Botan Reference Guide

Release 3.5.0

The Botan Authors

2024-07-08

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

Examples of usage are included in this documentation, some of which are listed below:

- *Block Ciphers*
- *Cipher Modes*
- *Hash Functions*
- *KDFs*
- *MACs*
- *PBKDFs*
- *Key Agreement*
- *ECDSA*
- *Kyber*
- *RSA*
- *XMSS*
- *Stream Ciphers*
- *TLS Client*
- *TLS Client (PQC/hybrid)*
- *HTTPS Client*
- *TLS Server*
- *X.509*

You'll find additional examples of usage in the src/examples (https://github.com/randombit/botan/tree/master/src/examples) directory.

An additional 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 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 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 3, the tier-1 supported platforms are

- Linux x86-64, GCC 11.2 or later
- Linux x86-64, Clang 14 or later
- Linux aarch64, GCC 11.2 or later
- Linux ppc64le, GCC 11.2 or later
- Windows x86-64, Visual C++ 2022 or later

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 3, the tier-2 supported platforms are

- macOS x86-64, latest XCode Clang
- iOS aarch64, latest XCode Clang
- Windows x86-64, latest MinGW GCC
- Android aarch64, latest NDK Clang
- Linux arm32, GCC 11.2 or later
- Linux x86-32, GCC 11.2 or later
- FreeBSD x86-64, Clang 14 or later

Note: Notice that the minimum version requirements for XCode and NDK is different from other compilers. With GCC or Clang, we fix the minimum required compiler version and aim to maintain that support for the entire lifecycle of Botan 3. In contrast, for XCode and NDK the minimum version is floating; namely, we will only support the very latest version. It's possible earlier versions will work, but this is not guaranteed.

Note: As of May 2024, it is known that at least XCode 15.0 is required, since earlier versions did not support certain C++20 language features that the library uses.

Note: For Android, NDK 26 is required

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++20 compiler is fine but in practice, we only regularly test with GCC, Clang, and Visual C++. Several other compilers (such as IBM XLC, Intel C++, and Sun Studio) 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 May 2024.

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

End of life dates may be extended as circumstances warrant.

Branch	First Release	End of Active Development	End of Life
Botan 1.8	2008-12-08	2010-08-31	2016-02-13
Botan 1.10	2011-06-20	2012-07-10	2018-12-31
Botan 2	2017-01-06	2020-11-05	2024-12-31 or later
Botan 3	2023-04-11	?	2027-12-31 or later

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 Apple macOS/Darwin), along with POSIX-ish systems like QNX.

Note: Botan is available already in nearly all packaging systems (https://repology.org/project/botan/versions) so you probably only need to build from source if you need unusual options or are building for an old system which has out of date packages.

Currently systems such as VMS, OS/390, and OS/400 are not supported by the build system, primarily due to lack of access and interest. 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 3.x 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 `python3` or `/path/to/python3`:

```
$ python3 ./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 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 modules 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
INFO: Skipping (no enabled compression schemes): compression
INFO: Skipping (requires external dependency): boost bzip2 lzma 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`.

Some modules may be marked as 'deprecated' or 'experimental'. Deprecated modules are available and built by default, but they will be removed in a future release of the library. Use `--disable-deprecated-features` to disable all of these modules or `--disable-modules=MODS` for finer grained control. Experimental modules are under active development and not built by default. Their API may change in future minor releases. Applications may still enable and use such modules using `--enable-modules=MODS` or using `--enable-experimental-features` to enable all experimental features.

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 Common Build Targets

Build everthing that is configured:

```
$ make all
```

Build the unit test binary (`./botan-test` to run):

```
$ make tests
```

Build and run the tests:

```
$ make check
```

Build the documentation (Doxygen API reference and Sphinx handbook):

```
$ make docs
```

Install the library:

```
$ make install
```

Remove all generated artefacts:

```
$ make clean
```

4.3 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.

4.4 On Unix

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

```
$ ./configure.py [various options]
$ 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=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.

4.5 On macOS

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

To build a universal binary for macOS, for older macOs releases, 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"
```

for mac M1 on arm64, you can build the x86_64 arch version via Rosetta separately. Do this with with *arch -x86_64 configure.py –library-suffix=-x86_64* Then using lipo to create a fat binary. *lipo -create libbotan-arm64.dylib libbotan-x86_64.dylib -o libbotan.dylib*

4.6 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:

```
$ python3 configure.py --cc=msvc --os=windows
$ nmake
$ nmake check
$ nmake install
```

Micosoft's `nmake` does not support building multiple jobs in parallel, which is unfortunate when building on modern multicore machines. It is possible to use the (somewhat unmaintained) Jom (https://wiki.qt.io/Jom) build tool, which is a `nmake` compatible build system that supports parallel builds. Alternately, starting in Botan 3.2, there is additionally support for using the `ninja` build tool as an alternative to `nmake`:

```
$ python3 configure.py --cc=msvc --os=windows --build-tool=ninja
$ ninja
$ ninja check
$ ninja install
```

For MinGW, use:

```
$ python3 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).

4.7 Ninja Support

Starting in Botan 3.2, there is additionally support for the ninja (https://ninja-build.org) build system.

This is particularly useful on Windows as there the default build tool `nmake` does not support parallel jobs. The `ninja` based build also works on Unix and macOs systems.

Support for `ninja` is still new and there are probably some rough edges.

4.8 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.

4.9 For Android

Modern versions of Android NDK use Clang and support C++20. Simply configure using the appropriate NDK compiler and `ar` (`ar` only needed if building the static library). Here we build for Aarch64 targeting Android API 28:

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

If you are building for mobile development consider restricting the build to only what you need (see *Minimized Builds*)

4.9.1 Docker

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

```
sudo ANDROID_SDK_VER=29 ANDROID_ARCH=aarch64 src/scripts/docker-android.sh
```

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

4.10 Emscripten (WebAssembly)

To build for WebAssembly using Emscripten, try:

```
./configure.py --cpu=wasm --os=emscripten
make
```

This will produce HTML files `botan-test.html` and `botan.html` along with a static archive `libbotan-3.a` which can be linked with other modules.

4.11 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`.

4.12 Other Build-Related Tasks

4.12.1 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`.

4.12.2 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.

4.12.3 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-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.

4.12.4 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.

4.12.5 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.

4.12.6 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.

4.12.7 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 `getrandom`, `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.

4.12.8 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.

4.13 Building Applications

4.13.1 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.

4.13.2 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.

4.13.3 CMake

Starting in Botan 3.3.0 we provide a `botan-config.cmake` module to discover the installed library binaries and headers. This hooks into CMake's `find_package()` and comes with common features like version detection. Also, library consumers may specify which botan modules they require in `find_package()`.

Examples:

```
find_package(Botan 3.3.0)
find_package(Botan 3.3.0 COMPONENTS rsa ecdsa tls13)
find_package(Botan 3.3.0 OPTIONAL_COMPONENTS tls13_pqc)
```

4.14 Language Wrappers

4.14.1 Building the Python wrappers

The Python wrappers for Botan use ctypes and the C89 API so no special build step is required, just import botan3.py

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

4.15 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`.

4.16 Configure Script Options

4.16.1 `--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.

4.16.2 `--os=OS`

Set the target operating system.

4.16.3 --cc=COMPILER

Set the desired build compiler

4.16.4 --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.

4.16.5 --cc-bin=BINARY

Set path to compiler binary

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

4.16.6 --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.

4.16.7 --cxxflags=FLAGS

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

4.16.8 --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.

4.16.9 --ldflags=FLAGS

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

4.16.10 --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.

4.16.11 --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.

4.16.12 --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.

4.16.13 --compiler-cache

Specify a compiler cache (like ccache) to use for each compiler invocation.

4.16.14 --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.

4.16.15 --with-os-features=FEAT

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

4.16.16 --without-os-features=FEAT

Specify an OS feature to disable.

4.16.17 --enable-experimental-features

Enable all experimental modules and features. Note that these are unstable and may change or even be removed in future releases. Also note that individual experimental modules can be explicitly enabled using `--enable-modules=MODS`.

4.16.18 --disable-experimental-features

Disable all experimental modules and features. This is the default.

4.16.19 --enable-deprecated-features

Enable all deprecated modules and features. Note that these are scheduled for removal in future releases. This is the default.

4.16.20 --disable-deprecated-features

Disable all deprecated modules and features. Note that individual deprecated modules can be explicitly disabled using `--disable-modules=MODS`.

4.16.21 --disable-sse2

Disable use of SSE2 intrinsics

4.16.22 --disable-ssse3

Disable use of SSSE3 intrinsics

4.16.23 --disable-sse4.1

Disable use of SSE4.1 intrinsics

4.16.24 --disable-sse4.2

Disable use of SSE4.2 intrinsics

4.16.25 --disable-avx2

Disable use of AVX2 intrinsics

4.16.26 --disable-bmi2

Disable use of BMI2 intrinsics

4.16.27 --disable-rdrand

Disable use of RDRAND intrinsics

4.16.28 --disable-rdseed

Disable use of RDSEED intrinsics

4.16.29 --disable-aes-ni

Disable use of AES-NI intrinsics

4.16.30 --disable-sha-ni

Disable use of SHA-NI intrinsics

4.16.31 --disable-altivec

Disable use of AltiVec intrinsics

4.16.32 --disable-neon

Disable use of NEON intrinsics

4.16.33 --disable-armv8crypto

Disable use of ARMv8 Crypto intrinsics

4.16.34 --disable-powercrypto

Disable use of POWER Crypto intrinsics

4.16.35 --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.

4.16.36 --with-debug-info

Include debug symbols.

4.16.37 --with-sanitizers

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

4.16.38 --enable-sanitizers=SAN

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

4.16.39 --without-stack-protector

Disable stack smashing protections. **not recommended**

4.16.40 --with-coverage-info

Add coverage info

4.16.41 --disable-shared-library

Disable building a shared library

4.16.42 --disable-static-library

Disable building static library

4.16.43 --optimize-for-size

Optimize for code size.

4.16.44 --no-optimizations

Disable all optimizations for debugging.

4.16.45 --debug-mode

Enable debug info and disable optimizations

4.16.46 --amalgamation

Use amalgamation to build

4.16.47 --name-amalgamation

Specify an alternative amalgamation file name. By default we use *botan_all*.

4.16.48 --with-build-dir=DIR

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

4.16.49 --with-external-includedir=DIR

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

4.16.50 --with-external-libdir=DIR

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

4.16.51 --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.

4.16.52 --with-sysroot-dir=DIR

Use specified dir for system root while cross-compiling

4.16.53 --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").

4.16.54 --with-local-config=FILE

Include the contents of FILE into the generated build.h

4.16.55 --distribution-info=STRING

Set distribution specific version information

4.16.56 --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.

4.16.57 --werror-mode

Turns most warnings into errors.

4.16.58 --no-install-python-module

Skip installing Python module.

4.16.59 --with-python-versions=N.M

Where to install botan3.py. By default this is chosen to be the version of Python that is running `configure.py`.

4.16.60 --with-valgrind

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

4.16.61 --unsafe-fuzzer-mode

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

4.16.62 --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.

4.16.63 --with-fuzzer-lib=LIB

Specify an additional library that fuzzer binaries must link with.

4.16.64 --build-targets=BUILD_TARGETS

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

4.16.65 --without-documentation

Skip building/installing documentation

4.16.66 --with-sphinx

Use Sphinx to generate the handbook

4.16.67 --with-pdf

Use Sphinx to generate PDF doc

4.16.68 --with-rst2man

Use rst2man to generate a man page for the CLI

4.16.69 --with-doxygen

Use Doxygen to generate API reference

4.16.70 --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
```

4.16.71 --enable-modules=MODS

Enable some specific modules

4.16.72 --disable-modules=MODS

Disable some specific modules

4.16.73 --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.

4.16.74 --with-boost

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

4.16.75 --with-bzip2

Enable bzip2 compression

4.16.76 --with-lzma

Enable lzma compression

4.16.77 --with-zlib

Enable using zlib compression

4.16.78 --with-commoncrypto

Enable using CommonCrypto for certain operations

4.16.79 --with-sqlite3

Enable using sqlite3 for data storage

4.16.80 --with-tpm

Enable support for TPM

4.16.81 --program-suffix=SUFFIX

A string to append to all program binaries.

4.16.82 --library-suffix=SUFFIX

A string to append to all library names.

4.16.83 --prefix=DIR

Set the install prefix.

4.16.84 --docdir=DIR

Set the documentation installation dir.

4.16.85 --bindir=DIR

Set the binary installation dir.

4.16.86 --libdir=DIR

Set the library installation dir.

4.16.87 --mandir=DIR

Set the man page installation dir.

4.16.88 --includedir=DIR

Set the include file installation dir.

4.16.89 --list-modules

List all modules that could be enabled or disabled using *–enable-modules* or *–disable-modules*.

CHAPTER

FIVE

SEMANTIC VERSIONING

Starting with 2.0.0, Botan adopted semantic versioning. This means we endevour to make no change which will either break compilation of existing code, or cause different behavior in a way that will cause compatability issues. Such changes are reserved for new major versions.

If on upgrading to a new minor version, you encounter a problem where your existing code either fails to compile, or the code behaves differently in some way that causes trouble, it is probably a bug; please report it on Github.

There are important exceptions to the SemVer guarantees that you should be aware of, described in the following list.

5.1 Exception #1: Deriving from Library Classes

If you in your application derive a new class from a class in the library, we do not guarantee a future minor release will not break your code. For example, we may in a minor release introduce a new pure virtual function to a base class like `BlockCipher`, and implement it for all subclasses within the library. In this case your code would fail to compile until you implemented the new virtual function. Or we might rename or remove a protected function, or a protected member variable.

There is also an exception to this exception! The following classes are intended for derivation by applications, and are fully covered by SemVer:

- `Credentials_Manager`
- `Entropy_Source`
- `TLS::Callbacks`
- `TLS::Policy` (and subclasses thereof)
- `TLS::Stream<T>`

5.2 Exception #2: `BOTAN_UNSTABLE_API`

Certain functionality is available to users, and marked in the header using the macro `BOTAN_UNSTABLE_API`. These interfaces are not covered by SemVer and may change or even vanish in a minor release.

Usually these interfaces are to enable applications that need to do something "interesting", but we are not confident that the API is any good. Examples include interfaces allowing applications to write custom TLS extensions and custom public key operations.

5.3 Exception #3: Experimental modules

Certain modules can be marked as experimental in the build system. Such modules are not built by default. Any functionality exposed by such modules may change or vanish at any time without warning. See *Building The Library* for more information on enabling or disabling these modules.

5.4 Exception #4: Any function starting with _

For various technical reasons, some functions are available for public use but are really only intended for use by the library itself.

The developers denote such functions by starting them with an underscore (_). Any such function may change or disappear at any time.

CHAPTER

SIX

BOTAN 2.X TO 3.X MIGRATION

This is a guide on migrating applications from Botan 2.x to 3.0.

This guide attempts to be, but is not, complete. If you run into a problem while converting code that does not seem to be described here, please open an issue on GitHub (https://github.com/randombit/botan/issues).

6.1 Headers

Many headers have been removed from the public API.

In some cases, such as `datastor.h` or `tls_blocking.h`, the functionality presented was entirely deprecated, in which case it has been removed.

In other cases (such as `loadstor.h` or `rotate.h`) the header was really an implementation header of the library and not intended to be consumed as a public API. In these cases the header is still used internally, but not installed for application use.

However in most cases there is a better way of performing the same operations, which usually works in both 2.x and 3.x. For example, in 3.0 all of the algorithm headers (such as `aes.h`) have been removed. Instead you should create objects via the factory methods (in the case of AES, `BlockCipher::create`) which works in both 2.x and 3.0

6.1.1 Errata: `pk_ops.h`

Between Botan 3.0 and 3.2 the public header `pk_ops.h` was removed accidentally. This header is typically required for specialized applications that interface with dedicated crypto hardware. If you are migrating such an application, please make sure to use Botan 3.3 or newer.

6.2 Build Artifacts

For consistency with other platforms the DLL is now suffixed with the library's major version on Windows as well.

6.3 TLS

Starting with Botan 3.0 TLS 1.3 is supported. This development required a number of backward-incompatible changes to accomodate the protocol differences to TLS 1.2, which is still supported.

6.3.1 Build modules

The build module `tls` is now internal and contains common TLS helpers. Users have to explicitly enable `tls12` and/or `tls13`. Note that for Botan 3.0 it is not (yet) possible to exclusively enable TLS 1.3 at build time.

6.3.2 Removed Functionality

Functionality removed from the TLS implementation includes

- TLS 1.0, 1.1 and DTLS 1.0
- DSA ciphersuites
- anonymous ciphersuites
- SRP ciphersuites
- SEED ciphersuites
- Camellia CBC ciphersuites
- AES-128 OCB ciphersuites
- DHE_PSK ciphersuites
- CECPQ1 ciphersuites

6.3.3 enum classes

The publicly available C++ enums in the TLS namespace are now *enum class* and their member naming scheme was converted from *SHOUTING_SNAKE_CASE* to *CamelCase*.

6.3.4 Callbacks

A number of new callbacks were added with TLS 1.3. None of those new callbacks is mandatory to implement by applications, though. Additionally there are a few backward incompatible changes in callbacks that might require attention by some applications:

tls_record_received() / tls_emit_data()

Those callbacks now take *std::span<const uint8_t>* instead of *const uint8_t** with a *size_t* buffer length.

tls_session_established()

This callback provides a summary of the just-negotiated connection. It used to have a bool return value letting an application decide to store or discard the connection's resumption information. This use case is now provided via: *tls_should_persist_resumption_information()* which might be called more than once for a single TLS 1.3 connection.

tls_session_established is not a mandatory callback anymore but still allows applications to abort a connection given a summary of the negotiated characteristics. Note that this summary is not a persistable *Session* anymore.

tls_verify_cert_chain()

The parameter *ocsp_responses*, which was previously *std::shared_ptr<OCSP::Response>*, is now *std::optional<OCSP::Response>*.

tls_modify_extensions() / tls_examine_extensions()

These callbacks now have an additional parameter of type *Handshake_Type* that identify the TLS handshake message the extensions in question are residing in. TLS 1.3 makes much heavier use of such extensions in a wider range of messages to implement core protocol functionality.

tls_dh_agree() / tls_ecdh_agree() / tls_decode_group_param()

These callbacks were used as customization points for the TLS 1.2 key exchange in the TLS client. To allow similar (and more) customizations with the introduction of TLS 1.3, these callbacks were replaced with a more generic approach.

Key agreement is split into two callbacks, namely *tls_generate_ephemeral_key()* and *tls_ephemeral_key_agreement()*. Those are used in both clients and servers and in all protocol versions. *tls_decode_group_param()* is removed as it became obsolete by the replacement of the other two callbacks.

6.3.5 Policy

choose_key_exchange_group()

The new parameter *offered_by_peer* identifies the key exchange groups a peer has sent public exchange offerings for (in TLS 1.3 handshakes only). Choosing a key exchange group that is not listed is legal but will result in an additional network round trip (cf. "Hello Retry Request"). In TLS 1.2, this vector is always empty and can be ignored.

session_ticket_lifetime()

Now returns *std::chrono::seconds* rather than a bare *uint32_t*.

6.3.6 Credentials Manager

find_cert_chain(), cert_chain() and cert_chain_single_type()

These methods now have a *cert_signature_schemes* parameter that identifies a list of signature schemes the peer is willing to accept for signatures in certificates. Notably, this *does not necessarily* mean that the leaf certificate must feature a public key type able to generate one of those schemes.

private_key_for()

Applications must now provide a *std::shared_ptr<>* to the requested private key object instead of a raw pointer to better communicate the implementation's life-time expectations of this private key object.

6.3.7 Session and Ticket Handling

Old (pre-Botan 3.0) sessions won't load in Botan 3.0 anymore and should be discarded. For applications using *Session_Manager_SQL* or *Session_Manager_SQLite* discarding happens automatically on first access after the update.

With Botan 3.0 the session manager now is responsible for stateful session handling (backed by a database) and creation and management of stateless session tickets. The latter was previously handled transparently by the TLS implementation itself.

Therefore, TLS server applications that relied on Botan's default session management implementations (most notably *Session_Manager_SQLite* or *Session_Manager_In_Memory*) are advised to re-evaluate their choice. Have a look at *Session_Manager_Hybrid* to retain support for both stateful and stateless TLS sessions. TLS client applications may safely keep relying on the above-mentioned default implementations.

Applications implementing their own *Session_Manager* will need to adapt to the new base class API.

New API of Session Manager

TLS 1.3 removed the legacy resumption procedures based on session IDs or session tickets and combined them under the protocol's Pre-Shared Key mechanism. This new approach allows TLS servers to handle sessions both stateless (as self-contained encrypted and authenticated tickets) and stateful (identified with unique database handles).

To accomodates this flexibility the *Session_Manager* base class API has changed drastically and is now responsible for creation, storage and management of both stateful sessions and stateless session tickets. Sub-classes therefore gain full control over the session ticket's structure and content.

API details are documented in the class' doxygen comments.

The Session Object and its Handle

Objects of class *Session* are not aware of their "session ID" or their "session ticket" anymore. Instead, the new class *Session_Handle* encapsulates the session's identifier or ticket and accompanies the *Session* object where necessary.

6.4 Algorithms Removed

The algorithms CAST-256, MISTY1, Kasumi, DESX, XTEA, PBKDF1, MCEIES, CBC-MAC, Tiger, CECPQ1, and NewHope have been removed.

6.5 Certificate API shared_ptr

Previously the certificate store used `shared_ptr<X509_Certificate>` in various APIs. However starting in 2.4.0, `X509_Certificate` itself is a pimpl to a `shared_ptr`, making the outer shared pointer pointless. In 3.0 the certificate interfaces have changed to just consume and return `X509_Certificate`.

6.6 All Or Nothing Package Transform

This code was deprecated and has been removed.

6.7 Exception Changes

Several exceptions, mostly ones not used by the library, were removed.

A few others that were very specific (such as Illegal_Point) were replaced by throws of their immediate base class exception type.

The base class of Encoding_Error and Decoding_Error changed from Invalid_Argument to Exception. If you are explicitly catching Invalid_Argument, verify that you do not need to now also explicitly catch Encoding_Error and/or Decoding_Error.

6.8 X.509 Certificate Info Access

Previously `X509_Certificate::subject_info` and `issuer_info` could be used to query information about extensions. This is not longer the case; instead you should either call a specific function on `X509_Certificate` which returns the same information, or lacking that, iterate over the result of `X509_Certificate::v3_extensions`.

6.9 OCSP Response Validation

After mitigating CVE-2022-43705 the OCSP response signature validation was refactored. This led to the removal of the *OCSP::Response::check_signature()* method. If you must validate OCSP responses directly in your application please use the new method *OCSP::Response::find_signing_certificate()* and *OCSP::Response::verify_signature()*.

6.10 Use of enum class

Several enumerations where modified to become enum class, including DL_Group::Format, CRL_Code, EC_Group_Encoding, Signature_Format, Cipher_Dir, TLS::Extension_Code, TLS::Connection_Side, TLS::Record_Type, and TLS::Handshake_Type

In many cases the enumeration values were renamed from SHOUTING_CASE to CamelCase. In some cases where the enumeration was commonly used by applications (for example Signature_Format and Cipher_Dir) the old enumeration names are retained as deprecated variants.

6.11 ASN.1 enums

The enum ASN1_Tag has been split into ASN1_Type and ASN1_Class. Unlike ASN1_Tag, these new enums are enum class. The members of the enums have changed from SHOUTING_CASE to CamelCase, eg CONSTRUCTED is now Constructed.

Also an important change related to ASN1_Tag::PRIVATE. This enum value was incorrect, and actually was used for explicitly tagged context specific values. Now, ASN1_Class::Private refers to the correct class, but would lead to a different encoding vs 2.x's ASN1_Tag::PRIVATE. The correct value to use in 3.0 to match ASN1_Tag::PRIVATE is ASN1_Class::ExplicitContextSpecific.

6.12 Cipher Mode Granularity

Previously Cipher_Mode::update_granularity specified the minimum buffer size that must be provided during processing. However the value returned was often much larger than what was strictly required. In particular some modes can easily accept inputs as small as 1 byte, but their update_granularity was much larger to encourage best performance.

Now update_granularity returns the true minimum value, and the new Cipher_Mode::ideal_granularity returns a value which is a multiple of update_granularity sized for good performance.

If you are sizing buffers on the basis of update_granularity consider using ideal_granularity instead. Otherwise you may encounter performance regressions due to creating and processing very small buffers.

6.13 "SHA-160" and "SHA1"

Previously the library accepted "SHA-160" and "SHA1" alternative names for "SHA-1". This is no longer the case, you must use "SHA-1". Botan 2.x also recognizes "SHA-1".

6.14 PointGFp

This type is now named EC_Point

6.15 X509::load_key

Previously these functions returned a raw pointer. They now return a std::unique_ptr

6.16 PKCS11_Request::subject_public_key and X509_Certificate::subject_publi

These functions now return a unique_ptr

6.17 choose_sig_format removed

The freestanding functions choose_sig_format have been removed. Use X509_Object::choose_sig_format

6.18 DLIES Constructors

Previously the constructors to the DLIES classes took raw pointers, and retained ownership of them. They now consume std::unique_ptrs

6.19 Credentials_Manager::private_key_for

Previously this function returned a raw pointer, which the Credentials_Manager implementation had to keep alive "forever", since there was no way for it to know when or if the TLS layer had completed using the returned key.

Now this function returns std::shared_ptr<Private_Key>

6.20 OID operator+

OID operator+ allowed concatenating new fields onto an object identifier. This was not used at all within the library or the tests, and seems of marginal value, so it was removed.

If necessary in your application, this can be done by retrieving the vector of components from your source OID, push the new element onto the vector and create an OID from the result.

6.21 RSA with "EMSA1" padding

EMSA1 indicates that effectively the plain hash is signed, with no other padding. It is typically used for algorithms like ECSDA, but was allowed for RSA. This is now no longer implemented.

If you must generate such signatures for some horrible reason, you can pre-hash the message using a hash function as usual, and then sign using a "Raw" padding, which will allow you to sign any arbitrary bits with no preprocessing.

6.22 ECDSA/DSA with "EMSA1" padding

Previous versions of Botan required using a hash specifier like "EMSA1(SHA-256)" when generating or verifying ECDSA/DSA signatures, with the specified hash. The "EMSA1" was a reference to a now obsolete IEEE standard.

In Botan 3 the "EMSA1" notation is still accepted, but now also it is possible to simply use the name of the hash, eg "EMSA1(SHA-256)" becomes "SHA-256".

6.23 Signature Algorithm OIDs

In line with the previous entries, previously Botan used a string like "ECDSA/EMSA1(SHA-256)" to identify the OID 1.2.840.10045.4.3.2. Now it uses the string "ECDSA/SHA-256" instead, and does not recognize the EMSA1 variant at all (for example in `OID::from_string`).

6.24 Public Key Signature Padding

In previous versions Botan was somewhat lenient about allowing the application to specify using a hash which was in fact incompatible with the algorithm. For example, Ed25519 signatures are *always* generated using SHA-512; there is no choice in the matter. In the past, requesting using some other hash, say SHA-256, would be silently ignored. Now an exception is thrown, indicating the desired hash is not compatible with the algorithm.

In previous versions, various APIs required that the application specify the hash function to be used. In most cases this can now be omitted (passing an empty string) and a suitable default will be chosen.

6.25 Discrete Logarithm Key Changes

Keys based on the discrete logarithm problem no longer derive from the DL_Scheme_PrivateKey and DL_Scheme_PublicKey classes; these classes have been removed.

Functions to access DL algorithm internal fields (such as the integer value of the private key using `get_x`) have been removed. If you need access to this information you can use the new `get_int_field` function.

The constructors of the DL scheme private keys have changed. Previously, loading and creating a key used the same constructor, namely one taking arguments `(DL_Group, RandomNumberGenerator&, BigInt x = 0)` and then the behavior of the constructor depend on if `x` was zero (in which case a new key was created) or otherwise if `x` was non-zero then it was taken as the private key. Now there are two constructors, one taking a random number generator and a group, which generates a new key, and a second taking a group and an integer, which loads an existing key.

6.26 XMSS Signature Changes

The logic to derive WOTS+ private keys from the seed contained in the XMSS private key has been updated according to the recommendations in NIST SP 800-208. While signatures created with old private keys are still valid using the old public key, new valid signatures cannot be created. To still support legacy private XMSS keys, they can be used by passing `WOTS_Derivation_Method::Botan2x` to the constructor of the `XMSS_PrivateKey`.

Private XMSS keys created this way use the old derivation logic and can therefore generate new valid signatures. It is recommended to use `WOTS_Derivation_Method::NIST_SP800_208` (default) when creating new XMSS keys.

6.27 Random Number Generator

Fetching a large number of bytes via *randomize_with_input()* from a stateful RNG will now incorporate the provided "input" data in the first request to the underlying DRBG only. This applies to such DRBGs that pose a limit on the number of bytes per request (most notable `HMAC_DRBG` with a 64kB default). Botan 2.x (erroneously) applied the input to *all* underlying DRBG requests in such cases.

Applications that rely on a static seed for deterministic RNG output might observe a different byte stream in such cases. As a workaround, users are advised to "mimick" the legacy behaviour by manually pulling from the RNG in "byte limit"-sized chunks and provide the "input" with each invocation.

CHAPTER

SEVEN

OPENSSL 1.1 TO BOTAN 3.X MIGRATION

This aims to be a rough guide for migrating applications from OpenSSL 1.1 to Botan 3.x.

This guide attempts to be, but is not, complete. If you run into a problem while migrating code that does not seem to be described here, please open an issue on GitHub (https://github.com/randombit/botan/issues).

Note: The OpenSSL code snippets in this guide may not be 100% correct. They are intended to show the differences in using OpenSSL's and Botan's APIs rather to be a complete and correct example.

7.1 General Remarks

- Botan is a C++ library, whereas OpenSSL is a C library
- Botan also provides a *C API* for most of its functionality, but it is not a 1:1 mapping of the C++ API
- With OpenSSL's API, there are sometimes multiple ways to achieve the same result, whereas Botan's API is more consistent
- OpenSSL's API is mostly underdocumented, whereas Botan targets 100% Doxygen coverage for all public API
- It is often hard to find example code for OpenSSL, whereas Botan provides many *examples* and lots of test code (https://github.com/randombit/botan/tree/master/src/tests).

7.2 X.509

Consider the following application code that uses OpenSSL to verify a certificate chain consisting of an end-entity certificate, two untrusted intermediate certificates, and a trusted root certificate.

