Zeutro LLC

Encryption & Data Security

OpenABE ライブラリ

C++ API ガイド

OpenABE C++ API

バージョン 1.0

内容

1 OpenABE ライブラリ

1.1 OpenABE とは?

OpenABE は、いくつかの属性ベースの暗号化を実装した C/C++ライブラリです。(ABE)方式は、さまざまなデータアットレスト・アプリケーションで使用できる。このツールキットの目標は、アーキテクチャ・レベルでセキュリティ保証を組み込んだ効率的な実装を開発することである。最新のアプリケーションをサポートするために、OpenABE は暗号化と署名のための単純化された API(または crypto_box のようなインターフェース)を提供する。また、OpenABE には以下の特長がある:

- **モジュール方式**:OpenABE では、開発者はアプリケーションのロジックを更新することなく、暗号方式を別の方式に置き換えることができます。
- **包括的**:OpenABE は、一般的な暗号タスクを実行するのに必要なルーチンをサポートしています。
- **拡張性**:OpenABE は、比較的少ない労力で、いくつかの機能的な暗号化方式をサポートすることができます。
- **ベストプラクティス**:OpenABE は、暗号化設計のベストプラクティスを取り入れています。これには、選択暗号文攻撃に対するセキュリティ、対称鍵の伝送のためのシンプルなインターフェース、大きなデータオブジェクトの暗号化の実行などが含まれます。

1.2 コア暗号アルゴリズム

図 1 の OpenABE アーキテクチャ図を参照:

OpenABE は多くのコア暗号アルゴリズムを提供している:

- キーポリシーABE や暗号文ポリシーABE など、複数のタイプの属性ベース暗号化(ABE)キーカプセル化(KEM)スキームをサポート。

- 各 ABE KEM スキームタイプで選択暗号文セキュリティ（CCA）をサポート。これには、選択暗号文セキュリティ付きの公開鍵暗号化も含まれます。
- 認証された共通鍵暗号化だけでなく、デジタル署名もサポートしています。
- 線形秘密分散方式（LSSS）、鍵導出関数（KDF）、擬似ランダム関数（PRF）、擬似ランダム生成器（PRG）など、一般的な暗号関数をサポート。

1.2　コア暗号アルゴリズム

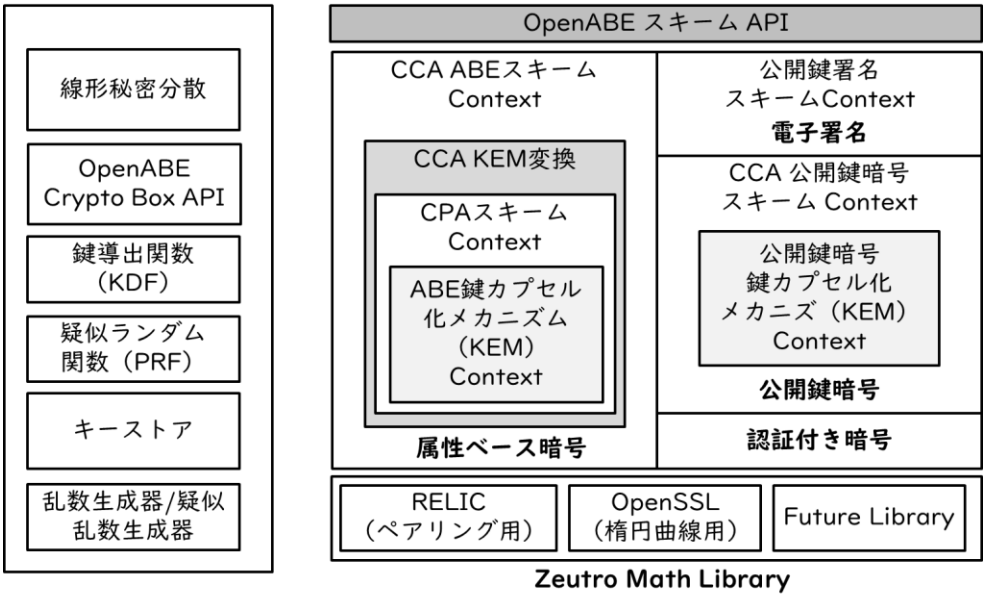

図 1：OpenABE アーキテクチャ図

OpenABE は他にも以下のような便利な機能をサポートしている：

- 楕円曲線とペアリング演算のための拡張可能な Zeutro 数学ライブラリ(ZML)。現時点では RELIC でのみインスタンス化できます。ZML は、C 抽象数学インターフェイスに準拠していれば、他の数学ライブラリもサポートするように設計されています。
- 交換可能な乱数生成器（RNG）。OpenABE ABE CCA 変換に必要な ZML で、RNG を擬似乱数（または PRNG）と入れ替えることができます。
- ABE および PKE/PKSIG 鍵をメモリ上で管理するための鍵ストア実装。

