Botan Reference Guide

Release 2.18.1

Jack Lloyd Daniel Neus René Korthaus
Juraj Somorovsky Tobias Niemann

2021-05-09

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 Examples

Some examples of usage are included in this documentation. However a better source for example code is in the implementation of the command line interface (https://github.com/randombit/botan/tree/master/src/cli), which was intentionally written to act as practical examples of usage.

1.2 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 at a place such as the metzdowd (http://www.metzdowd.com/mailman/listinfo/cryptography) or randombit (https://lists.randombit.net/mailman/listinfo/cryptography) mailing lists or the cryptography stack exchange (https://crypto.stackexchange.com/). 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 platforms like Windows, Linux, Android and iOS, and also promising new systems such as IncludeOS and Fuchsia.

- 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 in the past but never saw widespread deployment and have no compelling benefit over other designs have 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, allowing further simplifications in the codebase. 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, particularly when it comes to Level 1 software validations. 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 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++ 2015 and 2017

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
- macOS x86-64, XCode Clang
- iOS aarch64, XCode Clang
- Android aarch64, NDK Clang
- FreeBSD x86-64, Clang 3.8+

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

Of course most other modern OSes such as QNX, AIX, OpenBSD, NetBSD, and Solaris also work just fine. Some are tested occasionally, usually just before a new release. But very little code specific to these platforms is written by the primary developers. For example, any functionality in the library which utilizes OpenBSD specific APIs was likely contributed by someone interested in that platform.

In theory any working C++11 compiler is fine but in practice, we only regularly test with GCC, Clang, and Visual C++. Recent versions of IBM XLC can compile the library but occasionally codegen bugs occur. Several other compilers (such as Intel and PGI) are supported by the build system but are not tested by the developers and may have build or codegen problems. Patches to improve support for these compilers is welcome.

3.2 Branch Support Status

Following table provides the support status for Botan branches as of September 2020. 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.x	2017-01-06	2020-10-05	2024-01-01 or later
3.x	2021?	?	?

"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)

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): certstor_sqlite3 sessions_sqlite3
INFO: Skipping (incompatible CPU): aes_power8
INFO: Skipping (incompatible OS): darwin_secrandom getentropy win32_stats
INFO: Skipping (incompatible compiler): aes_armv8 pmull sha1_armv8 sha2_32_armv8
```

(continues on next page)

(continued from previous page)

```
INFO: Skipping (no enabled compression schemes): compression
INFO: Skipping (requires external dependency): boost bzip2 lzma openssl sqlite3 tpm␣
↪zlib
```

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=xtea,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* and *Minimized Builds*.

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. Note that a minimized build does not by default 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.

4.2 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-
↪command=i686-w64-mingw32-ar
...
$ make
...
$ file botan.exe
botan.exe: PE32 executable (console) Intel 80386, for MS Windows
```

Note: For whatever reason, some distributions of MinGW lack support for threading or mutexes in the C++ standard library. You can work around this by disabling thread support using `--without-os-feature=threads`

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.

4.3 On Unix

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

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

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.

4.4 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"
```

4.5 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
$ nmake check
$ 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).

4.6 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.

4.7 For Android

Modern versions of Android NDK use Clang and support C++11. Simply configure using the appropriate NDK compiler:

```
$ export CXX=/opt/android-ndk/toolchains/llvm/prebuilt/linux-x86_64/bin/aarch64-linux-
↪android28-clang++
$ ./configure.py --os=android --cc=clang --cpu=arm64
```

4.7.1 Docker

To build android version, there is the possibility to use the docker way:

```
sudo ANDROID_SDK_VER=21 ANDROID_ARCH=arm64 src/scripts/docker-android.sh
```

This will produce the docker-builds/android folder containing each architecture compiled.

4.8 Emscripten (WebAssembly)

To build for WebAssembly using Emscripten, try:

```
CXX=em++ ./configure.py --cc=clang --cpu=llvm --os=emscripten
make
```

This will produce bitcode files `botan-test.bc` and `botan.bc` along with a static archive `libbotan-2.a` which can linked with other modules. To convert the tests into a WASM file which can be executed on a browser, use:

```
em++ -s ALLOW_MEMORY_GROWTH=1 -s DISABLE_EXCEPTION_CATCHING=0 -s WASM=1 \
   --preload-file src/tests/data botan-test.bc -o botan-test.html
```

4.9 Supporting Older Distros

Some “stable” distributions, notably RHEL/CentOS, ship very obsolete versions of binutils, which do not support more recent CPU instructions. As a result when building you may receive errors like:

```
Error: no such instruction: `sha256rnds2 %xmm0,%xmm4,%xmm3'
```

Depending on how old your binutils is, you may need to disable BMI2, AVX2, SHA-NI, and/or RDSEED. These can be disabled by passing the flags `--disable-bmi2`, `--disable-avx2`, `--disable-sha-ni`, and `--disable-rdseed` to `configure.py`.

4.10 Other Build-Related Tasks

4.10.1 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`.

4.10.2 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`.

Note: The library will as usual be configured to target some specific operating system and CPU architecture. You can use the CPU target "generic" if you need to target multiple CPU architectures, but this has the effect of disabling *all* CPU specific features such as SIMD, AES instruction sets, or inline assembly. If you need to ship amalgamations for multiple targets, it would be better to create different amalgamation files for each individual target.

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 (assuming you are not making use of a compiler cache such as `ccache` or `sccache`) amalgamation builds usually have significantly shorter compile times for full rebuilds.

4.10.3 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 using sqlite3 databases in various contexts (TLS session cache, PSK database, etc).
- `--with-openssl` adds an engine that uses OpenSSL for some ciphers, hashes, and public key operations. OpenSSL 1.0.2 or later is supported. LibreSSL can also be used.
- `--with-tpm` adds support for using TPM hardware via the TrouSerS library.

- `--with-boost` enables using some Boost libraries. In particular Boost.Filesystem is used for a few operations (but on most platforms, a native API equivalent is available), and Boost.Asio is used to provide a few extra TLS related command line utilities.

4.10.4 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.

4.10.5 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.

4.10.6 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.

4.10.7 Enabling or Disabling Use of Certain OS Features

Botan uses compile-time flags to enable or disable use of certain operating specific functions. You can also override these at build time if desired.

The default feature flags are given in the files in `src/build-data/os` in the `target_features` block. For example Linux defines flags like `proc_fs`, `getauxval`, and `sockets`. The `configure.py` option `--list-os-features` will display all the feature flags for all operating system targets.

To disable a default-enabled flag, use `--without-os-feature=feat1,feat2,...`

To enable a flag that isn't otherwise enabled, use `--with-os-feature=feat`. For example, modern Linux systems support the `getentropy` call, but it is not enabled by default because many older systems lack it. However if you know you will only deploy to recently updated systems you can use `--with-os-feature=getentropy` to enable it.

A special case if dynamic loading, which applications for certain environments will want to disable. There is no specific feature flag for this, but `--disable-modules=dyn_load` will prevent it from being used.

Note: Disabling `dyn_load` module will also disable the PKCS #11 wrapper, which relies on dynamic loading.

4.10.8 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_RSA` is defined, then an application knows that this version of the library has RSA available.

`BOTAN_MP_WORD_BITS`: This macro controls the size of the words used for calculations with the MPI implementation in Botan. It must be set to either 32 or 64 bits. The default is chosen based on the target processor. There is normally no reason to change this.

`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.

4.11 Building Applications

4.11.1 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 itself).

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.

4.11.2 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.

4.12 Language Wrappers

4.12.1 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.

4.13 Minimized Builds

Many developers wish to configure a minimized build which contains only the specific features their application will use. In general this is straighforward: use `--minimized-build` plus `--enable-modules=` to enable the specific modules you wish to use. Any such configurations should build and pass the tests; if you encounter a case where it doesn't please file an issue.

The only trick is knowing which features you want to enable. The most common difficulty comes with entropy sources. By default, none are enabled, which means if you attempt to use `AutoSeeded_RNG`, it will fail. The easiest resolution is to also enable `system_rng` which can act as either an entropy source or used directly as the RNG.

If you are building for x86, ARM, or POWER, it can be beneficial to enable hardware support for the relevant instruction sets with modules such as `aes_ni` and `clmul` for x86, or `aes_armv8`, `pmull`, and `sha2_32_armv8` on ARMv8. SIMD optimizations such as `chacha_avx2` also can provide substantial performance improvements.

Note: In a future release, hardware specific modules will be enabled by default if the underlying "base" module is enabled.

If you are building a TLS application, you may (or may not) want to include `tls_cbc` which enables support for CBC ciphersuites. If `tls_cbc` is disabled, then it will not be possible to negotiate TLS v1.0/v1.1. In general this should be considered a feature; only enable this if you need backward compatability with obsolete clients or servers.

For TLS another useful feature which is not enabled by default is the ChaCha20Poly1305 ciphersuites. To enable these, add `chacha20poly1305`.

4.14 Configure Script Options

4.14.1 `--cpu=CPU`

Set the target CPU architecture. If not used, the arch of the current system is detected (using Python's platform module) and used.

4.14.2 `--os=OS`

Set the target operating system.

4.14.3 `--cc=COMPILER`

Set the desired build compiler

4.14.4 `--cc-min-version=MAJOR.MINOR`

Set the minimal version of the target compiler. Use –cc-min-version=0.0 to support all compiler versions. Default is auto detection.

4.14.5 `--cc-bin=BINARY`

Set path to compiler binary

If not provided, the value of the `CXX` environment variable is used if set.

4.14.6 `--cc-abi-flags=FLAGS`

Set ABI flags, which for the purposes of this option mean options which should be passed to both the compiler and linker.

4.14.7 `--cxxflags=FLAGS`

Override all compiler flags. This is equivalent to setting `CXXFLAGS` in the environment.

4.14.8 `--extra-cxxflags=FLAGS`

Set extra compiler flags, which are appended to the default set. This is useful if you want to set just one or two additional options but leave the normal logic for selecting flags alone.

4.14.9 `--ldflags=FLAGS`

Set flags to pass to the linker. This is equivalent to setting `LDFLAGS`

4.14.10 `--ar-command=AR`

Set the path to the tool to use to create static archives (`ar`). This is normally only used for cross-compilation.

If not provided, the value of the `AR` environment variable is used if set.

4.14.11 `--ar-options=AR_OPTIONS`

Specify the options to pass to `ar`.

If not provided, the value of the `AR_OPTIONS` environment variable is used if set.

4.14.12 `--msvc-runtime=RT`

Specify the MSVC runtime to use (MT, MD, MTd, or MDd). If not specified, picks either MD or MDd depending on if debug mode is set.

4.14.13 `--with-endian=ORDER`

The parameter should be either "little" or "big". If not used then if the target architecture has a default, that is used. Otherwise left unspecified, which causes less optimal codepaths to be used but will work on either little or big endian.

4.14.14 `--with-os-features=FEAT`

Specify an OS feature to enable. See `src/build-data/os` and `doc/os.rst` for more information.

4.14.15 `--without-os-features=FEAT`

Specify an OS feature to disable.

4.14.16 `--disable-sse2`

Disable use of SSE2 intrinsics

4.14.17 `--disable-ssse3`

Disable use of SSSE3 intrinsics

4.14.18 `--disable-sse4.1`

Disable use of SSE4.1 intrinsics

4.14.19 `--disable-sse4.2`

Disable use of SSE4.2 intrinsics

4.14.20 `--disable-avx2`

Disable use of AVX2 intrinsics

4.14.21 `--disable-bmi2`

Disable use of BMI2 intrinsics

4.14.22 `--disable-rdrand`

Disable use of RDRAND intrinsics

4.14.23 `--disable-rdseed`

Disable use of RDSEED intrinsics

4.14.24 `--disable-aes-ni`

Disable use of AES-NI intrinsics

4.14.25 `--disable-sha-ni`

Disable use of SHA-NI intrinsics

4.14.26 `--disable-altivec`

Disable use of AltiVec intrinsics

4.14.27 `--disable-neon`

Disable use of NEON intrinsics

4.14.28 `--disable-armv8crypto`

Disable use of ARMv8 Crypto intrinsics

4.14.29 `--disable-powercrypto`

Disable use of POWER Crypto intrinsics

4.14.30 `--with-debug-info`

Include debug symbols.

4.14.31 `--with-sanitizers`

Enable some default set of sanitizer checks. What exactly is enabled depends on the compiler.

4.14.32 `--enable-sanitizers=SAN`

Enable specific sanitizers. See `src/build-data/cc` for more information.

4.14.33 `--without-stack-protector`

Disable stack smashing protections. **not recommended**

4.14.34 `--with-coverage`

Add coverage info and disable optimizations

4.14.35 `--with-coverage-info`

Add coverage info, but leave optimizations alone

4.14.36 `--disable-shared-library`

Disable building a shared library

4.14.37 `--disable-static-library`

Disable building static library

4.14.38 `--optimize-for-size`

Optimize for code size.

4.14.39 `--no-optimizations`

Disable all optimizations for debugging.

4.14.40 `--debug-mode`

Enable debug info and disable optimizations

4.14.41 `--amalgamation`

Use amalgamation to build

4.14.42 `--system-cert-bundle=PATH`

Set a path to a file containing one or more trusted CA certificates in PEM format. If not given, some default locations are checked.

4.14.43 `--with-build-dir=DIR`

Setup the build in a specified directory instead of `./build`

4.14.44 `--with-external-includedir=DIR`

Search for includes in this directory. Provide this parameter multiple times to define multiple additional include directories.

4.14.45 `--with-external-libdir=DIR`

Add DIR to the link path. Provide this parameter multiple times to define multiple additional library link directories.

4.14.46 `--define-build-macro`

Set a compile-time pre-processor definition (i.e. add a -D… to the compiler invocations). Provide this parameter multiple times to add multiple compile-time definitions. Both KEY=VALUE and KEY (without specific value) are supported.

4.14.47 `--with-sysroot-dir=DIR`

Use specified dir for system root while cross-compiling

4.14.48 `--with-openmp`

Enable use of OpenMP

4.14.49 `--link-method=METHOD`

During build setup a directory linking to each header file is created. Choose how the links are performed (options are "symlink", "hardlink", or "copy").

4.14.50 `--with-local-config=FILE`

Include the contents of FILE into the generated build.h

4.14.51 `--distribution-info=STRING`

Set distribution specific version information

4.14.52 `--maintainer-mode`

A build configuration used by library developers, which enables extra warnings and turns most warnings into errors.

Warning: When this option is used, all relevant warnings available in the most recent release of GCC/Clang are enabled, so it may fail to build if your compiler is not sufficiently recent. In addition there may be non-default configurations or unusual platforms which cause warnings which are converted to errors. Patches addressing such warnings are welcome, but otherwise no support is available when using this option.

4.14.53 `--werror-mode`

Turns most warnings into errors.

4.14.54 `--no-install-python-module`

Skip installing Python module.

4.14.55 `--with-python-versions=N.M`

Where to install botan2.py. By default this is chosen to be the version of Python that is running `configure.py`.

4.14.56 `--with-valgrind`

Use valgrind API to perform additional checks. Not needed by end users.

4.14.57 `--unsafe-fuzzer-mode`

Disable essential checks for testing. **UNSAFE FOR PRODUCTION**

4.14.58 `--build-fuzzers=TYPE`

Select which interface the fuzzer uses. Options are "afl", "libfuzzer", "klee", or "test". The "test" mode builds fuzzers that read one input from stdin and then exit.

4.14.59 `--with-fuzzer-lib=LIB`

Specify an additional library that fuzzer binaries must link with.

4.14.60 `--build-targets=BUILD_TARGETS`

Build only the specific targets and tools (`static`, `shared`, `cli`, `tests`, `bogo_shim`).

4.14.61 `--boost-library-name`

Provide an alternative name for a boost library. Depending on the platform and boost's build configuration these library names differ significantly (see Boost docs (https://www.boost.org/doc/libs/1_70_0/more/getting_started/unix-variants.html#library-naming)). The provided library name must be suitable as identifier in a linker parameter, e.g on unix: `boost_system` or windows: `libboost_regex-vc71-x86-1_70`.

4.14.62 `--without-documentation`

Skip building/installing documentation

4.14.63 `--with-sphinx`

Use Sphinx to generate the handbook

4.14.64 `--with-pdf`

Use Sphinx to generate PDF doc

4.14.65 `--with-rst2man`

Use rst2man to generate a man page for the CLI

4.14.66 `--with-doxygen`

Use Doxygen to generate API reference

4.14.67 `--module-policy=POL`

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
```

4.14.68 `--enable-modules=MODS`

Enable some specific modules

4.14.69 `--disable-modules=MODS`

Disable some specific modules

4.14.70 `--minimized-build`

Start with the bare minimum. This is mostly useful in conjuction with `--enable-modules` to get a build that has just the features a particular application requires.

4.14.71 `--with-boost`

Use Boost.Asio for networking support. This primarily affects the command line utils.

4.14.72 `--with-bzip2`

Enable bzip2 compression

4.14.73 `--with-lzma`

Enable lzma compression

4.14.74 `--with-zlib`

Enable using zlib compression

4.14.75 `--with-openssl`

Enable using OpenSSL for certain operations

4.14.76 `--with-commoncrypto`

Enable using CommonCrypto for certain operations

4.14.77 `--with-sqlite3`

Enable using sqlite3 for data storage

4.14.78 `--with-tpm`

Enable support for TPM

4.14.79 `--program-suffix=SUFFIX`

A string to append to all program binaries.

4.14.80 `--library-suffix=SUFFIX`

A string to append to all library names.

4.14.81 `--prefix=DIR`

Set the install prefix.

4.14.82 `--docdir=DIR`

Set the documentation installation dir.

4.14.83 `--bindir=DIR`

Set the binary installation dir.

4.14.84 `--libdir=DIR`

Set the library installation dir.

4.14.85 `--mandir=DIR`

Set the man page installation dir.

4.14.86 `--includedir=DIR`

Set the include file installation dir.

CHAPTER

FIVE

API REFERENCE

5.1 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 corresponding 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 discrepancy. 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
```

5.2 Memory container

A major concern with mixing modern multi-user 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 retrieval.

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 privileged 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).

5.3 Random Number Generators

class RandomNumberGenerator

The base class for all RNG objects, is declared in `rng.h`.

void **randomize**(uint8_t **output_array*, size_t *length*)

Places *length* random bytes into the provided buffer.

void **randomize_with_input**(uint8_t **data*, size_t *length*, **const** uint8_t **extra_input*, size_t *extra_input_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, the value will be silently ignored when not supported.

void **randomize_with_ts_input**(uint8_t **data*, size_t *length*)

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

Note: When RDRAND is enabled and available at runtime, instead of timestamps the output of RDRAND is used as the additional data.

uint8_t **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.

void **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.

bool **accepts_input**() **const**

This function returns `false` if it is known that this RNG object cannot accept external inputs. In this case, any calls to `RandomNumberGenerator::add_entropy` will be ignored.

void **reseed_from_rng**(RandomNumberGenerator *&rng*, size_t *poll_bits* = BOTAN_RNG_RESEED_POLL_BITS)

Reseed by calling `rng` to acquire `poll_bits` data.

5.3.1 RNG Types

Several different RNG types are implemented. Some access hardware RNGs, which are only available on certain platforms. Others are mostly useful in specific situations.

Generally prefer using the system RNG, or if not available use `AutoSeeded_RNG` which is intended to provide best possible behavior in a userspace PRNG.

System_RNG

On systems which support it, in `system_rng.h` you can access a shared reference to a process global instance of the system PRNG (using interfaces such as `/dev/urandom`, `getrandom`, `arc4random`, or `RtlGenRandom`):

RandomNumberGenerator &**system_rng**()

Returns a reference to the system RNG

There is also a wrapper class `System_RNG` which simply invokes on the return value of `system_rng()`. This is useful in situations where you may sometimes want to use the system RNG and a userspace RNG in others, for example:

```
std::unique_ptr<Botan::RandomNumberGenerator> rng;
#if defined(BOTAN_HAS_SYSTEM_RNG)
rng.reset(new System_RNG);
#else
rng.reset(new AutoSeeded_RNG);
#endif
```

Unlike nearly any other object in Botan it is acceptable to share a single instance of `System_RNG` between threads, because the underlying RNG is itself thread safe due to being serialized by a mutex in the kernel itself.

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 compatibility concerns when changing any RNG since the only expectation is it produces bits indistinguishable from random.

Note: Starting in 2.16.0, AutoSeeded_RNG uses an internal lock and so is safe to share among threads. However if possible it is still better to use a RNG per thread as otherwise the RNG object needlessly creates a point of contention. In previous versions, the RNG does not have an internal lock and all access to it must be serialized.

The current version uses HMAC_DRBG with either 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.

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.

HMAC_DRBG's constructors are:

class HMAC_DRBG

HMAC_DRBG(std::unique_ptr<*MessageAuthenticationCode*> *prf*, *RandomNumberGenerator* *&underlying_rng*, size_t *reseed_interval* = BOTAN_RNG_DEFAULT_RESEED_INTERVAL, size_t *max_number_of_bytes_per_request* = 64 * 1024)

Creates a DRBG which will automatically reseed as required by making calls to `underlying_rng` either after being invoked `reseed_interval` times, or if use of `fork` system call is detected.

You can disable automatic reseeding by setting `reseed_interval` to zero, in which case `underlying_rng` will only be invoked in the case of `fork`.

The specification of HMAC DRBG requires that each invocation produce no more than 64 kibibytes of data. However, the RNG interface allows producing arbitrary amounts of data in a single request. To accommodate this, `HMAC_DRBG` treats requests for more data as if they were multiple requests each of (at most) the maximum size. You can specify a smaller maximum size with `max_number_of_bytes_per_request`. There is normally no reason to do this.

HMAC_DRBG(std::unique_ptr<*MessageAuthenticationCode*> *prf*, Entropy_Sources *&entropy_sources*, size_t *reseed_interval* = BOTAN_RNG_DEFAULT_RESEED_INTERVAL, size_t *max_number_of_bytes_per_request* = 64 * 1024)

Like above function, but instead of an RNG taking a set of entropy sources to seed from as required.

HMAC_DRBG(std::unique_ptr<*MessageAuthenticationCode*> *prf*, *RandomNumberGenerator* *&underlying_rng*, Entropy_Sources *&entropy_sources*, size_t *reseed_interval* = BOTAN_RNG_DEFAULT_RESEED_INTERVAL, size_t *max_number_of_bytes_per_request* = 64 * 1024)

Like above function, but taking both an RNG and a set of entropy sources to seed from as required.

HMAC_DRBG(std::unique_ptr<*MessageAuthenticationCode*> *prf*)

Creates an unseeded DRBG. You must explicitly provide seed data later on in order to use this RNG. This is primarily useful for deterministic key generation.

Since no source of data is available to automatically reseed, automatic reseeding is disabled when this constructor is used. If the RNG object detects that `fork` system call was used without it being subsequently reseeded, it will throw an exception.

HMAC_DRBG(**const** std::string *&hmac_hash*)

Like the constructor just taking a PRF, except instead of a PRF object, a string specifying what hash to use with HMAC is provided.

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 (basically running as fast as ChaCha can generate keystream bits), and certain applications need access to a very fast RNG.

One thing applications using `ChaCha_RNG` need to be aware of is that for performance reasons, no backtracking resistance is implemented in the RNG design. An attacker who recovers the `ChaCha_RNG` state can recover the output backwards in time to the last rekey and forwards to the next rekey.

An explicit reseeding (*`RandomNumberGenerator::add_entropy`*) or providing any input to the RNG (*`RandomNumberGenerator::randomize_with_ts_input`*, *`RandomNumberGenerator::randomize_with_input`*) is sufficient to cause a reseeding. Or, if a RNG or entropy source was provided to the `ChaCha_RNG` constructor, then reseeding will be performed automatically after a certain interval of requests.

Processor_RNG

This RNG type directly invokes a CPU instruction capable of generating a cryptographically secure random number. On x86 it uses `rdrand`, on POWER `darn`. If the relevant instruction is not available, the constructor of the class will throw at runtime. You can test beforehand by checking the result of `Processor_RNG::available()`.

TPM_RNG

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

PKCS11_RNG

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

5.3.2 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.

The following entropy sources are currently used:

- The system RNG (`arc4random`, `/dev/urandom`, or `RtlGenRandom`).
- RDRAND and RDSEED are used if available, but not counted as contributing entropy

- `/dev/random` and `/dev/urandom`. This may be redundant with the system RNG
- `getentropy`, only used on OpenBSD currently
- `/proc` walk: read files in `/proc`. Last ditch protection against flawed system RNG.
- Win32 stats: takes snapshot of current system processes. Last ditch protection against flawed system RNG.

5.3.3 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.

5.4 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

> **static** std::unique_ptr<*HashFunction*> **create**(**const** std::string &*name*)
>
> Return a newly allocated hash function object, or nullptr if the name is not recognized.
>
> **static** std::unique_ptr<*HashFunction*> **create_or_throw**(**const** std::string &*name*)
>
> Like `create` except that it will throw an exception instead of returning nullptr.
>
> 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`.

5.4.1 Code Example

Assume we want to calculate the SHA-256, SHA-384, 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-256"));
   std::unique_ptr<Botan::HashFunction> hash2(Botan::HashFunction::create("SHA-384"));
   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-256: " << Botan::hex_encode(hash1->final()) << std::endl;
   std::cout << "SHA-384: " << Botan::hex_encode(hash2->final()) << std::endl;
   std::cout << "SHA-3: " << Botan::hex_encode(hash3->final()) << std::endl;
   return 0;
   }
```

5.4.2 Available Hash Functions

The following cryptographic hash functions are implemented. If in doubt, any of SHA-384, SHA-3, or BLAKE2b are fine choices.

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.

Named like "Blake2b" which selects default 512-bit output, or as "Blake2b(256)" to select 256 bits of output.

GOST-34.11

Deprecated since version 2.11.

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.

Warning: As this hash function is no longer approved by the latest Russian standards, support for GOST 34.11 hash is deprecated and will be removed in a future major release.

Keccak-1600

Available if `BOTAN_HAS_KECCAK` is defined.

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

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.

Warning: Support for MD4 is deprecated and will be removed in a future major release.

MD5

Available if `BOTAN_HAS_MD5` is defined.

Widely used, now known to be broken.

RIPEMD-160

Available if `BOTAN_HAS_RIPEMD160` is defined.

A 160 bit hash function, quite old but still thought to be secure (up to the limit of 2**80 computation required for a collision which is possible with any 160 bit hash function). Somewhat deprecated these days.

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.

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.

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, which have the advantage of avoiding message extension attacks.

SHA-3

Available if `BOTAN_HAS_SHA3` is defined.

The new NIST standard hash. Fairly slow.

Supports 224, 256, 384 or 512 bit outputs. SHA-3 is faster with smaller outputs. Use as "SHA-3(256)" or "SHA-3(512)". Plain "SHA-3" selects default 512 bit output.

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 byte length. For example "SHAKE-128(1024)" will produce 1024 bits of output. The specified length must be a multiple of 8. Not specifying an output length, "SHAKE-128" defaults to a 128-bit output and "SHAKE-256" defaults to a 256-bit output.

Warning: In the case of SHAKE-128, the default output length in insufficient to ensure security. The choice of default lengths was a bug which is currently retained for compatability; they should have been 256 and 512 bits resp to match SHAKE's security level. Using the default lengths with SHAKE is deprecated and will be removed in a future major release. Instead, always specify the desired output length.

SM3

Available if `BOTAN_HAS_SM3` is defined.

Chinese national hash function, 256 bit output. Widely used in industry there. Fast and seemingly secure, but no reason to prefer it over SHA-2 or SHA-3 unless required.

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 an optional "personalization string" which can create variants of the hash. This is useful for domain separation.

To set a personalization string set the second param to any value, typically ASCII strings are used. Examples "Skein-512(256)" or "Skein-512(384,personalization_string)".

Streebog (Streebog-256, Streebog-512)

Available if `BOTAN_HAS_STREEBOG` is defined.

Newly designed Russian national hash function. Due to use of input-dependent table lookups, it is vulnerable to side channels. There is no reason to use it unless compatibility is needed.

Warning: The Streebog Sbox has recently been revealed to have a hidden structure which interacts with its linear layer in a way which may provide a backdoor when used in certain ways. Avoid Streebog if at all possible.

Tiger

Deprecated since version 2.15.

Available if `BOTAN_HAS_TIGER` is defined.

An older 192-bit hash function, optimized for 64-bit systems. Possibly vulnerable to side channels due to its use of table lookups.

Tiger supports variable length output (16, 20 or 24 bytes) and variable rounds (which must be at least 3). Default is 24 byte output and 3 rounds. Specify with names like "Tiger" or "Tiger(20,5)".

Warning: There are documented (albeit impractical) attacks on the full Tiger hash leading to preimage attacks. This indicates possibility of a serious weakness in the hash and for this reason it is deprecated and will be removed in a future major release of the library.

Whirlpool

Available if `BOTAN_HAS_WHIRLPOOL` is defined.

A 512-bit hash function standardized by ISO and NESSIE. Relatively slow, and due to the table based implementation it is potentially vulnerable to cache based side channels.

5.4.3 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.

Parallel

Available if `BOTAN_HAS_PARALLEL_HASH` is defined.

Parallel simply concatenates 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.

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.

5.4.4 Checksums

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

Adler32

Available if `BOTAN_HAS_ADLER32` is defined.

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

CRC24

Available if `BOTAN_HAS_CRC24` is defined.

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

CRC32

Available if `BOTAN_HAS_CRC32` is defined.

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

5.5 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.

Warning: In almost all cases, a bare block cipher is not what you should be using. You probably want an authenticated cipher mode instead (see *Cipher Modes*) This interface is used to build higher level operations (such as cipher modes or MACs), or in the very rare situation where ECB is required, eg for compatibility with an existing system.

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 acceptable for the algorithm.

size_t **maximum_keylength**() **const**
: Return the largest key length (in bytes) that is acceptable 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 parallelism 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.

5.5.1 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;
   }
```

5.5.2 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. If you are developing new code and have no particular opinion, use AES-256. If you desire an alternative to AES, consider Serpent, SHACAL2 or Threefish.

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

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. For x86 systems with SSSE3 but without AES-NI, Botan has an implementation which avoids known side channels.

Available if `BOTAN_HAS_AES` is defined.

ARIA

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

Available if `BOTAN_HAS_ARIA` is defined.

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.

CAST-128

A 64-bit cipher, commonly used in OpenPGP.

Available if `BOTAN_HAS_CAST128` is defined.

CAST-256

A 128-bit cipher that was a contestant in the NIST AES competition. Almost never used in practice. Prefer AES or Serpent.

Available if `BOTAN_HAS_CAST256` is defined.

Warning: Support for CAST-256 is deprecated and will be removed in a future major release.

Camellia

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

A Japanese design standardized by ISO, NESSIE and CRYPTREC. Rarely used outside of Japan.

Available if `BOTAN_HAS_CAMELLIA` is defined.

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.

DES, 3DES, DESX

Originally designed by IBM and NSA in the 1970s. Today, DES's 56-bit key renders it insecure to any well-resourced attacker. DESX and 3DES extend the key length, and are still thought to be secure, modulo the limitation of a 64-bit block. All are somewhat common in some industries such as finance. Avoid in new code.

Warning: Support for DESX is deprecated and it will be removed in a future major release.

Available if `BOTAN_HAS_DES` is defined.

GOST-28147-89

Aka “Magma”. An old 64-bit Russian cipher. Possible security issues, avoid unless compatibility is needed.

Available if `BOTAN_HAS_GOST_28147_89` is defined.

Warning: Support for this cipher is deprecated and will be removed in a future major release.

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.

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.

Warning: Support for Kasumi is deprecated and will be removed in a future major release.

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.

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.

Available if `BOTAN_HAS_MISTY1` is defined.

Warning: Support for MISTY1 is deprecated and will be removed in a future major release.

Noekeon

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

Available if `BOTAN_HAS_NOEKEON` is defined.

Warning: Support for Noekeon is deprecated and will be removed in a future major release.

SEED

A older South Korean cipher, widely used in industry there. No reason to choose it otherwise.

Available if `BOTAN_HAS_SEED` is defined.

SHACAL2

The 256-bit block cipher used inside SHA-256. Accepts up to a 512-bit key. Fast, especially when SIMD or SHA-2 acceleration instructions are available. Standardized by NESSIE but otherwise obscure.

Available if `BOTAN_HAS_SHACAL2` is defined.

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.

Serpent

An AES contender. Widely considered the most conservative design. Fairly slow unless SIMD instructions are available.

Available if `BOTAN_HAS_SERPENT` is defined.

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.

Twofish

A 128-bit block cipher that was one of the AES finalists. Has a somewhat complicated key setup and a “kitchen sink” design.

Available if `BOTAN_HAS_TWOFISH` is defined.

XTEA

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

Available if `BOTAN_HAS_XTEA` is defined.

5.6 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*).

Warning: Using a stream cipher without an authentication code is extremely insecure, because an attacker can trivially modify messages. Prefer using an authenticated cipher mode such as GCM or SIV.

Warning: Encrypting more than one message with the same key requires careful management of initialization vectors. Otherwise the keystream will be reused, which causes the security of the cipher to completely fail.

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 acceptable for the algorithm.

size_t **maximum_keylength**() **const**
: Return the largest key length (in bytes) that is acceptable 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.

size_t **default_iv_length**() **const**
: Returns some default IV size, normally the largest IV supported by the cipher. If this function returns zero, then IVs are not supported and any call to `set_iv` with a non-empty value will fail.

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.

  If the cipher does not support IVs, then a call with `len` equal to zero will be accepted and any other length will cause a `Invalid_IV_Length` exception.

void **seek**(uint64_t *offset*)
: Sets the state of the stream cipher and keystream according to the passed *offset*, exactly as if *offset* bytes had first been encrypted. The key and (if required) the IV have to be set before this can be called. Not all ciphers support seeking; such objects will throw `Not_Implemented` in this case.

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()).

5.6.1 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;
   }
```

5.6.2 Available Stream Ciphers

Botan provides the following stream ciphers. If in doubt use ChaCha20 or CTR(AES-256).

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, both useful properties.

CTR mode requires an IV which can be any length up to the block size of the underlying cipher. If it is shorter than the block size, sufficient zero bytes are appended.

It is possible to choose the width of the counter portion, which can improve performance somewhat, but limits the maximum number of bytes that can safely be encrypted. Different protocols have different conventions for the width of the counter portion. This is done by specifying with width (which must be at least 4 bytes, allowing to encrypt 2^{32} blocks of data) for example "CTR(AES-256,8)" to select a 64-bit counter.

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

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.

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.

ChaCha supports an optional IV (which defaults to all zeros). It can be of length 64, 96 or (since 2.8) 192 bits. Using ChaCha with a 192 bit nonce is also known as XChaCha.

Available if `BOTAN_HAS_CHACHA` is defined.

Salsa20

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

Salsa supports an optional IV (which defaults to all zeros). It can be of length 64 or 192 bits. Using Salsa with a 192 bit nonce is also known as XSalsa.

Available if `BOTAN_HAS_SALSA20` is defined.

SHAKE-128

This is the SHAKE-128 XOF exposed as a stream cipher. It is slower than ChaCha and somewhat obscure. It does not support IVs or seeking within the cipher stream.

Available if `BOTAN_HAS_SHAKE_CIPHER` is defined.

RC4

An old and very widely deployed stream cipher notable for its simplicity. It does not support IVs or seeking within the cipher stream.

Warning: RC4 is now badly broken. **Avoid in new code** and use only if required for compatibility with existing systems.

Available if `BOTAN_HAS_RC4` is defined.

5.7 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 acceptable for the algorithm.

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

5.7.1 Code Examples

The following example computes an HMAC with a random key then verifies the tag.

#include <botan/mac.h> #include <botan/hex.h> #include <botan/system_rng.h> #include <assert.h>

std::string compute_mac(const std::string& msg, const Botan::secure_vector<uint8_t>& key) {
auto hmac = Botan::MessageAuthenticationCode::create_or_throw("HMAC(SHA-256)");

hmac->set_key(key); hmac->update(msg);

return Botan::hex_encode(hmac->final()); }

int main() { Botan::System_RNG rng;

const auto key = rng.random_vec(32); // 256 bit random key

// "Message" != "Mussage" so tags will also not match std::string tag1 = compute_mac("Message", key); std::string tag2 = compute_mac("Mussage", key); assert(tag1 != tag2);

// Recomputing with original input message results in identical tag std::string tag3 = compute_mac("Message", key); assert(tag1 == tag3); }

The following example code computes a AES-256 GMAC and subsequently verifies the tag. Unlike most other MACs, GMAC requires a nonce *which must not repeat or all security is lost*.

```
#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> nonce = Botan::hex_decode("FFFFFFFFFFFFFFFFFFFFFFFF");
   const std::vector<uint8_t> data = Botan::hex_decode(
↪"6BC1BEE22E409F96E93D7E117393172A");
```

(continues on next page)

(continued from previous page)

```
   std::unique_ptr<Botan::MessageAuthenticationCode>␣
↪mac(Botan::MessageAuthenticationCode::create("GMAC(AES-256)"));
   if(!mac)
      return 1;
   mac->set_key(key);
   mac->start(nonce);
   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(nonce);
   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;
     }
```

5.7.2 Available MACs

Currently the following MAC algorithms are available in Botan. In new code, default to HMAC with a strong hash like SHA-256 or SHA-384.

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 compatibility.

Available if `BOTAN_HAS_CBC_MAC` is defined.

Warning: CBC-MAC support is deprecated and will be removed in a future major release.

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.

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.

Warning: Due to the nonce requirement, GMAC is exceptionally fragile. Avoid it unless absolutely required.

HMAC

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

Available if `BOTAN_HAS_HMAC` is defined.

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.

Warning: Due to the nonce requirement, Poly1305 is exceptionally fragile. Avoid it unless absolutely required.

SipHash

A modern and very fast PRF. Produces only a 64-bit output. Defaults to "SipHash(2,4)" which is the recommended configuration, using 2 rounds for each input block and 4 rounds for finalization.

Available if `BOTAN_HAS_SIPHASH` is defined.

X9.19-MAC

A CBC-MAC variant sometimes used in finance. Always uses DES. Sometimes called the "DES retail MAC", also standardized in ISO 9797-1.

It is slow and has known attacks. Avoid unless required.

Available if `BOTAN_HAS_X919_MAC` is defined.

5.8 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 acceptable for the algorithm.

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

size_t **`default_nonce_length`**() **`const`**
: Return the default (preferable) nonce size for this cipher mode.

bool **`valid_nonce_length`**(size_t *nonce_len*) **`const`**
: Return true if *nonce_len* is a valid length for a nonce with this algorithm.

bool **`authenticated`**() **`const`**
: Return true if this cipher mode is authenticated

size_t **`tag_size`**() **`const`**
: Return the length in bytes of the authentication tag this algorithm generates. If the mode is not authenticated, this will return 0. If the mode is authenticated, it will return some positive value (typically somewhere between 8 and 16).

void **clear** ()

Clear all internal state. The object will act exactly like one which was just allocated.

void **reset** ()

Reset all message state. For example if you called `start_msg`, then `process` to process some ciphertext, but then encounter an IO error and must abandon the current message, you can call *reset*. The object will retain the key (unlike calling `clear` which also resets the key) but the nonce and current message state will be erased.

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

Set up for processing a new message. This function must be called with a new random value for each message. For almost all modes (excepting SIV), if the same nonce is ever used twice with the same key, the encryption scheme loses its confidentiality and/or authenticity properties.

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 the number of 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.

5.8.1 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

Note: Simply replacing the string "AES-128/CBC/PKCS7" string in the example below with "AES-128/GCM" suffices to use authenticated encryption.

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

```

(continues on next page)

(continued from previous page)

```
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::Cipher_Mode::create("AES-128/CBC/
↪PKCS7", Botan::ENCRYPTION);
   enc->set_key(key);

   //generate fresh nonce (IV)
   Botan::secure_vector<uint8_t> iv = rng.random_vec(enc->default_nonce_length());

   // Copy input data to a buffer that will be encrypted
   Botan::secure_vector<uint8_t> pt(plaintext.data(), plaintext.data()+plaintext.
↪length());

   enc->start(iv);
   enc->finish(pt);

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

5.8.2 Available Unauthenticated Cipher Modes

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

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.

Ciphertext stealing (CTS) is also implemented. This scheme allows the ciphertext to have the same length as the plaintext, however using CTS requires the input be at least one full block plus one byte. It is also less commonly implemented.

CFB

Available if `BOTAN_HAS_MODE_CFB` is defined.

CFB uses a block cipher to create a self-synchronizing stream cipher. It is used for example in the OpenPGP protocol. There is no reason to prefer it, as it has worse performance characteristics than modes such as CTR or CBC.

XTS

Available if `BOTAN_HAS_MODE_XTS` is defined.

XTS is a mode specialized for encrypting disk or database storage where ciphertext expansion is not possible. XTS requires all inputs be at least one full block (16 bytes for AES), however for any acceptable input length, there is no ciphertext expansion.

5.8.3 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`*.

`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. It does not need to be random, simply unique (per key).

  Warning: With almost all AEADs, if the same nonce is ever used to encrypt two different messages under the same key, all security is lost. If reliably generating unique nonces is difficult in your environment, use SIV mode which retains security even if nonces are repeated.

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, if the supplied authentication tag does not validate, finish will throw an instance of Invalid_Authentication_Tag (aka Integrity_Failure, which was the name for this exception in versions before 2.10, a typedef is included for compatability).

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. This could be anything from echoing the plaintext back (perhaps in an error message), or by making an external RPC whose destination or contents depend on the plaintext. The only thing you can do is buffer it, and in the event of an invalid tag, erase the previously decrypted content from memory.

One simply way to assure this could never happen is to never call update, and instead always marshal 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.

5.8.4 Available AEAD Modes

If in doubt about what to use, pick ChaCha20Poly1305, AES-256/GCM, or AES-256/SIV. Both ChaCha20Poly1305 and AES with GCM are widely implemented. SIV is somewhat more obscure (and is slower than either GCM or ChaCha20Poly1305), but has excellent security properties.

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.

ChaCha20Poly1305 supports 64-bit, 96-bit, and (since 2.8) 192-bit nonces. 64-bit nonces are the "classic" ChaCha20Poly1305 design. 96-bit nonces are used by the IETF standard version of ChaCha20Poly1305. And 192-bit nonces is the XChaCha20Poly1305 construction, which is somewhat less common.

For best interop use the IETF version with 96-bit nonces. However 96 bits is small enough that it can be dangerous to generate nonces randomly if more than ~ 2^32 messages are encrypted under a single key, since if a nonce is ever reused ChaCha20Poly1305 becomes insecure. It is better to use a counter for the nonce in this case.

If you are encrypting many messages under a single key and cannot maintain a counter for the nonce, prefer XChaCha20Poly1305 since a 192 bit nonce is large enough that randomly chosen nonces are extremely unlikely to repeat.

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.

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.

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.

SIV

Available if `BOTAN_HAS_AEAD_SIV` is defined.

Requires a 128-bit block cipher. Unlike other AEADs, SIV is “misuse resistant”; 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 fact that the message was duplicated (this is simply because if both the nonce and the message are reused, SIV will output identical ciphertexts).

CCM

Available if `BOTAN_HAS_AEAD_CCM` is defined.

A composition of CTR mode and CBC-MAC. Requires a 128-bit block cipher. This is a NIST standard mode, but that is about all to recommend it. Prefer EAX.

5.9 Public Key Cryptography

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

5.9.1 Key Objects

Public and private keys are represented by classes `Public_Key` and it's subclass `Private_Key`. The use of inheritance 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 similarly 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 corresponding `Private_Key` classes for each of these algorithms.

5.9.2 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*.

RSA key generation is relatively slow, and can take an unpredictable amount of time. Generating a 2048 bit RSA key might take 5 to 10 seconds on a slow machine like a Raspberry Pi 2. Even on a fast desktop it might take up to half a second. In a GUI blocking for that long can be a problem. The usual approach is to perform key generation in a new thread, with a animated modal UI element so the user knows the application is still alive. If you wish to provide a progress estimate things get a bit complicated but some library users documented their approach in a blog post (https://medium.com/nexenio/indicating-progress-of-rsa-key-pair-generation-the-practical-approach-a049ba829dbe).

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 *Password Based Key Derivation* 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)

5.9.3 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.

The currently supported PBE is PBES2 from PKCS5. Format is as follows: `PBE-PKCS5v20(CIPHER, PBKDF)`. Since 2.8.0, `PBES2(CIPHER,PBKDF)` also works. Cipher can be any block cipher with /CBC or /GCM appended, for example "AES-128/CBC" or "Camellia-256/GCM". For best interop with other systems, use AES in CBC mode. The PBKDF can be either the name of a hash function (in which case PBKDF2 is used with that hash) or "Scrypt", which causes the scrypt memory hard password hashing function to be used. Scrypt is supported since version 2.7.0.

Use *PBE-PKCS5v20(AES-256/CBC,SHA-256)* if you want to ensure the keys can be imported by different software packages. Use *PBE-PKCS5v20(AES-256/GCM,Scrypt)* for best security assuming you do not care about interop.

For ciphers you can use anything which has an OID defined for CBC, GCM or SIV modes. Currently this includes AES, Camellia, Serpent, Twofish, and SM4. Most other libraries only support CBC mode for private key encryption. GCM has been supported in PBES2 since 1.11.10. SIV has been supported since 2.8.

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 unencrypted 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).

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 available.

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;
   }
```

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".

5.9.4 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>
```

(continues on next page)

(continued from previous page)

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

5.9.5 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 parameters 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

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.

```
#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
```

(continues on next page)

(continued from previous page)

```
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)");
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;
}
```

5.9.6 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 associated 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 associated 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

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>
#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;
  }
```

Ed25519 Variants

Most signature schemes in Botan follow a hash-then-sign paradigm. That is, the entire message is digested to a fixed length representative using a collision resistant hash function, and then the digest is signed. Ed25519 instead signs the message directly. This is beneficial, in that the Ed25519 design should remain secure even in the (extremely unlikely) event that a collision attack on SHA-512 is found. However it means the entire message must be buffered in memory, which can be a problem for many applications which might need to sign large inputs. To use this variety of Ed25519, use a padding name of “Pure”.

Ed25519ph (pre-hashed) instead hashes the message with SHA-512 and then signs the digest plus a special prefix specified in RFC 8032. To use it, specify padding name “Ed25519ph”.

Another variant of pre-hashing is used by GnuPG. There the message is digested with any hash function, then the digest is signed. To use it, specify any valid hash function. Even if SHA-512 is used, this variant is not compatible with Ed25519ph.

For best interop with other systems, prefer “Ed25519ph”.

5.9.7 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

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);
```

(continues on next page)

(continued from previous page)

```
   // 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;
   }
```

5.9.8 McEliece

McEliece is a cryptographic scheme based on error correcting codes which is thought to be resistant 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 intractable, 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 corresponding 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`

5.9.9 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 RFC 8391 "XMSS: eXtended Merkle Signature Scheme" (https://tools.ietf.org/html/rfc8391).

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

1. XMSS-SHA2_10_256 # XMSS-SHA2_16_256 # XMSS-SHA2_20_256 # XMSS-SHA2_10_512 # XMSS-SHA2_16_512 # XMSS-SHA2_20_512 # XMSS-SHAKE_10_256 # XMSS-SHAKE_16_256 # XMSS-SHAKE_20_256 # XMSS-SHAKE_10_512 # XMSS-SHAKE_16_512 # XMSS-SHAKE_20_512

The algorithm name contains the hash function name, tree height and digest width defined by the corresponding parameter set. Choosing *XMSS-SHA2_10_256* for instance will use the SHA2-256 hash function to generate a tree of height ten.

