CyaSSL ユーザ・マニュアル

（この資料は **CyaSSL User Manual**, Version 2.10
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 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。CyaSSL のビルド方法、使い方、CyaSSL の機能概説、移植性強化点、サポートその他を解説します。

なぜ CyaSSL 選択するか？

組込み向けソリューションとして CyaSSL を選択する理由が数多く存在します。簡単には、まずメモリーサイズ（CyaSSL は 30k バイトの小ささでビルド可能です）、そして最新の標準への準拠（TLS1.1 と 1.2、DTLS)、新しい暗号サポート（ストリーム型暗号を含む）、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. CyaSSL のビルド

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。CyaSSL 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.yassl.com/forums ）または support@yassl.com に直接お問い合わせください。

本章では Unix や Windows で CyaSSL をビルドする方法を解説するとともに、非標準環境での CyaSSL 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 CyaSSL SSL チュートリアルも合わせて参照してください。

2.1 CyaSSL ソース・コードの入手

CyaSSL の最新版は yaSSL 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。

http://yassl.com/yaSSL/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：CyaSSL2.0.0rc3 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 CyaSSL 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 CyaSSL のビルドでは autconf システムを使用してください。CyaSSL 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

CyaSSL 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、CyaSSL 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

または、testsuite とともに標準の CyaSSL API と crypto テストも実行するために autoconf を使用して：

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。CyaSSL 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 CyaSSL 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/libcyassl.la

2.3 Windows 上でのビルド

VS2005/VS2008：Visual Studio2005/2008 へのソリューションはインストールの root ディレクトリーに格納されています。

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。

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。

2.4 非標準環境でのビルド

正式サポートではありませんが、CyaSSL 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. src/ と ctaocrypt/src からのすべての .c ファイルを同じディレクトリーに置いてください。
2. cyassl/ インクルード・ディレクトリーを上記と同じディレクトリーにコピーしてください。このディレクトリーは CyaSSL と CTaoCrypt のためのすべての.h ファイルを含みます。
3. すべての CyaSSL ヘッダーはソースディレクトリー下の/cyassl サブディレクトリーにあってもなおかつ、いくつかのビルドシステムではヘッダーファイルの所在を明示的に知る必要があるものもあり、それを指定する必要があるかも知れません。
4. CyaSSL のデフォルトは、コンフィグレーション・プロセスでビッグ・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。ユーザはコンフィグレーション・プロセス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。

5. CyaSSL は 64 ビット型を有効利用して高速化します。コンフィグレーション・プロセスでは long か long long が 64bit かどうか判定し、もしそうなら、定義を設定します。ですから、ユーザのシステムで 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。
6. ライブラリーのビルドで何か問題があった場合は info@yassl.com にご連絡ください。
7. 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 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 CyaSSL 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_CYASSL_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_CYASSL_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 CyaSSL 用のものが src/internal.c に、CTaoCrypt 用のものが ctaocrypt/src/asn.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_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、お勧めしません。

SMALL_SESSION_CACHE は CyaSSL 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。

2.4.2デフォルトで無効の機能を有効にする

CYASSL_CERT_GEN は CyaSSL の証明書生成機能をオンにします。詳細は本マニュアルの第 7 章を参照してください。

CYASSL_DTLS は DTLS（データグラム TLS）の使用をオンにします。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。

CYASSL_KEY_GEN は CyaSSL の RSA 鍵生成機能をオンにします。詳しい情報は本マニュアル第七章を参照してください。

CYASSL_RIPEMD は RIPEMD-160 サポートを利用可にします。

CYASSL_SHA512 は SHA-512 サポートを利用可にします。

DEBUG_CYASSL はデバッグ・モードでビルドします。CyaSSL 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_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。デフォルトではオフで、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。

OPENSSL_EXTRA はライブラリーにさらなる OpenSSL 互換性を組込み、CyaSSL 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 CyaSSL、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オフです。

TEST_IPV6 はアプリケーションのテストにおいて IPv6 のテストをオンにします。
CyaSSL 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。

