libscapi Documentation

Release 1.0

libscapi team

December 12, 2018

Contents

Libscapi is an open-source c++ library for implementing secure two-party and multiparty computation protocols (SCAPI stands for the "Secure Computation API"). It provides a reliable, efficient, and highly flexible cryptographic infrastructure. SCAPI also has a java version, that can be found at https://scapi.readthedocs.io/en/latest/index.html. SCAPI is free and is licensed under an adaptation of the MIT license, you can read more about the license *here*.

CHAPTER 1

Introduction

LibSCAPI is an *open-source* general library tailored for **Secure Computation** implementations. libscapi provides a flexible and efficient infrastructure for the implementation of secure computation protocols, that is both easy to use and robust. We hope that SCAPI will help to promote the goal of making secure computation practical.

Why Should I Use libscapi?

- **libscapi provides uniformity.** As of today, different research groups are using different implementions. It is hard to compare different results, and implementations carried out by one group cannot be used by others. libscapi is trying to solve this problem by offering a modular codebase to be used as the standard library for Secure Computation.
- **libscapi is flexible.** libscapi's lower-level primitives inherit from modular interfaces, so that primitives can be replaced easily. libscapi leaves the choice of which concrete primitives to actually use to the high-level application calling the protocol. This flexibility can be used to find the most efficient primitives for each specific problem.
- **libscapi is efficient.** libscapi is implemented in c++ and wraps the most efficient libraries and implementations in order to run more efficiently. For example, elliptic curve operations in libscapi are implemented using the extremely efficient Miracl library written in C.
- **libscapi is built to please.** libscapi has been written with the understanding that others will be using it, and so an emphasis has been placed on clean design and coding, documentation, and so on.

Architecture

libscapi is composed of the following four layers:

1. **Low-level primitives:** these are functions that are basic building blocks for cryptographic constructions (e.g., pseudorandom functions, pseudorandom generators, discrete logarithm groups, and hash functions belong to this layer).
2. **Non-interactive mid-level protocols**: these are non-interactive functions that can be applications within themselves in addition to being tools (e.g., encryption and signature schemes belong to this layer).
3. **Interactive mid-level protocols:** these are interactive protocols involving two or more parties; typically, the protocols in this layer are popular building blocks like commitments, zero knowledge and oblivious transfer.
4. **High level Protocols:** these are implementations of known cryptographic multi-party and two-party protocols. For example, Yao and GMW.

In addition to these four main layers, there is an orthogonal communication layer that is used for setting up communication channels and sending messages and also a circuit package that contains boolean circuits and garbled boolean circuit implementations used in libscapi's layers.

CHAPTER 2

Installation

Scapi is simple enough to install, the installation varies on different operating systems. Scapi currently supports Linux and Windows.

Installing LibSCAPI - Linux

The following explains how to install libscapi on Ubuntu. For other Linux variants it should work as well with the appropriate adjustments.

Prerequisites

Update and install git, gcc, gmp, and open ssl. On Ubuntu environment is should look like:

```
$ sudo apt-get update
$ sudo apt-get install -y git build-essential
$ sudo apt-get install -y libssl-ocaml-dev libssl-dev
$ sudo apt-get install -y libgmp3-dev
```

Download and install boost (the last step might take some time. patience):

```
$ wget -O boost_1_64_0.tar.bz2 http://sourceforge.net/projects/boost/files/boost/1.64.0/boost_1_64_0
$ tar --bzip2 -xf boost_1_64_0.tar.bz2
$ cd boost_1_64_0
$  ./bootstrap.sh
$  ./b2
```

More details about boost here: http://www.boost.org/doc/libs/1_64_0/more/getting_started/unix-variants.html

Building libscapi and publishing libs

Download and build libscapi:

```
$ cd ~
$ git clone https://github.com/cryptobiu/libscapi.git
$ cd libscapi
$ make
```

Publish new libs:

```
$ sudo ldconfig ~/boost_1_60_0/stage/lib/ ~/libscapi/install/lib/
```

Building and Running the Tests

In order to build and run tests:

```
$ cd ~/libscapi/test
$ make
$ ./tests.exe
```

Samples

Build and run the samples program:

```
$ cd ~/libscapi/samples
$ make
```

In order to see all available samples:

```
$ ./libscapi_example.exe
```

In order to run simple examples (dlog or sha1):

```
$ ./libscapi_example.exe dlog
$ ./libscapi_example.exe sha1
```

You should get some print outs if everything works well.

In order to run the CommExample. Open two terminals. In the first run:

```
$ ./libscapi_example.exe comm 1 Comm/CommConfig.txt
```

And in the other run:

```
$ ./libscapi_example.exe comm 2 Comm/CommConfig.txt
```

In order to run Semi-honset YAO, run in the first terminal:

```
$ ./libscapi_example.exe yao 1 Yao/YaoConfig.txt
```

And in the second:

```
$ ./libscapi_example.exe yao 2 Yao/YaoConfig.txt
```

Finally in order to run the Sigma example - in the first terminal run:

```
$ ./libscapi_example.exe sigma 1 SigmaProtototocls/SigmaConfig.txt
```

And in the second terminal:

```
$ ./libscapi_example.exe sigma 1 SigmaProtototocls/SigmaConfig.txt
```

You can edit the config file in order to play with the different params in all examples.

Installing LibSCAPI - Windows

Installing scapi on windows will require git client and Visual Studio IDE. We tested it with VS2015.

Prerequisites:

1. Download and install open ssl for windows: https://slproweb.com/products/Win32OpenSSL.html (choose 64bit not light)
2. Download and install boost binaries for windos: https://sourceforge.net/projects/boost/files/boost-binaries/1.60.0/ choose 64 bit version 14

The windows solutions assume that boost is installed at `C:\local\boost_1_60_0` and that OpenSSL at: `C:\OpenSSL-Win64`

Pull libscapi from GitHub. For convenient we will assume that libscapi is located at: `c:\code\scapi\libscapi`‵. If it is located somewhere eles then the following paths should be adjusted accrodingly.

1. **Build Miracl for windows 64:**
   (a) Open solution MiraclWin64.sln at: `C:\code\libscapi\lib\MiraclCompilation`
   (b) Build the solution once for debug and once for release
2. **Build OTExtension for window 64:**
   (a) Open solution OTExtension.sln at `C:\code\libscapi\lib\OTExtension\Win64-sln`
   (b) Build solution once for debug and once for release
3. **Build GarbledCircuit project**
   (a) Open solution ScGarbledCircuitWin64.sln at `C:\code\libscapi\lib\ScGarbledCircuit\ScGarbledCi`
   (b) Build solution once for debug and once for release
4. **Build the NTL solution:**
   (a) Open solution NTL-WIN64.sln at `C:\code\libscapi\lib\NTL\windows\NTL-WIN64`
   (b) Build solution once for debug and once for release
5. **Build Scapi Solution including examples and test:**
   (a) Open solution ScapiCpp.sln at `C:\code\libscapi\windows-solutions\scapi-sln`
   (b) Build solution once for debug and once for release - (as needed)
6. **Run tests.**
   (a) Go to `C:\code\libscapi\windows-solutions\scapi-sln\x64\debug`
   (b) run ./scapi_tests.exe and make sure all is green
7. **Run example:**
   (a) open two terminals
   (b) in both of them go to: `C:\code\libscapi\windows-solutions\scapi-sln\x64\debug`
   (c) To see available samples run `libscapi_examples.exe`
   (d) Follow instruction of how to run the different samples as exaplained in the linux section
   (e) You can edit the different config file to play with the paramaters

Quickstart

Eager to get started? This page gives a good introduction to Libscapi. It assumes you already have libscapi installed. If you do not, head over to the *Installation* section.

Your First libscapi Application

We begin with a minimal application and go through some basic examples.

```
#include "../../include/primitives/DlogOpenSSL.hpp"

int main(int argc, char* argv[]){
    // initiate a discrete log group
    // (in this case the OpenSSL implementation of the elliptic curve group K-233)
    DlogGroup* dlog = new OpenSSLDlogECF2m("include/configFiles/NISTEC.txt", "K-233");

    // get the group generator and order
    auto g = dlog->getGenerator();
    biginteger q = dlog->getOrder();

    // create a random exponent r
    shared_ptr<PrgFromOpenSSLAES> gen = get_seeded_prg();
    biginteger r = getRandomInRange(0, q - 1, gen.get());

    // exponentiate g in r to receive a new group element
    auto g1 = dlog->exponentiate(g.get(), r);
    // create a random group element
    auto h = dlog->createRandomElement();
    // multiply elements
    auto gMult = dlog->multiplyGroupElements(g1.get(), h.get());
}
```

Pay attention to the definition of the discrete log group. In libscapi we will always use a generic data type such as `DlogGroup` instead of a more specified data type. This allows us to replace the group to a different implementation or a different group entirely, without changing our code.

Let's break it down:

We include the libscapi primitive `OpenSSLDlogECF2m` class that extends the `DlogGroup` abstract class (implements a discrete log group). This is a wrapper class to an implementation of an elliptic curve group in the OpenSSL library. Since `DlogGroup` is abstract class, we can easily choose a different group without changing a single line of code except the one in emphasis.

We also use the get_seeded_prg() function implemented by libscapi, that returns an object of type PrgFromOpenSSlAES. This is a libscapi's class that provides a cryptographically pseudo random generator.

In order to handle big numbers we use the `biginteger` define that represents boost::multiprecision::mpz_int in linux systems and boost::multiprecision::cpp_int in windows.

```
#include "../../include/primitives/DlogOpenSSL.hpp"
```

Our main class defines a discrete log group, and then extract the group properties (generator and order).

```
// initiate a discrete log group
// (in this case the OpenSSL implementation of the elliptic curve group K-233
// using the NISTEC.txt file that provided by libscapi that is a at libscapi/include/configFiles)
DlogGroup* dlog = new OpenSSLDlogECF2m("include/configFiles/NISTEC.txt", "K-233");

// get the group generator and order
auto g = dlog->getGenerator();
biginteger q = dlog->getOrder();
```

We then choose a random exponent, and exponentiate the generator in this exponent.

```
// create a random exponent r
shared_ptr<PrgFromOpenSSLAES> gen = get_seeded_prg();
biginteger r = getRandomInRange(0, q - 1, gen.get());

// exponentiate g in r to receive a new group element
auto g1 = dlog->exponentiate(g.get(), r);
```

We then select another group element randomly.

```
// create a random group element
auto h = dlog->createRandomElement();
```

Finally, we demonstrate how to multiply group elements.

```
// multiply elements
auto gMult = dlog->multiplyGroupElements(g1.get(), h.get());
```

Compiling and Running the libscapi Code

Save this example to a file called *DlogExample.cpp*. In order to compile this file, type in the terminal:

```
$ g++ example.cpp -I/home/moriya -I/home/moriya/boost_1_60_0 -std=c++11 scapi.a -lboost_system -L/hor
```

Note that we use the scapi.a which is the libscapi lirary. The -I command sets the include files to use in the program and the -l command sets the libraries to link to the program.

A file called *a.out* should be created as a result. In order to run this file, type in the terminal:

```
$ ./a.out
```

Establishing Secure Communication

The first thing that needs to be done to obtain communication services is to setup the connections between the different parties. Libscapi provides two communication types - tcp communication and ssl tcp communication. The abstract communication class called `commParty` and the concrete classes are `CommPartyTCPSynced` and `CommPartyTcpSslSynced`. Both communication types use `boost::asio::io_service` in order to set communication between the parties.

Let's get a look at the following code:

```
#include <libscapi/include/comm/Comm.hpp>

int main(int argc, char* argv[]) {

    boost::asio::io_service io_service;
    SocketPartyData me, other;
    if (atoi(argv[1]) == 0){
            me = SocketPartyData(boost_ip::address::from_string("127.0.0.1"), 8000);
            other = SocketPartyData(boost_ip::address::from_string("127.0.0.1"), 8001);
    } else {
            me = SocketPartyData(boost_ip::address::from_string("127.0.0.1"), 8001);
            other = SocketPartyData(boost_ip::address::from_string("127.0.0.1"), 8000);
    }

    shared_ptr<CommParty> channel = make_shared<CommPartyTCPSynced>(io_service, me, other);
    // connect to party one
    channel->join(500, 5000);
    cout<<"channel established"<<endl;
}
```

In this example, we establish a communication between two parties in the same machine, using ports 8000 and 8001.

A `CommParty` represents an established connection between two parties. It has two main functions:

```
void write(const byte* data, int size)
```

Sends a message *data* to the other party, the number of bytes in *data* should be equal to *size*.

```
size_t read(byte* buffer, int sizeToRead)
```

Receives a message with *sizeToRead* bytes from the channel. The buffer should have at least sizeToRead bytes.

This means that from the applications point of view, once it obtains the channels it can completely forget about it and just send and receive messages.

CHAPTER 4

The Communication Layer

Contents

Communication Design

The communication layer provides communication services for any interactive cryptographic protocol. We have two types of communication, plain (unauthenticated and unencrypted) communication and secure channels using ssl. This layer is heavily used by the interactive protocols in libscapi' third layer and by MPC protocols. It can also be used by any other cryptographic protocol that requires communication. Currently the communication layer is a two-party communication channel. MultiParty communication can be achieved by setting a communication between each pair of parties.

Class hierarchy

The main communication clas is `CommParty`. This is an abstract class that declares all communication functionalities. There are two concrete classes that derive the `CommParty` class:

- `CommPartyTCPSynced` - establish a plain channel between the parties.
- `CommPartyTcpSslSynced` - establish an ssl channel between the parties.

