Federal Office
for Information Security

A development project commissioned by the Federal Office for Information Security (BSI)

Project 197

Secure Implementation of a Universal Crypto Library

Architecture Description

Version 1.3.0 / 2018-05-07
Botan 2.4.0-RSCS1

ROHDE&SCHWARZ
Cybersecurity

HACKMANIT

Summary

The objective of this project is the secure implementation of a universal crypto library which contains all common cryptographic primitives that are necessary for the wide use of cryptographic operations. These include symmetric and asymmetric encryption and signature methods, PRFs, hash functions and RNGs. Additionally, security standards such as X.509 and SSL/TLS have to be supported. The library will be provided to manufacturers of VS-NfD products which will help the Federal Office for Information Security (BSI) to evaluate these products.

This document describes the architecture of Botan.

Authors

Daniel Neus (DN)
René Korthaus (RK)

Copyright

Secure Implementation of a Universal Crypto Library

Architecture Description

Botan 2.4.0-RSCS1

Changelog

Version	Authors	Comment	Date
1.0.0	DN, RK	Initial version	2016-11-11
1.1.0	RK, DN	Add CLI invocation example Fixes in labels for listings in chapter 8 Update configure.py options chapter Update chapter 8 with new test files Update BSI module policy Language fixes found during a review	2017-01-09
1.2.0	RK	Add section on RandomNumberGenerator interface Update to 2.0.1-RSCS1	2017-03-02
1.3.0	RK	Update to 2.4.0-RSCS1	2018-05-07

Table of Contents

1 Introduction

This document describes the architecture of Botan. Botan consists of three main parts: the library itself, a CLI tool, and the test suite. The library itself is divided into separate modules. Botan uses a self-written build system - explained in more detail in chapter 3 - which, e.g., allows to select which modules are compiled when building Botan.

2 Folder structure

This chapter describes the most important parts of the Botan folder structure. The root of the Botan folder structure is listed below. Folders are in bold type.

File-/Foldername	Description
`doc`	Contains the handbook and more general information such as PGP keys of the maintainer, a roadmap or a todo list for example.
`src`	Contains all the source code and the build system data.
`configure.py`	The main build system script that has to be executed before Botan can be compiled. See section 3.1.
`license.txt`	License information
`news.rst`	Release notes
`readme.rst`	A readme describing the most important parts of Botan to get started quickly.
`authors.txt`	Information about contributors

Furthermore, the `src` folder has the following structure:

Foldername	Description
`build-data`	All build system related files.
`cli`	Source code of the command line interface tool.
`contrib`	Currently contains a Perl wrapper and a patch for encrypted sqlite3 database storage.
`fuzzer`	Tests for fuzzing various message decoders and math functions using AFL[1], libFuzzer[2] and Klee[3].
`lib`	The library source code.
`python`	Python bindings
`scripts`	Build scripts mainly used for continuous

1 http://lcamtuf.coredump.cx/afl/
2 http://llvm.org/docs/LibFuzzer.html
3 https://klee.github.io/

	integration.
`tests`	The Botan test suite.

2.1 Structure of the Library Code

Botan has a modular design which is represented by the folder structure inside the `lib` folder. Each folder containing an `info.txt` file is a module that can be included or excluded from the build. More details about the build system and the Botan modules are described in chapter 3, especially 3.3. The root structure of the `lib` folder is shown below:

Foldername	Description
`asn1`	All ASN1 related code like BER or DER encoding and decoding.
`base`	Helper classes and basics like the secure_allocator. This module can't be excluded from the build.
`block`	Block ciphers like AES, DES or Blowfish.
`codec`	Currently base64 and hex encoding.
`compression`	Source code for the BZip2, LZMA and ZLIB compression formats.
`entropy`	Entropy sources like RdRand, CryptGenRandom() or /dev/random.
`ffi`	Foreign function interface: C wrapper
`filters`	Message processing using filters and pipes
`hash`	Hash functions
`kdf`	Key derivation functions
`mac`	Message authentication codes
`math`	Big integers, multi precision arithmetic
`misc`	Everything that does not fit into one of the other categories
`modes`	All block cipher modes

`passhash`	Password hashing algorithms
`pbkdf`	Password based key derivation
`pk_pad`	Public key padding mechanisms
`prov`	Botan can use external libraries (providers) to exchange certain own algorithms with algorithms from third party libraries like OpenSSL. Furthermore, it is possible to “disable” Botan's software implementation and run the crypto inside a TPM or via PKCS#11 in a smart card or a HSM. See chapter 6 for more information on this topic.
`pubkey`	All public key algorithms
`rng`	All random number generators
`stream`	All stream ciphers
`tls`	All TLS related source code
`utils`	Utilities like a certificate store, character set conversion, date time conversion functions and so on.
`x509`	All X.509 certificate related source code

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. For example, the `codec` folder contains the subfolders `base64` and `hex`. Both subfolders have source files and an `info.txt` file. So, these are both single modules, whereas the `codec` folder does not contain source code or an `info.txt` file and is not a module.

3 The Build System

To compile the Botan library it is necessary to configure the build first by executing the `configure.py` python script. The script creates various directories, configuration files and the Makefile that is necessary to compile the library for the specified configuration.

The script tries to detect various options like target CPU, OS and compiler automatically. These options can be overridden if the automatic detection fails. The following subsection gives more details about the `configure.py` script.

3.1 Configure.py Command Line Options

The `configure.py` script has various options which can be grouped into parser, target, build, module and installation specific options.

3.1.1 Parser Options

The parser options control how much output is generated during execution of the `configure.py` script and the tools that are invoked by the script itself, like the compiler.

Parameter	Definition
`--verbose`	Shows debug messages. Default: disabled
`--quiet`	Shows only warnings and errors. Default: disabled

3.1.2 Target Options

Target options control for which target architecture the library should be compiled.

Parameter	Definition
`--cpu {alpha, arm32, arm64, hppa, ia64, m68k, mips32, mips64, ppc32, ppc64, s390, s390x, sparc32, sparc64, superh, x86_32, x86_64}`	Sets the target CPU type/model. There are also a lot of aliases defined. So, for example it's possible to specify "x64" or "sandybridge". Supported architectures are described in "`src\build-data\arch`". Section 3.3.2System Architecture Description on page 26 gives more details on this topic.
`--os {aix, android, cygwin, darwin, dragonfly, freebsd, haiku, hpux, hurd, includeos, ios, irix, linux, mingw, nacl, netbsd, openbsd, qnx, solaris,`	Sets the target operating system. Supported OSes are defined in "`src\build-data\os`". See section Operating System Description on page 29 for more information.

Parameter	Definition
`windows}`	
`--cc {clang, ekopath, gcc, hpcc, icc, msvc, pgi, sunstudio, xlc}`	Sets the desired build compiler. Supported compilers are defined in "`src\build-data\cc`". See section Compiler Description on page 27 for details.
`--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.
`--cc-bin BINARY`	Set path to compiler binary
`--cc-abi-flags FLAG`	Set compiler ABI flags.
`--cxxflags=FLAG`	Set compiler flags.
`--ldflags=FLAG`	Set linker flags.
`--ar-command=AR`	Set path to static archive creator.
`--with-endian {little, big}`	Botan tries to guess the byte order automatically. Can be overwritten with this option.
`--with-unaligned-mem,` `--without-unaligned-mem`	Use/don't use unaligned memory access.
`--with-os-features FEAT, --without-os-features FEAT`	Specify OS features to use/to disable. For example "cryptgenrandom" or "posix_mlock".
`--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-aes-ni`	Disable aes-ni intrinsics
`--disable-sha`	Disable SHA intrinsics
`--disable-altivec`	Disable altivec intrinsics
`--disable-neon`	Disable NEON intrinsics

3.1.3 Build Options

The build options specify for example if debug output should be generated.