```
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>

int main() {
    // Create a new X.509 store
    X509_STORE *store = X509_STORE_new();

    // Load the root certificate
    FILE* rootCertFileHandle = fopen("root.crt", "r");
    X509* rootCert = PEM_read_X509(rootCertFileHandle, NULL, NULL, NULL);
```

(continues on next page)

(continued from previous page)

```
    X509_STORE_add_cert(store, rootCert);
    fclose(rootCertFileHandle);

    // Create a new X.509 store context
    X509_STORE_CTX *ctx = X509_STORE_CTX_new();
    X509_STORE_CTX_init(ctx, store, NULL, NULL);

    // Load the intermediate certificates
    FILE* intermediateCertFileHandle1 = fopen("int2.crt", "r");
    FILE* intermediateCertFileHandle2 = fopen("int1.crt", "r");
    X509* intermediateCert1 = PEM_read_X509(intermediateCertFileHandle1, NULL, NULL,␣
↪NULL);
    X509* intermediateCert2 = PEM_read_X509(intermediateCertFileHandle2, NULL, NULL,␣
↪NULL);
    X509_STORE_CTX_trusted_stack(ctx, sk_X509_new_null());
    sk_X509_push(X509_STORE_CTX_get0_untrusted(ctx), intermediateCert1);
    sk_X509_push(X509_STORE_CTX_get0_untrusted(ctx), intermediateCert2);
    fclose(intermediateCertFileHandle1);
    fclose(intermediateCertFileHandle2);

    // Load the end-entity certificate
    FILE* endEntityCertFileHandle = fopen("ee.crt", "r");
    X509* endEntityCert = PEM_read_X509(endEntityCertFileHandle, NULL, NULL, NULL);
    X509_STORE_CTX_set_cert(ctx, endEntityCert);
    fclose(endEntityCertFileHandle);

    // Verify the certificate chain
    int result = X509_verify_cert(ctx);
    if(result != 1) {
        // Verification failed
        X509_STORE_CTX_free(ctx);
        X509_STORE_free(store);
        return -1;
    }

    // Verification succeeded
    X509_STORE_CTX_free(ctx);
    X509_STORE_free(store);
    return 0;
}
```

First, we create a new `X509_STORE` object and add the trusted root certificate. Then we add the intermediate certificates to the untrusted certificate stack. Finally, we set the end-entity certificate and call `X509_verify_cert()` to verify the whole certificate chain.

Here is the equivalent C++ code using Botan:

```
#include <botan/certstor_system.h>
#include <botan/x509cert.h>
#include <botan/x509path.h>

int main() {
   // Create a certificate store and add a locally trusted CA certificate
```

(continues on next page)

(continued from previous page)

```
   Botan::Certificate_Store_In_Memory customStore;
   customStore.add_certificate(Botan::X509_Certificate("root.crt"));

   // Additionally trust all system-specific CA certificates
   Botan::System_Certificate_Store systemStore;
   std::vector<Botan::Certificate_Store*> trusted_roots{&customStore, &systemStore};

   // Load the end entity certificate and two untrusted intermediate CAs from file
   std::vector<Botan::X509_Certificate> end_certs;
   end_certs.emplace_back(Botan::X509_Certificate("ee.crt"));    // The end-entity␣
↪certificate, must come first
   end_certs.emplace_back(Botan::X509_Certificate("int2.crt"));  // intermediate 2
   end_certs.emplace_back(Botan::X509_Certificate("int1.crt"));  // intermediate 1

   // Optional: Set up restrictions, e.g. min. key strength, maximum age of OCSP␣
↪responses
   Botan::Path_Validation_Restrictions restrictions;

   // Optional: Specify usage type, compared against the key usage in end_certs[0]
   Botan::Usage_Type usage = Botan::Usage_Type::UNSPECIFIED;

   // Optional: Specify hostname, if not empty, compared against the DNS name in end_
↪certs[0]
   std::string hostname;

   Botan::Path_Validation_Result validationResult =
      Botan::x509_path_validate(end_certs, restrictions, trusted_roots, hostname, usage);

   if(!validationResult.successful_validation()) {
      // call validationResult.result() to get the overall status code
      return -1;
   }

   return 0;  // Verification succeeded
}
```

First, we create a `Certificate_Store_In_Memory` object and add the trusted root certificate. Additionally, we use `System_Certificate_Store` to load all trusted root certificates from the operating system's certificate store to trust. Botan provides several different *Certificate Stores*, including certificate stores that load certificates from a directory or from an SQL database. It even provides an interface for implementing your own certificate store. Then we add the end-entity certificate and the intermediate certificates to the `end_certs` chain. Optionally, we can set up path validation restrictions, specify usage and hostname for DNS, and then call `x509_path_validate()` to *verify the certificate chain*.

7.3 Random Number Generation

Consider the following application code to generate random bytes using OpenSSL.

```
#include <openssl/rand.h>
#include <iostream>

int main() {
    unsigned char buffer[16]; // Buffer to hold 16 random bytes

    if(RAND_bytes(buffer, sizeof(buffer)) != 1) {
        std::cerr << "Error generating random bytes.\n";
        return 1;
    }

    // Print the random bytes in hexadecimal format
    for(int i = 0; i < sizeof(buffer); i++) {
        printf("%02X", buffer[i]);
    }
    printf("\n");

    return 0;
}
```

This example uses the `RAND_bytes()` function to generate 16 random bytes, e.g., for a 128-bit AES key, and prints it on the console.

Here is the equivalent C++ code using Botan:

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

int main() {
    Botan::AutoSeeded_RNG rng;

    const Botan::secure_vector<uint8_t> buffer = rng.random_vec(16);

    // Print the random bytes in hexadecimal format
    std::cout << Botan::hex_encode(buffer) << std::endl;

    return 0;
}
```

This snippet uses the `AutoSeeded_RNG` class to generate the 16 random bytes. Botan provides different *Random Number Generators*, including system-specific as well as system-independent software and hardware-based generators, and a comprehensive interface for implementing your own random number generator, if required. `AutoSeeded_RNG` is the recommended random number generator for most applications. The `random_vec()` function returns the requested number of random bytes passed. Botan provides a `hex_encode()` function that converts the random bytes to a hexadecimal string.

7.4 Hash Functions

Consider the following application code to hash some data using OpenSSL.

```
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <iostream>
#include <vector>

void printHash(EVP_MD_CTX* ctx, const std::string& name) {
    unsigned char hash[EVP_MAX_MD_SIZE];
    unsigned int lengthOfHash = 0;

    EVP_DigestFinal_ex(ctx, hash, &lengthOfHash);

    std::cout << name << ": ";
    for(unsigned int i = 0; i < lengthOfHash; ++i) {
        std::cout << std::hex << std::setw(2) << std::setfill('0') << (int)hash[i];
    }
    std::cout << std::endl;
}

int main() {
    EVP_MD_CTX *ctx1 = EVP_MD_CTX_new();
    EVP_MD_CTX *ctx2 = EVP_MD_CTX_new();
    EVP_MD_CTX *ctx3 = EVP_MD_CTX_new();

    EVP_DigestInit_ex(ctx1, EVP_sha256(), NULL);
    EVP_DigestInit_ex(ctx2, EVP_sha384(), NULL);
    EVP_DigestInit_ex(ctx3, EVP_sha3_512(), NULL);

    std::vector<uint8_t> buffer(2048);
    while(std::cin.good()) {
        std::cin.read(reinterpret_cast<char*>(buffer.data()), buffer.size());
        std::streamsize bytesRead = std::cin.gcount();

        EVP_DigestUpdate(ctx1, buffer.data(), bytesRead);
        EVP_DigestUpdate(ctx2, buffer.data(), bytesRead);
        EVP_DigestUpdate(ctx3, buffer.data(), bytesRead);
    }

    printHash(ctx1, "SHA-256");
    printHash(ctx2, "SHA-384");
    printHash(ctx3, "SHA-3-512");

    EVP_MD_CTX_free(ctx1);
    EVP_MD_CTX_free(ctx2);
    EVP_MD_CTX_free(ctx3);

    return 0;
}
```

This example uses the `EVP_DigestInit_ex()`, `EVP_DigestUpdate()`, and `EVP_DigestFinal_ex()` functions to

hash data using SHA-256, SHA-384, and SHA-3-512. The `printHash()` function is used to print the hash values in hexadecimal format.

Here is the equivalent C++ code using Botan:

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

#include <iostream>

int main() {
   const auto hash1 = Botan::HashFunction::create_or_throw("SHA-256");
   const auto hash2 = Botan::HashFunction::create_or_throw("SHA-384");
   const auto hash3 = Botan::HashFunction::create_or_throw("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()) << '\n';
   std::cout << "SHA-384: " << Botan::hex_encode(hash2->final()) << '\n';
   std::cout << "SHA-3: " << Botan::hex_encode(hash3->final()) << '\n';
   return 0;
}
```

This example uses the `HashFunction` interface to hash data using SHA-256, SHA-384, and SHA-3-512. The `hash()` function is used to hash the data and the `output_length()` function is used to determine the length of the hash value. Botan provides a comprehensive list of *hash functions*, including all SHA-2 and SHA-3 variants, as well as *message authentication codes* and *key derivation functions*.

7.5 Symmetric Encryption

Consider the following application code to encrypt some data with AES using OpenSSL.

```
#include <openssl/aes.h>
#include <openssl/evp.h>
#include <iostream>
#include <iomanip>

int main() {
    // Hex-encoded key and plaintext block
    const char* key_hex =
↪"000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F";
    const char* plaintext_hex = "00112233445566778899AABBCCDDEEFF";

    // Convert hex-encoded key and plaintext block to binary
```

(continues on next page)

(continued from previous page)

```
    unsigned char key[32], plaintext[16];
    for(int i = 0; i < 32; i++) {
        sscanf(&key_hex[i*2], "%02x", &key[i]);
    }
    for(int i = 0; i < 16; i++) {
        sscanf(&plaintext_hex[i*2], "%02x", &plaintext[i]);
    }

    // Encrypt
    unsigned char ciphertext[16], iv_enc[AES_BLOCK_SIZE] = {0};
    EVP_CIPHER_CTX *ctx_enc = EVP_CIPHER_CTX_new();
    EVP_EncryptInit_ex(ctx_enc, EVP_aes_256_cbc(), NULL, key, iv_enc);
    int outlen1;
    EVP_EncryptUpdate(ctx_enc, ciphertext, &outlen1, plaintext, sizeof(plaintext));
    EVP_EncryptFinal_ex(ctx_enc, ciphertext + outlen1, &outlen1);

    // Print ciphertext in hexadecimal format
    for(int i = 0; i < 16; i++) {
        printf("%02X", ciphertext[i]);
    }
    printf("\n");

    return 0;
}
```

This example uses the `EVP_EncryptInit_ex()`, `EVP_EncryptUpdate()`, and `EVP_EncryptFinal_ex()` functions to encrypt a 128-bit plaintext block with a 256-bit key using AES. The key and plaintext block are hex-decoded and converted to binary before encryption.

Here is the equivalent C++ code using Botan:

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

#include <iostream>

int main() {
   Botan::AutoSeeded_RNG rng;

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

   const auto enc = Botan::Cipher_Mode::create_or_throw("AES-128/CBC/PKCS7",␣
↪Botan::Cipher_Dir::Encryption);
   enc->set_key(key);

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

(continues on next page)

(continued from previous page)

```
   // 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;
}
```

This example uses the `CipherMode` interface to encrypt a 128-bit plaintext block with a 256-bit key using AES in CBC mode with PKCS#7 padding. The `set_key()` function is used to set the key and the `start()` and `finish()` functions are used to encrypt the plaintext block.

To learn more about the `BlockCipher` and `CipherMode` interfaces, including a list of all available block ciphers and cipher modes, see the *Block Ciphers* and *Cipher Modes* handbook sections.

7.6 Asymmetric Encryption

Consider the following application code to encrypt some data with RSA using OpenSSL.

```
#include <openssl/evp.h>
#include <openssl/pem.h>
#include <openssl/rsa.h>
#include <openssl/err.h>
#include <string.h>
#include <stdio.h>

int main() {
    // Load public key
    FILE* pubKeyFile = fopen("public.pem", "r");
    if(pubKeyFile == NULL) {
        fprintf(stderr, "Error opening public key file.\n");
        return 1;
    }
    EVP_PKEY* pubKey = PEM_read_PUBKEY(pubKeyFile, NULL, NULL, NULL);
    fclose(pubKeyFile);

    // Load private key
    FILE* privKeyFile = fopen("private.pem", "r");
    if(privKeyFile == NULL) {
        fprintf(stderr, "Error opening private key file.\n");
        return 1;
    }
    EVP_PKEY* privKey = PEM_read_PrivateKey(privKeyFile, NULL, NULL, NULL);
    fclose(privKeyFile);

    // String to encrypt
```

(continues on next page)

(continued from previous page)

```
    unsigned char* 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.";
    size_t plaintext_len = strlen(plaintext);

    // Encrypt
    EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pubKey, NULL);
    EVP_PKEY_encrypt_init(ctx);
    EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_PADDING);
    EVP_PKEY_CTX_set_rsa_oaep_md(ctx, EVP_sha256());
    size_t encrypted_len;
    EVP_PKEY_encrypt(ctx, NULL, &encrypted_len, plaintext, plaintext_len);
    unsigned char* encrypted = (unsigned char*)malloc(encrypted_len);
    EVP_PKEY_encrypt(ctx, encrypted, &encrypted_len, plaintext, plaintext_len);

    // Decrypt
    EVP_PKEY_CTX *ctx2 = EVP_PKEY_CTX_new(privKey, NULL);
    EVP_PKEY_decrypt_init(ctx2);
    EVP_PKEY_CTX_set_rsa_padding(ctx2, RSA_PKCS1_OAEP_PADDING);
    EVP_PKEY_CTX_set_rsa_oaep_md(ctx2, EVP_sha256());
    size_t decrypted_len;
    EVP_PKEY_decrypt(ctx2, NULL, &decrypted_len, encrypted, encrypted_len);
    unsigned char* decrypted = (unsigned char*)malloc(decrypted_len + 1);
    EVP_PKEY_decrypt(ctx2, decrypted, &decrypted_len, encrypted, encrypted_len);
    decrypted[decrypted_len] = '\0';

    // Print encrypted and decrypted strings
    for(size_t i = 0; i < encrypted_len; i++) {
        printf("%02X", encrypted[i]);
    }
    printf("\n");
    printf("%s\n", decrypted);

    // Clean up
    EVP_PKEY_free(pubKey);
    EVP_PKEY_free(privKey);
    EVP_PKEY_CTX_free(ctx);
    EVP_PKEY_CTX_free(ctx2);
    free(encrypted);
    free(decrypted);

    return 0;
}
```

This example uses OpenSSL'S EVP interface, specifically `EVP_PKEY_encrypt()` and `EVP_PKEY_decrypt()` functions to encrypt and decrypt a string using RSA. The public and private keys are loaded from files. The `EVP_PKEY_CTX_set_rsa_padding()` and `EVP_PKEY_CTX_set_rsa_oaep_md()` functions are used to set the padding scheme and the hash function for RSA-OAEP.

Here is the equivalent C++ code using Botan:

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

(continues on next page)

(continued from previous page)

```
#include <botan/pkcs8.h>
#include <botan/pubkey.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());
   Botan::AutoSeeded_RNG rng;

   // load keypair
   Botan::DataSource_Stream in(argv[1]);
   auto kp = Botan::PKCS8::load_key(in);

   // encrypt with pk
   Botan::PK_Encryptor_EME enc(*kp, rng, "OAEP(SHA-256)");
   std::vector<uint8_t> ct = enc.encrypt(pt, rng);

   // decrypt with sk
   Botan::PK_Decryptor_EME dec(*kp, rng, "OAEP(SHA-256)");
   Botan::secure_vector<uint8_t> pt2 = dec.decrypt(ct);

   std::cout << "\nenc: " << Botan::hex_encode(ct) << "\ndec: " << Botan::hex_
↪encode(pt2);

   return 0;
}
```

This example uses the `PK_Encryptor_EME` and `PK_Decryptor_EME` classes to *encrypt and decrypt*. a message using *RSA*. The public and private keys are *loaded from files*. The padding scheme and *hash function* are passed as a string parameter.

7.7 Asymmetric Signatures

Consider the following application code to sign some data with ECDSA using OpenSSL.

```
#include <openssl/ec.h>
#include <openssl/obj_mac.h>
#include <openssl/err.h>
#include <openssl/ecdsa.h>
#include <openssl/pem.h>
#include <openssl/sha.h>
#include <iostream>

int main() {
```

(continues on next page)

(continued from previous page)

```
    EC_KEY *ec_key = EC_KEY_new_by_curve_name(NID_secp521r1);

    if(ec_key == NULL) {
        fprintf(stderr, "Error creating EC_KEY structure.\n");
        return 1;
    }

    if(!EC_KEY_generate_key(ec_key)) {
        fprintf(stderr, "Error generating key.\n");
        ERR_print_errors_fp(stderr);
        EC_KEY_free(ec_key);
        return 1;
    }

    // String to sign
    std::string plaintext = "This is a tasty burger!";

    // Hash the plaintext
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256((unsigned char*)plaintext.c_str(), plaintext.size(), hash);

    // Sign the hash
    ECDSA_SIG* sig = ECDSA_do_sign(hash, SHA256_DIGEST_LENGTH, ec_key);
    if(sig == NULL) {
        std::cerr << "Error signing: " << ERR_error_string(ERR_get_error(), NULL) << "\n
↪";
        return 1;
    }

    // Print the signature
    const BIGNUM* r;
    const BIGNUM* s;
    ECDSA_SIG_get0(sig, &r, &s);
    char* r_hex = BN_bn2hex(r);
    char* s_hex = BN_bn2hex(s);
    std::cout << "Signature: (" << r_hex << ", " << s_hex << ")\n";

    // Clean up
    EC_KEY_free(ec_key);
    ECDSA_SIG_free(sig);
    OPENSSL_free(r_hex);
    OPENSSL_free(s_hex);
    return 0;
}
```

This snippet uses OpenSSL's ECDSA interface, specifically `ECDSA_do_sign()`, to sign a string message using ECDSA. The private key is loaded from a file. The `SHA256()` function is used to hash the plaintext before signing.

Here is the equivalent C++ code using Botan:

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

(continues on next page)

(continued from previous page)

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

#include <iostream>

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

   const std::string message("This is a tasty burger!");

   // sign data
   Botan::PK_Signer signer(key, rng, "SHA-256");
   signer.update(message);
   std::vector<uint8_t> signature = signer.signature(rng);
   std::cout << "Signature:\n" << Botan::hex_encode(signature);

   // now verify the signature
   Botan::PK_Verifier verifier(key, "SHA-256");
   verifier.update(message);
   std::cout << "\nis " << (verifier.check_signature(signature) ? "valid" : "invalid");
   return 0;
}
```

This example uses the `PK_Signer` and `PK_Verifier` classes to sign and verify a message using *ECDSA*. The private key is similary *loaded from a file*. The *hash function* is passed as a string parameter. `PK_Verifier::check_signature()` is used to *verify the signature*.

CHAPTER

EIGHT

API REFERENCE

8.1 Footguns

This section notes areas where certain usages can cause confusing bugs or problems.

8.1.1 Static Objects

If you maintain `static` variables which hold Botan objects, you will perhaps find that in some circumstances your application crashes in strange ways on shutdown. That is because, at least on some operating systems, Botan uses a locked memory pool as backing storage for the `secure_vector` type. This pool allocates out of pages which have been locked into memory using `mlock` or `VirtualLock` system calls.

If your variable happens to be destroyed before the pool, all is well. If the pool happens to be destroyed before the variable, then when the object goes to free its memory, a crash will occur.

This is basically the famous C++ "Static Initialization Order Fiasco", except in reverse.

The best course of action is to avoid `static` variables. If that is impossible or inconvenient, one option is to disable the pool, either at build time (disable the `locking_allocator` module) or at runtime. Unfortunately the runtime setting requires setting an environment variable (see *Environment Variables*), and doing so consistently *prior to static intialization* is not trivial, due to the previously mentioned fiasco. One option might be to use GCC's `constructor` function attribute.

Another approach is to use the utility class `Allocator_Initializer` (declared in `mem_ops.h`) as an associated `static` variable in your code. As long as the `Allocator_Initializer` is created *before* your static variables, that means the allocator is created before your object, and thus will be destroyed after your object is destroyed.

Ideally a more satisfactory solution to this issue could be found, especially given the difficulty of disabling the pool at runtime.

8.1.2 Multithreaded Access

It is perfectly safe to use the library from multiple threads.

It is *not* safe to use the same object from multiple threads, without some form of external serialization or locking.

There are a few exceptions to this rule, where the type itself maintains an internal mutexes. This will be noted in the respective documentation for that type.

8.1.3 Use of *fork*

If you use the *fork* syscall in your code, and attempt to use the library in both processes, likely bad things will happen due to internal locks. You can avoid this problem by either at build time disabling the features associated with these locks (namely `locking_allocator` and `thread_utils`) or disabling them at runtime using *Environment Variables*, ideally at the very start of *main*.

8.2 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

Added 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

Added 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
```

8.3 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).

8.4 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.

8.4.1 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 `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`, `BCryptGenRandom`, 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 without locking, 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 again 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.

Note: There is no reason to use this class directly unless your application requires HMAC-DRBG with specific parameters or options. Usually this would be for some standards conformance reason. If you just want a userspace RNG, use `AutoSeeded_RNG`.

`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.

8.4.2 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 (`/dev/urandom`, `getrandom`, `arc4random`, `BCryptGenRandom`, or `RtlGenRandom`).
- Processor provided RNG outputs (RDRAND, RDSEED, DARN) are used if available, but not counted as contributing entropy
- The `getentropy` call is used on OpenBSD, FreeBSD, and macOS
- `/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.

8.4.3 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.

8.5 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`.

std::unique_ptr<*HashFunction*> **new_object**()

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

std::unique_ptr<*HashFunction*> **copy_state**()

Return a newly allocated HashFunction object of the same type as this one, whose internal state matches the current state of this.

8.5.1 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() {
   const auto hash1 = Botan::HashFunction::create_or_throw("SHA-256");
   const auto hash2 = Botan::HashFunction::create_or_throw("SHA-384");
   const auto hash3 = Botan::HashFunction::create_or_throw("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()) << '\n';
   std::cout << "SHA-384: " << Botan::hex_encode(hash2->final()) << '\n';
   std::cout << "SHA-3: " << Botan::hex_encode(hash3->final()) << '\n';
   return 0;
}
```

8.5.2 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.

Algorithm specification name: `BLAKE2b(<optional output bits>)` (reported name) / `Blake2b(<optional output bits>)`

- Output bits defaults to 512.
- Examples: `BLAKE2b(256)`, `BLAKE2b(512)`, `BLAKE2b`

BLAKE2s

Available if `BOTAN_HAS_BLAKE2S` is defined.

A recently designed hash function. Very fast on 32-bit processors. Can output a hash of any length between 1 and 32 bytes, this is specified by passing a value to the constructor with the desired length.

Named like "Blake2s" which selects default 256-bit output, or as "Blake2s(128)" to select 128 bits of output.

Algorithm specification name: `BLAKE2s(<optional output bits>)` (reported name) / `Blake2s(<optional output bits>)`

- Output bits defaults to 256.
- Examples: `BLAKE2s(128)`, `BLAKE2s(256)`, `BLAKE2s`

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.

Algorithm specification name: `GOST-R-34.11-94` (reported name) / `GOST-34.11`

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.

Algorithm specification name: `Keccak-1600(<optional output bits>)`

- Output bits defaults to 512.
- Examples: `Keccak-1600(256)`, `Keccak-1600(512)`, `Keccak-1600`

MD4

An old and now broken hash function. Available if `BOTAN_HAS_MD4` is defined.

> **Warning:** MD4 collisions can be easily created. There is no safe cryptographic use for this function.

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

Algorithm specification name: `MD4`

MD5

An old and now broken hash function. Available if `BOTAN_HAS_MD5` is defined.

> **Warning:** MD5 collisions can be easily created. MD5 should never be used for signatures.

Algorithm specification name: `MD5`

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. Prefer SHA-2 or SHA-3 in new code.

Algorithm specification name: `RIPEMD-160`

SHA-1

Available if `BOTAN_HAS_SHA1` is defined.

Widely adopted NSA designed hash function. Use SHA-2 or SHA-3 in new code.

> **Warning:** SHA-1 collisions can now be created by moderately resourced attackers. It must never be used for signatures.

Algorithm specification name: `SHA-1`

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.

Algorithm specification names:

- `SHA-224`
- `SHA-256`

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.

Algorithm specification names:

- `SHA-384`
- `SHA-512`
- `SHA-512-256`

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.

Algorithm specification name: `SHA-3(<optional output bits>)`

- Output bits defaults to 512.
- Examples: `SHA-3(256)`, `SHA-3(512)`, `SHA-3`

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.

Algorithm specification names:

- `SHAKE-128(<output bits>)`, e.g. `SHAKE-128(128)`
- `SHAKE-256(<output bits>`, e.g. `SHAKE-256(256)`

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)".

Algorithm specification name:

- `Skein-512(<optional output bits>)`
  - Output bits defaults to 512.
  - Examples: `Skein-512(256)`, `Skein-512(512)`, `Skein-512`
- `Skein-512(<output bits>,<personalization>)`, e.g. `Skein-512(512,Test)`

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.

Algorithm specification name: `SM3`

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.

Algorithm specification names:

- `Streebog-256`
- `Streebog-512`

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.

Algorithm specification name: Whirlpool

8.5.3 Hash Function Combiners and Modifiers

These are functions which combine multiple hash functions, or modify the output of 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.

Algorithm specification name: Parallel(<HashFunction>,<HashFunction>,...), e.g. Parallel(SHA-256,SHA-512), Parallel(MD5,SHA-1,SHA-256,SHA-512)

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.

Algorithm specification name: Comb4P(<HashFunction>,<HashFunction>), e.g. Comb4P(SHA-1,RIPEMD-160)

Truncated

Available if BOTAN_HAS_TRUNCATED_HASH is defined.

Wrapper class to truncate underlying hash function output to a given number of bits. The leading bits are retained.

Algorithm specification name: Truncated(<HashFunction>,<output bits>), e.g. Truncated(SHAKE-128(256),42)

8.5.4 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.

Algorithm specification name: `Adler32`

CRC24

Available if `BOTAN_HAS_CRC24` is defined.

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

Algorithm specification name: `CRC32`

CRC32

Available if `BOTAN_HAS_CRC32` is defined.

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

Algorithm specification name: `CRC32`

8.6 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.

std::unique_ptr<*BlockCipher*> **new_object**() const

Return a newly allocated BlockCipher object of the same type as this one. The new object is unkeyed.

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.

8.6.1 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");
   const auto cipher = Botan::BlockCipher::create_or_throw("AES-256");
   cipher->set_key(key);
   cipher->encrypt(block);
   std::cout << '\n' << cipher->name() << "single block encrypt: " << Botan::hex_
↪encode(block);

   // clear cipher for 2nd encryption with other key
   cipher->clear();
   key = Botan::hex_decode(
↪"13371337133713371337133713371337133713371337133713371337133713371337");
   cipher->set_key(key);
   cipher->encrypt(block);

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

8.6.2 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.

Algorithm specification names:

- `AES-128`
- `AES-192`
- `AES-256`

ARIA

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

Available if `BOTAN_HAS_ARIA` is defined.

Algorithm specification names:

- `ARIA-128`
- `ARIA-192`
- `ARIA-256`

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.

Algorithm specification name: `Blowfish`

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.

Algorithm specification names:

- `Camellia-128`
- `Camellia-192`
- `Camellia-256`

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.

Algorithm specification name: `Cascade(<BlockCipher 1>,<BlockCipher 2>)`, e.g. `Cascade(Serpent, AES-256)`

CAST-128

A 64-bit cipher, commonly used in OpenPGP.

Available if `BOTAN_HAS_CAST128` is defined.

Algorithm specification name:

- `CAST-128` (reported name) / `CAST5`

DES and 3DES

Originally designed by IBM and NSA in the 1970s. Today, DES's 56-bit key renders it insecure to any well-resourced attacker. 3DES extends the key length, and is 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.

Available if `BOTAN_HAS_DES` is defined.

Algorithm specification names:

- `DES`
- `TripleDES` (reported name) / `3DES` / `DES-EDE`

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.

Algorithm specification names:

- `GOST-28147-89` / `GOST-28147-89(R3411_94_TestParam)` (reported name)
- `GOST-28147-89(R3411_CryptoPro)`

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.

Algorithm specification name: `IDEA`

Kuznyechik

Added in version 3.2.

Newer Russian national cipher, also known as GOST R 34.12-2015 or "Grasshopper".

> **Warning:** The sbox of this cipher is supposedly random, but was found to have a mathematical structure which is exceedingly unlikely to have occured by chance. This may indicate the existence of a backdoor or other issue. Avoid using this cipher unless strictly required.

Available if `BOTAN_HAS_KUZNYECHIK` is defined.

Algorithm specification name: `Kuznyechik`

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.

Algorithm specification name: `Lion(<HashFunction>,<StreamCipher>,<optional block size>)`

- Block size defaults to 1024.
- Examples: `Lion(SHA-1,RC4,64)`

Noekeon

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

Available if `BOTAN_HAS_NOEKEON` is defined.

> **Warning:** Noekeon support is deprecated and will be removed in a future major release.

Algorithm specification name: `Noekeon`

SEED

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

Available if `BOTAN_HAS_SEED` is defined.

Algorithm specification name: `SEED`

Serpent

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

Available if `BOTAN_HAS_SERPENT` is defined.

Algorithm specification name: `Serpent`

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.

Algorithm specification name: `SHACAL2`

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.

Algorithm specification name: `SM4`

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.

Algorithm specification name: `Threefish-512`

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.

Algorithm specification name: `Twofish`

8.7 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.

std::unique_ptr<*StreamCipher*> **new_object**() const

Return a newly allocated object of the same type as this one. The new object is unkeyed.

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()).

