wolfSSL ユーザ・マニュアル

（この資料は **wolfSSL User Manual**, Version 3.9.0
第一章から第九章の日本語翻訳版です）

目次

1. はじめに

このマニュアルは wolfSSL(旧 CyaSSL)組込み向け SSL ライブラリーのテクニカルガイドとして書かれています。wolfSSL のビルド方法、使い方、wolfSSL の機能概説、移植性強化点、サポートその他を解説します。

wolfSSL 選択する理由

組込み向けソリューションとして wolfSSL を選択する理由はたくさんあります。簡単には、まずメモリーサイズ（wolfSSL は 30k バイトのサイズでビルド可能です）、そして最新の標準への準拠（TLS1.1 と 1.2、DTLS1.2)、新しい暗号サポート（ストリーム型暗号を含む）、多数のプラットホーム、ロイヤルティー・フリー、また古いアプリケーションの容易な移植を実現する OpenSSL 互換 API などなど。完全な機能リストについて、セッション 4.1 を参照してください。

2. wolfSSL のビルド

wolfSSL(旧 CyaSSL)はポータビリティーに配慮してコーディングされており、ほとんどのシステムに対して容易にビルドすることができるはずです。WolfSSL のビルドに関してお困りの点などございましたら、遠慮なくサポートフォーラム（ http://www.wolfssl.com/forums ）または support@wolfssl.com に直接お問い合わせください。

本章では Unix や Windows で wolfSSL をビルドする方法を解説するとともに、組み込みプラットフォームなど非標準環境での wolfSSL ビルドについてのガイダンスも提供します。第三章使用方法、第11章 wolfSSL SSL チュートリアルも合わせて参照してください。

Autoconf/automake システムを使って wolfSSL をビルドする場合は、wolfSSL は一つの Makefile を使ってライブラリーのすべてのパーツやサンプルプログラムをビルドします。この方法は Makefile を再帰的に使用するよりシンプルかつ高速です。

2.1 wolfSSL ソース・コードの入手

wolfSSL の最新版は wolfSSL ウェブサイトの下記ページから ZIP ファイルでダウンロードすることができます。

https://wolfssl.com/wolfSSL/download/downloadForm.php

ZIP ファイルをダウンロード後、unzip コマンドで unzip します。ネイティブの行端コードを使用するために unzip を使用する場合は ”-a” オプションをオンにします。Unzip のマニュアルページから “-a” オプションの機能について引用します。

“–a オプションは zip でテキストファイルに指定されたファイルに対して（zipinfo リスティングで、”b”ではなく ”t“ラベルのもの）以下のような形で自動的に抽出します。（例えば、Unix ファイルは行端（EOL）にラインフィード（LF）を使用し、ファイル末尾（EOF）には何も無し、アップル OS では EOL には改行（CR）、またほとんどの PC オペレーティングシステムでは EOL に CR+LF を使用します。加えて、IBM メインフレームやミシガン端末システムではより普及している ASCII コードではなく EBCDIC を、NT では Unicode をサポートします。）”

注：wolfSSL2.0.0rc3 以降では標準のインストール場所とともにディレクトリー構造が変更となっています。これらの変更は wolfSSL のオープンソース・プロジェクトへの統合を容易にするためのもので

す。さらに詳細のヘッダーと構造変更については、本マニュアルのセクション9．1および9．3を参照してください。

2.2 ＊nix 系でのビルド

Linux、各種 BSD、OS X、Solaris またはその他の*nix ライクのシステム上での WolfSSL のビルドでは autconf システムを使用してください。wolfSSL をビルドするためにはたった二つのコマンドが必要なだけです：

./configure
make

./configure コマンドには各種のオプションを指定することができます。利用可能なオプションについては 2.5 章を参照するか、下記のようにヘルプを実行してください。

./configure --help

wolfSSL をビルドするには make コマンドを実行します。

make

wolfSSL をインストールするためには下記のコマンドを実行してください：

make install

インストールのためにはスーパーユーザの権限が必要かもしれません。その場合はコマンドの直前に sudo をつけて実行します。

sudo make install

ビルドをテストするためには、wolfSSL ソースディレクトリの root から testsuite プログラム実行します：

./testsuite/testsuite.test

または、testsuite とともに標準の wolfSSL API と暗号アルゴリズムのテストも実行するためには下記のようにコマンドを実行します。

make test

testsuite の期待される出力に関してさらに詳細はユーザマニュアルの 3.2 を参照してください。wolfSSL ライブラリーだけをビルド、追加のものが無い場合（例えば testsuite, benchmark app など）は wolfSSL root ディレクトリーで以下のコマンドを実行することも可能です。

make src/libwolfssl.la

2.3 Windows 上でのビルド

Visual Sturdio によるビルドについては、”wolfSSL with Visual Studio” により詳細な説明があります。

VS2008：Visual Studio2008 用のソリューションは wolfSSL の root ディレクトリーに格納されています。Visual Studio2010 ないしそれ以降のバージョンを使用する場合、プロジェクト・ファイルはインポート処理に変換されます。

注意：
Visual Studio の新しいバージョンにインポートする場合、プロジェクトとインポートされるプロパティ・シートを上書きするか聞かれます。「いいえ」を選択することで以下を回避することができます。「はい」を選択すると、SAFESEH 仕様のせいで EDITANDCONTINUE の警告が無視されることになります testsuite, sslSniffer, server echoserver, echoclient と client をそれぞれ右クリックして Properties→Configuration Properties→Linker→Advanced を探します。Adviced ウインドウを一番下までスクロールして、”Image Has Save Exception Handlers”の右端のドロップダウンをクリックし、No (SAFESEH:NO) を選択します。これを前述の各々に対して行います。

VS2010: 一旦更新されてから、wolfSSL をビルドするために Service Pack 1 をダウンロードする必要があります。もし VS がリンカーエラーを出力するようなら、プロジェクトをクリーンして、プロジェクトを Rebuild すれば、リンカーエラーはなおるはずです。

VS2013(64 ビットソリューション): 一旦更新されてから、wolfSSL をビルドするために Service Pack 4 をダウンロードする必要があります。し VS がリンカーエラーを出力するようなら、プロジェクトを Rebuild すれば、リンカーエラーはなおるはずです。

それぞれのビルドをテストするには、Vidual Studio メニューの”Build ALL” を選択し、それから testsuite プログラムを実行します。Visual Studio プロジェクトのビルド・オプションをエディットするには、所定のプロジェクト(wolsssl, echoclient, echoserve など)を選択して、Properties パネルを見てください。

Cygwin：Cygwin を使用する場合、または、その他の *nix ライクのコマンドを提供する Windows 上のツールセットを使用する場合は、上記セクション２．“ *nix 上でのビルド”を参照してください。Windows 開発環境上での Windows 用の wolfSSL ビルドには、同梱の Visual Studio の使用をお勧めします。

2.4 (組込みシステム向け)非標準環境でのビルド

正式サポートではありませんが、wolfSSL を非標準の環境、特に組込み向けのクロスコンパイルシステムでビルドしようと思われる方々をできる限りお手伝いします。以下は、この環境での作業開始のための注意点です。

1. ソースファイルとヘッダーファイルを wolfSSL ダウンロード・パッケージ内のディレクトリ構成と同じに保つようにして、適当な格納ディレクトリーにコピーしてください。
2. 開発環境によってはヘッダーファイルの格納場所を明示的に指定する必要がある場合があります。ヘッダーファイルは<wolfssl_root>/wolfssl ディレクトリーにあります。通常、インクルード・バスに<wolfssl_root>ディレクトリーを追加することでヘッダー・パスは解決するはずです。
3. wolfSSL のデフォルトは、configure コマンド処理でビッグ・エンディアンを検出しない限りリトルエンディアンです。非標準環境のユーザは configure コマンドを使用していないので、ビッグエンディアンのシステムを使用する場合は BIG_ENDIAN_ORDER を定義する必要があります。

4. wolfSSL は 64 ビット型を有効利用して高速化します。configure コマンド処理を使用している場合は long か long long が 64bit かどうか判定し、定義を設定します。非標準環境のユーザはターゲット・システムで sizeof(long)が 8 バイトならば、SIZEOF_LONG 8 を定義してください。もしそうでなく sizeof(long long)が 8 バイトの場合は、SIZEOF_LONG_LONG 8 を定義してください。
5. ライブラリーのビルドで何か問題があった場合は info@wolfssl.com (日本語：info@wolfssl.jp)にご連絡ください。
6. ビルドを修正できる定義については以下のサブセクションに掲載されています。各オプションのさらに詳しい説明は 2.5.1 ビルドオプション・ノートで後述します。

2.4.1 機能の削除

以下の定義を使用して wolfSSL の機能を削除することができます。これはライブラリー全体の所要サイズを減らそうとするのに役立ちます。NO_<機能名> を定義することに加え、対応するソースファイルをビルドから取り除くこともできます（ただし、ヘッダーファイルを除く）。

NO_WOLFSSL_CLIENT はクライアント向け固有の関数呼び出しを削除します。これはサーバのみのビルドのためのものです。これはサイズ削減のためにいくつかの呼び出しを取り除きたい場合のみに使用してください。

NO_WOLFSSL_SERVER は同様にサーバ向け固有の関数呼び出しを削除します。

NO_DES は DES3 暗号化を削除します。DES3 は古いサーバで要求される場合があり、SSL3.0 要件でもあるので、デフォルトで組み込まれます。

NO_DH と NO_AES は上記と同じですが、広く使用されています。

NO_DSA は DSA を削除します。DSA は次第に使用されなくなりつつあります。

NO_ERROR_STRINGS エラー文字列を無効化します。エラー文字列は wolfSSL 用のものが src/internal.c に、wolfCrypt 用のものが wolfcrypt/src/error.c にあります。

NO_HMAC は HMAC をビルドから取り除きます。

NO_MD4 は MD4 をビルドから取り除きます。MD4 はすでに解読されているので、使用すべきではありません。

NO_MD5 は MD5 をビルドから取り除きます。

NO_SHA256 は SHA-256 をビルドから取り除きます。

NO_PSK は事前共有鍵の使用をオフにします。これはデフォルトで組み込みです。

NO_PWDBASED は PBKDF1、 PDKDF2、また PCKS#12 の BKDF などパスワードベース鍵導出関数を無効化します。

NO_RC4 はストリーム暗号化 ARC4 をビルドから取り除きます。ARC4 は現在も広く使われているので、デフォルトで組込みです。

NO_RABBIT と NO_HC128 はストリーム暗号化の拡張をビルドから取り除きます。

NO_SESSION_CACHE はセッション・キャッシュが不要の場合、定義することができます。これにより 3kB 近くのメモリー使用が削減されるはずです。

NO_TLS は TLS をオフにしますが、オフにすることはお勧めしません。

SMALL_SESSION_CACHE は wolfSSL が使用する SSL セッション・キャッシュのサイズを制限するために定義することができます。これにより、デフォルトのセッション・キャッシュを 33 セッションから 6 セッションに削減し、約 2.5kB 節約できます。

WC_NO_RSA_OAEP は OAEP パディングのためのコードを取り除きます。

2.4.2デフォルトで無効の機能を有効にする

WOLFSSL_CERT_GEN は wolfSSL の証明書生成機能を有効にします。詳細は本マニュアルの第 7 章を参照してください。

WOLFSSL_DER_LOAD は wolfSSL_CTX_CTX_der_load_veryfiy_locations()関数を使用して wolfSSL コンテクスト(WOLFSSL_CTX)に DER フォーマットの CA 証明書をロード可能にします。

WOLFSSL_DTLS は DTLS（データグラム TLS）を有効にします。これは広くサポートされておらず、また使用されていないので、デフォルトはオフとなっています。

WOLFSSL_KEY_GEN は wolfSSL の RSA 鍵生成機能を有効にします。詳しい情報は本マニュアル第七章を参照してください。

WOLFSSL_RIPEMD は RIPEMD-160 サポートを有効にします。

WOLFSSL_SHA384 は SHA-384 サポートを有効にします。

WOLFSSL_SHA512 は SHA-512 サポートを有効にします。

DEBUG_WOLFSSL はデバッグ・モードでビルドします。wolfSSL のデバッグについては本マニュアル第八章を参照してください。この機能はデフォルトではオフです。

HAVE_AESCCM は AES-CCM サポートを有効にします。

HAVE_AESGCM は AES-GCM サポートを有効にします。

HAVE_CAMELLIA は Camellia サポートを有効にします。

HAVE_CHACHA は ChaCha20 サポートを有効にします。

HAVE_POLY1305 は Poly1305 サポートを有効にします。

HAVE_CRL は証明書失効リスト(CRL)サポートを有効にします。

HAVE_ECC は楕円曲線暗号化(ECC)サポートを有効にします。

HAVE_LIBZ はコネクション時のデータ圧縮を可能にする拡張です。デフォルトでは無効で、通常は使用すべきでありません。後述のビルド・オプション注意点の libz の項目を参照してください。

HAVE_OCSP はオンライン証明書ステータス確認プロトコル(OCSP)サポートを有効にします。

OPENSSL_EXTRA はライブラリーにさらなる OpenSSL 互換性を組込み、wolfSSL を OpenSSL で動作するよう設計された既存アプリケーションに移植容易にする wolfSSL、OpenSSL 互換性レイヤーを有効にします。この機能はデフォルトでは無効です。

TEST_IPV6 はアプリケーションのテストにおいて IPv6 のテストを有効にします。
wolfSSL 自身は IP 中立ですが、デフォルトでは IPv4 を使用したアプリケーションをテストします。

HAVE_CSHARP は C＃ラッパーに必要なコンフィグレーション・オプションを有効化します。

CURVED25519_SMALL は curve25519 と ed25519 の両方に対してメモリ使用量を抑えます。これはメモリー使用量とスピードのトレードオフです。

HAVE_CURVE25519 は curve25519 を有効化します。

HAVE_ED25519 は ed25519 を有効化します。

WOLFSSL_DH_CONST はディフィー・ヘルマン処理時の浮動小数点の使用をオフにし、POW と LOG のためのテーブルを使用します。外部数学ライブラリへの依存を取り除きます。

WOLFSSL_TRUST_PEER_CERT は、信頼する相手方の証明書の使用を有効化します。これによって、CA 証明書ではなく相手方の証明書をロードして接続に対して照合します。有効化された場合、相手方証明書チェーンがロードされていなくても、信頼する相手方の証明書がマッチすれば、認証されたとして扱います。この機能はデフォルトではオフになっており、CA 証明書を使うことをお勧めします。

WOLFSSL_STATIC_MEMORY は静的メモリーバッファの使用と関数を有効化します。これによって、動的ではなく静的メモリーの使用が可能となります。

WOLFSSL_SESSION_EXPORT は、DTLS セッション・エキスポート、インポートの使用を有効化します。これによって、DTLS セッションの現在の状態を送受することが可能となります。

2.4.3 wolfSSL カスタマイズおよびポーティング

WOLFSSL_CALLBACKS はデバッガ無しの環境下でシグナルを使用してコールバックのデバッグができるようにするための拡張です。デフォルトではオフです。これはまたソケットをブロックしてタイマーを設定するためにも使用できます。詳細は第六章を参照してください。

WOLFSSL_USER_IO はデフォルト I/O 関数の EmbedSend() と EmbedReceive() の自動設定を取り除くことができるようにします。カスタム I/O 抽象化レイヤーのために使用されます（詳細は本マニュアル 5.1 を参照してください）。

NO_FILESYSTEM は証明書と鍵ファイルをロードするために stdio が利用できない場合に使用します。これによって、ファイル上のそれらの変わりにバッファー拡張の使用を可能になります。

NO_INLINE は小さな頻繁に使用される関数の自動インライン展開を不可にします。これらの関数は小さな関数で、通常関数呼び出しのセットアップ、リターンよりずっと小さいため、これをオンにすると wolfSSL の実行速度は低下し、サイズは大きくなります。

NO_DEV_RANDOM はデフォルトの /dev/random ランダム数値生成を使用しないようにします。これが定義された場合には、OS 固有の GenerateSeed()関数 (“wolfcrypt/src/random.c” にあります) を書く必要があります。

NO_MAIN_DRIVER は通常のビルド環境においてテスト・アプリケーションがそれ自身が main 関数を持つのか、別のドライバー・アプリケーションから呼ばれるのかを決定するために使用されます。以下のテストファイルを使用する際に指定します：test.c, client.c, server.c, echoclient.c, echoserver.c, および testsuite.c

NO_WRITEV は writev() セマンティクスのシミュレーションを不可にします。

SINGLE_THREADED は相互排他の使用をオフにするスイッチです。wolfSSL は現在これをセッションキャッシュのためだけに使用しています。wolfSSL の使用が常にシングル・スレッドの場合、これをオンにすることができます。

USER_TICKS は time(0)を使用したくない場合、自分自身のクロック・テイック関数を定義することができるようにします。カスタム関数は秒単位の精度を必要としますが、EPOCH と連携されている必要はありません。“wolfssl_int.c”の LowResTimer()関数を参照してください。

USER_TIME は自分自身のものを使用したい（必要がある）場合に、time.h の構造体定義を無効にします。ユーザは XTIME, XGMTIME、および XVALIDATE_DATE を自分で定義します。

USE_CERT_BUFFER_1024 は<wolfSSL_rppt>/wolfSSL/certs_test.h に格納されている1024 ビットのテスト用証明書と鍵用のバッファを有効にします。ファイルシステムの無い組み込みシステムに移植、テストする際に手助けとなります。

USE_CERT_BUFFER_2048 は<wolfSSL_rppt>/wolfSSL/certs_test.h に格納されている2048 ビットのテスト用証明書と鍵用のバッファを有効にします。ファイルシステムの無い組み込みシステムに移植、テストする際に手助けとなります。

CUSTOM_RAND_GENERATE_SEED によって、ユーザは wc_GenerateSeedGenerateSeed(byte* output, word32 sz)と等価な関数を定義することができます。

CUSTOM_RAND_GENERATE_BLOCK によって、ユーザはカスタム乱数生成関数を定義することができます。
以下に使用例を示します。

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

または、

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

2.4.4メモリー使用量削減

TFM_TIMING_RESISTANT はスタックサイズの小さなシステムで fast math (USE_FAST_MATH オプション) を使用する場合に定義することができます。これによって、大きな静的配列が排除されます。

WOLFSSL_SMALL_STACK はスタックサイズの小さなデバイスのために使用します。これによって wolfcrypt/src/integer.c の動的メモリーの使用は増加しますが、性能低下を引き起こす可能性があります。

RSA_LOW_MEM は定義された CRT が使用されない場合、幾らかのメモリーを削減します。ただし、RSA 処理の速度が低下します。デフォルトではオフとなっています。

2.4.5 性能改善

WOLFSSL_AESNI はいくつかのインテルチップセットに組込まれている AES 高速化処理の使用を有効にします。この定義を使用する場合、aes_asm.s ファイルが wolfSSL ビルドソースに追加されている必要があります。

USE_FAST_MATH は big integer ライブラリーを、可能な場合アセンブラー命令を使用するより高速なものに切り替えます。Fastmath は RSA, DH または DSA のような公開鍵の演算を高速化します。Big integer ライブラリーは通常もっとも移植性も高く、簡単に使用できますが、通常の big integer ライブラリーの短所は実行速度が遅く、動的メモリーを多量に使用する点です。Mastmath を使用する場合、スタックメモリーの使用量が大きくなるため、このオプションの使用する場合、TFM_TIMING_REGISTANT を併用することをお勧めします。

