InSiTo Library User Manual

Version 1.1

Oliver Uwira
Falko Strenzke
Martin Döring

August 13th, 2008

Contents

1 Introduction

1.1 Purpose of this document

This user manual is addressed to developers of applications requiring cryptographic functionality. The developers are supposed to know what functionality they are about to use. This manual describes how the InSiTo library is supposed to be used to obtain the required functionality.

The characteristics of the particular cryptographic algorithms provided by the InSiTo library are not subject of this manual, which focusses primarily on the aspect of programming. The reader is referred to cryptography handbooks for more detailed information about the particular algorithms.

Although this manual is focussing on programming, it is not an API reference. Instead, this document provides the reader with a survey of the functionality of the InSiTo library by means of explained source code listings.

1.2 How to use this manual

In Section 3, the global configuration of the InSiTo library is described. Section 4 introduces the mechanisms through which the various algorithms of the InSiTo library are provided with input data. From this section onwards, it will be assumed that said mechanics are known. That is, in the case that code listings would only differ in the input argument types passed to a function, this input data will not be further commented, and the reader is referred to Section 4.

The remainder of this manual consists of thematic sections, each of which covers one major group of cryptographic algorithms. The algorithms themselves are explained by means of code listings that demonstrate how to perform the usual operations associated with a particular algorithm. Following the description of these operations, the reader will find a reference of available algorithms and their respective instantiation code, which is easily transferable to the code listings mentioned above.

As an example, consider digital signatures. The code listing demonstrates how to instantiate an algorithm as well as how to sign and verify input data. This is followed by a list of available digital signature algorithms, the various parameters they require, and a demonstration of how to instantiate the respective algorithm.

2 Building the Library

In order to build the InSiTo library, the following prerequisites have to be met:

- A working C++-compiler is installed. The following compilers have been tested:
  - Linux: GCC (the GNU C++-compiler), version 4.1.2
  - Windows XP: MinGW version 5.1.3 (see `http://www.mingw.org`)
  - Windows XP: Visual C++ 2008 Express Edition (see `https://www.microsoft.com/express/download`). Both the IDE version and the command line version (NMake) are supported.
- CMake (version $\geq$ 2.4) is installed (see `http://www.cmake.org`). Due to an issue with the Windows version of CMake 2.6, the maximal supported version of CMake for Windows XP is 2.4.8. Building has been tested with version 2.4.8 for Windows and version 2.6 for Linux.
- Boost (version $\geq$ 1.34) is installed (see `http://www.boost.org`). Building has been tested with versions 1.34 and 1.35.

We introduce the following shortcuts for different configurations:

- "GCC" denotes Linux with the GNU C++-compiler.
- "MinGW" denotes Windows XP with the MinGW compiler.
- "VS 2008" denotes Windows XP with the Visual C++ 2008 Express Edition IDE.
- "NMake" denotes Windows XP with the command line compiler of the Visual C++ 2008 Express Edition.

CMake is used to generate build files for the different compilers. To do so, switch to the root of the InSiTo library sources. Then, type

- "`cmake .`" for GCC
- "`cmake -G "MinGW Makefiles"`" for MinGW
- "`cmake -G "Visual Studio 9 2008"`" for VS 2008
- "`cmake -G "NMake Makefiles"`" for NMake

Afterwards, start the CMake GUI as follows:

- "`ccmake .`" for Linux
- "`cmakesetup .`" for Windows XP

Now, a number of build options can be adjusted. In the following, we describe the frequently needed options which are not self-explanatory.

- `BUILD_SHARED`, `BUILD_STATIC`: toggle whether to build a shared and static version of the library, respectively. Note that the tests are always linked to the shared version. When building for VS 2008, always enable both the shared and the static version.

- `Boost_INCLUDE_DIR`: specify the location of the Boost header files on the system. The directory specified here must contain the `boost` subdirectory containing the headers.

- `USE_TR1`: specify which implementation of the TR1 functionality is used. "std" chooses the implementation provided by the compiler (assuming that it provides them in the first place), "boost" chooses the Boost implementation. The InSiTo library uses only the `tr1/memory` functionality from TR1. Note that the application build against the library has to make the same choice concerning the TR1 implementation to be used. Otherwise, a problem with multiple definitions will arise during the compilation. For information about TR1 see `http://aristeia.com/EC3E/TR1_info_frames.html`.

- `USE_INTERNAL_BOOST`: choose whether the pre-built Boost libraries shipped with the InSiTo-library shall be used. Pre-built libraries are only available for Windows. To use these libraries, set this option to "`yes`". If setting this option to "`no`" for Windows XP, the Boost library `boost_unit_test_ framework` has to be built and installed (needed for the tests to compile). For Linux, the Boost libraries `boost_unit_test_framework` and referenced libraries therein have to be built and installed (needed for the tests to compile). So it is advised to install the full set of the boost libraries (`http://www.boost.org/`). Either version 1.34 or 1.35 will do fine.

For the Linux version of CMake, a help for the currently selected option can be displayed by pressing "h", making it possible to read the full description of a property (which might be cut off in the main screen). To modify a parameter value, press "return". Pressing "return" again confirms the new value. Once all build parameters are adjusted, press "c", then "g".

For Windows, options can be selected with the mouse. When moving the mouse over an option, an explanation is shown. Once all build parameters are adjusted, press "Configure", then "OK". Finally, shut down the CMake GUI.

After all build parameters are set, the library is built as follows:

- GCC: execute "`make`".

- MinGW: execute "`mingw32-make`".

- NMake: execute "`nmake`".

- VS 2008: open the generated "`BOTAN.sln`" solution file with the Visual Studio IDE. Build the projects "`insito-bib-1.7.2`" and "`insito-bib_static-1.7.2`. Note that in case of the debug build, the string `_d` is appended to the projects' and resulting files' names. Afterwards, rename the built library file `<...>_static<...>.lib`: remove the string `_static` from the file's name.