1.3 インストール

このセクションでは、OpenABE ソースコード（libopenabe-）のインストールについて説明します。

1.0.0-src.tar.gz)を様々なプラットフォーム上で動作させることができます。OpenABE は現在、Linux、Mac OS X、Windows の複数のバージョン/ディストリビューションを含む複数のオペレーティングシステムをサポートしています。

1.3.1 Debian/Ubuntu ベースの Linux

Ubuntu または Debian Linux ベースのディストロで OpenABE をコンパイルするには、まずソースディレクトリから deps/install_pkgs.sh スクリプトを実行して、OpenABE のシステム固有の依存関係を以下のようにインストールします：

```
cd libopenabe-1.0.0/ sudo -
E ./deps/install_pkgs.sh
```

システム・セットアップごとに一度だけ行う必要があることに注意してください。完了したら、次のように OpenABE のコンパイルに進みます：

```
../env
make
make test
```

この時点ですべてのユニットテストがパスしているはずなので、次のように標準的な場所 (/usr/local) に OpenABE をインストールします。

インストールパスのプレフィックスを変更するには、 libopenabe-1.0.0/Makefile の INSTALL_PREFIX 変数を変更してください。

1.3.2 CentOS と RedHat Linux

前回と同様に、まずソースディレクトリからスクリプトを実行し、OpenABE の依存関係をセットアップします：

```
cd libopenabe-1.0.0/
sudo ./deps/install_pkgs.sh scl
enable devtoolset-3 bash
```

システム・セットアップごとに一度だけ行う必要があることに注意してください。完了したら、次のように OpenABE のコンパイルに進みます：

```
../env
make
make test
```

この時点ですべてのユニットテストがパスしているはずなので、次のように標準の場所 (/usr/local) に OpenABE をインストールすることができます： sudo make install

インストールパスのプレフィックスを変更するには、libopenabe-1.0.0/Makefile の INSTALL_PREFIX 変数を変更してください。

1.3.3 マック OS X

Mac OS X の場合、deps/install_pkgs.sh スクリプトを実行する前に、homebrew がインストールされている必要があります。その後、以下を実行する（2 つ目のステップでは（sudo）が必要かもしれない）：

```
cd libopenabe-1.0.0/ ./deps/install_pkgs.sh
```

システム・セットアップごとに一度だけ行う必要があることに注意してください。完了したら、次のように OpenABE のコンパイルに進みます：

```
../env
make
make test
```

この時点ですべてのユニットテストがパスしているはずなので、次のように標準的な場所（/usr/local）に OpenABE をインストールします。

インストールパスのプレフィックスを変更するには、libopenabe-1.0.0/Makefile の INSTALL_PREFIX 変数を変更してください。

1.3.4 ウィンドウズ

OpenABE を Windows 7、8、10 上でビルドするには、Windows ネイティブ・バイナリをビルドするための GNU ツールチェイン・ポートである Mingw-w64 をダウンロードしてインストールする必要があります。私たちは、Minimal SYStem 2（MSYS2）と一緒にパッケージ化された Mingw-w64 ポートを使用しています。MSYS2 は、GNU ツール（GCC など）、bash、Arch Linux の Pacman を使ったパッケージ管理でソフトウェアをビルドするための、エミュレートされた POSIX 準拠の環境です。これらのコンパイラでコンパイルされたバイナリはスタンドアロンなので cygwin.dll を必要としません。

1. msys2-x86_64-latest.exe をダウンロードして実行します。PATH 解決の問題を避けるため、インストール・ディレクトリに C:/を選択してください。

2. MSYS2 シェルを起動し、以下のコマンドを実行する： update-core

3. MSYS2 シェルを閉じて、MinGW-w64 Win64 シェルを起動します。MSYS2 を起動すると、プロンプトに起動したバージョンが表示されます。

4. プリインストールされている MSYS2 パッケージ（および関連ツールのインストール）を更新し、プロンプトが表示されたらシェルを閉じて、MinGW-w64 Win64 シェルを再起動します。以下のコマンドを実行して、プロセスを開始する：

   pacman -Sy pacman -Su base-devel unzip git wget mingw-w64-i686-toolchain ¥ mingw-w64-x86_64-toolchain mingw-w64-i686-cmake mingw-w64-x86_64-cmake

5. 以下のコマンドを実行して、必要なサードパーティ・ライブラリをインストールする：

pacman -S gmp-devel mingw-w64-i686-boost mingw-w64-x86_64-boost □ mingw-w64-x86_64-gtest mingw-w64-i686-gtest mingw-w64-x86_64-perl