2.4.6プロトコル・スタックおよびチップ固有の定義

wolfSSL は各種のプラットフォームと TCP/IP スタック向けにビルドすることができます。以下のオプションは ./wolfssl/wolfcrypt/settings.h の中に定義されていて、デフォルトではコメント化されています。それぞれは、参照されている特定チップやスタック向けのサポートを有効にするためにコメントを外すことができます。

IPHONE は iOS 上で使用のためのビルドの場合に定義します。

THREADX は ThreadX RTOS (www.rtos.com)上で使用のためのビルドの場合に定義します。

MICRIUM は Micrium の μC/OS (www.micrium.com)上で使用のためのビルドの場合に定義します。

MBED は mbed プロトタイピング・プラットフォーム（www.mbed.org）向けのビルドの際に定義します。

MICROCHIP_PIC32 は PIC32 プラットフォーム(www.microchip.com)上で使用のためのビルドの場合に定義します。

MICROCHIP_TCPIP_V5 は Microchip　TCP/IP スタック・バージョン 5 上で使用のためのビルドの場合に定義します。

MICROCHIP_TCPIP は Microchip　TCP/IP スタック・バージョン 6 以降で使用のためのビルドの場合に定義します。

WOLFSSL_MICROCHIP_PIC32MZ は PIC32MZ のハードウェア暗号化機能使用のためのビルドの場合に定義します。

FREERTOS は FreeRTOS (www.freertos.org) 向けビルドの場合、定義することができます。LwIP を使用する場合は WOLFSSL_LWIP も定義します。

FREERTOS_WINSIM は FreeRTOS Windows シミュレータ (www.freertos.org) 向けビルドの場合、定義することができます。

EBSNET は EMSnet 製品と RTIP 上で使用のためのビルドの場合に定義します。

WOLFSSL_LWIP は LwIP TCP/IP スタック(http://savannah.nongnu.org/projects/lwip/)上で wolfSSL を使用する場合に定義します。

WOLFSSL_GAME_BUILD はゲームコンソール向けに wolfSSL をビルドする際に定義することができます。

WOLFSSL_LSR は LSR 用のビルドの場合に定義します。

FREESCLAE_MQX は Freescale MQX/RTCS/MFS (www.freescale.com)のためのビルドの際、定義します。この定義によって Kinetis H/W 乱数生成アクセラレータ・サポートのための FREESCLAE_K70_RNGA が定義されます。

WOLFSSL_STM32F2 は STM32F2 のために定義します。この定義によって wolfSSL の STM32F2 ハードウェア暗号とハードウェア RNG サポートが有効となります。

COMVERGE は Comverge のための設定を使用するために定義します。

WOLFSSL_QL は QL SEP のための設定を使用するために定義します。

WOLFSSL_EROAD は EROAD のビルドのために定義します。

WOLFSSL_IAR_ARM は IAR EWARM 向けのビルドのために定義します。

WOLFSSL_TIRTOS は TI-RTOS 向けのビルド時に定義します。

WOLFSSL_ROWLEY_ARM はRowley CrossWorksのビルド時に使用します。

WOLFSSL_NRF51 は、Nordic nRF51へのポーティングの際に定義することができます。(NOT THERE)

WOLFSSL_NRF51_AES は、Nordic nRF51へのポーティングの際、AES128-ECB暗号化のためのビルドインAESハードウェアを使用するために定義することができます。

2.5 ビルド・オプション

以下は wolfSSL ライブラリーのビルド方法をカスタマイズするために ./configure スクリプトに追加できるオプションです。

デフォルトでは wolfSSL は、スタティック・モード無効で、共有モードのみでビルドします。これにより、倍のオーダーでビルド時間が高速化されます。必要ならばどちらのモードも明示的に無効化あるいは有効化することができます。

オプション	デフォルト値	説明
--enable-debug	無効	wolfSSL デバッグサポートを有効化

--enable-singleThreaded	無効	シングルスレッド・モードを有効化。マルチスレッド保護無し
--enable-dtls	無効	wolfSSL の DTLS サポートを有効化
--enable-openssh	無効	OpenSSH 互換性ビルドを有効化。
--enable-opensslextra	無効	拡張 OpenSSL API 互換性を有効化。サイズが増加します。
--enable-maxstrength	無効	最大暗号化強度のビルドを有効化。TSLv1.2-AEAD-PFS 暗号化のみ可能と成ります。
--enable-ipv6	無効	IPv6 のテストを有効化。wolfSSL 自身は IP 中立。
--enable-bump	無効	SSL Bump ビルド
--enable-leanpsk	無効	最小 PSK ビルド
--enable-bigcache	無効	大きなセッション・キャッシュの有効化
--enable-hugecache	無効	巨大なセッション・キャッシュの有効化
--enable-smallcache	無効	小さなセッション・キャッシュの有効化
--enable-savesession	無効	セッションキャッシュの永続化
--enable-savecert	無効	証明書キャッシュの永続化
--enable-atomicuser	無効	アトミックユーザレコード層の有効化
--enable-pkcallbacks	無効	公開鍵コールバックの有効化
--enable-sniffer	無効	Sniffer サポートの有効化
--enable-aesgcm	無効	AES-GCM サポートの有効化
--enable-aesccm	無効	AES-CCM サポートの有効化
--enable-aesni	無効	Intel AES-NI サポートの有効化
--enable-intelasm	無効	全ての Intel ASM スピードアップを有効化
--enable-poly1305	無効	Poly1305 サポートの有効化
--enable-camellia	無効	Camellia サポートの有効化
--enable-md2	無効	MD2 サポートの有効化
--enable-nullcipher	無効	NULL サイファーサポートの有効化
--enable-ripemd	無効	RIPEDM-160 サポートの有効化
--enable-blake2	無効	BLAKE2 サポートの有効化

--enable-sha512	無効	wolfSSL の SHA-512 サポートの有効化
--enable-sessioncerts	無効	セッション証明書ストアを有効化
--enable-keygen	無効	鍵生成を有効化
--enable-certgen	無効	証明書生成を有効化
--enable-certreq	無効	証明書要求を有効化
--enable-sep	無効	SEP を有効化
--enable-hkdf	無効	HKDF (HMAC-KDF)を有効化
--enable-dsa	無効	TLSDSA を有効化
--enable-eccshamir	X86_64 では有効	ECC Shamir を有効化
--enable-ecc	X86_64 では有効	ECC を有効化
--enable-fpecc	無効	固定小数点キャッシュ ECC を有効化
--enable-eccencrypt	無効	ECC 暗号化を有効化
--enable-psk	無効	PSK (事前共有鍵)を有効化
--enable-errorstrings	有効	エラーメッセージ文字列テーブルを有効化
--enable-oldtls	有効	TLS1.2 より古い TLS を有効化
--enable-sslv3	無効	SSL3.0 を有効化
--enable-stacksize	無効	サンプル・プログラムにおけるスタックサイズを有効化
--enable-memory	有効	メモリーコールバックを有効化
--enable-rsa	有効	RSA を有効化
--enable-dh	有効	DH を有効化
--enable-anon	無効	匿名を有効化
--enable-asn	有効	ASN を有効化
--enable-aes	有効	AES を有効化
--enable-coding	有効	ベース 16/64 コードを有効化
--enable-des3	有効	DES3 を有効化
--enable-idea	無効	IDEA 暗号を有効化
--enable-arc4	有効	ARC4 を有効化
--enable-md5	有効	MD5 を有効化
--enable-sha	有効	SHA を有効化

--enable-webserver	無効	Web サーバ向け SSL 機能を有効化
--enable-md4	無効	MD4 を有効化
--enable-pwdbased	無効	PWDBASED を有効化
--enable-hc128	無効	ストリーム暗号化 HC-128 を有効化
--enable-rabbit	無効	ストリーム暗号化 RABBIT を有効化
--enable-chacha	無効	ChaCha20 を有効化
--enable-fips	無効	FIPS140-2 を有効化
--enable-hashbrdg	無効	Hash DRBG を有効化
--enable-filesystem	有効	ファイルシステム・サポートを有効化
--enable-inline	有効	インライン関数を有効化
--enable-ocsp	無効	OCSP機能 (Online Certificate Status Protocol)を有効化
--enable-ocspstapling	無効	OCSP Stapling を有効化
--enable-ocspstapling2	無効	OCSP Stapling バージョン 2 を有効化
--enable-crl	無効	CRL 機能を有効化
--enable-crl-monitor	無効	CRL モニターを有効化
--enable-ntru	無効	NTRU のビルドを有効化（要ライセンス）
--enable-sni	無効	SNI (Server Name Indication)を有効化
--enable-maxfragment	無効	Maximum Fragment Length を有効化
--enable-alpn	無効	Application Layer Protocol Negotiation (ALPN) を有効化
--enable-trancatedmac	無効	Truncated Keyed-hash MAC (HMAC)を有効化
--enable-renegotiation-indication	無効	Renegotiation Indication を有効化
--enable-supportedcurves	無効	「サポートされる楕円曲線」拡張を有効化
--enable-session-ticket	無効	セッション・チケット機能を有効化
--enable-tlsx	無効	TLS 拡張を有効化
--enable-pkcs7	無効	PKCS#7 を有効化
--enable-scep	無効	wolfSCEP を有効化
--enable-smallstack	無効	小さいスタックサイズを有効化
--enable-valgrind	無効	単体テストのための Valgrind を有効化

--enable-testcert	無効	テスト用証明書を有効化
--enable-iopool	無効	I/O プールの例を有効化
--enable-certservice	無効	証明書サービス（Windows Servers）を有効化
--enable-fastmath	X86_64 にて有効	fast math を有効化
--enable-fastmathhugemath	無効	Fast math 有効＋大きなコード
--enable-examples	有効	サンプル・プログラムを有効化
--enable-mcapi	無効	Microchip API を有効化
--enable-asynccrypt	無効	非同期暗号化を有効化
--enable-sessionexport	無効	セッションのインポート、エクスポートを有効化
--enable-jobserver[=no/yes/#]	Yes	最大＃数のジョブを有効化。Yes: CPU カウント＋1 を有効化
--enable-shared[=PKGS]	無効	共有 wolfSSL ライブラリーのビルドを有効化 [default = no]
--enable-static[=PKGS]	無効	静的 wolfSSL ライブラリーのビルドを有効化 [default = no]
--with-ntru=PATH	無効	NTRU インストール・パス(デフォルトは /usr/)
--with-libz=PATH	無効	圧縮のための libz をオプションでインクルード
--with-cavium=PATH	無効	Cavium/software ディレクトリーへのパス
--enable-fast-rsa	無効	RSA のための Intel IPP ライブラリを有効化
--with-user-crypto=PATH	無効	USER_CRYPTO のインストールパス（デフォルト：/usr/local）
--enable-curve25519	無効	curve25519 を有効化
--enable-ed25519	無効	ed25519 を有効化
--disable-crypttests	有効	クロスコンパイル向けに wolfCrypt テストとベンチマークを無効化
--enable-leantls	無効	TLS1.2 クライアントのみ(クライアント認証なし)の構成、ECC256、AES128、および

		SHA256、Shamir 無し。このオプション単独で使用。他のビルドオプションとの組み合わせることはできない。
--enable-staticmemory	無効	静的メモリーバッファの使用

2.5.1ビルド・オプションの注意点

debug - デバッグサポートを有効化で、デバッグ情報と DEBUG_WOLFSSL 定数の定義とともにコンパイルすることにより stderr にメッセージが出力され、デバッグを簡単化することができます。実行時にデバッグをオンにするには wolfSSL_Debugging_ON()を呼び出してください。実行時にデバッグをオフにするには wolfSSL_Debugging_OFF()を呼び出してください。詳しい情報については、本マニュアル第八章を参照してください。

singleThreaded - シングルスレッド・モードを有効化するとセッション・キャッシュのマルチスレッド保護が無効化されます。ユーザアプリケーションがシングルスレッドの場合、またはアプロケーションがマルチスレッドでも一度に一つのスレッドだけがライブラリーにアクセスすると分かっている場合にのみ、シングルスレッドモードを有効化してください。

dtls - DTLS サポートを有効化することで、TLS と SSL に加えて DTLS の実行のためにライブラリーを使用できるようになります。DTLS はまだ試験的なものですので、コメント、質問、要望などお知らせください。詳しい情報は本マニュアル第四章を参照してください。

opensslextra – OpenSSL 拡張を有効化することで、より広範囲の OpenSSL 互換関数を含むことになります。基本ビルドでもほとんどの TLS/SSL への要求に対して十分な関数が有効化されますが、数十、数百種類の OpenSSL 関数呼び出しを使用するアプリケーションを移植する場合はこのオプションでより良いサポートが可能になります。wolfSSL OpenSSL 互換レイヤーはアクティブに開発中です。もし、必要な関数が見当たらない場合はお手伝いさせていただきますので、お知らせください。OpenSSL 互換レイヤーに関する詳細はユーザマニュアル第十三章を参照してください。

ipv6 – IPV6 を有効化するとテストアプリケーションを IPv4 ではなく IPv6 を使用するように切り替えます。wolfSSL 自身は IP 中立で、どちらのバージョンでも使用することができますが、現在テストアプリケーションは IP 依存で、IPv4 がデフォルトです。

leanpsk – PSK を使用し、ライブラリから多くの機能を削除した非常に小さいビルドです。このオプションを有効にした場合、組み込み向け wolfSSL のサイズはおおよそ 21kB となります。

fastmath – fastmath を有効化すると、RSA, DH または DSA のような公開鍵の演算が高速化されます。デフォルトでは wolfSSL は普通の整数数学ライブラリーを使用します。これは通常、最も移植性がよく使用も容易です。普通の整数数学ライブラリーの欠点は、実行速度が遅いことと動的メモリーを多量に使用する点です。このオプションは big integer ライブラリーを、可能な場合、アセンブラー命令を使用するより高速なものに切り替えます。アセンブラー言語のインクルードはコンパイラーとプロセッサの組合せに依存します。いくつかの組合せではさらなるコンフィグレーション・フラグを必要としたり、不可能な場合もあるかもしれません。新たなアセンブラー・ルーチンでの fastmath 最適化に対するヘルプは有償コンサルティング・ベースでご提供可能です。

例えば、ia32 においては、高い最適化とフレームポインタの必要性の管理を回避するため、すべてのレジスターが利用可能である必要があります。wolfSSL は 非デバッグビルドのために “-O3 -fomit-frame-pointer” を GCC に追加します。違うコンパイラーをご使用の場合はコンフィグレーション中の CFLAGS にマニュアルでこれらを追加してください。

OS X では “-mdynamic-no-pic” も CFLAGS に追加する必要があります。さらに、OS X で ia32 向けに共有モードでビルドする場合は、LDFLAGS にもオプションを渡す必要があります：

LDFLAGS="-Wl,-read_only_relocs,warning"

これはいくつかのシンボルに対して、エラーを起こす代わりに、ワーニングを与えます。

fastmath もまた、動的メモリーとスタックメモリーの使用方法を切り替えます。通常の数学ライブラリーは big integer のために動的メモリーを使用します。Fastmath はデフォルトで、2048 ビット×2048 ビットの乗算が可能となるように 4096 ビット整数を保持する固定長バッファーを使用します。4096 ビット×4096 ビットの乗算が必要な場合は、wolfssl/wolfcrypt/tmf.h の中の FP_MAX_BITS を変更してください。FP_MAX_BITS が増えるに従い公開鍵演算で使用されるバッファーが大きくなるので、実行時のスタック使用量が増えます。ライブラリー内の２、３の関数では一時的な big integer を使用します。これは、スタックが相対的に大きくなる可能性があるということです。このようなことは、スタックサイズが小さな値に設定される組込みシステムやスレッド環境でしか問題になりません。もし、そのような環境において fastmath で公開鍵演算中にスタック破壊が起きる場合は、スタックサイズをそのようなスタック使用に充分になるように増やしてください。

autoconf システムを使用しないで fastmath を有効化する場合は、USE_FAST_MATH を定義し、integer.c の代わりに tfm.c を wolfSSL ビルドに追加する必要があります。

fastmath を使用する場合、スタックメモリーが大きくなる場合があるので、fastmath ライブラリーを使用するときは TFM_TIMING_RESISTANT を定義することを推奨します。これにより大きな静的配列を排除することができます。

fasthugemath – fasthugemath を有効化すると、fastmath ライブラリー向けのサポートを含み、公開鍵オプション中によく使われる鍵サイズ向けにループを展開してコードサイズをおおはばに増加させます。fasthugemath の使用前と使用後にベンチマーク・ユーティリティーを使用してみて、コードサイズの増加がいくぶんかのスピードアップに見合っているかどうか確認してください。

bigcache - 「大きなセッションキャッシュ」を有効にすることで、セッションキャッシュは 33 セッションから 1055 セッションに増加します。デフォルトのセッションサイズ 33 は TLS クライアントや組込みサーバー向けに適切です。大きなセッションキャッシュは、基本的に新セッション数が毎分 200 以下程度向けのあまり重い負荷でないサーバを可能にします。

hugecache - 「巨大セッションキャッシュ」を有効にすると、セッションキャッシュのサイズは 65,791 セッションとなります。このオプションは、毎分 13,000 新セッション以上、あるいは毎秒 200 新セッション以上の高負荷サーバー向けのオプションです。

smallcache – 小さなセッション・キャッシュによってwolfSSLは６セッションのみを保持することに成ります。これは、デフォルトの3kB近いセッションはRAMに対して負担となるような組込みクライアントやシステムに対して有効と考えられます。

savesession – このオプションの有効化によって、アプリケーションはwolfSSLセッションキャッシュのメモリーバッファへの/からの永続化（保存）と復旧が可能と成ります。

savecert – このオプションの有効化によって、アプリケーションはwolfSSLの証明書キャッシュのメモリーバッファへの/からの永続化（保存）と復旧が可能と成ります。

atomicuser – このオプションの有効化で、User Atomic Record Layer Processing コールバックをオンにします。これによって、ユーザ独自のMAC/暗号化と復号化/検証コールバックを登録することが可能と成ります。

pkcallbacks – このオプションの有効化で、公開鍵コールバックをオンにし、ECC署名/検証、およびRSA署名/検証、および暗号/復号コールバックをオンにします。

sniffer – sniffer（SSL 検査）サポートを有効化すると、正しい鍵ファイルによって SSL トラフィックのパケット収集および正しいキーファイルによるそれらのパケットの復号化を可能にします。現在、sniffer は以下の暗号化をサポートしています。

CBC 暗号化:
- AES-CBC
- Camellia-CBC
- 3DES-CBC

Stream 暗号化:
- RC4
- Rabbit
- HC-128

aesgcm - AES-GCM の有効化でそれらの暗号化スイーツを wolfSSL に追加します。
wolfSSL は、スピードとメモリー消費のバランスで４つの異なる AES-GCM のインプリメ

