wolfSSL Documentation

2021-11-08

Contents

1 Introduction **10**
1.1 Why Choose wolfSSL? . . . 10

2 Building wolfSSL **11**
2.1 Getting wolfSSL Source Code . . . 11
2.2 Building on *nix . . . 11
2.3 Building on Windows . . . 12
2.3.1 VS 2008 . . . 12
2.3.2 VS 2010 . . . 12
2.3.3 VS 2013 (64 bit solution) . . . 12
2.3.4 Cygwin . . . 12
2.4 Building in a non-standard environment . . . 13
2.4.1 Building into Yocto Linux . . . 13
2.4.2 Building with Atollic TrueSTUDIO . . . 14
2.4.3 Removing Features . . . 14
2.4.4 Enabling Features Disabled by Default . . . 15
2.4.5 Customizing or Porting wolfSSL . . . 18
2.4.6 Reducing Memory or Code Usage . . . 20
2.4.7 Increasing Performance . . . 21
2.4.8 GCM Performance Tuning . . . 21
2.4.9 wolfSSL's proprietary Single Precision math support . . . 21
2.4.10 Stack or Chip Specific Defines . . . 22
2.4.11 OS Specific Defines . . . 23
2.5 Build Options . . . 23
2.5.1 `--enable-debug` . . . 24
2.5.2 `--enable-distro` . . . 24
2.5.3 `--enable-singlethread` . . . 24
2.5.4 `--enable-dtls` . . . 24
2.5.5 `--disable-rng` . . . 24
2.5.6 `--enable-sctp` . . . 24
2.5.7 `--enable-openssh` . . . 24
2.5.8 `--enable-apachehttpd` . . . 24
2.5.9 `--enable-openvpn` . . . 24
2.5.10 `--enable-opensslextra` . . . 24
2.5.11 `--enable-opensslall` . . . 24
2.5.12 `--enable-maxstrength` . . . 25
2.5.13 `--disable-harden` . . . 25
2.5.14 `--enable-ipv6` . . . 25
2.5.15 `--enable-bump` . . . 25
2.5.16 `--enable-leanpsk` . . . 25
2.5.17 `--enable-leantls` . . . 25
2.5.18 `--enable-bigcache` . . . 25
2.5.19 `--enable-hugecache` . . . 25
2.5.20 `--enable-smallcache` . . . 25
2.5.21 `--enable-savesession` . . . 25
2.5.22 `--enable-savecert` . . . 26
2.5.23 `--enable-atomicuser` . . . 26
2.5.24 `--enable-pkcallbacks` . . . 26
2.5.25 `--enable-sniffer` . . . 26
2.5.26 `--enable-aesgcm` . . . 26
2.5.27 `--enable-aesccm` . . . 26
2.5.28 `--disable-aescbc` . . . 26

2.5.29 --enable-aescfb . . . 27
2.5.30 --enable-aesctr . . . 27
2.5.31 --enable-aesni . . . 27
2.5.32 --enable-intelasm . . . 27
2.5.33 --enable-camellia . . . 27
2.5.34 --enable-md2 . . . 27
2.5.35 --enable-nullcipher . . . 27
2.5.36 --enable-ripemd . . . 27
2.5.37 --enable-blake2 . . . 27
2.5.38 --enable-blake2s . . . 27
2.5.39 --enable-sha3 . . . 27
2.5.40 --enable-sha512 . . . 27
2.5.41 --enable-sessioncerts . . . 28
2.5.42 --enable-keygen . . . 28
2.5.43 --enable-certgen . . . 28
2.5.44 --enable-certreq . . . 28
2.5.45 --enable-sep . . . 28
2.5.46 --enable-hkdf . . . 28
2.5.47 --enable-x963kdf . . . 28
2.5.48 --enable-dsa . . . 28
2.5.49 --enable-eccshamir . . . 28
2.5.50 --enable-ecc . . . 28
2.5.51 --enable-ecccustcurves . . . 28
2.5.52 --enable-compkey . . . 28
2.5.53 --enable-curve25519 . . . 29
2.5.54 --enable-ed25519 . . . 29
2.5.55 --enable-fpecc . . . 29
2.5.56 --enable-eccencrypt . . . 29
2.5.57 --enable-psk . . . 29
2.5.58 --disable-errorstrings . . . 29
2.5.59 --disable-oldtls . . . 29
2.5.60 --enable-sslv3 . . . 29
2.5.61 --enable-stacksize . . . 29
2.5.62 --disable-memory . . . 29
2.5.63 --disable-rsa . . . 29
2.5.64 --enable-rsapss . . . 29
2.5.65 --disable-dh . . . 29
2.5.66 --enable-anon . . . 30
2.5.67 --disable-asn . . . 30
2.5.68 --disable-aes . . . 30
2.5.69 --disable-coding . . . 30
2.5.70 --enable-base64encode . . . 30
2.5.71 --disable-des3 . . . 30
2.5.72 --enable-idea . . . 30
2.5.73 --enable-arc4 . . . 30
2.5.74 --disable-md5 . . . 30
2.5.75 --disable-sha . . . 30
2.5.76 --enable-webserver . . . 30
2.5.77 --enable-hc128 . . . 30
2.5.78 --enable-rabbit . . . 30
2.5.79 --enable-fips . . . 30
2.5.80 --enable-sha224 . . . 31
2.5.81 --disable-poly1305 . . . 31
2.5.82 --disable-chacha . . . 31

2.5.83 --disable-hashdrbg . . . 31
2.5.84 --disable-filesystem . . . 31
2.5.85 --disable-inline . . . 31
2.5.86 --enable-ocsp . . . 31
2.5.87 --enable-ocspstapling . . . 31
2.5.88 --enable-ocspstapling2 . . . 31
2.5.89 --enable-crl . . . 31
2.5.90 --enable-crl-monitor . . . 31
2.5.91 --enable-sni . . . 31
2.5.92 --enable-maxfragment . . . 32
2.5.93 --enable-alpn . . . 32
2.5.94 --enable-truncatedhmac . . . 32
2.5.95 --enable-renegotiation-indication . . . 32
2.5.96 --enable-secure-renegotiation . . . 32
2.5.97 --enable-supportedcurves . . . 32
2.5.98 --enable-session-ticket . . . 32
2.5.99 --enable-extended-master . . . 32
2.5.100 --enable-tlsx . . . 32
2.5.101 --enable-pkcs7 . . . 32
2.5.102 --enable-pkcs11 . . . 32
2.5.103 --enable-ssh . . . 32
2.5.104 --enable-scep . . . 33
2.5.105 --enable-srp . . . 33
2.5.106 --enable-smallstack . . . 33
2.5.107 --enable-valgrind . . . 33
2.5.108 --enable-testcert . . . 33
2.5.109 --enable-iopool . . . 33
2.5.110 --enable-certservice . . . 33
2.5.111 --enable-jni . . . 33
2.5.112 --enable-lighty . . . 33
2.5.113 --enable-stunnel . . . 33
2.5.114 --enable-md4 . . . 33
2.5.115 --enable-pwdbased . . . 33
2.5.116 --enable-scrypt . . . 33
2.5.117 --enable-cryptonly . . . 34
2.5.118 --enable-fastmath . . . 34
2.5.119 --enable-fasthugemath . . . 34
2.5.120 --disable-examples . . . 34
2.5.121 --disable-crypttests . . . 34
2.5.122 --enable-fast-rsa . . . 35
2.5.123 --enable-staticmemory . . . 35
2.5.124 --enable-mcapi . . . 35
2.5.125 --enable-asynccrypt . . . 35
2.5.126 --enable-sessionexport . . . 35
2.5.127 --enable-aeskeywrap . . . 35
2.5.128 --enable-jobserver . . . 35
2.5.129 --enable-shared[=PKGS] . . . 35
2.5.130 --enable-static[=PKGS] . . . 35
2.5.131 --with-ntru=PATH . . . 36
2.5.132 --with-libz=PATH . . . 36
2.5.133 --with-cavium . . . 36
2.5.134 --with-user-crypto . . . 36
2.5.135 --enable-rsavfy . . . 36
2.5.136 --enable-rsapub . . . 36

2.5.137 --enable-sp . . . 36
2.5.138 --enable-sp-asm . . . 36
2.5.139 --enable-armasm . . . 36
2.5.140 --disable-tlsv12 . . . 36
2.5.141 --enable-tls13 . . . 36
2.5.142 --enable-all . . . 37
2.5.143 --enable-xts . . . 37
2.5.144 --enable-asio . . . 37
2.5.145 --enable-qt . . . 37
2.5.146 --enable-qt-test . . . 37
2.5.147 --enable-apache-httpd . . . 37
2.5.148 --enable-afalg . . . 37
2.5.149 --enable-devcrypto . . . 37
2.5.150 --enable-mcast . . . 37
2.5.151 --disable-pkcs12 . . . 37
2.5.152 --enable-fallback-scsv . . . 37
2.5.153 --enable-psk-one-id . . . 38
2.6 Cross Compiling . . . 38
2.6.1 Example cross compile configure options for toolchain builds . . . 38
2.7 2.7 Building Ports . . . 40

3 Getting Started **41**
3.1 General Description . . . 41
3.2 Testsuite . . . 41
3.3 Client Example . . . 43
3.4 Server Example . . . 45
3.5 EchoServer Example . . . 46
3.6 EchoClient Example . . . 46
3.7 Benchmark . . . 47
3.7.1 Relative Performance . . . 48
3.7.2 Benchmarking Notes . . . 48
3.7.3 Benchmarking on Embedded Systems . . . 50
3.8 Changing a Client Application to Use wolfSSL . . . 50
3.9 Changing a Server Application to Use wolfSSL . . . 51

4 Features **53**
4.1 Features Overview . . . 53
4.2 Protocol Support . . . 53
4.2.1 Server Functions . . . 53
4.2.2 Client Functions . . . 53
4.2.3 Robust Client and Server Downgrade . . . 53
4.2.4 IPv6 Support . . . 54
4.2.5 DTLS . . . 54
4.2.6 LwIP (Lightweight Internet Protocol) . . . 54
4.2.7 TLS Extensions . . . 54
4.3 Cipher Support . . . 55
4.3.1 Cipher Suite Strength and Choosing Proper Key Sizes . . . 55
4.3.2 Supported Cipher Suites . . . 56
4.3.3 AEAD Suites . . . 59
4.3.4 Block and Stream Ciphers . . . 59
4.3.5 Hashing Functions . . . 60
4.3.6 Public Key Options . . . 60
4.3.7 ECC Support . . . 60
4.3.8 PKCS Support . . . 61
4.3.9 Forcing the Use of a Specific Cipher . . . 62

4.3.10 Quantum-Safe Handshake Ciphersuite . . . 62
4.4 Hardware Accelerated Crypto . . . 63
4.4.1 AES-NI . . . 63
4.4.2 STM32F2 . . . 63
4.4.3 Cavium NITROX . . . 63
4.4.4 ESP32-WROOM-32 . . . 64
4.5 SSL Inspection (Sniffer) . . . 64
4.6 Compression . . . 65
4.7 Pre-Shared Keys . . . 65
4.8 Client Authentication . . . 66
4.9 Server Name Indication . . . 66
4.10 Handshake Modifications . . . 67
4.10.1 Grouping Handshake Messages . . . 67
4.11 Truncated HMAC . . . 67
4.12 User Crypto Module . . . 67
4.13 Timing-Resistance in wolfSSL . . . 68
4.14 Fixed ABI . . . 68

5 Portability **70**
5.1 Abstraction Layers . . . 70
5.1.1 C Standard Library Abstraction Layer . . . 70
5.1.2 Custom Input/Output Abstraction Layer . . . 70
5.1.3 Operating System Abstraction Layer . . . 71
5.2 Supported Operating Systems . . . 71
5.3 Supported Chipmakers . . . 72
5.4 C# Wrapper . . . 72

6 Callbacks **74**
6.1 HandShake Callback . . . 74
6.2 Timeout Callback . . . 74
6.3 User Atomic Record Layer Processing . . . 75
6.4 Public Key Callbacks . . . 75

7 Keys and Certificates **78**
7.1 Supported Formats and Sizes . . . 78
7.2 Certificate Loading . . . 78
7.2.1 Loading CA Certificates** . . . 78
7.2.2 Loading Client or Server Certificates . . . 78
7.2.3 Loading Private Keys . . . 79
7.2.4 Loading Trusted Peer Certificates . . . 79
7.3 Certificate Chain Verification . . . 79
7.4 Domain Name Check for Server Certificates . . . 80
7.5 No File System and using Certificates . . . 80
7.5.1 Test Certificate and Key Buffers . . . 80
7.6 Serial Number Retrieval . . . 80
7.7 RSA Key Generation . . . 80
7.7.1 RSA Key Generation Notes . . . 81
7.8 Certificate Generation . . . 82
7.9 Certificate Signing Request (CSR) Generation . . . 84
7.9.1 Limitations . . . 85
7.10 Convert to raw ECC key . . . 85
7.10.1 Example . . . 85

8 Debugging **86**
8.1 Debugging and Logging . . . 86

8.2 Error Codes . . . 86

9 Library Design 87
9.1 Library Headers . . . 87
9.2 Startup and Exit . . . 87
9.3 Structure Usage . . . 87
9.4 Thread Safety . . . 87
9.5 Input and Output Buffers . . . 88

10 wolfCrypt Usage Reference 89
10.1 Hash Functions . . . 89
10.1.1 MD4 . . . 89
10.1.2 MD5 . . . 89
10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512 . . . 89
10.1.4 BLAKE2b . . . 90
10.1.5 RIPEMD-160 . . . 90
10.2 Keyed Hash Functions . . . 91
10.2.1 HMAC . . . 91
10.2.2 GMAC . . . 91
10.2.3 Poly1305 . . . 91
10.3 Block Ciphers . . . 92
10.3.1 AES . . . 92
10.4 Stream Ciphers . . . 93
10.4.1 ARC4 . . . 93
10.4.2 RABBIT . . . 93
10.4.3 HC-128 . . . 94
10.4.4 ChaCha . . . 94
10.5 Public Key Cryptography . . . 95
10.5.1 RSA . . . 95
10.5.2 DH (Diffie-Hellman) . . . 96
10.5.3 EDH (Ephemeral Diffie-Hellman) . . . 97
10.5.4 DSA (Digital Signature Algorithm) . . . 97

11 SSL Tutorial 99
11.1 Introduction . . . 99
11.1.1 Examples Used in this Tutorial . . . 99
11.2 Quick Summary of SSL/TLS . . . 99
11.3 Getting the Source Code . . . 99
11.4 Base Example Modifications . . . 100
11.4.1 Modifications to the echoserver (tcpserv04.c) . . . 100
11.4.2 Modifications to the echoclient (tcpcli01.c) . . . 100
11.4.3 Modifications to unp.h header . . . 100
11.5 Building and Installing wolfSSL . . . 100
11.6 Initial Compilation . . . 102
11.7 Libraries . . . 102
11.8 Headers . . . 102
11.9 Startup/Shutdown . . . 102
11.10 WOLFSSL Object . . . 104
11.10.1 EchoClient . . . 104
11.10.2 EchoServer . . . 105
11.11 Sending/Receiving Data . . . 106
11.11.1 EchoClient . . . 106
11.11.2 EchoServer . . . 106
11.12 Signal Handling . . . 107
11.12.1 Echoclient / Echoserver . . . 107

11.13 Certificates . . . 108
11.14 Conclusion . . . 108

12 Best Practices for Embedded Devices 110
12.1 Creating Private Keys . . . 110
12.2 Digitally Signing and Authenticating with wolfSSL . . . 110

13 OpenSSL Compatibility 111
13.1 Compatibility with OpenSSL . . . 111
13.2 Differences Between wolfSSL and OpenSSL . . . 111
13.3 Supported OpenSSL Structures . . . 112
13.4 Supported OpenSSL Functions . . . 112
13.5 x509 Certificates . . . 113

14 Licensing 114
14.1 Open Source . . . 114
14.2 Commercial Licensing . . . 114
14.3 Support Packages . . . 114

15 Support and Consulting 115
15.1 How to Get Support . . . 115
15.1.1 Bugs Reports and Support Issues . . . 115
15.2 Consulting . . . 115
15.2.1 Feature Additions and Porting . . . 115
15.2.2 Competitive Upgrade Program . . . 115
15.2.3 Design Consulting . . . 115

16 wolfSSL (formerly CyaSSL) Updates 117
16.1 Product Release Information . . . 117

17 wolfSSL API Reference 118

18 wolfCrypt API Reference 118

A SSL/TLS Overview 118
A.1 General Architecture . . . 118
A.2 SSL Handshake . . . 118
A.3 Differences between SSL and TLS Protocol Versions . . . 118
A.3.1 SSL 3.0 . . . 120
A.3.2 TLS 1.0 . . . 120
A.3.3 TLS 1.1 . . . 120
A.3.4 TLS 1.2 . . . 120
A.3.5 TLS 1.3 . . . 120

B RFCs, Specifications, and Reference 122
B.1 Protocols . . . 122
B.2 Stream Ciphers . . . 122
B.3 Block Ciphers . . . 122
B.4 Hashing Functions . . . 122
B.5 Public Key Cryptography . . . 122
B.6 Other . . . 122

C Error Codes 123
C.1 wolfSSL Error Codes . . . 123
C.2 wolfCrypt Error Codes . . . 126
C.3 Common Error Codes and their Solution . . . 129
C.3.1 `ASN_NO_SIGNER_E` (-188) . . . 129

C.3.2 WANT_READ (-323) 129

1 Introduction

This manual is written as a technical guide to the wolfSSL embedded SSL/TLS library. It will explain how to build and get started with wolfSSL, provide an overview of build options, features, portability enhancements, support, and much more.

1.1 Why Choose wolfSSL?

There are many reasons to choose wolfSSL as your embedded SSL solution. Some of the top reasons include size (typical footprint sizes range from 20-100 kB), support for the newest standards (SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3, DTLS 1.0, and DTLS 1.2), current and progressive cipher support (including stream ciphers), multi- platform, royalty free, and an OpenSSL compatibility API to ease porting into existing applications which have previously used the OpenSSL package. For a complete feature list, see Features Overview.

2 Building wolfSSL

wolfSSL was written with portability in mind and should generally be easy to build on most systems. If you have difficulty building wolfSSL, please don't hesitate to seek support through our support forums (https://www.wolfssl.com/forums) or contact us directly at support@wolfssl.com.

This chapter explains how to build wolfSSL on Unix and Windows, and provides guidance for building wolfSSL in a non-standard environment. You will find the "getting started" guide in Chapter 3 and an SSL tutorial in Chapter 11.

When using the autoconf / automake system to build wolfSSL, wolfSSL uses a single Makefile to build all parts and examples of the library, which is both simpler and faster than using Makefiles recursively.

2.1 Getting wolfSSL Source Code

The most recent version of wolfSSL can be downloaded from the wolfSSL website as a ZIP file:

https://www.wolfssl.com/download

After downloading the ZIP file, unzip the file using the `unzip` command. To use native line endings, enable the `-a` modifier when using unzip. From the unzip man page, the `-a` modifier functionality is described:

> [...] The -a option causes files identified by zip as text files (those with the 't' label in zipinfo listings, rather than 'b') to be automatically extracted as such, converting line endings, end-of-file characters and the character set itself as necessary. [...]

NOTE: Beginning with the release of wolfSSL 2.0.0rc3, the directory structure of wolfSSL was changed as well as the standard install location. These changes were made to make it easier for open source projects to integrate wolfSSL. For more information on header and structure changes, please see Library Headers and Structure Usage.

2.2 Building on *nix

When building wolfSSL on Linux, *BSD, OS X, Solaris, or other *nix-like systems, use the autoconf system. To build wolfSSL you only need to run two commands from the wolfSSL root directory, `./configure` and `make`.

The `./configure` script sets up the build environment and you cab append any number of build options to `./configure`. For a list of available build options, please see Build Options or run the following the command line to see a list of possible options to pass to the `./configure` script:

```
./configure --help
```

Once `./configure` has successfully executed, to build wolfSSL, run:

```
make
```

To install wolfSSL run:

```
make install
```

You may need superuser privileges to install, in which case precede the command with sudo:

```
sudo make install
```

To test the build, run the testsuite program from the root wolfSSL directory:

```
./testsuite/testsuite.test
```

Alternatively you can use autoconf to run the testsuite as well as the standard wolfSSL API and crypto tests:

```
make test
```

Further details about expected output of the testsuite program can be found in the Testsuite section. If you want to build only the wolfSSL library and not the additional items (examples, testsuite, benchmark app, etc.), you can run the following command fromthe wolfSSL root directory:

```
make src/libwolfssl.la
```

2.3 Building on Windows

In addition to the instructions below, you can find instructions and tips for building wolfSSL with Visual Studio here.

2.3.1 VS 2008

Solutions are included for Visual Studio 2008 in the root directory of the install. For use with Visual Studio 2010 and later, the existing project files should be able to be converted during the import process.

Note: If importing to a newer version of VS you will be asked: "Do you want to overwrite the project and its imported property sheets?" You can avoid the following by selecting "No". Otherwise if you select "Yes", you will see warnings about `EDITANDCONTINUE` being ignored due to `SAFESEH` specification. You will need to right click on the testsuite, sslSniffer, server, echoserver, echoclient, and client individually and modify their Properties->Configuration Properties->Linker->Advanced (scroll all the way to the bottom in Advanced window). Locate "Image Has Safe Exception Handlers" and click the drop down arrow on the far right. Change this to No (`/SAFESEH:NO`) for each of the aforementioned. The other option is to disable `EDITANDCONTINUE` which, we have found to be useful for debugging purposes and is therefore not recommended.

2.3.2 VS 2010

You will need to download Service Pack 1 to build wolfSSL solution once it has been updated. If VS reports a linker error, clean and rebuild the project; the linker error should be taken care of.

2.3.3 VS 2013 (64 bit solution)

You will need to download Service Pack 4 to build wolfSSL solution once it has been updated. If VS reports a linker error, clean the project then Rebuild the project and the linker error should be taken care of.

To test each build, choose "Build All" from the Visual Studio menu and then run the testsuite program. To edit build options in the Visual Studio project, select your desired project (wolfssl, echoclient, echoserver, etc.) and browse to the "Properties" panel.

Note: After the wolfSSL v3.8.0 release the build preprocessor macros were moved to a centralized file located at `IDE/WIN/user_settings.h`. This file can also be found in the project. To add features such as ECC or ChaCha20/Poly1305 add `#defines` here such as `HAVE_ECC` or `HAVE_CHACHA` / `HAVE_POLY1305`.

2.3.4 Cygwin

If building wolfSSL for Windows on a Windows development machine, we recommend using the included Visual Studio project files to build wolfSSL. However if Cygwin is required here is a short guide on how our team achieved a successful build:

1. Go to https://www.cygwin.com/install.html and download `setup-x86_64.exe`
2. Run `setup-x86_64.exe` and install however you choose. Click through the installation menus until you reach the "Select Packages" stage.
3. Click on the "+" icon to expand "All"
4. Now go to the "Archive" section and select "unzip" drop down, change "Skip" to 6.0-15 (or some other version).
5. Under "Devel" click "autoconf" drop down and change "Skip" to "10-1" (or some other version)
6. Under "Devel" click "automake" drop down and change "Skip" to "10-1" (or some other version)
7. Under "Devel" click the "gcc-core" drop down and change "Skip" to 7.4.0-1 (NOTE: wolfSSL has not tested GCC 9 or 10 and as they are fairly new does not recommend using them until they have had a bit more time to be fine-tuned for development).
8. Under "Devel" click the "git" drop down and change "Skip" to 2.29.0-1 (or some other version)
9. Under "Devel" click "libtool" drop down and change "Skip" to "2.4.6-5" (or some other version)
10. Under "Devel" click the "make" drop down and change "Skip" to 4.2.1-1 (or some other version)
11. Click "Next" and proceed through the rest of the installation.

The additional packages list should include:

- unzip

- autoconf
- automake
- gcc-core
- git
- libtool
- make

2.3.4.1 Post Install

Open a Cygwin terminal and clone wolfSSL:

```
git clone https://github.com/wolfssl/wolfssl.git
cd wolfssl
./autogen.sh
./configure
make
make check
```

2.4 Building in a non-standard environment

While not officially supported, we try to help users wishing to build wolfSSL in a non-standard environment, particularly with embedded and cross-compilation systems. Below are some notes on getting started with this.

1. The source and header files need to remain in the same directory structure as they are in the wolfSSL download package.
2. Some build systems will want to explicitly know where the wolfSSL header files are located, so you may need to specify that. They are located in the `<wolfssl_root>/wolfssl` directory. Typically, you can add the `<wolfssl_root>` directory to your include path to resolve header problems.
3. wolfSSL defaults to a little endian system unless the configure process detects big endian. Since users building in a non-standard environment aren't using the configure process, `BIG_ENDIAN_ORDER` will need to be defined if using a big endian system.
4. wolfSSL benefits speed-wise from having a 64-bit type available. The configure process determines if long or long long is 64 bits and if so sets up a define. So if `sizeof(long)` is 8 bytes on your system, define `SIZEOF_LONG 8`. If it isn't but `sizeof(long long)` is 8 bytes, then define `SIZEOF_LONG_LONG 8`.
5. Try to build the library, and let us know if you run into any problems. If you need help, contact us at info@wolfssl.com.
6. Some defines that can modify the build are listed in the following sub-sections, below. For more verbose descriptions of many options, please see Build Options.

2.4.1 Building into Yocto Linux

wolfSSL also includes recipes for building wolfSSL on Yocto Linux and OpenEmbedded. These recipes are maintained within the meta-wolfSSL layer as a GitHub repository, here: https://github.com/wolfSSL/meta-wolfssl. Building wolfSSL on Yocto Linux will require Git and bitbake. The following steps list how to get some wolfSSL products (that recipes exist for) built on Yocto Linux.

1. **Cloning wolfSSL meta**

   This can be done through a git-clone command of the following URL: https://github.com/wolfSSL/meta-wolfssl

2. **Insert the "meta-wolfSSL" layer into the build's bblayers.conf**

   Within the BBLAYERS section, add the path to the location where meta-wolfssl was cloned into. Example:

   ```
   BBLAYERS ?= "... \
   /path/to/meta-wolfssl/ \
   ..."
   ```

3. **Build a wolfSSL product recipe**

   bitbake can be used to build one of the three following wolfSSL product recipes: *wolfssl*, *wolfssh*, and *wolfmqtt*. Simply pass one of those recipes into the bitbake command (example: `bitbake wolfssl`). This allows the user to personally confirm compilation succeeds without issues.

4. **Edit local.conf**

   The final step is to edit the build's local.conf file, which allows desired libraries to be included with the image being built. Edit the IMAGE_INSTALL_append line to include the name of the desired recipe(s). An example of this is shown below:

   ```
   IMAGE_INSTALL_apped = "wolfssl wolfssh wolfmqtt"
   ```

Once the image has been built, wolfSSL's default location (or related products from recipes) will be the /usr/lib/ directory.

Additionally, wolfSSL can be customized when building into Yocto by using the enable and disable options listed in Build Options. This requires creating a .bbappend file and placing it within the wolfSSL application/recipe layer. The contents of this file should include a line specifying content to concatenate onto the EXTRA_OECONF variable. An example of this is shown below to enable TLS 1.3 support through the TLS 1.3 enable option:

```
EXTRA_OECONF += "--enable-tls13"
```

Further documentation on building into Yocto can be found in the meta-wolfssl README, located here: https://github.com/wolfSSL/meta-wolfssl/blob/master/README.md

2.4.2 Building with Atollic TrueSTUDIO

Versions of wolfSSL following 3.15.5 include a TrueSTUDIO project file that is used to build wolfSSL on ARM M4-Cortex devices. The TrueSTUDIO project file simplifies the process of building on STM32 devices, is free to download, and is createed by Atollic - a part of ST Microelectronics. To build the wolfSSL static library project file in TrueSTUDIO, it will require the user perform the following steps after opening TrueSTUDIO:

1. Import the project into the workspace (File > Import)
2. Build the project (Project > Build project)

The build then includes the settings located inside of user_settings.h at build-time. The default content of the user_settings.h file is minimal, and does not contain many features. Users are able to modify this file and add or remove features with options listed in the remainder of this chapter.

2.4.3 Removing Features

The following defines can be used to remove features from wolfSSL. This can be helpful if you are trying to reduce the overall library footprint size. In addition to defining a NO_<feature-name> define, you can also remove the respective source file as well from the build (but not the header file).

2.4.3.1 NO_WOLFSSL_CLIENT Removes calls specific to the client and is for a server-only builds. You should only use this if you want to remove a few calls for the sake of size.

2.4.3.2 NO_WOLFSSL_SERVER Likewise removes calls specific to the server side.

2.4.3.3 NO_DES3 Removes the use of DES3 encryptions. DES3 is built-in by default because some older servers still use it and it's required by SSL 3.0. NO_DH and NO_AES are the same as the two above, they are widely used.

2.4.3.4 NO_DSA Removes DSA since it's being phased out of popular use.

2.4.3.5 NO_ERROR_STRINGS Disables error strings. Error strings are located in src/internal.c for wolfSSL or wolfcrypt/src/asn.c for wolfCrypt.

2.4.3.6 NO_HMAC Removes HMAC from the build.

NOTE: SSL/TLS depends on HMAC but if you are only using wolfCrypt IE build option WOLFCRYPT_ONLY then HMAC can be disabled in this case.

2.4.3.7 NO_MD4 Removes MD4 from the build, MD4 is broken and shouldn't be used.

2.4.3.8 NO_MD5 Removes MD5 from the build.

2.4.3.9 NO_SHA Removes SHA-1 from the build.

2.4.3.10 NO_SHA256 Removes SHA-256 from the build.

2.4.3.11 NO_PSK Turns off the use of the pre-shared key extension. It is built-in by default.

2.4.3.12 NO_PWDBASED Disables password-based key derivation functions such as PBKDF1, PBKDF2, and PBKDF from PKCS #12.

2.4.3.13 NO_RC4 Removes the use of the ARC4 stream cipher from the build. ARC4 is built-in by default because it is still popular and widely used.

2.4.3.14 NO_RABBIT and NO_HC128 Remove stream cipher extensions from the build.

2.4.3.15 NO_SESSION_CACHE Can be defined when a session cache is not needed. This should reduce memory use by nearly 3 kB.

2.4.3.16 NO_TLS Turns off TLS. We don't recommend turning off TLS.

2.4.3.17 SMALL_SESSION_CACHE Can be defined to limit the size of the SSL session cache used by wolfSSL. This will reduce the default session cache from 33 sessions to 6 sessions and save approximately 2.5 kB.

2.4.3.18 NO_RSA Removes support for the RSA algorithm.

2.4.3.19 WC_NO_RSA_OAEP Removes code for OAEP padding.

2.4.3.20 NO_AES_CBC Turns off AES-CBC algorithm support.

2.4.3.21 NO_DEV_URANDOM Disables the use of `/dev/urandom`

2.4.3.22 WOLFSSL_NO_SIGALG Disables the signature algorithms extension

2.4.3.23 NO_RESUME_SUITE_CHECK Disables the check of cipher suite when resuming a TLS connection

2.4.3.24 NO_ASN Removes support for ASN formatted certificate processing.

2.4.3.25 NO_OLD_TLS Removes support for SSLv3, TLSv1.0 and TLSv1.1

2.4.3.26 WOLFSSL_AEAD_ONLY Removes support for non-AEAD algorithms. AEAD stands for "authenticated encryption with associated data" which means these algorithms (such as AES-GCM) do not just encrypt and decrypt data, they also assure confidentiality and authenticity of that data.

2.4.4 Enabling Features Disabled by Default

2.4.4.1 WOLFSSL_CERT_GEN Turns on wolfSSL's certificate generation functionality. See Keys and Certificates for more information.

2.4.4.2 WOLFSSL_DER_LOAD Allows loading DER-formatted CA certs into the wolfSSL context (`WOLFSSL_CTX`) using the function `wolfSSL_CTX_der_load_verify_locations()`.

2.4.4.3 WOLFSSL_DTLS Turns on the use of DTLS, or datagram TLS. This isn't widely supported or used.

2.4.4.4 WOLFSSL_KEY_GEN Turns on wolfSSL's RSA key generation functionality. See Keys and Certificates for more information.

2.4.4.5 WOLFSSL_RIPEMD Enables RIPEMD-160 support.

2.4.4.6 WOLFSSL_SHA384 Enables SHA-384 support.

2.4.4.7 WOLFSSL_SHA512 Enables SHA-512 support.

2.4.4.8 DEBUG_WOLFSSL Builds in the ability to debug. For more information regarding debugging wolfSSL, see Debugging.

2.4.4.9 HAVE_AESCCM Enables AES-CCM support.

2.4.4.10 HAVE_AESGCM Enables AES-GCM support.

2.4.4.11 WOLFSSL_AES_XTS Enables AES-XTS support.

2.4.4.12 HAVE_CAMELLIA Enables Camellia support.

2.4.4.13 HAVE_CHACHA Enables ChaCha20 support.

2.4.4.14 HAVE_POLY1305 Enables Poly1305 support.

2.4.4.15 HAVE_CRL Enables Certificate Revocation List (CRL) support.

2.4.4.16 HAVE_CRL_IO Enables blocking inline HTTP request on the CRL URL. It will load the CRL into the `WOLFSSL_CTX` and apply it to all WOLFSSL objects created from it.

2.4.4.17 HAVE_ECC Enables Elliptical Curve Cryptography (ECC) support.

2.4.4.18 HAVE_LIBZ Is an extension that can allow for compression of data over the connection. It normally shouldn't be used, see the note below under configure notes libz.

2.4.4.19 HAVE_OCSP Enables Online Certificate Status Protocol (OCSP) support.

2.4.4.20 OPENSSL_EXTRA Builds even more OpenSSL compatibility into the library, and enables the wolfSSL OpenSSL compatibility layer to ease porting wolfSSL into existing applications which had been designed to work with OpenSSL. It is off by default.

2.4.4.21 TEST_IPV6 Turns on testing of IPv6 in the test applications. wolfSSL proper is IP neutral, but the testing applications use IPv4 by default.

2.4.4.22 HAVE_CSHARP Turns on configuration options needed for C# wrapper.

2.4.4.23 HAVE_CURVE25519 Turns on the use of curve25519 algorithm.

2.4.4.24 HAVE_ED25519 Turns on use of the ed25519 algorithm.

2.4.4.25 WOLFSSL_DH_CONST Turns off use of floating point values when performing Diffie Hellman operations and uses tables for XPOW() and XLOG(). Removes dependency on external math library.

2.4.4.26 WOLFSSL_TRUST_PEER_CERT Turns on the use of trusted peer certificates. This allows for loading in a peer certificate to match with a connection rather than using a CA. When turned on if a trusted peer certificate is matched than the peer cert chain is not loaded and the peer is considered verified. Using CAs is preferred.

2.4.4.27 WOLFSSL_STATIC_MEMORY Turns on the use of static memory buffers and functions. This allows for using static memory instead of dynamic.

2.4.4.28 WOLFSSL_SESSION_EXPORT Turns on the use of DTLS session export and import. This allows for serializing and sending/receiving the current state of a DTLS session.

2.4.4.29 WOLFSSL_ARMASM Turns on the use of ARMv8 hardware acceleration.

2.4.4.30 WC_RSA_NONBLOCK Turns on fast math RSA non-blocking support for splitting RSA operations into smaller chunks of work. Feature is enabled by calling wc_RsaSetNonBlock() and checking for FP_WOULDBLOCK return code.

2.4.4.31 WOLFSSL_RSA_VERIFY_ONLY Turns on small build for RSA verify only use. Should be used with the macros WOLFSSL_RSA_PUBLIC_ONLY, WOLFSSL_RSA_VERIFY_INLINE, NO_SIG_WRAPPER, and WOLFCRYPT_ONLY.

2.4.4.32 WOLFSSL_RSA_PUBLIC_ONLY Turns on small build for RSA public key only use. Should be used with the macro WOLFCRYPT_ONLY.

2.4.4.33 WOLFSSL_SHA3 Turns on build for SHA3 use. This is support for SHA3 Keccak for the sizes SHA3-224, SHA3-256, SHA3-384 and SHA3-512. In addition WOLFSSL_SHA3_SMALL can be used to trade off performance for resource use.

2.4.4.34 USE_ECDSA_KEYSZ_HASH_ALGO Will choose a hash algorithm that matches the ephemeral ECDHE key size or the next highest available. This workaround resolves issues with some peers that do not properly support scenarios such as a P-256 key hashed with SHA512.

2.4.4.35 WOLFSSL_ALT_CERT_CHAIN Allows CA's to be presented by peer, but not part of a valid chain. Default wolfSSL behavior is to require validation of all presented peer certificates. This also allows loading intermediate CA's as trusted and ignoring no signer failures for CA's up the chain to root. The alternate certificate chain mode only requires that the peer certificate validate to a trusted CA.