Setting up communication

There are several steps involved in setting up a communication channel between parties. Each one of them will be explained below: First, let's take a look of an example for setting a cummunication between 3 parties:

```
#include <libscapi/include/comm/Comm.hpp>

int main(int argc, char* argv[]) {


    int numParties = 3;

    //open file
    ConfigFile cf("/home/moriya/libscapi/protocols/GMW/Parties");

    string portString, ipString;
    vector<int> ports(numParties);
    vector<string> ips(numParties);
    int counter = 0;
    for (int i = 0; i < numParties; i++) {
        portString = "party_" + to_string(i) + "_port";
        ipString = "party_" + to_string(i) + "_ip";
        //get partys IPs and ports data
        ports[i] = stoi(cf.Value("", portString));
        ips[i] = cf.Value("", ipString);
    }

    SocketPartyData me, other;
    boost::asio::io_service io_service;

    int id = atoi(argv[1]);
    for (int i=0; i<numParties; i++){
        if (i < id) {// This party will be the receiver in the protocol

            me = SocketPartyData(boost_ip::address::from_string(ips[id]), ports[id] + i);
            cout<<"my port = "<<ports[id] + i<<endl;
            other = SocketPartyData(boost_ip::address::from_string(ips[i]), ports[i] + id - 1);
            cout<<"other port = "<<ports[i] + id - 1<<endl;

            shared_ptr<CommParty> channel = make_shared<CommPartyTCPSynced>(io_service, me, other);
            // connect to party one
            channel->join(500, 5000);
            cout<<"channel established"<<endl;

        } else if (i>id) {// This party will be the sender in the protocol
            me = SocketPartyData(boost_ip::address::from_string(ips[id]), ports[id] + i - 1);
            cout<<"my port = "<<ports[id] + i - 1<<endl;
            other = SocketPartyData(boost_ip::address::from_string(ips[i]), ports[i] + id);
            cout<<"other port = "<< ports[i] + id<<endl;

            shared_ptr<CommParty> channel = make_shared<CommPartyTCPSynced>(io_service, me, other);
            // connect to party one
            channel->join(500, 5000);
            cout<<"channel established"<<endl;
        }
    }
}
```

Fetch the list of ips and ports

The first step towards obtaining communication services is to setup the connections between the different parties. In order start obtaining the communication, party should first get a list of the parties' ips and ports. Each pair of ip and port represents a party in the protocol. The ips and ports can be obtaind from a file or any other way. In the example above the reading from the file is done via ConfigFile wich is a libscapi's class that reads from a given file :

```
//open file
ConfigFile cf("/home/moriya/libscapi/protocols/GMW/Parties");

string portString, ipString;
vector<int> ports(numParties);
vector<string> ips(numParties);
int counter = 0;
for (int i = 0; i < numParties; i++) {
    portString = "party_" + to_string(i) + "_port";
    ipString = "party_" + to_string(i) + "_ip";
    //get partys IPs and ports data
    ports[i] = stoi(cf.Value("", portString));
    ips[i] = cf.Value("", ipString);
}
```

In the example, the parties file contains for each party in the protocol the ip and starting port number. The other port numbers are the next indices.

```
party_0_ip = 127.0.0.1
party_1_ip = 127.0.0.1
party_2_ip = 127.0.0.1
party_0_port = 8000
party_1_port = 8020
party_2_port = 8040
```

Setting up the actual communication

The actual communication is done by creating the channels and activate them. Once a channel has been activated, it can be used to write and read messages. Each channel communicates between two parties and uses a **single port** for each one of them. In order to create the channel, one should give the ips and ports of the parties on both channel's sides.

As we said before, the abstract communication class is `CommParty` and there are two concrete classes `CommPartyTCPSynced` and `CommPartyTcpSslSynced`. The constructors of the concrete classes are follow:

CommPartyTCPSynced(boost::asio::io_service& *ioService*, SocketPartyData *me*, SocketPartyData *other*)

CommPartyTcpSslSynced(boost::asio::io_service& *ioService*, SocketPartyData *me*, SocketPartyData *other*, string *certificateChainFile*, string *password*, string *privateKeyFile*, string *tmpDHFile*, string *clientVerifyFile*)

Parameters

- **out** – `boost::asio::io_service io_service` - Boost's object that used in the communication.
- **out** – `SocketPartyData me` - An object that contains the ip and the port of this party.
- **out** – `SocketPartyData other` - An object that contains the ip and the port of the party that we want to communicate with.

  `CommPartyTcpSslSynced` also accepts the parameters for the ssl protocol:
- **out** – string certificateChainFile
- **out** – string password
- **out** – string privateKeyFile

- **out** – string tmpDHFile
- **out** – string clientVerifyFile

After the channel has been creates, it needs to get activated. This is done by the `join` function of the channel:

void **join**(int *sleep_between_attempts*, int *timeout*)
This function setups a double edge connection with the the current party and the other party. The method blocks until both sides are connected to each other. In case of timeout, the communication fails and an error is thrown.

After the join function is complete, the channel is ready to send and receive messages.

In the example above the code that creates a channel and activate it is:

```
me = SocketPartyData(boost_ip::address::from_string(ips[id]), ports[id] + i);
cout<<"my port = "<<ports[id] + i<<endl;
other = SocketPartyData(boost_ip::address::from_string(ips[i]), ports[i] + id - 1);
cout<<"other port = "<<ports[i] + id - 1<<endl;

shared_ptr<CommParty> channel = make_shared<CommPartyTCPSynced>(io_service, me, other);
// connect to party one
channel->join(500, 5000);
```

First, we create a SocketPartyData for the current application with the ip and port. Second, we create a SocketPartyData for the other application and then we create the channel and activate it.

Using an established connection

A connection is represented by the `CommParty` interface. Once a channel is established, we can `write()` and `read()` data between parties. There are multiple write and read functions:

void **write**(const byte* *data*, int *size*)
Writes bytes from data to the other party. This function Will write exactly size bytes.

void **writeWithSize**(const byte* *data*, int *size*)
Writes the size of the data parameter, then writes the data itself.

size_t **read**(byte* *buffer*, int *sizeToRead*)
Reads exactly sizeToRead bytes and put them in buffer. This function Will block until all bytes are read.

There are also functions that working on strings and vectors:

void **write**(string *s*)

void **writeWithSize**(string *s*)

int **readSize**()

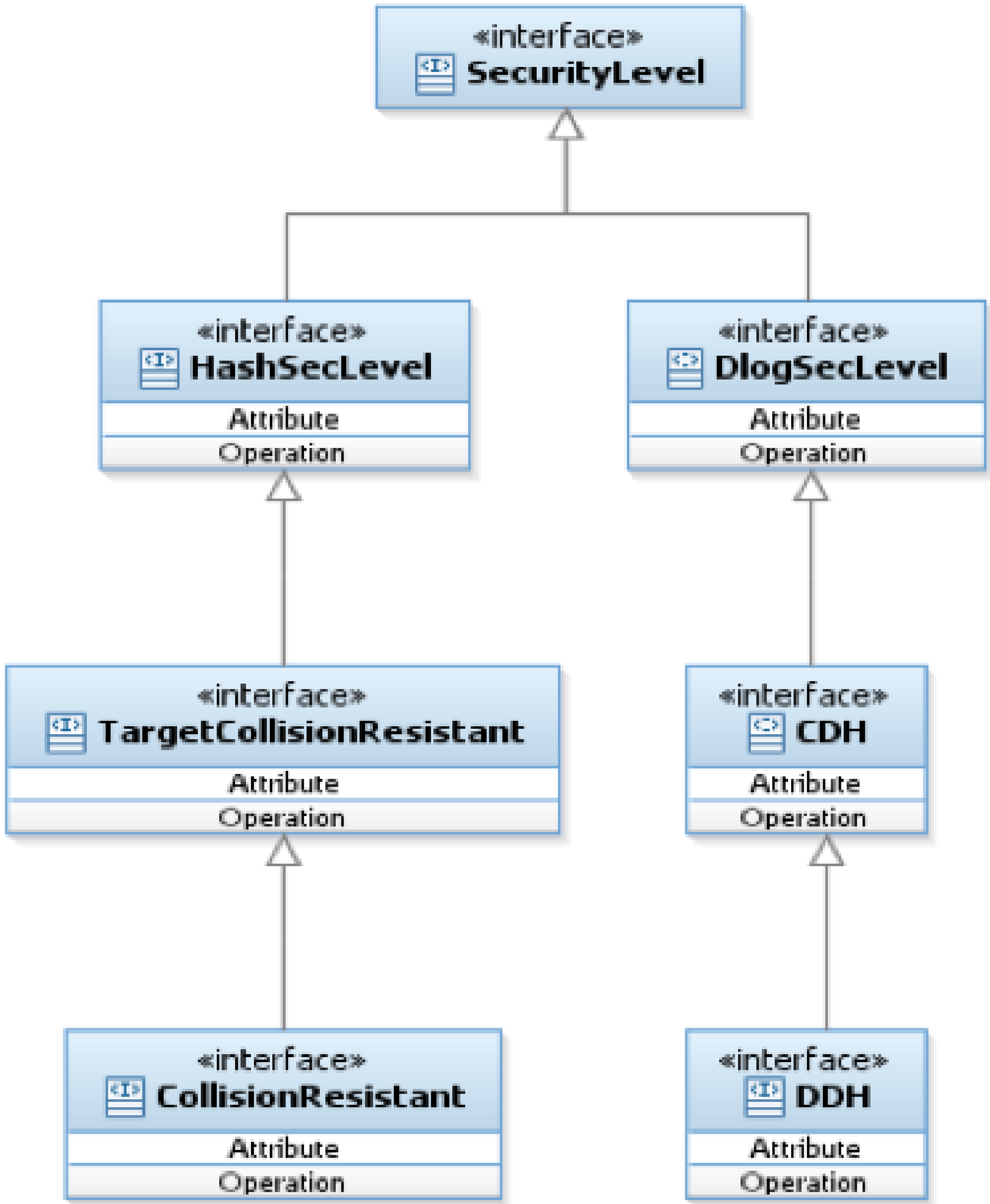
size_t **readWithSizeIntoVector**(vector<byte>& *targetVector*)

Security Levels

In many cases, a cryptographic primitive is not just "secure" or "insecure". Rather, it may meet some notion of security and not another. A classic example is encryption, where a scheme can be secure in the presence of eavesdropping adversaries (EAV), in the presence of chosen-plaintext attacks (CPA) or in the presence of chosen-ciphertext attacks (CCA). These three levels of security also form a hierarchy (any scheme that is secure in the presence of chosen-ciphertext attacks, is secure against chosen-plaintext attacks and so on). The choice of which level of security to require, depends on the application. We remark that it is not always wise to take the "most secure" scheme since this sometimes comes with a performance penalty. In addition, in some cases (like in commitments), the security levels are non-comparable.

Protocols that by definition need to work with primitives that hold a specific security level are responsible for checking that the primitives meet the security level requirements. For example, an encryption scheme that is secure under DDH should check that it receives a Dlog Group with security level `DDH`.

The library therefore includes a hierarchy of security level classes; These classes have no methods and are only markers. Each concrete class that is based on any security level should derive the relevant class, to declare itself as secure as the security level class.


«interface»
SecurityLevel
«interface»
HashSecLevel
Attribute
Operation
«interface»
DlogSecLevel
Attribute
Operation
«interface»
TargetCollisionResistant
Attribute
Operation
«interface»
CDH
Attribute
Operation
«interface»
CollisionResistant
Attribute
Operation
«interface»
DDH
Attribute
Operation

Circuits

Circuits are basic building block which often use in Scapi, espacially in MPC protocols.

Garbled Circuit refers the case of a circuit which we don't want the evaluator to know the values in the middle of the circuit evaluation. Meaning, each gate's output should tell nothing about its actual result. In order to get this goal we garble the circuit. In case of a Boolean circuit, each wire gets two garbled values (which we called "keys"). During the circuit computation, each gate outputs one of its output wire's garbled values. At the end, the circuit outputs garbled values for each output wire. In order to translate it to a meaningful result, one should call the translate function of the circuit.

All garbled circuits have four main operations:

1. The garble function that creates the garbled table.
2. The compute function that computes a result on a garbled circuit on the input which is the keys that were chosen for each input wire.
3. The verify method is used in the case of a malicious adversary to verify that the garbled circuit created is an honest garbling of the agreed upon non garbled circuit.
4. The translate method that translates the garbled output which usually is generated by the compute() function into meaningful output (a 0/1 result, rather than the keys outputed by compute).

Create the circuit

The best way to create a circuit is using a file. There are three formats of circuits that are quite similar:

Two-Party Boolean circuit

The format of the circuit file should be as follows.

1. Number of gates
2. Number of parties
3. For each party:

- Party number
- Number of inputs for that party
- A list of integer labels of each of these input wires.

4. Number of output wires

5. List of integer labels of each of these output wires.
6. For each gate:

- Number of input wires
- Number of output wires
- Input wires labels
- Output Wires labels
- Truth table (as a 0-1 string).

An example file:

```
1                // One gate
2                // Two parties
1 1 0            // Party one, has one input wire, labeled "0"
2 1 1            // Party two, has one input wire, labeled "1"
1                // One output wire
2                // The output wire, labeled "2"
2 1 0 1 2 0001   // The first (and only) gate has 2 input wire labeled "0" and "1", one output wire la
```

Multi-Party Boolean circuit

The format of the circuit file should be as follows.

1. Number of gates
2. Number of parties
3. For each party:

- Party number
- Number of inputs for that party
- A list of integer labels of each of these input wires.

4. For each party:

- Party number
- Number of outputs for that party
- A list of integer labels of each of these output wires.

5. For each gate:

- Number of input wires
- Number of output wires
- Input wires labels
- Output Wires labels
- Truth table (as a 0-1 string).

An example file:

```
1                // One gate
2                // Two parties
1 1 0            // Party one, has one input wire, labeled "0"
2 1 1            // Party two, has one input wire, labeled "1"
```

```
1 1 0          // Party one has no output wires
2 1 2          // Party two has one output wire, labeled "2"
2 1 0 1 2 0001 // The first (and only) gate has 2 input wire labeled "0" and "1", one output wire la
```

Multi-Party Arithmetic circuit

The format of the circuit file should be as follows.

1. Number of gates
2. Number of parties
3. For each party:

- Party number
- Number of inputs for that party
- A list of integer labels of each of these input wires.

4. For each party:

- Party number
- Number of outputs for that party
- A list of integer labels of each of these output wires.

5. For each gate:

- Number of input wires
- Number of output wires
- Input wires labels
- Output Wires labels
- A number that indicates the circuit type, see table below.

The available gates types are listed in the next table:

Gate type	Number
ADD	1
MULT	2
SCALAR MULTIPLICATION	5
SUBTRACT	6
SCALAR ADD	7

An example file:

```
1              // One gate
2              // Two parties
1 1 0          // Party one, has one input wire, labeled "0"
2 1 1          // Party two, has one input wire, labeled "1"
1 1 0          // Party one has no output wires
2 1 2          // Party two has one output wire, labeled "2"
2 1 0 1 2 1    // The first (and only) gate has 2 input wire labeled "0" and "1", one output wire la
```

CHAPTER 7

Layer 1: Basic Primitives

Cryptographic Hash

A **cryptographic hash** function is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string, the (cryptographic) hash value. There are two main levels of security that we will consider here:

- **target collision resistance:** meaning that given x it is hard to find y such that $H(y) = H(x)$.
- **collision resistance:** meaning that it is hard to find any x and y such that $H(x) = H(y)$.

Note: We do not include **preimage resistance** since cryptographically this is just a one-way function.

Contents

The `CryptographicHash` abstract class

The user may request to pass partial data to the hash and only after some iterations to obtain the hash of all the data. This is done by calling the function `update()`. After the user is done updating the data it can call the `hashFinal()` to obtain the hash output.

void **update**(const vector<byte>& *in*, int *inOffset*, int *inLen*)

Adds the vector to the existing msg to hash.