ンテーションを提供しています。可能ならば、64 ビットまたは 32 ビットの整数ライブラリーを使用します。組込みアプリケーション向けには、RAM ベースのルックアップ・テーブル (8k バイト/セッション)を使用した 64 ビット版に匹敵するスピードの高速の 8 ビット版と、追加の RAM を使用しない低速の 8 ビット版があります。--enable-aesgcm オプションは、例えば "--enable-aesgcm=table"のように、"=word32", "=table", または "=small"で修飾することができます。

aesccm – AES-CCM の有効化によって、８バイト認証（CCM-8）による CBC-MAC モードが可能となります。

aesni – AES-NI サポートを有効化すると、AES-NI がサポートされているチップの場合チップから直接に AES 命令が呼び出されます。これによって AES 関数が高速化されます。AIS-NI に関しては詳細は本マニュアル第四章を参照してください。

poly1305 – このオプションを有効化すると、wolfCrypt と wolfSSL に Poly1305 が追加されます。

camellia – このオプションを有効化すると、wolfCrypt と wolfSSL に Camellia が追加されます。

chacha – このオプションを有効化すると、wolfCrypt と wolfSSL に ChaCha が追加されます。

md2 – このオプションを有効化すると、wolfSSL に MD2 アルゴリズムが追加されます。MD２はセキュリティ脆弱性が知られており、デフォルトでは無効化されています。

ripemd – このオプションを有効化すると、wolfSSL に RIPEMD-160 のサポートが追加されます。

sha512 – このオプションを有効化すると、SHA-512 ハッシュアルゴリズムのサポートが追加されます。このアルゴリズムは word64 型が利用可能である必要があります。そのため、デフォルトでは無効化されています。組込みシステムの幾つかではこの型は利用可能と考えらます。

sessioncerts – このオプションの有効化によって、以下の
関数によるセッション・キャッシュ内の相手方の証明書チェーンのサポートが追加されます。

wolfSSL_get_peer_chain()chain(), wolfSSL_get_chain_count(), wolfSSL_get_chain_length(), wolfSSL_get_chain_cert(), wolfSSL_get_chain_cert_pem(), and wolfSSL_get_sessionID()

keygen – RSA 鍵生成サポートを有効化すると、最長 4096 ビットの各種長さの鍵を生成することができます。wolfSSL は DER および PEM フォーマットを提供します。

certgen – 自己署名の x509 v3 証明書生成サポートを有効化します。

certreq – 証明書要求の生成サポートを有効化します。

hc128 - HC-128 ストリーム暗号のスピードは非常に良いのですが、これを使用しないユーザーのためのスペースをとってしまいます。できる限り広い意味でデフォルトビルドを小さくするために、この暗号化アルゴリズムはデフォルトでは無効化しています。この暗号を使用するためには対応する暗号スイートをオンにするだけで、その他のアクションは不要です。

rabbit – このオプションを有効化すると RABBIT ストリーム暗号のサポートが追加されます。

psk –事前共有鍵サポートは、一般的には使用しないのでデフォルトではオフとなっています。この機能を有効化するためには、単にこれをオンにします。その他のアクションは不要です。

poly1305 - このオプションを有効化すると Poly1305 のサポートが wolfCrypt と wolfSSL に追加されます。

webserver – このオプションは、yaSSL 組込み Web サーバのビルドのために必要とされるすべての機能をオンにします。

noFilesystem – これによってファイルシステムの使用の無効化を容易にします。このオプションは NO_FILESYSTEM を定義します。

inline – このオプションの無効化によって wolfSSL 内の関数のインライン化が無効化されます。インライン化される場合は、関数はリンクされるのでなくそれぞれのコードブロックが関数呼び出しに挿入されインライン化されます。

ecc – このオプションを有効化すると ECC のサポートが wolfSSL に追加されます。

ocsp - このオプションを有効化すると OCSP (Online Certificate Status Protocol)のサポートが wolfSSL に追加されます。RFC6960 に説明されている X.509 証明書の取り消し状況を得るために使われます。

crl – このオプションを有効化すると CRL (Certificate Revocation List) のサポートが wolfSSL に追加されます。

crl-monitor - このオプションを有効化すると wolfSSL がアクティブに特定の CRL (Certificate Revocation List) ディレクトリを監視する機能が wolfSSL に追加されます。

ntru - このオプションを有効化すると wolfSSL で NTRU 暗号化スイーツを使用できるようになります。NTRU は今は Security Innovation から GPLv2 の元に利用可能です。NTRU バンドルは Security Innovation GitHub レポジトリからダウンロード可能です。
https://github.com/NTRUOpenSourceProject/ntru-crypto

sni – このオプションの有効化によって TLS Server Name Indication (SNI)拡張がオンになります。

maxfragment - このオプションの有効化によって TLS Maximum Fragment Length 拡張がオンになります。

truncatedhmac – このオプションの有効化によって TLS Truncated HMAC 拡張がオンになります。

supportedcurves - このオプションの有効化によって TLS Supported ECC Curves 拡張がオンになります。

tlsx - このオプションの有効化によって、すべての TLS 拡張がオンになります。

valgrind – このオプションの有効化によって、wolfSSL 単体テスト動作中 valgrant がオンになります。これは、開発サイクルにおいて問題を早期に発見するために有効です。

testcert - このオプションの有効化されると、通常開示されていない ASN 証明書 API が開示されます。これは、 wolfCrypt テストアプリケーションに見られるように (wolfcrypt/test/test.c)、テスト目的に便利です。

examples – このオプションはデフォルトで有効化されています。有効化されていると、wolfSSL サンプルアプリケーション(client, server, echoclient, echoserver) がビルドされます。

gcc-hardening – このオプションの有効化でさらにコンパイラのセキュリティチェックが追加されます。

jobserver – このオプションを有効化することで、複数のコンピュータ上の ”make” に対して複数ファイルのビルドを並行に行うための複数プロセスを可能にします。これによってビルド時間を大幅に削減することが可能です。ユーザはこのコマンドに対して異なるアーギュメント(yes/no/#)を渡すことができます。”yes”が使用されると、ジョブ数に対して CPU カウント＋１を使うよう”make” に対して伝えます。 “no” は明示的にこの機能を無効化します。ユーザはジョブの数を渡すこともできます。

shared の無効化 – 共有ライブラリーのビルド無効化オプションを有効化すると、wolfSSL 共有ライブラリーはビルドから外されます。デフォルトでは、時間とスペースを節約するために一つの共有ライブラリだけビルドされます。

static の無効化 – static の無効化オプションを有効化すると、wolfSSL 静的ライブラリーはビルドから外されます。--enable-stat ビルドオプションを使うことで、静的ライブラリをビルドすることができます。

libz – libz の有効化で、wolfSSL 内で libz ライブラリーから圧縮をサポート可能となります。このオプションを含めるかどうか、wolfSSL_set_commpression()を呼び出すかどうかについては慎重に考えてください。送信前にデータを圧縮すれば送受信される実際のメッセージサイズは減少する反面、圧縮のための解析時間はよほど遅いネットワークで送信するような場合でないかぎり生で送信したときの時間より長くかかってしまいます。

fast-rsa – fast-rsa を有効化によって、IPP ライブラリーを使用して RSA 処理を高速化します。wolfSSL デフォルトの RSA より大きなメモリを消費します。IIP ライブラリが見つからない場合、コンフィグレーション時にエラーが表示されます。Autoconf が最初に探す場所は wolfssl_root/IPP、次は Linux システムの/usr/lib のような該当マシンの標準のライブラリーの格納場所です。

RSA 処理に使用されるライブラリは “wolfssl-X.X.X/IPP” ディレクトリとなります。X.X.X は wolfSSL のバージョン番号です。バンドルされるライブラリからのビルドは、以下のサブディレクトリの IPP のファイル構成に変更がないようにディレクトリ・ロケーションと IPP の名前に依存するようになっています。

メモリーのアロケート時、fast-rsa 処理は DYNAMIC_TYPE_USER_CRYPTO メモリータグでアロケートします。これによって、fast-rsa オプションでの実行期間中の RSA 処理のメモリー消費を見ることを可能になります。

leantls – 有効化で小さなフットプリントの TLS クライアントを生成します。(クライアント認証なし) Shamir 無しの ECC256, AES128 と SHA256 だけの TLS1.2 クライアントを生成します。このオプションだけを指定し、その他のビルドオプションとの組み合わせての使用はできません。

renegotiation-indication – RFC5746 に説明されるように、この仕様は一旦成立した TLS 接続に対する再ネゴシエートに関連した SSL/TLS への攻撃を予防します。

scep – IETF(Internet Engineering Task Force) で定義されているように、SCEP (Simple Certificate Enrollment Protocol) はHTTPによるPKCS#7とPKCS#10を利用したPKIです。CERTはSCEPは証明書要求を強力に認証しないとコメントしています。

dsa – RSAやECDSAとともにNISTはDSAデジタル署名アルゴリズムをFIPS186-4に定義し、Secure Hash Standard (FIPS 180-4) として定義された承認されたハッシュ関数との組み合わせでデジタル署名の生成、検証に使用される場合、これを承認しています。

curve25519 - curve25519 オプションを有効化すると、curve25519 アルゴリズムの使用が可能となります。デフォルトの curve25519 は多量のメモリを使用し、処理速度が速くなるよう設定されています。メモリー消費を抑えるために ”--enable-curve25519=small” オプションを使用することができます。ただし、これはスピードとのトレードオフです。

ed25519 – ed25519 オプションを有効化することで、ed25519 アルゴリズムの使用が可能となります。デフォルトの ed25519 は多量のメモリを使用し、処理速度が速くなるよう設定されています。メモリー消費を抑えるために ”--enable-ed25519=small” オプションを使用することができます。ただし、curve25519 と同様に、これはスピードとのトレードオフです。

2.6 クロスコンパイル

多くの場合、組込み用プラットフォームでユーザ環境向けに wolfSSL をクロスコンパイルします。./configure システムを使用するともっとも簡単にクロスコンパイルすることができます。これによって、wolfSSL ビルドに使用することができる Makefile を生成します。
クロスコンパイルの場合、./configue に対してホストを指定する必要があります。例えば以下のように：

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

また、使用したいコンパイラー、リンカーなどの指定も必要となるかも知れません：

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

configure システムにはバグがあって、クロスコンパイル時にユーザの malloc をオーバーライドを検出すると、それに遭遇することがあるかも知れません。もし、'rpl_malloc' および/または 'rpl_realloc' の未定義エラーが起きてしまった場合は、./configure に以下を追加してください：

ac_cv_func_malloc_0_nonnull=yes ac_cv_func_realloc_0_nonnull=yes

クロスコンパイル向けに wolfSSL を正しくコンフィグレーションした後、標準の autoconf 実行でライブラリーのビルドとインストールを行ってください。

make
sudo make install

もし、wolfSSL クロスコンパイルに関してさらにヒントやフィードバックがありましたら info@wolfssl.com までお知らせください。

3. 使用方法

3.1. 全体説明

wolfSSL は本マニュアル第二章 wolfSSL のビルドで説明したオプションを使用した場合、yaSSL の約 10 分の 1、OpenSSL の 20 分の 1 以下のサイズとなります。ベンチマークとフィードバックによれば、ほとんど大多数の標準 SSL オペレーションにおいて、OpenSSL に対して wolfSSL は劇的に良い性能を示します。

ビルド手順については本マニュアル第二章 wolfSSL のビルドを参照してください。

3.2. テスト・スーツ

テストスーツ・プログラムは wolfSSL とその暗号ライブラリーwolfCrypt の能力をターゲットシステム上でテストするために設計されたものです。

wolfSSL はすべての例題とテストを wolfSSL ホーム・ディレクトリーから実行する必要があります。これは証明書と鍵を./certs から見つけるためです。テストスーツを実行するためには、以下を実行してください：

./testsuite/testsuite

または、

make test (autoconf 使用の場合)

*nix または Windows 上では、例題とテスト・スーツはカレント・ディレクトリーがソース・ディレクトリーかどうかを見てチェックし、wolfSSL ホーム・ディレクトリーに変更しようとします。これはほとんどのスタートアップのケースでうまく動作しますが、動作しない場合は、上記一つ目のほうの方法だけを使用して、フルパスを指定します。

実行が成功すると、以下のようなメッセージが出力されます：

MD5 test passed!
MD4 test passed!
SHA test passed!

SHA-256 test passed!
HMAC test passed!
ARC4 test passed!
HC-128 test passed!
Rabbit test passed!
DES test passed!
DES3 test passed!
AES test passed!
RANDOM test passed!
RSA test passed!
DH test passed!
DSA test passed!
OPENSSL test passed!
peer's cert info:
issuer : /C=US/ST=Oregon/L=Portland/O=yaSSL/CN=www.yassl.com/
emailAddress=info@yassl.com
subject: /C=US/ST=Oregon/L=Portland/O=yaSSL/CN=www.yassl.com/
emailAddress=info@yassl.com
peer's cert info:
issuer : /C=US/ST=Oregon/L=Portland/O=sawtooth/CN=www.sawtoothconsulting.
com/emailAddress=info@yassl.com
subject: /C=US/ST=Oregon/L=Portland/O=taoSoftDev/
Copyright 2012 Sawtooth Consulting Limited. All rights reserved.
CN=www.taosoftdev.com/emailAddress=info@yassl.com
Client message: hello wolfssl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
b88596cd2362310b2506f9d73693cefd input
b88596cd2362310b2506f9d73693cefd output
All tests passed!

これは、すべてのコンフィグレーションとビルドが正しく行われたことを示します。もし、何かテストがうまく行かない場合は、ビルド・システムが正しくセットアップされているかどうか確認してください。本当の原因となりそうなものとしては、誤ったエンディアン、64 ビット型が正しく設定していない、などが含まれます。何か非デフォルトの設定を使用している場合は、いったん外して wolfSSL を再ビルド、再テストしてみてください。

3.3. クライアントの例

example/client にあるクライアントの例を使用して、任意の SSL サーバに対して wolfSSL をテストすることができます。使用可能な実行時オプションの一覧は、このクライアントを—help アーギュメントで実行してください。

```
./examples/client/client --help
client 3.4.6 NOTE: All files relative to wolfSSL home dir
-?              Help, print this usage
-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 [0-3], SSLv3(0) - TLS1.2(3)), default 3
-V              Prints valid ssl version numbers, SSLv3(0)-TLS1.2(3)
-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 tats
-s              Use pre Shared keys
-t              Track wolfSSL memory use
-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
```

セキュアな Gmail に対してテストするには、以下のようにしてみてください。これは、--enable-opensslextra --enable-supportedcurves オプションでビルドしたものを使用します。

```
./examples/client/client -h gmail.google.com -p 443 -d -g
peer's cert info:
issuer : /C=US/O=Google Inc/CN=Google Internet Authority G2
```

subject: /C=US/ST=California/L=Mountain View/O=Google Inc/CN=*.google.com
altname = youtubeeducation.com
altname = youtube.com
altname = youtu.be
altname = www.goo.gl
altname = urchin.com
altname = googlecommerce.com
altname = google.com
altname = google-analytics.com
altname = goo.gl
altname = g.co
altname = android.com
altname = android.clients.google.com
altname = *.ytimg.com
altname = *.youtubeeducation.com
altname = *.youtube.com
altname = *.youtube-nocookie.com
altname = *.url.google.com
altname = *.urchin.com
altname = *.metric.gstatic.com
altname = *.gvt2.com
altname = *.gvt1.com
altname = *.gstatic.com
altname = *.gstatic.cn
altname = *.googlevideo.com
altname = *.googlecommerce.com
altname = *.googleapis.cn
altname = *.googleadapis.com
altname = *.google.pt
altname = *.google.pl
altname = *.google.nl
altname = *.google.it
altname = *.google.hu
altname = *.google.fr
altname = *.google.es
altname = *.google.de
altname = *.google.com.vn
altname = *.google.com.tr
altname = *.google.com.mx
altname = *.google.com.co
altname = *.google.com.br
altname = *.google.com.au
altname = *.google.com.ar
altname = *.google.co.uk

altname = *.google.co.jp
altname = *.google.co.in
altname = *.google.cl
altname = *.google.ca
altname = *.google-analytics.com
altname = *.cloud.google.com
altname = *.appengine.google.com
altname = *.android.com
altname = *.google.com
serial number:7e:4e:f0:c5:a2:31:59:e8
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
SSL connect ok, sending GET...
Server response: HTTP/1.0 404 Not Found
Content-Type: text/html; charset=UTF-8
Content-Length:
1571
Date: Fri, 03 Jun 2016 16:04:32 GMT

これは、クライアントが(-h) gmail.google.com に https のポート(-p) 443 でコネクトするために、通常の GET を送信（(-g) していることを示します。(-d) オプションはクライアントにサーバ認証を行わないよう指示するものです。以下はリードバッファへのサーバからの最初に出力です。

コマンドラインのアーギュメントが与えらえない場合、クライアントは localhost の wolfSSL のデフォルトポート 11111 にコネクトしようとします。サーバーがクライアント認証を要求する場合は、クライアント証明書もロードします。

“-b <num>” アーギュメントを使用して、クライアントは接続のベンチマークをすることが可能です。クライアントは特定のサーバ/ポートに対して、アーギュメントで指定された回数の接続を試み、SSL_connect()関数の処理の平均時間をミリ秒で表示します。

./examples/client/client -b 100
SSL_connect avg took: 0.653 milliseconds

デフォルト・ホストを localhost から変更したい場合、またはデフォルト・ポートを 11111 から変更したい場合は、これらの設定を/wolfssl/test.h で変更することができま

す。wolfsslIP と wolfsslPort 変数はこれらの設定をコントロールしています。これらの設定を変更した場合は、テスト・スーツを含むすべての例題を再ビルドしてください。そうしないと、テスト・プログラムはお互いにコネクトできなくなってしまいます。

デフォルトでは wolfSS サンプルクライアントは指定されたサーバに対して TLS1.2 を使用して接続を試みます。”-v”オプションを使用することで、SSL/TLS のバージョンを変更することができます。このオプショんで以下の値が利用可能です：

-v 0 = SSL 3.0 (デフォルトでは無効化されています)
-v 1 = TLS 1.0
-v 2 = TLS 1.1
-v 3 = TLS 1.2

サプンプルプログラムを使用していて、よくあるエラーとして　-155 があります。

err = -155, ASN sig error, confirm failure

これは通常、wolfSSL クライアントが接続しようとするサーバの証明書の認証ができないことが原因です。デフォルトでは wolfSSL クライアントは、yaSSL test CA 証明書を信頼するルート証明書としてロードします。この test CA 証明書は、異なる CA によって署名された外部のサーバ証明書を認証することはできません。このように、この問題を解決するには、”-d”オプションを使って相手がた(server) の認証をオフにするか、

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

“-A” コマンドライン・オプションを使って、正しい CA 証明書を wolfSSL クライアントにロードするか、しなければなりません。

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

3.4. サーバの例

サーバサンプルプログラムでは、オプションでクライアント認証をすることもできる簡単な SSL サーバをデモンストレートします。一つのクライアント接続だけをアクセプトし、サーバは停止します。通常モード（コマンドライン・アーギュメントなし）のクライアントサンプルプログラムはサーバサンプルに対して正常に動作しますが、クライアントプログラムに対してコマンドライン・アーギュメントを指定することで、クライアント証明書をロードさせないと wolfSSL_connect()は（サーバ側で ”-d”クライアント認証を無効化していない限り）異常終了します。サーバは “-245, peer didn’t send cert (相手がたが証明書を送ってこなかった)” をレポートします。クライアント・サンプルプログラムと同様に、サーバはコマンドライン・アーギュメントを使用することができます。

./examples/server/server --help

server 3.4.6 NOTE: All files relative to wolfSSL home dir

-?	Help, print this usage
-p <num>	Port to listen on, not 0, default 11111
-v <num>	SSL version [0-3], SSLv3(0) - TLS1.2(3)), 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
-t	Track wolfSSL memory use
-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
-i	Loop indefinitely (allow repeated connections)
-e	Echo data mode (return raw bytes received)
-B <num>	Benchmark throughput using <num> bytes and print stats

3.5. EchoServer の例

EchoServer の例では、無限ループで無限回のクライアント・コネクションを待ちます。クライアントが何を送ろうと、echoserver はそれをエコーバックします。クライアント例が echoserver に対して３つすべてのモードを使用できるように、クライアント認証は実行されないようになっています。以下の４つの特別なコマンドについてはエコーバックされず、echoserver に特定のアクションをとるように指示します。

1. “quit” echoserver が文字列 “quit” を受信した場合、ショットダウンします。
2. “break” echoserver が文字列 “break” を受信した場合、カレントのセッションを停止しますが、リクエストの処理は継続します。これは特に DTLS のテストのために便利です。
3. “printstats” echoserver が文字列 “printstats” を受信した場合、セッションキャッシュについての統計情報を出力します。
4. “GET” echoserver が文字列 “GET” を受信した場合、http get として扱い、 “greeting from wolfSSL”メッセージの簡単なページを送り返します。これによって、Safari, IE, Firefox, gnutls その他の各種 TLS/SSL クライアントが echoserver の例に対してテストすることができます。

NO_MAIN_DRIVER が定義されていない限り、echoserver の出力は標準出力に出力されます。シェルやコマンドラインの第一アーギュメントで出力先を変更することができます。Echoserver の出力で Output.txt という名前のファイルに出力するには以下のようなコマンドを実行します。

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

3.6. EchoClient の例

echoclient の例はインタラクティブ・モードまたはファイルとともにバッチ・モードで実行することができます。インタラクティブ・モードで実行して、”hell”、“wolfssl”と“quit” の３つの文字列を書くと、以下のようになります：

```
./examples/echoclient/echoclient
hello
hello
```

wolfssl
wolfssl
quit
sending server shutdown command: quit!