6. libopenabe ディレクトリで以下を実行する：

```
../env
make
make test
```

7. すべてのユニットテストがパスしたら、標準的な場所にライブラリをインストールする。

1.3.5 アンドロイド

Android 用の OpenABE をビルドするには、Android NDK をダウンロードしてインストールする必要があります。NDK は、ARM 用の C と C++、Android 固有のライブラリ、標準ライブラリ（GNU STL など）の実装をクロスコンパイルできるツールセットです。私たちは Android NDK r10e を使用し、Debian 7 上でビルドしています。Android NDK r10e は以下のリンクからダウンロードしてください：

1. Windows-x86_64 用
2. ダーウィン/Mac OS X-x86_64 用
3. Linux-x86_64 用

NDK を任意のディレクトリに解凍します。ここでは/opt/android-ndk-r10e/に解凍し、以後これを($ANDROID_NDK_ROOT)と呼ぶことにする。

すべてのライブラリは OpenABE deps ディレクトリの外でビルドします。スタンドアロンツールチェインでビルドプロセスを効率化し、含むために、以下の変数をエクスポートします：

```
export TOOLCHAIN_ARCH=arm-linux-androideabi-4.8
export MIN_PLATFORM=android-14 export
INSTALLDIR=$HOME/android
```

これらの変数を設定すれば、スタンドアローンのツールチェインを作ることができる：

```
ANDROID_NDK_ROOT/build/tools/make-standalone-toolchain.sh
    --toolchain=$TOOLCHAIN_ARCH --llvm-version=3.6 ¥
    --platform=$MIN_PLATFORM --install-dir=$INSTALLDIR
```

ただし、ARM ベースのプロセッサ用の RELIC ライブラリでは、64 ビットはサポートされていない。

アンドロイド用にビルドするには、以下を実行する：

```
./platforms/android.sh $ANDROID_NDK_ROOT $INSTALLDIR
```

libopenabe ディレクトリで、以下を実行する：

```
../env $ANDROID_NDK_ROOT $INSTALLDIR
make src
```

1.4　クイックスタート

高レベルの OpenABE crypto box API を使用するサンプル C++アプリをコンパイルするには、以 下の手順を実行する：

../env make
examples cd
examples/

次に、サポートされている暗号化モードごとにテストアプリを実行する：

./test_kp
./test_cp
./test_pk

また、KPABE 復号化でキーストアを使用する例を実行することもできる：

./test_km

test_km を実行すると、成功すれば次のような結果が得られる：

キー・マネージャーによる KP-ABE コンテキストのテスト
key1 の生成：（attr1 または attr2）と attr3
key2 の生成： attr1 と attr2
key3 の生成：attr2 と attr3
見つかったキー：'key2' => 'attr1 and attr2' 回収
されたメッセージ 1：hello world!発見されたキー：

'key4' => 'attr3 and attr4' 復元されたメッセージ

2：another hello!

メッセージ 3 の復旧に失敗!

2 OpenABE Crypto Box API

OpenABE Crypto Box インターフェイスは、属性ベースの暗号化（または ABE）のさまざまなフレーバー用の高レベル API を提供します。API を使用するには、以下の 6 つのステップを実行します：

1. OpenABE ライブラリを初期化する
2. 以下の 3 種類の ABE アルゴリズムから選んでコンテキストを構築する。
3. ABE の設定パラメータ（マスター公開パラメータ）の取得
4. 与えられた平文データと ABE 用のマスター公開パラメータを暗号化する。
5. 暗号文と秘密鍵（各ユーザの鍵オーソリティが生成）が与えられた場合の復号化
6. OpenABE ライブラリの分解

2.1 OpenABE の初期化

C++アプリケーションに OpenABE を統合するには、1 つのヘッダーファイルをインクルードし、oabe 名前空間を使用する必要があります：

```
#インクルード <openabe/openabe.h>

名前空間 oabe を使用しています;
```

Open-Key 暗号化方式が提供する共通鍵暗号機能が必要な場合は、Open-Key 暗号化方式を使用してください。
ABE を使用する場合は、以下のようにヘッダーを含め、oabe::crypto 名前空間を使用することもできます：

#インクルード <openabe/zsymcrypto.h>

名前空間 oabe::crypto;

2.1.1 シングルスレッド・アプリケーション

OpenABE ライブラリを初期化するには、シングルスレッド・アプリケーションの先頭で OpenABE init 関数を次のように呼び出します:

2.2暗号ボックス・コンテキストの構築

InitializeOpenABE();

このメソッドは、OpenSSL および RELIC ライブラリに必要な数学ライブラリやその他の機能を初期化することに注意してください。アプリケーションによっては、OpenABE の外部で openSSL の初期化を制御する必要があるかもしれません。そのような場合のために、openSSL の初期化を省略する別のルーチンを用意しています:
InitializeOpenABEwithoutOpenSSL()。