2.4.3CyaSSL カスタマイズおよびポーティング

CYASSL_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。

CYASSL_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 CyaSSL 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 ランダム数値生成の使用を不可にします。これが定義された場合には、OS 固有の enerateSeed()関数 (“ctaocrypt/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で呼び出されるのか、testsuite ドライバーアプリケーションを通して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における使用のためのみに必要となります：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。CyaSSL 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。CyaSSL 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。“cyassl_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 の構造体の使用を不可にします。XTIME, XGMTIME, XVALIDATE_DATE を定義、実現するために必要となります。

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 fastmath (USE_FAST_MATH) 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。

CYASSL_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 ctaocrypt/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ことはありません。

2.4.5 性能改善

CYASSL_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 CyaSSL 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。

2.4.6スタックおよびチップ固有の定義

CyaSSL 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 ./cyassl/ctaocrypt/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。

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

CYASSL_LWIP は LwIP TCP/IP スタック(http://savannah.nongnu.org/projects/lwip/)とともに CyaSSL を使用する場合に定義することができます。

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 CYASSL_LWIP も定義します。

IPHONE は iOS とともに使用するためのビルドの場合、定義することができます。

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することができます。

MICRIUM は Micrium の µC/OS (www.micrium.com) 向けのビルドの際に定義することができます。

MICROCHIP_PIC32 は Microchip の PIC32 プラットフォーム(www.microchip.com) 向けのビルドの際に定義することができます。

THREADX は ThreadX の RTOS (www.rtos.com) 向けのビルドの際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 CyaSSL 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 CyaSSL 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を高速化されます。どちらのモードも必要ならば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	CyaSSL デバッグサポートを有効化
--enable-small	無効	可能な限り最も小さいビルドを有効化
--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	CyaSSL の DTLS サポートを有効化

--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-ipv6	無効	IPv6 のテストを有効化。CyaSSL 自身は IP 中立。
--enable-fastmath	無効	BigInt 向け fast math を有効化
--enable-fasthugemath	無効	BigInt 向け fast math + huge code を有効化
--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-sniffer	無効	Sniffer サポートの有効化
--enable-aesni	無効	CyaSSL の Intel AES-NI サポートの有効化
--enable-ripemd	無効	CyaSSL の RIPEDM-160 サポートの有効化
--enable-sha512	無効	CyaSSL 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
--disable-shared	無効	共有 CyaSSL ライブラリーのビルドを無効化
--disable-static	無効	静的 CyaSSL ライブラリーのビルドを無効化
--with-libz	無効	libz 圧縮を含むよう選択
--enable-psk	無効	事前共有鍵を有効化
--enable-hc128	無効	HC-128 ストリーム暗号を有効化
--enable-ntru	無効	NTRU のビルドを有効化（要ライセンス）
--enable-webServer	無効	yaSSL 組込み Web サーバと互換のビルドを有効化
--enable-gcc-lots-o-warnings	無効	多くの追加のワーニングをオンに。
--enable-noFilesystem	無効	ファイルシステム使用を取り除く

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_CYASSL 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 CyaSSL_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 CyaSSL_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。

Small - small ビルドオプションの有効化すると、最小の CyaSSL ライブラリーを生成します。これにより、TLS、HMAC、SHA-256、エラーメッセージその他希望している機能も削除されるかも知れません。デフォルトのビルドが大きすぎ、機能が失われても問題無い場合にのみ使用してください。無効化される機能は以下の通りです。TLS, HMAC, AES, 3DES, SHA-256, Error Strings, HC-128, RABBIT, PSK, DSA, および DH。

Single Threaded - 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オフされます。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。

OpenSSL 拡張 – 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。CyaSSL 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。CyaSSL 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。

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 CyaSSL 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。CyaSSL 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 に手作業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、cyassl/ctaocrypt/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 CyaSSL 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。
bigcashe - 「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。

sniffer – sniffer（SSL 検査）サポートを有効化すると、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。

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。

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。CyaSSL 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。

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、CyaSSL 共有ライブラリーはビルドから外されます。

static の無効化 – static を無効化オプションを有効化すると、CyaSSL static ライブラリーはビルドから外されます。

libz – libz の有効化で、CyaSSL 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、CyaSSL_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じたときの時間より長くかかってしまいます。

PSK – 事前共有鍵サポートは、あまり広く使われていないので、デフォルトではオフになっています。この機能を有効化するには、単にオンにするだけで、そのほかの手順は必要ありません。

