is broken. [VersionOracle]. This is supported by a special version of PK_Decryptor::decrypt_or_random that additionally allows verifying one or more content bytes, in addition to the PKCS padding.

See eme_pkcs.cpp and pubkey.cpp.

34.3 Verification of PKCS #1 v1.5 Signatures

One way of verifying PKCS #1 v1.5 signature padding is to decode it with an ASN.1 BER parser. However such a design commonly leads to accepting signatures besides the (single) valid RSA PKCS #1 v1.5 signature for any given message, because often the BER parser accepts variations of the encoding which are actually invalid. It also needlessly exposes the BER parser to untrusted inputs.

It is safer and simpler to instead re-encode the hash value we are expecting using the PKCS #1 v1.5 encoding rules, and const time compare our expected encoding with the output of the RSA operation. So that is what Botan does.

See emsa_pkcs.cpp.

34.4 OAEP

RSA OAEP is (PKCS#1 v2) is the recommended version of RSA encoding standard, because it is not directly vulnerable to Bleichenbacher attack. However, if implemented incorrectly, a side channel can be presented to an attacker and create an oracle for decrypting RSA ciphertexts [OaepTiming].

This attack is avoided in Botan by making the OAEP decoding operation run without any conditional jumps or indexes, with the only variance in runtime coming from the length of the RSA key (which is public information).

See eme_oaep.cpp.

34.5 Modular Exponentiation

Modular exponentiation uses a fixed window algorithm with Montgomery representation. A side channel silent table lookup is used to access the precomputed powers. See powm_mnt.cpp.

The Karatsuba multiplication algorithm has some conditional branches that probably expose information through the branch predictor, but probably? does not expose a timing channel since the same amount of work is done on both sides of the conditional. There is certainly room for improvement here. See mp_karat.cpp for details.

The Montgomery reduction is written (and tested) to run in constant time. See mp_monty.cpp.

34.6 ECC point decoding

The API function OS2ECP, which is used to convert byte strings to ECC points, verifies that all points satisfy the ECC curve equation. Points that do not satisfy the equation are invalid, and can sometimes be used to break protocols

([InvalidCurve] [InvalidCurveTLS]). See point_gfp.cpp.

34.7 ECC scalar multiply

There are two implementations of scalar multiply, PointGFp::operator* and the class Blinded_Point_Multiply. The default scalar multiply uses the Montgomery ladder. However it currently leaks the size of the scalar, because the loop iterations are bounded by the scalar size.

Blinded_Point_Multiply (used by ECDH, ECDSA, etc) applies several additional side channel countermeasures. The scalar is masked by a small multiple of the group order (this is commonly called Coron's first countermeasure [CoronDpa]), the size of the scalar mask is currently controlled by build.h value BOTAN_POINTGFP_SCALAR_BLINDING_BITS which defaults to 20 bits.

Botan stores all ECC points in Jacobian representation. This form allows faster computation by representing points (x,y) as (X,Y,Z) where x=X/Z^2 and y=Y/Z^3. As the representation is redundant, for any randomly chosen r, (X*r^2,Y*r^3,Z*r) is an equivalent point. Changing the point values prevents an attacker from mounting attacks based on the input point remaining unchanged over multiple executions. This is commonly called Coron's third countermeasure, see again [CoronDpa].

Currently Blinded_Point_Multiply uses one of two different algorithms, depending on a build-time flag. If BOTAN_POINTGFP_BLINDED_MULTIPLY_USE_MONTGOMERY_LADDER is set in build.h (default is for it *not* to be set), then a randomized Montgomery ladder algorithm from [RandomMonty] is used. Otherwise, a simple fixed window exponentiation is used; the current version leaks exponent bits through memory index values. We rely on scalar blinding to reduce this leakage. It would obviously be better for Blinded_Point_Multiply to converge on a single side channel silent algorithm.

See point_gfp.cpp.

34.8 ECDH

ECDH verifies (through its use of OS2ECP) that all input points received from the other party satisfy the curve equation. This prevents twist attacks. The same check is performed on the output point, which helps prevent fault attacks.