8.7.1 Code Example

The following code encrypts a provided plaintext using ChaCha20.

```
#include <botan/auto_rng.h>
#include <botan/hex.h>
#include <botan/stream_cipher.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
↪");
   const auto cipher = Botan::StreamCipher::create_or_throw("ChaCha(20)");

   // generate fresh nonce (IV)
   Botan::AutoSeeded_RNG rng;
   const auto iv = rng.random_vec<std::vector<uint8_t>>(8);

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

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

8.7.2 Available Stream Ciphers

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

CTR-BE

Counter mode 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 a nonce, 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 the width (which must be at least 4 bytes, allowing to encrypt 2^{32} blocks of data) for example using "CTR(AES-256,8)" will select a 64-bit (8 byte) counter.

(The `-BE` suffix refers to big-endian convention for the counter. Little-endian counter mode is rarely used and not currently implemented.)

Algorithm specification name: `CTR-BE(<BlockCipher>,<optional counter size>)` (reported name) / `CTR(<BlockCipher>,<optional counter size>)`

- Counter size (in bytes) defaults to the block size of the underlying cipher
- If the counter size is the same as the underlying cipher, the name will be reported as `CTR-BE(<BlockCipher>)`.
- Examples: `CTR-BE(AES-128)`, `CTR-BE(AES-128,8)`

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.

Algorithm specification name: `OFB(<BlockCipher>)`, e.g. `OFB(AES-256)`

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.

Algorithm specification names:

- `ChaCha20`, alias for `ChaCha(20)`
- `ChaCha(<optional rounds>)`
  - Optional rounds defaults to 20
  - Examples: `ChaCha(20)`, `ChaCha(12)`

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.

Algorithm specification name: `Salsa20`

SHAKE-128

This is the SHAKE-128 XOF exposed as a stream cipher. It is slower than ChaCha and somewhat obscure, and was primarily implemented to support a particular post-quantum scheme which is no longer supported.

SHAKE does not support IVs, nor seeking within the cipher stream.

Available if `BOTAN_HAS_SHAKE_CIPHER` is defined.

> **Warning:** SHAKE support (as a stream cipher) is deprecated and will be removed in a future major release.

Algorithm specification names:

- `SHAKE-128` (reported name) / `SHAKE-128-XOF`
- `SHAKE-256` (reported name) / `SHAKE-256-XOF`

RC4

An old and very widely deployed stream cipher notable for its simplicity. It does not support IVs or seeking within the cipher stream. Compared to modern algorithms like ChaCha20, it is also quite slow.

> **Warning:** RC4 is prone to numerous attacks. **Avoid in new code** and use only if required for compatibility with existing systems.

Available if `BOTAN_HAS_RC4` is defined.

Algorithm specification names:

- `RC4` (reported name) / `ARC4`
- `MARK-4`
- `RC4(SKIP)` (reported name) / `ARC4(SKIP)`
  - `RC4(0)` is an alias for `RC4`
  - `RC4(256)` is an alias for `MARK-4`
  - Examples: `RC4(3)`

8.8 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.

std::unique_ptr<*MessageAuthenticationCode*> **new_object**() const

Return a newly allocated object of the same type as this one. The new object is unkeyed.

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.

8.8.1 Code Examples

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

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

#include <assert.h>

namespace {

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());
}

}  // namespace

int main() {
   Botan::AutoSeeded_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);

   return 0;
}
```

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

#include <iostream>

int main() {
   const std::vector<uint8_t> key =
      Botan::hex_decode("13371337133713371337133713371337133713371337133713371337133713371337
↪");
   const std::vector<uint8_t> nonce = Botan::hex_decode("FFFFFFFFFFFFFFFFFFFFFFFF");
   const std::vector<uint8_t> data = Botan::hex_decode("6BC1BEE22E409F96E93D7E117393172A
↪");
   const auto mac = Botan::MessageAuthenticationCode::create_or_throw("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) << '\n';

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

#include <iostream>

int main() {
   const std::vector<uint8_t> key = Botan::hex_decode("2B7E151628AED2A6ABF7158809CF4F3C
↪");
   std::vector<uint8_t> data = Botan::hex_decode("6BC1BEE22E409F96E93D7E117393172A");
   const auto mac = Botan::MessageAuthenticationCode::create_or_throw("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
```

(continues on next page)

(continued from previous page)

```
   mac->update(data);
   std::cout << "Verification with malformed data: " << (mac->verify_mac(tag) ? "success
↪" : "failure");
   return 0;
}
```

8.8.2 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.

Blake2B MAC

Available if `BOTAN_HAS_BLAKE2BMAC` is defined.

Algorithm specification name: `BLAKE2b(<optional output bits>)` (reported name) / `Blake2b(<optional output bits>)`

- Output bits defaults to 512.
- Examples: `BLAKE2b(256)`, `BLAKE2b`

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.

Algorithm specification name: `CMAC(<BlockCipher>)` (reported name) / `OMAC(<BlockCipher>)`, e.g. `CMAC(AES-256)`

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.

Algorithm specification name: `GMAC(<BlockCipher>)`, e.g. `GMAC(AES-256)`

HMAC

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

Available if `BOTAN_HAS_HMAC` is defined.

Algorithm specification name: `HMAC(<HashFunction>)`, e.g. `HMAC(SHA-512)`

KMAC

Added in version 3.2.

A SHA-3 derived message authentication code defined by NIST in SP 800-185.

There are two variants, `KMAC-128` and `KMAC-256`. Both take a parameter which specifies the output length in bits, for example `KMAC-128(256)`.

Available if `BOTAN_HAS_KMAC` is defined.

Algorithm specification names:

- `KMAC-128(<output size>)`, e.g. `KMAC-128(256)`
- `KMAC-256(<output size>)`, e.g. `KMAC-256(256)`

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.

Algorithm specification name: `Poly1305`

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.

Algorithm specification name: `SipHash(<optional C>,<optional D>)`

- C defaults to 2
- D defaults to 4
- Examples: `SipHash(2,4)`, `SipHash(2)`, `SipHash`

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.

Algorithm specification name: `X9.19-MAC`

8.9 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. Will return 1 if the mode implementation does not require buffering.

virtual size_t **ideal_granularity**() const

Returns a multiple of update_granularity sized for ideal performance.

In fact this is not truly the "ideal" buffer size but just reflects the smallest possible buffer that can reasonably take advantage of available parallelism (due to SIMD execution, etc). If you are concerned about performance, it may be advisable to take this return value and scale it to approximately 4 KB, and use buffers of that size.

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*. This is used for example when processing an AEAD message, to ensure the tag is available. In that case, the encryption side will return 0 (since the tag is generated, rather than being provided) while the decryption mode will return the size of the tag.

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.

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

#include <iostream>

int main() {
   Botan::AutoSeeded_RNG rng;

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

   const auto enc = Botan::Cipher_Mode::create_or_throw("AES-128/CBC/PKCS7",␣
↪Botan::Cipher_Dir::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;
}
```

8.9.2 Available Unauthenticated Cipher Modes

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

ESP (RFC 4303)
The first padding byte is set to 0x01, the next ones to 0x02, 0x03, ... (monotonically increasing sequence).

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.

Warning: Using CBC with padding without an authentication mode exposes your application to CBC padding oracle attacks, which allow recovering the plaintext of arbitrary messages. Always pair CBC with a MAC such as HMAC (or, preferably, use an AEAD such as GCM).

Algorithm specification name: `<BlockCipher>/CBC/<optional padding scheme>` (reported name) / `CBC(<BlockCipher>,<optional padding scheme>)`

- Available padding schemes:
  - `NoPadding`
  - `PKCS7` (default)
  - `OneAndZeros`
  - `X9.23`
  - `ESP`
  - `CTS`
- Examples: `AES-128/CBC/PKCS7`, `AES-256/CBC`

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.

Algorithm specification name: `<BlockCipher>/CFB(<optional feedback bits>)` (reported name) / `CFB(<BlockCipher>,<optional feedback bits>)`

- Feedback bits defaults to the size of the underlying block cipher.
- Examples: `AES-192/CFB`, `AES-128/CFB(8)`

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.

Algorithm specification name: `<BlockCipher>/XTS` (reported name) / `XTS(<BlockCipher>)`, e.g. `AES-256/XTS`

8.9.3 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.

8.9.4 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.

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.

Algorithm specification name: `<BlockCipher>/CCM(<optional tag size>,<optional L>)` (reported name) / `CCM(<BlockCipher>,<optional tag size>,<optional L>)`

- Tag size defaults to 16.
- L defaults to 3.
- Examples: `AES-128/CCM`, `AES-128/CCM(8)`, `AES-128/CCM(8,2)`

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.

Algorithm specification name: ChaCha20Poly1305

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.

Algorithm specification name: <BlockCipher>/EAX(<optional tag size>) / EAX(<BlockCipher>, <optional tag size>)

- Tag size defaults to 16.
- Reports name as <BlockCipher>/EAX, i.e. without the tag size.
- Examples: e.g. AES-128/EAX, AES-128/EAX(8)

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.

Algorithm specification name: <BlockCipher>/GCM(<optional tag size>) (reported name) / GCM(<BlockCipher>,<optional tag size>)

- Tag size defaults to 16.
- Examples: e.g. AES-128/GCM, AES-128/GCM(12)

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 until 2021 it was patented in the United States. The patent was allowed to lapse in early 2021.

Algorithm specification name: <BlockCipher>/OCB(<optional tag size>) / OCB(<BlockCipher>, <optional tag size>)

- Tag size defaults to 16.
- Reports name as <BlockCipher>/OCB, i.e. without the tag size.
- Examples: e.g. AES-128/OCB, AES-128/OCB(12)

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).

Algorithm specification name: <BlockCipher>/SIV (reported name) / SIV(<BlockCipher>), e.g. AES-128/SIV

8.10 Public Key Cryptography

Public key cryptography is a collection of techniques allowing for encryption, signatures, and key agreement.

8.10.1 Key Objects

Public and private keys are represented by classes Public_Key and Private_Key. Both derive from Asymmetric_Key.

Currently there is an inheritance relationship between Private_Key and Public_Key, so that a private key can also be used as the corresponding public key. It is best to avoid relying on this, as this inheritance will be removed in a future major release.

class **Asymmetric_Key**

std::string **algo_name**()

Return a short string identifying the algorithm of this key, eg "RSA" or "Dilithium".

size_t **estimated_strength**() const

Return an estimate of the strength of this key, in terms of brute force key search. For example if this function returns 128, then it is is estimated to be roughly as difficult to crack as AES-128.

OID **object_identifier**() const

Return an object identifier which can be used to identify this type of key.

bool **supports_operation**(PublicKeyOperation op) const

Check if this key could be used for the queried operation type.

class **Public_Key**

size_t **key_length**() const = 0;

Return an integer value that most accurately captures for the security level of the key. For example for RSA this returns the length of the public modules, while for ECDSA keys it returns the size of the elliptic curve group.

bool **check_key**(*RandomNumberGenerator* &rng, bool strong) const = 0;

Check if the key seems to be valid. If *strong* is set to true then more expensive tests are performed.

AlgorithmIdentifier **algorithm_identifier**() const = 0;

Return an X.509 algorithm identifier that can be used to identify the key.

std::vector<uint8_t> **public_key_bits**() const = 0;

Returns a binary representation of the public key. Typically this is a BER encoded structure that includes metadata like the algorithm and parameter set used to generate the key.

Note that pre-standard post-quantum algorithms of the NIST competition (e.g. Kyber, Dilithium, FrodoKEM, etc) do not have a standardized BER encoding, yet. For the time being, the raw public key bits are returned for these algorithms. That might change as the standards evolve.

std::vector<uint8_t> **raw_public_key_bits**() const = 0;

Returns a binary representation of the public key's canonical structure. Typically, this does not include any metadata like an algorithm identifier or parameter set. Note that some schemes (e.g. RSA) do not know such "raw" canonical structure and therefore throw *Not_Implemented*. For key agreement algorithms, this is the canonical public value of the scheme.

Decoding the resulting raw bytes typically requires knowledge of the algorithm and parameters used to generate the key.

std::vector<uint8_t> **subject_public_key**() const;

Return the X.509 SubjectPublicKeyInfo encoding of this key

std::string **fingerprint_public**(const std::string &alg = "SHA-256") const;

Return a hashed fingerprint of this public key.

8.10.2 Public Key Algorithms

Botan includes a number of public key algorithms, some of which are in common use, others only used in specialized or niche applications.

RSA

Based on the difficulty of factoring. Usable for encryption, signatures, and key encapsulation.

ECDSA

Fast signature scheme based on elliptic curves.

ECDH, DH, X25519 and X448

Key agreement schemes. DH uses arithmetic over finite fields and is slower and with larger keys. ECDH, X25519 and X448 use elliptic curves instead.

Dilithium

Post-quantum secure signature scheme based on lattice problems.

Kyber

Post-quantum key encapsulation scheme based on (structured) lattices.

Note: Currently two modes for Kyber are defined: the round3 specification from the NIST PQC competition, and the "90s mode" (which uses AES/SHA-2 instead of SHA-3 based primitives). The 90s mode Kyber is deprecated and will be removed in a future release.

The final NIST specification version of Kyber is not yet implemented.

Ed25519 and Ed448

Signature schemes based on a specific elliptic curve.

XMSS

A post-quantum secure signature scheme whose security is based (only) on the security of a hash function. Unfortunately XMSS is stateful, meaning the private key changes with each signature, and only a certain pre-specified number of signatures can be created. If the same state is ever used to generate two signatures, then the whole scheme becomes insecure, and signatures can be forged.

8.10.3 HSS-LMS

A post-quantum secure hash-based signature scheme similar to XMSS. Contains support for multitrees. It is stateful, meaning the private key changes after each signature.

SPHINCS+

A post-quantum secure signature scheme whose security is based (only) on the security of a hash function. Unlike XMSS, it is a stateless signature scheme, meaning that the private key does not change with each signature. It has high security but very long signatures and high runtime.

FrodoKEM

A post-quantum secure key encapsulation scheme based on (unstructured) lattices.

McEliece

Post-quantum secure key encapsulation scheme based on the hardness of certain decoding problems.

ElGamal

Encryption scheme based on the discrete logarithm problem. Generally unused except in PGP.

DSA

Finite field based signature scheme. A NIST standard but now quite obsolete.

ECGDSA, ECKCDSA, SM2, GOST-34.10

A set of signature schemes based on elliptic curves. All are national standards in their respective countries (Germany, South Korea, China, and Russia, resp), and are completely obscure and unused outside of that context.

8.10.4 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)

8.10.5 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)` or `PBES2(CIPHER,PBKDF)`.
>
> Cipher can be any block cipher using CBC or GCM modes, 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 2.0. 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:

std::unique_ptr<Private_Key> **load_key**(DataSource &source, std::function<std::string()> get_passphrase)

std::unique_ptr<Private_Key> **load_key**(DataSource &source, const std::string &pass)

std::unique_ptr<Private_Key> **load_key**(DataSource &source)

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 DataSource is usually a DataSource_Stream to read from a file or DataSource_Memory for an in-memory buffer.

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.

8.10.6 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)

std::unique_ptr<*Public_Key*> X509::**load_key**(DataSource &in)

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

std::unique_ptr<*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.

Note: In versions prior to 3.0, these functions returned a raw pointer instead of a `unique_ptr`.

8.10.7 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: DL_Group

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

#include <iostream>

int main() {
   Botan::AutoSeeded_RNG rng;
   auto group = std::make_unique<Botan::DL_Group>(rng, Botan::DL_Group::Strong, 2048);

   std::cout << "P = " << group->get_p().to_hex_string() << "\n"
             << "Q = " << group->get_q().to_hex_string() << "\n"
             << "G = " << group->get_g().to_hex_string() << "\n";

   std::cout << "\nPEM:\n" << group->PEM_encode(Botan::DL_Group_Format::ANSI_X9_42) << "\
↪n";

   return 0;
}
```

8.10.8 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".

8.10.9 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/auto_rng.h>
#include <botan/pk_keys.h>
#include <botan/rng.h>
```

(continues on next page)

(continued from previous page)

```
#include <botan/x509cert.h>
#include <iostream>

int main() {
   Botan::X509_Certificate cert("cert.pem");
   Botan::AutoSeeded_RNG rng;
   auto key = cert.subject_public_key();
   if(!key->check_key(rng, false)) {
      std::cerr << "Loaded key is invalid";
      return 1;
   }

   return 0;
}
```

8.10.10 Public Key Encryption/Decryption

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**

std::vector<uint8_t> **encrypt**(const uint8_t in[], size_t length, *RandomNumberGenerator* &rng) const

std::vector<uint8_t> **encrypt**(std::span<const 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.

size_t **ciphertext_length**(size_t ctext_len) const

Return an upper bound on the returned size of a ciphertext, if this particular key/padding scheme is used to encrypt a message of the provided length.

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 Diffie-Hellman 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 padding)

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

If you are not sure what padding to use, use "OAEP(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, std::unique_ptr<*KDF*> kdf, std::unique_ptr<*MessageAuthenticationCode*> mac, size_t mac_key_len = 20)

Where *kdf* is a key derivation function (see *Key Derivation Functions (KDF)*) 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, std::unique_ptr<*KDF*> kdf, std::unique_ptr<*Cipher_Mode*> cipher, size_t cipher_key_len, std::unique_ptr<*MessageAuthenticationCode*> mac, size_t mac_key_len = 20)

Instead of XORing the message with KDF output, a cipher mode can be used

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 (KDF)*), 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:

1. RSA. Requires a *padding scheme* as parameter.
2. DLIES
3. ECIES
4. SM2. Takes an optional `HashFunction` as parameter which defaults to SM3.
5. ElGamal. Requires a *padding scheme* as parameter.

Code Example: RSA Encryption

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 OAEP with SHA-256. For the sake of completeness, the ciphertext is then decrypted using the private key.

```
#include <botan/auto_rng.h>
#include <botan/hex.h>
#include <botan/pk_keys.h>
#include <botan/pkcs8.h>
#include <botan/pubkey.h>
#include <botan/rng.h>

#include <iostream>
```

(continues on next page)

(continued from previous page)

```
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());
   Botan::AutoSeeded_RNG rng;

   // load keypair
   Botan::DataSource_Stream in(argv[1]);
   auto kp = Botan::PKCS8::load_key(in);

   // encrypt with pk
   Botan::PK_Encryptor_EME enc(*kp, rng, "OAEP(SHA-256)");
   std::vector<uint8_t> ct = enc.encrypt(pt, rng);

   // decrypt with sk
   Botan::PK_Decryptor_EME dec(*kp, rng, "OAEP(SHA-256)");
   Botan::secure_vector<uint8_t> pt2 = dec.decrypt(ct);

   std::cout << "\nenc: " << Botan::hex_encode(ct) << "\ndec: " << Botan::hex_
↪encode(pt2);

   return 0;
}
```

Available encryption padding schemes

Note: Padding schemes in the context of encryption are sometimes also called *Encoding Method for Encryption* (EME).

OAEP

OAEP (called EME1 in IEEE 1363 and in earlier versions of the library) as specified in PKCS#1 v2.0 (RFC 2437) or PKCS#1 v2.1 (RFC 3447).

- Names: `OAEP` / `EME-OAEP` / `EME1`
- Parameters specification:
  - `(<HashFunction>)`
  - `(<HashFunction>,MGF1)`
  - `(<HashFunction>,MGF1(<HashFunction>))`
  - `(<HashFunction>,MGF1(<HashFunction>),<optional label>)`
- The only Mask generation function available is MGF1 which is also the default.
- By default the same hash function will be used for the label and MGF1.

- Examples: `OAEP(SHA-256)`, `EME-OAEP(SHA-256,MGF1)`, `OAEP(SHA-256,MGF1(SHA-512))`, `OAEP(SHA-512,MGF1(SHA-512),TCPA)`

PKCS #1 v1.5 Type 2 (encryption)

PKCS #1 v1.5 Type 2 (encryption) padding.

Names: `PKCS1v15` / `EME-PKCS1-v1_5`

Raw EME

Does not change the input during padding. Don't use this unless you know what you are doing. Un-padding will strip leading zeros.

Name: `Raw`

8.10.11 Public Key Signature Schemes

Signature generation is performed using

class **PK_Signer**

PK_Signer(const Private_Key &key, const std::string &padding, Signature_Format format = Siganture_Format::Standard)

Constructs a new signer object for the private key *key* using the hash/padding specified in *padding*. The key must support signature operations. In the current version of the library, this includes e.g. RSA, ECDSA, Dilithium, ECKCDSA, ECGDSA, GOST 34.10-2001, and SM2.

Note: Botan both supports non-deterministic and deterministic (as per RFC 6979) DSA and ECDSA signatures. Either type of signature can be verified by any other (EC)DSA library, regardless of which mode it prefers. If the `rfc6979` module is enabled at build time, deterministic DSA and ECDSA signatures will be created.

The proper value of *padding* depends on the algorithm. For many signature schemes including ECDSA and DSA, simply naming a hash function like "SHA-256" is all that is required.

For RSA, more complicated padding is required. The two most common schemes for RSA signature padding are PSS and PKCS1v1.5, so you must specify both the padding mechanism as well as a hash, for example "PSS(SHA-256)" or "PKCS1v15(SHA-256)".

Certain newer signature schemes, especially post-quantum based ones, hardcode the hash function associated with their signatures, and no configuration is possible. There *padding* should be left blank, or may possibly be used to identify some algorithm-specific option. For instance Dilithium may be parameterized with "Randomized" or "Deterministic" to choose if the generated signature is randomized or not. If left blank, a default is chosen.

Another available option, usable in certain specialized scenarios, is using padding scheme "Raw", where the provided input is treated as if it was already hashed, and directly signed with no other processing.

The *format* defaults to `Standard` which is either the usual, or the only, available formatting method, depending on the algorithm. For certain signature schemes including ECDSA, DSA, ECGDSA and ECKCDSA you can also use `DerSequence`, which will format the signature as an ASN.1 SEQUENCE value. This formatting is used in protocols such as TLS and Bitcoin.

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

void **update**(std::span<const 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.

std::vector<uint8_t> **signature**(*RandomNumberGenerator* &rng)

Creates the signature and returns it. The rng may or may not be used, depending on the scheme.

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

std::vector<uint8_t> **sign_message**(std::span<const 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 also be included in the signature.

size_t **signature_length**() const

Return an upper bound on the length of the signatures returned by this object.

AlgorithmIdentifier **algorithm_identifier**() const

Return an algorithm identifier appropriate to identify signatures generated by this object in an X.509 structure.

std::string **hash_function**() const

Return the hash function which is being used

Signatures are verified using

class **PK_Verifier**

PK_Verifier(const *Public_Key* &pub_key, const std::string &padding, Signature_Format format = Signature_Format::Standard)

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

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

void **update**(std::span<const 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**(std::span<const 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**(std::span<const uint8_t> msg, std::span<const uint8_t> sig)

These are equivalent to calling *PK_Verifier::update* on *msg* and then calling *PK_Verifier::check_signature* on *sig*. Any data previously provided to *PK_Verifier::update* will also be included.

Botan implements the following signature algorithms:

1. RSA. Requires a *padding scheme* as parameter.

2. DSA. Requires a *hash function* as parameter.
3. ECDSA. Requires a *hash function* as parameter.
4. ECGDSA. Requires a *hash function* as parameter.
5. ECKDSA. Requires a *hash function* as parameter, not supporting `Raw`.
6. GOST 34.10-2001. Requires a *hash function* as parameter.
7. Ed25519 and Ed448. See *Ed25519 and Ed448 Variants* for parameters.
8. SM2. Takes one of the following as parameter:
   - `<user ID>` (uses `SM3`)
   - `<user ID>,<HashFunction>`
9. Dilithium. Takes the optional parameter `Deterministic` (default) or `Randomized`.
10. SPHINCS+. Takes the optional parameter `Deterministic` (default) or `Randomized`.
11. XMSS. Takes no parameter.
12. HSS-LMS. Takes no parameter.

Code Example: ECDSA Signature

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

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

#include <iostream>

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

   const std::string message("This is a tasty burger!");

   // sign data
   Botan::PK_Signer signer(key, rng, "SHA-256");
   signer.update(message);
   std::vector<uint8_t> signature = signer.signature(rng);
   std::cout << "Signature:\n" << Botan::hex_encode(signature);

   // now verify the signature
   Botan::PK_Verifier verifier(key, "SHA-256");
   verifier.update(message);
   std::cout << "\nis " << (verifier.check_signature(signature) ? "valid" : "invalid");
   return 0;
}
```

Available signature padding schemes

Note: Padding schemes in the context of signatures are sometimes also called *Encoding methods for signatures with appendix* (EMSA).

PKCS #1 v1.5 Type 1 (signature)

PKCS #1 v1.5 Type 1 (signature) padding or EMSA3 from IEEE 1363.

- Names: `PKCS1v15` / `EMSA_PKCS1` / `EMSA-PKCS1-v1_5` / `EMSA3`
- Parameters specification:
  - `(<HashFunction>)`
  - `(Raw,<optional HashFunction>)`
- The raw variant encodes a precomputed hash, optionally with the digest ID of the given hash.
- Examples: `PKCS1v15(SHA-256)`, `PKCS1v15(Raw)`, `PKCS1v15(Raw,MD5)`,

EMSA-PSS

Probabilistic signature scheme (PSS) (called EMSA4 in IEEE 1363).

- Names: `PSS` / `EMSA-PSS` / `PSSR` / `PSS-MGF1` / `EMSA4`
- Parameters specification:
  - `(<HashFunction>)`
  - `(<HashFunction>,MGF1,<optional salt size>)`
- Examples: `PSS(SHA-256)`, `PSS(SHA-256,MGF1,32)`,

There also exists a raw version, which accepts a pre-hashed buffer instead of the message. Don't use this unless you know what you are doing.

- Names: `PSS_Raw` / `PSSR_Raw`
- Parameters specification:
  - `(<HashFunction>)`
  - `(<HashFunction>,MGF1,<optional salt size>)`

ISO-9796-2

ISO-9796-2 - Digital signature scheme 2 (probabilistic).

- Name: `ISO_9796_DS2`
- Parameters specification:
  - `(<HashFunction>)`
  - `(<HashFunction>,<exp|imp>,<optional salt size>)`
- Defaults to the explicit mode.

- Examples: `ISO_9796_DS2(RIPEMD-160)`, `ISO_9796_DS2(RIPEMD-160,imp)`

ISO-9796-2 - Digital signature scheme 3 (deterministic), i.e. DS2 without a salt.

- Name: `ISO_9796_DS3`
- Parameters specification:
  - `(<HashFunction>)`
  - `(<HashFunction>,<exp|imp>`
- Defaults to the explicit mode.
- Examples: `ISO_9796_DS3(RIPEMD-160)`, `ISO_9796_DS3(RIPEMD-160,imp)`,

X9.31

EMSA from X9.31 (EMSA2 in IEEE 1363).

- Names: `EMSA2` / `EMSA_X931` / `X9.31`
- Parameters specification: `(<HashFunction>)`
- Example: `EMSA2(SHA-256)`

Raw EMSA

Sign inputs directly. Don't use this unless you know what you are doing.

- Name: `Raw`
- Parameters specification: `(<optional HashFunction>)`
- Examples: `Raw`, `Raw(SHA-256)`

Signature with Hash

For many signature schemes including ECDSA and DSA, simply naming a hash function like `SHA-256` is all that is required.

Previous versions of Botan required using a hash specifier like `EMSA1(SHA-256)` when generating or verifying ECDSA/DSA signatures, with the specified hash. The `EMSA1` was a reference to a now obsolete IEEE standard.

Parameters specification:

- `<HashFunction>`
- `EMSA1(<HashFunction>)`

There also exists a raw mode, which accepts a pre-hashed buffer instead of the message. Don't use this unless you know what you are doing.

Parameters specification:

- `Raw`
- `Raw(<HashFunction>)`

Ed25519 and Ed448 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 and Ed448 instead sign the message directly. This is beneficial, in that the 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/Ed448, use a padding name of "Pure".

This is the default mode if no padding name is given.

Parameter specification: `Pure` / `Identity`

Ed25519ph (or Ed448) (pre-hashed) instead hashes the message with SHA-512 (or SHAKE256(512)) and then signs the digest plus a special prefix specified in RFC 8032. To use it, specify padding name "Ed25519ph" (or "Ed448ph").

Parameter specification: `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.

Parameter specification: <`HashFunction`>

For best interop with other systems, prefer "Ed25519ph".

8.10.12 Key Agreement

Key agreement is a scheme where two parties exchange public keys, after which it is possible for them to derive a secret key which is known only to the two of them.

There are different approaches possible for key agreement. In many protocols, both parties generate a new key, exchange public keys, and derive a secret, after which they throw away their private keys, using them only the once. However this requires the parties to both be online and able to communicate with each other.

In other protocols, one of the parties publishes their public key online in some way, and then it is possible for someone to send encrypted messages to that recipient by generating a new keypair, performing key exchange with the published public key, and then sending both the message along with their ephemeral public key. Then the recipient uses the provided public key along with their private key to complete the key exchange, recover the shared secret, and decrypt the message.

Typically the raw output of the key agreement function is not uniformly distributed, and may not be of an appropriate length to use as a key. To resolve these problems, key agreement will use a *Key Derivation Functions (KDF)* on the shared secret to produce an output of the desired length.

1. ECDH over GF(p) Weierstrass curves
2. ECDH over x25519
3. DH over prime fields

class **PK_Key_Agreement**

PK_Key_Agreement(const Private_Key &key, *RandomNumberGenerator* &rng, const std::string &kdf, const std::string &provider = "")

Set up to perform key derivation using the given private key and specified KDF.

SymmetricKey **derive_key**(size_t key_len, const uint8_t in[], size_t in_len, const uint8_t params[], size_t params_len) const

SymmetricKey **derive_key**(size_t key_len, std::span<const uint8_t> in, const uint8_t params[], size_t params_len) const

SymmetricKey **derive_key**(size_t key_len, const uint8_t in[], size_t in_len, const std::string &params = "") const

SymmetricKey **derive_key**(size_t key_len, const std::span<const uint8_t> in, const std::string &params = "") const

Return a shared key. The *params* will be hashed along with the shared secret by the KDF; this can be useful to bind the shared secret to a specific usage.

The *in* parameter must be the public key associated with the other party.

Code Example: ECDH Key Agreement

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

#include <iostream>

int main() {
   Botan::AutoSeeded_RNG rng;

   // ec domain and KDF
   const auto domain = Botan::EC_Group::from_name("secp521r1");
   const std::string kdf = "KDF2(SHA-256)";

   // the two parties generate ECDH keys
   Botan::ECDH_PrivateKey key_a(rng, domain);
   Botan::ECDH_PrivateKey key_b(rng, domain);

   // now they exchange their public values
   const auto key_apub = key_a.public_value();
   const auto key_bpub = key_b.public_value();

   // Construct key agreements and agree on a shared secret
   Botan::PK_Key_Agreement ka_a(key_a, rng, kdf);
   const auto sA = ka_a.derive_key(32, key_bpub).bits_of();

   Botan::PK_Key_Agreement ka_b(key_b, rng, kdf);
   const auto sB = ka_b.derive_key(32, key_apub).bits_of();

   if(sA != sB) {
      return 1;
   }

   std::cout << "agreed key:\n" << Botan::hex_encode(sA);
```

(continues on next page)

(continued from previous page)

```
   return 0;
}
```

8.10.13 Key Encapsulation

Key encapsulation (KEM) is a variation on public key encryption which is commonly used by post-quantum secure schemes. Instead of choosing a random secret and encrypting it, as in typical public key encryption, a KEM encryption takes no inputs and produces two values, the shared secret and the encapsulated key. The decryption operation takes in the encapsulated key and returns the shared secret.

class **PK_KEM_Encryptor**

PK_KEM_Encryptor(const *Public_Key* &key, const std::string &kdf = "", const std::string &provider = "")

Create a KEM encryptor

size_t **shared_key_length**(size_t desired_shared_key_len) const

Size in bytes of the shared key being produced by this PK_KEM_Encryptor.

size_t **encapsulated_key_length**() const

Size in bytes of the encapsulated key being produced by this PK_KEM_Encryptor.

KEM_Encapsulation **encrypt**(*RandomNumberGenerator* &rng, size_t desired_shared_key_len = 32, std::span<const uint8_t> salt = {})

Perform a key encapsulation operation with the result being returned as a convenient struct.

void **encrypt**(std::span<uint8_t> out_encapsulated_key, std::span<uint8_t> out_shared_key, *RandomNumberGenerator* &rng, size_t desired_shared_key_len = 32, std::span<const uint8_t> salt = {})

Perform a key encapsulation operation by passing in out-buffers of the correct output length. Use encapsulated_key_length() and shared_key_length() to pre-allocate the output buffers.

void **encrypt**(secure_vector<uint8_t> &out_encapsulated_key, secure_vector<uint8_t> &out_shared_key, size_t desired_shared_key_len, *RandomNumberGenerator* &rng, std::span<const uint8_t> salt)

Perform a key encapsulation operation by passing in out-vectors that will be re-allocated to the correct output size.

class **KEM_Encapsulation**

std::vector<uint8_t> **encapsulated_shared_key**() const

secure_vector<uint8_t> **shared_key**() const

class **PK_KEM_Decryptor**

PK_KEM_Decryptor(const *Public_Key* &key, const std::string &kdf = "", const std::string &provider = "")

Create a KEM decryptor

size_t **encapsulated_key_length**() const

Size in bytes of the encapsulated key expected by this PK_KEM_Decryptor.

size_t **shared_key_length**(size_t desired_shared_key_len) const

Size in bytes of the shared key being produced by this PK_KEM_Encryptor.

secure_vector<uint8> **decrypt**(std::span<const uint8> encapsulated_key, size_t desired_shared_key_len, std::span<const uint8_t> salt)

Perform a key decapsulation operation

void **decrypt**(std::span<uint8_t> out_shared_key, std::span<const uint8_t> encap_key, size_t desired_shared_key_len = 32, std::span<const uint8_t> salt = {})

Perform a key decapsulation operation by passing in a pre-allocated out-buffer. Use shared_key_length() to determine the byte-length required.

Botan implements the following KEM schemes:

1. RSA
2. Kyber
3. FrodoKEM
4. McEliece

Code Example: Kyber

The code below demonstrates key encapsulation using the Kyber post-quantum scheme.

```
#include <botan/kyber.h>
#include <botan/pubkey.h>
#include <botan/system_rng.h>
#include <array>
#include <iostream>

int main() {
   const size_t shared_key_len = 32;
   const std::string kdf = "HKDF(SHA-512)";

   Botan::System_RNG rng;

   std::array<uint8_t, 16> salt;
   rng.randomize(salt);

   Botan::Kyber_PrivateKey priv_key(rng, Botan::KyberMode::Kyber512_R3);
   auto pub_key = priv_key.public_key();

   Botan::PK_KEM_Encryptor enc(*pub_key, kdf);

   const auto kem_result = enc.encrypt(rng, shared_key_len, salt);

   Botan::PK_KEM_Decryptor dec(priv_key, rng, kdf);

   auto dec_shared_key = dec.decrypt(kem_result.encapsulated_shared_key(), shared_key_
↪len, salt);

   if(dec_shared_key != kem_result.shared_key()) {
      std::cerr << "Shared keys differ\n";
      return 1;
   }
```

(continues on next page)

(continued from previous page)

```
   return 0;
}
```

8.10.14 McEliece cryptosystem

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.

