Zeutro LLC

Encryption & Data Security

The OpenABE Library

C++ API Guide

OpenABE C++ API

Version 1.0

Contents

1 The OpenABE Library

1.1 What is OpenABE?

OpenABE is a C/C++ library that implements several attribute-based encryption (ABE) schemes that can be used across a variety of data-at-rest applications. The goal of this toolkit is to develop an efficient implementation that embeds security guarantees at the architectural level. To support modern applications, OpenABE provides a simplistic API (or `crypto_box` like interface) for encryption and signatures. OpenABE also has the following qualities:

- **Modular**: the OpenABE allows developers to swap one cryptographic scheme for another without updating application logic.
- **Comprehensive**: the OpenABE supports routines necessary to perform common cryptographic tasks.
- **Extensible**: the OpenABE can support several additional functional encryption scheme types with relatively little effort.
- **Best Practices**: the OpenABE incorporates best practices in encryption design which includes security against chosen ciphertext attacks, a simple interface for transporting symmetric keys, and performing encryption of large data objects.

1.2 Core Cryptographic Algorithms

See the OpenABE architecture diagram shown in Figure 1:

The OpenABE provides a number of core cryptographic algorithms:

- Supports multiple types of attribute-based encryption (ABE) key encapsulation (KEM) schemes including key-policy ABE and ciphertext-policy ABE.
- Supports chosen-ciphertext security (CCA) for each ABE KEM scheme type. This also includes public-key encryption with chosen-ciphertext security.
- Supports digital signatures as well as authenticated symmetric-key encryption.
- Supports several common cryptographic functions including linear secret sharing schemes (LSSS), key derivation functions (KDFs), pseud-random functions (PRFs), and pseudo-random generators (PRGs).

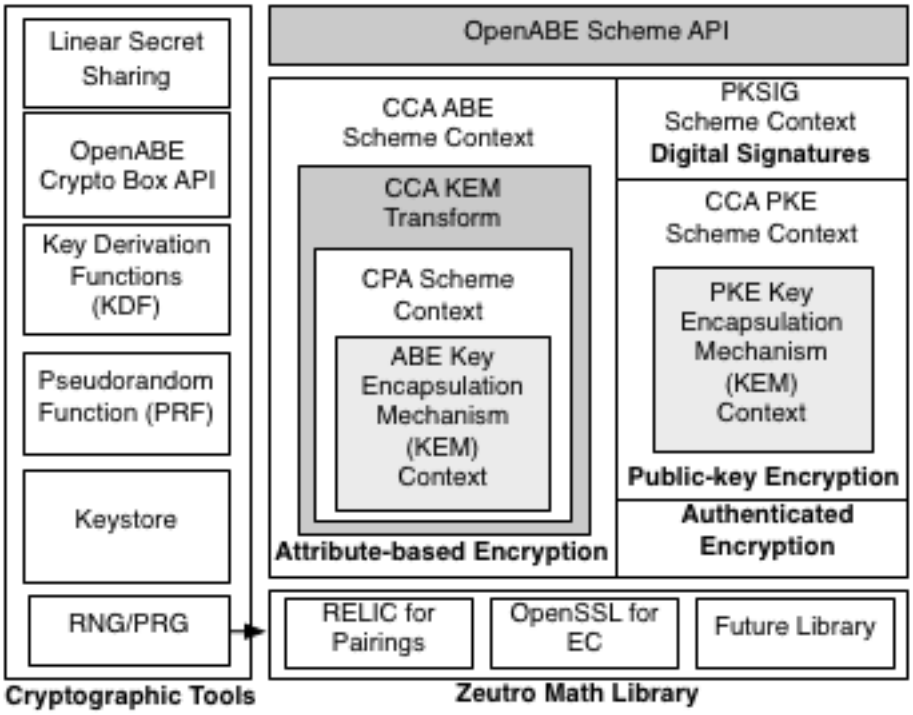

Figure 1: OpenABE Architecture Diagram

OpenABE supports other useful features such as:

- Extensible Zeutro math library (ZML) for elliptic curve and pairing operations. Can be instantiated with RELIC only at this time. ZML is designed to support other math libraries provided that they conform to our C abstract math interface.
- Replacable random number generator (RNG). Can swap a RNG with a pseudorandom one (or PRNG) in the ZML which is required for the OpenABE ABE CCA transformation.
- Keystore implementation for managing ABE and PKE/PKSIG keys in memory.