34.9 ECDSA

Inversion of the ECDSA nonce k must be done in constant time, as any leak of even a single bit of the nonce can be sufficient to allow recovering the private key. In Botan all inverses modulo an odd number are performed using a constant time algorithm due to Niels Möller.

34.10 x25519

The x25519 code is independent of the main Weiserstrass form ECC code, instead based on curve25519-donna-c64.c by Adam Langley. The code seems immune to cache based side channels. It does make use of integer multiplications; on some old CPUs these multiplications take variable time and might allow a side channel attack. This is not considered a problem on modern processors.

34.11 TLS CBC ciphersuites

The original TLS v1.0 CBC Mac-then-Encrypt mode is vulnerable to an oracle attack. If an attacker can distinguish padding errors through different error messages [TlsCbcOracle] or via a side channel attack like [Lucky13], they can abuse the server as a decryption oracle.

The side channel protection for Lucky13 follows the approach proposed in the Lucky13 paper. It is not perfectly constant time, but does hide the padding oracle in practice. Tools to test TLS CBC decoding are included in the timing tests. See https://github.com/randombit/botan/pull/675 for more information.

The Encrypt-then-MAC extension, which completely avoids the side channel, is implemented and used by default for CBC ciphersuites.

34.12 CBC mode padding

In theory, any good protocol protects CBC ciphertexts with a MAC. But in practice, some protocols are not good and cannot be fixed immediately. To avoid making a bad problem worse, the code to handle decoding CBC ciphertext padding bytes runs in constant time, depending only on the block size of the cipher.

34.13 AES

On x86 processors which support it, AES-NI instruction set is used, as it is fast and (presumed) side channel silent. There is no support at the moment for the similar ARMv8 or POWER AES instructions; patches would be welcome.

On x86 processors without AES-NI but with SSSE3 (which includes older Intel Atoms and Core2 Duos, and even now some embedded or low power x86 chips), a version of AES using pshufb is used which is both fast and side channel silent. It is based on code by Mike Hamburg [VectorAes], see aes_ssse3.cpp. This same technique could be applied with NEON or AltiVec, and the paper suggests some optimizations for the AltiVec shuffle.

On all other processors, a class 4K table lookup version based on the original Rijndael code is used. This approach relatively fast, but now known to be very vulnerable to side channels. The implementation does make modifications in the first and last rounds to reduce the cache signature, but these merely increase the number of observations required. See [AesCacheColl] for one paper which analyzes a number of implementations including Botan. Botan already follows both of their suggested countermeasures, which increased the number of samples required from 2**13 to the only slightly less pitiful 2**19 samples.

The Botan block cipher API already supports bitslicing implementations, so a const time 8x bitsliced AES could be integrated fairly easily.

34.14 GCM

On platforms that support a carryless multiply instruction (recent x86 and ARM), GCM is fast and constant time.

On all other platforms, GCM is slow and constant time. It uses a simple bit at at time loop. It would be much faster using a table lookup, but we wish to avoid side channels. One improvement here would be the option of using a 2K or 4K table, but using a side-channel silent (masked) table lookup.

34.15 OCB

It is straightforward to implement OCB mode in a efficient way that does not depend on any secret branches or lookups. See ocb.cpp for the implementation.

34.16 Poly1305

The Poly1305 implementation does not have any secret lookups or conditionals. The code is based on the public domain version by Andrew Moon.

34.17 DES/3DES

The DES implementation uses table lookups, and is likely vulnerable to side channel attacks. DES or 3DES should be avoided in new systems. The proper fix would be a scalar bitsliced implementation, this is not seen as worth the engineering investment given these algorithms end of life status.

34.18 Twofish

This algorithm uses table lookups with secret sboxes. No cache-based side channel attack on Twofish has ever been published, but it is possible nobody sufficiently skilled has ever tried.

34.19 ChaCha20, Serpent, Threefish, . . .

Some algorithms including ChaCha, Salsa, Serpent and Threefish are ‘naturally’ silent to cache and timing side channels on all recent processors.