Parameter	Definition
`--with-debug-info`	Enable debug info. Default: disabled
`--debug-mode`	Same as `--with-debug-info --no-optimizations`.
`--with-sanitizers`	Enable run time checks. Default: disabled
`--without-stack-protector`	Disable stack smashing protections
`--with-coverage`	Enable test coverage calculation and disable optimizations. Default: disabled
`--with-coverage-info`	Enable test coverage calculation. Optimizations enabled. Default: disabled
`--enable-shared-library`, `--disable-shared-library`	Enable/disable building shared library. Building shared library is the default option.
`--enable-static-library`, `--disable-static-library`	Enable/disable building static library. Building static library is disabled by default.
`--optimize-for-size`	Informs the build that a smaller binary is preferable. May impact performance. Default: disabled
`--no-optimizations`	Disables all optimizations. Default: disabled
`--amalgamation`	Create only a few big header/source files containing the library code: - botan_all.h - botan_all.cpp - botan_all_internal.h - botan_all_aesni.cpp - botan_all_ssse3.cpp - botan_all_rdrand.cpp - botan_all_rdseed.cpp - botan_all_avx2.cpp And use these files to build the library. Default: disabled
`--single-amalgamation-file`	Used in combination with "`--`

Parameter	Definition
	`amalgamation`" to build a single file instead of splitting on ABI: - botan_all.h - botan_all_internal.h - botan_all.cpp Default: disabled
`--with-build-dir DIR`	Setup the build in `DIR`
`--with-external-includedir DIR`	Allows to specify a path to external includes (like Boost or OpenSSL). This is especially needed on Windows where external libraries are not accessible via a default include path like on Linux for example.
`--with-external-libdir DIR`	Allows to specify a patch to external libs (like Boost or OpenSSL). This is especially needed on Windows where external libraries are not accessible via a default library path like on Linux for example
`--with-openmp`	Enable use of OpenMP.
`--link-method {symlink, hardlink, copy}`	Choose how links to include headers are created inside the build dir.
`--with-local-config FILE`	Include the contents of `FILE` into build.h (see also 3.4)
`--distribution-info STRING`	Set a distribution specific version. Generates "`BOTAN_DISTRIBUTION_INFO`" macro in build.h (see 3.4)

`--with-visibility,`
`--without-visibility`
	Turn on/off visibility[4] control. Default: activated

`--maintainer-mode`	Enable extra compiler warnings. `--with-sanitizers` is also activated. Default: disabled
`--dirty-tree`	Cleans build tree. Default: activated
`--with-python-versions N.M`	Where to install botan.py (the python wrapper/bindings). `N.M` = 2.7 for example.

4 https://gcc.gnu.org/wiki/Visibility

Parameter	Definition
`--with-valgrind`	Uses valgrind for dynamic analysis.
`--with-bakefile`	Generates a bakefile which can be used to create Visual Studio or Xcode project files for example.
`--unsafe-fuzzer-mode`	Disables essential checks for testing.
`—-with-stack-protector,` `--without-stack-protector`	Enable/disable stack smashing protections. Default: enabled

3.1.4 Documentation Options

Documentation options control how documentation is generated.

Parameter	Definition
`--with-documentation,` `--without-documentation`	Enable/disable building/installing the documentation. Default: enabled
`--with-sphinx,` `--without-sphinx`	Generate Sphinx[5] documentation.
`--with-pdf,` `--without-pdf`	Use Sphinx to generate a PDF document.
`--with-rst2man` `--without-rst2man`	Use rst2man[6] to generate man pages.
`--with-doxygen,` `--without-doxygen`	Generate Doxygen documentation.

3.1.5 Module Options

Everything that influences which modules are included in the build is controlled by these module options.

Parameter	Definition
`--module-policy`	Allows to specify a module policy file which defines required and prohibited modules and additionally modules that should be activated if available. There are currently two policies shipped with Botan: A `modern` policy and a

5 A tool to generate documentation in different output formats, see http://www.sphinx-doc.org.
6 http://docutils.sourceforge.net/sandbox/manpage-writer/rst2man.txt

Parameter	Definition
	`bsi` policy. Section 3.2 explains more details about this topic.
`--enable-modules`	Enable specific modules. Example: `--enable-modules "aes,tpm,rdrand"`
`--disable-modules`	Disable specific modules. Example: `--disable-modules "aes,tpm,rdrand"`
`--list-modules`	List available modules
`--no-autoload`	Only the core modules will be included. Default: deactivated
`--minimized-build`	Same as `--no-autoload`
`--with-bearssl,` `--without-bearssl`	Enables/disables `bearssl` provider. Adds an engine that uses BearlSSL[7] for ECDSA and some hash functions.
`--with-boost,` `--without-boost`	Enables/disables boost module
`--with-bzip2,` `--without-bzip2`	Enables/disables `bzip2` module for BZip2 compression/decompression. Requires BZip2 development libraries to be installed.
`--with-lzma,` `--without-lzma`	Enables/disables `lzma` module for LZMA compression/decompression. Requires LZMA development libraries to be installed.
`--with-openssl,` `--without-openssl`	Enables/disables `openssl` provider. Adds an engine that uses OpenSSL for some public key operations and ciphers/hashes. See chapter 6.1 for a description of this provider.
`--with-sqlite3,` `--without-sqlite3`	Enables/disables `sqlite3` module. Enables storing TLS session information to an encrypted SQLite database.
`--with-zlib,` `--without-zlib`	Enables/disables `zlib` module for Zlib compression/decompression. Requires Zlib development libraries to be installed.
`--with-tpm,`	Enables/disables the `tpm` module. The TPM

7 https://bearssl.org/

Parameter	Definition
`--without-tpm`	module allows to access a Trusted Platform Module via the external Trousers library.
`--with-everything`	Build all modules, except the ones marked as "load_on never" (see section 3.3.1). Default: disabled

3.1.6 Installation Options

Installation options control where the library should be installed.

Parameter	Definition
`--program-suffix SUFFIX`	Append `SUFFIX` to program names
`--prefix DIR`	Set the install prefix
`--docdir DIR`	Set the documentation install directory
`--bindir DIR`	Set the binary install directory
`--libdir DIR`	Set the library install directory
`--mandir DIR`	Set the man pages install directory
`--includedir DIR`	Set the include file install directory

3.1.7 Miscellaneous Options

Additional options that do not fit into one of the other option categories.

Parameter	Definition
`--house-curve`	Adds a custom in-house ECC curve of the format “curve.pem,NAME,OID,CURVEID”.

3.2 Module Policy

Module policy files define:

- Modules that must be activated.
- Modules that must be activated if available.
    - For example the `aesni` module is not compatible with all platforms.
- Modules that have to be disabled.

A module that is not mentioned in the module policy will only be enabled if it is a dependency of one of the modules that are activated by the policy or if it is explicitly enabled via the "`--enable-modules`" option.

The module policy itself is enforced by calling `configure.py` with the `--module-policy` parameter. There are currently three policies shipped with Botan. A "`modern`" policy that contains only modern algorithms, a "`nist`" policy that contains only NIST-approved algorithms and a "`bsi`" policy that represents the recommendations from BSI technical guideline TR-02102-1. The module policy files are located in `src\build-data\policy`. The `bsi` module policy is listed in the Appendix.

3.3 Build System Internals

Botan uses text files for describing information on Botan modules, supported architectures, supported compilers and supported operating systems.

3.3.1 Module Description

Each module must have an info.txt file which contains various information about the module. An info.txt file consists of group parameters and key-value pairs.

Group Parameters

For each group parameter, there are multiple entries possible. Each entry has to be placed on a new line. For example:

```
<source>
serpent.cpp
aes.cpp
</source>
```

This example adds the two source files serpent.cpp and aes.cpp to the list of files that should be compiled.

The following parameters are optional:

Parameter	Examples	Definition
`<source></source>`	`serpent.cpp`	Source files
`<header:internal></header:internal>`	`serpent_sbox.h`	Internal header
`<header:public></header:public>`	`serpent.h`	Public header
`<header:external></`	`pkcs11.h`	External header

`header:external>`		
`<requires></requires>`	`hash, modes`	Required modules (dependencies)
`<os></os>`	`windows, cygwin`	Only available on specific OS
`<arch></arch>`	`x86_64`	Only available on specific architecture
`<cc></cc>`	`gcc, clang`	Only available if targeting specific compiler
`<libs></libs>`	`windows -> advapi32.lib`	Required external libraries
`<frameworks></frameworks>`	`darwin -> Security`	Required external frameworks
`<comment></comment>`	`"Loading Module aes_ni"`	Shows informative comment if module is included in build during execution of `configure.py`
`<warning></warning>`	`"Deprecated Module"`	Shows warning if module is included in build during execution of `configure.py`

Key-Value Pairs

Key-Value pairs only consist of one key word and one value assigned to this.