HC-128 – このストリーム暗号化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にも暗号化 union の領域を占有してしまします。デフォルトビルドをできる限りさまざまな側面で小さくするために、この暗号化はデフォルトでは無効化されています。こお暗号化および対応する暗号化スーツを使用するためには、単にオプションをオンにしてください。そのほかの手順は必要ありません。

NTRU　- これは CyaSSL で NTRU 暗号化スーツ機能をオンにします。これらのビルドと使用には NTRU ライセンスが必要です。NTRU ライセンスとライブラリーが無いとビルドに失敗します。

webserver – これは yaSSL 組込み Web サーバをフル機能でビルド可能にする標準ビルドのために要求される関数をオンにします。

noFilesystem – これによってファイルシステムの使用を簡単に不可にできます。このオプションは NO_FILESYSTE を定義します。

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 CyaSSL 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、CyaSSL 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 CyaSSL 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、CyaSSL 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@yassl.com までお知らせください。

3. 使用方法

3.1. 全体説明

CyaSSL は本マニュアル第二章 CyaSSL 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 CyaSSL 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 CyaSSL 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 CyaSSL とその暗号ライブラリーCTaoCrypt 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。

CyaSSL はすべての例題とテストを CyaSSL 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、CyaSSL 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/

CN=www.taosoftdev.com/emailAddress=info@yassl.com
Client message: hello cyassl!
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 CyaSSL 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 CyaSSL をテストすることができます。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：

```
./examples/client/client gmail.google.com 443
peer's cert info:
issuer : /C=US/O=Google Inc/CN=Google Internet Authority
subject: /C=US/ST=California/L=Mountain View/O=Google Inc/
CN=*.google.com
SSL connect ok, sending GET...
Server response: HTTP/1.0 302 Found
Cache-Control: private
Location: http://www.google.com
Content-Type: text/html; charset=UTF-8
Content-Length: 218
Date: Tue, 16 Feb 2010 22:25:02 GMT
Server: GFE/2.0
X-XSS-Protection: 0
```

これは、クライアントが gmail.google.com に https のポート 443 でコネクトするために、通常の GET を送信していることを示します。

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 CyaSSL 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。
コマンドラインのアーギュメントが一つ与えらた場合、クライアントは localhost の CyaSSL のデフォルトポート 11111 に、アーギュメンで指定された回数コネクトしようと

し、SSL_connect()を実行するのにかかった時間をm秒単位の平均時間を、例えば以下のように表示します。

./examples/client/client 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/cyassl/test.h で変更することができます。yasslIP と yasslPort 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。

3.4. サーバの例

サーバの例ではクライアント認証とクライアントが証明書を提示しない場合に失敗する簡単な SSL サーバをデモします。ただ一つのクライアントのコネクトが受け付けられ、サーバは終了します。ノーマル・モードにおけるサンプルクライアント（コマンド行のアーギュメント無し）でサンプルサーバに対してうまく動作しますが、サンプル・クライアントのコマンド行にアーギュメントを指定した場合は、クライアントの証明書はロードされずCyaSSL_connect()はうまく行きません。サーバは "-245, peer didn't send cert" のエラーを出力します。

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 CyaSSL”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。

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”、“cyassl”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
cyassl
cyassl
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 CyaSSL 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。

CyaSSL/CTaoCrypt の SSL 性能評価をされる CyaSSL 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 CyaSSL にバンドル、提供されています。CyaSSL はデフォルトではすべての暗号操作に CTaoCrypt 暗号ライブラリーを使用しています。使用する暗号は SSL/TLS の性能に非常に依存性が高いので、ベンチマーク・アプリケーションは CTaoCrypt アルゴリズム上で性能テストを実行しています。

ctaocrypt/benchimark のベンチマーク・ユーティリティーが CTaoCrypt の暗号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：

./ctaocrypt/benchmark/benchmark
AES 5 megs took 0.034 seconds, 148.13 MB/s
ARC4 5 megs took 0.016 seconds, 312.54 MB/s
HC128 5 megs took 0.004 seconds, 1214.12 MB/s
RABBIT 5 megs took 0.011 seconds, 459.31 MB/s
3DES 5 megs took 0.233 seconds, 21.48 MB/s
MD5 5 megs took 0.011 seconds, 464.68 MB/s
SHA 5 megs took 0.018 seconds, 278.27 MB/s
SHA-256 5 megs took 0.040 seconds, 124.32 MB/s
RSA 1024 encryption took 0.04 milliseconds, avg over 100 iterations
RSA 1024 decryption took 0.45 milliseconds, avg over 100 iterations
DH 1024 key generation 0.21 milliseconds, avg over 100 iterations
DH 1024 key agreement 0.22 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便利です。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。