上記の初期化ルーチンが省略された場合、OpenABE コンテキストが呼び出されると std::runtime_error がスローされます。

OpenABE 操作の実行後にクリーンアップするには、アプリケーションを終了する前に shutdown メソッドを呼び出します:

ShutdownOpenABE();

このシャットダウン・ルーチンを省略した場合、RELIC ライブラリと OpenSSL の内部グローバル・ステートは、アプリケーションのメモリー・リークを引き起こす。

2.1.2 マルチスレッド・アプリケーション

マルチスレッド・アプリケーション用に、スレッドごとの OpenABE 初期化クラス OpenABEStateContext を用意しています。この OpenABEStateContext は、各スレッドの先頭で以下のように宣言します：

```
OpenABEStateContext oabe;
```

つまり、OpenABE の初期化はコンストラクタで行われ、シャットダウンとクリーンアップは(OpenABEStateContext)クラスのデストラクタで行われます。

2.2 暗号ボックス・コンテキストの構築

このセクションでは、ABE スキームコンテキストの構築方法について説明します。このインターフェイス内の各 ABE スキームは、Chosen-Ciphertext Attack（または CCA）セキュアスキームコンテキスト（OpenABE コンテキストの設計セクションで詳しく説明）を使用し、次のいずれかのタイプの ABE スキーム識別子でインスタンス化できます：
「CP-ABE」は暗号文ポリシーの ABE、「KP-ABE」は鍵ポリシーの ABE を意味する。

2.3 OpenABE 属性、属性リスト、およびポリシー

```
// クリプトボックス KP-ABE コンテキストをインスタンス化する
OpenABECryptoContext kpabe("KP-ABE");
```

(OpenABECryptoContext)は、生成されるパラメータ、鍵、暗号文を base64 エンコードするために、オプションで二番目の boolean 引数を受け付ける。デフォルトでは、base64 エンコーディングが有効になっている。このエンコードを無効にするには、以下のように crypto box をインスタンス化します：

```
// KP-ABE コンテキストをインスタンス化し、base64 エンコーディングを無効にする
OpenABECryptoContext kpabe("KP-ABE", false);
```

2.3 OpenABE 属性、属性リスト、およびポリシー

ABE の暗号化入力について、OpenABE はポリシーツリーと属性リストの 2 種類の入力をサポートするパーサーを実装している。属性は印刷可能なアスキー文字列であれば何でもよい。ポリシー・ツリーは属性で構成される論理式であり、属性リストは基本的に属性の配列です。

暗号文ポリシーABE（またはロールベースのアクセス制御）では、ユーザーはポリシーツリーを使って暗号化し、ユーザーの秘密鍵への入力は属性リストとなる。鍵の属性リストが暗号文のポリシーツリーを満たせば、ユーザーは復号できる。多権限 ABE では、異なる機関が発行した属性にまたがって書かれたポリシーに暗号文を関連付けることができる。同様に、ユーザーの秘密鍵に関連付けられた属性は複数の権限から構築される。 キーポリシーABE（またはコンテンツベースのアクセス制御）では、ユーザーは属性リストを使って暗号化し、ポリシーツリーが各ユーザーの秘密鍵に関連付けられる。鍵のポリシーツリーが暗号文の属性リストを満たせば、ユーザーは復号できる。

次のセクションでは、ポリシー・ツリーと属性リストを指定するための構文について説明する。

2.3.1 ポリシー・ツリー

ポリシー・ツリーは OR ゲートと AND ゲートで構成されるブーリアン式で、リーフ・ノードは属性である。さらに、< 、> 、<= 、>= 、DATE、RANGE のために、パーサーが内部的に数値属性に変換するいくつかの特殊な演算子を提供します。

2.4 パラメータの生成

適切な OR ゲートと AND ゲートを使用します。たとえば、「((マネージャーおよび経験 > 3) または管理者)」のようなポリシーを指定できます。

整数範囲: ここで[int]は最大 32 ビットの整数値である。パーサは以下のように変換します。

'<'と('>')を内部的に使用する。例えば、("Floor in (2, 5)")は、"(Floor > 2) and (Floor < 5)"と等価です。また、整数[int]を表すビット数を#で指定することもできます（例：[int]#4）。指定できるのは 2 の累乗のみであることに注意してください。つまり、4、8、16、32 です。

日付範囲: 日付の種類は、[接頭辞] = [月] [日] [年]、[接頭辞] = [月] [日]-[日] [年]で指定する。例：「日付=2015 年 3 月 1 日」。また、以下のように日付型で範囲を指定することもできます："(From:Alice and (Date = March 1-14, 2015 or Date = February 22-28, 2015))"。または、"To:Bob and (Date = January 5, 2016)"。