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 <iostream>
#include <botan/secmem.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_10_256,
      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, "", "");
```

(continues on next page)

(continued from previous page)

```
   // 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;
      }
   }
```

5.10 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 the class `X509_Certificate`. 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.

`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`.

X509_DN **`subject_dn`**() **const**
: Returns the distinguished name (DN) of the certificate's subject. This is the primary place where information about the subject of the certificate is stored. However "modern" information that doesn't fit in the X.500 framework, such as DNS name, email, IP address, or XMPP address, appears instead in the subject alternative name.

X509_DN **`issuer_dn`**() **const**
: Returns the distinguished name (DN) of the certificate's issuer, ie the CA that issued this certificate.

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::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> **subject_public_key_bits**() **const**
Return the binary encoding of the subject public key. This value (or a hash of it) is used in various protocols, eg for public key pinning.

AlgorithmIdentifier **subject_public_key_algo**() **const**
Return an algorithm identifier that identifies the algorithm used in the subject's public key.

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

std::vector<uint8> **raw_subject_dn**() **const**
Return the binary encoding of the subject DN.

std::vector<uint8> **raw_issuer_dn**() **const**
Return the binary encoding of the issuer DN.

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.

std::vector<uint8_t> **authority_key_id**() **const**
Return the authority key id, if set. This is an arbitrary string; in the issuing certificate this will be the subject key id.

std::vector<uint8_t> **subject_key_id**() **const**
Return the subject key id, if set.

bool **allowed_extended_usage**(**const** OID &*usage*) **const**
Return true if and only if the usage OID appears in the extended key usage extension. Also will return true if the extended key usage extension is not used in the current certificate.

std::vector<OID> **extended_key_usage**() **const**
Return the list of extended key usages. May be empty.

std::string **fingerprint**(**const** std::string &*hash_fn* = "SHA-1") **const**
Return a fingerprint for the certificate, which is basically just a hash of the binary contents. Normally SHA-1 or SHA-256 is used, but any hash function is allowed.

Key_Constraints **constraints**() **const**
Returns either an enumeration listing key constraints (what the associated key can be used

for) or `NO_CONSTRAINTS` if the relevant extension was not included. Example values are `DIGITAL_SIGNATURE` and `KEY_CERT_SIGN`. More than one value might be specified.

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.

std::string **`PEM_encode`**() **`const`**
: Returns the PEM encoding of the certificate

std::vector<uint8_t> **`BER_encode`**() **`const`**
: Returns the DER/BER encoding of the certificate

5.10.1 X.509 Distinguished Names

`class X509_DN`

bool **`has_field`**(**`const`** std::string *&attr*) **`const`**
: Returns true if `get_attribute` or `get_first_attribute` will return a value.

std::vector<std::string> **`get_attribute`**(**`const`** std::string *&attr*) **`const`**
: Return all attributes associated with a certain attribute type.

std::string **`get_first_attribute`**(**`const`** std::string *&attr*) **`const`**
: Like `get_attribute` but returns just the first attribute, or empty if the DN has no attribute of the specified type.

std::multimap<OID, std::string> **`get_attributes`**() **`const`**
: Get all attributes of the DN. The OID maps to a DN component such as 2.5.4.10 ("Organization"), and the strings are UTF-8 encoded.

std::multimap<std::string, std::string> **`contents`**() **`const`**
: Similar to `get_attributes`, but the OIDs are decoded to strings.

void **`add_attribute`**(**`const`** std::string *&key*, **`const`** std::string *&val*)
: Add an attribute to a DN.

void **`add_attribute`**(**`const`** OID *&oid*, **`const`** std::string *&val*)
: Add an attribute to a DN using an OID instead of string-valued attribute type.

The `X509_DN` type also supports iostream extraction and insertion operators, for formatted input and output.

5.10.2 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 extensions object. If an extension of the same type already exists, `extn` will replace it. If `critical` is true the extension will be marked as critical in the encoding.

bool **`add_new`**(Certificate_Extension **extn*, bool *critical* = false)
: Like `add` but an existing extension will not be replaced. Returns true if the extension was used, false if an extension of the same type was already in place.

void **`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.

5.10.3 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.

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.

5.10.4 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

5.10.5 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 revocation 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.

Path Validation

The process of validating a certificate 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* = std::vector<std::shared_ptr<**const** OCSP::*Response*>>())
: 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 completely. `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 flavors 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 relatively 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 corresponds 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.

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*)

5.10.6 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.

5.10.7 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).

5.10.8 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.

PKCS10_Request `PKCS10_Request::`**create**(**const** Private_Key *&key*, **const** *X509_DN &subject_dn*, **const** *Extensions &extensions*, **const** std::string *&hash_fn*, *RandomNumberGenerator &rng*, **const** std::string *&padding_scheme* = "", **const** std::string *&challenge* = "")

This function (added in 2.5) is similar to `create_cert_req` but allows specifying all the parameters directly. In fact `create_cert_req` just creates the DN and extensions from the options, then uses this call to actually create the `PKCS10_Request` object.

5.10.9 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`.

Additionally there are a small selection of `std::vector<std::string>` members, which allow space for repeating elements: `more_org_units` and `more_dns`.

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, which is useful if you want to encode a custom extension, or encode an extension in a way differently from how Botan defaults.

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** std::vector<std::shared<**const** *X509_Certificate*>>()) **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`.

5.11 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 analogous 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 pigeons, 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 asynchronous 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, corresponding 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.

void **`tls_verify_cert_chain`**(**`const`** std::vector<*X509_Certificate*> &*cert_chain*, **`const`** std::vector<std::shared_ptr<**`const`** OCSP::*Response*>> &*ocsp_responses*, **`const`** std::vector<Certificate_Store*> &*trusted_roots*, Usage_Type *usage*, **`const`** std::string &*hostname*, **`const`** Policy &*policy*)

Optional - default implementation should work for many users. It can be overridden for implementing extra validation routines such as public key pinning.

Verifies the certificate chain in *cert_chain*, assuming the leaf certificate is the first element. Throws an exception if any error makes this certificate chain unacceptable.

If usage is *Usage_Type::TLS_SERVER_AUTH*, then *hostname* should match the information in the server certificate. If usage is *TLS_CLIENT_AUTH*, then *hostname* specifies the host the client is authenticating against (from SNI); the callback can use this for any special site specific auth logic.

The *ocsp_responses* is a possibly empty list of OCSP responses provided by the server. In the current implementation of TLS OCSP stapling, only a single OCSP response can be returned. A existing TLS extension allows the server to send multiple OCSP responses, this extension may be supported in the future in which case more than one OCSP response may be given during this callback.

The *trusted_roots* parameter was returned by a call from the associated *Credentials_Manager*.

The *policy* provided is the policy for the TLS session which is being authenticated using this certificate chain. It can be consulted for values such as allowable signature methods and key sizes.

std::chrono::milliseconds **`tls_verify_cert_chain_ocsp_timeout`**() **`const`**

Called by default *tls_verify_cert_chain* to set timeout for online OCSP requests on the certificate chain. Return 0 to disable OCSP. Current default is 0.

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_session_activated`**()

Optional. By default does nothing. This is called when the session is activated, that is once it is possible to send or receive data on the channel. In particular it is possible for an implementation of this function to perform an initial write on the channel.

std::vector<uint8_t> **`tls_provide_cert_status`**(**`const`** std::vector<*X509_Certificate*> &*chain*, **`const`** Certificate_Status_Request &*csr*)

Optional. This can return a cached OCSP response. This is only used on the server side, and only if the client requests OCSP stapling.

std::string **tls_peer_network_identity**()

Optional. Return a string that identifies the peer in some unique way (for example, by formatting the remote IP and port into a string). This is currently used to bind DTLS cookies to the network identity.

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_modify_extensions**(*Extensions* *&extn*, Connection_Side *which_side*)

This callback is optional, and can be used to modify extensions before they are sent to the peer. For example this enables adding a custom extension, or replacing or removing an extension set by the library.

void **tls_examine_extensions**(**const** *Extensions* *&extn*, Connection_Side *which_side*)

This callback is optional, and can be used to examine extensions sent by the peer.

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)

std::string **tls_decode_group_param**(TLS::Group_Params *group_param*)

Optional. Called by the server when a client hello includes a list of supported groups in the supported_groups extension and by the client when decoding the server key exchange including the selected curve identifier. The function should return the name of the DH group or elliptic curve the passed TLS group identifier should be mapped to. Therefore this callback enables the use of custom elliptic curves or DH groups in TLS, if both client and server map the custom identifiers correctly. Please note that it is required to allow the group TLS identifier in in the used `TLS::Policy`.

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.

5.11.1 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 connection 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.

5.11.2 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* = std::vector<std::string>(), 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.

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

(continues on next page)

(continued from previous page)

```
#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:
      Client_Credentials()
         {
         // Here we base trust on the system managed trusted CA list
         m_stores.push_back(new Botan::System_Certificate_Store);
         }

      std::vector<Botan::Certificate_Store*> trusted_certificate_authorities(
         const std::string& type,
         const std::string& context) override
         {
```

(continues on next page)

(continued from previous page)

```
         // return a list of certificates of CAs we trust for tls server certificates
         // ownership of the pointers remains with Credentials_Manager
         return m_stores;
         }

      std::vector<Botan::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;
         }

   private:
       std::vector<Botan::Certificate_Store*> m_stores;
};

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

5.11.3 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 similarly 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.

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
         }
```

(continues on next page)

(continued from previous page)

```
      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 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::PKCS8::load_key("botan.randombit.net.key"))
         {
         }

      std::vector<Botan::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<Botan::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;
         }
```

(continues on next page)

(continued from previous page)

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

int main()
   {
   // prepare all the parameters
   Callbacks callbacks;
   Botan::AutoSeeded_RNG rng;
   Botan::TLS::Session_Manager_In_Memory session_mgr(rng);
   Server_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()
   // ...
   }
```

5.11.4 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:

Note: The serialization format of Session is not considered stable and is allowed to change even across minor releases. In the event of such a change, old sessions will no longer be able to be resumed.

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. The format is described in *TLS Session Encryption*.

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.

5.11.5 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.

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*.

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.

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*.

5.11.6 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"

Also allowed: "AES-256", "AES-128", "AES-256/CCM", "AES-128/CCM", "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)"

In versions up to 2.8.0, the CBC and CCM ciphersuites "AES-256", "AES-128", "AES-256/CCM" and "AES-128/CCM" were enabled by default.

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 overhead reasons, DH should also be avoided.

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.

Note: In order to enable RSA, SRP, or PSK ciphersuites one must also enable authentication method “IMPLICIT”, see `allowed_signature_methods`.

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”, “IMPLICIT”, “ANONYMOUS”

“IMPLICIT” enables ciphersuites which are authenticated not by a signature but through a side-effect of the key exchange. In particular this setting is required to enable PSK, SRP, and static RSA ciphersuites.

“ANONYMOUS” allows purely anonymous DH/ECDH key exchanges. **Enabling this is not recommended**

Note: Both DSA authentication and anonymous DH ciphersuites are deprecated, and will be removed in a future release.

std::vector<Group_Params> **`key_exchange_groups`**`()` **`const`**

Return a list of ECC curve and DH group TLS identifiers we are willing to use, in order of preference. The default ordering puts the best performing ECC first.

Default: Group_Params::X25519, Group_Params::SECP256R1, Group_Params::BRAINPOOL256R1, Group_Params::SECP384R1, Group_Params::BRAINPOOL384R1, Group_Params::SECP521R1, Group_Params::BRAINPOOL512R1, Group_Params::FFDHE_2048, Group_Params::FFDHE_3072, Group_Params::FFDHE_4096, Group_Params::FFDHE_6144, Group_Params::FFDHE_8192

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

Note: Support for EC point compression is deprecated and will be removed in a future major release.

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 v1.2 and DTLS v1.2, and rejects all older versions.

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.

Default: true

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: false

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 **`abort_connection_on_undesired_renegotiation`**`()` **`const`**

If a renegotiation attempt is being rejected due to the configuration of *`TLS::Policy::allow_client_initiated_renegotiation`* or *`TLS::Policy::allow_server_initiated_renegotiation`*, and this function returns true then the connection is closed with a fatal alert instead of the default warning 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

Group_Params **default_dh_group**() **const**

For ephemeral Diffie-Hellman key exchange, the server sends a group parameter. Return the 2 Byte TLS group identifier 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

bool **allow_tls10**() **const**

Return true from here to allow TLS v1.0. Since 2.8.0, returns `false` by default.

bool **allow_tls11**() **const**

Return true from here to allow TLS v1.1. Since 2.8.0, returns `false` by default.

bool **allow_tls12**() **const**

Return true from here to allow TLS v1.2. Returns `true` by default.

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)

5.11.7 TLS Ciphersuites

class `TLS::`**`Ciphersuite`**

uint16_t **`ciphersuite_code`**`()` **`const`**
: Return the numerical code for this ciphersuite

std::string **`to_string`**`()` **`const`**
: Return the full 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

bool **`acceptable_ciphersuite`**`(`**`const`** *Ciphersuite* &*suite*`)` **`const`**
: Return true if ciphersuite is accepted by the policy.

Allows an application to reject any ciphersuites, which are undesirable for whatever reason without having to reimplement *`TLS::Ciphersuite::ciphersuite_list`*

std::vector<uint16_t> **`ciphersuite_list`**`(`*Protocol_Version* *version*, bool *have_srp*`)` **`const`**
: Return allowed ciphersuites in order of preference

Allows an application to have full control over ciphersuites by returning desired ciphersuites in preference order.

5.11.8 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

5.11.9 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)

5.11.10 TLS Custom Curves

The supported_groups TLS extension is used in the client hello to advertise a list of supported elliptic curves and DH groups. The server subsequently selects one of the groups, which is supported by both endpoints. The groups are represented by their TLS identifier. This 2 Byte identifier is standardized for commonly used groups and curves. In addition, the standard reserves the identifiers 0xFE00 to 0xFEFF for custom groups or curves.

Using non standardized custom curves is however not recommended and can be a serious risk if an insecure curve is used. Still, it might be desired in some scenarios to use custom curves or groups in the TLS handshake.

To use custom curves with the Botan *`TLS::Client`* or *`TLS::Server`* the following additional adjustments have to be implemented as shown in the following code examples.

1. Registration of the custom curve
2. Implementation TLS callback `tls_decode_group_param`
3. Adjustment of the TLS policy by allowing the custom curve

Client Code Example

```
#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/ec_group.h>
```

(continues on next page)

(continued from previous page)

```
#include <botan/oids.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;
         }
      std::string tls_decode_group_param(Botan::TLS::Group_Params group_param)␣
↪override
          {
          // handle TLS group identifier decoding and return name as string
          // return empty string to indicate decoding failure

          switch(static_cast<uint16_t>(group_param))
             {
             case 0xFE00:
                return "testcurve1102";
             default:
                //decode non-custom groups
                return Botan::TLS::Callbacks::tls_decode_group_param(group_param);
             }
          }
};

/**
 * @brief Credentials storage for the tls client.
 *
```

(continues on next page)

(continued from previous page)

```
 * 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<Botan::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") };
         }

      std::vector<Botan::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;
         }
};

class Client_Policy : public Botan::TLS::Strict_Policy
{
   public:
      std::vector<Botan::TLS::Group_Params> key_exchange_groups() const override
         {
         // modified strict policy to allow our custom curves
         return
            {
            static_cast<Botan::TLS::Group_Params>(0xFE00)
            };
         }
};

int main()
   {
   // prepare rng
   Botan::AutoSeeded_RNG rng;

   // prepare custom curve

   // prepare curve parameters
```

(continues on next page)

(continued from previous page)

```
   const Botan::BigInt p(
↪"0x92309a3e88b94312f36891a2055725bb35ab51af96b3a651d39321b7bbb8c51575a76768c9b6b323
↪");
   const Botan::BigInt a(
↪"0x4f30b8e311f6b2dce62078d70b35dacb96aa84b758ab5a8dff0c9f7a2a1ff466c19988aa0acdde69
↪");
   const Botan::BigInt b(
↪"0x9045A513CFFF9AE1F1CC84039D852D240344A1D5C9DB203C844089F855C387823EB6FCDDF49C909C
↪");

   const Botan::BigInt x(
↪"0x9120f3779a31296cefcb5a5a08831f1a6d438ad5a3f2ce60585ac19c74eebdc65cadb96bb92622c7
↪");
   const Botan::BigInt y(
↪"0x836db8251c152dfee071b72c6b06c5387d82f1b5c30c5a5b65ee9429aa2687e8426d5d61276a4ede
↪");
   const Botan::BigInt order(
↪"0x248c268fa22e50c4bcda24688155c96ecd6ad46be5c82d7a6be6e7068cb5d1ca72b2e07e8b90d853
↪");

   const Botan::BigInt cofactor(4);

   const Botan::OID oid("1.2.3.1");

   // create EC_Group object to register the curve
   Botan::EC_Group testcurve1102(p, a, b, x, y, order, cofactor, oid);

   if(!testcurve1102.verify_group(rng))
      {
      // Warning: if verify_group returns false the curve parameters are insecure
      }

   // register name to specified oid
   Botan::OIDS::add_oid(oid, "testcurve1102");

   // prepare all the parameters
   Callbacks callbacks;
   Botan::TLS::Session_Manager_In_Memory session_mgr(rng);
   Client_Credentials creds;
   Client_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
      // ...
```

(continues on next page)

(continued from previous page)

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

       }
   }
```

Server Code Example

```
#include <botan/tls_server.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 <botan/pkcs8.h>

#include <botan/ec_group.h>
#include <botan/oids.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
         }

      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;
         }
```

(continues on next page)

(continued from previous page)

```
      std::string tls_decode_group_param(Botan::TLS::Group_Params group_param)␣
↪override
         {
         // handle TLS group identifier decoding and return name as string
         // return empty string to indicate decoding failure

         switch(static_cast<uint16_t>(group_param))
            {
            case 0xFE00:
               return "testcurve1102";
            default:
               //decode non-custom groups
               return Botan::TLS::Callbacks::tls_decode_group_param(group_param);
            }
         }
};

/**
 * @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::PKCS8::load_key("botan.randombit.net.key")
         {
         }

      std::vector<Botan::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<Botan::Certificate_Store*>();
         }

      std::vector<Botan::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
         {
```

(continues on next page)

(continued from previous page)

```
         // return the private key associated with the leaf certificate,
         // in this case the one associated with "botan.randombit.net.crt"
         return m_key.get();
         }

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

class Server_Policy : public Botan::TLS::Strict_Policy
{
   public:
      std::vector<Botan::TLS::Group_Params> key_exchange_groups() const override
         {
         // modified strict policy to allow our custom curves
         return
            {
            static_cast<Botan::TLS::Group_Params>(0xFE00)
            };
         }
};

int main()
   {

   // prepare rng
   Botan::AutoSeeded_RNG rng;

   // prepare custom curve

   // prepare curve parameters
   const Botan::BigInt p(
↪"0x92309a3e88b94312f36891a2055725bb35ab51af96b3a651d39321b7bbb8c51575a76768c9b6b323
↪");
   const Botan::BigInt a(
↪"0x4f30b8e311f6b2dce62078d70b35dacb96aa84b758ab5a8dff0c9f7a2a1ff466c19988aa0acdde69
↪");
   const Botan::BigInt b(
↪"0x9045A513CFFF9AE1F1CC84039D852D240344A1D5C9DB203C844089F855C387823EB6FCDDF49C909C
↪");

   const Botan::BigInt x(
↪"0x9120f3779a31296cefcb5a5a08831f1a6d438ad5a3f2ce60585ac19c74eebdc65cadb96bb92622c7
↪");
   const Botan::BigInt y(
↪"0x836db8251c152dfee071b72c6b06c5387d82f1b5c30c5a5b65ee9429aa2687e8426d5d61276a4ede
↪");
   const Botan::BigInt order(
↪"0x248c268fa22e50c4bcda24688155c96ecd6ad46be5c82d7a6be6e7068cb5d1ca72b2e07e8b90d853
↪");

   const Botan::BigInt cofactor(4);

   const Botan::OID oid("1.2.3.1");

   // create EC_Group object to register the curve
   Botan::EC_Group testcurve1102(p, a, b, x, y, order, cofactor, oid);
```

(continues on next page)

(continued from previous page)

```
if(!testcurve1102.verify_group(rng))
   {
   // Warning: if verify_group returns false the curve parameters are insecure
   }

// register name to specified oid
Botan::OIDS::add_oid(oid, "testcurve1102");

// prepare all the parameters
Callbacks callbacks;
Botan::TLS::Session_Manager_In_Memory session_mgr(rng);
Server_Credentials creds;
Server_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()
// ...
}
```

5.11.11 TLS Stream

`TLS::Stream` offers a Boost.Asio compatible wrapper around `TLS::Client` and `TLS::Server`. It can be used as an alternative to Boost.Asio's ssl::stream (https://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio/reference/ssl__stream.html) with minor adjustments to the using code. It offers the following interface:

template<class **StreamLayer**, class **ChannelT**>
class TLS::**Stream**

StreamLayer specifies the type of the stream's *next layer*, for example a Boost.Asio TCP socket (https://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio/reference/ip__tcp/socket.html). *ChannelT* is the type of the stream's *native handle*; it defaults to `TLS::Channel` and should not be specified manually.

template<typename ...**Args**>
explicit Stream(*Context* &*context*, *Args*&&... *args*)

Construct a new TLS stream. The *context* parameter will be used to initialize the underlying *native handle*, i.e. the *TLS::Client* or *TLS::Server*, when `handshake` is called. Using code must ensure the context is kept alive for the lifetime of the stream. The further *args* will be forwarded to the *next layer*'s constructor.

template<typename ...**Args**>
explicit Stream(Arg &&*arg*, *Context* &*context*)

Convenience constructor for `boost::asio::ssl::stream` compatibility. The parameters have the same meaning as for the first constructor, but their order is changed and only one argument can be passed to the *next layer* constructor.

void **handshake** (Connection_Side *side*, boost::system::error_code *&ec*)

Set up the *native handle* and perform the TLS handshake.

void **handshake** (Connection_Side *side*)

Overload of `handshake` that throws an exception if an error occurs.

template<typename **HandshakeHandler**>
DEDUCED **async_handshake** (Connection_Side *side*, *HandshakeHandler* *&&handler*)

Asynchronous variant of `handshake`. The function returns immediately and calls the *handler* callback function after performing asynchronous I/O to complete the TLS handshake. The return type is an automatically deduced specialization of `boost::asio::async_result`, depending on the *HandshakeHandler* type.

void **shutdown** (boost::system::error_code *&ec*)

Calls `TLS::Channel::close` on the native handle and writes the TLS alert to the *next layer*.

void **shutdown** ()

Overload of `shutdown` that throws an exception if an error occurs.

template<typename **ShutdownHandler**>
void **async_shutdown** (*ShutdownHandler* *&&handler*)

Asynchronous variant of `shutdown`. The function returns immediately and calls the *handler* callback function after performing asynchronous I/O to complete the TLS shutdown.

template<typename **MutableBufferSequence**>
std::size_t **read_some** (**const** *MutableBufferSequence* *&buffers*, boost::system::error_code *&ec*)

Reads encrypted data from the *next layer*, decrypts it, and writes it into the provided *buffers*. If an error occurs, *error_code* is set. Returns the number of bytes read.

template<typename **MutableBufferSequence**>
std::size_t **read_some** (**const** *MutableBufferSequence* *&buffers*)

Overload of `read_some` that throws an exception if an error occurs.

template<typename **MutableBufferSequence**, typename **ReadHandler**>
DEDUCED **async_read_some** (**const** *MutableBufferSequence* *&buffers*, *ReadHandler* *&&handler*)

Asynchronous variant of `read_some`. The function returns immediately and calls the *handler* callback function after writing the decrypted data into the provided *buffers*. The return type is an automatically deduced specialization of `boost::asio::async_result`, depending on the *ReadHandler* type. *ReadHandler* should suffice the requirements to a Boost.Asio read handler (https://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio/reference/ReadHandler.html).

template<typename **ConstBufferSequence**>
std::size_t **write_some** (**const** *ConstBufferSequence* *&buffers*, boost::system::error_code *&ec*)

Encrypts data from the provided *buffers* and writes it to the *next layer*. If an error occurs, *error_code* is set. Returns the number of bytes written.

template<typename **ConstBufferSequence**>
std::size_t **write_some** (**const** *ConstBufferSequence* *&buffers*)

Overload of `write_some` that throws an exception rather than setting an error code.

template<typename **ConstBufferSequence**, typename **WriteHandler**>
DEDUCED **async_write_some** (**const** *ConstBufferSequence* *&buffers*, *WriteHandler* *&&handler*)

Asynchronous variant of `write_some`. The function returns immediately and calls the *handler* callback function after writing the encrypted data to the *next layer*. The return type is an automatically deduced specialization of `boost::asio::async_result`, depending on the *WriteHandler* type. *WriteHandler* should suffice the requirements to a Boost.Asio write handler (https://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio/reference/WriteHandler.html).

class TLS::**Context**

A helper class to initialize and configure the Stream's underlying *native handle* (see `TLS::Client` and `TLS::Server`).

Context (*Credentials_Manager &credentialsManager*, *RandomNumberGenerator &randomNumberGenerator*, Session_Manager *&sessionManager*, *Policy &policy*, Server_Information *serverInfo* = Server_Information())

Constructor for TLS::Context.

void **set_verify_callback** (Verify_Callback_T *callback*)

Set a user-defined callback function for certificate chain verification. This will cause the stream to override the default implementation of the `tls_verify_cert_chain` callback.

TLS Stream Client Code Example

The code below illustrates how to build a simple HTTPS client based on the TLS Stream and Boost.Beast. When run, it fetches the content of *https://botan.randombit.net/news.html* and prints it to stdout.

```
#include <iostream>

#include <botan/asio_stream.h>
#include <botan/auto_rng.h>
#include <botan/certstor_system.h>

#include <boost/asio.hpp>
#include <boost/beast.hpp>
#include <boost/bind.hpp>

namespace http = boost::beast::http;
namespace _ = boost::asio::placeholders;

// very basic credentials manager
class Credentials_Manager : public Botan::Credentials_Manager
   {
   public:
      Credentials_Manager() {}

      std::vector<Botan::Certificate_Store*>
      trusted_certificate_authorities(const std::string&, const std::string&) override
         {
         return {&cert_store_};
         }

   private:
      Botan::System_Certificate_Store cert_store_;
   };

// a simple https client based on TLS::Stream
class client
   {
```

(continues on next page)

(continued from previous page)

```
  public:
     client(boost::asio::io_context&                  io_context,
            boost::asio::ip::tcp::resolver::iterator endpoint_iterator,
            http::request<http::string_body>         req)
        : request_(req)
        , ctx_(credentials_mgr_,
               rng_,
               session_mgr_,
               policy_,
               Botan::TLS::Server_Information())
        , stream_(io_context, ctx_)
        {
        boost::asio::async_connect(stream_.lowest_layer(), endpoint_iterator,
                                   boost::bind(&client::handle_connect, this, _
↪::error));
        }

     void handle_connect(const boost::system::error_code& error)
        {
        if(error)
           {
           std::cout << "Connect failed: " << error.message() << "\n";
           return;
           }
        stream_.async_handshake(Botan::TLS::Connection_Side::CLIENT,
                                boost::bind(&client::handle_handshake, this, _
↪::error));
        }

     void handle_handshake(const boost::system::error_code& error)
        {
        if(error)
           {
           std::cout << "Handshake failed: " << error.message() << "\n";
           return;
           }
        http::async_write(stream_, request_,
                          boost::bind(&client::handle_write, this, _::error, _
↪::bytes_transferred));
        }

     void handle_write(const boost::system::error_code& error, size_t)
        {
        if(error)
           {
           std::cout << "Write failed: " << error.message() << "\n";
           return;
           }
        http::async_read(stream_, reply_, response_,
                         boost::bind(&client::handle_read, this, _::error, _::bytes_
↪transferred));
        }

     void handle_read(const boost::system::error_code& error, size_t)
        {
        if(!error)
           {
```

(continues on next page)

(continued from previous page)

```
            std::cout << "Reply: ";
            std::cout << response_.body() << "\n";
            }
         else
            {
            std::cout << "Read failed: " << error.message() << "\n";
            }
         }

   private:
      http::request<http::dynamic_body> request_;
      http::response<http::string_body> response_;
      boost::beast::flat_buffer         reply_;

      Botan::TLS::Session_Manager_Noop session_mgr_;
      Botan::AutoSeeded_RNG            rng_;
      Credentials_Manager              credentials_mgr_;
      Botan::TLS::Policy               policy_;

      Botan::TLS::Context                              ctx_;
      Botan::TLS::Stream<boost::asio::ip::tcp::socket> stream_;
   };

int main()
   {
   boost::asio::io_context io_context;

   boost::asio::ip::tcp::resolver           resolver(io_context);
   boost::asio::ip::tcp::resolver::query    query("botan.randombit.net", "443");
   boost::asio::ip::tcp::resolver::iterator iterator = resolver.resolve(query);

   http::request<http::string_body> req;
   req.version(11);
   req.method(http::verb::get);
   req.target("/news.html");
   req.set(http::field::host, "botan.randombit.net");

   client c(io_context, iterator, req);

   io_context.run();
   }
```

5.11.12 TLS Session Encryption

A unified format is used for encrypting TLS sessions either for durable storage (on client or server) or when creating TLS session tickets. This format is *not stable* even across the same major version.

The current session encryption scheme was introduced in 2.13.0, replacing the format previously used since 1.11.13.

Session encryption accepts a key of any length, though for best security a key of 256 bits should be used. This master key is used to key an instance of HMAC using the SHA-512/256 hash.

First a "key name" or identifier is created, by HMAC'ing the fixed string "BOTAN TLS SESSION KEY NAME" and truncating to 4 bytes. This is the initial prefix of the encrypted session, and will remain fixed as long as the same ticket key is used. This allows quickly rejecting sessions which are encrypted using an unknown or incorrect key.

Then a key used for AES-256 in GCM mode is created by first choosing a 128 bit random seed, and HMAC'ing it to produce a 256-bit value. This means for any one master key as many as 2^{128} GCM keys can be created. This is done because NIST recommends that when using random nonces no one GCM key be used to encrypt more than 2^{32} messages (to avoid the possiblity of nonce reuse).

A random 96-bit nonce is created and included in the header.

AES in GCM is used to encrypt and authenticate the serialized session. The key name, key seed, and AEAD nonce are all included as additional data.

5.12 Credentials Manager

A `Credentials_Manager` is a way to abstract how the application stores credentials. The main user is the *Transport Layer Security (TLS)* implementation.

`class Credentials_Manager`

std::vector<Certificate_Store*> **`trusted_certificate_authorities`**(**`const`** std::string &*type*, **`const`** std::string &*context*)

Return the list of certificate stores, each of which is assumed to contain (only) trusted certificate authorities. The `Credentials_Manager` retains ownership of the Certificate_Store pointers.

Note: It would have been a better API to return a vector of `shared_ptr` here. This may change in a future major release.

When *type* is “tls-client”, *context* will be the hostname of the server, or empty if the hostname is not known. This allows using a different set of certificate stores in different contexts, for example using the system certificate store unless contacting one particular server which uses a cert issued by an internal CA.

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*> **`find_cert_chain`**(**`const`** std::vector<std::string> &*cert_key_types*, **`const`** std::vector<*X509_DN*> &*acceptable_CAs*, **`const`** std::string &*type*, **`const`** std::string &*context*)

Return the certificate chain to use to identify ourselves. The `acceptable_CAs` parameter gives a list of CAs the peer trusts. This may be empty.

Warning: If this function returns a certificate that is not one of the types given in `cert_key_types` confusing handshake failures will result.

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. Starting in 2.5, prefer `find_cert_chain` which additionally provides the CA list.

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_type* 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*.

5.12.1 SRP Authentication

`Credentials_Manager` contains the hooks used by TLS clients and servers for SRP authentication.

Note: Support for TLS-SRP is deprecated, and will be removed in a future major release. When that occurs these APIs will be removed. Prefer instead performing a standard TLS handshake, then perform a PAKE authentication inside of (and cryptographically bound to) the TLS channel.

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)

5.12.2 Preshared Keys

TLS supports 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.

A similar special case exists for DTLS cookie verification. In this case *type* will be "tls-server" and *context* is "dtls-cookie-secret". If no key is returned, then DTLS cookies are not used. Similar to the session ticket key, the DTLS cookie secret can be chosen during server startup and rotated at any time with no ill effect.

Warning: If DTLS cookies are not used then the server is prone to be abused as a DoS amplifier, where the attacker sends a relatively small client hello in a UDP packet with a forged return address, and then the server replies to the victim with several messages that are larger. This not only hides the attackers address from the victim, but increases their effective bandwidth. This is not an issue when using DTLS over SCTP or TCP.

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.

5.13 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, and not all of them are documented here. You can find the complete declarations in `botan/bigint.h` and `botan/numthry.h`.

`class BigInt`

`BigInt` ()
Create a BigInt with value zero

`BigInt` (uint64_t *n*)
Create a BigInt with value *n*

`BigInt` (**`const`** std::string &*str*)
Create a BigInt from a string. By default decimal is expected. With an 0x prefix instead it is treated as hexadecimal.

`BigInt` (**`const`** uint8_t *buf*[], size_t *length*)
Create a BigInt from a binary array (big-endian encoding).

`BigInt` (*RandomNumberGenerator* &*rng*, size_t *bits*, bool *set_high_bit* = true)
Create a random BigInt of the specified size.

BigInt **`operator+`** (**`const`** *BigInt* &*x*, **`const`** *BigInt* &*y*)
Add `x` and `y` and return result.

BigInt **`operator+`** (**`const`** *BigInt* &*x*, word *y*)
Add `x` and `y` and return result.

BigInt **`operator+`** (word *x*, **`const`** *BigInt* &*y*)
Add `x` and `y` and return result.

BigInt **`operator-`** (**`const`** *BigInt* &*x*, **`const`** *BigInt* &*y*)
Subtract `y` from `x` and return result.

BigInt **`operator-`** (**`const`** *BigInt* &*x*, word *y*)
Subtract `y` from `x` and return result.

BigInt **`operator*`** (**`const`** *BigInt* &*x*, **`const`** *BigInt* &*y*)
Multiply `x` and `y` and return result.

BigInt **operator/**(**const** *BigInt* &*x*, **const** *BigInt* &*y*)
Divide `x` by `y` and return result.

BigInt **operator%**(**const** *BigInt* &*x*, **const** *BigInt* &*y*)
Divide `x` by `y` and return remainder.

word **operator%**(**const** *BigInt* &*x*, word *y*)
Divide `x` by `y` and return remainder.

word **operator<<**(**const** *BigInt* &*x*, size_t *n*)
Left shift `x` by `n` and return result.

word **operator>>**(**const** *BigInt* &*x*, size_t *n*)
Right shift `x` by `n` and return result.

BigInt &**operator+=**(**const** *BigInt* &*y*)
Add y to `*this`

BigInt &**operator+=**(word *y*)
Add y to `*this`

BigInt &**operator-=**(**const** *BigInt* &*y*)
Subtract y from `*this`

BigInt &**operator-=**(word *y*)
Subtract y from `*this`

BigInt &**operator*=**(**const** *BigInt* &*y*)
Multiply `*this` with y

BigInt &**operator*=**(word *y*)
Multiply `*this` with y

BigInt &**operator/=**(**const** *BigInt* &*y*)
Divide `*this` by y

BigInt &**operator%=**(**const** *BigInt* &*y*)
Divide `*this` by y and set `*this` to the remainder.

word **operator%=**(word *y*)
Divide `*this` by y and set `*this` to the remainder.

word **operator<<=**(size_t *shift*)
Left shift `*this` by *shift* bits

word **operator>>=**(size_t *shift*)
Right shift `*this` by *shift* bits

BigInt &**operator++**()
Increment `*this` by 1

BigInt &**operator--**()
Decrement `*this` by 1

BigInt **operator++**(int)
Postfix increment `*this` by 1

BigInt **operator--**(int)
Postfix decrement `*this` by 1

BigInt **operator-**() **const**
Negation operator

bool **operator!**() **const**
: Return true unless `*this` is zero

void **clear**()
: Set `*this` to zero

size_t **bytes**() **const**
: Return number of bytes need to represent value of `*this`

size_t **bits**() **const**
: Return number of bits need to represent value of `*this`

bool **is_even**() **const**
: Return true if `*this` is even

bool **is_odd**() **const**
: Return true if `*this` is odd

bool **is_nonzero**() **const**
: Return true if `*this` is not zero

bool **is_zero**() **const**
: Return true if `*this` is zero

void **set_bit**(size_t *n*)
: Set bit *n* of `*this`

void **clear_bit**(size_t *n*)
: Clear bit *n* of `*this`

bool **get_bit**(size_t *n*) **const**
: Get bit *n* of `*this`

uint32_t **to_u32bit**() **const**
: Return value of `*this` as a 32-bit integer, if possible. If the integer is negative or not in range, an exception is thrown.

bool **is_negative**() **const**
: Return true if `*this` is negative

bool **is_positive**() **const**
: Return true if `*this` is negative

BigInt **abs**() **const**
: Return absolute value of `*this`

void **binary_encode**(uint8_t *buf*[]) **const**
: Encode this BigInt as a big-endian integer. The sign is ignored.

void **binary_encode**(uint8_t *buf*[], size_t *len*) **const**
: Encode this BigInt as a big-endian integer. The sign is ignored. If `len` is less than `bytes()` then only the low `len` bytes are output. If `len` is greater than `bytes()` then the output is padded with leading zeros.

void **binary_decode**(uint8_t *buf*[])
: Decode this BigInt as a big-endian integer.

std::string **to_dec_string**() **const**
: Encode the integer as a decimal string.

std::string **to_hex_string**() **const**
: Encode the integer as a hexadecimal string.

5.13.1 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 **jacobi**(*BigInt a*, *BigInt n*)
Return Jacobi symbol of (a|n).

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 probabilistic 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.

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`.

5.14 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 label or identifier, such as a session id.

You can create a *KDF* using

KDF ***get_kdf**(**const** std::string &*algo_spec*)

5.14.1 Available KDFs

Botan includes many different KDFs simply because different protocols and standards have created subtly different approaches to this problem. For new code, use HKDF which is conservative, well studied, widely implemented and NIST approved.

HKDF

Defined in RFC 5869, HKDF uses HMAC to process inputs. Also available are variants HKDF-Extract and HKDF-Expand. HKDF is the combined Extract+Expand operation. Use the combined HKDF unless you need compatibility with some other system.

Available if `BOTAN_HAS_HKDF` is defined.

KDF2

KDF2 comes from IEEE 1363. It uses a hash function.

Available if `BOTAN_HAS_KDF2` is defined.

KDF1-18033

KDF1 from ISO 18033-2. Very similar to (but incompatible with) KDF2.

Available if `BOTAN_HAS_KDF1_18033` is defined.

KDF1

KDF1 from IEEE 1363. It can only produce an output at most the length of the hash function used.

Available if `BOTAN_HAS_KDF1` is defined.

X9.42 PRF

A KDF from ANSI X9.42. Sometimes used for Diffie-Hellman.

Available if `BOTAN_HAS_X942_PRF` is defined.

Warning: Support for X9.42 KDF is deprecated and will be removed in a future major release.

SP800-108

KDFs from NIST SP 800-108. Variants include "SP800-108-Counter", "SP800-108-Feedback" and "SP800-108-Pipeline".

Available if `BOTAN_HAS_SP800_108` is defined.

SP800-56A

KDF from NIST SP 800-56A.

Available if `BOTAN_HAS_SP800_56A` is defined.

SP800-56C

KDF from NIST SP 800-56C.

Available if `BOTAN_HAS_SP800_56C` is defined.

5.15 Password Based Key Derivation

Often one needs to convert a human readable password into a cryptographic key. It is useful to slow down the computation of these computations in order to reduce the speed of brute force search, thus they are parameterized in some way which allows their required computation to be tuned.

5.15.1 PBKDF

PBKDF is the older API for this functionality, presented in header `pbkdf.h`. It does not support Scrypt, nor will it be able to support other future hashes (such as Argon2) that may be added in the future. In addition, this API requires the passphrase be entered as a `std::string`, which means the secret will be stored in memory that will not be zeroed.

class PBKDF

void **pbkdf_iterations**(uint8_t *out*[], size_t *out_len*, **const** std::string &*passphrase*, **const** uint8_t *salt*[], size_t *salt_len*, size_t *iterations*) **const**

Run the PBKDF algorithm for the specified number of iterations, with the given salt, and write output to the buffer.

void **pbkdf_timed**(uint8_t *out*[], size_t *out_len*, **const** std::string &*passphrase*, **const** uint8_t *salt*[], size_t *salt_len*, std::chrono::milliseconds *msec*, size_t &*iterations*) **const**

Choose (via short run-time benchmark) how many iterations to perform in order to run for roughly msec milliseconds. Writes the number of iterations used to reference argument.

OctetString **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.

5.15.2 PasswordHash

New in version 2.8.0.

This API has two classes, one representing the algorithm (such as "PBKDF2(SHA-256)", or "Scrypt") and the other representing a specific instance of the problem which is fully specified (say "Scrypt" with N=8192,r=64,p=8).

class PasswordHash

void **derive_key**(uint8_t *out*[], size_t *out_len*, **const** char **password*, **const** size_t *password_len*, **const** uint8_t *salt*[], size_t *salt_len*) **const**

Derive a key, placing it into output

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

Return a descriptive string including the parameters (iteration count, etc)

The `PasswordHashFamily` creates specific instances of `PasswordHash`:

class PasswordHashFamily

static std::unique_ptr<*PasswordHashFamily*> **create**(**const** std::string &*what*)

For example "PBKDF2(SHA-256)", "Scrypt", "OpenPGP-S2K(SHA-384)". Returns null if not available.

std::unique_ptr<*PasswordHash*> **default_params**() **const**

Create a default instance of the password hashing algorithm. Be warned the value returned here may change from release to release.

std::unique_ptr<*PasswordHash*> **tune**(size_t *output_len*, std::chrono::milliseconds *msec*) **const**

Return a password hash instance tuned to run for approximately `msec` milliseconds when producing an output of length `output_len`. (Accuracy may vary, use the command line utility `botan pbkdf_tune` to check.)

std::unique_ptr<*PasswordHash*> **from_params**(size_t *i1*, size_t *i2* = 0, size_t *i3* = 0) **const**

Create a password hash using some scheme specific format. Eg PBKDF2 and PGP-S2K set iterations in i1 Scrypt uses N,r,p in i{1-3} Bcrypt-PBKDF just has iterations Argon2{i,d,id} would use iterations, memory, parallelism for i{1-3}, and Argon2 type is part of the family.

Values not needed should be set to 0.

5.15.3 Available Schemes

PBKDF2

PBKDF2 is the "standard" password derivation scheme, widely implemented in many different libraries. It uses HMAC internally.

Scrypt

Scrypt is a relatively newer design which is "memory hard" - in addition to requiring large amounts of CPU power it uses a large block of memory to compute the hash. This makes brute force attacks using ASICs substantially more expensive.

Scrypt is not supported through `PBKDF`, only `PasswordHash`, starting in 2.8.0. In addition, starting in version 2.7.0, scrypt is available with this function:

void **scrypt** (uint8_t *output*[], size_t *output_len*, **const** std::string &*password*, **const** uint8_t *salt*[], size_t *salt_len*, size_t *N*, size_t *r*, size_t *p*)

Computes the Scrypt using the password and salt, and produces an output of arbitrary length.

The N, r, p parameters control how much work and memory Scrypt uses. N is the primary control of the workfactor, and must be a power of 2. For interactive logins use 32768, for protection of secret keys or backups use 1048576.

The r parameter controls how 'wide' the internal hashing operation is. It also increases the amount of memory that is used. Values from 1 to 8 are reasonable.

Setting p parameter to greater than one splits up the work in a way that up to p processors can work in parallel.

As a general recommendation, use N=32768, r=8, p=1

Argon2

New in version 2.11.0.

Argon2 is the winner of the PHC (Password Hashing Competition) and provides a tunable memory hard PBKDF.

OpenPGP S2K

Warning: The OpenPGP algorithm is weak and strange, and should be avoided unless implementing OpenPGP.

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.

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. It is not supported through `PasswordHash`.

5.16 AES Key Wrapping

NIST specifies two mechanisms for wrapping (encrypting) symmetric keys using another key. The first (and older, more widely supported) method requires 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 5649.

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 Invalid_Authentication_Tag 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 Invalid_Authentication_Tag on error.

5.16.1 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 with rfc3394_keywrap.

5.17 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 authenticating party 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 effectiveness 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 three techniques for password hashing: Argon2, bcrypt, and passhash9 (based on PBKDF2).

5.17.1 Argon2

New in version 2.11.0.

Argon2 is the winner of the PHC (Password Hashing Competition) and provides a tunable memory hard password hash. It has a standard string encoding, which looks like:

```
"$argon2i$v=19$m=8192,t=10,p=3$YWFhYWFhYWE$itkWB9ODqTd85wUsoib7pfpVTNGMOu0ZJan1odl25V8
↪"
```

Argon2 has three tunable parameters: `M`, `p`, and `t`. `M` gives the total memory consumption of the algorithm in kilobytes. Increasing `p` increases the available parallelism of the computation. The `t` parameter gives the number of passes which are made over the data.

Note: Currently Botan does not make use of `p` > 1, so it is best to set it to 1 to minimize any advantage to highly parallel cracking attempts.

There are three variants of Argon2, namely Argon2d, Argon2i and Argon2id. Argon2d uses data dependent table lookups with may leak information about the password via side channel attacks, and is **not recommended** for password hashing. Argon2i uses data independent table lookups and is immune to these attacks, but at the cost of requiring higher `t` for security. Argon2id uses a hybrid approach which is thought to be highly secure. The algorithm designers recommend using Argon2id with `t` and `p` both equal to 1 and `M` set to the largest amount of memory usable in your environment.

std::string **argon2_generate_pwhash** (**const** char **password*, size_t *password_len*, RandomNumberGenerator &*rng*, size_t *p*, size_t *M*, size_t *t*, size_t *y* = 2, size_t *salt_len* = 16, size_t *output_len* = 32)

Generate an Argon2 hash of the specified password. The `y` parameter specifies the variant: 0 for Argon2d, 1 for Argon2i, and 2 for Argon2id.

bool **argon2_check_pwhash** (**const** char **password*, size_t *password_len*, **const** std::string &*hash*)

Verify an Argon2 password hash against the provided password. Returns false if the input hash seems malformed or if the computed hash does not match.

5.17.2 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"
```

Note: Due to the design of bcrypt, the password is effectively truncated at 72 characters; further characters are ignored and do not change the hash. To support longer passwords, one common approach is to pre-hash the password with SHA-256, then run bcrypt using the hex or base64 encoding of the hash as the password. (Many bcrypt implementations truncate the password at the first NULL character, so hashing the raw binary SHA-256 may cause problems. Botan's bcrypt implementation will hash whatever values are given in the `std::string` including any embedded NULLs so this is not an issue, but might cause interop problems if another library needs to validate the password hashes.)

std::string **generate_bcrypt** (**const** std::string &*password*, RandomNumberGenerator &*rng*, uint16_t *work_factor* = 12, char *bcrypt_version* = "a")

Takes the password to hash, a rng, and a work factor. The resulting password hash is returned as a string.

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 12 takes roughly twice as long as work factor 11. The default work factor was set to 10 up until the 2.8.0 release.

It is recommended to set the work factor as high as your system can tolerate (from a performance and latency perspective) since higher work factors greatly improve the security against GPU-based attacks. For example, for protecting high value administrator passwords, consider using work factor 15 or 16; at these work factors each bcrypt computation takes several seconds. Since admin logins will be relatively uncommon, it might be acceptable for each login attempt to take some time. As of 2018, a good password cracking rig (with 8 NVIDIA 1080 cards) can attempt about 1 billion bcrypt computations per month for work factor 13. For work factor 12, it can do twice as many. For work factor 15, it can do only one quarter as many attempts.

Due to bugs affecting various implementations of bcrypt, several different variants of the algorithm are defined. As of 2.7.0 Botan supports generating (or checking) the 2a, 2b, and 2y variants. Since Botan has never been affected by any of the bugs which necessitated these version upgrades, all three versions are identical beyond the version identifier. Which variant to use is controlled by the `bcrypt_version` argument.

The bcrypt work factor must be at least 4 (though at this work factor bcrypt is not very secure). The bcrypt format allows up to 31, but Botan currently rejects all work factors greater than 18 since even that work factor requires roughly 15 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.

5.17.3 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 foreseeable future, but it is specific to Botan rather than being a widely used password hash. Prefer bcrypt or Argon2.

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* = 15, uint8_t *alg_id* = 4)

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)

The work factor must be greater than zero and less than 512. This performs 10000 * `work_factor` PBKDF2 iterations, using 96 bits of salt taken from `rng`. Using work factor of 10 or more is recommended.

bool **check_passhash9**(**const** std::string &*password*, **const** std::string &*hash*)

Functions much like `check_bcrypt`

5.18 Cryptobox

5.18.1 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.

5.19 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 (however brute force attacks are possible). Later, the client and server can perform an SRP exchange, which results in a shared secret key. This key can be used for mutual authentication and/or encryption.

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 protected as carefully as a plaintext password would be.

BigInt **generate_srp6_verifier** (**const** std::string &*username*, **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. Later, the given username and password are used to by the client during the key agreement step.

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, and output a value *B* which is provided to the client.

SymmetricKey **step2**(**const** *BigInt* *&A*)

Takes the parameter A generated by srp6_client_agree, and return the shared secret key.

In the event of an impersonation attack (or wrong username/password, etc) no error occurs, but the key returned will be different on the two sides. The two sides must verify each other, for example by using the shared secret to key an HMAC and then exchanging authenticated messages.

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).

5.20 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, along with some implementations of same, are 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, but will not be able to tell how many other master keys are in use.

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.

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)`.

5.21 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 occurring.

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"
```

Byte streams 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 this, using the bzip2 compression filter:

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

Pipe pipe(new Compression_Filter("bzip2", 9));

```

(continues on next page)

(continued from previous page)

```
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 Compression_Filter("bzip2", 9), 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)
```

(continues on next page)

(continued from previous page)

```
      {
      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();
```

5.21.1 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,
                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)
          )
   );

```

(continues on next page)

(continued from previous page)

```
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*.

There is also a `Threaded_Fork` which acts the same as `Fork`, except it runs each of the filters in its own thread.

5.21.2 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 pass through the other version used:

```
Pipe pipe(new Fork(
              new Chain(new Hash_Filter("SHA-256"), new Hex_Encoder),
              new Hash_Filter("SHA-512")
              )
         );
```

5.21.3 Sources and Sinks

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.

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.

5.21.4 The Pipe API

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).

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.

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.

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.

5.21.5 Filter Catalog

This section documents most of the useful filters included in the library.

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
```

(continues on next page)

(continued from previous page)

```
                )
             )
   );
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*.

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.

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`.

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`.

5.21.6 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`:

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

This should just return a useful decription of the filter object.

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.

Note: Normally a filter does not need to override `send`, though it can for special handling. It does however need to call this function whenever it wants to produce output.

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.

5.22 Format Preserving Encryption

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 similarly 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 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. Or, if you were encrypting words in a dictionary, you could rank the words by their lexicographical order, and choose the modulus to be the number of words in the dictionary.

The interfaces for FE1 are defined in the header `fpe_fe1.h`:

New in version 2.5.0.

class FPE_FE1

FPE_FE1 (**const** *BigInt* &*n*, size_t *rounds* = 5, bool *compat_mode* = false, std::string *mac_algo* = "HMAC(SHA-256)")

Initialize an FPE operation to encrypt/decrypt integers less than *n*. It is expected that *n* is trivially factorable into small integers. Common usage would be n to be a power of 10.

Note that the default parameters to this constructor are **incompatible** with the `fe1_encrypt` and `fe1_decrypt` function originally added in 1.9.17. For compatibility, use 3 rounds and set `compat_mode` to true.

BigInt **encrypt** (**const** *BigInt* &*x*, **const** uint8_t *tweak*[], size_t *tweak_len*) **const**

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. The same tweak value must be used during decryption.

BigInt **decrypt** (**const** *BigInt* &*x*, **const** uint8_t *tweak*[], size_t *tweak_len*) **const**

Decrypts an FE1 ciphertext. The *tweak* must be the same as that provided to the encryption function. Returns the plaintext integer.

Note that there is not any implicit authentication or checking of data in FE1, so if you provide an incorrect key or tweak the result is simply a random integer.

BigInt **encrypt** (**const** *BigInt* &*x*, uint64_t *tweak*)

Convenience version of encrypt taking an integer tweak.

BigInt **decrypt** (**const** *BigInt* &*x*, uint64_t *tweak*)

Convenience version of decrypt taking an integer tweak.

There are two functions that handle the entire FE1 encrypt/decrypt operation. These are the original interface to FE1, first added in 1.9.17. However because they do the entire setup cost for each operation, they are significantly slower than the class-based API presented above.

Warning: These functions are hardcoded to use 3 rounds, which may be insufficient depending on the chosen modulus.

BigInt `FPE::`**fe1_encrypt** (**const** *BigInt* &*n*, **const** *BigInt* &*X*, **const** SymmetricKey &*key*, **const** std::vector<uint8_t> &*tweak*)

This creates an FPE_FE1 object, sets the key, and encrypts *X* using the provided tweak.

BigInt `FPE::`**fe1_decrypt** (**const** *BigInt* &*n*, **const** *BigInt* &*X*, **const** SymmetricKey &*key*, **const** std::vector<uint8_t> &*tweak*)

This creates an FPE_FE1 object, sets the key, and decrypts *X* using the provided tweak.