Parameter	Examples	Definition
`load_on KEYWORD`	`auto (default), request, vendor, dep, always, never`	`auto`: loaded unless "`--no-autoload`" or "`--minimized-build`" is set `request`: only when explicitly set on command line `vendor`: requires external dependency `dep`: can only be loaded if needed by dependency `always`: load always `never`: loading disabled
`<defines></defines>`	`AES_NI -> 20131128`	API-version of the module is increased on each API change. Can be checked from

		application code during compile time (see 3.4).
`need_isa INTRINSIC`	`sse2, ssse3, avx2, aes-ni, altivec`	Required CPU intrinsic
`mp_bits BITS`	`64, default: 0`	Required architecture word size for this module

"`<defines>MODULNAME -> API-VERSION</defines>`" is the only mandatory parameter. See appendix: Module Description Example for a concrete module file example.

3.3.2 System Architecture Description

Each supported architecture must have a text file with architecture specific information in `src/build-data/`arch consisting of group parameters and key-value pairs.

Group Parameters

For each group parameter, there are multiple entries possible. Each entry has to be placed in a new line. See the example in section 3.3.1 for more information.

Parameter	Examples	Definition
`<aliases></aliases>`	`amd64, x64`	Names under which the architecture is known.
`<submodels></submodels>`	`atom, core2, sandybridge, corei7`	Processor models that implement the architecture

`<submodel_aliases></submodel_aliases>`	`core2duo -> core2`
	`nehalem -> corei7`
		Names under which the processor models are also known

`<isa_extensions></isa_extensions>`	`sse2, ssse3, sse4.1, aesni, rdrand`	Instruction set extensions available on this architecture

Key-Value Pairs

Key-Value pairs only consist of one key word and one value assigned to this.

Parameter	Examples	Definition
`endian`	`big, little`	Endianness of the architecture
`unaligned`	`no (default), ok`	Unaligned memory access supported?
`wordsize`	`32 (default), 64`	Word size of the architecture

`family`	`x86, sparc, ppc`	Architecture family name

See appendix: Architecture Description Example for a concrete architecture description example.

3.3.3 Compiler Description

Each supported compiler must have a text-file with compiler specific information in `src/build-data/cc`, again consisting of group parameters and key-value pairs.

Group Parameters

For each group parameter, there are multiple entries possible. Each entry has to be placed in a new line. See the example in section 3.3.1 for more information.

Parameter	**Examples**	**Definition**

`so_link_commands`	`default       -> "$(CXX) -shared -fPIC -Wl,-soname,$(SONAME_ABI)"`
	
	`default-debug -> "$(CXX) -shared -fPIC -Wl,-soname,$(SONAME_ABI)"`
	
	`solaris -> "$(CXX) -shared -fPIC -Wl,-h,$(SONAME_ABI)"`
	
	`openbsd -> "$(CXX) -shared -fPIC"`
		Command needed to link shared object

`binary_link_commands`	`linux         -> "$(LINKER) -Wl,-rpath=\$$ORIGIN"`
	
	`linux-debug   -> "$(LINKER) -Wl,-rpath=\$$ORIGIN"`
	
	`default       -> "$(LINKER)"`
	
	`default-debug -> "$(LINKER)"`
		Command needed to link application object

`mach_opt`	`i386       -> "-mtune=generic"`
	
	`nehalem     -> "-`
		Machine specific options which are passed to the compiler

	`march=corei7"`
	
	`sandybridge -> "-march=corei7-avx"`
	
	`x86_32    -> "-march=SUBMODEL"`	

`mach_abi_linking`	`all     -> "-pthread -fstack-protector"`
	
	`x86_64  -> "-m64"`
		Machine specific ABI flags. Flags set here are included at compile and link time

`isa_flags`	`sse2    -> "-msse2"`
	`sse4.1  -> "-msse4.1"`
	`aesni   -> "-maes -mpclmul -mssse3"`
	`rdrand  -> "-mrdrnd"`
		Instruction set extension flags

Key-Value Pairs

Key-Value pairs only consist of one key word and one value assigned to this.

Parameter	Examples	Definition
`binary_name`	`g++`	Binary name of the compiler
`linker_name`	`link`	Binary name of the linker
`macro_name`	`GCC`	Name of the compiler
`output_to_exe`	`-o`	Output option of the compiler. Default: `-o`
`add_include_dir_option`	`-I`	Include directory option of the compiler. Default: `-I`
`add_lib_dir_option`	`-L`	Library directory option of the compiler. Default: `-L`
`add_lib_option`	`-l`	Library file option of the compiler. Default: `-l`
`add_framework_option`	`-framework`	Framework option of the compiler. Default: `-framework`
`preproc_flags`	`-E`	Flags for the preprocessor
`compile_flags`	`-c`	Compiler specific flag to compile a file

`debug_info_flags`	`-g`	Flag to generate debug information
`optimization_flags`	`-O2`	Flags for optimized code generation
`size_optimization_flags`	`-Os`	Flags for optimized code generation when optimizing for code size
`coverage_flags`	`--coverage`	Flags to generate coverage data
`sanitizer_flags`	`-D_GLIBCXX_DEBUG -fsanitize=address`	Flags to activate sanitizer
`stack_protector_flags`	`-fstack-protector-all`	Flags to enable stack smashing protection
`shared_flags`	`-fPIC`	Flags for dynamic library generation
`lang_flags`	`-std=c++11 -D_REENTRANT`	Language specific flags
`warning_flags`	`-Wall -Wextra`	Default Warning flags
`maintainer_warning_flags`	`-Wold-style-cast -Werror`	Activated warnings if `--maintainer-mode` is specified
`visibility_build_flags`	`-fvisibility=hidden`	Flags to control symbol visibility
`visibility_attribute`	`__attribute__((visibility("default")))`	Compiler specific visibility attributes
`ar_command`	`pathCC -ar -o`	Command used to create library archives
`ar_options`	`-ar -o`	Options of the static archive builder.
`ar_output_to`	`/OUT:`	Output option of the static archive builder.

See appendix: Compiler Description Example for a concrete compiler description example.

3.3.4 Operating System Description

Each supported operating system must have a text file with OS specific information in `src/build-data/os` consisting of group parameters and key-value pairs.

Group Parameters

For each group parameter, there are multiple entries possible. Each entry has to be placed in a new line. See the example in section 3.3.1 for more information.

Parameter	Examples	Definition
`<aliases></aliases>`	`win32, MSWin32`	Names under which the OS is also known
`<target_features></target_features>`	`cryptgenrandom, virtual_lock, rtlsecurezeromemory`	Features/API-calls that are available on this OS. Generates macros in Build.h (see 3.4). For example: `BOTAN_TARGET_OS_HAS_CRYPTGENRANDOM`

Key-Value Pairs

Key-Value pairs only consist of one key word and one value assigned to this.

Parameter	Examples	Definition
`os_type`	`windows`	To which OS family does the OS belong. For example Android, Linux, Darwin, Cygwin, Hurd, OpenBSD all have `os_type = unix` Default: `None`
`program_suffix`	`exe`	File extension for executables
`obj_suffix`	`obj`	File extension for object files. Default: `o`
`soname_suffix`	`so`	File extension of dynamic shared libraries.
`soname_pattern_patch`	`"libbotan-{version_major}.{version_minor}.so.{abi_rev}.{version_patch}"`	Shared library name with ABI version and patch level information. For example: libbotan-1.11.so.24.24
`soname_pattern_abi`	`"libbotan-{version_major}.{version_minor}.so.{abi_rev}"`	Shared library name with ABI version information. For example: libbotan-1.11.so.24
`soname_pattern_base`	`"libbotan-{version_major}.{version_minor}.so"`	Base shared library name. For example: libbotan-1.11.so
`static_suffix`	`lib`	File extension for static libraries. Default: `a`
`ar_command`	`"ar"`	Command used to create library archives. Default: `ar`
`ar_options`	`crs`	Command options passed to the `ar_command`
`ar_output_to`	`/OUT:`	Output option of the static archive builder.
`install_root`	`c:\\Botan`	Default install path. Default:

		`/usr/local`
`header_dir`	`include`	Default name of the folder where the header files are stored. Default: `include`
`bin_dir`	`bin`	Default name of the folder where the binary files are stored. Default: `bin`
`lib_dir`	`lib`	Default name of the folder where the library files are stored. Default: `lib`
`doc_dir`	`docs`	Default name of the folder where the documentation files are stored. Default: `share/doc`
`man_dir`	`man`	Default name of the folder where the documentation files are stored. Default: `share/man`
`use_stack_protector`	`true`	Whether tp use the stack smashing protector. Default: `true`
`so_post_link_command`	`install_name_tool -change … …`	Specifies an additional post-link command to be executed after linking.
`cli_exe_name`	`"botan"`	Name of the command line tool executable. Default: `botan`
`lib_prefix`	`"lib"`	Prefix for shared libraries. Default: `lib`
`library_name`	`“botan-{major}”`	Name of the shared library. Default: `botan-{major}`

See appendix: Operating System Description Example for a concrete operating system description example.