For VS 2008, it is advised to disable the compiler warnings 4244, 4250, and 4290 for all projects contained in the "`BOTAN.sln`" solution.

The tests are built as follows:

- GCC/MinGW/NMake: execute "`make/mingw32-make/nmake tests`".
- VS 2008: build the "`tests`" project. Afterwards, copy the following files to the InSiTo base dir:
  - from the `Release` or `Debug` directory (depends on the build type you chose) the files `tests.exe`, `insito-bib<...>.dll` and `insito-bib<...>.lib`;
  - all files in the `misc/lib/win32_vs2008` directory.

The original Botan tests are built with

- GCC/MinGW/NMake: execute "`make/mingw32-make/nmake check`".
- VS 2008: build the "`check` project".

This also allows benchmarking of certain algorithms.

To run the tests, execute either of the following commands:

- `./tests`
- `./check --validate`
- `./check --benchmark`

For Windows XP, the "`pthread`" library is needed in order to run the "`tests`" program. Pre-built libraries for MinGW and VS 2008 are shipped with the InSiTo library. They have to be copied into the same directory as the "`tests`" executable. To do so, type

- MinGW: `copy misc\lib\win32_mingw\pthreadGCE2.dll .`
- VS 2008: `copy misc\lib\win32_vs2008\pthreadVCE2.dll .`

For Linux, to install the headers and the library, execute the following command with root privileges:

```
make install
```

For Windows, the library is contained in the root of the InSiTo library sources. The headers are in the `build/include/` subdirectory.

3 Global Configuration

3.1 Default Configuration

The InSiTo library uses a centralized global set of configuration parameters in the form of key-value pairs. If no user defined configuration is provided, the InSiTo library will use the default configuration, defined in the file `policy.cpp`.

As can be seen from this file, the keys of the configuration parameters are divided into different sections, such as `x509` or `x509/ca`. Finally, the full key is defined by an additional key name, e.g. `x509/ca/default_expire`.

3.2 User defined Configuration

To use a user defined configuration, a text file of the following form has to be created:

```
[<section name>]
<key 1> = <value for key 1>
<key 2> = <value for key 2>
.
.
.
```

An example excerpt from such a file would be:

```
[x509/ca]
allow_ca          = false
basic_constraints = always
default_expire    = 1y       # default expiry time for new certs
signing_offset    = 30s      # offset the PKCS #10 validity times
                               by this amount
```

As it can be seen, comments can be provided after a `#` character.

Note that all keys that can be found in the file `policy.cpp` should appear in the the text file with the user configuration. Otherwise, runtime errors might occur because of missing entries.

To tell the InSiTo library that it should use the text file with the user defined configuration instead of the default configuration, a constructor of the `LibraryInitializer` has to be called with a specific argument:

```
Botan::InitializerOptions init_options("config=<filename>");
```

where `<filename>` has to be replaced by the name (including path) to the configuration file.

4 Working with data

4.1 Information flow – pipes and filters

Many common uses of cryptography involve processing one or more streams of data (be it from sockets, files, or hardware devices). The InSiTo library provides a mechanism which facilitates setting up data flows through various operations, such as compression, encryption, and base64 encoding.

Each of these operations is implemented in what are called *filters* in the terminology of the InSiTo library. A set of filters are created and placed into a *pipe*. Information "flows" through the pipe, being passed from one filter to the next, until the end of the pipe is reached, where the output is collected for retrieval.

This design is similar to the usage of pipes in the Unix shell environment. To demonstrate the usage of pipes in the InSiTo library, the following listing shows an example, which uses a pipe to base64-encode some strings:

```
Pipe pipe(create_shared_ptr<Base64_Encoder>());
pipe.start_msg();
pipe.write("message 1");
pipe.end_msg();

// process_msg(x) is a short form for start_msg() &&
   write(x) && end_msg()
pipe.process_msg("message2");
// first encoded message
std::string m1 = pipe.read_all_as_string(0);
// second encoded message
std::string m2 = pipe.read_all_as_string(1);
```

The pipe constructor is passed a `std::tr1::shared_ptr<Filter>`. This is the major difference between the InSiTo library and its base library Botan, which only supports ordinary C/C++ pointers and thus leaves the task of freeing allocated memory to the programmer. Apart from this difference, the pipe API of the InSiTo library is the same as in Botan, and therefore we refer the reader to the Botan API reference for more detailed information about pipes.

4.2 Data input and output

The member functions `Pipe::write()` and `Pipe::process_msg()` are overloaded to support various means of passing data to a pipe. These are:

- `void Pipe::write(const byte input[], u32bit length)` – Write the content of a byte array into the pipe. The second argument specifies the number of bytes to be written.
- `void Pipe::write(const MemoryRegion<byte>& input)` – Write the content of a memory region into a pipe. The `MemoryRegion<T>` class

is the base class of container classes such as `SecureVector<T>`. All classes derived from `MemoryRegion<T>` implement secure memory management. That is, swapping of virtual memory pages occupied by a `MemoryRegion<T>` to the hard disk is prevented, which keeps sensible data in non-permanent storage only.

- `void Pipe::write(const std::string& str)` – Write a string into a pipe. This is more of a convenience method, because normally data representations based on a stream of bytes are much closer to the real-life applications of the library.
- `void Pipe::write(DataSource& source)` – Write a data source into a pipe. Data sources are a group of interfaces specified in `botan/data_src.h`, that should be implemented by any data stream classes that an application using InSiTo might specify. Doing this will allow the writing of those streams into a pipe. Note that a `Pipe` is itself derived from `DataSource`.
- `void Pipe::write(byte input)` – Write a single byte into the pipe. This is the only overloaded version of `Pipe::write()` that does not have an equivalent version of `Pipe::process_msg()`.