3.8. クライアント・アプリケーションを CyaSSL 用に変更

このセクションでは、CyaSSL 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 CyaSSL 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。「CyaSSL 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したい方は、ユーザ・マニュアル第十三章を参照してください。

1. CyaSSL ヘッダーをインクルードする

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

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

```
result = read(fd, buffer, bytes);
```
は
```
result = CyaSSL_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 CyaSSL_write()に変更する。

```
result = write(fd, buffer, bytes);
```
は
```
result = CyaSSL_write(ssl, buffer, bytes);
```
となります。

4. CyaSSL_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 CyaSSL_read() または CyaSSL_write() 呼び出しで、まだならば CyaSSL_connect() が呼び出されます。

5. CyaSSL と CYASSL_CTX を初期化します。CYASSL オブジェクトをいくつ生成する場合でも、一つの CYASSL_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:

```
CyaSSL_Init();
CYASSL_CTX* ctx;
if ( (ctx = CyaSSL_CTX_new(CyaTLSv1_client_method())) == NULL) {
    fprintf(stderr, "CyaSSL_CTX_new error.¥n");
    exit(EXIT_FAILURE);
}
if (CyaSSL_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、CYASSL を生成します:

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

7. エラーチェック。各 CyaSSL_read() または CyaSSL_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 = CyaSSL_get_error(ssl, 0);
CyaSSL_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 CYASSL 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:

```
CyaSSL_free(ssl);
```

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

```
CyaSSL_CTX_free(ctx);
CyaSSL_Cleanup();
```

3.9. サーバ・アプリケーションを CyaSSL 用に変更

このセクションでは、CyaSSL のネイティブ API を使用してサーバ・アプリケーションに CyaSSL 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。「CyaSSL 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、
CyaSSL_CTX_new(CyaTLSv1_client_method())
を
CyaSSL_CTX_new(CyaTLSv1_server_method())