3.4 Build.h

The build.h header file is generated and overwritten each time the `configure.py` script is executed. This is done by extending and modifying the template file "`src/build-data/buildh.in`". The build.h header file can be included in a Botan header or source file.

This header file is helpful in order to verify which modules are included in the currently used version of the library. This information can be derived from the macros that begin with "`BOTAN_HAS`" followed by the module name. For example the macro "`BOTAN_HAS_TLS 20150319`" that shows the application developer that TLS support is enabled and which API-version is included.

There are a lot more configuration macros present. The most important ones are listed below:

3.4.1 Informational Parameters

Option	Definition
`BOTAN_VERSION_MAJOR`	Botan major version number
`BOTAN_VERSION_MINOR`	Botan minor version number
`BOTAN_VERSION_PATCH`	Botan patch level
`BOTAN_VERSION_DATESTAMP`	Expands to an integer of the form YYYYMMDD if this is an official release, or 0 otherwise
`BOTAN_VERSION_RELEASE_TYPE`	unreleased / released
`BOTAN_VERSION_VC_REVISION`	Git commit SHA1 hash of the release
`BOTAN_DISTRIBUTION_INFO`	A macro expanding to a string that is set at build time using the `--distribution-info` option

3.4.2 Configurable Options

The following parameters are configurable after execution of the `configure.py` script inside the build.h header file.

General Options

Option	Definition
`BOTAN_DEFAULT_BUFFER_SIZE`	Default value: 1024. How much to allocate for a buffer of no particular size
`BOTAN_MLOCK_ALLOCATOR_MIN_ALLOCATION`	Default value: 16. Minimum size to allocate out of the mlock[8] pool in bytes
`BOTAN_MLOCK_ALLOCATOR_MAX_ALLOCATION`	Default value: 128. Maximum size to allocate out of the mlock pool in bytes
`BOTAN_MLOCK_ALLOCATOR_MAX_LOCKED`	Default: 512. Total maximum amount of RAM

8 A memory pool that is locked into volatile memory on supported platforms, see section 4.4.

`_KB`	(in KiB) that is locked into memory, even if the OS allows more
`BOTAN_BLOCK_CIPHER_PAR_MULT`	Default: 4. Multiplier on a block cipher's native parallelism
`BOTAN_MP_WORD_BITS`	How many bits per limb in a BigInt
`BOTAN_USE_VOLATILE_MEMSET_FOR_ZERO`	Default: 1. If enabled uses memset via volatile function pointer to zero memory, otherwise does a byte at a time write via a volatile pointer.

Blinding Options

Option	Definition
`BOTAN_POINTGFP_USE_SCALAR_BLINDING`	Default: 1. If enabled the ECC implementation will use scalar blinding with order.bits()/2 bit long masks.
`BOTAN_POINTGFP_RANDOMIZE_BLINDING_BITS`	Default: 80. Set number of bits used to generate mask for blinding the representation of an ECC point. Set to zero to disable this side-channel countermeasure.
`BOTAN_BLINDING_REINIT_INTERVAL`	Default: 32. Normally blinding is performed by choosing a random starting point (plus its inverse, of a form appropriate to the algorithm being blinded), and then choosing new blinding operands by successive squaring of both values. This is much faster than computing a new starting point but introduces some possible correlation. To avoid possible leakage problems in long-running processes, the blinder periodically reinitializes the sequence. This value specifies how often a new sequence should be started.

Random Number Generation Options

Option	Definition
`BOTAN_RNG_DEFAULT_RESEED_INTERVAL`	Default: 1024. Enforce reseed after asking the RNG X times for randomness.
`BOTAN_RNG_RESEED_POLL_BITS`	Default: 256. Number of bits of entropy to attempt to gather from the entropy sources
`BOTAN_RNG_AUTO_RESEED_TIMEOUT`	Default: 10 milliseconds. Stops automatic reseeding after X milliseconds even if not

Option	Definition
	enough entropy is collected.
`BOTAN_RNG_RESEED_DEFAULT_TIMEOUT`	Default: 50 milliseconds. Stops manual reseeding after X milliseconds even if not enough entropy is collected.
`BOTAN_ENTROPY_DEFAULT_SOURCES`	Specifies (in order) the list of entropy sources that will be used to seed an in-memory RNG: `"rdseed", "rdrand", "darwin_secrandom", "dev_random", "system_rng", "proc_walk", "system_stats"`
`BOTAN_SYSTEM_RNG_DEVICE`	Controls the RNG used by the system RNG interface. For example: "`/dev/urandom`"
`BOTAN_SYSTEM_RNG_POLL_DEVICES`	List of devices that are polled for randomness if “`/dev/random`” is used as System_RNG. Default: "`/dev/urandom", "/dev/random", "/dev/srandom`"
`BOTAN_ENTROPY_PROC_FS_PATH`	Default: `/proc`. Directory which will be monitored by the `ProcWalking_EntropySource` and the contents provided to the RNG. Set to empty string to disable.
`BOTAN_SYSTEM_RNG_POLL_REQUEST`	Default: 64. How many bytes to read from the system PRNG.
`BOTAN_SYSTEM_RNG_POLL_TIMEOUT_MS`	Default: 20. Maximum block time in ms even if not enough entropy is collected.
`BOTAN_ENTROPY_INTEL_RNG_POLLS`	Default: 32. Specifies how many times to read from the RdRand/RdSeed RNG. Each read generates 32 bits of output.
`BOTAN_ENTROPY_RDRAND_RETRIES`	Default: 10. Within each poll there are at maximum 10 retries to make a new poll to the RNG. According to Intel, RdRand is guaranteed to generate a random number within 10 retries on a working CPU.
`BOTAN_ENTROPY_RDSEED_RETRIES`	Default: 20. RdSeed is not guaranteed to generate a random number within a specific number of retries.

3.5 Makefile

The Makefile is generated from a template in `src/build-data/makefile`. Currently only Nmake and Gmake are supported. Nmake is used on Windows with Visual C++ and Gmake on all other platforms.

The following targets are supported:

Target	**Definition**
`nmake/make all`	Compiles library, CLI and test suite
`nmake/make`	same as "`nmake/make all`"

`nmake/make botan`
`nmake/make libs`
	Compiles the library

`nmake/make botan-test`
`nmake/make tests`
	Compiles the test suite

`nmake/make cli`	Compiles the CLI
`nmake/make install`	Install Botan to default directory
`nmake/make docs`	Generate documentation and also API-documentation if `--with-sphinx` or `--with-doxygen` was used during configure stage
`nmake/make clean`	Cleans output of compilation
`nmake/make distclean`	Same as "`clean`" + remove output of `configure.py` artifacts
`make valgrind`	Runs test in valgrind environment

4 Interfaces

Botan provides specific interfaces for algorithms and operations. In the following, an overview of the most important interfaces is given. For the basic cryptographic primitives, these are:

Interface Name	Defined in
`BlockCipher`	`block_cipher.h`
`StreamCipher`	`stream_cipher.h`
`HashFunction`	`hash.h`
`MessageAuthenticationCode`	`mac.h`
`KDF`	`kdf.h`
`PBKDF`	`pbkdf.h`

Other commonly used interfaces are:

Interface Name	Definiton in	Description
`Cipher_Mode`	`cipher_mode.h`	Common interface of all cipher modes.
`AEAD_Mode`	`aead.h`	Interface for AEAD (Authenticated Encryption with Associated Data) modes. Inherits from Cipher_Mode.
`Stream_Cipher_Mode`	`stream_mode.h`	Interface for stream cipher modes. Inherits from Cipher_Mode.
`BlockCipherModePaddingMethod`	`mode_pad.h`	Interface for block cipher mode padding methods
`EME`	`eme.h`	Message-encoding methods for encryption
`EMSA`	`emsa.h`	Message-encoding methods for signatures with appendix
`PK_Encryptor`	`pubkey.h`	Public key encryption
`PK_Decryptor`	`pubkey.h`	Public key decryption
`PK_Signer`	`pubkey.h`	Public key signing
`PK_Verifier`	`pubkey.h`	Public key verification

`PK_Key_Agreement`	`pubkey.h`	Public key key agreement
`PK_Encryptor_EME`	`pubkey.h`	Public key encryption paired with an encoding scheme
`PK_Decryptor_EME`	`pubkey.h`	Public key decryption paired with an encoding scheme
`PK_KEM_Encryptor`	`pubkey.h`	Public key key encapsulation mechanism encryption
`PK_KEM_Decryptor`	`pubkey.h`	Public key key encapsulation mechanism decryption
`Public_Key`	`pk_keys.h`	Interface for all public keys
`Private_Key`	`pk_keys.h`	Interface for all private keys
`PK_Key_Agreement_Key`	`pk_keys.h`	Interface for all key agreement keys. Inherits from Private_Key.
`RandomNumberGenerator`	`rng.h`	Interface for all cryptographic random number generators

4.1 Object Creation

Object creation is identical for all basic cryptographic primitives. This is explained in subsection 4.1.1. Other important factory functions with a different method signature are explained in subsection 4.1.2.