日付の比較: 場合によっては、< 、> 、<= 、>= を使って日付を比較すると便利です。このように、次のようなポリシーを指定することもできる：「(日付 > 2015 年 3 月 1 日)」や「(日付 <= 2017 年 1 月 1 日)」のように指定することもできる。日付は unix のエポック時間の始まりからの日数に変換されることに注意してください（つまり、1970 年 1 月 1 日以降）。

2.3.2 属性リスト

属性は'|'区切り文字で区切られ、属性リストとなる。例えば、"|Manager|IT|Experience=5|Date = December 20, 2015|"のように。

2.4 パラメータの生成

システムのセットアップ段階（通常は管理者が行う）で新しい ABE パラメータセットを生成するには、次のようにするだけでよい（ここでは KP-ABE システムを実行例として使用する）：

```
// 新しいマスター公開パラメータとマスターシークレットを生成する kpabe.generateParams();
```

マスター公開パラメータとマスターシークレットは生成され、インメモリ鍵ストアに保存される。生成されたパラメータをエクスポートするには、以下の手順に従う：

2.5キージェネレーション

```
// マスターのパブリックパラメータをエクスポート
std::string mpk; kpabe.exportPublicParams(mpk);

// マスターシークレットをエクスポート std::string msk;
kpabe.exportSecretParams(msk);
```

デフォルトでは、*mpk* と *msk* は base64 エンコードされた blob です（OpenABECryptoContext コンストラクタでバイナリ blob が指定されている場合を除く）。*mpk* はファイルシステムやデータベースに保存/キャッシュすることができる。しかし、*msk* はどこに保存するにしても、秘密にして保護しなければならない。最低限、安全なパスフレーズの下で *msk* を暗号化するか、ハードウェア・セキュリティ・モジュール（HSM）内に保存することを推奨する。既存の ABE パラメータのセットをコンテキストにロードし直す：

```
// マスターのパブリックパラメータをロードする
kpabe.importPublicParams(mpk);

// マスターシークレットパラメータをロード kpabe.importSecretParams(msk);
```

暗号化および復号化処理を実行するには、ABE *マスター公開パラメータ*を生成またはロードする必要があります。このステップに失敗したり省略したりすると、次の例外がスローされます: oabe::ZCryptoBoxException：グローバルパラメータが無効です。

鍵ジェネレーターを実行するエンティティは、ユーザーの秘密鍵を生成する前に、 マスターシークレットもロードする必要がある。

2.5 キー・ジェネレーション

管理者は、以下の手順でユーザーの秘密鍵を生成する：

```
// 以下のポリシーで新しいキーを生成する kpabe.keygen("attr1 and attr2", "key0");
```

2.6 単一メッセージの暗号化

ユーザーの KP-ABE 秘密鍵はポリシー・ツリーに関連付けられているが、CP-ABE 鍵は属性リストに関連付けられていることを思い出してほしい。鍵が生成されると、それはメモリ内の鍵ストアに格納され、指定された鍵識別子が 与えられれば、base64 エンコードされた文字列（またはバイナリ blob）にエクスポートできる：

```
std::string key0Blob; kpabe.exportUserKey("key0",
key0Blob);
```

エクスポートされた文字列は、アプリケーションに応じてファイルシステムまたは データベースに保存することができるが、ディスク上では秘密にして**保護する**必要がある。例えば、鍵ブロブを安全なパスフレーズで暗号化することで、さらに保護することができます。キー文字列をコンテキストに読み込むには： kpabe.importUserKey("key0", key0Blob);

多くの異なる鍵をコンテキストに格納／ロードできることに注意。しかし、暗号文を復号するときにどの鍵を使うかは、ユーザーが決めなければならない。

2.6 単一メッセージの暗号化

暗号化と復号化には、**マスター公開**パラメータをメモリにロードする必要がある。以下に、キーポリシーABE における crypto box API の使用例を示す：

```
// マスターのパブリックパラメータをロードする
kpabe.importPublicParams(mpk);

// 以下のポリシーでメッセージを暗号化する // 1 ステップで ABE と
AES-GCM 暗号化 std::string ct, pt1 = "message", pt2;
kpabe.encrypt("attr1|attr2", pt1, ct);
```

暗号化に成功すると、暗号文は base64 エンコードされた文字列（コンストラクタで base64 エンコードが無効になっている場合はバイナリ blob）として ct に格納される。以下の点に注意。

2.7 キー・マネージャーによる復号化

OpenABECryptoContext の encrypt は、生成された ABE 暗号文と AES-GCM 暗号化を ct 文字列で結合するように設計されています。

復号化のために、ユーザーは 2 つのものを提供する。ユーザーの秘密鍵識別子（あらかじめコンテキストにロードされていなければならない）と暗号文の文字列 ct である。

```
// 以下のキーで 1 つのメッセージを復元する bool result =
kpabe.decrypt("key0", ct, pt2);
```