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=") {}

      std::string group() const override
         {
```

(continues on next page)

(continued from previous page)

```
         return "misc";
         }

      std::string description() const override
         {
         return "Encrypt the passed valid credit card number using FPE encryption";
         }

      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() << 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=") {}

      std::string group() const override
         {
         return "misc";
         }

      std::string description() const override
         {
         return "Decrypt the passed valid ciphertext credit card number using FPE␣
↪decryption";
         }

      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)");
            }
```

(continues on next page)

(continued from previous page)

```
         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
```

5.23 Threshold Secret Sharing

New in version 1.9.1.

Threshold secret sharing allows splitting a secret into `N` shares such that `M` (for specified `M` <= `N`) is sufficient to recover the secret, but an attacker with `M - 1` shares cannot derive any information about the secret.

The implementation in Botan follows an expired Internet draft "draft-mcgrew-tss-03". Several other implementations of this TSS format exist.

class RTSS_Share

static std::vector<*RTSS_Share*> **split** (uint8_t *M*, uint8_t *N*, **const** uint8_t *secret*[], uint16_t *secret_len*, **const** std::vector<uint8_t> &*identifier*, **const** std::string &*hash_fn*, *RandomNumberGenerator* &*rng*)

Split a secret. The identifier is an optional key identifier which may be up to 16 bytes long. Shorter identifiers are padded with zeros.

The hash function must be either "SHA-1", "SHA-256", or "None" to disable the checksum.

This will return a vector of length `N`, any `M` of these shares is sufficient to reconstruct the data.

static secure_vector<uint8_t> **reconstruct** (**const** std::vector<*RTSS_Share*> &*shares*)

Given a sufficient number of shares, reconstruct a secret.

RTSS_Share (**const** uint8_t *data*[], size_t *len*)

Read a TSS share as a sequence of bytes.

const secure_vector<uint8> &**data** () **const**

Return the data of this share.

uint8_t **share_id** () **const**

Return the share ID which will be in the range 1. . . 255

5.24 Elliptic Curve Operations

In addition to high level operations for signatures, key agreement, and message encryption using elliptic curve cryptography, the library contains lower level interfaces for performing operations such as elliptic curve point multiplication.

Only curves over prime fields are supported.

Many of these functions take a workspace, either a vector of words or a vector of BigInts. These are used to minimize memory allocations during common operations.

> **Warning:** You should only use these interfaces if you know what you are doing.

class EC_Group

EC_Group(**const** OID &*oid*)
Initialize an `EC_Group` using an OID referencing the curve parameters.

EC_Group(**const** std::string &*name*)
Initialize an `EC_Group` using a name or OID (for example "secp256r1", or "1.2.840.10045.3.1.7")

EC_Group(**const** *BigInt* &*p*, **const** *BigInt* &*a*, **const** *BigInt* &*b*, **const** *BigInt* &*base_x*, **const** *BigInt* &*base_y*, **const** *BigInt* &*order*, **const** *BigInt* &*cofactor*, **const** OID &*oid* = OID())
Initialize an elliptic curve group from the relevant parameters. This is used for example to create custom (application-specific) curves.

EC_Group(**const** std::vector<uint8_t> &*ber_encoding*)
Initialize an `EC_Group` by decoding a DER encoded parameter block.

std::vector<uint8_t> **DER_encode**(EC_Group_Encoding *form*) **const**
Return the DER encoding of this group.

std::string **PEM_encode**() **const**
Return the PEM encoding of this group (base64 of DER encoding plus header/trailer).

bool **a_is_minus_3**() **const**
Return true if the `a` parameter is congruent to -3 mod p.

bool **a_is_zero**() **const**
Return true if the `a` parameter is congruent to 0 mod p.

size_t **get_p_bits**() **const**
Return size of the prime in bits.

size_t **get_p_bytes**() **const**
Return size of the prime in bytes.

size_t **get_order_bits**() **const**
Return size of the group order in bits.

size_t **get_order_bytes**() **const**
Return size of the group order in bytes.

const *BigInt* &**get_p**() **const**
Return the prime modulus.

const *BigInt* &**get_a**() **const**
Return the `a` parameter of the elliptic curve equation.

const *BigInt* &**get_b**() **const**
    Return the b parameter of the elliptic curve equation.

const *PointGFp* &**get_base_point**() **const**
    Return the groups base point element.

const *BigInt* &**get_g_x**() **const**
    Return the x coordinate of the base point element.

const *BigInt* &**get_g_y**() **const**
    Return the y coordinate of the base point element.

const *BigInt* &**get_order**() **const**
    Return the order of the group generated by the base point.

const *BigInt* &**get_cofactor**() **const**
    Return the cofactor of the curve. In most cases this will be 1.

BigInt **mod_order**(**const** *BigInt* &*x*) **const**
    Reduce argument x modulo the curve order.

BigInt **inverse_mod_order**(**const** *BigInt* &*x*) **const**
    Return inverse of argument x modulo the curve order.

BigInt **multiply_mod_order**(**const** *BigInt* &*x*, **const** *BigInt* &*y*) **const**
    Multiply x and y and reduce the result modulo the curve order.

bool **verify_public_element**(**const** *PointGFp* &*y*) **const**
    Return true if y seems to be a valid group element.

const OID &**get_curve_oid**() **const**
    Return the OID used to identify the curve. May be empty.

PointGFp **point**(**const** *BigInt* &*x*, **const** *BigInt* &*y*) **const**
    Create and return a point with affine elements x and y. Note this function *does not* verify that x and y satisfy the curve equation.

PointGFp **point_multiply**(**const** *BigInt* &*x*, **const** *PointGFp* &*pt*, **const** *BigInt* &*y*) **const**
    Multi-exponentiation. Returns base_point*x + pt*y. Not constant time. (Ordinarily used for signature verification.)

PointGFp **blinded_base_point_multiply**(**const** *BigInt* &*k*, *RandomNumberGenerator* &*rng*, std::vector<*BigInt*> &*ws*) **const**
    Return base_point*k in a way that attempts to resist side channels.

BigInt **blinded_base_point_multiply_x**(**const** *BigInt* &*k*, *RandomNumberGenerator* &*rng*, std::vector<*BigInt*> &*ws*) **const**
    Like *blinded_base_point_multiply* but returns only the x coordinate.

PointGFp **blinded_var_point_multiply**(**const** *PointGFp* &*point*, **const** *BigInt* &*k*, *RandomNumberGenerator* &*rng*, std::vector<*BigInt*> &*ws*) **const**
    Return point*k in a way that attempts to resist side channels.

BigInt **random_scalar**(*RandomNumberGenerator* &*rng*) **const**
    Return a random scalar (ie an integer between 1 and the group order).

PointGFp **zero_point**() **const**
    Return the zero point (aka the point at infinity).

PointGFp **OS2ECP**(**const** uint8_t *bits*[], size_t *len*) **const**
    Decode a point from the binary encoding. This function verifies that the decoded point is a valid element on the curve.

bool **verify_group**(*RandomNumberGenerator* &*rng*, bool *strong* = false) **const**
  Attempt to verify the group seems valid.

static const std::set<std::string> &**known_named_groups**()
  Return a list of known groups, ie groups for which `EC_Group(name)` will succeed.

class PointGFp
  Stores elliptic curve points in Jacobian representation.

  std::vector<uint8_t> **encode**(*PointGFp*::Compression_Type *format*) **const**
    Encode a point in a way that can later be decoded with *EC_Group::OS2ECP*.

  PointGFp &**operator+=**(**const** *PointGFp* &*rhs*)
    Point addition.

  PointGFp &**operator-=**(**const** *PointGFp* &*rhs*)
    Point subtraction.

  PointGFp &**operator*=**(**const** *BigInt* &*scalar*)
    Point multiplication using Montgomery ladder.

> **Warning:** Prefer the blinded functions in `EC_Group`

  PointGFp &**negate**()
    Negate this point.

  BigInt **get_affine_x**() **const**
    Return the affine `x` coordinate of the point.

  BigInt **get_affine_y**() **const**
    Return the affine `y` coordinate of the point.

  void **force_affine**()
    Convert the point to its equivalent affine coordinates. Throws if this is the point at infinity.

  static void **force_all_affine**(std::vector<*PointGFp*> &*points*, secure_vector<word> &*ws*)
    Force several points to be affine at once. Uses Montgomery's trick to reduce number of inversions required, so this is much faster than calling `force_affine` on each point in sequence.

  bool **is_affine**() **const**
    Return true if this point is in affine coordinates.

  bool **is_zero**() **const**
    Return true if this point is zero (aka point at infinity).

  bool **on_the_curve**() **const**
    Return true if this point is on the curve.

  void **randomize_repr**(*RandomNumberGenerator* &*rng*)
    Randomize the point representation.

  bool **operator==**(**const** *PointGFp* &*other*) **const**
    Point equality. This compares the affine representations.

  void **add**(**const** *PointGFp* &*other*, std::vector<*BigInt*> &*workspace*)
    Point addition, taking a workspace.

  void **add_affine**(**const** *PointGFp* &*other*, std::vector<*BigInt*> &*workspace*)
    Mixed (Jacobian+affine) addition, taking a workspace.

> **Warning:** This function assumes that `other` is affine, if this is not correct the result will be invalid.

void **mult2** (std::vector<*BigInt*> &*workspace*)
: Point doubling.

void **mult2i** (size_t *i*, std::vector<*BigInt*> &*workspace*)
: Repeated point doubling.

PointGFp **plus** (**const** *PointGFp* &*other*, std::vector<*BigInt*> &*workspace*) **const**
: Point addition, returning the result.

PointGFp **double_of** (std::vector<*BigInt*> &*workspace*) **const**
: Point doubling, returning the result.

PointGFp **zero** () **const**
: Return the point at infinity

5.25 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 adapter 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.

5.26 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`

5.26.1 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`.

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 behavior `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.

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.

5.26.2 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

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;
```

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.

static std::vector<SlotId> **get_available_slots** (*Module* &*module*, bool *token_present*)
: Retrieve available slot ids by calling this static method.

  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*.

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 ) );
```

(continues on next page)

(continued from previous page)

```
// 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
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 );
```

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. The destructor calls `C_Logout` if a user is logged in to this session and always `C_CloseSession`.

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`.

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
↪' };
```

(continues on next page)

(continued from previous page)

```
session.login( Botan::PKCS11::UserType::SO, so_pin );

// change pin to old pin
session.init_pin( pin );
```

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 themselves 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␣
↪);
```

(continues on next page)

(continued from previous page)

```
// 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 );

// 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();
```

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
```

(continues on next page)

(continued from previous page)

```
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 );

// 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 );
```

(continues on next page)

(continued from previous page)

```
auto ok = verifier.verify_message( plaintext, signature );
```

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";
```

(continues on next page)

(continued from previous page)

```
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();

/************ 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 );
```

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
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();
```

(continues on next page)

(continued from previous page)

```
/************ 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 ) );

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;
```

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() );

/************ 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() );
```

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␣
↪);

/************* 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 );
```

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() );
```

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:

```
// 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());
```

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]
CardOS 5.3	mostly works	Windows 10, 64-bit, version 1903	API Version 5.5.1 (Cryptoki v2.11)	2.12.0 unreleased, Cryptoki v2.40	[53]
Gemalto IDPrime MD 3840	mostly works	Windows 10, 64-bit, version 1709	IDGo 800, v1.2.4 (Cryptoki v2.20)	2.4.0, Cryptoki v2.40	[54]
SoftHSM 2.3.0 (OpenSSL 1.0.2g)	works	Windows 10, 64-bit, version 1709	Cryptoki v2.40	2.4.0, Cryptoki v2.40	
SoftHSM 2.5.0 (OpenSSL 1.1.1)	works	Windows 10, 64-bit, version 1803	Cryptoki v2.40	2.11.0, Cryptoki v2.40	

[50] Failing operations for CardOS 4.4:

- object_copy[Page 171, 20]
- rsa_privkey_export[Page 171, 21]
- rsa_generate_private_key[Page 171, 22]
- rsa_sign_verify[Page 171, 23]
- ecdh_privkey_import[Page 171, 3]
- ecdh_privkey_export[Page 171, 2]
- ecdh_pubkey_import[Page 171, 4]
- ecdh_pubkey_export[Page 171, 4]
- ecdh_generate_private_key[Page 171, 3]
- ecdh_generate_keypair[Page 171, 3]
- ecdh_derive[Page 171, 3]
- ecdsa_privkey_import[Page 171, 3]
- ecdsa_privkey_export[Page 171, 2]
- ecdsa_pubkey_import[Page 171, 4]
- ecdsa_pubkey_export[Page 171, 4]

Error descriptions

- ecdsa_generate_private_key[Page 171, 3]
- ecdsa_generate_keypair[Page 171, 3]
- ecdsa_sign_verify[Page 171, 3]
- rng_add_entropy[Page 171, 5]

[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)
[51] Failing operations for CardOS 5.0

- object_copy[?]
- rsa_privkey_export[?]
- rsa_generate_private_key[?]
- rsa_sign_verify[?]
- ecdh_privkey_export[?]
- ecdh_pubkey_import[?]
- ecdh_generate_private_key[Page 171, 32]
- ecdh_generate_keypair[?]
- ecdh_derive[Page 171, 33]
- ecdsa_privkey_export[?]
- ecdsa_generate_private_key[Page 171, 30]
- ecdsa_generate_keypair[Page 171, 30]
- ecdsa_sign_verify[Page 171, 30]
- rng_add_entropy[?]

[32] Invalid argument OS2ECP: Unknown format type 155
[33] Invalid argument OS2ECP: Unknown format type 92
[30] Invalid argument Decoding error: BER: Value truncated
[52] Failing operations for CardOS 5.3

- object_copy[?]
- rsa_privkey_export[?]
- rsa_generate_private_key[?]
- rsa_sign_verify[?]
- ecdh_privkey_export[?]
- ecdh_pubkey_import[Page 172, 6]
- ecdh_pubkey_export[Page 172, 6]
- ecdh_generate_private_key[?]
- ecdh_generate_keypair[Page 172, 31]
- ecdh_derive[?]
- ecdsa_privkey_export[?]
- ecdsa_pubkey_import[Page 172, 6]
- ecdsa_pubkey_export[Page 172, 6]
- ecdsa_generate_private_key[Page 172, 31]
- ecdsa_generate_keypair[Page 172, 31]

5.27 Trusted Platform Module (TPM)

New in version 1.11.26.

- ecdsa_sign_verify[Page 172, 34]
- rng_add_entropy[?]

[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
[53] Failing operations for CardOS 5.3 (middelware 5.5.1)

- ecdh_privkey_export[?]
- ecdh_generate_private_key[Page 172, 35]
- ecdsa_privkey_export[?]
- ecdsa_generate_private_key[Page 172, 36]
- c_copy_object[?]
- object_copy[?]
- rng_add_entropy[?]
- rsa_sign_verify[?]
- rsa_privkey_export[?]
- rsa_generate_private_key[Page 172, 9]

[35] Invalid argument OS2ECP: Unknown format type 82
[36] Invalid argument OS2ECP: Unknown format type 102
[9] CKR_TEMPLATE_INCOMPLETE (0xD0=208)
[54] Failing operations for Gemalto IDPrime MD 3840

- session_login_logout[?]
- session_info[?]
- set_pin[?]
- initialize[?]
- change_so_pin[?]
- object_copy[?]
- rsa_generate_private_key[Page 172, 7]
- rsa_encrypt_decrypt[Page 172, 8]
- rsa_sign_verify[?]
- rng_add_entropy[?]

[7] CKR_TEMPLATE_INCONSISTENT (0xD1=209)
[8] CKR_ENCRYPTED_DATA_INVALID | CKM_SHA256_RSA_PKCS (0x40=64)

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

5.28 One Time Passwords

New in version 2.2.0.

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.

5.28.1 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 associated 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.

5.28.2 TOTP

TOTP is based on the same algorithm as HOTP, but instead of a counter a timestamp is used.

class TOTP

TOTP (**const** SymmetricKey &*key*, **const** std::string &*hash_algo* = "SHA-1", size_t *digits* = 6, size_t *time_step* = 30)
Setup to perform TOTP authentication using secret key *key*.

uint32_t **generate_totp** (std::chrono::system_clock::time_point *time_point*)

uint32_t **generate_totp** (uint64_t *unix_time*)
Generate and return a TOTP code based on a timestamp.

bool **verify_totp** (uint32_t *otp*, std::chrono::system_clock::time_point *time*, size_t *clock_drift_accepted* = 0)

bool **verify_totp** (uint32_t *otp*, uint64_t *unix_time*, size_t *clock_drift_accepted* = 0)
Return true if the provided OTP code is correct for the provided timestamp. If required, use *clock_drift_accepted* to deal with the client and server having slightly different clocks.

5.29 Roughtime

New in version 2.13.0.

Botan includes a Roughtime client, available in `botan/roughtime.h`

5.30 FFI (C Binding)

New in version 1.11.14.

Botan's ffi module provides a C89 binding 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 the tests (https://github.com/randombit/botan/blob/master/src/tests/test_ffi.cpp).

5.30.1 Return Codes

Almost all functions in the Botan C interface return an `int` error code. The only exceptions are a handful of functions (like *`botan_ffi_api_version`*) which cannot fail in any circumstances.

The FFI functions return a non-negative integer (usually 0) to indicate success, or a negative integer to represent an error. A few functions (like *`botan_block_cipher_block_size`*) return positive integers instead of zero on success.

The error codes returned in certain error situations may change over time. This especially applies to very generic errors like *`BOTAN_FFI_ERROR_EXCEPTION_THROWN`* and *`BOTAN_FFI_ERROR_UNKNOWN_ERROR`*. For instance, before 2.8, setting an invalid key length resulted in *`BOTAN_FFI_ERROR_EXCEPTION_THROWN`* but now this is specially handled and returns *`BOTAN_FFI_ERROR_INVALID_KEY_LENGTH`* instead.

The following enum values are defined in the FFI header:

`enumerator BOTAN_FFI_SUCCESS` = 0
: Generally returned to indicate success

`enumerator BOTAN_FFI_INVALID_VERIFIER` = 1
: Note this value is positive, but still represents an error condition. In indicates that the function completed successfully, but the value provided was not correct. For example *`botan_bcrypt_is_valid`* returns this value if the password did not match the hash.

`enumerator BOTAN_FFI_ERROR_INVALID_INPUT` = -1
: The input was invalid. (Currently this error return is not used.)

`enumerator BOTAN_FFI_ERROR_BAD_MAC` = -2
: While decrypting in an AEAD mode, the tag failed to verify.

`enumerator BOTAN_FFI_ERROR_INSUFFICIENT_BUFFER_SPACE` = -10
: Functions which write a variable amount of space return this if the indicated buffer length was insufficient to write the data. In that case, the output length parameter is set to the size that is required.

`enumerator BOTAN_FFI_ERROR_EXCEPTION_THROWN` = -20
: An exception was thrown while processing this request, but no further details are available.

> **Note:** If the environment variable `BOTAN_FFI_PRINT_EXCEPTIONS` is set to any non-empty value, then any exception which is caught by the FFI layer will first print the exception message to stderr before returning an error. This is sometimes useful for debugging.

`enumerator BOTAN_FFI_ERROR_OUT_OF_MEMORY` = -21
: Memory allocation failed

`enumerator BOTAN_FFI_ERROR_BAD_FLAG` = -30
: A value provided in a *flag* variable was unknown.

`enumerator BOTAN_FFI_ERROR_NULL_POINTER` = -31
: A null pointer was provided as an argument where that is not allowed.

`enumerator BOTAN_FFI_ERROR_BAD_PARAMETER` = -32
: An argument did not match the function.

`enumerator BOTAN_FFI_ERROR_KEY_NOT_SET` = -33
: An object that requires a key normally must be keyed before use (eg before encrypting or MACing data). If this is not done, the operation will fail and return this error code.

`enumerator BOTAN_FFI_ERROR_INVALID_KEY_LENGTH` = -34
: An invalid key length was provided with a call to `x_set_key`.

enumerator BOTAN_FFI_ERROR_NOT_IMPLEMENTED = -40
This is returned if the functionality is not available for some reason. For example if you call `botan_hash_init` with a named hash function which is not enabled, this error is returned.

enumerator BOTAN_FFI_ERROR_INVALID_OBJECT = -50
This is used if an object provided did not match the function. For example calling `botan_hash_destroy` on a `botan_rng_t` object will cause this return.

enumerator BOTAN_FFI_ERROR_UNKNOWN_ERROR = -100
Something bad happened, but we are not sure why or how.

5.30.2 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-form string describing the version. The return value is a statically allocated string.

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 YYYYMMDD, or 0 if an unreleased version

FFI Versions

This maps the FFI API version to the first version of the library that supported it.

FFI Version	Supported Starting
20191214	2.13.0
20180713	2.8.0
20170815	2.3.0
20170327	2.1.0
20150515	2.0.0

5.30.3 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.

int **botan_hex_decode**(**const** char **hex_str*, size_t *in_len*, uint8_t **out*, size_t **out_len*)
Hex decode some data

5.30.4 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`, "user-threadsafe": serialized `AutoSeeded_RNG`, "null": `Null_RNG` (always fails), "hwrnd" or "rdrand": `Processor_RNG` (if available)

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_reseed_from_rng**(*botan_rng_t* *rng*, *botan_rng_t* *src*, size_t *bits*)
Reseeds the random number generator with *bits* number of bits taken from the given source RNG.

int **botan_rng_add_entropy**(*botan_rng_t* *rng*, **const** uint8_t *seed*[], size_t *len*)
Adds the provided seed material to the internal RNG state.

This call may be ignored by certain RNG instances (such as RDRAND or, on some systems, the system RNG).

int **botan_rng_destroy**(*botan_rng_t* *rng*)
Destroy the object created by `botan_rng_init`.

5.30.5 Block Ciphers

New in version 2.1.0.

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_name**(*botan_block_cipher_t cipher*, char **name*, size_t **name_len*)
: Return the name of this block cipher algorithm, which may nor may not exactly match what was passed to `botan_block_cipher_init`.

int **botan_block_cipher_get_keyspec**(*botan_block_cipher_t cipher*, size_t **out_minimum_keylength*, size_t **out_maximum_keylength*, size_t **out_keylength_modulo*)
: Return the limits on the key which can be provided to this cipher. If any of the parameters are null, no output is written to that field. This allows retrieving only (say) the maximum supported keylength, if that is the only information needed.

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`.

5.30.6 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_name**(*botan_hash_t hash*, char **name*, size_t **name_len*)
: Write the name of the hash function to the provided buffer.

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.

5.30.7 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.

5.30.8 Symmetric Ciphers

typedef opaque ***botan_cipher_t**
An opaque data type for a symmetric cipher object. 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 bitof `flags` specifies if encrypt or decrypt, ie use 0 for encryption and 1 for decryption.

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_is_authenticated**(*botan_cipher_t cipher*)

size_t **botan_cipher_get_tag_length**(*botan_cipher_t cipher*, size_t **tag_len*)
Write the tag length of the cipher to `tag_len`. This will be zero for non-authenticated ciphers.

int **botan_cipher_valid_nonce_length**(*botan_cipher_t cipher*, size_t *nl*)
Returns 1 if the nonce length is valid, or 0 otherwise. Returns -1 on error (such as the cipher object being invalid).

size_t **botan_cipher_get_default_nonce_length**(*botan_cipher_t cipher*, size_t **nl*)
Return the default nonce length

int **botan_cipher_set_associated_data**(*botan_cipher_t cipher*, **const** uint8_t **ad*, size_t *ad_len*)
Set associated data. Will fail unless the cipher is an AEAD.

int **botan_cipher_start**(*botan_cipher_t cipher*, **const** uint8_t **nonce*, size_t *nonce_len*)
Start processing a message using the provided nonce.

int **botan_cipher_update**(*botan_cipher_t cipher*, uint32_t *flags*, uint8_t *output*[], size_t *output_size*, size_t **output_written*, **const** uint8_t *input_bytes*[], size_t *input_size*, size_t **input_consumed*)

Encrypt or decrypt data.

5.30.9 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*.

5.30.10 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*.

5.30.11 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` 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` 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` 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`

5.30.12 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 `BOTAN_SUCCESS` if if this password/hash combination is valid, *`BOTAN_FFI_INVALID_VERIFIER`* if the combination is not valid (but otherwise well formed), negative on error.

5.30.13 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*)
Create an RSA key of the given size

int **botan_privkey_create_ecdsa** (*botan_privkey_t *key*, *botan_rng_t rng*, **const** char **curve*)
Create a ECDSA key of using a named curve

int **botan_privkey_create_ecdh** (*botan_privkey_t *key*, *botan_rng_t rng*, **const** char **curve*)
Create a ECDH key of using a named curve

int **botan_privkey_create_mceliece** (*botan_privkey_t *key*, *botan_rng_t rng*, size_t *n*, size_t *t*)
Create a McEliece key using the specified parameters. See *McEliece* for details on choosing parameters.

int **botan_privkey_create_dh**(*botan_privkey_t *key*, *botan_rng_t rng*, **const** char **params*)
: Create a finite field Diffie-Hellman key using the specified named group, for example "modp/ietf/3072".

int **botan_privkey_load**(*botan_privkey_t *key*, *botan_rng_t rng*, **const** uint8_t *bits*[], size_t *len*, **const** char **password*)
: Load a private key. If the key is encrypted, `password` will be used to attempt decryption.

int **botan_privkey_destroy**(*botan_privkey_t key*)
: Destroy the object.

int **botan_privkey_export**(*botan_privkey_t key*, uint8_t *out*[], size_t **out_len*, uint32_t *flags*)
: Export a public key. If flags is 1 then PEM format is used.

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`

int **botan_privkey_export_pubkey**(*botan_pubkey_t *out*, *botan_privkey_t in*)

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.

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_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.

5.30.14 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.

5.30.15 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.

5.30.16 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.

5.30.17 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.

5.30.18 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*)
Create a new operation object which can be used to encrypt using the provided key and the specified padding scheme (such as "OAEP(SHA-256)" for use with RSA). Flags should be 0 in this version.

int **botan_pk_op_encrypt_destroy** (*botan_pk_op_encrypt_t op*)
Destroy the object.

int **botan_pk_op_encrypt_output_length** (*botan_pk_op_encrypt_t op*, size_t *ptext_len*, size_t **ctext_len*)
Returns an upper bound on the output length if a plaintext of length `ptext_len` is encrypted with this key/parameter setting. This allows correctly sizing the buffer that is passed to `botan_pk_op_encrypt`.

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*)
  Encrypt the provided data using the key, placing the output in *out*. If *out* is NULL, writes the length of what the ciphertext would have been to **out_len*. However this is computationally expensive (the encryption actually occurs, then the result is discarded), so it is better to use `botan_pk_op_encrypt_output_length` to correctly size the buffer.

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_output_length** (*botan_pk_op_decrypt_t op*, size_t *ctext_len*, size_t **ptext_len*)
  For a given ciphertext length, returns the upper bound on the size of the plaintext that might be enclosed. This allows properly sizing the output buffer passed to `botan_pk_op_decrypt`.

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*)

5.30.19 Signature Generation

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*)
  Create a signature operator for the provided key. The padding string specifies what hash function and padding should be used, for example "PKCS1v15(SHA-256)" or "EMSA1(SHA-384)".

int **botan_pk_op_sign_destroy** (*botan_pk_op_sign_t op*)
  Destroy an object created by `botan_pk_op_sign_create`.

int **botan_pk_op_sign_output_length** (*botan_pk_op_sign_t op*, size_t **sig_len*)
  Writes the length of the signatures that this signer will produce. This allows properly sizing the buffer passed to `botan_pk_op_sign_finish`.

int **botan_pk_op_sign_update** (*botan_pk_op_sign_t op*, **const** uint8_t *in*[], size_t *in_len*)
  Add bytes of the message to be signed.

int **botan_pk_op_sign_finish** (*botan_pk_op_sign_t op*, *botan_rng_t rng*, uint8_t *sig*[], size_t **sig_len*)
  Produce a signature over all of the bytes passed to `botan_pk_op_sign_update`. Afterwards, the sign operator is reset and may be used to sign a new message.

5.30.20 Signature Verification

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*)
  Add bytes of the message to be verified

int **botan_pk_op_verify_finish** (*botan_pk_op_verify_t op*, **const** uint8_t *sig*[], size_t *sig_len*)
Verify if the signature provided matches with the message provided as calls to `botan_pk_op_verify_update`.

5.30.21 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*)

5.30.22 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*)
Load a certificate from the DER or PEM representation

int **botan_x509_cert_load_file** (*botan_x509_cert_t *cert_obj*, **const** char **filename*)
Load a certificate from a file.

int **botan_x509_cert_dup** (*botan_x509_cert_t *cert_obj*, *botan_x509_cert_t cert*)
Create a new object that refers to the same certificate.

int **botan_x509_cert_destroy** (*botan_x509_cert_t cert*)
Destroy the certificate object

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*)
Return the time the certificate becomes valid, as a string in form "YYYYMMDDHHMMSSZ" where Z is a literal character reflecting that this time is relative to UTC. Prefer `botan_x509_cert_not_before`.

int **botan_x509_cert_get_time_expires** (*botan_x509_cert_t cert*, char *out*[], size_t **out_len*)
Return the time the certificate expires, as a string in form "YYYYMMDDHHMMSSZ" where Z is a literal character reflecting that this time is relative to UTC. Prefer `botan_x509_cert_not_after`.

int **botan_x509_cert_not_before** (*botan_x509_cert_t cert*, uint64_t **time_since_epoch*)
Return the time the certificate becomes valid, as seconds since epoch.

int **botan_x509_cert_not_after** (*botan_x509_cert_t cert*, uint64_t **time_since_epoch*)
Return the time the certificate expires, as seconds since epoch.

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*)
Return the serial number of the certificate.

int **botan_x509_cert_get_authority_key_id**(*botan_x509_cert_t cert*, uint8_t *out*[], size_t **out_len*)
Return the authority key ID set in the certificate, which may be empty.

int **botan_x509_cert_get_subject_key_id**(*botan_x509_cert_t cert*, uint8_t *out*[], size_t **out_len*)
Return the subject key ID set in the certificate, which may be empty.

int **botan_x509_cert_get_public_key_bits**(*botan_x509_cert_t cert*, uint8_t *out*[], size_t **out_len*)
Get the serialized representation of the public key included in this certificate

int **botan_x509_cert_get_public_key**(*botan_x509_cert_t cert*, *botan_pubkey_t* **key*)
Get the public key included in this certificate as a newly allocated object

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*)
Get a value from the issuer DN field.

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*)
Get a value from the subject DN field.

int **botan_x509_cert_to_string**(*botan_x509_cert_t cert*, char *out*[], size_t **out_len*)
Format the certificate as a free-form string.

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*)

int **botan_x509_cert_verify**(int **validation_result*, *botan_x509_cert_t cert*, **const** *botan_x509_cert_t* **intermediates*, size_t *intermediates_len*, **const** *botan_x509_cert_t* **trusted*, size_t *trusted_len*, **const** char **trusted_path*, size_t *required_strength*, **const** char **hostname*, uint64_t *reference_time*)
Verify a certificate. Returns 0 if validation was successful, 1 if unsuccessful, or negative on error.

Sets `validation_result` to a code that provides more information.

If not needed, set `intermediates` to NULL and `intermediates_len` to zero.

If not needed, set `trusted` to NULL and `trusted_len` to zero.

The `trusted_path` refers to a directory where one or more trusted CA certificates are stored. It may be NULL if not needed.

Set `required_strength` to indicate the minimum key and hash strength that is allowed. For instance setting to 80 allows 1024-bit RSA and SHA-1. Setting to 110 requires 2048-bit RSA and SHA-256 or higher. Set to zero to accept a default.

Set `reference_time` to be the time which the certificate chain is validated against. Use zero to use the current system clock.

int **botan_x509_cert_verify_with_crl** (int **validation_result*, *botan_x509_cert_t* *cert*, **const** *botan_x509_cert_t* **intermediates*, size_t *intermediates_len*, **const** *botan_x509_cert_t* **trusted*, size_t *trusted_len*, **const** *botan_x509_crl_t* **crls*, size_t *crls_len*, **const** char **trusted_path*, size_t *required_strength*, **const** char **hostname*, uint64_t *reference_time*)

Certificate path validation supporting Certificate Revocation Lists.

Works the same as `botan_x509_cert_cerify`.

`crls` is an array of `botan_x509_crl_t` objects, `crls_len` is its length.

const char ***botan_x509_cert_validation_status** (int *code*)

Return a (statically allocated) string associated with the verification result.

5.30.23 X.509 Certificate Revocation Lists

typedef opaque ***botan_x509_crl_t**

An opaque data type for an X.509 CRL.

int **botan_x509_crl_load** (*botan_x509_crl_t* **crl_obj*, **const** uint8_t *crl*[], size_t *crl_len*)

Load a CRL from the DER or PEM representation.

int **botan_x509_crl_load_file** (*botan_x509_crl_t* **crl_obj*, **const** char **filename*)

Load a CRL from a file.

int **botan_x509_crl_destroy** (*botan_x509_crl_t* *crl*)

Destroy the CRL object.

int **botan_x509_is_revoked** (*botan_x509_crl_t* *crl*, *botan_x509_cert_t* *cert*)

Check whether a given `crl` contains a given `cert`. Return `0` when the certificate is revoked, `-1` otherwise.

5.31 Environment Variables

Certain environment variables can affect or tune the behavior of the library. The variables and their behavior are described here.

- `BOTAN_THREAD_POOL_SIZE` controls the number of threads which will be created for a thread pool used for some purposes within the library. If not set then it defaults to the number of CPUs available on the system.
- `BOTAN_MLOCK_POOL_SIZE` controls the total amount of memory which will be locked in memory using `mlock` or `VirtualLock` and managed in a memory pool. If set to `0` (or indeed any value smaller than the system page size), then the memory pool is disabled.
- `BOTAN_FFI_PRINT_EXCEPTIONS` if this variable is set (to any value), then if an exception is caught by the FFI layer, before returning an error code, it will print the text message of the exception to stderr. This is primarily intended for debugging.

5.32 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.

Starting in 2.8, the class names were renamed to match Python standard conventions. However aliases are defined which allow older code to continue to work; the older names are mentioned as "previously X". These aliases will be removed in a future major release.

5.32.1 Versioning

`botan.`**`version_major`**`()`
: Returns the major number of the library version.

`botan.`**`version_minor`**`()`
: Returns the minor number of the library version.

`botan.`**`version_patch`**`()`
: Returns the patch number of the library version.

`botan.`**`version_string`**`()`
: Returns a free form version string for the library

5.32.2 Random Number Generators

`class` `botan.`**`RandomNumberGenerator`**`(`*rng_type='system'*`)`

Previously `rng`

Type 'user' also allowed (userspace HMAC_DRBG 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.RandomNumberGenerator().get(32)*.

`get``(`*length*`)`
: Return some bytes

`reseed``(`*bits=256*`)`
: Meaningless on system RNG, on userspace RNG causes a reseed/rekey

`reseed_from_rng``(`*source_rng*, *bits=256*`)`
: Take bits from the source RNG and use it to seed `self`

`add_entropy``(`*seed*`)`
: Add some unpredictable seed data to the RNG

5.32.3 Hash Functions

`class` `botan.`**`HashFunction`**`(`*algo*`)`

Previously `hash_function`

The `algo` param is a string (eg 'SHA-1', 'SHA-384', 'BLAKE2b')

`algo_name``()`
: Returns the name of this algorithm

clear()
Clear state

output_length()
Return output length in bytes

update(*x*)
Add some input

final()
Returns the hash of all input provided, resets for another message.

5.32.4 Message Authentication Codes

class botan.**MsgAuthCode**(*algo*)
Previously `message_authentication_code`

Algo is a string (eg 'HMAC(SHA-256)', 'Poly1305', 'CMAC(AES-256)')

algo_name()
Returns the name of this algorithm

clear()
Clear internal state including the key

output_length()
Return the output length in bytes

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.

5.32.5 Ciphers

class botan.**SymmetricCipher**(*object*, *algo*, *encrypt=True*)

Previously `cipher`

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

algo_name()
Returns the name of this algorithm

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

set_assoc_data(*ad*)
: Sets the associated data. Fails if this is not an AEAD mode

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

5.32.6 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.

5.32.7 PBKDF

botan.**pbkdf**(*algo*, *password*, *out_len*, *iterations=100000*, *salt=None*)
: Runs a PBKDF2 algo specified as a string (eg 'PBKDF2(SHA-256)', 'PBKDF2(CMAC(Blowfish))'). Runs with specified 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

5.32.8 Scrypt

New in version 2.8.0.

`botan.`**`scrypt`**(*out_len*, *password*, *salt*, *N=1024*, *r=8*, *p=8*)
Runs Scrypt key derivation function over the specified password and salt using Scrypt parameters N, r, p.

5.32.9 KDF

`botan.`**`kdf`**(*algo*, *secret*, *out_len*, *salt*)
Performs a key derviation function (such as "HKDF(SHA-384)") over the provided secret and salt values. Returns a value of the specified length.

5.32.10 Public Key

`class` `botan.`**`PublicKey`**(*object*)
Previously `public_key`

`classmethod load`(*val*)
Load a public key. The value should be a PEM or DER blob.

`classmethod load_rsa`(*n*, *e*)
Load an RSA public key giving the modulus and public exponent as integers.

`classmethod load_dsa`(*p*, *q*, *g*, *y*)
Load an DSA public key giving the parameters and public value as integers.

`classmethod load_dh`(*p*, *g*, *y*)
Load an Diffie-Hellman public key giving the parameters and public value as integers.

`classmethod load_elgamal`(*p*, *q*, *g*, *y*)
Load an ElGamal public key giving the parameters and public value as integers.

`classmethod load_ecdsa`(*curve*, *pub_x*, *pub_y*)
Load an ECDSA public key giving the curve as a string (like "secp256r1") and the public point as a pair of integers giving the affine coordinates.

`classmethod load_ecdh`(*curve*, *pub_x*, *pub_y*)
Load an ECDH public key giving the curve as a string (like "secp256r1") and the public point as a pair of integers giving the affine coordinates.

`classmethod load_sm2`(*curve*, *pub_x*, *pub_y*)
Load a SM2 public key giving the curve as a string (like "sm2p256v1") and the public point as a pair of integers giving the affine coordinates.

`check_key(rng_obj, strong=True):`
Test the key for consistency. If `strong` is `True` then more expensive tests are performed.

`export`(*pem=False*)
Exports the public key using the usual X.509 SPKI representation. If `pem` is True, the result is a PEM encoded string. Otherwise it is a binary DER value.

`to_der`()
Like `self.export(False)`

`to_pem`()
Like `self.export(True)`

`get_field`(*field_name*)
Return an integer field related to the public key. The valid field names vary depending on the algorithm. For example RSA public modulus can be extracted with `rsa_key.get_field("n")`.

`fingerprint`(*hash='SHA-256'*)
Returns a hash of the public key

`algo_name`()
Returns the algorithm name

`estimated_strength`()
Returns the estimated strength of this key against known attacks (NFS, Pollard's rho, etc)

5.32.11 Private Key

`class` `botan.`**`PrivateKey`**
Previously `private_key`

`classmethod create`(*algo*, *param*, *rng*)
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).

`classmethod load`(*val*, *passphrase=''*)
Return a private key (DER or PEM formats accepted)

`classmethod load_rsa`(*p*, *q*, *e*)
Return a private RSA key

`classmethod load_dsa`(*p*, *q*, *g*, *x*)
Return a private DSA key

`classmethod load_dh`(*p*, *g*, *x*)
Return a private DH key

`classmethod load_elgamal`(*p*, *q*, *g*, *x*)
Return a private ElGamal key

`classmethod load_ecdsa`(*curve*, *x*)
Return a private ECDSA key

`classmethod load_ecdh`(*curve*, *x*)
Return a private ECDH key

`classmethod load_sm2`(*curve*, *x*)
Return a private SM2 key

`get_public_key`()
Return a public_key object

`to_pem`()
Return the PEM encoded private key (unencrypted). Like `self.export(True)`

`to_der`()
Return the PEM encoded private key (unencrypted). Like `self.export(False)`

`check_key(rng_obj, strong=True):`
Test the key for consistency. If `strong` is `True` then more expensive tests are performed.

`algo_name`()
Returns the algorithm name

export (*pem=False*)
Exports the private key in PKCS8 format. If `pem` is True, the result is a PEM encoded string. Otherwise it is a binary DER value. The key will not be encrypted.

export_encrypted (*passphrase*, *rng*, *pem=False*, *msec=300*, *cipher=None*, *pbkdf=None*)
Exports the private key in PKCS8 format, encrypted using the provided passphrase. If `pem` is True, the result is a PEM encoded string. Otherwise it is a binary DER value.

get_field (*field_name*)
Return an integer field related to the public key. The valid field names vary depending on the algorithm. For example first RSA secret prime can be extracted with `rsa_key.get_field("p")`. This function can also be used to extract the public parameters.

5.32.12 Public Key Operations

class botan.**PKEncrypt** (*pubkey*, *padding*)
Previously `pk_op_encrypt`

encrypt (*msg*, *rng*)

class botan.**PKDecrypt** (*privkey*, *padding*)
Previously `pk_op_decrypt`

decrypt (*msg*)

class botan.**PKSign** (*privkey*, *hash_w_padding*)
Previously `pk_op_sign`

update (*msg*)

finish (*rng*)

class botan.**PKVerify** (*pubkey*, *hash_w_padding*)
Previously `pk_op_verify`

update (*msg*)

check_signature (*signature*)

class botan.**PKKeyAgreement** (*privkey*, *kdf*)
Previously `pk_op_key_agreement`

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.

5.32.13 Multiple Precision Integers (MPI)

New in version 2.8.0.

class botan.**MPI** (*initial_value=None*, *radix=None*)
Initialize an MPI object with specified value, left as zero otherwise. The `initial_value` should be an `int`, `str`, or `MPI`. The `radix` value should be set to 16 when initializing from a base 16 *str* value.

Most of the usual arithmetic operators (`__add__`, `__mul__`, etc) are defined.

inverse_mod (*modulus*)
Return the inverse of `self` modulo `modulus`, or zero if no inverse exists

is_prime(*rng*, *prob=128*)
Test if `self` is prime

pow_mod(exponent, modulus):
Return `self` to the `exponent` power modulo `modulus`

mod_mul(other, modulus):
Return the multiplication product of `self` and `other` modulo `modulus`

gcd(other):
Return the greatest common divisor of `self` and `other`

5.32.14 Format Preserving Encryption (FE1 scheme)

New in version 2.8.0.

class botan.**FormatPreservingEncryptionFE1**(*modulus*, *key*, *rounds=5*, *compat_mode=False*)
Initialize an instance for format preserving encryption

encrypt(*msg*, *tweak*)
The msg should be a botan2.MPI or an object which can be converted to one

decrypt(*msg*, *tweak*)
The msg should be a botan2.MPI or an object which can be converted to one

5.32.15 HOTP

New in version 2.8.0.

class botan.**HOTP**(*key*, *hash='SHA-1'*, *digits=6*)

generate(*counter*)
Generate an HOTP code for the provided counter

check(*code*, *counter*, *resync_range=0*)
Check if provided `code` is the correct code for `counter`. If `resync_range` is greater than zero, HOTP also checks up to `resync_range` following counter values.

Returns a tuple of (bool,int) where the boolean indicates if the code was valid, and the int indicates the next counter value that should be used. If the code did not verify, the next counter value is always identical to the counter that was passed in. If the code did verify and resync_range was zero, then the next counter will always be counter+1.

5.32.16 X509Cert

class botan.**X509Cert**(*filename=None*, *buf=None*)

time_starts()
Return the time the certificate becomes valid, as a string in form "YYYYMMDDHHMMSSZ" where Z is a literal character reflecting that this time is relative to UTC.

time_expires()
Return the time the certificate expires, as a string in form "YYYYMMDDHHMMSSZ" where Z is a literal character reflecting that this time is relative to UTC.

to_string()
Format the certificate as a free-form string.

fingerprint(*hash_algo='SHA-256'*)
Return a fingerprint for the certificate, which is basically just a hash of the binary contents. Normally SHA-1 or SHA-256 is used, but any hash function is allowed.

serial_number()
Return the serial number of the certificate.

authority_key_id()
Return the authority key ID set in the certificate, which may be empty.

subject_key_id()
Return the subject key ID set in the certificate, which may be empty.

subject_public_key_bits()
Get the serialized representation of the public key included in this certificate.

subject_public_key()
Get the public key included in this certificate as an object of class `PublicKey`.

subject_dn(*key*, *index*)
Get a value from the subject DN field.

`key` specifies a value to get, for instance `"Name"` or *"Country"*.

issuer_dn(*key*, *index*)
Get a value from the issuer DN field.

`key` specifies a value to get, for instance `"Name"` or *"Country"*.

hostname_match(*hostname*)
Return True if the Common Name (CN) field of the certificate matches a given `hostname`.

not_before()
Return the time the certificate becomes valid, as seconds since epoch.

not_after()
Return the time the certificate expires, as seconds since epoch.

allowed_usage(*usage_list*)
Return True if the certificates Key Usage extension contains all constraints given in `usage_list`. Also return True if the certificate doesn't have this extension. Example usage constraints are: `"DIGITAL_SIGNATURE"`, `"KEY_CERT_SIGN"`, `"CRL_SIGN"`.

verify(*intermediates=None*, *trusted=None*, *trusted_path=None*, *required_strength=0*, *hostname=None*, *reference_time=0 crls=None*)
Verify a certificate. Returns 0 if validation was successful, returns a positive error code if the validation was unsuccesful.

`intermediates` is a list of untrusted subauthorities.

`trusted` is a list of trusted root CAs.

The *trusted_path* refers to a directory where one or more trusted CA certificates are stored.

Set `required_strength` to indicate the minimum key and hash strength that is allowed. For instance setting to 80 allows 1024-bit RSA and SHA-1. Setting to 110 requires 2048-bit RSA and SHA-256 or higher. Set to zero to accept a default.

If `hostname` is given, it will be checked against the certificates CN field.

Set `reference_time` to be the time which the certificate chain is validated against. Use zero (default) to use the current system clock.

`crls` is a list of CRLs issued by either trusted or untrusted authorities.

classmethod validation_status(*error_code*)
Return an informative string associated with the verification return code.

is_revoked(*self*, *crl*)
Check if the certificate (`self`) is revoked on the given `crl`.

5.32.17 X509CRL

class botan.**X509CRL**(*filename=None*, *buf=None*)
Class representing an X.509 Certificate Revocation List.

A CRL in PEM or DER format can be loaded from a file, with the `filename` argument, or from a bytestring, with the `buf` argument.

CHAPTER

SIX

COMMAND LINE INTERFACE

6.1 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 will be printed. If a particular command is run with the `--help` option (like `botan <command> --help`) some information about the usage of the command is printed.

Starting in version 2.9, commands that take a passphrase (such as `gen_bcrypt` or `pkcs8`) will also accept the literal `-` to mean ask for the passphrase on the terminal. If supported by the operating system, echo will be disabled while reading the passphrase.

Most arguments that take a path to a file will also accept the literal `-` to mean the file content should be read from STDIN instead.

6.2 Hash Function

`hash --algo=SHA-256 --buf-size=4096 --no-fsname --format=hex *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. Unless the `--no-fsname` option is given, the filename is printed alongside the hash, in the style of tools such as `sha256sum`.

6.3 Password Hash

`gen_argon2 --mem=65536 --p=1 --t=1 password` Calculate the Argon2 password digest of *password*. *mem* is the amount of memory to use in Kb, *p* the parallelization parameter and *t* the number of iterations to use.

`check_argon2 password hash` Checks if the Argon2 hash of the passed *password* equals the passed *hash* value.

`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.

`pbkdf_tune --algo=Scrypt --max-mem=256 --output-len=32 --check *times` Tunes the PBKDF algorithm specified with `--algo=` for the given *times*.

6.4 HMAC

hmac --hash=SHA-256 --buf-size=4096 --no-fsname 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. Unless the `--no-fsname` option is given, the filename is printed alongside the HMAC value.

6.5 Encryption

encryption --buf-size=4096 --decrypt --mode= --key= --iv= --ad= Encrypt a given file with the specified *mode*. If `--decrypt` is provided the file is decrypted instead.

6.6 Public Key Cryptography

keygen --algo=RSA --params= --passphrase= --pbe= --pbe-millis=300 --provider= --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 bit length 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.

The default *pbe* algorithm is "PBES2(AES-256/CBC,SHA-256)".

With PBES2 scheme, you can select any CBC or GCM mode cipher which has an OID defined (such as 3DES, Camellia, SM4, Twofish or Serpent). However most other implementations support only AES or 3DES in CBC mode. You can also choose Scrypt instead of PBKDF2, by using "Scrypt" instead of the name of a hash function, for example "PBES2(AES-256/CBC,Scrypt)". Scrypt is also supported by some other implementations including OpenSSL.

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 --der-format --passphrase= --hash=SHA-256 --emsa= --provider= 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)

For ECDSA and DSA, the option `--der-format` outputs the signature as an ASN.1 encoded blob. Some other tools (including `openssl`) default to this format.

The signature is formatted for your screen using base64.

verify --der-format --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 --seed= --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.
- If *type=dsa* is used, p and q are generated by the algorithm specified in FIPS 186-4. If the `--seed` parameter is used, it allows to select the seed value, instead of one being randomly generated. If the seed does not in fact generate a valid DSA group, the command will fail.

dl_group_info --pem name Print raw Diffie-Hellman parameters (p,g) of the standardized 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 standardized curve *name*. If *pem* is set, the encoded domain is printed.

pk_encrypt --aead=AES-256/GCM rsa_pubkey datafile Encrypts `datafile` using the specified AEAD algorithm, under a key protected by the specified RSA public key.

pk_decrypt rsa_privkey datafile Decrypts a file encrypted with `pk_encrypt`. If the key is encrypted using a password, it will be prompted for on the terminal.

fingerprint --no-fsname --algo=SHA-256 *keys Calculate the public key fingerprint of the *keys*.

pk_workfactor --type=rsa bits Provide an estimate of the strength of a public key based on it's size. `--type=` can be "rsa", "dl" or "dl_exp".

6.7 X.509

gen_pkcs10 key CN --country= --organization= --ca --path-limit=1 --email= --dns= --ext-ku=
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= --path-limit=1 --days=365
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 --timeout=3000 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 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 successfully validated. The list of trusted CA certificates is passed with *ca_certs*, which is a list of one or more certificates.

trust_roots --dn --dn-only --display List the certificates in the system trust store.

6.8 TLS Server/Client

The `--policy=` argument of the TLS commands specifies the TLS policy to use. The policy can be any of the the strings "default", "suiteb_128", "suiteb_192", "bsi", "strict", or "all" to denote built-in policies, or it can name a file from which a policy description will be read.

tls_ciphers --policy=default --version=tls1.2 Prints the list of ciphersuites that will be offered under a particular policy/version.

tls_client host --port=443 --print-certs --policy=default --tls1.0 --tls1.1 --tls1.2 --skip
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 specified with *policy* and prints all certificates in the chain, if *print-certs* is passed. *next-protocols* is a comma separated list and specifies the protocols to advertise with Application-Layer Protocol Negotiation (ALPN).

tls_server cert key --port=443 --type=tcp --policy=default --dump-traces= --max-clients=0 -
Implements a testing TLS server, which allows TLS clients to connect and which echos any data that is sent to it. 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 specified with *policy*. *socket-id* is only available on FreeBSD and sets the *so_user_cookie* value of the used socket.

tls_http_server cert key --port=443 --policy=default --threads=0 --max-clients=0 --session-
Only available if Boost.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 specified with *policy*.

tls_proxy listen_port target_host target_port server_cert server_key--policy=default --thre
Only available if Boost.Asio support was enabled. Listens on a port and forwards all connects to a target server specified at `target_host` and `target_port`.

tls_client_hello --hex input Parse and print a TLS client hello message.

6.9 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. It can in reasonable time factor integers up to 110 bits or so.

gen_prime --count=1 bits Samples *count* primes with a length of *bits* bits.

mod_inverse n mod Calculates a modular inverse.

6.10 PSK Database

The PSK database commands are 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
```

6.11 Secret Sharing

Split a file into several shares.

tss_split M N data_file --id= --share-prefix=share --share-suffix=tss --hash=SHA-256 Split a file into `N` pieces any `M` of which suffices to recover the original input. The ID allows specifying a unique key ID which may be up to 16 bytes long, this ensures that shares can be uniquely matched. If not specified a random 16 byte value is used. A checksum can be appended to the data to help verify correct recovery, this can be disabled using `--hash=None`.

tss_recover *shares Recover some data split by `tss_split`. If insufficient number of shares are provided an error is printed.