4.1.1 Common Factory Functions

Botan has a common way of creating an object that implements basic cryptographic primitives. For this purpose, two methods are defined as explained in the following:

T::create

To create an algorithm object, the following function is provided:

```
std::unique_ptr<T> T::create( const std::string& algo, const
std::string& provider = "" )
```

Parameters:

- `algo:` name of the algorithm
- `provider:` (optional) provider to use (see chapter 6)

Return value:

- `std::unique_ptr<T>` to the algorithm object or `nullptr` if the algorithm/provider combination cannot be found.

Examples:
- `BlockCipher::create( "AES-128" );`
- `BlockCipher::create( "AES-128", "base" );`
- `BlockCipher::create( "AES-256" );`
- `BlockCipher::create( "AES-256", "openssl" );`

The first and second example have the same effect. If no provider is specified, then the base provider is automatically selected.

T::create_or_throw

There is an additional function to create algorithm objects which in contrast to the previous one does not return a `nullptr` if the desired algorithm/provider combination cannot be found but instead throws a `Lookup_Error` exception. The function signature is the same as before:

```
std::unique_ptr<T> T::create_or_throw( const std::string& algo,
const std::string& provider = "" )
```

4.1.2 Other Factory Functions

Some of the other commonly used interfaces also provide factory functions to get an object for this mechanism:

Cipher modes

```
Cipher_Mode* get_cipher_mode( const std::string& algo, Cipher_Dir
direction );
```

Examples:

- `get_cipher_mode( "AES-128/XTS", ENCRYPTION );`
- `get_cipher_mode( "AES-128/CBC/NoPadding", ENCRYPTION );`
- `get_cipher_mode( "AES-128/CBC/PKCS7", DECRYPTION );`

Authenticated Encryption with Associated Data Modes

```
AEAD_Mode* get_aead( const std::string& name, Cipher_dir direction
);
```

Examples:
- `get_aead( "AES-128/GCM", ENCRYPTION );`
- `get_aead( "Serpent/EAX", DECRYPTION );`

Message-encoding Methods for Encryption

```
EME* get_eme( const std::string& algo_spec );
```

Examples:

- `get_eme( "PKCS1v15" );`
- `get_eme( "OAEP" );`
- `get_eme( "Raw" );`

Message-encoding Methods for Signatures with Appendix

```
EMSA* get_emsa( const std::string& algo_spec );
```

Example:

- `get_emsa( "PSSR" );`
- `get_emsa( "EMSA_PKCS1" );`
- `get_emsa( "Raw" );`

Block Cipher Padding Mode Methods

```
BlockCipherModePaddingMethod* get_bc_pad( const std::string&
algo_spec );
```

Examples:

- `get_bc_pad( "NoPadding" );`
- `get_bc_pad( "PKCS7" );`

4.2 Other Common Functions

Currently there is another function common to all basic cryptographic primitives:

T::providers

The interface can be called to get a list of available providers for a specific algorithm:

```
std::vector<std::string> T::providers( const std::string& algo )
```

Parameter:

- `algo`: name of the algorithm

Return value:

- vector of strings containing available providers for the desired algorithm or empty vector if no providers implementing this algorithm were found.

Example:

- `BlockCipher::providers( "AES-128" );` → Returns “base” and if Botan was configured with OpenSSL support, also “openssl”.

4.3 Random Number Generation

All interfaces in Botan that require cryptographically secure random numbers require passing a reference to a `Botan::RandomNumberGenerator` in their function signature. This has the advantage that it’s always clear where the random numbers come from respectively which random number generator is used. For example, the factory function to create a private key has the following signature:

```
std::unique_ptr<Private_Key> create_private_key( const
std::string& algo_name, RandomNumberGenerator& rng, const
std::string& algo_params="" );
```

By passing the random number generator reference it is transparent to the caller that the private key is generated from the random number generator the caller has passed as a reference to the function.

An application developer can use one of the random number generators provided by Botan, e.g., the HMAC_DRBG deterministic random number generator implemented in `src/lib/rng/hmac_drbg/hmac_drbg.cpp`. An application developer may also define a custom random number generator type and pass an instance to Botan interfaces that accept a `RandomNumberGenerator` reference.

All random number generators in Botan implement the `RandomNumberGenerator` interface. This interface provides the following important member functions. Functions marked with `= 0` are pure virtual and thus *must* be implemented by classes deriving from the `RandomNumberGenerator` interface.

- `virtual void randomize(uint8_t output[], size_t length) = 0`: Extracts *length* random bytes from the random number generator and writes the output to *output*.
- `virtual void add_entropy(const uint8_t input[], size_t length) = 0`: Incorporates *length* bytes of entropy from the input buffer *input* into the random number generator's entropy pool.
- `virtual void randomize_with_input(uint8_t output[], size_t output_len, const uint8_t input[], size_t input_len)`: Incorporates *input_len* bytes of entropy from the input buffer *input* into the random number generator's entropy pool and then extracts *output_len* random bytes from the random number generator and writes the output to *output*.
- `virtual void randomize_with_ts_input(uint8_t output[], size_t output_len)`: Incorporates a 64 bit system timestamp and a 64 bit processor timestamp into the random number generator's entropy pool and then extracts *output_len* random bytes

from the random number generator and writes the output to *output*.

- `virtual size_t reseed(Entropy_Sources& srcs, size_t poll_bits = BOTAN_RNG_RESEED_POLL_BITS, std::chrono::milliseconds poll_timeout = BOTAN_RNG_RESEED_DEFAULT_TIMEOUT)`: Polls the *entropy_sources* for up to *poll_bits* bits of entropy or until the *poll_timeout* expires, calls `add_entropy()` on this random generator and returns an estimate of the number of bits collected. The default value for *poll_bits* is `BOTAN_RNG_RESEED_POLL_BITS`, which defaults to 128. The default value for *poll_timeout* is `BOTAN_RNG_RESEED_DEFAULT_TIMEOUT`, which defaults to 50 milliseconds.
- `virtual void reseed_from_rng(RandomNumberGenerator& rng, size_t poll_bits = BOTAN_RNG_RESEED_POLL_BITS)`: Polls the *rng* for *poll_bits* bits of entropy and calls `add_entropy()` on this random generator. The default value for *poll_bits* is `BOTAN_RNG_RESEED_POLL_BITS`, which defaults to 128.

4.4 Secure Memory

Sensitive information is stored and processed in a special type `secure_vector`. This type is defined in the header file “`secmem.h`”. A secure vector is an `std::vector` with a custom allocator.. This custom allocator is named `secure_allocator` and is also defined in “`secmem.h`”. The `secure_allocator` makes sure that the memory is securely deleted and overwritten after usage. Additionally, if supported by the platform, it is ensured that the memory locations which contain sensitive information are locked into volatile memory and not swapped out to disk. All Unix-like operating systems as well as Windows provide support for memory locking.

5 CPU specific optimizations

Sometimes multiple implementations for the same algorithm are available in Botan. These alternate implementations use 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:

- A table-based implementation vulnerable to side channels.
- A constant time version using SSSE3 SIMD extensions on modern x86 CPUs.
- A constant time and fast version which using x86 AES-NI instructions.

If the CPU specific optimizations are available at runtime, they are automatically used if enabled in the build.

6 Provider Model / Using External Libraries

The idea behind the provider model is to allow exchanging certain implementations in Botan with either other software implementations from external libraries or with a hardware implementation inside a HSM, e.g, a smart card or TPM. Botan currently ships three different providers: OpenSSL, PKCS#11 and TPM.