For reading the results of a pipe operation, the following member functions can be used. Every member function listed below specifies an `u32bit`-argument that serves as an index into the list of processed messages of the pipe (please see the example listing in Section 4.1).

- `u32bit Pipe::read(byte& out, u32bit msg)` – Read a single byte from the specified message. The number of read bytes is returned.
- `u32bit Pipe::read(byte output[], u32bit length, u32bit msg)` – Read a number of bytes, specified by the second argument, from the specified message. The number of read bytes is returned.
- `SecureVector<byte> Pipe::read_all(u32bit msg)` – Read a complete message into a `SecureVector<byte>`.
- `std::string Pipe::read_all_as_string(u32bit msg)` – Read a complete message into a `std:string`.

Reading data out of a pipe will advance the pipe, such that the read data cannot be read again. Therefore, potential lookahead functionality cannot be implemented by means of the `read()`-functions. Instead, `Pipe` provides a group of related `peek()`-functions, which will not be discussed here in further detail.

5 Symmetric Algorithms

Symmetric algorithms provided by the InSiTo library cover block cipher algorithms, message authentication codes (MACs), and message digests (hash functions),

Except for message digests, which require no cryptographic key at all, the keys used by the various algorithms are of the type `SymmetricKey`. This type implements an octet string of variable length. The class provides a constructor that takes the number of bytes (i.e octets). This constructor may be used to create random octet string keys as needed.

5.1 Message Digests

Message digests are computed by means of a `Hash_Filter`, which can be used with a `Pipe`. The code listing below demonstrates how message digests of the InSiTo library are computed:

```
#include <botan/api/botan.h>

...

std::string text = "message";
std::string md;

...

Pipe pipe(create_shared_ptr<Hash_Filter>("SHA-1"),
          create_shared_ptr<Hex_Encoder>());
pipe.process_msg(text);
md = pipe.read_all_as_string(0);
```

A `std::tr1::shared_ptr<Hash_Filter>` is created by using the template function `create_shared_ptr<T>()`. The arguments passed to this function are passed to the respective constructors of the class `T`. In case of a `Hash_Filter`, the constructor must be passed a string alias of the required algorithm.

The following message digests are provided by the InSiTo library:

- `SHA-1`
- `SHA-224`
- `SHA-256`
- `SHA-384`
- `SHA-512`
- `RIPEMD-160`

These codes are also used as string aliases with `create_shared_ptr<Hash_Filter>()`.

5.2 Message Authentication Codes

Message authentication codes (MACs) are computed by means of a `MAC_Filter`, which can be used with a `Pipe`. The code listing below demonstrates how the MACs of the InSiTo library are computed:

```
#include <botan/api/botan.h>

...

std::string text = "message";
std::string mac;

...

// create a symmetric key of size 20 byte
SymmetricKey key(20);
Pipe pipe(create_shared_ptr<MAC_Filter>("HMAC(SHA-1)",
    key),
          create_shared_ptr<Hex_Encoder>());
pipe.process_msg(text);
mac = pipe.read_all_as_string(0);
```

A `std::tr1::shared_ptr<MAC_Filter>` is created by using the template function `create_shared_ptr<T>()`. The arguments passed to this function are passed to the respective constructors of the class `T`. In case of a `MAC_Filter`, the constructor must be passed a string alias of the required algorithm and a `SymmetricKey`.

The following MACs are provided by the InSiTo library:

- `HMAC` (in combination with a message digest)
- `CBC-MAC` (in combination with a block cipher)
- `CMAC` (in combination with a block cipher)

These codes are also used as string aliases with `create_shared_ptr<MAC_Filter>()`.

As can be seen in the listing above, the string alias of HMAC is constructed by appending a message digest alias in braces. The same is valid for CBC-MAC and CMAC, but for these algorithms, a block cipher must be specified in braces instead of a message digest, e.g `CMAC(AES)`.

The list of available message digest aliases is given in Section 5.1. The block cipher aliases are given in Section 5.3.

5.3 Block Ciphers

Block ciphers are computed by means of a `Keyed_Filter`, which can be used with a `Pipe`. The code listing below demonstrates how block ciphers of the

InSiTo library are used:

```
#include <botan/api/botan.h>

std::string text = "message";

...

// Create a symmetric key of size 16 bytes
SymmetricKey key(16);

// Create an initialization vector of size 16 bytes
InitializationVector iv(16);

Pipe enc_pipe(get_cipher("AES/CBC/NoPadding", key, iv,
              ENCRYPTION),
              create_shared_ptr<Hex_Encoder>());
enc_pipe.process_msg(text);

SecureVector<byte> ciphertext = enc_pipe.read_all(0);

Pipe dec_pipe(create_shared_ptr<Hex_Decoder>(),
              get_cipher("AES/CBC/NoPadding", key, iv,
              DECRYPTION));
dec_pipe.process_msg(ciphertext);

SecureVector<byte> plaintext = dec_pipe.read_all(0);
```

A `std::tr1::shared_ptr<Keyed_Filter>` is created by using the global function `get_cipher()`. The arguments passed to this function are:

- `const std::string& algo_spec` – The algorithm alias
- `const SymmetricKey& key` – The symmetric key
- `const InitializationVector& iv` – The initialization vector
- `Cipher_Dir direction` – The direction of the cipher operation. `Cipher_Dir` is an `enum` that specifies two directions: `ENCRYPTION` and `DECRYPTION`.