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`

8.10.15 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).

Warning: XMSS is stateful, meaning the private key updates after each signature creation. Applications are responsible for updating their persistent secret with the new output of `Private_Key::private_key_bits()` after each signature creation. If the same private key is ever used to generate two different signatures, then the scheme becomes insecure. For this reason, it can be challenging to use XMSS securely.

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

1. XMSS-SHA2_10_256
2. XMSS-SHA2_16_256
3. XMSS-SHA2_20_256
4. XMSS-SHA2_10_512
5. XMSS-SHA2_16_512
6. XMSS-SHA2_20_512
7. XMSS-SHAKE_10_256
8. XMSS-SHAKE_16_256
9. XMSS-SHAKE_20_256
10. XMSS-SHAKE_10_512
11. XMSS-SHAKE_16_512
12. 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: XMSS

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

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

#include <iostream>
#include <vector>

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);
   const Botan::XMSS_PublicKey& public_key(private_key);

   // create Public Key Signer using the private key.
   Botan::PK_Signer signer(private_key, rng, "");

   // create and sign a message using the Public Key Signer.
   Botan::secure_vector<uint8_t> msg{0x01, 0x02, 0x03, 0x04};
   signer.update(msg.data(), msg.size());
```

(continues on next page)

(continued from previous page)

```
   std::vector<uint8_t> sig = signer.signature(rng);

   // create Public Key Verifier using the public key
   Botan::PK_Verifier verifier(public_key, "");

   // verify the signature for the previously generated message.
   verifier.update(msg.data(), msg.size());
   if(verifier.check_signature(sig.data(), sig.size())) {
      std::cout << "Success.\n";
      return 0;
   } else {
      std::cout << "Error.\n";
      return 1;
   }
}
```

8.10.16 Hierarchical Signature System with Leighton-Micali Hash-Based Signatures (HSS-LMS)

HSS-LMS is a stateful hash-based signature scheme which is defined in RFC 8554 "Leighton-Micali Hash-Based Signatures" (https://datatracker.ietf.org/doc/html/rfc8554).

It is a multitree scheme, which is highly configurable. Multitree means, it consists of multiple layers of Merkle trees, which can be defined individually. Moreover, the used hash function and the Winternitz Parameter of the underlying one-time signature can be chosen for each tree layer. For a sensible selection of parameters refer to RFC 8554 Section 6.4. (https://datatracker.ietf.org/doc/html/rfc8554#section-6.4).

> **Warning:** HSS-LMS is stateful, meaning the private key updates after each signature creation. Applications are responsible for updating their persistent secret with the new output of `Private_Key::private_key_bits()` after each signature creation. If the same private key is ever used to generate two different signatures, then the scheme becomes insecure. For this reason, it can be challenging to use HSS-LMS securely.

HSS-LMS uses the Botan interfaces for public key cryptography. The `params` argument of the HSS-LMS private key is used to define the parameter set. The syntax of this argument must be the following:

`HSS-LMS(<hash>,HW(<h>,<w>),HW(<h>,<w>),...)`

e.g. `HSS-LMS(SHA-256,HW(5,1),HW(5,1))` to use SHA-256 in a two-layer HSS instance with LMS tree height 5 and Winternitz parameter 1. This results in a private key that can be used to create up to 2^(5+5)=1024 signatures.

The following parameters are allowed (which are specified in RFC 8554 (https://datatracker.ietf.org/doc/html/rfc8554) and and draft-fluhrer-lms-more-parm-sets-11 (https://datatracker.ietf.org/doc/html/draft-fluhrer-lms-more-parm-sets-11)):

- hash: `SHA-256`, `Truncated(SHA-256,192)`, `SHAKE-256(256)`, `SHAKE-256(192)`
- h: `5`, `10`, `15`, `20`, `25`
- w: `1`, `2`, `4`, `8`

8.11 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 a basic list of constraints which govern usage of the key embedded in this certificate.

The Key_Constraints is a class that behaves somewhat like an `enum`. The easiest way to use it is with its `includes` method. For example:

```
constraints().includes(Key_Constraints::DigitalSignature)
```

checks if the certificate key is valid for generating digital signatures.

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

8.11.1 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.

8.11.2 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.

8.11.3 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::optional` 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::optional` of `X509_CRL`. The `std::optional` return type makes it easy to return no CRLs by returning `nullopt` (eg, if the certificate store doesn't support retrieving CRLs). Implementing the function is optional, and by default will return `nullopt`.

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

Note: In the 2.x library, the certificate store interface relied on `shared_ptr<X509_Certificate>` to avoid copies. However since 2.4.0, the `X509_Certificate` was internally shared, and thus the outer `shared_ptr` was just a cause of needless runtime overhead and API complexity. Starting in version 3.0, the certificate store interface is defined in terms of plain `X509_Certificate`.

8.11.4 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_crl**(const X509_CRL &crl)

Add a certificate revocation list (CRL) to the store.

8.11.5 System Certificate Stores

An interface to use the system provided certificate stores is available for Unix, macOS and Windows systems, `System_Certificate_Store`

8.11.6 Flatfile Certificate Stores

`Flatfile_Certificate_Store` is an implementation of certificate store that reads certificates as files from a directory. This is also used as the implementation of the Unix/Linux system certificate store.

The constructor takes a path to the directory to read, along with an optional boolean indicating if non-CA certificates should be ignored.

8.11.7 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<*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::optional<OCSP::*Response*>> &ocsp_resp = std::vector<std::optional<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.

Code Example

For sheer demonstrative purposes, the following code verifies an end entity certificate against a trusted Root CA certificate.

```
#include <botan/certstor_system.h>
#include <botan/x509cert.h>
#include <botan/x509path.h>

int main() {
   // Create a certificate store and add a locally trusted CA certificate
   Botan::Certificate_Store_In_Memory customStore;
   customStore.add_certificate(Botan::X509_Certificate("root.crt"));

   // Additionally trust all system-specific CA certificates
   Botan::System_Certificate_Store systemStore;
   std::vector<Botan::Certificate_Store*> trusted_roots{&customStore, &systemStore};

   // Load the end entity certificate and two untrusted intermediate CAs from file
   std::vector<Botan::X509_Certificate> end_certs;
```

(continues on next page)

(continued from previous page)

```
   end_certs.emplace_back(Botan::X509_Certificate("ee.crt"));    // The end-entity␣
↪certificate, must come first
   end_certs.emplace_back(Botan::X509_Certificate("int2.crt")); // intermediate 2
   end_certs.emplace_back(Botan::X509_Certificate("int1.crt")); // intermediate 1

   // Optional: Set up restrictions, e.g. min. key strength, maximum age of OCSP␣
↪responses
   Botan::Path_Validation_Restrictions restrictions;

   // Optional: Specify usage type, compared against the key usage in end_certs[0]
   Botan::Usage_Type usage = Botan::Usage_Type::UNSPECIFIED;

   // Optional: Specify hostname, if not empty, compared against the DNS name in end_
↪certs[0]
   std::string hostname;

   Botan::Path_Validation_Result validationResult =
      Botan::x509_path_validate(end_certs, restrictions, trusted_roots, hostname, usage);

   if(!validationResult.successful_validation()) {
      // call validationResult.result() to get the overall status code
      return -1;
   }

   return 0;  // Verification succeeded
}
```

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 some default. The only time you need this alternate interface is for creating RSA-PSS certificates.

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

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)

If you need more control over the signing process, you can use the methods

static *X509_Certificate* **X509_CA::make_cert**(*PK_Signer* &signer, *RandomNumberGenerator* &rng, const *BigInt* &serial_number, const AlgorithmIdentifier &sig_algo, const std::vector<uint8_t> &pub_key, const X509_Time &not_before, const X509_Time &not_after, const *X509_DN* &issuer_dn, const *X509_DN* &subject_dn, const *Extensions* &extensions)

static *Extensions* **X509_CA::choose_extensions**(const PKCS10_Request &req, const *X509_Certificate* &ca_certificate, const std::string &hash_fn)

Returns the extensions that would be created by sign_request if it was used. You can call this and then modify the extensions list before invoking *X509_CA::make_cert*

8.11.8 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.

8.11.9 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).

8.11.10 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.

8.11.11 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): `DigitalSignature`, `NonRepudiation`, `KeyEncipherment`, `DataEncipherment`, `KeyAgreement`, `KeyCertSign`, `CrlSign`, `EncipherOnly`, or `DecipherOnly`. 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 based on the key type.

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://datatracker.ietf.org/doc/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<*X509_Certificate*> &cert_path = const std::vector<*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://datatracker.ietf.org/doc/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`.

8.12 Transport Layer Security (TLS)

Botan has client and server implementations of TLS 1.2 and 1.3. Support for older versions of the protocol was removed with Botan 3.0.

There is also support for DTLS (currently v1.2 only), 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.

Note that we support *an optional Boost ASIO stream* that is a convenient way to use Botan's TLS implementation as an almost drop-in replacement of ASIO's *ssl::stream*. Applications that build their network layer on Boost ASIO are advised to use this wrapper of `TLS::Client` and `TLS::Server`.

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

void **`tls_emit_data`**(std::span<const uint8_t> data)

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, std::span<const uint8_t> data)

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.

void **`tls_session_established`**(const Botan::TLS::Session_Summary &session)

Optional - default implementation is a no-op Called whenever a negotiation completes. This can

happen more than once on TLS 1.2 connections, if renegotiation occurs. The *session* parameter provides information about the session which was just established.

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, std::string_view 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. The server may also throw an exception to reject the connection; this is recommended when the client sends a list of protocols and the server does not understand any of them.

> **Warning:** The ALPN RFC requires that if the server does not understand any of the protocols offered by the client, it should close the connection using an alert. Carrying on the connection (for example by ignoring ALPN when the server does not understand the protocol list) can expose applications to cross-protocol attacks.

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

Optional. This may be called by TLS 1.3 clients or servers when OCSP stapling was negotiated. In contrast to `tls_provide_cert_status`, this allows providing OCSP responses for each certificate in the chain.

Note that the returned list of encoded OCSP responses must be of the same length as the input list of certificates in the chain. By default, this will call `tls_provide_cert_status` to obtain an OCSP response for the end-entity only.

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)

8.12.1 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**(std::span<const 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**(std::string_view str)

void **send**(std::span<const 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 **is_closed_for_reading**()

TLS 1.3 supports half-open connections. If the peer notified a connection closure, this will return true. For TLS 1.2 this will always return the same `is_closed`.

bool **is_closed_for_writing**()

TLS 1.3 supports half-open connections. After calling `close` on the channel, this will return true. For TLS 1.2 this will always return the same `is_closed`.

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.

Note that TLS 1.3 does not support renegotiation. This method will throw when called on a channel that uses TLS 1.3.

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.

void **update_traffic_keys**(bool request_peer_update = false)

After a successful handshake, this will update our traffic keys and may send a request to do the same to the peer.

Note that this is a TLS 1.3 feature and invocations on a channel using TLS 1.2 will throw.

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. Acting as a server, this value will ordinarily be empty, unless the server requested a certificate and the client responded with one.

std::optional<std::string> **external_psk_identity**() const

When this connection was established using a user-defined Preshared Key this will return the identity of the PSK used. If no PSK was used in the establishment of the connection this will return std::nullopt.

Note that TLS 1.3 session resumption is based on PSKs internally. Neverthelees, connections that were established using a session resumption will return std::nullopt here.

SymmetricKey **key_material_export**(std::string_view label, std::string_view 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://datatracker.ietf.org/doc/html/rfc5705.html), *label* should begin with "EXPERIMENTAL" unless the label has been standardized in an RFC.

8.12.2 TLS Clients

class TLS::**Client**

Client(const std::shared_ptr<Callbacks> &callbacks, const std::shared_ptr<Session_Manager> &session_manager, const std::shared_ptr<*Credentials_Manager*> &creds, const std::shared_ptr<const *Policy*> &policy, const std::shared_ptr<*RandomNumberGenerator*> &rng, Server_Information server_info = Server_Information(), *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, private keys, or pre-shared keys; 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: TLS Client

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/auto_rng.h>
#include <botan/certstor.h>
#include <botan/certstor_system.h>
#include <botan/tls.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(std::span<const uint8_t> data) override {
         // send data to tls server, e.g., using BSD sockets or boost asio
         BOTAN_UNUSED(data);
      }

      void tls_record_received(uint64_t seq_no, std::span<const uint8_t> data) override {
         // process full TLS record received by tls server, e.g.,
         // by passing it to the application
         BOTAN_UNUSED(seq_no, data);
      }

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

/**
 * @brief Credentials storage for the tls client.
 *
 * It returns a list of trusted CA certificates.
```

(continues on next page)

(continued from previous page)

```
 * Here we base trust on the system managed trusted CA list.
 * 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 {
         BOTAN_UNUSED(type, context);
         // return a list of certificates of CAs we trust for tls server certificates
         // ownership of the pointers remains with Credentials_Manager
         return {&m_cert_store};
      }

      std::vector<Botan::X509_Certificate> cert_chain(
         const std::vector<std::string>& cert_key_types,
         const std::vector<Botan::AlgorithmIdentifier>& cert_signature_schemes,
         const std::string& type,
         const std::string& context) override {
         BOTAN_UNUSED(cert_key_types, cert_signature_schemes, type, context);

         // when using tls client authentication (optional), return
         // a certificate chain being sent to the tls server,
         // else an empty list
         return {};
      }

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

   private:
      Botan::System_Certificate_Store m_cert_store;
};

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

   // open the tls connection
```

(continues on next page)

(continued from previous page)

```
   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()
   }

   return 0;
}
```

8.12.3 TLS Servers

class TLS::**Server**

> **Server**(const std::shared_ptr<Callbacks> &callbacks, const std::shared_ptr<Session_Manager> &session_manager, const std::shared_ptr<*Credentials_Manager*> &creds, const std::shared_ptr<const *Policy*> &policy, const std::shared_ptr<*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: TLS Server

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/auto_rng.h>
#include <botan/certstor.h>
#include <botan/pk_keys.h>
#include <botan/pkcs8.h>
#include <botan/tls.h>

#include <memory>
```

(continues on next page)

(continued from previous page)

```
/**
 * @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(std::span<const uint8_t> data) override {
         // send data to tls client, e.g., using BSD sockets or boost asio
         BOTAN_UNUSED(data);
      }

      void tls_record_received(uint64_t seq_no, std::span<const uint8_t> data) override {
         // process full TLS record received by tls client, e.g.,
         // by passing it to the application
         BOTAN_UNUSED(seq_no, data);
      }

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

/**
 * @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() {
         Botan::DataSource_Stream in("botan.randombit.net.key");
         m_key.reset(Botan::PKCS8::load_key(in).release());
      }

      std::vector<Botan::Certificate_Store*> trusted_certificate_authorities(const␣
↪std::string& type,
                                                                              const␣
↪std::string& context) override {
         BOTAN_UNUSED(type, context);
         // 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 {};
      }
```

(continues on next page)

(continued from previous page)

```
      std::vector<Botan::X509_Certificate> cert_chain(
         const std::vector<std::string>& cert_key_types,
         const std::vector<Botan::AlgorithmIdentifier>& cert_signature_schemes,
         const std::string& type,
         const std::string& context) override {
         BOTAN_UNUSED(cert_key_types, cert_signature_schemes, type, context);

         // 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")};
      }

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

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

int main() {
   // prepare all the parameters
   auto callbacks = std::make_shared<Callbacks>();
   auto rng = std::make_shared<Botan::AutoSeeded_RNG>();
   auto session_mgr = std::make_shared<Botan::TLS::Session_Manager_In_Memory>(rng);
   auto creds = std::make_shared<Server_Credentials>();
   auto policy = std::make_shared<Botan::TLS::Strict_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()
   // ...

   return 0;
}
```

8.12.4 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 the handshake callback (`TLS::Callbacks::tls_session_established`) is called, a new session has been established, and a `TLS::Session_Summary` is included that provides information about that session:

class `TLS::Session_Summary`

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.

bool **was_resumption**() const

Returns true if the session resulted from a resumption of a previously established session.

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

Returns the certificate chain of the peer

std::optional<std::string> **external_psk_identity**() const

If the session was established using a user-provided Preshared Key, its identity will be provided here. If no PSK was used, std::nullopt will be reported.

bool **psk_used**() const

Returns true if the session was established using a user-provided Preshared Key.

8.12.5 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.

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.

The interface of the `TLS::Session_Manager` was completely redesigned with Botan 3.0 to accommodate the new requirements of TLS 1.3. Please also see *the migration guide* for an outline of the differences between the Botan 2.x and 3.x API.

In Botan 3.0 the server-side `TLS::Session_Manager` gained the competency to decide whether to store sessions in a stateful database and just return a handle to it. Or to serialize the session into an encrypted ticket and pass it back to the client. To distinguish those use cases, Botan 3.0 introduced a `TLS::Session_Handle` class that is used throughout this API.

Below is a brief overview of the most important methods that a custom implementation must implement. There are more methods that provide applications with full flexibility to handle session objects. More detail can be found in the API documentation inline.

class TLS::**Session_Mananger**

void **store**(const *Session* &session, const Session_Handle &handle)

Attempts to save a new *session*. Typical implementations will use `TLS::Session::encrypt`, `TLS::Session::DER_encode` or `TLS::Session::PEM_encode` to obtain an opaque and serialized session object for storage. It is legal to simply drop an incoming session for whatever reason.

size_t **remove**(const Session_Handle &handle)

Remove the session identified by *handle*. Future attempts at resumption should fail for this session. Returns the number of sessions actually removed.

size_t **remove_all**()

Empties the session storage. Returns the number of sessions actually removed.

std::optional<*Session*> **retrieve_one**(const Session_Handle &handle)

Attempts to retrieve a single session that corresponds to *handle* from storage. Typical implementations will use `TLS::Session::decrypt` or the `TLS::Session` constructors that deserialize a session from DER or PEM. If no session was found for the given *handle*, return std::nullopt. This method is called in TLS servers to find a specific session for a given handle.

std::vector<Session_with_Handle> **find_some**(const Server_Information &info, size_t max_sessions_hint)

Try to find some saved sessions using information about the server. TLS 1.3 clients may offer more than one session for resumption to the server. It is okay to ignore the *max_sessions_hint* and just return exactly one or no sessions at all.

recursive_mutex_type &**mutex**()

Derived implementations may use this mutex to serialize concurrent requests.

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)

Limits the maximum number of saved sessions to *max_sessions*.

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(std::string_view passphrase, const std::shared_ptr<*RandomNumberGenerator*> &rng, std::string_view db_filename, size_t max_sessions = 1000)

Uses the sqlite3 database named by *db_filename*.

Stateless Session Manager

This session manager is useful for servers that want to implement stateless session resumption. If supported by the client, sessions are always encoded as opaque and encrypted session tickets. Sessions are encrypted with a symmetric secret obtained via `TLS::Credentials_Manager::session_ticket_key()`.

Session_Manager_Stateless(const std::shared_ptr<*Credentials_Manager*> &credentials_manager, const std::shared_ptr<*RandomNumberGenerator*> &rng)

Creates a stateless session manager.

Hybrid Session Manager

This is a meta-manager that combines a `TLS::Session_Manager_Stateless` with any (built-in or user-provided) stateful session manager. Typically, such a hybrid manager is useful for TLS servers that want to support both stateless session tickets and stateful session storage.

Session_Manager_Hybrid(std::unique_ptr<Session_Manager> stateful_manager, const std::shared_ptr<*Credentials_Manager*> &credentials_manager, const std::shared_ptr<*RandomNumberGenerator*> &rng, bool prefer_tickets = true)

Creates a hybrid session manager that uses *stateful_manager* as its storage backend when session tickets are not supported or desired.

8.12.6 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"

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**): "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: All CBC ciphersuites 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: "ECDH", "DH"

Also allowed: "RSA", "ECDHE_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 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): "IMPLICIT"

"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 and static RSA ciphersuites.

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.

std::vector<Group_Param> **key_exchange_groups_to_offer**() const

Return a list of groups to opportunistically offer key exchange information for in the initial ClientHello when offering TLS 1.3. This policy has no effect on TLS 1.2 connections.

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_tls12**() const

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

bool **allow_tls13**() const

Return true from here to allow TLS v1.3. 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 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

std::chrono::seconds **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)

size_t **new_session_tickets_upon_handshake_success**() const

Return the number of session tickets a TLS 1.3 server should issue automatically once a successful handshake was made. Alternatively, users may manually call `TLS::Server::send_new_session_tickets()` at any time after a successful handshake.

Default: 1

std::optional<uint16_t> **record_size_limit**() const

Defines the maximum TLS record length this peer is willing to receive or std::nullopt in case of no preference (will use the maximum allowed).

This is currently implemented for TLS 1.3 only and will not be negotiated if TLS 1.2 is used or allowed.

Default: no preference (use maximum allowed by the protocol)

bool **tls_13_middlebox_compatibility_mode**() const

Enables middlebox compatibility mode as defined in RFC 8446 Appendix D.4.

Default: true

8.12.7 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.

8.12.8 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

8.12.9 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)

8.12.10 Post-quantum-secure key exchange

Added in version ::: 3.2

Botan allows TLS 1.3 handshakes using both pure post-quantum secure algorithms or a hybrid key exchange that combines a classical and a post-quantum secure algorithm. For the latter it implements the recent IETF draft-ietf-tls-hybrid-design (https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design).

Note that post-quantum key exchanges in TLS 1.3 are not conclusively standardized. Therefore, the key exchange group identifiers used by various TLS 1.3 implementations are not consistent. Applications that wish to enable hybrid key exchanges must enable the hybrid algorithms in their TLS policy. Override *TLS::Policy::key_exchange_groups()* and return a list of the desired exchange groups. For text-based policy configurations use the identifiers in parenthesis.

Currently, Botan supports the following post-quantum secure key exchanges:

- used in Open Quantum Safe (https://github.com/open-quantum-safe/oqs-provider/blob/main/oqs-template/oqs-kem-info.md) (PQC algorithm without a classical algorithm)
  - `KYBER_512_R3` ("Kyber-512-r3")
  - `KYBER_768_R3` ("Kyber-768-r3")
  - `KYBER_1024_R3` ("Kyber-1024-r3")
- used in Open Quantum Safe (https://github.com/open-quantum-safe/oqs-provider/blob/main/oqs-template/oqs-kem-info.md) (hybrid between Kyber and a classical ECDH algorithm)
  - `HYBRID_X25519_KYBER_512_R3_OQS` ("x25519/Kyber-512-r3")

  - HYBRID_X25519_KYBER_768_R3_OQS ("x25519/Kyber-768-r3")
  - HYBRID_SECP256R1_KYBER_512_R3_OQS ("secp256r1/Kyber-512-r3")
  - HYBRID_SECP384R1_KYBER_768_R3_OQS ("secp384r1/Kyber-768-r3")
  - HYBRID_SECP521R1_KYBER_1024_R3_OQS ("secp521r1/Kyber-1024-r3")
- used by Cloudflare (https://blog.cloudflare.com/post-quantum-for-all/) (hybrid between Kyber and the classical X25519 algorithm)
  - HYBRID_X25519_KYBER_512_R3_CLOUDFLARE ("x25519/Kyber-512-r3/cloudflare")
  - HYBRID_X25519_KYBER_768_R3_CLOUDFLARE ("x25519/Kyber-768-r3/cloudflare")

Code Example: Hybrid TLS Client

```
#include <botan/auto_rng.h>
#include <botan/certstor.h>
#include <botan/tls.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(std::span<const uint8_t> data) override {
         BOTAN_UNUSED(data);
         // send data to tls server, e.g., using BSD sockets or boost asio
      }

      void tls_record_received(uint64_t seq_no, std::span<const uint8_t> data) override {
         BOTAN_UNUSED(seq_no, data);
         // process full TLS record received by tls server, e.g.,
         // by passing it to the application
      }

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

/**
 * @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:
```

(continues on next page)

(continued from previous page)

```
      std::vector<Botan::Certificate_Store*> trusted_certificate_authorities(const␣
↪std::string& type,
                                                                              const␣
↪std::string& context) override {
         BOTAN_UNUSED(type, context);
         // return a list of certificates of CAs we trust for tls server certificates,
         // e.g., all the certificates in the local directory "cas"
         return {&m_cert_store};
      }

   private:
      Botan::Certificate_Store_In_Memory m_cert_store{"cas"};
};

class Client_Policy : public Botan::TLS::Default_Policy {
   public:
      // This needs to be overridden to enable the hybrid PQ/T groups
      // additional to the default (classical) key exchange groups
      std::vector<Botan::TLS::Group_Params> key_exchange_groups() const override {
         auto groups = Botan::TLS::Default_Policy::key_exchange_groups();
         groups.push_back(Botan::TLS::Group_Params::HYBRID_X25519_KYBER_512_R3_
↪CLOUDFLARE);
         groups.push_back(Botan::TLS::Group_Params::HYBRID_X25519_KYBER_512_R3_OQS);
         return groups;
      }

      // Define that the client should exclusively pre-offer hybrid groups
      // in its initial Client Hello.
      std::vector<Botan::TLS::Group_Params> key_exchange_groups_to_offer() const␣
↪override {
         return {Botan::TLS::Group_Params::HYBRID_X25519_KYBER_512_R3_CLOUDFLARE,
                 Botan::TLS::Group_Params::HYBRID_X25519_KYBER_512_R3_OQS};
      }
};

int main() {
   // prepare all the parameters
   auto rng = std::make_shared<Botan::AutoSeeded_RNG>();
   auto callbacks = std::make_shared<Callbacks>();
   auto session_mgr = std::make_shared<Botan::TLS::Session_Manager_In_Memory>(rng);
   auto creds = std::make_shared<Client_Credentials>();
   auto policy = std::make_shared<Botan::TLS::Strict_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);
```

(continues on next page)

(continued from previous page)

```
   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()
   }

   return 0;
}
```

8.12.11 TLS Custom Key Exchange Mechanisms

Applications can override the ephemeral key exchange mechanism used in TLS. This is not necessary for typical applications and might pose a serious security risk. Though, it allows the usage of custom groups or curves, offloading of cryptographic calculations to special hardware or the addition of entirely different algorithms (e.g. for post-quantum resilience).

From a technical point of view, 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. Groups are represented by their TLS identifier. This two-byte identifier is standardized for commonly used groups and curves. In addition, the standard reserves the identifiers 0xFE00 to 0xFEFF for custom groups, curves or other algorithms.

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 callbacks `tls_generate_ephemeral_key` and `tls_ephemeral_key_agreement`
3. Adjustment of the TLS policy by allowing the custom curve

Below is a code example for a TLS client using a custom curve. For servers, it works exactly the same.

Code Example: TLS Client using Custom Curve

```
#include <botan/auto_rng.h>
#include <botan/certstor.h>
#include <botan/ecdh.h>
#include <botan/tls.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(std::span<const uint8_t> data) override {
         BOTAN_UNUSED(data);
         // send data to tls server, e.g., using BSD sockets or boost asio
```

(continues on next page)

(continued from previous page)

```
    }

    void tls_record_received(uint64_t seq_no, std::span<const uint8_t> data) override {
       BOTAN_UNUSED(seq_no, data);
       // process full TLS record received by tls server, e.g.,
       // by passing it to the application
    }

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

    std::unique_ptr<Botan::PK_Key_Agreement_Key> tls_generate_ephemeral_key(
       const std::variant<Botan::TLS::Group_Params, Botan::DL_Group>& group,
       Botan::RandomNumberGenerator& rng) override {
       if(std::holds_alternative<Botan::TLS::Group_Params>(group) &&
          std::get<Botan::TLS::Group_Params>(group) == Botan::TLS::Group_
↪Params(0xFE00)) {
          // generate a private key of my custom curve
          const auto ec_group = Botan::EC_Group::from_name("numsp256d1");
          return std::make_unique<Botan::ECDH_PrivateKey>(rng, ec_group);
       } else {
          // no custom curve used: up-call the default implementation
          return tls_generate_ephemeral_key(group, rng);
       }
    }

    Botan::secure_vector<uint8_t> tls_ephemeral_key_agreement(
       const std::variant<Botan::TLS::Group_Params, Botan::DL_Group>& group,
       const Botan::PK_Key_Agreement_Key& private_key,
       const std::vector<uint8_t>& public_value,
       Botan::RandomNumberGenerator& rng,
       const Botan::TLS::Policy& policy) override {
       if(std::holds_alternative<Botan::TLS::Group_Params>(group) &&
          std::get<Botan::TLS::Group_Params>(group) == Botan::TLS::Group_
↪Params(0xFE00)) {
          // perform a key agreement on my custom curve
          const auto ec_group = Botan::EC_Group::from_name("numsp256d1");
          Botan::ECDH_PublicKey peer_key(ec_group, ec_group.OS2ECP(public_value));
          Botan::PK_Key_Agreement ka(private_key, rng, "Raw");
          return ka.derive_key(0, peer_key.public_value()).bits_of();
       } else {
          // no custom curve used: up-call the default implementation
          return tls_ephemeral_key_agreement(group, private_key, public_value, rng,␣
↪policy);
       }
    }
};

/**
 * @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 {
         BOTAN_UNUSED(type, context);
         // return a list of certificates of CAs we trust for tls server certificates,
         // e.g., all the certificates in the local directory "cas"
         return {&m_cert_store};
      }

   private:
      Botan::Certificate_Store_In_Memory m_cert_store{"cas"};
};

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

         // NOLINTNEXTLINE(clang-analyzer-optin.core.EnumCastOutOfRange)
         return {static_cast<Botan::TLS::Group_Params>(0xFE00)};
      }
};

int main() {
   // prepare rng
   auto rng = std::make_shared<Botan::AutoSeeded_RNG>();

   // prepare custom curve

   // prepare curve parameters

   // In this case we will use numsp256d1 from https://datatracker.ietf.org/doc/html/
↪draft-black-numscurves-02

   const Botan::BigInt p(
↪"0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF43");
   const Botan::BigInt a(
↪"0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF40");
   const Botan::BigInt b("0x25581");
   const Botan::BigInt n(
↪"0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE43C8275EA265C6020AB20294751A825");

   const Botan::BigInt g_x("0x01");
   const Botan::BigInt g_y(
↪"0x696F1853C1E466D7FC82C96CCEEEDD6BD02C2F9375894EC10BF46306C2B56C77");
```

(continues on next page)

(continued from previous page)

```
   // This is an OID reserved in Botan's private arc for numsp256d1
   // If you use some other curve you should create your own OID
   const Botan::OID oid("1.3.6.1.4.1.25258.4.1");

   // create EC_Group object to register the curve
   Botan::EC_Group numsp256d1(oid, p, a, b, g_x, g_y, n);

   if(!numsp256d1.verify_group(*rng)) {
      return 1;
      // Warning: if verify_group returns false the curve parameters are insecure
   }

   // register name to specified oid
   Botan::OID::register_oid(oid, "numsp256d1");

   // prepare all the parameters
   auto callbacks = std::make_shared<Callbacks>();
   auto session_mgr = std::make_shared<Botan::TLS::Session_Manager_In_Memory>(rng);
   auto creds = std::make_shared<Client_Credentials>();
   auto policy = std::make_shared<Botan::TLS::Strict_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()
   }

   return 0;
}
```

8.12.12 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.

To use the asio stream wrapper, a relatively recent version of boost is required. Include `botan/asio_compat.h` and check that `BOTAN_FOUND_COMPATIBLE_BOOST_ASIO_VERSION` is defined before including `botan/asio_stream.h` to be ensure compatibility at compile time of your application.

The asio Stream 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.

Code Examples: HTTPS Client using Boost Beast

Starting with Botan 3.3.0 (and assuming a recent version of Boost), one may use Botan's TLS using C++20 coroutines. The following example implements a minimal HTTPS client using Botan's default settings to fetch content from web servers.

To establish trust in the server's certificate, Botan attempts to use the system's trust store (supported on macOS, Linux and Windows). If that does not work, you might get an error indicating that the certificate is not trusted. In that case, you can provide a custom trust store by subclassing the *Credentials_Manager* and passing it to the *TLS::Stream* as shown in *this example*.

Note that Botan's default TLS policy requires servers to provide a valid CRL or OCSP response for their certificate. To disable this, derive the default policy class *TLS::Policy*, override `require_cert_revocation_info` accordingly and pass an object of your policy via the *TLS::Context* to the *TLS::Stream*.