復号化ルーチンは、成功したか否かを示すブール値を返す。結果が真の場合、保護されたデータは pt2 に格納される。

```
// 復号化に成功し、平文が復元されたかどうかをチェックする assert(result && pt1 == pt2);
```

2.7 キー・マネージャーによる復号化

復号化を支援するために、キー・マネージャーを組み込み、与えられた暗号文を復号化するためのキーを手動で指定する必要をなくしました。この機能を利用するには、秘密鍵を読み込む**前に** enableKeyManager() API を呼び出す必要があります。

```
// 指定されたユーザ識別子でキー・マネージャの使用を有効にする
kpabe.enableKeyManager("user1");
```

importUserKey()API を使用して、任意の数の秘密鍵をインポートすると、これらの鍵はメモリ上に保存され、鍵マネージャによって検索可能になる。読み込まれたすべてのユーザー秘密鍵は、指定されたユーザー識別子（例：user1）の下に保存されることに注意してください。

```
// 無制限の数のキーをキー・マネージャにインポートする
kpabe.importUserKey(keyId1, keyBlob1);
kpabe.importUserKey(keyId2, keyBlob2); ...
```

秘密鍵がロードされると、暗号文は鍵識別子なしで以下のように復号化できる：

2.8 公開鍵暗号化と公開鍵署名

```
// キー・マネージャーを使用して単一メッセージを復元 bool result =
kpabe.decrypt(ct, pt2);
```

KP-ABE を使った動作例については(examples/test_km.cpp)を参照のこと。

2.8 公開鍵暗号化と公開鍵署名

また、公開鍵暗号化や署名など、他の暗号プリミティブ用の crypto box インターフェースもサポートしている。以下に、各コンテキストタイプの完全なコード例を示す：

```
// PKE コンテキスト PKEContext pke を
作成する；

// user0 の PK/SK を生成 pke.keygen("user0")；

// user0 へのメッセージを暗号化する std::string pt =
"message", ct; pke.encrypt("user0", pt1, ct)；

// user0 宛のメッセージを復号化 bool result =
pke.decrypt("user0", ct, pt2)；

// 復号が成功し、平文が復元されたことを確認する assert(result && pt1 == pt2)；
```

公開鍵署名（EC-DSA アルゴリズムのラッパー）、

```
// PKSIG コンテキストを作成する
PKSIGContext pksig；

// user1 用の PK/SK を生成 pksig.keygen("user1")；

// 送信者を user1 としたメッセージに署名する。
```

```
std::string msg = "hello world!"; std::string
sig; pksig.sign("user1", msg, sig);

// メッセージが user1 からのものであることを誰でも確認できる bool
result = pksig.verify("user1", msg, sig);

// 検証結果を確認する assert(result);
```

3 OpenABE コンテキストデザイン

OpenABE は、暗号アルゴリズムの種類ごとに、そのアルゴリズムをアプリケーションで使用するためのカプセル化されたコンテキストを提供する。各コンテキストには、ローカルのキーストアへの参照、楕円曲線（またはペアリング・ベース）演算用の数学ライブラリ、乱数のソースが含まれる。

3.1 属性ベースの暗号化（ABE）：

OpenABE は、Chosen-Plaintext Attack（または CPA）セキュリティと Chosen-Ciphertext Attack（または CCA）セキュリティを備えた ABE のための 4 つのコンテキストを提供します。以下に各コンテキストと、それらが提供する特定のセキュリティ特性について説明する。また
各コンテキストを実装している OpenABE の関連ソースファイルを参照します。

1. **CPA KEM スキームコンテキスト**：鍵ポリシー、暗号文ポリシーの ABE など、さまざまな ABE を実装。セットアップ、鍵生成、暗号化、復号化のための汎用インターフェースを提供。

CPA セキュリティは、OpenABE に実装されている ABE アルゴリズムが満たす基本的なセキュリティ定義であることに注意。鍵カプセル化メカニズム（KEM）とは、単に AES 鍵を ABE で暗号化することを意味する。加えて、暗号化は ABE スキームタイプに対応する汎用的な機能入力を入力とする。

ソースファイル src/abe/zcontextcpwaters.cpp および src/abe/zcontextkpgpsw.cpp を参照してください。

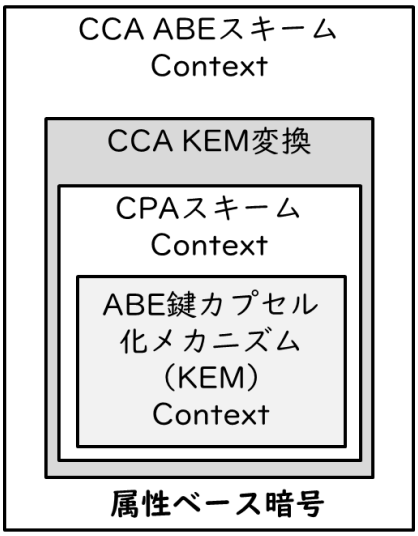

図 2：OpenABE ABE コンテキスト

2. **CPA スキームコンテキスト**：CPA KEM スキームコンテキストを標準的な暗号化アルゴリズムに変換するラッパー。この汎用コンテキストは、任意の CPA KEM コンテキストと擬似乱数生成器（PRG）を組み合わせて平文を暗号化する。