入力ファイルを使用するには、ファイル名をコマンド行の第一アーギュメントに指定します。ファイル input.txt の内容をエコーさせる場合、以下のように発行してください：

./examples/echoclient/echoclient input.txt

結果をファイルに出力したい場合、出力ファイル名を次のコマンド行アーギュメントに指定することができます。以下のコマンドは、input.txt ファイルの内容をエコーし、output.txt ファイルにサーバーからの結果を書き出します。

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

テストスーツプログラムは入力をハッシュし、出力ファイルに書き出し、クライアントとサーバが正しく期待された結果を送受していることを確認しているだけです。

3.7. ベンチマーク

多くのユーザの皆様は wolfSSL 組込み SSL ライブラリーが特定のハードウェアの上、もしくは特定の環境下でどのような性能になるのか興味をお持ちかと思います。今日の組込み、企業システム、またクラウド・ベースの環境などを使った、非常に多様なプラットフォームやコンパイラーに対して、さまざまなボード相互の一般的な性能評価は難しくなっています。

wolfSSL/wolfCrypt の SSL 性能評価をされる wolfSSL ユーザの皆様のために、ベンチマーク・アプリケーションが wolfSSL にバンドルされ提供されています。wolfSSL はデフォルトではすべての暗号操作に wolfCrypt 暗号ライブラリーを使用しています。ベースとなってる暗号は SSL/TLS の性能に非常にクリティカルな側面なので、我々のベンチマーク・アプリケーションは wolfCrypt の各アルゴリズムに対して性能テストを実行しています。

wolfcrypt/benchimark (./wolfcrypt/benchmar/benchmark) にあるベンチマーク・ユーティリティーをwolfCryptの暗号化機能のベンチマークに使用することができます。典型的な出力は下記のような感じになります（この出力例では、HC-128, RABBIT, ECC, SHA-256, SHA-512, AES-GCM, AES-CCMおよびCamelliaを含むオプショナルなアルゴリズム、暗号を含んでいます。）：

./wolfcrypt/benchmark/benchmark

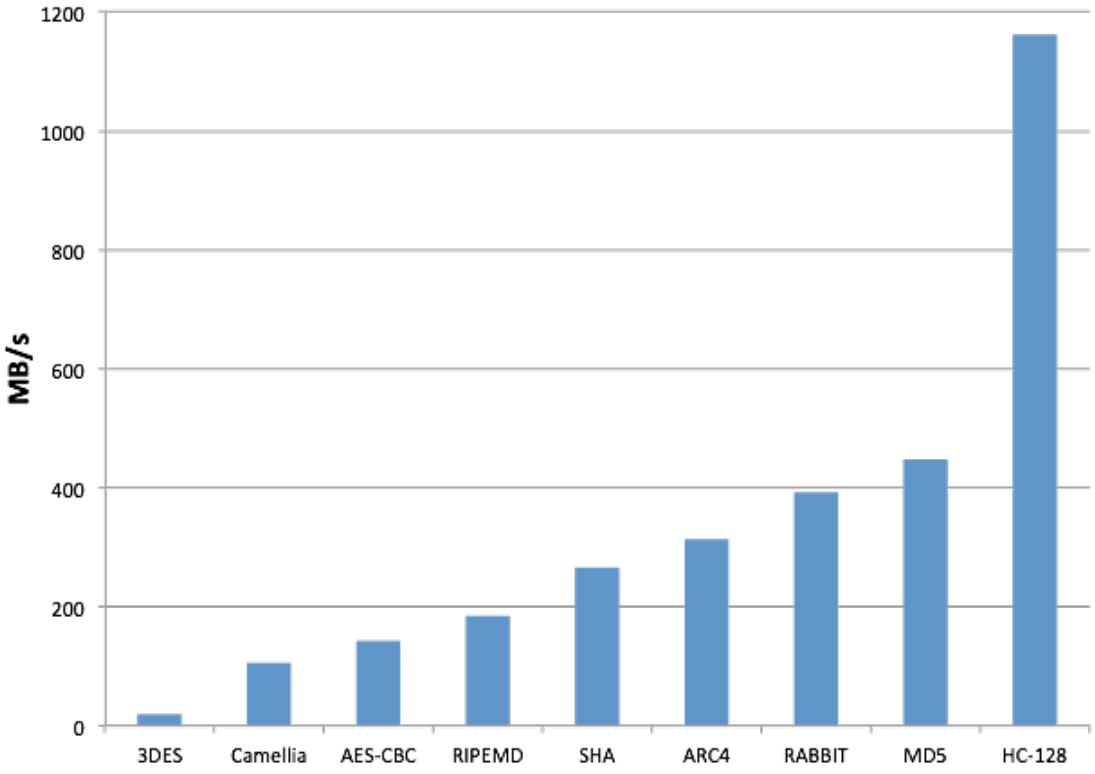
```
RNG             50 megs took 0.570 seconds, 87.769 MB/s Cycles per byte = 28.19
AES enc         50 megs took 0.361 seconds, 138.582 MB/s Cycles per byte = 17.85
AES dec         50 megs took 0.341 seconds, 146.649 MB/s Cycles per byte = 16.87
AES-GCM         50 megs took 2.924 seconds, 17.099 MB/s Cycles per byte = 144.68
CHACHA          50 megs took 0.214 seconds, 233.603 MB/s Cycles per byte = 10.59
CHA-POLY        50 megs took 0.263 seconds, 190.226 MB/s Cycles per byte = 13.00
3DES            50 megs took 2.109 seconds, 23.703 MB/s Cycles per byte = 104.37
MD5             50 megs took 0.093 seconds, 539.136 MB/s Cycles per byte = 4.59
POLY1305        50 megs took 0.057 seconds, 881.508 MB/s Cycles per byte = 2.81
SHA             50 megs took 0.155 seconds, 322.495 MB/s Cycles per byte = 7.67
SHA-256         50 megs took 0.262 seconds, 191.039 MB/s Cycles per byte = 12.95
SHA-384         50 megs took 0.173 seconds, 289.818 MB/s Cycles per byte = 8.54
SHA-512         50 megs took 0.215 seconds, 232.494 MB/s Cycles per byte = 10.64
RSA             2048 encryption took 0.087 milliseconds, avg over 100 iterations
RSA             2048 decryption took 2.055 milliseconds, avg over 100 iterations
DH              2048 key generation 0.746 milliseconds, avg over 100 iterations
DH              2048 key agreement 0.757 milliseconds, avg over 100 iterations
ECC 256         key generation 0.552 milliseconds, avg over 100 iterations
EC-DHE          key agreement 0.528 milliseconds, avg over 100 iterations
EC-DSA          sign time 0.558 milliseconds, avg over 100 iterations
EC-DSA          verify time 0.721 milliseconds, avg over 100 iterations
```

これは特に、数学ライブラリーの変更前後の公開鍵のスピードを比較するのに有効です。通常の数学ライブラリー(./counfigure)、fastmath (./configure --enable-fastmath) またはfasthugemath ライブラリー (./configure --enable-fasthugemath) を使用した結果をテストすることができます。

さらに詳細とベンチマーク結果については、wolfSSL のベンチマーク・ページを参照してください（http://www.wolfssl.com/yaSSL/benchmarks-wolfssl.html）。

3.7.1 相対的性能

個々の暗号化とアルゴリズムの性能はホストとなるプラットフォームに依存しますが、次のグラフは wolfCrypt の暗号間での相対的性能を示しています。これらのテストは MacbookRro (OSx 10.6.8) Intel Core i7, 2.2GHz で実行されたものです。



SSL/TLS 接続時に特定のスイートのみに限定したい場合、wolfSSL が使用する暗号化スイーツをカスタマイズすることができます。wolfSSL_CTX_new(SSL_CTX_net)で次のように指定します。

wolfSSL_CTX_set_cipher_list(ctx, “AES128-SHA”);

3.7.2 ベンチマークに関するコメント

1. プロセッサのネイティブのレジスタサイズ(32 対 64 ビット)は 1000 ビット以上の公開鍵の処理を行う場合大きな違いをもたらします

2. keygen (--enable-keygen) オプションで、ベンチマーク・ユーティリティの実行時に鍵生成のスピードをベンチマークすることもできます。

3. fastmath (--enable-fastmath) オプションにより公開鍵処理における動的メモリーを削減、処理速度が改善します。32 ビットプラットフォーム上で fastmath のビルドに問題がある場合、PIC が一つのレジスタをホギングしないように共有ライブラリを無効化します。

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

注：wolfSSL でコンフィグ・オプションを変更した場合、”make clean” することをお勧めします。

4. デフォルトでは、fastmath は可能な限りアセンブリー最適化を使用します。アセンブリー最適化が機能しない場合も、wolfSSL のビルド時に FTM_NO_ASM を CFLAGS に加えることでアセンブリー最適化無しでビルド可能です。

```
./configure --enable-fastmath CFLAGS=DTFM_NO_ASM
```

5. 組込みプラットフォームでの実行ではないユーザは、masthugemath オプション使用で fastmath をさらにプッシュすることが可能です。

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

6. wolfSSL のデフォルトビルドで、メモリー使用と性能の良いバランスについてこれまで説明の通り試行できます。どちらか一方により関心が高いユーザは第 2 章に戻って、さらなる wolfSSL コンフィグレーション・オプションを参照ください。

7. バルク転送：wolfSSL のデフォルトでは、128 バイトの I/O バッファを使用します。およそ 80%の SSL トラフィックはこのサイズにおさまり、動的メモリの使用を抑えるといわれています。バルク転送が要求される場合は、16K バイト（SSL の最大サイズ）を使用するよう構成が可能です。

3.7.3 組込みシステムに対するベンチマーク

組込みシステムでのベンチマークアプリケーションのビルドを容易にするために幾つかのビルドオプションが用意されています。

BENCH_EMBEDDED – このオプションの有効化でベンチマークアプリケーションをメガバイトからキロバイドを使用するように切り替え、メモリ使用量を削減します。このオプションを使用した場合、デフォルトでは暗号とアルゴリズムは 25k バイトでベンチマークされます。公開鍵アルゴリズムは 1 回の繰り返しのみ（幾つかの組みこみプロセッサでの公開鍵処理はかなり遅いので）に対してベンチマークされます。これらは benchmark.c の BENCH_EMBEDDED 定義内の “numBlocks” と “times” 変数を取り替えることで調整可能です。

USE_CERT_BUFFER?1024 - この定義を有効化することで、ベンチマークアプリケーションはテスト用の鍵と証明書をファイルシステムからのロードではなく、<wolfssl_root>?wolfssl/cert_test.h にある 1024 ビット鍵と証明書バッファを使用するように切り替わります。これは、組込み向けプラットフォームにファイルシステムが無い場合、2048 ビット公開鍵処理が妥当でない遅いプロセッサに有効です（NO_FILESYSTEM と共に使用)。

USE_CERT_BUFFER_2048 - この定義を有効化すると USE_CERT_BUFFER_1024 と似ていますが、 1024 ビットではなく 2048 ビットの鍵と証明書バッファを受け付けます。この定義はプロセッサの処理が 2048 ビット公開鍵処理のために十分速位けれど鍵と証明書をロードしてくるためのファイルシステムがない場合、有効です。

3.8. クライアント・アプリケーションを wolfSSL 用に修正

このセクションでは、wolfSSL のネイティブ API を使用してクライアント・アプリケーションに wolfSSL を加えるのに必要な基本的ステップを説明します。サーバ側の例については次のセクション 9 を参照してください。「wolfSSL SSL チュートリアル」ではサンプルコードを使ったより詳細なウォークスルーを掲載しています。OpenSSL 互換レイヤーの使用を希望する場合は、ユーザ・マニュアル第十三章を参照してください。

1. wolfSSL ヘッダーをインクルードする
   #include <wolfssl/ssl.h>

2. すべての read() (または recv())関数呼び出しを wolfSSL_read() に変更する。

result = read(fd, buffer, bytes);

は

result = wolfSSL_read(ssl, buffer, bytes);

となります。

3. すべての write (または send) 関数呼び出しを wolfSSL_write()に変更する。

result = write(fd, buffer, bytes);

は

result = wolfSSL_write(ssl, buffer, bytes);

となります。

4. wolfSSL_connect() を手動で呼び出すこともできますが、必須ではありません。最初の wolfSSL_read() または wolfSSL_write() 呼び出しで、まだならば wolfSSL_connect() が呼び出されます。
5. wolfSSL と WOLFSSL_CTX を初期化します。WOLFSSL オブジェクトをいくつ生成する場合でも、一つの WOLFSSL_CTX を使用できます。基本的に、コネクトしようとするサーバーに対して証明するための CA 証明書をロードしなければならないだけです。基本的な初期化は以下のような感じになります:

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
if ( (ctx = wolfSSL_CTX_new(CyaTLSv1_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);
}
```

6. 各 TCP コネクトし、ファイル・ディスクリプタをセッションに関連付けた後、WOLFSSL を生成します:

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

7. エラーチェック。各 wolfSSL_read() または wolfSSL_write() 呼び出しは、read()または write()と同じように、成功した書き込みバイト数、コネクション・クローズに対して０、エラーに対して－１を返します。エラー時に、以下のような二つの関数を使って、エラー情報をさらに取得できます:

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

ノンブロック型ソケットを使用している場合は、EAGAIN / EWOULDBLOCK でエラーを、あるいは、より正確には特定のエラー・コードを SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE でテストすることができます。

8. 後処理。各 WOLFSSL オブジェクトを使用した後、下記の呼び出しでクリーンアップすることができます:

```
wolfSSL_free(ssl);
```

SSL/TLS の使用をすべて完全に完了した場合は、WOLFSSL_CTX を以下のように呼び出して解放することができます:

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

3.9. サーバ・アプリケーションを wolfSSL 用に変更

このセクションでは、wolfSSL のネイティブ API を使用してサーバ・アプリケーションに wolfSSL を加えるのに必要な基本的ステップを説明します。クライアント側の例につい

ては前のセクション８を参照してください。「wolfSSL SSL チュートリアル」ではサンプル・コードを使ったより詳細なウォークスルーを掲載しています。

1. 前述のクライアント向けの説明に従ってください。ただし、ステップ５のクライアント・メソッドはサーバ・メソッドに読み替えてください。

   クライアントに SSLv3 と TLSv1+でサーバーコネクトを許可するには、
   wolfSSL_CTX_new(wolfTLSv1_client_method())
   を
   wolfSSL_CTX_new(wolfTLSv1_server_method())
   または
   wolfSSL_CTX_new(wolfSSLv23_server_method())
   とします。

2. 上記ステップ５の初期化に、サーバーの証明書と鍵ファイルを加えます。