6.12 Data Encoding/Decoding

base32_dec file Encode *file* to Base32.

base32_enc file Decode Base32 encoded *file*.

base58_enc --check file Encode *file* to Base58. If `--check` is provided Base58Check is used.

base58_dec --check file Decode Base58 encoded *file*. If `--check` is provided Base58Check is used.

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*.

6.13 Miscellaneous Commands

`version --full` Print the version number. If option `--full` is provided, additional details are printed.

`has_command cmd` Test if the command *cmd* is available.

`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`. This is similar to information provided by the `pkg-config` tool.

`cpuid` List available processor flags (AES-NI, SIMD extensions, . . .).

`cpu_clock --test-duration=500` Estimate the speed of the CPU cycle counter.

`asn1print --skip-context-specific --print-limit=4096 --bin-limit=2048 --max-depth=64 --pem`
Decode and print *file* with ASN.1 Basic Encoding Rules (BER). If flag `--pem` is used, or the filename ends in `.pem`, then PEM encoding is assumed. Otherwise the input is assumed to be binary DER/BER.

`http_get --redirects=1 --timeout=3000 url` Retrieve resource from the passed http *url*.

`speed --msec=500 --format=default --ecc-groups= --provider= --buf-size=1024 --clear-cpuid=`
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. The *buf-size* option allows testing the same algorithm on one or more input sizes, for example `speed --buf-size=136,1500 AES-128/GCM` tests the performance of GCM for small and large packet sizes. *format* can be "default", "table" or "json".

`timing_test test_type --test-data-file= --test-data-dir=src/tests/data/timing --warmup-runs`
Run various timing side channel tests.

`rng --format=hex --system --rdrand --auto --entropy --drbg --drbg-seed= *bytes`
Sample *bytes* random bytes from the specified random number generator. If *system* is set, the system RNG is used. If *rdrand* is set, the hardware RDRAND instruction is used. If *auto* is set, AutoSeeded_RNG is used, seeded with the system RNG if available or the global entropy source otherwise. If *entropy* is set, AutoSeeded_RNG is used, seeded with the global entropy source. If *drbg* is set, HMAC_DRBG is used seeded with *drbg-seed*.

`entropy --truncate-at=128 source` Sample a raw entropy source.

`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*.

`roughtime_check --raw-time chain-file` Parse and validate a Roughtime chain file.

`roughtime --raw-time --chain-file=roughtime-chain --max-chain-size=128 --check-local-clock=`
Retrieve time from a Roughtime server and store it in a chain file.

`uuid` Generate and print a random UUID.

`compress --type=gzip --level=6 --buf-size=8192 file` Compress a given file.

`decompress --buf-size=8192 file` Decompress a given compressed archive.

CHAPTER

SEVEN

DEPRECATED FEATURES

Certain functionality is deprecated and is likely to be removed in a future major release.

To help warn users, macros are used to annotate deprecated functions and headers. These warnings are enabled by default, but can be disabled by defining the macro `BOTAN_NO_DEPRECATED_WARNINGS` prior to including any Botan headers.

Warning: Not all of the functionality which is currently deprecated has an associated warning.

If you are using something which is currently deprecated and there doesn't seem to be an obvious alternative, contact the developers to explain your use case if you want to make sure your code continues to work.

7.1 TLS Protocol Deprecations

The following TLS protocol features are deprecated and will be removed in a future major release:

- Support for TLSv1.0/v1.1 and DTLS v1.0
- All support for DSA ciphersuites/certificates
- Support for point compression in TLS. This is supported in v1.2 but removed in v1.3. For simplicity it will be removed in v1.2 also.
- Support for using SHA-1 to sign TLS v1.2 ServerKeyExchange.
- All CBC mode ciphersuites. This includes all available 3DES and SEED ciphersuites. This implies also removing Encrypt-then-MAC extension.
- All ciphersuites using DH key exchange (DHE-DSS, DHE-RSA, DHE-PSK, anon DH)
- Support for renegotiation in TLS v1.2
- All ciphersuites using static RSA key exchange
- All anonymous (DH/ECDH) ciphersuites. This does not include PSK and ECDHE-PSK, which will be retained.
- SRP ciphersuites. This is implied by the removal of CBC mode, since all available SRP ciphersuites use CBC. To avoid use of obsolete ciphers, it would be better to instead perform a standard TLS negotiation, then a PAKE authentication within (and bound to) the TLS channel.
- OCB ciphersuites using 128-bit keys

7.2 Deprecated Functionality

This section lists cryptographic functionality which will be removed in a future major release.

- Block ciphers CAST-256, GOST 28147, Kasumi, MISTY1, DESX, XTEA, Noekeon
- Hash functions GOST 34.11-94, Tiger, MD4
- X9.42 KDF
- DLIES
- MCEIES
- CBC-MAC
- PBKDF1 key derivation
- GCM support for 64-bit tags
- Weak or rarely used ECC builtin groups including "secp160k1", "secp160r1", "secp160r2", "secp192k1", "secp224k1", "brainpool160r1", "brainpool192r1", "brainpool224r1", "brainpool320r1", "x962_p192v2", "x962_p192v3", "x962_p239v1", "x962_p239v2", "x962_p239v3".
- All built in MODP groups < 2048 bits
- Support for explicit ECC curve parameters and ImplicitCA encoded parameters in EC_Group and all users (including X.509 certificates and PKCS#8 private keys).
- All pre-created DSA groups
- All support for loading, generating or using RSA keys with a public exponent larger than 2**64-1
- All or nothing package transform (`package.h`)

7.3 Deprecated Headers

- The following headers and all functionality contained within them are outright deprecated, and will be removed entirely in a future major release. Most are either simply forwarding includes to another (still public) header, or contain functionality which is entirely deprecated. Consult the relevent file for more information. `basefilt.h`, `botan.h`, `buf_filt.h`, `cipher_filter.h`, `comp_filter.h`, `compiler.h`, `init.h`, `key_filt.h`, `lookup.h`, `sm2_enc.h`, `threefish.h`, `xmss_key_pair.h`
- The following headers have useful functionality but which we wish to hide from applications to allow easier library evolution. They will be made internal in a future major release, and will only be available to the library itself. In most cases, there is an alternative available. For example instead of using algorithm specific interfaces, use X::create to create the object dynamically.

  Block cipher headers (interact using BlockCipher interface): `aes.h`, `aria.h`, `blowfish.h`, `camellia.h`, `cascade.h`, `cast128.h`, `cast256.h`, `des.h`, `desx.h`, `gost_28147.h`, `idea.h`, `kasumi.h`, `lion.h`, `misty1.h`, `noekeon.h`, `seed.h`, `serpent.h`, `shacal2.h`, `sm4.h`, `threefish_512.h`, `twofish.h`, `xtea.h`,

  Hash function headers (interact using HashFunction interface): `adler32.h`, `blake2b.h`, `comb4p.h`, `crc24.h`, `crc32.h`, `gost_3411.h`, `keccak.h`, `md4.h`, `md5.h`, `par_hash.h`, `rmd160.h`, `sha160.h`, `sha2_32.h`, `sha2_64.h`, `sha3.h`, `shake.h`, `skein_512.h`, `sm3.h`, `streebog.h`, `tiger.h`, `whrlpool.h`,

  MAC headers: `cbc_mac.h`, `cmac.h`, `gmac.h`, `hmac.h`, `poly1305.h`, `siphash.h`, `x919_mac.h`,

  Stream cipher headers: `chacha.h`, `ctr.h`, `ofb.h`, `rc4.h`, `salsa20.h`, Cipher mode headers: `cbc.h`, `ccm.h`, `cfb.h`, `chacha20poly1305.h`, `eax.h`, `gcm.h`, `ocb.h`, `shake_cipher.h`, `siv.h`, `xts.h`,

KDF headers: `hkdf.h`, `kdf1.h`, `kdf1_iso18033.h`, `kdf2.h`, `prf_tls.h`, `prf_x942.h`, `sp800_108.h`, `sp800_56a.h`, `sp800_56c.h`,

PBKDF headers: `bcrypt_pbkdf.h`, `pbkdf1.h`, `pbkdf2.h`, `pgp_s2k.h`, `scrypt.h`,

Internal implementation headers - seemingly no reason for applications to use: `blinding.h`, `curve_gfp.h`, `curve_nistp.h`, `datastor.h`, `divide.h`, `eme.h`, `eme_pkcs.h`, `eme_raw.h`, `emsa.h`, `emsa1.h`, `emsa_pkcs1.h`, `emsa_raw.h`, `emsa_x931.h`, `gf2m_small_m.h`, `ghash.h`, `iso9796.h`, `keypair.h`, `mdx_hash.h`, `mode_pad.h`, `mul128.h`, `oaep.h`, `pbes2.h`, `polyn_gf2m.h`, `pow_mod.h`, `pssr.h`, `reducer.h`, `rfc6979.h`, `scan_name.h`, `stream_mode.h`, `tls_algos.h`, `tls_magic.h`, `xmss_common_ops.h`, `xmss_hash.h`, `xmss_index_registry.h`, `xmss_tools.h`,

Utility headers, nominally useful in applications but not a core part of the library API and most are just sufficient for what the library needs to implement other functionality. `atomic.h`, `bswap.h`, `charset.h`, `compiler.h`, `cpuid.h`, `http_util.h`, `loadstor.h`, `locking_allocator.h`, `parsing.h`, `rotate.h`, `secqueue.h`, `stl_compatibility.h`, `uuid.h`,

Merged into other headers: `alg_id.h`, `asn1_oid.h`, `asn1_str.h`, and `asn1_time.h` - use `asn1_obj.h`

7.4 Other API deprecations

- Directly accessing the member variables of types `calendar_point`, `ASN1_Attribute`, `AlgorithmIdentifier`, and `BER_Object`
- Using a default output length for "SHAKE-128" and "SHAKE-256". Instead, always specify the desired output length.
- Currently, for certain KDFs, if KDF interface is invoked with a requested output length larger than supported by the KDF, it returns instead a truncated key. In a future major release, instead if KDF is called with a length larger than it supports an exception will be thrown.
- 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 assessor 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.

7.5 Deprecated Build Targets

- Configuring a build (with `configure.py`) using Python2. In a future major release, Python3 will be required.
- Platform support for Google Native Client
- Support for PathScale and HP compilers

CHAPTER

EIGHT

DEVELOPMENT ROADMAP

8.1 Near Term Plans

Here is an outline for the development plans over the next 12-18 months, as of June 2019.

8.1.1 TLS Hardening/Testing

Leverage TLS-Attacker better, for example using custom workflows. Add interop testing with OpenSSL as part of CI. Improve fuzzer coverage.

8.1.2 Expose TLS at FFI layer

Exposing TLS to C would allow for many new applications to make use of Botan.

8.1.3 TLS v1.3

A complete implementation of TLS v1.3 is planned. DTLS v1.3 may or may not be supported as well.

8.2 Botan 3.x

Botan 3 is currently planned for release in 2021. Botan 2 will remain supported for several years past that, to allow plenty of time for applications to switch over.

This version will adopt C++17 and use new std types such as string_view, optional, and any, along with adopting memory span and guarded integer types. All deprecated features/APIs of 2.x (which notably includes TLS v1.0/v1.1 support) will be removed. Beyond explicitly deprecated functionality, there should be no breaking API changes in the transition to 3.x

Features currently targeted for Botan 3 include

- New post-quantum algorithms: especially a CCA2 secure encryption scheme and a lattice-based signature scheme are of interest.
- Password Authenticated Key Exchanges: one or more modern PAKEs (such as SPAKE2+ or OPAQUE) to replace SRP.
- Elliptic Curve Pairings: useful in many interesting protocols. BN-256 and BLS12-381 seem the most likely.
- New ASN.1 library

Some of these features may end being backported to Botan 2 as well.

CHAPTER

NINE

CREDITS

This is at least a partial credits-file of people that have contributed to botan. It is sorted by name and formatted to allow easy grepping and beautification by scripts. The fields are name (N), email (E), web-address (W), PGP key ID and fingerprint (P), description (D), snail-mail address (S), and Bitcoin address (B).

```
N: Alexander Bluhm
W: https://www.genua.de/
P: 1E3B BEA4 6C20 EA00 2FFC  DE4D C5F4 83AD DEE8 6380
D: improve support for OpenBSD
S: Kirchheim, Germany

N: Charles Brockman
W: http://www.securitygenetics.com/
D: documentation editing
S: Oregon, USA

N: Simon Cogliani
E: simon.cogliani@tanker.io
W: https://www.tanker.io/
P: EA73 D0AF 5A81 A61A 8931  C2CA C9AB F2E4 3820 4F25
D: Getting keystream of ChaCha
S: Paris, France

N: Martin Doering
E: doering@cdc.informatik.tu-darmstadt.de
D: GF(p) arithmetic

N: Olivier de Gaalon
D: SQLite encryption codec (src/contrib/sqlite)

N: Matthias Gierlings
E: matthias.gierlings@hackmanit.de
W: https://www.hackmanit.de/
P: 39E0 D270 19A4 B356 05D0 29AE 1BD3 49CF 744A 02FF
D: GMAC, Extended Hash-Based Signatures (XMSS)
S: Bochum, Germany

N: Matthew Gregan
D: Binary file I/O support, allocator fixes

N: Hany Greiss
D: Windows porting

N: Manuel Hartl
E: hartl@flexsecure.de
```

(continues on next page)

(continued from previous page)

```
W: http://www.flexsecure.de/
D: ECDSA, ECDH

N: Yves Jerschow
E: yves.jerschow@uni-duesseldorf.de
D: Optimizations for memory load/store and HMAC
D: Support for IPv4 addresses in X.509 alternative names
S: Germany

N: Matt Johnston
D: Allocator fixes and optimizations, decompressor fixes

N: Peter J. Jones
E: pjones@pmade.org
D: Bzip2 compression module
S: Colorado, USA

N: Justin Karneges
D: Qt support modules (mutexes and types), X.509 API design

N: Vojtech Kral
E: vojtech@kral.hk
D: LZMA compression module
S: Czech Republic

N: Matej Kenda
E: matej.kenda@topit.si
D: Locking in Algo_Registry for Windows OS
S: Slovenia

N: René Korthaus
E: r.korthaus@sirrix.com
W: https://www.sirrix.com
P: C196 FF9D 3DDC A5E7 F98C E745 9AD0 F9FA 587E 74D6
D: CI, ECGDSA, ECKCDSA
S: Bochum, Germany

N: Adam Langley
E: agl@imperialviolet.org
D: Curve25519

N: Jack Lloyd
E: jack@randombit.net
W: https://www.randombit.net/
P: 3F69 2E64 6D92 3BBE E7AE  9258 5C0F 96E8 4EC1 6D6B
B: 1DwxWb2J4vuX4vjsbzaCXW696rZfeamahz
D: Original designer/author, maintainer 2001-current
S: Vermont, USA

N: Joel Low
D: DLL symbol visibility and Windows DLL support in general
D: Threaded_Fork

N: Christoph Ludwig
E: ludwig@fh-worms.de
D: GP(p) arithmetic
```

(continues on next page)

(continued from previous page)

```
N: Vaclav Ovsik
E: vaclav.ovsik@i.cz
D: Perl XS module (src/contrib/perl-xs)

N: Luca Piccarreta
E: luca.piccarreta@gmail.com
D: x86/amd64 assembler, BigInt optimizations, Win32 mutex module
S: Italy

N: Daniel Seither
E: post@tiwoc.de
D: iOS support, improved Android support, improved MSVC support

N: Falko Strenzke
E: fstrenzke@cryptosource.de
W: http://www.cryptosource.de
D: McEliece, GF(p) arithmetic, CVC, Shanks-Tonelli algorithm
S: Darmstadt, Germany

N: Simon Warta
E: simon@kullo.net
W: https://www.kullo.net
D: Build system
S: Germany

N: Philipp Weber
E: p.weber@sirrix.com
W: https://sirrix.com/
D: KDF1-18033, ECIES
S: Saarland, Germany

N: Daniel Neus
E: d.neus@sirrix.com
W: https://sirrix.com/
D: CI, PKCS#11, RdSeed, BSI module policy
S: Bochum, Germany

N: Erwan Chaussy
D: Base32, Base64 matching Base32 implementation
S: France

N: Daniel Wyatt (on behalf of Ribose Inc)
E: daniel.wyatt@ribose.com
W: https://www.ribose.com/
D: SM3, Streebog, various minor contributions
```

CHAPTER

TEN

ABI STABILITY

Botan uses semantic versioning for the API; if API features are added the minor version increases, whereas if API compatibility 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/

CHAPTER

ELEVEN

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.

11.1 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.

Even though OpenSSL is also typically already installed, using `--with-openssl` by default is *not recommended*. OpenSSL is sometimes faster and sometimes slower than Botan, and the relative speeds vary depending on the algorithm and CPU.

11.2 Set Path to the System CA bundle

Most Unix/Linux systems maintain a list of trusted CA certificates at some well known path like `/etc/ssl/certs/ca-certificates.crt` or `/etc/ssl/cert.pem`. Unfortunately the exact path varies between systems. Use `--system-cert-bundle=PATH` to set this path. If the option is not used, `configure.py` tries a list of known locations.

11.3 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.

11.4 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 TWELVE

SECURITY ADVISORIES

If you think you have found a security bug in Botan please contact Jack Lloyd (jack@randombit.net). If you would like to encrypt your mail please use:

```
pub   rsa3072/57123B60 2015-03-23
      Key fingerprint = 4E60 C735 51AF 2188 DF0A  5A62 78E9 8043 5712 3B60
      uid         Jack Lloyd <jack@randombit.net>
```

This key can be found in the file `doc/pgpkey.txt` or online at https://keybase.io/jacklloyd and on most PGP keyservers.

12.1 2020

- 2020-07-05: Failure to enforce name constraints on alternative names

  The path validation algorithm enforced name constraints on the primary DN included in the certificate but failed to do so against alternative DNs which may be included in the subject alternative name. This would allow a corrupted sub-CA which was constrained by a name constraints extension in its own certificate to issue a certificate containing a prohibited DN. Until 2.15.0, there was no API to access these alternative name DNs so it is unlikely that any application would make incorrect access control decisions on the basis of the incorrect DN. Reported by Mario Korth of Ruhr-Universität Bochum.

  Introduced in 1.11.29, fixed in 2.15.0

- 2020-03-24: Side channel during CBC padding

  The CBC padding operations were not constant time and as a result would leak the length of the plaintext values which were being padded to an attacker running a side channel attack via shared resources such as cache or branch predictor. No information about the contents was leaked, but the length alone might be used to make inferences about the contents. This issue affects TLS CBC ciphersuites as well as CBC encryption using PKCS7 or other similar padding mechanisms. In all cases, the unpadding operations were already constant time and are not affected. Reported by Maximilian Blochberger of Universität Hamburg.

  Fixed in 2.14.0, all prior versions affected.

12.2 2018

- 2018-12-17 (CVE-2018-20187): Side channel during ECC key generation

  A timing side channel during ECC key generation could leak information about the high bits of the secret scalar. Such information allows an attacker to perform a brute force attack on the key somewhat more efficiently than they would otherwise. Found by Ján Jančár using ECTester.

  Introduced in 1.11.20, fixed in 2.8.0.

- 2018-06-13 (CVE-2018-12435): ECDSA side channel

  A side channel in the ECDSA signature operation could allow a local attacker to recover the secret key. Found by Keegan Ryan of NCC Group.

  Bug introduced in 2.5.0, fixed in 2.7.0. The 1.10 branch is not affected.

- 2018-04-10 (CVE-2018-9860): Memory overread in TLS CBC decryption

  An off by one error in TLS CBC decryption meant that for a particular malformed ciphertext, the receiver would miscompute a length field and HMAC exactly 64K bytes of data following the record buffer as if it was part of the message. This cannot be used to leak information since the MAC comparison will subsequently fail and the connection will be closed. However it might be used for denial of service. Found by OSS-Fuzz.

  Bug introduced in 1.11.32, fixed in 2.6.0

- 2018-03-29 (CVE-2018-9127): Invalid wildcard match

  RFC 6125 wildcard matching was incorrectly implemented, so that a wildcard certificate such as `b*.domain.com` would match any hosts `*b*.domain.com` instead of just server names beginning with `b`. The host and certificate would still have to be in the same domain name. Reported by Fabian Weißberg of Rohde and Schwarz Cybersecurity.

  Bug introduced in 2.2.0, fixed in 2.5.0

12.3 2017

- 2017-10-02 (CVE-2017-14737): Potential side channel using cache information

  In the Montgomery exponentiation code, a table of precomputed values is used. An attacker able to analyze which cache lines were accessed (perhaps via an active attack such as Prime+Probe) could recover information about the exponent. Identified in "CacheD: Identifying Cache-Based Timing Channels in Production Software" by Wang, Wang, Liu, Zhang, and Wu (Usenix Security 2017).

  Fixed in 1.10.17 and 2.3.0, all prior versions affected.

- 2017-07-16: Failure to fully zeroize memory before free

  The secure_allocator type attempts to zeroize memory before freeing it. Due to a error sometimes only a portion of the memory would be zeroed, because of a confusion between the number of elements vs the number of bytes that those elements use. So byte vectors would always be fully zeroed (since the two notions result in the same value), but for example with an array of 32-bit integers, only the first 1/4 of the elements would be zeroed before being deallocated. This may result in information leakage, if an attacker can access memory on the heap. Reported by Roman Pozlevich.

  Bug introduced in 1.11.10, fixed in 2.2.0

- 2017-04-04 (CVE-2017-2801): Incorrect comparison in X.509 DN strings

  Botan's implementation of X.509 name comparisons had a flaw which could result in an out of bound memory read while processing a specially formed DN. This could potentially be exploited for information disclosure or denial of service, or result in incorrect validation results. Found independently by Aleksandar Nikolic of Cisco Talos, and OSS-Fuzz automated fuzzing infrastructure.

Bug introduced in 1.6.0 or earlier, fixed in 2.1.0 and 1.10.16

- 2017-03-23 (CVE-2017-7252): Incorrect bcrypt computation

  Botan's implementation of bcrypt password hashing scheme truncated long passwords at 56 characters, instead of at bcrypt's standard 72 characters limit. Passwords with lengths between these two bounds could be cracked more easily than should be the case due to the final password bytes being ignored. Found and reported by Solar Designer.

  Bug introduced in 1.11.0, fixed in 2.1.0.

12.4 2016

- 2016-11-27 (CVE-2016-9132) Integer overflow in BER decoder

  While decoding BER length fields, an integer overflow could occur. This could occur while parsing untrusted inputs such as X.509 certificates. The overflow does not seem to lead to any obviously exploitable condition, but exploitation cannot be positively ruled out. Only 32-bit platforms are likely affected; to cause an overflow on 64-bit the parsed data would have to be many gigabytes. Bug found by Falko Strenzke, cryptosource GmbH.

  Fixed in 1.10.14 and 1.11.34, all prior versions affected.

- 2016-10-26 (CVE-2016-8871) OAEP side channel

  A side channel in OAEP decoding could be used to distinguish RSA ciphertexts that did or did not have a leading 0 byte. For an attacker capable of precisely measuring the time taken for OAEP decoding, this could be used as an oracle allowing decryption of arbitrary RSA ciphertexts. Remote exploitation seems difficult as OAEP decoding is always paired with RSA decryption, which takes substantially more (and variable) time, and so will tend to mask the timing channel. This attack does seems well within reach of a local attacker capable of a cache or branch predictor based side channel attack. Finding, analysis, and patch by Juraj Somorovsky.

  Introduced in 1.11.29, fixed in 1.11.33

- 2016-08-30 (CVE-2016-6878) Undefined behavior in Curve25519

  On systems without a native 128-bit integer type, the Curve25519 code invoked undefined behavior. This was known to produce incorrect results on 32-bit ARM when compiled by Clang.

  Introduced in 1.11.12, fixed in 1.11.31

- 2016-08-30 (CVE-2016-6879) Bad result from X509_Certificate::allowed_usage

  If allowed_usage was called with more than one Key_Usage set in the enum value, the function would return true if *any* of the allowed usages were set, instead of if *all* of the allowed usages are set. This could be used to bypass an application key usage check. Credit to Daniel Neus of Rohde & Schwarz Cybersecurity for finding this issue.

  Introduced in 1.11.0, fixed in 1.11.31

- 2016-03-17 (CVE-2016-2849): ECDSA side channel

  ECDSA (and DSA) signature algorithms perform a modular inverse on the signature nonce k. The modular inverse algorithm used had input dependent loops, and it is possible a side channel attack could recover sufficient information about the nonce to eventually recover the ECDSA secret key. Found by Sean Devlin.

  Introduced in 1.7.15, fixed in 1.10.13 and 1.11.29

- 2016-03-17 (CVE-2016-2850): Failure to enforce TLS policy

  TLS v1.2 allows negotiating which signature algorithms and hash functions each side is willing to accept. However received signatures were not actually checked against the specified policy. This had the effect of allowing a server to use an MD5 or SHA-1 signature, even though the default policy prohibits it. The same issue affected client cert authentication.

  The TLS client also failed to verify that the ECC curve the server chose to use was one which was acceptable by the client policy.

  Introduced in 1.11.0, fixed in 1.11.29

- 2016-02-01 (CVE-2016-2196): Overwrite in P-521 reduction

  The P-521 reduction function would overwrite zero to one word following the allocated block. This could potentially result in remote code execution or a crash. Found with AFL

  Introduced in 1.11.10, fixed in 1.11.27

- 2016-02-01 (CVE-2016-2195): Heap overflow on invalid ECC point

  The PointGFp constructor did not check that the affine coordinate arguments were less than the prime, but then in curve multiplication assumed that both arguments if multiplied would fit into an integer twice the size of the prime.

  The bigint_mul and bigint_sqr functions received the size of the output buffer, but only used it to dispatch to a faster algorithm in cases where there was sufficient output space to call an unrolled multiplication function.

  The result is a heap overflow accessible via ECC point decoding, which accepted untrusted inputs. This is likely exploitable for remote code execution.

  On systems which use the mlock pool allocator, it would allow an attacker to overwrite memory held in secure_vector objects. After this point the write will hit the guard page at the end of the mmap'ed region so it probably could not be used for code execution directly, but would allow overwriting adjacent key material.

  Found by Alex Gaynor fuzzing with AFL

  Introduced in 1.9.18, fixed in 1.11.27 and 1.10.11

- 2016-02-01 (CVE-2016-2194): Infinite loop in modular square root algorithm

  The ressol function implements the Tonelli-Shanks algorithm for finding square roots could be sent into a nearly infinite loop due to a misplaced conditional check. This could occur if a composite modulus is provided, as this algorithm is only defined for primes. This function is exposed to attacker controlled input via the OS2ECP function during ECC point decompression. Found by AFL

  Introduced in 1.7.15, fixed in 1.11.27 and 1.10.11

12.5 2015

- 2015-11-04: TLS certificate authentication bypass

  When the bugs affecting X.509 path validation were fixed in 1.11.22, a check in Credentials_Manager::verify_certificate_chain was accidentally removed which caused path validation failures not to be signaled to the TLS layer. So for affected versions, certificate authentication in TLS is bypassed. As a workaround, applications can override the call and implement the correct check. Reported by Florent Le Coz in GH #324

  Introduced in 1.11.22, fixed in 1.11.24

- 2015-10-26 (CVE-2015-7824): Padding oracle attack on TLS

  A padding oracle attack was possible against TLS CBC ciphersuites because if a certain length check on the packet fields failed, a different alert type than one used for message authentication failure would be returned to the sender. This check triggering would leak information about the value of the padding bytes and could be used to perform iterative decryption.

  As with most such oracle attacks, the danger depends on the underlying protocol - HTTP servers are particularly vulnerable. The current analysis suggests that to exploit it an attacker would first have to guess several bytes of plaintext, but again this is quite possible in many situations including HTTP.

  Found in a review by Sirrix AG and 3curity GmbH.

  Introduced in 1.11.0, fixed in 1.11.22

- 2015-10-26 (CVE-2015-7825): Infinite loop during certificate path validation

  When evaluating a certificate path, if a loop in the certificate chain was encountered (for instance where C1 certifies C2, which certifies C1) an infinite loop would occur eventually resulting in memory exhaustion. Found in a review by Sirrix AG and 3curity GmbH.

  Introduced in 1.11.6, fixed in 1.11.22

- 2015-10-26 (CVE-2015-7826): Acceptance of invalid certificate names

  RFC 6125 specifies how to match a X.509v3 certificate against a DNS name for application usage.

  Otherwise valid certificates using wildcards would be accepted as matching certain hostnames that should they should not according to RFC 6125. For example a certificate issued for `*.example.com` should match `foo.example.com` but not `example.com` or `bar.foo.example.com`. Previously Botan would accept such a certificate as also valid for `bar.foo.example.com`.

  RFC 6125 also requires that when matching a X.509 certificate against a DNS name, the CN entry is only compared if no subjectAlternativeName entry is available. Previously X509_Certificate::matches_dns_name would always check both names.

  Found in a review by Sirrix AG and 3curity GmbH.

  Introduced in 1.11.0, fixed in 1.11.22

- 2015-10-26 (CVE-2015-7827): PKCS #1 v1.5 decoding was not constant time

  During RSA decryption, how long decoding of PKCS #1 v1.5 padding took was input dependent. If these differences could be measured by an attacker, it could be used to mount a Bleichenbacher million-message attack. PKCS #1 v1.5 decoding has been rewritten to use a sequence of operations which do not contain any input-dependent indexes or jumps. Notations for checking constant time blocks with ctgrind (https://github.com/agl/ctgrind) were added to PKCS #1 decoding among other areas. Found in a review by Sirrix AG and 3curity GmbH.

  Fixed in 1.11.22 and 1.10.13. Affected all previous versions.

- 2015-08-03 (CVE-2015-5726): Crash in BER decoder

  The BER decoder would crash due to reading from offset 0 of an empty vector if it encountered a BIT STRING which did not contain any data at all. This can be used to easily crash applications reading untrusted ASN.1 data, but does not seem exploitable for code execution. Found with afl.

  Fixed in 1.11.19 and 1.10.10, affected all previous versions of 1.10 and 1.11

- 2015-08-03 (CVE-2015-5727): Excess memory allocation in BER decoder

  The BER decoder would allocate a fairly arbitrary amount of memory in a length field, even if there was no chance the read request would succeed. This might cause the process to run out of memory or invoke the OOM killer. Found with afl.

Fixed in 1.11.19 and 1.10.10, affected all previous versions of 1.10 and 1.11

12.6 2014

- 2014-04-10 (CVE-2014-9742): Insufficient randomness in Miller-Rabin primality check

  A bug in the Miller-Rabin primality test resulted in only a single random base being used instead of a sequence of such bases. This increased the probability that a non-prime would be accepted by is_prime or that a randomly generated prime might actually be composite. The probability of a random 1024 bit number being incorrectly classed as prime with a single base is around 2^-40. Reported by Jeff Marrison.

  Introduced in 1.8.3, fixed in 1.10.8 and 1.11.9

CHAPTER

THIRTEEN

SIDE CHANNELS

Many cryptographic systems can be easily 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.

13.1 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. The caller provides the maximum possible bit length of the exponent, and the exponent is zero-padded as required. For example, in a DSA signature with 256-bit q, the caller will specify a maximum length of exponent of 256 bits, even if the k that was generated was 250 bits. This avoids leaking the length of the exponent through the number of loop iterations. See monty_exp.cpp and monty.cpp

Karatsuba multiplication algorithm avoids any conditional branches; in cases where different operations must be performed it instead uses masked operations. See mp_karat.cpp for details.

The Montgomery reduction is written to run in constant time. The final reduction is handled with a masked subtraction. See mp_monty.cpp.

13.2 Barrett Reduction

The Barrett reduction code is written to avoid input dependent branches. The Barrett algorithm only works for inputs up to a certain size, and larger values fall back on a different (slower) division algorithm. This secondary algorithm is also const time, but the branch allows detecting when a value larger than 2^{2k} was reduced, where k is the word length of the modulus. This leaks only the size of the two values, and not anything else about their value.

13.3 RSA

Blinding is always used to protect private key operations (there is no way to turn it off). Both base blinding and exponent blinding are used.

For base blinding, 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 64) operations, a new starting point is chosen.

Exponent blinding uses new values for each signature, with 64 bit masks.

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.

13.4 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].

As a first step, the PKCS v1.5 decoding operation runs without any conditional jumps or indexes, with the only variance in runtime being based on the length of the public modulus, which is public information.

Preventing the attack 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.

13.5 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.

13.6 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.

13.7 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.

13.8 ECC scalar multiply

There are several different implementations of ECC scalar multiplications which depend on the API invoked. This include `PointGFp::operator*`, `EC_Group::blinded_base_point_multiply` and `EC_Group::blinded_var_point_multiply`.

The `PointGFp::operator*` implementation uses the Montgomery ladder, which is fairly resistant to side channels. However it leaks the size of the scalar, because the loop iterations are bounded by the scalar size. It should not be used in cases when the scalar is a secret.

Both `blinded_base_point_multiply` and `blinded_var_point_multiply` apply side channel countermeasures. The scalar is masked by a multiple of the group order (this is commonly called Coron's first countermeasure [CoronDpa]), currently the mask is an 80 bit random value.

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 non-zero 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].

The base point multiplication algorithm is a comb-like technique which precomputes `P^i,(2*P)^i,(3*P)^i` for all `i` in the range of valid scalars. This means the scalar multiplication involves only point additions and no doublings, which may help against attacks which rely on distinguishing between point doublings and point additions. The elements of the table are accessed by masked lookups, so as not to leak information about bits of the scalar via a cache side channel. However, whenever 3 sequential bits of the (masked) scalar are all 0, no operation is performed in that iteration of the loop. This exposes the scalar multiply to a cache-based side channel attack; scalar blinding is necessary to prevent this attack from leaking information about the scalar.

The variable point multiplication algorithm uses a fixed-window algorithm. Since this is normally invoked using untrusted points (eg during ECDH key exchange) it randomizes all inputs to prevent attacks which are based on chosen input points. The table of precomputed multiples is accessed using a masked lookup which should not leak information about the secret scalar to an attacker who can mount a cache-based side channel attack.

See point_gfp.cpp and point_mul.cpp

13.9 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.

13.10 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.

13.11 x25519

The x25519 code is independent of the main Weierstrass 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.

13.12 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.

13.13 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.

13.14 AES

Some x86, ARMv8 and POWER processors support AES instructions which are fast and are thought to be side channel silent. These instructions are used when available.

On CPUs which do not have hardware AES instructions but do support SIMD vectors with a byte shuffle (including x86's SSSE3, ARM's NEON and PowerPC AltiVec), a version of AES is implemented which is side channel silent. This implementation is based on code by Mike Hamburg [VectorAes], see aes_vperm.cpp.

On all other processors, a constant time bitsliced implementation is used. This is typically slower than the vector permute implementation, and additionally for best performance multiple blocks must be processed in parellel. So modes such as CTR, GCM or XTS are relatively fast, but others such as CBC encryption suffer.

13.15 GCM

On platforms that support a carryless multiply instruction (ARMv8 and recent x86), GCM is fast and constant time.

On all other platforms, GCM uses an algorithm based on precomputing all powers of H from 1 to 128. Then for every bit of the input a mask is formed which allows conditionally adding that power without leaking information via a cache side channel. There is also an SSSE3 variant of this algorithm which is somewhat faster on processors which have SSSE3 but no AES-NI instructions.

13.16 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.

13.17 Poly1305

The Poly1305 implementation does not have any secret lookups or conditionals. The code is based on the public domain version by Andrew Moon.

13.18 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.

13.19 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.

13.20 ChaCha20, Serpent, Threefish, . . .

Some algorithms including ChaCha, Salsa, Serpent and Threefish are 'naturally' silent to cache and timing side channels on all recent processors.

13.21 IDEA

IDEA encryption, decryption, and key schedule are implemented to take constant time regardless of their inputs.

13.22 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. However the implementations of Whirlpool and Streebog use table lookups and probably can be attacked by side channels.

13.23 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.

13.24 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. If possible an OS provided routine (such as `RtlSecureZeroMemory` or `explicit_bzero`) is used.

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.

13.25 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 locked by processes without extra credentials.

If available, Botan uses such a region for storing key material. A page-aligned block of memory is allocated and locked, then the memory is scrubbed before freeing. 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.

13.26 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.

There is also a test utility built into the command line util, *timing_test*, which runs an operation on several different inputs many times in order to detect simple timing differences. The output can be processed using the Mona timing report library (https://github.com/seecurity/mona-timing-report). To run a timing report (here for example pow_mod):

```
$ ./botan timing_test pow_mod > pow_mod.raw
```

This must be run from a checkout of the source, or otherwise `--test-data-dir=` must be used to point to the expected input files.

Build and run the Mona report as:

```
$ git clone https://github.com/seecurity/mona-timing-report.git
$ cd mona-timing-report
$ ant
$ java -jar ReportingTool.jar --lowerBound=0.4 --upperBound=0.5 --inputFile=pow_mod.
↪raw --name=PowMod
```

This will produce plots and an HTML file in subdirectory starting with `reports_` followed by a representation of the current date and time.

13.27 References

[Aes256Sc] Neve, Tiri "On the complexity of side-channel attacks on AES-256" (https://eprint.iacr.org/2007/318.pdf)

[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

FOURTEEN

DEVELOPER REFERENCE

This section contains information useful to people making contributions to the library

14.1 Notes for New Contributors

14.1.1 Source Code Layout

Under `src` there are directories

- `lib` is the library itself, more on that below
- `cli` is the command line application `botan`
- `tests` contain what you would expect. Input files go under `tests/data`.
- `build-data` contains files read by the configure script. For example `build-data/cc/gcc.txt` describes various gcc options.
- `scripts` contains misc scripts: install, distribution, various codegen things. Scripts controlling CI go under `scripts/ci`.
- `configs` contains configuration files for emacs, astyle, pylint, etc
- `python/botan2.py` is the Python ctypes wrapper

14.1.2 Library Layout

- `base` defines some high level types
- `utils` contains various utility functions and types
- `codec` has hex, base64
- `block` contains the block cipher implementations
- `modes` contains block cipher modes (CBC, GCM, etc)
- `stream` contains the stream ciphers
- `hash` contains the hash function implementations
- `passhash` contains password hashing algorithms for authentication
- `kdf` contains the key derivation functions
- `mac` contains the message authentication codes
- `pbkdf` contains password hashing algorithms for key derivation

- `math` is the big integer math library. It is divided into three parts: `mp` which are the low level algorithms; `bigint` which is a C++ wrapper around `mp`, and `numbertheory` which contains higher level algorithms like primality testing and exponentiation
- `pubkey` contains the public key algorithms
- `pk_pad` contains padding schemes for public key algorithms
- `rng` contains the random number generators
- `entropy` has various entropy sources used by some of the RNGs
- `asn1` is the DER encoder/decoder
- `x509` is X.509 certificates, PKCS #10 requests, OCSP
- `tls` contains the TLS implementation
- `filters` is a filter/pipe API for data transforms
- `compression` has the compression wrappers (zlib, bzip2, lzma)
- `ffi` is the C99 API
- `prov` contains bindings to external libraries like OpenSSL and PKCS #11
- `misc` contains odds and ends: format preserving encryption, SRP, threshold secret sharing, all or nothing transform, and others

14.1.3 Sending patches

All contributions should be submitted as pull requests via GitHub (https://github.com/randombit/botan). If you are planning a large change email the mailing list or open a discussion ticket on github before starting out to make sure you are on the right path. And once you have something written, free to open a [WIP] PR for early review and comment.

If possible please sign your git commits using a PGP key. See https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work for instructions on how to set this up.

Depending on what your change is, your PR should probably also include an update to `news.rst` with a note explaining the change. If your change is a simple bug fix, a one sentence description is perhaps sufficient. If there is an existing ticket on GitHub with discussion or other information, reference it in your change note as 'GH #000'.

Update `doc/credits.txt` with your information so people know what you did!

If you are interested in contributing but don't know where to start check out `doc/dev_ref/todo.rst` for some ideas - these are changes we would almost certainly accept once they've passed code review.

Also, try building and testing it on whatever hardware you have handy, especially unusual platforms, or using C++ compilers other than the regularly tested GCC, Clang, and Visual Studio.

14.1.4 FFI Additions

If adding a new function declaration to `ffi.h`, the same PR must also add the same declaration in the Python binding `botan2.py`, in addition the new API functionality must be exposed to Python and a test written in Python.

14.1.5 Git Usage

Do *NOT* merge `master` into your topic branch, this creates needless commits and noise in history. Instead, as needed, rebase your branch against master (`git rebase -i master`) and force push the branch to update the PR. If the GitHub PR page does not report any merge conflicts and nobody asks you to rebase, you don't need to rebase.

Try to keep your history clean and use rebase to squash your commits as needed. If your diff is less than roughly 100 lines, it should probably be a single commit. Only split commits as needed to help with review/understanding of the change.

14.1.6 Python

Scripts should be in Python whenever possible.

For configure.py (and helper scripts install.py, cleanup.py and build_docs.py) the target is stock (no modules outside the standard library) CPython 2.7 plus latest CPython 3.x. Support for CPython 2.6, PyPy, etc is great when viable (in the sense of not causing problems for 2.7 or 3.x, and not requiring huge blocks of version dependent code). As running this program successfully is required for a working build, making it as portable as possible is considered key.

The python wrapper botan2.py targets CPython 2.7, 3.x, and latest PyPy. Note that a single file is used to avoid dealing with any of Python's various crazy module distribution issues.

For random scripts not typically run by an end-user (codegen, visualization, and so on) there isn't any need to worry about 2.6 and even just running under Python2 xor Python3 is acceptable if needed. Here it's fine to depend on any useful modules such as graphviz or matplotlib, regardless if it is available from a stock CPython install. Since Python2 is now end of life, prefer Python3 for new scripts of this sort.

14.1.7 Build Tools and Hints

If you don't already use it for all your C/C++ development, install `ccache` (or on Windows, `sccache`) right now, and configure a large cache on a fast disk. It allows for very quick rebuilds by caching the compiler output.

Use `--enable-sanitizers=` flag to enable various sanitizer checks. Supported values including "address" and "undefined" for GCC and Clang. GCC also supports "iterator" (checked iterators), and Clang supports "memory" (MSan) and "coverage" (for fuzzing).

On Linux if you have the `lcov` and `gcov` tools installed, then running `./src/scripts/ci_build.py coverage` will produce a coverage enabled build, run the tests, test the fuzzers against a corpus, and produce an HTML report of total coverage. This coverage build requires the development headers for zlib, bzip2, liblzma, OpenSSL, TrouSerS (libtspi), and Sqlite3.

14.1.8 Copyright Notice

At the top of any new file add a comment with a copyright and a reference to the license, for example:

```
/*
* (C) 20xx Copyright Holder
* Botan is released under the Simplified BSD License (see license.txt)
*/
```

If you are making a substantial or non-trivial change to an existing file, add or update your own copyright statement at the top of each file.

14.1.9 Style Conventions

When writing your code remember the need for it to be easily understood by reviewers and auditors, both at the time of the patch submission and in the future.

Avoid complicated template metaprogramming where possible. It has its places but should be used judiciously.

When designing a new API (for use either by library users or just internally) try writing out the calling code first. That is, write out some code calling your idealized API, then just implement that API. This can often help avoid cut-and-paste by creating the correct abstractions needed to solve the problem at hand.

The C++11 `auto` keyword is very convenient but only use it when the type truly is obvious (considering also the potential for unexpected integer conversions and the like, such as an apparent uint8_t being promoted to an int).

If a variable is defined and not modified, declare it `const`. Some exception for very short-lived variables, but generally speaking being able to read the declaration and know it will not be modified is useful.

Use `override` annotations whenever overriding a virtual function. If introducing a new type that is not intended for derivation, mark it `final`.

Avoid explicit `delete` - use RAII.

Use `m_` prefix on all member variables.

For formatting, there are configs for emacs and astyle in `src/configs`. No tabs, and remove trailing whitespace.

Prefer using braces on both sides of if/else blocks, even if only using a single statement. The current code doesn't always do this.

Avoid `using namespace` declarations, even inside of single functions. One allowed exception is `using namespace std::placeholders` in functions which use `std::bind`. (But, don't use `std::bind` - use a lambda instead).

Use `::` to explicitly refer to the global namespace (eg, when calling an OS or external library function like `::select` or `::sqlite3_open`).

14.1.10 Use of External Dependencies

Compiler Dependencies

The library should always be as functional as possible when compiled with just C++11. However, feel free to use the full C++11 language. No accomodations are made for compilers that are incomplete or buggy.

Use of compiler extensions is fine whenever appropriate; this is typically restricted to a single file or an internal header. Compiler extensions used currently include native uint128_t, SIMD intrinsics, inline asm syntax and so on, so there are some existing examples of appropriate use.

Generally intrinsics or inline asm is preferred over bare assembly to avoid calling convention issues among different platforms; the improvement in maintainability is seen as worth any potential performance tradeoff. One risk with intrinsics is that the compiler might rewrite your clever const-time SIMD into something with a conditional jump, but code intended to be const-time should in any case be annotated so it can be checked at runtime with tools.

Operating System Dependencies

If you're adding a small OS dependency in some larger piece of code, try to contain the actual non-portable operations to utils/os_utils.* and then call them from there.

As a policy, operating systems which are not supported by their original vendor are not supported by Botan either. Patches that complicate the code in order to support obsolete operating systems will likely be rejected. In writing OS specific code, feel free to assume roughly POSIX 2008, or for Windows, Windows 8 /Server 2012 (which are as of this writing the oldest versions still supported by Microsoft).

Some operating systems, such as OpenBSD, only support the latest release. For such cases, it's acceptable to add code that requires APIs added in the most recent release of that OS as soon as the release is available.

Library Dependencies

Any external library dependency - even optional ones - is met with as one PR submitter put it "great skepticism".

At every API boundary there is potential for confusion that does not exist when the call stack is all contained within the boundary. So the additional API really needs to pull its weight. For example a simple text parser or such which can be trivially implemented is not really for consideration. As a rough idea of the bar, equate the viewed cost of an external dependency as at least 1000 additional lines of code in the library. That is, if the library really does need this functionality, and it can be done in the library for less than that, then it makes sense to just write the code. Yup.

Currently the (optional) external dependencies of the library are OpenSSL (for access to fast and side channel hardened RSA, ECDSA, AES), some compression libraries (zlib, bzip2, lzma), sqlite3 database, Trousers (TPM integration), plus various operating system utilities like basic filesystem operations. These provide major pieces of functionality which seem worth the trouble of maintaining an integration with.

At this point the most plausible examples of an appropriate new external dependency are all deeper integrations with system level cryptographic systems (CommonCrypto, CryptoAPI, /dev/crypto, iOS keychain, TPM 2.0, etc)

14.2 Understanding configure.py

Botan's build is handled with a custom Python script, `configure.py`. This document tries to explain how configure works.

Note: You only need to read this if you are modifying the library, or debugging some problem with your build. For how to use it, see *Building The Library*.

14.2.1 Build Structure

Modules are a group of related source and header files, which can be individually enabled or disabled at build time. Modules can depend on other modules; if a dependency is not available then the module itself is also removed from the list. Examples of modules in the existing codebase are `asn1` and `x509`, Since `x509` depends on (among other things) `asn1`, disabling `asn1` will also disable `x509`.

Most modules define one or more macros, which application code can use to detect the modules presence or absence. The value of each macro is a datestamp, in the form YYYYMMDD which indicates the last time this module changed in a way that would be visible to an application. For example if a class gains a new function, the datestamp should be incremented. That allows applications to detect if the new feature is available.

14.2.2 What `configure.py` does

First, all command line options are parsed.

Then all of the files giving information about target CPUs, compilers, etc are parsed and sanity checked.

In `calculate_cc_min_version` the compiler version is detected using the preprocessor.

Then in `check_compiler_arch` the target architecture are detected, again using the preprocessor.

Now that the target is identified and options have been parsed, the modules to include into the artifact are picked, in `ModulesChooser`.

In `create_template_vars`, a dictionary of variables is created which describe different aspects of the build. These are serialized to `build/build_config.json`.

Up until this point no changes have been made on disk. This occurs in `do_io_for_build`. Build output directories are created, and header files are linked into `build/include/botan`. Templates are processed to create the Makefile, `build.h` and other artifacts.

14.2.3 When Modifying `configure.py`

For now, any changes to `configure.py` must work under both CPython 2.7 and CPython 3.x. In a future major release, support for CPython2 will be dropped, but until then if making modifications verify the code works as expected on both versions.

Run `./src/scripts/ci_build.py lint` to run Pylint checks after any change.

14.2.4 Template Language

Various output files are generated by processing input files using a simple template language. All input files are stored in `src/build-data` and use the suffix `.in`. Anything not recognized as a template command is passed through to the output unmodified. The template elements are:

- Variable substitution, `%{variable_name}`. The configure script creates many variables for various purposes, this allows getting their value within the output. If a variable is not defined, an error occurs.

  If a variable reference ends with `|upper`, the value is uppercased before being inserted into the template output.

- Iteration, `%{for variable} block %{endfor}`. This iterates over a list and repeats the block as many times as it is included. Variables within the block are expanded. The two template elements `%{for ...}` and `%{endfor}` must appear on lines with no text before or after.

- Conditional inclusion, `%{if variable} block %{endif}`. If the variable named is defined and true (in the Python sense of the word; if the variable is empty or zero it is considered false), then the block will be included and any variables expanded. As with the for loop syntax, both the start and end of the conditional must be on their own lines with no additional text.