34.20 IDEA

IDEA encryption, decryption, and key schedule are implemented to take constant time regardless of their inputs.

34.21 Hash Functions

Most hash functions included in Botan such as MD5, SHA-1, SHA-2, SHA-3, Skein, and BLAKE2 do not require any input-dependent memory lookups, and so seem to not be affected by common CPU side channels.

34.22 Memory comparisons

The function same_mem in header mem_ops.h provides a constant-time comparison function. It is used when comparing MACs or other secret values. It is also exposed for application use.

34.23 Memory zeroizing

There is no way in portable C/C++ to zero out an array before freeing it, in such a way that it is guaranteed that the compiler will not elide the 'additional' (seemingly unnecessary) writes to zero out the memory.

The function secure_scrub_memory (in mem_ops.cpp) uses some system specific trick to zero out an array. On Windows it uses the directly supported API function RtlSecureZeroMemory.

On other platforms, by default the trick of referencing memset through a volatile function pointer is used. This approach is not guaranteed to work on all platforms, and currently there is no systematic check of the resulting binary function that it is compiled as expected. But, it is the best approach currently known and has been verified to work as expected on common platforms.

If BOTAN_USE_VOLATILE_MEMSET_FOR_ZERO is set to 0 in build.h (not the default) a byte at a time loop through a volatile pointer is used to overwrite the array.

34.24 Memory allocation

Botan's secure_vector type is a std::vector with a custom allocator. The allocator calls secure_scrub_memory before freeing memory.

Some operating systems support an API call to lock a range of pages into memory, such that they will never be swapped out (mlock on POSIX, VirtualLock on Windows). On many POSIX systems mlock is only usable by root, but on Linux, FreeBSD and possibly other systems a small amount of memory can be mlock'ed by processes without extra credentials.

If available, Botan uses such a region for storing key material. It is created in anonymous mapped memory (not disk backed), locked in memory, and scrubbed on free. This memory pool is used by secure_vector when available. It can be disabled at runtime setting the environment variable BOTAN_MLOCK_POOL_SIZE to 0.

34.25 Automated Analysis

Currently the main tool used by the Botan developers for testing for side channels at runtime is valgrind; valgrind's runtime API is used to taint memory values, and any jumps or indexes using data derived from these values will cause a valgrind warning. This technique was first used by Adam Langley in ctgrind. See header ct_utils.h.

To check, install valgrind, configure the build with –with-valgrind, and run the tests.

34.26 References

[AesCacheColl] Bonneau, Mironov "Cache-Collision Timing Attacks Against AES" (http://www.jbonneau.com/doc/BM06-CHES-aes_cache_timing.pdf)

[CoronDpa] Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems" (https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.5695)

[InvalidCurve] Biehl, Meyer, Müller: Differential fault attacks on elliptic curve cryptosystems (https://www.iacr.org/archive/crypto2000/18800131/18800131.pdf)

[InvalidCurveTLS] Jager, Schwenk, Somorovsky: Practical Invalid Curve Attacks on TLS-ECDH (https://www.nds.rub.de/research/publications/ESORICS15/)

[SafeCurves] Bernstein, Lange: SafeCurves: choosing safe curves for elliptic-curve cryptography. (https://safecurves.cr.yp.to)

[Lucky13] AlFardan, Paterson “Lucky Thirteen: Breaking the TLS and DTLS Record Protocols” (http://www.isg.rhul.ac.uk/tls/TLStiming.pdf)

[MillionMsg] Bleichenbacher “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS1” (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.8543)

[MillionMsgTiming] Meyer, Somorovsky, Weiss, Schwenk, Schinzel, Tews: Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks (https://www.nds.rub.de/research/publications/mswsst2014-bleichenbacher-usenix14/)

[OaepTiming] Manger, “A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0” (http://archiv.infsec.ethz.ch/education/fs08/secsem/Manger01.pdf)

[RsaFault] Boneh, Demillo, Lipton “On the importance of checking cryptographic protocols for faults” (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.9764)