1.3 Installation

This section describes the installation of the OpenABE source code (`libopenabe-1.0.0-src.tar.gz`) on various platforms. The OpenABE currently supports several operating systems including multiple versions/distros of Linux, Mac OS X and Windows.

1.3.1 Debian/Ubuntu-based Linux

To compile OpenABE on Ubuntu or Debian Linux-based distro, first run the `deps/install_pkgs.sh` script from the source directory to install the OpenABE system-specific dependencies as follows:

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

Note that you only have to do this once per system setup. After completion, you can proceed to compile the OpenABE as follows:

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

All the unit tests should pass at this point and you can proceed to install the OpenABE in a standard location (`/usr/local`) as follows:

```
sudo make install
```

To change the installation path prefix, modify the `INSTALL_PREFIX` variable in `libopenabe-1.0.0/Makefile`.

1.3.2 CentOS and RedHat Linux

As before, first run the script from the source directory to setup OpenABE dependencies:

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

Note that you only have to do this once per system setup. After completion, you can proceed to compile the OpenABE as follows:

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

All the unit tests should pass at this point and you can proceed to install the OpenABE in a standard location (`/usr/local`) as follows:

```
sudo make install
```

To change the installation path prefix, modify the `INSTALL_PREFIX` variable in `libopenabe-1.0.0/Makefile`.

1.3.3 Mac OS X

Note that for Mac OS X, you will need homebrew installed prior to running the `deps/install_pkgs.sh` script. Then, do the following (you may require `sudo` on the second step):

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

Note that you only have to do this once per system setup. After completion, you can proceed to compile the OpenABE as follows:

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

All the unit tests should pass at this point and you can proceed to install the OpenABE in a standard location (`/usr/local`) as follows:

```
sudo make install
```

To change the installation path prefix, modify the `INSTALL_PREFIX` variable in `libopenabe-1.0.0/Makefile`.

1.3.4 Windows

To build OpenABE on Windows 7, 8, and 10, you will need to download and install Mingw-w64, the GNU toolchain port for building Windows-native binaries. We use the Mingw-w64 port packaged with Minimal SYStem 2 (MSYS2). MSYS2 is an emulated POSIX-compliant environment for building software with GNU tooling (e.g., GCC), bash, and package management using Arch Linux's Pacman. Binaries compiled with these compilers do not require `cygwin.dll` as they are standalone.

1. Download `msys2-x86_64-latest.exe` and run it. Select `C:\` for the installation directory to avoid `PATH` resolution problems.

2. Launch the MSYS2 shell and execute the following command:

```
update-core
```

3. Close the MSYS2 shell and launch the MinGW-w64 Win64 Shell. Note that after starting MSYS2, the prompt will indicate which version you have launched.

4. Update the pre-installed MSYS2 packages (and install related tooling), close the shell when prompted to, and relaunch the MinGW-w64 Win64 Shell. Execute the following command to start the process:

```
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. Install the required third-party libraries by executing the following command:

```
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. In the libopenabe directory, execute the following:

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

7. If all the unit tests pass, then proceed to install the library in a standard location:

   make install

1.3.5 Android

To build OpenABE for Android, you will need to download and install the Android NDK. The NDK is a toolset that enables cross-compiling C and C++ for ARM and Android-specific libraries and implementations of standard libraries (e.g., GNU STL). We use Android NDK r10e and build on Debian 7.

Download the Android NDK r10e at the following links:

1. For Windows-x86_64
2. For Darwin/Mac OS X-x86_64
3. For Linux-x86_64

Unzip the NDK to a directory of your choice. We unzip it to `/opt/android-ndk-r10e/` and will refer to this as `$ANDROID_NDK_ROOT` hereafter.

We build all libraries outside of the OpenABE deps directory. We export the following variables to streamline and contain the build process with a standalone toolchain:

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

With these variables set, you can now make the standalone toolchain:

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

Note that 32- and 64-bit architectures are supported for any platform API greater than android-14; However, 64-bit is not supported in the RELIC library for ARM-based processors.