2.4.4.36 WOLFSSL_CUSTOM_CURVES Allow non-standard curves. Includes the curve "a" variable in calculation. Additional curve types can be enabled using HAVE_ECC_SECPR2, HAVE_ECC_SECPR3, HAVE_ECC_BRAINPOOL and HAVE_ECC_KOBLITZ.

2.4.4.37 HAVE_COMP_KEY Enables ECC compressed key support.

2.4.4.38 WOLFSSL_EXTRA_ALERTS Enables additional alerts to be sent during a TLS connection. This feature is also enabled automatically when --enable-opensslextra is used.

2.4.4.39 WOLFSSL_DEBUG_TLS Enables additional debugging print outs during a TLS connection

2.4.4.40 HAVE_BLAKE2 Enables Blake2s algorithm support

2.4.4.41 HAVE_FALLBACK_SCSV Enables Signaling Cipher Suite Value(SCSV) support on the server side. This handles the cipher suite 0x56 0x00 sent from a client to signal that no downgrade of TLS version should be allowed.

2.4.4.42 WOLFSSL_PSK_ONE_ID Enables support for only one PSK ID with TLS 1.3.

2.4.4.43 SHA256_MANY_REGISTERS A SHA256 version that keeps all data in registers and partially unrolls loops.

2.4.4.44 WOLFCRYPT_HAVE_SRP Enables wolfcrypt secure remote password support

2.4.4.45 WOLFSSL_MAX_STRENGTH Enables the strongest security features only and disables any weak or deprecated features. Results in slower performance due to near constant time execution to protect against timing based side-channel attacks.

2.4.4.46 HAVE_QSH Turns on support for cipher suites resistant to Shor's algorithm. QSH stands for "Quantum Safe Handshake".

2.4.4.47 WOLFSSL_STATIC_RSA Static ciphers are strongly discouraged and should never be used if avoidable. However there are still legacy systems that ONLY support static cipher suites. To that end if you need to connect to a legacy peer only supporting static RSA cipher suites use this to enable support for static RSA in wolfSSL. (See also `WOLFSSL_STATIC_PSK` and `WOLFSSL_STATIC_DH`)

2.4.4.48 WOLFSSL_STATIC_PSK Static ciphers are highly discouraged see `WOLFSSL_STATIC_RSA`

2.4.4.49 WOLFSSL_STATIC_DH Static ciphers are highly discouraged see `WOLFSSL_STATIC_RSA`

2.4.4.50 HAVE_NTRU Turns on support for NTRU cipher suites. NTRU offers a Quantum resistant Public Key solution. Read more about it on the WIKI page: https://en.wikipedia.org/wiki/NTRU

2.4.4.51 HAVE_NULL_CIPHER Turns on support for NULL ciphers. This option is highly discouraged from a security standpoint however some systems are too small to perform encrypt/decrypt operations and it is better to at least authenticate messages and peers to prevent message tampering than nothing at all!

2.4.4.52 HAVE_ANON Turns on support for anonymous cipher suites. (Never recommended, some valid use cases involving closed or private networks detached from the web)

2.4.5 Customizing or Porting wolfSSL

2.4.5.1 WOLFSSL_USER_SETTINGS If defined allows a user specific settings file to be used. The file must be named `user_settings.h` and exist in the include path. This is included prior to the standard `settings.h` file, so default settings can be overridden.

2.4.5.2 WOLFSSL_CALLBACKS Is an extension that allows debugging callbacks through the use of signals in an environment without a debugger, it is off by default. It can also be used to set up a timer with blocking sockets. Please see Callbacks for more information.

2.4.5.3 WOLFSSL_USER_IO Allows the user to remove automatic setting of the default I/O functions `EmbedSend()` and `EmbedReceive()`. Used for custom I/O abstraction layer (see Abstraction Layers for more details).

2.4.5.4 NO_FILESYSTEM Is used if stdio isn't available to load certificates and key files. This enables the use of buffer extensions to be used instead of the file ones.

2.4.5.5 NO_INLINE Disables the automatic inlining of small, heavily used functions. Turning this on will slow down wolfSSL and actually make it bigger since these are small functions, usually much smaller than function call setup/return. You'll also need to add `wolfcrypt/src/misc.c` to the list of compiled files if you're not using autoconf.

2.4.5.6 NO_DEV_RANDOM Disables the use of the default `/dev/random` random number generator. If defined, the user needs to write an OS-specific `GenerateSeed()` function (found in `wolfcrypt/src/random.c`).

2.4.5.7 NO_MAIN_DRIVER Is used in the normal build environment to determine whether a test application is called on its own or through the testsuite driver application. You'll only need to use it with the test files: `test.c`, `client.c`, `server.c`, `echoclient.c`, `echoserver.c`, and `testsuite.c`

2.4.5.8 NO_WRITEV Disables simulation of `writev()` semantics.

2.4.5.9 SINGLE_THREADED Is a switch that turns off the use of mutexes. wolfSSL currently only uses one for the session cache. If your use of wolfSSL is always single threaded you can turn this on.

2.4.5.10 USER_TICKS Allows the user to define their own clock tick function if time(0) is not wanted. Custom function needs second accuracy, but doesn't have to be correlated to Epoch. See `LowResTimer()` function in `wolfssl_int.c`.

2.4.5.11 USER_TIME Disables the use of time.h structures in the case that the user wants (or needs) to use their own. See `wolfcrypt/src/asn.c` for implementation details. The user will need to define and/or implement `XTIME()`, `XGMTIME()`, and `XVALIDATE_DATE()`.

2.4.5.12 USE_CERT_BUFFERS_1024 Enables 1024-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.4.5.13 USE_CERT_BUFFERS_2048 Enables 2048-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.4.5.14 CUSTOM_RAND_GENERATE_SEED Allows user to define custom function equivalent to `wc_GenerateSeed(byte* output, word32 sz)`.

2.4.5.15 CUSTOM_RAND_GENERATE_BLOCK Allows user to define custom random number generation function. Examples of use are as follows.

```
./configure --disable-hashdrbg
CFLAGS="-DCUSTOM_RAND_GENERATE_BLOCK= custom_rand_generate_block"
```

Or

```
/* RNG */
/* #define HAVE_HASHDRBG */
extern int custom_rand_generate_block(unsigned char* output, unsigned int sz);
```

2.4.5.16 NO_PUBLIC_GCM_SET_IV Use this if you have done your own custom hardware port and not provided a public implementation of `wc_AesGcmSetIV()`

2.4.5.17 NO_PUBLIC_CCM_SET_NONCE Use this if you have done your own custom hardware port and not provided a public implementation of `wc_AesGcmSetNonce()`

2.4.5.18 NO_GCM_ENCRYPT_EXTRA Use this if you have done your own custom hardwareport and not provided an implementation of `wc_AesGcmEncrypt_ex()`

2.4.5.19 WOLFSSL_STM32[F1 | F2 | F4 | F7 | L4] Use one of these defines when building for the appropriate STM32 device. Update `wolfssl-root/wolfssl/wolfcrypt/settings.h` section with regards to the wolfSSL porting guide (https://www.wolfssl.com/docs/porting-guide/) as appropriate.

2.4.5.20 WOLFSSL_STM32_CUBEMX When using the CubeMX tool to generate Hardware Abstraction Layer (HAL) API's use this setting to add appropriate support in wolfSSL.

2.4.5.21 WOLFSSL_CUBEMX_USE_LL When using the CubeMX tool to generate APIs there are two options, HAL (Hardware Abstraction Layer) or Low Layer (LL). Use this define to control which headers are include in `wolfssl-root/wolfssl/wolfcrypt/settings.h` in the `WOLFSSL_STM32[F1/F2/F4/F7/L4]` section.

2.4.5.22 NO_STM32_CRYPTO For when an STM32 part does not offer hardware crypto support

2.4.5.23 NO_STM32_HASH For when an STM32 part does not offer hardware hash support

2.4.5.24 NO_STM32_RNG For when an STM32 part does not offer hardware RNG support

2.4.5.25 XTIME_MS Macro to map a function for use to get the time in milliseconds when using TLS 1.3. Example being:

```
extern time_t m2mb_xtime_ms(time_t * timer);
#define XTIME_MS(tl) m2mb_xtime_ms((tl))
```

2.4.6 Reducing Memory or Code Usage

2.4.6.1 TFM_TIMING_RESISTANT Can be defined when using fast math (`USE_FAST_MATH`) on systems with a small stack size. This will get rid of the large static arrays.

2.4.6.2 WOLFSSL_SMALL_STACK Can be used for devices which have a small stack size. This increases the use of dynamic memory in `wolfcrypt/src/integer.c`, but can lead to slower performance.

2.4.6.3 ALT_ECC_SIZE If using fast math and RSA/DH you can define this to reduce your ECC memory consumption. Instead of using stack for ECC points it will allocate from the heap.

2.4.6.4 ECC_SHAMIR Uses variation of ECC math that is slightly faster, but doubles heap usage.

2.4.6.5 RSA_LOW_MEM When defined CRT is not used which saves on some memory but slows down RSA operations. It is off by default.

2.4.6.6 WOLFSSL_SHA3_SMALL When SHA3 is enabled this macro will reduce build size.

2.4.6.7 WOLFSSL_SMALL_CERT_VERIFY Verify the certificate signature without using DecodedCert. Doubles up on some code but allows smaller peak heap memory usage. Cannot be used with `WOLFSSL_NONBLOCK_OCSP`.

2.4.6.8 GCM_SMALL Option to reduce AES GCM code size by calculating at runtime instead of using tables. Possible options are: `GCM_SMALL`, `GCM_WORD32` or `GCM_TABLE`.

2.4.6.9 CURVED25519_SMALL Defines `CURVE25519_SMALL` and `ED25519_SMALL`.

2.4.6.10 CURVE25519_SMALL Use small memory option for curve25519. This uses less memory, but is slower.

2.4.6.11 ED25519_SMALL Use small memory option for ed25519. This uses less memory, but is slower.

2.4.6.12 USE_SLOW_SHA Reduces code size by not unrolling loops, which reduces performance for SHA.

2.4.6.13 USE_SLOW_SHA256 Reduces code size by not unrolling loops, which reduces performance for SHA. About 2k smaller and about 25% slower.

2.4.6.14 USE_SLOW_SHA512 Reduces code size by not unrolling loops, which reduces performance for SHA. Over twice as small, but 50% slower.

2.4.6.15 ECC_USER_CURVES Allow user to choose ECC curve sizes that are enabled. Only the 256-bit curve is enabled by default. To enable others use HAVE_ECC192, HAVE_ECC224, etc...

2.4.6.16 WOLFSSL_SP_SMALL If using SP math this will use smaller versions of the code.

2.4.6.17 WOLFSSL_SP_MATH Enable only SP math to reduce code size. Eliminates big integer math code such as normal (`integer.c`) or fast (`tfm.c`). Restricts key sizes and curves to only ones supported by SP.

2.4.7 Increasing Performance

2.4.7.1 USE_INTEL_SPEEDUP Enables use of Intel's AVX/AVX2 instructions for accelerating AES, ChaCha20, Poly1305, SHA256, SHA512, ED25519 and Curve25519.

2.4.7.2 WOLFSSL_AESNI Enables use of AES accelerated operations which are built into some Intel and AMD chipsets. When using this define, the `aes_asm.asm` (for Windows with at&t syntax) or `aes_asm.S` file is used to optimize via the Intel AES new instruction set (AESNI).

2.4.7.3 HAVE_INTEL_RDSEED Enable Intel's RDSEED for DRBG seed source.

2.4.7.4 HAVE_INTEL_RDRAND Enable Intel's RDRAND instruction for wolfSSL's random source.

2.4.7.5 USE_FAST_MATH Switches the big integer library to a faster one that uses assembly if possible. fastmath will speed up public key operations like RSA, DH, and DSA. The big integer library is generally the most portable and generally easiest to get going with, but the negatives to the normal big integer library are that it is slower and it uses a lot of dynamic memory. Because the stack memory usage can be larger when using fastmath, we recommend defining `TFM_TIMING_RESISTANT` as well when using this option.

2.4.7.6 FP_ECC Enables ECC Fixed Point Cache, which speeds up repeated operations against same private key. Can also define number of entries and LUT bits using FP_ENTRIES and FP_LUT to reduce default static memory usage.

2.4.8 GCM Performance Tuning

There are 4 variants of GCM performance:

- GCM_SMALL - Smallest footprint, slowest (FIPS validated)
- GCM_WORD32 - Medium (FIPS validated)
- GCM_TABLE - Fast (FIPS validated)
- GCM_TABLE_4BIT - Fastest (Not yet FIPS validated, will be included in FIPS 140-3!)

2.4.9 wolfSSL's proprietary Single Precision math support

Use these to speed up public key operations for specific keys sizes and curves that are common. Make sure to include the correct code files such as:

- `sp_c32.c`
- `sp_c64.c`
- `sp_arm32.c`
- `sp_arm64.c`
- `sp_armthumb.c`
- `sp_cortexm.c`
- `sp_int.c`
- `sp_x86_64.c`
- `sp_x86_64_asm.S`

2.4.9.1 WOLFSSL_SP Enable Single Precision math support.

2.4.9.2 WOLFSSL_SP_ASM Enable assembly speedups for Single Precision

2.4.9.3 WOLFSSL_HAVE_SP_RSA Single Precision RSA for 2048, 3072 and 4096 bit.

2.4.9.4 WOLFSSL_HAVE_SP_DH Single Precision DH for 2048, 3072 and 4096 bit.

2.4.9.5 WOLFSSL_HAVE_SP_ECC Single Precision ECC for SECP256R1.

2.4.9.6 WOLFSSL_ASYNC_CRYPT Adds support for Asynchronous Crypto[1]

2.4.10 Stack or Chip Specific Defines

wolfSSL can be built for a variety of platforms and TCP/IP stacks. Most of the following defines are located in `./wolfssl/wolfcrypt/settings.h` and are commented out by default. Each can be uncommented to enable support for the specific chip or stack referenced below.

2.4.10.1 IPHONE Can be defined if building for use with iOS.

2.4.10.2 THREADX Can be defined when building for use with the ThreadX RTOS (https://www.rtos.com).

2.4.10.3 MICRIUM Can be defined when building wolfSSL to enable support for Micrium's µC/OS-III RTOS (https://www.micrium.com).

2.4.10.4 MBED Can be defined when building for the mbed prototyping platform (https://www.mbed.org).

2.4.10.5 MICROCHIP_PIC32 Can be defined when building for Microchip's PIC32 platform (https://www.microchip.com).

2.4.10.6 MICROCHIP_TCPIP_V5 Can be defined specifically version 5 of microchip tcp/ip stack.

2.4.10.7 MICROCHIP_TCPIP Can be defined for microchip tcp/ip stack version 6 or later.

2.4.10.8 WOLFSSL_MICROCHIP_PIC32MZ Can be defined for PIC32MZ hardware cryptography engine.

2.4.10.9 FREERTOS Can be defined when building for FreeRTOS (https://www.freertos.org). If using LwIP, define `WOLFSSL_LWIP` as well.

2.4.10.10 FREERTOS_WINSIM Can be defined when building for the FreeRTOS windows simulator (https://www.freertos.org).

2.4.10.11 EBSNET Can be defined when using EBSnet products and RTIP.

2.4.10.12 WOLFSSL_LWIP Can be defined when using wolfSSL with the LwIP TCP/IP stack (https://savannah.nongnu.org/projects/lwip/).

2.4.10.13 WOLFSSL_GAME_BUILD Can be defined when building wolfSSL for a game console.

2.4.10.14 WOLFSSL_LSR Can be defined if building for LSR.

2.4.10.15 FREESCALE_MQX Can be defined when building for Freescale MQX/RTCS/MFS (https://www.freescale.com). This in turn defines `FREESCALE_K70_RNGA` to enable support for the Kinetis H/W Random Number Generator Accelerator

[1] Limited Software support, works best with Intel® QuickAssist Technology (Intel® QAT) and Cavium Nitrox V Processors

2.4.10.16 WOLFSSL_STM32F2 Can be defined when building for STM32F2. This define also enables STM32F2 hardware crypto and hardware RNG support in wolfSSL (https://www.st.com/internet/mcu/subclass/1520.jsp).

2.4.10.17 COMVERGE Can be defined if using Comverge settings.

2.4.10.18 WOLFSSL_QL Can be defined if using QL SEP settings.

2.4.10.19 WOLFSSL_EROAD Can be defined building for EROAD.

2.4.10.20 WOLFSSL_IAR_ARM Can be defined if build for IAR EWARM.

2.4.10.21 WOLFSSL_TIRTOS Can be defined when building for TI-RTOS.

2.4.10.22 WOLFSSL_ROWLEY_ARM Can be defined when building with Rowley CrossWorks.

2.4.10.23 WOLFSSL_NRF51 Can be defined when porting to Nordic nRF51.

2.4.10.24 WOLFSSL_NRF51_AES Can be defined to use built-in AES hardware for AES 128 ECB encrypt when porting to Nordic nRF51.

2.4.10.25 WOLFSSL_CONTIKI Can be defined to enable support for the Contiki operating system.

2.4.10.26 WOLFSSL_APACHE_MYNEWT Can be defined to enable the Apache Mynewt port layer.

2.4.10.27 WOLFSSL_APACHE_HTTPD Can be defined to enable support for the Apache HTTPD web server.

2.4.10.28 ASIO_USE_WOLFSSL Can be defined to make wolfSSL build as an ASIO-compatible version. ASIO then relies on the `BOOST_ASIO_USE_WOLFSSL` preprocessor define.

2.4.10.29 WOLFSSL_CRYPTOCELL Can be defined to enable using ARM CRYPTOCELL.

2.4.10.30 WOLFSSL_SIFIVE_RISC_V Can be defined to enable using RISC-V SiFive/HiFive port.

2.4.10.31 WOLFSSL_MDK_ARM Adds support for MDK ARM

2.4.10.32 WOLFSSL_MDK5 Adds support for MDK5 ARM

2.4.11 OS Specific Defines

2.4.11.1 USE_WINDOWS_API Specify use of windows library APIs' as opposed to Unix/Linux APIs'

2.4.11.2 WIN32_LEAN_AND_MEAN Adds support for the Microsoft win32 lean and mean build.

2.4.11.3 FREERTOS_TCP Adds support for the FREERTOS TCP stack

2.4.11.4 WOLFSSL_SAFERTOS Adds support for SafeRTOS

2.5 Build Options

The following are options which may be appended to the `./configure` script to customize how the wolfSSL library is built.

By default, wolfSSL only builds in shared mode, with static mode being disabled. This speeds up build times by a factor of two. Either mode can be explicitly disabled or enabled if desired.

2.5.1 --enable-debug

Enable wolfSSL debugging support. Enabling debug support allows easier debugging by compiling with debug information and defining the constant DEBUG_WOLFSSL which outputs messages to stderr. To turn debug on at runtime, call wolfSSL_Debugging_ON(). To turn debug logging off at runtime, call wolfSSL_Debugging_OFF(). For more information, see Debugging.

2.5.2 --enable-distro

Enable wolfSSL distro build.

2.5.3 --enable-singlethread

Enable single threaded mode, no multi thread protections.

Enabling single threaded mode turns off multi thread protection of the session cache. Only enable single threaded mode if you know your application is single threaded or your application is multithreaded and only one thread at a time will be accessing the library.

2.5.4 --enable-dtls

Enable wolfSSL DTLS support

Enabling DTLS support allows users of the library to also use the DTLS protocol in addition to TLS and SSL. For more information, see the DTLS section.

2.5.5 --disable-rng

Disable compiling and using RNG

2.5.6 --enable-sctp

Enable wolfSSL DTLS-SCTP support

2.5.7 --enable-openssh

Enable OpenSSH compatibility build

2.5.8 --enable-apachehttpd

Enable Apache httpd compatibility build

2.5.9 --enable-openvpn

Enable OpenVPN compatibility build

2.5.10 --enable-opensslextra

Enable extra OpenSSL API compatibility, increases the size

Enabling OpenSSL Extra includes a larger set of OpenSSL compatibility functions. The basic build will enable enough functions for most TLS/SSL needs, but if you're porting an application that uses 10s or 100s of OpenSSL calls, enabling this will allow better support. The wolfSSL OpenSSL compatibility layer is under active development, so if there is a function missing which you need, please contact us and we'll try to help. For more information about the OpenSSL Compatibility Layer, please see OpenSSL Compatibility.

2.5.11 --enable-opensslall

Enable all OpenSSL API.

2.5.12 --enable-maxstrength

Enable Max Strength build, allows TSLv1.2-AEAD-PFS ciphers only

2.5.13 --disable-harden

Disable Hardened build, Enables Timing Resistance and Blinding

2.5.14 --enable-ipv6

Enable testing of IPv6, wolfSSL proper is IP neutral

Enabling IPV6 changes the test applications to use IPv6 instead of IPv4. wolfSSL proper is IP neutral, either version can be used, but currently the test applications are IP dependent.

2.5.15 --enable-bump

Enable SSL Bump build

2.5.16 --enable-leanpsk

Enable Lean PSK build.

Very small build using PSK, and eliminating many features from the library. Approximate build size for wolfSSL on an embedded system with this enabled is 21kB.

2.5.17 --enable-leantls

Implements a lean TLS 1.2 client only (no client auth), ECC256, AES128 and SHA256 w/o Shamir. Meant to be used by itself at the moment and not in conjunction with other build options.

Enabling produces a small footprint TLS client that supports TLS 1.2 client only (no client auth), ECC256, AES128 and SHA256 w/o Shamir. Meant to be used by itself at the moment and not in conjunction with other build options.

2.5.18 --enable-bigcache

Enable a big session cache.

Enabling the big session cache will increase the session cache from 33 sessions to 20,027 sessions. The default session cache size of 33 is adequate for TLS clients and embedded servers. The big session cache is suitable for servers that aren't under heavy load, basically allowing 200 new sessions per minute or so.

2.5.19 --enable-hugecache

Enable a huge session cache.

Enabling the huge session cache will increase the session cache size to 65,791 sessions. This option is for servers that are under heavy load, over 13,000 new sessions per minute are possible or over 200 new sessions per second.

2.5.20 --enable-smallcache

Enable small session cache.

Enabling the small session cache will cause wolfSSL to only store 6 sessions. This may be useful for embedded clients or systems where the default of nearly 3kB is too much RAM. This define uses less than 500 bytes of RAM.

2.5.21 --enable-savesession

Enable persistent session cache.

Enabling this option will allow an application to persist (save) and restore the wolfSSL session cache to/from memory buffers.

2.5.22 --enable-savecert

Enable persistent cert cache.

Enabling this option will allow an application to persist (save) and restore the wolfSSL certificate cache to/from memory buffers.

2.5.23 --enable-atomicuser

Enable Atomic User Record Layer.

Enabling this option will turn on User Atomic Record Layer Processing callbacks. This will allow the application to register its own MAC/encrypt and decrypt/verify callbacks.

2.5.24 --enable-pkcallbacks

Enable Public Key Callbacks

2.5.25 --enable-sniffer

Enable wolfSSL sniffer support.

Enabling sniffer (SSL inspection) support will allow the collection of SSL traffic packets as well as the ability to decrypt those packets with the correct key file.

Currently the sniffer supports the following RSA ciphers:

CBC ciphers:

- AES-CBC
- Camellia-CBC
- 3DES-CBC

Stream ciphers:

- RC4
- Rabbit
- HC-128

2.5.26 --enable-aesgcm

Enable AES-GCM support.

Eabling this option will turn on Public Key callbacks, allowing the application to register its own ECC sign/verify and RSA sign/verify and encrypt/decrypt callbacks.

2.5.27 --enable-aesccm

Enable AES-CCM support

Enabling AES-GCM will add these cipher suites to wolfSSL. wolfSSL offers four different implementations of AES-GCM balancing speed versus memory consumption. If available, wolfSSL will use 64-bit or 32-bit math. For embedded applications, there is a speedy 8-bit version that uses RAM-based lookup tables (8KB per session) which is speed comparable to the 64-bit version and a slower 8-bit version that doesn't take up any additional RAM. The –enable-aesgcm configure option may be modified with the options `=word32`, `=table`, or `=small`, i.e. `--enable-aesgcm=table`.

2.5.28 --disable-aescbc

Used to with `--disable-aescbc` to compile out AES-CBC

AES-GCM will enable Counter with CBC-MAC Mode with 8-byte authentication (CCM-8) for AES.

2.5.29 --enable-aescfb

Turns on AES-CFB mode support

2.5.30 --enable-aesctr

Enable wolfSSL AES-CTR support

Enabling AES-CTR will enable Counter mode.

2.5.31 --enable-aesni

Enable wolfSSL Intel AES-NI support

Enabling AES-NI support will allow AES instructions to be called directly from the chip when using an AES-NI supported chip. This provides speed increases for AES functions. See Features for more details regarding AES-NI.

2.5.32 --enable-intelasm

Enable ASM speedups for Intel and AMD processors.

Enabling the intelasm option for wolfSSL will utilize expanded capabilities of your processor that dramatically enhance AES performance. The instruction sets leveraged when configure option is enabled include AVX1, AVX2, BMI2, RDRAND, RDSEED, AESNI, and ADX. These were first introduced into Intel processors and AMD processors have started adopting them in recent years. When enabled, wolfSSL will check the processor and take advantage of the instruction sets your processor supports.

2.5.33 --enable-camellia

Enable Camellia support

2.5.34 --enable-md2

Enable MD2 support

2.5.35 --enable-nullcipher

Enable wolfSSL NULL cipher support (no encryption)

2.5.36 --enable-ripemd

Enable wolfSSL RIPEMD-160 support

2.5.37 --enable-blake2

Enable wolfSSL BLAKE2 support

2.5.38 --enable-blake2s

Enable wolfSSL BLAKE2s support

2.5.39 --enable-sha3

Enabled by default on x86_64 and Aarch64.

Enables wolfSSL SHA3 support (=small for small build)

2.5.40 --enable-sha512

Enabled by default on x86_64.

Enable wolfSSL SHA-512 support

2.5.41 --enable-sessioncerts

Enable session cert storing

2.5.42 --enable-keygen

Enable key generation

2.5.43 --enable-certgen

Enable cert generation

2.5.44 --enable-certreq

Enable cert request generation

2.5.45 --enable-sep

Enable SEP extensions

2.5.46 --enable-hkdf

Enable HKDF (HMAC-KDF)

2.5.47 --enable-x963kdf

Enable X9.63 KDF support

2.5.48 --enable-dsa

Enable Digital Signature Algorithm (DSA).

NIST approved digital signature algorithm along with RSA and ECDSA as defined by FIPS 186-4 and are used to generate and verify digital signatures if used in conjunction with an approved hash function as defined by the Secure Hash Standard (FIPS 180-4).

2.5.49 --enable-eccshamir

Enabled by default on x86_64

Enable ECC Shamir

2.5.50 --enable-ecc

Enabled by default on x86_64

Enable ECC.

Enabling this option will build ECC support and cipher suites into wolfSSL.

2.5.51 --enable-ecccustcurves

Enable ECC custom curves (=all to enable all curve types)

2.5.52 --enable-compkey

Enable compressed keys support

2.5.53 --enable-curve25519

Enable Curve25519 (or `--enable-curve25519=small` for CURVE25519_SMALL).

An elliptic curve offering 128 bits of security and to be used with ECDH key agreement (see Cross Compiling). Enabling curve25519 option allows for the use of the curve25519 algorithm. The default curve25519 is set to use more memory but have a faster run time. To have the algorithm use less memory the option `--enable-curve25519=small` can be used. Although using less memory there is a trade off in speed.

2.5.54 --enable-ed25519

Enable ED25519 (or `--enable-ed25519=small` for ED25519_SMALL)

Enabling ed25519 option allows for the use of the ed25519 algorithm. The default ed25519 is set to use more memory but have a faster run time. To have the algorithm use less memory the option `--enable-ed25519=small` can be used. Like with curve25519 using this enable option less is a trade off between speed and memory.

2.5.55 --enable-fpecc

Enable Fixed Point cache ECC

2.5.56 --enable-eccencrypt

Enable ECC encrypt

2.5.57 --enable-psk

Enable PSK (Pre Shared Keys)

2.5.58 --disable-errorstrings

Disable the error strings table

2.5.59 --disable-oldtls

Disable old TLS version < 1.2

2.5.60 --enable-sslv3

Enable SSL version 3.0

2.5.61 --enable-stacksize

Enable stack size info on examples

2.5.62 --disable-memory

Disable memory callbacks

2.5.63 --disable-rsa

Disable RSA

2.5.64 --enable-rsapss

Enable RSA-PSS

2.5.65 --disable-dh

Disable DH

2.5.66 --enable-anon

Enable Anonymous

2.5.67 --disable-asn

Disable ASN

2.5.68 --disable-aes

Disable AES

2.5.69 --disable-coding

Disable Coding base 16/64

2.5.70 --enable-base64encode

Enabled by default on x86_64

Enable Base64 encoding

2.5.71 --disable-des3

Disable DES3

2.5.72 --enable-idea

Enable IDEA Cipher

2.5.73 --enable-arc4

Enable ARC4

2.5.74 --disable-md5

Disable MD5

2.5.75 --disable-sha

Disable SHA

2.5.76 --enable-webserver

Enable Web Server.

This turns on functions required over the standard build that will allow full functionality for building with the yaSSL Embedded Web Server.

2.5.77 --enable-hc128

Enable streaming cipher HC-128

2.5.78 --enable-rabbit

Enable streaming cipher RABBIT

2.5.79 --enable-fips

Enable FIPS 140-2 (Must have license to implement.)

2.5.80 --enable-sha224

Enabled by default on x86_64

Enable wolfSSL SHA-224 support

2.5.81 --disable-poly1305

Disable wolfSSL POLY1305 support

2.5.82 --disable-chacha

Disable CHACHA

2.5.83 --disable-hashdrbg

Disable Hash DRBG support

2.5.84 --disable-filesystem

Disable Filesystem support.

This makes it easier to disable filesystem use. This option defines `NO_FILESYSTEM`.

2.5.85 --disable-inline

Disable inline functions.

Disabling this option disables function inlining in wolfSSL. Function placeholders that are not linked against but, rather, the code block is inserted into the function call when function inlining is enabled.

2.5.86 --enable-ocsp

Enable Online Certificate Status Protocol (OCSP).

Enabling this option adds OCSP (Online Certificate Status Protocol) support to wolfSSL. It is used to obtain the revocation status of x.509 certificates as described in RFC 6960.

2.5.87 --enable-ocspstapling

Enable OCSP Stapling

2.5.88 --enable-ocspstapling2

Enable OCSP Stapling version 2

2.5.89 --enable-crl

Enable CRL (Certificate Revocation List)

2.5.90 --enable-crl-monitor

Enable CRL Monitor.

Enabling this option adds the ability to have wolfSSL actively monitor a specific CRL (Certificate Revocation List) directory.

2.5.91 --enable-sni

Enable Server Name Indication (SNI).

Enabling this option will turn on the TLS Server Name Indication (SNI) extension.

2.5.92 --enable-maxfragment

Enable Maximum Fragment Length.

Enabling this option will turn on the TLS Maximum Fragment Length extension.

2.5.93 --enable-alpn

Enable Application Layer Protocol Negotiation (ALPN)

2.5.94 --enable-truncatedhmac

Enable Truncated Keyed-hash MAC (HMAC).

Enabling this option will turn on the TLS Truncated HMAC extension.

2.5.95 --enable-renegotiation-indication

Enable Renegotiation Indication.

As described in RFC 5746, this specification prevents an SSL/TLS attack involving renegotiation splicing by tying the renegotiations to the TLS connection they are performed over.

2.5.96 --enable-secure-renegotiation

Enable Secure Renegotiation

2.5.97 --enable-supportedcurves

Enable Supported Elliptic Curves.

Enabling this option will turn on the TLS Supported ECC Curves extension.

2.5.98 --enable-session-ticket

Enable Session Ticket

2.5.99 --enable-extended-master

Enable Extended Master Secret

2.5.100 --enable-tlsx

Enable all TLS extensions.

Enabling this option will turn on all TLS extensions currently supported by wolfSSL.

2.5.101 --enable-pkcs7

Enable PKCS#7 support

2.5.102 --enable-pkcs11

Enable PKCS#11 access

2.5.103 --enable-ssh

Enable wolfSSH options

2.5.104 --enable-scep

Enable wolfSCEP (Simple Certificate Enrollment Protocol)

As defined by the Internet Engineering Task Force, Simple Certificate Enrollment Protocol is a PKI that leverages PKCS#7 and PKCS#10 over HTTP. CERT notes that SCEP does not strongly authenticate certificate requests.

2.5.105 --enable-srp

Enable Secure Remote Password

2.5.106 --enable-smallstack

Enable Small Stack Usage

2.5.107 --enable-valgrind

Enable valgrind for unit tests.

Enabling this option will turn on valgrind when running the wolfSSL unit tests. This can be useful for catching problems early on in the development cycle.

2.5.108 --enable-testcert

Enable Test Cert.

When this option is enabled, it exposes part of the ASN certificate API that is usually not exposed. This can be useful for testing purposes, as seen in the wolfCrypt test application (`wolfcrypt/test/test.c`).

2.5.109 --enable-iopool

Enable I/O Pool example

2.5.110 --enable-certservice

Enable certificate service (Windows Servers)

2.5.111 --enable-jni

Enable wolfSSL JNI

2.5.112 --enable-lighty

Enable lighttpd/lighty

2.5.113 --enable-stunnel

Enable stunnel

2.5.114 --enable-md4

Enable MD4

2.5.115 --enable-pwdbased

Enable PWDBASED

2.5.116 --enable-scrypt

Enable SCRYPT

2.5.117 --enable-cryptonly

Enable wolfCrypt Only build

2.5.118 --enable-fastmath

Enabled by default on x86_64

Enable fast math ops.

Enabling fastmath will speed up public key operations like RSA, DH, and DSA. By default, wolfSSL uses the normal big integer math library. This is generally the most portable and generally easiest to get going with. The negatives to the normal big integer library are that it is slower and it uses a lot of dynamic memory. This option switches the big integer library to a faster one that uses assembly if possible. Assembly inclusion is dependent on compiler and processor combinations. Some combinations will need additional configure flags and some may not be possible. Help with optimizing fastmath with new assembly routines is available on a consulting basis.

On ia32, for example, all of the registers need to be available so high optimization and omitting the frame pointer needs to be taken care of. wolfSSL will add `-O3 -fomit-frame-pointer` to GCC for non debug builds. If you're using a different compiler you may need to add these manually to `CFLAGS` during configure.

OS X will also need `-mdynamic-no-pic` added to `CFLAGS`. In addition, if you're building in shared mode for ia32 on OS X you'll need to pass options to `LDFLAGS` as well:

```
LDFLAGS="-Wl,-read_only_relocs,warning"
```

This gives warning for some symbols instead of errors.

fastmath also changes the way dynamic and stack memory is used. The normal math library uses dynamic memory for big integers. Fastmath uses fixed size buffers that hold 4096 bit integers by default, allowing for 2048 bit by 2048 bit multiplications. If you need 4096 bit by 4096 bit multiplications then change `FP_MAX_BITS` in `wolfssl/wolfcrypt/tfm.h`. As `FP_MAX_BITS` is increased, this will also increase the runtime stack usage since the buffers used in the public key operations will now be larger. A couple of functions in the library use several temporary big integers, meaning the stack can get relatively large. This should only come into play on embedded systems or in threaded environments where the stack size is set to a low value. If stack corruption occurs with fastmath during public key operations in those environments, increase the stack size to accommodate the stack usage.

If you are enabling fastmath without using the autoconf system, you'll need to define `USE_FAST_MATH` and add `tfm.c` to the wolfSSL build instead of `integer.c`.

Since the stack memory can be large when using fastmath, we recommend defining `TFM_TIMING_RESISTANT` when using the fastmath library. This will get rid of large static arrays.

2.5.119 --enable-fasthugemath

Enable fast math + huge code

Enabling fasthugemath includes support for the fastmath library and greatly increases the code size by unrolling loops for popular key sizes during public key operations. Try using the benchmark utility before and after using fasthugemath to see if the slight speedup is worth the increased code size.

2.5.120 --disable-examples

Disable building examples.

When enabled, the wolfSSL example applications will be built (client, server, echoclient, echoserver).

2.5.121 --disable-crypttests

Disable Crypt Bench/Test

2.5.122 --enable-fast-rsa

Enable RSA using Intel IPP.

Enabling fast-rsa speeds up RSA operations by using IPP libraries. It has a larger memory consumption then the default RSA set by wolfSSL. If IPP libraries can not be found an error message will be displayed during configuration. The first location that autoconf will look is in the directory `wolfssl_root/IPP` the second is standard location for libraries on the machine such as `/usr/lib/` on linux systems.

The libraries used for RSA operations are in the directory `wolfssl-X.X.X/IPP/` where X.X.X is the current wolfSSL version number. Building from the bundled libraries is dependent on the directory location and name of IPP so the file structure of the subdirectory IPP should not be changed.