The following block cipher algorithms are provided by the InSiTo library:

- `TripleDES` (key length of 168 bits)
- `AES` (supports key lengths of 128 bits, 192 bits and 256 bits)

The algorithm alias string for a block cipher is of the form `ALGORITHM/MODE/PADDING`. If any part of the alias except for `ALGORITHM` is omitted, defaults will be used. The following modes are provided by the InSiTo library:

- `ECB`
- `CBC`
- `CFB`
- `OFB`
- `CTR`

These codes are also used as string aliases for `MODE`. The following padding methods are provided by the InSiTo library:

- `PKCS7`
- `OneAndZeros`
- `X9.23`
- `NoPadding`

These codes are also used as string aliases for `PADDING`.

6 Asymmetric Algorithms

Asymmetric algorithms provided by the InSiTo library cover asymmetric cipher (encryption) algorithms, digital signatures, and key agreement schemes.

The public and private keys used by the various algorithms are of the type `Public_Key` and `Private_Key`, respectively. In the InSiTo library, `Private_Key` is a subclass of `Public_Key` and encapsulates all the information that is needed by its associated algorithm. This design allows for a very easy method of obtaining a key pair, which is demonstrated in the next listing:

```
#include <botan/api/rsa.h>

...

// Create an RSA private key of size 1024 bits
RSA_PrivateKey prv_key(1024);

// Create the corresponding RSA public key object via
   a cast to the super type
RSA_PublicKey pub_key = prv_key;
```

6.1 Asymmetric Ciphers

The InSiTo library provides one asymmetric cipher algorithm, RSA, and two encoding methods, `PKCS1v15` and `RSA-OAEP`. It is also possible to use the algorithm in raw mode.

Asymmetric ciphers are computed by means of two special objects, `PK_Encryptor` and `PK_Decryptor`. These are used to construct the suitable `Filter` classes, `PK_Encryptor_Filter` and `PK_Decryptor_Filter`, which can be used with a `Pipe`.

The code listing below demonstrates how asymmetric ciphers of the InSiTo library are used:

```
#include <botan/api/botan.h>
#include <botan/api/look_pk.h>
#include <botan/api/rsa.h>

...

std::string plaintext = "message";
std::string ciphertext;

...

// Create an RSA private key of size 1024 bits
RSA_PrivateKey prv_key(1024);

// Create PK_Encryptor and PK_Decryptor filters
std::auto_ptr<PK_Encryptor> enc_aptr =
   get_pk_encryptor(prv_key, "PKCS1v15");
std::auto_ptr<PK_Decryptor> dec_aptr =
   get_pk_decryptor(prv_key, "PKCS1v15");
std::tr1::shared_ptr<PK_Encryptor> enc(enc_aptr);
std::tr1::shared_ptr<PK_Decryptor> dec(dec_aptr);

// Encrypt the plaintext
Pipe enc_pipe(create_shared_ptr<PK_Encryptor_Filter>(
   enc),
              create_shared_ptr<Hex_Encoder>());
enc_pipe.process_msg(plaintext);

ciphertext = enc_pipe.read_all_as_string(0);

// Decrypt the ciphertext
Pipe dec_pipe(create_shared_ptr<Hex_Decoder>(),
              create_shared_ptr<PK_Decryptor_Filter>(
                 dec));
dec_pipe.process_msg(ciphertext);
```

```
plaintext = dec_pipe.read_all_as_string(0);
```

A `std::tr1::shared_ptr<PK_Encryptor>` is created by passing the result of the global function `get_pk_encryptor()` to the constructor of `std::tr1::shared_ptr<PK_Encryptor>`. A `std::tr1::shared_ptr<PK_Decryptor>` is created by using the global function `get_pk_decrpytor()`.

The arguments passed to `get_pk_encryptor()` are:

- `const PK_Encrypting_Key& key` – An encrypting key suitable for the specified algorithm
- `const std::string& eme` – A string alias for an encoding method.

The arguments passed to `get_pk_decryptor()` are:

- `const PK_Decrypting_Key& key` – An encrypting key suitable for the specified algorithm
- `const std::string& eme` – A string alias for an encoding method.

The following encoding methods are provided by the InSiTo library.

- `Raw` – no encoding will be used
- `PKCS1v15` – PKCS #1 v1.5 encoding
- `EME1` – OAEP (PKCS #1 v2.1) encoding (in combination with a message digest)

A string alias of the OAEP encoding method is constructed by appending a message digest alias in braces. This is analogous to the creation of message authentication code filters described in Section 5.2. The list of available message digest aliases is given in Section 5.1.

6.2 Digital Signature Algorithms

The InSiTo library provides two digital signatures, `RSA` and `ECDSA`. For RSA, the encoding methods `PKCS1v15` and `RSA-PSS` are available. It is also possible to use RSA in raw mode. EC-DSA supports EMSA1 and EMSA1_BSI encoding. The latter is a non standard or respectively future standard encoding variant and is explained in Section 6.4.

Digital signatures are computed by means of two special objects, `PK_Signer` and `PK_Verifier`. These are used to construct the suitable `Filter` classes, `PK_Signer_Filter` and `PK_Verifier_Filter`, which can be used with a `Pipe`.