Parameters

- **in** – input vector
- **inOffset** – the offset within the vector
- **inLen** – the length. The number of bytes to take after the offset

void **hashFinal**(vector<byte>& *out*, int *outOffset*)

Completes the hash computation.

Parameters

- **out** – the output in vector

- **outOffset** – the offset which to put the result bytes from

Usage

Below is an example of using Cryptographic hash:

```
//create an input array in and an output array out
...

//create  an OpenSSL sha224 function.
CryptographicHash* hash = new OpenSSLSHA224();

//call the update function in the Hash interface.
hash->update(in, 0, in.length);

//get the result of hashing the updated input.
hash->hashFinal(out, 0);
```

Supported Hash Types

In this section we present the hash functions provided by libscapi.

The OpenSSL implementation:

Class Name	Class Location
OpenSSLSHA1	libscapi/include/primitives/hashOpenSSL.hpp
OpenSSLSHA224	libscapi/include/primitives/hashOpenSSL.hpp
OpenSSLSHA256	libscapi/include/primitives/hashOpenSSL.hpp
OpenSSLSHA384	libscapi/include/primitives/hashOpenSSL.hpp
OpenSSLSHA512	libscapi/include/primitives/hashOpenSSL.hpp

The Blake2 implementation:

Class Name	Class Location
Blake2SHA1	libscapi/include/primitives/hashBlake2.hpp
Blake2SHA224	libscapi/include/primitives/hashBlake2.hpp
Blake2SHA256	libscapi/include/primitives/hashBlake2.hpp
Blake2SHA384	libscapi/include/primitives/hashBlake2.hpp
Blake2SHA512	libscapi/include/primitives/hashBlake2.hpp

Pseudorandom Function (PRF)

In cryptography, a **pseudorandom function family**, abbreviated **PRF**, is a collection of efficiently-computable functions which emulate a random function in the following way: no efficient algorithm can distinguish (with significant advantage) between a function chosen randomly from the PRF family and a random oracle (a function whose outputs are fixed completely at random).

Contents

The `PseudorandomFunction` abstract class

The main function of this class is `computeBlock()`. We supply several versions for compute, with and without length. Since both PRP's and PRF's may have varying input/output length, for such algorithms the length should be supplied. We provide the version without the lengths and not just the versions with length of input and output, although it suffices, to avoid confusion and misuse from a basic user that only knows how to use block ciphers. A user that uses the block cipher TripleDES, may be confused by the "compute with length" functions since TripleDES has a pre-defined length and it cannot be changed.

Block Manipulation

void `PseudorandomFunction::`**`computeBlock`**(const vector<byte>& *inBytes*, int *inOff*, vector<byte>& *outBytes*, int *outOff*)

Computes the function using the secret key. The user supplies the input vector and the offset from which to take the data from. The user also supplies the output vector as well as the offset. The computeBlock function will put the output in the output vector starting at the offset. This function is suitable for block ciphers where the input/output length is known in advance.

Parameters

- **inBytes** – input bytes to compute
- **inOff** – input offset in the inBytes array
- **outBytes** – output bytes. The resulted bytes of compute
- **outOff** – output offset in the outBytes array to put the result from

void `PseudorandomFunction::`**`computeBlock`**(const vector<byte>& *inBytes*, int *inOff*, int *inLen*, vector<byte>& *outBytes*, int *outOff*, int *outLen*)

Computes the function using the secret key. This function is provided in the abstract class especially for the sub-family PrfVaryingIOLength, which may have variable input and output length. If the implemented algorithm is a block cipher then the size of the input as well as the output is known in advance and the use may call the other computeBlock function where length is not require.

Parameters

- **inBytes** – input bytes to compute
- **inOff** – input offset in the inBytes vector
- **inLen** – the length of the input vector
- **outBytes** – output bytes. The resulted bytes of compute
- **outOff** – output offset in the outBytes vector to put the result from
- **outLen** – the length of the output vector

void `PseudorandomFunction::computeBlock`(const vector<byte>& *inBytes*, int *inOffset*, int *inLen*, vector<byte>& *outBytes*, int *outOffset*)

Computes the function using the secret key.

This function is provided in this PseudorandomFunction abstract class for the sake of classes for which the input length can be different for each computation. Hmac and Prf/Prp with variable input length are examples of such classes.

Parameters

- **inBytes** – input bytes to compute
- **inOffset** – input offset in the inBytes vector
- **inLen** – the length of the input vector
- **outBytes** – output bytes. The resulted bytes of compute.
- **outOffset** – output offset in the outBytes vector to put the result from

int `PseudorandomFunction::getBlockSize`()

Returns the input block size in bytes

Setting the Secret Key

SecretKey `PseudorandomFunction::generateKey`(AlgorithmParameterSpec& *keyParams*)

Generates a secret key to initialize this prf object.

Parameters keyParams algorithmParameterSpec contains the required parameters for the key generation

Returns the generated secret key

SecretKey `PseudorandomFunction::generateKey`(int *keySize*)

Generates a secret key to initialize this prf object.

Parameters keySize is the required secret key size in bits

Returns the generated secret key

bool `PseudorandomFunction::isKeySet`()

An object trying to use an instance of prf needs to check if it has already been initialized.

Returns true if the object was initialized by calling the function setKey.

void `PseudorandomFunction::setKey`(SecretKey& *secretKey*)

Sets the secret key for this prf. The key can be changed at any time.

Parameters secretKey secret key

Basic Usage

```
//Create secretKey and in, in2, out vectors
...

// create a PRF of type TripleDES using openssl library
PseudorandomFunction* prf = new OpenSSLTripleDES();

//set the key
prf->setKey(secretKey);
```

```
//compute the function with input in and output out.
prf->computeBlock(in, 0, out, 0);
```

Pseudorandom Function with Varying Input-Output Lengths

A pseudorandom function with varying input/output lengths does not have pre-defined input and output lengths. The input and output length may be different for each compute function call. The length of the input as well as the output is determined upon user request. The class `IteratedPrfVarying` implements this functionality using an inner PRF that must implement the `PrfVaryingInputLength` abstract class. An example for such PRF is `Hmac`.

How to use the Varying Input-Output Length PRF

```
//Create secret key and in, out byte vectors
...

//create the Prf varying.
PseudorandomFunction* prf = new IteratedPrfVarying(make_shared<OpenSSLHMAC>());

//set the key
prf->setKey(secretKey);

//compute the function with input in of size 10 and output out of size 20.
prf->computeBlock(in, 0, 10, out, 0, 20);
```

Supported Prf Types

In this section we present the prf functions provided by libscapi.

Class Name	Class Location
IteratedPrfVarying	libscapi/include/primitives/Prf.hpp
LubyRackoffPrpFromPrfVarying	libscapi/include/primitives/Prf.hpp

The OpenSSL implementation:

Class Name	Class Location
OpenSSLHMAC	libscapi/include/primitives/PrfOpenSSL.hpp

Pseudorandom Permutation (PRP)

Pseudorandom permutations are bijective pseudorandom functions that are *efficiently invertible*. As such, they are of the pseudorandom function type and their input length always equals their output length. In addition (and unlike general pseudorandom functions), they are efficiently invertible.

The `PseudorandomPermutation` abstract class

The `PseudorandomPermutation` class derives the `PseudorandomFunction` abstract class, and adds the following functionality.

void `PseudorandomPermutation::`**`invertBlock`**(const vector<byte>& *inBytes*, int *inOff*, vector<byte>& *outBytes*, int *outOff*)

Inverts the permutation using the given key.

This function is a part of the PseudorandomPermutation class since any PseudorandomPermutation must be efficiently invertible (given the key). For block ciphers, for example, the length is known in advance and so there is no need to specify the length.

Parameters

- **inBytes** – input bytes to invert.
- **inOff** – input offset in the inBytes vector
- **outBytes** – output bytes. The resulted bytes of invert
- **outOff** – output offset in the outBytes vector to put the result from

void `PseudorandomPermutation::`**`invertBlock`**(const vector<byte>& *inBytes*, int *inOff*, vector<byte>& *outBytes*, int *outOff*, int *len*)

Inverts the permutation using the given key.

Since PseudorandomPermutation can also have varying input and output length (although the input and the output should be the same length), the common parameter `len` of the input and the output is needed.

Parameters

- **inBytes** – input bytes to invert.
- **inOff** – input offset in the inBytes vector
- **outBytes** – output bytes. The resulted bytes of invert
- **outOff** – output offset in the outBytes vector to put the result from
- **len** – the length of the input and the output

Basic Usage

```
//Create secretKey and in, out, inv vectors
...

//create the prp object
PseudorandomPermutation* prp = new OpenSSLAES();

//set the key
prp->setKey(secretKey);

//run the permutation on a block-size prefix of in
prp->computeBlock(in, 0, out, 0);

//invert the permutation
prp->invertBlock(out, 0, inv, 0);
```

Pseudorandom Permutation with Varying Input-Output Lengths

A pseudorandom permutation with varying input/output lengths does not have pre-defined input/output lengths. The input and output length (that must be equal) may be different for each function call. The length of the input/output is determined upon user request.

We implement the Luby-Rackoff algorithm as an example of PRP with varying I/O lengths. The class that implements the algorithm is `LubyRackoffPrpFromPrfVarying`.

How to use the Varying Input-Output Length PRP

```
//Create secretKey and in, out vectors
...

//create the prp object
PseudorandomPermutation* prp = new LubyRackoffPrpFromPrfVarying();

//set the key
prp->setKey(secretKey);

//invert the permutation with input in and output out of common size 20.
prp->invertBlock(in, 0, out, 0, 20);
```

Supported Prp Types

In this section we present the prp functions provided by libscapi.

The OpenSSL implementation:

Class Name	Class Location
OpenSSLAES	libscapi/include/primitives/PrfOpenSSL.hpp
OpenSSLTripleDes	libscapi/include/primitives/PrfOpenSSL.hpp

Pseudorandom Generator (PRG)

A **pseudorandom generator (PRG)** is a deterministic algorithm that takes a "short" uniformly distributed string, known as *the seed*, and outputs a longer string that cannot be efficiently distinguished from a uniformly distributed string of that length.

The `PseudorandomGenerator` abstract class

The main function of this class is `getPrgBytes()`. It streams the prg bytes and return the reauired amount of pseudo random bytes:

void `PseudorandomGenerator::`**`getPRGBytes`**(vector<byte>& *outBytes*, int *outOffset*, int *outlen*)
Streams the prg bytes.

Parameters

- **outBytes** – output bytes. The result of streaming the bytes.
- **outOffset** – output offset
- **outlen** – the required output length

Setting the Secret Key

SecretKey `PseudorandomGenerator::`**`generateKey`**(AlgorithmParameterSpec& *keyParams*)
Generates a secret key to initialize this prg object.

Parameters keyParams algorithmParameterSpec contains the required parameters for the key generation

Returns the generated secret key

SecretKey `PseudorandomGenerator::generateKey(int keySize)`
Generates a secret key to initialize this prg object.

Parameters keySize is the required secret key size in bits

Returns the generated secret key

bool `PseudorandomGenerator::isKeySet()`
An object trying to use an instance of prg needs to check if it has already been initialized.

Returns true if the object was initialized by calling the function setKey.

void `PseudorandomGenerator::setKey(SecretKey& secretKey)`
Sets the secret key for this prg. The key can be changed at any time.

Parameters secretKey secret key

Basic Usage

```
//Create secret key and out byte vector
...

//Create a prg
PseudorandomGenerator* prg = new PrgFromOpenSSLAES();
SecretKey secretKey = prg->generateKey(256); //256 is the key size in bits.

//set the key
prg->setKey(secretKey);

//get PRG bytes. The caller is responsible for allocating the out array.
//The result will be put in the out array.
prg->getPRGBytes(out.length, out);
```

Supported Prg Types

In this section we present the prg functions provided by libscapi.

Class Name	Class Location
ScPrgFromPrf	libscapi/include/primitives/Prg.hpp
PrgFromOpenSSLAES	libscapi/include/primitives/Prf.hpp

The OpenSSL implementation:

Class Name	Class Location
OpenSSLRC4	libscapi/include/primitives/Prg.hpp

Trapdoor Permutation

A trapdoor permutation is a bijection (1-1 and onto function) that is easy to compute for everyone, yet is hard to invert unless given special additional information, called the “trapdoor”. The public key is essentially the function description and the private key is the trapdoor.

Contents

The `TPElement` abstract class

The `TPElement` class represents a trapdoor permutation element.

biginteger `TPElement::`**`getElement`**`()`

Returns the trapdoor element value as bigInteger.

Returns the value of the element

The `TrapdoorPermutation` abstract class

This class is the general class of trapdoor permutation.

Core Functionality

shared_ptr<TPElement> `TrapdoorPermutation::`**`compute`**`(`TPElement* *tpEl*`)`

Computes the operation of this trapdoor permutation on the given TPElement.

Parameters tpEl the input for the computation

Returns the result TPElement from the computation

shared_ptr<TPElement> `TrapdoorPermutation::`**`invert`**`(`TPElement* *tpEl*`)`

Inverts the operation of this trapdoor permutation on the given TPElement.

Parameters tpEl the input to invert

Returns the result TPElement from the invert operation

byte `TrapdoorPermutation::`**`hardCorePredicate`**`(`TPElement* *tpEl*`)`

Computes the hard core predicate of the given tpElement.

A hard-core predicate of a one-way function f is a predicate b (i.e., a function whose output is a single bit) which is easy to compute given x but is hard to compute given $f(x)$. In formal terms, there is no probabilistic polynomial time algorithm that computes $b(x)$ from $f(x)$ with probability significantly greater than one half over random choice of x.

Parameters tpEl the input to the hard core predicate

Returns (byte) the hard core predicate.

vector<byte> `TrapdoorPermutation::`**`hardCoreFunction`**`(`TPElement* *tpEl*`)`

Computes the hard core function of the given tpElement.

A hard-core function of a one-way function f is a function g which is easy to compute given x but is hard to compute given $f(x)$. In formal terms, there is no probabilistic polynomial time algorithm that computes $g(x)$ from $f(x)$ with probability significantly greater than one half over random choice of x.

Parameters tpEl the input to the hard core function

Returns byte[] the result of the hard core function

Generating TPElements

shared_ptr<TPElement> `TrapdoorPermutation::`**`generateRandomTPElement`**`()`

creates a random TPElement that is valid for this trapdoor permutation

Returns the created random element

shared_ptr<TPElement> `TrapdoorPermutation::`**`generateUncheckedTPElement`**`(`const biginteger& *x*`)`

Creates a TPElement from a specific value x. This function does not guarantee that the the returned `TPElement` object is valid. It is the caller's responsibility to pass a legal x value.