To build for Android, run the following:

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

In the libopenabe directory, execute the following:

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

1.4 Quick Start

To compile example C++ apps that use the high-level OpenABE crypto box API, do the following:

```
. ./env
make examples
cd examples/
```

Then, execute the test apps for each mode of encryption supported:

```
./test_kp
./test_cp
./test_pk
```

You can also execute the example that demonstrates use of the keystore with KP-ABE decryption:

```
./test_km
```

Executing `test_km` will lead to the following results if successful:

```
Testing KP-ABE context with Key Manager
Generate key1: (attr1 or attr2) and attr3
Generate key2: attr1 and attr2
Generate key3: attr2 and attr3
Generate key4: attr3 and attr4
Found Key: 'key2' => 'attr1 and attr2'
Recovered message 1: hello world!
Found Key: 'key4' => 'attr3 and attr4'
Recovered message 2: another hello!
Correctly failed to recover message 3!
```

2 OpenABE Crypto Box API

The OpenABE Crypto Box interface provides a high-level API for different flavors of attribute-based encryption (or ABE). Using the API comprises the following six steps:

1. Initialize the OpenABE library
2. Construct a context by picking from three possible types of ABE algorithms
3. Retrieve the setup parameters for ABE (master public parameters)
4. Encrypt given plaintext data and master public parameters for the ABE
5. Decrypt given a ciphertext and a private key (generated by key authority for each user)
6. Teardown the OpenABE library

2.1 Initializing the OpenABE

To integrate the OpenABE into your C++ application, you will need to include a single header file and use the `oabe` namespace:

```
#include <openabe/openabe.h>

using namespace oabe;
```

If you need the symmetric key cryptographic functionality provided by the OpenABE, you may also include the header below and use the `oabe::crypto` namespace as follows:

```
#include <openabe/zsymcrypto.h>

using namespace oabe::crypto;
```

2.1.1 Single-threaded Application

To initialize the OpenABE library, call the OpenABE init function in the beginning of your single-threaded application as follows:

```
InitializeOpenABE();
```

Note that this method initializes the math library and other functionality required from the OpenSSL and RELIC library. For some applications, it might be necessary to control openSSL initialization outside of the OpenABE. We provide a separate routine that omits openSSL initialization for such cases: `InitializeOpenABEwithoutOpenSSL()`.

If either of the above initialization routines are ommitted, then a `std::runtime_error` is thrown when any OpenABE contexts are invoked.

To cleanup after performing OpenABE operations, simply call the shutdown method prior to exiting your application:

```
ShutdownOpenABE();
```

If this shutdown routine is ommitted, then the internal global state for the RELIC library and OpenSSL will result in a memory leak in your application.

2.1.2 Multi-threaded Applications

We provide a per-thread OpenABE initialization class `OpenABEStateContext` for multi-threaded applications. This `OpenABEStateContext` should be declared at the beginning of each thread as follows:

```
OpenABEStateContext oabe;
```

Note that this follows **resource acquisition is initialization (or RAII)** which means that OpenABE initialization occurs in the constructor and shutdown/cleanup occurs in the destructor of the `OpenABEStateContext` class.

2.2 Construct Crypto Box Context

In this section, we describe how to construct an ABE scheme context. Each ABE scheme within this interface uses the Chosen-Ciphertext Attack (or CCA) secure scheme context (described in more detail in the OpenABE context design section) and can be instantiated with one of the following types of ABE scheme identifiers: "CP-ABE" stands for ciphertext-policy ABE and "KP-ABE" for key-policy ABE.

```
// instantiate a crypto box KP-ABE context
OpenABECryptoContext kpabe("KP-ABE");
```

The `OpenABECryptoContext` accepts an optional second boolean argument to base64 encode the generated parameters, keys and ciphertexts. By default, base64 encoding is enabled. To disable this encoding, simply instantiate the crypto box as follows:

```
// instantiate KP-ABE context and disable base64 encoding
OpenABECryptoContext kpabe("KP-ABE", false);
```

2.3 OpenABE Attributes, Attribute Lists and Policies

For ABE encryption inputs, the OpenABE implements a parser that supports two types of inputs: policy trees and attribute lists. Attributes can be any printable ascii strings. Policy trees are logical expressions comprised of attributes and attribute lists are essentially an array of attributes.