The code listing below demonstrates how digital signatures of the InSiTo library are used:

```
#include <botan/api/botan.h>
#include <botan/api/look_pk.h>
#include <botan/api/rsa.h>

...

std::string text = "message";

...

// Create an RSA key pair
RSA_PrivateKey prv_key(1024);
RSA_PublicKey  pub_key = prv_key;

// Create PK_Signer and PK_Verifier filters
std::auto_ptr<PK_Signer> sig_aptr = get_pk_signer(
   prv_key, "Raw");
std::tr1::shared_ptr<PK_Signer> signer(sig_aptr);
std::auto_ptr<PK_Verifier> ver_aptr = get_pk_verifier(
   prv_key, "Raw");
std::tr1::shared_ptr<PK_Verifier> verifier(ver_aptr);

// Sign the message
Pipe sig_pipe(create_shared_ptr<PK_Signer_Filter>(
   signer));
sig_pipe.process_msg(text);
SecureVector<byte> sig_bytes = sig_pipe.read_all(0);

// Verify the signature
// Create a PK_Verifier_Filter by passing a
   PK_Verifier AND the digital signature
Pipe ver_pipe(create_shared_ptr<PK_Verifier_Filter>(
   verifier, sig_bytes));
ver_pipe.process_msg(text);
SecureVector<byte> ver_result = ver_pipe.read_all(0);
```

A `std::tr1::shared_ptr<PK_Signer>` is created by passing the result of the global function `get_pk_signer()` to the constructor of `std::tr1::shared_ptr<PK_Signer>`. The arguments passed to `get_pk_signer()` are:

- `const PK_Signing_Key& key` – An signing key suitable for the specified algorithm
- `const std::string& eme` – A string alias for an encoding method.
- `Signature_Format sig_format` – The signature format. `Signature_Format` is an `enum` that specifies two formats: `IEEE_1363`, which is the default argument, and `DER_SEQUENCE`.

A `std::tr1::shared_ptr<PK_Decryptor>` is created by using the global function `get_pk_verifier()` respectively. The arguments of `get_pk_verifier()` are:

- `const PK_Verifying_with_MR_Key& key` – An verifying key suitable for the specified algorithm
- `const std::string& eme` – A string alias for an encoding method.
- `Signature_Format sig_format` – see `get_pk_signer()` above

In order to create a `PK_Verifier_Filter`, the digital signature must be passed to the constructor along with the `PK_Verifier`.

Please note: The code listing uses the RSA algorithm and therefore RSA keys are generated in the beginning. For using the EC-DSA algorithm, EC-DSA keys must be used. These keys are of the types `ECDSA_PublicKey` and `ECDSA_PrivateKey`. Once these keys are available, the process of calculating and verifying signatures is analogous to the way shown in the listing. Please see Section 6.4 for information about the usage of elliptic curve functionality of the InSiTo library.

6.3 Key Agreement Algorithms

The InSiTo library provides two variations of a Diffie-Hellman key agreement scheme. One scheme (PKCS #3) operates on a classical $(\mathbb{Z}/p\mathbb{Z})^*$ group, whereas the other one (EC-KAEG) operates on an elliptic curve point group.

The operations necessary to compute a shared secret are encapsulated by the class `PK_Key_Agreement`. Any party involved in a key agreement process can use a `PK_Key_Agreement` object to update received public data and, finally, compute the shared secret.

The code listing below demonstrates how key agreement schemes of the InSiTo library are used:

```
#include <botan/api/botan.h>
#include <botan/api/look_pk.h>
#include <botan/api/dh.h>

...

// Create the public and private values for the
// key agreement scheme. Its participants be
// Alice and Bob.

// We also create a domain. In this case, we will use
// the (Z/nZ)* method and therefore the domain is a
// DL_Group object.

DL_Group domain ("modp/ietf/1024");
```

```
DH_PrivateKey prv_key_alice(domain);
DH_PublicKey  pub_key_alice = prv_key_alice;

DH_PrivateKey prv_key_bob(domain);
DH_PublicKey  pub_key_bob = prv_key_bob;

// Create the PK_Key_Agreement objects for Alice and
   Bob

std::auto_ptr<PK_Key_Agreement> kas_alice_aptr =
    get_pk_kas(prv_key_alice, "Raw");
std::tr1::shared_ptr<PK_Key_Agreement> kas_alice(
   kas_alice_aptr);

std::auto_ptr<PK_Key_Agreement> kas_bob_aptr =
    get_pk_kas(prv_key_bob, "Raw");
std::tr1::shared_ptr<PK_Key_Agreement> kas_bob(
   kas_bob_aptr);

// Use the PK_Key_Agreement object to derive a
   symmetric key.
// The computation of the shared secret and the
   derivation of the symmetric key are bundled by the
   PK_Key_Agreement object.

SymmetricKey sym_key_alice = kas_alice->derive_key(16,
    pub_key_bob);

// Repeat the last step for Bob. The results are equal
   .
SymmetricKey sym_key_bob = kas_bob->derive_key(16,
   pub_key_alice);
```

A `std::tr1::shared_ptr<PK_Key_Agreement>` is created by passing the result of the global function `get_pk_kas()` to the constructor of `std::tr1::shared_ptr<PK_Key_Agreement>`.

The arguments passed to `get_pk_kas()` are:

- `const PK_Key_Agreement_Key& key` – A (private) key agreement key suitable for the specified algorithm
- `const std::string& kdf` – A string alias for a key derivation function

The following key derivation methods are provided by the InSiTo library:

- `Raw`

- `KDF1`
- `KDF2`
- `X9.42-PRF`

The InSiTo library also provides some built-in $(\mathbb{Z}/p\mathbb{Z})^*$ groups of different size, which can be instantiated by passing a group identifier to the constructor of `DL_Group`. The following group identifiers can be used:

- `modp/ietf/768` (768 bits)
- `modp/ietf/1024` (1024 bits)
- `modp/ietf/1536` (1536 bits)
- `modp/ietf/2048` (2048 bits)
- `modp/ietf/3072` (3072 bits)
- `modp/ietf/4096` (4096 bits)

Please note: The code listing uses a $(\mathbb{Z}/p\mathbb{Z})^*$ group as the domain of the key agreement scheme (PKCS #3). Therefore, keys of the types `DH_PrivateKey` and `DH_PublicKey` are generated in the beginning. For using the ECKAEG algorithm, ECKAEG keys must be used. These keys are of the types `ECKAEG_PublicKey` and `ECKAEG_PrivateKey`. Once these keys are available, the process of computing the shared secret and deriving a symmetric key is analogous to the way shown in the listing. Please see Section 6.4 for information about the usage of elliptic curve functionality of the InSiTo library.