Returns Set the x value and return the created random element

Checking Element Validity

TPElValidity `TrapdoorPermutation::`**`isElement`**`(`TPElement* *tpEl*`)`

Checks if the given element is valid for this trapdoor permutation

Parameters tpEl the element to check

Returns (TPElValidity) enum number that indicate the validation of the element

Throws IllegalArgumentException if the given element is invalid for this permutation

Encryption Keys Functionality

void **`setKey`**`(`const shared_ptr<PublicKey>& *publicKey*, const shared_ptr<PrivateKey>& *privateKey*`)`

Sets this trapdoor permutation with public key and private key.

Parameters

- **publicKey** – the public key
- **privateKey** – the private key that without it the permutation cannot be inverted efficiently. If the private key is not given, the object can compute but canot invert.

bool **`isKeySet`**`()`

Checks if this trapdoor permutation object has been previously initialized. To initialize the object the `setKey()` function has to be called with corresponding parameters after construction.

return `true` if the object was initialized, `false` otherwise.

shared_ptr<PublicKey> **`getPubKey`**`()`

Returns returns the public key

BasicUsage

We demonstrate a basic usage scenario with a sender party that wish to hide a secret using the trapdoor permutation, and a receiver who is not able to invert the permutation on the secret.

Here is the code of the sender:

```
//Create public key, private key and secret
...

//instantiate the rsa permutation using the openssl library:
OpenSSLRSAPermutation trapdoorPermutation;
//set the keys for this trapdoor permutation
trapdoorPermutation.setKey(publicKey, privateKey);

// represent the secret (originally was of BigInteger type) using TPElement
TPElement secretElement = trapdoorPermutation.generateTPElement(secret);
//hide the secret using the trapdoor permutation
TPElement maskedSecret = trapdoorPermutation.compute(secretElement);

// this line will succeed, because the private key is known to the sender
TPElement invertedElement = trapdoorPermutation.invert(maskedSecret);

// send the public key and the secret to the other side
...
```

Here is the code of the receiver:

```
// receive public key and secretMsg
...

//instantiate the rsa permutation using the openssl library:
OpenSSLRSAPermutation trapdoorPermutation;
//set the keys for this trapdoor permutation
trapdoorPermutation.setKey(publicKey);

// reconstruct a TPElement from a biginteger
TPElement maskedSecret = trapdoorPermutation.generateTPElement(secretMsg);

// this line will fail, because the private key is not known to the receiver
TPElement secretElement = trapdoorPermutation.invert(maskedSecret);
```

Supported Trapdoor Permutations

In this section we present the trapddor permutations provided by libscapi.

OpenSSL implementation of RSA trapdoor permutation:

Key	Class Location
OpenSSLRSAPermutation	libscapi/include/primitives/TrapdoorPermutationOpenSSL.hpp

Discrete Log Group (DLOG)

The **discrete logarithm problem** is as follows: given a generator g of a finite group G and a random element $h \in G$, find the (unique) integer x such that $g^x = h$. In cryptography, we are interested in groups for which the discrete logarithm problem (Dlog for short) is assumed to be hard (or other discrete-log type assumptions like **CDH** and

DDH). The two most common classes are a prime subgroup of the group Z_p^* for a large p, and some Elliptic curve groups.

We provide the implementation of the most important Dlog groups in cryptography (see diagram below):

- Z_p^*
- Elliptic curve over the field $GF[2^m]$
- Elliptic curve over the field Z_p

Although Elliptic curves groups look very different, the discrete log problem over them can be described as follows. Given an elliptic curve E over a finite field F, a base point on that curve P (i.e., a generator of the group defined from the curve), and a random point Q on the curve, the problem is to find the integer n such that $nP = Q$.

We have currently incorporated the elliptic curves recommended by NIST.

Contents

Class Hierarchy:

The root of the family is a general Dlog Group that presents functionality that all Dlog Groups should implement.

At the second level we encounter three interfaces:

1. PrimeOrderSubGroup: The order q of the group must be a prime.
2. DlogZp: Dlog Group over the Z_p^* field.
3. DlogEllipticCurve: Any elliptic curve.

At the third level we have:

1. DlogZpSafePrime: The order q is not only a prime but also is such that prime $p = 2 * q + 1$.
2. DlogEcFp: Any elliptic curve over F_p.
3. DlogEcF2m: Any elliptic curve over $F_2[m]$.

All these are general interfaces. Specifically, we implement Dlog Groups that are of prime order; therefore all the concrete classes presented here implement this interface. Other implementations may choose to add Dlog Groups that are not of prime order, and they are at liberty of doing so. They just need not to declare that they implement the PrimeOrderSubGroup interface.

We also see in the diagram two other interfaces that are **used** by DlogGroup:

1. GroupParams.
2. GroupElement.

The `DlogGroup` abstract class

Group Parameters

shared_ptr<GroupElement> `DlogGroup::`**`getGenerator`**`()`

The generator g of the group is an element of the group such that, when written multiplicatively, every element of the group is a power of g.

Returns the generator of this Dlog group

shared_ptr<GroupElement> `DlogGroup::`**`createRandomGenerator`**`()`

Creates a random generator of this Dlog group

Returns the random generator

biginteger `DlogGroup::`**`getOrder`**`()`

Returns the order of this Dlog group

shared_ptr<GroupParams> `DlogGroup::`**`getGroupParams`**`()`

GroupParams is a structure that holds the actual data that makes this group a specific Dlog group. For example, for a Dlog group over Zp* what defines the group is p.

Returns the GroupParams of that Dlog group

string `DlogGroup::`**`getGroupType`**`()`

Each concrete class implementing this interface returns a string with a meaningful name for this type of Dlog group. For example: “elliptic curve over F2m” or “Zp*”

Returns the name of the group type

shared_ptr<GroupElement> `DlogGroup::`**`getIdentity`**`()`

Returns the identity of this Dlog group

Exponentiation

shared_ptr<GroupElement> `DlogGroup::`**`exponentiate`**`(`GroupElement* *base*, const biginteger& *exponent*`)`

Raises the base GroupElement to the exponent. The result is another GroupElement.

Returns the result of the exponentiation

shared_ptr<GroupElement> `DlogGroup::`**`exponentiateWithPreComputedValues`**`(`const shared_ptr<GroupElement>& *base*, const biginteger& *exponent*`)`

Computes the product of several exponentiations of the same base and distinct exponents. An optimization is used to compute it more quickly by keeping in memory the result of h1, h2, h4,h8,... and using it in the calculation.

Note that if we want a one-time exponentiation of h it is preferable to use the basic exponentiation function since there is no point to keep anything in memory if we have no intention to use it.

Returns the exponentiation result

void DlogGroup::**endExponentiateWithPreComputedValues**(const shared_ptr<GroupElement>& *base*)

This function cleans up any resources used by exponentiateWithPreComputedValues for the requested base. It is recommended to call it whenever an application does not need to continue calculating exponentiations for this specific base.

shared_ptr<GroupElement> DlogGroup::**simultaneousMultipleExponentiations**(vector<shared_ptr<GroupElement>>& *groupElements*, vector<biginteger>& *exponentiations*)

Computes the product of several exponentiations with distinct bases and distinct exponents. Instead of computing each part separately, an optimization is used to compute it simultaneously.

Parameters

- **groupElements** – vector of base elements to exponentiate
- **exponentiations** – vector of exponents

Returns the exponentiation result

Multiplication and Inverse

shared_ptr<GroupElement> DlogGroup::**getInverse**(GroupElement* *groupElement*)

Calculates the inverse of the given GroupElement.

Parameters groupElement to invert

Returns the inverse element of the given GroupElement

shared_ptr<GroupElement> DlogGroup::**multiplyGroupElements**(GroupElement* *groupElement1*, GroupElement* *groupElement2*)

Multiplies two GroupElements

Returns the multiplication result

Group Element Generation

shared_ptr<GroupElement> DlogGroup::**createRandomElement**()

Creates a random member of this Dlog group

Returns the random element

shared_ptr<GroupElement> DlogGroup::**generateElement**(bool *bCheckMembership*, vector<biginteger>& *values*)

This function allows the generation of a group element by a protocol that holds a Dlog Group but does not know if it is a Zp Dlog Group or an Elliptic Curve Dlog Group. It receives the possible values of a group element and whether to check membership of the group element to the group or not.

It may be not necessary to check membership if the source of values is a trusted source (it can be the group itself after some calculation). On the other hand, to work with a generated group element that is not really an element in the group is wrong. It is up to the caller of the function to decide if to check membership or not. If bCheckMembership is false always generate the element. Else, generate it only if the values are correct.

Parameters

- **bCheckMembership** –
- **values** –

Returns the generated GroupElement

Validation

bool `DlogGroup::`**`isGenerator`**`()`

Checks if the element set as the generator is indeed the generator of this group.

Returns `true` if the generator is valid, `false` otherwise.

bool `DlogGroup::`**`isMember`**`(`GroupElement* *element*`)`

Checks if the given element is a member of this Dlog group

Parameters element possible group element for which to check that it is a member of this group

Returns `true` if the given element is a member of this group, `false` otherwise.

bool `DlogGroup::`**`validateGroup`**`()`

Checks parameters of this group to see if they conform to the type this group is supposed to be.

Returns `true` if valid, `false` otherwise.

Group Classification

bool `DlogGroup::`**`isOrderGreaterThan`**`(`int *numBits*`)`

Checks if the order of this group is greater than 2^numBits

Returns `true` if the order is greater than 2^numBits, `false` otherwise.

bool `DlogGroup::`**`isPrimeOrder`**`()`

Checks if the order is a prime number

Returns true if the order is a prime number, false otherwise.

Group Element Serialization

shared_ptr<GroupElement> `DlogGroup::`**`reconstructElement`**`(`bool *bCheckMembership*, GroupElementSendableData* *data*`)`

Reconstructs a GroupElement given the GroupElementSendableData data, which might have been received through a Channel open between the party holding this DlogGroup and some other party.

Parameters

- **bCheckMembership** – whether to check that the data provided can actually reconstruct an element of this DlogGroup. Since this action is expensive it should be used only if necessary.
- **data** – the GroupElementSendableData from which we wish to "reconstruct" an element of this DlogGroup

Returns the reconstructed GroupElement

Byte Array Encoding

shared_ptr<GroupElement> `DlogGroup::encodeByteArrayToGroupElement(const vector<unsigned char>& binaryString)`

This function takes any string of length up to k bytes and encodes it to a Group Element. k can be obtained by calling getMaxLengthOfByteArrayForEncoding() and it is calculated upon construction of this group; it depends on the length in bits of p.

The encoding-decoding functionality is not a bijection, that is, it is a 1-1 function but **is not onto**. Therefore, any string of length in bytes up to k can be encoded to a group element but not every group element can be decoded to a binary string in the group of binary strings of length up to 2^k.

Thus, the right way to use this functionality is first to encode a byte array and then to decode it, and not the opposite.

Parameters binaryString the byte array to encode

Returns the encoded group Element **or null** if the string could not be encoded

const vector<unsigned char> `DlogGroup::decodeGroupElementToByteArray(GroupElement* groupElement)`

This function decodes a group element to a byte array. This function is guaranteed to work properly **ONLY** if the group element was obtained as a result of encoding a binary string of length in bytes up to k.

This is because the encoding-decoding functionality is not a bijection, that is, it is a 1-1 function but **is not onto**. Therefore, any string of length in bytes up to k can be encoded to a group element but not any group element can be decoded to a binary sting in the group of binary strings of length up to 2^k.

Parameters groupElement the element to decode

Returns the decoded byte array

int `DlogGroup::getMaxLengthOfByteArrayForEncoding()`

This function returns the value *k* which is the maximum length of a string to be encoded to a Group Element of this group. Any string of length *k* has a numeric value that is less than (p-1)/2. *k* is the maximum length a binary string is allowed to be in order to encode the said binary string to a group element and vice-versa. If a string exceeds the *k* length it cannot be encoded.

Returns k the maximum length of a string to be encoded to a Group Element of this group. k can be zero if there is no maximum.

const vector<byte> `DlogGroup::mapAnyGroupElementToByteArray(GroupElement* groupElement)`

This function maps a group element of this dlog group to a byte array. This function does not have an inverse function, that is, it is not possible to re-construct the original group element from the resulting byte array.

Returns a byte array representation of the given group element

The `GroupElement` abstract class

shared_ptr<GroupElementSendableData> `GroupElement::generateSendableData()`

This function is used when a group element needs to be sent via a channel or any other means of sending data. It retrieves all the data needed to reconstruct this Group Element at a later time and/or in a different VM. It puts all the data in an instance of the relevant class that implements the GroupElementSendableData interface.

Returns the GroupElementSendableData object

bool `GroupElement::isIdentity()`

checks if this element is the identity of the group.

Returns `true` if this element is the identity of the group, `false` otherwise.

The `GroupParams` class

biginteger `GroupParams::getQ()`

Returns the group order q

Basic Usage

```
// initiate a discrete log group (in this case the OpenSSL implementation of the elliptic curve grou
DlogGroup* dlog = new OpenSSLDlogECF2m("include/configFiles/NISTEC.txt", "K-233");

// get the group generator and order
shared_ptr<GroupElement> g = dlog->getGenerator();
biginteger q = dlog->getOrder();
auto random = get_seeded_prg();

// create a random exponent r
biginteger r = getRandomInRange(0, q - 1, random.get());
// exponentiate g in r to receive a new group element
shared_ptr<GroupElement> g1 = dlog->exponentiate(g.get(), r);
// create a random group element

shared_ptr<GroupElement> h = dlog->createRandomElement();
// multiply elements
shared_ptr<GroupElement> gMult = dlog->multiplyGroupElements(g1.get(), h.get());
```

Supported Dlog Types

In this section we present the Discrete log groups provided by libscapi.

The OpenSSL implementation:

Class Name	Class Location
OpenSSLDlogZpSafePrime	libscapi/include/primitives/DlogOpenSSL.hpp
OpenSSLDlogECFp	libscapi/include/primitives/DlogOpenSSL.hpp
OpenSSLDlogECF2m	libscapi/include/primitives/DlogOpenSSL.hpp

Key Derivation Function (KDF)

A key derivation function (or KDF) is used to derive (close to) uniformly distributed string/s from a secret value with high entropy (but no other guarantee regarding its distribution).

Contents

The `Key Derivation Function` abstract class:

SecretKey `KeyDerivationFunction::`**`deriveKey`**(const vector<byte>& *entropySource*, int *inOff*, int *inLen*, int *outLen*, const vector<byte>& *iv=vector<byte>()*)

Generates a new secret key from the given seed and iv (if given).

Parameters

- **entropySource** – the secret key that is the seed for the key generation
- **inOff** – the offset within the entropySource to take the bytes from
- **inLen** – the length of the seed
- **outLen** – the required output key length
- **iv** – info for the key generation

Returns SecretKey the derivated key.