   ソースファイル src/abe/zcontextabe.cpp を参照。

3. **CCA KEM 変換コンテキスト**：任意の CPA スキーム・コンテキストを使用して、キーのカプセル化を伴う CCA セキュア ABE スキームを構築する汎用 CCA 変換コンテキスト。この汎用変換は、OpenABE に実装されている ABE スキームに対応します。

   ソースファイル src/abe/zcontextcca.cpp を参照。

4. **CCA スキームコンテキスト**：平文メッセージの暗号化に適した、選択された安全な ABE スキームに変換する CCA KEM 変換コンテキストのラッパー。特に、CCA KEM に由来するキーを使用して、Authenticated Encryption（AES-GCM）を使用して暗号化します。

ソースファイル src/abe/zcontextcca.cpp および src/zsymcrypto.cpp を参照してください。

ZCrypto_box API の OpenABECryptoContext は、**CCA Scheme Context を**中心に実装されているため、デフォルトで ABE のベストプラクティスレベルのセキュリティを提供することに注意してください。

3.2 *公開鍵暗号化（PKE）：*

図 3: OpenABE PKE コンテキスト

3.2 公開鍵暗号化（PKE）：

OpenABE は、CPA および CCA セキュリティを持つ PKE に対して同様のコンテキストを提供する。以下の各コンテキストの説明では、関連するソースファイルを参照しています。

1. **CPA KEM スキームコンテキスト**：これは、鍵のカプセル化を伴う公開鍵暗号化スキーム用の CPA KEM コンテキストを提供する。

   ソースファイル src/pke/zcontextpke.cpp を参照。

2. **CCA スキームコンテキスト**：これは、上記の CPA KEM コンテキストをラップし、Authenticated Encryption と組み合わせた CCA スキームコンテキストを提供する。

ソースファイル src/pke/zcontextpke.cpp および src/zsymcrypto.cpp を参照してください。

4 低レベル OpenABE API

OpenABE にはいくつかのコンテキストがありますが、どうしても必要な場合を除き、低レベルの OpenABE API を直接使用することはお勧めしません。ほとんどの場合、OpenABE crypto box API で十分であり、一般的な暗号実装のエラーを回避できるように特別に設計されています。開発者は、アプリケーションに選択暗号文攻撃セキュアスキームコンテキストを使用することを推奨します。

数値識別子を指定してコンテキストを作成するヘルパーメソッドをいくつか提供する。以下のコード・スニペットは、OpenABE が ABE および公開鍵暗号化でサポートするさまざまな暗号化コンテキストの作成方法を示しています。

4.1　属性ベースの暗号化（ABE）

高レベル API と同様に、ABE コンテキスト関数を呼び出す前に、OpenABE を InitializeOpenABE()で初期化する必要がある。CCA セキュア暗号化コンテキストの場合、API は以下のようになる。

```
STD::UNIQUE_PTR<OpenABECONTEXTSchemeCCA> ¥
    OpenABE_createContextABESchemeCCA(OpenABE_SCHEME s);
```

ここで OpenABE_SCHEME は、現在実装されている 3 つのオプションのうちの 1 つである：

- 暗号文ポリシー ABE 用 OpenABE_SCHEME_CP_WATERS
- 鍵ポリシー ABE 用 OpenABE_SCHEME_KP_GPSW

同様に、CPA セキュアな暗号化コンテキストのために OpenABE_createContextABESchemeCPA() メソッドが存在します（ただし、CCA セキュリティを推奨します）。

4.1.1　スキーム・コンテキストの構築

例えば、CCA セキュアな KP-ABE スキームのコンテキストを次のように構築する：

```
std::unique_ptr<OpenABEContextSchemeCCA> ccaCpAbeContext = nullptr;
ccaKpAbeContext =
OpenABE_createContextABESchemeCCA(OpenABE_SCHEME_KP_GPSW);
```

この例の KP-ABE スキーム CCA コンテキストが構築されると、ハンドルを使用してシステムパラメータを生成し、ユーザー秘密鍵を導出することができる：

```
// セットアップパラメータを生成する
// BN_P254 "はスキームのペアリング曲線である、
// ccaKpAbeContext->generateParams("BN_P254", "mpk", "msk");

// 属性リスト "one|two|three "のキー元 std::unique_ptr<OpenABEFunctionInput> keyInput
= ¥createPolicyTree("((one and two) or three)");
ccaKpAbeContext->keygen(keyInput.get(), "deckey", "mpk", "msk");
```