6.1 OpenSSL

The OpenSSL provider is enabled during configure stage with the parameter `--with-openssl`. This will result in the following changes: The OpenSSL module is enabled and build.h (see 3.4) is extended with the following macro: BOTAN_HAS_OPENSSL 20151219.

All files in “`botan\src\lib\prov\openssl`” belong to the OpenSSL provider/module:

- `info.txt`
- `openssl.h`
- `openssl_block.cpp`
- `openssl_ec.cpp`
- `openssl_hash.cpp`
- `openssl_rc4.cpp`
- `openssl_rsa.cpp`

The OpenSSL module provides implementations for block ciphers, elliptic curve cryptography, hash functions, RC4 and RSA. For example, to use the OpenSSL implementation for AES-128 instead of Botans own, the block cipher object has to be created as follows:

```
BlockCipher::create( "AES-128", "openssl" );
```

instead of:

```
BlockCipher::create( "AES-128" );
```

which would use Botans own implementation.

6.2 PKCS#11

The PKCS#11 provider is enabled by default and allows to run the cryptographic operations in hardware on a smart card or HSM. Additionally, it provides for example functionality to initialize smart cards / HSMs, change PINs and iterate over objects on the device. The following example shows how to encrypt something with RSA using the PKCS#11 provider:

```
std::string padding = “Raw”;
PK_Encryptor_EME encryptor( public_key, rng, padding, "pkcs11" );
```

```
auto encrypted = encryptor.encrypt( plaintext, rng );
```

The provider is the last argument in the `PK_Encryptor_EME` constructor. By specifying "pkcs11" we make sure that this operation is performed using the PKCS#11 provider.

6.3 TPM

The TPM provider usage is equivalent to the one of the OpenSSL and PKCS#11 providers. It can be enabled with the `configure.py` parameter `–with-tpm`. Internally it uses the external Trousers library to access the Trusted Platform Module.

6.4 BearSSL

The BearSSL provider is enabled during configure stage with the parameter `--with-bearssl`. This will result in the following changes: The BearlSSL module is enabled and build.h (see 3.4) is extended with the following macro: BOTAN_HAS_BEARSSL 20170628.

All files in "`botan\src\lib\prov\bearssl`" belong to the BearSSL provider/module:

- `info.txt`
- `openssl.h`
- `openssl_ec.cpp`
- `openssl_hash.cpp`

The BearSSL module provides implementations for elliptic curve cryptography and hash functions. For example, to use the BearSSL implementation for SHA-256 instead of Botans own, the hash function object has to be created as follows:

```
HashFunction::create( "SHA-256", "bearssl" );
```

instead of:

```
HashFunction::create( "SHA-256" );
```

which would use Botans own implementation.

7 Command Line Interface (CLI)

Botan offers a set of command line tools to handle some common tasks on the command line. The command line tool is invoked with `botan <cmd> <cmd-options>`. It offers the following commands:

Command	Description
`asn1print`	Prints the ASN.1 structure of the given FILE.
`base64_dec`	Base64 decodes the given FILE.
`base64_enc`	Base64 encodes the given FILE.
`cert_info`	Prints the contents of the given X.509 certificate FILE.
`cert_verify`	Verifies a certificate chain.
`config`	Print the library configuration.
`cpuid`	Prints information about the supported CPUID flags of the current CPU.
`dl_group_info`	Prints parameters of a given DL group.
`ec_group_info`	Prints parameters of a given ECC group.
`factor`	Factors a given integer using a combination of trial division by small primes, and Pollard's Rho algorithm.
`gen_dl_group`	Generates a DL group.
`gen_pkcs10`	Generates a PKCS#10 certificate signing request (CSR).
`gen_prime`	Generates a random prime.
`gen_self_signed`	Generates a self-signed certificate.
`hash`	Calculates the message digest of given files.
`help`	Prints available commands.
`hex_dec`	Hex decode a given file.
`hex_enc`	Hex encode a given file.
`http_get`	Performs a HTTP GET query against the given URL and prints the result.
`is_prime`	Checks if the given integer is prime.
`keygen`	Generates a new public key keypair.
`mod_inverse`	Calculate the modular inverse.
`ocsp_check`	Performs an OCSP online check for a given certificate and prints the result.

Command	Description
`pkcs8`	Provides PKCS#8 key container handling.
`rng`	Generates random bytes.
`sign`	Signs a given file using a public key signature algorithm and prints the base64 encoded signature.
`sign_cert`	Signs a given PKCS#10 CSR using a CA key.
`speed`	Performs speed tests of given algorithms.
`timing_test`	Performs timing tests.
`tls_ciphers`	Print the ciphersuites supported by the given TLS policy and TLS version.
`tls_client`	Provides a TLS command line client.
`tls_client_hello`	Decodes the given TLS client hello.
`tls_server`	Provides a TLS command line server.
`verify`	Verifies the public key signature on a given file.
`version`	Prints the Botan version.

The following command line example verifies the peer certificate from file “peer.crt” using intermediate CA cert from file “inter.crt” and root CA from file “root.crt”:

```
$ botan cert_verify peer.crt inter.crt root.crt
Certificate passes validation checks
```

8 Test Suite

Botan contains an extensive test suite that aims to cover the library source code with positive and negative tests. The test suite is organized in the `src/tests/` folder as follows. Folders are typed in **bold**.

File/Folder Name	Description
`data`	Test vectors for known answer tests
`main.cpp`	Test runner main loop
`test_aead.cpp`	Tests for AEAD modes
`test_asn1.cpp`	Tests for the ASN1 module
`test_bigint.cpp`	Tests for the BigInt module
`test_block.cpp`	Tests for block ciphers
`test_certstor.cpp`	Tests for the certificate store
`test_compression.cpp`	Tests for the compression module
`test_datastore.cpp`	Tests for the DataStore class
`test_dh.cpp`	Tests for Diffie Hellman
`test_dl_group.cpp`	Tests for discrete logarithm
`test_dlies.cpp`	Tests for DLIES
`test_dsa.cpp`	Tests for DSA
`test_ecc_pointmul.cpp`	Tests for ECC point multiplication
`test_ecdh.cpp`	Tests for ECDH
`test_ecdsa.cpp`	Tests for ECDSA
`test_ecgdsa.cpp`	Tests for ECGDSA
`test_ecies.cpp`	Tests for ECIES
`test_eckcdsa.cpp`	Tests for ECKDSA
`test_entropy.cpp`	Tests for entropy sources
`test_ffi.cpp`	Tests for the FFI (C bindings)
`test_filters.cpp`	Tests for the filters module
`test_fuzzer.cpp`	Basic fuzzing tests
`test_gf2m.cpp`	Tests for GF(2^m)
`test_hash.cpp`	Tests for hash functions
`test_hash_id.cpp`	Tests for module hash_id
`test_kdf.cpp`	Tests for key derivation functions
`test_mac.cpp`	Tests for message authentication codes

File/Folder Name	Description
`test_modes.cpp`	Tests for block cipher modes of operation
`test_mp.cpp`	Tests for multi-precision integer handling
`test_name_constraint.cpp`	Tests for X.509 name constraints
`test_ocsp.cpp`	Tests for OCSP
`test_octetstring.cpp`	Unit tests for the OctetString class
`test_pad.cpp`	Tests for block cipher padding modes
`test_pk_pad.cpp`	Public key padding tests
`test_pkcs11_high_level.cpp`	PKCS#11 high level tests
`test_pkcs11_low_level.cpp`	PKCS#11 low level tests
`test_pkcs11.cpp`	Base class for PKCS#11 tests
`test_pkcs11.h`	Base class for PKCS#11 tests
`test_pubkey.cpp`	Base classes for public key algorithm tests
`test_pubkey.h`	Base classes for public key algorithm tests
`test_rng_kat.cpp`	Known-answer tests for random number generators
`test_rng.cpp`	Tests for random number generators
`test_rng.h`	Random number generators useful for testing
`test_rsa.cpp`	Tests for RSA
`test_runner.cpp`	Implementation of a test runner
`test_runner.h`	Header file of the test runner
`test_simd.cpp`	Tests for SIMD extensions
`test_stream.cpp`	Stream cipher tests
`test_tls_cbc.cpp`	Tests for TLS CBC padding
`test_tls_messages.cpp`	TLS messages unit tests
`test_utils.cpp`	Tests of the utils module
`test_workfactor.cpp`	Workfactor tests
`test_x509_path.cpp`	X.509 path validation tests
`test_xmss.cpp`	XMSS tests
`tests.cpp`	Base classes for tests
`tests.h`	Base classes for tests
`unit_ecc.cpp`	ECC unit tests
`unit_ecdh.cpp`	ECDH unit tests
`unit_ecdsa.cpp`	ECDSA unit tests