Basic Usage

```
KeyDerivationFunction* kdf = new HKDF(make_shared<OpenSSLHMAC>());
vector<byte> source(3, 1);
int targetLen = 128;
vector<byte> kdfed = kdf->deriveKey(source, 0, source.size(), targetLen).getEncoded();
```

Supported KDF Types

In this section we present the key derivation functions provided by libscapi.

Class Name	Class Location
HKDF	libscapi/include/primitives/Kdf.hpp

CHAPTER 8

Layer 2: Non Interactive Protocols

The second layer of libscapi currently includes different symmetric and asymmetric encryption schemes. In the future this layer will also include message authentication codes and digital signatures. It heavily uses the primitives of the first layer to perform internal operations. For example, the ElGamal encryption scheme uses DlogGroup.

Message Authentication Codes

In cryptography, a Message Authentication Code (MAC) is a short piece of information used to authenticate a message and to provide integrity and authenticity assurances on the message. Integrity assurances detect accidental and intentional message changes, while authenticity assurances affirm the message's origin. libscapi currently provides only one implementation of message authentication codes: HMAC.

Contents

The Mac abstract class

This is the general class for Mac. Every class in this family must derive this class.

class `Mac`

Basic Mac and Verify Functionality

vector<byte> `Mac::mac`(const vector<byte>& *msg*, int *offset*, int *msgLen*)

Computes the mac operation on the given msg and return the calculated tag.

Parameters

- **msg** – the message to operate the mac on.

- **offset** – the offset within the message vector to take the bytes from.
- **msgLen** – the length of the message in bytes.

Returns vector<byte> the return tag from the mac operation.

bool Mac::**verify**(const vector<byte>& *msg*, int *offset*, int *msgLength*, vector<byte>& *tag*)
Verifies that the given tag is valid for the given message.

Parameters

- **msg** – the message to compute the mac on to verify the tag.
- **offset** – the offset within the message array to take the bytes from.
- **msgLength** – the length of the message in bytes.
- **tag** – the tag to verify.

Returns true if the tag is the result of computing mac on the message. false, otherwise.

Calulcating the Mac when not all the message is known up front

void Mac::**update**(vector<byte>& *msg*, int *offset*, int *msgLen*)
Adds the byte array to the existing message to mac.

Parameters

- **msg** – the message to add.
- **offset** – the offset within the message array to take the bytes from.
- **msgLen** – the length of the message in bytes.

void Mac::**doFinal**(vector<byte>& *msg*, int *offset*, int *msgLength*, vector<byte>& *tag_res*)
Completes the mac computation and puts the result tag in the tag array.

Parameters

- **msg** – the end of the message to mac.
- **offset** – the offset within the message array to take the bytes from.
- **msgLength** – the length of the message in bytes.

Returns the result tag from the mac operation.

Key Handling

SecretKey Mac::**generateKey**(int *keySize*)
Generates a secret key to initialize this mac object.

Parameters keySize is the required secret key size in bits.

Returns the generated secret key.

SecretKey Mac::**generateKey**(AlgorithmParameterSpec& *keyParams*)
Generates a secret key to initialize this mac object.

Parameters keyParams algorithmParameterSpec contains parameters for the key generation of this mac algorithm.

Returns the generated secret key.

bool Mac::**isKeySet**()

An object trying to use an instance of mac needs to check if it has already been initialized.

Returns true if the object was initialized by calling the function setKey.

void Mac::**setMacKey**(SecretKey& *secretKey*)

Sets the secret key for this mac. The key can be changed at any time.

Parameters secretKey secret key

Mac Properties

int Mac::**getMacSize**()

Returns the input block size in bytes.

Returns the input block size.

HMAC

We presented the same HMAC algorithm in the first layer of libscapi. However, there it was only presented as a PRF. In order to make HMAC become also a MAC and not just a PRF, all we have to do is to derive the Mac class. This means that now our HMAC needs to know how to mac and verify. HMAC is a mac that does not require knowing the length of the message in advance.

The Hmac class

Hmac is a Marker interface. Every class that implements it is signed as Hmac. Hmac has varying input length and thus implements the interface PrfVaryingInputLength. Currenty the `BcHMAC` class implements the `Hmac` interface.

Hmac : public virtual PrfVaryingInputLength, public virtual UniqueTagMac, public virtual U

Basic Usage

Sender usage:

```
//Create an hmac object.
OpenSSLHMAC hmac("SHA-1");

//Generate a SecretKey
Hmac.generateKey(128);

//Set the secretKey.
hmac.setKey(secretKey);

//Get the message to mac and calculate the mac tag.
auto tag = hmac.mac(msg, offset, length);

//Send the msg and tag to the receiver.
...
```

Receiver usage:

```
//Get secretKey, msg and tag byte arrays.
...
//Create the same hmac object as the sender’s hmac object and set the key.
...
```

```
// receive the message and the tag
...
// Verify the tag with the given msg.
If (hmac.verify(tag, msg, offset, length)) { //Tag is valid.
    //Continue working...
} else return ERROR; //Tag is not valid.
```

Symmetric Encryption

There are three main categories of symmetric encryption:

1. An encryption based on modes of operation using a pseudo-random permutation and a randomized IV. The randomized IV is crucial for security. **CBCEnc** and **CTREnc** belong to this category.
2. An authenticated encryption where the message gets first encrypted and then mac-ed. **EncryptThenMac** belongs to this category.
3. Homomorphic encryption.

Libscapi currently implemented the CTR encryption only. In the future we may add more implementations.

The symmetric encryption class implements three main functionalities that correspond to the cryptographer's language in which an encryption scheme is composed of three algorithms:

1. Generation of the key.
2. Encryption of the plaintext.
3. Decryption of the ciphertext.

Contents

The SymmetricEnc abstract class

class `SymmetricEnc` : **public** *Eav*, **public** *Indistinguishable*

This is the main class for the Symmetric Encryption family. Any symmetric encryption scheme belongs by default at least to the Eavsdropper Security Level and to the Indistinguishable Security Level.

Encryption and Decryption

shared_ptr<SymmetricCiphertext> `SymmetricEnc::`**`encrypt`**(Plaintext* *plaintext*)

Encrypts a plaintext. It lets the system choose the random IV.

Returns an IVCiphertext, which contains the IV used and the encrypted data.

shared_ptr<SymmetricCiphertext> `SymmetricEnc::`**`encrypt`**(Plaintext* *plaintext*, vector<byte>& *iv*)
    This function encrypts a plaintext. It lets the user choose the random IV.

    Parameters iv random bytes to use in the encryption pf the message.

    Returns an IVCiphertext, which contains the IV used and the encrypted data.

shared_ptr<Plaintext> `SymmetricEnc::`**`decrypt`**(SymmetricCiphertext* *ciphertext*)
    This function performs the decryption of a ciphertext returning the corresponding decrypted plaintext.

    Parameters ciphertext The Ciphertext to decrypt.

    Returns the decrypted plaintext.

Key Generation

SecretKey `SymmetricEnc::`**`generateKey`**(AlgorithmParameterSpec& *keyParams*)
    Generates a secret key to initialize this symmetric encryption.

    Parameters keyParams algorithmParameterSpec contains parameters for the key generation of this symmetric encryption.

    Returns the generated secret key.

SecretKey `SymmetricEnc::`**`generateKey`**(int *keySize*)
    Generates a secret key to initialize this symmetric encryption.

    Parameters keySize is the required secret key size in bits.

    Returns the generated secret key.

Key Handling

bool `SymmetricEnc::`**`isKeySet`**()
    An object trying to use an instance of symmetric encryption needs to check if it has already been initialized.

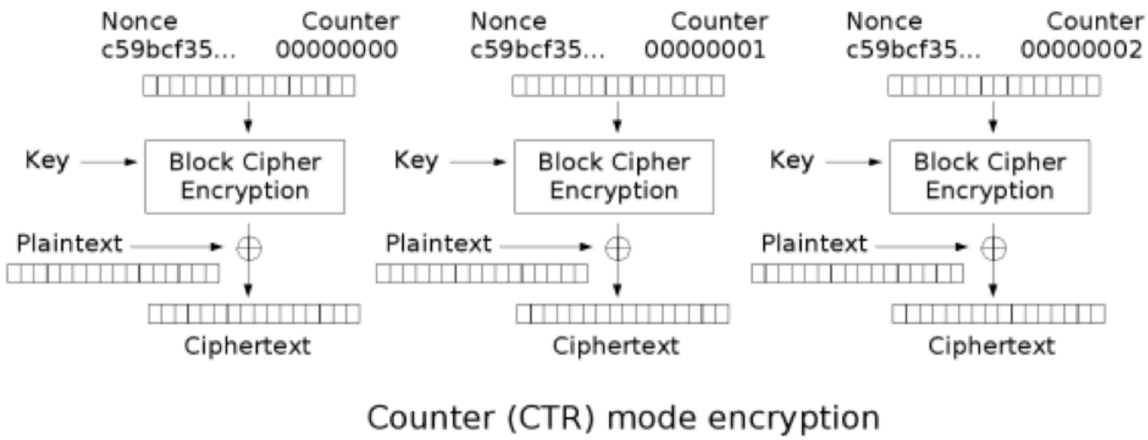
    Returns true if the object was initialized by calling the function setKey.

void `SymmetricEnc::`**`setKey`**(SecretKey& *secretKey*)
    Sets the secret key for this symmetric encryption. The key can be changed at any time.

    Parameters secretKey secret key.

The CTREnc abstract class

This is a marker class, for the CTR method:



CTREnc : public virtual SymmetricEnc, public Cpa

Basic Usage

Sender usage:

```
OpenSSLCTREncRandomIV encryptor("AES");

//Generate a SecretKey using the created object and set it.
SecretKey key = encryptor.generateKey(128);
encryptor.setKey(key);

//Get a plaintext to encrypt, and encrypt the plaintext.
...
SymmetricCiphertext cipher = encryptor.encrypt(plaintext);

//Send the cipher to the decryptor.
...
```

Receiver usage:

```
//Create the same SymmetricEnc object as the sender's encryption object, and set the key.
//Get the ciphertext and decrypt it to get the plaintext.
Plaintext plaintext = decryptor.decrypt(cipher);
```

Supported Encryption Types

In this section we present the symmetric encryptions provided by libscapi.

The OpenSSL implementation:

Class Name	Class Location
OpenSSLCTREncRandomIV	libscapi/include/mid_layer/OpenSSLSymmetricEnc.hpp

Asymmetric Encryption

Asymmetric encryption refers to a cryptographic system requiring two separate keys, one to encrypt the plaintext, and one to decrypt the ciphertext. Neither key will do both functions. One of these keys is public and the other is kept private. If the encryption key is the one published then the system enables private communication from the public to the decryption key's owner.

Contents

Asymmetric encryption can be used by a protocol or a user in two different ways:

1. The protocol works on an abstract level and does not know the concrete algorithm of the asymmetric encryption. This way the protocol cannot create a specific Plaintext to the encrypt function because it does not know which concrete Plaintext the encrypt function should get. Similarly, the protocol does not know how to treat the Plaintext returned from the decrypt function. In these cases the protocol has a byte array that needs to be encrypted.

2. The protocol knows the concrete algorithm of the asymmetric encryption. This way the protocol knows which Plaintext implementation the encrypt function gets and the decrypt function returns. Therefore, the protocol can be specific and cast the plaintext to the concrete implementation. For example, the protocol knows that it has a DamgardJurikEnc object, so the encrypt function gets a BigIntegerPlaintext and the decrypt function returns a BigIntegerPlaintext. The protocol can create such a plaintext in order to call the encrypt function or cast the returned plaintext from the decrypt function to get the BigInteger value that was encrypted.

The AsymmetricEnc abstract class

class `AsymmetricEnc` : **public** *Cpa*, *Indistinguishable*

General class for asymmetric encryption. Each class of this family must derive this class.

Encryption and Decryption

shared_ptr<AsymmetricCiphertext> AsymmetricEnc::**encrypt**(const shared_ptr<Plaintext>& *plainText*)

Encrypts the given plaintext using this asymmetric encryption scheme.

Parameters plainText message to encrypt

Returns Ciphertext the encrypted plaintext

shared_ptr<AsymmetricCiphertext> AsymmetricEnc::**encrypt**(const shared_ptr<Plaintext>& *plainText*, const biginteger& *r*)

Decrypts the given ciphertext using this asymmetric encryption scheme.

Parameters cipher ciphertext to decrypt

Returns Plaintext the decrypted cipher

Plaintext Manipulation

shared_ptr<Plaintext> AsymmetricEnc::**generatePlaintext**(vector<byte>& *text*)

Generates a Plaintext suitable for this encryption scheme from the given message.

A Plaintext object is needed in order to use the encrypt function. Each encryption scheme might generate a different type of Plaintext according to what it needs for encryption. The encryption function receives as argument an object of type Plaintext in order to allow a protocol holding the encryption scheme to be oblivious to the exact type of data that needs to be passed for encryption.

Parameters text byte array to convert to a Plaintext object.

vector<byte> AsymmetricEnc::**generateBytesFromPlaintext**(Plaintext* *plaintext*)

Generates a byte array from the given plaintext. This function should be used when the user does not know the specific type of the Asymmetric encryption he has, and therefore he is working on byte array.

Parameters plaintext to generates byte array from.

Returns the byte array generated from the given plaintext.

int AsymmetricEnc::**getMaxLengthOfByteArrayForPlaintext**()

Returns the maximum size of the byte array that can be passed to generatePlaintext function. This is the maximum size of a byte array that can be converted to a Plaintext object suitable to this encryption scheme.

Returns the maximum size of the byte array that can be passed to generatePlaintext function.

bool AsymmetricEnc::**hasMaxByteArrayLengthForPlaintext**()

There are some encryption schemes that have a limit of the byte array that can be passed to the generatePlaintext. This function indicates whether or not there is a limit. Its helps the user know if he needs to pass an array with specific length or not.

Returns true if this encryption scheme has a maximum byte array length to generate a plaintext from; false, otherwise.

Key Generation

pair<shared_ptr<PublicKey>, shared_ptr<PrivateKey>> AsymmetricEnc::**generateKey**(AlgorithmParameterSpec* *keyParams*)

Generates public and private keys for this asymmetric encryption.

Parameters keyParams hold the required parameters to generate the encryption scheme's keys

Returns KeyPair holding the public and private keys relevant to the encryption scheme

pair<shared_ptr<PublicKey>, shared_ptr<PrivateKey>> `AsymmetricEnc::generateKey()`
Generates public and private keys for this asymmetric encryption.

Returns KeyPair holding the public and private keys

Key Handling

shared_ptr<PublicKey> `AsymmetricEnc::getPublicKey()`
Returns the PublicKey of this encryption scheme.

This function should not be use to check if the key has been set. To check if the key has been set use isKeySet function.

Returns the PublicKey

bool `AsymmetricEnc::isKeySet()`
Checks if this AsymmetricEnc object has been previously initialized with corresponding keys.