6.4 Using Elliptic Curves

The InSiTo library also provides seven standard elliptic curve point groups, which can be instantiated by passing the OID of the curve to the global function `get_EC_Dom_Pars_by_oid()`. These standard curves and their OIDs are:

- `OID = "1.3.36.3.3.2.8.1.1.1"` (BrainpoolP160r1)
- `OID = "1.3.36.3.3.2.8.1.1.3"` (BrainpoolP192r1)
- `OID = "1.3.36.3.3.2.8.1.1.5"` (BrainpoolP224r1)
- `OID = "1.3.36.3.3.2.8.1.1.7"` (BrainpoolP256r1)
- `OID = "1.3.36.3.3.2.8.1.1.9"` (BrainpoolP320r1)
- `OID = "1.3.36.3.3.2.8.1.1.11"` (BrainpoolP384r1)
- `OID = "1.3.36.3.3.2.8.1.1.13"` (BrainpoolP512r1)

Once an elliptic curve has been instantiated, it can be used to create keys for either the EC-KAEG key agreement scheme or the EC-DSA digital signature. The process is demonstrated in the following listing:

```
#include <botan/api/botan.h>
#include <botan/api/look_pk.h>
#include <botan/api/ec.h>
#include <botan/api/ec_dompar.h>

// Create an elliptic curve
EC_Domain_Params dom_pars = get_EC_Dom_Pars_by_oid
   ("1.3.36.3.3.2.8.1.1.1");

// Create keys for EC-DSA digital signature algorithm.
// Further proceeding is similar to the RSA digital
   signature algorithm described before.
ECDSA_PrivateKey prv_key(dom_pars);
ECDSA_PublicKey  pub_key = prv_key;

// Create keys for EC-KAEG key agreement scheme
// Further proceeding is similar to the PCKS #3 key
   agreement scheme described before.
ECKAEG_PrivateKey prv_key_alice(dom_pars);
ECKAEG_PublicKey  pub_key_alice = prv_key_alice;
```

Once the elliptic curve and the EC-DSA or EC-KAEG keys have been created, the further process is analogous to the RSA digital signature (see Section 6.2) and the PKCS #3 key agreement scheme (see Section 6.3), respectively.

Signature Encoding Besides the standardized signature encoding methods like EMSA1, EMSA2, etc., the InSiTo library supports a yet non standardized method named EMSA1_BSI. In contrast to EMSA1, this encoding type does not allow for the length of the output of the hash function to be larger than the input size of the core signature algorithm. For ECDSA, this means that the bit length of the order of the base point must be at least equal to the bit length of the output of the chosen hash function, otherwise an exception will be thrown when attempting to perform a signing or verifying operation with the signer or verifier.

To create a `PK_Signer` using this encoding, simply type

```
auto_ptr<Botan::PK_Signer> pk_signer = Botan::get_pk_signer(priv_key, "EM
```

In the default global configuration, this encoding is associated with the ECDSA OIDs. To override this, you have to provide your own configuration. See section 3 to learn how to do this.

7 Random Number Generation

The InSiTo library provides three pseudo random number generators:

- the PRNG defined in ANSI X9.31, Appendix A,
- `BBS` and
- `SHA1-PRNG`.

Both generators need a seeded base random number generator as an entropy source. Attempting to use any random number generator unseeded will cause an exception. For more detailed information about the random number generation of the InSiTo library, the reader is referred to the API documentation of the base library Botan.

The code listing below demonstrates how random number generators of the InSiTo library are used:

```
#include <botan/api/botan.h>
#include <botan/api/base.h>
#include <botan/int/bbs.h>
#include <botan/int/randpool.h>

#define SEEDLEN 600
#define RNUMLEN 20

...

byte seed[SEEDLEN] = {0,0,0, ... ,0};
byte out [RNUMLEN] = {0,0,0, ... ,0};

...

// Create a base random number generator and seed it.
// Note that an exception will also be thrown here if
   the seed length is too small.
std::tr1::shared_ptr<RandomNumberGenerator> rn_ptr(new
    Randpool());
rn_ptr->add_entropy(seed, SEEDLEN);

// Create a BBS random number generator and seed it.
// Note that an exception will also be thrown here if
   the seed length is too small.
RandomNumberGenerator* rng = (RandomNumberGenerator*)(
   new BBS(rn_ptr));
rng->add_entropy(seed, SEEDLEN);

// Randomize the contents of the byte array out[].
```

```
// The contents of the array can be interpreted as a
   random number of the size (in bytes) of RNUMLEN
rng->randomize(out, RNUMLEN);

// Delete the RNG after usage because we have not used
    a smart pointer but an ordinary C/C++ pointer.
delete rng;
```

For instantiating a SHA1-PRNG, we would simply use the `SHA1PRNG` constructor instead of the `BBS` constructor:

```
RandomNumberGenerator* rng =
                      (RandomNumberGenerator*)(new SHA1PRNG());
```

Both constructors also have a default argument, which will result in the usage of the global random number generator of the InSiTo library as the base random number generator.