With ciphertext-policy ABE (or role-based access control), users encrypt using a policy tree and the input to the user's secret key is an attribute list. If the list of attributes on the key satisfy the policy tree on the ciphertext, then the user will be able to decrypt. Multi-authority ABE allows one to associate a ciphertext with a policy written across attributes issued by different authorities. Similarly, the attributes associated with a user's secret key are built from multiple authorities

With key-policy ABE (or content-based access control), users encrypt using an attribute list and a policy tree is associated with each user's secret key. If the policy tree on the key satisfies the list of attributes on the ciphertext, then the user will be able to decrypt.

In the next section, we will describe the syntax for specifying policy trees and attribute lists.

2.3.1 Policy Trees

Policy trees are boolean formulas comprised of `OR` and `AND` gates where leaf nodes are attributes. In addition, we provide some specialized operators for <,>,<=,>=, `DATE` and `RANGE` that the parser internally transforms into numerical attributes

using the appropriate `OR` and `AND` gates. For example, you can specify policies such as `"((Manager and Experience > 3) or Admin)"`.

Integer Ranges: The range type is specified as `[Attribute] in ([int], [int])` where `[int]` can be up to a 32-bit integer value. The parser does the translation to `'<'` and `'>'` internally. For example, `"Floor in (2, 5)"` is equivalent to `"(Floor > 2) and (Floor < 5)"`. You can also specify the number of bits to represent the integer `[int]` using a `#` (e.g., `[int]#4`). Note that only powers of two are allowed. That is, 4, 8, 16, and 32.

Date Ranges: The date type is specified as `[Prefix] = [Month] [Day], [Year]` and `[Prefix] = [Month] [Day]-[Day], [Year]`. E.g., `"Date = March 1, 2015"`. You can also specify a range with the date type as follows: `"(From:Alice and (Date = March 1-14, 2015 or Date = February 22-28, 2015))"`. Or, `"To:Bob and (Date = January 5, 2016)"`.

Date Comparison: In certain cases, it maybe useful to compare dates using <,>,<=, and >=. As such, you can also specify policies like: `"(Date > March 1, 2015)"` or `"(Date <= January 1, 2017)"`. Note that the date is translated into days since the beginning of the unix epoch time (i.e., since Jan 1, 1970).

2.3.2 Attribute Lists

Attributes are separated by the `'|'` delimiter to form an attribute list. For example, `"|Manager|IT|Experience=5|Date = December 20, 2015|"`.

2.4 Generate Parameters

To generate a fresh set of ABE parameters during the system setup phase (typically done by an administrator), simply do the following (using a KP-ABE system as the running example here):

```
// generate a fresh master public and master secret parameters
kpabe.generateParams();
```

The master public parameters and master secret parameters are generated and stored in an in-memory keystore. To export the parameters after they have been generated:

```
// export master public params
std::string mpk;
kpabe.exportPublicParams(mpk);

// export master secret params
std::string msk;
kpabe.exportSecretParams(msk);
```

By default, the *mpk* and *msk* are base64 encoded blobs (unless binary blob is specified in the `OpenABECryptoContext` constructor). The *mpk* can be stored/cached on the file system or in a database. However, the *msk* must be kept secret and protected wherever it is stored. At a minimum, we recommend encrypting the *msk* under a secure passphrase or stored inside a hardware security module (HSM).

To load an existing set of ABE parameters back into a context:

```
// load master public params
kpabe.importPublicParams(mpk);

// load master secret params
kpabe.importSecretParams(msk);
```

Generating or loading ABE *master public parameters* is required to perform encrypt and decrypt operations. If this step fails or is ommitted, then the following exception will be thrown: `oabe::ZCryptoBoxException: Invalid global parameters`.

For the entity that runs the key generator, it is necessary to also load the *master secret parameters* prior to generating user's private keys.