When allocating memory the fast-rsa operations have a memory tag of `DYNAMIC_TYPE_USER_CRYPTO`. This allows for viewing the memory consumption of RSA operations during run time with the fast-rsa option.

2.5.123 --enable-staticmemory

Enable static memory use

2.5.124 --enable-mcapi

Enable Microchip API

2.5.125 --enable-asynccrypt

Enable Asynchronous Crypto

2.5.126 --enable-sessionexport

Enable export and import of sessions

2.5.127 --enable-aeskeywrap

Enable AES key wrap support

2.5.128 --enable-jobserver

Values: `yes` (default) / no / #

When using `make` this builds wolfSSL using a multithreaded build, `yes` (default) detects the number of CPU cores and builds using a recommended amount of jobs for that count, # to specify an exact number. This works in a similar way to the `make -j` option.

2.5.129 --enable-shared[=PKGS]

Building shared wolfSSL libraries [default=yes]

Disabling the shared library build will exclude a wolfSSL shared library from being built. By default only a shared library is built in order to save time and space.

2.5.130 --enable-static[=PKGS]

Building static wolfSSL libraries [default=no]

2.5.131 --with-ntru=PATH

Path to NTRU install (default `/usr/`).

This turns on the ability for wolfSSL to use NTRU cipher suites. NTRU is now available under the GPLv2 from Security Innovation. The NTRU bundle may be downloaded from the Security Innovation GitHub repository available at https://github.com/NTRUOpenSourceProject/ntru-crypto.

2.5.132 --with-libz=PATH

Optionally include libz for compression.

Enabling libz will allow compression support in wolfSSL from the libz library. Think twice about including this option and using it by calling `wolfSSL_set_compression()`. While compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

2.5.133 --with-cavium

Path to cavium/software directory.

2.5.134 --with-user-crypto

Path to USER_CRYPTO install (default `/usr/local`).

2.5.135 --enable-rsavfy

Enables RSA verify only support (**note** requires `--enable-cryptonly`)

2.5.136 --enable-rsapub

Default value: Enabled RSA public key only support (**note** requires `--enable-cryptonly`)

2.5.137 --enable-sp

Enable single-precision math for RSA, DH, and ECC to improve performance.

2.5.138 --enable-sp-asm

Enable single-precision assembly implementation.

Can be used to enable single-precision performance improvements through assembly with ARM and 64-bit ARM architectures.

2.5.139 --enable-armasm

Enables ARMv8 ASM support.

The default configure sets mcpu or mfpu based on 64 vs 32 bit system. It does not overwrite mcpu or mfpu setting passed in by use of CPPFLAGS. On some compilers `-mstrict-align` may be needed due to the constraiants and `-mstrict-align` is now also set by default unless a user passes in mcpu/mfpu flags with CPPFLAGS.

2.5.140 --disable-tlsv12

Disable TLS 1.2 support

2.5.141 --enable-tls13

Enable TLS 1.3 support

This build option can be combined with `--disable-tlsv12` and `--disable-oldtls` to produce a wolfSSL build that is only TLS 1.3.

2.5.142 --enable-all

Enables all wolfSSL features, excluding SSL v3

2.5.143 --enable-xts

Enables AES-XTS mode

2.5.144 --enable-asio

Enables ASIO.

Requires that the options `--enable-opensslextra` and `--enable-opensslall` be enabled when configuring wolfSSL. If these two options are not enabled, then the autoconf tool will automatically enable these options to enable ASIO when configuring wolfSSL.

2.5.145 --enable-qt

Enables Qt 5.12 onwards support.

Enables wolfSSL build settings compatible with the wolfSSL Qt port. Patch file is required from wolfSSL for patching Qt source files.

2.5.146 --enable-qt-test

Enable Qt test compatibility build.

Enables support for building wolfSSL for compatibility with running the built-in Qt tests.

2.5.147 --enable-apache-httpd

Enables Apache httpd support

2.5.148 --enable-afalg

Enables use of Linux module AF_ALG for hardware accleration. Additional Xilinx use with `=xilinx`, `=xilinx-rsa`, `=xilinx-aes`, `=xilinx-sha3`

Is similar to `--enable-devcrypto` in that it leverages a Linux kernel module (AF_ALG) for offloading crypto operations. On some hardware the module has performance accelerations available through the Linux crypto drivers. In the case of Petalinux with Xilinx the flag `--enable-afalg=xilinx` can be used to tell wolfSSL to use the Xilinx interface for AF_ALG.

2.5.149 --enable-devcrypto

Enables use of Linux `/dev/crypto` for hardware acceleration.

Has the ability to receive arguments, being able to receive any combination of `aes` (all aes support), `hash` (all hash algorithms), and `cbc` (aes-cbc only). If no options are given, it will default to using `all`.

2.5.150 --enable-mcast

Enable wolfSSL DTLS multicast support

2.5.151 --disable-pkcs12

Disable PKCS12 code

2.5.152 --enable-fallback-scsv

Enables Signaling Cipher Suite Value(SCSV)

2.5.153 --enable-psk-one-id

Enables support for single PSK ID with TLS 1.3

2.6 Cross Compiling

Many users on embedded platforms cross compile wolfSSL for their environment. The easiest way to cross compile the library is to use the `./configure` system. It will generate a Makefile which can then be used to build wolfSSL.

When cross compiling, you'll need to specify the host to `./configure`, such as:

```
./configure --host=arm-linux
```

You may also need to specify the compiler, linker, etc. that you want to use:

```
./configure --host=arm-linux CC=arm-linux-gcc AR=arm-linux-ar RANLIB=arm-linux
```

There is a bug in the configure system which you might see when cross compiling and detecting user overriding malloc. If you get an undefined reference to `rpl_malloc` and/or `rpl_realloc`, please add the following to your `./configure` line:

```
ac_cv_func_malloc_0_nonnull=yes ac_cv_func_realloc_0_nonnull=yes
```

After correctly configuring wolfSSL for cross-compilation, you should be able to follow standard autoconf practices for building and installing the library:

```
make
sudo make install
```

If you have any additional tips or feedback about cross compiling wolfSSL, please let us know at info@wolfssl.com.

2.6.1 Example cross compile configure options for toolchain builds

2.6.1.1 armebv7-eabihf-glibc

```
./configure --host=armeb-linux \
        CC=armeb-linux-gcc LD=armeb-linux-ld \
        AR=armeb-linux-ar \
        RANLIB=armeb-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.6.1.2 armv5-eabi-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.6.1.3 armv6-eabihf-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.6.1.4 armv7-eabihf-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.6.1.5 armv7m-uclibc

```
./configure --enable-static --disable-shared \--host=arm-linux CC=arm-linux-gcc \
        LD=arm-linux-ld AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.6.1.6 arm-none-eabi-gcc

```
./configure --host=arm-none-eabi \
        CC=arm-none-eabi-gcc LD=arm-none-eabi-ld \
        AR=arm-none-eabi-ar RANLIB=arm-none-eabi-ranlib \
        CFLAGS="-DNO_WOLFSSL_DIR \
        -DWOLFSSL_USER_IO -DNO_WRITEV \
        -mcpu=cortex-m4 -mthumb -Os \
        -specs=rdimon.specs" CPPFLAGS="-I./"
```

2.6.1.7 mips32–glibc

```
./configure --host=mips-linux \
        CC=mips-linux-gcc LD=mips-linux-ld \
        AR=mips-linux-ar \
        RANLIB=mips-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.6.1.8 PowerPc64le-Power8-Glibc

```
./configure --host=powerpc64le-buildroot-linux-gnu \
        CC=powerpc64le-buildroot-linux-gnu-gcc \
        LD=powerpc64le-buildroot-linux-gnu-ld \
        AR=powerpc64le-buildroot-linux-gnu-ar \
        RANLIB=powerpc64le-buildroot-linux-gnu-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.6.1.9 x86-64-core-i7-glibc

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.6.1.10 x86-64-core-i7-musl

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \CPPFLAGS="-I./"
```

2.6.1.11 x86-64-core-i7-uclibc

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.7 2.7 Building Ports

wolfSSL has been ported to many environments and devices. A portion of these ports and accompanying documentation for them is located in the directory `wolfssl-X.X.X/IDE`, where X.X.X is the current wolfSSL version number. This directory also contains helpful information and code for IDE's used to build wolfSSL for the environments.

PORT Lists:

- Arduino
- LPCXPRESSO
- Wiced Studio
- CSBench
- SGX Windows and Linux
  - These directories (`wolfssl/IDE/WIN-SGX` and `wolfssl/IDE/LINUX-SGX`) contain Makefiles for and Visual Studio solutions for building wolfSSL as a library to be used in an Intel SGX project.
- Hexagon
  - This directory (`wolfssl/IDE/HEXAGON`) contains a Makefile for building with the Hexagon tool chain. It can be used to build wolfSSL for offloading ECC verify operations to a DSP processor. The directory contains a README file to help with the steps required to build.
- Hexiwear
- NetBurner M68K
  - In the directory (`wolfssl/IDE/M68K`) there is a Makefile for building wolfSSL for a MCF5441X device using the Netburner RTOS.
- Renesas
  - This directory (`wolfssl/IDE/Renesas`) contains multiple builds for different Renesas devices. It also has example builds that demonstrate using hardware acceleration.
- XCode
- Eclipse
- Espressif
- IAR-EWARM
- Kinetis Design Studio (KDS)
- Rowley Crossworks ARM
- OpenSTM32
- RISCV
- Zephyr
- Mynewt
- INTIME-RTOS

3 Getting Started

3.1 General Description

wolfSSL, formerly CyaSSL, is about 10 times smaller than yaSSL and up to 20 times smaller than OpenSSL when using the compile options described in Chapter 2. User benchmarking and feedback also reports dramatically better performance from wolfSSL vs. OpenSSL in the vast majority of standard SSL operations.

For instructions on the build process please see Chapter 2.

3.2 Testsuite

The testsuite program is designed to test the ability of wolfSSL and its cryptography library, wolfCrypt, to run on the system.

wolfSSL needs all examples and tests to be run from the wolfSSL home directory. This is because it finds certs and keys from ./certs. To run testsuite, execute:

```
./testsuite/testsuite.test
```

Or when using autoconf:

```
make test
```

On *nix or Windows the examples and testsuite will check to see if the current directory is the source directory and if so, attempt to change to the wolfSSL home directory. This should work in most setup cases, if not, just use the first method above and specify the full path.

On a successful run you should see output like this, with additional output for unit tests and cipher suite tests:

```
------------------------------------------------------------------------------
 wolfSSL version 4.8.1
------------------------------------------------------------------------------
error    test passed!
MEMORY   test passed!
base64   test passed!
base16   test passed!
asn      test passed!
RANDOM   test passed!
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
SHA-3    test passed!
Hash     test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
HMAC-SHA3   test passed!
HMAC-KDF    test passed!
GMAC     test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
```

```
AES      test passed!
AES192   test passed!
AES256   test passed!
AES-GCM  test passed!
RSA      test passed!
DH       test passed!
PWDBASED test passed!
OPENSSL  test passed!
OPENSSL (EVP MD) passed!
OPENSSL (PKEY0) passed!
OPENSSL (PKEY1) passed!
OPENSSL (EVP Sign/Verify) passed!
ECC      test passed!
logging  test passed!
mutex    test passed!
memcb    test passed!
Test complete
Alternate cert chain used
 issuer : /C=US/ST=Montana/L=Bozeman/O=Sawtooth/OU=Consulting/CN=www.wolfssl.com/emailAddress=
     info@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL/OU=Support/CN=www.wolfssl.com/emailAddress=
     info@wolfssl.com
 altname = example.com
Alternate cert chain used
 issuer : /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www.wolfssl.com/
     emailAddress=info@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www.wolfssl.com/
     emailAddress=info@wolfssl.com
 altname = example.com
 serial number:01
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL signature algorithm is RSA-SHA256
SSL curve name is SECP256R1
Session timeout set to 500 seconds
Client Random :  serial number:f1:5c:99:43:66:3d:96:04
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL signature algorithm is RSA-SHA256
SSL curve name is SECP256R1
1DC16A2C0D3AC49FC221DD5B8346B7B38CB9899B7A402341482183Server Random : 1679
    E50DBBBB3DB88C90F600C4C578F4F5D3CEAEC9B16BCCA215C276B448
765A1385611D6A
Client message: hello wolfssl!
I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers = TLS13-AES128-GCM-SHA256:TLS13-AES256-GCM-SHA384:TLS13-CHACHA20-POLY1305-SHA256:DHE-RSA-
    AES128-SHA:DHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES128-
    SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-GCM-
    SHA256:DHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:
    ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-SHA256:ECDHE-
    ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-CHACHA20-
    POLY1305:ECDHE-ECDSA-CHACHA20-POLY1305:DHE-RSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305-
```

```
    OLD:ECDHE-ECDSA-CHACHA20-POLY1305-OLD:DHE-RSA-CHACHA20-POLY1305-OLD
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  input
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  /tmp/output-gNQWZL

All tests passed!
```

This indicates that everything is configured and built correctly. If any of the tests fail, make sure the build system was set up correctly. Likely culprits include having the wrong endianness or not properly setting the 64-bit type. If you've set anything to the non-default settings try removing those, rebuilding wolfSSL, and then re-testing.

3.3 Client Example

You can use the client example found in examples/client to test wolfSSL against any SSL server. To see a list of available command line runtime options, run the client with the `--help` argument:

```
./examples/client/client --help
```

Which returns:

```
wolfSSL client 4.8.1 NOTE: All files relative to wolfSSL home dir
Max RSA key size in bits for build is set at : 4096
-? <num>    Help, print this usage
            0: English, 1: Japanese
--help      Help, in English
-h <host>   Host to connect to, default 127.0.0.1
-p <num>    Port to connect on, not 0, default 11111
-v <num>    SSL version [0-4], SSLv3(0) - TLS1.3(4)), default 3
-V          Prints valid ssl version numbers, SSLv3(0) - TLS1.3(4)
-l <str>    Cipher suite list (: delimited)
-c <file>   Certificate file,           default ./certs/client-cert.pem
-k <file>   Key file,                   default ./certs/client-key.pem
-A <file>   Certificate Authority file, default ./certs/ca-cert.pem
-Z <num>    Minimum DH key bits,        default 1024
-b <num>    Benchmark <num> connections and print stats
-B <num>    Benchmark throughput using <num> bytes and print stats
-d          Disable peer checks
-D          Override Date Errors example
-e          List Every cipher suite available,
-g          Send server HTTP GET
-u          Use UDP DTLS, add -v 2 for DTLSv1, -v 3 for DTLSv1.2 (default)
-m          Match domain name in cert
-N          Use Non-blocking sockets
-r          Resume session
-w          Wait for bidirectional shutdown
-M <prot>   Use STARTTLS, using <prot> protocol (smtp)
-f          Fewer packets/group messages
-x          Disable client cert/key loading
-X          Driven by eXternal test case
-j          Use verify callback override
-n          Disable Extended Master Secret
-H <arg>    Internal tests [defCipherList, exitWithRet, verifyFail, useSupCurve,
                            loadSSL, disallowETM]
-J          Use HelloRetryRequest to choose group for KE
-K          Key Exchange for PSK not using (EC)DHE
-I          Update keys and IVs before sending data
-y          Key Share with FFDHE named groups only
```

```
-Y          Key Share with ECC named groups only
-1 <num>    Display a result by specified language.
            0: English, 1: Japanese
-2          Disable DH Prime check
-6          Simulate WANT_WRITE errors on every other IO send
-7          Set minimum downgrade protocol version [0-4]  SSLv3(0) - TLS1.3(4)
```

To test against example.com:443 try the following. This is using wolfSSL compiled with the `--enable-opensslextra` and `--enable-supportedcurves` build options:

```
./examples/client/client -h example.com -p 443 -d -g
```

Which returns:

```
Alternate cert chain used
 issuer : /C=US/O=DigiCert Inc/CN=DigiCert TLS RSA SHA256 2020 CA1
 subject: /C=US/ST=California/L=Los Angeles/O=Internet Corporation for Assigned Names and Numbers
     /CN=www.example.org
 altname = www.example.net
 altname = www.example.edu
 altname = www.example.com
 altname = example.org
 altname = example.net
 altname = example.edu
 altname = example.com
 altname = www.example.org
 serial number:0f:be:08:b0:85:4d:05:73:8a:b0:cc:e1:c9:af:ee:c9
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
SSL curve name is SECP256R1
Session timeout set to 500 seconds
Client Random : 20640B8131D8E542646D395B362354F9308057B1624C2442C0B5FCDD064BFE29
SSL connect ok, sending GET...
HTTP/1.0 200 OK
Accept-Ranges: bytes
Content-Type: text/html
Date: Thu, 14 Oct 2021 16:50:28 GMT
Last-Modified: Thu, 14 Oct 2021 16:45:10 GMT
Server: ECS (nyb/1D10)
Content-Length: 94
Connection: close
```

This tells the client to connect to (`-h`) example.com on the HTTPS port (`-p`) of 443 and sends a generic (`-g`) GET request. The (`-d`) option tells the client not to verify the server. The rest is the initial output from the server that fits into the read buffer.

If no command line arguments are given, then the client attempts to connect to the localhost on the wolfSSL default port of 11111. It also loads the client certificate in case the server wants to perform client authentication.

The client is able to benchmark a connection when using the `-b <num>` argument. When used, the client attempts to connect to the specified server/port the argument number of times and gives the average time in milliseconds that it took to perform `SSL_connect()`. For example:

```
/examples/client/client -b 100 -h example.com -p 443 -d
```

Returns:

```
wolfSSL_connect avg took:  296.417 milliseconds
```

If you'd like to change the default host from localhost, or the default port from 11111, you can change these settings in `/wolfssl/test.h`. The variables wolfSSLIP and wolfSSLPort control these settings. Re-build all of the examples including testsuite when changing these settings otherwise the test programs won't be able to connect to each other.

By default, the wolfSSL example client tries to connect to the specified server using TLS 1.2. The user is able to change the SSL/TLS version which the client uses by using the `-v` command line option. The following values are available for this option:

- `-v 0` - SSL 3.0 (disabled by default)
- `-v 1` - TLS 1.0
- `-v 2` - TLS 1.1
- `-v 3` - TLS 1.2 (selected by default)
- `-v 4` - TLS 1.3

A common error users see when using the example client is -188:

```
wolfSSL_connect error -188, ASN no signer error to confirm failure
wolfSSL error: wolfSSL_connect failed
```

This is typically caused by the wolfSSL client not being able to verify the certificate of the server it is connecting to. By default, the wolfSSL client loads the yaSSL test CA certificate as a trusted root certificate. This test CA certificate will not be able to verify an external server certificate which was signed by a different CA. As such, to solve this problem, users either need to turn off verification of the peer (server), using the `-d` option:

```
./examples/client/client -h myhost.com -p 443 -d
```

Or load the correct CA certificate into the wolfSSL client using the `-A` command line option:

```
./examples/client/client -h myhost.com -p 443 -A serverCA.pem
```

3.4 Server Example

The server example demonstrates a simple SSL server that optionally performs client authentication. Only one client connection is accepted and then the server quits. The client example in normal mode (no command line arguments) will work just fine against the example server, but if you specify command line arguments for the client example, then a client certificate isn't loaded and the `wolfSSL_connect()` will fail (unless client cert check is disabled using the `-d` option). The server will report an error "-245, peer didn't send cert". Like the example client, the server can be used with several command line arguments as well:

```
./examples/server/server --help
```

Which returns:

```
server 4.8.1 NOTE: All files relative to wolfSSL home dir
-? <num>    Help, print this usage
            0: English, 1: Japanese
--help      Help, in English
-p <num>    Port to listen on, not 0, default 11111
-v <num>    SSL version [0-4], SSLv3(0) - TLS1.3(4)), default 3
-l <str>    Cipher suite list (: delimited)
-c <file>   Certificate file,           default ./certs/server-cert.pem
-k <file>   Key file,                   default ./certs/server-key.pem
-A <file>   Certificate Authority file, default ./certs/client-cert.pem
-R <file>   Create Ready file for external monitor default none
-D <file>   Diffie-Hellman Params file, default ./certs/dh2048.pem
-Z <num>    Minimum DH key bits,        default 1024
-d          Disable client cert check
-b          Bind to any interface instead of localhost only
-s          Use pre Shared keys
-u          Use UDP DTLS, add -v 2 for DTLSv1, -v 3 for DTLSv1.2 (default)
```

```
-f          Fewer packets/group messages
-r          Allow one client Resumption
-N          Use Non-blocking sockets
-S <str>    Use Host Name Indication
-w          Wait for bidirectional shutdown
-x          Print server errors but do not close connection
-i          Loop indefinitely (allow repeated connections)
-e          Echo data mode (return raw bytes received)
-B <num>    Benchmark throughput using <num> bytes and print stats
-g          Return basic HTML web page
-C <num>    The number of connections to accept, default: 1
-H <arg>    Internal tests [defCipherList, exitWithRet, verifyFail, useSupCurve,
                        loadSSL, disallowETM]
-U          Update keys and IVs before sending
-K          Key Exchange for PSK not using (EC)DHE
-y          Pre-generate Key Share using FFDHE_2048 only
-Y          Pre-generate Key Share using P-256 only
-F          Send alert if no mutual authentication
-2          Disable DH Prime check
-1 <num>    Display a result by specified language.
            0: English, 1: Japanese
-6          Simulate WANT_WRITE errors on every other IO send
-7          Set minimum downgrade protocol version [0-4]  SSLv3(0) - TLS1.3(4)
```

3.5 EchoServer Example

The echoserver example sits in an endless loop waiting for an unlimited number of client connections. Whatever the client sends the echoserver echoes back. Client authentication isn't performed so the example client can be used against the echoserver in all 3 modes. Four special commands aren't echoed back and instruct the echoserver to take a different action.

1. `quit` - If the echoserver receives the string "quit" it will shutdown.
2. `break` If the echoserver receives the string "break" it will stop the current session but continue handling requests. This is particularly useful for DTLS testing.
3. `printstats` - If the echoserver receives the string "printstats" it will print out statistics for the session cache.
4. `GET` - If the echoserver receives the string "GET" it will handle it as an http get and send back a simple page with the message "greeting from wolfSSL". This allows testing of various TLS/SSL clients like Safari, IE, Firefox, gnutls, and the like against the echoserver example.

The output of the echoserver is echoed to stdout unless `NO_MAIN_DRIVER` is defined. You can redirect output through the shell or through the first command line argument. To create a file named output.txt with the output from the echoserver run:

```
./examples/echoserver/echoserver output.txt
```

3.6 EchoClient Example

The echoclient example can be run in interactive mode or batch mode with files. To run in interactive mode and write 3 strings "hello", "wolfssl", and "quit" results in:

```
./examples/echoclient/echoclient
hello
hello
wolfssl
wolfssl
quit
sending server shutdown command: quit!
```

To use an input file, specify the filename on the command line as the first argument. To echo the contents of the file `input.txt` issue:

```
./examples/echoclient/echoclient input.txt
```

If you want the result to be written out to a file, you can specify the output file name as an additional command line argument. The following command will echo the contents of file `input.txt` and write the result from the server to `output.txt`:

```
./examples/echoclient/echoclient input.txt output.txt
```

The testsuite program does just that, but hashes the input and output files to make sure that the client and server were getting/sending the correct and expected results.

3.7 Benchmark

Many users are curious about how the wolfSSL embedded SSL library will perform on a specific hardware device or in a specific environment. Because of the wide variety of different platforms and compilers used today in embedded, enterprise, and cloud-based environments, it is hard to give generic performance calculations across the board.

To help wolfSSL users and customers in determining SSL performance for wolfSSL / wolfCrypt, a benchmark application is provided which is bundled with wolfSSL. wolfSSL uses the wolfCrypt cryptography library for all crypto operations by default. Because the underlying crypto is a very performance-critical aspect of SSL/TLS, our benchmark application runs performance tests on wolfCrypt's algorithms.

The benchmark utility located in wolfcrypt/benchmark (`./wolfcrypt/benchmark/benchmark`) may be used to benchmark the cryptographic functionality of wolfCrypt. Typical output may look like the following (in this output, several optional algorithms/ciphers were enabled including HC-128, RABBIT, ECC, SHA-256, SHA-512, AES-GCM, AES-CCM, and Camellia):

```
./wolfcrypt/benchmark/benchmark
```

```
------------------------------------------------------------------------------
 wolfSSL version 4.8.1
------------------------------------------------------------------------------
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
RNG                105 MB took 1.004 seconds,  104.576 MB/s Cycles per byte =  20.94
AES-128-CBC-enc    310 MB took 1.008 seconds,  307.434 MB/s Cycles per byte =   7.12
AES-128-CBC-dec    290 MB took 1.002 seconds,  289.461 MB/s Cycles per byte =   7.56
AES-192-CBC-enc    265 MB took 1.010 seconds,  262.272 MB/s Cycles per byte =   8.35
AES-192-CBC-dec    240 MB took 1.013 seconds,  236.844 MB/s Cycles per byte =   9.24
AES-256-CBC-enc    240 MB took 1.011 seconds,  237.340 MB/s Cycles per byte =   9.22
AES-256-CBC-dec    235 MB took 1.018 seconds,  230.864 MB/s Cycles per byte =   9.48
AES-128-GCM-enc    160 MB took 1.011 seconds,  158.253 MB/s Cycles per byte =  13.83
AES-128-GCM-dec    160 MB took 1.016 seconds,  157.508 MB/s Cycles per byte =  13.90
AES-192-GCM-enc    150 MB took 1.022 seconds,  146.815 MB/s Cycles per byte =  14.91
AES-192-GCM-dec    150 MB took 1.039 seconds,  144.419 MB/s Cycles per byte =  15.16
AES-256-GCM-enc    130 MB took 1.017 seconds,  127.889 MB/s Cycles per byte =  17.12
AES-256-GCM-dec    140 MB took 1.030 seconds,  135.943 MB/s Cycles per byte =  16.10
GMAC Table 4-bit   321 MB took 1.002 seconds,  320.457 MB/s Cycles per byte =   6.83
CHACHA             420 MB took 1.002 seconds,  419.252 MB/s Cycles per byte =   5.22
CHA-POLY           330 MB took 1.013 seconds,  325.735 MB/s Cycles per byte =   6.72
MD5                655 MB took 1.007 seconds,  650.701 MB/s Cycles per byte =   3.36
POLY1305          1490 MB took 1.002 seconds, 1486.840 MB/s Cycles per byte =   1.47
SHA                560 MB took 1.004 seconds,  557.620 MB/s Cycles per byte =   3.93
SHA-224            240 MB took 1.011 seconds,  237.474 MB/s Cycles per byte =   9.22
SHA-256            250 MB took 1.020 seconds,  245.081 MB/s Cycles per byte =   8.93
SHA-384            380 MB took 1.005 seconds,  377.963 MB/s Cycles per byte =   5.79
SHA-512            380 MB took 1.007 seconds,  377.260 MB/s Cycles per byte =   5.80
SHA3-224           385 MB took 1.009 seconds,  381.679 MB/s Cycles per byte =   5.74
```

```
SHA3-256          360 MB took 1.004 seconds,  358.583 MB/s Cycles per byte =   6.11
SHA3-384          270 MB took 1.020 seconds,  264.606 MB/s Cycles per byte =   8.27
SHA3-512          185 MB took 1.019 seconds,  181.573 MB/s Cycles per byte =  12.06
HMAC-MD5          665 MB took 1.004 seconds,  662.154 MB/s Cycles per byte =   3.31
HMAC-SHA          590 MB took 1.004 seconds,  587.535 MB/s Cycles per byte =   3.73
HMAC-SHA224       240 MB took 1.018 seconds,  235.850 MB/s Cycles per byte =   9.28
HMAC-SHA256       245 MB took 1.013 seconds,  241.805 MB/s Cycles per byte =   9.05
HMAC-SHA384       365 MB took 1.006 seconds,  362.678 MB/s Cycles per byte =   6.04
HMAC-SHA512       365 MB took 1.009 seconds,  361.674 MB/s Cycles per byte =   6.05
PBKDF2             30 KB took 1.000 seconds,   29.956 KB/s Cycles per byte = 74838.56
RSA     2048 public      18400 ops took 1.004 sec, avg 0.055 ms, 18335.019 ops/sec
RSA     2048 private       300 ops took 1.215 sec, avg 4.050 ms, 246.891 ops/sec
DH      2048 key gen      1746 ops took 1.000 sec, avg 0.573 ms, 1745.991 ops/sec
DH      2048 agree         900 ops took 1.060 sec, avg 1.178 ms, 849.210 ops/sec
ECC   [      SECP256R1]   256 key gen       901 ops took 1.000 sec, avg 1.110 ms, 900.779 ops/sec
ECDHE [      SECP256R1]   256 agree        1000 ops took 1.105 sec, avg 1.105 ms, 904.767 ops/sec
ECDSA [      SECP256R1]   256 sign          900 ops took 1.022 sec, avg 1.135 ms, 880.674 ops/sec
ECDSA [      SECP256R1]   256 verify       1300 ops took 1.012 sec, avg 0.779 ms, 1284.509 ops/
    sec
Benchmark complete
```

This is especially useful for comparing the public key speed before and after changing the math library. You can test the results using the normal math library (`./configure`), the fastmath library (`./configure --enable-fastmath`), and the fasthugemath library (`./configure --enable-fasthugemath`).

For more details and benchmark results, please refer to the wolfSSL Benchmarks page: https://www.wolfssl.com/docs/benchmarks

3.7.1 Relative Performance

Although the performance of individual ciphers and algorithms will depend on the host platform, the following graph shows relative performance between wolfCrypt's ciphers. These tests were conducted on a Macbook Pro (OS X 10.6.8) running a 2.2 GHz Intel Core i7.

If you want to use only a subset of ciphers, you can customize which specific cipher suites and/or ciphers wolfSSL uses when making an SSL/TLS connection. For example, to force 128-bit AES, add the following line after the call to `wolfSSL_CTX_new(SSL_CTX_new)`:

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

3.7.2 Benchmarking Notes

1. The processors native register size (32 vs 64-bit) can make a big difference when doing 1000+ bit public key operations.
2. **keygen** (`--enable-keygen`) will allow you to also benchmark key generation speeds when running the benchmark utility.
3. **fastmath** (`--enable-fastmath`) reduces dynamic memory usage and speeds up public key operations. If you are having trouble building on 32-bit platform with fastmath, disable shared libraries so that PIC isn't hogging a register (also see notes in the README):

   ```
   ./configure --enable-fastmath --disable-shared
   make clean
   make
   ```

   Note: doing a `make clean` is good practice with wolfSSL when switching configure options.
4. By default, fastmath tries to use assembly optimizations if possible. If assembly optimizations don't work, you can still use fastmath without them by adding `TFM_NO_ASM` to `CFLAGS` when building wolfSSL:

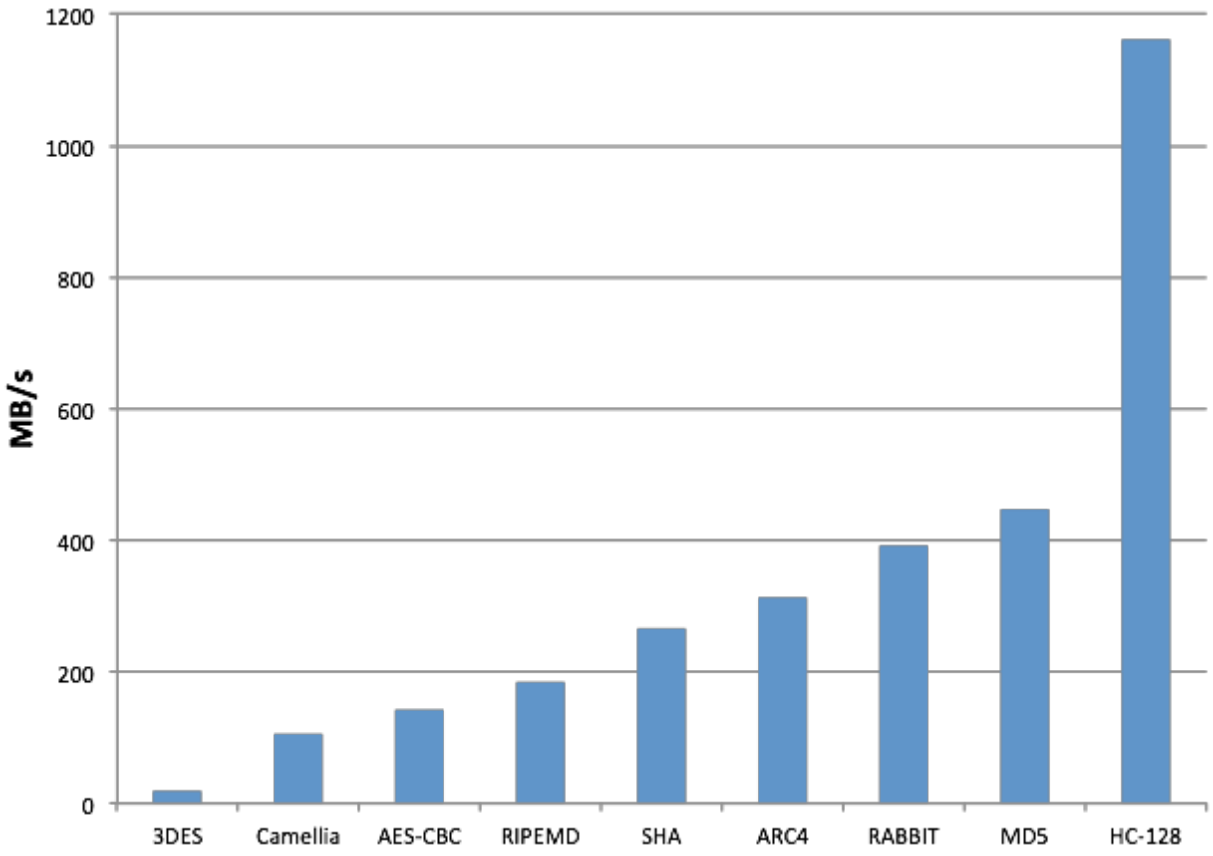

Figure 1: Benchmark

```
./configure --enable-fastmath C_EXTRA_FLAGS="-DTFM_NO_ASM"
```

5. Using fasthugemath can try to push fastmath even more for users who are not running on embedded platforms:

   ```
   ./configure --enable-fasthugemath
   ```

6. With the default wolfSSL build, we have tried to find a good balance between memory usage and performance. If you are more concerned about one of the two, please refer back to Build Options for additional wolfSSL configuration options.

7. **Bulk Transfers**: wolfSSL by default uses 128 byte I/O buffers since about 80% of SSL traffic falls within this size and to limit dynamic memory use. It can be configured to use 16K buffers (the maximum SSL size) if bulk transfers are required.

3.7.3 Benchmarking on Embedded Systems

There are several build options available to make building the benchmark application on an embedded system easier. These include:

3.7.3.1 BENCH_EMBEDDED Enabling this define will switch the benchmark application from using Megabytes to using Kilobytes, therefore reducing the memory usage. By default, when using this define, ciphers and algorithms will be benchmarked with 25kB. Public key algorithms will only be benchmarked over 1 iteration (as public key operations on some embedded processors can be fairly slow). These can be adjusted in `benchmark.c` by altering the variables `numBlocks` and `times` located inside the `BENCH_EMBEDDED` define.

3.7.3.2 USE_CERT_BUFFERS_1024 Enabling this define will switch the benchmark application from loading test keys and certificates from the file system and instead use 1024-bit key and certificate buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. It is useful to use this define when an embedded platform has no filesystem (used with `NO_FILESYSTEM`) and a slow processor where 2048-bit public key operations may not be reasonable.

3.7.3.3 USE_CERT_BUFFERS_2048 Enabling this define is similar to `USE_CERT_BUFFERS_1024` accept that 2048-bit key and certificate buffers are used instead of 1024-bit ones. This define is useful when the processor is fast enough to do 2048-bit public key operations but when there is no filesystem available to load keys and certificates from files.

3.8 Changing a Client Application to Use wolfSSL

This section will explain the basic steps needed to add wolfSSL to a client application, using the wolfSSL native API. For a server explanation, please see Changing a Server Application to Use wolfSSL. A more complete walk-through with example code is located in the SSL Tutorial in Chapter 11. If you want more information about the OpenSSL compatibility layer, please see OpenSSL Compatibility.

1. Include the wolfSSL header:

   ```
   #include <wolfssl/ssl.h>
   ```

2. Initialize wolfSSL and the `WOLFSSL_CTX`. You can use one `WOLFSSL_CTX` no matter how many WOLFSSL objects you end up creating. Basically you'll just need to load CA certificates to verify the server you are connecting to. Basic initialization looks like:

   ```
   wolfSSL_Init();
   WOLFSSL_CTX* ctx;
   if ((ctx = wolfSSL_CTX_new(wolfTLSv1_client_method())) == NULL)
   {
       fprintf(stderr, "wolfSSL_CTX_new error.\n");
       exit(EXIT_FAILURE);
   }
   if (wolfSSL_CTX_load_verify_locations(ctx,"./ca-cert.pem",0) != SSL_SUCCESS) {
       fprintf(stderr, "Error loading ./ca-cert.pem,"
                       " please check the file.\n");
   ```