または
CyaSSL_CTX_new(CyaSSLv23_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 (CyaSSL_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 (CyaSSL_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);
}
```

4. 機能

CyaSSL 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 CyaSSL 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 CyaSSL 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 CyaSSL リリースに含まれる機能、特徴の一覧を示します。

CyaSSL機能、特徴（Ver 2.0.8）	利点
SSLバージョン3.0、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
ビルド・オプションと実行環境により、最小30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
3-36kBの実行時メモリー	最小の動的メモリー使用
DTLS1.0サポート（クライアントおよびサーバ）	ストリーム・メディア対応
OpenSSL互換レイヤー	標準APIによりOpenSSLからの移行を容易化
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
RSA鍵生成	高速な実行時鍵生成をサポート
PSK（事前共有鍵）	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

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
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
クライアント認証サポート	証明書をクライアント検証に使用
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 ・ OS抽象化レイヤー ・ カスタムI/O抽象化レイヤー	開発者に移植性と柔軟性を提供
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
各種アルゴリズムのサポート - MD2, MD4, MD5, SHA-1, SHA-256, SHA-512, RIPEMD-160 - AES, DES, 3DES, ARC4, RABBIT, HC-128 - RSA, DSS, DH, EDH, NTRU - HMAC, PBKDF2, PKCS#5	– 複数のハッシュ関数が利用可能 – 3つのブロック暗号化、3つのストリーム暗号化 – 4つの公開鍵の選択肢 – パスワード・ベースの鍵
Webサーバのサポート – GoAhead, Mongoose, Lighttpd等々	複数の軽量組込みWebサーバの選択肢。 CyaSSLは弊社yaSSL組込みWebサーバでも使用

表1：CyaSSL の機能

4.2　プロトコル・サポート

CyaSSL は SSL3.0, TLS(1.0, 1.1 および 1.2)および DTLS1.0 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）。CyaSSL 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サーバ機能

CyaDTLSv1_server_method(void);　　// DTLS 1.0

CyaSSLv3_server_method(void);　　// SSL 3.0

CyaTLSv1_server_method(void);　　// TLS 1.0

CyaTLSv1_1_server_method(void);　　// TLS 1.1

CyaTLSv1_2_server_method(void);　　// TLS 1.2

CyaSSLv23_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

CyaSSLはCyaSSLv23_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　　クライアント機能

CyaDTLSv1_client_method(void);　　// DTLS 1.0

CyaSSLv3_client_method(void);　　// SSL 3.0

CyaTLSv1_client_method(void);　　// TLS 1.0

CyaTLSv1_1_client_method(void);　　// TLS 1.1

CyaTLSv1_2_client_method(void);　　// TLS 1.2

CyaSSLv23_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

CyaSSLはCyaSSLv23_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

CyaSSL 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、CyaSSLv23_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、CyaSSLv23_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。CyaSSL 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、CyaSSL 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。CyaSSL は IPv6 上で動作します。

CyaSSL 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、CyaSSL 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

4.2.5DTLS

上のリストに挙げたように、CyaSSL 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。

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

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	k =7680	F = 384 - 511

		N = 384		
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、CyaSSL2.0.0 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 CyaSSL 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。

CyaSSL 暗号化スーツ（Ver 2.0.8）	
TLS_DHE_RSA_WITH_AES_256_CBC_SHA TLS_DHE_RSA_WITH_AES_128_CBC_SHA TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 TLS_RSA_WITH_AES_256_CBC_SHA TLS_RSA_WITH_AES_128_CBC_SHA TLS_RSA_WITH_AES_256_CBC_SHA256	TLS 暗号化スーツ

TLS_RSA_WITH_AES_128_CBC_SHA256 TLS_PSK_WITH_AES_256_CBC_SHA TLS_PSK_WITH_AES_128_CBC_SHA 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	
SSL_RSA_WITH_RC4_128_SHA SSL_RSA_WITH_RC4_128_MD5 SSL_RSA_WITH_3DES_EDE_CBC_SHA	SSL 暗号化スーツ
TLS_RSA_WITH_HC_128_CBC_MD5 TLS_RSA_WITH_HC_128_CBC_SHA TLS_RSA_WITH_RABBIT_CBC_SHA	TLS 暗号化スーツ（ストリーム暗号化）

表４：CyaSSL 暗号化スーツ

4.3.3 ブロックおよびストリーム暗号化

CyaSSL は AES、DES および 3 DES のブロック暗号化および RC4、RABBIT および HC-128 のストリーム暗号化をサポートします。AES、DES、３DES、RC4 および RABBIT はデフォルトで有効化されています。HC-128 は CyaSSL 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 CTaoCrypt 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。CyaSSL 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、CyaSSL 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。

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

CyaSSL は、MD2、MD4、MD5、SHA-1、SHA-256、SHA-512 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 CTaoCrypt 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

CyaSSL は RSA、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、CyaSSL 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 CTaoCrypt レファレンスのセッション 10.5 を参照ください。

CyaSSL は NTRU 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。CyaSSL 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 CyaSSL ライブラリーに組み込まれます。CyaSSL 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、CyaSSL 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。CyaSSL 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 は非公開鍵 Blob です。

CyaSSL 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 CyaSSL 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 AES-NI サポート

AES は全世界の政府関係で使用されている暗号化標準で、CyaSSL 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。CyaSSL 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

CyaSSL 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 CyaSSL 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 CyaSSL 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 GCC4.3.3 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/

4.3.7 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。CyaSSL は PKCS #5, PKCS #8 および PKCS #12 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。CyaSSL 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。

CyaSSL 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。

フルの例題については ctaoscrypt/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 特定暗号化を強制指定

デフォルトで、CyaSSL 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”);
```

4.4 SSL 検査（Sniffer）

CyaSSL1.5.0 リリース以降、CyaSSL 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理由で便利なことです。

- 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 CyaSSL 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秘密鍵を設定する。
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。

CyaSSL の sniffer サポートと例題全体を参照するには、CyaSSL 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、CyaSSL 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。

4.5 圧縮

CyaSSL 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 CyaSSL_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。