[RandomMonty] Le, Tan, Tunstall “Randomizing the Montgomery Powering Ladder” (https://eprint.iacr.org/2015/657)

[VectorAes] Hamburg, “Accelerating AES with Vector Permute Instructions” https://shiftleft.org/papers/vector_aes/vector_aes.pdf

[VersionOracle] Klíma, Pokorný, Rosa “Attacking RSA-based Sessions in SSL/TLS” (https://eprint.iacr.org/2003/052)

CHAPTER

THIRTYFIVE

NOTES FOR DISTRIBUTORS

This document has information for anyone who is packaging copies of Botan for use by downstream developers, such as through a Linux distribution or other package management system.

35.1 Recommended Options

In most environments, zlib, bzip2, and sqlite are already installed, so there is no reason to not include support for them in Botan as well. Build with options `--with-zlib --with-bzip2 --with-sqlite3` to enable these features.

35.2 Set Distribution Info

If your distribution of Botan involves creating library binaries, use the configure.py flag `--distribution-info=` to set the version of your packaging. For example Foonix OS might distribute its 4th revision of the package for Botan 2.1.3 using `--distribution-info='Foonix 2.1.3-4'`. The string is completely free-form, since it depends on how the distribution numbers releases and packages.

Any value set with `--distribution-info` flag will be included in the version string, and can read through the `BOTAN_DISTRIBUTION_INFO` macro.

35.3 Minimize Distribution Patches

We (Botan upstream) *strongly* prefer that downstream distributions maintain no long-term patches against Botan. Even if it is a build problem which probably only affects your environment, please open an issue on github and include the patch you are using. Perhaps the issue does affect other users, and even if not it would be better for everyone if the library were improved so it were not necessary for the patch to be created in the first place. For example, having to modify or remove a build data file, or edit the makefile after generation, suggests an area where the build system is insufficiently flexible.

Obviously nothing in the BSD-2 license prevents you from distributing patches or modified versions of Botan however you please. But long term patches by downstream distributors have a tendency to bitrot and sometimes even result in security problems (such as in the Debian OpenSSL RNG fiasco) because the patches are never reviewed by the library developers. So we try to discourage them, and work to ensure they are never necessary.

CHAPTER

THIRTYSIX

FUZZING THE LIBRARY

Botan comes with a set of fuzzing endpoints which can be used to test the library.

36.1 Fuzzing with libFuzzer

To fuzz with libFuzzer (https://llvm.org/docs/LibFuzzer.html), you'll first need to compile libFuzzer:

```
$ svn co https://llvm.org/svn/llvm-project/compiler-rt/trunk/lib/fuzzer libFuzzer
$ cd libFuzzer && clang -c -g -O2 -std=c++11 *.cpp
$ ar cr libFuzzer.a libFuzzer/*.o
```

Then build the fuzzers:

```
$ ./configure.py --cc=clang --build-fuzzer=libfuzzer --unsafe-fuzzer-mode \
      --cc-abi-flags='-fsanitize=address,undefined -fsanitize-coverage=edge,indirect-
↪calls,8bit-counters -fno-sanitize-recover=undefined'
$ make fuzzers
```

The fuzzer binaries will be in *build/fuzzer*. Simply pick one and run it, optionally also passing a directory containing corpus inputs.

Using *libfuzzer* build mode implicitly assumes the fuzzers need to link with *libFuzzer*; if another library is needed (for example in OSS-Fuzz, which uses *libFuzzingEngine*), use the flag *–with-fuzzer-lib* to specify the desired name.

36.2 Fuzzing with AFL

To fuzz with AFL (http://lcamtuf.coredump.cx/afl/):

```
$ ./configure.py --with-sanitizers --build-fuzzer=afl --unsafe-fuzzer-mode --cc-
↪bin=afl-g++
$ make fuzzers
```

For AFL, *–with-sanitizers* is optional.

You can also use *afl-clang-fast++* or *afl-clang++*.