File/Folder Name	Description
`unit_tls_policy.cpp`	TLS_Policy unit tests
`unit_tls.cpp`	TLS client/server tests
`unit_x509.cpp`	X.509 unit tests

8.1 Test Vectors

Many of the tests are so called known answer tests (KAT). These tests use test vectors from different sources such as RFCs, scientific papers or the NIST Cryptographic Algorithm Validation Program[9] to make sure cryptographic algorithms are implemented correctly. Test vectors are stored as text files in the `tests/data` folder and have the file ending `.vec`. Vector files can contain comments, prefixed with the pound sign, which are ignored during parsing. The following excerpt show parts of the vector file for the AES block cipher `aes.vec`.

```
[AES-128]
Key = 000102030405060708090A0B0C0D0E0F
In = 00112233445566778899AABBCCDDEEFF
Out = 69C4E0D86A7B0430D8CDB78070B4C55A

Key = 00010203050607080A0B0C0D0F101112
In = 506812A45F08C889B97F5980038B8359
Out = D8F532538289EF7D06B506A4FD5BE9C9
```

Listing 1: Excerpt of the aes.vec test vector file

First, the algorithm to be used is written in angle brackets [], in this case AES-128. Following, the test vectors for this algorithm are listed, in this case two test vectors each containing of a *Key*, an input value *In* and an output value *Out*. The aes.vec vector file also contains test vectors for AES-192 and AES-256, each followed by a new section starting with the algorithm name, in this case [AES-192] and [AES-256].

8.2 Test Framework

The test suite offers several classes for testing library functionality, explained in more detail in the following.

9 http://csrc.nist.gov/groups/STM/cavp/

8.2.1 Class Test

The `Test` class offers basic functionality for writing a test. A test that does not require test vectors, such as a unit test, uses this class. Adding a test involves deriving from the `Test` class and overriding the `run()` member function. Listing 2 shows an excerpt of the `HMAC_DRBG` unit test class. The actual test functions are omitted here to keep the example short.

```
class HMAC_DRBG_Unit_Tests : public Test
   {
   public:
      std::vector<Test::Result> run() override
         {
         std::vector<Test::Result> results;
         results.push_back(test_reseed_kat());
         results.push_back(test_reseed());
         results.push_back(test_max_number_of_bytes_per_request());
         results.push_back(test_broken_entropy_input());
         results.push_back(test_check_nonce());
         results.push_back(test_prediction_resistance());
         results.push_back(test_fork_safety());
         results.push_back(test_randomize_with_ts_input());
         return results;
         }
   };
```

Listing 2: Excerpt from the HMAC_DRBG unit test class (test_rng.cpp)

The Test class also offers some utility functions, such as access to a random number generator useful for testing via `Test::rng()` or a timestamp via `Test::timestamp()`.

8.2.2 Class Text_Based_Test

The `Text_Based_Test` class is used for implementing a test with test vectors. Adding a test involves deriving from the `Text_Based_Test` class and overriding the `run_one_test()` member function. Listing 3 shows an excerpt of the block cipher known answer test class. The code was modified in order to provide a minimal example.

```
class Block_Cipher_Tests : public Text_Based_Test
   {
   public:
      Block_Cipher_Tests() : Text_Based_Test("block", {"Key", "In", "Out"}) {}

      Test::Result run_one_test(const std::string& algo, const VarMap& vars)
override
         {
         const std::vector<uint8_t> key      = get_req_bin(vars, "Key");
         const std::vector<uint8_t> input    = get_req_bin(vars, "In");
         const std::vector<uint8_t> expected = get_req_bin(vars, "Out");

         std::unique_ptr<Botan::BlockCipher> cipher(
                  Botan::BlockCipher::create(algo));

         Test::Result result(algo);
         cipher->set_key(key);
         std::vector<uint8_t> buf = input;

         cipher->encrypt(buf);

         result.test_eq(provider, "encrypt", buf, expected);

         // always decrypt expected ciphertext vs what we produced above
         buf = expected;
         cipher->decrypt(buf);

         cipher->clear();

         result.test_eq(provider, "decrypt", buf, input);

         return result;
         }

   };
```

Listing 3: Modified Block cipher known answer test class from test_block.cpp

In this example, the `Text_Based_Test` base class is initialized such that it looks for test vectors in the `block` folder and that each test vector shall contain the keys *Key*, *In* and *Out*. Upon test execution, the test runner will create an instance of the `Block_Cipher_Tests` class and iterate over all .vec files in the `block` folder, invoking `run_one_test()` for each test vector in each .vec file. The algorithm name given in angle brackets in the .vec file, e,g., “AES-128”, and the test vector values, e.g., *Key*, *In* and *Out*, are given as parameters `algo` and `vars`.

8.2.3 Class Test::Result

The test runner collects the test results from all registered tests. Each test function, such as `Test::run()` and `Text_Based_Test::run_one_test()`, returns a `Test::Result` or `std::vector<Test::Result>` object containing the results of this specific test. At the beginning of a test function, a `Test::Result` object is created, giving the test name as a parameter. For known answer tests, this is usually the algorithm name, as in Listing 3. For each test case, a suitable function is called on the `Test::Result` object, automatically reporting the result to the test runner. It offers several functions for this, the most important listed below.

- `confirm(provider, what, expression)`: Test that the given expression evaluates to `true`.
- `test_eq(provider, what, produced, expected)`: Test that `produced` and `expected` are equal. This functions is offered for many different types, including container types.
- `test_ne(provider, what, produced, expected)`: Test that `produced` and `expected` are not equal. This functions is offered for many different types, including container types.
- `test_lt(provider, what, produced, expected)`: Test that `produced` is less than `expected`.
- `test_gte(provider, what, produced, expected)`: Test that `produced` is greater than or equal to `expected`.
- `test_throws(provider, what, function)`: Test that the `function` throws an exception.

8.2.4 Test Registration

Finally, a test must be registered with the test runner using the `BOTAN_REGISTER_TEST` macro. This macro automatically creates code to register the given test class with the test runner under the given name, as seen in Listing 4. For each test class, this macro must appear exactly once.

```
class Block_Cipher_Tests : public Text_Based_Test
   {
   public:
      Block_Cipher_Tests() : Text_Based_Test("block", {"Key", "In", "Out"}) {}

      Test::Result run_one_test(const std::string& algo, const VarMap& vars)
override
      {
      ...
      }
   };

BOTAN_REGISTER_TEST("block", Block_Cipher_Tests);
```

Listing 4: Registering the block cipher known answer tests

Appendix

1 Module Description Example

This is not a real example, just one that shows all available options:

```
<defines>
ASN1 -> 20131128
</defines>

load_on auto
mp_bits 32
need_isa aesni

<requires>
bigint
oid_lookup
</requires>

<header:internal>
mp_generic:mp_madd.h
mp_asmi.h
</header:internal>

<header:public>
serpent.h
</header:public>

<source>
serpent.cpp
</source>

<arch>
x86_32
</arch>

<cc>
msvc
</cc>

<os>
```

```
windows
cygwin
mingw
</os>

<libs>
windows -> advapi32.lib
mingw -> advapi32
</libs>

<frameworks>
darwin -> Security
</frameworks>

<comment>”Loading module ASN1”</comment>

<warning>”Deprecated Module”</warning>
```

2 Architecture Description Example

The following example shows the x86_64 architecture description:

```
endian little
unaligned ok
wordsize 64

family x86

<aliases>
amd64
x86-64
em64t
x64
</aliases>

<submodels>
k8
barcelona
atom
nocona
core2
corei7
sandybridge
ivybridge
</submodels>

<submodel_aliases>
core2duo -> core2
intelcore2 -> core2
intelcore2duo -> core2

nehalem -> corei7
westmere -> corei7

sledgehammer -> k8
opteron -> k8
amdopteron -> k8
athlon64 -> k8
```

```
</submodel_aliases>

<isa_extensions>
sse2
ssse3
sse4.1
sse4.2
avx2
aesni
clmul
rdrand
rdseed
sha
bmi2
</isa_extensions>
```