4.1　属性ベースの暗号化（ABE）

4.1.2　暗号化と復号化

低レベルの OpenABE API は、デタッチド・モードでの暗号化のみをサポートしています。分離モードとは、ABE 暗号文と AES-GCM 暗号文が**別々の**文字列バッファに格納されることを意味します。生成された暗号文のエクスポートと保存は、各自の責任で行ってください。以下の例は、KP-ABE CCA コンテキストの暗号化ルーチンと復号化ルーチンを示しています：

```
// 暗号化
OpenABECiphertext ct1, ct2; std::string pt1 = "some sample
plaintext"; std::unique_ptr<OpenABEFunctionInput> encInput =
¥ createAttributeList("one|two|four|five");
ccaKpAbeContext->encrypt("mpk", encInput.get(), pt1, &ct1, &ct2);

// decrypt std::string pt2; ccaKpAbeContext->decrypt("mpk",
"deckey", pt2, &ct1, &ct2);

// 成功するはず assert(pt1 ==
pt2);
```

適切な入力は、ABE の他のフレーバーに対して供給される。例えば、encrypt ルーチンは CP-ABE スキームに対して（属性リストの代わりに）ポリシーを期待する。

4.1.3　暗号文のインポートとエクスポート

暗号文の内容を文字列にエクスポートするには、OpenABECiphertext オブジェクトの exportToBytes メソッドを呼び出します：

```
OpenABEByteString ct1Buf;
// ABE 暗号文のヘッダーとボディをエクスポート ct1.exportToBytes(ct1Buf);
```

バイナリ文字列から暗号文をインポートするには、OpenABECiphertext オブジェクトの loadFromBytes メソッドを呼び出します：

4.2　公開鍵暗号化 (PKE)

```
OpenABECiphertext ct2;
// ABE 暗号文のヘッダーとボディをロードする ct2.loadFromBytes(ct1Buf);
```

4.1.4　ABE キーのインポートとエクスポート

CCA スキーム・コンテキストによって以前にロードまたは生成された ABE 秘密鍵の内容をエクスポートするには、コンテキストの exportKey ルーチンを呼び出します：

```
OpenABEByteString skBlob;
// ccaKpAbeContext->exportKey("deckey", skBlob);
```

ABE 秘密鍵のコンテンツをインポートするには、CCA スキーム・コンテキストの loadUserSecretParams を呼び出します：

```
// skBlob からユーザ秘密鍵をロードし、"decKey "として格納 ccaKpAbeContext-
>loadUserSecretParams("deckey", skBlob);
```

メモリ内の ABE 秘密鍵が不要になったら、CCA スキーム・コンテキストの deleteKey ルーチンを呼び出して、機密鍵を消去する：

```
// ID "deckey "のキーを削除する。
ccaKpAbeContext->deleteKey("deckey");
```

4.2　公開鍵暗号化 (PKE)

CCA セキュアな encrypton コンテキストについては、以下の API を呼び出す：

```
std::unique_ptr<OpenABEContextSchemePKE> ¥
        OpenABE_createContextPKESchemeCCA(OpenABE_SCHEME s);
```

OpenABE_SCHEME は OpenABE_SCHEME_PK_OPDH です。

4.2　公開鍵暗号化 (PKE)

4.2.1　スキーム・コンテキストの構築

```
// PKE ECC MQV スキーム用の新しい KEM コンテキストを作成する
std::unique_ptr<OpenABEContextSchemePKE> pkSchemeContext = ¥
    OpenABE_createContextPKESchemeCCA(OpenABE_SCHEME_PK_OPDH);
```

4.2.2　鍵の生成

```
// アリスの静的公開鍵と秘密鍵を計算する pkSchemeContext->keygen("ID_A", "public_A",
"private_A");

// ボブの静的公開鍵と秘密鍵を計算する pkSchemeContext->keygen("ID_B", "public_B",
"private_B");
```

4.2.3　暗号化と復号化

```
std::string pt1, pt2; OpenABECiphertext ct;

// 最初の引数は NULL (ユーザが RNG をカスタマイズしたい場合を除く)
// 2 番目の引数は受信者の公開鍵 ID // 3 番目の引数は送信者の公開鍵 ID
pkSchemeContext->encrypt(NULL, "public_B", "public_A", pt1, &ct);

// 最初の引数は送信者の公開鍵
// 2 番目の引数は受信者の秘密鍵 bool result = pkSchemeContext->decrypt("public_A",
"private_B", pt2, &ct);

// 復号化が成功し、平文が復元されたことを確認する assert(result && pt1 == pt2);
```