```
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);
}
```

   ファイルシステムが無い場合、証明書と鍵をバッファからロードすることも可能です。
   wolfSSL_CTX_use_certificate_buffer() とwolfSSL_CTX_use_PrivateKey_buffer() につてい詳しくはAPIドキュメントを参照してください。

   wolfSSL を使ったサーバアプリケーションの例については、
   <wolfsssl_root>/examples/serve.cファイルにあるサーバ・サンプルを参照してください。

4. 機能

wolfSSL は基本インタフェースとして C 言語をサポートしますが、そのほかにも Java, PHP, Perl および Python（swig インタフェース経由）などを含むホスト言語をサポートします。その他のプログラミング言語から wolfSSL を利用されることをご検討の際は弊社までお問合わせください。

この章では、ストリーム暗号化、AES-NI、IPv6、SSL 検査のサポートなど wolfSSL のいくつかの機能について、より詳細に説明します。

4.1 概要

以下の表に wolfSSL リリースに含まれる機能、特徴の一覧を示します。

wolfSSL機能、特徴（Ver 2.0.8）	利点
SSLバージョン3、TLS1、1.1、1.2 （クライアントおよびサーバ）	バックワード互換性を維持しつつ、最新の標準をサポート
DTLS1．0、1．2サポート（クライアントとサーバ）	ストリーム・メディア
ビルド・オプションと実行環境により、最小20から100kBのコードサイズ	リソースの制約環境のもとで使用するために、小さな構成サイズを実現
1-36kBの実行時メモリー（I/Oバッファサイズ、公開鍵アルゴリズム、鍵サイズに依存）	最小の動的メモリー使用、組込みシステム、スケーラブルな企業向けサーバに最適。
OpenSSL互換レイヤー	標準APIによりOpenSSLからの移行を容易化
OCSPとCRLサポート	証明書の有効性検証
ハッシュ関数サポート：	AES (CBC, CTR, GCM, CCM-8), Camellia, DES, 3DES, ARC4, RABBIT, HC-128, ChaCha20
ブロックとストリーム暗号化	AES (CBC, CTR, GCM, CCM-8), Camellia, DES, 3DES, ARC4, RABBIT, HC-128,ChaCha20
パスワードベース鍵生成	HMAC, PBKDF2, PKCS #5

ECCサポート	ECDH-ECDSA, ECDHE-ECDSA, ECDHRSA, ECDHE-RSA, ECDHE-PSK
RSA鍵生成	高速な実行時鍵生成
クライアント認証サポート	クライアントの認証機能で、相互認証の実現
PSK（事前共有鍵）サポート	制約された環境下でRSAオペレーションを避けることが可能
簡易なAPI	導入、使用が容易なAPI
MySQLインテグレーション	大規模なディストリビューションとテスト
zlib圧縮サポート	高度に構成可能な圧縮サポート
PEMおよびDER形式の証明書をサポート	証明書または鍵の再構成が不要
X509 v3署名の証明書生成	自分自身の証明書生成
証明書管理	証明書の検証、SSLの外でのCRLチェック
インテルAES-NIサポート	超高速のチップレベルのAES暗号化
STM32F2/4ハードウェア暗号サポート	STM32プロセッサの暗号高速化
Cavium NITROXサポート	Cavium NITROXを使用した暗号化とSSLの高速化
スニファー（SSL検査）サポート	SSL暗号化パケットを容易にデコード
抽象化レイヤー ・ C言語ライブラリー抽象化レイヤー ・ カスタムI/O ・ メモリーフック ・ ロギング・コールバック ・ ユーザ・アトミックレコード層処理 ・ 公開鍵(RSA, ECC) コールバック	開発者に移植性と柔軟性を提供

・ ・ OS抽象化レイヤー ・ カスタムI/O抽象化レイヤー	
IPv4およびIPv6をサポート	現行と今後のプロトコル互換性
PKCS#8 (PKCS#5, #12フォーマット)	非公開鍵暗号化
MySQLインテグレーション	幅広いディストリビューションとテスト
Webサーバのサポート – yaSSLEWS, GoAhead, Mongoose, Lighttpd等々	複数の軽量組込みWebサーバの選択肢。 wolfSSLは弊社yaSSL組込みWebサーバでも使用

表1:wolfSSL の機能

4.1.2 AEAD スイーツ

wolfSSL は、AES-GCM, AES-CCM, および CHACHAPOLY1305 を含む AEADw をサポートします。AEAD スイーツとそれ以外の大きな違いは、AEAD では暗号化されたデータを認証する点です。これによって、中間者攻撃によるデータ改ざんを防ぐのに有効です。AEAD スイーツではブロック暗号（最近はストリーム暗号化とも）アルゴリズムが鍵つきハッシュアルゴリズムと組み合わされます。これら二つのアルゴリズムを組み合わせは wolfSSL 暗号化と復号化処理によって扱われることで、ユーザに取って使いやすいものいなっています。特定の AEAD スイーツを使用するために必要なものは、単にサポートされているそのアルゴリズムを有効化するだけです。

4.2 プロトコル・サポート

wolfSSL は SSL3.0, TLS(1.0, 1.1 および 1.2)および DTLS（1.0 と 1.2）をサポートします。ユーザは以下の関数のうちの一つを利用して使用するプロトコルを容易に選択することができます（クライアントまたはサーバとして）。wolfSSL は、セキュリティー上問題があるため SSL2.0 はサポートしていません。以下のクライアントおよびサーバ関数は OpenSSL 互換レイヤーを利用する場合、若干違いがあります。OpenSSL 互換関数については、ユーザ・マニュアルの第 13 章を参照ください。

4.2.1サーバ関数

wolfDTLSv1_server_method(void); /*DTLS 1.0 */
wolfDTLSv1_2_server_method(void); /*DTLS 1.2 */
wolfSSLv3_server_method(void); /* SSL 3.0 */
wolfTLSv1_server_method(void); /* TLS 1.0 */
wolfTLSv1_1_server_method(void); /* TLS 1.1 */
wolfTLSv1_2_server_method(void); /* TLS 1.2 */
wolfSSLv23_server_method(void); /* SSLv3 - TLS 1.2のうち利用可能な最も高いバージョンを利用 */

wolfSSLはwolfSSLv23_server_method()関数を利用して堅牢なサーバ・ダウングレードをサポートします。詳細は2.3を参照してください。

4.2.2 クライアント関数

wolfDTLSv1_client_method(void); /*DTLS 1.0 */
wolfDTLSv1_2_client_method(void); /* DTLS 1.2 */
wolfSSLv3_client_method(void); /* SSL 3.0 */
wolfTLSv1_client_method(void); /* TLS 1.0 */
wolfTLSv1_1_client_method(void); /* TLS 1.1 */
wolfTLSv1_2_client_method(void); /* TLS 1.2 */
wolfSSLv23_client_method(void); /*SSLv3 - TLS 1.2のうち利用可能な最も高いバージョ

ンを利用*/

wolfSSLはwolfSSLv23_client_method()関数を利用して堅牢なクライアント・ダウングレードをサポートします。詳細は2.3を参照してください。

これらの関数の利用方法の詳細は本マニュアルの”第三章:使用方法”参照してください。また、SSL3.0, TLS1.0, 1.1, 1.2およびDTLSの比較についてはユーザ・マニュアルのAppendix Aを参照してください。

4.2.3堅牢なクライアントおよびサーバ・ダウングレード

wolfSSL クライアントおよびサーバは双方とも堅牢なバージョン・ダウングレード機能を持っています。もし、ある特定のプロトコル・バージョンのメソッドがどちらかの側で使用されていると、そのバージョンだけがネゴシエートされるか、エラーが返却されてしまいます。たとえば、クライアントが TLSv1 を使用していて、SSL v 3 のみのサーバにコネクトしようとすると失敗となってしまいます。同じように、TLS1.1 にコネクトしようとしても失敗となってしまいます。

この問題を解決するために、wolfSSLv23_client_method()関数を使用するクライアントはサーバ側でサポートされているもっとも高いプロトコル・バージョンを利用し、必要ならば SSLv3 までダウングレードします。この場合、クライアントは SSLv3 から TLSv1.2 が動作しているサーバにコネクトすることができます。長年セキュリティー上問題があるとされる SSLv2 にだけはコネクトすることができません。

似たように、wolfSSLv23_server_method()を使用するサーバは SSLv3 から TLSv1.2 のプロトコル・バージョンをサポートするクライアントを取り扱うことができます。wolfSSL サーバはセキュリティー上問題のある SSLv2 からのコネクションについては受け入れません。

4.2.4 IPv6 サポート

IPv6 対応で組込み SSL を利用したいユーザは、wolfSSL が IPv6 サポートかどうか疑問に思われているかもしれません。答は Yes。wolfSSL は IPv6 上で動作します。

wolfSSL は IP 中立に設計されており、IPv4 でも IPv6 でも動作しますが、現行のテストアプリケーションでは IPv4 をデフォルトとしています（他の多くのシステムと同様に）。テスト・アプリケーションを IPv6 に変更するには、wolfSSL ビルド時に--enable-ipv6 オプションを使用してください。

IPv6 に関する詳しい情報は：http://en.wikipedia.org/wiki/IPv6

4.2.5 DTLS

上のリストに挙げたように、wolfSSL はクライアント、サーバ両方の DTLS（「データグラム」TLS）をサポートします。現在のサポートバージョンは DTLS1.0 です。

TLS プロトコルは（TCP のように）信頼性のある媒体におけるセキュアなトランスポート・チャネルを提供するよう設計されました。アプリケーション層のプロトコルが（SIP や各種ゲーム・プロトコルのように）UDP トランスポートを使用して開発されはじめるに従って、通信遅れに対して敏感なアプリケーションのための通信セキュリティーを提供する方法に対するニーズが高まってきました。そのようなニーズが DTLS プロトコルの誕生を導きました。

多くの人々は TLS と DTLS の違いは TCP と UDP と同様だと理解していますが、これは誤解です。UDP は（TCP と比べ）ハンドシェイク無し、通信切れサポート無し、またはパケットロスに対する遅れ無しなどの利点があります。一方、DTLS は拡張された SSL ハンドシェイク、通信切れに対するサポート、また、ハンドシェイクに対して TCP のような挙動を実現しなければなりません。つまり、DTLS は UDP が信頼できるコネクションと引き換えに提供する利点を無効にしてしまっています。

--enable-dtls ビルド・オプションを使用して wolfSSL をビルドすることで、DTLS を有効化することができます。

4.2.6 Lightweight Internet Protocol (lwIP)

wolfSSL はそのまま変更なしで Lightweight Internet Protocol (lwIP)をサポートします。このプロトコルを使用するためには単に WOLFSSL_LWIP を定義するだけです。または、 settings.h ファイルにて次の行のコメントを外します。

/*#define WOLFSSL_LWIP*/

lwIP はフルの TCP スタックを提供しながら RAM 消費量の削減にフォーカスしています。このフォーカスは wolfSSL の SSL/TLS におけるフォーカスと理想的なマッチングです。

4.3 暗号化サポート

4.3.1暗号化スーツ強度と適切な鍵サイズの選択

現在どの暗号化が使用されているかを見るには、次のメソッドを呼び出すことができます。

wolfSSL_get_ciphers()

この関数はカレントの暗号化スイーツを返却します。

暗号化スーツはそれぞれ異なった強度を実現します。異なる種類のアルゴリズム（認証、暗号化おおよびメッセージ認証コード（MAC））で作られるため選択する鍵サイズによりそれぞれの強度が異なることになります。暗号化スーツの強度に関してグレード付けする方法は様々で、対照型か公開鍵アルゴリズムの鍵サイズ、アルゴリズム種別、性能、または既知の脆弱性などとも絡んで、プロジェクトや企業によっても異なってくるようです。

NIST（Nathional Institute of Standards and Technology）はそれぞれの異なる鍵サイズに対して比較可能なアルゴリズムの強度を提供することで、採用可能な暗号化スーツ選択についてレコメンデーションを作成しています。暗号化アルゴリズムの強度はアルゴリズムに使用される鍵サイズに依存します。NIST Special Publication、SP800-57 では、以下のように、二つのアルゴリズムは等価な強度を持つと述べています。

「…二つのアルゴリズムは、与えられた鍵サイズ（X と Y）に対して『アルゴリズムを破る』または鍵（与えられた鍵サイズにおいて）を決定するのに必要とされた仕事量が与えられた資源を使用してほぼ同じであるならば、互換の強度と考えられる。与えられた鍵サイズにおけるあるアルゴリズムのセキュリティー強度は、伝統的に、対照アルゴリズムのある鍵サイズ“X”がショートカット・アタックを持っていない場合（すなわち、すべての可能な鍵を試すことが最も効率が良いような場合）においてすべての鍵を試すのに必要な仕事量として説明される。」

次の二つの表は NIST SP800-57 の表 2（64 ページ）と表 4（66 ページ）から引用して、（NIST の、セキュリティーのビットを使用するときのセキュリティー寿命に対する推奨をベースに）アルゴリズムと強度の計測とともにアルゴリズム間のセキュリティー強度比較を示したものです。

注：以下の表で“L”は有限体暗号（FFC）のための公開鍵のサイズ、“N”は FFC のための秘密鍵のサイズ、“k”は素因数分解暗号化（IFC）のための鍵サイズ、“f”は楕円曲線暗号化のための鍵サイズです。

セキュリティービット数	対照鍵アルゴリズム	FFC 鍵サイズ (DSA, DH, etc)	IFC 鍵サイズ (RSA, etc.)	ECC 鍵サイズ (ECDSA, etc)
80	2TDEA ほか	L = 1024 N = 160	k = 1024	F = 160 - 223
128	AES-128 ほか	L = 3072 N = 256	k =3072	F = 256 - 383
192	AES-192 ほか	L = 7680 N = 384	k =7680	F = 384 - 511
256	AES-256 ほか	L = 15360 N = 512	k =15360	F = 512 +

表２：相対ビットと鍵強度

セキュリティービット	説明
80	2010 年まで有効
128	2030 年まで有効
192	長期間プロテクション
256	予見できる限りセキュア

表 3 ：ビット強度の説明

この表をガイドとして使用し暗号化スーツの分類を始めるため、対照鍵暗号化アルゴリズムの強度を基本に分類しました。このようにして、おおざっぱなグレード分類をセキュリティーのビット数をベースに暗号化スーツ毎に分類することができます（対照鍵暗号の鍵サイズを考慮にいれるだけで）。

強度「低」　=　128 ビット未満のセキュリティー
強度「中」　=　128 ビット程度のセキュリティー
強度「高」　=　128 ビット以上のセキュリティー

対照鍵暗号の強度の外では、暗号化スーツの強度は鍵交換と認証アルゴリズム鍵の鍵サイズに多く依存しています。その強度は暗号化スーツのもっとも弱いリンクと同程度となります。

上記のグレーディング手法（対照鍵暗号アルゴリズムの強度だけに基づいた）に従って、wolfSSL2.0.0 では強度「低」の暗号化スーツ 0 個、強度「中」の暗号化スーツ 12 個、強度「高」の暗号化スーツ 8 個をサポートしています（この後に示す通り）。この強度分類は関係する他のアルゴリズムに選択された鍵サイズに依存して変わる可能性があります。ハッシュ関数のセキュリティー強度に関しては NISTSP800-57 の表 3 （64 ページ）を参照してください。

いくつかのケースで、「輸出」暗号化と示される暗号化を目にするかと思います。これらの暗号化は、（1992 年以前の）合衆国の歴史上、米国から強固な暗号化を持つソフトウェアの輸出は違法であったころに端を発しています。強固な暗号化は米国政府によって「軍需品」（核兵器、戦車、弾道ミサイルなどと同じ）として分類されていました。この

制限により、輸出されるソフトウェアには「弱められた」暗号化（おもに小さな鍵サイズで）を添付していました。今日、この制限は撤廃され、「輸出」暗号化のようなものは必要とされなくなりました。

4.3.2サポートされる暗号化スーツ

以下の暗号化スーツが wolfSSL によってサポートされます。暗号化スーツは認証、暗号化、コネクションの設定のために TLS または SSL のハンドシェークで利用されるメッセージ認証コード（MAC）アルゴリズムの組み合わせになります。

それぞれの暗号化スーツは鍵交換アルゴリズム、バルク暗号アルゴリズム、およびメッセージ認証コード・アルゴリズムを定義します。**鍵交換アルゴリズム**（RSA、DDS、DH、EDH）は、クライアントとサーバがハンドシェイク・プロセスの間に認証する方法について決定します。ブロック暗号およびストリーム暗号を含む**バルク暗号アルゴリズム**（DES、３DES、AES、ARC4、RABBIT、HC-128）はメッセージ・ストリームを暗号化するのに使用されます。**メッセージ認証コード**(MAC)**アルゴリズム**(MD2、MD5、SHA-1SHA-256、SHA-512、RIPEMD)はメッセージ・ダイジェストを生成するためのハッシュ関数です。

以下の表は<wolfssl_root>/wolfssl/internal.h（706 行目あたりから)の暗号スイーツに対応しています。もし表に探している暗号スイートが見つからない場合は追加に関して wolfSSL にコンタクト下さい。

wolfSSL 暗号スイーツ (3.4.6 版)	
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	ECC 暗号化スイーツ
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	静的 ECDE 暗号化スイーツ
TLS_RSA_WITH_HC_128_MD5 TLS_RSA_WITH_HC_128_SHA TLS_RSA_WITH_RABBIT_SHA	wolfSSL 拡張 - eSTREAM 暗号化スイーツ
TLS_RSA_WITH_AES_128_CBC_B2B256 TLS_RSA_WITH_AES_256_CBC_B2B256 TLS_RSA_WITH_HC_128_B2B256	Blake2b 暗号化スイーツ
TLS_QSH	wolfSSL 拡張 -

	量子安全ハンドシェイク
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	wolfSSL 拡張 - NTRU 暗号化スイーツ
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-256 暗号化スイーツ
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384 TLS_DHE_PSK_WITH_NULL_SHA384	SHA-384 暗号化スイーツ
TLS_RSA_WITH_AES_128_GCM_SHA256	AES-GCM 暗号化

表４：wolfSSL 暗号化スーツ

4.3.3 ブロックおよびストリーム暗号化

wolfSSL は AES、DES、３DES および Camellia ブロック暗号化および RC4、RABBIT、HC-128 および ChaCha ストリーム暗号化をサポートします。AES、３DES, RABBIT はデフォルトで有効化されています。HC-128 は wolfSSL ビルド時に（--enable-hc128 ビルド・オプションで）有効化することができます。使用情報に関しては使用例および wolfCrypt レファレンス（第１０章）をご参照ください。

SSL は RC4 をデフォルトのストリーム暗号化として使用します。これは少々古くなりつつありますが、かなり有効です。wolfSSL は eStream プロジェクトから次の二つの暗号化をコードベースに追加しました。RABBIT と HC-128 は RC4 にくらべ２倍近く、HC-128 は約５倍高速です。ですから、もしスピードが心配で SSL を使用したくとがないということであれば、wolfSSL のストリーム暗号化はそうした性能への疑いを軽減ないし解消するはずです。wolfSSL は ChaCha20 も追加しました。RC4 の性能は ChaCha より 11

倍ほど速いのですが、RC４は ChaCha より通常安全性が低いと考えられています。ChaCha はトレードオフとしてセキュリティを高めています。

暗号化の性能比較に関しては wolfSSL ベンチマーク Web ページを参照下さい。
http://wolfssl.com/yaSSL/benchmarks-wolfssl.html

4.3.3.1　両者の違い

ブロック暗号化とストリーム暗号化の違いは何か疑問に思われたことがありますか？

ブロック暗号化は暗号表のブロックサイズの区切り毎に暗号化されます。例えば、AES は 16 バイトのブロックサイズを持っています。従って、もし２、３バイトのたくさんの小さな断片を暗号でやり取りするようですと、データの 80%以上は無駄なパディングになってしまい、暗号・復号プロセスのスピードを低下させ、ネットワークの必要な帯域幅を無駄にしてしまいます。基本的にブロック暗号化は大きなデータの塊のために設計されており、ブロックサイズのためにパディング・サイズがあり、固定の変化の無い転送を使用します。

ストリーム暗号化はデータの塊が大きくても小さくてもうまく機能します。ブロックサイズというものが不要なので、非常に小さなデータサイズに適切です。もしスピードが心配ならば、ストリーム暗号化が答となります。ストリーム暗号化では、通常 XOR された鍵ストリームを利用した単純な変換を使用するからです。従って、もし小さなサイズを含むいろいろなサイズの暗号化を行うストリーム・メディアや高速な暗号化へのニーズをお持ちでしたら、ストリーム暗号化が最良の答となります。

4.3.4ハッシュ関数

wolfSSL は、MD2、MD4、MD5、SHA-1、SHA-2(SHA-256、SHA-384、SHA-512)、SHA-3(BLAKE2)、および RIPEMD-160 などを含む複数のハッシュ関数をサポートします。これらの関数使用方法の詳細は wolfCrypt レファレンスのセッション 10.1 を参照ください。

4.3.5公開鍵オプション

wolfSSL は RSA、ECC、DSA/DSS、DH および NTRU の各公開鍵オプションをサポートします。また、wolfSSL サーバーでは EDH（Ephemeral Diffie-Hellman）をサポートします。これらの関数使用方法の詳細は wolfCrypt レファレンスのセッション 10.5 を参照ください。

wolfSSL は NTRU 公開鍵を利用して以下の４つの暗号化スーツをサポートします。

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

最も強固な AES-256 がデフォルトです。wolfSSL が NTRU 利用可が選択されていて NTRU パッケージが利用可能な場合、これらの暗号化スーツが wolfSSL ライブラリーに組み込まれます。wolfSSL クライアントでは、ユーザ・インタラクション無しにこれらの暗号化スーツを利用可能となります。一方、wolfSSL サーバ・アプリケーションでは、これらの暗号化スーツを利用可能とするためには NTRU プライベート鍵と NTRU x509 証明書をロードする必要があります。

例題の、echoserver と server の両方のサーバは（NTRU を利用可とする）HAVE_NTRU 定義を使用し NTRU 鍵と証明書をロードするかどうかを示します。wolfSSL パッケージは certs/ ディレクトリー下にテスト鍵と証明書を含みます。Ntru-cert.pem は証明書、ntru-key.raw はプライベート鍵 Blob です。

wolfSSL NTRU 暗号化スーツは、プロトコルがスーツを選択するときもっとも高い選択肢となっています。その選択順序は、上に示したものの逆順になっています。例えば、AES-256 が最初に選択され、3DES が最後に、「標準の」暗号化スーツに行く前に選択されます。基本的に、NTRU を wolfSSL に組み込んであり、両者のコネクションが NTRU をサ

ポートするならば、他の暗号化スーツしか使用しないことを宣言して明示的に排除しない限り、NTRU 暗号化スーツが選択されます。

RSA 上の NTRU の利用は 20-200 倍の速度改善をもたらします。鍵サイズが大きくなるほど改善も大きくなり、短い鍵（1024 ビット）に対して長い鍵（8192 ビット）を利用すればずっと大きな速度の恩恵を受けるということです。

4.3.6 ECC サポート

wolfSSL は楕円曲線暗号(ECC)をサポートします。サポートは、
ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, ECDHE-PSK, ECDHE-RSA、その他を含みます。

wolfSSL の ECC のプログラムについては以下に含まれています。
<wolfssl_root>/wolfssl/wolfcrypt/ecc.h ヘッダファイル
<wolfssl_root>/wolfcrypt/src/ecc.c source file　ソースコード

サポートしている暗号スイーツに関しては表 4 を参照してください。ECC はデフォルトでは無効化されていますが、HAVE_ECC 定義によって wolfSSL をビルドすることで有効化することができます。また、autoconf を使用する場合は、次の通りです。

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

“make check”実行時には、多くの暗号スイーツがチェックされることに注目してください(make checkが暗号スイーツのリストを生成しない場合は ./testsuite/testsuite.test 自身を実行してください)。それらの暗号スイーツは、

例えば ECDH-ECDSA with AES256-SHAなど、個別にテストすることもできます。wolfSSLサーバは次のように起動することができます。

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

(-d) は証明書チェックを無効化し、(-l) は暗号スイーツリストを指定します。(-c)は使用すべき証明書、(-k)は対応するプライベイト鍵を指定します。接続するクライアント側は下記のように起動します：

./examples/client/client -A ./certs/server-ecc.pem
(-A) はサーバ認証に使用する CA 証明書です。

4.3.7 PKCS サポート

PKCS（公開鍵暗号化標準）とは、RSA Security 社によって開発、公開された一連の標準を言います。wolfSSL は PKCS #5, PKCS #8 および PKCS #12 から PBKD をサポートします。

4.3.7.1 PKCS#5、PBKDF1、PBKDF2、PKCS#12

PKCS#5 は、パスワード・ベースの鍵導出方法で、パスワード、ソルト、および繰り返し回数の組合せたものです。wolfSSL は PBKDF1 および PBKDF2 鍵導出関数をサポートします。鍵導出関数はベース鍵およびその他のパラメータ（上で説明したように、ソフト、繰り返し回数など）から鍵を導出生成します。PBKDF1 はハッシュ関数（MD5、SHA1 その他）を適用し鍵を導出します。そのとき鍵の長さはハッシュ関数の出力の長さに制限されます。PBKDF2 では疑似ランダム関数（HMAC-SHA-1 など）が鍵の導出に適用されます。PBKDF2 の場合、導出される鍵の長さに制限はありません。

wolfSSL は PBKDF1 と PBKDF2 に加えて PKCS#12 から PBKDF 関数をサポートします。関数プロトタイプはこのような感じです：