3 Compiler Description Example

The following example shows the compiler description for gcc.

```
macro_name GCC

binary_name g++

output_to_exe "-o "
add_include_dir_option -I
add_lib_dir_option -L
add_lib_option -l

lang_flags "-std=c++11 -D_REENTRANT"

# This should only contain flags which are included in GCC 4.8
warning_flags "-Wall -Wextra -Wpedantic -Wstrict-aliasing -
Wstrict-overflow=5 -Wcast-align -Wmissing-declarations -Wpointer-
arith -Wcast-qual -Wzero-as-null-pointer-constant -Wnon-virtual-
dtor"

maintainer_warning_flags "-Wold-style-cast -Wsuggest-override -
Wshadow -Werror -Wno-error=old-style-cast -Wno-error=zero-as-null-
pointer-constant -Wno-error=strict-overflow -Wno-error=deprecated-
declarations"

compile_flags "-c"
debug_info_flags "-g"
optimization_flags "-O3"
size_optimization_flags "-Os"

shared_flags "-fPIC"
coverage_flags "--coverage"

# GCC 4.8
sanitizer_flags "-D_GLIBCXX_DEBUG -fsanitize=address"

# GCC 4.9 and later
#sanitizer_flags "-D_GLIBCXX_DEBUG -fsanitize=address,undefined -
fno-sanitize-recover=undefined"
```

```
visibility_build_flags "-fvisibility=hidden"
visibility_attribute '__attribute__((visibility("default")))'


<so_link_commands>
# The default works for GNU ld and several other Unix linkers
default       -> "$(CXX) -shared -fPIC -Wl,-soname,$(SONAME_ABI)"
default-debug -> "$(CXX) -shared -fPIC -Wl,-soname,$(SONAME_ABI)"

# Darwin, HP-UX and Solaris linkers use different syntax
darwin  -> "$(CXX) -dynamiclib -fPIC -install_name $(LIBDIR)/$
(SONAME_ABI)"
hpux    -> "$(CXX) -shared -fPIC -Wl,+h,$(SONAME_ABI)"
solaris -> "$(CXX) -shared -fPIC -Wl,-h,$(SONAME_ABI)"

# AIX and OpenBSD don't use sonames at all
aix     -> "$(CXX) -shared -fPIC"
openbsd -> "$(CXX) -shared -fPIC"
</so_link_commands>

<binary_link_commands>
linux         -> "$(LINKER) -Wl,-rpath=\$$ORIGIN"
linux-debug   -> "$(LINKER) -Wl,-rpath=\$$ORIGIN"
default       -> "$(LINKER)"
default-debug -> "$(LINKER)"
</binary_link_commands>

<isa_flags>
sse2    -> "-msse2"
ssse3   -> "-mssse3"
sse4.1  -> "-msse4.1"
sse4.2  -> "-msse4.2"
avx2    -> "-mavx2"
bmi2    -> "-mbmi2"
aesni   -> "-maes -mpclmul -mssse3"
rdrand  -> "-mrdrnd"
rdseed    -> "-mrdseed"
sha     -> "-msha"
altivec -> "-maltivec"
```

```
</isa_flags>

<mach_opt>
# Avoid using -march=i[3456]86, instead tune for generic
i386        -> "-mtune=generic"
i486        -> "-mtune=generic"
i586        -> "-mtune=generic"
i686        -> "-mtune=generic"

# Translate to GCC-speak
nehalem     -> "-march=corei7"
sandybridge -> "-march=corei7-avx"
ivybridge   -> "-march=core-avx-i"

ppc601      -> "-mpowerpc -mcpu=601"
cellppu     -> "-mcpu=cell"
e500v2      -> "-mcpu=8548"

# No scheduler in GCC for anything after EV67
alpha-ev68  -> "-mcpu=ev67"
alpha-ev7   -> "-mcpu=ev67"

# The patch from Debian bug 594159 has this, don't know why
though...
sh4         -> "-m4 -mieee"

# Default family options (SUBMODEL is substitued with the actual
# submodel name). Anything after the quotes is what should be
# *removed* from the submodel name before it's put into SUBMODEL.

alpha     -> "-mcpu=SUBMODEL" alpha-
arm32     -> "-march=SUBMODEL"
arm64     -> "-march=SUBMODEL"
superh    -> "-mSUBMODEL" sh
hppa      -> "-march=SUBMODEL" hppa
ia64      -> "-mtune=SUBMODEL"
m68k      -> "-mSUBMODEL"
mips32    -> "-mips1 -mcpu=SUBMODEL" mips32-
mips64    -> "-mips3 -mcpu=SUBMODEL" mips64-
ppc32     -> "-mcpu=SUBMODEL" ppc
```

```
ppc64     -> "-mcpu=SUBMODEL" ppc
sparc32   -> "-mcpu=SUBMODEL -Wa,-xarch=v8plus" sparc32-
sparc64   -> "-mcpu=v9 -mtune=SUBMODEL"
x86_32    -> "-march=SUBMODEL"
x86_64    -> "-march=SUBMODEL"

all_x86_32 -> "-momit-leaf-frame-pointer"
all_x86_64 -> "-momit-leaf-frame-pointer"
</mach_opt>

# Flags set here are included at compile and link time
<mach_abi_linking>
all     -> "-pthread -fstack-protector"

cilkplus -> "-fcilkplus"
openmp   -> "-fopenmp"

mips64  -> "-mabi=64"
s390    -> "-m31"
s390x   -> "-m64"
sparc32 -> "-m32 -mno-app-regs"
sparc64 -> "-m64 -mno-app-regs"
ppc64   -> "-m64"
x86_64  -> "-m64"

netbsd  -> "-D_NETBSD_SOURCE"
qnx     -> "-fexceptions -D_QNX_SOURCE"
</mach_abi_linking>
```

4 Operating System Description Example

The following example shows the operating system description for Linux.

```
os_type unix

soname_suffix "so"
lib_prefix "lib"

<target_features>
clock_gettime
gettimeofday
posix_mlock
gmtime_r
dlopen
readdir
timegm
sockets
threads
filesystem
</target_features>

<aliases>
linux-gnu
</aliases>
```

5 BSI Module Policy File

The following listing shows the bsi module policy file. Note that some modules can not be prohibited because they are dependencies of other modules listed as required, e.g., sha1 is required by the tls module. Such modules are listed as prohibited, but commented out and thus ignored by `configure.py`.

```
<required>
# block
aes

# modes
gcm
cbc
mode_pad

# stream
ctr

# hash
sha2_32
sha2_64
sha3

# mac
cmac
hmac
gmac

# kdf
kdf1_iso18033
sp800_108
sp800_56c

# pk_pad
eme_oaep
emsa_pssr
emsa1
iso9796

# pubkey
dlies
dh
rsa
dsa
ecdsa
ecgdsa
```

```
ecies
eckcdsa
ecdh
xmss

# rng
auto_rng
hmac_drbg
</required>

<if_available>
# block
aes_ni
aes_ssse3

# modes
clmul
clmul_ssse3
pmull

# hash
sha2_32_x86
sha2_32_armv8

# entropy sources
darwin_secrandom
dev_random
proc_walk
rdrand
rdseed
win32_stats

# rng
rdrand_rng
system_rng

# utils
http_util # needed by x509 for OCSP online checks
locking_allocator
simd
</if_available>

<prohibited>
# block
aria
blowfish
camellia
cascade
```

```
cast
des
gost_28147
idea
idea_sse2
kasumi
lion
misty1
noekeon
noekeon_simd
seed
serpent
serpent_simd
shacal2
shacal2_x86
shacal2_simd
sm4
threefish
threefish_avx2
twofish
xtea

# modes
ccm
chacha20poly1305
eax
ocb
siv
cfb

# stream
chacha
chacha_sse2
ofb
rc4
salsa20
shake_cipher

# kdf
hkdf
kdf1
kdf2
prf_x942
sp800_56a

# pubkey
cecpq1
curve25519
```

```
ed25519
elgamal
gost_3410
mce
mceies
rfc6979
newhope
sm2

# pk_pad
#eme_pkcs1 // needed for tls
#emsa_pkcs1 // needed for tls
emsa_raw
emsa_x931

# hash
blake2
comb4p
gost_3411
md4
#md5 // needed for tls
rmd160
#sha1 // needed for tls
#sha1_sse2 // needed for tls
shake
skein
sm3
streebog
tiger
whirlpool
keccak

# rng
chacha_rng

# mac
cbc_mac
poly1305
siphash
x919_mac

# misc
bcrypt
</prohibited>
```