4.6 事前共有鍵

CyaSSL 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_SHA

これらのスーツは自動的に CyaSSL 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 CyaSSL_CTX_set_cipher_list()関数を使用してください。

クライアント側では、CyaSSL_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。<CyaSSL_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 cyassl/test.h でインプリメントされています。

CyaSSL 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。

4.7 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、CyaSSL 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。

CyaSSL_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
CyaSSL_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。
CyaSSL_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。

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

CyaSSL 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。

4.8 ハンドシェイク修正

4.8.1 ハンドシェイク・メッセージのグループ化

必要の場合、CyaSSL 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、

CyaSSL_CTX_set_group_messages(ctx);

を使用するか、SSL オブジェクト・レベルで、

CyaSSL_set_group_messages(ssl);

で行うことが可能です。

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

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です。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。CyaSSL メモリー関数は cyassl/ctaocrypt/type.h にあります。

CyaSSL 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。Cyassl/ctaocrypt/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 CyaSSL_SetAllocators(CyaSSL_Malloc_cb malloc_function,
                         CyaSSL_Free_cb free_function,
                         CyaSSL_Realloc_cb realloc_function);
```

コールバックの関数プロトタイプは cyassl/ctaocrypt/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

CyaSSL 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、CyaSSL 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

CyaSSL 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 ctaocrypt/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、CyaSSL 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

CyaSSL 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 CYASSL_CTX ごとに CyaSSL_SetIOSend() と CyaSSL_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 CyaSSL_SetIORecv(CYASSL_CTX *ctx, CallbackIORecv CBIORecv)
{
        ctx->CBIORecv = CBIORecv;
}
void CyaSSL_SetIOSend(CYASSL_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、CYASSL オブジェクト（セッション）毎に CyaSSL_SetIOWriteCtx() と CyaSSL_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。

CYASSL_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

CyaSSL の OS 抽象化レイヤーは CyaSSL 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。cyassl/ctaocrypt/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 CTaoCrypt 向けは cyassl/ctaocrypt/types.h、CyaSSL 向けのものは cyassl/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

CyaSSL 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 CyaSSL 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, Micrium's µC/OS, FreeRTOS, Freescale MQX, Nucleus

5.3. サポートするチップ・メーカー

CyaSSL 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, Motorola, mbed, Freescale そのほかが含まれます。

6. コールバック

6.1.ハンドシェイク・コールバック

CyaSSL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。CyaSSL HandShakeCallBackを使用するためには、拡張関数*CyaSSL_connect_ex()* または *CyaSSL_accept_ex()* を使用して：

int CyaSSL_connect_ex(CYASSL*, HandShakeCallBack, TimeoutCallBack,Timeval)
int CyaSSL_accept_ex(CYASSL*, HandShakeCallBack, TimeoutCallBack,Timeval)

HandShakeCallBack は以下のように定義されます:

typedef int (*HandShakeCallBack)(HandShakeInfo*);

HandShakeInfo はcyassl/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

CyaSSL 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 CyaSSL 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。

7. 鍵と証明書

7.1 サポートするフォーマットとサイズ

CyaSSL 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 CyaSSL_CTX_load_verify_locations()関数を使用してロードすることができます：

```
int CyaSSL_CTX_load_verify_locations(CYASSL_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、CyaSSL 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 CyaSSL_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 CyaSSL_CTX_use_certificate_file(CYASSL_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 CyaSSL_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 CyaSSL_CTX_use_certificate_chain_file(CYASSL_CTX *ctx,
                                          const char *file);
```

7.2.3非公開鍵のロード

サーバー非公開鍵は CyaSSL_CTX_use_PrivateKey_file()関数を使用してロードすることができます。