14.2.5 Adding a new module

Create a directory in the appropriate place and create a info.txt file.

14.2.6 Syntax of `info.txt`

Warning: The syntax described here is documented to make it easier to use and understand, but it is not considered part of the public API contract. That is, the developers are allowed to change the syntax at any time on the assumption that all users are contained within the library itself. If that happens this document will be updated.

Modules and files describing information about the system use the same parser and have common syntactical elements.

Comments begin with '#' and continue to end of line.

There are three main types: maps, lists, and variables.

A map has a syntax like:

```
<MAP_NAME>
NAME1 -> VALUE1
NAME2 -> VALUE2
...
</MAP_NAME>
```

The interpretation of the names and values will depend on the map's name and what type of file is being parsed.

A list has similar syntax, it just doesn't have values:

```
<LIST_NAME>
ELEM1
ELEM2
...
</LIST_NAME>
```

Lastly there are single value variables like:

```
VAR1 SomeValue
VAR2 "Quotes Can Be Used (And will be stripped out)"
VAR3 42
```

Variables can have string, integer or boolean values. Boolean values are specified with 'yes' or 'no'.

14.2.7 Module Syntax

The info.txt files have the following elements. Not all are required; a minimal file for a module with no dependencies might just contain a macro define.

Lists:

- comment and warning provides block-comments which are displayed to the user at build time.
- requires is a list of module dependencies. An os_features can be specified as a condition for needing the dependency by writing it before the module name and separated by a ?, e.g. rtlgenrandom? dyn_load.
- header:internal is the list of headers (from the current module) which are internal-only.
- header:public is a the list of headers (from the current module) which should be exported for public use. If neither header:internal nor header:public are used then all headers in the current directory are assumed public.

  Note: If you omit a header from both internal and public lists, it will be ignored.

- header:external is used when naming headers which are included in the source tree but might be replaced by an external version. This is used for the PKCS11 headers.
- arch is a list of architectures this module may be used on.
- isa lists ISA features which must be enabled to use this module. Can be proceeded by an arch name followed by a : if it is only needed on a specific architecture, e.g. x86_64:ssse3.
- cc is a list of compilers which can be used with this module. If the compiler name is suffixed with a version (like "gcc:5.0") then only compilers with that minimum version can use the module.
- os_features is a list of OS features which are required in order to use this module. Each line can specify one or more features combined with ','. Alternatives can be specified on additional lines.

Maps:

- defines is a map from macros to datestamps. These macros will be defined in the generated build.h.
- libs specifies additional libraries which should be linked if this module is included. It maps from the OS name to a list of libraries (comma seperated).
- frameworks is a macOS/iOS specific feature which maps from an OS name to a framework.

Variables:

- load_on Can take on values never, always, auto, dep or vendor. TODO describe the behavior of these
- endian Required endian for the module (any (default), little, big)

An example:

```
# Disable this by default
load_on never

<isa>
sse2
</isa>

<defines>
```

(continues on next page)

(continued from previous page)

```
DEFINE1 -> 20180104
DEFINE2 -> 20190301
</defines>

<comment>
I have eaten
the plums
that were in
the icebox
</comment>

<warning>
There are no more plums
</warning>

<header:public>
header1.h
</header:public>

<header:internal>
header_helper.h
whatever.h
</header:internal>

<arch>
x86_64
</arch>

<cc>
gcc:4.9 # gcc 4.8 doesn't work for <reasons>
clang
</cc>

# Can work with POSIX+getentropy or Win32
<os_features>
posix1,getentropy
win32
</os_features>

<frameworks>
macos -> FramyMcFramerson
</frameworks>

<libs>
qnx -> foo,bar,baz
solaris -> socket
</libs>
```

14.2.8 Supporting a new CPU type

CPU information is stored in `src/build-data/arch`.

There is also a file `src/build-data/detect_arch.cpp` which is used for build-time architecture detection using the compiler preprocessor. Supporting this is optional but recommended.

Lists:

- `aliases` is a list of alternative names for the CPU architecture.
- `isa_extensions` is a list of possible ISA extensions that can be used on this architecture. For example x86-64 has extensions "sse2", "ssse3", "avx2", "aesni", . . .

Variables:

- `endian` if defined should be "little" or "big". This can also be controlled or overridden at build time.
- `family` can specify a family group for several related architecture. For example both x86_32 and x86_64 use `family` of "x86".
- `wordsize` is the default wordsize, which controls the size of limbs in the multi precision integers. If not set, defaults to 32.

14.2.9 Supporting a new compiler

Compiler information is stored in `src/build-data/cc`. Looking over those files will probably help understanding, especially the ones for GCC and Clang which are most complete.

In addition to the info file, for compilers there is a file `src/build-data/detect_version.cpp`. The `configure.py` script runs the preprocessor over this file to attempt to detect the compiler version. Supporting this is not strictly necessary.

Maps:

- `binary_link_commands` gives the command to use to run the linker, it maps from operating system name to the command to use. It uses the entry "default" for any OS not otherwise listed.
- `cpu_flags_no_debug` unused, will be removed
- `cpu_flags` used to emit CPU specific flags, for example LLVM bitcode target uses `-emit-llvm` flag. Rarely needed.
- `isa_flags` maps from CPU extensions (like NEON or AES-NI) to compiler flags which enable that extension. These have the same name as the ISA flags listed in the architecture files.
- `lib_flags` has a single possible entry "debug" which if set maps to additional flags to pass when building a debug library. Rarely needed.
- `mach_abi_linking` specifies flags to enable when building and linking on a particular CPU. This is usually flags that modify ABI. There is a special syntax supported here "all!os1,arch1,os2,arch2" which allows setting ABI flags which are used for all but the named operating systems and/or architectures.
- `sanitizers` is a map of sanitizers the compiler supports. It must include "default" which is a list of sanitizers to include by default when sanitizers are requested. The other keys should map to compiler flags.
- `so_link_commands` maps from operating system to the command to use to build a shared object.

Variables:

- `binary_name` the default name of the compiler binary.
- `linker_name` the name of the linker to use with this compiler.

- `macro_name` a macro of the for `BOTAN_BUILD_COMPILER_IS_XXX` will be defined.
- `output_to_object` (default "-o") gives the compiler option used to name the output object.
- `output_to_exe` (default "-o") gives the compiler option used to name the output object.
- `add_include_dir_option` (default "-I") gives the compiler option used to specify an additional include dir.
- `add_lib_dir_option` (default "-L") gives the compiler option used to specify an additional library dir.
- `add_sysroot_option` gives the compiler option used to specify the sysroot.
- `add_lib_option` (default "-l%s") gives the compiler option to link in a library. `%s` will be replaced with the library name.
- `add_framework_option` (default "-framework") gives the compiler option to add a macOS framework.
- `preproc_flags` (default "-E") gives the compiler option used to run the preprocessor.
- `compile_flags` (default "-c") gives the compiler option used to compile a file.
- `debug_info_flags` (default "-g") gives the compiler option used to enable debug info.
- `optimization_flags` gives the compiler optimization flags to use.
- `size_optimization_flags` gives compiler optimization flags to use when compiling for size. If not set then `--optimize-for-size` will use the default optimization flags.
- `sanitizer_optimization_flags` gives compiler optimization flags to use when building with sanitizers.
- `coverage_flags` gives the compiler flags to use when generating coverage information.
- `stack_protector_flags` gives compiler flags to enable stack overflow checking.
- `shared_flags` gives compiler flags to use when generation shared libraries.
- `lang_flags` gives compiler flags used to enable the required version of C++.
- `warning_flags` gives warning flags to enable.
- `maintainer_warning_flags` gives extra warning flags to enable during maintainer mode builds.
- `visibility_build_flags` gives compiler flags to control symbol visibility when generation shared libraries.
- `visibility_attribute` gives the attribute to use in the `BOTAN_DLL` macro to specify visibility when generation shared libraries.
- `ar_command` gives the command to build static libraries
- `ar_options` gives the options to pass to `ar_command`, if not set here takes this from the OS specific information.
- `ar_output_to` gives the flag to pass to `ar_command` to specify where to output the static library.
- `werror_flags` gives the complier flags to treat warnings as errors.

14.2.10 Supporting a new OS

Operating system information is stored in `src/build-data/os`.

Lists:

- `aliases` is a list of alternative names which will be accepted
- `target_features` is a list of target specific OS features. Some of these are supported by many OSes (for example "posix1") others are specific to just one or two OSes (such as "getauxval"). Adding a value here causes a new macro `BOTAN_TARGET_OS_HAS_XXX` to be defined at build time. Use `configure.py --list-os-features` to list the currently defined OS features.
- `feature_macros` is a list of macros to define.

Variables:

- `ar_command` gives the command to build static libraries
- `ar_options` gives the options to pass to `ar_command`
- `ar_output_to` gives the flag to pass to `ar_command` to specify where to output the static library.
- `bin_dir` (default "bin") specifies where binaries should be installed, relative to install_root.
- `cli_exe_name` (default "botan") specifies the name of the command line utility.
- `default_compiler` specifies the default compiler to use for this OS.
- `doc_dir` (default "doc") specifies where documentation should be installed, relative to install_root
- `header_dir` (default "include") specifies where include files should be installed, relative to install_root
- `install_root` (default "/usr/local") specifies where to install by default.
- `lib_dir` (default "lib") specifies where library should be installed, relative to install_root.
- `lib_prefix` (default "lib") prefix to add to the library name
- `library_name`
- `man_dir` specifies where man files should be installed, relative to install_root
- `obj_suffix` (default "o") specifies the suffix used for object files
- `program_suffix` (default "") specifies the suffix used for executables
- `shared_lib_symlinks` (default "yes) specifies if symbolic names should be created from the base and patch soname to the library name.
- `soname_pattern_abi`
- `soname_pattern_base`
- `soname_pattern_patch`
- `soname_suffix` file extension to use for shared library if `soname_pattern_base` is not specified.
- `static_suffix` (default "a") file extension to use for static library.
- `use_stack_protector` (default "true") specify if by default stack smashing protections should be enabled.
- `uses_pkg_config` (default "yes") specify if by default a pkg-config file should be created.

14.3 Test Framework

Botan uses a custom-built test framework. Some portions of it are quite similar to assertion-based test frameworks such as Catch or Gtest, but it also includes many features which are well suited for testing cryptographic algorithms.

The intent is that the test framework and the test suite evolve symbiotically; as a general rule of thumb if a new function would make the implementation of just two distinct tests simpler, it is worth adding to the framework on the assumption it will prove useful again. Feel free to propose changes to the test system.

When writing a new test, there are three key classes that are used, namely `Test`, `Test::Result`, and `Text_Based_Test`. A `Test` (or `Test_Based_Test`) runs and returns one or more `Test::Result`.

14.3.1 Namespaces in Test

The test code lives in a distinct namespace (`Botan_Tests`) and all code in the tests which calls into the library should use the namespace prefix `Botan::` rather than a `using namespace` declaration. This makes it easier to see where the test is actually invoking the library, and makes it easier to reuse test code for applications.

14.3.2 Test Data

The test framework is heavily data driven. As of this writing, there is about 1 Mib of test code and 17 MiB of test data. For most (though certainly not all) tests, it is better to add a data file representing the input and outputs, and run the tests over it. Data driven tests make adding or editing tests easier, for example by writing scripts which produce new test data and output it in the expected format.

14.3.3 Test

class Test

virtual std::vector<*Test*::*Result*> **run**() = 0

This is the key function of a `Test`: it executes and returns a list of results. Almost all other functions on `Test` are static functions which just serve as helper functions for `run`.

static std::string **read_data_file**(**const** std::string &*path*)

Return the contents of a data file and return it as a string.

static std::vector<uint8_t> **read_binary_data_file**(**const** std::string &*path*)

Return the contents of a data file and return it as a vector of bytes.

static std::string **data_file**(**const** std::string &*what*)

An alternative to `read_data_file` and `read_binary_file`, use only as a last result, typically for library APIs which themselves accept a filename rather than a data blob.

static bool **run_long_tests**() **const**

Returns true if the user gave option `--run-long-tests`. Use this to gate particularly time-intensive tests.

static Botan::RandomNumberGenerator &**rng**()

Returns a reference to a fast, not cryptographically secure random number generator. It is deterministicly seeded with the seed logged by the test runner, so it is possible to reproduce results in "random" tests.

Tests are registered using the macro `BOTAN_REGISTER_TEST` which takes 2 arguments: the name of the test and the name of the test class. For example given a `Test` instance named `MyTest`, use:

```
BOTAN_REGISTER_TEST("mytest", MyTest);
```

All test names should contain only lowercase letters, numbers, and underscore.

14.3.4 Test::Result

class `Test::`**`Result`**

A `Test::Result` records one or more tests on a particular topic (say "AES-128/CBC" or "ASN.1 date parsing"). Most of the test functions return true or false if the test was successful or not; this allows performing conditional blocks as a result of earlier tests:

```
if(result.test_eq("first value", produced, expected))
   {
   // further tests that rely on the initial test being correct
   }
```

Only the most commonly used functions on `Test::Result` are documented here, see the header `tests.h` for more.

`Test::`**`Result`**(**const** std::string &*who*)

Create a test report on a particular topic. This will be displayed in the test results.

bool **test_success**()

Report a test that was successful.

bool **test_success**(**const** std::string &*note*)

Report a test that was successful, including some comment.

bool **test_failure**(**const** std::string &*err*)

Report a test failure of some kind. The error string will be logged.

bool **test_failure**(**const** std::string &*what*, **const** std::string &*error*)

Report a test failure of some kind, with a description of what failed and what the error was.

void **test_failure**(**const** std::string &*what*, **const** uint8_t *buf*[], size_t *buf_len*)

Report a test failure due to some particular input, which is provided as arguments. Normally this is only used if the test was using some randomized input which unexpectedly failed, since if the input is hardcoded or from a file it is easier to just reference the test number.

bool **test_eq**(**const** std::string &*what*, **const** std::string &*produced*, **const** std::string &*expected*)

Compare to strings for equality.

bool **test_ne**(**const** std::string &*what*, **const** std::string &*produced*, **const** std::string &*expected*)

Compare to strings for non-equality.

bool **test_eq**(**const** char **producer*, **const** std::string &*what*, **const** uint8_t *produced*[], size_t *produced_len*, **const** uint8_t *expected*[], size_t *expected_len*)

Compare two arrays for equality.

bool **test_ne**(**const** char **producer*, **const** std::string &*what*, **const** uint8_t *produced*[], size_t *produced_len*, **const** uint8_t *expected*[], size_t *expected_len*)

Compare two arrays for non-equality.

bool **test_eq**(**const** std::string &*producer*, **const** std::string &*what*, **const** std::vector<uint8_t> &*produced*, **const** std::vector<uint8_t> &*expected*)

Compare two vectors for equality.

bool **test_ne**(**const** std::string &*producer*, **const** std::string &*what*, **const** std::vector<uint8_t> &*produced*, **const** std::vector<uint8_t> &*expected*)

Compare two vectors for non-equality.

bool **confirm**(**const** std::string &*what*, bool *expr*)
: Test that some expression evaluates to `true`.

template<typename **T**>
bool **test_not_null**(**const** std::string &*what*, *T* **ptr*)
: Verify that the pointer is not null.

bool **test_lt**(**const** std::string &*what*, size_t *produced*, size_t *expected*)
: Test that `produced` < `expected`.

bool **test_lte**(**const** std::string &*what*, size_t *produced*, size_t *expected*)
: Test that `produced` <= `expected`.

bool **test_gt**(**const** std::string &*what*, size_t *produced*, size_t *expected*)
: Test that `produced` > `expected`.

bool **test_gte**(**const** std::string &*what*, size_t *produced*, size_t *expected*)
: Test that `produced` >= `expected`.

bool **test_throws**(**const** std::string &*what*, std::function<void)
: > *fn*Call a function and verify it throws an exception of some kind.

bool **test_throws**(**const** std::string &*what*, **const** std::string &*expected*, std::function<void)
: > *fn*Call a function and verify it throws an exception of some kind and that the exception message exactly equals `expected`.

14.3.5 Text_Based_Test

A `Text_Based_Text` runs tests that are produced from a text file with a particular format which looks somewhat like an INI-file:

```
# Comments begin with # and continue to end of line
[Header]
# Test 1
Key1 = Value1
Key2 = Value2

# Test 2
Key1 = Value1
Key2 = Value2
```

class VarMap
: An object of this type is passed to each invocation of the text-based test. It is used to access the test variables. All access takes a key, which is one of the strings which was passed to the constructor of `Text_Based_Text`. Accesses are either required (`get_req_foo`), in which case an exception is throwing if the key is not set, or optional (`get_opt_foo`) in which case the test provides a default value which is returned if the key was not set for this particular instance of the test.

std::vector<uint8_t> **get_req_bin**(**const** std::string &*key*) **const**
: Return a required binary string. The input is assumed to be hex encoded.

std::vector<uint8_t> **get_opt_bin**(**const** std::string &*key*) **const**
: Return an optional binary string. The input is assumed to be hex encoded.

std::vector<std::vector<uint8_t>> **get_req_bin_list**(**const** std::string &*key*) **const**

Botan::BigInt **get_req_bn**(**const** std::string &*key*) **const**
: Return a required BigInt. The input can be decimal or (with “0x” prefix) hex encoded.

Botan::BigInt **get_opt_bn**(**const** std::string &*key*, **const** Botan::BigInt &*def_value*) **const**
Return an optional BigInt. The input can be decimal or (with "0x" prefix) hex encoded.

std::string **get_req_str**(**const** std::string &*key*) **const**
Return a required text string.

std::string **get_opt_str**(**const** std::string &*key*, **const** std::string &*def_value*) **const**
Return an optional text string.

size_t **get_req_sz**(**const** std::string &*key*) **const**
Return a required integer. The input should be decimal.

size_t **get_opt_sz**(**const** std::string &*key*, **const** size_t *def_value*) **const**
Return an optional integer. The input should be decimal.

class Text_Based_Test : **public** *Test*

Text_Based_Test(**const** std::string &*input_file*, **const** std::string &*required_keys*, **const** std::string &*optional_keys* = "")
This constructor is

Note: The final element of required_keys is the "output key", that is the key which signifies the boundary between one test and the next. When this key is seen, `run_one_test` will be invoked. In the test input file, this key must always appear least for any particular test. All the other keys may appear in any order.

Test::*Result* **run_one_test**(**const** std::string &*header*, **const** *VarMap* &*vars*)
Runs a single test and returns the result of it. The `header` parameter gives the value (if any) set in a `[Header]` block. This can be useful to distinguish several types of tests within a single file, for example "[Valid]" and "[Invalid]".

bool **clear_between_callbacks**() **const**
By default this function returns `false`. If it returns `true`, then when processing the data in the file, variables are not cleared between tests. This can be useful when several tests all use some common parameters.

14.3.6 Test Runner

If you are simply writing a new test there should be no need to modify the runner, however it can be useful to be aware of its abilities.

The runner can run tests concurrently across many cores. By default single threaded execution is used, but you can use `--test-threads` option to specify the number of threads to use. If you use `--test-threads=0` then the runner will probe the number of active CPUs and use that (but limited to at most 16). If you want to run across many cores on a large machine, explicitly specify a thread count. The speedup is close to linear.

The RNG used in the tests is deterministic, and the seed is logged for each execution. You can cause the random sequence to repeat using `--drbg-seed` option.

Note: Currently the RNG is seeded just once at the start of execution. So you must run the exact same sequence of tests as the original test run in order to get reproducible results.

If you are trying to track down a bug that happens only occasionally, two very useful options are `--test-runs` and `--abort-on-first-fail`. The first takes an integer and runs the specified test cases that many times. The second causes abort to be called on the very first failed test. This is sometimes useful when tracing a memory corruption bug.

14.4 Continuous Integration and Automated Testing

14.4.1 CI Build Script

The Travis and AppVeyor builds are orchestrated using a script `src/scripts/ci_build.py`. This allows one to easily reproduce the CI process on a local machine.

14.4.2 Travis CI

https://travis-ci.com/github/randombit/botan

This is the primary CI, and tests the Linux, macOS, and iOS builds. Among other things it runs tests using valgrind, compilation on various architectures (currently including ARM, PPC64, and S390x), MinGW build, and a build that produces the coverage report.

The Travis configurations is in `src/scripts/ci/travis.yml`, which executes a setup script `src/scripts/ci/setup_travis.sh` to install needed packages. Then `src/scripts/ci_build.py` is invoked.

14.4.3 AppVeyor

https://ci.appveyor.com/project/randombit/botan

Runs a build/test cycle using MSVC on Windows. Like Travis it uses `src/scripts/ci_build.py`. The AppVeyor setup script is in `src/scripts/ci/setup_appveyor.bat`

The AppVeyor build uses sccache (https://github.com/mozilla/sccache) as a compiler cache. Since that is not available in the AppVeyor images, the setup script downloads a release binary from the upstream repository.

14.4.4 LGTM

https://lgtm.com/projects/g/randombit/botan/

An automated linter that is integrated with Github. It automatically checks each incoming PR. It also supports custom queries/alerts, which likely would be worth investigating but is not something currently in use.

14.4.5 Coverity

https://scan.coverity.com/projects/624

An automated source code scanner. Use of Coverity scanner is rate-limited, sometimes it is very slow to produce a new report, and occasionally the service goes offline for days or weeks at a time. New reports are kicked off manually by rebasing branch `coverity_scan` against the most recent master and force pushing it.

14.4.6 Sonar

https://sonarcloud.io/dashboard?id=botan

Sonar scanner is another software quality scanner. Unfortunately a recent update of their scanner caused it to take over an hour to produce a report which caused Travis CI timeouts, so it has been disabled. It should be re-enabled to run on demand in the same way Coverity is.

14.4.7 OSS-Fuzz

https://github.com/google/oss-fuzz/

OSS-Fuzz is a distributed fuzzer run by Google. Every night, each library fuzzers in `src/fuzzer` are built and run on many machines, with any findings reported to the developers via email.

14.5 Fuzzing The Library

Botan comes with a set of fuzzing endpoints which can be used to test the library.

14.5.1 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 \
      --enable-sanitizers=coverage,address,undefined
$ make fuzzers
```

Enabling 'coverage' sanitizer flags is required for libFuzzer to work. Address sanitizer and undefined sanitizer are optional.

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.

14.5.2 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 sanitizers are optional. You can also use *afl-clang-fast++* or *afl-clang++*, be sure to set *–cc=clang* also.

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
```

14.5.3 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 repository:

```
$ 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_attacker.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.

14.5.4 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

14.5.5 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.

14.6 Release Process and Checklist

Releases are done quarterly, normally on the first non-holiday Monday of January, April, July and October. A feature freeze goes into effect starting 9 days before the release.

Note: This information is only useful if you are a developer of botan who is creating a new release of the library.

14.6.1 Pre Release Testing

Kick off a Coverity scan a day or so before the planned release.

Do maintainer-mode builds with Clang and GCC to catch any warnings that should be corrected. Also check Visual C++ build logs for any warnings that should be addressed.

And remember that CI doesn't test everything. In particular, not all tests run under valgrind or on the qemu cross builds due to time constraints. So before release:

- Run under valgrind, building with `--with-valgrind` flag
- Using Clang sanitizers (ASan + UbSan)
- Native compile on FreeBSD x86-64
- Native compile on at least one unusual platform (AIX, NetBSD, . . .)
- Build the website content to detect any Doxygen problems
- Test many build configurations (using *src/scripts/test_all_configs.py*)
- Build/test SoftHSM

Confirm that the release notes in `news.rst` are accurate and complete and that the version number in `version.txt` is correct.

14.6.2 Tag the Release

Update the release date in the release notes and change the entry for the appropriate branch in `readme.rst` to point to the new release.

Now check in, and backport changes to the release branch:

```
$ git commit readme.rst news.rst -m "Update for 2.6.13 release"
$ git checkout release-2
$ git merge master
$ git tag 2.6.13
```

14.6.3 Build The Release Tarballs

The release script is `src/scripts/dist.py` and must be run from a git workspace.

$ src/scripts/dist.py 2.6.13

One useful option is `--output-dir`, which specifies where the output will be placed.

Now do a final build/test of the released tarball.

The `--pgp-key-id` option is used to specify a PGP keyid. If set, the script assumes that it can execute GnuPG and will attempt to create signatures for the tarballs. The default value is `EFBADFBC`, which is the official signing key. You can use `--pgp-key-id=none` to avoid creating any signature, though official distributed releases *should not* be released without signatures.

The releases served on the official site are taken from the contents in a git repository:

```
$ git checkout git@botan.randombit.net:/srv/git/botan-releases.git
$ src/scripts/dist.py 2.6.13 --output-dir=botan-releases
$ cd botan-releases
$ sha256sum Botan-2.6.13.tgz >> sha256sums.txt
```

(continues on next page)

(continued from previous page)

```
$ git add .
$ git commit -m "Release version 2.6.13"
$ git push origin master
```

A cron job updates the live site every 10 minutes.

14.6.4 Push to GitHub

Don't forget to also push tags:

```
$ git push origin --tags release-2 master
```

14.6.5 Build The Windows Installer

Note: We haven't distributed Windows binaries for some time.

On Windows, run `configure.py` to setup a build:

```
$ python ./configure.py --cc=msvc --cpu=$ARCH --distribution-info=unmodified
```

After completing the build (and running the tests), use InnoSetup (http://www.jrsoftware.org/isinfo.php) to create the installer. A InnoSetup script is created from `src/build-data/innosetup.in` and placed in `build/botan.iss` by `configure.py`. Create the installer either via the InnoSetup GUI by opening the `iss` file and selecting the 'Compile' option, or using the `iscc` command line tool. If all goes well it will produce an executable with a name like `botan-2.6.13-x86_64.exe`. Sign the installers with GPG.

14.6.6 Update The Website

The website content is created by `src/scripts/website.py`.

The website is mirrored automatically from a git repository which must be updated:

```
$ git checkout git@botan.randombit.net:/srv/git/botan-website.git
$ ./src/scripts/website.py --output botan-website
$ cd botan-website
$ git add .
$ git commit -m "Update for 2.6.13"
$ git push origin master
```

14.6.7 Announce The Release

Send an email to the botan-announce and botan-devel mailing lists noting that a new release is available.

14.7 Todo List

Feel free to take one of these on if it interests you. Before starting out on something, send an email to the dev list or open a discussion ticket on GitHub to make sure you're on the right track.

Request a new feature by opening a pull request to update this file.

14.7.1 Ciphers, Hashes, PBKDF

- Stiched AES/GCM mode for CPUs supporting both AES and CLMUL
- Combine AES-NI, ARMv8 and POWER AES implementations (as already done for CLMUL)
- Vector permute AES only supports little-endian systems; fix for big-endian
- SM4 using AES-NI (https://github.com/mjosaarinen/sm4ni) or vector permute
- Poly1305 using AVX2
- ChaCha using SSSE3
- Skein-MAC
- PMAC
- SIV-PMAC
- GCM-SIV (RFC 8452)
- EME* tweakable block cipher (https://eprint.iacr.org/2004/125)
- FFX format preserving encryption (NIST 800-38G)
- SHA-512 using BMI2+AVX2
- Constant time DES using bitslicing and/or BMI2
- Threefish-1024
- SIMD evaluation of SHA-2 and SHA-3 compression functions
- Adiantum (https://eprint.iacr.org/2018/720)
- CRC using clmul/pmull

14.7.2 Public Key Crypto, Math

- Short vector optimization for BigInt
- Abstract representation of ECC point elements to allow specific implementations of the field arithmetic depending upon the curve.
- Use NAF (joint sparse form) for ECC multi-exponentiation
- Curves for pairings (BN-256, BLS12-381)
- Identity based encryption
- Paillier homomorphic cryptosystem
- Socialist Millionaires Protocol (needed for OTRv3)
- Hashing onto an elliptic curve (draft-irtf-cfrg-hash-to-curve)
- New PAKEs (pending CFRG bakeoff results)

- New post quantum schemes (pending NIST contest results)
- SPHINX password store (https://eprint.iacr.org/2018/695)
- X448 and Ed448
- Use GLV decomposition to speed up secp256k1 operations

14.7.3 Utility Functions

- Add a memory span type
- Make Memory_Pool more concurrent (currently uses a global lock)
- Guarded integer type to prevent overflow bugs
- Add logging callbacks
- Add latency tracing framework

14.7.4 Multiparty Protocols

- Distributed key generation for DL, RSA
- Threshold signing, decryption

14.7.5 External Providers, Hardware Support

- Add support ARMv8.4-A SHA-512, SHA-3, SM3 and RNG
- Aarch64 inline asm for BigInt
- Extend OpenSSL provider (DH, HMAC, CMAC, GCM)
- Support using BoringSSL instead of OpenSSL or LibreSSL
- /dev/crypto provider (ciphers, hashes)
- Windows CryptoNG provider (ciphers, hashes)
- Extend Apple CommonCrypto provider (HMAC, CMAC, RSA, ECDSA, ECDH)
- Add support for iOS keychain access
- POWER8 SHA-2 extensions (GH #1486 + #1487)
- Add support VPSUM on big-endian PPC64 (GH #2252)
- Better TPM support: NVRAM, PCR measurements, sealing
- Add support for TPM 2.0 hardware
- Support Intel QuickAssist accelerator cards

14.7.6 TLS

- Make DTLS support optional at build time
- Improve/optimize DTLS defragmentation and retransmission
- Implement logging callbacks for TLS
- Make RSA optional at build time
- Make finite field DH optional at build time
- Authentication using TOFU (sqlite3 storage)
- Certificate pinning (using TACK?)
- Certificate Transparency extensions
- TLS supplemental authorization data (RFC 4680, RFC 5878)
- DTLS-SCTP (RFC 6083)

14.7.7 PKIX

- Further tests of validation API (see GH #785)
- Test suite for validation of 'real world' cert chains (GH #611)
- Improve output of X509_Certificate::to_string This is a free-form string for human consumption so the only constraints are being informative and concise. (GH #656)
- X.509 policy constraints
- OCSP responder logic

14.7.8 New Protocols / Formats

- ACME protocol
- PKCS7 / Cryptographic Message Syntax
- PKCS12 / PFX
- Off-The-Record v3 https://otr.cypherpunks.ca/
- Certificate Management Protocol (RFC 5273); requires CMS
- Fernet symmetric encryption (https://cryptography.io/en/latest/fernet/)
- RNCryptor format (https://github.com/RNCryptor/RNCryptor)
- Useful OpenPGP subset 1: symmetrically encrypted files. Not aiming to process arbitrary OpenPGP, but rather produce something that happens to be readable by *gpg* and is relatively simple to process for decryption. Require AEAD mode (EAX/OCB).
- Useful OpenPGP subset 2: Process OpenPGP public keys
- Useful OpenPGP subset 3: Verification of OpenPGP signatures

14.7.9 Cleanups

- Split test_ffi.cpp into multiple files
- Unicode path support on Windows (GH #1615)
- The X.509 path validation tests have much duplicated logic

14.7.10 Compat Headers

- OpenSSL compatible API headers: EVP, TLS, certificates, etc

14.7.11 New C APIs

- PKCS10 requests
- Certificate signing
- Expose TLS
- Expose NIST key wrap with padding
- Expose secret sharing
- Expose deterministic PRNG
- base32
- base58
- DL_Group
- EC_Group

14.7.12 Python

- Anywhere Pylint warnings too-many-locals, too-many-branches, or too-many-statements are skipped, fix the code so Pylint no longer warns.
- Write a CLI or HTTPS client in Python

14.7.13 Build/Test

- Start using GitHub Actions for CI, especially Windows builds
- Create Docker image for Travis that runs 18.04 and has all the tools we need pre-installed.
- Code signing for Windows installers
- Test runner python script that captures backtraces and other debug info during CI
- Support hardcoding all test vectors into the botan-test binary so it can run as a standalone item (copied to a device, etc)
- Run iOS binary under simulator in CI
- Run Android binary under simulator in CI
- Run the TPM tests against an emulator (https://github.com/PeterHuewe/tpm-emulator)

- Add clang-tidy, clang-analyzer, cppcheck to CI
- Add support for vxWorks
- Add support for Fuschia OS
- Add support for CloudABI
- Add support for SGX

14.7.14 CLI

- Add a `--completion` option to dump autocomplete info, write support for autocompletion in bash/zsh.
- Refactor `speed`
- Change *tls_server* to be a tty<->socket app, like *tls_client* is, instead of a bogus echo server.
- *encrypt* / *decrypt* tools providing password based file encryption
- Add ECM factoring
- Clone of *minisign* signature utility
- Implementation of *tlsdate*
- Password store utility
- TOTP calculator

14.7.15 Documentation

- X.509 certs, path validation
- Specific docs covering one major topic (RSA, ECDSA, AES/GCM, . . .)
- Some howto style docs (setting up CA, . . .)

14.8 OS Features

A summary of OS features as defined in `src/build-data/os`.

```
a: aix
a: android
c: cygwin
d: dragonfly
e: emscripten
f: freebsd
h: haiku
h: hpux
h: hurd
i: includeos
i: ios
l: linux
l: llvm
m: macos
m: mingw
n: nacl
```

(continues on next page)

(continued from previous page)

```
n: netbsd
o: openbsd
q: qnx
s: solaris
u: uwp
w: windows
```

Feature	a	a	c	d	e	f	h	h	h	i	i	l	l	m	m	n	n	o	q	s	u	w
apple_keychain														X								
arc4random		X		X		X					X			X			X	X				
cap_enter						X																
certificate_store																						X
clock_gettime	X	X		X		X	X	X	X			X		X			X	X	X	X		
commoncrypto											X			X								
crypto_ng																					X	
dev_random	X	X	X	X	X	X	X	X	X	X		X		X			X	X	X	X		
elf_aux_info						X																
explicit_bzero						X												X				
explicit_memset																	X					
filesystem	X	X	X	X	X	X	X	X	X		X	X	X	X	X		X	X	X	X	X	X
getauxval		X										X										
getentropy						X								X				X				
pledge																		X				
posix1	X	X	X	X	X	X	X	X	X	X	X	X		X			X	X	X	X		
posix_mlock	X	X		X		X		X	X		X	X		X			X	X	X	X		
proc_fs	X			X								X								X		
rtlgenrandom															X							X
rtlsecurezeromemory																					X	X
sockets	X	X	X	X		X	X	X	X		X	X		X			X	X	X	X		
threads	X	X	X	X		X	X	X	X		X	X		X	X	X	X	X	X	X	X	X
virtual_lock															X							X
win32															X						X	X
winsock2																					X	X

Note: This file is auto generated by `src/scripts/gen_os_features.py`. Dont modify it manually.

14.9 Private OID Assignments

The library uses some OIDs under a private arc assigned by IANA, 1.3.6.1.4.1.25258

Values currently assigned are:

```
randombit   OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 25258 }

publicKey   OBJECT IDENTIFIER ::= { randombit 1 }

mceliece    OBJECT IDENTIFIER ::= { publicKey 3 }
-- { publicKey 4 } previously used as private X25519
-- { publicKey 5 } previously used for XMSS draft 6
gost-3410-with-sha256 OBJECT IDENTIFIER ::= { publicKey 6 1 }
kyber       OBJECT IDENTIFIER ::= { publicKey 7 }
xmss        OBJECT IDENTIFIER ::= { publicKey 8 }

symmetricKey OBJECT IDENTIFIER ::= { randombit 3 }

ocbModes OBJECT IDENTIFIER ::= { symmetricKey 2 }

aes-128-ocb      OBJECT IDENTIFIER ::= { ocbModes 1 }
aes-192-ocb      OBJECT IDENTIFIER ::= { ocbModes 2 }
aes-256-ocb      OBJECT IDENTIFIER ::= { ocbModes 3 }
serpent-256-ocb  OBJECT IDENTIFIER ::= { ocbModes 4 }
twofish-256-ocb  OBJECT IDENTIFIER ::= { ocbModes 5 }
camellia-128-ocb OBJECT IDENTIFIER ::= { ocbModes 6 }
camellia-192-ocb OBJECT IDENTIFIER ::= { ocbModes 7 }
camellia-256-ocb OBJECT IDENTIFIER ::= { ocbModes 8 }