Returns `true` if either the Public Key has been set or the key pair (Public Key, Private Key) has been set; `false` otherwise.

void `AsymmetricEnc::setKey`(const shared_ptr<PublicKey>& *publicKey*, const shared_ptr<PrivateKey>& *privateKey*)
Sets this asymmetric encryption with public key and private key.

void `AsymmetricEnc::setKey`(const shared_ptr<PublicKey>& *publicKey*)
Sets this asymmetric encryption with a public key

In this case the encryption object can be used only for encryption.

Reconstruction (from communication channel)

shared_ptr<AsymmetricCiphertext> `AsymmetricEnc::reconstructCiphertext`(AsymmetricCiphertextSendableData* *data*)
Reconstructs a suitable AsymmetricCiphertext from data that was probably obtained via a Channel or any other means of sending data (including serialization).

We emphasize that this is NOT in any way an encryption function, it just receives ENCRYPTED DATA and places it in a ciphertext object.

Parameters data contains all the necessary information to construct a suitable ciphertext.

Returns the AsymmetricCiphertext that corresponds to the implementing encryption scheme, for ex: CramerShoupCiphertext

shared_ptr<PrivateKey> `AsymmetricEnc::reconstructPrivateKey`(KeySendableData* *data*)
Reconstructs a suitable PrivateKey from data that was probably obtained via a Channel or any other means of sending data (including serialization).

We emphasize that this function does NOT in any way generate a key, it just receives data and recreates a PrivateKey object.

Parameters data a KeySendableData object needed to recreate the original key. The actual type of KeySendableData has to be suitable to the actual encryption scheme used, otherwise it throws an IllegalArgumentException

Returns a new PrivateKey with the data obtained as argument

shared_ptr<PublicKey> `AsymmetricEnc::reconstructPublicKey`(KeySendableData* *data*)
Reconstructs a suitable PublicKey from data that was probably obtained via a Channel or any other means of sending data (including serialization).

We emphasize that this function does NOT in any way generate a key, it just receives data and recreates a PublicKey object.

Parameters data a KeySendableData object needed to recreate the original key. The actual type of KeySendableData has to be suitable to the actual encryption scheme used, otherwise it throws an IllegalArgumentException

Returns a new PublicKey with the data obtained as argument

Using the Generic Interface

Sender Usage:

```
//Get an abstract Asymmetric encryption object from somewhere.
//Generate a key pair using the encryptor.
auto pair = encryptor.generateKey();

//Publish your public key.
Publish(pair.first);

//Set private key and party2's public key:
encryptor.setKey(party2PublicKey, pair.second);

//Generate a plaintext suitable for this encryption object using the encryption object.
Plaintext plaintext = encryptor.generatePlaintext(msg);

//Encrypt the plaintext
AsymmetricCiphertext cipher = encryptor.encrypt(plaintext);

//Send cipher and keys to the receiver.
...
```

Receiver Usage:

```
//Get the same asymmetric encryption object as the sender's object. //Generate a keyPair using the e
auto pair = encryptor.generateKey();

//Publish your public key.
Publish(pair.getPublic());

//Set private key and party1's public key:
encryptor.setKey(party1PublicKey, pair.second);

//Get the ciphertext and decrypt it to get the plaintext.
...

Plaintext plaintext = encryptor.decrypt(cipher);
//Get the plaintext bytes using the encryption object and use it as needed.
auto text = encryptor.generatesBytesFromPlaintext(plaintext);
...
```

El Gamal Encryption Scheme

The El Gamal encryption scheme's security is based on the hardness of the decisional Diffie-Hellman (DDH) problem. ElGamal encryption can be defined over any cyclic group G. Its security depends upon the difficulty of a certain problem in G related to computing discrete logarithms. We implement El Gamal over a Dlog Group (G, q, g) where q is the order of group G and g is the generator.

ElGamal encryption scheme can encrypt a group element and a byte array. The general case that accepts a message that should be encrypted usually uses the encryption on a byte array, but in other cases there are protocols that do multiple calculations and might want to keep working on a close group. For those cases we provide encryption on a group element.

In order to allow these two encryption types, we provide two ElGamal concrete classes. One implements the encrypt function on a group element and is called *ElGamalOnGroupElementEnc*, and the other one implements the encrypt function on a byte array and is called *ElGamalOnByteArrayEnc*.

Note: Note that ElGamal on a groupElement is an asymmetric multiplicative homomorphic encryption, while ElGamal on a ByteArray is not.

ElGamalEnc abstract class

class **`ElGamalEnc`** : **public** *AsymmetricEnc*

General class for El Gamal encryption scheme. Every concrete implementation of ElGamal should derive this class. By definition, this encryption scheme is CPA-secure and Indistinguishable.

ElGamalOnByteArrayEnc class

class **`ElGamalOnByteArrayEnc`** : **public** *ElGamalEnc*

This class performs the El Gamal encryption scheme that perform the encryption on a ByteArray. The general encryption of a message usually uses this type of encryption. By definition, this encryption scheme is CPA-secure and Indistinguishable.

Constructors

`ElGamalOnByteArrayEnc::`**`ElGamalOnByteArrayEnc`**`()`

Default constructor. Uses the default implementations of DlogGroup and KDF.

`ElGamalOnByteArrayEnc::`**`ElGamalOnByteArrayEnc`**`(`const shared_ptr<DlogGroup>& *dlogGroup*, const shared_ptr<KeyDerivationFunction>& *kdf*, const shared_ptr<PrgFromOpenSSLAES>& *random*`)`

Constructor that gets a DlogGroup and source of randomness.

Parameters

- **dlogGroup** – must be DDH secure.
- **kdf** – a key derivation function.
- **random** – source of randomness

Complete Encryption

shared_ptr<AsymmetricCiphertext> `ElGamalOnByteArrayEnc::completeEncryption(const shared_ptr<GroupElement>& c1, GroupElement* hy, Plaintext* plaintext)`

This is a protected function. It completes the encryption operation.

Parameters plaintext contains message to encrypt. MUST be of type ByteArrayPlaintext.

Returns Ciphertext of type ElGamalOnByteArrayCiphertext containing the encrypted message.

ElGamalOnGroupElementEnc class

class `ElGamalOnGroupElementEnc` : **public** *ElGamalEnc*, **public** *AsymMultiplicativeHomomorphicEnc*

This class performs the El Gamal encryption scheme that perform the encryption on a GroupElement.

In some cases there are protocols that do multiple calculations and might want to keep working on a close group. For those cases we provide encryption on a group element. By definition, this encryption scheme is CPA-secure and Indistinguishable.

Constructors

`ElGamalOnGroupElementEnc::ElGamalOnGroupElementEnc()`

Default constructor. Uses the default implementations of DlogGroup and random.

`ElGamalOnGroupElementEnc::ElGamalOnGroupElementEnc(const shared_ptr<DlogGroup>& dlogGroup, const shared_ptr<PrgFromOpenSSLAES>& random)`

Constructor that gets a DlogGroup and source of randomness.

Parameters

- **dlogGroup** – must be DDH secure.
- **random** – source of randomness.

Complete Encryption

shared_ptr<AsymmetricCiphertext> `ElGamalOnGroupElementEnc::completeEncryption(const shared_ptr<GroupElement>& c1, GroupElement* hy, Plaintext* plaintext)`

This is a protected function. It completes the encryption operation.

Parameters plaintext contains message to encrypt. MUST be of type GroupElementPlaintext.

Returns Ciphertext of type ElGamalOnGroupElementCiphertext containing the encrypted message.

Multiply Ciphertexts (Homomorphic Encryption operation)

shared_ptr<AsymmetricCiphertext> `ElGamalOnGroupElementEnc::multiply`(AsymmetricCiphertext* *cipher1*, AsymmetricCiphertext* *cipher2*)

Calculates the ciphertext resulting of multiplying two given ciphertexts. Both ciphertexts have to have been generated with the same public key and DlogGroup as the underlying objects of this ElGamal object.

Returns Ciphertext of the multiplication of the plaintexts p1 and p2 where alg.encrypt(p1)=cipher1 and alg.encrypt(p2)=cipher2

shared_ptr<AsymmetricCiphertext> `ElGamalOnGroupElementEnc::multiply`(AsymmetricCiphertext* *cipher1*, AsymmetricCiphertext* *cipher2*, biginteger& *r*)

Calculates the ciphertext resulting of multiplying two given ciphertexts using the given random value r. Both ciphertexts have to have been generated with the same public key and DlogGroup as the underlying objects of this ElGamal object.

Returns Ciphertext of the multiplication of the plaintexts p1 and p2 where alg.encrypt(p1)=cipher1 and alg.encrypt(p2)=cipher2

Basic Usage

Sender usage:

```
shared_ptr<DlogGroup> dlog = make_shared<OpenSSLDlogECF2m>();
//Create an ElGamalOnGroupElement encryption object.
ElGamalOnGroupElementEnc elGamal(dlog);

//Generate a keyPair using the ElGamal object.
auto pair = elGamal.generateKey();

//Publish your public key.
Publish(pair.first);

//Set private key and party2's public key:
elGamal.setKey(party2PublicKey, pair.second);

//Create a GroupElementPlaintext to encrypt and encrypt the plaintext.
GroupElementPlaintext plaintext(dlog->createRandomElement());
AsymmetricCiphertext cipher = elGamal.encrypt(plaintext);

//Sends cipher to the receiver.
```

Receiver usage:

```
//Create an ElGamal object with the same DlogGroup definition as party1.
//Generate a keyPair using the ElGamal object.
auto pair = elGamal.generateKey();

//Publish your public key.
Publish(pair.first);
```

```
//Set private key and party1's public key:
elGamal.setKey(party1PublicKey, pair.second);

//Get the ciphertext and decrypt it to get the plaintext.
...
shared_ptr<Plaintext> plaintext = elGamal.decrypt(cipher);

//Get the plaintext element and use it as needed.
GroupElement element = ((GroupElementPlaintext*)plaintext.get()).getElement();
...
```

Cramer Shoup DDH Encryption Scheme

The Cramer Shoup encryption scheme's security is based on the hardness of the decisional Diffie-Hellman (DDH) problem, like El Gamal encryption scheme. Cramer Shoup encryption can be defined over any cyclic group G. Its security depends upon the difficulty of a certain problem in G related to computing discrete logarithms.

We implement Cramer Shoup over a Dlog Group (G, q, g) where q is the order of group G and g is the generator.

In contrast to El Gamal, which is extremely malleable, Cramer–Shoup adds other elements to ensure non-malleability even against a resourceful attacker. This non-malleability is achieved through the use of a hash function and additional computations, resulting in a ciphertext which is twice as large as in El Gamal.

Similary to ElGamal, Cramer Shoup encryption scheme can encrypt a group element and a byte array. libscapi only provides the group element version.

The CramerShoupOnGroupElementEnc class

class **`CramerShoupOnGroupElementEnc`** : **public** *AsymmetricEnc*, *Cca2*

Implementation of CramerShoup encryption scheme over group elements.

`CramerShoupOnGroupElementEnc::`**`CramerShoupOnGroupElementEnc`**(const shared_ptr<DlogGroup>& *dlogGroup*, const shared_ptr<CryptographicHash>& *hash*, const shared_ptr<PrgFromOpenSSLAES>& *random*)

Constructor that lets the user choose the underlying dlog, hash and random.

Parameters

- **dlogGroup** – underlying DlogGroup to use, it has to have DDH security level
- **hash** – underlying hash to use, has to have CollisionResistant security level
- **random** – source of randomness.

Basic Usage

Sender usage:

```
//Create an underlying DlogGroup.
shared_ptr<DlogGroup> dlog = make_shared<OpenSSLDlogECF2m>();

//Create a CramerShoupOnByteArray encryption object.
```

```
CramerShoupOnGroupElementEnc encryptor (dlog);

//Generate a keyPair using the CramerShoup object.
auto pair = encryptor.generateKey();

//Publish your public key.
Publish(pair.first);

//Set private key and party2's public key:
encryptor.setKey(party2PublicKey, pair.second);

//Get a vector message to encrypt. Check if the length of the given msg is valid.
if (encryptor.hasMaxByteArrayLengthForPlaintext()){
    if (msg.size() > encryptor.getMaxLengthOfByteArrayForPlaintext()) {
        throw invalid_argument("message too long");
    }
}

//Generate a plaintext suitable to this CramerShoup object.
auto plaintext = encryptor.generatePlaintext(msg);

//Encrypt the plaintext
auto cipher = encrypor.encrypt(plaintext);

//Send cipher and keys to the receiver.
```

Receiver usage:

```
//Create a CramerShoup object with the same DlogGroup definition as party1.
//Generate a keyPair using the CramerShoup object.
auto pair = encryptor.generateKey();

//Publish your public key.
Publish(pair.first);

//Set private key and party1's public key:
encryptor.setKey(party1PublicKey, pair.second);

//Get the ciphertext and decrypt it to get the plaintext. ...
auto plaintext = encryptor.decrypt(cipher);

//Get the plaintext element and use it as needed.
GroupElement element = ((GroupElementPlaintext*)plaintext.get()).getElement();
```

Damgard Jurik Encryption Scheme

Damgard Jurik is an asymmetric encryption scheme that is based on the Paillier encryption scheme. This encryption scheme is CPA-secure and Indistinguishable.

DamgardJurikEnc class

class `DamgardJurikEnc` : **public** *AsymAdditiveHomomorphicEnc*

Damgard Jurik is an asymmetric encryption scheme based on the Paillier encryption scheme. By definition, this encryption scheme is CPA-secure and Indistinguishable.

`DamgardJurikEnc::`**`DamgardJurikEnc`**(const shared_ptr<PrgFromOpenSSLAES>& *random*)
Constructor that lets the user choose the source of randomness.

shared_ptr<AsymmetricCiphertext> `DamgardJurikEnc::`**`reRandomize`**(AsymmetricCiphertext* *cipher*)
This function takes an encryption of some plaintext (let's call it originalPlaintext) and returns a cipher that "looks" different but it is also an encryption of originalPlaintext.

Basic Usage

The code example below is used when the sender and receiver know the specific type of asymmetric encryption object.