```
    exit(EXIT_FAILURE);
}
```

3. Create the WOLFSSL object after each TCP connect and associate the file descriptor with the session:

```
/*after connecting to socket fd*/
WOLF SSL* ssl;
if ((ssl = wolfSSL_new(ctx)) == NULL) {
    fprintf(stderr, "wolfSSL_new error.\n");
    exit(EXIT_FAILURE);
}
wolfSSL_set_fd(ssl, fd);
```

4. Change all calls from `read()` (or `recv()`) to `wolfSSL_read()` so:

   ```
   result = read(fd, buffer, bytes);
   ```

   becomes:

   ```
   result = wolfSSL_read(ssl, buffer, bytes);
   ```

5. Change all calls from `write()` (or `send()`) to `wolfSSL_write()` so:

   ```
   result = write(fd, buffer, bytes);
   ```

   becomes

   ```
   result = wolfSSL_write(ssl, buffer, bytes);
   ```

6. You can manually call `wolfSSL_connect()` but that's not even necessary; the first call to `wolfSSL_read()` or `wolfSSL_write()` will initiate the `wolfSSL_connect()` if it hasn't taken place yet.

7. Error checking. Each `wolfSSL_read()` and `wolfSSL_write()` call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like `read()` and `write()`. In the event of an error you can use two calls to get more information about the error:

   ```
   char errorString[80];
   int err = wolfSSL_get_error(ssl, 0);
   wolfSSL_ERR_error_string(err, errorString);
   ```

   If you are using non-blocking sockets, you can test for errno `EAGAIN/EWOULDBLOCK` or more correctly you can test the specific error code returned by `wolfSSL_get_error()` for `SSL_ERROR_WANT_READ` or`SSL_ERROR_WANT_WRITE`.

8. Cleanup. After each WOLFSSL object is done being used you can free it up by calling:

   ```
   wolfSSL_free(ssl);
   ```

   When you are completely done using SSL/TLS altogether you can free the `WOLFSSL_CTX` object by calling:

   ```
   wolfSSL_CTX_free(ctx);
   wolfSSL_Cleanup();
   ```

   For an example of a client application using wolfSSL, see the client example located in the `<wolfssl_root>/examples/client.c` file.

3.9 Changing a Server Application to Use wolfSSL

This section will explain the basic steps needed to add wolfSSL to a server application using the wolfSSL native API. For a client explanation, please see Changing a Client Application to Use wolfSSL. A more complete walk-through, with example code, is located in the SSL Tutorial chapter.

1. Follow the instructions above for a client, except change the client method call in step 5 to a server one, so:

   ```
   wolfSSL_CTX_new(wolfTLSv1_client_method());
   ```

becomes:

```
wolfSSL_CTX_new(wolfTLSv1_server_method());
```

or even:

```
wolfSSL_CTX_new(wolfSSLv23_server_method());
```

To allow SSLv3 and TLSv1+ clients to connect to the server.

2. Add the server's certificate and key file to the initialization in step 5 above:

```
if (wolfSSL_CTX_use_certificate_file(ctx,"./server-cert.pem", SSL_FILETYPE_PEM) !=
 ↪  SSL_SUCCESS) {
    fprintf(stderr, "Error loading ./server-cert.pem,"
                    " please check the file.\n");
    exit(EXIT_FAILURE);
}
if (wolfSSL_CTX_use_PrivateKey_file(ctx,"./server-key.pem", SSL_FILETYPE_PEM) !=
 ↪  SSL_SUCCESS) {
    fprintf(stderr, "Error loading ./server-key.pem,"
                    " please check the file.\n");
    exit(EXIT_FAILURE);
}
```

It is possible to load certificates and keys from buffers as well if there is no filesystem available. In this case, see the wolfSSL_CTX_use_certificate_buffer() and wolfSSL_CTX_use_PrivateKey_buffer() API documentation, linked here, for more information.

For an example of a server application using wolfSSL, see the server example located in the <wolfssl_root>/examples/server.c file.

4 Features

wolfSSL (formerly CyaSSL) supports the C programming language as a primary interface, but also supports several other host languages, including Java, PHP, Perl, and Python (through a SWIG interface). If you have interest in hosting wolfSSL in another programming language that is not currently supported, please contact us.

This chapter covers some of the features of wolfSSL in more depth, including Stream Ciphers, AES-NI, IPv6 support, SSL Inspection (Sniffer) support, and more.

4.1 Features Overview

For an overview of wolfSSL features, please reference the wolfSSL product webpage: https://www.wolfssl.com/products/wolfssl

4.2 Protocol Support

wolfSSL supports **SSL 3.0**, **TLS** (**1.0**, **1.1**, **1.2, 1.3**), and **DTLS** (**1.0** and **1.2**). You can easily select a protocol to use by using one of the following functions (as shown for either the client or server). wolfSSL does not support SSL 2.0, as it has been insecure for several years. The client and server functions below change slightly when using the OpenSSL compatibility layer. For the OpenSSL-compatible functions, please see OpenSSL Compatibility.

4.2.1 Server Functions

- `wolfDTLSv1_server_method()` - DTLS 1.0
- `wolfDTLSv1_2_server_method()` - DTLS 1.2
- `wolfSSLv3_server_method()` - SSL 3.0
- `wolfTLSv1_server_method()` - TLS 1.0
- `wolfTLSv1_1_server_method()` - TLS 1.1
- `wolfTLSv1_2_server_method()` - TLS 1.2
- `wolfTLSv1_3_server_method()` - TLS 1.3
- `wolfSSLv23_server_method()` - Use highest possible version from SSLv3 - TLS 1.2

wolfSSL supports robust server downgrade with the `wolfSSLv23_server_method()` function. See Robust Client and Server Downgrade for a details.

4.2.2 Client Functions

- `wolfDTLSv1_client_method()` - DTLS 1.0
- `wolfDTLSv1_2_client_method_ex()` - DTLS 1.2
- `wolfSSLv3_client_method()` - SSL 3.0
- `wolfTLSv1_client_method()` - TLS 1.0
- `wolfTLSv1_1_client_method()` - TLS 1.1
- `wolfTLSv1_2_client_method()` - TLS 1.2
- `wolfTLSv1_3_client_method()` - TLS 1.3
- `wolfSSLv23_client_method()` - Use highest possible version from SSLv3 - TLS 1.2

wolfSSL supports robust client downgrade with the `wolfSSLv23_client_method()` function. See Robust Client and Server Downgrade for a details.

For details on how to use these functions, please see the Getting Started chapter. For a comparison between SSL 3.0, TLS 1.0, 1.1, 1.2, and DTLS, please see Appendix A.

4.2.3 Robust Client and Server Downgrade

Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLS 1.0 and tries to connect to an SSL 3.0 only server, the connection will fail, likewise connecting to a TLS 1.1 will fail as well.

To resolve this issue, a client that uses the `wolfSSLv23_client_method()` function will support the highest protocol version supported by the server by downgrading if necessary. In this case, the client will be able to connect to a server running TLS 1.0 - TLS 1.3 (or a subset or superset that includes SSL 3.0 depending on which protocol versions are configured in wolfSSL). The only versions it can't connect to is SSL 2.0 which has been insecure for years, and SSL 3.0 which has been disabled by default.

Similarly, a server using the `wolfSSLv23_server_method()` function can handle clients supporting protocol versions from TLS 1.0 - TLS 1.2. A wolfSSL server can't accept a connection from SSLv2 because no security is provided.

4.2.4 IPv6 Support

If you are an adopter of IPv6 and want to use an embedded SSL implementation then you may have been wondering if wolfSSL supports IPv6. The answer is yes, we do support wolfSSL running on top of IPv6.

wolfSSL was designed as IP neutral, and will work with both IPv4 and IPv6, but the current test applications default to IPv4 (so as to apply to a broader range of systems). To change the test applications to IPv6, use the **–enable-ipv6** option while building wolfSSL.

Further information on IPv6 can be found here:

https://en.wikipedia.org/wiki/IPv6.

4.2.5 DTLS

wolfSSL has support for DTLS ("Datagram" TLS) for both client and server. The current supported version is DTLS 1.0.

The TLS protocol was designed to provide a secure transport channel across a **reliable** medium (such as TCP). As application layer protocols began to be developed using UDP transport (such as SIP and various electronic gaming protocols), a need arose for a way to provide communications security for applications which are delay sensitive. This need lead to the creation of the DTLS protocol.

Many people believe the difference between TLS and DTLS is the same as TCP vs. UDP. This is incorrect. UDP has the benefit of having no handshake, no tear-down, and no delay in the middle if something gets lost (compared with TCP). DTLS on the other hand, has an extended SSL handshake and tear-down and must implement TCP-like behavior for the handshake. In essence, DTLS reverses the benefits that are offered by UDP in exchange for a secure connection.

DTLS can be enabled when building wolfSSL by using the `--enable-dtls` build option.

4.2.6 LwIP (Lightweight Internet Protocol)

wolfSSL supports the lightweight internet protocol implementation out of the box. To use this protocol all you need to do is define `WOLFSSL_LWIP` or navigate to the `settings.h` file and uncomment the line:

```
/*#define WOLFSSL_LWIP*/
```

The focus of lwIP is to reduce RAM usage while still providing a full TCP stack. That focus makes lwIP great for use in embedded systems, an area where wolfSSL is an ideal match for SSL/TLS needs.

4.2.7 TLS Extensions

A list of TLS extensions supported by wolfSSL and note of which RFC can be referenced for the given extension.

RFC	Extension	wolfSSL Type
6066	Server Name Indication	`TLSX_SERVER_NAME`
6066	Maximum Fragment Length Negotiation	`TLSX_MAX_FRAGMENT_LENGTH`
6066	Truncated HMAC	`TLSX_TRUNCATED_HMAC`
6066	Status Request	`TLSX_STATUS_REQUEST`
7919	Supported Groups	`TLSX_SUPPORTED_GROUPS`
5246	Signature Algorithm	`TLSX_SIGNATURE_ALGORITHMS`
7301	Application Layer Protocol Negotiation	`TLSX_APPLICATION_LAYER_PROTOCOL`

RFC	Extension	wolfSSL Type
6961	Multiple Certificate Status Request	TLSX_STATUS_REQUEST_V2
Draft	Quantum-Safe Hybrid Key Exchange	TLSX_QUANTUM_SAFE_HYBRID
5077	Session Ticket	TLSX_SESSION_TICKET
5746	Renegotiation Indication	TLSX_RENEGOTIATION_INFO
8446	Key Share	TLSX_KEY_SHARE
8446	Pre Shared Key	TLSX_PRE_SHARED_KEY
8446	PSK Key Exchange Modes	TLSX_PSK_KEY_EXCHANGE_MODES
8446	Early Data	TLSX_EARLY_DATA
8446	Cookie	TLSX_COOKIE
8446	Supported Versions	TLSX_SUPPORTED_VERSIONS
8446	Post Handshake Authorization	TLSX_POST_HANDSHAKE_AUTH

4.3 Cipher Support

4.3.1 Cipher Suite Strength and Choosing Proper Key Sizes

To see what ciphers are currently being used you can call the method: `wolfSSL_get_ciphers()`.

This function will return the currently enabled cipher suites.

Cipher suites come in a variety of strengths. Because they are made up of several different types of algorithms (authentication, encryption, and message authentication code (MAC)), the strength of each varies with the chosen key sizes.

There can be many methods of grading the strength of a cipher suite - the specific method used seems to vary between different projects and companies and can include things such as symmetric and public key algorithm key sizes, type of algorithm, performance, and known weaknesses.

NIST (National Institute of Standards and Technology) makes recommendations on choosing an acceptable cipher suite by providing comparable algorithm strengths for varying key sizes of each. The strength of a cryptographic algorithm depends on the algorithm and the key size used. The NIST Special Publication, SP800-57, states that two algorithms are considered to be of comparable strength as follows:

> Two algorithms are considered to be of comparable strength for the given key sizes (X and Y) if the amount of work needed to "break the algorithms" or determine the keys (with the given key sizes) is approximately the same using a given resource. The security strength of an algorithm for a given key size is traditionally described in terms of the amount of work it takes to try all keys for a symmetric algorithm with a key size of "X" that has no shortcut attacks (i.e., the most efficient attack is to try all possible keys).

The following two tables are based off of both Table 2 (pg. 56) and Table 4 (pg. 59) from NIST SP800-57, and shows comparable security strength between algorithms as well as a strength measurement (based off of NIST's suggested algorithm security lifetimes using bits of security).

Note: In the following table "L" is the size of the public key for finite field cryptography (FFC), "N" is the size of the private key for FFC, "k" is considered the key size for integer factorization cryptography (IFC), and "f" is considered the key size for elliptic curve cryptography.

Bits of Security	Symmetric Key Algorithms	**FFC** Key Size (DSA, DH, etc.)	**IFC** Key Size (RSA, etc.)	**ECC** Key Size (ECDSA, etc.)	Description
80	2TDEA, etc.	L = 1024, N = 160	k = 1024	f = 160-223	Security good through 2010
128	AES-128, etc.	L = 3072, N = 256	k = 3072	f = 256-383	Security good through 2030

Bits of Security	Symmetric Key Algorithms	**FFC** Key Size (DSA, DH, etc.)	**IFC** Key Size (RSA, etc.)	**ECC** Key Size (ECDSA, etc.)	Description
192	AES-192, etc.	L = 7680, N = 384	k = 7680	f = 384-511	Long Term Protection
256	AES-256, etc.	L = 15360, N = 512	k = 15360	f = 512+	Secure for the fore-see-able future

Using this table as a guide, to begin to classify a cipher suite, we categorize it based on the strength of the symmetric encryption algorithm. In doing this, a rough grade classification can be devised to classify each cipher suite based on bits of security (only taking into account symmetric key size):

- **LOW** - bits of security smaller than 128 bits
- **MEDIUM** - bits of security equal to 128 bits
- **HIGH** - bits of security larger than 128 bits

Outside of the symmetric encryption algorithm strength, the strength of a cipher suite will depend greatly on the key sizes of the key exchange and authentication algorithm keys. The strength is only as good as the cipher suite's weakest link.

Following the above grading methodology (and only basing it on symmetric encryption algorithm strength), wolfSSL 2.0.0 currently supports a total of 0 LOW strength cipher suites, 12 MEDIUM strength cipher suites, and 8 HIGH strength cipher suites – as listed below. The following strength classification could change depending on the chosen key sizes of the other algorithms involved. For a reference on hash function security strength, see Table 3 (pg. 56) of NIST SP800-57.

In some cases, you will see ciphers referenced as "**EXPORT**" ciphers. These ciphers originated from the time period in US history (as late as 1992) when it was illegal to export software with strong encryption from the United States. Strong encryption was classified as "Munitions" by the US Government (under the same category as Nuclear Weapons, Tanks, and Ballistic Missiles). Because of this restriction, software being exported included "weakened" ciphers (mostly in smaller key sizes). In the current day, this restriction has been lifted, and as such, EXPORT ciphers are no longer a mandated necessity.

4.3.2 Supported Cipher Suites

The following cipher suites are supported by wolfSSL. A cipher suite is a combination of authentication, encryption, and message authentication code (MAC) algorithms which are used during the TLS or SSL handshake to negotiate security settings for a connection.

Each cipher suite defines a key exchange algorithm, a bulk encryption algorithm, and a message authentication code algorithm (MAC). The **key exchange algorithm** (RSA, DSS, DH, EDH) determines how the client and server will authenticate during the handshake process. The **bulk encryption algorithm** (DES, 3DES, AES, ARC4, RABBIT, HC-128), including block ciphers and stream ciphers, is used to encrypt the message stream. The **message authentication code (MAC) algorithm** (MD2, MD5, SHA-1, SHA-256, SHA-512, RIPEMD) is a hash function used to create the message digest.

The table below matches up to the cipher suites (and categories) found in `<wolfssl_root>/wolfssl/internal.h` (starting at about line 706). If you are looking for a cipher suite which is not in the following list, please contact us to discuss getting it added to wolfSSL.

ECC cipher suites:

- `TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA`
- `TLS_DHE_RSA_WITH_AES_256_CBC_SHA`
- `TLS_DHE_RSA_WITH_AES_128_CBC_SHA`
- `TLS_DH_anon_WITH_AES_128_CBC_SHA`
- `TLS_RSA_WITH_AES_256_CBC_SHA`

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_NULL_SHA
- TLS_PSK_WITH_AES_256_CBC_SHA
- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_PSK_WITH_AES_256_CBC_SHA384
- TLS_PSK_WITH_AES_128_CBC_SHA
- TLS_PSK_WITH_NULL_SHA256
- TLS_PSK_WITH_NULL_SHA384
- TLS_PSK_WITH_NULL_SHA
- SSL_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_IDEA_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_RC4_128_SHA
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_PSK_WITH_NULL_SHA256
- TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_NULL_SHA

Static ECC cipher suites:

- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_RSA_WITH_RC4_128_SHA
- TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384

wolfSSL extension - eSTREAM cipher suites:

- TLS_RSA_WITH_HC_128_MD5
- TLS_RSA_WITH_HC_128_SHA
- TLS_RSA_WITH_RABBIT_SHA

Blake2b cipher suites:

- TLS_RSA_WITH_AES_128_CBC_B2B256
- TLS_RSA_WITH_AES_256_CBC_B2B256
- TLS_RSA_WITH_HC_128_B2B256

wolfSSL extension - Quantum-Safe Handshake:

- TLS_QSH

wolfSSL extension - NTRU cipher suites:

- TLS_NTRU_RSA_WITH_RC4_128_SHA
- TLS_NTRU_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_NTRU_RSA_WITH_AES_128_CBC_SHA
- TLS_NTRU_RSA_WITH_AES_256_CBC_SHA

SHA-256 cipher suites:

- TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_NULL_SHA256
- TLS_DHE_PSK_WITH_AES_128_CBC_SHA256
- TLS_DHE_PSK_WITH_NULL_SHA256

SHA-384 cipher suites:

- TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
- TLS_DHE_PSK_WITH_NULL_SHA384

AES-GCM cipher suites:

- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_PSK_WITH_AES_128_GCM_SHA256
- TLS_PSK_WITH_AES_256_GCM_SHA384
- TLS_DHE_PSK_WITH_AES_128_GCM_SHA256
- TLS_DHE_PSK_WITH_AES_256_GCM_SHA384

ECC AES-GCM cipher suites:

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384

AES-CCM cipher suites:

- TLS_RSA_WITH_AES_128_CCM_8
- TLS_RSA_WITH_AES_256_CCM_8
- TLS_ECDHE_ECDSA_WITH_AES_128_CCM
- TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8
- TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8
- TLS_PSK_WITH_AES_128_CCM
- TLS_PSK_WITH_AES_256_CCM
- TLS_PSK_WITH_AES_128_CCM_8
- TLS_PSK_WITH_AES_256_CCM_8
- TLS_DHE_PSK_WITH_AES_128_CCM
- TLS_DHE_PSK_WITH_AES_256_CCM

Camellia cipher suites:

- TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
- TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256

ChaCha cipher suites:

- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_RSA_WITH_CHACHA20_OLD_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_OLD_POLY1305_SHA256
- TLS_DHE_RSA_WITH_CHACHA20_OLD_POLY1305_SHA256

Renegotiation Indication Extension Special Suite:

- TLS_EMPTY_RENEGOTIATION_INFO_SCSV

4.3.3 AEAD Suites

wolfSSL supports AEAD suites, including AES-GCM, AES-CCM, and CHACHA-POLY1305. The big difference between these AEAD suites and others is that they authenticate the encrypted data with any additional cleartext data. This helps with mitigating man in the middle attacks that result in having data tampered with. AEAD suites use a combination of a block cipher (or more recently also a stream cipher) algorithm combined with a tag produced by a keyed hash algorithm. Combining these two algorithms is handled by the wolfSSL encrypt and decrypt process which makes it easier for users. All that is needed for using a specific AEAD suite is simply enabling the algorithms that are used in a supported suite.

4.3.4 Block and Stream Ciphers

wolfSSL supports the **AES**, **DES**, **3DES**, and **Camellia** block ciphers and the **RC4**, **RABBIT**, **HC-128** and **CHACHA20** stream ciphers. AES, DES, 3DES, RC4 and RABBIT are enabled by default. Camellia, HC-128, and ChaCha20 can be enabled when building wolfSSL (with the `--enable-hc128`, `--enable-camellia`, and `--disable-chacha` build options, respectively). The default mode of AES is CBC mode. To enable GCM or CCM mode with AES, use the `--enable-aesgcm` and `--enable-aesccm` build options. Please see the examples for usage and the wolfCrypt Usage Reference for specific usage information.

While SSL uses RC4 as the default stream cipher, it has been obsoleted due to compromise. wolfSSL has added two ciphers from the eStream project into the code base, RABBIT and HC-128. RABBIT is nearly twice as fast as RC4 and HC-128 is about 5 times as fast! So if you've ever decided not to use SSL because of speed concerns, using wolfSSL's stream ciphers should lessen or eliminate that performance doubt. Recently wolfSSL also added ChaCha20. While RC4 is about 11% more performant than ChaCha, RC4 is generally considered less secure than ChaCha. ChaCha can put up very nice times of it's own with added security as a tradeoff.

To see a comparison of cipher performance, visit the wolfSSL Benchmark web page, located here: https://www.wolfssl.com/docs/benchmarks.

4.3.4.1 What's the Difference? A block cipher has to be encrypted in chunks that are the block size for the cipher. For example, AES has a block size of 16 bytes. So if you're encrypting a bunch of small, 2 or 3 byte chunks back and forth, over 80% of the data is useless padding, decreasing the speed of the encryption/decryption process and needlessly wasting network bandwidth to boot. Basically block ciphers are designed for large chunks of data, have block sizes requiring padding, and use a fixed, unvarying transformation.

Stream ciphers work well for large or small chunks of data. They are suitable for smaller data sizes because no block size is required. If speed is a concern, stream ciphers are your answer, because they use a simpler transformation that typically involves an xor'd keystream. So if you need to stream media, encrypt various data sizes including small ones, or have a need for a fast cipher then stream ciphers are your best bet.

4.3.5 Hashing Functions

wolfSSL supports several different hashing functions, including **MD2**, **MD4**, **MD5**, **SHA-1**, **SHA-2** (SHA-224, SHA-256, SHA-384, SHA-512), **SHA-3** (BLAKE2), and **RIPEMD-160**. Detailed usage of these functions can be found in the wolfCrypt Usage Reference, Hash Functions.

4.3.6 Public Key Options

wolfSSL supports the **RSA**, **ECC**, **DSA/DSS**, **DH**, and **NTRU** public key options, with support for **EDH** (Ephemeral Diffie-Hellman) on the wolfSSL server. Detailed usage of these functions can be found in the wolfCrypt Usage Reference, Public Key Cryptography.

wolfSSL has support for four cipher suites utilizing NTRU public key:

- `TLS_NTRU_RSA_WITH_3DES_EDE_CBC_SHA`
- `TLS_NTRU_RSA_WITH_RC4_128_SHA`
- `TLS_NTRU_RSA_WITH_AES_128_CBC_SHA`
- `TLS_NTRU_RSA_WITH_AES_256_CBC_SHA`

The strongest one, AES-256, is the default. If wolfSSL is enabled with NTRU and the NTRU library is available, these cipher suites are built into the wolfSSL library. A wolfSSL client will have these cipher suites available without any interaction needed by the user. On the other hand, a wolfSSL server application will need to load an NTRU private key and NTRU x509 certificate in order for those cipher suites to be available for use.

The example servers, echoserver and server, both use the define HAVE_NTRU (which is turned on by enabling NTRU) to specify whether or not to load NTRU keys and certificates. The wolfSSL package comes with test keys and certificates in the /certs directory. ntru-cert.pem is the certificate and ntru-key.raw is the private key blob.

The wolfSSL NTRU cipher suites are given the highest preference order when the protocol picks a suite. Their exact preference order is the reverse of the above listed suites, i.e., AES-256 will be picked first and 3DES last before moving onto the "standard" cipher suites. Basically, if a user builds NTRU into wolfSSL and both sides of the connection support NTRU then an NTRU cipher suite will be picked unless a user on one side has explicitly excluded them by stating to only use different cipher suites. Using NTRU over RSA can provide a 20 - 200X speed improvement. The improvement increases as the size of keys increases, meaning a much larger speed benefit when using large keys (8192-bit) versus smaller keys (1024-bit).

4.3.7 ECC Support

wolfSSL has support for Elliptic Curve Cryptography (ECC) including but not limited to: ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, ECDHE-PSK and ECDHE-RSA.

wolfSSL's ECC implementation can be found in the `<wolfssl_root>/wolfssl/wolfcrypt/ecc.h` header file and the `<wolfssl_root>/wolfcrypt/src/ecc.c` source file.

Supported cipher suites are shown in the table above. ECC is disabled by default on non x86_64 builds, but can be turned on when building wolfSSL with the HAVE_ECC define or by using the autoconf system:

```
./configure --enable-ecc
make
make check
```

When `make check` runs, note the numerous cipher suites that wolfSSL checks (if make check doesn't produce a list of cipher suites run `./testsuite/testsuite.test` on its own). Any of these cipher suites can be tested individually, e.g., to try ECDH-ECDSA with AES256-SHA, the example wolfSSL server can be started like this:

```
./examples/server/server -d -l ECDHE-ECDSA-AES256-SHA -c ./certs/server-ecc.pem -k
↪  ./certs/ecc-key.pem
```

(-d) disables client cert check while (-l) specifies the cipher suite list. (-c) is the certificate to use and (-k) is the corresponding private key to use. To have the client connect try:

```
./examples/client/client -A ./certs/server-ecc.pem
```

where (-A) is the CA certificate to use to verify the server.

4.3.8 PKCS Support

PKCS (Public Key Cryptography Standards) refers to a group of standards created and published by RSA Security, Inc. wolfSSL has support for **PKCS #1, PKCS #3, PKCS #5, PKCS #7, PKCS #8, PKCS #9, PKCS #10, PKCS #11,** and **PKCS #12.**

Additionally, wolfSSL also provides support for RSA-Probabilistic Signature Scheme (PSS), which is standardized as part of PKCS #1.

4.3.8.1 PKCS #5, PBKDF1, PBKDF2, PKCS #12 PKCS #5 is a password based key derivation method which combines a password, a salt, and an iteration count to generate a password-based key. wolfSSL supports both PBKDF1 and PBKDF2 key derivation functions. A key derivation function produces a derived key from a base key and other parameters (such as the salt and iteration count as explained above). PBKDF1 applies a hash function (MD5, SHA1, etc) to derive keys, where the derived key length is bounded by the length of the hash function output. With PBKDF2, a pseudorandom function is applied (such as HMAC-SHA-1) to derive the keys. In the case of PBKDF2, the derived key length is unbounded.

wolfSSL also supports the PBKDF function from PKCS #12 in addition to PBKDF1 and PBKDF2. The function prototypes look like this:

```
int PBKDF2(byte* output, const byte* passwd, int pLen,
           const byte* salt,int sLen, int iterations,
           int kLen, int hashType);

int PKCS12_PBKDF(byte* output, const byte* passwd, int pLen,
                 const byte* salt, int sLen, int iterations,
                 int kLen, int hashType, int purpose);
```

`output` contains the derived key, `passwd` holds the user password of length `pLen`, `salt` holds the salt input of length `sLen`, `iterations` is the number of iterations to perform, `kLen` is the desired derived key length, and `hashType` is the hash to use (which can be MD5, SHA1, or SHA2).

If you are using `./configure` to build wolfssl, the way to enable this functionality is to use the option `--enable-pwdbased`

A full example can be found in `<wolfSSL Root>/wolfcrypt/test.c`. More information can be found on PKCS #5, PBKDF1, and PBKDF2 from the following specifications:

PKCS#5, PBKDF1, PBKDF2: https://tools.ietf.org/html/rfc2898

4.3.8.2 PKCS #8 PKCS #8 is designed as the Private-Key Information Syntax Standard, which is used to store private key information - including a private key for some public-key algorithm and set of attributes.

The PKCS #8 standard has two versions which describe the syntax to store both encrypted private keys and non-encrypted keys. wolfSSL supports both unencrypted and encrypted PKCS #8. Supported formats include PKCS #5 version 1 - version 2, and PKCS#12. Types of encryption available include DES, 3DES, RC4, and AES.

PKCS#8: https://tools.ietf.org/html/rfc5208

4.3.8.3 PKCS #7 PKCS #7 is designed to transfer bundles of data whether is an enveloped certificate or unencrypted but signed string of data. The functionality is turned on by using the enable option (`--enable-pkcs7`) or by using the macro `HAVE_PKCS7`. Note that degenerate cases are allowed by default as per the RFC having an empty set of signers. To toggle allowing degenerate cases on and off the function `wc_PKCS7_AllowDegenerate()` can be called.

Supported features include:

- Degenerate bundles
- KARI, KEKRI, PWRI, ORI, KTRI bundles
- Detached signatures
- Compressed and Firmware package bundles
- Custom callback support
- Limited streaming capability

4.3.8.3.1 PKCS #7 Callbacks Additional callbacks and supporting functions were added to allow for a user to choose their keys after the PKCS7 bundle has been parsed.For unwrapping the CEK the function `wc_PKCS7_SetWrapCEKCb()` can be called. The callback set by this function gets called in the case of KARI and KEKRI bundles. The keyID or SKID gets passed from wolfSSL to the user along with the originator key in the case of KARI. After the user unwraps the CEK with their KEK the decrypted key to be used should then be passed back to wolfSSL. An example of this can be found in the wolfssl-examples repository in the file `signedData-EncryptionFirmwareCB.c`.

An additional callback was added for decryption of PKCS7 bundles. For setting a decryption callback function the API `wc_PKCS7_SetDecodeEncryptedCb()` can be used. To set a user defined context the API `wc_PKCS7_SetDecodeEncryptedCtx()` should be used. This callback will get executed on calls to `wc_PKCS7_DecodeEncryptedData()`.

4.3.8.3.2 PKCS #7 Streaming Stream oriented API for PKCS7 decoding gives the option of passing inputs in smaller chunks instead of all at once.By default the streaming functionality with PKCS7 is on. To turn off support for streaming PKCS7 API the macro `NO_PKCS7_STREAM` can be defined. An example of doing this with autotools would be `./configure --enable-pkcs7 CFLAGS=-DNO_PKCS7_STREAM`.

For streaming when decoding/verifying bundles the following functions are supported:

1. `wc_PKCS7_DecodeEncryptedData()`
2. `wc_PKCS7_VerifySignedData()`
3. `wc_PKCS7_VerifySignedData_ex()`
4. `wc_PKCS7_DecodeEnvelopedData()`
5. `wc_PKCS7_DecodeAuthEnvelopedData()`

Note: that when calling `wc_PKCS7_VerifySignedData_ex` it is expected that the argument pkiMsgFoot is the full buffer. The internal structure only supports streaming of one buffer which in this case would be `pkiMsgHead`.

4.3.9 Forcing the Use of a Specific Cipher

By default, wolfSSL will pick the “best” (highest security) cipher suite that both sides of the connection can support. To force a specific cipher, such as 128 bit AES, add something similar to:

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

after the call to `wolfSSL_CTX_new()` so that you have:

```
ctx = wolfSSL_CTX_new(method);
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

4.3.10 Quantum-Safe Handshake Ciphersuite

wolfSSL has support for the cipher suite utilizing post quantum handshake cipher suite such as with NTRU: `TLS_QSH`

If wolfSSL is enabled with NTRU and the NTRU package is available, the `TLS_QSH` cipher suite is built into the wolfSSL library. A wolfSSL client and server will have this cipher suite available without any interaction needed by the user.

The wolfSSL quantum safe handshake ciphersuite is given the highest preference order when the protocol picks a suite. Basically, if a user builds NTRU into wolfSSL and both sides of the connection support NTRU then an NTRU cipher suite will be picked unless a user on one side has explicitly excluded them by stating to only use different cipher suites.

Users can adjust what crypto algorithms and if the client sends across public keys by using the function examples:

```
wolfSSL_UseClientQSHKeys(ssl, 1);
wolfSSL_UseSupportedQSH(ssl, WOLFSSL_NTRU_EESS439);
```

To test if a QSH connection was established after a client has connected the following function example can be used:

```
wolfSSL_isQSH(ssl);
```

4.4 Hardware Accelerated Crypto

wolfSSL is able to take advantage of several hardware accelerated (or "assisted") crypto functionalities in various processors and chips. The following sections explain which technologies wolfSSL supports out-of-the-box.

4.4.1 AES-NI

AES is a key encryption standard used by governments worldwide, which wolfSSL has always supported. Intel has released a new set of instructions that is a faster way to implement AES. wolfSSL is the first SSL library to fully support the new instruction set for production environments.

Essentially, Intel and AMD have added AES instructions at the chip level that perform the computationally-intensive parts of the AES algorithm, boosting performance. For a list of Intel's chips that currently have support for AES-NI, you can look here:

https://ark.intel.com/search/advanced/?s=t&AESTech=true

We have added the functionality to wolfSSL to allow it to call the instructions directly from the chip, instead of running the algorithm in software. This means that when you're running wolfSSL on a chipset that supports AES-NI, you can run your AES crypto 5-10 times faster!

If you are running on an AES-NI supported chipset, enable AES-NI with the `--enable-aesni` build option. To build wolfSSL with AES-NI, GCC 4.4.3 or later is required to make use of the assembly code. wolfSSL supports the ASM instructions on AMD processors using the same build options.

References and further reading on AES-NI, ordered from general to specific, are listed below. For information about performance gains with AES-NI, please see the third link to the Intel Software Network page.

- AES (Wikipedia)
- AES-NI (Wikipedia)
- AES-NI (Intel Software Network page)

AES-NI will accelerate the following AES cipher modes: AES-CBC, AES-GCM, AES-CCM-8, AES-CCM, and AES-CTR. AES-GCM is further accelerated with the use of the 128-bit multiply function added to the Intel chips for the GHASH authentication.

4.4.2 STM32F2

wolfSSL is able to use the STM32F2 hardware-based cryptography and random number generator through the STM32F2 Standard Peripheral Library.

For necessary defines, see the `WOLFSSL_STM32F2` define in `settings.h`. The `WOLFSSL_STM32F2` define enables STM32F2 hardware crypto and RNG support by default. The defines for enabling these individually are `STM32F2_CRYPTO` (for hardware crypto support) and `STM32F2_RNG` (for hardware RNG support).

Documentation for the STM32F2 Standard Peripheral Library can be found in the following document: https://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/DM00023896.pdf

4.4.3 Cavium NITROX

wolfSSL has support for Cavium NITROX (https://www.cavium.com/processor_security.html). To enable Cavium NITROX support when building wolfSSL use the following configure option:

```
./configure --with-cavium=/home/user/cavium/software
```

Where the --with-cavium=** option is pointing to your licensed cavium/software directory. Since Cavium doesn't build a library wolfSSL pulls in the cavium_common.o file which gives a libtool warning about the portability of this. Also, if you're using the github source tree you'll need to remove the -Wredundant-decls warning from the generated Makefile because the cavium headers don't conform to this warning.

Currently wolfSSL supports Cavium RNG, AES, 3DES, RC4, HMAC, and RSA directly at the crypto layer. Support at the SSL level is partial and currently just does AES, 3DES, and RC4. RSA and HMAC are slower until the Cavium calls can be utilized in non-blocking mode. The example client turns on cavium support as does the crypto test and benchmark. Please see the HAVE_CAVIUM define.

4.4.4 ESP32-WROOM-32

wolfSSL is able to use the ESP32-WROOM-32 hardware-based cryptography.

For necessary defines, see the WOLFSSL_ESPWROOM32 define in settings.h. The WOLFSSL_ESPWROOM32 define enables ESP32-WROOM-32 hardware crypto and RNG support by default. Currently wolfSSL supports RNG, AES, SHA and RSA primitive at the crypt layer. The example projects including TLS server/client, wolfCrypt test and benchmark can be found at /examples/protocols directory in ESP-IDF after deploying files.

4.5 SSL Inspection (Sniffer)

Beginning with the wolfSSL 1.5.0 release, wolfSSL has included a build option allowing it to be built with SSL Sniffer (SSL Inspection) functionality. This means that you can collect SSL traffic packets and with the correct key file, are able to decrypt them as well. The ability to "inspect" SSL traffic can be useful for several reasons, some of which include:

- Analyzing Network Problems
- Detecting network misuse by internal and external users
- Monitoring network usage and data in motion
- Debugging client/server communications

To enable sniffer support, build wolfSSL with the --enable-sniffer option on *nix or use the **vcproj** files on Windows. You will need to have **pcap** installed on *nix or **WinPcap** on Windows. The main sniffer functions which can be found in sniffer.h are listed below with a short description of each:

- ssl_SetPrivateKey - Sets the private key for a specific server and port.
- ssl_SetNamedPrivateKey - Sets the private key for a specific server, port and domain name.
- ssl_DecodePacket - Passes in a TCP/IP packet for decoding.
- ssl_Trace - Enables / Disables debug tracing to the traceFile.
- ssl_InitSniffer - Initialize the overall sniffer.
- ssl_FreeSniffer - Free the overall sniffer.
- ssl_EnableRecovery - Enables option to attempt to pick up decoding of SSL traffic in the case of lost packets.
- ssl_GetSessionStats - Obtains memory usage for the sniffer sessions.