sivModes OBJECT IDENTIFIER ::= { symmetricKey 4 }

aes-128-siv      OBJECT IDENTIFIER ::= { sivModes 1 }
aes-192-siv      OBJECT IDENTIFIER ::= { sivModes 2 }
aes-256-siv      OBJECT IDENTIFIER ::= { sivModes 3 }
serpent-256-siv  OBJECT IDENTIFIER ::= { sivModes 4 }
twofish-256-siv  OBJECT IDENTIFIER ::= { sivModes 5 }
camellia-128-siv OBJECT IDENTIFIER ::= { sivModes 6 }
camellia-192-siv OBJECT IDENTIFIER ::= { sivModes 7 }
camellia-256-siv OBJECT IDENTIFIER ::= { sivModes 8 }
sm4-128-siv      OBJECT IDENTIFIER ::= { sivModes 9 }
```

14.10 Reading List

These are papers, articles and books that are interesting or useful from the perspective of crypto implementation.

14.10.1 Papers

Implementation Techniques

- "Randomizing the Montgomery Powering Ladder" Le, Tan, Tunstall https://eprint.iacr.org/2015/657 A variant of Algorithm 7 is used for GF(p) point multplications when BOTAN_POINTGFP_BLINDED_MULTIPLY_USE_MONTGOMERY_LADDER is set
- "Accelerating AES with vector permute instructions" Mike Hamburg https://shiftleft.org/papers/vector_aes/ His public doman assembly code was rewritten into SSS3 intrinsics for aes_ssse3.
- "Elliptic curves and their implementation" Langley http://www.imperialviolet.org/2010/12/04/ecc.html Describes sparse representations for ECC math

Random Number Generation

- "On Extract-then-Expand Key Derivation Functions and an HMAC-based KDF" Hugo Krawczyk http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.8254 RNG design underlying HMAC_RNG

AES Side Channels

- "Software mitigations to hedge AES against cache-based software side channel vulnerabilities" https://eprint.iacr.org/2006/052.pdf
- "Cache Games - Bringing Access-Based Cache Attacks on AES to Practice" http://www.ieee-security.org/TC/SP2011/PAPERS/2011/paper031.pdf
- "Cache-Collision Timing Attacks Against AES" Bonneau, Mironov http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.88.4753

Public Key Side Channels

- "Fast Elliptic Curve Multiplications Resistant against Side Channel Attacks" http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.1028&rep=rep1&type=pdf
- "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems" Coron http://www.jscoron.fr/publications/dpaecc.pdf
- "Further Results and Considerations on Side Channel Attacks on RSA" Klima, Rosa https://eprint.iacr.org/2002/071 Side channel attacks on RSA-KEM and MGF1-SHA1
- "Side-Channel Attacks on the McEliece and Niederreiter Public-Key Cryptosystems" Avanzi, Hoerder, Page, and Tunstall https://eprint.iacr.org/2010/479
- "Minimum Requirements for Evaluating Side-Channel Attack Resistance of Elliptic Curve Implementations" BSI https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_ECCGuide_e_pdf.pdf

14.10.2 Books

- "Handbook of Elliptic and Hyperelliptic Curve Cryptography" Cohen and Frey https://www.hyperelliptic.org/HEHCC/ An excellent reference for ECC math, algorithms, and side channels
- "Post-Quantum Cryptography" Bernstein, Buchmann, Dahmen Covers code, lattice, and hash based cryptography

14.10.3 Standards

- IEEE 1363 http://grouper.ieee.org/groups/1363/ Very influential early in the library lifetime, so a lot of terminology used in the public key (such as "EME" for message encoding) code comes from here.
- ISO/IEC 18033-2 http://www.shoup.net/iso/std4.pdf RSA-KEM, PSEC-KEM
- NIST SP 800-108 http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf KDF schemes
- NIST SP 800-90A http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf HMAC_DRBG, Hash_DRBG, CTR_DRBG, maybe one other thing?

14.11 Mistakes Were Made

These are mistakes made early on in the project's history which are difficult to fix now, but mentioned in the hope they may serve as an example for others.

14.11.1 C++ API

As an implementation language, I still think C++ is the best choice (or at least the best choice available in early '00s) at offering good performance, reasonable abstractions, and low overhead. But the user API should have been pure C with opaque structs (rather like the FFI layer, which was added much later). Then an expressive C++ API could be built on top of the C API. This would have given us a stable ABI, allowed C applications to use the library, and (these days) make it easier to progressively rewrite the library in Rust.

14.11.2 Public Algorithm Specific Classes

Classes like AES_128 and SHA_256 should never have been exposed to applications. Intead such operations should have been accessible only via the higher level interfaces (here BlockCipher and HashFunction). This would substantially reduce the overall API and ABI surface.

These interfaces are now deprecated, and perhaps will be able to be removed eventually.

14.11.3 Header Directories

It would have been better to install all headers as `X/header.h` where `X` is the base dir in the source, eg `block/aes128.h`, `hash/md5.h`, ...

14.11.4 Exceptions

Constant ABI headaches from this, and it impacts performance and makes APIs harder to understand. Should have been handled with a result<> type instead.

14.11.5 Virtual inheritance

This was used in the public key interfaces and the hierarchy is a tangle. Public and private keys should be distinct classes, with a function on private keys that creates a new object corresponding to the public key.

14.11.6 Cipher Interface

The cipher interface taking a secure_vector that it reads from and writes to was an artifact of an earlier design which supported both compression and encryption in a single API. But it leads to inefficient copies.

(I am hoping this issue can be somewhat fixed by introducing a new cipher API and implementing the old API in terms of the new one.)

14.11.7 Pipe Interface

On the surface this API seems very convenient and easy to use. And it is. But the downside is it makes the application code totally opaque; some bytes go into a Pipe object and then come out the end transformed in some way. What happens in between? Unless the Pipe was built in the same function and you can see the parameters to the constructor, there is no way to find out.

The problems with the Pipe API are documented, and it is no longer used within the library itself. But since many people seem to like it and many applications use it, we are stuck at least with maintaining it as it currently exists.

14.11.8 License

MIT is more widely used and doesn't have the ambiguity surrounding the various flavors of BSD.

CHAPTER

FIFTEEN

RELEASE NOTES: 0.7.0 TO 1.11.34

15.1 Version 1.10.17, 2017-10-02

- Address a side channel affecting modular exponentiation. An attacker capable of a local or cross-VM cache analysis attack may be able to recover bits of secret exponents as used in RSA, DH, etc. CVE-2017-14737
- Workaround a miscompilation bug in GCC 7 on x86-32 affecting GOST-34.11 hash function. (GH #1192 #1148 #882)
- Add SecureVector::data() function which returns the start of the buffer. This makes it slightly simpler to support both 1.10 and 2.x APIs in the same codebase.
- When compiled by a C++11 (or later) compiler, a template typedef of SecureVector, secure_vector, is added. In 2.x this class is a std::vector with a custom allocator, so has a somewhat different interface than SecureVector in 1.10. But this makes it slightly simpler to support both 1.10 and 2.x APIs in the same codebase.
- Fix a bug that prevented *configure.py* from running under Python3
- Botan 1.10.x does not support the OpenSSL 1.1 API. Now the build will *#error* if OpenSSL 1.1 is detected. Avoid *–with-openssl* if compiling against 1.1 or later. (GH #753)
- Import patches from Debian adding basic support for building on aarch64, ppc64le, or1k, and mipsn32 platforms.

15.2 Version 1.10.16, 2017-04-04

- Fix a bug in X509 DN string comparisons that could result in out of bound reads. This could result in information leakage, denial of service, or potentially incorrect certificate validation results. (CVE-2017-2801)
- Avoid throwing during a destructor since this is undefined in C++11 and rarely a good idea. (GH #930)

15.3 Version 1.10.15, 2017-01-12

- Fix a bug causing modular exponentiations done modulo even numbers to almost always be incorrect, unless the values were small. This bug is not known to affect any cryptographic operation in Botan. (GH #754)
- Avoid use of C++11 std::to_string in some code added in 1.10.14 (GH #747 #834)

15.4 Version 1.11.34, 2016-11-28

- Fix integer overflow during BER decoding, found by Falko Strenzke. This bug is not thought to be directly exploitable but upgrading ASAP is advised. (CVE-2016-9132)
- Add post-quantum signature scheme XMSS. Provides either 128 or 256 bit (post-quantum) security, with small public and private keys, fast verification, and reasonably small signatures (2500 bytes for 128-bit security). Signature generation is very slow, on the order of seconds. And very importantly the signature scheme is stateful: each leaf index must only be used once, or all security is lost. In the appropriate system where signatures are rarely generated (such as code signing) XMSS makes an excellent choice. (GH #717 #736)
- Add support for CECPQ1 TLS ciphersuites. These use a combination of x25519 ECDH and NewHope to provide post-quantum security. The ciphersuites are not IETF standard, but is compatible with BoringSSL. (GH #729)
- Add support for client-side OCSP stapling to TLS. (GH #738)
- Previously both public and private keys performed automatic self testing after generation or loading. However this often caused unexpected application performance problems, and so has been removed. Instead applications must call check_key explicitly. (GH #704)
- Fix TLS session resumption bugs which caused resumption failures if an application used a single session cache for both TLS and DTLS. (GH #688)
- Add SHAKE-128 and SHAKE-256 XOFs as hash functions supporting paramaterized output lengths.
- Add MessageAuthenticationCode::start_msg interface, for MACs which require or can use a nonce (GH #691)
- Add GMAC, a MAC based on GCM (GH #488 / #691)
- Add ESP block cipher padding from RFC 4304. GH #724
- Incompatible change to HKDF: previously the HKDF type in Botan was only the Expand half of HKDF. Now HKDF is the full Extract-then-Expand KDF, and HKDF_Extract and HKDF_Expand are available. If you previously used HKDF, you must switch to using HKDF_Expand. (GH #723)
- Add Cipher_Mode::reset which resets message-specific state, allowing discarding state but allowing continued processing under the same key. (GH #552)
- The ability to add OIDs at runtime has been removed. This additionally removes a global lock which was acquired on each OID lookup. (GH #706)
- The default TLS policy now disables static RSA ciphersuites, all DSA ciphersuites, and the AES CCM-8 ciphersuites. Disabling static RSA by default protects servers from oracle attacks, as well as enforcing a forward secure ciphersuite. Some applications may be forced to re-enable RSA for interop reasons. DSA and CCM-8 are rarely used, and likely should not be negotiated outside of special circumstances.
- The default TLS policy now prefers ChaCha20Poly1305 cipher over any AES mode.
- The default TLS policy now orders ECC curve preferences in order by performance, with x25519 first, then P-256, then P-521, then the rest.
- Add a BSD sockets version of the HTTP client code used for OCSP. GH #699
- Export the public key workfactor functions (GH #734) and add tests for them.
- HMAC_DRBG allows configuring maximum number of bytes before reseed check (GH #690)
- Salsa20 now accepts a null IV as equivalent to an all-zero one (GH #697)
- Optimize ECKCDSA verification (GH #700 #701 #702)
- The deprecated RNGs HMAC_RNG and X9.31 RNG have been removed. Now the only userspace PRNG included in the library is HMAC_DRBG. (GH #692)

- The entropy sources for EGD and BeOS, as well as the Unix entropy source which executed processes to get statistical data have been removed. (GH #692)
- The openpgp module (which just implemented OpenPGP compatible base64 encoding and decoding, nothing else) has been removed.
- Added new configure.py argument *–optimize-for-size*. Currently just sets the flag for code size optimizations with the compiler, but may have other effects in the future.
- Fixed bug in Threaded_Fork causing incorrect computations (GH #695 #716)
- Add DSA deterministic parameter generation test from FIPS 186-3.
- Fix PKCS11_ECDSA_PrivateKey::check_key (GH #712)
- Fixed problems running configure.py outside of the base directory
- The BOTAN_ENTROPY_PROC_FS_PATH value in build.h was being ignored (GH #708)
- Add speed tests for ECGDSA and ECKCDSA (GH #696)
- Fix a crash in speed command for Salsa20 (GH #697)
- Allow a custom ECC curve to be specified at build time, for application or system specific curves. (GH #636 #710)
- Use NOMINMAX on Windows to avoid problems in amalgamation build. (GH #740)
- Add support to output bakefiles with new *configure.py* option *–with-bakefile*. (GH #360 #720)
- The function *zero_mem* has been renamed *secure_scrub_memory*
- More tests for pipe/filter (GH #689 #693), AEADs (GH #552), KDF::name (GH #727),
- Add a test suite for timing analysis for TLS CBC decryption, OAEP decryption, and PKCS #1 v1.5 decryption. These operations all have the feature that if an attacker can distinguish internal operations, such as through a variance in timing, they can use this oracle to decrypt arbitrary ciphertexts. GH #733
- Add a test suite for testing and fuzzing with TLS-Attacker, a tool for analyzing TLS libraries. (https://github.com/RUB-NDS/TLS-Attacker)
- Add a fuzzing framework. Supports fuzzing some APIs using AFL and libFuzzer.
- Added documentation for PKCS #11 (GH #725)
- The LibraryInitializer type is no longer needed and is now deprecated.
- The license and news files were moved from doc to the top level directory. There should not be any other visible change (eg, to the installed version) as a result of this move.
- Fixed some problems when running configure.py outside of the base directory, especially when using relative paths.
- Add (back) the Perl XS wrapper and sqlite encryption code.

15.5 Version 1.10.14, 2016-11-28

- NOTE WELL: Botan 1.10.x is supported for security patches only until 2017-12-31
- Fix integer overflow during BER decoding, found by Falko Strenzke. This bug is not thought to be directly exploitable but upgrading ASAP is advised. (CVE-2016-9132)
- Fix two cases where (in error situations) an exception would be thrown from a destructor, causing a call to std::terminate.
- When RC4 is disabled in the build, also prevent it from being included in the OpenSSL provider. (GH #638)

15.6 Version 1.11.33, 2016-10-26

- Avoid side channel during OAEP decryption. (CVE-2016-8871)
- A countermeasure for the Lucky13 timing attack against CBC-based TLS ciphersuites has been added. (GH #675)
- Added X25519-based key exchange for TLS (GH #673)
- Add Certificate_Store_In_SQL which supports storing certs, keys, and revocation information in a SQL database. Subclass Certificate_Store_In_SQLite specializes with support for SQLite3 databases. (GH #631)
- The Certificate_Store interface has been changed to deal with `std::shared_ptrs` instead of raw pointers (GH #471 #631)
- Add support for official SHA-3. Keccak-1600 was already supported but used different padding from FIPS 202. (GH #669)
- Add SHAKE-128 based stream cipher. (GH #669)
- NewHope key exchange now supports the SHA-256/AES-128-CTR scheme used by BoringSSL in addition to the SHA-3/SHAKE-128 parameters used by the reference implementation. (GH #669)
- Add support for the TLS Supported Point Formats Extension from RFC 4492. Adds `TLS::Policy::use_ecc_point_compression` policy option. If supported on both sides, ECC points can be sent in compressed format which saves a few bytes during the handshake. (GH #645)
- Fix entropy source selection bug on Windows, which caused the CryptoAPI entropy source to be not available under its normal name “win32_cryptoapi” but instead “dev_random”. GH #644
- Accept read-only access to `/dev/urandom`. System_RNG previously required read-write access, to allow applications to provide inputs to the system PRNG. But local security policies might only allow read-only access, as is the case with Ubuntu’s AppArmor profile for applications in the Snappy binary format. If opening read/write fails, System_RNG silently backs down to read-only, in which case calls to `add_entropy` on that object will fail. (GH #647 #648)
- Fix use of Win32 CryptoAPI RNG as an entropy source, which was accidentally disabled due to empty list of acceptable providers being specified. Typically the library would fall back to gathering entropy from OS functions returning statistical information, but if this functionality was disabled in the build a `PRNG_Unseeded` exception would result. (GH #655)
- Add support for building the library as part of the IncludeOS unikernel. This included making filesystem and threading support optional. (GH #665)
- Added ISA annotations so that with GCC (all supported versions) and Clang (since 3.7) it is no longer required to compile amalgamation files with ABI specific flags such as `-maes`. (GH #665)

- Internal cleanups to TLS CBC record handling. TLS CBC ciphersuites can now be disabled by disabling `tls_cbc` module. (GH #642 #659)
- Internal cleanups to the object lookup code eliminates most global locks and all use of static initializers (GH #668 #465)
- Avoid `static_assert` triggering under MSVC debug builds (GH #646)
- The antique PBKDF1 password hashing scheme is deprecated and will be removed in a future release. It was only used to support the equally ancient PBES1 private key encryption scheme, which was removed in 1.11.8.
- Added MSVC debug/checked iterator builds (GH #666 #667)
- Added Linux ppc64le cross compile target to Travis CI (GH #654)
- If RC4 is disabled, also disable it coming from the OpenSSL provider (GH #641)
- Add TLS message parsing tests (GH #640)
- Updated BSI policy to prohibit DES, HKDF, HMAC_RNG (GH #649)
- Documentation improvements (GH #660 #662 #663 #670)

15.7 Version 1.11.32, 2016-09-28

- Add support for the NewHope Ring-LWE key encapsulation algorithm. This scheme provides an estimated ~200 bit security level against a quantum attacker while also being very fast and requiring only modest message sizes of 1824 and 2048 bytes for initiator and responder, resp. This version is tested as having bit-for-bit identical output as the reference implementation by the authors.

  Be warned that NewHope is still a very new scheme and may yet fall to analysis. For best assurance, NewHope should be used only in combination with another key exchange mechanism, such as ECDH.
- New TLS callbacks API. Instead of numerous std::function callbacks, the application passes an object implementing the TLS::Callbacks interface, which has virtual functions matching the previous callbacks (plus some extras). Full source compatability with previous versions is maintained for now, but the old interface is deprecated and will be removed in a future release. The manual has been updated to reflect the changes. (GH #457 and #567)
- Add support for TLS Encrypt-then-MAC extension (GH #492 and #578), which fixes the known issues in the TLS CBC-HMAC construction.
- The format of the TLS session struct has changed (to support EtM), so old TLS session caches will be invalidated.
- How the library presents optimized algorithm implementations has changed. For example with the algorithm AES-128, previously there were three BlockCipher classes AES_128, AES_128_SSSE3, and AES_128_NI which used (resp) a table-based implementation vulnerable to side channels, a constant time version using SSSE3 SIMD extensions on modern x86, and x86 AES-NI instructions. Using the correct version at runtime required using `BlockCipher::create`. Now, only the class AES_128 is presented, and the best available version is always used based on CPUID checks. The tests have been extended to selectively disable CPUID bits to ensure all available versions are tested. (GH #477 #623)

  Removes API classes AES_128_NI, AES_192_NI, AES_256_NI, AES_128_SSSE3, AES_192_SSSE3 AES_256_SSSE3, IDEA_SSE2, Noekeon_SIMD, Serpent_SIMD, Threefish_512_AVX2, SHA_160_SSE2
- The deprecated algorithms Rabin-Williams, Nyberg-Rueppel, MARS, RC2, RC5, RC6, SAFER-SK, TEA, MD2, HAS-160, and RIPEMD-128 have been removed. (GH #580)
- A new Cipher_Mode interface `process` allows encryption/decryption of buffers without requiring copying into `secure_vector` first. (GH #516)

- Fix verification of self-issued certificates (GH #634)
- SSE2 optimizations for ChaCha, 60% faster on both Westmere and Skylake (GH #616)
- The HMAC_RNG constructor added in 1.11.31 that took both an RNG and an entropy source list ignored the entropy sources.
- The configure option `--via-amalgamation` was renamed to `--amalgamation`. The configure option `--gen-amalgamation` was removed. It did generate amalgamations but build Botan without amalgamation. Users should migrate to `--amalgamation`. (GH #621)
- DH keys did not automatically self-test after being generated, contrary to the current behavior for other key types.
- Add tests for TLS 1.2 PRF (GH #628)

15.8 Version 1.11.31, 2016-08-30

- Fix undefined behavior in Curve25519 on platforms without a native 128-bit integer type. This was known to produce incorrect results on 32-bit ARM under Clang. GH #532 (CVE-2016-6878)
- If X509_Certificate::allowed_usage was called with more than one Key_Usage set in the enum value, the function would return true if *any* of the allowed usages were set, instead of if *all* of the allowed usages are set. GH #591 (CVE-2016-6879)
- Incompatible changes in DLIES: Previously the input to the KDF was the concatenation of the (ephemeral) public key and the secret value derived by the key agreement operation. Now the input is only the secret value obtained by the key agreement operation. That's how it is specified in the original paper "DHIES: An encryption scheme based on Diffie-Hellman Problem" or in BSI technical guideline TR-02102-1 for example. In addition to the already present XOR-encrypion/decryption mode it's now possible to use DLIES with a block cipher. Furthermore the order of the output was changed from {public key, tag, ciphertext} to {public key, ciphertext, tag}. Both modes are compatible with BouncyCastle.
- Add initial PKCS #11 support (GH #507). Currently includes a low level wrapper to all of PKCS #11 (p11.h) and high level code for RSA and ECDSA signatures and hardware RNG access.
- Add ECIES encryption scheme, compatible with BouncyCastle (GH #483)
- Add ECKCDSA signature algorithm (GH #504)
- Add KDF1 from ISO 18033 (GH #483)
- Add FRP256v1 curve (GH #551)
- Changes for userspace PRNGs HMAC_DRBG and HMAC_RNG (GH #520 and #593)

  These RNGs now derive from Stateful_RNG which handles issues like periodic reseeding and (on Unix) detecting use of fork. Previously these measures were included only in HMAC_RNG.

  Stateful_RNG allows reseeding from another RNG and/or a specified set of entropy sources. For example it is possible to configure a HMAC_DRBG to reseed using a PKCS #11 token RNG, the CPU's RDSEED instruction, and the system RNG but disabling all other entropy polls.
- AutoSeeded_RNG now uses NIST SP800-90a HMAC_DRBG(SHA-384). (GH #520)
- On Windows and Unix systems, the system PRNG is used as the sole reseeding source for a default AutoSeeded_RNG, completely skipping the standard entropy polling code. New constructors allow specifying the reseed RNG and/or entropy sources. (GH #520)
- The *hres_timer* entropy source module has been removed. Timestamp inputs to the RNG are now handled as additional_data inputs to HMAC_DRBG.

- Add RDRAND_RNG which directly exposes the CPU RNG (GH #543)
- Add PKCS #1 v1.5 id for SHA-512/256 (GH #554)
- Add X509_Time::to_std_timepoint (GH #560)
- Fix a bug in ANSI X9.23 padding mode, which returned one byte more than the given block size (GH #529).
- Fix bug in SipHash::clear, which did not reset all state (GH #547)
- Fixes for FreeBSD (GH #517) and OpenBSD (GH #523). The compiler defaults to Clang on FreeBSD now.
- SonarQube static analysis integration (GH #592)
- Switched Travis CI to Ubuntu 14.04 LTS (GH #592)
- Added ARM32, ARM64, PPC32, PPC64, and MinGW x86 cross compile targets to Travis CI (GH #608)
- Clean up in TLS ciphersuite handling (GH #583)
- Threefish-512 AVX2 optimization work (GH #581)
- Remove build configuration host and timestamp from build.h This makes this header reproducible and allows using ccache's direct mode (GH #586 see also #587)
- Prevent building for x86-64 with x86-32 compiler and the reverse (GH #585)
- Avoid build problem on 32-bit userspace ARMv8 (GH #563)
- Refactor of internal MP headers (GH #549)
- Avoid MSVC C4100 warning (GH #525)
- Change botan.exe to botan-cli.exe on Windows to workaround VC issue (GH #584)
- More tests for RSA-KEM (GH #538), DH (GH #556), EME (GH #553), cipher mode padding (GH #529), CTS mode (GH #531), KDF1/ISO18033 (GH #537), OctetString (GH #545), OIDs (GH #546), parallel hash (GH #548), charset handling (GH #555), BigInt (GH #558), HMAC_DRBG (GH #598 #600)
- New deprecations. See the full list in doc/deprecated.txt

  The X9.31 and HMAC_RNG RNGs are deprecated. If you need a userspace PRNG, use HMAC_DRBG (or AutoSeeded_RNG which is HMAC_DRBG with defaults).

  Support for getting entropy from EGD is deprecated, and will be removed in a future release. The developers believe that it is unlikely that any modern system requires EGD and so the code is now dead weight. If you rely on EGD support, you should contact the developers by email or GitHub ASAP.

  The TLS ciphersuites using 3DES and SEED are deprecated and will be removed in a future release.

  ECB mode Cipher_Mode is deprecated and will be removed in a future release.

  Support for BeOS/Haiku has not been tested in 5+ years and is in an unknown state. Unless reports are received of successful builds and use on this platform, support for BeOS/Haiku will be removed in a future release.

15.9 Version 1.11.30, 2016-06-19

- In 1.11.23 a bug was introduced such that CBC-encrypted TLS packets containing no plaintext bytes at all were incorrectly rejected with a MAC failure. Records like this are used by OpenSSL in TLS 1.0 connections in order to randomize the IV.
- A bug in GCM caused incorrect results if the 32-bit counter field overflowed. This bug has no implications on the security but affects interoperability.

  With a 96-bit nonce, this could only occur if at least 2**32 128-bit blocks (64 GiB) were encrypted. This actually exceeds the maximum allowable length of a GCM plaintext; when messages longer than 2**32 - 2 blocks are encrypted, GCM loses its security properties.

  In addition to 96-bit nonces, GCM also supports nonces of arbitrary length using a different method which hashes the provided nonce under the authentication key. When using such a nonce, the last 4 bytes of the resulting CTR input might be near the overflow boundary, with the probability of incorrect overflow increasing with longer messages. when encrypting 256 MiB of data under a random 128 bit nonce, an incorrect result would be produced about 1/256 of the time. With 1 MiB texts, the probability of error is reduced to 1/65536.

  Since TLS uses GCM with 96 bit nonces and limits the length of any record to far less than 64 GiB, TLS GCM ciphersuites are not affected by this bug.

  Reported by Juraj Somorovsky, described also in "Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS" (https://eprint.iacr.org/2016/475.pdf)
- Previously when generating a new self-signed certificate or PKCS #10 request, the subject DN was required to contain both common name (CN) and country (C) fields. These restrictions have been removed. GH #496
- The Transform and Keyed_Transform interfaces has been removed. The two concrete implementations of these interfaces were Cipher_Mode and Compressor_Transform. The Cipher_Mode interface remains unchanged as the Transform and Keyed_Transform signatures have moved to it; no changes to Cipher_Mode usage should be necessary. Any uses of Transform& or Keyed_Transform& to refer to a cipher should be replaced by Cipher_Mode&. The compression algorithm interface has changed; the start function now takes the per-message compression ratio to use. Previously the compression level to use had to be set once, at creation time, and the required `secure_vector` argument to `start` was required to be empty. The new API is documented in *compression.rst* in the manual.
- Add IETF versions of the ChaCha20Poly1305 TLS ciphersuites from draft-ietf-tls-chacha20-poly1305-04. The previously implemented (non-standard) ChaCha20Poly1305 ciphersuites from draft-agl-tls-chacha20poly1305 remain but are deprecated.
- The OCB TLS ciphersuites have been updated to use the new nonce scheme from draft-zauner-tls-aes-ocb-04. This is incompatible with previous versions of the draft, and the ciphersuite numbers used for the (still experimental) OCB ciphersuites have changed.
- Previously an unknown critical extension caused X.509 certificate parsing to fail; such a cert could not be created at all. Now parsing succeeds and the certificate validation fails with an error indicating an unknown critical extension. GH #469
- X509_CRL previously had an option to cause it to ignore unknown critical extensions. This has been removed.
- Added StreamCipher::seek allowing seeking to arbitrary position in the key stream. Currently only implemented for ChaCha. (GH #497)
- Added support for ChaCha stream cipher with 8 or 12 rounds.
- Add ECGDSA signature algorithm (GH #479)
- Add support for label argument to KDFs (GH #495)
- Add NIST SP800-108 and 56C KDFs (GH #481)

- Support for Card Verifiable Certificates and the obsolete EMSA1_BSI signature padding scheme have been removed. (GH #487)
- A bug in the IETF version of ChaCha20Poly1305 (with 96 bit nonces) caused incorrect computation when the plaintext or AAD was exactly a multiple of 16 bytes.
- Fix return type of TLS_Reader::get_u32bit, which was truncated to 16 bits. This only affected decoding of session ticket lifetimes. GH #478
- Fix OS X dylib naming problem (GH #468 #467)
- Fix bcrypt function under Python 3 (GH #461)
- The `unix_procs` entropy source is deprecated and will be removed in a future release. This entropy source attempts to get entropy by running Unix programs like `arp`, `netstat`, and `dmesg` which produce information which may be difficult for a remote attacker to guess. This exists primarily as a last-ditch for Unix systems without `/dev/random`. But at this point such systems effectively no longer exist, and the use of `fork` and `exec` by the library complicates effective application sandboxing.
- Changes to avoid implicit cast warnings in Visual C++ (GH #484)

15.10 Version 1.10.13, 2016-04-23

- Use constant time modular inverse algorithm to avoid possible side channel attack against ECDSA (CVE-2016-2849)
- Use constant time PKCS #1 unpadding to avoid possible side channel attack against RSA decryption (CVE-2015-7827)
- Avoid a compilation problem in OpenSSL engine when ECDSA was disabled. Gentoo bug 542010

15.11 Version 1.11.29, 2016-03-20

- CVE-2016-2849 DSA and ECDSA used a modular inverse function which had input dependent loops. It is possible a side channel attack on this function could be used to recover sufficient information about the nonce k to mount a lattice attack and recover the private key. Found by Sean Devlin.
- CVE-2016-2850 The TLS client did not check that the signature algorithm or ECC curve a v1.2 server used was actually acceptable by the policy. This would allow a server who ignored the preferences indicated in the client to use a weak algorithm, and may allow MITM attacks by an attacker who can break MD5 signatures or 160 bit ECC in real time. The server similarly failed to check on the hash a client used during client certificate authentication.
- Reject empty TLS records at the record processing layer since such a record is not valid regardless of the record type. Later checks already correctly rejected empty records, but during processing such a record, a pointer to the end of the vector was created, causing a assertion failure under checked iterators. Found by Juraj Somorovsky.
- Add PK_Decryptor::decrypt_or_random which allows an application to atomically (in constant time) check that a decrypted ciphertext has the expected length and/or apply content checks on the result. This is used by the TLS server for decrypting PKCS #1 v1.5 RSA ciphertexts. Previously the server used a implementation which was potentially vulnerable to side channels.
- Add support for processing X.509 name constraint extension during path validation. GH #454
- Add X509_Certificate::v3_extensions which allows retreiving the raw binary of all certificate extensions, including those which are not known to the library. This allows processing of custom extensions. GH #437

- Add support for module policies which are a preconfigured set of acceptable or prohibited modules. A policy based on BSI TR-02102-1 is included. GH #439 #446
- Support for the deprecated TLS heartbeat extension has been removed.
- Support for the deprecated TLS minimum fragment length extension has been removed.
- SRP6 support is now optional in TLS
- Support for negotiating MD5 and SHA-224 signatures in TLS v1.2 has been removed. MD5 signatures are demonstratably insecure in TLS, SHA-224 is rarely used.
- Support for negotiating ECC curves secp160r1, secp160r2, secp160k1, secp192k1, secp192r1 (P-192), secp224k1, secp224r1 (P-224), and secp256k1 have been removed from the TLS implementation. All were already disabled in the default policy.
- HMAC_RNG now has an explicit check for fork using pid comparisons. It also includes the pid and system and CPU clocks into the PRF computation to help reduce the risk of pid wraparound. Even so, applications using fork and userspace RNGs should explicitly reseed all such RNGs whenever possible.
- Deprecation warning: support for DSA certificates in TLS is deprecated and will be removed in a future release.
- Deprecation warning: in addition to the algorithms deprecated in 1.11.26, the following algorithms are now deprecated and will be removed in a future release: Rabin-Williams signatures, TEA, XTEA.
- Deprecation warning: the library has a number of compiled in MODP and ECC DL parameters. All MODP parameter sets under 2048 bits and all ECC parameters under 256 bits are deprecated and will be removed in a future release. This includes the MODP groups "modp/ietf/1024", "modp/srp/1024", "modp/ietf/1536", "modp/srp/1536" and the ECC groups "secp160k1", "secp160r1", "secp160r2", "secp192k1", "secp192r1", "secp224k1", "secp224r1", "brainpool160r1", "brainpool192r1", "brainpool224r1", "x962_p192v2", "x962_p192v3", "x962_p239v1", "x962_p239v2" and "x962_p239v3". Additionally all compiled in DSA parameter sets ("dsa/jce/1024", "dsa/botan/2048", and "dsa/botan/3072") are also deprecated.
- RDSEED/RDRAND polling now retries if the operation fails. GH #373
- Fix various minor bugs found by static analysis with PVS-Studio (GH#421), Clang analyzer (GH #441), cppcheck (GH #444, #445), and Coverity.
- Add –with-valgrind configure option to enable building against the valgrind client API. This currently enables checking of const time operations using memcheck.
- Fix remaining Wshadow warnings. Enable Wshadow in build. GH #427
- Use noexcept in VS 2015 GH #429
- On Windows allow the user to explicitly request symlinks be used as part of the build. Likely only useful for someone working on the library itself. GH #430
- Remove use of TickCount64 introduced in 1.11.27 which caused problem with downstream distributors/users building XP compatiable binaries which is still an option even in VS 2015
- MCEIES requires KDF1 at runtime but did not require it be enabled in the build. GH #369
- Small optimizations to Keccak hash
- Support for locking allocator on Windows using VirtualLock. GH #450

15.12 Version 1.8.15, 2016-02-13

- NOTE WELL: Botan 1.8 is not supported for security issues anymore. Moving to 1.10 or 1.11 is certainly recommended.
- Fix CVE-2014-9742: Insufficient randomness in Miller-Rabin primality check
- Fix CVE-2016-2194: Infinite loop in modulur square root algorithm
- Fix CVE-2015-5726: Crash in BER decoder
- Fix CVE-2015-5727: Excess memory allocation in BER decoder Note: Unlike the fix in 1.10 which checks that the source actually contains enough data to satisfy the read before allocating the memory, 1.8.15 simply rejects all ASN.1 blocks larger than 1 MiB. This simpler check avoids the problem without breaking ABI.

15.13 Version 1.10.12, 2016-02-03

- In 1.10.11, the check in PointGFp intended to check the affine y argument actually checked the affine x again. Reported by Remi Gacogne

  The CVE-2016-2195 overflow is not exploitable in 1.10.11 due to an additional check in the multiplication function itself which was also added in that release, so there are no security implications from the missed check. However to avoid confusion the change was pushed in a new release immediately.

  The 1.10.11 release notes incorrectly identified CVE-2016-2195 as CVE-2016-2915

15.14 Version 1.10.11, 2016-02-01

- Resolve heap overflow in ECC point decoding. CVE-2016-2195
- Resolve infinite loop in modular square root algorithm. CVE-2016-2194
- Correct BigInt::to_u32bit to not fail on integers of exactly 32 bits. GH #239

15.15 Version 1.11.28, 2016-02-01

- One of the checks added while addressing CVE-2016-2195 was incorrect and could cause needless assertion failures.

15.16 Version 1.11.27, 2016-02-01

- SECURITY: Avoid heap overflow in ECC point decoding. This could likely result in remote code execution. CVE-2016-2195
- SECURITY: Avoid one word heap overflow in P-521 reduction function. This could potentially lead to remote code execution or other attack. CVE-2016-2196.
- SECURITY: Avoid infinite or near-infinite loop during modular square root algorithm with invalid inputs. CVE-2016-2194
- Add Blake2b hash function. GH #413
- Use `m_` prefix on all member variables. GH #398 and #407

- Use final qualifier on many classes. GH #408
- Use noreturn attribute on assertion failure function to assist static analysis. GH #403
- Use TickCount64 and MemoryStatusEx in the Windows entropy source. Note these calls are only available in Vista/Server 2008. No accomodations are made for XP or Server 2003, both of which are no longer patched by the vendor. GH #365

15.17 Version 1.11.26, 2016-01-04

- Deprecation warnings: Nyberg-Rueppel signatures, MARS, RC2, RC5, RC6, SAFER, HAS-160, RIPEMD-128, MD2 and support for the TLS minimum fragment length extensions are all being considered for removal in a future release. If there is a compelling use case for keeping any of them in the library, please open a discussion ticket on GitHub.
- Support for the TLS extended master secret extension (RFC 7627) has been added.
- The format of serialized TLS sessions has changed to add a flag indicating support for the extended master secret flag, which is needed for proper handling of the extension.
- Root all exceptions thrown by the library in the `Botan::Exception` class. Previously the library would in many cases throw `std::runtime_error` or `std::invalid_argument` exceptions which would make it hard to determine the source of the error in some cases.
- The command line interface has been mostly rewritten. The syntax of many of the sub-programs has changed, and a number have been extended with new features and options.
- Correct an error in PointGFp multiplication when multiplying a point by the scalar value 3. PointGFp::operator* would instead erronously compute it as if the scalar was 1 instead.
- Enable RdRand entropy source on Windows/MSVC. GH #364
- Add Intel's RdSeed as entropy source. GH #370
- Add preliminary support for accessing TPM v1.2 devices. Currently random number generation, RSA key generation, and signing are supported. Tested using Trousers and an ST TPM
- Add generalized interface for KEM (key encapsulation) techniques. Convert McEliece KEM to use it. The previous interfaces McEliece_KEM_Encryptor and McEliece_KEM_Decryptor have been removed. The new KEM interface now uses a KDF to hash the resulting keys; to get the same output as previously provided by McEliece_KEM_Encryptor, use "KDF1(SHA-512)" and request exactly 64 bytes.
- Add support for RSA-KEM from ISO 18033-2
- Add support for ECDH in the OpenSSL provider
- Fix a bug in DataSource::discard_next() which could cause either an infinite loop or the discarding of an incorrect number of bytes. Reported on mailing list by Falko Strenzke.
- Previously if BOTAN_TARGET_UNALIGNED_MEMORY_ACCESS_OK was defined, the code doing low level loads/stores would use pointer casts to access larger words out of a (potentially misaligned) byte array, rather than using byte-at-a-time accesses. However even on platforms such as x86 where this works, it triggers UBSan errors under Clang. Instead use memcpy, which the C standard says is usable for such purposes even with misaligned values. With recent GCC and Clang, the same code seems to be emitted for either approach.
- Avoid calling memcpy, memset, or memmove with a length of zero to avoid undefined behavior, as calling these functions with an invalid or null pointer, even with a length of zero, is invalid. Often there are corner cases where this can occur, such as pointing to the very end of a buffer.

- The function `RandomNumberGenerator::gen_mask` (added in 1.11.20) had undefined behavior when called with a bits value of 32 or higher, and was tested to behave in unpleasant ways (such as returning zero) when compiled by common compilers. This function was not being used anywhere in the library and rather than support something without a use case to justify it it seemed simpler to remove it. Undefined behavior found by Daniel Neus.
- Support for using `ctgrind` for checking const time blocks has been replaced by calling the valgrind memcheck APIs directly. This allows const-time behavior to be tested without requiring a modified valgrind binary. Adding the appropriate calls requires defining BOTAN_HAS_VALGRIND in build.h. A binary compiled with this flag set can still run normally (though with some slight runtime overhead).
- Export MGF1 function mgf1_mask GH #380
- Work around a problem with some antivirus programs which causes the `shutil.rmtree` and `os.makedirs` Python calls to occasionally fail. The could prevent `configure.py` from running sucessfully on such systems. GH #353
- Let `configure.py` run under CPython 2.6. GH #362

15.18 Version 1.11.25, 2015-12-07

- In this release the test suite has been largely rewritten. Previously the tests had internally used several different test helper frameworks created or adopted over time, each of which was insufficient on its own for testing the entire library. These have been fully converged on a new framework which suffices for all of the tests. There should be no user-visible change as a result of this, except that the output format of *botan-test* has changed.
- Improved side channel countermeasures for the table based AES implementation. The 4K T tables are computed (once) at runtime to avoid various cache based attacks which are possible due to shared VMM mappings of read only tables. Additionally every cache line of the table is read from prior to processing the block(s).
- Support for the insecure ECC groups secp112r1, secp112r2, secp128r1, and secp128r2 has been removed.
- The portable version of GCM has been changed to run using only constant time operations.
- Work around a bug in MSVC 2013 std::mutex which on some Windows versions can result in a deadlock during static initialization. On Windows a CriticalSection is used instead. Analysis and patch from Matej Kenda (TopIT d.o.o.). GH #321
- The OpenSSL implementation of RC4 would return the wrong value from *name* if leading bytes of the keystream had been skipped in the output.
- Fixed the signature of the FFI function botan_pubkey_destroy, which took the wrong type and was not usable.
- The TLS client would erronously reject any server key exchange packet smaller than 6 bytes. This prevented negotiating a plain PSK TLS ciphersuite with an empty identity hint. ECDHE_PSK and DHE_PSK suites were not affected.
- Fixed a bug that would cause the TLS client to occasionally reject a valid server key exchange message as having an invalid signature. This only affected DHE and SRP ciphersuites.
- Support for negotiating use of SHA-224 in TLS has been disabled in the default policy.
- Added *remove_all* function to the *TLS::Session_Manager* interface
- Avoid GCC warning in pedantic mode when including bigint.h GH #330

15.19 Version 1.11.24, 2015-11-04

- When the bugs affecting X.509 path validation were fixed in 1.11.23, a check in Credentials_Manager::verify_certificate_chain was accidentally removed which caused path validation failures not to be signaled to the TLS layer. Thus in 1.11.23 certificate authentication in TLS is bypassed. Reported by Florent Le Coz in GH #324
- Fixed an endian dependency in McEliece key generation which caused keys to be generated differently on big and little endian systems, even when using a deterministic PRNG with the same seed.
- In *configure,py*, the flags for controlling use of debug, sanitizer, and converage information have been split out into individual options *–with-debug-info*, *–with-sanitizers*, and *–with-coverage*. These allow enabling more than one in a build in a controlled way. The *–build-mode* flag added in 1.11.17 has been removed.

15.20 Version 1.11.23, 2015-10-26

- CVE-2015-7824: An information leak allowed padding oracle attacks against TLS CBC decryption. Found in a review by Sirrix AG and 3curity GmbH.
- CVE-2015-7825: Validating a malformed certificate chain could cause an infinite loop. Found in a review by Sirrix AG and 3curity GmbH.
- CVE-2015-7826: X.509 path validation violated RFC 6125 and would accept certificates which should not validate under those rules. In particular botan would accept wildcard certificates as matching in situations where it should not (for example it would erroneously accept `*.example.com` as a valid wildcard for `foo.bar.example.com`)
- CVE-2015-7827: The routines for decoding PKCS #1 encryption and OAEP blocks have been rewritten to run without secret indexes or branches. These cryptographic operations are vulnerable to oracle attacks, including via side channels such as timing or cache-based analysis. In theory it would be possible to attack the previous implementations using such a side channel, which could allow an attacker to mount a plaintext recovery attack.

  By writing the code such that it does not depend on secret inputs for branch or memory indexes, such a side channel would be much less likely to exist.

  The OAEP code has previously made an attempt at constant time operation, but it used a construct which many compilers converted into a conditional jump.
- Add support for using ctgrind (https://github.com/agl/ctgrind) to test that sections of code do not use secret inputs to decide branches or memory indexes. The testing relies on dynamic checking using valgrind.

  So far PKCS #1 decoding, OAEP decoding, Montgomery reduction, IDEA, and Curve25519 have been notated and confirmed to be constant time on Linux/x86-64 when compiled by gcc.
- Public key operations can now be used with specified providers by passing an additional parameter to the constructor of the PK operation.
- OpenSSL RSA provider now supports signature creation and verification.
- The blinding code used for RSA, Diffie-Hellman, ElGamal and Rabin-Williams now periodically reinitializes the sequence of blinding values instead of always deriving the next value by squaring the previous ones. The reinitializion interval can be controlled by the build.h parameter BOTAN_BLINDING_REINIT_INTERVAL.
- A bug decoding DTLS client hellos prevented session resumption for succeeding.
- DL_Group now prohibits creating a group smaller than 1024 bits.
- Add System_RNG type. Previously the global system RNG was only accessible via *system_rng* which returned a reference to the object. However is at times useful to have a unique_ptr<RandomNumberGenerator> which

will be either the system RNG or an AutoSeeded_RNG, depending on availability, which this additional type allows.

- New command line tools *dl_group* and *prime*
- The *configure.py* option *–no-autoload* is now also available under the more understandable name *–minimized-build*.
- Note: 1.11.22 was briefly released on 2015-10-26. The only difference between the two was a fix for a compilation problem in the OpenSSL RSA code. As the 1.11.22 release had already been tagged it was simpler to immediately release 1.11.23 rather than redo the release.

15.21 Version 1.11.21, 2015-10-11

- Add new methods for creating types such as BlockCiphers or HashFunctions, T::providers() returning list of provider for a type, and T::create() creating a new object of a specified provider. The functions in lookup.h forward to these new APIs. A change to the lookup system in 1.11.14 had caused problems with static libraries (GH #52). These problems have been fixed as part of these changes. GH #279
- Fix loading McEliece public or private keys with PKCS::load_key / X509::load_key
- Add *mce* command line tool for McEliece key generation and file encryption
- Add Darwin_SecRandom entropy source which uses *SecRandomCopyBytes* API call for OS X and iOS, as this call is accessible even from a sandboxed application. GH #288
- Add new HMAC_DRBG constructor taking a name for the MAC to use, rather than a pointer to an object.
- The OCaml module is now a separate project at https://github.com/randombit/botan-ocaml
- The encrypted sqlite database support in contrib has moved to https://github.com/randombit/botan-sqlite
- The Perl XS module has been removed as it was no longer maintained.

15.22 Version 1.11.20, 2015-09-07

- Additional countermeasures were added to ECC point multiplications including exponent blinding and randomization of the point representation to help protect against side channel attacks.
- An ECDSA provider using OpenSSL has been added.
- The ordering of algorithm priorities has been reversed. Previously 255 was the lowest priority and 0 was the highest priority. Now it is the reverse, with 0 being lowest priority and 255 being highest. The default priority for the base algorithms is 100. This only affects external providers or applications which directly set provider preferences.
- On OS X, rename libs to avoid trailing version numbers, e.g. libbotan-1.11.dylib.19 -> libbotan-1.11.19.dylib. This was requested by the Homebrew project package audit. GH #241, #260
- Enable use of CPUID interface with clang. GH #232
- Add support for MSVC 2015 debug builds by satisfying C++ allocator requirements. SO 31802806, GH #236
- Make *X509_Time* string parsing and *to_u32bit()* more strict to avoid integer overflows and other potentially dangerous misinterpretations. GH #240, #243
- Remove all ‘extern “C”’ declarations from src/lib/math/mp/ because some of those did throw exceptions and thus cannot be C methods. GH #249
- Fix build configuration for clang debug on Linux. GH #250

- Fix zlib error when compressing an empty buffer. GH #265
- Fix iOS builds by allowing multiple compiler flags with the same name. GH #266
- Fix Solaris build issue caused by *RLIMIT_MEMLOCK*. GH #262

15.23 Version 1.11.19, 2015-08-03

- SECURITY: The BER decoder would crash due to reading from offset 0 of an empty vector if it encountered a BIT STRING which did not contain any data at all. As the type requires a 1 byte field this is not valid BER but could occur in malformed data. Found with afl. CVE-2015-5726
- SECURITY: The BER decoder would allocate a fairly arbitrary amount of memory in a length field, even if there was no chance the read request would succeed. This might cause the process to run out of memory or invoke the OOM killer. Found with afl. CVE-2015-5727
- The TLS heartbeat extension is deprecated and unless strong arguments are raised in its favor it will be removed in a future release. Comment at https://github.com/randombit/botan/issues/187
- The x86-32 assembly versions of MD4, MD5, SHA-1, and Serpent and the x86-64 version of SHA-1 have been removed. With compilers from this decade the C++ versions are significantly faster. The SSE2 versions of SHA-1 and Serpent remain, as they are still the fastest version for processors with SIMD extensions. GH #216
- BigInt::to_u32bit would fail if the value was exactly 32 bits. GH #220
- Botan is now fully compaitible with _GLIBCXX_DEBUG. GH #73
- BigInt::random_integer distribution was not uniform. GH #108
- Added unit testing framework Catch. GH #169
- Fix *make install*. GH #181, #186
- Public header *fs.h* moved to *internal/filesystem.h*. Added filesystem support for MSVC 2013 when boost is not available, allowing tests to run on those systems. GH #198, #199
- Added os "android" and fix Android compilation issues. GH #203
- Drop support for Python 2.6 for all Botan Python scripts. GH #217

15.24 Version 1.10.10, 2015-08-03

- SECURITY: The BER decoder would crash due to reading from offset 0 of an empty vector if it encountered a BIT STRING which did not contain any data at all. As the type requires a 1 byte field this is not valid BER but could occur in malformed data. Found with afl. CVE-2015-5726
- SECURITY: The BER decoder would allocate a fairly arbitrary amount of memory in a length field, even if there was no chance the read request would succeed. This might cause the process to run out of memory or invoke the OOM killer. Found with afl. CVE-2015-5727
- Due to an ABI incompatible (though not API incompatible) change in this release, the version number of the shared object has been increased.
- The default TLS policy no longer allows RC4.
- Fix a signed integer overflow in Blue Midnight Wish that may cause incorrect computations or undefined behavior.

15.25 Version 1.11.18, 2015-07-05

- In this release Botan has switched VCS from `monotone` to `git`, and is now hosted on github at https://github.com/randombit/botan
- The TLS client called `std::set_difference` on an invalid iterator pair. This could potentially lead to a crash depending on the compiler and STL implementation. It also would trigger assertion failures when using checked iterators. GH #73
- Remove code constructs which triggered errors under MSVC and GCC debug iterators. The primary of these was an idiom of `&vec[x]` to create a pointer offset of a `std::vector`. This failed when x was set equal to `vec.size()` to create the one-past-the-end address. The pointer in question was never dereferenced, but it triggered the iterator debugging checks which prevented using these valuble analysis tools. From Simon Warta and Daniel Seither. GH #125
- Several incorrect or missing module dependencies have been fixed. These often prevented a successful build of a minimized amalgamation when only a small set of algorithms were specified. GH #71 From Simon Warta.
- Add an initial binding to OCaml. Currently only hashes, RNGs, and bcrypt are supported.
- The default key size generated by the `keygen` tool has increased to 2048 bits. From Rene Korthaus.
- The `Botan_types` namespace, which contained `using` declarations for (just) `Botan::byte` and `Botan::u32bit`, has been removed. Any use should be replaced by `using` declarations for those types directly.

15.26 Version 1.11.17, 2015-06-18

- All support for the insecure RC4 stream cipher has been removed from the TLS implementation.
- Fix decoding of TLS maximum fragment length. Regardless of what value was actually negotiated, TLS would treat it as a negotiated limit of 4096.
- Fix the configure.py flag `--disable-aes-ni` which did nothing of the sort.
- Fixed nmake clean target. GitHub #104
- Correct buffering logic in `Compression_Filter`. GitHub #93 and #95

15.27 Version 1.11.16, 2015-03-29

- TLS has changed from using the non-standard NPN extension to the IETF standardized ALPN extension for negotiating an application-level protocol. Unfortunately the semantics of the exchange have changed with ALPN. Using NPN, the server offered a list of protocols it advertised, and then the client chose its favorite. With ALPN, the client offers a list of protocols and the server chooses. The the signatures of both the TLS::Client and TLS::Server constructors have changed to support this new flow.
- Optimized ECDSA signature verification thanks to an observation by Dr. Falko Strenzke. On some systems verifications are between 1.5 and 2 times faster than in 1.11.15.
- RSA encrypt and decrypt operations using OpenSSL have been added.
- Public key operation types now handle all aspects of the operation, such as hashing and padding for signatures. This change allows supporting specialized implementations which only support particular padding types.

- Added global timeout to HMAC_RNG entropy reseed. The defaults are the values set in the build.h macros `BOTAN_RNG_AUTO_RESEED_TIMEOUT` and `BOTAN_RNG_RESEED_DEFAULT_TIMEOUT`, but can be overriden on a specific poll with the new API call reseed_with_timeout.
- Fixed Python cipher update_granularity() and default_nonce_length() functions
- The library now builds on Visual C++ 2013
- The GCM update granularity was reduced from 4096 to 16 bytes.
- Fix a bug that prevented building the amalgamation until a non-amalgamation configuration was performed first in the same directory.
- Add Travis CI integration. Github pull 60.

15.28 Version 1.11.15, 2015-03-08

- Support for RC4 in TLS, already disabled by default, is now deprecated. The RC4 ciphersuites will be removed entirely in a future release.
- A bug in ffi.cpp meant Python could only encrypt. Github issue 53.
- When comparing two ASN.1 algorithm identifiers, consider empty and NULL parameters the same.
- Fixed memory leaks in TLS and cipher modes introduced in 1.11.14
- MARK-4 failed when OpenSSL was enabled in the build in 1.11.14 because the OpenSSL version ignored the skip parameter.
- Fix compilation problem on OS X/clang
- Use BOTAN_NOEXCEPT macro to work around lack of noexcept in VS 2013

15.29 Version 1.11.14, 2015-02-27

- The global state object previously used by the library has been removed. This includes the global PRNG. The library can be safely initialized multiple times without harm.

  The engine code has also been removed, replaced by a much lighter-weight object registry system which provides lookups in faster time and with less memory overhead than the previous approach.

  One caveat of the current system with regards to static linking: because only symbols already mentioned elsewhere in the program are included in the final link step, few algorithms will be available through the lookup system by default, even though they were compiled into the library. Your application must explicitly reference the types you require or they will not end up being available in the final binary. See also Github issue #52

  If you intend to build your application against a static library and don't want to explicitly reference each algo object you might attempt to look up by string, consider either building with `--via-amalgamation`, or else (much simpler) using the amalgamation directly.
- The new `ffi` submodule provides a simple C API/ABI for a number of useful operations (hashing, ciphers, public key operations, etc) which is easily accessed using the FFI modules included in many languages.
- A new Python wrapper (in `src/lib/python/botan.py`) using `ffi` and the Python `ctypes` module is available. The old Boost.Python wrapper has been removed.
- Add specialized reducers for P-192, P-224, P-256, and P-384