Sender code:

```
//Create a DamgardJurik encryption object.
DamgardJurikEnc encryptor;

//Generate a keyPair using the DamgardJurik object.
DJKeyGenParameterSpec spec(128, 40)
auto pair = encryptor.generateKey(spec);

//Publish your public key.
Publish(pair.first);

//Set private key and party2's public key:
encryptor.setKey(party2PublicKey, pair.second);

//Get the biginteger value to encrypt, create a BigIntegerPlaintext with it and encrypt the plaintext
...
BigIntegerPlainText plaintext(num);
auto cipher = encryptor.encrypt(plaintext);

//Send cipher and keys to the receiver.
```

Receiver code:

```
//Create a DamgardJurik object with the same definition as party1.
//Generate a keyPair using the DamgardJurik object.
auto pair = encryptor.generateKey();

//Publish your public key.
Publish(pair.first);

//Set private key and party1's public key:
encryptor.setKey(party1PublicKey, pair.second);

//Get the ciphertext and decrypt it to get the plaintext. ...
auto plaintext = elGamal.decrypt(cipher);

//Get the plaintext element and use it as needed.
biginteger element = ((BigIntegerPlainText)plaintext.get()).getX();
```

Layer 3: Interactive Protocols

The Interactive Protocol layer contains interactive protocols which can be used as a standalone protocols or as building blocks of higher cryptographic schemes. The protocols in this layer are two-party protocols, meaning that there are two participants in the protocol execution when each one has a different role. For example, OT protocol consists of a sender and a receiver, ZK protocol consists of a prover and a verifier, etc. The communication between the parties is done through the SCAPI's Communication Layer.

This layer contains the following components:

Oblivious Transfer Protocols

In Oblivious Transfer, a party called **the sender** has n messages, and a party called **the receiver** has an index i. The receiver wishes to receive the i^{th} message of the sender, without the sender learning i, while the sender wants to ensure that the receiver receives only one of the n messages.

Contents

Class Hierarchy

The general structure of OT protocols contains three components:

- Sender and receiver abstract classes
- Sender and receiver concrete classes

abstract classes

Both Sender and Receiver abstract classes declare the `transfer()` function, which executes the OT protocol. The `transfer()` function of the sender runs the protocol from the sender's point of view, while the transfer function of the receiver runs the protocol from the receiver's point of view. There are two types of abstract classes. One is for the regular OT case and the other for the batch OT case .

In the regular OT case, both transfer functions accept two parameters:

- A channel that is used to send and receive messages during the protocol execution.
- An input object that holds the required parameter to the sender/receiver execution.

In the batch OT case, the transfer functions accept just the input object, since all concrete implementations use their own communication rether than libscapi's channel.

The input types are `OTSInput` and `OTRInput` for the regular case, and `OTBatchSInput` and `OTBatchRInput` for the batch case. These are abstract classes for the sender's and receiver's input, respectively. Each concrete implementation may have some different parameters and should implement a dedicated input class that holds them. The transfer functions of the sender and the receiver differ in their return value. In the regular case, the sender's transfer function returns void, and the receiver's transfer function returns `OTROutput`. In the batch case, the sender's transfer function returns `OTBatchSOutput`, and the receiver's transfer function returns `OTBatchROutput`. All types of output are abstract classes and work as marker classes. Each concrete OT receiver should implement a dedicated output class that holds the necessary output objects.

The OTSender abstract class

class OTSender

void `OTSender::`**`transfer`**(CommParty* *channel*, OTSInput* *input*)

The transfer stage of OT protocol which can be called several times in parallel. The OT implementation support usage of many calls to transfer, with single preprocess execution. This way, one can execute multiple OTs by creating the OT sender once and call the transfer function for each input couple. In order to enable parallel calls, each transfer call should use a different channel to send and receive messages. This way the parallel executions of the function will not block each other.

Parameters

- **channel** – each call should get a different one.
- **input** – The parameters given in the input must match the DlogGroup member of this class, which given in the constructor.

The OTReciever abstract class

class OTReceiver

shared_ptr<OTROutput> `OTReceiver::`**`transfer`**(CommParty* *channel*, OTRInput* *input*)

The transfer stage of OT protocol which can be called several times in parallel. The OT implementation support usage of many calls to transfer, with single preprocess execution. This way, one can execute multiple OT by creating the OT receiver once and call the transfer function for each input couple. In order to enable parallel calls, each transfer call should use a different channel to send and receive messages. This way the parallel executions of the function will not block each other.

Parameters

- **channel** – each call should get a different one.

- **input** – The parameters given in the input must match the DlogGroup member of this class, which given in the constructor.

Returns OTROutput, the output of the protocol.

The OTBatchSender abstract class

class OTBatchSender

shared_ptr<OTBatchSOutput> `OTBatchSender::transfer`(OTBatchSInput* *input*)

The transfer stage of OT protocol which does mulptiple OTs in parallel.

Parameters input The parameters used in the

The OTBatchReceiver abstract class

class OTBatchReceiver

shared_ptr<OTBatchROutput> `OTBatchReceiver::transfer`(OTBatchRInput* *input*)

The transfer stage of OT protocol which does mulptiple OTs in parallel.

Parameters input The parameters given in the input must match the DlogGroup member of this class, which given in the constructor.

Returns OTROutput, the output of the protocol.

The Input/Output Interfaces

Every OT sender and receiver need inputs during the protocol execution, but every concrete protocol needs different inputs. The following classes are marker classes for regular and batch OT sender/receiver inputs, where there is an implementing class for each OT protocol.

class OTSInput

class OTRInput

class OTBatchSInput

class OTBatchRInput

Similar, every regular OT receiver and every batch sender and receiver outputs a result in the end of the protocol execution, but every concrete protocol output different data. The following classes are marker classes for OT output, where there is an implementing class for each OT protocol.

class OTROutput

class OTBatchSOutput

class OTBatchROutput

Concrete implementations

As we have already said, each concrete OT implementation should implement dedicated sender and receiver classes. These classes implement the functionalities that are unique for the specific implementation. Most OT protocols can work on two different types of inputs: byte arrays and DlogGroup elements. Each input type should be treated differently, thus we decided to have concrete sender/receiver classes for each input option.

Concrete *regular* OT implemented so far are:

- Semi Honest
- Privacy Only
- One Sided Simulation
- Full Simulation
- Full Simulation – ROM
- UC

Concrete *batch* OT implemented so far are:

- Batch Semi Honest Extension. This is a wrapper of Michael Zohner's implementation.
- Batch Malicious Extension. There are two wrappers: One wraps the Michael Zohner's implementation and the other wraps the Bristol's implementation.

Basic Usage

In order to execute the OT protocol, both sender and receiver should be created as separate programs (Usually not on the same machine). The main function in the sender and the receiver is the transfer function, that gets the communication channel between them and input.

Steps in sender creation:

- Given a `Channel` object channel do:
- Create an `OTSender` (for example, `OTSemiHonestDDHOnGroupElementSender`).
- Create input for the sender. Usually, the input for the receiver contains x0 and x1.
- Call the transfer function of the sender with channel and the created input.

```
//Creates the OT sender object.
OTSemiHonestDDHOnGroupElementSender sender;

//Creates input for the sender.
auto x0 = dlog.createRandomElement();
auto x1 = dlog.createRandomElement();
OTSOnGroupElementInput input(x0, x1);

//call the transfer part of the OT protocol
sender.transfer(&channel, &input);
```

Steps in receiver creation:

- Given a `Channel` object channel do:
- Create an `OTReceiver` (for example, `OTSemiHonestDDHOnGroupElementReceiver`).
- Create input for the receiver. Usually, the input for the receiver contains only sigma parameter.
- Call the transfer function of the receiver with channel and the created input.

```
//Creates the OT receiver object.
OTSemiHonestDDHOnGroupElementReceiver receiver;

//Creates input for the receiver.
byte sigma = 1;
OTRBasicInput input(sigma);
```

```
OTROutput output = receiver.transfer(&channel, &input);
//use output...
```

Sigma Protocols

Sigma Protocols are a basic building block for Zero-knowledge proofs, Zero-Knowledge Proofs Of Knowledge and more. A sigma protocol is a 3-round proof, comprised of:

1. A first message from the prover to the verifier
2. A random challenge from the verifier
3. A second message from the prover.

Sigma Protocol can be executed as a standalone protocol or as a building block for another protocol, like Zero Knowledge proofs. As a standalone protocol, Sigma protocol should execute the protocol as is, including the communication between the prover and the verifier. As a building block for other protocols, Sigma protocol should only compute the prover's first and second messages and the verifier's challenge and verification. This is, in other words, the protocol functions without communication between the parties.

To enable both options, there is a separation between the communication part and the actual protocol computations. The general structure of Sigma Protocol contains the following components:

- Prover, Verifier and Simulator generic classes.
- ProverComputation and VerifierComputation abstract classes.
- ProverComputation and VerifierComputation concrete classes (Specific to each protocol).

Contents

The Prover class

The `SigmaProtocolProver` class has two modes of operation:

1. Explicit mode - call processFirstMessage() to process the first message and afterwards call processSecondMessage() to process the second message.
2. Implicit mode - Call prove() function that calls the above two functions. This way is more easy to use since the user should not be aware of the order in which the functions must be called.

class **SigmaProtocolProver**

General class for Sigma Protocol prover. This class manages the communication functionality of all the sigma protocol provers. It sends the first message, receives the challenge from the prover and sends the second message. It uses SigmaProverComputation instance of a concrete sigma protocol to compute the actual messages.

Sigma protocols are a basic building block for zero-knowledge, zero-knowledge proofs of knowledge and more.

A sigma protocol is a 3-round proof, comprised of a first message from the prover to the verifier, a random challenge from the verifier and a second message from the prover. See Hazay-Lindell (chapter 6) for more information.

void `SigmaProtocolProver::`**`processSecondMsg`**`()`
Processes the second step of the sigma protocol. It receives the challenge from the verifier, computes the second message and then sends it to the verifier.

This is a blocking function!

void `SigmaProtocolProver::`**`prove`**`(`const shared_ptr<SigmaProverInput>& *input*`)`
Runs the proof of this protocol.

This function executes the proof at once by calling the above functions one by one. This function can be called when a user does not want to save time by doing operations in parallel.

The Verifier class

The `SigmaProtocolVerifier` also has two modes of operation:

1. Explicit mode – call sampleChallenge() to sample the challenge, then sendChallenge() to receive the prover's first message and then call processVerify() to receive the prover's second message and verify the proof.
2. Implicit mode - Call verify() function that calls the above three functions. Same as the prove function of the prover, this way is much simpler, since the user should not know the order of the functions.

class `SigmaProtocolVerifier`
General class for Sigma Protocol verifier. This class manages the communication functionality of all the sigma protocol verifiers, such as send the challenge to the prover and receive the prover messages. It uses SigmaVerifierComputation instance of a concrete sigma protocol to compute the actual calculations.

vector<byte> `SigmaProtocolVerifier::`**`getChallenge`**`()`
Returns the sampled challenge.

Returns the challenge.

bool `SigmaProtocolVerifier::`**`processVerify`**`(`SigmaCommonInput* *input*`)`
Waits to the prover's second message and then verifies the proof. **This is a blocking function!**

Returns true if the proof has been verified; false, otherwise.

void `SigmaProtocolVerifier::`**`sampleChallenge`**`()`
Samples the challenge for this protocol.

void `SigmaProtocolVerifier::`**`sendChallenge`**`()`
Waits for the prover's first message and then sends the chosen challenge to the prover. **This is a blocking function!**

void `SigmaProtocolVerifier::`**`setChallenge`**`(`const vector<byte>& *challenge*`)`
Sets the given challenge.

bool `SigmaProtocolVerifier::`**`verify`**`(`SigmaCommonInput* *input*`)`
Runs the verification of this protocol.

This function executes the verification protocol at once by calling the following functions one by one. This function can be called when a user does not want to save time by doing operations in parallel.

Returns true if the proof has been verified; false, otherwise.

The Simulator class

The `SigmaSimulator` has two simulate() functions. Both functions simulate the sigma protocol. The difference between them is the source of the challenge; one function receives the challenge as an input argument, while the other samples a random challenge. Both simulate functions return `SigmaSimulatorOutput` object that holds the simulated a, e, z.

class SigmaSimulator
General class for Sigma Protocol Simulator. The simulator is a probabilistic polynomial-time function, that on input x and challenge e outputs a transcript of the form (a, e, z) with the same probability distribution as transcripts between the honest prover and verifier on common input x.

int `SigmaSimulator::`**getSoundnessParam**()
Returns the soundness parameter for this Sigma simulator.

Returns t soundness parameter

shared_ptr<SigmaSimulatorOutput> `SigmaSimulator::`**simulate**(SigmaCommonInput* *input*, const vector<byte>& *challenge*)
Computes the simulator computation.

Returns the output of the computation - (a, e, z).

shared_ptr<SigmaSimulatorOutput> `SigmaSimulator::`**simulate**(SigmaCommonInput* *input*)
Chooses random challenge and computes the simulator computation.

Returns the output of the computation - (a, e, z).

Computation classes

The classes that operate the **actual** protocol phases derive the `SigmaProverComputation` and `SigmaVerifierComputation` abstract classes. SigmaProverComputation computes the prover's messages and SigmaVerifierComputation computes the verifier's challenge and verification. Each operation is done in a dedicated function.

In case that Sigma Protocol is used as a building block, the protocol which uses it will hold an instance of SigmaProverComputation or SigmaVerifierComputation and will call the required function. Each concrete sigma protocol should implement the computation classes.

SigmaProverComputation

class SigmaProverComputation
This abstract class manages the mathematical calculations of the prover side in the sigma protocol. It samples random values and computes the messages.

shared_ptr<SigmaProtocolMsg> `SigmaProverComputation::`**computeFirstMsg**(const shared_ptr<SigmaProverInput>& *input*)
Computes the first message of the sigma protocol.

shared_ptr<SigmaProtocolMsg> `SigmaProverComputation::`**computeSecondMsg**(const vector<byte>& *challenge*)
Computes the second message of the sigma protocol.

SigmaVerifierComputation

class **SigmaVerifierComputation**
This abstract class manages the mathematical calculations of the verifier side in the sigma protocol. It samples random challenge and verifies the proof.

void SigmaVerifierComputation::**sampleChallenge**()
Samples the challenge for this protocol.

void SigmaVerifierComputation::**setChallenge**(const vector<byte>& *challenge*)
Sets the given challenge.

vector<byte> SigmaVerifierComputation::**getChallenge**()
Returns the sampled challenge.

Returns the challenge.

bool SigmaVerifierComputation::**verify**(SigmaCommonInput* *input*, SigmaProtocolMsg* *a*, SigmaProtocolMsg* *z*)
Verifies the proof.

Returns true if the proof has been verified; false, otherwise.

Supported Protocols

Concrete Sigma protocols implemented so far are:

- Dlog
- DH
- Extended DH
- Pedersen commitment knowledge
- Pedersen committed value
- El Gamal commitment knowledge
- El Gamal committed value
- El Gamal private key
- El Gamal encrypted value
- Cramer-Shoup encrypted value
- Damgard-Jurik encrypted zero
- Damgard-Jurik encrypted value
- Damgard-Jurik product
- AND (of multiple statements)
- OR of two statements
- OR of multiple statements

Example of Usage

Steps in prover creation:

- Given a `Channel` object channel and input for the concrete Sigma protocol prover (In the example below, x and h) do:
  - Create a `SigmaProverComputation` (for example, `SigmaDlogProverComputation`).
  - Create a `SigmaProtocolProver` with channel and the proverComputation.
  - Create input object for the prover.
  - Call the `prove()` function of the prover with the input.