8 Key Storage

For storing key objects of the InSiTo library, two encoding methods are provided: `PEM` and `DER`. The following functions (declared in the header file `botan/x509_keys.h`) can be used to encode key objects:

- `void encode(const Public_Key key&, Pipe& pipe, X509_Encoding encoding)` – Encode a `Public_Key` in the specified `X509_Encoding` and pass the encoded key into a `Pipe`. Possible encodings are `PEM`, which is the default argument, and `RAW_BER` (i.e DER).
- The following overloaded functions, each returning a `std::auto_ptr <Public_Key>`:
  - `load_key(std::tr1::shared_ptr<DataSource>& source)` – load key from a data source
  - `load_key(const std::string& fsname)` – load key from a file with the given filename
  - `load_key(const MemoryRegion<byte>& mem)` – load key from a memory region

```
#include <botan/x509_keys.h>
#include <botan/rsa.h>
...

RSA_PrivateKey prv_key(1024);
RSA_PublicKey  pub_key = prv_key
```

```
// PEM encode pub_key

// A pipe without filters
Pipe pipe();
pipe.start_msg();
X509::encode(pub_key, pipe);
pipe.end_msg();

// Reload pub_key from pipe
// Note: Pipe is derived from DataSource
std::auto_ptr<PublicKey> pub_key_aptr = load_key(pipe)
   ;
```

9 X509 Certificates

9.1 Creating a CA certificate

In the InSiTo library, a CA certificate can be created by using the function `X509_Certificate X509::create_self_signed_cert()`. The method takes the following arguments:

- `const X509_Cert_Options& opts` – certificate parameters
- `const Private_Key& key` – the private key of the CA

The following listing demonstrates the use of this function:

```
// Create Certificate options
X509_Cert_Options ca_opts("Test CA/DE/InSiTo/Examples
   ");
ca_opts.uri = "http://botan.randombit.net";
ca_opts.dns = "botan.randombit.net";
ca_opts.email = "testing@randombit.net";
ca_opts.CA_key(1);

// Create a private key for the CA
RSA_PrivateKey prv_key_ca(1024);

// Create the CA certificate by self-signing
X509_Certificate cert_ca = X509::
   create_self_signed_cert(ca_opts, prv_key_ca);
```

9.2 Creating a CA object

A CA object is creating by passing the constructor of the class `X509_CA` the CA certificate as well as the private key of the CA. The following listing demonstrates the use:

```
// Assume we have these objects from section 8.1
RSA_PrivateKey prv_key_ca = ...
X509_Certificate cert_ca = ...

// Create a CA object
X509_CA ca(cert_ca, prv_key_ca);
```

9.3 Creating a certificate request

In the InSiTo library, a certificate request can be created by using the function `PKCS10_Request X509::create_cert_req()`.

- `const X509_Cert_Options& opts` – certificate parameters
- `const Private_Key& key` – the private key to be certified

The following listing demonstrates the use of this function:

```
// Create Certificate options
X509_Cert_Options req_opts_alice("Test Alice/DE/InSiTo
   /Examples");
req_opts_alice.uri = "http://www.foobar.net";
req_opts_alice.dns = "www.foobar.net";
req_opts_alice.email = "alice@foobar.net";
req_opts_alice.add_ex_constraint("PKIX.EmailProtection
   ");

// Create a private key for the CA
RSA_PrivateKey prv_key_alice(768);

// Create a certificate request
PKCS10_Request req_alice = X509::create_cert_req(
   req_opts_alice, prv_key_alice);
```

9.4 Creating a certificate

A certificate can be created by letting a CA sign a certificate request. This can be done by using the member function `X509_Certificate X509_CA::sign_request()`.

- `const PKCS10_Request& req` – the PCKCS #10 request object
- `u32bit expire_time = 0`

The following listing demonstrates this usage:

```
// Assume we have these objects from section 8.2 and
   8.3, respectively
X509_CA ca = ...
PKCS10_Request req_alice = ...

// Sign the request to create a certificate
X509_Certificate cert_alice = ca.sign_request(
   req_alice);
```

9.5 Creating and Using CRLs

The following listing demonstrates the following operations concerning CRLs:

- Creating a CRL
- Adding a CRL a certificate to a `X509_Store` key store object
- Revoking a certificate

```
// Assume we have these objects from section 8.2 and
   8.4, respectively
X509_CA ca = ...
X509_Certificate cert_alice = ...

// Create a new CRL
X509_CRL crl1 = ca.new_crl();

// Add the CRL and Alice's certificate to a key store.
X509_Store store;
store.add_cert(cert_ca, true); // second arg == true:
   trusted CA cert

if(store.add_crl(crl1) != VERIFIED)
{
    // CRL invalid
}
else
{
    // CRL valid
}

if(store.validate_cert(cert_alice) != VERIFIED)
{
    // Alice's certificate invalid
}
else
```

```
{
    // Alice's certificate valid
}

// Revoke Alice's certificate
std::vector<CRL_Entry> revoked;
revoked.push_back(cert_alice);
X509_CRL crl2 = ca.update_crl(crl1, revoked);
store.add_crl(crl2) != VERIFIED
```

9.6 Storing and loading certificates

A certificate can be encoded by using the member functions

```
SecureVector<byte> BER_encode() const
```

or

```
std::string PEM_encode() const
```

for DER and PEM encoding, respectively. Loading a certificate is done by passing the file name of an encoded certificate to the constructor of `X509_Certificate`.

The following listing demonstrates this use:

```
// Assume we have this object from section 8.4
X509_Certificate cert_alice = ...

// Export Alice' certificate into a PEM encoded file
std::ofstream x509_file("cert_alice.pem", std::ios::
   binary);
x509_file << cert_alice.PEM_encode();
x509_file.close();

// Import Alice's certificate from a PEM encoded file
   and validate it
X509_Certificate cert_tmp("cert_alice.pem");
```

10 EAC 1.1 CV Certificates

This section explains the use of the EAC 1.1 Card Verifiable Certificates (CVC).

10.1 Overview

InSiTo library supports the following EAC related objects:

CVC's The CVC's themselves are represented by the class `EAC1_1_CVC`.

CVC Requests CVC Requests, i.e. requests that only contain an internal self signature, are represented by the class `EAC1_1_Req`.

CVC ADO Requests The signed CVC ADO Requests are represented by the class `EAC1_1_ADO`.