2.5 Key Generation

An administrator generates user's private keys by doing the following:

```
// generate a new key with the following policy
kpabe.keygen("attr1 and attr2", "key0");
```

Recall that a user's KP-ABE private key has a policy tree associated with it while a CP-ABE key is associated with an attribute list. Once the key is generated, it is stored in an in-memory keystore that can be exported to a base64 encoded string (or binary blob) given the key identifier specified:

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

The exported string can be stored on the file system or a database depending on the application, but must be kept **secret** and **protected** on disk. For example, you could further protect the key blob by encrypting it with a secure passphrase.

To load a key string into the context:

```
kpabe.importUserKey("key0", key0Blob);
```

Note that many different keys can be stored/loaded into a context. However, the user must decide which key to use when decrypting a ciphertext.

2.6 Encrypting a Single Message

For encryption and decryption, it is required that the **master public parameters** are loaded into memory. An example use of the crypto box API for key-policy ABE is shown below:

```
// load master public params
kpabe.importPublicParams(mpk);

// encrypt a message with the following policy
// ABE & AES-GCM encryption in one step
std::string ct, pt1 = "message", pt2;
kpabe.encrypt("attr1|attr2", pt1, ct);
```

If encryption is successful, then the ciphertext is stored in `ct` as a base64 encoded string (or binary blob if base64 encoding disabled in the constructor). Note that

encrypt in the `OpenABECryptoContext` is designed to combine the generated ABE ciphertext and AES-GCM encryption in the `ct` string.

For decryption, users supply two things: the user's private key identifier (must be loaded into the context beforehand) and the ciphertext string `ct`.

```
// recover a single message with the following key
bool result = kpabe.decrypt("key0", ct, pt2);
```

The decryption routine returns a boolean value to indicate whether it was successful or not. If the result is `true`, the protected data will be stored in `pt2`.

```
// check if decrypt was successful and plaintexts recovered
assert(result && pt1 == pt2);
```

2.7 Decrypting with a Key Manager

For assisted decryption, we have included a key manager to eliminate the need to manually specify a key to decrypt a given ciphertext. You are simply required to call the `enableKeyManager()` API **before** loading any secret keys to take advantage of this feature.

```
// enable the use of the key manager with a given user identifier
kpabe.enableKeyManager("user1");
```

Import any number of secret keys via `importUserKey()` API and these keys will get stored and be searchable in memory by the key manager. Note that all user secret keys that are loaded are stored under the specified user identifier (e.g., `user1`).

```
// import an unlimited number of keys into the key manager
kpabe.importUserKey(keyId1, keyBlob1);
kpabe.importUserKey(keyId2, keyBlob2);
...
```

Once the secret keys have been loaded, a ciphertext can be decrypted without a key identifier as follows:

```
// recover a single message using the key manager
bool result = kpabe.decrypt(ct, pt2);
```

See `examples/test_km.cpp` for a working example using KP-ABE.

2.8 Public-key Encryption & Public-key Signatures

We also support a crypto box interface for other cryptographic primitives including for public-key encryption and signatures. We show a full code example for each context type below:

```
// create PKE context
PKEContext pke;

// generate PK/SK for user0
pke.keygen("user0");

// encrypt a message to user0
std::string pt = "message", ct;
pke.encrypt("user0", pt1, ct);

// decrypt a message intended for user0
bool result = pke.decrypt("user0", ct, pt2);

// check that decryption is succesful and plaintext recovered
assert(result && pt1 == pt2);
```

For public-key signatures (wrapper around the EC-DSA algorithm),

```
// create PKSIG context
PKSIGContext pksig;

// generate PK/SK for user1
pksig.keygen("user1");

// sign a message with the sender being user1
```

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

// anyone can verify the message came from user1
bool result = pksig.verify("user1", msg, sig);