To look at wolfSSL's sniffer support and see a complete example, please see the snifftest app in the sslSniffer/sslSnifferTest folder from the wolfSSL download.

Keep in mind that because the encryption keys are setup in the SSL Handshake, the handshake needs to be decoded by the sniffer in order for future application data to be decoded. For example, if you are using "snifftest" with the wolfSSL example echoserver and echoclient, the snifftest application must be started before the handshake begins between the server and client.

The sniffer can only decode streams encrypted with the following algorithms: AES-CBC, DES3-CBC, ARC4, HC-128, RABBIT, Camellia-CBC, and IDEA. If ECDHE or DHE key agreement is used the stream cannot be sniffed; only RSA or ECDH key-exchange is supported.

Watch callbacks with wolfSSL sniffer can be turned on with WOLFSSL_SNIFFER_WATCH. With the sniffer watch feature compiled in, the function ssl_SetWatchKeyCallback() can be used to set a custom callback. The callback is then used to inspect the certificate chain, error value, and digest of the certificate sent from the peer. If a non 0 value is returned from the callback then an error state is set when processing the peer's certificate. Additional supporting functions for the watch callbacks are:

- ssl_SetWatchKeyCtx: Sets a custom user context that gets passed to the watch callback.
- ssl_SetWatchKey_buffer: Loads a new DER format key into server session.
- ssl_SetWatchKey_file: File version of ssl_SetWatchKey_buffer.

Statistics collecting with the sniffer can be compiled in with defining the macro WOLFSSL_SNIFFER_STATS. The statistics are kept in a SSLStats structure and are copied to an applications SSLStats structure by a call to ssl_ReadStatistics. Additional API to use with sniffer statistics is ssl_ResetStatistics (resets the collection of statistics) and ssl_ReadResetStatistics (reads the current statistic values and then resets the internal state). The following is the current statistics kept when turned on:

- sslStandardConns
- sslClientAuthConns
- sslResumedConns
- sslEphemeralMisses
- sslResumeMisses
- sslCiphersUnsupported
- sslKeysUnmatched
- sslKeyFails
- sslDecodeFails
- sslAlerts
- sslDecryptedBytes
- sslEncryptedBytes
- sslEncryptedPackets
- sslDecryptedPackets
- sslKeyMatches
- sslEncryptedConns

4.6 Compression

wolfSSL supports data compression with the **zlib** library. The ./configure build system detects the presence of this library, but if you're building in some other way define the constant HAVE_LIBZ and include the path to zlib.h for your includes.

Compression is off by default for a given cipher. To turn it on, use the function wolfSSL_set_compression() before SSL connecting or accepting. Both the client and server must have compression turned on in order for compression to be used.

Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer time to analyze than it does to send it raw on all but the slowest of networks.

4.7 Pre-Shared Keys

wolfSSL has support for these ciphers with static pre-shared keys:

- TLS_PSK_WITH_AES_256_CBC_SHA
- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_PSK_WITH_AES_256_CBC_SHA384
- TLS_PSK_WITH_AES_128_CBC_SHA
- TLS_PSK_WITH_NULL_SHA256
- TLS_PSK_WITH_NULL_SHA384
- TLS_PSK_WITH_NULL_SHA
- TLS_PSK_WITH_AES_128_GCM_SHA256
- TLS_PSK_WITH_AES_256_GCM_SHA384
- TLS_PSK_WITH_AES_128_CCM
- TLS_PSK_WITH_AES_256_CCM
- TLS_PSK_WITH_AES_128_CCM_8
- TLS_PSK_WITH_AES_256_CCM_8
- TLS_PSK_WITH_CHACHA20_POLY1305

These suites are built into wolfSSL with WOLFSSL_STATIC_PSK on, all PSK suites can be turned off at build time with the constant NO_PSK. To only use these ciphers at runtime use the function wolfSSL_CTX_set_cipher_list() with the desired ciphersuite.

wolfSSL has support for ephemeral key PSK suites:

- ECDHE-PSK-AES128-CBC-SHA256
- ECDHE-PSK-NULL-SHA256
- ECDHE-PSK-CHACHA20-POLY1305
- DHE-PSK-CHACHA20-POLY1305
- DHE-PSK-AES256-GCM-SHA384
- DHE-PSK-AES128-GCM-SHA256
- DHE-PSK-AES256-CBC-SHA384
- DHE-PSK-AES128-CBC-SHA256
- DHE-PSK-AES128-CBC-SHA256

On the client, use the function wolfSSL_CTX_set_psk_client_callback() to setup the callback. The client example in <wolfSSL_Home>/examples/client/client.c gives example usage for setting up the client identity and key, though the actual callback is implemented in wolfssl/test.h.

On the server side two additional calls are required:

- wolfSSL_CTX_set_psk_server_callback()
- wolfSSL_CTX_use_psk_identity_hint()

The server stores its identity hint to help the client with the 2nd call, in our server example that's "wolfssl server". An example server psk callback can also be found in my_psk_server_cb() in wolfssl/test.h.

wolfSSL supports identities and hints up to 128 octets and pre-shared keys up to 64 octets.

4.8 Client Authentication

Client authentication is a feature which enables the server to authenticate clients by requesting that the clients send a certificate to the server for authentication when they connect. Client authentication requires an X.509 client certificate from a CA (or self-signed if generated by you or someone other than a CA).

By default, wolfSSL validates all certificates that it receives - this includes both client and server. To set up client authentication, the server must load the list of trusted CA certificates to be used to verify the client certificate against:

```
wolfSSL_CTX_load_verify_locations(ctx, caCert, 0);
```

To turn on client verification and control its behavior, the wolfSSL_CTX_set_verify() function is used. In the following example, SSL_VERIFY_PEER turns on a certificate request from the server to the client. SSL_VERIFY_FAIL_IF_NO_PEER_CERT instructs the server to fail if the client does not present a certificate to validate on the server side. Other options to wolfSSL_CTX_set_verify() include SSL_VERIFY_NONE and SSL_VERIFY_CLIENT_ONCE.

```
wolfSSL_CTX_set_verify(ctx,SSL_VERIFY_PEER | ((usePskPlus)?
                       SSL_VERIFY_FAIL_EXCEPT_PSK :
                       SSL_VERIFY_FAIL_IF_NO_PEER_CERT),0);
```

An example of client authentication can be found in the example server (server.c) included in the wolfSSL download (/examples/server/server.c).

4.9 Server Name Indication

SNI is useful when a server hosts multiple 'virtual' servers at a single underlying network address. It may be desirable for clients to provide the name of the server which it is contacting. To enable SNI with wolfSSL you can simply do:

```
./configure --enable-sni
```

Using SNI on the client side requires an additional function call, which should be one of the following functions:

- wolfSSL_CTX_UseSNI()
- wolfSSL_UseSNI()

wolfSSL_CTX_UseSNI() is most recommended when the client contacts the same server multiple times. Setting the SNI extension at the context level will enable the SNI usage in all SSL objects created from that same context from the moment of the call forward.

wolfSSL_UseSNI() will enable SNI usage for one SSL object only, so it is recommended to use this function when the server name changes between sessions.

On the server side one of the same function calls is required. Since the wolfSSL server doesn't host multiple 'virtual' servers, the SNI usage is useful when the termination of the connection is desired in the case of SNI mismatch. In this scenario, wolfSSL_CTX_UseSNI() will be more efficient, as the server will set it only once per context creating all subsequent SSL objects with SNI from that same context.

4.10 Handshake Modifications

4.10.1 Grouping Handshake Messages

wolfSSL has the ability to group handshake messages if the user desires. This can be done at the context level with wolfSSL_CTX_set_group_messages(ctx); or at the SSL object level with wolfSSL_set_group_messages(ssl);.

4.11 Truncated HMAC

Currently defined TLS cipher suites use the HMAC to authenticate record-layer communications. In TLS, the entire output of the hash function is used as the MAC tag. However, it may be desirable in constrained environments to save bandwidth by truncating the output of the hash function to 80 bits when forming MAC tags. To enable the usage of Truncated HMAC at wolfSSL you can simply do:

```
./configure --enable-truncatedhmac
```

Using Truncated HMAC on the client side requires an additional function call, which should be one of the following functions:

- wolfSSL_CTX_UseTruncatedHMAC()
- wolfSSL_UseTruncatedHMAC()

wolfSSL_CTX_UseTruncatedHMAC() is most recommended when the client would like to enable Truncated HMAC for all sessions. Setting the Truncated HMAC extension at context level will enable it in all SSL objects created from that same context from the moment of the call forward.

wolfSSL_UseTruncatedHMAC() will enable it for one SSL object only, so it's recommended to use this function when there is no need for Truncated HMAC on all sessions.

On the server side no call is required. The server will automatically attend to the client's request for Truncated HMAC.

All TLS extensions can also be enabled with:

```
./configure --enable-tlsx
```

4.12 User Crypto Module

User Crypto Module allows for a user to plug in custom crypto that they want used during supported operations (Currently RSA operations are supported). An example of a module is located in the directory root_wolfssl/wolfcrypt/user-crypto/ using IPP libraries. Examples of the configure option when building wolfSSL to use a crypto module is as follows:

```
./configure --with-user-crypto
```

or

```
./configure --with-user-crypto=/dir/to
```

When creating a user crypto module that performs RSA operations, it is mandatory that there is a header file for RSA called `user_rsa.h`. For all user crypto operations it is mandatory that the users library be called `libusercrypto`. These are the names that wolfSSL autoconf tools will be looking for when linking and using a user crypto module. In the example provided with wolfSSL, the header file `user_rsa.h` can be found in the directory `wolfcrypt/user-crypto/include/` and the library once created is located in the directory `wolfcrypt/user-crypto/lib/`. For a list of required API look at the header file provided.

To build the example, after having installed IPP libraries, the following commands from the root wolfSSL directory should be ran.

```
cd wolfcrypt/user-crypto/
./autogen.sh
./configure
make
sudo make install
```

The included example in wolfSSL requires the use of IPP, which will need to be installed before the project can be built. Though even if not having IPP libraries to build the example it is intended to provide users with an example of file name choice and API interface. Once having made and installed both the library libusercrypto and header files, making wolfSSL use the crypto module does not require any extra steps. Simply using the configure flag `--with-user-crypto` will map all function calls from the typical wolfSSL crypto to the user crypto module.

Memory allocations, if using wolfSSL's XMALLOC, should be tagged with `DYNAMIC_TYPE_USER_CRYPTO`. Allowing for analyzing memory allocations used by the module.

User crypto modules **cannot** be used in conjunction with the wolfSSL configure options fast-rsa and/or fips. Fips requires that specific, certified code be used and fast-rsa makes use of the example user crypto module to perform RSA operations.

4.13 Timing-Resistance in wolfSSL

wolfSSL provides the function "ConstantCompare" which guarantees constant time when doing comparison operations that could potentially leak timing information. This API is used at both the TLS and crypto level in wolfSSL to deter against timing based, side-channel attacks.

The wolfSSL ECC implementation has the define `ECC_TIMING_RESISTANT` to enable timing-resistance in the ECC algorithm. Similarly the define `TFM_TIMING_RESISTANT` is provided in the fast math libraries for RSA algorithm timing-resistance. The function exptmod uses the timing resistant Montgomery ladder.

See also: `--disable-harden`

Timing resistance and cache resistance defines enabled with `--enable-harden`:

- `WOLFSSL_SP_CACHE_RESISTANT`: Enables logic to mask the address used.
- `WC_RSA_BLINDING`: Enables blinding mode, to prevent timing attacks.
- `ECC_TIMING_RESISTANT`: ECC specific timing resistance.
- `TFM_TIMING_RESISTANT`: Fast math specific timing resistance.

4.14 Fixed ABI

wolfSSL provides a fixed Application Binary Interface (ABI) for a subset of the Application Programming Interface (API). Starting with wolfSSL v4.3.0, the following functions will be compatible across all future releases of wolfSSL:

- `wolfSSL_Init()`
- `wolfTLSv1_2_client_method()`
- `wolfTLSv1_3_client_method()`
- `wolfSSL_CTX_new()`
- `wolfSSL_CTX_load_verify_locations()`
- `wolfSSL_new()`
- `wolfSSL_set_fd()`

- wolfSSL_connect()
- wolfSSL_read()
- wolfSSL_write()
- wolfSSL_get_error()
- wolfSSL_shutdown()
- wolfSSL_free()
- wolfSSL_CTX_free()
- wolfSSL_check_domain_name()
- wolfSSL_UseALPN()
- wolfSSL_CTX_SetMinVersion()
- wolfSSL_pending()
- wolfSSL_set_timeout()
- wolfSSL_CTX_set_timeout()
- wolfSSL_get_session()
- wolfSSL_set_session()
- wolfSSL_flush_sessions()
- wolfSSL_CTX_set_session_cache_mode()
- wolfSSL_get_sessionID()
- wolfSSL_UseSNI()
- wolfSSL_CTX_UseSNI()
- wc_ecc_init_ex()
- wc_ecc_make_key_ex()
- wc_ecc_sign_hash()
- wc_ecc_free()
- wolfSSL_SetDevId()
- wolfSSL_CTX_SetDevId()
- wolfSSL_CTX_SetEccSignCb()
- wolfSSL_CTX_use_certificate_chain_file()
- wolfSSL_CTX_use_certificate_file()
- wolfSSL_use_certificate_chain_file()
- wolfSSL_use_certificate_file()
- wolfSSL_CTX_use_PrivateKey_file()
- wolfSSL_use_PrivateKey_file()
- wolfSSL_X509_load_certificate_file()
- wolfSSL_get_peer_certificate()
- wolfSSL_X509_NAME_oneline()
- wolfSSL_X509_get_issuer_name()
- wolfSSL_X509_get_subject_name()
- wolfSSL_X509_get_next_altname()
- wolfSSL_X509_notBefore()
- wolfSSL_X509_notAfter()
- wc_ecc_key_new()
- wc_ecc_key_free()

5 Portability

5.1 Abstraction Layers

5.1.1 C Standard Library Abstraction Layer

wolfSSL (formerly CyaSSL) can be built without the C standard library to provide a higher level of portability and flexibility to developers. The user will have to map the functions they wish to use instead of the C standard ones.

5.1.1.1 Memory Use Most C programs use `malloc()` and `free()` for dynamic memory allocation. wolfSSL uses `XMALLOC()` and `XFREE()` instead. By default, these point to the C runtime versions. By defining `XMALLOC_USER`, the user can provide their own hooks. Each memory function takes two additional arguments over the standard ones, a heap hint, and an allocation type. The user is free to ignore these or use them in any way they like. You can find the wolfSSL memory functions in `wolfssl/wolfcrypt/types.h`.

wolfSSL also provides the ability to register memory override functions at runtime instead of compile time. `wolfssl/wolfcrypt/memory.h` is the header for this functionality and the user can call the following function to set up the memory functions:

```
int wolfSSL_SetAllocators(wolfSSL_Malloc_cb  malloc_function,
                          wolfSSL_Free_cb    free_function,
                          wolfSSL_Realloc_cb realloc_function);
```

See the header `wolfssl/wolfcrypt/memory.h` for the callback prototypes and `memory.c` for the implementation.

5.1.1.2 string.h wolfSSL uses several functions that behave like `string.h`'s `memcpy()`, `memset()`, and `memcmp()` amongst others. They are abstracted to `XMEMCPY()`, `XMEMSET()`, and `XMEMCMP()` respectively. And by default, they point to the C standard library versions. Defining `STRING_USER` allows the user to provide their own hooks in `types.h`. For example, by default `XMEMCPY()` is:

```
#define XMEMCPY(d,s,l)    memcpy((d),(s),(l))
```

After defining `STRING_USER` you could do:

```
#define XMEMCPY(d,s,l)    my_memcpy((d),(s),(l))
```

Or if you prefer to avoid macros:

```
external void* my_memcpy(void* d, const void* s, size_t n);
```

to set wolfSSL's abstraction layer to point to your version `my_memcpy()`.

5.1.1.3 math.h wolfSSL uses two functions that behave like `math.h`'s `pow()` `log()`. They are only required by Diffie-Hellman, so if you exclude DH from the build, then you don't have to provide your own. They are abstracted to `XPOW()` and `XLOG()` and found in `wolfcrypt/src/dh.c`.

5.1.1.4 File System Use By default, wolfSSL uses the system's file system for the purpose of loading keys and certificates. This can be turned off by defining `NO_FILESYSTEM`, see item V. If instead, you'd like to use a file system but not the system one, you can use the `XFILE()` layer in `ssl.c` to point the file system calls to the ones you'd like to use. See the example provided by the MICRIUM define.

5.1.2 Custom Input/Output Abstraction Layer

wolfSSL provides a custom I/O abstraction layer for those who wish to have higher control over I/O of their SSL connection or run SSL on top of a different transport medium other than TCP/IP.

The user will need to define two functions:

1. The network Send function
2. The network Receive function

These two functions are prototyped by CallbackIOSend and CallbackIORecv in ssl.h:

```
typedef int (*CallbackIORecv)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
```

The user needs to register these functions per WOLFSSL_CTX with wolfSSL_SetIOSend() and wolfSSL_SetIORecv(). For example, in the default case, CBIORecv() and CBIOSend() are registered at the bottom of io.c:

```
void wolfSSL_SetIORecv(WOLFSSL_CTX *ctx, CallbackIORecv CBIORecv)
{
     ctx->CBIORecv = CBIORecv;
}

void wolfSSL_SetIOSend(WOLFSSL_CTX *ctx, CallbackIOSend CBIOSend)
{
    ctx->CBIOSend = CBIOSend;
}
```

The user can set a context per WOLFSSL object (session) with wolfSSL_SetIOWriteCtx() and wolfSSL_SetIOReadCtx(), as demonstrated at the bottom of io.c. For example, if the user is using memory buffers, the context may be a pointer to a structure describing where and how to access the memory buffers. The default case, with no user overrides, registers the socket as the context.

The CBIORecv and CBIOSend function pointers can be pointed to your custom I/O functions. The default Send() and Receive() functions, EmbedSend() and EmbedReceive(), located in io.c, can be used as templates and guides.

WOLFSSL_USER_IO can be defined to remove the automatic setting of the default I/O functions EmbedSend() and EmbedReceive().

5.1.3 Operating System Abstraction Layer

The wolfSSL OS abstraction layer helps facilitate easier porting of wolfSSL to a user's operating system. The wolfssl/wolfcrypt/settings.h file contains settings which end up triggering the OS layer.

OS-specific defines are located in wolfssl/wolfcrypt/types.h for wolfCrypt and wolfssl/internal.h for wolfSSL.

5.2 Supported Operating Systems

One factor which defines wolfSSL is its ability to be easily ported to new platforms. As such, wolfSSL has support for a long list of operating systems out-of-the-box. Currently-supported operating systems include:

- Win32/64
- Linux
- Mac OS X
- Solaris
- ThreadX
- VxWorks
- FreeBSD
- NetBSD
- OpenBSD
- embedded Linux
- Yocto Linux
- OpenEmbedded
- WinCE
- Haiku
- OpenWRT
- iPhone (iOS)
- Android
- Nintendo Wii and Gamecube through DevKitPro

- QNX
- MontaVista
- NonStop
- TRON/ITRON/µITRON
- Micrium's µC/OS-III
- FreeRTOS
- SafeRTOS
- NXP/Freescale MQX
- Nucleus
- TinyOS
- HP/UX
- AIX
- ARC MQX
- TI-RTOS
- uTasker
- embOS
- INtime
- Mbed
- µT-Kernel
- RIOT
- CMSIS-RTOS
- FROSTED
- Green Hills INTEGRITY
- Keil RTX
- TOPPERS
- PetaLinux
- Apache Mynewt

5.3 Supported Chipmakers

wolfSSL has support for chipsets including ARM, Intel, Motorola, mbed, Freescale, Microchip (PIC32), STMicro (STM32F2/F4), NXP, Analog Devices, Texas Instruments, AMD and more.

5.4 C# Wrapper

wolfSSL has limited support for use in C#. A Visual Studio project containing the port can be found in the directory `root_wolfSSL/wrapper/CSharp/`. After opening the Visual Studio project set the "Active solution configuration" and "Active solution platform" by clicking on BUILD->Configuration Manager... The supported "Active solution configuration"s are DLL Debug and DLL Release. The supported platforms are Win32 and x64.

Once having set the solution and platform the preprocessor flag `HAVE_CSHARP` will need to be added. This turns on the options used by the C# wrapper and used by the examples included.

To then build simply select build solution. This creates the `wolfssl.dll`, `wolfSSL_CSharp.dll` and examples. Examples can be ran by targeting them as an entry point and then running debug in Visual Studio.

Adding the created C# wrapper to C# projects can be done a couple of ways. One way is to install the created `wolfssl.dll` and `wolfSSL_CSharp.dll` into the directory `C:/Windows/System/`. This will allow projects that have:

```
using wolfSSL.CSharp

public some_class {

    public static main(){
    wolfssl.Init()
    ...
```

```
}
...
```

to make calls to the wolfSSL C# wrapper. Another way is to create a Visual Studio project and have it reference the bundled C# wrapper solution in wolfSSL.

6 Callbacks

6.1 HandShake Callback

wolfSSL (formerly CyaSSL) has an extension that allows a HandShake Callback to be set for connect or accept. This can be useful in embedded systems for debugging support when another debugger isn't available and sniffing is impractical. To use wolfSSL HandShake Callbacks, use the extended functions, `wolfSSL_connect_ex()` and `wolfSSL_accept_ex()`:

```
int wolfSSL_connect_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                       Timeval)
int wolfSSL_accept_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                      Timeval)
```

HandShakeCallBack is defined as:

```
typedef int (*HandShakeCallBack)(HandShakeInfo*);
```

HandShakeInfo is defined in `wolfssl/callbacks.h` (which should be added to a non-standard build):

```
typedef struct handShakeInfo_st {
    char   cipherName[MAX_CIPHERNAME_SZ + 1];  /*negotiated name */
    char   packetNames[MAX_PACKETS_HANDSHAKE][MAX_PACKETNAME_SZ+1];
                                        /* SSL packet names */
    int    numberPackets;               /*actual # of packets  */
    int    negotiationError;            /*cipher/parameter err */
} HandShakeInfo;
```

No dynamic memory is used since the maximum number of SSL packets in a handshake exchange is known. Packet names can be accessed through `packetNames[idx]` up to `numberPackets`. The callback will be called whether or not a handshake error occurred. Example usage is also in the client example.

6.2 Timeout Callback

The same extensions used with wolfSSL Handshake Callbacks can be used for wolfSSL Timeout Callbacks as well. These extensions can be called with either, both, or neither callbacks (Handshake and/or Timeout). `TimeoutCallback` is defined as:

```
typedef int (*TimeoutCallBack)(TimeoutInfo*);
```

Where `TimeoutInfo` looks like:

```
typedef struct timeoutInfo_st {
    char       timeoutName[MAX_TIMEOUT_NAME_SZ +1]; /*timeout Name*/
    int        flags;                          /* for future use*/
    int        numberPackets;               /*actual # of packets */
    PacketInfo packets[MAX_PACKETS_HANDSHAKE];  /*list of packets */
    Timeval    timeoutValue;               /*timer that caused it */
} TimeoutInfo;
```

Again, no dynamic memory is used for this structure since a maximum number of SSL packets is known for a handshake. `Timeval` is just a typedef for struct timeval.

`PacketInfo` is defined like this:

```
typedef struct packetInfo_st {
    char           packetName[MAX_PACKETNAME_SZ + 1]; /* SSL name */
    Timeval        timestamp;                 /* when it occured */
    unsigned char  value[MAX_VALUE_SZ];     /* if fits, it's here */
    unsigned char* bufferValue;         /* otherwise here (non 0) */
    int            valueSz;              /* sz of value or buffer */
} PacketInfo;
```

Here, dynamic memory may be used. If the SSL packet can fit in `value` then that's where it's placed. `valueSz` holds the length and `bufferValue` is 0. If the packet is too big for `value`, only **Certificate** packets should cause this, then the packet is placed in `bufferValue`. `valueSz` still holds the size.

If memory is allocated for a **Certificate** packet then it is reclaimed after the callback returns. The timeout is implemented using signals, specifically `SIGALRM`, and is thread safe. If a previous alarm is set of type `ITIMER_REAL` then it is reset, along with the correct handler, afterwards. The old timer will be time adjusted for any time wolfSSL spends processing. If an existing timer is shorter than the passed timer, the existing timer value is used. It is still reset afterwards. An existing timer that expires will be reset if has an interval associated with it. The callback will only be issued if a timeout occurs.

See the client example for usage.

6.3 User Atomic Record Layer Processing

wolfSSL provides Atomic Record Processing callbacks for users who wish to have more control over MAC/encrypt and decrypt/verify functionality during the SSL/TLS connection.

The user will need to define 2 functions:

1. MAC/encrypt callback function
2. Decrypt/verify callback function

These two functions are prototyped by `CallbackMacEncrypt` and `CallbackDecryptVerify` in `ssl.h`:

```
typedef int (*CallbackMacEncrypt)(WOLFSSL* ssl,
    unsigned char* macOut,const unsigned char* macIn,
    unsigned int macInSz,int macContent, int macVerify,
    unsigned char* encOut, const unsigned char* encIn,
    unsigned int encSz,void* ctx);

typedef int (*CallbackDecryptVerify)(WOLFSSL* ssl,
    unsigned char* decOut, const unsigned char* decIn,
    unsigned int decSz, int content, int verify,
    unsigned int* padSz, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (`WOLFSSL_CTX`) with `wolfSSL_CTX_SetMacEncryptCb()` and `wolfSSL_CTX_SetDecryptVerifyCb()`.

The user can set a context per WOLFSSL object (session) with `wolfSSL_SetMacEncryptCtx()` and `wolfSSL_SetDecryptVerifyCtx()`. This context may be a pointer to any user-specified context, which will then in turn be passed back to the MAC/encrypt and decrypt/verify callbacks through the `void* ctx` parameter.

1. Example callbacks can be found in `wolfssl/test.h`, under `myMacEncryptCb()` and `myDecryptVerifyCb()`. Usage can be seen in the wolfSSL example client (`examples/client/client.c`), when using the `-U` command line option.

To use Atomic Record Layer callbacks, wolfSSL needs to be compiled using the `--enable-atomicuser` configure option, or by defining the `ATOMIC_USER` preprocessor flag.

6.4 Public Key Callbacks

wolfSSL provides Public Key callbacks for users who wish to have more control over ECC sign/verify functionality as well as RSA sign/verify and encrypt/decrypt functionality during the SSL/TLS connection.

The user can optionally define 7 functions:

1. ECC sign callback
2. ECC verify callback
3. ECC shared secret callback
4. RSA sign callback
5. RSA verify callback

6. RSA encrypt callback
7. RSA decrypt callback

These two functions are prototyped by CallbackEccSign, CallbackEccVerify, CallbackEccSharedSecret, CallbackRsaSign, CallbackRsaVerify, CallbackRsaEnc, and CallbackRsaDec in ssl.h:

```
typedef int (*CallbackEccSign)(WOLFSSL* ssl, const unsigned
        char* in, unsigned int inSz, unsigned char* out,
        unsigned int* outSz, const unsigned char* keyDer,
        unsigned int keySz, void* ctx);

typedef int (*CallbackEccVerify)(WOLFSSL* ssl,
    const unsigned char* sig, unsigned int sigSz,
    const unsigned char* hash, unsigned int hashSz,
    const unsigned char* keyDer, unsigned int keySz,
    int* result, void* ctx);

typedef int (*CallbackEccSharedSecret)(WOLFSSL* ssl,
        struct ecc_key* otherKey,
    unsigned char* pubKeyDer, unsigned int* pubKeySz,
    unsigned char* out, unsigned int* outlen,
    int side, void* ctx);

typedef int (*CallbackRsaSign)(WOLFSSL* ssl,
    const unsigned char* in, unsigned int inSz,
    unsigned char* out, unsigned int* outSz,
    const unsigned char* keyDer, unsigned int keySz,
    void* ctx);

typedef int (*CallbackRsaVerify)(WOLFSSL* ssl,
    unsigned char* sig, unsigned int sigSz,
    unsigned char** out, const unsigned char* keyDer,
    unsigned int keySz, void* ctx);

typedef int (*CallbackRsaEnc)(WOLFSSL* ssl,
    const unsigned char* in, unsigned int inSz,
    Unsigned char* out, unsigned int* outSz,
    const unsigned char* keyDer,
    unsigned int keySz, void* ctx);

typedef int (*CallbackRsaDec)(WOLFSSL* ssl, unsigned char* in,
    unsigned int inSz, unsigned char** out,
    const unsigned char* keyDer, unsigned int keySz,
    void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with:

- wolfSSL_CTX_SetEccSignCb()
- wolfSSL_CTX_SetEccVerifyCb()
- wolfSSL_CTX_SetEccSharedSecretCb()
- wolfSSL_CTX_SetRsaSignCb()
- wolfSSL_CTX_SetRsaVerifyCb()
- wolfSSL_CTX_SetRsaEncCb()
- wolfSSL_CTX_SetRsaDecCb()

The user can set a context per WOLFSSL object (session) with:

- wolfSSL_SetEccSignCtx()
- wolfSSL_SetEccVerifyCtx()
- wolfSSL_SetEccSharedSecretCtx()
- wolfSSL_SetRsaSignCtx()
- wolfSSL_SetRsaVerifyCtx()
- wolfSSL_SetRsaEncCtx()
- wolfSSL_SetRsaDecCtx()

These contexts may be pointers to any user-specified context, which will then in turn be passed back to the respective public key callback through the void* ctx parameter.

Example callbacks can be found in wolfssl/test.h, under myEccSign(), myEccVerify(), myEccSharedSecret(), myRsaSign(), myRsaVerify(), myRsaEnc(), and myRsaDec(). Usage can be seen in the wolfSSL example client (examples/client/client.c), when using the -P command line option.

To use Atomic Record Layer callbacks, wolfSSL needs to be compiled using the --enable-pkcallbacks configure option, or by defining the HAVE_PK_CALLBACKS preprocessor flag.

7 Keys and Certificates

For an introduction to X.509 certificates, as well as how they are used in SSL and TLS, please see Appendix A.

7.1 Supported Formats and Sizes

wolfSSL (formerly CyaSSL) has support for **PEM**, and **DER** formats for certificates and keys, as well as PKCS#8 private keys (with PKCS#5 or PKCS#12 encryption).

PEM, or "Privacy Enhanced Mail" is the most common format that certificates are issued in by certificate authorities. PEM files are Base64 encoded ASCII files which can include multiple server certificates, intermediate certificates, and private keys, and usually have a `.pem`, `.crt`, `.cer`, or `.key` file extension. Certificates inside PEM files are wrapped in the "`-----BEGIN CERTIFICATE-----`" and "`-----END CERTIFICATE-----`" statements.

DER, or "Distinguished Encoding Rules", is a binary format of a certificate. DER file extensions can include `.der` and `.cer`, and cannot be viewed with a text editor.

An X.509 certificate is encoded using ASN.1 format. The DER format is the ASN.1 encoding. The PEM format is Base64 encoded and wrapped with a human readable header and footer. TLS send certificates in DER format.

7.2 Certificate Loading

Certificates are normally loaded using the file system (although loading from memory buffers is supported as well - see No File System and using Certificates).

7.2.1 Loading CA Certificates**

CA certificate files can be loaded using the `wolfSSL_CTX_load_verify_locations()` function:

```
int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX *ctx,
                                      const char *CAfile,
                                      const char *CApath);
```

CA loading can also parse multiple CA certificates per file using the above function by passing in a `CAfile` in PEM format with as many certs as possible. This makes initialization easier, and is useful when a client needs to load several root CAs at startup. This makes wolfSSL easier to port into tools that expect to be able to use a single file for CAs.

NOTE: If you have to load a chain of Roots and Intermediate certificates you must load them in the order of trust. Load ROOT CA first followed by Intermediate 1 followed by Intermediate 2 and so on. You may call `wolfSSL_CTX_load_verify_locations()` for each cert to be loaded or just once with a file containing the certs in order (Root at the top of the file and certs ordered by the chain of trust)

7.2.2 Loading Client or Server Certificates

Loading single client or server certificates can be done with the `wolfSSL_CTX_use_certificate_file()` function. If this function is used with a certificate chain, only the actual, or "bottom" certificate will be sent.

```
int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX *ctx,
                                     const char *CAfile,
                                     int type);
```

`CAfile` is the CA certificate file, and `type` is the format of the certificate - such as `SSL_FILETYPE_PEM`.

The server and client can send certificate chains using the `wolfSSL_CTX_use_certificate_chain_file()` function. The certificate chain file must be in PEM format and must be sorted starting with the subject's certificate (the actual client or server cert), followed by any intermediate certificates and ending (optionally) at the root "top" CA. The example server (`/examples/server/server.c`) uses this functionality.

NOTE: This is the exact reverse of the order necessary when loading a certificate chain for verification! Your file contents in this scenario would be Entity cert at the top of the file followed by the next cert up the chain and so on with Root CA at the bottom of the file.

```
int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX *ctx,
                                  const char *file);
```

7.2.3 Loading Private Keys

Server private keys can be loaded using the `wolfSSL_CTX_use_PrivateKey_file()` function.

```
int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX *ctx,
                      const char *keyFile, int type);
```

`keyFile` is the private key file, and `type` is the format of the private key (e.g. `SSL_FILETYPE_PEM`).

7.2.4 Loading Trusted Peer Certificates

Loading a trusted peer certificate to use can be done with `wolfSSL_CTX_trust_peer_cert()`.

```
int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX *ctx,
                 const char *trustCert, int type);
```

`trustCert` is the certificate file to load, and `type` is the format of the private key (i.e. `SSL_FILETYPE_PEM`).

7.3 Certificate Chain Verification

wolfSSL requires that only the top or "root" certificate in a chain to be loaded as a trusted certificate in order to verify a certificate chain. This means that if you have a certificate chain (A -> B -> C), where C is signed by B, and B is signed by A, wolfSSL only requires that certificate A be loaded as a trusted certificate in order to verify the entire chain (A->B->C).

For example, if a server certificate chain looks like:

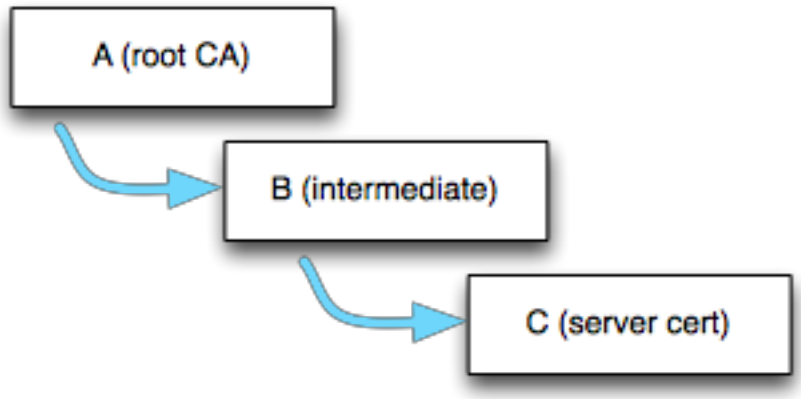

Figure 2: Certificate Chain

The wolfSSL client should already have at least the root cert (A) loaded as a trusted root (with `wolfSSL_CTX_load_verify_locations()`). When the client receives the server cert chain, it uses the signature of A to verify B, and if B has not been previously loaded into wolfSSL as a trusted root, B gets stored in wolfSSL's internal cert chain (wolfSSL just stores what is necessary to verify a certificate: common name hash, public key and key type, etc.). If B is valid, then it is used to verify C.

Following this model, as long as root cert "A" has been loaded as a trusted root into the wolfSSL server, the server certificate chain will still be able to be verified if the server sends (A->B->C), or (B->C). If the server just sends (C), and not the intermediate certificate, the chain will not be able to be verified unless the wolfSSL client has already loaded B as a trusted root.

7.4 Domain Name Check for Server Certificates

wolfSSL has an extension on the client that automatically checks the domain of the server certificate. In OpenSSL mode nearly a dozen function calls are needed to perform this. wolfSSL checks that the date of the certificate is in range, verifies the signature, and additionally verifies the domain if you call `wolfSSL_check_domain_name(WOLFSSL* ssl, const char* dn)` before calling `wolfSSL_connect()`. wolfSSL will match the X.509 issuer name of peer's server certificate against dn (the expected domain name). If the names match `wolfSSL_connect()` will proceed normally, however if there is a name mismatch, `wolfSSL_connect()` will return a fatal error and `wolfSSL_get_error()` will return `DOMAIN_NAME_MISMATCH`.