- OCB mode, which provides a fast and constant time AEAD mode without requiring hardware support, is now supported in TLS, following draft-zauner-tls-aes-ocb-01. Because this specification is not yet finalized is not yet enabled by the default policy, and the ciphersuite numbers used are in the experimental range and may conflict with other uses.
- Add ability to read TLS policy from a text file using `TLS::Text_Policy`.
- The amalgamation now splits off any ISA specific code (for instance, that requiring SSSE3 instruction sets) into a new file named (for instance) `botan_all_ssse3.cpp`. This allows the main amalgamation file to be compiled without any special flags, so `--via-amalgamation` builds actually work now. This is disabled with the build option `--single-amalgamation-file`
- PBKDF and KDF operations now provide a way to write the desired output directly to an application-specified area rather than always allocating a new heap buffer.
- HKDF, previously provided using a non-standard interface, now uses the standard KDF interface and is retrievable using get_kdf.
- It is once again possible to build the complete test suite without requiring any boost libraries. This is currently only supported on systems supporting the readdir interface.
- Remove use of memset_s which caused problems with amalgamation on OS X. Github 42, 45
- The memory usage of the counter mode implementation has been reduced. Previously it encrypted 256 blocks in parallel as this leads to a slightly faster counter increment operation. Instead CTR_BE simply encrypts a buffer equal in size to the advertised parallelism of the cipher implementation. This is not measurably slower, and dramatically reduces the memory use of CTR mode.
- The memory allocator available on Unix systems which uses mmap and mlock to lock a pool of memory now checks environment variable BOTAN_MLOCK_POOL_SIZE and interprets it as an integer. If the value set to a smaller value then the library would originally have allocated (based on resource limits) the user specified size is used instead. You can also set the variable to 0 to disable the pool entirely. Previously the allocator would consume all available mlocked memory, this allows botan to coexist with an application which wants to mlock memory for its own uses.
- The botan-config script previously installed on Unix systems has been removed. Its functionality is replaced by the `config` command of the `botan` tool executable, for example `botan config cflags` instead of `botan-config --cflags`.
- Added a target for POWER8 processors

15.30 Version 1.11.13, 2015-01-11

- All support for the insecure SSLv3 protocol and the server support for processing SSLv2 client hellos has been removed.
- The command line tool now has `tls_proxy` which negotiates TLS with clients and forwards the plaintext to a specified port.
- Add MCEIES, a McEliece-based integrated encryption system using AES-256 in OCB mode for message encryption/authentication.
- Add DTLS-SRTP negotiation defined in RFC 5764
- Add SipHash
- Add SHA-512/256
- The format of serialized TLS sessions has changed. Additiionally, PEM formatted sessions now use the label of "TLS SESSION" instead of "SSL SESSION"

- Serialized TLS sessions are now encrypted using AES-256/GCM instead of a CBC+HMAC construction.
- The cryptobox_psk module added in 1.11.4 and previously used for TLS session encryption has been removed.
- When sending a TLS heartbeat message, the number of pad bytes to use can now be specified, making it easier to use for PMTU discovery.
- If available, zero_mem now uses RtlSecureZeroMemory or memset_s instead of a byte-at-a-time loop.
- The functions base64_encode and base64_decode would erroneously throw an exception if passed a zero-length input. Github issue 37.
- The Python install script added in version 1.11.10 failed to place the headers into a versioned subdirectory.
- Fix the install script when running under Python3.
- Avoid code that triggers iterator debugging asserts under MSVC 2013. Github pull 36 from Simon Warta.

15.31 Version 1.11.12, 2015-01-02

- Add Curve25519. The implementation is based on curve25519-donna-c64.c by Adam Langley. New (completely non-standard) OIDs and formats for encrypting Curve25519 keys under PKCS #8 and including them in certificates and CRLs have been defined.
- Add Poly1305, based on the implementation poly1305-donna by Andrew Moon.
- Add the ChaCha20Poly1305 AEADs defined in draft-irtf-cfrg-chacha20-poly1305-03 and draft-agl-tls-chacha20poly1305-04.
- Add ChaCha20Poly1305 ciphersuites for TLS compatible with Google's servers following draft-agl-tls-chacha20poly1305-04
- When encrypted as PKCS #8 structures, Curve25519 and McEliece private keys default to using AES-256/GCM instead of AES-256/CBC
- Define OIDs for OCB mode with AES, Serpent and Twofish.

15.32 Version 1.11.11, 2014-12-21

- The Sqlite3 wrapper has been abstracted to a simple interface for SQL dbs in general, though Sqlite3 remains the only implementation. The main logic of the TLS session manager which stored encrypted sessions to a Sqlite3 database (`TLS::Session_Manager_SQLite`) has been moved to the new `TLS::Session_Manager_SQL`. The Sqlite3 manager API remains the same but now just subclasses `TLS::Session_Manager_SQL` and has a constructor instantiate the concrete database instance.

  Applications which would like to use a different db can now do so without having to reimplement the session cache logic simply by implementing a database wrapper subtype.
- The CryptGenRandom entropy source is now also used on MinGW.
- The system_rng API is now also available on systems with CryptGenRandom
- With GCC use -fstack-protector for linking as well as compiling, as this is required on MinGW. Github issue 34.
- Fix missing dependency in filters that caused compilation problem in amalgamation builds. Github issue 33.
- SSLv3 support is officially deprecated and will be removed in a future release.

15.33 Version 1.10.9, 2014-12-13

- Fixed EAX tag verification to run in constant time
- The default TLS policy now disables SSLv3.
- A crash could occur when reading from a blocking random device if the device initially indicated that entropy was available but a concurrent process drained the entropy pool before the read was initiated.
- Fix decoding indefinite length BER constructs that contain a context sensitive tag of zero. Github pull 26 from Janusz Chorko.
- The `botan-config` script previously tried to guess its prefix from the location of the binary. However this was error prone, and now the script assumes the final installation prefix matches the value set during the build. Github issue 29.

15.34 Version 1.11.10, 2014-12-10

- An implementation of McEliece code-based public key encryption based on INRIA's HyMES and secured against a variety of side-channels was contributed by cryptosource GmbH. The original version is LGPL but cryptosource has secured permission to release an adaptation under a BSD license. A CCA2-secure KEM scheme is also included.

  The implementation is further described in http://www.cryptosource.de/docs/mceliece_in_botan.pdf and http://cryptosource.de/news_mce_in_botan_en.html
- DSA and ECDSA now create RFC 6979 deterministic signatures.
- Add support for TLS fallback signaling (draft-ietf-tls-downgrade-scsv-00). Clients will send a fallback SCSV if the version passed to the Client constructor is less than the latest version supported by local policy, so applications implementing fallback are protected. Servers always check the SCSV.
- In previous versions a TLS::Server could service either TLS or DTLS connections depending on policy settings and what type of client hello it received. This has changed and now a Server object is initialized for either TLS or DTLS operation. The default policy previously prohibited DTLS, precisely to prevent a TCP server from being surprised by a DTLS connection. The default policy now allows TLS v1.0 or higher or DTLS v1.2.
- Fixed a bug in CCM mode which caused it to produce incorrect tags when used with a value of L other than 2. This affected CCM TLS ciphersuites, which use L=3. Thanks to Manuel Pégourié-Gonnard for the anaylsis and patch. Bugzilla 270.
- DTLS now supports timeouts and handshake retransmits. Timeout checking is triggered by the application calling the new TLS::Channel::timeout_check.
- Add a TLS policy hook to disable putting the value of the local clock in hello random fields.
- All compression operations previously available as Filters are now performed via the Transformation API, which minimizes memory copies. Compression operations are still available through the Filter API using new general compression/decompression filters in comp_filter.h
- The zlib module now also supports gzip compression and decompression.
- Avoid a crash in low-entropy situations when reading from /dev/random, when select indicated the device was readable but by the time we start the read the entropy pool had been depleted.
- The Miller-Rabin primality test function now takes a parameter allowing the user to directly specify the maximum false negative probability they are willing to accept.
- PKCS #8 private keys can now be encrypted using GCM mode instead of unauthenticated CBC. The default remains CBC for compatibility.

- The default PKCS #8 encryption scheme has changed to use PBKDF2 with SHA-256 instead of SHA-1
- A specialized reducer for P-521 was added.
- On Linux the mlock allocator will use MADV_DONTDUMP on the pool so that the contents are not included in coredumps.
- A new interface for directly using a system-provided PRNG is available in system_rng.h. Currently only systems with /dev/urandom are supported.
- Fix decoding indefinite length BER constructs that contain a context sensitive tag of zero. Github pull 26 from Janusz Chorko.
- The GNU MP engine has been removed.
- Added AltiVec detection for POWER8 processors.
- Add a new install script written in Python which replaces shell hackery in the makefiles.
- Various modifications to better support Visual C++ 2013 and 2015. Github issues 11, 17, 18, 21, 22.

15.35 Version 1.10.8, 2014-04-10

- SECURITY: Fix a bug in primality testing introduced in 1.8.3 which caused only a single random base, rather than a sequence of random bases, to be used in the Miller-Rabin test. This increased the probability that a non-prime would be accepted, for instance a 1024 bit number would be incorrectly classed as prime with probability around 2^-40. Reported by Jeff Marrison. CVE-2014-9742
- The key length limit on HMAC has been raised to 512 bytes, allowing the use of very long passphrases with PBKDF2.

15.36 Version 1.11.9, 2014-04-10

- SECURITY: Fix a bug in primality testing introduced in 1.8.3 which caused only a single random base, rather than a sequence of random bases, to be used in the Miller-Rabin test. This increased the probability that a non-prime would be accepted, for instance a 1024 bit number would be incorrectly classed as prime with probability around 2^-40. Reported by Jeff Marrison. CVE-2014-9742
- X.509 path validation now returns a set of all errors that occurred during validation, rather than immediately returning the first detected error. This prevents a seemingly innocuous error (such as an expired certificate) from hiding an obviously serious error (such as an invalid signature). The Certificate_Status_Code enum is now ordered by severity, and the most severe error is returned by Path_Validation_Result::result(). The entire set of status codes is available with the new all_statuses call.
- Fixed a bug in OCSP response decoding which would cause an error when attempting to decode responses from some widely used responders.
- An implementation of HMAC_DRBG RNG from NIST SP800-90A has been added. Like the X9.31 PRNG implementation, it uses another underlying RNG for seeding material.
- An implementation of the RFC 6979 deterministic nonce generator has been added.
- Fix a bug in certificate path validation which prevented successful validation if intermediate certificates were presented out of order.
- Fix a bug introduced in 1.11.5 which could cause crashes or other incorrect behavior when a cipher mode filter was followed in the pipe by another filter, and that filter had a non-empty start_msg.
- The types.h header now uses stdint.h rather than cstdint to avoid problems with Clang on OS X.

15.37 Version 1.11.8, 2014-02-13

- The `botan` command line application introduced in 1.11.7 is now installed along with the library.
- A bug in certificate path validation introduced in 1.11.6 which caused all CRL signature checks to fail has been corrected.
- The ChaCha20 stream cipher has been added.
- The `Transformation` class no longer implements an interface for keying, this has been moved to a new subclass `Keyed_Transformation`.
- The `Algorithm` class, which previously acted as a global base for various types (ciphers, hashes, etc) has been removed.
- CMAC now supports 256 and 512 bit block ciphers, which also allows the use of larger block ciphers with EAX mode. In particular this allows using Threefish in EAX mode.
- The antique PBES1 private key encryption scheme (which only supports DES or 64-bit RC2) has been removed.
- The Square, Skipjack, and Luby-Rackoff block ciphers have been removed.
- The Blue Midnight Wish hash function has been removed.
- Skein-512 no longer supports output lengths greater than 512 bits.
- Skein did not reset its internal state properly if clear() was called, causing it to produce incorrect results for the following message. It was reset correctly in final() so most usages should not be affected.
- A number of public key padding schemes have been renamed to match the most common notation; for instance EME1 is now called OAEP and EMSA4 is now called PSSR. Aliases are set which should allow all current applications to continue to work unmodified.
- A bug in CFB encryption caused a few bytes past the end of the final block to be read. The actual output was not affected.
- Fix compilation errors in the tests that occurred with minimized builds. Contributed by Markus Wanner.
- Add a new `--destdir` option to `configure.py` which controls where the install target will place the output. The `--prefix` option continues to set the location where the library expects to be eventually installed.
- Many class destructors which previously deleted memory have been removed in favor of using `unique_ptr`.
- Various portability fixes for Clang, Windows, Visual C++ 2013, OS X, and x86-32.

15.38 Version 1.11.7, 2014-01-10

- Botan's basic numeric types are now defined in terms of the C99/C++11 standard integer types. For instance `u32bit` is now a typedef for `uint32_t`, and both names are included in the library namespace. This should not result in any application-visible changes.
- There are now two executable outputs of the build, `botan-test`, which runs the tests, and `botan` which is used as a driver to call into various subcommands which can also act as examples of library use, much in the manner of the `openssl` command. It understands the commands `base64`, `asn1`, `x509`, `tls_client`, `tls_server`, `bcrypt`, `keygen`, `speed`, and various others. As part of this change many obsolete, duplicated, or one-off examples were removed, while others were extended with new functionality. Contributions of new subcommands, new bling for exising ones, or documentation in any form is welcome.
- Fix a bug in Lion, which was broken by a change in 1.11.0. The problem was not noticed before as Lion was also missing a test vector in previous releases.

15.39 Version 1.10.7, 2013-12-29

- OAEP had two bugs, one of which allowed it to be used even if the key was too small, and the other of which would cause a crash during decryption if the EME data was too large for the associated key.

15.40 Version 1.11.6, 2013-12-29

- The Boost filesystem and asio libraries are now being used by default. Pass `--without-boost` to `configure.py` to disable.
- The default TLS policy no longer allows SSLv3 or RC4.
- OAEP had two bugs, one of which allowed it to be used even if the key was too small, and the other of which would cause a crash during decryption if the EME data was too large for the associated key.
- GCM mode now uses the Intel clmul instruction when available
- Add the Threefish-512 tweakable block cipher, including an AVX2 version
- Add SIV (from **RFC 5297** (https://tools.ietf.org/html/rfc5297.html)) as a nonce-based AEAD
- Add HKDF (from **RFC 5869** (https://tools.ietf.org/html/rfc5869.html)) using an experimental PRF interface
- Add HTTP utility functions and OCSP online checking
- Add TLS::Policy::acceptable_ciphersuite hook to disable ciphersuites on an ad-hoc basis.
- TLS::Session_Manager_In_Memory's constructor now requires a RNG

15.41 Version 1.10.6, 2013-11-10

- The device reading entropy source now attempts to read from all available devices. Previously it would break out early if a partial read from a blocking source occurred, not continuing to read from a non-blocking device. This would cause the library to fall back on slower and less reliable techniques for collecting PRNG seed material. Reported by Rickard Bellgrim.
- HMAC_RNG (the default PRNG implementation) now automatically reseeds itself periodically. Previously reseeds only occurred on explicit application request.
- Fix an encoding error in EC_Group when encoding using EC_DOMPAR_ENC_OID. Reported by fxdupont on github.
- In EMSA2 and Randpool, avoid calling name() on objects after deleting them if the provided algorithm objects are not suitable for use. Found by Clang analyzer, reported by Jeffrey Walton.
- If X509_Store was copied, the u32bit containing how long to cache validation results was not initialized, potentially causing results to be cached for significant amounts of time. This could allow a certificate to be considered valid after its issuing CA's cert expired. Expiration of the end-entity cert is always checked, and reading a CRL always causes the status to be reset, so this issue does not affect revocation. Found by Coverity scanner.
- Avoid off by one causing a potentially unterminated string to be passed to the connect system call if the library was configured to use a very long path name for the EGD socket. Found by Coverity Scanner.
- In PK_Encryptor_EME, PK_Decryptor_EME, PK_Verifier, and PK_Key_Agreement, avoid dereferencing an unitialized pointer if no engine supported operations on the key object given. Found by Coverity scanner.
- Avoid leaking a file descriptor in the /dev/random and EGD entropy sources if stdin (file descriptor 0) was closed. Found by Coverity scanner.

- Avoid a potentially undefined operation in the bit rotation operations. Not known to have caused problems under any existing compiler, but might have caused problems in the future. Caught by Clang sanitizer, reported by Jeffrey Walton.
- Increase default hash iterations from 10000 to 50000 in PBES1 and PBES2
- Add a fix for mips64el builds from Brad Smith.

15.42 Version 1.11.5, 2013-11-10

- The TLS callback signatures have changed - there are now two distinct callbacks for application data and alerts. TLS::Client and TLS::Server have constructors which continue to accept the old callback and use it for both operations.
- The entropy collector that read from randomness devices had two bugs - it would break out of the poll as soon as any read succeeded, and it selected on each device individually. When a blocking source was first in the device list and the entropy pool was running low, the reader might either block in select until eventually timing out (continuing on to read from /dev/urandom instead), or read just a few bytes, skip /dev/urandom, fail to satisfy the entropy target, and the poll would continue using other (slower) sources. This caused substantial performance/latency problems in RNG heavy applications. Now all devices are selected over at once, with the effect that a full read from urandom always occurs, along with however much (if any) output is available from blocking sources.
- Previously AutoSeeded_RNG referenced a globally shared PRNG instance. Now each instance has distinct state.
- The entropy collector that runs Unix programs to collect statistical data now runs multiple processes in parallel, greatly reducing poll times on some systems.
- The Randpool RNG implementation was removed.
- All existing cipher mode implementations (such as CBC and XTS) have been converted from filters to using the interface previously provided by AEAD modes which allows for in-place message processing. Code which directly references the filter objects will break, but an adaptor filter allows usage through get_cipher as usual.
- An implementation of CCM mode from RFC 3601 has been added, as well as CCM ciphersuites for TLS.
- The implementation of OCB mode now supports 64 and 96 bit tags
- Optimized computation of XTS tweaks, producing a substantial speedup
- Add support for negotiating Brainpool ECC curves in TLS
- TLS v1.2 will not negotiate plain SHA-1 signatures by default.
- TLS channels now support sending a `std::vector`
- Add a generic 64x64->128 bit multiply instruction operation in mul128.h
- Avoid potentially undefined operations in the bit rotation operations. Not known to have caused problems under existing compilers but might break in the future. Found by Clang sanitizer, reported by Jeffrey Walton.

15.43 Version 1.11.4, 2013-07-25

- CPU specific extensions are now always compiled if support for the operations is available at build time, and flags enabling use of extra operations (such as SSE2) are only included when compiling files which specifically request support. This means, for instance, that the SSSE3 and AES-NI implementations of AES are always included in x86 builds, relying on runtime cpuid checking to prevent their use on CPUs that do not support those operations.
- The default TLS policy now only accepts TLS, to minimize surprise for servers which might not expect to negotiate DTLS. Previously a server would by default negotiate either protocol type (clients would only accept the same protocol type as they offered). Applications which use DTLS or combined TLS/DTLS need to override `Policy::acceptable_protocol_version`.
- The TLS channels now accept a new parameter specifying how many bytes to preallocate for the record handling buffers, which allows an application some control over how much memory is used at runtime for a particular connection.
- Applications can now send arbitrary TLS alert messages using `TLS::Channel::send_alert`
- A new TLS policy `NSA_Suite_B_128` is available, which will negotiate only the 128-bit security NSA Suite B. See **RFC 6460** (https://tools.ietf.org/html/rfc6460.html) for more information about Suite B.
- Adds a new interface for benchmarking, `time_algorithm_ops`, which returns a map of operations to operations per second. For instance now both encrypt and decrypt speed of a block cipher can be checked, as well as the key schedule of all keyed algorithms. It additionally supports AEAD modes.
- Rename ARC4 to RC4

15.44 Version 1.11.3, 2013-04-11

- Add a new interface for AEAD modes (`AEAD_Mode`).
- Implementations of the OCB and GCM authenticated cipher modes are now included.
- Support for TLS GCM ciphersuites is now available.
- A new TLS policy mechanism `TLS::Policy::server_uses_own_ciphersuite_preferences` controls how a server chooses a ciphersuite. Previously it always chose its most preferred cipher out of the client's list, but this can allow configuring a server to choose by the client's preferences instead.
- `Keyed_Filter` now supports returning a `Key_Length_Specification` so the full details of what keylengths are supported is now available in keyed filters.
- The experimental and rarely used Turing and WiderWAKE stream ciphers have been removed
- New functions for symmetric encryption are included in cryptobox.h though interfaces and formats are subject to change.
- A new function `algorithm_kat_detailed` returns a string providing information about failures, instead of just a pass/fail indicator as in `algorithm_kat`.

15.45 Version 1.10.5, 2013-03-02

- A potential crash in the AES-NI implementation of the AES-192 key schedule (caused by misaligned loads) has been fixed.
- A previously conditional operation in Montgomery multiplication and squaring is now always performed, removing a possible timing channel.
- Use correct flags for creating a shared library on OS X under Clang.
- Fix a compile time incompatibility with Visual C++ 2012.

15.46 Version 1.11.2, 2013-03-02

- A bug in the release script caused the `botan_version.py` included in 1.11.1`` to be invalid, which required a manual edit to fix (Bugzilla 226)
- Previously `clear_mem` was implemented by an inlined call to `std::memset`. However an optimizing compiler might notice cases where the memset could be skipped in cases allowed by the standard. Now `clear_mem` calls `zero_mem` which is compiled separately and which zeros out the array through a volatile pointer. It is possible some compiler with some optimization setting (especially with something like LTO) might still skip the writes. It would be nice if there was an automated way to test this.
- The new filter `Threaded_Fork` acts like a normal `Fork`, sending its input to a number of different filters, but each subchain of filters in the fork runs in its own thread. Contributed by Joel Low.
- The default TLS policy formerly preferred AES over RC4, and allowed 3DES by default. Now the default policy is to negotiate only either AES or RC4, and to prefer RC4.
- New TLS `Blocking_Client` provides a thread per connection style API similar to that provided in 1.10
- The API of `Credentials_Manager::trusted_certificate_authorities` has changed to return a vector of `Certificate_Store*` instead of `X509_Certificate`. This allows the list of trusted CAs to be more easily updated dynamically or loaded lazily.
- The `asn1_int.h` header was split into `asn1_alt_name.h`, `asn1_attribute.h` and `asn1_time.h`.

15.47 Version 1.10.4, 2013-01-07

- Avoid a conditional operation in the power mod implementations on if a nibble of the exponent was zero or not. This may help protect against certain forms of side channel attacks.
- The SRP6 code was checking for invalid values as specified in RFC 5054, specifically values equal to zero mod p. However SRP would accept negative A/B values, or ones larger than p, neither of which should occur in a normal run of the protocol. These values are now rejected. Credits to Timothy Prepscius for pointing out these values are not normally used and probably signal something fishy.
- The return value of version_string is now a compile time constant string, so version information can be more easily extracted from binaries.

15.48 Version 1.11.1, 2012-10-30

Initial support for DTLS (both v1.0 and v1.2) is available in this release, though it should be considered highly experimental. Currently timeouts and retransmissions are not handled.

The `TLS::Client` constructor now takes the version to offer to the server. The policy hook `TLS::Policy` function *pref_version* `, which previously controlled this, has been removed.

TLS::Session_Manager_In_Memory ` now chooses a random 256-bit key at startup and encrypts all sessions (using the existing *TLS::Session::encrypt* ` mechanism) while they are stored in memory. This is primarily to reduce pressure on locked memory, as each session normally requires 48 bytes of locked memory for the master secret, whereas now only 32 bytes are needed total. This change may also make it slightly harder for an attacker to extract session data from memory dumps (eg with a cold boot attack).

The keys used in TLS session encryption were previously uniquely determined by the master key. Now the encrypted session blob includes two 80 bit salts which are used in the derivation of the cipher and MAC keys.

The `secure_renegotiation` flag is now considered an aspect of the connection rather than the session, which matches the behavior of other implementations. As the format has changed, sessions saved to persistent storage by 1.11.0 will not load in this version and vice versa. In either case this will not cause any errors, the session will simply not resume and instead a full handshake will occur.

New policy hooks `TLS::Policy::acceptable_protocol_version`, *TLS::Policy::allow_server_initiated_renegotiation* `, and *TLS::Policy::negotiate_heartbeat_support* ` were added.

TLS clients were not sending a next protocol message during a session resumption, which would cause resumption failures with servers that support NPN if NPN was being offered by the client.

A bug caused heartbeat requests sent by the counterparty during a handshake to be passed to the application callback as if they were heartbeat responses.

Support for TLS key material export as specified in **RFC 5705** (https://tools.ietf.org/html/rfc5705.html) has been added, available via `TLS::Channel::key_material_export`

A new function `Public_Key::estimated_strength` returns an estimate for the upper bound of the strength of the key. For instance for an RSA key, it will return an estimate of how many operations GNFS would take to factor the key.

A new `Path_Validation_Result` code has been added `SIGNATURE_METHOD_TOO_WEAK`. By default signatures created with keys below 80 bits of strength (as estimated by `estimated_strength`) are rejected. This level can be modified using a parameter to the `Path_Validation_Restrictions` constructor.

The SRP6 code was checking for invalid values as specified in **RFC 5054** (https://tools.ietf.org/html/rfc5054.html), ones equal to zero mod p, however it would accept negative A/B values, or ones larger than p, neither of which should occur in a normal run of the protocol. These values are now rejected. Credits to Timothy Prepscius for pointing out these values are not normally used and probably signal something fishy.

Several `BigInt` functions have been removed, including `operator[]`, `assign`, `get_reg`, and `grow_reg`. The version of `data` that returns a mutable pointer has been renamed `mutable_data`. Support for octal conversions has been removed.

The constructor `BigInt(NumberType type, size_t n)` has been removed, replaced by `BigInt::power_of_2`.

In 1.11.0, when compiled by GCC, the AES-NI implementation of AES-192 would crash if the mlock-based allocator was used due to an alignment issue.

15.49 Version 1.11.0, 2012-07-19

Note: In this release, many new features of C++11 are being used in the library. Currently GCC 4.7 and Clang 3.1 are known to work well. This version of the library cannot be compiled by or used with a C++98 compiler.

There have been many changes and improvements to TLS. The interface is now purely event driven and does not directly interact with sockets. New TLS features include TLS v1.2 support, client certificate authentication, renegotiation, session tickets, and session resumption. Session information can be saved in memory or to an encrypted SQLite3 database. Newly supported TLS ciphersuite algorithms include using SHA-2 for message authentication, pre shared keys and SRP for authentication and key exchange, ECC algorithms for key exchange and signatures, and anonymous DH/ECDH key exchange.

Support for OCSP has been added. Currently only client-side support exists.

The API for X.509 path validation has changed, with `x509_path_validate` in x509path.h now handles path validation and `Certificate_Store` handles storage of certificates and CRLs.

The memory container types have changed substantially. The `MemoryVector` and `SecureVector` container types have been removed, and an alias of `std::vector` using an allocator that clears memory named `secure_vector` is used for key material, with plain `std::vector` being used for everything else.

The technique used for mlock'ing memory on Linux and BSD systems is much improved. Now a single page-aligned block of memory (the exact limit of what we can mlock) is mmap'ed, with allocations being done using a best-fit allocator and all metadata held outside the mmap'ed range, in an effort to make best use of the very limited amount of memory current Linux kernels allow unpriveledged users to lock.

A filter using LZMA was contributed by Vojtech Kral. It is available if LZMA support was enabled at compilation time by passing `--with-lzma` to `configure.py`.

RFC 5915 (https://tools.ietf.org/html/rfc5915.html) adds some extended information which can be included in ECC private keys which the ECC key decoder did not expect, causing an exception when such a key was loaded. In particular, recent versions of OpenSSL use these fields. Now these fields are decoded properly, and if the public key value is included it is used, as otherwise the public key needs to be rederived from the private key. However the library does not include these fields on encoding keys for compatibility with software that does not expect them (including older versions of botan).

15.50 Version 1.8.14, 2012-07-18

- The malloc allocator would return null instead of throwing in the event of an allocation failure, which could cause an application crash due to null pointer dereference where normally an exception would occur.
- Recent versions of OpenSSL include extra information in ECC private keys, the presence of which caused an exception when such a key was loaded by botan. The decoding of ECC private keys has been changed to ignore these fields if they are set.
- AutoSeeded_RNG has been changed to prefer `/dev/random` over `/dev/urandom`
- Fix detection of s390x (Debian bug 638347)

15.51 Version 1.10.3, 2012-07-10

A change in 1.10.2 accidentally broke ABI compatibility with 1.10.1 and earlier versions, causing programs compiled against 1.10.1 to crash if linked with 1.10.2 at runtime.

Recent versions of OpenSSL include extra information in ECC private keys, the presence of which caused an exception when such a key was loaded by botan. The decoding of ECC private keys has been changed to ignore these fields if they are set.

15.52 Version 1.10.2, 2012-06-17

Several TLS bugs were fixed in this release, including a major omission that the renegotiation extension was not being used. As the 1.10 implementation of TLS does not properly support renegotiation, the approach in this release is simply to send the renegotiation extension SCSV, which should protect the client against any handshake splicing. In addition renegotiation attempts are handled properly instead of causing handshake failures - all hello requests, and all client hellos after the initial negotiation, are ignored. Some bugs affecting DSA server authentication were also fixed.

By popular request, `Pipe::reset` no longer requires that message processing be completed, a requirement that caused problems when a Filter's end_msg call threw an exception, after which point the Pipe object was no longer usable.

Support for getting entropy using the rdrand instruction introduced in Intel's Ivy Bridge processors has been added. In previous releases, the `CPUID::has_rdrand` function was checking the wrong cpuid bit, and would false positive on AMD Bulldozer processors.

An implementation of SRP-6a compatible with the specification in RFC 5054 is now available in `srp6.h`. In 1.11, this is being used for TLS-SRP, but may be useful in other environments as well.

An implementation of the Camellia block cipher was added, again largely for use in TLS.

If `clock_gettime` is available on the system, hres_timer will poll all the available clock types.

AltiVec is now detected on IBM POWER7 processors and on OpenBSD systems. The OpenBSD support was contributed by Brad Smith.

The Qt mutex wrapper was broken and would not compile with any recent version of Qt. Taking this as a clear indication that it is not in use, it has been removed.

Avoid setting the soname on OpenBSD, as it doesn't support it (Bugzilla 158)

A compilation problem in the dynamic loader that prevented using dyn_load under MinGW GCC has been fixed.

A common error for people using MinGW is to target GCC on Windows, however the 'Windows' target assumes the existence of Visual C++ runtime functions which do not exist in MinGW. Now, configuring for GCC on Windows will cause the configure.py to warn that likely you wanted to configure for either MinGW or Cygwin, not the generic Windows target.

A bug in configure.py would cause it to interpret `--cpu=s390x` as `s390`. This may have affected other CPUs as well. Now configure.py searches for an exact match, and only if no exact match is found will it search for substring matches.

An incompatibility in configure.py with the subprocess module included in Python 3.1 has been fixed (Bugzilla 157).

The exception catching syntax of configure.py has been changed to the Python 3.x syntax. This syntax also works with Python 2.6 and 2.7, but not with any earlier Python 2 release. A simple search and replace will allow running it under Python 2.5: `perl -pi -e 's/except (.*) as (.*):/except $1, $2:/g' configure.py`

Note that Python 2.4 is not supported at all.

15.53 Version 1.10.1, 2011-07-11

- A race condition in `Algorithm_Factory` could cause crashes in multithreaded code.
- The return value of `name` has changed for GOST 28147-89 and Skein-512. GOST's `name` now includes the name of the sbox, and Skein's includes the personalization string (if nonempty). This allows an object to be properly roundtripped, which is necessary to fix the race condition described above.
- A new distribution script is now included, as `src/build-data/scripts/dist.py`
- The `build.h` header now includes, if available, an identifier of the source revision that was used. This identifier is also included in the result of `version_string`.

15.54 Version 1.8.13, 2011-07-02

- A race condition in `Algorithm_Factory` could cause crashes in multithreaded code.

15.55 Version 1.10.0, 2011-06-20

- Detection for the rdrand instruction being added to upcoming Intel Ivy Bridge processors has been added.
- A template specialization of std::swap was added for the memory container types.

15.56 Version 1.8.12, 2011-06-20

- If EMSA3(Raw) was used for more than one signature, it would produce incorrect output.
- Fix the –enable-debug option to configure.py
- Improve OS detection on Cygwin
- Fix compilation under Sun Studio 12 on Solaris
- Fix a memory leak in the constructors of DataSource_Stream and DataSink_Stream which would occur if opening the file failed (Bugzilla 144)

15.57 Version 1.9.18, 2011-06-03

- Fourth release candidate for 1.10.0
- The GOST 34.10 verification operation was not ensuring that s and r were both greater than zero. This could potentially have meant it would have accepted an invalid all-zero signature as valid for any message. Due to how ECC points are internally represented it instead resulted in an exception being thrown.
- A simple multiexponentation algorithm is now used in ECDSA and GOST-34.10 signature verification, leading to 20 to 25% improvements in ECDSA and 25% to 40% improvements in GOST-34.10 verification performance.
- The internal representation of elliptic curve points has been modified to use Montgomery representation exclusively, resulting in reduced memory usage and a 10 to 20% performance improvement for ECDSA and ECDH.
- In OAEP decoding, scan for the delimiter bytes using a loop that is written without conditionals so as to help avoid timing analysis. Unfortunately GCC at least is 'smart' enough to compile it to jumps anyway.

- The SSE2 implementation of IDEA did not work correctly when compiled by Clang, because the trick it used to emulate a 16 bit unsigned compare in SSE (which doesn't contain one natively) relied on signed overflow working in the 'usual' way. A different method that doesn't rely on signed overflow is now used.
- Add support for compiling SSL using Visual C++ 2010's TR1 implementation.
- Fix a bug under Visual C++ 2010 which would cause `hex_encode` to crash if given a zero-sized input to encode.
- A new build option `--via-amalgamation` will first generate the single-file amalgamation, then build the library from that single file. This option requires a lot of memory and does not parallelize, but the resulting library is smaller and may be faster.
- On Unix, the library and header paths have been changed to allow parallel installation of different versions of the library. Headers are installed into `<prefix>/include/botan-1.9/botan`, libraries are named `libbotan-1.9`, and `botan-config` is now namespaced (so in this release `botan-config-1.9`). All of these embedded versions will be 1.10 in the upcoming stable release.
- The soname system has been modified. In this release the library soname is `libbotan-1.9.so.0`, with the full library being named `libbotan-1.9.so.0.18`. The `0` is the ABI version, and will be incremented whenever a breaking ABI change is made.
- TR1 support is not longer automatically assumed under older versions of GCC
- Functions for base64 decoding that work standalone (without needing to use a pipe) have been added to `base64.h`
- The function `BigInt::to_u32bit` was inadvertently removed in 1.9.11 and has been added back.
- The function `BigInt::get_substring` did not work correctly with a *length* argument of 32.
- The implementation of `FD_ZERO` on Solaris uses `memset` and assumes the caller included `string.h` on its behalf. Do so to fix compilation in the `dev_random` and `unix_procs` entropy sources. Patch from Jeremy C. Reed.
- Add two different configuration targets for Atom, since some are 32-bit and some are 64-bit. The 'atom' target now refers to the 64-bit implementations, use 'atom32' to target the 32-bit processors.
- The (incomplete) support for CMS and card verifiable certificates are disabled by default; add `--enable-modules=cms` or `--enable-modules=cvc` during configuration to turn them back on.

15.58 Version 1.9.17, 2011-04-29

- Third release candidate for 1.10.0
- The format preserving encryption method currently available was presented in the header `fpe.h` and the functions `fpe_encrypt` and `fpe_decrypt`. These were renamed as it is likely that other FPE schemes will be included in the future. The header is now `fpe_fe1.h`, and the functions are named `fe1_encrypt` and `fe1_decrypt`.
- New options to `configure.py` control what tools are used for documentation generation. The `--with-sphinx` option enables using Sphinx to convert ReST into HTML; otherwise the ReST sources are installed directly. If `--with-doxygen` is used, Doxygen will run as well. Documentation generation can be triggered via the `docs` target in the makefile; it will also be installed by the install target on Unix.
- A bug in 1.9.16 effectively disabled support for runtime CPU feature detection on x86 under GCC in that release.
- A mostly internal change, all references to "ia32" and "amd64" have been changed to the vendor neutral and probably easier to understand "x86-32" and "x86-64". For instance, the "mp_amd64" module has been re-

named "mp_x86_64", and the macro indicating x86-32 has changed from `BOTAN_TARGET_ARCH_IS_IA32` to `BOTAN_TARGET_ARCH_IS_X86_32`. The classes calling assembly have also been renamed.

- Similiarly to the above change, the AES implemenations using the AES-NI instruction set have been renamed from AES_XXX_Intel to AES_XXX_NI.
- Systems that are identified as `sun4u` will default to compiling for 32-bit SPARCv9 code rather than 64-bit. This matches the still common convention for 32-bit SPARC userspaces. If you want 64-bit code on such as system, use `--cpu=sparc64`.
- Some minor fixes for compiling botan under the BeOS clone/continuation Haiku (http://haiku-os.org).
- Further updates to the documentation

15.59 Version 1.9.16, 2011-04-11

- Second release candidate for 1.10.0
- The documentation, previously written in LaTeX, is now in reStructuredText suitable for processing by Sphinx (http://sphinx.pocoo.org), which can generate nicely formatted HTML and PDFs. The documentation has also been greatly updated and expanded.
- The class `EC_Domain_Params` has been renamed `EC_Group`, with a typedef for backwards compatibility.
- The `EC_Group` string constructor didn't understand the standard names like "secp160r1", forcing use of the OIDs.
- Two constructors for ECDSA private keys, the one that creates a new random key, and the one that provides a preset private key as a `BigInt`, have been merged. This matches the existing interface for DSA and DH keys. If you previously used the version taking a `BigInt` private key, you'll have to additionally pass in a `RandomNumberGenerator` object starting in this release.
- It is now possible to create ECDH keys with a preset `BigInt` private key; previously no method for this was available.
- The overload of `generate_passhash9` that takes an explicit algorithm identifier has been merged with the one that does not. The algorithm identifier code has been moved from the second parameter to the fourth.
- Change shared library versioning to match the normal Unix conventions. Instead of `libbotan-X.Y.Z.so`, the shared lib is named `libbotan-X.Y.so.Z`; this allows the runtime linker to do its runtime linky magic. It can be safely presumed that any change in the major or minor version indicates ABI incompatibility.
- Remove the socket wrapper code; it was not actually used by anything in the library, only in the examples, and you can use whatever kind of (blocking) socket interface you like with the SSL/TLS code. It's available as socket.h in the examples directory if you want to use it.
- Disable the by-default 'strong' checking of private keys that are loaded from storage. You can always request key material sanity checking using Private_Key::check_key.
- Bring back removed functions `min_keylength_of`, `max_keylength_of`, `keylength_multiple_of` in `lookup.h` to avoid breaking applications written against 1.8

15.60 Version 1.9.15, 2011-03-21

- First release candidate for 1.10.0
- Modify how message expansion is done in SHA-256 and SHA-512. Instead of expanding the entire message at the start, compute them in the minimum number of registers. Values are computed 15 rounds before they are needed. On a Core i7-860, GCC 4.5.2, went from 143 to 157 MiB/s in SHA-256, and 211 to 256 MiB/s in SHA-512.
- Pipe will delete empty output queues as soon as they are no longer needed, even if earlier messages still have data unread. However an (empty) entry in a deque of pointers will remain until all prior messages are completely emptied.
- Avoid reading the SPARC `%tick` register on OpenBSD as unlike the Linux and NetBSD kernels, it will not trap and emulate it for us, causing a illegal instruction crash.
- Improve detection and autoconfiguration for ARM processors. Thanks go out to the the Tahoe-LAFS Software Foundation (http://tahoe-lafs.org), who donated a Sheevaplug that I'll be using to figure out how to make the cryptographic primitives Tahoe-LAFS relies on faster, particularly targeting the ARMv5TE.

15.61 Version 1.9.14, 2011-03-01

- Add support for bcrypt, OpenBSD's password hashing scheme.
- Add support for NIST's AES key wrapping algorithm, as described in **RFC 3394** (https://tools.ietf.org/html/rfc3394.html). It is available by including `rfc3394.h`.
- Fix an infinite loop in zlib filters introduced in 1.9.11 (Bugzilla 142)

15.62 Version 1.9.13, 2011-02-19

GOST 34.10 signatures were being formatted in a way that was not compatible with other implemenations, and specifically how GOST is used in DNSSEC.

The Keccak hash function was updated to the tweaked variant proposed for round 3 of the NIST hash competition. This version is not compatible with the previous algorithm.

A new option `--distribution-info` was added to the configure script. It allows the user building the library to set any distribution-specific notes on the build, which are available as a macro `BOTAN_DISTRIBUTION_INFO`. The default value is 'unspecified'. If you are building an unmodified version of botan (especially for distribution), and want to indicate to applications that this is the case, consider using `--distribution-info=pristine`. If you are making any patches or modifications, it is recommended to use `--distribution-info=[Distribution Name] [Version]`, for instance 'FooNix 1.9.13-r3'.

Some bugs preventing compilation under Clang 2.9 and Sun Studio 12 were fixed.

The DER/BER codecs use `size_t` instead of `u32bit` for small integers

15.63 Version 1.9.12, 2010-12-13

- Add the Keccak hash function
- Fix compilation problems in Python wrappers
- Fix compilation problem in OpenSSL engine
- Update SQLite3 database encryption codec

15.64 Version 1.9.11, 2010-11-29

- The TLS API has changed substantially and now relies heavily on TR1's `std::function` is now required. Additionally, it is required that all callers derive a subclass of TLS_Policy and pass it to a client or server object. Please remember that the TLS interface/API is currently unstable and will very likely change further before TLS is included in a stable release. A handshake failure that occurred when RC4 was negotiated has also been fixed.
- Some possible timing channels in the implementations of Montgomery reduction and the IDEA key schedule were removed. The table-based AES implementation uses smaller tables in the first round to help make some timing/cache attacks harder.
- The library now uses size_t instead of u32bit to represent lengths. Also the interfaces for the memory containers have changed substantially to better match STL container interfaces; MemoryRegion::append, MemoryRegion::destroy, and MemoryRegion::set were all removed, and several other functions, like clear and resize, have changed meaning.
- Update Skein-512 to match the v1.3 specification
- Fix a number of CRL encoding and decoding bugs
- Counter mode now always encrypts 256 blocks in parallel
- Use small tables in the first round of AES
- Removed AES class: app must choose AES-128, AES-192, or AES-256
- Add hex encoding/decoding functions that can be used without a Pipe
- Add base64 encoding functions that can be used without a Pipe
- Add to_string function to X509_Certificate
- Add support for dynamic engine loading on Windows
- Replace BlockCipher::BLOCK_SIZE attribute with function block_size()
- Replace HashFunction::HASH_BLOCK_SIZE attribute with hash_block_size()
- Move PBKDF lookup to engine system
- The IDEA key schedule has been changed to run in constant time
- Add Algorithm and Key_Length_Specification classes
- Switch default PKCS #8 encryption algorithm from AES-128 to AES-256
- Allow using PBKDF2 with empty passphrases
- Add compile-time deprecation warnings for GCC, Clang, and MSVC
- Support use of HMAC(SHA-256) and CMAC(Blowfish) in passhash9
- Improve support for Intel Atom processors

- Fix compilation problems under Sun Studio and Clang

15.65 Version 1.8.11, 2010-11-02

- Fix a number of CRL encoding and decoding bugs
- When building a debug library under VC++, use the debug runtime
- Fix compilation under Sun Studio on Linux and Solaris
- Add several functions for compatibility with 1.9
- In the examples, read most input files as binary
- The Perl build script has been removed in this release

15.66 Version 1.8.10, 2010-08-31

- Switch default PKCS #8 encryption algorithm from 3DES to AES-256
- Increase default hash iterations from 2048 to 10000 in PBES1 and PBES2
- Use small tables in the first round of AES
- Add PBKDF typedef and get_pbkdf for better compatibility with 1.9
- Add version of S2K::derive_key taking salt and iteration count
- Enable the /proc-walking entropy source on NetBSD
- Fix the doxygen makefile target

15.67 Version 1.9.10, 2010-08-12

- Add a constant-time AES implementation using SSSE3. This code is based on public domain assembly written by Mike Hamburg (http://crypto.stanford.edu/vpaes/), and described in his CHES 2009 paper "Accelerating AES with Vector Permute Instructions". In addition to being constant time, it is also significantly faster than the table-based implementation on some processors. The current code has been tested with GCC 4.5, Visual C++ 2008, and Clang 2.8.
- Support for dynamically loading Engine objects at runtime was also added. Currently only system that use `dlopen`-style dynamic linking are supported.
- On GCC 4.3 and later, use the byteswap intrinsic functions.
- Drop support for building with Python 2.4
- Fix benchmarking of block ciphers in ECB mode
- Consolidate the two x86 assembly engines
- Rename S2K to PBKDF

15.68 Version 1.9.9, 2010-06-28

A new pure virtual function has been added to `Filter`, `name` which simply returns some useful identifier for the object. Any out-of-tree `Filter` implementations will need to be updated.

Add `Keyed_Filter::valid_iv_length` which makes it possible to query as to what IV length(s) a particular filter allows. Previously, partially because there was no such query mechanism, if a filter did not support IVs at all, then calls to `set_iv` would be silently ignored. Now an exception about the invalid IV length will be thrown.

The default iteration count for the password based encryption schemes has been increased from 2048 to 10000. This should make password-guessing attacks against private keys encrypted with versions after this release somewhat harder.

New functions for encoding public and private keys to binary, `X509::BER_encode` and `PKCS8::BER_encode` have been added.

Problems compiling under Apple's version of GCC 4.2.1 and on 64-bit MIPS systems using GCC 4.4 or later were fixed.

The coverage of Doxygen documentation comments has significantly improved in this release.

15.69 Version 1.8.9, 2010-06-16

- Use constant time multiplication in IDEA
- Avoid possible timing attack against OAEP decoding
- Add new X509::BER_encode and PKCS8::BER_encode
- Enable DLL builds under Windows
- Add Win32 installer support
- Add support for the Clang compiler
- Fix problem in semcem.h preventing build under Clang or GCC 3.4
- Fix bug that prevented creation of DSA groups under 1024 bits
- Fix crash in GMP_Engine if library is shutdown and reinitialized and a PK algorithm was used after the second init
- Work around problem with recent binutils in x86-64 SHA-1
- The Perl build script is no longer supported and refuses to run by default. If you really want to use it, pass `--i-know-this-is-broken` to the script.

15.70 Version 1.9.8, 2010-06-14

- Add support for wide multiplications on 64-bit Windows
- Use constant time multiplication in IDEA
- Avoid possible timing attack against OAEP decoding
- Removed FORK-256; rarely used and it has been broken
- Rename `--use-boost-python` to `--with-boost-python`
- Skip building shared libraries on MinGW/Cygwin

- Fix creation of 512 and 768 bit DL groups using the DSA kosherizer
- Fix compilation on GCC versions before 4.3 (missing cpuid.h)
- Fix compilation under the Clang compiler

15.71 Version 1.9.7, 2010-04-27

- TLS: Support reading SSLv2 client hellos
- TLS: Add support for SEED ciphersuites (RFC 4162)
- Add Comb4P hash combiner function
- Fix checking of EMSA_Raw signatures with leading 0 bytes, valid signatures could be rejected in certain scenarios.

15.72 Version 1.9.6, 2010-04-09

- TLS: Add support for TLS v1.1
- TLS: Support server name indicator extension
- TLS: Fix server handshake
- TLS: Fix server using DSA certificates
- TLS: Avoid timing channel between CBC padding check and MAC verification

15.73 Version 1.9.5, 2010-03-29

- Numerous ECC optimizations
- Fix GOST 34.10-2001 X.509 key loading
- Allow PK_Signer's fault protection checks to be toggled off
- Avoid using pool-based locking allocator if we can't mlock
- Remove all runtime options
- New BER_Decoder::{decode_and_check, decode_octet_string_bigint}
- Remove SecureBuffer in favor of SecureVector length parameter
- HMAC_RNG: Perform a poll along with user-supplied entropy
- Fix crash in MemoryRegion if Allocator::get failed
- Fix small compilation problem on FreeBSD

15.74 Version 1.9.4, 2010-03-09

- Add the Ajisai SSLv3/TLSv1.0 implementation
- Add GOST 34.10-2001 public key signature scheme
- Add SIMD implementation of Noekeon
- Add SSE2 implementation of IDEA
- Extend Salsa20 to support longer IVs (XSalsa20)
- Perform XTS encryption and decryption in parallel where possible
- Perform CBC decryption in parallel where possible
- Add SQLite3 db encryption codec, contributed by Olivier de Gaalon
- Add a block cipher cascade construction
- Add support for password hashing for authentication (passhash9.h)
- Add support for Win32 high resolution system timers
- Major refactoring and API changes in the public key code
- PK_Signer class now verifies all signatures before releasing them to the caller; this should help prevent a wide variety of fault attacks, though it does have the downside of hurting signature performance, particularly for DSA/ECDSA.
- Changed S2K interface: derive_key now takes salt, iteration count
- Remove dependency on TR1 shared_ptr in ECC and CVC code
- Renamed ECKAEG to its more usual name, ECDH
- Fix crash in GMP_Engine if library is shutdown and reinitialized
- Fix an invalid memory read in MD4
- Fix Visual C++ static builds
- Remove Timer class entirely
- Switch default PKCS #8 encryption algorithm from 3DES to AES-128
- New configuration option, `--gen-amalgamation`, creates a pair of files (`botan_all.cpp` and `botan_all.h`) which contain the contents of the library as it would have normally been compiled based on the set configuration.
- Many headers are now explicitly internal-use-only and are not installed
- Greatly improve the Win32 installer
- Several fixes for Visual C++ debug builds

15.75 Version 1.9.3, 2009-11-19

- Add new AES implementation using Intel's AES instruction intrinsics
- Add an implementation of format preserving encryption
- Allow use of any hash function in X.509 certificate creation
- Optimizations for MARS, Skipjack, and AES
- Set macros for available SIMD instructions in build.h
- Add support for using InnoSetup to package Windows builds
- By default build a DLL on Windows

15.76 Version 1.8.8, 2009-11-03

- Alter Skein-512 to match the tweaked 1.2 specification
- Fix use of inline asm for access to x86 bswap function
- Allow building the library without AES enabled
- Add 'powerpc64' alias to ppc64 arch for Gentoo ebuild

15.77 Version 1.9.2, 2009-11-03

- Add SIMD version of XTEA
- Support both SSE2 and AltiVec SIMD for Serpent and XTEA
- Optimizations for SHA-1 and SHA-2
- Add AltiVec runtime detection
- Fix x86 CPU identification with Intel C++ and Visual C++

15.78 Version 1.9.1, 2009-10-23

- Better support for Python and Perl wrappers
- Add an implementation of Blue Midnight Wish (Round 2 tweak version)
- Modify Skein-512 to match the tweaked 1.2 specification
- Add threshold secret sharing (draft-mcgrew-tss-02)
- Add runtime cpu feature detection for x86/x86-64
- Add code for general runtime self testing for hashes, MACs, and ciphers
- Optimize XTEA; twice as fast as before on Core2 and Opteron
- Convert CTR_BE and OFB from filters to stream ciphers
- New parsing code for SCAN algorithm names
- Enable SSE2 optimizations under Visual C++

- Remove all use of C++ exception specifications
- Add support for GNU/Hurd and Clang/LLVM

15.79 Version 1.8.7, 2009-09-09

- Fix processing multiple messages in XTS mode
- Add –no-autoload option to configure.py, for minimized builds

15.80 Version 1.9.0, 2009-09-09

- Add support for parallel invocation of block ciphers where possible
- Add SSE2 implementation of Serpent
- Add Rivest's package transform (an all or nothing transform)
- Minor speedups to the Turing key schedule
- Fix processing multiple messages in XTS mode
- Add –no-autoload option to configure.py, for minimized builds
- The previously used configure.pl script is no longer supported

15.81 Version 1.8.6, 2009-08-13

- Add Cryptobox, a set of simple password-based encryption routines
- Only read world-readable files when walking /proc for entropy