```
#include <iostream>

#include <botan/asio_compat.h>

// Boost 1.81.0 introduced support for the finalized C++20 coroutines
// in clang 14 and newer. Older versions of Boost might work with other
// compilers, though.
#if defined(BOTAN_FOUND_COMPATIBLE_BOOST_ASIO_VERSION) && BOOST_VERSION >= 108100

   #include <botan/asio_stream.h>
   #include <botan/version.h>

   #include <boost/asio/awaitable.hpp>
   #include <boost/asio/co_spawn.hpp>
   #include <boost/asio/detached.hpp>
   #include <boost/asio/use_awaitable.hpp>
   #include <boost/beast/core.hpp>
   #include <boost/beast/http.hpp>
   #include <boost/beast/version.hpp>

namespace beast = boost::beast;
namespace http = beast::http;
namespace net = boost::asio;
namespace tls = Botan::TLS;
using tcp = boost::asio::ip::tcp;

namespace {

http::request<http::string_body> create_GET_request(const std::string& host, const␣
↪std::string& target) {
   http::request<http::string_body> req;
   req.version(11);
   req.method(http::verb::get);
   req.target(target);
   req.set(http::field::host, host);
   req.set(http::field::user_agent, Botan::version_string());
   return req;
}

net::awaitable<void> request(std::string host, std::string port, std::string target) {
   // Lookup host address
   auto resolver = net::use_awaitable.as_default_on(tcp::resolver(co_await net::this_
↪coro::executor));
   const auto dns_result = co_await resolver.async_resolve(host, port);

   // Connect to host and establish a TLS session
   auto tls_stream =
      tls::Stream(tls::Server_Information(host),
                  net::use_awaitable.as_default_on(beast::tcp_stream(co_await net::this_
↪coro::executor)));
   tls_stream.next_layer().expires_after(std::chrono::seconds(30));
   co_await tls_stream.next_layer().async_connect(dns_result);
   co_await tls_stream.async_handshake(tls::Connection_Side::Client);
```

(continues on next page)

(continued from previous page)

```
   // Send HTTP GET request
   tls_stream.next_layer().expires_after(std::chrono::seconds(30));
   co_await http::async_write(tls_stream, create_GET_request(host, target));

   // Receive HTTP response and print result
   beast::flat_buffer b;
   http::response<http::dynamic_body> res;
   co_await http::async_read(tls_stream, b, res);
   std::cout << res << std::endl;

   // Terminate connection
   co_await tls_stream.async_shutdown();
   tls_stream.next_layer().close();
}

}  // namespace

int main(int argc, char* argv[]) {
   if(argc != 4) {
      std::cerr << "Usage: tls_stream_coroutine_client <host> <port> <target>\n"
                << "Example:\n"
                << "    tls_stream_coroutine_client botan.randombit.net 443 /news.html\n
↪";
      return 1;
   }

   const auto host = argv[1];
   const auto port = argv[2];
   const auto target = argv[3];

   int return_code = 0;

   try {
      net::io_context ioc;

      net::co_spawn(ioc, request(host, port, target), [&](const std::exception_ptr&␣
↪eptr) {
         if(eptr) {
            try {
               std::rethrow_exception(eptr);
            } catch(std::exception& ex) {
               std::cerr << "Error: " << ex.what() << "\n";
               return_code = 1;
            }
         }
      });

      ioc.run();
   } catch(std::exception& e) {
      std::cerr << e.what() << "\n";
   }
```

(continues on next page)

(continued from previous page)

```
   return return_code;
}

#else

int main() {
   std::cout << "Your boost version is too old, sorry.\n"
             << "Or did you compile Botan without --with-boost?\n";
   return 1;
}

#endif
```

Aside of the modern coroutines-based approach, the ASIO stream may also be used in a more traditional way, using callback handler methods instead of coroutines.

Also, this example shows how to use a custom *Credentials_Manager* and pass it to the *TLS::Stream* via a *TLS::Context* object.

```
#include <iostream>

#include <botan/asio_compat.h>
#if defined(BOTAN_FOUND_COMPATIBLE_BOOST_ASIO_VERSION)

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

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

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

// very basic credentials manager
class Credentials_Manager : public Botan::Credentials_Manager {
   public:
      Credentials_Manager() = default;

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

   private:
      Botan::System_Certificate_Store m_cert_store;
```

(continues on next page)

(continued from previous page)

```
};

// a simple https client based on TLS::Stream
class client {
   public:
      client(boost::asio::io_context& io_context,
             boost::asio::ip::tcp::resolver::iterator endpoint_iterator,
             std::string_view host,
             const http::request<http::string_body>& req) :
            m_request(req),
            m_ctx(std::make_shared<Botan::TLS::Context>(std::make_shared<Credentials_
↪Manager>(),
                                                        std::make_shared
↪<Botan::AutoSeeded_RNG>(),
                                                        std::make_shared
↪<Botan::TLS::Session_Manager_Noop>(),
                                                        std::make_shared
↪<Botan::TLS::Policy>(),
                                                        host)),
            m_stream(io_context, m_ctx) {
         boost::asio::async_connect(m_stream.lowest_layer(),
                                    std::move(endpoint_iterator),
                                    boost::bind(&client::handle_connect, this,␣
↪ap::error));
      }

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

      void handle_handshake(const boost::system::error_code& error) {
         if(error) {
            std::cout << "Handshake failed: " << error.message() << '\n';
            return;
         }
         http::async_write(
            m_stream, m_request, boost::bind(&client::handle_write, this, ap::error,␣
↪ap::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(
```

(continues on next page)

(continued from previous page)

```
            m_stream, m_reply, m_response, boost::bind(&client::handle_read, this,␣
↪ap::error, ap::bytes_transferred));
      }

      void handle_read(const boost::system::error_code& error, size_t) {
         if(!error) {
            std::cout << "Reply: ";
            std::cout << m_response.body() << '\n';
         } else {
            std::cout << "Read failed: " << error.message() << '\n';
         }
      }

   private:
      http::request<http::dynamic_body> m_request;
      http::response<http::string_body> m_response;
      boost::beast::flat_buffer m_reply;

      std::shared_ptr<Botan::TLS::Context> m_ctx;
      Botan::TLS::Stream<boost::asio::ip::tcp::socket> m_stream;
};

int main(int argc, char* argv[]) {
   if(argc != 4) {
      std::cerr << "Usage: tls_stream_client <host> <port> <target>\n"
                << "Example:\n"
                << "    tls_stream_client botan.randombit.net 443 /news.html\n";
      return 1;
   }

   const auto host = argv[1];
   const auto port = argv[2];
   const auto target = argv[3];

   try {
      boost::asio::io_context io_context;

      boost::asio::ip::tcp::resolver resolver(io_context);
      boost::asio::ip::tcp::resolver::query query(host, port);
      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(target);
      req.set(http::field::host, host);
      req.set(http::field::user_agent, Botan::version_string());

      client c(io_context, iterator, host, req);

      io_context.run();
   } catch(std::exception& e) {
```

(continues on next page)

(continued from previous page)

```
      std::cerr << e.what();
      return 1;
   }

   return 0;
}

#else

int main() {
   std::cout << "Your boost version is too old, sorry.\n"
             << "Or did you compile Botan without --with-boost?\n";
   return 1;
}

#endif
```

8.12.13 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.

8.13 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*.

For servers *type* will be "tls-server" and the *context* will be the server name that the client requested via SNI (or empty, if the client did not send SNI).

Warning: To avoid cross-protocol attacks it is recommended that if a server receives an SNI request for a name it does not expect, it should close the connection with an alert. This can be done by throwing an exception from the implementation of this function.

std::shared_ptr<Private_Key> **private_key_for**(const *X509_Certificate* &cert, const std::string &type, const std::string &context)

Return a shared pointer to 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*.

8.13.1 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)

8.13.2 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://datatracker.ietf.org/doc/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.

8.14 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::**from_u64**(uint64_t n)

Create a BigInt with value *n*

BigInt(std::string_view str)

Create a BigInt from a string. By default decimal is expected. With an 0x prefix instead it is treated as hexadecimal. A - prefix to indicate negative numbers is also accepted.

BigInt(std::span<const uint8_t> buf)

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 **serialize_to**(std::span<uint8_t> buf)

Encode this BigInt as a big-endian integer. The sign is ignored.

There must be sufficient space to encode the entire integer in `buf`. If `buf` is larger than required, sufficient zero bytes will be prefixed.

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, with "0x" prefix

8.14.1 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`.

8.15 Key Derivation Functions (KDF)

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.

Typically a KDF is also used with a *salt* and a *label*. The *salt* should be some random information which is available to all of the parties that would need to use the KDF; this could be performed by setting the salt to some kind of session identifier, or by having one of the parties generate a random salt and including it in a message.

The *label* is used to bind the KDF output to some specific context. For instance if you were using the KDF to derive a specific key referred to as the "message key" in the protocol description, you might use a label of "FooProtocol v2 MessageKey". This labeling ensures that if you accidentally use the same input key and salt in some other context, you still use different keys in the two contexts.

class **KDF**

std::unique_ptr<KDF> KDF::**create**(const std::string &algo)

Create a new KDF object. Returns nullptr if the named key derivation function was not available

std::unique_ptr<KDF> KDF::**create_or_throw**(const std::string &algo)

Create a new KDF object. Throws an exception if the named key derivation function was not available

template<concepts::resizable_byte_buffer **T** = secure_vector<uint8_t>>
T **derive_key**(size_t key_len, std::span<const uint8_t> secret, std::span<const uint8_t> salt, std::span<const uint8_t> label) const

This version is parameterized to the output buffer type, so it can be used to return a `std::vector`, a `secure_vector`, or anything else satisfying the `resizable_byte_buffer` concept.

secure_vector<uint8_t> **derive_key**(const uint8_t secret[], size_t secret_len, const uint8_t salt[], size_t salt_len, const uint8_t label[], size_t label_len) const

secure_vector<uint8_t> **derive_key**(size_t key_len, const std::vector<uint8_t> &secret, const std::vector<uint8_t> &salt, const std::vector<uint8_t> &label) 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*, *salt*, and *label*, whose meaning is described above.

8.15.1 Code Example

An example demonstrating using the API to hash a secret using HKDF

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

int main() {
   // Replicate a test from RFC 5869
   // https://www.rfc-editor.org/rfc/rfc5869#appendix-A.1
   const std::vector<uint8_t> input_secret(22, 0x0b);
   const std::vector<uint8_t> salt = Botan::hex_decode("000102030405060708090a0b0c");
   const std::vector<uint8_t> label = Botan::hex_decode("f0f1f2f3f4f5f6f7f8f9");
   const size_t derived_key_len = 42;

   auto kdf = Botan::KDF::create_or_throw("HKDF(SHA-256)");
   auto derived_key = kdf->derive_key(derived_key_len, input_secret, salt, label);

   // OKM = 0x3cb25f25faacd57a90434f64d0362f2a...
   std::cout << Botan::hex_encode(derived_key) << '\n';
}
```

8.15.2 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. There is no technical reason (besides compatability) to choose any other KDF.

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.

Algorithm specification names:

- `HKDF(<MessageAuthenticationCode|HashFunction>)`, e.g. `HKDF(HMAC(SHA-256))`
- `HKDF-Extract(<MessageAuthenticationCode|HashFunction>)`
- `HKDF-Expand(<MessageAuthenticationCode|HashFunction>)`

If a `HashFunction` is provided as an argument, it will create `HMAC(HashFunction)` as the `MessageAuthenticationCode`. I.e. `HKDF(SHA-256)` will result in `HKDF(HMAC(SHA-256))`.

KDF1-18033

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

Available if `BOTAN_HAS_KDF1_18033` is defined.

Algorithm specification name: `KDF1-18033(<HashFunction>)`, e.g. `KDF1-18033(SHA-512)`

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.

Algorithm specification name: `KDF1(<HashFunction>)`, e.g. `KDF1(SHA-512)`

KDF2

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

Available if `BOTAN_HAS_KDF2` is defined.

Algorithm specification name: `KDF2(<HashFunction>)`, e.g. `KDF2(SHA-512)`

X9.42 PRF

A KDF from ANSI X9.42. Sometimes used for Diffie-Hellman. However it is overly complicated and is fixed to use only SHA-1.

Available if `BOTAN_HAS_X942_PRF` is defined.

> **Warning:** X9.42 PRF is deprecated and will be removed in a future major release.

Algorithm specification name: `X9.42-PRF(<OID>)`, e.g. `X9.42-PRF(KeyWrap.TripleDES)`, `X9.42-PRF(1.2.840.113549.1.9.16.3.7)`

SP800-56A

KDF from NIST SP 800-56Ar2 or One-Step KDF of SP 800-56Cr2.

Available if `BOTAN_HAS_SP800_56A` is defined.

Algorithm specification names:

- `SP800-56A(<HashFunction>)`, e.g. `SP800-56A(SHA-256)`
- `SP800-56A(HMAC(<HashFunction>))`, e.g. `SP800-56A(HMAC(SHA-256))`
- `SP800-56A(KMAC-128)` or `SP800-56A(KMAC-256)`

SP800-56C

Two-Step KDF from NIST SP 800-56Cr2.

Available if `BOTAN_HAS_SP800_56C` is defined.

Algorithm specification name: `SP800-56C(<MessageAuthenticationCode|HashFunction>)`, e.g. `SP800-56C(HMAC(SHA-256))`

If a `HashFunction` is provided as an argument, it will create `HMAC(HashFunction)` as the `MessageAuthenticationCode`. I.e. `SP800-56C(SHA-256)` will result in `SP800-56C(HMAC(SHA-256))`.

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.

Algorithm specification names:

- `SP800-108-Counter(<MessageAuthenticationCode|HashFunction>)`, e.g. `SP800-108-Counter(HMAC(SHA-256))`
- `SP800-108-Feedback(<MessageAuthenticationCode|HashFunction>)`
- `SP800-108-Pipeline(<MessageAuthenticationCode|HashFunction>)`

If a `HashFunction` is provided as an argument, it will create `HMAC(HashFunction)` as the `MessageAuthenticationCode`. I.e. `SP800-108-Counter(SHA-256)` will result in `SP800-108-Counter(HMAC(SHA-256))`.

TLS 1.2 PRF

Implementation of the Pseudo-Random Function as used in TLS 1.2.

Available if `BOTAN_HAS_TLS_V12_PRF` is defined.

Algorithm specification name: `TLS-12-PRF(<MessageAuthenticationCode|HashFunction>)`, e.g. `TLS-12-PRF(HMAC(SHA-256))`

If a `HashFunction` is provided as an argument, it will create `HMAC(HashFunction)` as the `MessageAuthenticationCode`. I.e. `TLS-12-PRF(SHA-256)` will result in `TLS-12-PRF(HMAC(SHA-256))`.

8.16 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.

8.16.1 PasswordHash

Added in version 2.8.0.

This API, declared in `pwdhash.h`, has two classes, `PasswordHashFamily` representing the general algorithm, such as "PBKDF2(SHA-256)", or "Scrypt", and `PasswordHash` representing a specific instance of the problem which is fully specified with all parameters (say "Scrypt" with `N` = 8192, `r` = 64, and `p` = 8) and which can be used to derive keys.

class **PasswordHash**

void **hash**(std::span<uint8_t> out, std::string_view password, std::span<uint8> salt)

Derive a key from the specified *password* and *salt*, placing it into *out*.

void **hash**(std::span<uint8_t> out, std::string_view password, std::span<const uint8> salt, std::span<const uint8> ad, std::span<const uint8> key)

Derive a key from the specified *password*, *salt*, associated data (*ad*), and secret *key*, placing it into *out*. The *ad* and *key* are both allowed to be empty. Currently non-empty AD/key is only supported with Argon2.

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

Same functionality as the 3 argument variant of *PasswordHash::hash*.

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 uint8_t ad[], size_t ad_len, const uint8_t key[], size_t key_len) const

Same functionality as the 5 argument variant of *PasswordHash::hash*.

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

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

size_t **iterations**() const

Return the iteration parameter

size_t **memory_param**() const

Return the memory usage parameter, or 0 if the algorithm does not offer a memory usage option.

size_t **parallelism**() const

Returns the parallelism parameter, or 0 if the algorithm does not offer a parallelism option.

size_t **total_memory_usage**() const

Return a guesstimate of the total number of bytes of memory consumed when running this algorithm. If the function is not intended to be memory-hard and uses an effictively fixed amount of memory when running, this function returns 0.

bool **supports_keyed_operation**() const

Returns true if this password hash supports supplying a secret key to *PasswordHash::hash*.

bool **supports_associated_data**() const

Returns true if this password hash supports supplying associated data to *PasswordHash::hash*.

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", "Argon2id". Returns null if the algorithm is 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, size_t max_memory_usage_mb = 0, std::chrono::milliseconds tuning_msec = std::chrono::milliseconds(10)) 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.)

The parameters will be selected to use at most *max_memory_usage_mb* megabytes of memory, or if left as zero any size is allowed.

This function works by runing a short tuning loop to estimate the performance of the algorithm, then scaling the parameters appropriately to hit the target size. The length of time the tuning loop runs can be controlled using the *tuning_msec* parameter.

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. Parameters are as follows:

- For PBKDF2, PGP-S2K, and Bcrypt-PBKDF, `i1` is iterations
- Scrypt uses `i1` == `N`, `i2` == `r`, and `i3` == `p`
- Argon2 family uses `i1` == `M`, `i2` == `t`, and `i3` == `p`

All unneeded parameters should be set to 0 or left blank.

8.16.2 Code Examples

An example demonstrating using the API to hash a password using Argon2i:

```
#include <botan/hex.h>
#include <botan/pwdhash.h>
#include <botan/system_rng.h>
#include <array>
#include <iostream>

int main() {
   // You can change this to "PBKDF2(SHA-512)" or "Scrypt" or "Argon2id" or ...
   const std::string pbkdf_algo = "Argon2i";
   auto pbkdf_runtime = std::chrono::milliseconds(300);
   const size_t output_hash = 32;
   const size_t max_pbkdf_mb = 128;

   auto pwd_fam = Botan::PasswordHashFamily::create_or_throw(pbkdf_algo);

   auto pwdhash = pwd_fam->tune(output_hash, pbkdf_runtime, max_pbkdf_mb);

   std::cout << "Using params " << pwdhash->to_string() << '\n';

   std::array<uint8_t, 32> salt;
   Botan::system_rng().randomize(salt);

   const std::string password = "tell no one";

   std::array<uint8_t, output_hash> key;
   pwdhash->hash(key, password, salt);

   std::cout << Botan::hex_encode(key) << '\n';

   return 0;
}
```

Combining a password based key derivation with an authenticated cipher yields an application that can encrypt and decrypt data using a password. Note that this example does not incorporate any "associated data" into the AEAD. For instance, a real application might want to include a version number of their file format as associated data. See *AEAD Mode* for more information.

```
#include <botan/aead.h>
#include <botan/assert.h>
#include <botan/auto_rng.h>
#include <botan/hex.h>
#include <botan/pwdhash.h>

#include <iostream>

namespace {

template <typename OutT = std::vector<uint8_t>, typename... Ts>
OutT concat(const Ts&... buffers) {
   OutT out;
```

(continues on next page)

(continued from previous page)

```
   out.reserve((buffers.size() + ... + 0));
   (out.insert(out.end(), buffers.begin(), buffers.end()), ...);
   return out;
}

template <typename Out, typename In>
Out as(const In& data) {
   return Out(data.data(), data.data() + data.size());
}

constexpr size_t salt_length = 16;

Botan::secure_vector<uint8_t> derive_key_material(std::string_view password,
                                                  std::span<const uint8_t> salt,
                                                  size_t output_length) {
   // Here, we use statically defined password hash parameters. Alternatively
   // you could use Botan::PasswordHashFamily::tune() to automatically select
   // parameters based on your desired runtime and memory usage.
   //
   // Defining those parameters highly depends on your use case and the
   // available compute and memory resources on your target platform.
   const std::string pbkdf_algo = "Argon2id";
   constexpr size_t M = 256 * 1024;  // kiB
   constexpr size_t t = 4;           // iterations
   constexpr size_t p = 2;           // parallelism

   auto pbkdf = Botan::PasswordHashFamily::create_or_throw(pbkdf_algo)->from_params(M, t,
↪ p);
   BOTAN_ASSERT_NONNULL(pbkdf);

   Botan::secure_vector<uint8_t> key(output_length);
   pbkdf->hash(key, password, salt);

   return key;
}

std::unique_ptr<Botan::AEAD_Mode> prepare_aead(std::string_view password,
                                               std::span<const uint8_t> salt,
                                               Botan::Cipher_Dir direction) {
   auto aead = Botan::AEAD_Mode::create_or_throw("AES-256/GCM", direction);

   const size_t key_length = aead->key_spec().maximum_keylength();
   const size_t nonce_length = aead->default_nonce_length();

   // Stretch the password into enough cryptographically strong key material
   // to initialize the AEAD with a key and nonce (aka. initialization vector).
   const auto keydata = derive_key_material(password, salt, key_length + nonce_length);
   BOTAN_ASSERT_NOMSG(keydata.size() == key_length + nonce_length);
   const auto key = std::span{keydata}.first(key_length);
   const auto nonce = std::span{keydata}.last(nonce_length);

   aead->set_key(key);
```

(continues on next page)

(continued from previous page)

```
   aead->start(nonce);

   return aead;
}

/**
 * Encrypts the data in @p plaintext using the given @p password.
 *
 * To resist offline brute-force attacks we stretch the password into key
 * material using a password-based key derivation function (PBKDF). The key
 * material is then used to initialize an AEAD for encryption and authentication
 * of the plaintext. This ensures that on-one can read or manipulate the data
 * without knowledge of the password.
 */
std::vector<uint8_t> encrypt_by_password(std::string_view password,
                                         Botan::RandomNumberGenerator& rng,
                                         std::span<const uint8_t> plaintext) {
   const auto kdf_salt = rng.random_vec(salt_length);
   auto aead = prepare_aead(password, kdf_salt, Botan::Cipher_Dir::Encryption);

   Botan::secure_vector<uint8_t> out(plaintext.begin(), plaintext.end());
   aead->finish(out);

   // The random salt used by the key derivation function is not secret and is
   // therefore prepended to the ciphertext.
   return concat(kdf_salt, out);
}

/**
 * Decrypts the output of `encrypt_by_password` given the correct @p password
 * or throws an exception if decryption is not possible.
 */
Botan::secure_vector<uint8_t> decrypt_by_password(std::string_view password, std::span
↪<const uint8_t> wrapped_data) {
   if(wrapped_data.size() < salt_length) {
      throw std::runtime_error("Encrypted data is too short");
   }

   const auto kdf_salt = wrapped_data.first(salt_length);
   auto aead = prepare_aead(password, kdf_salt, Botan::Cipher_Dir::Decryption);

   const auto ciphertext = wrapped_data.subspan(salt_length);
   Botan::secure_vector<uint8_t> out(ciphertext.begin(), ciphertext.end());

   try {
      aead->finish(out);
   } catch(const Botan::Invalid_Authentication_Tag&) {
      throw std::runtime_error("Failed to decrypt, wrong password?");
   }

   return out;
}
```

(continues on next page)

(continued from previous page)

```
} // namespace

int main() {
   Botan::AutoSeeded_RNG rng;

   // Note: For simplicity we omit the authentication of any associated data.
   //       If your use case would benefit from it, you should add it. Perhaps
   //       to both the password hashing and the AEAD.
   const std::string password = "geheimnis";
   const std::string message = "Attack at dawn!";

   try {
      const auto ciphertext = encrypt_by_password(password, rng, as<Botan::secure_vector
↪<uint8_t>>(message));
      std::cout << "Ciphertext: " << Botan::hex_encode(ciphertext) << "\n";

      const auto decrypted_message = decrypt_by_password(password, ciphertext);
      BOTAN_ASSERT_NOMSG(message.size() == decrypted_message.size() &&
                         std::equal(message.begin(), message.end(), decrypted_message.
↪begin()));

      std::cout << "Decrypted message: " << as<std::string>(decrypted_message) << "\n";
   } catch(const std::exception& ex) {
      std::cerr << "Something went wrong: " << ex.what() << "\n";
      return 1;
   }

   return 0;
}
```

8.16.3 Available Schemes

General Recommendations

If you need wide interoperability use PBKDF2 with HMAC-SHA256 and at least 50K iterations. If you don't, use Argon2id with p=1, t=3 and M as large as you can reasonably set (say 1 gigabyte).

You can test how long a particular PBKDF takes to execute using the cli tool `pbkdf_tune`:

```
$ ./botan pbkdf_tune --algo=Argon2id 500 --max-mem=192 --check
For 500 ms selected Argon2id(196608,3,1) using 192 MiB took 413.159 msec to compute
```

This returns the parameters chosen by the fast auto-tuning algorithm, and because `--check` was supplied the hash is also executed with the full set of parameters and timed.

PBKDF2

PBKDF2 is the "standard" password derivation scheme, widely implemented in many different libraries. It uses HMAC internally and requires choosing a hash function to use. (If in doubt use SHA-256 or SHA-512). It also requires choosing an iteration count, which makes brute force attacks more expensive. Use *at least* 50000 and preferably much more. Using 250,000 would not be unreasonable.

Algorithm specification name: `PBKDF2(<MessageAuthenticationCode|HashFunction>)`, e.g. `PBKDF2(HMAC(SHA-256))`

If a `HashFunction` is provided as an argument, it will create `HMAC(HashFunction)` as the `MessageAuthenticationCode`. I.e. `PBKDF2(SHA-256)` will result in `PBKDF2(HMAC(SHA-256))`.

Scrypt

Added in version 2.7.0.

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 has three parameters, usually termed `N`, `r`, and `p`. `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 1 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

Algorithm specification name: `Scrypt`

Argon2

Added in version 2.11.0.

Argon2 is the winner of the PHC (Password Hashing Competition) and provides a tunable memory hard PBKDF. There are three minor variants of Argon2 - Argon2d, Argon2i, and Argon2id. All three are implemented.

Algorithm specification names:

- `Argon2d`
- `Argon2i`
- `Argon2id`

Bcrypt

Added in version 2.11.0.

Bcrypt-PBKDF is a variant of the well known `bcrypt` password hashing function. Like `bcrypt` it is based around using Blowfish for the key expansion, which requires 4 KiB of fast random access memory, making hardware based attacks more expensive. Unlike Argon2 or Scrypt, the memory usage is not tunable.

This function is relatively obscure but is used for example in OpenSSH. Prefer Argon2 or Scrypt in new systems.

Algorithm specification name: `Bcrypt-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.

Algorithm specification name: `OpenPGP-S2K(<HashFunction>)`, e.g. `OpenPGP-S2K(SHA-384)`

8.16.4 PBKDF

PBKDF is the older API for this functionality, presented in header `pbkdf.h`. It only supports PBKDF2 and the PGP S2K algorithm, not Scrypt, Argon2, or bcrypt. This interface is deprecated and will be removed in a future major release.

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**

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

Return a newly created PBKDF object. The name should be in the format "PBKDF2(HASHNAME)", "PBKDF2(HMAC(HASHNAME))", or "OpenPGP-S2K". Returns null if the algorithm is not available.

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.

8.17 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.

8.17.1 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.

8.18 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).

8.18.1 Argon2

Added 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.

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.

8.18.2 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.

8.18.3 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`

8.19 Cryptobox

8.19.1 Encryption using a passphrase

Added in version 1.8.6.

Deprecated since version 3.0.

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.

8.20 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).

8.21 PSK Database

Added 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)`.

8.22 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, Cipher_Dir::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));

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));
```

(continues on next page)

(continued from previous page)

```
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, Cipher_Dir::Encryption),
               new Base64_Encoder);

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

You can read from a pipe while you are still writing to it, which allows you to bound the amount of memory that is in use at any one time. A common idiom for this is:

```
pipe.start_msg();
std::vector<uint8_t> buffer(4096); // arbitrary size
while(infile.good())
   {
   infile.read((char*)&buffer[0], buffer.size());
   const size_t got_from_infile = infile.gcount();
   pipe.write(buffer, got_from_infile);

   if(infile.eof())
      pipe.end_msg();

   while(pipe.remaining() > 0)
      {
      const size_t buffered = pipe.read(buffer, buffer.size());
      outfile.write((const char*)&buffer[0], buffered);
      }
   }
if(infile.bad() || (infile.fail() && !infile.eof()))
   throw Some_Exception();
```

8.22.1 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, Cipher_Dir::Decryption),
          new Fork(
             0, // this message gets plaintext
             new MAC_Filter("HMAC(SHA-1)", mac_key)
          )
   );

pipe.process_msg(ciphertext);
std::string plaintext = pipe.read_all_as_string(0);
secure_vector<uint8_t> mac = pipe.read_all(1);

if(mac != auth_code)
   error();
```

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

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

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

And then called `start_msg`, inserted some data, then `end_msg`. Then `pipe` would contain two messages. The first one (message number 0) would contain the SHA-256 sum of the input in hex encoded form, and the other would contain the SHA-512 sum of the input in raw binary. In many situations you'll want to perform a sequence of operations on multiple branches of the fork; in which case, use the filter described in *Chain*.

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

8.22.2 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")
              )
          );
```

8.22.3 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.

8.22.4 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.

8.22.5 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(
```

(continues on next page)

(continued from previous page)

```
            0, // Read the section 'Fork' to understand this
            new Chain(
               hmac = new MAC_Filter("HMAC(SHA-1)", mac_key, 12),
               new Base64_Encoder
               )
            )
   );
pipe.start_msg();
// use pipe for a while, decrypt some stuff, derive new keys and IVs
pipe.end_msg();

aes->set_key(aes_key2);
aes->set_iv(iv2);
hmac->set_key(mac_key2);

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

There are some requirements to using `Keyed_Filter` that you must follow. If you call `set_key` or `set_iv` on a filter that is owned by a `Pipe`, you must do so while the `Pipe` is "unlocked". This refers to the times when no messages are being processed by `Pipe` – either before `Pipe`'s `start_msg` is called, or after `end_msg` is called (and no new call to `start_msg` has happened yet). Doing otherwise will result in undefined behavior, probably silently getting invalid output.

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

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 `Cipher_Dir::Encryption` or `Cipher_Dir::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`.

8.22.6 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.