// check the verification result
assert(result);
```

3 OpenABE Context Design

For each type of cryptographic algorithm, the OpenABE provides an encapsulated context for using that algorithm in an application. Each context includes a reference to a local keystore, a math library for elliptic curve (or pairing-based) operations and a source for random numbers.

3.1 Attribute-based Encryption (ABE):

The OpenABE provides four contexts for ABE with Chosen-Plaintext Attack (or CPA) security and Chosen-Ciphertext Attack (or CCA) security. We describe each context below and the specific security properties that they provide. In addition, we refer to the relevant source files in the OpenABE that implements each context.

1. **CPA KEM Scheme Context**: Implements different flavors of ABE including key-policy and ciphertext-policy ABE. Provides a generic interface for setup, key generation, encryption, and decryption.

   Note that CPA security is the basic security definition satisfied by ABE algorithms implemented in the OpenABE. Key encapsulation mechanism (KEM) simply means that an AES key is encrypted with ABE. In addition, encryption takes as input a generic functional input for the ABE scheme type.

   See source files: `src/abe/zcontextcpwaters.cpp` and `src/abe/zcontextkpgpsw.cpp`

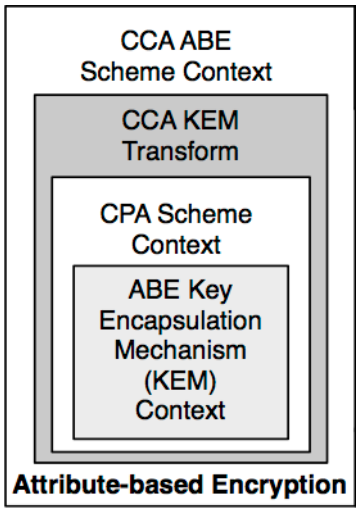

Figure 2: OpenABE ABE Contexts

2. **CPA Scheme Context**: A wrapper for the CPA KEM scheme context that transforms it into a standard encryption algorithm. This generic context combines any CPA KEM context with a pseudo-random number generator (PRG) to encrypt plaintext.

   See source file: `src/abe/zcontextabe.cpp`

3. **CCA KEM Transform Context**: A generic CCA transform context that takes any CPA scheme context to build a CCA secure ABE scheme with key encapsulation. This generic transform works for the ABE schemes implemented in the OpenABE.

   See source file: `src/abe/zcontextcca.cpp`

4. **CCA Scheme Context**: A wrapper for the CCA KEM transform context that transforms it into a chosen-secure ABE scheme suitable for encrypting plaintext messages. In particular, it uses the key derived from the CCA KEM to encrypt using Authenticated Encryption (i.e., AES-GCM).

   See source files: `src/abe/zcontextcca.cpp` and `src/zsymcrypto.cpp`

Note that the `OpenABECryptoContext` in the `ZCrypto_box` API is implemented around the **CCA Scheme Context** and thus provides the best practice level of security for ABE by default.

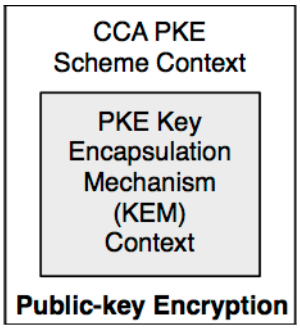

Figure 3: OpenABE PKE Contexts

3.2 Public-Key Encryption (PKE):

The OpenABE provides similar contexts for PKE with CPA and CCA security. The relevant source files are referenced in each context description below.

1. **CPA KEM Scheme Context**: This provides a CPA KEM context for public-key encryption schemes with key encapsulation.

   See source file: `src/pke/zcontextpke.cpp`

2. **CCA Scheme Context**: This provides a CCA scheme context which wraps the above CPA KEM context and combines with Authenticated Encryption.

   See source files: `src/pke/zcontextpke.cpp` and `src/zsymcrypto.cpp`

4 Low-level OpenABE API

Although there are several contexts to choose from in the OpenABE, we do not recommend using the low-level OpenABE API directly unless absolutely necessary. In most cases, the OpenABE crypto box API should be sufficient and it has been specially designed to help avoid common cryptography implementation errors. We recommend that developers use the chosen-ciphertext attack secure scheme contexts for applications; this is a best practice.

We provide some helper methods to create the contexts given a numeric identifier. The following code snippets demonstrate how to construct the various encryption contexts supported by the OpenABE for ABE and public-key encryption.

4.1 Attribute-based Encryption (ABE)

Similar to the high-level API, prior to calling any ABE context functions the OpenABE must be initialized via `InitializeOpenABE()`.

For a CCA-secure encrypton context, the API is