- Fix building with TR1 disabled
- Fix x86 bswap support for Visual C++
- Fixes for compilation under Sun C++
- Add support for Dragonfly BSD (contributed by Patrick Georgi)
- Add support for the Open64 C++ compiler
- Build fixes for MIPS systems running Linux
- Minor changes to license, now equivalent to the FreeBSD/NetBSD license

15.82 Version 1.8.5, 2009-07-23

- Change configure.py to work on stock Python 2.4
- Avoid a crash in Skein_512::add_data processing a zero-length input
- Small build fixes for SPARC, ARM, and HP-PA processors
- The test suite now returns an error code from main() if any tests failed

15.83 Version 1.8.4, 2009-07-12

- Fix a bug in nonce generation in the Miller-Rabin test

15.84 Version 1.8.3, 2009-07-11

- Add a new Python configuration script
- Add the Skein-512 SHA-3 candidate hash function
- Add the XTS block cipher mode from IEEE P1619
- Fix random_prime when generating a prime of less than 7 bits
- Improve handling of low-entropy situations during PRNG seeding
- Change random device polling to prefer /dev/urandom over /dev/random
- Use an input insensitive implementation of same_mem instead of memcmp
- Correct DataSource::discard_next to return the number of discarded bytes
- Provide a default value for AutoSeeded_RNG::reseed
- Fix Gentoo bug 272242

15.85 Version 1.8.2, 2009-04-07

- Make entropy polling more flexible and in most cases faster
- GOST 28147 now supports multiple sbox parameters
- Added the GOST 34.11 hash function
- Fix botan-config problems on MacOS X

15.86 Version 1.8.1, 2009-01-20

- Avoid a valgrind warning in es_unix.cpp on 32-bit Linux
- Fix memory leak in PKCS8 load_key and encrypt_key
- Relicense api.tex from CC-By-SA 2.5 to BSD
- Fix botan-config on MacOS X, Solaris

15.87 Version 1.8.0, 2008-12-08

- Fix compilation on Solaris with GCC

15.88 Version 1.7.24, 2008-12-01

- Fix a compatibility problem with SHA-512/EMSA3 signature padding
- Fix bug preventing EGD/PRNGD entropy poller from working
- Fix integer overflow in Pooling_Allocator::get_more_core (bug id #27)
- Add EMSA3_Raw, a variant of EMSA3 called CKM_RSA_PKCS in PKCS #11
- Add support for SHA-224 in EMSA2 and EMSA3 PK signature padding schemes
- Add many more test vectors for RSA with EMSA2, EMSA3, and EMSA4
- Wrap private structs in SSE2 SHA-1 code in anonymous namespace
- Change configure.pl's CPU autodetection output to be more consistent
- Disable using OpenSSL's AES due to crashes of unknown cause
- Fix warning in /proc walking entropy poller
- Fix compilation with IBM XLC for Cell 0.9-200709

15.89 Version 1.7.23, 2008-11-23

- Change to use TR1 (thus enabling ECDSA) with GCC and ICC
- Optimize almost all hash functions, especially MD4 and Tiger
- Add configure.pl options –{with,without}-{bzip2,zlib,openssl,gnump}
- Change Timer to be pure virtual, and add ANSI_Clock_Timer
- Cache socket descriptors in the EGD entropy source
- Avoid bogging down startup in /proc walking entropy source
- Remove Buffered_EntropySource helper class
- Add a Default_Benchmark_Timer typedef in benchmark.h
- Add examples using benchmark.h and Algorithm_Factory
- Add ECC tests from InSiTo
- Minor documentation updates

15.90 Version 1.7.22, 2008-11-17

- Add provider preferences to Algorithm_Factory
- Fix memory leaks in PBE_PKCS5v20 and get_pbe introduced in 1.7.21
- Optimize AES encryption and decryption (about 10% faster)
- Enable SSE2 optimized SHA-1 implementation on Intel Prescott CPUs
- Fix nanoseconds overflow in benchmark code
- Remove Engine::add_engine

15.91 Version 1.7.21, 2008-11-11

- Make algorithm lookup much more configuable
- Add facilities for runtime performance testing of algorithms
- Drop use of entropy estimation in the PRNGs
- Increase intervals between HMAC_RNG automatic reseeding
- Drop InitializerOptions class, all options but thread safety

15.92 Version 1.7.20, 2008-11-09

- Namespace pkg-config file by major and minor versions
- Cache device descriptors in Device_EntropySource
- Split base.h into {block_cipher,stream_cipher,mac,hash}.h
- Removed get_mgf function from lookup.h

15.93 Version 1.7.19, 2008-11-06

- Add HMAC_RNG, based on a design by Hugo Krawczyk
- Optimized the Turing stream cipher (about 20% faster on x86-64)
- Modify Randpool's reseeding algorithm to poll more sources
- Add a new AutoSeeded_RNG in auto_rng.h
- OpenPGP_S2K changed to take hash object instead of name
- Add automatic identification for Intel's Prescott processors

15.94 Version 1.7.18, 2008-10-22

- Add Doxygen comments from InSiTo
- Add ECDSA and ECKAEG benchmarks
- Add configure.pl switch –with-tr1-implementation
- Fix configure.pl's –with-endian and –with-unaligned-mem options
- Added support for pkg-config
- Optimize byteswap with x86 inline asm for Visual C++ by Yves Jerschow
- Use const references to avoid copying overhead in CurveGFp, GFpModulus

15.95 Version 1.7.17, 2008-10-12

- Add missing ECDSA object identifiers
- Fix error in x86 and x86-64 assembler affecting GF(p) math
- Remove Boost dependency from GF(p) math
- Modify botan-config to not print -L/usr/lib or -L/usr/local/lib
- Add BOTAN_DLL macro to over 30 classes missing it
- Rename the two SHA-2 base classes for consistency

15.96 Version 1.7.16, 2008-10-09

- Add several missing pieces needed for ECDSA and ECKAEG
- Add Card Verifiable Certificates from InSiTo
- Add SHA-224 from InSiTo
- Add BSI variant of EMSA1 from InSiTo
- Add GF(p) and ECDSA tests from InSiTo
- Split ECDSA and ECKAEG into distinct modules
- Allow OpenSSL and GNU MP engines to be built with public key algos disabled
- Rename sha256.h to sha2_32.h and sha_64.h to sha2_64.h

15.97 Version 1.7.15, 2008-10-07

- Add GF(p) arithmetic from InSiTo
- Add ECDSA and ECKAEG implementations from InSiTo
- Minimize internal dependencies, allowing for smaller build configurations
- Add new User Manual and Architecture Guide from FlexSecure GmbH
- Alter configure.pl options for better autotools compatibility

- Update build instructions for recent changes to configure.pl
- Fix CPU detection using /proc/cpuinfo

15.98 Version 1.7.14, 2008-09-30

- Split library into parts allowing modular builds
- Add (very preliminary) CMS support to the main library
- Some constructors now require object pointers instead of names
- Support multiple implementations of the same algorithm
- Build support for Pentium-M processors, from Derek Scherger
- Build support for MinGW/MSYS, from Zbigniew Zagorski
- Use inline assembly for bswap on 32-bit x86

15.99 Version 1.7.13, 2008-09-27

- Add SSLv3 MAC, SSLv3 PRF, and TLS v1.0 PRF from Ajisai
- Allow all examples to compile even if compression not enabled
- Make CMAC's polynomial doubling operation a public class method
- Use the -m64 flag when compiling with Sun Forte on x86-64
- Clean up and slightly optimize CMAC::final_result

15.100 Version 1.7.12, 2008-09-18

- Add x86 assembly for Visual Studio C++, by Luca Piccarreta
- Add a Perl XS module, by Vaclav Ovsik
- Add SWIG-based wrapper for Botan
- Add SSE2 implementation of SHA-1, by Dean Gaudet
- Remove the BigInt::sig_words cache due to bugs
- Combined the 4 Blowfish sboxes, suggested by Yves Jerschow
- Changed BigInt::grow_by and BigInt::grow_to to be non-const
- Add private assignment operators to classes that don't support assignment
- Benchmark RSA encryption and signatures
- Added test programs for random_prime and ressol
- Add high resolution timers for IA-64, HP-PA, S390x
- Reduce use of the RNG during benchmarks
- Fix builds on STI Cell PPU
- Add support for IBM's XLC compiler

- Add IETF 8192 bit MODP group

15.101 Version 1.7.11, 2008-09-11

- Added the Salsa20 stream cipher
- Optimized Montgomery reduction, Karatsuba squaring
- Added 16x16->32 word Comba multiplication and squaring
- Use a much larger Karatsuba cutoff point
- Remove bigint_mul_add_words
- Inlined several BigInt functions
- Add useful information to the generated build.h
- Rename alg_{ia32,amd64} modules to asm_{ia32,amd64}
- Fix the Windows build

15.102 Version 1.7.10, 2008-09-05

- Public key benchmarks run using a selection of random keys
- New benchmark timer options are clock_gettime, gettimeofday, times, clock
- Including reinterpret_cast optimization for xor_buf in default header
- Split byte swapping and word rotation functions into distinct headers
- Add IETF modp 6144 group and 2048 and 3072 bit DSS groups
- Optimizes BigInt right shift
- Add aliases in DL_Group::Format enum
- BigInt now caches the significant word count

15.103 Version 1.6.5, 2008-08-27

- Add noexec stack marker for GNU linker in assembly code
- Fix autoconfiguration problem on x86 with GCC 4.2 and 4.3

15.104 Version 1.7.9, 2008-08-27

- Make clear() in most algorithm base classes a pure virtual
- Add noexec stack marker for GNU linker in assembly code
- Avoid string operations in ressol
- Compilation fixes for MinGW and Visual Studio C++ 2008
- Some autoconfiguration fixes for Windows

15.105 Version 1.7.8, 2008-07-15

- Added the block cipher Noekeon
- Remove global deref_alias function
- X509_Store takes timeout options as constructor arguments
- Add Shanks-Tonelli algorithm, contributed by FlexSecure GmbH
- Extend random_prime() for generating primes of any bit length
- Remove Config class
- Allow adding new entropy via base RNG interface
- Reseeding a X9.31 PRNG also reseeds the underlying PRNG

15.106 Version 1.7.7, 2008-06-28

- Remove the global PRNG object
- The PK filter objects were removed
- Add a test suite for the ANSI X9.31 PRNG
- Much cleaner and (mostly) thread-safe reimplementation of es_ftw
- Remove both default arguments to ANSI_X931_RNG's constructor
- Remove the randomizing version of OctetString::change
- Make the cipher and MAC to use in Randpool configurable
- Move RandomNumberGenerator declaration to rng.h
- RSA_PrivateKey will not generate keys smaller than 1024 bits
- Fix an error decoding BER UNIVERSAL types with special taggings

15.107 Version 1.7.6, 2008-05-05

- Initial support for Windows DLLs, from Joel Low
- Reset the position pointer when a new block is generated in X9.32 PRNG
- Timer objects are now treated as entropy sources
- Moved several ASN.1-related enums from enums.h to an appropriate header
- Removed the AEP module, due to inability to test
- Removed Global_RNG and rng.h
- Removed system_clock
- Removed Library_State::UI and the pulse callback logic

15.108 Version 1.7.5, 2008-04-12

- The API of X509_CA::sign_request was altered to avoid race conditions
- New type Pipe::message_id to represent the Pipe message number
- Remove the Named_Mutex_Holder for a small performance gain
- Removed several unused or rarely used functions from Config
- Ignore spaces inside of a decimal string in BigInt::decode
- Allow using a std::istream to initialize a DataSource_Stream object
- Fix compilation problem in zlib compression module
- The chunk sized used by Pooling_Allocator is now a compile time setting
- The size of random blinding factors is now a compile time setting
- The install target no longer tries to set a particular owner/group

15.109 Version 1.7.4, 2008-03-10

- Use unaligned memory read/writes on systems that allow it, for performance
- Assembly for x86-64 for accessing the bswap instruction
- Use larger buffers in ARC4 and WiderWAKE for significant throughput increase
- Unroll loops in SHA-160 for a few percent increase in performance
- Fix compilation with GCC 3.2 in es_ftw and es_unix
- Build fix for NetBSD systems
- Prevent es_dev from being built except on Unix systems

15.110 Version 1.6.4, 2008-03-08

- Fix a compilation problem with Visual Studio C++ 2003

15.111 Version 1.7.3, 2008-01-23

- New invocation syntax for configure.pl with several new options
- Support for IPv4 addresses in a subject alternative name
- New fast poll for the generic Unix entropy source (es_unix)
- The es_file entropy source has been replaced by the es_dev module
- The malloc allocator does not inherit from Pooling_Allocator anymore
- The path that es_unix will search in are now fully user-configurable
- Truncate X9.42 PRF output rather than allow counter overflow
- PowerPC is now assumed to be big-endian

15.112 Version 1.7.2, 2007-10-13

- Initialize the global library state lazily
- Add plain CBC-MAC for backwards compatibility with old systems
- Clean up some of the self test code
- Throw a sensible exception if a DL_Group is not found
- Truncate KDF2 output rather than allowing counter overflow
- Add newly assigned OIDs for SHA-2 and DSA with SHA-224/256
- Fix a Visual Studio compilation problem in x509stat.cpp

15.113 Version 1.6.3, 2007-07-23

- Fix a race condition in the algorithm lookup cache
- Fix problems building the memory pool on some versions of Visual C++

15.114 Version 1.7.1, 2007-07-23

- Fix a race condition in the algorithm object cache
- HMAC key schedule optimization
- The build header sets a macro defining endianness, if known
- New word load/store abstraction allowing further optimization
- Modify most of the library to avoid use the C-style casts
- Use higher resolution timers in symmetric benchmarks

15.115 Version 1.7.0, 2007-05-19

- DSA parameter generation now follows FIPS 186-3
- Added OIDs for Rabin-Williams and Nyberg-Rueppel
- Somewhat better support for out of tree builds
- Minor optimizations for RC2 and Tiger
- Documentation updates
- Update the todo list

15.116 Version 1.6.2, 2007-03-24

- Fix autodection on Athlon64s running Linux
- Fix builds on QNX and compilers using STLport
- Remove a call to abort() that crept into production

15.117 Version 1.6.1, 2007-01-20

- Fix some base64 decoder bugs
- Add a new option to base64 encoding, to always append a newline
- Fix some build problems under Visual Studio with debug enabled
- Fix a bug in BER_Decoder that was triggered under some compilers

15.118 Version 1.6.0, 2006-12-17

- Minor cleanups versus 1.5.13

15.119 Version 1.5.13, 2006-12-10

- Compilation fixes for the bzip2, zlib, and GNU MP modules
- Better support for Intel C++ and EKOpath C++ on x86-64

15.120 Version 1.5.12, 2006-10-27

- Cleanups in the initialization routines
- Add some x86-64 assembly for multiply-add
- Fix problems generating very small (below 384 bit) RSA keys
- Support out of tree builds
- Bring some of the documentation up to date
- More improvements to the Python bindings

15.121 Version 1.5.11, 2006-09-10

- Removed the Algorithm base class
- Various cleanups in the public key inheritance hierarchy
- Major overhaul of the configure/build setup
- Added x86 assembler implementations of Serpent and low-level MPI code
- Optimizations for the SHA-1 x86 assembler
- Various improvements to the Python wrappers
- Work around a Visual Studio compiler bug

15.122 Version 1.5.10, 2006-08-13

- Add x86 assembler versions of MD4, MD5, and SHA-1
- Expand InitializerOptions' language to support on/off switches
- Fix definition of OID 2.5.4.8; was accidentally changed in 1.5.9
- Fix possible resource leaks in the mmap allocator
- Slightly optimized buffering in MDx_HashFunction
- Initialization failures are dealt with somewhat better
- Add an example implementing Pollard's Rho algorithm
- Better option handling in the test/benchmark tool
- Expand the xor_ciph example to support longer keys
- Some updates to the documentation

15.123 Version 1.5.9, 2006-07-12

- Fixed bitrot in the AEP engine
- Fix support for marking certificate/CRL extensions as critical
- Significant cleanups in the library state / initialization code
- LibraryInitializer takes an explicit InitializerOptions object
- Make Mutex_Factory an abstract class, add Default_Mutex_Factory
- Change configuration access to using global_state()
- Add support for global named mutexes throughout the library
- Add some STL wrappers for the delete operator
- Change how certificates are created to be more flexible and general

15.124 Version 1.5.8, 2006-06-23

- Many internal cleanups to the X.509 cert/CRL code
- Allow for application code to support new X.509 extensions
- Change the return type of X509_Certificate::{subject,issuer}_info
- Allow for alternate character set handling mechanisms
- Fix a bug that was slowing squaring performance somewhat
- Fix a very hard to hit overflow bug in the C version of word3_muladd
- Minor cleanups to the assembler modules
- Disable es_unix module on FreeBSD due to build problem on FreeBSD 6.1
- Support for GCC 2.95.x has been dropped in this release

15.125 Version 1.5.7, 2006-05-28

- Further, major changes to the BER/DER coding system
- Updated the Qt mutex module to use Mutex_Factory
- Moved the library global state object into an anonymous namespace
- Drop the Visual C++ x86 assembly module due to bugs

15.126 Version 1.5.6, 2006-03-01

- The low-level DER/BER coding system was redesigned and rewritten
- Portions of the certificate code were cleaned up internally
- Use macros to substantially clean up the GCC assembly code
- Added 32-bit x86 assembly for Visual C++ (by Luca Piccarreta)
- Avoid a couple of spurious warnings under Visual C++
- Some slight cleanups in X509_PublicKey::key_id

15.127 Version 1.5.5, 2006-02-04

- Fixed a potential infinite loop in the memory pool code (Matt Johnston)
- Made Pooling_Allocator::Memory_Block an actual class of sorts
- Some small optimizations to the division and modulo computations
- Cleaned up the implementation of some of the BigInt operators
- Reduced use of dynamic memory allocation in low-level BigInt functions
- A few simplifications in the Randpool mixing function
- Removed power(), as it was not particularly useful (or fast)

- Fixed some annoying bugs in the benchmark code
- Added a real credits file

15.128 Version 1.5.4, 2006-01-29

- Integrated x86 and amd64 assembly code, contributed by Luca Piccarreta
- Fixed a memory access off-by-one in the Karatsuba code
- Changed Pooling_Allocator's free list search to a log(N) algorithm
- Merged ModularReducer with its only subclass, Barrett_Reducer
- Fixed sign-handling bugs in some of the division and modulo code
- Renamed the module description files to modinfo.txt
- Further cleanups in the initialization code
- Removed BigInt::add and BigInt::sub
- Merged all the division-related functions into just divide()
- Modified the <mp_asmi.h> functions to allow for better optimizations
- Made the number of bits polled from an EntropySource user configurable
- Avoid including <algorithm> in <botan/secmem.h>
- Fixed some build problems with Sun Forte
- Removed some dead code from bigint_modop
- Fix the definition of same_mem

15.129 Version 1.5.3, 2006-01-24

- Many optimizations in the low-level multiple precision integer code
- Added hooks for assembly implementations of the MPI code
- Support for the X.509 issuer alternative name extension in new certs
- Fixed a bug in the decompression modules; found and patched by Matt Johnston
- New Windows mutex module (mux_win32), by Luca Piccarreta
- Changed the Windows timer module to use QueryPerformanceCounter
- mem_pool.cpp was using std::set iterators instead of std::multiset ones
- Fixed a bug in X509_CA preventing users from disabling particular extensions
- Fixed the mp_asm64 module, which was entirely broken in 1.5.2
- Fixed some module build problems on FreeBSD and Tru64

15.130 Version 1.4.12, 2006-01-15

- Fixed an off-by-one memory read in MISTY1::key()
- Fixed a nasty memory leak in Output_Buffers::retire()
- Changed maximum HMAC keylength to 1024 bits
- Fixed a build problem in the hardware timer module on 64-bit PowerPC

15.131 Version 1.5.2, 2006-01-15

- Fixed an off-by-one memory read in MISTY1::key()
- Fixed a nasty memory leak in Output_Buffers::retire()
- Reimplemented the memory allocator from scratch
- Improved memory caching in Montgomery exponentiation
- Optimizations for multiple precision addition and subtraction
- Fixed a build problem in the hardware timer module on 64-bit PowerPC
- Changed default Karatsuba cutoff to 12 words (was 14)
- Removed MemoryRegion::bits(), which was unused and incorrect
- Changed maximum HMAC keylength to 1024 bits
- Various minor Makefile and build system changes
- Avoid using std::min in <secmem.h> to bypass Windows libc macro pollution
- Switched checks/clock.cpp back to using clock() by default
- Enabled the symmetric algorithm tests, which were accidentally off in 1.5.1
- Removed the Default_Mutex's unused clone() member function

15.132 Version 1.5.1, 2006-01-08

- Implemented Montgomery exponentiation
- Implemented generalized Karatsuba multiplication and squaring
- Implemented Comba squaring for 4, 6, and 8 word inputs
- Added new Modular_Exponentiator and Power_Mod classes
- Removed FixedBase_Exp and FixedExponent_Exp
- Fixed a performance regression in get_allocator introduced in 1.5.0
- Engines can now offer S2K algorithms and block cipher padding methods
- Merged the remaining global 'algolist' code into Default_Engine
- The low-level MPI code is linked as C again
- Replaced BigInt's get_nibble with the more general get_substring
- Some documentation updates

15.133 Version 1.5.0, 2006-01-01

- Moved all global/shared library state into a single object
- Mutex objects are created through mutex factories instead of a global
- Removed ::get_mutex(), ::initialize_mutex(), and Mutex::clone()
- Removed the RNG_Quality enum entirely
- There is now only a single global-use PRNG
- Removed the no_aliases and no_oids options for LibraryInitializer
- Removed the deprecated algorithms SEAL, ISAAC, and HAVAL
- Change es_ftw to use unbuffered I/O

15.134 Version 1.4.11, 2005-12-31

- Changed Whirlpool diffusion matrix to match updated algorithm spec
- Fixed several engine module build errors introduced in 1.4.10
- Fixed two build problems in es_capi; reported by Matthew Gregan
- Added a constructor to DataSource_Memory taking a std::string
- Placing the same Filter in multiple Pipes triggers an exception
- The configure script accepts –docdir and –libdir
- Merged doc/rngs.txt into the main API document
- Thanks to Joel Low for several bug reports on early tarballs of 1.4.11

15.135 Version 1.4.10, 2005-12-18

- Added an implementation of KASUMI, the block cipher used in 3G phones
- Refactored Pipe; output queues are now managed by a distinct class
- Made certain Filter facilities only available to subclasses of Fanout_Filter
- There is no longer any overhead in Pipe for a message that has been read out
- It is now possible to generate RSA keys as small as 128 bits
- Changed some of the core classes to derive from Algorithm as a virtual base
- Changed Randpool to use HMAC instead of a plain hash as the mixing function
- Fixed a bug in the allocators; found and fixed by Matthew Gregan
- Enabled the use of binary file I/O, when requested by the application
- The OpenSSL engine's block cipher code was missing some deallocation calls
- Disabled the es_ftw module on NetBSD, due to header problems there
- Fixed a problem preventing tm_hard from building on MacOS X on PowerPC
- Some cleanups for the modules that use inline assembler

- config.h is now stored in build/ instead of build/include/botan/
- The header util.h was split into bit_ops.h, parsing.h, and util.h
- Cleaned up some redundant include directives

15.136 Version 1.4.9, 2005-11-06

- Added the IBM-created AES candidate algorithm MARS
- Added the South Korean block cipher SEED
- Added the stream cipher Turing
- Added the new hash function FORK-256
- Deprecated the ISAAC stream cipher
- Twofish and RC6 are significantly faster with GCC
- Much better support for 64-bit PowerPC
- Added support for high-resolution PowerPC timers
- Fixed a bug in the configure script causing problems on FreeBSD
- Changed ANSI X9.31 to support arbitrary block ciphers
- Make the configure script a bit less noisy
- Added more test vectors for some algorithms, including all the AES finalists
- Various cosmetic source code cleanups

15.137 Version 1.4.8, 2005-10-16

- Resolved a bad performance problem in the allocators; fix by Matt Johnston
- Worked around a Visual Studio 2003 compilation problem introduced in 1.4.7
- Renamed OMAC to CMAC to match the official NIST naming
- Added single byte versions of update() to PK_Signer and PK_Verifier
- Removed the unused reverse_bits and reverse_bytes functions

15.138 Version 1.4.7, 2005-09-25

- Fixed major performance problems with recent versions of GNU C++
- Added an implementation of the X9.31 PRNG
- Removed the X9.17 and FIPS 186-2 PRNG algorithms
- Changed defaults to use X9.31 PRNGs as global PRNG objects
- Documentation updates to reflect the PRNG changes
- Some cleanups related to the engine code
- Removed two useless headers, base_eng.h and secalloc.h

- Removed PK_Verifier::valid_signature
- Fixed configure/build system bugs affecting MacOS X builds
- Added support for the EKOPath x86-64 compiler
- Added missing destructor for BlockCipherModePaddingMethod
- Fix some build problems with Visual C++ 2005 beta
- Fix some build problems with Visual C++ 2003 Workshop

15.139 Version 1.4.6, 2005-03-13

- Fix an error in the shutdown code introduced in 1.4.5
- Setting base/pkcs8_tries to 0 disables the builtin fail-out
- Support for XMPP identifiers in X.509 certificates
- Duplicate entries in X.509 DNs are removed
- More fixes for Borland C++, from Friedemann Kleint
- Add a workaround for buggy iostreams

15.140 Version 1.4.5, 2005-02-26

- Add support for AES encryption of private keys
- Minor fixes for PBES2 parameter decoding
- Internal cleanups for global state variables
- GCC 3.x version detection was broken in non-English locales
- Work around a Sun Forte bug affecting mem_pool.h
- Several fixes for Borland C++ 5.5, from Friedemann Kleint
- Removed inclusion of init.h into base.h
- Fixed a major bug in reading from certificate stores
- Cleaned up a couple of mutex leaks
- Removed some left-over debugging code
- Removed SSL3_MAC, SSL3_PRF, and TLS_PRF

15.141 Version 1.4.4, 2004-12-02

- Further tweaks to the pooling allocator
- Modified EMSA3 to support SSL/TLS signatures
- Changes to support Qt/QCA, from Justin Karneges
- Moved mux_qt module code into mod_qt
- Fixes for HP-UX from Mike Desjardins

15.142 Version 1.4.3, 2004-11-06

- Split up SecureAllocator into Allocator and Pooling_Allocator
- Memory locking allocators are more likely to be used
- Fixed the placement of includes in some modules
- Fixed broken installation procedure
- Fixes in configure script to support alternate install programs
- Modules can specify the minimum version they support

15.143 Version 1.4.2, 2004-10-31

- Fixed a major CRL handling bug
- Cipher and hash operations can be offloaded to engines
- Added support for cipher and hash offload in OpenSSL engine
- Improvements for 64-bit CPUs without a widening multiply instruction
- Support for SHA2-* and Whirlpool with EMSA2
- Fixed a long-standing build problem with conflicting include files
- Fixed some examples that hadn't been updated for 1.4.x
- Portability fixes for Solaris, BSD, HP-UX, and others
- Lots of fixes and cleanups in the configure script
- Updated the Gentoo ebuild file

15.144 Version 1.4.1, 2004-10-10

- Fixed major errors in the X.509 and PKCS #8 copy_key functions
- Added a LAST_MESSAGE meta-message number for Pipe
- Added new aliases (3DES and DES-EDE) for Triple-DES
- Added some new functions to PK_Verifier
- Cleaned up the KDF interface
- Disabled tm_posix on BSD due to header issues
- Fixed a build problem on PowerPC with GNU C++ pre-3.4

15.145 Version 1.4.0, 2004-06-26

- Added the FIPS 186 RNG back
- Added copy_key functions for X.509 public keys and PKCS #8 private keys
- Fixed PKCS #1 signatures with RIPEMD-128
- Moved some code around to avoid warnings with Sun ONE compiler
- Fixed a bug in botan-config affecting OpenBSD
- Fixed some build problems on Tru64, HP-UX
- Fixed compile problems with Intel C++, Compaq C++

15.146 Version 1.3.14, 2004-06-12

- Added support for AEP's AEP1000/AEP2000 crypto cards
- Added a Mutex module using Qt, from Justin Karneges
- Added support for engine loading in LibraryInitializer
- Tweaked SecureAllocator, giving 20% better performance under heavy load
- Added timer and memory locking modules for Win32 (tm_win32, ml_win32)
- Renamed PK_Engine to Engine_Core
- Improved the Karatsuba cutoff points
- Fixes for compiling with GCC 3.4 and Sun C++ 5.5
- Fixes for Linux/s390, OpenBSD, and Solaris
- Added support for Linux/s390x
- The configure script was totally broken for 'generic' OS
- Removed Montgomery reduction due to bugs
- Removed an unused header, pkcs8alg.h
- check –validate returns an error code if any tests failed
- Removed duplicate entry in Unix command list for es_unix
- Moved the Cert_Usage enumeration into X509_Store
- Added new timing methods for PK benchmarks, clock_gettime and RDTSC
- Fixed a few minor bugs in the configure script
- Removed some deprecated functions from x509cert.h and pkcs10.h
- Removed the 'minimal' module, has to be updated for Engine support
- Changed MP_WORD_BITS macro to BOTAN_MP_WORD_BITS to clean up namespace
- Documentation updates

15.147 Version 1.3.13, 2004-05-15

- Major fixes for Cygwin builds
- Minor MacOS X install fixes
- The configure script is a little better at picking the right modules
- Removed ml_unix from the 'unix' module set for Cygwin compatibility
- Fixed a stupid compile problem in pkcs10.h

15.148 Version 1.3.12, 2004-05-02

- Added ability to remove old entries from CRLs
- Swapped the first two arguments of X509_CA::update_crl()
- Added an < operator for MemoryRegion, so it can be used as a std::map key
- Changed X.509 searching by DNS name from substring to full string compares
- Renamed a few X509_Certificate and PKCS10_Request member functions
- Fixed a problem when decoding some PKCS #10 requests
- Hex_Decoder would not check inputs, reported by Vaclav Ovsik
- Changed default CRL expire time from 30 days to 7 days
- X509_CRL's default PEM header is now "X509 CRL", for OpenSSL compatibility
- Corrected errors in the API doc, fixes from Ken Perano
- More documentation about the Pipe/Filter code

15.149 Version 1.3.11, 2004-04-01

- Fixed two show-stopping bugs in PKCS10_Request
- Added some sanity checks in Pipe/Filter
- The DNS and URI entries would get swapped in subjectAlternativeNames
- MAC_Filter is now willing to not take a key at creation time
- Setting the expiration times of certs and CRLs is more flexible
- Fixed problems building on AIX with GCC
- Fixed some problems in the tutorial pointed out by Dominik Vogt
- Documentation updates

15.150 Version 1.3.10, 2004-03-27

- Added support for OpenPGP's ASCII armor format
- Cleaned up the RNG system; seeding is much more flexible
- Added simple autoconfiguration abilities to configure.pl
- Fixed a GCC 2.95.x compile problem
- Updated the example configuration file
- Documentation updates

15.151 Version 1.3.9, 2004-03-07

- Added an engine using OpenSSL (requires 0.9.7 or later)
- X509_Certificate would lose email addresses stored in the DN
- Fixed a missing initialization in a BigInt constructor
- Fixed several Visual C++ compile problems
- Fixed some BeOS build problems
- Fixed the WiderWake benchmark

15.152 Version 1.3.8, 2003-12-30

- Initial introduction of engine support, which separates PK keys from the underlying operations. An engine using GNU MP was added.
- DSA, DH, NR, and ElGamal constructors accept taking just the private key again since the public key is easily derived from it.
- Montgomery reduction support was added.
- ElGamal keys now support being imported/exported as ASN.1 objects
- Added Montgomery reductions
- Added an engine that uses GNU MP (requires 4.1 or later)
- Removed the obsolete mp_gmp module
- Moved several initialization/shutdown functions to init.h
- Major refactoring of the memory containers
- New non-locking container, MemoryVector
- Fixed 64-bit problems in BigInt::set_bit/clear_bit
- Renamed PK_Key::check_params() to check_key()
- Some incompatible changes to OctetString
- Added version checking macros in version.h
- Removed the fips140 module pending rewrite

- Added some functions and hooks to help GUIs
- Moved more shared code into MDx_HashFunction
- Added a policy hook for specifying the encoding of X.509 strings

15.153 Version 1.3.7, 2003-12-12

- Fixed a big security problem in es_unix (use of untrusted PATH)
- Fixed several stability problems in es_unix
- Expanded the list of programs es_unix will try to use
- SecureAllocator now only preallocates blocks in special cases
- Added a special case in Global_RNG::seed for forcing a full poll
- Removed the FIPS 186 RNG added in 1.3.5 pending further testing
- Configure updates for PowerPC CPUs
- Removed the (never tested) VAX support
- Added support for S/390 Linux

15.154 Version 1.3.6, 2003-12-07

- Added a new module 'minimal', which disables most algorithms
- SecureAllocator allocates a few blocks at startup
- A few minor MPI cleanups
- RPM spec file cleanups and fixes

15.155 Version 1.3.5, 2003-11-30

- Major improvements in ASN.1 string handling
- Added partial support for ASN.1 UTF8 STRINGs and BMP STRINGs
- Added partial support for the X.509v3 certificate policies extension
- Centralized the handling of character set information
- Added FIPS 140-2 startup self tests
- Added a module (fips140) for doing extra FIPS 140-2 tests
- Added FIPS 186-2 RNG
- Improved ASN.1 BIT STRING handling
- Removed a memory leak in PKCS10_Request
- The encoding of DirectoryString now follows PKIX guidelines
- Fixed some of the character set dependencies
- Fixed a DER encoding error for tags greater than 30

- The BER decoder can now handle tags larger than 30
- Fixed tm_hard.cpp to recognize SPARC on more systems
- Workarounds for a GCC 2.95.x bug in x509find.cpp
- RPM changed to install into /usr instead of /usr/local
- Added support for QNX

15.156 Version 1.2.8, 2003-11-21

- Merged several important bug fixes from 1.3.x

15.157 Version 1.3.4, 2003-11-21

- Added a module that does certain MPI operations using GNU MP
- Added the X9.42 Diffie-Hellman PRF
- The Zlib and Bzip2 objects now use custom allocators
- Added member functions for directly hashing/MACing SecureVectors
- Minor optimizations to the MPI addition and subtraction algorithms
- Some cleanups in the low-level MPI code
- Created separate AES-{128,192,256} objects

15.158 Version 1.3.3, 2003-11-17

- The library can now be repeatedly initialized and shutdown without crashing
- Fixed an off-by-one error in the CTS code
- Fixed an error in the EMSA4 verification code
- Fixed a memory leak in mutex.cpp (pointed out by James Widener)
- Fixed a memory leak in Pthread_Mutex
- Fixed several memory leaks in the testing code
- Bulletproofed the EMSA/EME/KDF/MGF retrieval functions
- Minor cleanups in SecureAllocator
- Removed a needless mutex guarding the (stateless) global timer
- Fixed a piece of bash-specific code in botan-config
- X.509 objects report more information about decoding errors
- Cleaned up some of the exception handling
- Updated the example config file with new OIDSs
- Moved the build instructions into a separate document, building.tex

15.159 Version 1.3.2, 2003-11-13

- Fixed a bug preventing DSA signatures from verifying on X.509 objects
- Made the X509_Store search routines more efficient and flexible
- Added a function to X509_PublicKey to do easy public/private key matching
- Added support for decoding indefinite length BER data
- Changed Pipe's peek() to take an offset
- Removed Filter::set_owns in favor of the new incr_owns function
- Removed BigInt::zero() and BigInt::one()
- Renamed the PEM related options from base/pem_* to pem/*
- Added an option to specify the line width when encoding PEM
- Removed the "rng/safe_longterm" option; it's always on now
- Changed the cipher used for RNG super-encryption from ARC4 to WiderWake4+1
- Cleaned up the base64/hex encoders and decoders
- Added an ASN.1/BER decoder as an example
- AES had its internals marked 'public' in previous versions
- Changed the value of the ASN.1 NO_OBJECT enum
- Various new hacks in the configure script
- Removed the already nominal support for SunOS

15.160 Version 1.3.1, 2003-11-04

- Generalized a few pieces of the DER encoder
- PKCS8::load_key would fail if handed an unencrypted key
- Added a failsafe so PKCS #8 key decoding can't go into an infinite loop

15.161 Version 1.3.0, 2003-11-02

- Major redesign of the PKCS #8 private key import/export system
- Added a small amount of UI interface code for getting passphrases
- Added heuristics that tell if a key, cert, etc is stored as PEM or BER
- Removed CS-Cipher, SHARK, ThreeWay, MD5-MAC, and EMAC
- Removed certain deprecated constructors of RSA, DSA, DH, RW, NR
- Made PEM decoding more forgiving of extra text before the header

15.162 Version 1.2.7, 2003-10-31

- Added support for reading configuration files
- Added constructors so NR and RW keys can be imported easily
- Fixed mp_asm64, which was completely broken in 1.2.6
- Removed tm_hw_ia32 module; replaced by tm_hard
- Added support for loading certain oddly formed RSA certificates
- Fixed spelling of NON_REPUDIATION enum
- Renamed the option default_to_ca to v1_assume_ca
- Fixed a minor bug in X.509 certificate generation
- Fixed a latent bug in the OID lookup code
- Updated the RPM spec file
- Added to the tutorial

15.163 Version 1.2.6, 2003-07-04

- Major performance increase for PK algorithms on most 64-bit systems
- Cleanups in the low-level MPI code to support asm implementations
- Fixed build problems with some versions of Compaq's C++ compiler
- Removed useless constructors for NR public and private keys
- Removed support for the patch_file directive in module files
- Removed several deprecated functions

15.164 Version 1.2.5, 2003-06-22

- Fixed a tricky and long-standing memory leak in Pipe
- Major cleanups and fixes in the memory allocation system
- Removed alloc_mlock, which has been superseded by the ml_unix module
- Removed a denial of service vulnerability in X509_Store
- Fixed compilation problems with VS .NET 2003 and Codewarrior 8
- Added another variant of PKCS8::load_key, taking a memory buffer
- Fixed various minor/obscure bugs which occurred when MP_WORD_BITS != 32
- BigInt::operator%=(word) was a no-op if the input was a power of 2
- Fixed portability problems in BigInt::to_u32bit
- Fixed major bugs in SSL3-MAC
- Cleaned up some messes in the PK algorithms
- Cleanups and extensions for OMAC and EAX

- Made changes to the entropy estimation function
- Added a 'beos' module set for use on BeOS
- Officially deprecated a few X509:: and PKCS8:: functions
- Moved the contents of primes.h to numthry.h
- Moved the contents of x509opt.h to x509self.h
- Removed the (empty) desx.h header
- Documentation updates

15.165 Version 1.2.4, 2003-05-29

- Fixed a bug in EMSA1 affecting NR signature verification
- Fixed a few latent bugs in BigInt related to word size
- Removed an unused function, mp_add2_nc, from the MPI implementation
- Reorganized the core MPI files

15.166 Version 1.2.3, 2003-05-20

- Fixed a bug that prevented DSA/NR key generation
- Fixed a bug that prevented importing some root CA certs
- Fixed a bug in the BER decoder when handing optional bit or byte strings
- Fixed the encoding of authorityKeyIdentifier in X509_CA
- Added a sanity check in PBKDF2 for zero length passphrases
- Added versions of X509::load_key and PKCS8::load_key that take a file name
- X509_CA generates 128 bit serial numbers now
- Added tests to check PK key generation
- Added a simplistic X.509 CA example
- Cleaned up some of the examples

15.167 Version 1.2.2, 2003-05-13

- Add checks to prevent any BigInt bugs from revealing an RSA or RW key
- Changed the interface of Global_RNG::seed
- Major improvements for the es_unix module
- Added another Win32 entropy source, es_win32
- The Win32 CryptoAPI entropy source can now poll multiple providers
- Improved the BeOS entropy source
- Renamed pipe_unixfd module to fd_unix

- Fixed a file descriptor leak in the EGD module
- Fixed a few locking bugs

15.168 Version 1.2.1, 2003-05-06

- Added ANSI X9.23 compatible CBC padding
- Added an entropy source using Win32 CryptoAPI
- Removed the Pipe I/O operators taking a FILE*
- Moved the BigInt encoding/decoding functions into the BigInt class
- Integrated several fixes for VC++ 7 (from Hany Greiss)
- Fixed the configure.pl script for Windows builds

15.169 Version 1.2.0, 2003-04-28

- Tweaked the Karatsuba cut-off points
- Increased the allowed keylength of HMAC and Blowfish
- Removed the 'mpi_ia32' module, pending rewrite
- Workaround a GCC 2.95.x bug in eme1.cpp

15.170 Version 1.1.13, 2003-04-22

- Added OMAC
- Added EAX authenticated cipher mode
- Diffie-Hellman would not do blinding in some cases
- Optimized the OFB and CTR modes
- Corrected Skipjack's word ordering, as per NIST clarification
- Support for all subject/issuer attribute types required by RFC 3280
- The removeFromCRL CRL reason code is now handled correctly
- Increased the flexibility of the allocators
- Renamed Rijndael to AES, created aes.h, deleted rijndael.h
- Removed support for the 'no_timer' LibraryInitializer option
- Removed 'es_pthr' module, pending further testing
- Cleaned up get_ciph.cpp

15.171 Version 1.1.12, 2003-04-15

- Fixed a ASN.1 string encoding bug
- Fixed a pair of X509_DN encoding problems
- Base64_Decoder and Hex_Decoder can now validate input
- Removed support for the LibraryInitializer option 'egd_path'
- Added tests for DSA X.509 and PKCS #8 key formats
- Removed a long deprecated feature of DH_PrivateKey's constructor
- Updated the RPM .spec file
- Major documentation updates

15.172 Version 1.1.11, 2003-04-07

- Added PKCS #10 certificate requests
- Changed X509_Store searching interface to be more flexible
- Added a generic Certificate_Store interface
- Added a function for generating self-signed X.509 certs
- Cleanups and changes to X509_CA
- New examples for PKCS #10 and self-signed certificates
- Some documentation updates

15.173 Version 1.1.10, 2003-04-03

- X509_CA can now generate new X.509 CRLs
- Added blinding for RSA, RW, DH, and ElGamal to prevent timing attacks
- More certificate and CRL extensions/attributes are supported
- Better DN handling in X.509 certificates/CRLs
- Added a DataSink hierarchy (suggested by Jim Darby)
- Consolidated SecureAllocator and ManagedAllocator
- Many cleanups and generalizations
- Added a (slow) pthreads based EntropySource
- Fixed some threading bugs

15.174 Version 1.1.9, 2003-02-25

- Added support for using X.509v2 CRLs
- Fixed several bugs in the path validation algorithm
- Certificates can be verified for a particular usage
- Algorithm for comparing distinguished names now follows X.509
- Cleaned up the code for the es_beos, es_ftw, es_unix modules
- Documentation updates

15.175 Version 1.1.8, 2003-01-29

- Fixes for the certificate path validation algorithm in X509_Store
- Fixed a bug affecting X509_Certificate::is_ca_cert()
- Added a general configuration interface for policy issues
- Cleanups and API changes in the X.509 CA, cert, and store code
- Made various options available for X509_CA users
- Changed X509_Time's interface to work around time_t problems
- Fixed a theoretical weakness in Randpool's entropy mixing function
- Fixed problems compiling with GCC 2.95.3 and GCC 2.96
- Fixed a configure bug (reported by Jon Wilson) affecting MinGW

15.176 Version 1.0.2, 2003-01-12

- Fixed an obscure SEGFAULT causing bug in Pipe
- Fixed an obscure but dangerous bug in SecureVector::swap

15.177 Version 1.1.7, 2003-01-12

- Fixed an obscure but dangerous bug in SecureVector::swap
- Consolidated SHA-384 and SHA-512 to save code space
- Added SSL3-MAC and SSL3-PRF
- Documentation updates, including a new tutorial

15.178 Version 1.1.6, 2002-12-10

- Initial support for X.509v3 certificates and CAs
- Major redesign/rewrite of the ASN.1 encoding/decoding code
- Added handling for DSA/NR signatures encoded as DER SEQUENCEs
- Documented the generic cipher lookup interface
- Added an (untested) entropy source for BeOS
- Various cleanups and bug fixes

15.179 Version 1.1.5, 2002-11-17

- Added the discrete logarithm integrated encryption system (DLIES)
- Various optimizations for BigInt
- Added support for assembler optimizations in modules
- Added BigInt x86 optimizations module (mpi_ia32)

15.180 Version 1.1.4, 2002-11-10

- Speedup of 15-30% for PK algorithms
- Implemented the PBES2 encryption scheme
- Fixed a potential bug in decoding RSA and RW private keys
- Changed the DL_Group class interface to handle different formats better
- Added support for PKCS #3 encoded DH parameters
- X9.42 DH parameters use a PEM label of 'X942 DH PARAMETERS'
- Added key pair consistency checking
- Fixed a compatibility problem with gcc 2.96 (pointed out by Hany Greiss)
- A botan-config script is generated at configure time
- Documentation updates

15.181 Version 1.1.3, 2002-11-03

- Added a generic public/private key loading interface
- Fixed a small encoding bug in RSA, RW, and DH
- Changed the PK encryption/decryption interface classes
- ECB supports using padding methods
- Added a function-based interface for library initialization
- Added support for RIPEMD-128 and Tiger PKCS#1 v1.5 signatures

- The cipher mode benchmarks now use 128-bit AES instead of DES
- Removed some obsolete typedefs
- Removed OpenCL support (opencl.h, the OPENCL_* macros, etc)
- Added tests for PKCS #8 encoding/decoding
- Added more tests for ECB and CBC

15.182 Version 1.1.2, 2002-10-21

- Support for PKCS #8 encoded RSA, DSA, and DH private keys
- Support for Diffie-Hellman X.509 public keys
- Major reorganization of how X.509 keys are handled
- Added PKCS #5 v2.0's PBES1 encryption scheme
- Added a generic cipher lookup interface
- Added the WiderWake4+1 stream cipher
- Added support for sync-able stream ciphers
- Added a 'paranoia level' option for the LibraryInitializer
- More security for RNG output meant for long term keys
- Added documentation for some of the new 1.1.x features
- CFB's feedback argument is now specified in bits
- Renamed CTR class to CTR_BE
- Updated the RSA and DSA examples to use X.509 and PKCS #8 key formats

15.183 Version 1.1.1, 2002-10-15

- Added the Korean hash function HAS-160
- Partial support for RSA and DSA X.509 public keys
- Added a mostly functional BER encoder/decoder
- Added support for non-deterministic MAC functions
- Initial support for PEM encoding/decoding
- Internal cleanups in the PK algorithms
- Several new convenience functions in Pipe
- Fixed two nasty bugs in Pipe
- Messed with the entropy sources for es_unix
- Discrete logarithm groups are checked for safety more closely now
- For compatibility with GnuPG, ElGamal now supports DSA-style groups

15.184 Version 1.0.1, 2002-09-14

- Fixed a minor bug in Randpool::random()
- Added some new aliases and typedefs for 1.1.x compatibility
- The 4096-bit RSA benchmark key was decimal instead of hex
- EMAC was returning an incorrect name

15.185 Version 1.1.0, 2002-09-14

- Added entropy estimation to the RNGs
- Improved the overall design of both Randpool and ANSI_X917_RNG
- Added a separate RNG for nonce generation
- Added window exponentiation support in power_mod
- Added a get_s2k function and the PKCS #5 S2K algorithms
- Added the TLSv1 PRF
- Replaced BlockCipherModeIV typedef with InitializationVector class
- Renamed PK_Key_Agreement_Scheme to PK_Key_Agreement
- Renamed SHA1 -> SHA_160 and SHA2_x -> SHA_x
- Added support for RIPEMD-160 PKCS#1 v1.5 signatures
- Changed the key agreement scheme interface
- Changed the S2K and KDF interfaces
- Better SCAN compatibility for HAVAL, Tiger, MISTY1, SEAL, RC5, SAFER-SK
- Added support for variable-pass Tiger
- Major speedup for Rabin-Williams key generation

15.186 Version 1.0.0, 2002-08-26

- Octal I/O of BigInt is now supported
- Fixed portability problems in the es_egd module
- Generalized IV handling in the block cipher modes
- Added Karatsuba multiplication and k-ary exponentiation
- Fixed a problem in the multiplication routines

15.187 Version 0.9.2, 2002-08-18

- DH_PrivateKey::public_value() was returning the wrong value
- Various BigInt optimizations
- The filters.h header now includes hex.h and base64.h
- Moved Counter mode to ctr.h
- Fixed a couple minor problems with VC++ 7
- Fixed problems with the RPM spec file

15.188 Version 0.9.1, 2002-08-10

- Grand rename from OpenCL to Botan
- Major optimizations for the PK algorithms
- Added ElGamal encryption
- Added Whirlpool
- Tweaked memory allocation parameters
- Improved the method of seeding the global RNG
- Moved pkcs1.h to eme_pkcs.h
- Added more test vectors for some algorithms
- Fixed error reporting in the BigInt tests
- Removed Default_Timer, it was pointless
- Added some new example applications
- Removed some old examples that weren't that interesting
- Documented the compression modules

15.189 Version 0.9.0, 2002-08-03

- EMSA4 supports variable salt size
- PK_* can take a string naming the encoding method to use
- Started writing some internals documentation

15.190 Version 0.8.7, 2002-07-30

- Fixed bugs in EME1 and EMSA4
- Fixed a potential crash at shutdown
- Cipher modes returned an ill-formed name
- Removed various deprecated types and headers
- Cleaned up the Pipe interface a bit
- Minor additions to the documentation
- First stab at a Visual C++ makefile (doc/Makefile.vc7)

15.191 Version 0.8.6, 2002-07-25

- Added EMSA4 (aka PSS)
- Brought the manual up to date; many corrections and additions
- Added a parallel hash function construction
- Lookup supports all available algorithms now
- Lazy initialization of the lookup tables
- Made more discrete logarithm groups available through get_dl_group()
- StreamCipher_Filter supports seeking (if the underlying cipher does)
- Minor optimization for GCD calculations
- Renamed SAFER_SK128 to SAFER_SK
- Removed many previously deprecated functions
- Some now-obsolete functions, headers, and types have been deprecated
- Fixed some bugs in DSA prime generation
- DL_Group had a constructor for DSA-style prime gen but it wasn’t defined
- Reversed the ordering of the two arguments to SEAL’s constructor
- Fixed a threading problem in the PK algorithms
- Fixed a minor memory leak in lookup.cpp
- Fixed pk_types.h (it was broken in 0.8.5)
- Made validation tests more verbose
- Updated the check and example applications

15.192 Version 0.8.5, 2002-07-21

- Major changes to constructors for DL-based cryptosystems (DSA, NR, DH)
- Added a DL_Group class
- Reworking of the pubkey internals
- Support in lookup for aliases and PK algorithms
- Renamed CAST5 to CAST_128 and CAST256 to CAST_256
- Added EMSA1
- Reorganization of header files
- LibraryInitializer will install new allocator types if requested
- Fixed a bug in Diffie-Hellman key generation
- Did a workaround in pipe.cpp for GCC 2.95.x on Linux
- Removed some debugging code from init.cpp that made FTW ES useless
- Better checking for invalid arguments in the PK algorithms
- Reduced Base64 and Hex default line length (if line breaking is used)
- Fixes for HP's aCC compiler
- Cleanups in BigInt

15.193 Version 0.8.4, 2002-07-14

- Added Nyberg-Rueppel signatures
- Added Diffie-Hellman key exchange (kex interface is subject to change)
- Added KDF2
- Enhancements to the lookup API
- Many things formerly taking pointers to algorithms now take names
- Speedups for prime generation
- LibraryInitializer has support for seeding the global RNG
- Reduced SAFER-SK128 memory consumption
- Reversed the ordering of public and private key values in DSA constructor
- Fixed serious bugs in MemoryMapping_Allocator
- Fixed memory leak in Lion
- FTW_EntropySource was not closing the files it read
- Fixed line breaking problem in Hex_Encoder

15.194 Version 0.8.3, 2002-06-09

- Added DSA and Rabin-Williams signature schemes
- Added EMSA3
- Added PKCS#1 v1.5 encryption padding
- Added Filters for PK algorithms
- Added a Keyed_Filter class
- LibraryInitializer processes arguments now
- Major revamp of the PK interface classes
- Changed almost all of the Filters for non-template operation
- Changed HMAC, Lion, Luby-Rackoff to non-template classes
- Some fairly minor BigInt optimizations
- Added simple benchmarking for PK algorithms
- Added hooks for fixed base and fixed exponent modular exponentiation
- Added some examples for using RSA
- Numerous bugfixes and cleanups
- Documentation updates

15.195 Version 0.8.2, 2002-05-18

- Added an (experimental) algorithm lookup interface
- Added code for directly testing BigInt
- Added SHA2-384
- Optimized SHA2-512
- Major optimization for Adler32 (thanks to Dan Nicolaescu)
- Various minor optimizations in BigInt and related areas
- Fixed two bugs in X9.19 MAC, both reported by Darren Starsmore
- Fixed a bug in BufferingFilter
- Made a few fixes for MacOS X
- Added a workaround in configure.pl for GCC 2.95.x
- Better support for PowerPC, ARM, and Alpha
- Some more cleanups

15.196 Version 0.8.1, 2002-05-06

- Major code cleanup (check doc/deprecated.txt)
- Various bugs fixed, including several portability problems
- Renamed MessageAuthCode to MessageAuthenticationCode
- A replacement for X917 is in x917_rng.h
- Changed EMAC to non-template class
- Added ANSI X9.19 compatible CBC-MAC
- TripleDES now supports 128 bit keys

15.197 Version 0.8.0, 2002-04-24

- Merged BigInt: many bugfixes and optimizations since alpha2
- Added RSA (rsa.h)
- Added EMSA2 (emsa2.h)
- Lots of new interface code for public key algorithms (pk_base.h, pubkey.h)
- Changed some interfaces, including SymmetricKey, to support the global rng
- Fixed a serious bug in ManagedAllocator
- Renamed RIPEMD128 to RIPEMD_128 and RIPEMD160 to RIPEMD_160
- Removed some deprecated stuff
- Added a global random number generator (rng.h)
- Added clone functions to most of the basic algorithms
- Added a library initializer class (init.h)
- Version macros in version.h
- Moved the base classes from opencl.h to base.h
- Renamed the bzip2 module to comp_bzip2 and zlib to comp_zlib
- Documentation updates for the new stuff (still incomplete)
- Many new deprecated things: check doc/deprecated.txt

15.198 Version 0.7.10, 2002-04-07

- Added EGD_EntropySource module (es_egd)
- Added a file tree walking EntropySource (es_ftw)
- Added MemoryLocking_Allocator module (alloc_mlock)
- Renamed the pthr_mux, unix_rnd, and mmap_mem modules
- Changed timer mechanism; the clock method can be switched on the fly.
- Renamed MmapDisk_Allocator to MemoryMapping_Allocator

- Renamed ent_file.h to es_file.h (ent_file.h is around, but deprecated)
- Fixed several bugs in MemoryMapping_Allocator
- Added more default sources for Unix_EntropySource
- Changed SecureBuffer to use same allocation methods as SecureVector
- Added bigint_divcore into mp_core to support BigInt alpha2 release
- Removed some Pipe functions deprecated since 0.7.8
- Some fixes for the configure program

15.199 Version 0.7.9, 2002-03-19

- Memory allocation substantially revamped
- Added memory allocation method based on mmap(2) in the mmap_mem module
- Added ECB and CTS block cipher modes (ecb.h, cts.h)
- Added a Mutex interface (mutex.h)
- Added module pthr_mux, implementing the Mutex interface
- Added Threaded Filter interface (thr_filt.h)
- All algorithms can now by keyed with SymmetricKey objects
- More testing occurs with –validate (expected failures)
- Fixed two bugs reported by Hany Greiss, in Luby-Rackoff and RC6
- Fixed a buffering bug in Bzip_Decompress and Zlib_Decompress
- Made X917 safer (and about 1/3 as fast)
- Documentation updates

15.200 Version 0.7.8, 2002-02-28

- More capabilities for Pipe, inspired by SysV STREAMS, including peeking, better buffering, and stack ops. NOT BACKWARDS COMPATIBLE: SEE DOCUMENTATION
- Added a BufferingFilter class
- Added popen() based EntropySource for generic Unix systems (unix_rnd)
- Moved ‘devrand’ module into main distribution (ent_file.h), renamed to File_EntropySource, and changed interface somewhat.
- Made Randpool somewhat more conservative and also 25% faster
- Minor fixes and updates for the configure script
- Added some tweaks for memory allocation
- Documentation updates for the new Pipe interface
- Fixed various minor bugs
- Added a couple of new example programs (stack and hasher2)

15.201 Version 0.7.7, 2001-11-24

- Filter::send now works in the constructor of a Filter subclass
- You may now have to include <opencl/pipe.h> explicitly in some code
- Added preliminary PK infrastructure classes in pubkey.h and pkbase.h
- Enhancements to SecureVector (append, destroy functions)
- New infrastructure for secure memory allocation
- Added IEEE P1363 primitives MGF1, EME1, KDF1
- Rijndael optimizations and cleanups
- Changed CipherMode<B> to BlockCipherMode(B*)
- Fixed a nasty bug in pipe_unixfd
- Added portions of the BigInt code into the main library
- Support for VAX, SH, POWER, PowerPC-64, Intel C++

15.202 Version 0.7.6, 2001-10-14

- Fixed several serious bugs in SecureVector created in 0.7.5
- Square optimizations
- Fixed shared objects on MacOS X and HP-UX
- Fixed static libs for KCC 4.0; works with KCC 3.4g as well
- Full support for Athlon and K6 processors using GCC
- Added a table of prime numbers < 2**16 (primes.h)
- Some minor documentation updates

15.203 Version 0.7.5, 2001-08-19

- Split checksum.h into adler32.h, crc24.h, and crc32.h
- Split modes.h into cbc.h, cfb.h, and ofb.h
- CBC_wPadding* has been replaced by CBC_Encryption and CBC_Decryption
- Added OneAndZeros and NoPadding methods for CBC
- Added Lion, a very fast block cipher construction
- Added an S2K base class (s2k.h) and an OpenPGP_S2K class (pgp_s2k.h)
- Basic types (ciphers, hashes, etc) know their names now (call name())
- Changed the EntropySource type somewhat
- Big speed-ups for ISAAC, Adler32, CRC24, and CRC32
- Optimized CAST-256, DES, SAFER-SK, Serpent, SEAL, MD2, and RIPEMD-160
- Some semantics of SecureVector have changed slightly

- The mlock module has been removed for the time being
- Added string handling functions for hashes and MACs
- Various non-user-visible cleanups
- Shared library soname is now set to the full version number

15.204 Version 0.7.4, 2001-07-15

- New modules: Zlib, gettimeofday and x86 RTC timers, Unix I/O for Pipe
- Fixed a vast number of errors in the config script/makefile/specfile
- Pipe now has a stdio(3) interface as well as C++ iostreams
- ARC4 supports skipping the first N bytes of the cipher stream (ala MARK4)
- Bzip2 supports decompressing multiple concatenated streams, and flushing
- Added a simple 'overall average' score to the benchmarks
- Fixed a small bug in the POSIX timer module
- Removed a very-unlikely-to-occur bug in most of the hash functions
- filtbase.h now includes <iosfwd>, not <iostream>
- Minor documentation updates

15.205 Version 0.7.3, 2001-06-08

- Fix build problems on Solaris/SPARC
- Fix build problems with Perl versions < 5.6
- Fixed some stupid code that broke on a few compilers
- Added string handling functions to Pipe
- MISTY1 optimizations

15.206 Version 0.7.2, 2001-06-03

- Build system supports modules
- Added modules for mlock, a /dev/random EntropySource, POSIX1.b timers
- Added Bzip2 compression filter, contributed by Peter Jones
- GNU make no longer required (tested with 4.4BSD pmake and Solaris make)
- Fixed minor bug in several of the hash functions
- Various other minor fixes and changes
- Updates to the documentation

15.207 Version 0.7.1, 2001-05-16

- Rewrote configure script: more consistent and complete
- Made it easier to find out parameters of types at run time (opencl.h)
- New functions for finding the version being used (version.h)
- New SymmetricKey interface for Filters (symkey.h)
- InvalidKeyLength now records what the invalid key length was
- Optimized DES, CS-Cipher, MISTY1, Skipjack, XTEA
- Changed GOST to use correct S-box ordering (incompatible change)
- Benchmark code was almost totally rewritten
- Many more entries in the test vector file
- Fixed minor and idiotic bug in check.cpp

15.208 Version 0.7.0, 2001-03-01

- First public release