The convenience functions intended for clients to use for the creation of the above EAC objects are found in the file `cvc_self.h`. The functions therein are divided into two namespaces: `CVC_EAC` and `DE_EAC`. The former contains the lower level functions for the creation of general EAC objects, while the latter specifically implements the german EAC rules (they build up on the former).

For a list and description of the EAC objects and the related convenience functions, please refer to the API reference documentation.

In the following, examples are given for the use of the various EAC objects.

10.2 Global Configuration

The following global configuration parameters are used in the context of CVC's:

- `eac/ca/cvca_validity_months`, defaults to 12,
- `eac/ca/dvca_validity_months`, defaults to 3,
- `eac/ca/is_validity_months`, defaults to 1.

10.3 Examples

10.3.1 Creation of CVCA

To create a CVCA, the user has to provide the `EC_Domain_Params` object and create a private key for these domain parameters. Furthermore, the user has to choose the hash algorithm used by this CA for signing and determine the CAR/CHR value.

```
Botan::EC_Domain_Params dom_pars(Botan::
   get_EC_Dom_Pars_by_oid("1.3.36.3.3.2.8.1.1.5"));
Botan::ECDSA_PrivateKey cvca_privk(dom_pars);
string hash("SHA-224");
Botan::ASN1_Car car("DECVCA00001");
Botan::EAC1_1_CVC cvca_cert = Botan::DE_EAC::
   create_cvca(cvca_privk, hash, car, true, true);
```

The last two boolean parameters are the 'iris" and "fingerprint" flags that will be incorporated in the certificate holder authorization (CHA). So in this example, this CVCA will be allowed to read both biometric properties, and all certificates signed with this CVCA will inherit this property.

10.3.2 Creating a link certificate

The following piece of code creates a link certificate between two CVCA certificates:

```
Botan::EAC1_1_CVC link12 = Botan::DE_EAC::link_cvca(
   cvca_cert1, cvca_privk1, cvca_cert2);
```

Here, the certificate `cvca_cert1` (whose private key is `cvca_privk1`) signs the link certificate with the certificate holder being `cvca_cert2`.

10.3.3 Creating an unsigned DVCA request

To create a DVCA request (in constrast to an ADO request, this object type carries only a self signature), do the following:

```
Botan::EC_Domain_Params dom_pars(Botan::
   get_EC_Dom_Pars_by_oid("1.3.36.3.3.2.8.1.1.5"));
string hash("SHA-224");
Botan::ECDSA_PrivateKey dvca_priv_key(dom_pars);
Botan::EAC1_1_Req dvca_req = Botan::DE_EAC::
   create_cvc_req(dvca_private_key, Botan::ASN1_Chr("
   DEDVCAEPASS"), hash);
```

10.3.4 Creating an ADO DVCA request

The following piece of code shows how to create an ADO request:

```
Botan::EAC1_1_ADO dvca_ado2 = Botan::CVC_EAC::
   create_ado_req(dvca_priv_key, dvca_req, Botan::
   ASN1_Car(dvca_cert1.get_chr().iso_8859()));
```

The ADO is created from the request object it encapsulates, the private key to create the ADO signature, and the CAR. In the example code above, `dvca_cert1` is the certificate associated with the private key `dvca_priv_key`.

10.3.5 Signing DVCA requests

In order to sign a DVCA request with a CVCA, simply do the following:

```
Botan::EAC1_1_CVC dvca_cert1 = Botan::DE_EAC::
   sign_request(cvca_cert, cvca_private_key, dvca_req,
    1, 5, true);
```

Here, the first three parameters should be self-explanatory. The number `1` indicates the serial number of the request to be signed, whereas `5` indicates that the serial number will be encoded in 5 digits and appended to the CHR found in the request. The last boolean parameter indicates that this request is signed as a domestic DVCA.

10.3.6 Verifying and signing ADO requests

The following piece of code shows how to perform the verification of outer ADO signature. Assume that `dvca_ado` is the `EAC1_1_ADO` to be verified, and that `dvca_cert1` is the `EAC1_1_CVC` object that has signed the ADO.

```
auto_ptr<Botan::Public_Key> ap_pk = dvca_cert1.
    subject_public_key();
Botan::ECDSA_PublicKey* cert_pk = dynamic_cast<Botan::
    ECDSA_PublicKey*>(ap_pk.get());
cert_pk->set_domain_parameters(dom_pars);
dvca_ado.check_signature(*cert_pk)
```

Note that the function `ECDSA_PublicKey::set_domain_parameters()` is called. This is necessary whenever a certificate without explicit domain parameters is loaded.

10.3.7 IS requests and IS ADO requests

The creation, verification, and signing of IS related requests works analogously to the handling of DVCA requests. When letting a DVCA sign requests, it will automatically sign them as IS requests, just as CVCA automatically treats all requests as DVCA requests.

10.3.8 Writing a certificate file to disk in DER format

The following code shows how to write a certificate named `cvca_cert` to disk in DER format:

```
ofstream cvca_file("checks/temp/cvc_chain_cvca.cer",
    ios::binary);
Botan::SecureVector<Botan::byte> cvca_sv = cvca_cert.
    BER_encode();
cvca_file.write((char*)cvca_sv.begin(), cvca_sv.size()
    );
cvca_file.close();
```

10.3.9 Loading a DER encoded certificate file from disk

The following code shows how to decode a DER encoded CVC from the disk:

```
Botan::EAC1_1_CVC cert("checks/testdata/cvca01.cv.crt
    ");
```

The same constructors exist for requests and ADO requests.

Note that the public EC-DSA key of certificates can only be used when the domain parameters are set. After an encoded CVC without domain parameters is decoded, the certificate's public key cannot be used. In this case, first call the key's member function `ECDSA_PublicKey::set_domain_parameters()` and provide the associated domain parameters as the argument.