8.23 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`:

Added 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)
*/
```

(continues on next page)

(continued from previous page)

```
#include "cli.h"
#include <botan/hex.h>

#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::SymmetricKey& 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::SymmetricKey& 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);
}

}  // namespace

class CC_Encrypt final : public Command {
   public:
      CC_Encrypt() : Command("cc_encrypt CC passphrase --tweak=") {}

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

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

      void go() override {
```

(continues on next page)

(continued from previous page)

```
         const uint64_t cc_number = std::stoull(get_arg("CC"));
         const std::vector<uint8_t> tweak = Botan::hex_decode(get_arg("tweak"));
         const std::string pass = get_arg("passphrase");

         auto pbkdf = Botan::PBKDF::create("PBKDF2(SHA-256)");
         if(!pbkdf) {
            throw CLI_Error_Unsupported("PBKDF", "PBKDF2(SHA-256)");
         }

         auto key = Botan::SymmetricKey(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");

         auto pbkdf = Botan::PBKDF::create("PBKDF2(SHA-256)");
         if(!pbkdf) {
            throw CLI_Error_Unsupported("PBKDF", "PBKDF2(SHA-256)");
         }

         auto key = Botan::SymmetricKey(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);

}  // namespace Botan_CLI

#endif  // FPE && PBKDF
```

8.24 Threshold Secret Sharing

Added 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

8.25 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::**from_OID**(const OID &oid)

Initialize an `EC_Group` using an OID referencing the curve parameters.

EC_Group::**from_name**(std::string_view name)

Initialize an `EC_Group` using a name (such as "secp256r1")

EC_Group(const OID &oid, const *BigInt* &p, const *BigInt* &a, const *BigInt* &b, const *BigInt* &base_x, const *BigInt* &base_y, const *BigInt* &order)

Create an application specific elliptic curve.

This constructor imposes the following restrictions:

- The prime must be between 128 and 512 bits, and a multiple of 32 bits.
- As a special extension regarding the above restriction, the prime may alternately be 521 bits, in which case it must be exactly 2**521-1
- The prime must be congruent to 3 modulo 4
- The group order must have identical bitlength to the prime
- No cofactor is allowed
- An object identifier must be specified

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())

This is a deprecated alternative interface for creating application specific elliptic curves.

This does not impose the same restrictions regarding use of arbitrary sized groups, use of a cofactor, etc, and the object identifier is optional.

> **Warning:** If you are using this constructor, and cannot use the non-deprecated constructor due to the restrictions it places on the curve parameters, be aware that this constructor will be dropped in Botan 4. Please open an issue on Github describing your usecase.

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).

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 *EC_Point* &**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.

> **Warning:** In a future release all support for elliptic curves group with a cofactor > 1 will be removed.

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 *EC_Point* &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.

EC_Point **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.

EC_Point **point_multiply**(const *BigInt* &x, const *EC_Point* &pt, const *BigInt* &y) const

Multi-exponentiation. Returns base_point*x + pt*y. Not constant time. (Ordinarily used for signature verification.)

EC_Point **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.

EC_Point **blinded_var_point_multiply**(const *EC_Point* &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).

EC_Point **zero_point**() const

Return the zero point (aka the point at infinity).

EC_Point **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 **EC_Point**

Stores elliptic curve points in Jacobian representation.

std::vector<uint8_t> **encode**(*EC_Point*::Compression_Type format) const

Encode a point in a way that can later be decoded with *EC_Group::OS2ECP*.

EC_Point &**operator+=**(const *EC_Point* &rhs)

Point addition.

EC_Point &**operator-=**(const *EC_Point* &rhs)

Point subtraction.

EC_Point &**operator*=**(const *BigInt* &scalar)

Point multiplication using Montgomery ladder.

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

EC_Point &**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<*EC_Point*> &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 *EC_Point* &other) const

Point equality. This compares the affine representations.

8.26 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::create` and `Decompression_Algorithm::create` which both accept a string argument which can take values 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.

Two older functions for this are

Compression_Algorithm ***make_compressor**(std::string type)*

Decompression_Algorithm ***make_decompressor**(std::string type)*

which call the relevant `create` function and then `release` the returned `unique_ptr`. Avoid these in new code.

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.

8.27 External Providers

Botan ships with a variety of cryptographic algorithms in both pure software as well as with support from *hardware acceleration*.

Additionally, Botan allows to use external implementations to provide algorithms ("providers").

8.27.1 Integrated Providers

PKCS#11

PKCS#11 is a standard API for accessing cryptographic hardware. Botan ships a *PKCS#11 provider* for interacting with PKCS#11 devices which provide cryptographic algorithms. It is enabled by default.

TPM 1.2

The TPM 1.2 standard is a specification for a hardware device which provides cryptographic algorithms. Botan ships a *TPM provider* for interacting with TPM devices. It is disabled by default.

CommonCrypto

CommonCrypto is a library provided by Apple for accessing cryptographic algorithms. Botan ships a *CommonCrypto* provider for interacting with CommonCrypto. It is disabled by default.

The CommonCrypto provider supports the following algorithms:

- SHA-1, SHA-256, SHA-384, SHA-512
- AES-128, AES-192, AES-256, DES, TDES, Blowfish, CAST-128
- CBC, CTR, OFB

8.27.2 Provider Interfaces

Symmetric Algorithms

The following interfaces can be used to implement providers for symmetric algorithms:

- `AEAD_Mode`
- `BlockCipher`
- `Cipher_Mode`
- `Hash`
- `KDF`
- `MAC`
- `PasswordHashFamily`
- `PBKDF`
- `StreamCipher`
- `XOF`

Each of the interfaces provide a factory method which takes string arguments and returns an object implementing the interface. The strings are the name of the algorithm to be instantiated and the provider to be used. For example, the following code creates a SHA-256 hash object using the *CommonCrypto* provider:

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

auto hash = Botan::HashFunction::create_or_throw("SHA-256", "CommonCrypto");

hash->update("Hello");
hash->update(" ");
hash->update("World");
auto digest = hash->final();

// query the provider currently used
std::string provider = hash->provider(); // "CommonCrypto"
```

Omitting the provider string or leaving it empty means the default provider is used. The default provider is the first provider which supports the requested algorithm. Depending on how Botan was configured at build time, the default provider may be a pure software implementation, a hardware accelerated implementation or an implementation using an integrated provider, e.g., CommonCrypto.

The following rules apply:

1. If Botan was built with an integrated provider that is hooked into the `T::create()`/`T::create_or_throw()` factory methods (currently only *CommonCrypto* is), the default provider is the integrated provider.
2. If Botan was not built with an integrated provider as in (1), but with hardware acceleration support, e.g., AES-NI, and the hardware acceleration is available at runtime, the default provider is the hardware accelerated provider.
3. If Botan was not built with an integrated provider as in (1) and not built with hardware acceleration support, the default provider is the pure software implementation.

Regardless of the default provider, a specific provider can always be requested by passing the provider name as the second argument to `T::create()`/`T::create_or_throw()`. Specifically, the special provider name `"base"` can always be used to request the hardware accelerated (preferred, if available at runtime) or pure software implementation (last fallback).

Public Key Algorithms

The following interfaces support using providers for *public key algorithms*. The interfaces are used in a similar way as the interfaces for symmetric algorithms described above.

- `PK_Signer`
- `PK_Verifier`
- `PK_Key_Agreement`
- `PK_Encryptor_EME`
- `PK_Decryptor_EME`
- `PK_KEM_Encryptor`
- `PK_KEM_Decryptor`

Each of the interfaces provides a constructor which takes a key object, optional parameters, and a string specifying the provider to be used. For example, the following code signs a message using an RSA key with the *CommonCrypto* provider:

Note: No integrated provider currently supports using any public key algorithm in the way described above, so the example is purely for illustrative purposes.

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

Botan::AutoSeeded_RNG rng;
auto key = Botan::create_private_key("RSA", rng, "3072");

Botan::PK_Signer signer(key, rng, "EMSA3(SHA-256)", Botan::Signature_Format::Standard,
↪"CommonCrypto");

signer.update("Hello");
signer.update(" ");
signer.update("World");
auto signature = signer.signature(rng);
```

To create a key object, use `Botan::create_private_key()`, which takes a string specifying the algorithm and the provider to be used. For example, to create a 3072 bit RSA key with the *CommonCrypto* provider:

Note: No integrated provider currently supports creating any private key in the way described above, so the example is purely for illustrative purposes.

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

Botan::AutoSeeded_RNG rng;

auto key = Botan::create_private_key("RSA", rng, "3072", "CommonCrypto");
```

Another way to implement a provider for public key algorithms is to implement the `Private_Key` and `Public_Key` interfaces. This allows for different use cases, e.g., to use a key stored in a hardware security module, handled by a different operating system process (to avoid leaking the key material), or even implement an algorithm not supported by Botan. The resulting key class can be stored outside Botan and still be used with the `PK_Signer`, `PK_Verifier`, `PK_Key_Agreement`, `PK_Encryptor_EME`, `PK_Decryptor_EME`, `PK_KEM_Encryptor`, and `PK_KEM_Decryptor` interfaces.

8.28 PKCS#11

Added 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`

8.28.1 Low Level API

The PKCS#11 standards committee provides header files (`pkcs11.h`, `pkcs11f.h` and `pkcs11t.h`) which define the PKCS#11 API in the C programming language. These header files could be used directly to access PKCS#11 compatible smart cards or HSMs. The external header files are shipped with Botan in version v2.4 of the standard. The PKCS#11 low level API wraps the original PKCS#11 API, but still allows to access all functions described in the standard and has the advantage that it is a C++ interface with features like RAII, exceptions and automatic memory management.

The low level API is implemented by the *LowLevel* class and can be accessed by including the header `botan/p11.h`.

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 accessed by the `Module::operator->()` method.

Code Example:

```
#include <botan/p11.h>
#include <botan/p11_types.h>

#include <vector>

int main() {
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");

   // C_Initialize is automatically called by the constructor of the Module

   // work with the token

   std::vector<Botan::PKCS11::SlotId> slot_ids;
   [[maybe_unused]] bool success = module->C_GetSlotList(true, slot_ids);

   // C_Finalize is automatically called by the destructor of the Module

   return 0;
}
```

8.28.2 High Level API

The high level API provides access to the most commonly used PKCS#11 functionality in an object oriented manner. Functionality of the high level API includes:

- Loading/unloading of PKCS#11 modules
- Initialization of tokens
- Change of PIN/SO-PIN
- Session management
- Random number generation

- Enumeration of objects on the token (certificates, public keys, private keys)
- Import/export/deletion of certificates
- Generation/import/export/deletion of RSA and EC public and private keys
- Encryption/decryption using RSA with support for OAEP and PKCS1-v1_5 (and raw)
- Signature generation/verification using RSA with support for PSS and PKCS1-v1_5 (and raw)
- Signature generation/verification using ECDSA
- Key derivation using ECDH

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:

```
#include <botan/p11.h>
#include <botan/p11_types.h>

#include <iostream>
#include <string>

int main() {
   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();
```

(continues on next page)

(continued from previous page)

```
   // print library version
   std::cout << std::to_string(info.libraryVersion.major) << "." << std::to_string(info.
↪libraryVersion.minor) << '\n';

   return 0;
}
```

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:

```
#include <botan/p11.h>
#include <botan/p11_types.h>

#include <iostream>
#include <string>
#include <vector>

int main() {
```

(continues on next page)

(continued from previous page)

```
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");

   // only slots with connected token
   std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_available_
↪slots(module, true);

   // use first slot
   Botan::PKCS11::Slot slot(module, slots.at(0));

   // print firmware version of the slot
   Botan::PKCS11::SlotInfo slot_info = slot.get_slot_info();
   std::cout << std::to_string(slot_info.firmwareVersion.major) << "."
             << std::to_string(slot_info.firmwareVersion.minor) << '\n';

   // 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) << '\n';

   // 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 << '\n';

   // initialize the token
   Botan::PKCS11::secure_string so_pin(8, '0');
   slot.initialize("Botan PKCS11 documentation test label", so_pin);

   return 0;
}
```

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.

The session is passed to most other PKCS#11 operations, and must remain alive as long as any other PKCS#11 object which the session 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 **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:

```
#include <botan/p11.h>
#include <botan/p11_types.h>

#include <iostream>
#include <vector>

int main() {
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");
   // use first slot with connected token
   std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_available_
↪slots(module, true);
   Botan::PKCS11::Slot slot(module, slots.at(0));

   // 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);
```

(continues on next page)

(continued from previous page)

```
   }

   // 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 << '\n';

   // login
   Botan::PKCS11::secure_string pin = {'1', '2', '3', '4', '5', '6'};
   session.login(Botan::PKCS11::UserType::User, pin);

   // set pin
   Botan::PKCS11::secure_string new_pin = {'6', '5', '4', '3', '2', '1'};
   session.set_pin(pin, new_pin);

   // logoff
   session.logoff();

   // log in as security officer
   Botan::PKCS11::secure_string so_pin = {'0', '0', '0', '0', '0', '0', '0', '0'};
   session.login(Botan::PKCS11::UserType::SO, so_pin);

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

   return 0;
}
```

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::**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::**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::**Object**(*Session* &session, ObjectHandle handle)

Takes ownership over an existing object.

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::**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:

```
#include <botan/der_enc.h>
#include <botan/p11.h>
#include <botan/p11_object.h>
#include <botan/p11_types.h>
#include <botan/secmem.h>

#include <cstddef>
#include <string>
#include <vector>

int main() {
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");
   // open write session to first slot with connected token
   std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_available_
↪slots(module, true);
   Botan::PKCS11::Slot slot(module, slots.at(0));
   Botan::PKCS11::Session session(slot, false);

   // 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);
   std::vector<uint8_t> encoded_id;
```

(continues on next page)

(continued from previous page)

```
   Botan::DER_Encoder(encoded_id).encode(id);
   data_obj_props.set_object_id(encoded_id);

   // create the object
   Botan::PKCS11::Object data_obj(session, data_obj_props);

   // get label of this object
   Botan::PKCS11::secure_string retrieved_label = data_obj.get_attribute_
↪value(Botan::PKCS11::AttributeType::Label);

   // set a new label
   Botan::PKCS11::secure_string new_label = {'B', 'o', 't', 'a', 'n'};
   data_obj.set_attribute_value(Botan::PKCS11::AttributeType::Label, new_label);

   // copy the object
   Botan::PKCS11::AttributeContainer copy_attributes;
   copy_attributes.add_string(Botan::PKCS11::AttributeType::Label, "copied object");
   [[maybe_unused]] 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();

   return 0;
}
```

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:

```
#include <botan/auto_rng.h>
#include <botan/p11.h>
#include <botan/p11_rsa.h>
#include <botan/p11_types.h>
#include <botan/pubkey.h>
#include <botan/rsa.h>

#include <vector>

int main() {
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");
   // open write session to first slot with connected token
   std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_available_
↪slots(module, true);
   Botan::PKCS11::Slot slot(module, slots.at(0));
   Botan::PKCS11::Session session(slot, false);

   Botan::PKCS11::secure_string pin = {'1', '2', '3', '4', '5', '6'};
   session.login(Botan::PKCS11::UserType::User, pin);
```

(continues on next page)

(continued from previous page)

```
   /************* import RSA private key *************/

   // create private key in software
   Botan::AutoSeeded_RNG rng;
   Botan::RSA_PrivateKey priv_key_sw(rng, 2048);

   // set the private key import properties
   Botan::PKCS11::RSA_PrivateKeyImportProperties priv_import_props(priv_key_sw.get_n(),␣
↪priv_key_sw.get_d());

   priv_import_props.set_pub_exponent(priv_key_sw.get_e());
   priv_import_props.set_prime_1(priv_key_sw.get_p());
   priv_import_props.set_prime_2(priv_key_sw.get_q());
   priv_import_props.set_coefficient(priv_key_sw.get_c());
   priv_import_props.set_exponent_1(priv_key_sw.get_d1());
   priv_import_props.set_exponent_2(priv_key_sw.get_d2());

   priv_import_props.set_token(true);
   priv_import_props.set_private(true);
   priv_import_props.set_decrypt(true);
   priv_import_props.set_sign(true);

   // import
   Botan::PKCS11::PKCS11_RSA_PrivateKey priv_key(session, priv_import_props);

   /************* export PKCS#11 RSA private key *************/
   Botan::RSA_PrivateKey exported = priv_key.export_key();

   /************* import RSA public key *************/

   // set the public key import properties
   Botan::PKCS11::RSA_PublicKeyImportProperties pub_import_props(priv_key.get_n(), priv_
↪key.get_e());
   pub_import_props.set_token(true);
   pub_import_props.set_encrypt(true);
   pub_import_props.set_private(false);

   // 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);
```

(continues on next page)

(continued from previous page)

```
   /************* 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::Signature_
↪Format::Standard);
   auto signature = signer.sign_message(plaintext, rng);

   /************* RSA verify *************/

   Botan::PK_Verifier verifier(rsa_keypair.first, "EMSA4(SHA-256)", Botan::Signature_
↪Format::Standard);
   auto ok = verifier.verify_message(plaintext, signature);

   return ok ? 0 : 1;
}
```

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:

```
#include <botan/asn1_obj.h>
#include <botan/auto_rng.h>
#include <botan/der_enc.h>
#include <botan/ec_group.h>
#include <botan/ecdsa.h>
#include <botan/p11.h>
#include <botan/p11_ecc_key.h>
#include <botan/p11_ecdsa.h>
#include <botan/p11_types.h>
#include <botan/pubkey.h>

#include <string>
#include <vector>

int main() {
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");
   // open write session to first slot with connected token
   std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_available_
↪slots(module, true);
   Botan::PKCS11::Slot slot(module, slots.at(0));
   Botan::PKCS11::Session session(slot, false);

   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::from_name("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();
```

(continues on next page)

(continued from previous page)

```
   /************* import ECDSA public key *************/

   // import to card
   std::vector<uint8_t> ec_point;
   Botan::DER_Encoder(ec_point).encode(priv_key_sw.public_point().encode(Botan::EC_Point_
↪Format::Uncompressed),
                                       Botan::ASN1_Type::OctetString);
   Botan::PKCS11::EC_PublicKeyImportProperties pub_import_props(priv_key_sw.DER_domain(),
↪ ec_point);

   pub_import_props.set_token(true);
   pub_import_props.set_verify(true);
   pub_import_props.set_private(false);

   // label
   label = "test ECDSA pub key";
   pub_import_props.set_label(label);

   Botan::PKCS11::PKCS11_ECDSA_PublicKey public_key(session, pub_import_props);

   /************* export PKCS#11 ECDSA public key *************/
   Botan::ECDSA_PublicKey pub_exported = public_key.export_key();

   /************* 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::from_name("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::from_name("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);
```

(continues on next page)

(continued from previous page)

```
   /************* PKCS#11 ECDSA sign and verify *************/

   std::vector<uint8_t> plaintext(20, 0x01);

   Botan::PK_Signer signer(key_pair.second, rng, "Raw", Botan::Signature_
↪Format::Standard, "pkcs11");
   auto signature = signer.sign_message(plaintext, rng);

   Botan::PK_Verifier token_verifier(key_pair.first, "Raw", Botan::Signature_
↪Format::Standard, "pkcs11");
   bool ecdsa_ok = token_verifier.verify_message(plaintext, signature);

   return ecdsa_ok ? 0 : 1;
}
```

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_Key

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:

```
#include <botan/asn1_obj.h>
#include <botan/auto_rng.h>
#include <botan/der_enc.h>
#include <botan/ec_group.h>
#include <botan/ecdh.h>
#include <botan/p11.h>
#include <botan/p11_ecc_key.h>
#include <botan/p11_ecdh.h>
#include <botan/p11_types.h>
#include <botan/pubkey.h>
#include <botan/symkey.h>

#include <string>
#include <vector>

int main() {
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");
   // open write session to first slot with connected token
   std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_available_
↪slots(module, true);
   Botan::PKCS11::Slot slot(module, slots.at(0));
   Botan::PKCS11::Session session(slot, false);

   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::from_name("secp256r1"));
   priv_key_sw.set_parameter_encoding(Botan::EC_Group_Encoding::EC_DOMPAR_ENC_OID);

   // set import properties
```

(continues on next page)

(continued from previous page)

```
   Botan::PKCS11::EC_PrivateKeyImportProperties priv_import_props(priv_key_sw.DER_
↪domain(),
                                                                  priv_key_sw.private_
↪value());

   priv_import_props.set_token(true);
   priv_import_props.set_private(true);
   priv_import_props.set_derive(true);
   priv_import_props.set_extractable(true);

   // label
   std::string label = "test ECDH key";
   priv_import_props.set_label(label);

   // import to card
   Botan::PKCS11::PKCS11_ECDH_PrivateKey priv_key(session, priv_import_props);

   /************* export ECDH private key *************/
   Botan::ECDH_PrivateKey exported = priv_key.export_key();

   /************* import ECDH public key *************/

   // set import properties
   std::vector<uint8_t> ec_point;
   Botan::DER_Encoder(ec_point).encode(priv_key_sw.public_point().encode(Botan::EC_Point_
↪Format::Uncompressed),
                                       Botan::ASN1_Type::OctetString);
   Botan::PKCS11::EC_PublicKeyImportProperties pub_import_props(priv_key_sw.DER_domain(),
↪ ec_point);

   pub_import_props.set_token(true);
   pub_import_props.set_private(false);
   pub_import_props.set_derive(true);

   // label
   label = "test ECDH pub key";
   pub_import_props.set_label(label);

   // import
   Botan::PKCS11::PKCS11_ECDH_PublicKey pub_key(session, pub_import_props);

   /************* export ECDH private key *************/
   Botan::ECDH_PublicKey exported_pub = pub_key.export_key();

   /************* generate ECDH private key *************/

   Botan::PKCS11::EC_PrivateKeyGenerationProperties priv_generate_props;
   priv_generate_props.set_token(true);
   priv_generate_props.set_private(true);
   priv_generate_props.set_derive(true);

   Botan::PKCS11::PKCS11_ECDH_PrivateKey priv_key2(
```

(continues on next page)

(continued from previous page)

```
      session,
      Botan::EC_Group::from_name("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::from_name("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, key_pair_other.first.public_point().encode(Botan::EC_Point_
↪Format::Uncompressed));

   Botan::SymmetricKey bob_key =
      kb.derive_key(32, key_pair.first.public_point().encode(Botan::EC_Point_
↪Format::Uncompressed));

   bool eq = alice_key == bob_key;

   return eq ? 0 : 1;
}
```

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:

```
#include <botan/auto_rng.h>
#include <botan/hmac_drbg.h>
#include <botan/mac.h>
#include <botan/p11.h>
#include <botan/p11_randomgenerator.h>
#include <botan/p11_types.h>

#include <vector>

int main() {
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");
   // open write session to first slot with connected token
   std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_available_
↪slots(module, true);
   Botan::PKCS11::Slot slot(module, slots.at(0));
   Botan::PKCS11::Session session(slot, false);

   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());

   return 0;
}
```

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:

```
#include <botan/p11.h>
#include <botan/p11_types.h>

#include <vector>

int main() {
   /************* set pin *************/

   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");

   // 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
```

(continues on next page)

(continued from previous page)

```
   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);

   return 0;
}
```

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:

```
#include <botan/p11.h>
#include <botan/p11_types.h>
#include <botan/p11_x509.h>
#include <botan/pkix_types.h>
#include <botan/x509cert.h>

#include <vector>

int main() {
   Botan::PKCS11::Module module("C:\\pkcs11-middleware\\library.dll");
   // open write session to first slot with connected token
   std::vector<Botan::PKCS11::SlotId> slots = Botan::PKCS11::Slot::get_available_
↪slots(module, true);
   Botan::PKCS11::Slot slot(module, slots.at(0));
   Botan::PKCS11::Session session(slot, false);

   // load existing certificate
   Botan::X509_Certificate root("test.crt");

   // set props
   Botan::PKCS11::X509_CertificateProperties props(root.subject_dn().DER_encode(), root.
↪BER_encode());

   props.set_label("Botan PKCS#11 test certificate");
```

(continues on next page)

(continued from previous page)

```
   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());

   return 0;
}
```

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	

Error descriptions

[50] Failing operations for CardOS 4.4:

- object_copyPage 230, 20
- rsa_privkey_exportPage 230, 21
- rsa_generate_private_keyPage 230, 22
- rsa_sign_verifyPage 230, 23
- ecdh_privkey_importPage 230, 3
- ecdh_privkey_exportPage 230, 2
- ecdh_pubkey_importPage 230, 4
- ecdh_pubkey_exportPage 230, 4
- ecdh_generate_private_keyPage 230, 3
- ecdh_generate_keypairPage 230, 3
- ecdh_derivePage 230, 3
- ecdsa_privkey_importPage 230, 3
- ecdsa_privkey_exportPage 230, 2
- ecdsa_pubkey_importPage 230, 4
- ecdsa_pubkey_exportPage 230, 4
- ecdsa_generate_private_keyPage 230, 3
- ecdsa_generate_keypairPage 230, 3
- ecdsa_sign_verifyPage 230, 3
- rng_add_entropyPage 230, 5

8.29 Trusted Platform Module (TPM)

Added in version 1.11.26.

Some computers come with a TPM, which is a small side processor which can perform certain operations which include RSA key generation and signing, a random number generator, accessing a small amount of NVRAM, and a set of PCRs which can be used to measure software state (this is TPMs most famous use, for authenticating a boot sequence).

The TPM NVRAM and PCR APIs are not supported by Botan at this time, patches welcome.

Currently only v1.2 TPMs are supported, and the only TPM library supported is TrouSerS (http://trousers.sourceforge.net/). Hopefully both of these limitations will be removed in a future release, in order to support newer TPM v2.0 systems. The current code has been tested with an ST TPM running in a Lenovo laptop.

Test for TPM support with the macro `BOTAN_HAS_TPM`, include `<botan/tpm.h>`.

First, create a connection to the TPM with a `TPM_Context`. The context is passed to all other TPM operations, and should remain alive as long as any other TPM object which the context was passed to is still alive, otherwise errors or even an application crash are possible. In the future, the API may change to using `shared_ptr` to remove this problem.

class **TPM_Context**

TPM_Context(pin_cb cb, const char *srk_password)

The (somewhat improperly named) pin_cb callback type takes a std::string as an argument, which is an informative message for the user. It should return a string containing the password entered by the user.

Normally the SRK password is null. Use nullptr to signal this.

The TPM contains a RNG of unknown design or quality. If that doesn't scare you off, you can use it with `TPM_RNG` which implements the standard `RandomNumberGenerator` interface.

class **TPM_RNG**

---

[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_copyPage 230, 20
- rsa_privkey_export[21]
- rsa_generate_private_key[22]
- rsa_sign_verify[23]
- ecdh_privkey_export[2]
- ecdh_pubkey_import[4]
- ecdh_generate_private_key[32]
- ecdh_generate_keypair[3]
- ecdh_derive[33]
- ecdsa_privkey_export[2]
- ecdsa_generate_private_key[30]
- ecdsa_generate_keypair[30]
- ecdsa_sign_verify[30]
- rng_add_entropy[5]

[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[20]
- rsa_privkey_export[21]
- rsa_generate_private_key[22]
- rsa_sign_verify[23]
- ecdh_privkey_export[2]

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).

---

[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[2]
- ecdh_generate_private_key[35]
- ecdsa_privkey_export[Page 230, 2]
- ecdsa_generate_private_key[36]
- c_copy_object[Page 230, 4]
- object_copy[Page 230, 4]
- rng_add_entropy[Page 230, 5]
- rsa_sign_verify[Page 230, 3]
- rsa_privkey_export[Page 230, 2]
- rsa_generate_private_key[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[Page 230, 2]
- session_info[Page 230, 2]
- set_pin[Page 230, 2]
- initialize[Page 230, 2]
- change_so_pin[Page 230, 2]
- object_copy[Page 230, 20]
- rsa_generate_private_key[7]
- rsa_encrypt_decrypt[8]
- rsa_sign_verify[Page 230, 2]
- rng_add_entropy[Page 230, 5]

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

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

8.30 One Time Passwords

Added 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.

8.30.1 HOTP

HOTP generates OTPs that are a short numeric sequence, between 6 and 8 digits (most applications use 6 digits), created using the HMAC of a 64-bit counter value. If the counter ever repeats the OTP will also repeat, thus both parties must assure the counter only increments and is never repeated or decremented. Thus both client and server must keep track of the next counter expected.

Anyone with access to the client-specific secret key can authenticate as that client, so it should be treated with the same security consideration as would be given to any other symmetric key or plaintext password.

class **HOTP**

Implement counter-based OTP

HOTP(const SymmetricKey &key, const std::string &hash_algo = "SHA-1", size_t digits = 6)

Initialize an HOTP instance with a secret key (specific to each client), a hash algorithm (must be SHA-1, SHA-256, or SHA-512), and the number of digits with each OTP (must be 6, 7, or 8).

In RFC 4226, HOTP is only defined with SHA-1, but many HOTP implementations support SHA-256 as an extension. The collision attacks on SHA-1 do not have any known effect on HOTP's security.

uint32_t **generate_hotp**(uint64_t counter)

Return the OTP 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.

8.30.2 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.

8.31 Roughtime

Added in version 2.13.0.

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

8.32 ZFEC Forward Error Correction

Added in version 3.0.0.

The `ZFEC` class provides forward error correction compatible with the zfec (https://github.com/tahoe-lafs/zfec) library.

Forward error correction takes an input and creates multiple "shares", such that any `K` of `N` shares is sufficient to recover the entire original input.

Note: Specific to the ZFEC format, the first `K` generated shares are identical to the original input data, followed by `N-K` shares of error correcting code. This is very different from threshold secret sharing, where having fewer than `K` shares gives no information about the original input.

Warning: If a corrupted share is provided to the decoding algorithm, the resulting decoding will be invalid. It is recommended to protect shares using a technique such as a MAC or public key signature, if corruption is likely in your application.

`ZFEC` requires that the input length be exactly divisible by `K`; if needed define a padding scheme to pad your input to the necessary size.

An example application that adds padding and a hash checksum is available in `src/cli/zfec.cpp` and invokable using `botan fec_encode` and `botan fec_decode`.

class **ZFEC**

ZFEC(size_t k, size_t n)

Set up for encoding or decoding using parameters `k` and `n`. Both must be less than 256, and `k` must be less than `n`.

void **encode_shares**(const std::vector<const uint8_t*> &shares, size_t share_size, std::function<void(size_t, const uint8_t[], size_t)> output_cb) const

Encode `K` shares in `shares` each of length `share_size` into `N` shares, also each of length `share_size`. The `output_cb` function will be called once for each output share (in some unspecified and possibly non-deterministic order).

The parameters to `output_cb` are: the share being output, the share contents, and the length of the encoded share (which will always be equal to `share_size`).

void **decode_shares**(const std::map<size_t, const uint8_t*> &shares, size_t share_size, std::function<void(size_t, const uint8_t[], size_t)> output_cb) const

Decode some set of shares into the original input. Each share is of `share_size` bytes. The shares are identified by a small integer (between 0 and 255).

The parameters to `output_cb` are similar to that of `encode_shares`.

8.33 FFI (C Binding)

Added in version 2.0.0.

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) as well as the implementations of the various language bindings (https://github.com/randombit/botan/wiki/Language-Bindings). At the time of this writing, the Python and Rust bindings are probably the most comprehensive.

8.33.1 Rules of Engagement

Writing language bindings for C or C++ libraries is typically a tedious and bug-prone experience. This FFI layer was designed to make the experience, if not pleasant, at least straighforward.

- All objects manipulated by the API are opaque structs. Each struct is tagged with a 32-bit magic number which is unique to its type; accidentally passing the wrong object type to a function will result in a *BOTAN_FFI_ERROR_INVALID_OBJECT* error, instead of a crash or memory corruption.
- (Almost) all functions return an integer error code indicating success or failure. The exception is a small handful of version query functions, which are guaranteed to never fail. All functions returning errors use the same set of error codes.
- The set of types used is small and commonly supported: `uint8_t` arrays for binary data, `size_t` for lengths, and NULL-terminated UTF-8 encoded strings.
- No ownership of pointers crosses the boundary. If the library is producing output, it does so by either writing to a buffer that was provided by the application, or calling a view callback.

  In the first case, the application typically passes both an output buffer and a pointer to a length field. On entry, the length field should be set to the number of bytes available in the output buffer. If there is sufficient room, the output is written to the buffer, the actual number of bytes written is returned in the length field, and the function returns 0 (success). Otherwise, the number of bytes required is placed in the length parameter, and then *BOTAN_FFI_ERROR_INSUFFICIENT_BUFFER_SPACE* is returned.

  In most cases, for this style of function, there is also a function which allows querying the actual (or possibly upper bound) number of bytes in the function's output. For example calling *botan_hash_output_length* allows the application to determine in advance the number of bytes that *botan_hash_final* will want to write.

  In some situations, it is not possible to determine exactly what the output size of the function will be in advance. Here the FFI layer uses what it terms *View Functions*; callbacks that are allowed to view the entire output of the function, but once the callback returns, no further access is allowed. View functions are called with an opaque pointer provided by the application, which allows passing arbitrary context information.

8.33.2 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_STRING_CONVERSION_ERROR** = -11

A string view function which attempts to convert a string to a specified charset, and fails, can use this function to indicate the error.