```
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 は導出される鍵を制約します。Passwd は長さ pLen のユーザパスワードを保持します。Salt は長さ sLen のソルト入力、iteration は実行されるべき繰り返しの回数、kLen は導出したい鍵の長さ、hashType は使用するハッシュ（MD5, SHA1 または SHA2）です。

wolfSSL のビルドに./configure を使用している場合、この機能を有効化するには--enable-pwdbased オプションを使用します。

フルの例題については wolfcrypt/src/test.c を参照してください。PKCS#5、PBKDF1 および PBKDF2 に関する詳細情報は以下の仕様をご参照ください。

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

4.3.7.2 PKCS#8

PKCS#8 は、公開鍵アルゴリズムおよび属性集合のための秘密鍵を保持するために使用する秘密鍵関連情報の構文に関する標準（Private-Key Information Syntax Standard）として設計されました。

PKCS#8 には、暗号化された秘密鍵および暗号化されない秘密鍵の両方を保持するための構文を説明した二つのバージョンの標準があります。サポートするフォーマットには PKCS#5 バージョン１、バージョン２および PKCS12 が含まれます。暗号化のタイプとしては DES、3DES、RC4 および AES が利用可能です。

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

4.3.8　特定の暗号化の使用を強制

デフォルトで、wolfSSL はコネクションの双方がサポート可能な「最適な（最高のセキュリティ）」暗号化スーツを選択します。例えば 128 ビット AES のように、特定の暗号化を強制するためには、SSL_CTX_new()を呼び出した後に以下のような感じの指定を追加します。

SSL_CTX_set_cipher_list(ctx, “AES128-SHA”);

従って、次のようになります。

ctx = SSL_CTX_new(method);
SSL_CTX_set_cipher_list(ctx, “AES128-SHA”);

4.3.9　量子安全ハンドシェイク暗号スイート

wolfSSL は NTRU のようなものを利用した量子ハンドシェーク暗号スイート（post quantum handshake cipher suite）を利用した暗号スイーツをサポートします：

TLS_QSH

wolfSSL が NTRU 有効化されていて、NTRU パッケージが利用可能ならば、TLS_QSH 暗号スイートが wolfSSL ライブラリにビルドされます。wolfSSL クライアントとサーバはユーザによる何のインタラクションもなしにこの暗号スイートを利用可能になります。

wolfSSL 量子安全ハンドシェーク暗号スイートはプロトコルがスイートを選択する時に最も高い選択順位が与えられます。基本的に、ユーザが wolfSSL に NTRU を組み込み、接続の両側が NTRU をサポートするならば、片方のユーザが異なる暗号スイーツだけを使用するように主張して明示的にそれらを排除しない限り NTRU 暗号スイートが選択されます。

ユーザはどの暗号アルゴリズムか、そしてクライアント側が公開鍵を送ってくるかどうかを、下記のような関数例を使って、調整することができます。

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

クライアントが接続された後、QSH 接続が確立されたかどうかテストするためには以下の関数例を使用することができます。

```
wolfSSL_isQSH(ssl);
```

4.4 ハードウェア高速暗号

wolfSSL では様々なプロセッサやチップにおいてハードウェア高速化(またはアシステト)暗号機能の恩恵を受けることができます。以降のセクションでは wolfSSL がそのままでサポートできるテクノロジーについて説明します。

4.4.1 インテル AES-NI

AES は全世界の政府関係で使用されている暗号化標準で、wolfSSL も常にサポートしてきました。インテル社は AES 実行の高速化の方法として新しい命令セットをリリースしました。wolfSSL は製品化環境でこの新しい命令セットをフルにサポートした最初の SSL ライブラリーです。

本質的に、性能を改善するために、インテルは AES アルゴリズムのうち演算が集中している部分を実行するチップ・レベルの AES 命令群を追加しました。現在 AES-NI をサポートしているインテルのチップについては、下記を参照ください。
http://ark.intel.com/search/advanced/?s=t&AESTech=true

wolfSSL では、アルゴリズムをソフトウェアで実行するのではなく、チップから直接命令を呼び出す機能を追加しました。これによって、AES-NI をサポートするチップ・セット上で wolfSSL 実行時には、AES 暗号を 5～10 倍高速に実行することができます。

AES-NI がサポートされているチップ・セットで実行する場合は、--enable-aesni ビルドオプションを有効にしてください。AES-NI で wolfSSL をビルドするには、アセンブリー・コードを使用するために GCC4.4.3 かそれ以降が必要です。

AES-NI に関してさらに参照できるものを、一般的なものから特定のものの順に、以下にまとめます。AES-NI の性能改善に関する情報は、３番目の Intel Software Network ページを参照ください。

AES (Wikipedia)	http://en.wikipedia.org/wiki/Advanced_Encryption_Standard
AES-NI (Wikipedia)	http://en.wikipedia.org/wiki/AES_instruction_set
AES-NI (Intel Software Network page)	http://software.intel.com/en-us/articles/intel-advancedencryption-standard-instructions-aes-ni/

4.4.2 STM32F2

wolfSSL では STRM32F2 標準周辺ライブラリ(STM32F2 Standard Peripheral Library）を通じてハードウェア・ベースの暗号かと乱数生成器を使用することができます。

settings.h の中で WOLFSSL_STM32F2 の定義を探してください。
WOLFSSL_STM32F2 定義は STM32F2 ハードウェア暗号化と RNG サポートを有効化します。これらを個別に有効化する場合は STM32F2_CRYPTO（ハードウェア暗号サポート）と STM32F2_RNG (ハードウェア RNG サポート）です。

STRM32F2 標準周辺ライブラリ(STM32F2 Standard Peripheral Library)に関しては下記を参照してください。
http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/DM00023896.pdf

4.4.3 Cavium NITROX

wolfSSLはCavium NITROX (http://www.cavium.com/processor_security.html) をサポートします。Cavium NITROXサポートを有効化するにはwolfSSLのビルド時に次のコンフィグレーション・オプションを使用します。

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

ここにおいて、“--with-cavium=” オプションはライセンスされたcavium/softwareディレクトリを指しています。Caviumは、wolfSSLをcavium_common.oファイルに引き込むようにビルドしないため、libtoolはこのポータビリティに関してワーニングを出します。また、github source treeを使っている場合、Camiumのヘッダーはこのワーニングを認めないので、生成されたMakefileから –Wredundant-decls ワーニング・オプションを削除する必要があります。

現在の所、wolfSSLはCaviumのRNG, AES, 3DES, RC4, HMAC、およびRSAを直接暗号化層でサポートします。SSLレベルでのサポートは一部で、現在の所AES、3DESとRC4のみです。RSAとHMACは、Cavium呼び出しが非同期モードで利用できるようになるまで、低速での実行になります。暗号テスト、ベンチマークと同様にサンプルクライアントでもCaviumサポートが有効になります。HAVE_CAVIUM定義を参照してください。

4.5 SSL 検査（Sniffer）

wolfSSL1.5.0 リリース以降、wolfSSL は SSL Sniffer（SSL 検査）とともにビルドするためのオプションを提供しています。これによって、SSL のトラフィックパケットを収集することができ、正しい鍵ファイルによって復号化することが可能になります。SSL トラフィックを「検査」する機能は以下のように複数の用途で便利です。

- ネットワークの問題を解析する
- 内部または外部ユーザによるネットワークの不正な利用を検出する
- ネットワークの使用状況と動いているデータをモニターする

・　クライアント・サーバ間のコミュニケーションをデバックする

Sniffer サポートを有効にするためには、*nix 上または Windows 上で vcproj ファイルを使用して　--enable-sniffer オプションで wolfSSL をビルドしてください。pcap がインストールされた*nix、または Windows 上の WinPcap が必要です。sniffer.h にあるように５つの主要な関数があります。以下はそれらの簡単な説明です。

ssl_SetPrivateKey – 指定したサーバーとポートにプライベート鍵を設定する。
ssl_SetNamedPrivateKey – 指定したサーバのプライベート鍵にポートとドメイン名を設定する。
ssl_DecodePacket – 復号のための TCP/IP パケットを渡す。
ssl_Trace – traceFile のデバッグ・トレースを有効/無効にする。
ssl_InitSniffer – sniffer 全体を初期化する。
ssl_FreeSniffer – sniffer 全体を解放する。
ssl_EnableRecovery – ロスト・パケットの場合において、SSL トラッフィクの復号を取り上げる試みを有効化するオプション
ssl_GetSessionStats – Sniffer セッションのメモリ使用量をえる

wolfSSL の sniffer サポートと例題全体を参照するには、wolfSSL ダウンロードから "sslSniffer/sslSnifferTest" フォルダーの ”sniffertest” を参照してください。

SSL ハンドシェイクに暗号化鍵が設定されているので、さらなるアプリケーション・データのデコードには、そのハンドシェイクが sniffer によってデコードされる必要がある点に注意してください。例えば、wolfSSL 例題の echoserver と echoclient とともに に”snifftest”を使用した場合、snifftest アプリケーションはサーバとクライアント間のハンドシェイク開始前に開始されていなければなりません。

Sniffer 機能は以下のアルゴリズムで暗号化されたストリームの復号のみ可能です：AES-CBC, DES3-CBC, ARC4, HC-128, RABBIT, Camellia-CBC, および IDEA。ECDHEまたはDHE鍵合意が使用されている場合、ストリームを見ることはできません。RSA鍵交換のみサポートされています。

4.6 圧縮

wolfSSL は zlib ライブラリーのデータ圧縮をサポートしています。./configure ビルドシステムはこのライブラリーの存在を検出しますが、何か別の方法でビルドする場合は HAVE_LIBZ 定数を定義し、zlib.h へのパスをインクルードのために含めてください。

与えられた暗号化では圧縮はデフォルトではオフになっています。オンにするには、SSL コネクションまたはアクセプトの前に wolfSSL_set_compression()を使用してください。圧縮を使用するためにはクライアントとサーバ双方が圧縮がオンになっている必要があります。

送出前の圧縮は実際に送受されるメッセージのサイズを減らす一方で、圧縮により削減されたデータの解析時間は、よほど遅いネットワークで無い限り、そのまま送信するよりも長くなるという点に注意してください。

4.7 事前共有鍵

wolfSSL は以下の二つの事前共有鍵暗号化をサポートします。

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

これらのスーツは WOLFSSL_STATIC_PSK オプションにって自動的に wolfSSL に組込まれますが、NO_PSL 定数によってビルド時にオフにすることができます。実行時のみにこ

れらの暗号化を使用する場合には、希望する暗号化スーツにおて
wolfSSL_CTX_set_cipher_list()関数を使用してください。

wolfSSL は以下の一時鍵（ephemeral key) PSK をサポートします。

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

クライアント側では、wolfSSL_CTX_set_psk_client_callback() を使用してコールバックを設定してください。<wolfSSL_Home>/examples/client/client.c のクライアントの事例では、クライアント・アイデンティティーと鍵を設定するための例を提供しています。実際のコールバックは wolfssl/test.h でインプリメントされています。

サーバ側では、二つの追加の関数呼び出しが必要です。

wolfSSL_CTX_set_psk_server_callback()
wolfSSL_CTX_use_psk_identity_hint()

我々のサーバ・サンプルプログラムでは、サーバは二つ目の呼び出しでクライアントを助けるために、識別のヒント(identity hint) 保持しています。サーバ PSK コールバックの例は wolfssl/test.h の中の my_psk_server_cb()にあります。

wolfSSL は最長 128 オクテットのアイデンティティーとヒント、および最長 64 オクテットの事前共有鍵をサポートします。

4.8 クライアント認証

クライアント認証は、クライアントがコネクト時に認証のための証明書を

サーバに送ることを要求することで、サーバがクライアントを認証できるようにする機能です。クライアント認証は CA から得た（または、ユーザか CA 以外の誰かが生成し自己署名した）X.509 クライアント証明書を要求します。

デフォルトでは、wolfSSL はクライアントとサーバ双方の受け取ったすべての証明書の有効性をチェックします。クライアント側の認証を設定するには、サーバ側は、クライアント側の証明書をチェックするのに使用する、信頼できる CA 証明書のリストをロードしなければなりません。

wolfSSL_CTX_load_verify_locations(ctx, caCert, 0);

クライアント認証をオフにしたり、その挙動を管理するためには
wolfSSL_CTX_set_verify() 関数が使用されます。以下の例では、

SSL_VERIFY_PEER はサーバからクライアントに対する認証をオンにします。
SSL_VERIFY_FAIL_IF_NO_PEER_CERT はクライアントがサーバ側でチェックすべき証明書を提示しない場合にフェイルするように指示します。
wolfSSL_CTX_set_verify() のその他のオプションとし SSL_VERIFY_NONE、
SSL_VERIFY_CLIENT_ONCE などがあります。

```
wolfSSL_CTX_set_verify(ctx,SSL_VERIFY_PEER |
                       SSL_VERIFY_FAIL_IF_NO_PEER_CERT,0);
```

wolfSSL ダウンロード(/examples/server/server.c)に含まれる例題サーバ(server.c)のクライアント認証の例も参照してください。

4.9 Server Name Indication (SNI)

SNI は、単一のネットワークアドレスにおいて 1 サーバ複数仮想サーバの場合に有効です。多分、クライアントに対してコンタクトしているサーバの名前を用意することが望ましい。SNI を有効化するにはビルド時に以下を実行します：

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

クライアント側で SNI を利用する場合、追加で以下のうちどちらかもう一つの関数呼び出しが必要です。

wolfSSL_CTX_UseSNI()
wolfSSL_UseSNI()

wolfSSL_CTX_UseSNI()はクライアントが同じサーバに複数回コンタクトする場合に推奨されます。SNI拡張をコンテクストレベルで設定することで、その呼び出しがされたて以降、同じコンテクストから生成される全てのオブジェクトにおいてSNIの使用を可能にします。

wolfSSL_UseSNI()は一つの SSL オブジェクトのみに SNI を有効化するので、サーバ名がセッション間で変わるような場合に推奨されます。

サーバ側でも同じ関数呼び出しが必要です。wolfSSLサーバは複数”仮想“サーバをホストしないので、SNIの使用はSNIミスマッチのケースに接続の終了が望まれるような場合に有効です。このシナリオでは、サーバはコンテクストごとに一回だけで、同じコンテクストからのすべてのそれ以降のSSLオブジェクトに設定されるので、wolfSSL_CTX_UseSNI()が効率的です。

4.10 ハンドシェーク修正

4.10.1 ハンドシェイク・メッセージのグループ化

必要の場合、wolfSSL はハンドシェイク・メッセージをグループ化する機能を持っています。これはコンテクスト・レベルで、

wolfSSL_CTX_set_group_messages(ctx);

を使用するか、SSL オブジェクト・レベルで、

wolfSSL_set_group_messages(ssl);

4.11 切り詰めた（Truncated） HMAC

現在定義されている TLS 暗号スイーツはレコード層通信の認証に HMAC を使用しています。TLS では、ハッシュ関数の全体出力に対して MAC タグとして使用されています。しかし、制限された環境では、MAC タグを形成する時ハッシュ関数の出力を 80 ビットに切り詰める事によってバンド幅を節約したくなる場合があります。切り詰めた（Truncated） HMAC を使用可能にするためには wolfSSL ではビルド時に単に次のように指定します：