The fuzzer binaries will be in *build/fuzzer*. To run them you need to run under *afl-fuzz*:

```
$ afl-fuzz -i corpus_path -o output_path ./build/fuzzer/binary
```

36.3 Fuzzing with TLS-Attacker

TLS-Attacker (https://github.com/RUB-NDS/TLS-Attacker) includes a mode for fuzzing TLS servers. A prebuilt copy of TLS-Attacker is available in a git repo:

```
$ git clone --depth 1 https://github.com/randombit/botan-ci-tools.git
```

To run it against Botan's server:

```
$ ./configure.py --with-sanitizers
$ make botan
$ ./src/scripts/run_tls_fuzzer.py ./botan ./botan-ci-tools
```

Output and logs from the fuzzer are placed into */tmp*. See the TLS-Attacker documentation for more information about how to use this tool.

36.4 Input Corpus

AFL requires an input corpus, and libFuzzer can certainly make good use of it.

Some crypto corpus repositories include

- https://github.com/randombit/crypto-corpus
- https://github.com/mozilla/nss-fuzzing-corpus
- https://github.com/google/boringssl/tree/master/fuzz
- https://github.com/openssl/openssl/tree/master/fuzz/corpora

36.5 Adding new fuzzers

New fuzzers are created by adding a source file to *src/fuzzers* which have the signature:

`void fuzz(const uint8_t in[], size_t len)`

After adding your fuzzer, rerun *./configure.py* and build.

CHAPTER

THIRTYSEVEN

DEPRECATED FEATURES

The following functionality is currently deprecated, and will likely be removed in a future release. If you think you have a good reason to be using one of the following, contact the developers to explain your use case if you want to make sure your code continues to work.

This is in addition to specific API calls marked with BOTAN_DEPRECATED in the source.

- Directly accessing the member variables of types calendar_point, ASN1_Attribute, and AlgorithmIdentifier
- The headers `botan.h`, `init.h`, `lookup.h`
- All or nothing package transform (`package.h`)
- The Perl-XS wrapper
- The TLS constructors taking *std::function* for callbacks. Instead use the TLS::Callbacks interface.
- Using `X509_Certificate::subject_info` and `issuer_info` to access any information that is not included in the DN or subject alternative name. Prefer using the specific accessor functions for other data, eg instead of `cert.subject_info("X509.Certificate.serial")` use `cert.serial_number()`.
- The Buffered_Computation base class. In a future release the class will be removed, and all of member functions instead declared directly on MessageAuthenticationCode and HashFunction. So this only affects you if you are directly referencing *Botan::Buffered_Computation* in some way.
- The SymmetricAlgorithm base class. Similarly to Buffered_Computation, in a future release the class will be removed and its member functions copied to classes which currently subclass it. This only affects your code if you are referencing *Botan::SymmetricAlgorithm* directly.
- Platform support for Google Native Client
- Support for PathScale and HP compilers
- TLS: 3DES and SEED ciphersuites
- TLS: Anonymous DH/ECDH ciphersuites
- TLS: DSA ciphersuites/certs
- TLS: static RSA key exchange ciphersuites
- TLS: CCM_8 ciphersuites
- Block ciphers CAST-256, Kasumi, MISTY1, and DESX.
- CBC-MAC, X9.19-MAC
- PBKDF1 key derivation
- GCM support for 64-bit tags
- Old (Google specific) ChaCha20 TLS ciphersuites

- All built in ECC groups < 256 bits
- All built in MODP groups < 2048 bits
- All pre-created DSA groups

CHAPTER THIRTYEIGHT

ABI STABILITY

Botan uses semantic versioning for the API; if API features are added the minor version increases, whereas if API compatability breaks occur the major version is increased.

However no guarantees about ABI are made between releases. Maintaining an ABI compatible release in a complex C++ API is exceedingly expensive in development time; just adding a single member variable or virtual function is enough to cause ABI issues.

If ABI changes, the soname revision will increase to prevent applications from linking against a potentially incompatible version at runtime.

If you are concerned about long-term ABI issues, considering using the C API instead; this subset *is* ABI stable.

You can review a report on ABI changes to Botan at https://abi-laboratory.pro/tracker/timeline/botan/