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_SYSTEM_ERROR** = -22

A system call failed

enumerator **BOTAN_FFI_ERROR_INTERNAL_ERROR** = -23

An internal bug was encountered (please open a ticket on github)

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 `foo_set_key`.

enumerator **BOTAN_FFI_ERROR_INVALID_OBJECT_STATE** = -35

An operation was invoked that makes sense for the object, but it is in the wrong state to perform it.

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 error.

enumerator **BOTAN_FFI_ERROR_UNKNOWN_ERROR** = -100

Something bad happened, but we are not sure why or how.

Error values below -10000 are reserved for the application (these can be returned from view functions).

Further information about the error that occured is available via

const char ***botan_error_last_exception_message**()

Added in version 3.0.0.

Returns a static string stored in a thread local variable which contains the last exception message thrown.

> **Warning:** This string buffer is overwritten on the next call to the FFI layer

8.33.3 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
20230403	3.0.0
20210220	2.18.0
20191214	2.13.0
20180713	2.8.0
20170815	2.3.0
20170327	2.1.0
20150515	2.0.0

8.33.4 View Functions

Added in version 3.0.0.

Starting in Botan 3.0, certain functions were added which produce a "view". That is instead of copying data to a user provided buffer, they instead invoke a callback, passing the data that was requested. This avoids an issue where in some cases it is not possible for the caller to know what the output length of the FFI function will be. In these cases, the best they can do is set a large length, invoke the function, and then accept that they may need to retry the (potentially expensive) operation.

View functions avoid this by always providing the full data, and allowing the caller to allocate memory as necessary to copy out the result, without having to guess the length in advance.

In all cases the pointer passed to the view function is deallocated after the view function returns, and should not be retained.

The view functions return an integer value; if they return non-zero, then the overall FFI function will also return this integer. To avoid confusion when mapping the errors, any error returns should either match Botan's FFI error codes, or else use an integer value in the application reserved range.

typedef void ***botan_view_ctx**

The application context, which is passed back to the view function.

typedef int (***botan_view_bin_fn**)(*botan_view_ctx* view_ctx, const uint8_t *data, size_t len)

A viewer of arbitrary binary data.

typedef int (***botan_view_str_fn**)(*botan_view_ctx* view_ctx, const char *str, size_t len)

A viewer of a null terminated C-style string. The length *includes* the null terminator byte. The string should be UTF-8 encoded, but in certain circumstances may not be. (Typically this would be due to a bug or oversight; please report the issue.) *BOTAN_FFI_ERROR_STRING_CONVERSION_ERROR* is reserved to allow the FFI call to indicate the problem, should it be unable to convert the data.

8.33.5 Utility Functions

int **botan_constant_time_compare**(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

8.33.6 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_init_custom**(*botan_rng_t* *rng, const char *rng_name, void *context, int (*get_cb)(void *context, uint8_t *out, size_t out_len), int (*add_entropy_cb)(void *context, const uint8_t input[], size_t length), void (*destroy_cb)(void *context));

Added in version 2.18.0.

Create a new custom RNG object, which will invoke the provided callbacks.

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*.

8.33.7 Block Ciphers

Added 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*.

8.33.8 Hash Functions

typedef opaque ***botan_hash_t**

An opaque data type for a hash. Don't mess with it.

int **botan_hash_init**(*botan_hash_t* hash, const char *hash_name, uint32_t flags)

Creates a hash of the given name, e.g., "SHA-384".

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.

int **botan_hash_output_length**(*botan_hash_t* hash, size_t *output_length)

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.

8.33.9 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.

int **botan_mac_init**(*botan_mac_t* *mac, const char *mac_name, uint32_t flags)

Creates a MAC of the given name, e.g., "HMAC(SHA-384)". 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.

int **botan_mac_output_length**(*botan_mac_t* mac, size_t *output_length)

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_set_nonce**(*botan_mac_t* mac, const uint8_t *key, size_t key_len)

Set a nonce for the MAC. This is used for certain (relatively uncommon) MACs such as GMAC

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.

8.33.10 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.

int **botan_cipher_init**(*botan_cipher_t* *cipher, 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)

int **botan_cipher_requires_entire_message**(*botan_cipher_t* cipher)

int **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).

int **botan_cipher_get_default_nonce_length**(*botan_cipher_t* cipher, size_t *nl)

Return the default nonce length

int **botan_cipher_get_update_granularity**(*botan_cipher_t* cipher, size_t *ug)

Return the minimum update granularity, ie the size of a buffer that must be passed to *botan_cipher_update*

int **botan_cipher_get_ideal_update_granularity**(*botan_cipher_t* cipher, size_t *ug)

Return the ideal update granularity, ie the size of a buffer that must be passed to *botan_cipher_update* that maximizes performance.

Note: Using larger buffers than the value returned here is unlikely to hurt (within reason). Typically the returned value is a small multiple of the minimum granularity, with the multiplier depending on the algorithm and hardware support.

---

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.

8.33.11 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(SHA-512)".

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(SHA-512)". 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*.

8.33.12 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*.

8.33.13 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`

8.33.14 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.

8.33.15 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_destroy**(*botan_privkey_t* key)

Destroy an object.

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 cryptosystem* 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_export**(*botan_privkey_t* key, uint8_t out[], size_t *out_len, uint32_t flags)

Export a private key. If flags is 1 then PEM format is used.

int **botan_privkey_view_encrypted_der**(*botan_privkey_t* key, *botan_rng_t* rng, const char *passphrase, const char *cipher_algo, const char *pbkdf_hash, size_t pbkdf_iterations, *botan_view_ctx* ctx, *botan_view_bin_fn* view)

View the encrypted DER private key. In this version the number of PKBDF2 iterations is specified.

Set cipher_algo and pbkdf_hash to NULL to select defaults.

int **botan_privkey_view_encrypted_der_timed**(*botan_privkey_t* key, *botan_rng_t* rng, const char *passphrase, const char *cipher_algo, const char *pbkdf_hash, size_t pbkdf_runtime_msec, *botan_view_ctx* ctx, *botan_view_bin_fn* view)

View the encrypted DER private key. In this version the desired PBKDF runtime is specified in milliseconds.

Set cipher_algo and pbkdf_hash to NULL to select defaults.

int **botan_privkey_view_encrypted_pem**(*botan_privkey_t* key, *botan_rng_t* rng, const char *passphrase, const char *cipher_algo, const char *pbkdf_hash, size_t pbkdf_iterations, *botan_view_ctx* ctx, *botan_view_str_fn* view)

View the encrypted PEM private key. In this version the number of PKBDF2 iterations is specified.

Set cipher_algo and pbkdf_hash to NULL to select defaults.

int **botan_privkey_view_encrypted_pem_timed**(*botan_privkey_t* key, *botan_rng_t* rng, const char *passphrase, const char *cipher_algo, const char *pbkdf_hash, size_t pbkdf_runtime_msec, *botan_view_ctx* ctx, *botan_view_str_fn* view)

View the encrypted PEM private key. In this version the desired PBKDF runtime is specified in milliseconds.

Set cipher_algo and pbkdf_hash to NULL to select defaults.

int **botan_privkey_view_der**(*botan_privkey_t* key, *botan_view_ctx* ctx, *botan_view_bin_fn* view)

View the unencrypted DER encoding of the private key

int **botan_privkey_view_pem**(*botan_privkey_t* key, *botan_view_ctx* ctx, *botan_view_str_fn* view)

View the unencrypted PEM encoding of the private key

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_view_der**(*botan_pubkey_t* key, *botan_view_ctx* ctx, *botan_view_bin_fn* view)

View the DER encoding of the public key

int **botan_pubkey_view_pem**(*botan_pubkey_t* key, *botan_view_ctx* ctx, *botan_view_str_fn* view)

View the PEM encoding of the public key

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.

8.33.16 RSA specific functions

Note: These functions are deprecated. Instead use *botan_privkey_get_field* and *botan_pubkey_get_field*.

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.

8.33.17 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.

8.33.18 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.

8.33.19 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.

8.33.20 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)

8.33.21 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)" for PKCS #1 v1.5 padding (used with RSA) or "SHA-384". Generally speaking only RSA has special padding modes; for other algorithms like ECDSA one just names the hash.

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.

8.33.22 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*.

8.33.23 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_view_public**(*botan_privkey_t* key, *botan_view_ctx* ctx, *botan_view_bin_fn* view)

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)

8.33.24 Public Key Encapsulation

Added in version 3.0.0.

typedef opaque ***botan_pk_op_kem_encrypt_t**

An opaque data type for a KEM operation. Don't mess with it.

int **botan_pk_op_kem_encrypt_create**(*botan_pk_op_kem_encrypt_t* *op, *botan_pubkey_t* key, const char *kdf)

Create a KEM operation, encrypt version

int **botan_pk_op_kem_encrypt_destroy**(*botan_pk_op_kem_encrypt_t* op)

Destroy the operation, freeing memory

int **botan_pk_op_kem_encrypt_shared_key_length**(*botan_pk_op_kem_encrypt_t* op, size_t desired_shared_key_length, size_t *output_shared_key_length)

Return the output shared key length, assuming *desired_shared_key_length* is provided.

Note: Normally this will just return *desired_shared_key_length* but may return a different value if a "raw" KDF is used (returning the unhashed output), or potentially depending on KDF limitations.

---

int **botan_pk_op_kem_encrypt_encapsulated_key_length**(*botan_pk_op_kem_encrypt_t* op, size_t *output_encapsulated_key_length)

Return the length of the encapsulated key

int **botan_pk_op_kem_encrypt_create_shared_key**(*botan_pk_op_kem_encrypt_t* op, *botan_rng_t* rng, const uint8_t salt[], size_t salt_len, size_t desired_shared_key_len, uint8_t shared_key[], size_t *shared_key_len, uint8_t encapsulated_key[], size_t *encapsulated_key_len)

Create a new encapsulated key. Use the length query functions beforehand to correctly size the output buffers, otherwise an error will be returned.

typedef opaque ***botan_pk_op_kem_decrypt_t**

An opaque data type for a KEM operation. Don't mess with it.

int **botan_pk_op_kem_decrypt_create**(*botan_pk_op_kem_decrypt_t* *op, *botan_pubkey_t* key, const char *kdf)

Create a KEM operation, decrypt version

int **botan_pk_op_kem_decrypt_shared_key_length**(*botan_pk_op_kem_decrypt_t* op, size_t desired_shared_key_length, size_t *output_shared_key_length)

See *botan_pk_op_kem_encrypt_shared_key_length*

int **botan_pk_op_kem_decrypt_shared_key**(*botan_pk_op_kem_decrypt_t* op, const uint8_t salt[], size_t salt_len, const uint8_t encapsulated_key[], size_t encapsulated_key_len, size_t desired_shared_key_len, uint8_t shared_key[], size_t *shared_key_len)

Decrypt an encapsulated key and return the shared secret

int **botan_pk_op_kem_decrypt_destroy**(*botan_pk_op_kem_decrypt_t* op)

Destroy the operation, freeing memory

8.33.25 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 (DER) representation of the public key included in this certificate

int **botan_x509_cert_view_public_key_bits**(*botan_x509_cert_t* cert, *botan_view_ctx* ctx, *botan_view_bin_fn* view)

View the serialized (DER) 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.

int **botan_x509_cert_view_as_string**(*botan_x509_cert_t* cert, *botan_view_ctx* ctx, *botan_view_str_fn* view)

View 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, or NULL if the code is not known.

8.33.26 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.

8.33.27 ZFEC (Forward Error Correction)

Added in version 3.0.0.

int botan_zfec_encode(size_t K, size_t N, const uint8_t *input, size_t size, uint8_t **outputs)

Perform forward error correction encoding. The input length must be a multiple of *K* bytes. The *outputs* parameter must point to *N* output buffers, each of length *size / K*.

Any *K* of the *N* output shares is sufficient to recover the original input.

int botan_zfec_decode(size_t K, size_t N, const size_t *indexes, uint8_t *const *const inputs, size_t shareSize, uint8_t **outputs)

Decode some FEC shares. The indexes and inputs must be exactly K in length. The *indexes* array specifies which shares are presented in *inputs*. Each input must be of length *shareSize*. The output is written to the *K* buffers in *outputs*, each buffer must be *shareSize* long.

8.34 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, or set to 0, then it defaults to the number of CPUs available on the system. If it is set to the string "none" then the thread pool is disabled; instead all work passed to the thread pool will be executed immediately by the calling thread.

  As of version 3.2.0, on MinGW the thread pool is by default disabled, due to a bug which causes deadlock on application shutdown. Enabling the pool can be explicitly requested by setting `BOTAN_THREAD_POOL_SIZE` to an integer value.

- `BOTAN_MLOCK_POOL_SIZE` controls the total amount of memory, in bytes, which will be locked in memory using `mlock` or `VirtualLock` and managed in a memory pool. This should be a multiple of the system page size. If set to `0`, 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.
- BOTAN_CLEAR_CPUID: this variable can be set to a comma-separated list of CPUID fields to ignore. For example setting BOTAN_CLEAR_CPUID=avx2,avx512 will cause AVX2 and AVX-512 codepaths to be avoided. Note that disabling basic features (notably NEON or SSE2/SSSE3) can cause other higher level features like AES-NI to also become disabled.

8.35 Python Binding

Added 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.

The versioning of the Python module follows the major versioning of the C++ library. So for Botan 2, the module is named botan2 while for Botan 3 it is botan3.

8.35.1 Versioning

botan3.**version_major**()

Returns the major number of the library version.

botan3.**version_minor**()

Returns the minor number of the library version.

botan3.**version_patch**()

Returns the patch number of the library version.

botan3.**version_string**()

Returns a free form version string for the library

8.35.2 Random Number Generators

class botan3.**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

8.35.3 Hash Functions

class botan3.**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.

8.35.4 Message Authentication Codes

class botan3.**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.

8.35.5 Ciphers

class botan3.**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

8.35.6 Bcrypt

botan3.**bcrypt**(*passwd*, *rng*, *work_factor=10*)

Provided the password and an RNG object, returns a bcrypt string

botan3.**check_bcrypt**(*passwd*, *bcrypt*)

Check a bcrypt hash against the provided password, returning True iff the password matches.

8.35.7 PBKDF

botan3.**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

botan3.**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

8.35.8 Scrypt

Added in version 2.8.0.

botan3.**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.

8.35.9 KDF

botan3.**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.

8.35.10 Public Key

class botan3.**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)

8.35.11 Private Key

class botan3.**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 groups "curve25519" and "x448" (which are actually completely distinct key types 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.

8.35.12 Public Key Operations

class botan3.**PKEncrypt**(*pubkey*, *padding*)

Previously `pk_op_encrypt`

encrypt(*msg*, *rng*)

class botan3.**PKDecrypt**(*privkey*, *padding*)

Previously `pk_op_decrypt`

decrypt(*msg*)

class botan3.**PKSign**(*privkey*, *hash_w_padding*)

Previously `pk_op_sign`

update(*msg*)

finish(*rng*)

class botan3.**PKVerify**(*pubkey*, *hash_w_padding*)

Previously `pk_op_verify`

update(*msg*)

check_signature(*signature*)

class botan3.**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.

8.35.13 Multiple Precision Integers (MPI)

Added in version 2.8.0.

class botan3.**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`

8.35.14 Format Preserving Encryption (FE1 scheme)

Added in version 2.8.0.

class botan3.**FormatPreservingEncryptionFE1**(*modulus*, *key*, *rounds=5*, *compat_mode=False*)

Initialize an instance for format preserving encryption

encrypt(*msg*, *tweak*)

The msg should be a botan3.MPI or an object which can be converted to one

decrypt(*msg*, *tweak*)

The msg should be a botan3.MPI or an object which can be converted to one

8.35.15 HOTP

Added in version 2.8.0.

class botan3.**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.

8.35.16 X509Cert

class botan3.**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`.

8.35.17 X509CRL

class botan3.**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

NINE

COMMAND LINE INTERFACE

9.1 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.

All options for the command line are displayed in the summary line, and in the help output. All options are, as the name suggests, optional, and the default values are shown. For example `hash file` prints the SHA-256 of the file encoded as hex, while `hash --format=base64 --algo=SHA-384 file` prints the base64 encoded SHA-384 hash of the same file.

9.2 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`.

9.3 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*.

9.4 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.

9.5 Encryption

`cipher --buf-size=4096 --decrypt --cipher= --key= --nonce= --ad=`

Encrypt a given file with the specified *cipher*, eg "AES-256/GCM". If `--decrypt` is provided the file is decrypted instead.

9.6 Public Key Cryptography

keygen --algo=RSA --params= --passphrase= --cipher= --pbkdf= --pbkdf-ms=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*.

If a passphrase is used, *cipher* specifies the name of the desired encryption algorithm (such as "AES-256/CBC", or leave empty to use a default), and *pbkdf* can be used to specify the password hashing mechanism (either a hash such as "SHA-256" to select PBKDF2, or "Scrypt").

The cipher mode must have an object identifier defined, this allows use of ciphers such as AES, Twofish, Serpent, and SM4. Ciphers in CBC, GCM, and SIV modes are supported. However most other implementations support only AES or 3DES in CBC mode.

If encryption is used, the parameter *pbkdf-ms* controls how long the password hashing function will run 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.

`pkcs8 --pass-in= --pub-out --der-out --pass-out= --cipher= --pbkdf= --pbkdf-ms=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 *cipher*, *pbkdf*, and *pbkdf-ms* work similarly to `keygen`.

`sign --der-format --passphrase= --hash=SHA-256 --padding= --provider= key file`

Sign the data in *file* using the PKCS #8 private key *key* and cryptographic hash *hash*. If *key* is encrypted, the used passphrase must be passed as *pass-in*.

The *padding* option can be used to control padding for algorithms that have divergent methods; this mostly applies to RSA. For RSA, if the option is not specified PSS signatures are used. You can select generating a PKCS #1 v1.5 formatted signature instead by providing `--padding=PKCS1v15`.

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. This option does not make sense for other algorithms such as RSA.

The signature is formatted for your screen using base64.

verify --der-format --hash=SHA-256 --padding= 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, *padding* specifies the padding scheme and *hash* the cryptographic hash function to use.

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".

9.7 X.509

gen_pkcs10 key CN --country= --organization= --ca --path-limit=1 --email= --dns= --ext-ku= --key-pass= --hash=SHA-256 --emsa=
Generate a PKCS #10 certificate signing request (CSR) using the passed PKCS #8 private key *key*. If the private key is encrypted, the decryption passphrase *key-pass* has to be passed.*emsa* specifies the padding scheme to be used when calculating the signature.

- For RSA keys EMSA4 (RSA-PSS) is the default scheme.
- For ECDSA, DSA, ECGDSA, ECKCDSA and GOST-34.10 keys *emsa* defaults to EMSA1.

gen_self_signed key CN --country= --dns= --organization= --email= --path-limit=1 --days=365 --key-pass= --ca --hash=SHA-256 --emsa= --der
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.

9.8 TLS Server/Client

The `--policy=` argument of the TLS commands specifies the TLS policy to use. The policy can be any of 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-system-cert-store --trusted-cas= --session-db= --session-db-pass= --next-protocols= --type=tcp --client-cert= --client-cert-key=
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). Pass a path to a client certificate PEM and unencrypted PKCS8 encoded private key if client authentication is required.

tls_server cert key --port=443 --type=tcp --policy=default --dump-traces= --max-clients=0 --socket-id=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-db --session-db-pass=
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 --threads=0 --max-clients=0 --session-db= --session-db-pass=
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.

9.9 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.

9.10 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
```

9.11 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.

9.12 Data Encoding/Decoding

base32_dec file
Decode *file* to Base32.

base32_enc file
Encode 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
Decode *file* to Base64.

base64_enc file
Encode Base64 encoded *file*.

hex_dec file
Decode *file* to Hex.

hex_enc file
Encode Hex encoded *file*.

9.13 Forward Error Correction

fec_encode --suffix=fec --prefix= --output-dir= k n input
Split a given `input` file into `n` shares where `k` shares are required to recreate the original file. The output shares a written to files with the file extension specified in `--suffix` and either the original file name or the one specified in `--prefix`. The output directory is either equal to the input file's directory or the one specified in `--output-dir`.

fec_decode *shares
If given enough shares, this will output the original input file's content to stdout. Otherwise an error is printed on stderr.

fec_info share
Given a single share this will print information about the share. For instance: `FEC share 4/4 with 3 needed for recovery`

9.14 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 file`
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= --cpu-clock-speed=0 --cpu-clock-ratio=1.0 *algos
Measures the speed of the passed *algos*. If no *algos* are passed all available speed tests are executed. *msec* (in milliseconds) sets the period of measurement for each algorithm. 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=1000 --measurement-runs=10000
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=60 --host= --pubkey= --servers-file=
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

TEN

HARDWARE ACCELERATION

Botan provides built-in support for hardware acceleration of certain algorithms on certain platforms. These alternate implementations use special CPU instructions that are not available on all platforms and either speed up the algorithm or improve security in terms of side channel resistance.

A "base" software implementation is always provided. For example, for the AES-128 block cipher three implementations are available. All of the AES-128 implementations are immune to common cache/timing based side channels.

- If AES hardware support is available (AES-NI, POWER8, Aarch64) use that
- If 128-bit SIMD with byte shuffles are available (SSSE3, NEON, or Altivec), use the vperm technique published by Mike Hamburg at CHES 2009
- If no hardware or SIMD support, fall back to a constant time bitsliced implementation

The following sections list the platforms and algorithms for which hardware acceleration is available. If the CPU specific optimizations are available at runtime, they are automatically used if enabled in the build. If not, the base implementation is used.

10.1 x86

On x86-64 and x86-32 platforms, the following CPU specific optimizations are available:

Algorithm	Extension	Module	Added in
AES	AES-NI	*aes_ni*	1.9.3
	SSSE3	*aes_vperm*	1.9.10
AES-GCM	CLMUL	*ghash_cpu*	1.11.6
	SSSE3	*ghash_vperm*	1.9.10
Argon2	AVX2	*argon2_avx2*	3.0.0
	SSSE3	*argon2_ssse3*	2.19.2
ChaCha	AVX512 (x86-64 only)	*chacha_avx512*	3.1.0
	AVX2	*chacha_avx2*	2.8.0
	SSE2	*chacha_simd32*	1.11.32
KMAC	BMI2	*keccak_perm_bmi2*	3.2.0
NOEKEON	SSE2	*noekeon_simd*	1.9.4
RDRAND	RDRAND	*processor_rng*	1.11.31
RDSEED	RDSEED	*rdseed*	1.11.36
Serpent	AVX512 (x86-64 only)	*serpent_avx512*	3.1.0
	AVX2	*serpent_avx2*	2.8.0
	SSE2	*serpent_simd*	1.9.0
SHACAL2	Intel SHA Extensions	*shacal2_x86*	2.3.0
	AVX2	*shacal2_avx2*	2.13.0
SHAKE	BMI2	*keccak_perm_bmi2*	2.13.0
SHA-1	Intel SHA Extensions	*sha1_x86*	2.2.0
	SSE2	*sha1_sse2*	1.7.12
SHA-256	Intel SHA Extensions	*sha2_32_x86*	2.2.0
	BMI2	*sha2_32_bmi2*	2.7.0
SHA-3	BMI2	*keccak_perm_bmi2*	2.10.0

10.2 ARM

On arm64 and arm32 platforms, the following CPU specific optimizations are available:

Algorithm	Extension	Module	Added in
AES	NEON	*aes_armv8*	1.9.3
AES-GCM	PMULL (arm64 only)	*ghash_cpu*	2.3.0
	NEON	*ghash_vperm*	2.12.0
ChaCha	NEON	*chacha_simd32*	2.8.0
NOEKEON	NEON	*noekeon_simd*	1.9.4
Serpent	NEON	*serpent_simd*	1.9.2
SHACAL2	NEON	*shacal2_simd*	2.3.0
SM4	ARMv8 Cryptography Extensions (arm64 only)	*sm4_armv8*	2.8.0
SHA-1	ARMv8 Cryptography Extensions (arm64 only)	*sha1_armv8*	2.2.0
SHA-256	ARMv8 Cryptography Extensions (arm64 only)	*sha2_32_armv8*	2.2.0
SHA-384	ARMv8 Cryptography Extensions (arm64 only)	*sha2_64_armv8*	3.3.0
SHA-512	ARMv8 Cryptography Extensions (arm64 only)	*sha2_64_armv8*	3.3.0

10.3 PowerPC

On ppc64 and ppc32 platforms, the following CPU specific optimizations are available:

Algorithm	Extension	Module	Added in
AES	POWER8/POWER9	*aes_power8*	2.14.0
	AltiVec	*aes_vperm*	2.12.0
AES-GCM	AltiVec	*ghash_vperm*	2.12.0
ChaCha	AltiVec	*chacha_simd32*	2.8.0
DARN	POWER9	*processor_rng*	2.15.0
Serpent	AltiVec	*serpent_simd*	1.9.2
SHACAL2	AltiVec	*shacal2_simd*	2.3.0
NOEKEON	AltiVec	*noekeon_simd*	1.9.4

10.4 Configuring Acceleration

Hardware acceleration can be disabled at during configuring the build by passing certain `--disable-*` options to `configure.py`. This will cause the base software implementation to be used instead of the hardware accelerated one. The following options are currently supported:

`--disable-sse2`
: disable SSE2 intrinsics

`--disable-ssse3`
: disable SSSE3 intrinsics

`--disable-sse4.1`
: disable SSE4.1 intrinsics

`--disable-sse4.2`
: disable SSE4.2 intrinsics

`--disable-avx2`
: disable AVX2 intrinsics

`--disable-bmi2`
: disable BMI2 intrinsics

`--disable-rdrand`
: disable RDRAND intrinsics

`--disable-rdseed`
: disable RDSEED intrinsics

`--disable-aes-ni`
: disable AES-NI intrinsics

`--disable-sha-ni`
: disable SHA-NI intrinsics

`--disable-altivec`
: disable AltiVec intrinsics

`--disable-neon`
: disable NEON intrinsics

`--disable-armv8crypto`
: disable ARMv8 Crypto intrinsics

`--disable-powercrypto`
: disable POWER Crypto intrinsics

Additionally, `--disable-modules=MODS` can be used to remove a certain module, if desirable.

Last but not least, the `BOTAN_CLEAR_CPUID` *environment variable* can be set to a non-empty value *at runtime* to cause Botan to clear the CPUID bits for the CPU extensions it uses.

CHAPTER

ELEVEN

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.

11.1 Platform Support Deprecations

- Support for building for Windows systems prior to Windows 10 is deprecated.

11.2 TLS Protocol Deprecations

The following TLS protocol features are deprecated and will be removed in a future major release:

- 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.
- All CBC mode ciphersuites. This includes all available 3DES ciphersuites. This implies also removing Encrypt-then-MAC extension.
- All DHE ciphersuites
- Support for renegotiation in TLS v1.2
- All ciphersuites using static RSA key exchange
- `Credentials_Manager::psk()` to provide various TLS-specific keys and secrets, most notably "session-ticket", "dtls-cookie-secret" and the actual TLS PSKs for given identities and hosts. Instead, use the dedicated methods in `Credentials_Manager` and do not override the `psk()` method any longer.

11.3 Elliptic Curve Deprecations

A number of features relating to elliptic curves are deprecated. As a typical user you would probably not notice these; their removal would not affect for example using ECDSA signatures or TLS, but only applications doing unusual things such as custom elliptic curve parameters, or creating your own protocol using elliptic curve points.

- 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).
- Currently it is possible to create an `EC_Group` with cofactor > 1. None of the builtin groups have composite order, and in the future it will be impossible to create composite order `EC_Group`.
- Currently it is possible to create an application specific `EC_Group` with parameters of effectively arbitrary size. In a future release the parameters of application provided elliptic curve will be limited in the following ways.
  a) The bitlength must be between 128 and 512 bits, and a multiple of 32
  b) As an extension of (a) you can also use the 521 bit Mersenne prime
  c) The prime must be congruent to 3 modulo 4
  d) The bitlength of the prime and the bitlength of the order must be equal
- Elliptic curve points can be encoded in several different ways. The most common are "compressed" and "uncompressed"; both are widely used in various systems. Botan additionally supports a "hybrid" encoding format which is effectively uncompressed but with an additional indicator of the parity of the y coordinate. This format is quite obscure and seemingly rarely implemented. Support for this encoding will be removed in a future release.
- Botan currently contains support for a number of relatively weak or little used elliptic curves. These are deprecated. These include "secp160k1", "secp160r1", "secp160r2", "secp192k1", "secp224k1", "brainpool160r1", "brainpool192r1", "brainpool224r1", "brainpool320r1", "x962_p192v2", "x962_p192v3", "x962_p239v1", "x962_p239v2", "x962_p239v3", "gost_256A", "gost_512A"
- Currently *EC_Point* offers a wide variety of functionality almost all of which was intended only for internal implementation. In a future release, the only operations available for EC_Points will be to extract the byte encoding of their affine x and y coordinates.

11.4 Deprecated Modules

In a number of cases an entire module is deprecated. If the build is configured with `--disable-deprecated` then these will not be included. In a future major release the source for these modules will be entirely removed.

Deprecated modules include

- Kyber mode `kyber_90s`: Kyber's "90s mode" is not in the NIST ML-KEM standard, and seems to have been never implemented widely.
- Dilithium mode `dilithium_aes`: Similar situation to Kyber 90s mode.
- Block cipher `gost_28147`: This cipher was obsolete 20 years ago.
- Block cipher `noekeon`: An interesting design but not widely implemented.
- Block cipher `lion`: Similar situation to Noekeon
- Hash function `gost_3411`: Very weak and questionable hash function.
- Hash function `streebog`: Incredibly sketchy situation with the sbox
- Hash function `md4`: It's time to let go
- Signature scheme `gost_3410`

- McEliece implementation `mce`. Will be replaced by the proposal Classic McEliece.
- Stream cipher `shake_cipher`. Note this deprecation affects only using SHAKE as a `StreamCipher` not as a hash or XOF
- *cryptobox*: A not unreasonable password based encryption utility but neither modern (these days) nor widely implemented.
- `dlies`: DLIES is considered quite obsolete
- `tpm` (TPM 1.2 only, rarely tested)

11.5 Other Deprecated Functionality

This section lists other functionality which will be removed in a future major release, or where a backwards incompatible change is expected.

- Support for OtherNames in X.509 certificates is deprecated
- The `PBKDF` class is deprecated in favor of `PasswordHash` and `PasswordHashFamily`.
- Implicit conversion of a private key into a public key. Currently `Private_Key` derives from `Public_Key` (and likewise for each of the algorithm specfic classes, eg `RSA_PrivateKey` derives from `RSA_PublicKey`). In a future release these derivations will not exist. To correctly extract the public key from a private key, use the function `Private_Key::public_key()`
- Prior to 2.8.0, SM2 algorithms were implemented as two distinct key types, one used for encryption and the other for signatures. In 2.8, the two types were merged. However it is still possible to refer to SM2 using the split names of "SM2_Enc" or "SM2_Sig". In a future major release this will be removed, and only "SM2" will be recognized.
- DSA, ECDSA, ECGDSA, ECKCDSA, and GOST-34.10 previously (before Botan 3) required that the hash be named as "EMSA1(HASH_NAME)". This is no longer required. In a future major release, only "HASH_NAME" will be accepted.
- 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.
- GCM support for 64-bit tags
- All built in MODP groups < 2048 bits
- All pre-created DSA groups
- All support for loading, generating or using RSA keys with a public exponent larger than 2**64-1

11.6 Deprecated Headers

These headers are currently publically available, but will be made internal to the library in the future.

PBKDF headers: `bcrypt_pbkdf.h`, `pbkdf2.h`, `pgp_s2k.h`, `scrypt.h`, and `argon2.h`: Use the `PasswordHash` interface instead.

Internal implementation headers - seemingly no reason for applications to use: `curve_gfp.h`, `reducer.h`, `tls_algos.h`, `tls_magic.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. `compiler.h`, `uuid.h`,

CHAPTER

TWELVE

DEVELOPMENT ROADMAP

12.1 Near Term Plans

Here is an outline for the development plans over the next 12-24 months, as of May 2024.

12.2 Botan 2

Botan 2 is still supported, but no further feature work is planned. Only security issues and serious bugs will be addressed.

12.3 Botan 3

The following future work is currently planned for Botan 3:

- BSI Project 481 [https://github.com/randombit/botan/issues/3108] will add several new post-quantum algorithms including LMS signatures and Classic McEliece.
- New ECC based password authenticated key exchanges, to replace SRP. The most likely candidate algorithms are CPace and OPAQUE.
- Adding an implementation of BLS12-381 elliptic curve pairing.
- Low level integer math and elliptic curve arithmetic optimizations.

12.4 Botan 4

At this time there is no immediate plan for a new major version. When it occurs, it will remove functionality currently marked as deprecated, and adopt a new C++ version. This is unlikely to occur before 2027, at the earliest.

One major change already planned for Botan 4 is that in this release, Public_Key will no longer derive from Private_Key. And similarly, specific private keys (for example RSA_PrivateKey) will no longer derive from their corresponding public key type.

CHAPTER

THIRTEEN

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
```

(continues on next page)

(continued from previous page)

```
D: Windows porting

N: Manuel Hartl
E: hartl@flexsecure.de
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 Fischer (formerly Korthaus)
E: rene.fischer@rohde-schwarz.com
W: https://www.rohde-schwarz.com/cybersecurity
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
```

(continues on next page)

(continued from previous page)

```
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

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: philipp.weber@rohde-schwarz.com
W: https://www.rohde-schwarz.com/cybersecurity
D: KDF1-18033, ECIES
S: Saarland, Germany

N: Daniel Neus
E: daniel.neus@rohde-schwarz.com
W: https://www.rohde-schwarz.com/cybersecurity
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
```

(continues on next page)

(continued from previous page)

```
W: https://www.ribose.com/
D: SM3, Streebog, various minor contributions

N: Rostyslav Khudolii
E: rhudoliy@gmail.com
D: SRP6 FFI
S: Ukraine/Denmark

N: René Meusel
E: rene.meusel@rohde-schwarz.com
W: https://www.rohde-schwarz.com/cybersecurity
D: CI, TLS 1.3, Kyber, Dilithium, SPHINCS+, FrodoKEM
S: Berlin, Germany

N: Philippe Lieser
E: philippe.lieser@rohde-schwarz.com
W: https://www.rohde-schwarz.com/cybersecurity
D: CI, BSI module policy, HSS/LMS, various minor contributions
S: Saarland, Germany

N: Fabian Albert
E: fabian.albert@rohde-schwarz.com
W: https://www.rohde-schwarz.com/cybersecurity
D: SPHINCS+, HSS/LMS
S: Bochum, Germany
```

CHAPTER

FOURTEEN

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

FIFTEEN

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.