```
int CyaSSL_CTX_use_PrivateKey_file(CYASSL_CTX *ctx,
                                   const char *keyFile,
                                   int type);
```

keyFile は非公開鍵、type は非公開鍵のフォーマット（例えば SSL_FILETYPE_PEM）です。

7.3 証明書チェーンの検証

CyaSSL 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、CyaSSL 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。

7.4 サーバー証明書のドメイン名チェック

CyaSSL 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。CyaSSL 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、CyaSSL_connect()の前に以下の関数呼び出しで行います：

CyaSSL_check_domain_name(CYASSL* ssl, cons char* dn)；

CyaSSL 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、CyaSSL_connect()は正常に処理しますが、マッチしない場合は CyaSSL_connect()は致命的エラーを返却し、CyaSSL_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。CyaSSL 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 CyaSSL_CTX_load_verify_buffer(CYASSL_CTX*, const unsigned char*, long)
int CyaSSL_CTX_use_certificate_buffer(CYASSL_CTX*, const unsigned char*,
long, int)
int CyaSSL_CTX_use_PrivateKey_buffer(CYASSL_CTX*, const unsigned char*,
long, int)
int CyaSSL_CTX_use_certificate_chain_buffer(CYASSL_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。

7.6 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 CyaSSL 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 CyaSSL_X509_get_serial_number(CYASSL_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 cyassl/test.h に含まれています。

7.7 RSA 鍵生成

CyaSSL 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 CYASSL_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 ctaocrypt/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、CyaSSL 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。

7.7.1RSA 鍵生成の注意点

RSA 非公開鍵は公開鍵も含みますが、CyaSSL 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。CyaSSL 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。

CyaSSL 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 CyaSSL にロードすることができます。

7.8　証明書の生成

CyaSSL 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 CYASSL_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 cyassl/ctaocrypt/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 MD5_WITH_RSA, SHA_WITH_RSA, また SHA256_WITH_RSA などがあります。

これで、ctaocrypt/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, "yaSSL", NAME_SIZE);
strncpy(myCert.subject.unit, "Development", NAME_SIZE);
strncpy(myCert.subject.commonName, "www.yassl.com", NAME_SIZE);
strncpy(myCert.subject.email, "info@yassl.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。

8. デバッグ

8.1 デバッグとログ

CyaSSL 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 CyaSSL_Debugging_ON()関数、オフにするには CyaSSL_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_CYASSL 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。

CyaSSL2.0 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 CyaSSL_SetLoggingCb(CyaSSL_Logging_cb log_function);
typedef void (*CyaSSL_Logging_cb)(const int logLevel, const char *const logMessage);
```

ログレベルについては cyassl/ctaocrypt/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

CyaSSL 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 CyaSSL_read()または CyaSSL_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。

CyaSSL_get_error()はその時点のエラー・コードを返却します。CYASSL オブジェクトと CyaSSL_read()または CyaSSL_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 = CyaSSL_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 CyaSSL_ERR_error_string() 関数を使用することができます。CyaSSL_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];
CyaSSL_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。

CyaSSL と CTaoCrypt 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

CyaSSL2.0.0 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：

CyaSSL:	/cyassl
CTaoCrypt:	/cyassl/ctaocrypt
CyaSSL OpenSSL Compatibility Layer:	/cyassl/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、/cyassl/openssl/ssl.h ヘッダーをインクルードする必要があります：

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

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

```
#include <cyassl/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 CyaSSL_Init()を呼び出し、プログラム終了時には CyaSSL_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、CyaSSL2.0.0 RC3 リリースではネイティブ CyaSSL API と CyaSSL 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 CyaSSL 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。

CYASSL (previously SSL)
CYASSL_CTX (previously SSL_CTX)
CYASSL_METHOD (previously SSL_METHOD)
CYASSL_SESSION (previously SSL_SESSION)
CYASSL_X509 (previously X509)
CYASSL_X509_NAME (previously X509_NAME)
CYASSL_X509_CHAIN (previously X509_CHAIN)

9.4 スレッド安全性

CyaSSL はスレッドに対して安全に設計されています。CyaSSL 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 CYASSL 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。

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

CYASSL 構造体を生成することができますが、いったん CYASSL オブジェクトが生成された後は CyaSSL_new()生成と CYASSL_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、CYASSL_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、CYASSL_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

CyaSSL 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、cyassl/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、CyaSSL 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 CyaSSL_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。

9.6 安全な再ネゴシエーション

数年前、SSL の再ネゴシエーション（CVE-2009-3555）関係で脆弱性が発見されました。この脆弱性に対応して、問題を緩和するために RFC4756（安全な再ネゴシエーション：Secure Renegotiation）が作られました。

再ネゴシエーションのサポートは設計のはじめから CyaSSL から除外されました。これは CyaSSL は RFC4756 を実現しておらず、上記の脆弱性に感染しにくいということを意味します。