./configure --enable-truncatedhmac

切り詰めた（Truncated） HMAC をクライアント側で使用するには、次の二つのいずれかの関数を呼び出す必要があります。

wolfSSL_CTX_UseTruncatedHMAC();
wolfSSL_UseTruncatedHMAC();
クライアント側で SNI を利用する場合、追加で以下のうちどちらかもう一つの関数呼び出しが必要です。

wolfSSL_CTX_UseTruncatedHMAC ()はクライアントが同じサーバに複数回コンタクトする場合に推奨されます。Truncated HMAC拡張をコンテクストレベルで設定することで、その呼び出しがされたて以降、同じコンテクストから生成される全てのオブジェクトにおいてTruncated HMACの使用を可能にします。

wolfSSL_UseTruncatedHMAC ()は一つの SSL オブジェクトのみに Truncated HMAC を有効化するので、TruncatedHMAC が必要かどうかがセッション間で変わるような場合に推奨されます。

サーバ側では関数呼び出しは不要です。サーバは自動的にクライアントの Truncated HMAC の要求に対応します。

なお、すべての TLS 拡張は下記でも有効化されます：

./configure --enable-tlsx

4.12 ユーザ定義暗号モジュール

ユーザ定義暗号モジュールによって、ユーザはサポートされている処理の間に使用したいカスタム暗号をプラグインすることができます(現在の所、RSA 処理がサポートされています)。IPP を使ったモジュールの例は root_wolfssl/wolfcrypt/user-crypto/ ディレクトリにあります。ユーザ定義暗号モジュールを使用する wolfSSL のビルドの場合の configure に対するオプションの例は次の通りです：

./configure --with-user-crypto
または
./configure --with-user-crypto=/dir/to

RSA 処理を実行するユーザ暗号モジュールを生成する時には、user_rsa.h という名前の RSA のためのヘッダーファイルがあることが必須です。すべてのユーザ暗号処理のために、libusercrypt という名前のユーザライブラリが必須です。これらは、ユーザ暗号モジュールをリンクする際に wolfSSL の autoconf ツールが探しに行く名前です。wolfSSL で提供されるサンプルでは、user_rsa.h は wolfcrypt/user-crypto/include/ディレクトリにあります。また、一旦生成されたライブラリは wolfcrypt/user-crypto/lib/に置かれます。必要な API のリストについては、提供されたヘッダーファイルを参照してください。

サンプルをビルドする伊波、IPP ライブラリーをインストールした後、次のコマンドを wolfSSL のルートディレクトリで実行させます。

cd wolfcrypt/user-crypto/
./autogen.sh

./configure
make
sudo make install

wolfSSL に含まれているサンプルは IPP を使用する必要があります。これはプロジェクトがビルドされる前にインストールされている必要があります。サンプルをビルドするために IPP ライブラリーが無い場合でも、ユーザにファイル名の選択と API インターフェースの例を提供するためです。一旦、libusercrypto とヘッダーファイルの両方が make され、インストールされると、暗号モジュールを使った wolfSSL の make に追加ステップは必要なくなります。単に、-with-user-crypto フラグを configure に使用するだけで、通常の wolfSSL 暗号からユーザ暗号モジュールにすべての関数呼び出しがマップされます。

もし wolfSSL の XMALLOC を使用している場合、メモリーアロケーションは DYNAMIC_TYPE_USER_CRYPTO とタグ付けしなければなりません。モジュールで使われたすべてのメモリーアロケーションの分析を可能にします。

ユーザ暗号モジュールは wolfSSL の configure オプション fast-rsa と/または fips オプションと組み合わせ使用することはできません。FIPS は、特定の認証されたコードが使用されることを要求します。fast-rsa は RSA 処理の実行にサンプル・ユーザ暗号モジュールを使用させます。

5. 移植性

5.1. 抽象化レイヤー

5.1.1. C 標準ライブラリー抽象化レイヤー

wolfSSL（旧 CyaSSL）は、開発者に対してより高レベルの移植性と柔軟性を提供するために、C 標準ライブラリー無しでビルドすることが可能になっています。そのために、ユーザは C 標準ライブラリーの関数に対応した自分の使用したい関数をマップする必要があります。

5.1.1.1. メモリーの使用

ほとんどの C プログラムは動的メモリー・アロケーションに malloc()と free()を使用しています。CyaSLL では、代わりに XMALLOC()と XFREE()を使用しています。デフォルトでは、これらは C 実行バージョンを指しています。XMALLOC_USER を定義することで、独自のフックを提供することができます。それぞれのメモリ関数は、標準のものに加えて heap hint とアロケーション・タイプの２つのアーギュメントが追加されています。これらを無視するのも、任意の用途に使用するのも自由です。wolfSSL メモリー関数は wolfssl/wolfcrypt/type.h にあります。

wolfSSL はメモリーオーバーライド関数をコンパイル時ではなく実行時に登録する機能を提供しています。wolfssl/wolfcrypt/memory.h はこの機能のためのヘッダーで、次の関数を呼び出しメモリー関数を設定することができます。

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

コールバックの関数プロトタイプは wolfssl/wolfcrypt/memory.h ヘッダーを、実現に関しては memory.c を参照してください。

5.1.1.2. string.h

wolfSSL は string.h の memcpy()、memset() 、memcmp()、その他の関数のような動作をする関数を使用します。これらは、それぞれ XMEMCPY()、XMEMSET()、

XMEMCMP()に抽象化されています。デフォルトでは、それらは C 標準ライブラリーのバージョンを指しています。XSTRING_USER を定義することで、types.h に独自フックを提供できるようになっています。例えば、XMEMCPY()は：

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

となっています。XSTRING_USER を定義して下記のようにもできます:

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

あるいは、マクロを避けたいようでしたら:

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

のようにして、wolfSSL 抽象化レイヤーが独自バージョンの my_memcpy()を指すようにすることもできます。

5.1.1.3. math.h

wolfSSL は math.h の pow()と log()のような動作をする二つの関数を使用します。それらは Diffie-Hellman に必要とされるだけですので、もし DH をビルドから外すようでしたら、独自のものを用意する必要はありません。これらは wolfcrypt/src/dh.h に XPOW()と XLOG()として抽象化されています。

5.1.1.4. ファイルシステムの使用

デフォルトでは、wolfSSL は鍵と証明書のロードの目的でターゲットシステムのファイルシステムを使用します。これは NO_FILESYSTEM を定義することでオフにすることができます。「第５章：ビルド・オプション」を参照してください。もし代わりに、システムのものではないファイル・システムを使用したい場合は、ssl.c の XFILE()レイヤーを使用して、ファイル・システムへの呼び出しを自分の使用したいと思うものを指すようにすることができます。MICRIUM 定義で提供されている例を参照してください。

5.1.2. カスタム入出力抽象化レイヤー

wolfSSL は、SSL コネクションや SSL を TCP/IP 以外のトランスポート層で実行する場合により高レベルのコントロールを望むような場合のためのカスタム I/O 抽象化レイヤーを提供します。

使用するには、二つの関数を定義する必要があります。

1. ネットワーク送信関数
2. ネットワーク受信関数

これら二つの関数は ssl.h の CallbackIOSend と CallbackIORecv によってプロトタイプ宣言されています。

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

これらの関数を WOLFSSL_CTX ごとに wolfSSL_SetIOSend() と wolfSSL_SetIORecv() として登録する必要があります。デフォルトでは CBIORecv() と CBIOSend()が io.c の最後尾に登録されています：

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

ユーザは、io.c の一番下に示しているように、WOLFSSL オブジェクト（セッション）毎に wolfSSL_SetIOWriteCtx() と wolfSSL_SetIOReadCtx()でコンテクストをセットできます。例えば、メモリー・バッファーを使用するような場合は、コンテクストはそのメモリー・バッファーの場所とアクセス方法を説明するような構造体へのポインターとなるでしょう。デフォルトでは、ユーザ・オーバーライド無しで、ソケットをコンテクストとして登録しています。

CBIORecv と CBIOSend 関数のポインターはユーザ独自のカスタム I/O 関数を指すことができます。デフォルトの Send()と Receive()関数は、io.c にある EmbedSend()と EmbedReceive()で、テンプレートやガイドとして使用することができます。

WOLFSSL_USER_IO を定義して、デフォルト I/O 関数、EmbedSend()と EmbedReceive()の自動設定を取り除くことができます。

5.1.3. オペレーティングシステム抽象化レイヤー

wolfSSL の OS 抽象化レイヤーは wolfSSL をユーザのオペレーティング・システムへの移植をより容易にします。wolfssl/wolfcrypt/settings.h ファイルには OS レイヤーを起動するための設定が格納されています。

OS 固有の定義は wolfCrypt 向けは wolfssl/wolfcrypt/types.h、wolfSSL 向けのものは wolfssl/internal.h にあります。

5.2. サポートするオペレーティング・システム

wolfSSL は新しいプラットフォームへの容易な移植性という特徴をもっていますが、一方で wolfSSL にはそのまますぐに利用いただけるたくさんのサポート対象オペレーティング・システムがあります。現在サポートされるオペレーティング・システムとしては以下のようなものが含まれます：

Win32/64, Linux, Mac OS X, Solaris, ThreadX, VxWorks, FreeBSD, NetBSD, OpenBSD, embedded Linux, WinCE, Haiku, OpenWRT, iPhone (iOS), Android, Nintendo Wii and Gamecube through DevKitPro, QNX, MontaVista, OpenCL, NonStop, µITRON/, µT-Kernel, Micrium's µC/OS, FreeRTOS, Freescale MQX, Nucleus, TinyOS, HP/UX, TIRTOS

5.3. サポートするチップ・メーカー

wolfSSLがサポートするチップ・メーカーのチップ・セットとしてはARM, Intel, ST

(STM32F2/F4), Motorola, mbed, Freescale, Microchip (PIC32), Texas Instruments そのほかが含まれます。

5.4. C#ラッパー

wolfSSL はC#に対して限定されたサポートを提供しています。Visual Studioプロジェクトが“root_wolfSSL/wrapper/CSharp/”ディレクトリにあります。Visual Studio プロジェクトを開いた後、”BUILD->Configuration Manager…” をクリックして、“Active solution configuration” と “Active solution platform” をセットします。“Active solution configuration” はDLL Debug とDLL Release です。サポートされるプラットフォームはWin32とWin64です。

ソリューションとプラットフォームをセットしたら、プリプロセッサフラグにHAVE_CSHARPが必要となります。これは、C#ラッパーと同梱のサンプルによって使用されるオプションをオンにします。

次に、“Build Solusion”を選択してビルドします。これによって、wolfssl.dll. wolfSSL_CSharp.dllとexamplesが生成されます。”examplese” はエントリーポイントとしてそれをターゲッティングし、Visual Satudio をDebug実行することで実行できます。

生成されたC#ラッパーをC#プロジェクトに加えるのは、２つの方法で可能です。一つの方法は、 生成されたwolfssl.dll とwolfSSL_CSharp.dllをC:/Windows/System/にインストールします。これによって、プロジェクトは以下のように、C#ラッパーを呼び出すことができるようになります。

```
using wolfSSL.CSharp
public some_class {
public static main(){
wolfssl.Init()
...
}
...
```

もう一つの方法は、Visual Studio プロジェクトを生成して、wolfSSL 内の C#ラッパーソリューションを呼び出します。

6. コールバック

6.1.ハンドシェイク・コールバック

wolfSSLは、コネクトまたはアクセプトを設定するためのハンドシェイク・コールバックのための拡張機能を提供しています。これは、組込みシステムのデバッグサポートにおいて他のデバッガが利用可能でない場合やスニファリングが現実的でない場合に有用です。wolfSSL HandShakeCallBackを使用するためには、拡張関数*wolfSSL_connect_ex()* または*wolfSSL_accept_ex()* を使用して：

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

HandShakeCallBack は以下のように定義されます:

typedef int (*HandShakeCallBack)(HandShakeInfo*);

HandShakeInfo はwolfssl/callbacks.h (非標準のビルドに追加)に定義されます:

```
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;
```

ハンドシェイクにおけるSSLパケットの最大数はわかっているので、動的メモリーは使用されません。パケット名はpacketNames[idx]をmuberPacketsで指定された数だけ評価されます。コールバックはハンドシェイクエラーが発生したかどうかにかかわらず呼び出されます。使用例はclientの例にもあります。

6.2 タイムアウト・コールバック

wolfSSL ハンドシェイク・コールバックに使用されたものと同じ拡張が wolfSSL タイムアウト・コールバックにも使用することができます。これらの拡張はどちらか一方、両方の呼び出し、またはどちらのコールバックも（ハンドシェイク/タイムアウト）使用しないことができます。TimeoutCallback は次のように定義されます：

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

その中の *TimeoutInfo* は以下のように:

```
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 ;
```

繰り返しになりますが、ハンドシェイクにおける SSL パケットの最大数はわかっているので、動的メモリーは使用されません。*Timeval* は timeval 構造体の typedef です。

PacketInfo は次のような定義になります：

```
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;
```

ここにおいては、動的メモリーが使用されるかもしれません。SSL パケットが *value* に収まるようならそこに収めます。*valueSz* は長さを保持し、*bufferValue* は 0 です。パケットサイズが *value* に対して大きすぎるようなら、パケットは *bufferValue* のほうに置かれ、*valueSz* は変わらずサイズを保持します。このようなことは**証明書**パケットにのみ発生するはずです。

証明書パケット向けにメモリーがアロケートされる場合、コールバックからリターンした後、再クレームされます。タイムアウトはシグナル（SIGALRM）を使用して実現されており、スレッドセイフです。以前のアラームが ITIMER_REAL タイプにセットされている場合は、後でその時点のハンドラーとともにリセットされます。既存のタイマーが過去のタイマーより短い場合は、既存のタイマー値が使用されます。これも、その後リセットされます。タイムアウトする既存のタイマーは、それと関連付けられたインターバルを持っている場合はリセットされます。コールバックはタイムアウトが発生した場合だけに発行されます。

使用方法については client の例を参照してください。

6.3 ユーザ定義のアトミック・レコード層処理

wolfSSL はユーザ定義のアトミック・レコード層処理のコールバックを提供している。これは、SSL/TLS 接続中に MAC/暗号と復号化/検証機能を自分で管理したいユーザのためのものです。

ユーザは二つの関数を定義する必要があります。

4.9.6　MAC/暗号化コールバック関数

4.9.7　復号化/検証コールバック関数

これらの二つの関数は ssl.h の中で CallbackMacEncrypt と CallbackDecryptVerify としてプロトタイプ定義されています。

```
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);
```

ユーザはこれらの関数を書き、wolfSSL コンテクスト(WOLFSSL_CTX)ごとに wolfSSL_CTX_SetMacEncryptCb() と wolfSSL_CTX_SetDecryptVerifyCb() で登録する必要があります。

ユーザは wolfSSL_SetMacEncryptCtx() と wolfSSL_SetDecryptVerifyCtx() で WOLFSSL オブジェクト (セッション) ごとにユーザコンテクストを設定することができます。このコンテクストは任意のユーザ定義コンテクストに対するポインタが指定可能です。ここで指定したポインタは、”void* ctx”パラメータとして MAC/暗号化と復号化/検証コールバックに送り返されます。

サンプルコールバックが wolfssl/test.h の myMacEncryptCb()と myDecryptVerifyCb()にあります。使用方法は wolfSSL サンプルクライアント(examples/client/client.c) の”-U”オプションを指定した場合を参照してください。

アトミックなレコード層コールバックを使用するために、wolfSSL は configure コマンドのオプションで “--enable-atomicuser”を指定するか、プリプロセッサフラブに ATOMIC_USER を定義してください。

6.4 公開鍵コールバック

wolfSSL は、SSL/TLS 接続中に ECC 署名/検証機能、RSA 署名/検証機能、および暗号化、復号化に対して自分で管理したいユーザのために公開鍵コールバックを提供しています。

ユーザは以下の 6 つの関数を使うことができます。

1. ECC 署名コールバック
2. ECC 検証コールバック
3. RSA 署名コールバック
4. RSA 検証コールバック
5. RSA 暗号化コールバック
6. RSA 復号化コールバック

これら６つの関数はssl.hの中でCallbackEccSign, CallbackEccVerify, CallbackRsaSign, CallbackRsaVerify, CallbackRsaEnc, および CallbackRsaDec としてプロトタイプ定義されています。

```
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 (*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);
```

ユーザはこれらの関数を書き、wolfSSLコンテクスト (WOLFSSL_CTX)ごとに wolfSSL_CTX_SetEccSignCb(), wolfSSL_CTX_SetEccVerifyCb(), wolfSSL_CTX_SetRsaSignCb(), wolfSSL_CTX_SetRsaVerifyCb(), wolfSSL_CTX_SetRsaEncCb(), およびwolfSSL_CTX_SetRsaDecCb()する必要があります。

ユーザはWOLFSSL オブジェクト (セッション) ごとにユーザコンテクストを wolfSSL_SetEccSignCtx(), wolfSSL_SetEccVerifyCtx(), wolfSSL_SetRsaSignCtx(),

wolfSSL_SetRsaVerifyCtx(), wolfSSL_SetRsaEncCtx(), およびwolfSSL_SetRsaDecCtx()関数で登録することができます。

このコンテクストは任意のユーザ定義コンテクストに対するポインタが指定可能です。ここで指定したポインタは、”void* ctx”パラメータとして MAC/暗号化と復号化/検証コールバックに送り返されます。

サンプルコールバックが wolfssl/test.h の myEccSignCb(), myEccVerfyCb(), myRsaSignCb(), myRsaVerfyCb(), myRsaEncCb()と myRsaDecCb()にあります。使用方法は wolfSSL サンプルクライアント(examples/client/client.c) の”-P”オプションを指定した場合を参照してください。

公開鍵コールバックを使用するために、wolfSSL は configure コマンドのオプションで “--enable-atomicuser”を指定するか、プリプロセッサフラブに ATOMIC_USER を定義してください。

7. 鍵と証明書

X.509 署名入門、また SSL と TLS における使用方法については付録 A を参照してください。

7.1 サポートするフォーマットとサイズ

wolfSSL は、PKCS#8 秘密鍵（PKCS#5 または PKCS#12 暗号化による）とともに PEM と DER フォーマットの証明書と鍵をサポートします。

PEM（Privacy Enhanced Mail）は認証局から発行される証明書のもっとも一般的なフォーマットです。PEM ファイルは、複数のサーバーの証明書、中間認証、非公開鍵を含むことができる Base64 でエンコードされた ASCII ファイルで、.pem, .crt, .cer および.key のファイル拡張子を持ちます。PEM ファイルを含む証明書は“-----BEGIN CERTIFICATE-----”と“-----END CERTIFICATE-----”ステートメントで包まれます。

DER（Distinguished Encoding Rules）は証明書のバイナリー・フォーマットです。DER ファイルの拡張子は.der または.cer などで、テキスト・エディターで見ることができます。

7.2 証明書のロード

証明書は通常、ファイル・システムを使用してロードされます（メモリ・バッファーからのロードもサポートされています。セクション 7.5 参照）

7.2.1CA 証明書のロード

CA 証明書ファイルは wolfSSL_CTX_load_verify_locations()関数を使用してロードすることができます：

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

CA のロードは、上記の関数を使用して PEM フォーマットで CAfile を渡すことで、証明書の数に制限なくファイル当たり複数の CA 証明書を解析することができます。これによ

って初期化が簡単になり、スタートアップ時に複数のルート CA をロードする必要がある場合便利です。これは、wolfSSL を複数の CA を単一ファイルで扱うことを期待しているツールへの移植を容易にします。

7.2.2 クライアントまたはサーバー証明書のロード

単一のクライアントまたはサーバー証明書のロードは wolfSSL_CTX_use_certificate_file() 関数で行われます。この関数は証明書チェーンに使用した場合、実際の（または“ボトムの”）証明書だけが送られます。