15.1 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.

15.2 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.

15.3 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.

15.4 CMake Integration

Starting in Botan 3.3.0, we ship `botan-config.cmake` files. While this config file is somewhat relocatable, it assumes the default installation directory structure as generated by `make install`. If your distribution changes the directory layout of the installed files you might want to either adapt the final `botan-config.cmake` file accordingly or leave it out entirely using `--without-cmake-config`.

Please don't hesitate to give your feedback on this new feature by opening a ticket on the upstream GitHub.

15.5 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

SIXTEEN

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.

16.1 2024

- 2024-07-08 (CVE-2024-34702): Denial of Service Due to Excessive Name Constraints

  Checking name constraints in X.509 certificates is quadratic in the number of names and name constraints. An attacker who presented a certificate chain which contained a very large number of names in the SubjectAlternativeName, signed by a CA certificate which contained a large number of name constraints, could cause a denial of service.

  Introduced in 2.0.0, fixed in 2.19.5 and 3.5.0

  Found and reported by Bing Shi.

- 2024-07-08 (CVE-2024-39312): Authorization Error due to Name Constraint Decoding Bug

  A bug in the parsing of name constraint extensions in X.509 certificates meant that if the extension included both permitted subtrees and excluded subtrees, only the permitted subtree would be checked. If a certificate included a name which was permitted by the permitted subtree but also excluded by excluded subtree, it would be accepted.

  Introduced in 2.0.0, fixed in 2.19.5 and 3.5.0

- 2024-02-20: Kyber side channel

  The Kyber implementation was vulnerable to the KyberSlash1 and KyberSlash2 side channel issues.

  Introduced in 3.0.0, fixed in 3.3.0

- 2024-02-20 (CVE-2024-34703): DoS due to oversized elliptic curve parameters

  When decoding an ASN.1 encoded elliptic curve, Botan would verify the p parameter was actually prime, and at least some minimum size. However it failed to check if the prime was far too large (for example thousands of bits), in which case checking the prime would take a significant amount of computation. Now the maximum size of arbitrary elliptic curves when decoding from ASN.1 is limited.

  Reported by Bing Shi

Fixed in 3.3.0 and 2.19.4

16.2 2022

- 2022-11-16 (CVE-2022-43705): Failure to correctly check OCSP responder embedded certificate

  OCSP responses for some end entity are either signed by the issuing CA certificate of the PKI, or an OCSP responder certificate that the PKI authorized to sign responses in their name. In the latter case, the responder certificate (and its validation path certificate) may be embedded into the OCSP response and clients must verify that such certificates are indeed authorized by the CA when validating OCSP responses.

  The OCSP implementation failed to verify that an authorized responder certificate embedded in an OCSP response is authorized by the issuing CA. As a result, any valid signature by an embedded certificate passed the check and was allowed to make claims about the revocation status of certificates of any CA.

  Attackers that are in a position to spoof OCSP responses for a client could therefore render legitimate certificates of a 3rd party CA as revoked or even use a compromised (and actually revoked) certificate by spoofing an OCSP-"OK" response. E.g. an attacker could exploit this to impersonate a legitimate TLS server using a compromised certificate of that host and get around the revocation check using OCSP stapling.

  Introduced in 1.11.34, fixed in 2.19.3 and 3.0.0

16.3 2020

- 2020-12-21 (CVE-2021-24115): Codec encoding/decoding was not constant time

  The base64, base32, base58 and hex encoding/decoding routines used lookup tables which could leak information via a cache-based side channel attack. The encoding tables were small and unlikely to be exploitable, but the decoding tables were large enough to cause non-negligible information leakage. In particular parsing an unencrypted PEM-encoded private key within an SGX enclave could be easily attacked to leak key material.

  Identified and reported by Jan Wichelmann, Thomas Eisenbarth, Sebastian Berndt, and Florian Sieck.

  Fixed in 2.17.3

- 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.

16.4 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

16.5 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.

16.6 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

16.7 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

16.8 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

SEVENTEEN

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.

17.1 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.

17.2 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.

17.3 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.

RSA key generation is also prone to side channel vulnerabilities due to the need to calculate the CRT parameters. The GCD computation, LCM computations, modulo, and inversion of `q` modulo `p` are all done via constant time algorithms. An additional inversion, of `e` modulo `phi(n)`, is also required. This one is somewhat more complicated because `phi(n)` is even and the primary constant time algorithm for inversions only works for odd moduli. This is worked around by a technique based on the CRT - `phi(n)` is factored to `2**e * z` for some `e > 1` and some odd `z`. Then `e` is inverted modulo `2**e` and also modulo `z`. The inversion modulo `2**e` is done via a specialized constant-time algoirthm which only works for powers of 2. Then the two inversions are combined using the CRT. This process does leak the value of `e`; to avoid problems `p` and `q` are chosen so that `e` is always 1.

See blinding.cpp and rsa.cpp.

17.4 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.

17.5 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.

17.6 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.

17.7 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 ec_point.cpp.

17.8 ECC scalar multiply

There are several different implementations of ECC scalar multiplications which depend on the API invoked. This include `EC_Point::operator*`, `EC_Group::blinded_base_point_multiply` and `EC_Group::blinded_var_point_multiply`.

The `EC_Point::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 scaled to be half the bit length of the order of the group.

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 ec_point.cpp and point_mul.cpp in src/lib/pubkey/ec_group

17.9 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.

17.10 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.

17.11 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.

17.12 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.

17.13 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.

17.14 base64 decoding

Base64 (and related encodings base32, base58 and hex) are sometimes used to encode or decode secret data. To avoid possible side channels which might leak key material during the encoding or decoding process, these functions avoid any input-dependent table lookups.

17.15 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.

17.16 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.

17.17 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.

17.18 Poly1305

The Poly1305 implementation does not have any secret lookups or conditionals. The code is based on the public domain version by Andrew Moon.

17.19 DES/3DES

The DES implementation relies on table lookups but they are limited to tables which are exactly 64 bytes in size. On systems with 64 byte (or larger) cache lines, these should not leak information. It may still be vulnerable to side channels on processors which leak cache line access offsets via cache bank conflicts; vulnerable hardware includes Sandy Bridge processors, but not later Intel or AMD CPUs.

17.20 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.

17.21 ChaCha20, Serpent, Threefish, ...

Some algorithms including ChaCha, Salsa, Serpent and Threefish are 'naturally' silent to cache and timing side channels on all recent processors.

17.22 IDEA

IDEA encryption, decryption, and key schedule are implemented to take constant time regardless of their inputs.

17.23 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.

17.24 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.

17.25 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.

17.26 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.

17.27 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.

17.28 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

EIGHTEEN

DEVELOPER REFERENCE

This section contains information useful to people making contributions to the library

18.1 Notes for New Contributors

18.1.1 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`.
- `python/botan3.py` is the Python ctypes wrapper
- `bogo_shim` contains the shim binary and configuration for BoringSSL's TLS test suite (https://github.com/google/boringssl/tree/master/ssl/test)
- `fuzzer` contains fuzz targets for various modules of the library
- `build-data` contains files read by the configure script. For example `build-data/cc/gcc.txt` describes various gcc options.
- `examples` contains usage examples used in the documentation.
- `scripts` contains misc scripts: install, distribution, various codegen things. Scripts controlling CI go under `scripts/ci`.
- `configs` contains configuration files tools like pylint
- `editors` contains configuration files for editors like vscode and emacs

Under `doc` one finds the sources of this documentation

18.1.2 Library Layout

Under `src/lib` are several directories

- `asn1` is the DER encoder/decoder
- `base` defines some high level types
- `block` contains the block cipher implementations
- `codec` has hex, base64, base32, base58
- `compat` a (partial) compatibility layer for the libsodium API
- `compression` has the compression wrappers (zlib, bzip2, lzma)
- `entropy` has various entropy sources used by some of the RNGs
- `ffi` is the C99 API
- `filters` is a filter/pipe API for data transforms
- `hash` contains the hash function implementations
- `kdf` contains the key derivation functions
- `mac` contains the message authentication codes
- `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
- `misc` contains odds and ends: format preserving encryption, SRP, threshold secret sharing, all or nothing transform, and others
- `modes` contains block cipher modes (CBC, GCM, etc)
- `passhash` contains password hashing algorithms for authentication
- `pbkdf` contains password hashing algorithms for key derivation
- `pk_pad` contains padding schemes for public key algorithms
- `prov` contains bindings to external libraries such as PKCS #11
- `psk_db` contains a generic interface for a Pre-Shared-Key database
- `pubkey` contains the public key algorithms
- `rng` contains the random number generators
- `stream` contains the stream ciphers
- `tls` contains the TLS implementation
- `utils` contains various utility functions and types
- `x509` is X.509 certificates, PKCS #10 requests, OCSP

Each of these folders can contain subfolders which are treated as modules if they contain an `info.txt` file. These submodules have an implicit dependency on their parent module. The chapter *Understanding configure.py* contains more information on Botan's module architecture.

18.1.3 Sending patches

All contributions should be submitted as pull requests via GitHub (https://github.com/randombit/botan). If you are planning a large change, open a discussion ticket on github before starting out to make sure you are on the right path. And once you have something written, even if it is not complete/ready to go, feel free to open a draft 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.

18.1.4 FFI Additions

If adding a new function declaration to `ffi.h`, the same PR must also add the same declaration in the Python binding `botan3.py`, in addition the new API functionality must be exposed to Python and a test written in Python.

18.1.5 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.

18.1.6 Python

Scripts should be in Python 3 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 3.x. Support for PyPy, etc is great when viable (in the sense of not causing problems for 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 botan3.py targets CPython 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 platform independence. 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.

18.1.7 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, TrouSerS (libtspi), and Sqlite3.

18.1.8 Copyright Notice

At the top of any new file add a comment with a copyright and a reference to the license, for example:

```
/*
* (C) 202x <You>
*
* 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.

18.1.9 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).

Unless there is a specific reason otherwise (eg due to calling some C API which requires exactly a `long*` be provided) integer types should be either `(u)intXX_t` or `size_t`. If the variable is used for integer values of "no particular size", as in the loop `for(some_type i = 0; i != 100; ++i)` then the type should be `size_t`. Use one of the specific size integer types only when there is a algorithmic/protocol reason to use an integer of that size. For example if a parsing a protocol that uses 16-bit integer fields to encode a length, naturally one would use `uint16_t` there.

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 further derivation, mark it `final`.

Avoid explicit `new` or (especially) explicit `delete`: use RAII, `make_unique`, etc.

Use `m_` prefix on all member variables.

`clang-format` is used for all C++ formatting. The configuration is in `.clang-format` in the root directory. You can rerun the formatter using `make fmt` or by invoking the script `src/scripts/dev_tools/run_clang_format.py`. If the output would be truly horrible, it is allowed to disable formatting for a specific area using `// clang-format off` annotations.

Note: Since the output of clang-format varies from version to version, we currently require using exactly `clang-format 17`.

Use braces on both sides of if/else blocks, even if only using a single statement.

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`).

18.1.10 Use of External Dependencies

Compiler Dependencies

The library should always be as functional as possible when compiled with just Standard C++20. However, feel free to use the full language.

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 (using `CT::poison`) 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 several 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 interfaces (CommonCrypto, CryptoAPI, /dev/crypto, iOS keychain, TPM 2.0, etc)

18.2 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*.

18.2.1 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.

18.2.2 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.

18.2.3 When Modifying `configure.py`

Run `./src/scripts/ci_build.py lint` to run Pylint checks after any change.

18.2.4 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.

  Using `|concat:<some string>` as a suffix, it is possible to conditionally concatenate the variable value with a static string defined in the template. This is useful for compiler switches that require a template-defined parameter value. If the substitution value is not set (i.e. "empty"), also the static concatenation value is omitted.

- 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.

18.2.5 Build.h

The `build.h` header file is generated and overwritten each time the `configure.py` script is executed. This header can be included in any header or source file and provides plenty of compile-time information in the form of preprocessor `#define`s.

It is helpful to check which modules are included in the current build of the library via macro defines of the form "BOTAN_HAS" followed by the module name. Also, it contains *version information macros* and compile-time library configurations.

18.2.6 Adding a new module

Create a directory in the appropriate place and create a `info.txt` file.

18.2.7 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'.

18.2.8 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 and module_info.

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 internal.

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. If you need to exclude just one specific compiler (for example because that compiler miscompiles the code in the module), you can prefix a compiler name with `!` - like `!msvc`.
- `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`.
- `module_info` contains documentation-friendly information about the module. Available mappings:
  - `name` must contain a human-understandable name for the module
  - `brief` may provide a short description about the module's contents
  - `type` specifies the type of the module (defaults to `Public`)
    - `Public` Library users can directly interact with this module. E.g. they may enable or disable the module at will during build.
    - `Internal` Library users must not directly interact with this module. It is enabled and used as required by other modules.
    - `Virtual` This module does not contain any implementation but acts as a container for other submodules. It cannot be interacted with by the library user and cannot be depended upon directly.
  - `lifecycle` specifies the module's lifecycle (defaults to `Stable`)
    - `Stable` The module is stable and will not change in a way that would break backwards compatibility.
    - `Experimental` The module is experimental and may change in a way that would break backwards compatibility. Not enabled in a default build. Either use `--enable-modules` or `--enable-experimental-features`.
    - `Deprecated` The module is deprecated and will be removed in a future release. It remains to be enabled in a default build. Either use `--disable-modules` or `--disable-deprecated-features`.
- `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>
DEFINE1 -> 20180104
DEFINE2 -> 20190301
</defines>

<module_info>
name -> "This Is Just To Say"
brief -> "Contains a poem by William Carlos Williams"
</module_info>

<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>
```

(continues on next page)

(continued from previous page)

```
macos -> FramyMcFramerson
</frameworks>

<libs>
qnx -> foo,bar,baz
solaris -> socket
</libs>
```

18.2.9 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.

18.2.10 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++.
- lang_binary_linker_flags gives flags to be passed to the linker when creating a binary
- 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.

- `ninja_header_deps_style` style of include dependency tracking for Ninja, see also https://ninja-build.org/manual.html#ref_headers.
- `header_deps_flag` flag to write out dependency information in the style required by `ninja_header_deps_style`.
- `header_deps_out` flag to specify name of the dependency output file.
- `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.

18.2.11 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.

18.3 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`.

18.3.1 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.

18.3.2 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.

18.3.3 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.

18.3.4 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`.

18.3.5 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
```

(continues on next page)

(continued from previous page)

```
# 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.

18.3.6 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.

18.4 Continuous Integration and Automated Testing

18.4.1 CI Build Script

The Github Actions builds are orchestrated using a script `src/scripts/ci_build.py`. This allows one to easily reproduce the CI process on a local machine.

18.4.2 Github Actions

https://github.com/randombit/botan/actions/workflows/ci.yml

Github Actions is the primary CI, and tests the Linux, Windows, macOS, and iOS builds. Among other things it runs tests using valgrind, cross-compilation for various architectures (currently including ARM and PPC64), MinGW build, and a build that produces the coverage report.

The Github Actions configuration is in `.github/workflows/ci.yml` which executes platform dependent setup scripts `src/scripts/ci/setup_gh_actions.sh` or `src/scripts/ci/setup_gh_actions.ps1` and `.../setup_gh_actions_after_vcvars.ps1` to install needed packages and detect certain platform specifics like compiler cache locations.

Then `src/scripts/ci_build.py` is invoked to steer the actual build and test runs.

18.4.3 Github Actions (nightly)

https://github.com/randombit/botan/actions/workflows/nightly.yml

Some checks are just too slow to include in the main CI builds. These are instead delegated to a scheduled job that runs every night against master.

Currently these checks include a full run of `valgrind` (the valgrind build in CI only runs a subset of the tests), and a run of `clang-tidy` with all warnings (that we are currently clean for) enabled. Each of these jobs takes about an hour to run. In the main CI, we aim to have no job take more than half an hour.

18.4.4 OSS-Fuzz

https://github.com/google/oss-fuzz/

OSS-Fuzz is a distributed fuzzer run by Google. Every night, the fuzzer harnesses in `src/fuzzer` are built and run on many machines, with any findings reported to the developers via email.

18.5 Fuzzing The Library

Botan comes with a set of fuzzing endpoints which can be used to test the library.

18.5.1 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 \
      --with-debug-info --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.

18.5.2 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
```

18.5.3 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.

18.5.4 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

18.5.5 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.

18.6 Release Process and Checklist

Releases are done quarterly, normally on the second 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.

18.6.1 Pre Release Testing

Do maintainer-mode builds with Clang and GCC to catch any warnings that should be corrected.

Other checks which are not in CI:

- 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.

18.6.2 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 3.8.2 release"
$ git checkout release-3
$ git merge master
$ git tag 3.8.2
```

18.6.3 Build The Release Tarballs

The release script is `src/scripts/dist.py` and must be run from a git workspace.

```
$ src/scripts/dist.py 3.8.2
```

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 3.8.2 --output-dir=botan-releases
$ cd botan-releases
$ sha256sum Botan-3.8.2.tgz >> sha256sums.txt
$ git add .
$ git commit -m "Release version 3.8.2"
$ git push origin master
```

A cron job updates the live site every 10 minutes.

18.6.4 Push to GitHub

Don't forget to also push tags:

```
$ git push origin --tags release-3 master
```

18.6.5 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-dir botan-website
$ cd botan-website
$ git add .
$ git commit -m "Update for 3.8.2"
$ git push origin master
```

18.6.6 Announce The Release

Send an email to the botan-announce and botan-devel mailing lists noting that a new release is available.

18.7 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.

18.7.1 New Ciphers/Hashes/MACs

- GCM-SIV (RFC 8452)
- EME* tweakable block cipher (https://eprint.iacr.org/2004/125)
- PMAC
- SIV-PMAC
- Threefish-1024
- Skein-MAC
- FFX format preserving encryption (NIST 800-38G)
- Adiantum (https://eprint.iacr.org/2018/720)
- HPKE (RFC 9180)
- Blake3

18.7.2 Improved Ciphers Implementations

- Stiched AES/GCM mode for CPUs supporting both AES and CLMUL
- Combine AES-NI, ARMv8 and POWER AES implementations (as already done for CLMUL)
- Support for VAES (Zen4/Ice Lake)
- NEON/VMX support for the SIMD based GHASH
- 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
- SHA-512 using BMI2+AVX2
- Constant time bitsliced DES
- SIMD evaluation of SHA-2 and SHA-3 compression functions
- Improved Salsa implementations (SIMD_4x32 and/or AVX2)
- Add CLMUL/PMULL implementations for CRC24/CRC32

18.7.3 Public Key Crypto, Math

- Short vector optimization for BigInt
- BLS12-381 pairing, BLS signatures
- Identity based encryption
- Paillier homomorphic cryptosystem
- New PAKEs (pending CFRG bakeoff results)
- SPHINX password store (https://eprint.iacr.org/2018/695)

18.7.4 Utility Functions

- Make Memory_Pool more concurrent (currently uses a global lock)
- Guarded integer type to prevent overflow bugs

18.7.5 External Providers, Hardware Support

- Add support for ARMv8.4-A SHA-3, SM3 and RNG instructions
- Aarch64 inline asm for mp
- /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 for VPSUM on big-endian PPC64 (GH #2252)

18.7.6 TLS

- Make DTLS support optional at build time
- Improve/optimize DTLS defragmentation and retransmission
- Make RSA optional at build time
- Make finite field DH optional at build time
- Certificate Transparency extensions
- TLS supplemental authorization data (RFC 4680, RFC 5878)
- DTLS-SCTP (RFC 6083)

18.7.7 PKIX

- Further tests of validation API (see GH #785)
- Test suite for validation of 'real world' cert chains (GH #611)
- X.509 policy constraints
- OCSP responder logic

18.7.8 New Protocols / Formats

- Noise protocol
- ACME protocol (needs a story for JSON)
- Cryptographic Message Syntax (RFC 5652)
- Fernet symmetric encryption (https://cryptography.io/en/latest/fernet/)
- RNCryptor format (https://github.com/RNCryptor/RNCryptor)
- Age format (https://age-encryption.org/v1)
- 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

18.7.9 Cleanups

- Unicode path support on Windows (GH #1615)
- The X.509 path validation tests have much duplicated logic

18.7.10 New C APIs

- PKCS10 requests
- Certificate signing
- CRLs
- Expose TLS
- Expose secret sharing
- Expose deterministic PRNG
- base32
- base58
- DL_Group
- EC_Group

18.7.11 Build/Test

- 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
- Add support for vxWorks

18.7.12 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
- Password store utility
- TOTP calculator
- Clone of magic wormhole
- ACVP client (https://github.com/usnistgov/ACVP)

18.7.13 Documentation

- Always needs help

18.8 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
g: generic
h: haiku
h: hpux
h: hurd
i: ios
l: linux
l: llvm
m: macos
m: mingw
n: netbsd
n: none
o: openbsd
q: qnx
s: solaris
u: uwp
w: windows
```

Feature	a	a	c	d	e	f	g	h	h	h	i	l	l	m	m	n	n	o	q	s	u	w
alloc_conceal																		X				
apple_keychain														X								
arc4random		X		X		X					X			X		X		X				
atomics	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X
auxinfo																X						
cap_enter						X																
ccrandom														X								
certificate_store															X							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		
elf_aux_info						X																
explicit_bzero				X		X						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	X
getauxval		X										X										
getentropy						X						X		X				X		X		
getrandom				X								X										
pledge																		X				
posix1	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		
prctl		X										X										
proc_fs	X			X								X								X		
rtlgenrandom															X							X
rtlsecurezeromemory																					X	X
sandbox_proc														X								
setppriv																				X		
sockets	X	X	X	X		X		X	X	X	X	X		X		X		X	X	X		
thread_local	X	X	X	X		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.

18.9 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 }
```

(continues on next page)

(continued from previous page)

```
-- { publicKey 4 } previously used as private X25519
-- { publicKey 5 } previously used for XMSS draft 6
gost-3410-with-sha256 OBJECT IDENTIFIER ::= { publicKey 6 1 }

frodokem-shake  OBJECT IDENTIFIER ::= { publicKey 14 }
efrodokem-shake OBJECT IDENTIFIER ::= { publicKey 16 }
frodokem-aes    OBJECT IDENTIFIER ::= { publicKey 15 }
efrodokem-aes   OBJECT IDENTIFIER ::= { publicKey 17 }

frodokem-640-shake   OBJECT_IDENTIFIER : { frodokem-shake  1  }
frodokem-976-shake   OBJECT_IDENTIFIER : { frodokem-shake  2  }
frodokem-1344-shake  OBJECT_IDENTIFIER : { frodokem-shake  3  }
frodokem-640-aes     OBJECT_IDENTIFIER : { frodokem-aes    1  }
frodokem-976-aes     OBJECT_IDENTIFIER : { frodokem-aes    2  }
frodokem-1344-aes    OBJECT_IDENTIFIER : { frodokem-aes    3  }
efrodokem-640-shake  OBJECT_IDENTIFIER : { efrodokem-shake 1  }
efrodokem-976-shake  OBJECT_IDENTIFIER : { efrodokem-shake 2  }
efrodokem-1344-shake OBJECT_IDENTIFIER : { efrodokem-shake 3  }
efrodokem-640-aes    OBJECT_IDENTIFIER : { efrodokem-aes   1  }
efrodokem-976-aes    OBJECT_IDENTIFIER : { efrodokem-aes   2  }
efrodokem-1344-aes   OBJECT_IDENTIFIER : { efrodokem-aes   3  }

kyber       OBJECT IDENTIFIER ::= { publicKey 7 }
kyber-90s   OBJECT IDENTIFIER ::= { publicKey 11 }

kyber-512      OBJECT IDENTIFIER ::= { kyber 1 }
kyber-768      OBJECT IDENTIFIER ::= { kyber 2 }
kyber-1024     OBJECT IDENTIFIER ::= { kyber 3 }
kyber-512-90s  OBJECT IDENTIFIER ::= { kyber-90s 1 }
kyber-768-90s  OBJECT IDENTIFIER ::= { kyber-90s 2 }
kyber-1024-90s OBJECT IDENTIFIER ::= { kyber-90s 3 }

xmss        OBJECT IDENTIFIER ::= { publicKey 8 }

-- The current dilithium implementation is compatible with the reference
-- implementation commit 3e9b9f1412f6c7435dbeb4e10692ea58f181ee51
dilithium     OBJECT IDENTIFIER ::= { publicKey 9 }
dilithium-aes OBJECT IDENTIFIER ::= { publicKey 10 }

dilithium-4x4     OBJECT IDENTIFIER ::= { dilithium 1 }
dilithium-6x5     OBJECT IDENTIFIER ::= { dilithium 2 }
dilithium-8x7     OBJECT IDENTIFIER ::= { dilithium 3 }
dilithium-aes-4x4 OBJECT IDENTIFIER ::= { dilithium-aes 1 }
dilithium-aes-6x5 OBJECT IDENTIFIER ::= { dilithium-aes 2 }
dilithium-aes-8x7 OBJECT IDENTIFIER ::= { dilithium-aes 3 }

SphincsPlus OBJECT IDENTIFIER ::= { publicKey 12 }

SphincsPlus-shake  OBJECT IDENTIFIER ::= { SphincsPlus 1 }
SphincsPlus-sha2   OBJECT IDENTIFIER ::= { SphincsPlus 2 }
SphincsPlus-haraka OBJECT IDENTIFIER ::= { SphincsPlus 3 }
```

(continues on next page)

(continued from previous page)

```
SphincsPlus-shake-128s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-shake256 1 }
SphincsPlus-shake-128f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-shake256 2 }
SphincsPlus-shake-192s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-shake256 3 }
SphincsPlus-shake-192f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-shake256 4 }
SphincsPlus-shake-256s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-shake256 5 }
SphincsPlus-shake-256f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-shake256 6 }

SphincsPlus-sha2-128s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-sha256 1 }
SphincsPlus-sha2-128f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-sha256 2 }
SphincsPlus-sha2-192s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-sha256 3 }
SphincsPlus-sha2-192f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-sha256 4 }
SphincsPlus-sha2-256s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-sha256 5 }
SphincsPlus-sha2-256f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-sha256 6 }

SphincsPlus-haraka-128s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-haraka 1 }
SphincsPlus-haraka-128f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-haraka 2 }
SphincsPlus-haraka-192s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-haraka 3 }
SphincsPlus-haraka-192f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-haraka 4 }
SphincsPlus-haraka-256s-r3.1 OBJECT IDENTIFIER ::= { SphincsPlus-haraka 5 }
SphincsPlus-haraka-256f-r3.1  OBJECT IDENTIFIER ::= { SphincsPlus-haraka 6 }

HSS-LMS-Private-Key OBJECT IDENTIFIER ::= { publicKey 13 }

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 }

ellipticCurve OBJECT IDENTIFIER ::= { randombit 4 }

numsp256d1     OBJECT IDENTIFIER ::= { ellipticCurve 1 }
numsp384d1     OBJECT IDENTIFIER ::= { ellipticCurve 2 }
```

(continues on next page)

(continued from previous page)

```
numsp512d1     OBJECT IDENTIFIER ::= { ellipticCurve 3 }

-- These are just for testing purposes internally in the library
-- and are not included in oids.txt
sm2test     OBJECT IDENTIFIER ::= { ellipticCurve 5459250 }
iso18003    OBJECT IDENTIFIER ::= { ellipticCurve 18003 }
```

18.10 Checklist For Next Major Version

- Remove most/all explicitly deprecated modules, interfaces, and features. Check deprecated.rst plus BOTAN_DEPRECATED annotations.
- Make the remaining PasswordHash interfaces internal
- Remove EC_Point/CurveGFp

18.10.1 Big Project: Public Key Split

Some complications of this aren't going to become clear until we get into it...

A number of operations currently defined on Public_Key can be moved to Asymetric_Key, for example key_length and algorithm_identifier.

Due to Private_Key deriving from Public_Key, the fingerprint functions are oddly named. Otherwise we can't correctly disambiguate sk->fingerprint(); should this be the fingerprint of the public or private key. With the split we can move this to Asymetric_Key::fingerprint and know that the correct thing happens.

The public and private key encoding functions (pkcs8.h, x509_key.h) are also complicated by the combined keys. For example we have to use PKCS8::PEM_encode(key) because key.PEM_encode() would be ambigious (similar situation as with the fingerprint APIs currently). Once the key types are split, we can move all of this to the key types themselves, or again (for the shared cases, like unencrypted PEM) to Asymetric_Key.

Decoding also can become simpler. We could consider moving to a model that doesn't use DataSource? Maybe just a span even?

Put _ prefixes on all of the internal operations getters (create_signature_op, etc)

18.11 Reading List

These are papers, articles and books that are interesting or useful from the perspective of crypto implementation.

18.11.1 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

18.11.2 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

18.11.3 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?

18.12 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.

18.12.1 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.

18.12.2 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 were made internal in 3.0]

18.12.3 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`, ...

18.12.4 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.

Alternatively, and possibly more practically, there should have not been any exception hierarchy (or at least not one visible to users) - instead only the high level Exception type with contains an error type enum.

18.12.5 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.

18.12.6 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.)

18.12.7 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.

18.12.8 License

MIT is more widely used and doesn't have the ambiguity surrounding the various flavors of BSD.