Checking the domain name of the certificate is an important step that verifies the server is actually who it claims to be. This extension is intended to ease the burden of performing the check.

7.5 No File System and using Certificates

Normally a file system is used to load private keys, certificates, and CAs. Since wolfSSL is sometimes used in environments without a full file system an extension to use memory buffers instead is provided. To use the extension define the constant `NO_FILESYSTEM` and the following functions will be made available:

- `int wolfSSL_CTX_load_verify_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,long sz, int format);`
- `int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, long sz, int format);`
- `int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, long sz, int format);`
- `int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,long sz);`
- `int wolfSSL_CTX_trust_peer_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, Long sz, int format);`

Use these functions exactly like their counterparts that are named `*_file` instead of `*_buffer`. And instead of providing a filename provide a memory buffer. See API documentation for usage details.

7.5.1 Test Certificate and Key Buffers

wolfSSL has come bundled with test certificate and key files in the past. Now it also comes bundled with test certificate and key buffers for use in environments with no filesystem available. These buffers are available in `certs_test.h` when defining one or more of `USE_CERT_BUFFERS_1024`, `USE_CERT_BUFFERS_2048`, or `USE_CERT_BUFFERS_256`.

7.6 Serial Number Retrieval

The serial number of an X.509 certificate can be extracted from wolfSSL using `wolfSSL_X509_get_serial_number()`. The serial number can be of any length.

```
int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509,
            unsigned char* buffer, int* inOutSz)
```

`buffer` will be written to with at most `*inOutSz` bytes on input. After the call, if successful (return of 0), `*inOutSz` will hold the actual number of bytes written to `buffer`. A full example is included `wolfssl/test.h`.

7.7 RSA Key Generation

wolfSSL supports RSA key generation of varying lengths up to 4096 bits. Key generation is off by default but can be turned on during the `./configure` process with `--enable-keygen` or by defining `WOLFSSL_KEY_GEN` in Windows or non-standard environments. Creating a key is easy, only requiring one function from `rsa.h`:

```
int MakeRsaKey(RsaKey* key, int size, long e, RNG* rng);
```

Where `size` is the length in bits and `e` is the public exponent, using 65537 is usually a good choice for `e`. The following from `wolfcrypt/test/test.c` gives an example creating an RSA key of 1024 bits:

```
RsaKey genKey;
RNG    rng;
int    ret;

InitRng(&rng);
InitRsaKey(&genKey, 0);

ret = MakeRsaKey(&genKey, 1024, 65537, &rng);
if (ret != 0)
    /* ret contains error */;
```

The RsaKey genKey can now be used like any other RsaKey. If you need to export the key, wolfSSL provides both DER and PEM formatting in asn.h. Always convert the key to DER format first, and then if you need PEM use the generic `DerToPem()` function like this:

```
byte der[4096];
int  derSz = RsaKeyToDer(&genKey, der, sizeof(der));
if (derSz < 0)
    /* derSz contains error */;
```

The buffer `der` now holds a DER format of the key. To convert the DER buffer to PEM use the conversion function:

```
byte pem[4096];
int  pemSz = DerToPem(der, derSz, pem, sizeof(pem),
                      PRIVATEKEY_TYPE);
if (pemSz < 0)
    /* pemSz contains error */;
```

The last argument of *DerToPem()* takes a type parameter, usually either `PRIVATEKEY_TYPE` or `CERT_TYPE`. Now the buffer `pem` holds the PEM format of the key. Supported types are:

- `CA_TYPE`
- `TRUSTED_PEER_TYPE`
- `CERT_TYPE`
- `CRL_TYPE`
- `DH_PARAM_TYPE`
- `DSA_PARAM_TYPE`
- `CERTREQ_TYPE`
- `DSA_TYPE`
- `DSA_PRIVATEKEY_TYPE`
- `ECC_TYPE`
- `ECC_PRIVATEKEY_TYPE`
- `RSA_TYPE`
- `PRIVATEKEY_TYPE`
- `ED25519_TYPE`
- `EDDSA_PRIVATEKEY_TYPE`
- `PUBLICKEY_TYPE`
- `ECC_PUBLICKEY_TYPE`
- `PKCS8_PRIVATEKEY_TYPE`
- `PKCS8_ENC_PRIVATEKEY_TYPE`

7.7.1 RSA Key Generation Notes

The RSA private key contains the public key as well. The private key can be used as both a private and public key by wolfSSL as used in test.c. The private key and the public key (in the form of a certificate) is all that is typically needed for SSL.

A separate public key can be loaded into wolfSSL manually using the RsaPublicKeyDecode() function if need be. Additionally, the `wc_RsaKeyToPublicDer()` function can be used to export the public RSA key.

7.8 Certificate Generation

wolfSSL supports X.509 v3 certificate generation. Certificate generation is off by default but can be turned on during the `./configure` process with `--enable-certgen` or by defining `WOLFSSL_CERT_GEN` in Windows or non-standard environments.

Before a certificate can be generated the user needs to provide information about the subject of the certificate. This information is contained in a structure from `wolfssl/wolfcrypt/asn_public.h` named `Cert`:

```
/* for user to fill for certificate generation */
typedef struct Cert {
    int      version;                   /* x509 version  */
    byte     serial[CTC_SERIAL_SIZE];   /* serial number */
    int      sigType;                   /*signature algo type */
    CertName issuer;                    /* issuer info */
    int      daysValid;                 /* validity days */
    int      selfSigned;                /* self signed flag */
    CertName subject;                   /* subject info */
    int      isCA;                      /*is this going to be a CA*/
    ...
} Cert;
```

Where CertName looks like:

```
typedef struct CertName {
char country[CTC_NAME_SIZE];
    char countryEnc;
    char state[CTC_NAME_SIZE];
    char stateEnc;
    char locality[CTC_NAME_SIZE];
    char localityEnc;
    char sur[CTC_NAME_SIZE];
    char surEnc;
    char org[CTC_NAME_SIZE];
    char orgEnc;
    char unit[CTC_NAME_SIZE];
    char unitEnc;
    char commonName[CTC_NAME_SIZE];
    char commonNameEnc;
    char email[CTC_NAME_SIZE];  /* !!!! email has to be last!!!! */
} CertName;
```

Before filling in the subject information an initialization function needs to be called like this:

```
Cert myCert;
InitCert(&myCert);
```

`InitCert()` sets defaults for some of the variables including setting the version to **3** (0x02), the serial number to **0** (randomly generated), the sigType to CTC_SHAwRSA, the daysValid to **500**, and selfSigned to **1** (TRUE). Supported signature types include:

- CTC_SHAwDSA
- CTC_MD2wRSA
- CTC_MD5wRSA
- CTC_SHAwRSA
- CTC_SHAwECDSA
- CTC_SHA256wRSA
- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA
- CTC_SHA512wRSA

- CTC_SHA512wECDSA

Now the user can initialize the subject information like this example from wolfcrypt/test/test.c:

```
strncpy(myCert.subject.country, "US", CTC_NAME_SIZE);
strncpy(myCert.subject.state, "OR", CTC_NAME_SIZE);
strncpy(myCert.subject.locality, "Portland", CTC_NAME_SIZE);
strncpy(myCert.subject.org, "yaSSL", CTC_NAME_SIZE);
strncpy(myCert.subject.unit, "Development", CTC_NAME_SIZE);
strncpy(myCert.subject.commonName, "www.wolfssl.com", CTC_NAME_SIZE);
strncpy(myCert.subject.email, "info@wolfssl.com", CTC_NAME_SIZE);
```

Then, a self-signed certificate can be generated using the variables genKey and rng from the above key generation example (of course any valid RsaKey or RNG can be used):

```
byte derCert[4096];

int certSz = MakeSelfCert(&myCert, derCert, sizeof(derCert), &key, &rng);
if (certSz < 0)
  /* certSz contains the error */;
```

The buffer derCert now contains a DER format of the certificate. If you need a PEM format of the certificate you can use the generic DerToPem() function and specify the type to be CERT_TYPE like this:

```
byte* pem;

int pemSz = DerToPem(derCert, certSz, pem, sizeof(pemCert), CERT_TYPE);
if (pemCertSz < 0)
  /* pemCertSz contains error */;
```

Supported types are:

- CA_TYPE
- TRUSTED_PEER_TYPE
- CERT_TYPE
- CRL_TYPE
- DH_PARAM_TYPE
- DSA_PARAM_TYPE
- CERTREQ_TYPE
- DSA_TYPE
- DSA_PRIVATEKEY_TYPE
- ECC_TYPE
- ECC_PRIVATEKEY_TYPE
- RSA_TYPE
- PRIVATEKEY_TYPE
- ED25519_TYPE
- EDDSA_PRIVATEKEY_TYPE
- PUBLICKEY_TYPE
- ECC_PUBLICKEY_TYPE
- PKCS8_PRIVATEKEY_TYPE
- PKCS8_ENC_PRIVATEKEY_TYPE

Now the buffer pemCert< holds the PEM format of the certificate.

If you wish to create a CA signed certificate then a couple of steps are required. After filling in the subject information as before, you'll need to set the issuer information from the CA certificate. This can be done with SetIssuer() like this:

```
ret = SetIssuer(&myCert, "ca-cert.pem");
if (ret < 0)
    /* ret contains error */;
```

Then you'll need to perform the two-step process of creating the certificate and then signing it (MakeSelfCert() does these both in one step). You'll need the private keys from both the issuer (caKey) and the subject (key). Please see the example in test.c for complete usage.

```
byte derCert[4096];

int certSz = MakeCert(&myCert, derCert, sizeof(derCert), &key, NULL, &rng);
if (certSz < 0);
   /*certSz contains the error*/;

certSz = SignCert(myCert.bodySz, myCert.sigType, derCert,
            sizeof(derCert), &caKey, NULL, &rng);
if (certSz < 0);
   /*certSz contains the error*/;
```

The buffer derCert now contains a DER format of the CA signed certificate. If you need a PEM format of the certificate please see the self signed example above. Note that MakeCert() and SignCert() provide function parameters for either an RSA or ECC key to be used. The above example uses an RSA key and passes NULL for the ECC key parameter.

7.9 Certificate Signing Request (CSR) Generation

wolfSSL supports X.509 v3 certificate signing request (CSR) generation. CSR generation is off by default but can be turned on during the ./configure process with --enable-certreq --enable-certgen or by defining WOLFSSL_CERT_GEN **and** WOLFSSL_CERT_REQ in Windows or non-standard environments.

Before a CSR can be generated the user needs to provide information about the subject of the certificate. This information is contained in a structure from wolfssl/wolfcrypt/asn_public.h named Cert:

For details on the Cert and CertName structures please reference Certificate Generation above.

Before filling in the subject information an initialization function needs to be called like this:

```
Cert request;
InitCert(&request);
```

InitCert() sets defaults for some of the variables including setting the version to **3** (0x02), the serial number to **0** (randomly generated), the sigType to CTC_SHAwRSA, the daysValid to **500**, and selfSigned to **1** (TRUE). Supported signature types include:

- CTC_SHAwDSA
- CTC_MD2wRSA
- CTC_MD5wRSA
- CTC_SHAwRSA
- CTC_SHAwECDSA
- CTC_SHA256wRSA
- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA
- CTC_SHA512wRSA
- CTC_SHA512wECDSA

Now the user can initialize the subject information like this example from https://github.com/wolfSSL/wolfssl-examples/blob/master/certgen/csr_example.c:

```
strncpy(req.subject.country, "US", CTC_NAME_SIZE);
strncpy(req.subject.state, "OR", CTC_NAME_SIZE);
strncpy(req.subject.locality, "Portland", CTC_NAME_SIZE);
strncpy(req.subject.org, "wolfSSL", CTC_NAME_SIZE);
strncpy(req.subject.unit, "Development", CTC_NAME_SIZE);
```

```
strncpy(req.subject.commonName, "www.wolfssl.com", CTC_NAME_SIZE);
strncpy(req.subject.email, "info@wolfssl.com", CTC_NAME_SIZE);
```

Then, a valid signed CSR can be generated using the variable key from the above key generation example (of course any valid ECC/RSA key or RNG can be used):

```
byte der[4096]; /* Store request in der format once made */

ret = wc_MakeCertReq(&request, der, sizeof(der), NULL, &key);
/* check ret value for error handling, <= 0 indicates a failure */
```

Next you will want to sign your request making it valid, use the rng variable from the above key generation example. (of course any valid ECC/RSA key or RNG can be used)

```
derSz = ret;

req.sigType = CTC_SHA256wECDSA;
ret = wc_SignCert(request.bodySz, request.sigType, der, sizeof(der), NULL, &key, &rng);
/* check ret value for error handling, <= 0 indicates a failure */
```

Lastly it is time to convert the CSR to PEM format for sending to a CA authority to use in issueing a certificate:

```
ret = wc_DerToPem(der, derSz, pem, sizeof(pem), CERTREQ_TYPE);
/* check ret value for error handling, <= 0 indicates a failure */
printf("%s", pem); /* or write to a file */
```

7.9.1 Limitations

There are fields that are mandatory in a certificate that are excluded in a CSR. There are other fields in a CSR that are also deemed “optional” that are otherwise mandatory when in a certificate. Because of this the wolfSSL certificate parsing engine, which strictly checks all certificate fields AND considers all fields mandatory, does not support consuming a CSR at this time. Therefore while CSR generation AND certificate generation from scratch are supported, wolfSSL does not support certificate generation FROM a CSR. Passing in a CSR to the wolfSSL parsing engine will return a failure at this time. Check back for updates once we support consuming a CSR for use in certificate generation!

See also: Certificate Generation

7.10 Convert to raw ECC key

With our recently added support for raw ECC key import comes the ability to convert an ECC key from PEM to DER. Use the following with the specified arguments to accomplish this:

```
EccKeyToDer(ecc_key*, byte* output, word32 inLen);
```

7.10.1 Example

```
#define FOURK_BUF 4096
byte  der[FOURK_BUF];
ecc_key userB;

EccKeyToDer(&userB, der, FOURK_BUF);
```

8 Debugging

8.1 Debugging and Logging

wolfSSL (formerly CyaSSL) has support for debugging through log messages in environments where debugging is limited. To turn logging on use the function wolfSSL_Debugging_ON() and to turn it off use wolfSSL_Debugging_OFF(). In a normal build (release mode) these functions will have no effect. In a debug build, define DEBUG_WOLFSSL to ensure these functions are turned on.

As of wolfSSL 2.0, logging callback functions may be registered at runtime to provide more flexibility with how logging is done. The logging callback can be registered with the function wolfSSL_SetLoggingCb():

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);

typedef void (*wolfSSL_Logging_cb)(const int logLevel,
                  const char *const logMessage);
```

The log levels can be found in wolfssl/wolfcrypt/logging.h, and the implementation is located in logging.c. By default, wolfSSL logs to stderr with fprintf.

8.2 Error Codes

wolfSSL tries to provide informative error messages in order to help with debugging.

Each wolfSSL_read() and wolfSSL_write() call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like read() and write(). In the event of an error you can use two calls to get more information about the error.

The function wolfSSL_get_error() will return the current error code. It takes the current WOLFSSL object, and wolfSSL_read() or wolfSSL_write() result value as an arguments and returns the corresponding error code.

```
int err = wolfSSL_get_error(ssl, result);
```

To get a more human-readable error code description, the wolfSSL_ERR_error_string() function can be used. It takes the return code from wolfSSL_get_error and a storage buffer as arguments, and places the corresponding error description into the storage buffer (errorString in the example below).

```
char errorString[80];
wolfSSL_ERR_error_string(err, errorString);
```

If you are using non blocking sockets, you can test for errno EAGAIN/EWOULDBLOCK or more correctly you can test the specific error code for SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE.

For a list of wolfSSL and wolfCrypt error codes, please see Appendix C (Error Codes).

9 Library Design

9.1 Library Headers

With the release of wolfSSL 2.0.0 RC3, library header files are now located in the following locations:

- wolfSSL: `wolfssl/`
- wolfCrypt: `wolfssl/wolfcrypt/`
- wolfSSL OpenSSL Compatibility Layer: `wolfssl/openssl/`

When using the OpenSSL Compatibility layer (see OpenSSL Compatibility), the `/wolfssl/openssl/ssl.h` header is required to be included:

```
#include <wolfssl/openssl/ssl.h>
```

When using only the wolfSSL native API, only the `/wolfssl/ssl.h` header is required to be included:

```
#include <wolfssl/ssl.h>
```

9.2 Startup and Exit

All applications should call `wolfSSL_Init()` before using the library and call `wolfSSL_Cleanup()` at program termination. Currently these functions only initialize and free the shared mutex for the session cache in multi-user mode but in the future they may do more so it's always a good idea to use them.

9.3 Structure Usage

In addition to header file location changes, the release of wolfSSL 2.0.0 RC3 created a more visible distinction between the native wolfSSL API and the wolfSSL OpenSSL Compatibility Layer. With this distinction, the main SSL/TLS structures used by the native wolfSSL API have changed names. The new structures are as follows. The previous names are still used when using the OpenSSL Compatibility Layer (see OpenSSL Compatibility).

- `WOLFSSL` (previously SSL)
- `WOLFSSL_CTX` (previously SSL_CTX)
- `WOLFSSL_METHOD` (previously SSL_METHOD)
- `WOLFSSL_SESSION` (previously SSL_SESSION)
- `WOLFSSL_X509` (previously X509)
- `WOLFSSL_X509_NAME` (previously X509_NAME)
- `WOLFSSL_X509_CHAIN` (previously X509_CHAIN)

9.4 Thread Safety

wolfSSL (formerly CyaSSL) is thread safe by design. Multiple threads can enter the library simultaneously without creating conflicts because wolfSSL avoids global data, static data, and the sharing of objects. The user must still take care to avoid potential problems in two areas.

1. A client may share an WOLFSSL object across multiple threads but access must be synchronized, i.e., trying to read/write at the same time from two different threads with the same SSL pointer is not supported.

   wolfSSL could take a more aggressive (constrictive) stance and lock out other users when a function is entered that cannot be shared but this level of granularity seems counter-intuitive. All users (even single threaded ones) will pay for the locking and multi-thread ones won't be able to re-enter the library even if they aren't sharing objects across threads. This penalty seems much too high and wolfSSL leaves the responsibility of synchronizing shared objects in the hands of the user.

2. Besides sharing WOLFSSL pointers, users must also take care to completely initialize an `WOLFSSL_CTX` before passing the structure to `wolfSSL_new()`. The same `WOLFSSL_CTX` can create multiple `WOLFSSL` structs but the `WOLFSSL_CTX` is only read during `wolfSSL_new()` creation and any future (or simultaneous changes) to the `WOLFSSL_CTX` will not be reflected once the `WOLFSSL` object is created.

Again, multiple threads should synchronize writing access to a WOLFSSL_CTX and it is advised that a single thread initialize the WOLFSSL_CTX to avoid the synchronization and update problem described above.

9.5 Input and Output Buffers

wolfSSL now uses dynamic buffers for input and output. They default to 0 bytes and are controlled by the RECORD_SIZE define in wolfssl/internal.h. If an input record is received that is greater in size than the static buffer, then a dynamic buffer is temporarily used to handle the request and then freed. You can set the static buffer size up to the MAX_RECORD_SIZE which is 2^16 or 16,384.

If you prefer the previous way that wolfSSL operated, with 16Kb static buffers that will never need dynamic memory, you can still get that option by defining LARGE_STATIC_BUFFERS.

If dynamic buffers are used and the user requests a wolfSSL_write() that is bigger than the buffer size, then a dynamic block up to MAX_RECORD_SIZE is used to send the data. Users wishing to only send the data in chunks of at most RECORD_SIZE size can do this by defining STATIC_CHUNKS_ONLY. This will cause wolfSSL to use I/O buffers which grow up to RECORD_SIZE, which is 128 bytes by default.

10 wolfCrypt Usage Reference

wolfCrypt is the cryptography library primarily used by wolfSSL. It is optimized for speed, small footprint, and portability. wolfSSL interchanges with other cryptography libraries as required.

Types used in the examples:

```
typedef unsigned char byte;
typedef unsigned int  word32;
```

10.1 Hash Functions

10.1.1 MD4

NOTE: MD4 is outdated and considered insecure. Please consider using a different hashing function if possible.

To use MD4 include the MD4 header `wolfssl/wolfcrypt/md4.h`. The structure to use is `Md4`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitMd4()` call. Use `wc_Md4Update()` to update the hash and `wc_Md4Final()` to retrieve the final hash.

```
byte md4sum[MD4_DIGEST_SIZE];
byte buffer[1024];
/* fill buffer with data to hash*/

Md4 md4;
wc_InitMd4(&md4);

wc_Md4Update(&md4, buffer, sizeof(buffer));  /*can be called again
                                 and again*/
wc_Md4Final(&md4, md4sum);
```

`md4sum` now contains the digest of the hashed data in buffer.

10.1.2 MD5

NOTE: MD5 is outdated and considered insecure. Please consider using a different hashing function if possible.

To use MD5 include the MD5 header `wolfssl/wolfcrypt/md5.h`. The structure to use is `Md5`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitMd5()` call. Use `wc_Md5Update()` to update the hash and `wc_Md5Final()` to retrieve the final hash

```
byte md5sum[MD5_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

Md5 md5;
wc_InitMd5(&md5);

wc_Md5Update(&md5, buffer, sizeof(buffer));  /*can be called again
                                 and again*/
wc_Md5Final(&md5, md5sum);
```

`md5sum` now contains the digest of the hashed data in buffer.

10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512

To use SHA include the SHA header `wolfssl/wolfcrypt/sha.h`. The structure to use is `Sha`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitSha()` call. Use `wc_ShaUpdate()` to update the hash and `wc_ShaFinal()` to retrieve the final hash:

```
byte shaSum[SHA_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

Sha sha;
wc_InitSha(&sha);

wc_ShaUpdate(&sha, buffer, sizeof(buffer));  /*can be called again
                                               and again*/
wc_ShaFinal(&sha, shaSum);
```

shaSum now contains the digest of the hashed data in buffer.

To use either SHA-224, SHA-256, SHA-384, or SHA-512, follow the same steps as shown above, but use either the wolfssl/wolfcrypt/sha256.h or wolfssl/wolfcrypt/sha512.h (for both SHA-384 and SHA-512). The SHA-256, SHA-384, and SHA-512 functions are named similarly to the SHA functions.

For **SHA-224**, the functions wc_InitSha224(), wc_Sha224Update(), and wc_Sha224Final() will be used with the structure Sha224.

For **SHA-256**, the functions wc_InitSha256(), wc_Sha256Update(), and wc_Sha256Final() will be used with the structure Sha256.

For **SHA-384**, the functions InitSha384(), wc_Sha384Update(), and wc_Sha384Final() will be used with the structure Sha384.

For **SHA-512**, the functions wc_InitSha512(), Sha512Update(), and Sha512Final() will be used with the structure Sha512.

10.1.4 BLAKE2b

To use BLAKE2b (a SHA-3 finalist) include the BLAKE2b header wolfssl/wolfcrypt/blake2.h. The structure to use is Blake2b, which is a typedef. Before using, the hash initialization must be done with the wc_InitBlake2b() call. Use wc_Blake2bUpdate() to update the hash and wc_Blake2bFinal() to retrieve the final hash:

```
byte digest[64];
byte input[64];     /*fill input with data to hash*/

Blake2b b2b;
wc_InitBlake2b(&b2b, 64);

wc_Blake2bUpdate(&b2b, input, sizeof(input));
wc_Blake2bFinal(&b2b, digest, 64);
```

The second parameter to wc_InitBlake2b() should be the final digest size. digest now contains the digest of the hashed data in buffer.

Example usage can be found in the wolfCrypt test application (wolfcrypt/test/test.c), inside the blake2b_test() function.

10.1.5 RIPEMD-160

To use RIPEMD-160, include the header wolfssl/wolfcrypt/ripemd.h. The structure to use is RipeMd, which is a typedef. Before using, the hash initialization must be done with the wc_InitRipeMd() call. Use wc_RipeMdUpdate() to update the hash and wc_RipeMdFinal() to retrieve the final hash

```
byte ripeMdSum[RIPEMD_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

RipeMd ripemd;
```

```
wc_InitRipeMd(&ripemd);

wc_RipeMdUpdate(&ripemd, buffer, sizeof(buffer));  /*can be called
                                        again and again*/
wc_RipeMdFinal(&ripemd, ripeMdSum);
```

ripeMdSum now contains the digest of the hashed data in buffer.

10.2 Keyed Hash Functions

10.2.1 HMAC

wolfCrypt currently provides HMAC for message digest needs. The structure Hmac is found in the header wolfssl/wolfcrypt/hmac.h. HMAC initialization is done with wc_HmacSetKey(). 5 different types are supported with HMAC: MD5, SHA, SHA-256, SHA-384, and SHA-512. Here's an example with SHA-256.

```
Hmac    hmac;
byte    key[24];        /*fill key with keying material*/
byte    buffer[2048];   /*fill buffer with data to digest*/
byte    hmacDigest[SHA256_DIGEST_SIZE];

wc_HmacSetKey(&hmac, SHA256, key, sizeof(key));
wc_HmacUpdate(&hmac, buffer, sizeof(buffer));
wc_HmacFinal(&hmac, hmacDigest);
```

hmacDigest now contains the digest of the hashed data in buffer.

10.2.2 GMAC

wolfCrypt also provides GMAC for message digest needs. The structure Gmac is found in the header wolfssl/wolfcrypt/aes.h, as it is an application AES-GCM. GMAC initialization is done with wc_GmacSetKey().

```
Gmac gmac;
byte    key[16];         /*fill key with keying material*/
byte iv[12];         /*fill iv with an initialization vector*/
byte    buffer[2048];   /*fill buffer with data to digest*/
byte    gmacDigest[16];

wc_GmacSetKey(&gmac, key, sizeof(key));
wc_GmacUpdate(&gmac, iv, sizeof(iv), buffer, sizeof(buffer),
gmacDigest, sizeof(gmacDigest));
```

gmacDigest now contains the digest of the hashed data in buffer.

10.2.3 Poly1305

wolfCrypt also provides Poly1305 for message digest needs. The structure Poly1305 is found in the header wolfssl/wolfcrypt/poly1305.h. Poly1305 initialization is done with wc_Poly1305SetKey(). The process of setting a key in Poly1305 should be done again, with a new key, when next using Poly1305 after wc_Poly1305Final() has been called.

```
Poly1305    pmac;
byte           key[32];        /*fill key with keying material*/
byte           buffer[2048];   /*fill buffer with data to digest*/
byte           pmacDigest[16];

wc_Poly1305SetKey(&pmac, key, sizeof(key));
wc_Poly1305Update(&pmac, buffer, sizeof(buffer));
wc_Poly1305Final(&pmac, pmacDigest);
```

pmacDigest now contains the digest of the hashed data in buffer.

10.3 Block Ciphers

10.3.1 AES

wolfCrypt provides support for AES with key sizes of 16 bytes (128 bits), 24 bytes (192 bits), or 32 bytes (256 bits). Supported AES modes include CBC, CTR, GCM, and CCM-8.

CBC mode is supported for both encryption and decryption and is provided through the wc_AesSetKey(), wc_AesCbcEncrypt() and wc_AesCbcDecrypt() functions. Please include the header wolfssl/wolfcrypt/aes.h to use AES. AES has a block size of 16 bytes and the IV should also be 16 bytes. Function usage is usually as follows:

```
Aes enc;
Aes dec;

const byte key[] = {  /*some 24 byte key*/ };
const byte iv[] = { /*some 16 byte iv*/ };

byte plain[32];   /*an increment of 16, fill with data*/
byte cipher[32];

/*encrypt*/
wc_AesSetKey(&enc, key, sizeof(key), iv, AES_ENCRYPTION);
wc_AesCbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_AesSetKey(&dec, key, sizeof(key), iv, AES_DECRYPTION);
wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

wolfCrypt also supports CTR (Counter), GCM (Galois/Counter), and CCM-8 (Counter with CBC-MAC) modes of operation for AES. When using these modes, like CBC, include the wolfssl/wolfcrypt/aes.h header.

GCM mode is available for both encryption and decryption through the wc_AesGcmSetKey(), wc_AesGcmEncrypt(), and wc_AesGcmDecrypt() functions. For a usage example, see the aesgcm_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

CCM-8 mode is supported for both encryption and decryption through the wc_AesCcmSetKey(), wc_AesCcmEncrypt(), and wc_AesCcmDecrypt() functions. For a usage example, see the aesccm_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

CTR mode is available for both encryption and decryption through the wc_AesCtrEncrypt() function. The encrypt and decrypt actions are identical so the same function is used for both. For a usage example, see the function aes_test() in file wolfcrypt/test/test.c.

10.3.1.1 DES and 3DES wolfCrypt provides support for DES and 3DES (Des3 since 3 is an invalid leading C identifier). To use these include the header wolfssl/wolfcrypt/des.h. The structures you can use are Des and Des3. Initialization is done through wc_Des_SetKey() or wc_Des3_SetKey(). CBC encryption/decryption is provided through wc_Des_CbcEnrypt() / wc_Des_CbcDecrypt() and wc_Des3_CbcEncrypt() / wc_Des3_CbcDecrypt(). Des has a key size of 8 bytes (24 for 3DES) and the block size is 8 bytes, so only pass increments of 8 bytes to encrypt/decrypt functions. If your data isn't in a block size increment you'll need to add padding to make sure it is. Each SetKey() also takes an IV (an initialization vector that is the same size as the key size). Usage is usually like the following:

```
Des3 enc;
Des3 dec;

const byte key[] = {  /*some 24 byte key*/ };
```

```
const byte iv[]  = {  /*some 24 byte iv*/  };

byte plain[24];   /*an increment of 8, fill with data*/
byte cipher[24];

/*encrypt*/
wc_Des3_SetKey(&enc, key, iv, DES_ENCRYPTION);
wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
wc_Des3_CbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.3.1.2 Camellia wolfCrypt provides support for the Camellia block cipher. To use Camellia include the header wolfssl/wolfcrypt/camellia.h. The structure you can use is called Camellia. Initialization is done through wc_CamelliaSetKey(). CBC encryption/decryption is provided through wc_CamelliaCbcEnrypt() and wc_CamelliaCbcDecrypt() while direct encryption/decryption is provided through wc_CamelliaEncryptDirect() and wc_CamelliaDecryptDirect().

For usage examples please see the camellia_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

10.4 Stream Ciphers

10.4.1 ARC4

NOTE: ARC4 is outdated and considered insecure. Please consider using a different stream cipher.

The most common stream cipher used on the Internet is ARC4. wolfCrypt supports it through the header wolfssl/wolfcrypt/arc4.h. Usage is simpler than block ciphers because there is no block size and the key length can be any length. The following is a typical usage of ARC4.

```
Arc4 enc;
Arc4 dec;

const byte key[] = {  /*some key any length*/};

byte plain[27];   /*no size restriction, fill with data*/
byte cipher[27];

/*encrypt*/
wc_Arc4SetKey(&enc, key, sizeof(key));
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Arc4SetKey(&dec, key, sizeof(key));
wc_Arc4Process(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.4.2 RABBIT

A newer stream cipher gaining popularity is RABBIT. This stream cipher can be used through wolfCrypt by including the header wolfssl/wolfcrypt/rabbit.h. RABBIT is very fast compared to ARC4, but has key constraints of 16 bytes (128 bits) and an optional IV of 8 bytes (64 bits). Otherwise usage is exactly like ARC4:

```
Rabbit enc;
Rabbit dec;

const byte key[] = {  /*some key 16 bytes*/};
const byte iv[]  = {  /*some iv 8 bytes*/  };

byte plain[27];   /*no size restriction, fill with data*/
byte cipher[27];

/*encrypt*/
wc_RabbitSetKey(&enc, key, iv);     /*iv can be a NULL pointer*/
wc_RabbitProcess(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_RabbitSetKey(&dec, key, iv);
wc_RabbitProcess(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.4.3 HC-128

Another stream cipher in current use is HC-128, which is even faster than RABBIT (about 5 times faster than ARC4). To use it with wolfCrypt, please include the header wolfssl/wolfcrypt/hc128.h. HC-128 also uses 16-byte keys (128 bits) but uses 16-byte IVs (128 bits) unlike RABBIT.

```
HC128 enc;
HC128 dec;

const byte key[] = {  /*some key 16 bytes*/};
const byte iv[]  = {  /*some iv 16 bytes*/ };

byte plain[37];   /*no size restriction, fill with data*/
byte cipher[37];

/*encrypt*/
wc_Hc128_SetKey(&enc, key, iv);     /*iv can be a NULL pointer*/
wc_Hc128_Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Hc128_SetKey(&dec, key, iv);
wc_Hc128_Process(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.4.4 ChaCha

ChaCha with 20 rounds is slightly faster than ARC4 while maintaining a high level of security. To use it with wolfCrypt, please include the header wolfssl/wolfcrypt/chacha.h. ChaCha typically uses 32 byte keys (256 bit) but can also use 16 byte keys (128 bits).

```
CHACHA enc;
CHACHA dec;

const byte key[] = {  /*some key 32 bytes*/};
const byte iv[]  = {  /*some iv 12 bytes*/ };
```

```
byte plain[37];   /*no size restriction, fill with data*/
byte cipher[37];

/*encrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter); /*counter is the start block
                        counter is usually set as 0*/
wc_Chacha_Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter);
wc_Chacha_Process(&enc, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

wc_Chacha_SetKey only needs to be set once but for each packet of information sent wc_Chacha_SetIV() must be called with a new iv (nonce). Counter is set as an argument to allow for partially decrypting/encrypting information by starting at a different block when performing the encrypt/decrypt process, but in most cases is set to 0. **ChaCha should not be used without a mac algorithm (e.g. Poly1305 , hmac).**

10.5 Public Key Cryptography

10.5.1 RSA

wolfCrypt provides support for RSA through the header wolfssl/wolfcrypt/rsa.h. There are two types of RSA keys, public and private. A public key allows anyone to encrypt something that only the holder of the private key can decrypt. It also allows the private key holder to sign something and anyone with a public key can verify that only the private key holder actually signed it. Usage is usually like the following:

```
RsaKey rsaPublicKey;

byte publicKeyBuffer[]  = { /*holds the raw data from the key, maybe
                        from a file like RsaPublicKey.der*/ };
word32 idx = 0;            /*where to start reading into the buffer*/

RsaPublicKeyDecode(publicKeyBuffer, &idx, &rsaPublicKey, sizeof(publicKeyBuffer));

byte in[] = { /*plain text to encrypt*/ };
byte out[128];
RNG rng;

wc_InitRng(&rng);

word32 outLen = RsaPublicEncrypt(in, sizeof(in), out, sizeof(out), &rsaPublicKey, &rng);
```

Now out holds the ciphertext from the plain text in. wc_RsaPublicEncrypt() will return the length in bytes written to out or a negative number in case of an error. wc_RsaPublicEncrypt() needs a RNG (Random Number Generator) for the padding used by the encryptor and it must be initialized before it can be used. To make sure that the output buffer is large enough to pass you can first call wc_RsaEncryptSize() which will return the number of bytes that a successful call to wc_RsaPublicEnrypt() will write.

In the event of an error, a negative return from wc_RsaPublicEnrypt(), or wc_RsaPublicKeyDecode() for that matter, you can call wc_ErrorString() to get a string describing the error that occurred.

```
void wc_ErrorString(int error, char* buffer);
```

Make sure that buffer is at least MAX_ERROR_SZ bytes (80).

Now to decrypt out:

```
RsaKey rsaPrivateKey;

byte privateKeyBuffer[] = { /*hold the raw data from the key, maybe
                             from a file like RsaPrivateKey.der*/ };
word32 idx = 0;            /*where to start reading into the buffer*/

wc_RsaPrivateKeyDecode(privateKeyBuffer, &idx, &rsaPrivateKey,
                       sizeof(privateKeyBuffer));

byte plain[128];
word32 plainSz = wc_RsaPrivateDecrypt(out, outLen, plain,
                       sizeof(plain), &rsaPrivateKey);
```

Now plain will hold plainSz bytes or an error code. For complete examples of each type in wolfCrypt please see the file wolfcrypt/test/test.c. Note that the wc_RsaPrivateKeyDecode function only accepts keys in raw DER format.

10.5.2 DH (Diffie-Hellman)

wolfCrypt provides support for Diffie-Hellman through the header wolfssl/wolfrypt/dh.h. The Diffie-Hellman key exchange algorithm allows two parties to establish a shared secret key. Usage is usually similar to the following example, where **sideA** and **sideB** designate the two parties.

In the following example, dhPublicKey contains the Diffie-Hellman public parameters signed by a Certificate Authority (or self-signed). privA holds the generated private key for sideA, pubA holds the generated public key for sideA, and agreeA holds the mutual key that both sides have agreed on.