```
std::unique_ptr<OpenABEContextSchemeCCA> \
    OpenABE_createContextABESchemeCCA(OpenABE_SCHEME s);
```

where `OpenABE_SCHEME` can be one of three currently implemented options:

- `OpenABE_SCHEME_CP_WATERS` for Ciphertext-Policy ABE
- `OpenABE_SCHEME_KP_GPSW` for Key-Policy ABE

Similarly, the `OpenABE_createContextABESchemeCPA()` method exists for CPA-secure encryption contexts (though we recommend CCA security).

4.1.1 Construct Scheme Context

For example, construct a CCA-secure KP-ABE scheme context as follows:

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

Once the KP-ABE scheme CCA context in our example has been constructed, the handle can be used to generate system parameters and derive user private keys:

```
// generate setup params where
// "BN_P254" is the pairing curve for the scheme,
// "mpk" is the master public param ID and
// "msk" is the master secret param ID.
ccaKpAbeContext->generateParams("BN_P254", "mpk", "msk");

// keygen for attribute list "one|two|three"
std::unique_ptr<OpenABEFunctionInput> keyInput = \
            createPolicyTree("((one and two) or three)");
ccaKpAbeContext->keygen(keyInput.get(), "deckey", "mpk", "msk");
```

4.1.2 Encryption and Decryption

The low-level OpenABE API supports encryption in detached mode only. Detached mode means that the ABE ciphertext and AES-GCM ciphertext are stored in **separate** string buffers. You are responsible for exporting and storing the generated ciphertexts. The following example demonstrates the encrypt and decrypt routines for the KP-ABE CCA context:

```
// encrypt
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);

// should succeed
assert(pt1 == pt2);
```

The appropriate inputs would be supplied for the other flavors of ABE. For example, the encrypt routine expects a policy (instead of an attribute list) for the CP-ABE scheme.

4.1.3 Import and Export Ciphertexts

To export the contents of the ciphertext to a string, call the `exportToBytes` method of the `OpenABECiphertext` object:

```
OpenABEByteString ct1Buf;
// exports the ABE ciphertext header and body
ct1.exportToBytes(ct1Buf);
```

To import the ciphertext from a binary string, call the `loadFromBytes` method of the `OpenABECiphertext` object:

```
OpenABECiphertext ct2;
// load the ABE ciphertext header and body
ct2.loadFromBytes(ct1Buf);
```

4.1.4 Import and Export ABE Keys

To export the contents of an ABE private key that was previously loaded or generated by the CCA scheme context, call the `exportKey` routine of the context:

```
OpenABEByteString skBlob;
// export the private key contents in "deckey" into the skBlob buffer
ccaKpAbeContext->exportKey("deckey", skBlob);
```

To import the contents of an ABE private key, call the `loadUserSecretParams` of the CCA scheme context:

```
// load the user secret key from the skBlob and store as "decKey"
ccaKpAbeContext->loadUserSecretParams("deckey", skBlob);
```

Once you no longer require the ABE private key in memory, then call the `deleteKey` routine of the CCA scheme context to erase the sensitive keys:

```
// delete the key with ID "deckey"
ccaKpAbeContext->deleteKey("deckey");
```

4.2 Public-key Encryption (PKE)

For a CCA-secure encrypton context, call the following API:

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

where `OpenABE_SCHEME` is `OpenABE_SCHEME_PK_OPDH`.

4.2.1 Construct Scheme Context

```
// create new KEM context for PKE ECC MQV scheme
std::unique_ptr<OpenABEContextSchemePKE> pkSchemeContext = \
       OpenABE_createContextPKESchemeCCA(OpenABE_SCHEME_PK_OPDH);
```

4.2.2 Generate Keys

```
// Compute Alice's static public and private key
pkSchemeContext->keygen("ID_A", "public_A", "private_A");

// Compute Bob's static public and private key
pkSchemeContext->keygen("ID_B", "public_B", "private_B");
```

4.2.3 Encryption and Decryption

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

// first arg is NULL (unless the user wants to customize the RNG)
// second arg is public key ID of receiver
// third arg is for sender's public key ID
pkSchemeContext->encrypt(NULL, "public_B", "public_A", pt1, &ct);

// first arg is the public key of the sender
// second arg is the private key of the receiver
bool result = pkSchemeContext->decrypt("public_A", "private_B", pt2, &ct);

// check that decryption was successful and plaintext recovered
assert(result && pt1 == pt2);
```