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

CAfile は CA 証明書ファイルです。Type は SSL_FILETYPE_PEM などの証明書のフォーマットです。

サーバーまたはクライアントは wolfSSL_CTX_use_certificate_chain_file()関数を使って証明書チェーンを送ることができます。証明書チェーン・ファイルは PEM フォーマットでなければならず、subject の証明書（実際のクライアントまたはサーバー証明書）で始まり、以降は中間証明書および（必要に応じて）ルート（トップ）CA で終わるように順序がソートされていなければなりません。サーバの例（/examples/server/server.c）はこの機能を使用しています。

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

7.2.3 プライベート鍵のロード

サーバーのプライベート鍵は wolfSSL_CTX_use_PrivateKey_file()関数を使用してロードすることができます。

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

keyFile はプライベート鍵、type はプライベート鍵のフォーマット（例えば SSL_FILETYPE_PEM）です。

7.2.4　信頼する相手の証明書のロード

信頼する相手の証明書のロードは wolfSSL_CTX_trust_peer_cert()関数で行います。

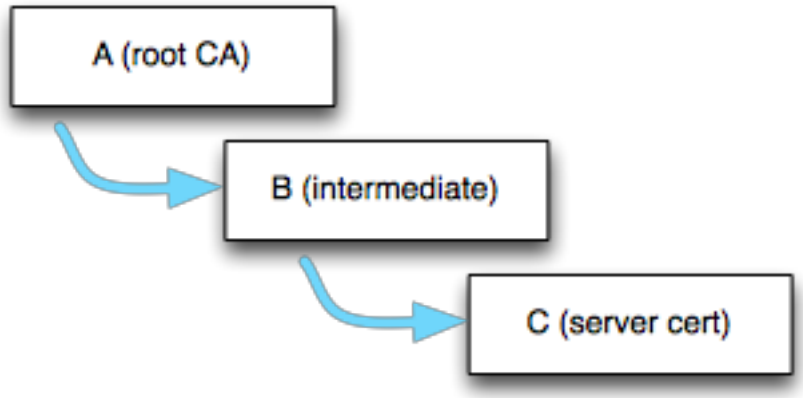
```
int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX *ctx,
const char *trustCert, int type);
```

trustCert はロードすべき証明書、type は証明書のフォーマット（例えば、SSL_FILETYPE_PEM）です。

7.3　証明書チェーンの検証

wolfSSL は、証明書チェーンを検証するために、信頼できる証明書としてロードされるチェーンのトップ（ルート）証明書だけを必要とします。つまり、C は B によって署名され、B は A によって署名されたような証明書チェーン（A→B→C）を持っている場合、wolfSSL は、チェーン（A→B→C）全体を検証するために信頼できる証明書としてロードされた証明書 A だけを要求するということです。

例えばサーバ証明書チェーンは次のような感じになっています：



wolfSSL クライアントは少なくとも一つの信頼する CA をすでに持っているはずです。クライアントがサーバの証明書チェーンを受診すると、まず署名 A を使って署名 B を検証します。もし、B が以前に信頼する CA として wolfSSL にロードされていなければ、B は

wolfSSL 内部の信頼する CA チェーンに保存されます（wolfSSL は証明書の検証に必要な物だけを保存します：共通名ハッシュ、公開鍵と鍵タイプ他）。B が検証されたら、それを C の検証に使います。

このモデルに従って、A が信頼するルートとして wolfSSL クライアントにロードされている限り、サーバが(A->B->C)または(B->C)を送れば、サーバ証明書チェーンの検証は可能なのです。もし、サーバが中間証明書なしで C のみをを送った場合、wolfSSL クライントが信頼するルートとして B をロードしていない限り検証することはできません。

7.4 サーバー証明書のドメイン名チェック

wolfSSL は、サーバー証明書のドメインを自動的にチェックするクライアントに対する拡張機能を持っています。OpenSSL モードでは、これを実現するのに十数個近くの関数コールを必要とします。wolfSSL は証明書の日付のレンジのチェック、署名の検証、また必要ならドメインの検証などを、wolfSSL_connect()の前に以下の関数呼び出しで行います：

wolfSSL_check_domain_name(WOLFSSL* ssl, cons char* dn)；

wolfSSL はピアのサーバー証明書の X509 発行者名と dn を照合します。名前がマッチする場合、wolfSSL_connect()は正常に処理しますが、マッチしない場合は wolfSSL_connect()は致命的エラーを返却し、wolfSSL_get_error() は DOMAIN_NAME_MISMATCH を返却します。

証明書のドメイン名チェックはサーバーが実際にだれを名乗っているのかを検証する重要なステップです。この拡張はチェックを実現する負荷を軽減することを意図しています。

7.5 ファイル・システム無しでの証明書使用

通常は、非公開鍵、証明書、CA などのロードのためにはファイル・システムが使用されます。wolfSSL は本格的ファイル・システム無しの環境で使用されることもあるので、代

わりにメモリー・バッファを使用した拡張を提供しています。この拡張を利用するには定数 NO_FILESYSTEM を定義して下記の関数を使用可能にしてください：

int wolfSSL_CTX_load_verify_buffer(WOLFSSL_CTX*, const unsigned char*, long)
int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX*, const unsigned char*,
long, int)
int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX*, const unsigned char*,
long, int)
int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX*, const unsigned
char*,long)

関数名の buffer が file となっている対応する関数とまったく同じように、これらの関数を使用してください。また、ファイル名を与える代わりにメモリー・バッファーを与えてください。

7.5.1 テスト用証明書と鍵バッファ

wolfSSL は、以前はテスト用の証明書と鍵ファイルをバンドルしていました。今は、ファイルシステムの使えない環境のために、テスト用の証明書バッファと鍵バッファをバンドルしています。これはのバッファは cert_test.h にあって、USE_CERT_BUFFERS_1024 または USE_CERT_BUFFERS_2048 を定義することで使用できます。

7.6 シリアル番号のリトリーブ

X509 証明書のシリアル番号は以下の関数を使って wolfSSL から抽出することができます。シリアル番号の長さは任意です。

int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509, byte* buffer,
int* inOutSz)

buffer は入力の最大*inOutSz バイトで書き込まれます。呼び出し後、正常の場合（返却値 0）、*inOutSz は buffer に実際に書き込まれたバイト数を保持します。例全体は wolfssl/test.h に含まれています。

7.7 RSA 鍵生成

wolfSSL は最長 4096 ビット長の RSA 鍵生成をサポートします。鍵生成はデフォルトではオフとなっていますが、./coufigure プロセスで：

--enable-keygen

で、または Windows や非標準環境では WOLFSSL_KEY_GEN を定義によってオンにすることができます。鍵の生成は簡単で、rsa.h から一つの関数を要求するだけです：

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

size はビット長、e は公開鍵暗号化指数で通常 65537 がおすすめです。以下の wolfcrypt/test/test.c からの例では 1024 ビットの RSA 鍵を生成します：

```
RsaKey genKey;
RNG rng;
int ret;
InitRng(&rng);
InitRsaKey(&genKey, 0);
ret = MakeRsaKey(&genKey, 1024, 65537, &rng);
if (ret < 0)
/* ret contains error */;
```

これで RsaKey genKey は他の RsaKey と同じように使用できます。鍵をエキスポートする必要がある場合は、wolfSSL は asn.h 中の DER と PEM フォーマットの両方を提供します。常に最初 DER フォーマットに変換して、それから PEM が必要なら汎用の DerToPem()関数を以下のように使用してください：

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

```
        /* derSz contains error */;
```

これで、バッファーdef は DER フォーマットの鍵を保持します。DER バッファを PEM に変換するために変換関数を使用します：

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

DerToPem()の最後のアーギュメントは type パラメータで、通常、PRIVATEKEY_TYPE か CERT_TYPE のいずれかです。これで、バッファーpem は PEM フォーマットの鍵を保持します。

7.7.1 RSA 鍵生成の注意点

RSA 非公開鍵は公開鍵も含みますが、wolfSSL は現在、スタンド・アロンの RSA 公開鍵を生成する機能は持っていません。wolfSSL では test.c で使用されているように、非公開鍵を非公開と公開鍵どちらとしても使用することができます。

wolfSSL で個々の RSA 公開鍵生成が無い理由は、非公開鍵と公開鍵（証明書の形で）はどちらも通常 SSL にとって必要とされるものだからです。

分離した公開鍵は RsaPublicKeyDecode()関数を使用して手動で wolfSSL にロードすることができます。

7.8 証明書の生成

wolfSSL は x509 v3 証明書の生成をサポートします。証明書生成はデフォルトではオフとなっていますが、./coufigure 過程で以下を使ってオンにすることができます：

--enable-certgen

また、Windows や非標準の環境では WOLFSSL_CERT_GEN を定義してオンにすることもできます。

証明書を生成できるようにする前に、証明書のサブジェクトについての情報を準備する必要があります。この情報は Cert と命名された wolfssl/wolfcrypt /asn.h の構造体に含まれています：

```
/* for user to fill for certificate generation */
typedef struct Cert {
        int version; /* x509 version */
        byte serial[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 */
} Cert;
```

ここにおいて、CertName は以下のような感じです：

```
typedef struct CertName {
        char country[NAME_SIZE];
        char state[NAME_SIZE];
        char locality[NAME_SIZE];
        char org[NAME_SIZE];
        char unit[NAME_SIZE];
        char commonName[NAME_SIZE];
        char email[NAME_SIZE];
} CertName;
```

サブジェクト情報を格納する前に、以下のような初期化関数を呼び出す必要があります：

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

InitCert()は、バージョン番号を 3(0x02)、シリアル番号を 0（ランダムに生成）、sigType を SHA_WITH_RSA、daysValid を 500、また selfSigned を 1 (TRUE)に.設定するほか、いくつかの変数にデフォルト値を設定します。サポートされている署名タイプには以下のものがあります。

CTC_SHAwDSA
CTC_MD2wRSA
CTC_MD5wRSA
CTC_SHAwRSA
CTC_SHAwECDSA
CTC_SHA256wRSA
CTC_SHA256wECDSA
CTC_SHA384wRSA
CTC_SHA384wECDSA
CTC_SHA512wRSA
CTC_SHA512wECDSA

これで、wolfcrypt/test/test.c からの例のように、サブジェクト情報を初期化することができるようになりました：

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

次に、自己署名の証明書を genKey と mg 変数を使って上記の鍵生成例から（もちろん、有効な RsaKey や RNG を使用可能です）生成することができます：

```
byte derCert[4096];
int certSz = MakeSelfCert(&myCert, derCert, sizeof(derCert), &key, &rng);
if (certSz < 0)
        /* certSz contains the error */;
```

これで derCert バッファーには DER フォーマットの証明書が格納されます。PEN フォーマットの証明書が必要な場合は汎用の DerToPem 関数で CERT_TYPE を type に指定して、以下のように生成することができます：

```
byte pemCert[4096];
int pemCertSz = DerToPem(derCert, certSz, pemCert,
```

```
sizeof(pemCert), CERT_TYPE);
if (pemCertSz < 0)
        /* pemCertSz contains error */;
```

以上で pemCert は PEM フォーマットの証明書を保持します。

CA 署名の証明書を生成したい場合は、さらに２、３のステップを必要とします。まず、サブジェクト情報を入れた後、CA 証明書から発行者情報を設定する必要があります。これは SetIssuer()で以下のようにできます：

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

次に、証明書を生成する２ステップのプロセスと、さらにそれに署名します（MakeSelfCert()はこれら二つを一つのステップで行います）。発行者（caKey）とサブジェクト(key)の両方から非公開鍵を生成する必要があります。使用方法全体については test.c の例を参照してください。

```
byte derCert[4096];
int certSz = MakeCert(&myCert, derCert, sizeof(derCert), &key, &rng);
if (certSz < 0);
        /* certSz contains the error */;
certSz = SignCert(&myCert, derCert, sizeof(derCert), &caKey, &rng);
if (certSz < 0);
        /* certSz contains the error */;
```

以上で derCert バッファーには DER フォーマットの CA 署名された証明書が格納されます。PEM フォーマットの証明書が必要な場合は上の自己署名の例を参照してください。MakeCert() と SignCert()は関数パラメータとして RSA または ECC のどちらかが使用されるように指定している点に留意してください。上の例では、RSA 鍵を与えて、ECC 鍵は NULL を渡しています。

7.9 生 ECC 鍵への変換

最近追加されたサポートの一環として、生 ECC 鍵インポート ECC 鍵の PEM から DER への変換が可能となりました。次のような関数をアーギュメントを指定して実行してください。

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

例：

```
#define FOURK_BUF 4096
byte der[FOURK_BUF];
ecc_key userB;
EccKeyToDer(&userB, der, FOURK_BUF);
```

8. デバッグ

8.1 デバッグとログ

wolfSSL はデバッグ機能が限られている環境のために、ログ・メッセージを通じてデバッグをサポートします。ログ機能をオンにするには wolfSSL_Debugging_ON()関数、オフにするには wolfSSL_Debugging_OFF()関数を使用してください。通常ビルド（リリースモード）ではこれらの関数は作用しないようになります。デバッグビルドでは、DEBUG_WOLFSSL を定義するとこれらの関数をオンにすることができます。

wolfSSL2.0 では、ログ機能のコールバック関数が実行時に登録でき、ログがどのようにされるか柔軟に指定できます。ログ機能のコールバックは次のような関数で登録できます：

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

ログレベルについては wolfssl/wolfcrypt/logging.h に定義があり、その実体は logging.c にあります。デフォルトでログは fprintf の stderr に出力されます。

8.2 エラー・コード

wolfSSL は、デバッグを手助けするために有用なエラー・メッセージを提供します。

それぞれの wolfSSL_read()または wolfSSL_write()呼び出しは、ちょうど read()や write()と同じように、成功した場合書き込みバイト数、コネクションのクローズ時には 0、エラー時には-1 を返却します。エラー時には二つの呼び出しを使ってエラーに関してさらに情報を得ることができます。

wolfSSL_get_error()はその時点のエラー・コードを返却します。WOLFSSL オブジェクトと wolfSSL_read()または wolfSSL_write()の返却値をアーギュメントとして受け取り、対応するエラー・コードを返却します

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

さらに人が読める形のエラー・コードの説明を得るためには wolfSSL_ERR_error_string()関数を使用することができます。wolfSSL_get_error()の返却するコードとストレージ・バッファをアーギュメントとして受け取り、対応するエラーの説明をストレージ・バッファ（下の例では errorString）に置きます。

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

ノンブロッキングのソケットを使用している場合、EAGAIN/EWOULDBLOCK でエラーチェックすることができます。さらにより正確には、特定のエラーコードを SSL_ERROR_WANT_READ と SSL_ERROR_WANT_WRITE でテストすることができます。

wolfSSL と wolfCrypt のエラー・コードのリストはユーザ・マニュアル Appendix C (エラー・コード)を参照してください。

9. ライブラリー設計

9.1 ライブラリー・ヘッダー

wolfSSL2.0.0 RC3 リリースでは、ライブラリー・ヘッダーファイルは以下の場所に格納されています：

wolfSSL:	/wolfssl
wolfCrypt:	/wolfssl/wolfcrypt
wolfSSL OpenSSL 互換レイヤー:	/wolfssl/openssl

OpenSSL 互換レイヤー（ユーザマニュアル第 13 章参照）を使用する場合、/wolfssl/openssl/ssl.h ヘッダーをインクルードする必要があります：

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

wolfSSL ネイティブ API のみを使用する場合は /wolfssl/ssl.h ヘッダーをインクルードする必要があります：

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

9.2 開始と終了

すべてのアプリケーションはライブラリーを使用する前に wolfSSL_Init()を呼び出し、プログラム終了時には wolfSSL_Cleanup()を呼び出す必要があります。現在は、これらの関数はマルチユーザーモードにおいてセッションキャッシュのための共有の相互排他を初期化あるいは解放するだけですが、将来、拡張の可能性があるのでこれらを使用することをお勧めします。

9.3 構造体の使用

ヘッダーファイルの格納場所の変更のほかに、wolfSSL2.0.0 RC3 リリースではネイティブ wolfSSL API と wolfSSL OpenSSL 互換レイヤーの間に明らかな区別をするようになりました。この区別によって、ネイティブ wolfSSL API によって使用されるメイン

SSL/TLS 構造体は名前を変更しました。新しい構造体は以下の通りです。OpenSSL 互換レイヤー（ユーザ・マニュアル第 13 章参照）を使用する場合は以前の名前も使用されています。

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 スレッド安全性

wolfSSL はスレッドに対して安全に設計されています。wolfSSL はグローバル・データ、静的データおよび共有オブジェクトを避けているので、マルチスレッドが競合を起こすことなしに並列にライブラリーに入ることができます。しかし、それでも二つの場合で潜在的問題に注意してください。

1. クライアントは WOLFSSL オブジェクトをマルチスレッド間で共有してもかまいませんが、アクセスは同期しなければいけません。例えば、二つの異なるスレッドから同時に同じ SSL ポインタに read/write するようなことはサポートされていません。

   wolfSSL はより過激な（制限のきつい）スタンス、つまり「複数ユーザ間で共有する関数に誰かが入った場合、他のユーザをロックアウトする」というようなスタンスもとり得たかもしれません。しかし、この粒度レベルは直観に反します。すべてのユーザ（シングルスレッドでさえ）はロッキングの代償を払うことになり、マルチスレッドではスレッド間で共有されていなくてもライブラリーに再入不可となってしまいます。この代償は大きすぎるように思われ、wolfSSL は共有オブジェクトの同期責任をユーザの手にゆだねています。

2. WOLFSSL ポインターを共有するほかに、ユーザは wolfSSL_new()に構造体を渡す前に WOLFSSL_CTX を完全に初期化する責任があります。同じ WOLFSSL_CTX は複

数の WOLFSSL 構造体を生成することができますが、いったん WOLFSSL オブジェクトが生成された後は wolfSSL_new()生成と WOLFSSL_CTX に対するその後の（あるいは同時の）変更は反映されません。

繰り返しになりますが、WOLFSSL_CTX に対する書き込みアクセスはマルチスレッドに対して同期する必要があり、上記説明の同期とアップデート問題を回避するために、WOLFSSL_CTX のシングルスレッドの初期化をお勧めします。

9.5 入力と出力バッファー

wolfSSL は入力と出力のために小さな静的バッファーを使用しています。デフォルトでは 128 バイト、wolfssl/internal.h の RECORD_SIZE でコントロールされています。受け取る入力レコードが静的バッファーよりサイズ的に大きい場合は、要求を処理するために動的バッファーが一時的に使用され、解放されます。静的バッファーのサイズは最大 MAX_RECORD_SIZE、2 の 16 乗または 16,384 まで指定することができます。

する動的メモリーを必要としない場合で、wolfSSL の以前の方法、16Kb の静的バッファーのほうの使用を希望する場合、LARGE_STATIC_BUFFER を定義してそのオプションを選択することができます。

小さな静的バッファーが使用され、バッファー・サイズより大きな wolfSSL_write()を要求した場合、最大 MAX_RECORD_SIZE の動的ブロックが使用されデーターを送信します。現時点のバッファー・サイズの分のデータだけを送信したい（そして、動的メモリーを避けたい）場合は STATIC_CHUNKS_ONLY を定義して、これを行うことができます。