```
DhKey   dhPublicKey;
word32 idx = 0;  /*where to start reading into the
                   publicKeyBuffer*/
word32 pubASz, pubBSz, agreeASz;
byte   tmp[1024];
RNG    rng;

byte privA[128];
byte pubA[128];
byte agreeA[128];

wc_InitDhKey(&dhPublicKey);

byte publicKeyBuffer[] = { /*holds the raw data from the public key
                             parameters, maybe from a file like
                             dh1024.der*/ }
wc_DhKeyDecode(tmp, &idx, &dhPublicKey, publicKeyBuffer);
wc_InitRng(&rng);  /*Initialize random number generator*/
```

wc_DhGenerateKeyPair() will generate a public and private DH key based on the initial public parameters in dhPublicKey.

```
wc_DhGenerateKeyPair(&dhPublicKey, &rng, privA, &privASz,
            pubA, &pubASz);
```

After sideB sends their public key (pubB) to sideA, sideA can then generate the mutually-agreed key(agreeA) using the wc_DhAgree() function.

```
wc_DhAgree(&dhPublicKey, agreeA, &agreeASz, privA, privASz,
           pubB, pubBSz);
```

Now, agreeA holds sideA's mutually-generated key (of size agreeASz bytes). The same process will have been done on sideB.

For a complete example of Diffie-Hellman in wolfCrypt, see the file wolfcrypt/test/test.c.

10.5.3 EDH (Ephemeral Diffie-Hellman)

A wolfSSL server can do Ephemeral Diffie-Hellman. No build changes are needed to add this feature, though an application will have to register the ephemeral group parameters on the server side to enable the EDH cipher suites. A new API can be used to do this:

```
int wolfSSL_SetTmpDH(WOLFSSL* ssl, unsigned char* p,
                     int pSz,unsigned char* g,int gSz);
```

The example server and echoserver use this function from SetDH().

10.5.4 DSA (Digital Signature Algorithm)

wolfCrypt provides support for DSA and DSS through the header wolfssl/wolfcrypt/dsa.h. DSA allows for the creation of a digital signature based on a given data hash. DSA uses the SHA hash algorithm to generate a hash of a block of data, then signs that hash using the signer's private key. Standard usage is similar to the following.

We first declare our DSA key structure (key), initialize our initial message (message) to be signed, and initialize our DSA key buffer (dsaKeyBuffer).

```
DsaKey key;
Byte   message[]      = { /*message data to sign*/ }
byte   dsaKeyBuffer[] = { /*holds the raw data from the DSA key,
                            maybe from a file like dsa512.der*/ }
```

We then declare our SHA structure (sha), random number generator (rng), array to store our SHA hash (hash), array to store our signature (signature), idx (to mark where to start reading in our dsaKeyBuffer), and an int (answer) to hold our return value after verification.

```
Sha    sha;
RNG    rng;
byte   hash[SHA_DIGEST_SIZE];
byte   signature[40];
word32 idx = 0;
int    answer;
```

Set up and create the SHA hash. For more information on wolfCrypt's SHA algorithm, see SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512. The SHA hash of message is stored in the variable hash.

```
wc_InitSha(&sha);
wc_ShaUpdate(&sha, message, sizeof(message));
wc_ShaFinal(&sha, hash);
```

Initialize the DSA key structure, populate the structure key value, and initialize the random number generator (rng).

```
wc_InitDsaKey(&key);
wc_DsaPrivateKeyDecode(dsaKeyBuffer, &idx, &key,
                       sizeof(dsaKeyBuffer));
wc_InitRng(&rng);
```

The wc_DsaSign() function creates a signature (signature) using the DSA private key, hash value, and random number generator.

```
wc_DsaSign(hash, signature, &key, &rng);
```

To verify the signature, use wc_DsaVerify(). If verification is successful, answer will be equal to "**1**". Once finished, free the DSA key structure using wc_FreeDsaKey().

```
wc_DsaVerify(hash, signature, &key, &answer);
wc_FreeDsaKey(&key);
```

11 SSL Tutorial

11.1 Introduction

The wolfSSL (formerly CyaSSL) embedded SSL library can easily be integrated into your existing application or device to provide enhanced communication security through the addition of SSL and TLS. wolfSSL has been targeted at embedded and RTOS environments, and as such, offers a minimal footprint while maintaining excellent performance. Minimum build sizes for wolfSSL range between 20-100kB depending on the selected build options and platform being used.

The goal of this tutorial is to walk through the integration of SSL and TLS into a simple application. Hopefully the process of going through this tutorial will also lead to a better understanding of SSL in general. This tutorial uses wolfSSL in conjunction with simple echoserver and echoclient examples to keep things as simple as possible while still demonstrating the general procedure of adding SSL support to an application. The echoserver and echoclient examples have been taken from the popular book titled Unix Network Programming, Volume 1, 3rd Edition by Richard Stevens, Bill Fenner, and Andrew Rudoff.

This tutorial assumes that the reader is comfortable with editing and compiling C code using the GNU GCC compiler as well as familiar with the concepts of public key encryption. Please note that access to the Unix Network Programming book is not required for this tutorial.

11.1.1 Examples Used in this Tutorial

- echoclient - Figure 5.4, Page 124
- echoserver - Figure 5.12, Page 139

11.2 Quick Summary of SSL/TLS

TLS (Transport Layer Security) and **SSL** (Secure Sockets Layer) are cryptographic protocols that allow for secure communication across a number of different transport protocols. The primary transport protocol used is TCP/IP. The most recent version of SSL/TLS is TLS 1.3. wolfSSL supports SSL 3.0, TLS 1.0, 1.1, 1.2, 1.3 in addition to DTLS 1.0 and 1.2.

SSL and TLS sit between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the underlying transport medium. Application protocols are layered on top of SSL and can include protocols such as HTTP, FTP, and SMTP. A diagram of how SSL fits into the OSI model, as well as a simple diagram of the SSL handshake process can be found in Appendix A.

11.3 Getting the Source Code

All of the source code used in this tutorial can be downloaded from the wolfSSL website, specifically from the following location. The download contains both the original and completed source code for both the echoserver and echoclient used in this tutorial. Specific contents are listed below the link.

https://www.wolfssl.com/documentation/ssl-tutorial-2.3.zip

The downloaded ZIP file has the following structure:

```
/finished_src
    /echoclient (Completed echoclient code)
    /echoserver (Completed echoserver code)
    /include    (Modified unp.h)
    /lib    (Library functions)
/original_src
    /echoclient (Starting echoclient code)
    /echoserver (Starting echoserver code)
    /include    (Modified unp.h)
    /lib    (Library functions)
README
```

11.4 Base Example Modifications

This tutorial, and the source code that accompanies it, have been designed to be as portable as possible across platforms. Because of this, and because we want to focus on how to add SSL and TLS into an application, the base examples have been kept as simple as possible. Several modifications have been made to the examples taken from Unix Network Programming in order to either remove unnecessary complexity or increase the range of platforms supported. If you believe there is something we could do to increase the portability of this tutorial, please let us know at support@wolfssl.com.

The following is a list of modifications that were made to the original echoserver and echoclient examples found in the above listed book.

11.4.1 Modifications to the echoserver (tcpserv04.c)

- Removed call to the `Fork()` function because `fork()` is not supported by Windows. The result of this is an echoserver which only accepts one client simultaneously. Along with this removal, Signal handling was removed.
- Moved `str_echo()` function from `str_echo.c` file into `tcpserv04.c` file
- Added a printf statement to view the client address and the port we have connected through:

```
printf("Connection from %s, port %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
        ntohs(cliaddr.sin_port));
```

- Added a call to `setsockopt()` after creating the listening socket to eliminate the "Address already in use" bind error.
- Minor adjustments to clean up newer compiler warnings

11.4.2 Modifications to the echoclient (tcpcli01.c)

- Moved `str_cli()` function from `str_cli.c` file into `tcpcli01.c` file
- Minor adjustments to clean up newer compiler warnings

11.4.3 Modifications to unp.h header

- This header was simplified to contain only what is needed for this example.

Please note that in these source code examples, certain functions will be capitalized. For example, `Fputs()` and `Writen()`. The authors of Unix Network Programming have written custom wrapper functions for normal functions in order to cleanly handle error checking. For a more thorough explanation of this, please see **Section 1.4** (page 11) in the *Unix Network Programming* book.

11.5 Building and Installing wolfSSL

Before we begin, download the example code (`echoserver` and `echoclient`) from the Getting the Source Code section, above. This section will explain how to download, configure, and install the wolfSSL embedded SSL library on your system.

You will need to download and install the most recent version of wolfSSL from the wolfSSL download page.

For a full list of available build options, see the Building wolfSSL guide. wolfSSL was written with portability in mind, and should generally be easy to build on most systems. If you have difficulty building wolfSSL, please feel free to ask for support on the wolfSSLproduct support forums.

When building wolfSSL on Linux, *BSD, OS X, Solaris, or other* nix like systems, you can use the autoconf system. For Windows-specific instructions, please refer to the Building wolfSSL section of the wolfSSL Manual. To configure and build wolfSSL, run the following two commands from the terminal. Any desired build options may be appended to `./configure` (ex: `./configure --enable-opensslextra`):

```
./configure
make
```

To install wolfSSL, run:

```
sudo make install
```

This will install wolfSSL headers into `/usr/local/include/wolfssl` and the wolfSSL libraries into `/usr/local/lib` on your system. To test the build, run the testsuite application from the wolfSSL root directory:

```
./testsuite/testsuite.test
```

A set of tests will be run on wolfCrypt and wolfSSL to verify it has been installed correctly. After a successful run of the testsuite application, you should see output similar to the following:

```
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
GMAC     test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES      test passed!
AES-GCM  test passed!
RANDOM   test passed!
RSA      test passed!
DH       test passed!
ECC      test passed!
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
Client message: hello wolfssl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers = DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:ECDHE-RSA-AES256-SHA:ECDHE-
    ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-RSA-
    AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-
    SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-SHA256:
    ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-
    CHACHA20-POLY1305:ECDHE-ECDSA-CHACHA20-POLY1305:DHE-RSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-
    POLY1305-OLD:ECDHE-ECDSA-CHACHA20-POLY1305-OLD:DHE-RSA-CHACHA20-POLY1305-OLD
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  input
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  /tmp/output-N0Xq9c

All tests passed!
```

Now that wolfSSL has been installed, we can begin modifying the example code to add SSL functionality. We will first begin by adding SSL to the echoclient and subsequently move on to the echoserver.

11.6 Initial Compilation

To compile and run the example echoclient and echoserver code from the SSL Tutorial source bundle, you can use the included Makefiles. Change directory (cd) to either the echoclient or echoserver directory and run:

```
make
```

This will compile the example code and produce an executable named either echoserver or echoclient depending on which one is being built. The GCC command which is used in the Makefile can be seen below. If you want to build one of the examples without using the supplied Makefile, change directory to the example directory and replace `tcpcli01.c` (echoclient) or `tcpserv04.c` (echoserver) in the following command with correct source file for the example:

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include
```

This will compile the current example into an executable, creating either an "echoserver" or "echoclient" application. To run one of the examples after it has been compiled, change your current directory to the desired example directory and start the application. For example, to start the echoserver use:

```
./echoserver
```

You may open a second terminal window to test the echoclient on your local host and you will need to supply the IP address of the server when starting the application, which in our case will be 127.0.0.1. Change your current directory to the "echoclient" directory and run the following command. Note that the echoserver must already be running:

```
./echoclient 127.0.0.1
```

Once you have both the echoserver and echoclient running, the echoserver should echo back any input that it receives from the echoclient. To exit either the echoserver or echoclient, use *Ctrl + C* to quit the application. Currently, the data being echoed back and forth between these two examples is being sent in the clear - easily allowing anyone with a little bit of skill to inject themselves in between the client and server and listen to your communication.

11.7 Libraries

The wolfSSL library, once compiled, is named `libwolfssl`, and unless otherwise configured the wolfSSL build and install process creates only a shared library under the following directory. Both shared and static libraries may be enabled or disabled by using the appropriate build options:

```
/usr/local/lib
```

The first step we need to do is link the wolfSSL library to our example applications. Modifying the GCC command (using the echoserver as an example), gives us the following new command. Since wolfSSL installs header files and libraries in standard locations, the compiler should be able to find them without explicit instructions (using `-l` or `-L`). Note that by using `-lwolfssl` the compiler will automatically choose the correct type of library (static or shared):

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include -lm -lwolfssl
```

11.8 Headers

The first thing we will need to do is include the wolfSSL native API header in both the client and the server. In the `tcpcli01.c` file for the client and the tcpserv04.c file for the server add the following line near the top:

```
#include <wolfssl/ssl.h>
```

11.9 Startup/Shutdown

Before we can use wolfSSL in our code, we need to initialize the library and the `WOLFSSL_CTX`. wolfSSL is initialized by calling `wolfSSL_Init()`. This must be done first before anything else can be done with the library.

The `WOLFSSL_CTX` structure (wolfSSL Context) contains global values for each SSL connection, including certificate information. A single `WOLFSSL_CTX` can be used with any number of `WOLFSSL` objects created. This allows us to load certain information, such as a list of trusted CA certificates only once.

To create a new WOLFSSL_CTX, use wolfSSL_CTX_new(). This function requires an argument which defines the SSL or TLS protocol for the client or server to use. There are several options for selecting the desired protocol. wolfSSL currently supports SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, DTLS 1.0, and DTLS 1.2. Each of these protocols have a corresponding function that can be used as an argument to wolfSSL_CTX_new(). The possible client and server protocol options are shown below. SSL 2.0 is not supported by wolfSSL because it has been insecure for several years.

EchoClient:

- wolfSSLv3_client_method(); - SSL 3.0
- wolfTLSv1_client_method(); - TLS 1.0
- wolfTLSv1_1_client_method(); - TLS 1.1
- wolfTLSv1_2_client_method(); - TLS 1.2
- wolfSSLv23_client_method(); - Use highest version possible from SSLv3 - TLS 1.2
- wolfDTLSv1_client_method(); - DTLS 1.0
- wolfDTLSv1_2_client_method_ex(); - DTLS 1.2

EchoServer:

- wolfSSLv3_server_method(); - SSLv3
- wolfTLSv1_server_method(); - TLSv1
- wolfTLSv1_1_server_method(); - TLSv1.1
- wolfTLSv1_2_server_method(); - TLSv1.2
- wolfSSLv23_server_method(); - Allow clients to connect with TLSv1+
- wolfDTLSv1_server_method(); - DTLS
- wolfDTLSv1_2_server_method(); - DTLS 1.2

We need to load our CA (Certificate Authority) certificate into the WOLFSSL_CTX so that the when the echoclient connects to the echoserver, it is able to verify the server's identity. To load the CA certificates into the WOLFSSL_CTX, use wolfSSL_CTX_load_verify_locations(). This function requires three arguments: a WOLFSSL_CTX pointer, a certificate file, and a path value. The path value points to a directory which should contain CA certificates in PEM format. When looking up certificates, wolfSSL will look at the certificate file value before looking in the path location. In this case, we don't need to specify a certificate path because we will specify one CA file - as such we use the value 0 for the path argument. The wolfSSL_CTX_load_verify_locations function returns either SSL_SUCCESS or SSL_FAILURE:

```
wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file, const char* path)
```

Putting these things together (library initialization, protocol selection, and CA certificate), we have the following. Here, we choose to use TLS 1.2:

EchoClient:

```
WOLFSSL_CTX* ctx;

wolfSSL_Init();/* Initialize wolfSSL */

/* Create the WOLFSSL_CTX */
if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method())) == NULL){
     fprintf(stderr, "wolfSSL_CTX_new error.\n");
     exit(EXIT_FAILURE);
}

/* Load CA certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_load_verify_locations(ctx,"../certs/ca-cert.pem",0) !=
    SSL_SUCCESS) {
    fprintf(stderr, "Error loading ../certs/ca-cert.pem, please check
            the file.\n");
    exit(EXIT_FAILURE);
}
```

EchoServer:

When loading certificates into the WOLFSSL_CTX, the server certificate and key file should be loaded in addition to the CA certificate. This will allow the server to send the client its certificate for identification verification:

```
  WOLFSSL_CTX* ctx;

  wolfSSL_Init();  /* Initialize wolfSSL */

  /* Create the WOLFSSL_CTX */
  if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method())) == NULL){
       fprintf(stderr, "wolfSSL_CTX_new error.\n");
       exit(EXIT_FAILURE);
  }

  /* Load CA certificates into CYASSL_CTX */
  if (wolfSSL_CTX_load_verify_locations(ctx, "../certs/ca-cert.pem", 0) !=
           SSL_SUCCESS) {
       fprintf(stderr, "Error loading ../certs/ca-cert.pem, "
           "please check the file.\n");
       exit(EXIT_FAILURE);
  }

/* Load server certificates into WOLFSSL_CTX */
  if (wolfSSL_CTX_use_certificate_file(ctx,"../certs/server-cert.pem",
        SSL_FILETYPE_PEM) != SSL_SUCCESS){
      fprintf(stderr, "Error loading ../certs/server-cert.pem, please
        check the file.\n");
      exit(EXIT_FAILURE);
  }

  /* Load keys */
  if (wolfSSL_CTX_use_PrivateKey_file(ctx,"../certs/server-key.pem",
        SSL_FILETYPE_PEM) != SSL_SUCCESS){
      fprintf(stderr, "Error loading ../certs/server-key.pem, please check
        the file.\n");
      exit(EXIT_FAILURE);
  }
```

The code shown above should be added to the beginning of `tcpcli01.c` and `tcpserv04.c`, after both the variable definitions and the check that the user has started the client with an IP address (client). A version of the finished code is included in the SSL tutorial ZIP file for reference.

Now that wolfSSL and the WOLFSSL_CTX have been initialized, make sure that the WOLFSSL_CTX object and the wolfSSL library are freed when the application is completely done using SSL/TLS. In both the client and the server, the following two lines should be placed at the end of the `main()` function (in the client right before the call to `exit()`):

```
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

11.10 WOLFSSL Object

11.10.1 EchoClient

A WOLFSSL object needs to be created after each TCP Connect and the socket file descriptor needs to be associated with the session. In the echoclient example, we will do this after the call to `Connect()`, shown below:

```
/* Connect to socket file descriptor */
Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
```

Directly after connecting, create a new WOLFSSL object using the wolfSSL_new() function. This function returns a pointer to the WOLFSSL object if successful or NULL in the case of failure. We can then associate the socket file descriptor (sockfd) with the new WOLFSSL object (ssl):

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if( (ssl = wolfSSL_new(ctx)) == NULL) {
    fprintf(stderr, "wolfSSL_new error.\n");
    exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, sockfd);
```

One thing to notice here is that we haven't made a call to the wolfSSL_connect() function. wolfSSL_connect() initiates the SSL/TLS handshake with the server, and is called during wolfSSL_read() if it hadn't been called previously. In our case, we don't explicitly call wolfSSL_connect(), as we let our first wolfSSL_read() do it for us.

11.10.2 EchoServer

At the end of the for loop in the main method, insert the WOLFSSL object and associate the socket file descriptor (connfd) with the WOLFSSL object (ssl), just as with the client:

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if ( (ssl = wolfSSL_new(ctx)) == NULL) {
            fprintf(stderr, "wolfSSL_new error.\n");
            exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, connfd);
```

A WOLFSSL object needs to be created after each TCP Connect and the socket file descriptor needs to be associated with the session.

Create a new WOLFSSL object using the wolfSSL_new() function. This function returns a pointer to the WOLFSSL object if successful or NULL in the case of failure. We can then associate the socket file descriptor (sockfd) with the new WOLFSSL object (ssl):

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if( (ssl = wolfSSL_new(ctx)) == NULL) {
    fprintf(stderr, "wolfSSL_new error.\n");
    exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, sockfd);
```

11.11 Sending/Receiving Data

11.11.1 EchoClient

The next step is to begin sending data securely. Take note that in the echoclient example, the `main()` function hands off the sending and receiving work to `str_cli()`. The `str_cli()` function is where our function replacements will be made. First we need access to our `WOLFSSL` object in the `str_cli()` function, so we add another argument and pass the ssl variable to `str_cli()`. Because the `WOLFSSL` object is now going to be used inside of the `str_cli()` function, we remove the `sockfd` parameter. The new `str_cli()` function signature after this modification is shown below:

```
void str_cli(FILE *fp, WOLFSSL* ssl)
```

In the `main()` function, the new argument (`ssl`) is passed to `str_cli()`:

```
str_cli(stdin, ssl);
```

Inside the `str_cli()` function, `Writen()` and `Readline()` are replaced with calls to `wolfSSL_write()` and `wolfSSL_read()` functions, and the `WOLFSSL` object (`ssl`) is used instead of the original file descriptor(`sockfd`). The new `str_cli()` function is shown below. Notice that we now need to check if our calls to `wolfSSL_write` and `wolfSSL_read` were successful.

The authors of the Unix Programming book wrote error checking into their `Writen()` function which we must make up for after it has been replaced. We add a new int variable, n, to monitor the return value of `wolfSSL_read` and before printing out the contents of the buffer, recvline, the end of our read data is marked with a `\0`:

```
void
str_cli(FILE *fp, WOLFSSL* ssl)
{
    char  sendline[MAXLINE], recvline[MAXLINE];
    int   n = 0;

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        if(wolfSSL_write(ssl, sendline, strlen(sendline)) !=
                    strlen(sendline)){
            err_sys("wolfSSL_write failed");
        }

        if ((n = wolfSSL_read(ssl, recvline, MAXLINE)) <= 0)
            err_quit("wolfSSL_read error");

        recvline[n] = '\0';
        Fputs(recvline, stdout);
    }
}
```

The last thing to do is free the `WOLFSSL` object when we are completely done with it. In the `main()` function, right before the line to free the `WOLFSSL_CTX`, call to `wolfSSL_free()`:

```
str_cli(stdin, ssl);

wolfSSL_free(ssl);      /* Free WOLFSSL object */
wolfSSL_CTX_free(ctx);  /* Free WOLFSSL_CTX object */
wolfSSL_Cleanup();      /* Free wolfSSL */
```

11.11.2 EchoServer

The echo server makes a call to `str_echo()` to handle reading and writing (whereas the client made a call to `str_cli()`). As with the client, modify `str_echo()` by replacing the sockfd parameter with a `WOLFSSL` object (`ssl`) parameter in the function

signature:

```
void str_echo(WOLFSSL* ssl)
```

Replace the calls to `Read()` and `Writen()` with calls to the `wolfSSL_read()` and `wolfSSL_write()` functions. The modified `str_echo()` function, including error checking of return values, is shown below. Note that the type of the variable n has been changed from `ssize_t` to `int` in order to accommodate for the change from `read()` to `wolfSSL_read()`:

```
void
str_echo(WOLFSSL* ssl)
{
    int n;
    char buf[MAXLINE];

    while ( (n = wolfSSL_read(ssl, buf, MAXLINE)) > 0) {
        if(wolfSSL_write(ssl, buf, n) != n) {
            err_sys("wolfSSL_write failed");
        }
    }

    if( n < 0 )
        printf("wolfSSL_read error = %d\n", wolfSSL_get_error(ssl,n));
    else if( n == 0 )
        printf("The peer has closed the connection.\n");
}
```

In `main()` call the `str_echo()` function at the end of the for loop (soon to be changed to a while loop). After this function, inside the loop, make calls to free the `WOLFSSL` object and close the connfd socket:

```
str_echo(ssl);                  /* process the request */

wolfSSL_free(ssl);              /* Free WOLFSSL object */
Close(connfd);
```

We will free the `ctx` and cleanup before the call to exit.

11.12 Signal Handling

11.12.1 Echoclient / Echoserver

In the echoclient and echoserver, we will need to add a signal handler for when the user closes the app by using “Ctrl+C”. The echo server is continually running in a loop. Because of this, we need to provide a way to break that loop when the user presses “Ctrl+C”. To do this, the first thing we need to do is change our loop to a while loop which terminates when an exit variable (cleanup) is set to true.

First, define a new static int variable called cleanup at the top of `tcpserv04.c` right after the `#include` statements:

```
static int cleanup;  /* To handle shutdown */
```

Modify the echoserver loop by changing it from a for loop to a while loop:

```
while(cleanup != 1)
{
    /* echo server code here */
}
```

For the echoserver we need to disable the operating system from restarting calls which were being executed before the signal was handled after our handler has finished. By disabling these, the operating system will not restart calls to `accept()` after the signal has been handled. If we didn’t do this, we would have to wait for another client to connect and disconnect before

the echoserver would clean up resources and exit. To define the signal handler and turn off SA_RESTART, first define act and oact structures in the echoserver's main() function:

```
struct sigaction     act, oact;
```

Insert the following code after variable declarations, before the call to wolfSSL_Init() in the main function:

```
/* Signal handling code */
struct sigaction act, oact;         /* Declare the sigaction structs    */
act.sa_handler = sig_handler;       /* Tell act to use sig_handler      */
sigemptyset(&act.sa_mask);          /* Tells act to exclude all sa_mask *
                                     * signals during execution of      *
                                     * sig_handler.                     */
act.sa_flags = 0;                   /* States that act has a special    *
                                     * flag of 0                        */
sigaction(SIGINT, &act, &oact);     /* Tells the program to use (o)act  *
                                     * on a signal or interrupt         */
```

The echoserver's sig_handler function is shown below:

```
void sig_handler(const int sig)
{
    printf("\nSIGINT handled.\n");
    cleanup = 1;
    return;
}
```

That's it - the echoclient and echoserver are now enabled with TLSv1.2!!

What we did:

- Included the wolfSSL headers
- Initialized wolfSSL
- Created a WOLFSSL_CTX structure in which we chose what protocol we wanted to use
- Created a WOLFSSL object to use for sending and receiving data
- Replaced calls to Writen() and Readline() with wolfSSL_write() and wolfSSL_read()
- Freed WOLFSSL, WOLFSSL_CTX
- Made sure we handled client and server shutdown with signal handler

There are many more aspects and methods to configure and control the behavior of your SSL connections. For more detailed information, please see additional wolfSSL documentation and resources.

Once again, the completed source code can be found in the downloaded ZIP file at the top of this section.

11.13 Certificates

For testing purposes, you may use the certificates provided by wolfSSL. These can be found in the wolfSSL download, and specifically for this tutorial, they can be found in the finished_src folder.

For production applications, you should obtain correct and legitimate certificates from a trusted certificate authority.

11.14 Conclusion

This tutorial walked through the process of integrating the wolfSSL embedded SSL library into a simple client and server application. Although this example is simple, the same principles may be applied for adding SSL or TLS into your own application. The wolfSSL embedded SSL library provides all the features you would need in a compact and efficient package that has been optimized for both size and speed.

Being dual licensed under GPLv2 and standard commercial licensing, you are free to download the wolfSSL source code directly from our website. Feel free to post to our support forums (https://www.wolfssl.com/forums) with any questions or comments

you might have. If you would like more information about our products, please contact info@wolfssl.com.

We welcome any feedback you have on this SSL tutorial. If you believe it could be improved or enhanced in order to make it either more useful, easier to understand, or more portable, please let us know at support@wolfssl.com.

12 Best Practices for Embedded Devices

12.1 Creating Private Keys

Embedding a private key into firmware allows anyone to extract the key and turns an otherwise secure connection into something nothing more secure than TCP.

We have a few ideas about creating private keys for SSL enabled devices.

1. Each device acting as a server should have a unique private key, just like in the non-embedded world.
2. If the key can't be placed onto the device before delivery, have it generated during setup.
3. If the device lacks the power to generate its own key during setup, have the client setting up the device generate the key and send it to the device.
4. If the client lacks the ability to generate a private key, have the client retrieve a unique private key over an SSL/TLS connection from the devices known website (for example).

wolfSSL (formerly CyaSSL) can be used in all of these steps to help ensure an embedded device has a secure unique private key. Taking these steps will go a long way towards securing the SSL connection itself.

12.2 Digitally Signing and Authenticating with wolfSSL

wolfSSL is a popular tool for digitally signing applications, libraries, or files prior to loading them on embedded devices. Most desktop and server operating systems allow creation of this type of functionality through system libraries, but stripped down embedded operating systems do not. The reason that embedded RTOS environments do not include digital signature functionality is because it has historically not been a requirement for most embedded applications. In today's world of connected devices and heightened security concerns, digitally signing what is loaded onto your embedded or mobile device has become a top priority.

Examples of embedded connected devices where this requirement was not found in years past include set top boxes, DVR's, POS systems, both VoIP and mobile phones, connected home, and even automobile-based computing systems. Because wolfSSL supports the key embedded and real time operating systems, encryption standards, and authentication functionality, it is a natural choice for embedded systems developers to use when adding digital signature functionality.

Generally, the process for setting up code and file signing on an embedded device are as follows:

1. The embedded systems developer will generate an RSA key pair.
2. A server-side script-based tool is developed
    1. The server side tool will create a hash of the code to be loaded on the device (with SHA-256 for example).
    2. The hash is then digitally signed, also called RSA private encrypt.
    3. A package is created that contains the code along with the digital signature.
3. The package is loaded on the device along with a way to get the RSA public key. The hash is re-created on the device then digitally verified (also called RSA public decrypt) against the existing digital signature.

Benefits to enabling digital signatures on your device include:

1. Easily enable a secure method for allowing third parties to load files to your device.
2. Ensure against malicious files finding their way onto your device.
3. Digitally secure firmware updates
4. Ensure against firmware updates from unauthorized parties

General information on code signing:

https://en.wikipedia.org/wiki/Code_signing

13 OpenSSL Compatibility

13.1 Compatibility with OpenSSL

wolfSSL (formerly CyaSSL) provides an OpenSSL compatibility header, `wolfssl/openssl/ssl.h`, in addition to the wolfSSL native API, to ease the transition into using wolfSSL or to aid in porting an existing OpenSSL application over to wolfSSL. For an overview of the OpenSSL Compatibility Layer, please continue reading below. To view the complete set of OpenSSL functions supported by wolfSSL, please see the `wolfssl/openssl/ssl.h` file.

The OpenSSL Compatibility Layer maps a subset of the most commonly-used OpenSSL commands to wolfSSL's native API functions. This should allow for an easy replacement of OpenSSL by wolfSSL in your application or project without changing much code.

Our test beds for OpenSSL compatibility are stunnel and Lighttpd, which means that we build both of them with wolfSSL as a way to test our OpenSSL compatibility API.

Building wolfSSL With Compatibility Layer:

1. Enable with (`--enable-opensslextra`) or by defining the macro `OPENSSL_EXTRA`.

   ```
   ./configure --enable-opensslextra
   ```

2. Include <`wolfssl/options.h`> as first wolfSSL header
3. Header files for migration are located under:
   - `./wolfssl/openssl/*.h`
   - Ex: <`wolfssl/openssl/ssl.h`>

13.2 Differences Between wolfSSL and OpenSSL

Many people are curious how wolfSSL compares to OpenSSL and what benefits there are to using an SSL/TLS library that has been optimized to run on embedded platforms. Obviously, OpenSSL is free and presents no initial costs to begin using, but we believe that wolfSSL will provide you with more flexibility, an easier integration of SSL/TLS into your existing platform, current standards support, and much more – all provided under a very easy-to-use license model.

The points below outline several of the main differences between wolfSSL and OpenSSL.

1. With a 20-100 kB build size, wolfSSL is up to 20 times smaller than OpenSSL. wolfSSL is a better choice for resource constrained environments – where every byte matters.
2. wolfSSL is up to date with the most current standards of TLS 1.3 with DTLS. The wolfSSL team is dedicated to continually keeping wolfSSL up-to-date with current standards.
3. wolfSSL offers the best current ciphers and standards available today, including ciphers for streaming media support. In addition, the recently-introduced NTRU cipher allows speed increases of 20-200x over standard RSA.
4. wolfSSL is dual licensed under both the GPLv2 as well as a commercial license, where OpenSSL is available only under their unique license from multiple sources.
5. wolfSSL is backed by an outstanding company who cares about its users and about their security, and is always willing to help. The team actively works to improve and expand wolfSSL. The wolfSSL team is based primarily out of Bozeman, MT, Portland, OR, and Seattle, WA, along with other team members located around the globe.
6. wolfSSL is the leading SSL/TLS library for real time, mobile, and embedded systems by virtue of its breadth of platform support and successful implementations on embedded environments. Chances are we've already been ported to your environment. If not, let us know and we'll be glad to help.
7. wolfSSL offers several abstraction layers to make integrating SSL into your environment and platform as easy as possible. With an OS layer, a custom I/O layer, and a C Standard Library abstraction layer, integration has never been so easy.
8. wolfSSL offers several support packages for wolfSSL. Available directly through phone, email or the wolfSSL product support forums, your questions are answered quickly and accurately to help you make progress on your project as quickly as possible.

13.3 Supported OpenSSL Structures

- SSL_METHOD holds SSL version information and is either a client or server method. (Same as WOLFSSL_METHOD in the native wolfSSL API).
- SSL_CTX holds context information including certificates. (Same as WOLFSSL_CTX in the native wolfSSL API).
- SSL holds session information for a secure connection. (Same as WOLFSSL in the native wolfSSL API).

13.4 Supported OpenSSL Functions

The three structures shown above are usually initialized in the following way:

```
SSL_METHOD* method = SSLv3_client_method();
SSL_CTX* ctx = SSL_CTX_new(method);
SSL* ssl = SSL_new(ctx);
```

This establishes a client side SSL version 3 method, creates a context based on the method, and initializes the SSL session with the context. A server side program is no different except that the SSL_METHOD is created using SSLv3_server_method(), or one of the available functions. For a list of supported functions, please see the Protocol Support section. When using the OpenSSL Compatibility layer, the functions in this section should be modified by removing the "wolf" prefix. For example, the native wolfSSL API function:

```
wolfTLSv1_client_method()
```

Becomes:

```
TLSv1_client_method()
```

When an SSL connection is no longer needed the following calls free the structures created during initialization.

```
SSL_CTX_free(ctx);
SSL_free(ssl);
```

SSL_CTX_free() has the additional responsibility of freeing the associated SSL_METHOD. Failing to use the XXX_free() functions will result in a resource leak. Using the system's free() instead of the SSL ones results in undefined behavior.

Once an application has a valid SSL pointer from SSL_new(), the SSL handshake process can begin. From the client's view, SSL_connect() will attempt to establish a secure connection.

```
SSL_set_fd(ssl, sockfd);
SSL_connect(ssl);
```

Before the SSL_connect() can be issued, the user must supply wolfSSL with a valid socket file descriptor, sockfd in the example above. sockfd is typically the result of the TCP function socket() which is later established using TCP connect(). The following creates a valid client side socket descriptor for use with a local wolfSSL server on port 11111, error handling is omitted for simplicity.

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
sockaddr_in servaddr;
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(11111);
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
connect(sockfd, (const sockaddr*)&servaddr, sizeof(servaddr));
```

Once a connection is established, the client may read and write to the server. Instead of using the TCP functions send() and receive(), wolfSSL and yaSSL use the SSL functions SSL_write() and SSL_read(). Here is a simple example from the client demo:

```
char msg[] = "hello wolfssl!";
int wrote = SSL_write(ssl, msg, sizeof(msg));
char reply[1024];
int read = SSL_read(ssl, reply, sizeof(reply));
```

```
reply[read] = 0;
printf("Server response: %s\n", reply);
```

The server connects in the same way, except that it uses SSL_accept() instead of SSL_connect(), analogous to the TCP API. See the server example for a complete server demo program.

13.5 x509 Certificates

Both the server and client can provide wolfSSL with certificates in either **PEM** or **DER**.

Typical usage is like this:

```
SSL_CTX_use_certificate_file(ctx, "certs/cert.pem",
SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "certs/key.der",
SSL_FILETYPE_ASN1);
```

A key file can also be presented to the Context in either format. SSL_FILETYPE_PEM signifies the file is PEM formatted while SSL_FILETYPE_ASN1 declares the file to be in DER format. To verify that the key file is appropriate for use with the certificate the following function can be used:

```
SSL_CTX_check_private_key(ctx);
```

14 Licensing

14.1 Open Source

wolfSSL (formerly CyaSSL), yaSSL, wolfCrypt, yaSSH and TaoCrypt software are free software downloads and may be modified to the needs of the user as long as the user adheres to version two of the GPL License. The GPLv2 license can be found on the gnu.org website https://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

wolfSSH software is a free software download and may be modified to the needs of the user as long as the user adheres to version three of the GPL license. The GPLv3 license can be found on the gnu.org website (https://www.gnu.org/licenses/gpl.html).

14.2 Commercial Licensing

Businesses and enterprises who wish to incorporate wolfSSL products into proprietary appliances or other commercial software products for re-distribution must license commercial versions. Commercial licenses for wolfSSL, yaSSL, and wolfCrypt are available for $5,000 USD per end product or SKU. Licenses are generally issued for one product and include unlimited royalty-free distribution. Custom licensing terms are also available.

Commercial licenses are also available for wolfMQTT and wolfSSH. Please contact licensing@wolfssl.com with inquiries.

14.3 Support Packages

Support packages for wolfSSL products are available on an annual basis directly from wolfSSL. With three different package options, you can compare them side-by-side and choose the package that best fits your specific needs. Please see our Support Packages page (https://www.wolfssl.com/products/support-and-maintenance) for more details.

15 Support and Consulting

15.1 How to Get Support

For general product support, wolfSSL (formerly CyaSSL) maintains an online forum for the wolfSSL product family. Please post to the forums or contact wolfSSL directly with any questions.

- wolfSSL (yaSSL) Forums: https://www.wolfssl.com/forums
- Email Support: support@wolfssl.com

For information regarding wolfSSL products, questions regarding licensing, or general comments, please contact wolfSSL by emailing info@wolfssl.com. For support packages, please see Licensing.

15.1.1 Bugs Reports and Support Issues

If you are submitting a bug report or asking about a problem, please include the following information with your submission:

1. wolfSSL version number
2. Operating System version
3. Compiler version
4. The exact error you are seeing
5. A description of how we can reproduce or try to replicate this problem

With the above information, we will do our best to resolve your problems. Without this information, it is very hard to pinpoint the source of the problem. wolfSSL values your feedback and makes it a top priority to get back to you as soon as possible.

15.2 Consulting

wolfSSL offers both on and off site consulting - providing feature additions, porting, a Competitive Upgrade Program (see Competitive Upgrade Program), and design consulting.

15.2.1 Feature Additions and Porting

We can add additional features that you may need which are not currently offered in our products on a contract or co-development basis. We also offer porting services on our products to new host languages or new operating environments.

15.2.2 Competitive Upgrade Program

We will help you move from an outdated or expensive SSL/TLS library to wolfSSL with low cost and minimal disturbance to your code base.

Program Outline:

1. You need to currently be using a commercial competitor to wolfSSL.
2. You will receive up to one week of on-site consulting to switch out your old SSL library for wolfSSL. Travel expenses are not included.
3. Normally, up to one week is the right amount of time for us to make the replacement in your code and do initial testing. Additional consulting on a replacement is available as needed.
4. You will receive the standard wolfSSL royalty free license to ship with your product.
5. The price is $10,000.

The purpose of this program is to enable users who are currently spending too much on their embedded SSL implementation to move to wolfSSL with ease. If you are interested in learning more, then please contact us at info@wolfssl.com.

15.2.3 Design Consulting

If your application or framework needs to be secured with SSL/TLS but you are uncertain about how the optimal design of a secured system would be structured, we can help!

We offer design consulting for building SSL/TLS security into devices using wolfSSL. Our consultants can provide you with the following services:

1. *Assessment*: An evaluation of your current SSL/TLS implementation. We can give you advice on your current setup and how we think you could improve upon this by using wolfSSL.
2. *Design*: Looking at your system requirements and parameters, we'll work closely with you to make recommendations on how to implement wolfSSL into your application such that it provides you with optimal security.

If you would like to learn more about design consulting for building SSL into your application or device, please contact info@wolfssl.com for more information.

16 wolfSSL (formerly CyaSSL) Updates

16.1 Product Release Information

We regularly post update information on Twitter. For additional release information, you can keep track of our projects on GitHub, follow us on Facebook, or follow our daily blog.

- wolfSSL on GitHub - https://www.github.com/wolfssl/wolfssl
- wolfSSL on Twitter - https://twitter.com/wolfSSL
- wolfSSL on Facebook - https://www.facebook.com/wolfSSL
- wolfSSL on Reddit - https://www.reddit.com/r/wolfssl/
- Daily Blog - https://www.wolfssl.com/blog

17 wolfSSL API Reference

18 wolfCrypt API Reference

A SSL/TLS Overview

A.1 General Architecture

The wolfSSL (formerly CyaSSL) embedded SSL library implements SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3 protocols. TLS 1.3 is currently the most secure and up to date version of the standard. wolfSSL does not support SSL 2.0 due to the fact that it has been insecure for several years.

The TLS protocol in wolfSSL is implemented as defined in RFC 5246 (https://tools.ietf.org/html/rfc5246). Two record layer protocols exist within SSL - the message layer and the handshake layer. Handshake messages are used to negotiate a common cipher suite, create secrets, and enable a secure connection. The message layer encapsulates the handshake layer while also supporting alert processing and application data transfer.

A general diagram of how the SSL protocol fits into existing protocols can be seen in **Figure 1**. SSL sits in between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the transport medium. Application protocols are layered on top of SSL (such as HTTP, FTP, and SMTP).

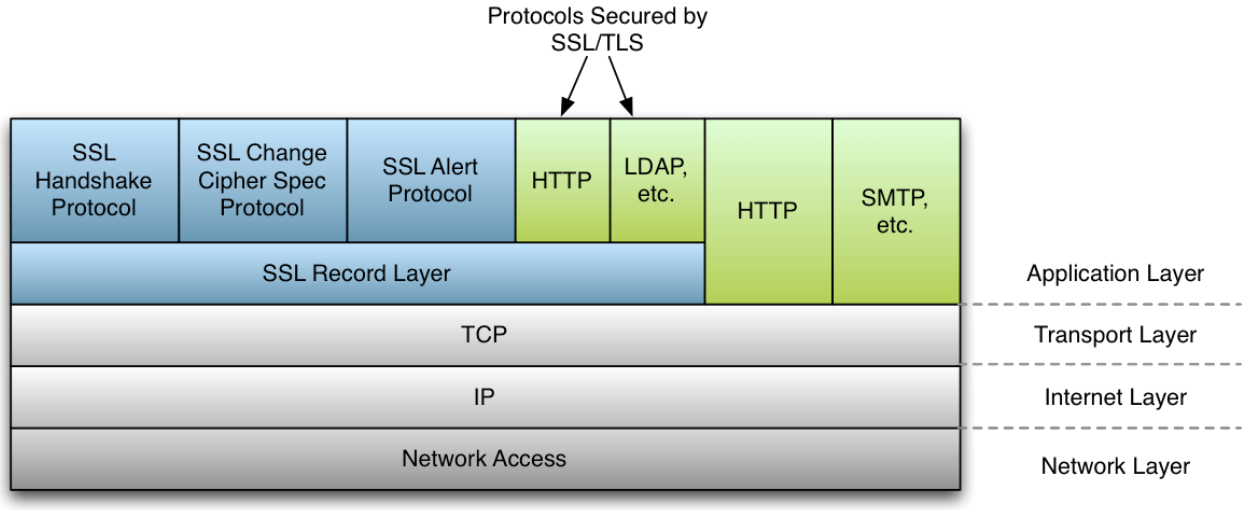

Figure 3: SSL Protocol Diagram

A.2 SSL Handshake

The SSL handshake involves several steps, some of which are optional depending on what options the SSL client and server have been configured with. Below, in **Figure 2**, you will find a simplified diagram of the SSL handshake process.

A.3 Differences between SSL and TLS Protocol Versions

SSL (Secure Sockets Layer) and TLS (Transport Security Layer) are both cryptographic protocols which provide secure communication over networks. These two protocols (and the several versions of each) are in widespread use today in applications ranging from web browsing to e-mail to instant messaging and VoIP. Each protocol, and the underlying versions of each, are slightly different from the other.

Below you will find both an explanation of, and the major differences between the different SSL and TLS protocol versions. For specific details about each protocol, please reference the RFC specification mentioned.

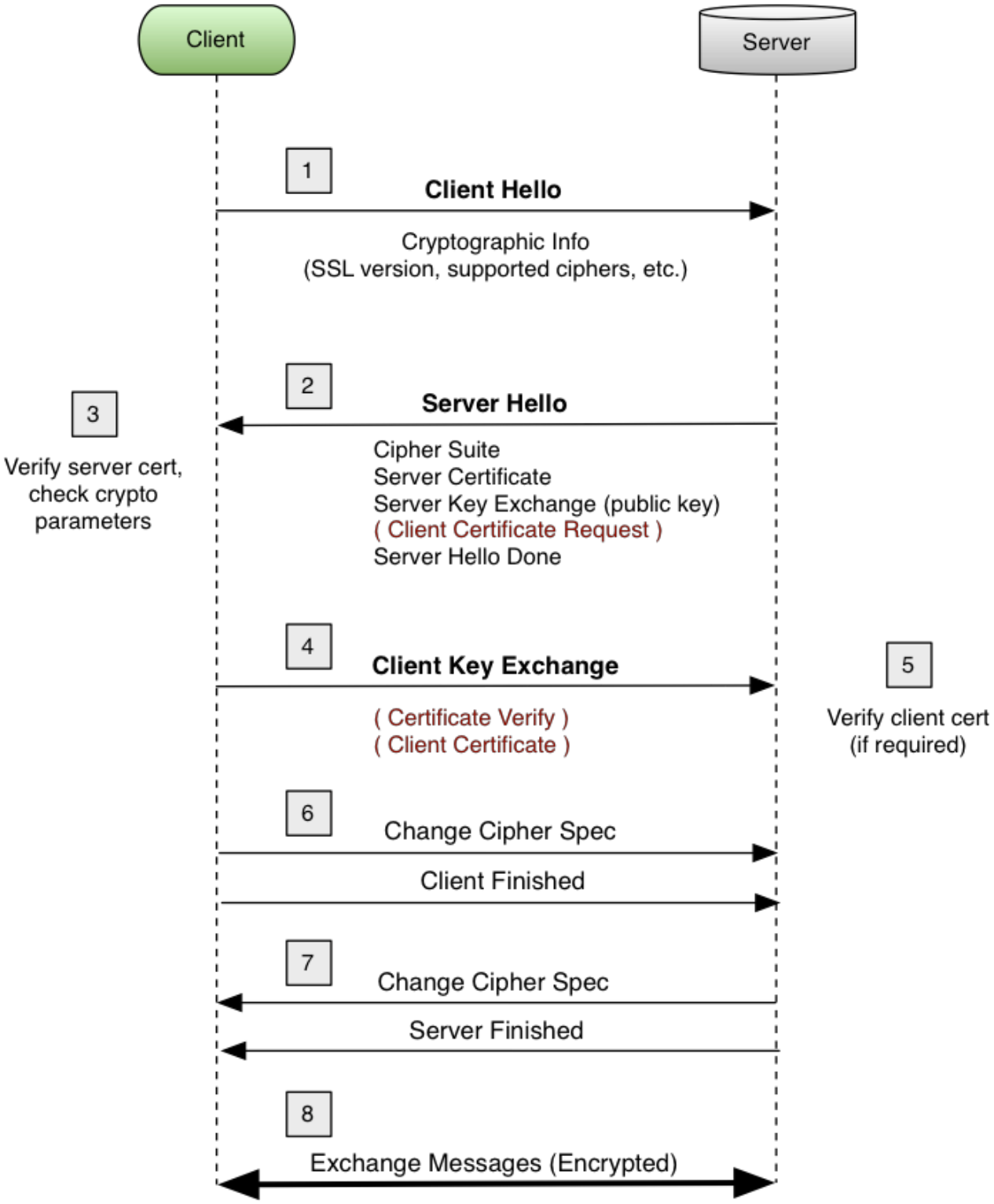

Figure 4: SSL Handshake Diagram

A.3.1 SSL 3.0

This protocol was released in 1996 but began with the creation of SSL 1.0 developed by Netscape. Version 1.0 wasn't released, and version 2.0 had a number of security flaws, thus leading to the release of SSL 3.0. Some major improvements of SSL 3.0 over SSL 2.0 are:

- Separation of the transport of data from the message layer
- Use of a full 128 bits of keying material even when using the Export cipher
- Ability of the client and server to send chains of certificates, thus allowing organizations to use certificate hierarchy which is more than two certificates deep.
- Implementing a generalized key exchange protocol, allowing Diffie-Hellman and Fortezza key exchanges as well as non-RSA certificates.
- Allowing for record compression and decompression
- Ability to fall back to SSL 2.0 when a 2.0 client is encountered

A.3.2 TLS 1.0

This protocol was first defined in RFC 2246 in January of 1999. This was an upgrade from SSL 3.0 and the differences were not dramatic, but they are significant enough that SSL 3.0 and TLS 1.0 don't interoperate. Some of the major differences between SSL 3.0 and TLS 1.0 are:

- Key derivation functions are different
- MACs are different - SSL 3.0 uses a modification of an early HMAC while TLS 1.0 uses HMAC.
- The Finished messages are different
- TLS has more alerts
- TLS requires DSS/DH support

A.3.3 TLS 1.1

This protocol was defined in RFC 4346 in April of 2006, and is an update to TLS 1.0. The major changes are:

- The Implicit Initialization Vector (IV) is replaced with an explicit IV to protect against Cipher block chaining (CBC) attacks.
- Handling of padded errors is changed to use the `bad_record_mac` alert rather than the `decryption_failed` alert to protect against CBC attacks.
- IANA registries are defined for protocol parameters
- Premature closes no longer cause a session to be non-resumable.

A.3.4 TLS 1.2

This protocol was defined in RFC 5246 in August of 2008. Based on TLS 1.1, TLS 1.2 contains improved flexibility. The major differences include:

- The MD5/SHA-1 combination in the pseudorandom function (PRF) was replaced with cipher-suite-specified PRFs.
- The MD5/SHA-1 combination in the digitally-signed element was replaced with a single hash. Signed elements include a field explicitly specifying the hash algorithm used.
- There was substantial cleanup to the client's and server's ability to specify which hash and signature algorithms they will accept.
- Addition of support for authenticated encryption with additional data modes.
- TLS Extensions definition and AES Cipher Suites were merged in.
- Tighter checking of EncryptedPreMasterSecret version numbers.
- Many of the requirements were tightened
- `Verify_data` length depends on the cipher suite
- Description of Bleichenbacher/Dlima attack defenses cleaned up.

A.3.5 TLS 1.3

This protocol was defined in RFC 8446 in August of 2018. TLS 1.3 contains improved security and speed. The major differences include:

- The list of supported symmetric algorithms has been pruned of all legacy algorithms. The remaining algorithms all use Authenticated Encryption with Associated Data (AEAD) algorithms.
- A zero-RTT (0-RTT) mode was added, saving a round-trip at connection setup for some application data at the cost of certain security properties.
- All handshake messages after the ServerHello are now encrypted.
- Key derivation functions have been re-designed, with the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) being used as a primitive.
- The handshake state machine has been restructured to be more consistent and remove superfluous messages.
- ECC is now in the base spec and includes new signature algorithms. Point format negotiation has been removed in favor of single point format for each curve.
- Compression, custom DHE groups, and DSA have been removed, RSA padding now uses PSS.
- TLS 1.2 version negotiation verification mechanism was deprecated in favor of a version list in an extension.
- Session resumption with and without server-side state and the PSK-based ciphersuites of earlier versions of TLS have been replaced by a single new PSK exchange.

B RFCs, Specifications, and Reference

B.1 Protocols

- SSL v3.0 - IETF Draft
- TLS v1.0 - RFC2246
- TLS v1.1 - RFC4346
- TLS v1.2 - RFC5246
- TLS v1.3 - RFC8446
- DTLS - RFC4347 Specification document
- IPv4 - Wikipedia
- IPv6 - Wikipedia

B.2 Stream Ciphers

- Stream Cipher Information - Wikipedia
- HC-128 - Specification document
- RABBIT - Specification document
- RC4 / ARC4 - IETF Draft Wikipedia

B.3 Block Ciphers

- Block Cipher Information - Wikipedia
- AES - NIST Publication Wikipedia
- AES-GCM - NIST Specification
- AES-NI - Intel Sofware Network
- DES/3DES - NIST Publication Wikipedia

B.4 Hashing Functions

- SHA - NIST FIPS180-1 Publication NIST FIPS180-2 Publication Wikipedia
- MD4 - RFC1320
- MD5 - RFC1321
- RIPEMD-160 - Specification document

B.5 Public Key Cryptography

- Diffie-Hellman - Wikipedia
- RSA - MIT Paper Wikipedia
- DSA/DSS - NIST FIPS186-3
- ECDSA - Specification Document
- NTRU - Wikipedia
- X.509 - RFC3279
- ASN.1 - Specification Document Wikipedia
- PSK - RFC4279

B.6 Other

- PKCS#5, PBKDF1, PBKDF2 - RFC2898
- PKCS#8 - RFC5208
- PKCS#12 - Wikipedia

C Error Codes

C.1 wolfSSL Error Codes

wolfSSL (formerly CyaSSL) error codes can be found in `wolfssl/ssl.h`. For detailed descriptions of the following errors, see the OpenSSL man page for `SSL_get_error` (`man SSL_get_error`).

Error Code Enum	Error Code	Error Description
SSL_ERROR_WANT_READ	2	
SSL_ERROR_WANT_WRITE	3	
SSL_ERROR_WANT_CONNECT	7	
SSL_ERROR_WANT_ACCEPT	8	
SSL_ERROR_SYSCALL	5	
SSL_ERROR_WANT_X509_LOOKUP	83	
SSL_ERROR_ZERO_RETURN	6	
SSL_ERROR_SSL	85	

Additional wolfSSL error codes can be found in `wolfssl/error-ssl.h`

Error Code Enum	Error Code	Error Description
INPUT_CASE_ERROR	-301	process input state error
PREFIX_ERROR	-302	bad index to key rounds
MEMORY_ERROR	-303	out of memory
VERIFY_FINISHED_ERROR	-304	verify problem on finished
VERIFY_MAC_ERROR	-305	verify mac problem
PARSE_ERROR	-306	parse error on header
UNKNOWN_HANDSHAKE_TYPE	-307	weird handshake type
SOCKET_ERROR_E	-308	error state on socket
SOCKET_NODATA	-309	expected data, not there
INCOMPLETE_DATA	-310	don't have enough data to complete task
UNKNOWN_RECORD_TYPE	-311	unknown type in record hdr
DECRYPT_ERROR	-312	error during decryption
FATAL_ERROR	-313	revcd alert fatal error
ENCRYPT_ERROR	-314	error during encryption
FREAD_ERROR	-315	fread problem
NO_PEER_KEY	-316	need peer's key
NO_PRIVATE_KEY	-317	need the private key
RSA_PRIVATE_ERROR	-318	error during rsa priv op
NO_DH_PARAMS	-319	server missing DH params
BUILD_MSG_ERROR	-320	build message failure
BAD_HELLO	-321	client hello malformed
DOMAIN_NAME_MISMATCH	-322	peer subject name mismatch
WANT_READ	-323	want read, call again
NOT_READY_ERROR	-324	handshake layer not ready
VERSION_ERROR	-326	record layer version error
WANT_WRITE	-327	want write, call again
BUFFER_ERROR	-328	malformed buffer input
VERIFY_CERT_ERROR	-329	verify cert error
VERIFY_SIGN_ERROR	-330	verify sign error
CLIENT_ID_ERROR	-331	psk client identity error
SERVER_HINT_ERROR	-332	psk server hint error
PSK_KEY_ERROR	-333	psk key error
GETTIME_ERROR	-337	gettimeofday failed ???

Error Code Enum	Error Code	Error Description
GETITIMER_ERROR	-338	getitimer failed ???
SIGACT_ERROR	-339	sigaction failed ???
SETITIMER_ERROR	-340	setitimer failed ???
LENGTH_ERROR	-341	record layer length error
PEER_KEY_ERROR	-342	cant decode peer key
ZERO_RETURN	-343	peer sent close notify
SIDE_ERROR	-344	wrong client/server type
NO_PEER_CERT	-345	peer didn't send key
NTRU_KEY_ERROR	-346	NTRU key error
NTRU_DRBG_ERROR	-347	NTRU drbg error
NTRU_ENCRYPT_ERROR	-348	NTRU encrypt error
NTRU_DECRYPT_ERROR	-349	NTRU decrypt error
ECC_CURVETYPE_ERROR	-350	Bad ECC Curve Type
ECC_CURVE_ERROR	-351	Bad ECC Curve
ECC_PEERKEY_ERROR	-352	Bad Peer ECC Key
ECC_MAKEKEY_ERROR	-353	Bad Make ECC Key
ECC_EXPORT_ERROR	-354	Bad ECC Export Key
ECC_SHARED_ERROR	-355	Bad ECC Shared Secret
NOT_CA_ERROR	-357	Not CA cert error
BAD_CERT_MANAGER_ERROR	-359	Bad Cert Manager
OCSP_CERT_REVOKED	-360	OCSP Certificate revoked
CRL_CERT_REVOKED	-361	CRL Certificate revoked
CRL_MISSING	-362	CRL Not loaded
MONITOR_SETUP_E	-363	CRL Monitor setup error
THREAD_CREATE_E	-364	Thread Create Error
OCSP_NEED_URL	-365	OCSP need an URL for lookup
OCSP_CERT_UNKNOWN	-366	OCSP responder doesn't know
OCSP_LOOKUP_FAIL	-367	OCSP lookup not successful
MAX_CHAIN_ERROR	-368	max chain depth exceeded
COOKIE_ERROR	-369	dtls cookie error
SEQUENCE_ERROR	-370	dtls sequence error
SUITES_ERROR	-371	suites pointer error
OUT_OF_ORDER_E	-373	out of order message
BAD_KEA_TYPE_E	-374	bad KEA type found
SANITY_CIPHER_E	-375	sanity check on cipher error
RECV_OVERFLOW_E	-376	RXCB returned more than rqed
GEN_COOKIE_E	-377	Generate Cookie Error
NO_PEER_VERIFY	-378	Need peer cert verify Error
FWRITE_ERROR	-379	fwrite problem
CACHE_MATCH_ERROR	-380	cache hrd match error
UNKNOWN_SNI_HOST_NAME_E	-381	Unrecognized host name Error
UNKNOWN_MAX_FRAG_LEN_E	-382	Unrecognized max frag len Error
KEYUSE_SIGNATURE_E	-383	KeyUse digSignature error
KEYUSE_ENCIPHER_E	-385	KeyUse KeyEncipher error
EXTKEYUSE_AUTH_E	-386	ExtKeyUse server
SEND_OOB_READ_E	-387	Send Cb out of bounds read
SECURE_RENEGOTIATION_E	-388	Invalid renegotiation info
SESSION_TICKET_LEN_E	-389	Session Ticket too large
SESSION_TICKET_EXPECT_E	-390	Session Ticket missing
SCR_DIFFERENT_CERT_E	-391	SCR Different cert error
SESSION_SECRET_CB_E	-392	Session secret CB fcn failure
NO_CHANGE_CIPHER_E	-393	Finished before change cipher

Error Code Enum	Error Code	Error Description
SANITY_MSG_E	-394	Sanity check on msg order error
DUPLICATE_MST_E	-395	Duplicate message error
SNI_UNSUPPORTED	-396	SSL 3.0 does not support SNI
SOCKET_PEER_CLOSED_E	-397	Underlying transport closed
BAD_TICKET_KEY_CB_SZ	-398	Bad session ticket key cb size
BAD_TICKET_MSG_SZ	-399	Bad session ticket msg size
BAD_TICKET_ENCRYPT	-400	Bad user ticket encrypt
DH_KEY_SIZE_E	-401	DH key too small
SNI_ABSENT_ERROR	-402	No SNI request
RSA_SIGN_FAULT	-403	RSA sign fault
HANDSHAKE_SIZE_ERROR	-404	Handshake message too large
UNKNOWN_ALPN_PROTOCOL_NAME_E	-405	Unrecognized protocol name error
BAD_CERTIFICATE_STATUS_ERROR	-406	Bad certificate status message
OCSP_INVALID_STATUS	-407	Invalid OCSP status
OCSP_WANT_READ	-408	OCSP callback response
RSA_KEY_SIZE_E	-409	RSA key too small
ECC_KEY_SIZE_E	-410	ECC key too small
DTLS_EXPORT_VER_E	-411	Export version error
INPUT_SIZE_E	-412	Input size too big error
CTX_INIT_MUTEX_E	-413	Initialize ctx mutex error
EXT_MASTER_SECRET_NEEDED_E	-414	Need EMS enabled to resume
DTLS_POOL_SZ_E	-415	Exceeded DTLS pool size
DECODE_E	-416	Decode handshake message error
HTTP_TIMEOUT	-417	HTTP timeout for OCSP or CRL req
WRITE_DUP_READ_E	-418	Write dup write side can't read
WRITE_DUP_WRITE_E	-419	Write dup read side can't write
INVALID_CERT_CTX_E	-420	TLS cert ctx not matching
BAD_KEY_SHARE_DATA	-421	Key share data invalid
MISSING_HANDSHAKE_DATA	-422	Handshake message missing data
BAD_BINDER	-423	Binder does not match
EXT_NOT_ALLOWED	-424	Extension not allowed in msg
INVALID_PARAMETER	-425	Security parameter invalid
MCAST_HIGHWATER_CB_E	-426	Multicast highwater cb err
ALERT_COUNT_E	-427	Alert count exceeded err
EXT_MISSING	-428	Required extension not found
UNSUPPORTED_EXTENSION	-429	TLSX not requested by client
PRF_MISSING	-430	PRF not compiled in
DTLS_RETX_OVER_TX	-431	Retransmit DTLS flight over
DH_PARAMS_NOT_FFDHE_E	-432	DH params from server not FFDHE
TCA_INVALID_ID_TYPE	-433	TLSX TCA ID type invalid
TCA_ABSENT_ERROR	-434	TLSX TCA ID no response

Negotiation Parameter Errors

Error Code Enum	Error Code	Error Description
UNSUPPORTED_SUITE	-500	Unsupported cipher suite
MATCH_SUITE_ERROR	-501	Can't match cipher suite
COMPRESSION_ERROR	-502	Compression mismatch
KEY_SHARE_ERROR	-503	Key share mismatch
POST_HAND_AUTH_ERROR	-504	Client won't do post-hand auth
HRR_COOKIE_ERROR	-505	HRR msg cookie mismatch

C.2 wolfCrypt Error Codes

wolfCrypt error codes can be found in `wolfssl/wolfcrypt/error.h`.

Error Code Enum	Error Code	Error Description
OPEN_RAN_E	-101	opening random device error
READ_RAN_E	-102	reading random device error
WINCRYPT_E	-103	windows crypt init error
CRYPTGEN_E	-104	windows crypt generation error
RAN_BLOCK_E	-105	reading random device would block
BAD_MUTEX_E	-106	Bad mutex operation
MP_INIT_E	-110	mp_init error state
MP_READ_E	-111	mp_read error state
MP_EXPTMOD_E	-112	mp_exptmod error state
MP_TO_E	-113	mp_to_xxx error state, can't convert
MP_SUB_E	-114	mp_sub error state, can't subtract
MP_ADD_E	-115	mp_add error state, can't add
MP_MUL_E	-116	mp_mul error state, can't multiply
MP_MULMOD_E	-117	mp_mulmod error state, can't multiply mod
MP_MOD_E	-118	mp_mod error state, can't mod
MP_INVMOD_E	-119	mp_invmod error state, can't inv mod
MP_CMP_E	-120	mp_cmp error state
MP_ZERO_E	-121	got a mp zero result, not expected
MEMORY_E	-125	out of memory error
RSA_WRONG_TYPE_E	-130	RSA wrong block type for RSA function
RSA_BUFFER_E	-131	RSA buffer error, output too small or input too large
BUFFER_E	-132	output buffer too small or input too large
ALGO_ID_E	-133	setting algo id error
PUBLIC_KEY_E	-134	setting public key error
DATE_E	-135	setting date validity error
SUBJECT_E	-136	setting subject name error
ISSUER_E	-137	setting issuer name error
CA_TRUE_E	-138	setting CA basic constraint true error
EXTENSIONS_E	-139	setting extensions error
ASN_PARSE_E	-140	ASN parsing error, invalid input
ASN_VERSION_E	-141	ASN version error, invalid number
ASN_GETINT_E	-142	ASN get big int error, invalid data
ASN_RSA_KEY_E	-143	ASN key init error, invalid input
ASN_OBJECT_ID_E	-144	ASN object id error, invalid id
ASN_TAG_NULL_E	-145	ASN tag error, not null
ASN_EXPECT_0_E	-146	ASN expect error, not zero
ASN_BITSTR_E	-147	ASN bit string error, wrong id
ASN_UNKNOWN_OID_E	-148	ASN oid error, unknown sum id
ASN_DATE_SZ_E	-149	ASN date error, bad size
ASN_BEFORE_DATE_E	-150	ASN date error, current date before
ASN_AFTER_DATE_E	-151	ASN date error, current date after
ASN_SIG_OID_E	-152	ASN signature error, mismatched oid
ASN_TIME_E	-153	ASN time error, unknown time type
ASN_INPUT_E	-154	ASN input error, not enough data
ASN_SIG_CONFIRM_E	-155	ASN sig error, confirm failure
ASN_SIG_HASH_E	-156	ASN sig error, unsupported hash type
ASN_SIG_KEY_E	-157	ASN sig error, unsupported key type
ASN_DH_KEY_E	-158	ASN key init error, invalid input
ASN_NTRU_KEY_E	-159	ASN ntru key decode error, invalid input

Error Code Enum	Error Code	Error Description
ASN_CRIT_EXT_E	-160	ASN unsupported critical extension
ECC_BAD_ARG_E	-170	ECC input argument of wrong type
ASN_ECC_KEY_E	-171	ASN ECC bad input
ECC_CURVE_OID_E	-172	Unsupported ECC OID curve type
BAD_FUNC_ARG	-173	Bad function argument provided
NOT_COMPILED_IN	-174	Feature not compiled in
UNICODE_SIZE_E	-175	Unicode password too big
NO_PASSWORD	-176	no password provided by user
ALT_NAME_E	-177	alt name size problem, too big
AES_GCM_AUTH_E	-180	AES-GCM Authentication check failure
AES_CCM_AUTH_E	-181	AES-CCM Authentication check failure
CAVIUM_INIT_E	-182	Cavium Init type error
COMPRESS_INIT_E	-183	Compress init error
COMPRESS_E	-184	Compress error
DECOMPRESS_INIT_E	-185	DeCompress init error
DECOMPRESS_E	-186	DeCompress error
BAD_ALIGN_E	-187	Bad alignment for operation, no alloc
ASN_NO_SIGNER_E	-188	ASN sig error, no CA signer to verify certificate
ASN_CRL_CONFIRM_E	-189	ASN CRL no signer to confirm failure
ASN_CRL_NO_SIGNER_E	-190	ASN CRL no signer to confirm failure
ASN_OCSP_CONFIRM_E	-191	ASN OCSP signature confirm failure
BAD_ENC_STATE_E	-192	Bad ecc enc state operation
BAD_PADDING_E	-193	Bad padding, msg not correct length
REQ_ATTRIBUTE_E	-194	Setting cert request attributes error
PKCS7_OID_E	-195	PKCS#7, mismatched OID error
PKCS7_RECIP_E	-196	PKCS#7, recipient error
FIPS_NOT_ALLOWED_E	-197	FIPS not allowed error
ASN_NAME_INVALID_E	-198	ASN name constraint error
RNG_FAILURE_E	-199	RNG Failed, Reinitialize
HMAC_MIN_KEYLEN_E	-200	FIPS Mode HMAC Minimum Key Length error
RSA_PAD_E	-201	RSA Padding Error
LENGTH_ONLY_E	-202	Returning output length only
IN_CORE_FIPS_E	-203	In Core Integrity check failure
AES_KAT_FIPS_E	-204	AES KAT failure
DES3_KAT_FIPS_E	-205	DES3 KAT failure
HMAC_KAT_FIPS_E	-206	HMAC KAT failure
RSA_KAT_FIPS_E	-207	RSA KAT failure
DRBG_KAT_FIPS_E	-208	HASH DRBG KAT failure
DRBG_CONT_FIPS_E	-209	HASH DRBG Continuous test failure
AESGCM_KAT_FIPS_E	-210	AESGCM KAT failure
THREAD_STORE_KEY_E	-211	Thread local storage key create failure
THREAD_STORE_SET_E	-212	Thread local storage key set failure
MAC_CMP_FAILED_E	-213	MAC comparison failed
IS_POINT_E	-214	ECC is point on curve failed
ECC_INF_E	-215	ECC point infinity error
ECC_PRIV_KEY_E	-216	ECC private key not valid error
SRP_CALL_ORDER_E	-217	SRP function called in the wrong order
SRP_VERIFY_E	-218	SRP proof verification failed
SRP_BAD_KEY_E	-219	SRP bad ephemeral values
ASN_NO_SKID	-220	ASN no Subject Key Identifier found
ASN_NO_AKID	-221	ASN no Authority Key Identifier found
ASN_NO_KEYUSAGE	-223	ASN no Key Usage found

Error Code Enum	Error Code	Error Description
SKID_E	-224	Setting Subject Key Identifier error
AKID_E	-225	Setting Authority Key Identifier error
KEYUSAGE_E	-226	Bad Key Usage value
CERTPOLICIES_E	-227	Setting Certificate Policies error
WC_INIT_E	-228	wolfCrypt failed to initialize
SIG_VERIFY_E	-229	wolfCrypt signature verify error
BAD_PKCS7_SIGNEEDS_CHECKCOND_E	-230	Bad condition variable operation
SIG_TYPE_E	-231	Signature Type not enabled/available
HASH_TYPE_E	-232	Hash Type not enabled/available
WC_KEY_SIZE_E	-234	Key size error, either too small or large
ASN_COUNTRY_SIZE_E	-235	ASN Cert Gen, invalid country code size
MISSING_RNG_E	-236	RNG required but not provided
ASN_PATHLEN_SIZE_E	-237	ASN CA path length too large error
ASN_PATHLEN_INV_E	-238	ASN CA path length inversion error
BAD_KEYWRAP_ALG_E	-239	Algorithm error with keywrap
BAD_KEYWRAP_IV_E	-240	Decrypted AES key wrap IV incorrect
WC_CLEANUP_E	-241	wolfCrypt cleanup failed
ECC_CDH_KAT_FIPS_E	-242	ECC CDH known answer test failure
DH_CHECK_PUB_E	-243	DH check public key error
BAD_PATH_ERROR	-244	Bad path for opendir
ASYNC_OP_E	-245	Async operation error
ECC_PRIVATEONLY_E	-246	Invalid use of private only ECC key
EXTKEYUSAGE_E	-247	Bad extended key usage value
WC_HW_E	-248	Error with hardware crypto use
WC_HW_WAIT_E	-249	Hardware waiting on resource
PSS_SALTLEN_E	-250	PSS length of salt is too long for hash
PRIME_GEN_E	-251	Failure finding a prime
BER_INDEF_E	-252	Cannot decode indefinite length BER
RSA_OUT_OF_RANGE_E	-253	Ciphertext to decrypt out of range
RSAPSS_PAT_FIPS_E	-254	RSA-PSS PAT failure
ECDSA_PAT_FIPS_E	-255	ECDSA PAT failure
DH_KAT_FIPS_E	-256	DH KAT failure
AESCCM_KAT_FIPS_E	-257	AESCCM KAT failure
SHA3_KAT_FIPS_E	-258	SHA-3 KAT failure
ECDHE_KAT_FIPS_E	-259	ECDHE KAT failure
AES_GCM_OVERFLOW_E	-260	AES-GCM invocation counter overflow
AES_CCM_OVERFLOW_E	-261	AES-CCM invocation counter overflow
RSA_KEY_PAIR_E	-262	RSA Key Pair-Wise consistency check fail
DH_CHECK_PRIVE_E	-263	DH check private key error
WC_AFALG_SOCK_E	-264	AF_ALG socket error
WC_DEVCRYPTO_E	-265	/dev/crypto error
ZLIB_INIT_ERROR	-266	Zlib init error
ZLIB_COMPRESS_ERROR	-267	Zlib compression error
ZLIB_DECOMPRESS_ERROR	-268	Zlib decompression error
PKCS7_NO_SIGNER_E	-269	No signer in PKCS7 signed data msg
WC_PKCS7_WANT_READ_E	-270	PKCS7 stream operation wants more input
CRYPTOCB_UNAVAILABLE	-271	Crypto callback unavailable
PKCS7_SIGNEEDS_CHECK	-272	Signature needs verified by caller
ASN_SELF_SIGNED_E	-273	ASN self-signed certificate error
MIN_CODE_E	-300	errors -101 - -299

C.3 Common Error Codes and their Solution

There are several error codes that commonly happen when getting an application up and running with wolfSSL.

C.3.1 ASN_NO_SIGNER_E (-188)

This error occurs when using a certificate and the signing CA certificate was not loaded. This can be seen using the wolfSSL example server or client against another client or server, for example connecting to Google using the wolfSSL example client:

```
$ ./examples/client/client -g -h www.google.com -p 443
```

This fails with error -188 because Google's CA certificate wasn't loaded with the "-A" command line option.

C.3.2 WANT_READ (-323)

The WANT_READ error happens often when using non-blocking sockets, and isn't actually an error when using non-blocking sockets, but it is passed up to the caller as an error. When a call to receive data from the I/O callback would block as there isn't data currently available to receive, the I/O callback returns WANT_READ. The caller should wait and try receiving again later. This is usually seen from calls to wolfSSL_read(), wolfSSL_negotiate(), wolfSSL_accept(), and wolfSSL_connect(). The example client and server will indicate the WANT_READ incidents when debugging is enabled.