Prover code example:

```
//Creates the dlog group, use the koblitz curve.
auto dlog = make_shared<OpenSSLDlogECF2m>("K-233");

//Creates sigma prover computation.
shared_ptr<SigmaProverComputation> proverComputation = make_shared<SigmaDlogProverComputation>(dlog,

//Create Sigma Prover with the given SigmaProverComputation.
SigmaProver prover(channel, proverComputation);

//Creates input for the prover.
shared_ptr<SigmaProverInput> input = make_shared<SigmaDlogProverInput>(h, w);

//Calls the prove function of the prover.
prover.prove(input);
```

Steps in verifier creation:

- Given a `Channel` object channel and input for the concrete Sigma protocol verifier (In the example below, h) do:
  - Create a `SigmaVerifierComputation` (for example, `SigmaDlogVerifierComputation`).
  - Create a `SigmaProtocolVerifier` with channel and verifierComputation.
  - Create input object for the verifier.
  - Call the `verify()` function of the verifier with the input.

Verifier code example:

```
//Creates the dlog group, use the koblitz curve.
auto dlog = make_shared<OpenSSLDlogECF2m>("K-233");

//Creates sigma verifier computation.
shared_ptr<SigmaVerifierComputation> verifierComputation = make_shared<SigmaDlogVerifierComputation>

//Creates Sigma verifier with the given SigmaVerifierComputation.
SigmaVerifier verifier(channel, verifierComputation);

// Creates input for the verifier.
shared_ptr<SigmaCommonInput> input = make_shared<SigmaDlogCommonInput>(h);

//Calls the verify function of the verifier.
verifier.verify(input);
```

Zero Knowledge Proofs and Zero Knowledge Proofs of Knowledge

A **zero-knowledge proof** or a zero-knowledge protocol is a method by which one party (the prover) can prove to another party (the verifier) that a given statement is true, without conveying any additional information apart from the fact that the statement is indeed true. A **zero-knowledge proof of knowledge (ZKPOK)** is a sub case of zero knowledge proofs, in which the prover proves to the verifier that he knows how to prove a statement, without actually proving it.

Contents

Zero Knowledge abstract classes

ZKProver

The `ZKProver` abstract class declares the `prove()` function that accepts an input and runs the ZK proof. The input type is `ZKProverInput`, which is a marker class. Every concrete protocol should have a dedicated input class that extends it.

class ZKProver

A zero-knowledge proof or zero-knowledge protocol is a method by which one party (the prover) can prove to another party (the verifier) that a given statement is true, without conveying any additional information apart from the fact that the statement is indeed true.

This is a general class that simulates the prover side of the Zero Knowledge proof. Every class that derive this class is signed as Zero Knowledge prover.

void `ZKProver::`**prove**(const shared_ptr<ZKProverInput>& *input*)

Runs the prover side of the Zero Knowledge proof.

Parameters input holds necessary values to the proof calculations.

ZKVerifier

The `ZKVerifier` abstract class declares the `verify()` function that accepts an input and runs the ZK proof verification. The input type is `ZKCommonInput`, which is a marker class of inputs that are common for the prover and the verifier. Every concrete protocol should have a dedicated input class that extends it.

class ZKVerifier

A zero-knowledge proof or zero-knowledge protocol is a method by which one party (the prover) can prove to another party (the verifier) that a given statement is true, without conveying any additional information apart from the fact that the statement is indeed true.

This is a general class that simulates the verifier side of the Zero Knowledge proof. Every class that derive this class is signed as Zero Knowledge verifier.

bool ZKVerifier::**verify**(ZKCommonInput* *input*, const shared_ptr<SigmaProtocolMsg>& *emptyA*, const shared_ptr<SigmaProtocolMsg>& *emptyZ*)
Runs the verifier side of the Zero Knowledge proof.

Parameters input holds necessary values to the varification calculations.

Returns true if the proof was verified; false, otherwise.

ZKProverInput

class ZKProverInput
Marker class. Each concrete ZK prover's input class should derive this class.

ZKCommonInput

class ZKCommonInput
This is a marker class for Zero Knowledge input, where there is an implementing class for each concrete Zero Knowledge protocol.

Zero Knowledge Proof of Knowledge classes

`ZKPOKProver` and `ZKPOKVerifier` are marker classes that extend the `ZKProver` and `ZKVerifier` classes. ZKPOK concrete protocol should extend these marker classes instead of the general ZK classes.

class ZKPOKProver : public *ZKProver*
This is a general class that simulates the prover side of the Zero Knowledge proof of knowledge. Every class that derive it is signed as ZKPOK prover.

ZKPOKVerifier : public virtual ZKVerifier
This is a general class that simulates the verifier side of the Zero Knowledge proof of knowledge. Every class that derive it is signed as ZKPOK verifier.

Implemented Protocols

Concrete Zero Knowledge protocols implemented so far are:

- Zero Knowledge from any sigma protocol
- Zero Knowledge Proof of Knowledge from any sigma protocol (currently implemented using Pedersen Commitment scheme)
- Zero Knowledge Proof of Knowledge from any sigma protocol Fiat Shamir (Random Oracle Model)

Example of Usage

Steps in prover creation:

- Given a Channel object channel and input for the underlying SigmaProverComputation (in the following case, h and x) do:
  - Create a SigmaProverComputation (for example, SigmaDlogProverComputation).
  - Create a ZKProver with channel and the proverComputation (ForExample, ZKFromSigmaProver).
  - Create input object for the prover.

– Call the prove function of the prover with the input.

Prover code example:

```
//create the ZK prover
auto dlog = make_shared<OpenSSLDlogECF2m>("K-233");
ZKFromSigmaProver prover(channel, make_shared<SigmaDlogProverComputation>(dlog, 40, get_seeded_prg()

//create the input for the prover
shared_ptr<SigmaDlogProverInput> input = make_shared<SigmaDlogProverInput>(h, x);

//Call prove function
prover.prove(input);
```

Steps in verifier creation:

- Given a Channel object channel and input for the underlying SigmaVerifierComputation (In the example below, h) do:
  - Create a SigmaVerifierComputation (for example, SigmaDlogVerifierComputation).
  - Create a ZKVerifier with channel and verifierComputation (For example, ZKFromSigmaVerifier).
  - Create input object for the verifier.
  - Call the verify function of the verifier with the input.

Verifier code example:

```
 //create the ZK prover
auto dlog = make_shared<OpenSSLDlogECF2m>("K-233");
 ZKFromSigmaVerifier verifier(channel, make_shared<SigmaDlogVerifierComputation>(dlog, 40, get_seede

 //create the input for the verifier
 shared_ptr<SigmaDlogCommonInput> input = make_shared<SigmaDlogCommonInput>(h);

 //Call verify function
 cout << verifier.verify(input) << endl;
```

Commitment Schemes

A commitment scheme allows one to commit to a chosen value (or a chosen statement) while keeping it hidden from others, with the ability to reveal the committed value later. There exist some commitment schemes that can be proven by ZK protocols.

Contents

The Committer class

class `CmtCommitter`

This the general class of the Committer side of a Commitment Scheme. A commitment scheme has a commitment phase in which the committer send the commitment to the Receiver, and a decommitment phase in which the the Committer sends the decommitment to the Receiver.

Commit and Decommit

void `CmtCommitter::`**`commit`**(const shared_ptr<CmtCommitValue>& *input*, long *id*)

This function is the heart of the commitment phase from the Committer's point of view.

Parameters

- **input** – The value that the committer commits about.
- **id** – Unique value attached to the input to keep track of the commitments in the case that many commitments are performed one after the other without decommiting them yet.

void `CmtCommitter::`**`decommit`**(long *id*)

This function is the heart of the decommitment phase from the Committer's point of view.

Parameters id Unique value used to identify which previously committed value needs to be decommitted now.

There are cases when the user wants to commit the input but remain non-interactive, meaning not to send the generate message yet. The reasons for doing that are vary, for example the user wants to prepare a lot of commitments and send together. In these cases the commit function is not useful since it sends the generates commit message to the other party. The following function provide the ability to generate the commitment and decommitment messages and get them without send to the other party:

shared_ptr<CmtCCommitmentMsg> `CmtCommitter::`**`generateCommitmentMsg`**(const shared_ptr<CmtCommitValue>& *input*, long *id*)

This function generates a commitment message using the given input and ID.

shared_ptr<CmtCDecommitmentMessage> `CmtCommitter::`**`generateDecommitmentMsg`**(long *id*)

This function generate a decommitment message using the given id.

Conversion to and from CmtCommitValue

vector<byte> CmtCommitter::**generateBytesFromCommitValue**(CmtCommitValue* *value*)
This function converts the given commit value to a byte array.

Parameters value to get its bytes.

Returns the generated bytes.

shared_ptr<CmtCommitValue> CmtCommitter::**generateCommitValue**(const vector<byte>& *x*)
This function wraps the raw data x with a suitable CommitValue instance according to the actual implementaion.

Parameters x array to convert into a commitValue.

Returns the created CommitValue.

Inner state functions

CmtCommitmentPhaseValues* CmtCommitter::**getCommitmentPhaseValues**(long *id*)
This function returns the values calculated during the commit phase for a specific commitment. This function is used for protocols that need values of the commitment, like ZK protocols during proofs on the commitment. We recommended not to call this function from somewhere else.

Parameters id of the specific commitment

Returns values calculated during the commit phase

vector<shared_ptr<void>> CmtCommitter::**getPreProcessValues**()
This function returns the values calculated during the preprocess phase. This function is used for protocols that need values of the commitment, like ZK protocols during proofs on the commitment. We recommended not to call this function from somewhere else.

Returns values calculated during the preprocess phase

shared_ptr<CmtCommitValue> CmtCommitter::**sampleRandomCommitValue**()
This function samples random commit value to commit on.

Returns the sampled commit value.

The Receiver class

class **CmtReceiver**
This the general class of the Receiver side of a Commitment Scheme. A commitment scheme has a commitment phase in which the Receiver waits for the commitment sent by the Committer; and a decommitment phase in which the Receiver waits for the decommitment sent by the Committer and checks whether to accept or reject the decommitment.

Receive Commitment and Decommitment

shared_ptr<CmtRCommitPhaseOutput> CmtReceiver::**receiveCommitment**()
This function is the heart of the commitment phase from the Receiver's point of view.

Returns the id of the commitment and some other information if necessary according to the implementing class.

shared_ptr<CmtCommitValue> CmtReceiver::**receiveDecommitment**(long *id*)
This function is the heart of the decommitment phase from the Receiver's point of view.

Parameters id wait for a specific message according to this id

Returns the commitment

shared_ptr<CmtCommitValue> `CmtReceiver::`**`verifyDecommitment`**(CmtCCommitmentMsg* *commitmentMsg*, CmtCDecommitmentMessage* *decommitmentMsg*)

There are cases when the receiver gets the commitment and decommitments in the application (not by the channel), and the receiver does not use the receiveCommitment and receiveDecommitment function. In these cases this function should be called for each pair of commitment and decommitment messages. The reasons for doing that are vary, for example a protocol that prepare a lot of commitments and send together. In these cases the receiveCommitment and receiveDecommitment functions are not useful since they receive the generates messages separately to the other party. This function generates the message without sending it and this allows the user to save it and send it later if he wants.

Conversion to and from CmtCommitValue

vector<byte> `CmtReceiver::`**`generateBytesFromCommitValue`**(CmtCommitValue* *value*)

This function converts the given commit value to a byte array.

Parameters value to get its bytes.

Returns the generated bytes.

Inner state functions

shared_ptr<void> `CmtReceiver::`**`getCommitmentPhaseValues`**(long *id*)

Return the intermediate values used during the commitment phase.

Parameters id get the commitment values according to this id.

Returns a general array of Objects.

vector<shared_ptr<void>> `CmtReceiver::`**`getPreProcessedValues`**()

Return the values used during the pre-process phase (usually upon construction). Since these values vary between the different implementations this function returns a general array of Objects.

Returns a general array of Objects

Implemented Protocols

Each concrete commitment protocol should have committer and receiver classes that extends the `CmtCommitter` and `CmtReceiver` abstract classes mentioned above or the `CmtCommitterWithProofs` and `CmtReceiverWithProofs`, in case the scheme can be proven.

Concrete Commitments protocols implemented so far are: * Pedersen commitment * Pedersen Hash commitment * Pedersen Trapdoor commitment * El Gamal commitment * El Gamal Hash commitment * Simple Hash commitment * Equivoqal commitments

Example of Usage

Commitment protocol has two sides: committer and receiver. In order to execute the commitment protocol, both committer and receiver should be created as separate programs (Usually not on the same machine).

Steps in committer creation:

- Given a `Channel` object ch do:

  - Create a CmtCommitter (for example, CmtPedersenCommitter).
  - Create an instance of the concrete CommitValue that suits the commitment scheme (This can be done by calling the function generateCommitValue(byte[]).
  - Call the commit() function of the committer with the committed value and id.
  - Call the decommit() function of the committer with the same id sent to the commit() function.

Code example:

```
//create the committer
auto dlog = make_shared<OpenSSLDlogECF2m>("K-233");
CmtPedersenCommitter committer(ch, dlog, get_seeded_prg());

//generate CommitValue from string
vector<byte> msg(10, 0);
auto val = committer.generateCommitValue(msg);

//Commit on the commit value with id 2
committer.commit(val, 2);

//decommit id 2
committer.decommit(2);
```

Steps in receiver creation:

- Given a Channel object ch do:
  - Create a CmtReceiver (for example, CmtPedersenReceiver).
  - Call the receiverCommitment() function of the receiver.
  - Call the receiveDecommitment() function of the receiver with the id given in the output of the receiverCommitment() function.
  - The CommitValue returned from the receiveDecommitment() can be converted to bytes using the generateBytesFromCommitValue() function of the receiver.

Code example:

```
//create the receiver
auto dlog = make_shared<OpenSSLDlogECF2m>("K-233");
CmtPedersenReceiver receiver(ch, dlog, get_seeded_prg());

//Receive the commitment on the commit value
auto output = receiver.receiveCommitment();

//Receive the decommit
auto val = receiver.receiveDecommitment(output.getCommitmentId());

//Convert the commitValue to bytes.
vector<byte> committedVector = receiver.generateBytesFromCommitValue(val.get());

for (int i=0; i<committedVector.size(); i++){
    cout << committedVector[i];
}
cout<<endl;
```

CHAPTER 10

License

We request that any publication and/or code referring to and/or based on SCAPI contain an appropriate citation to SCAPI, including a reference to https://github.com/cryptobiu/libscapi.

SCAPI uses other open source libraries: Crypto++, Miracl, NTL, OpenSSL, OtExtension and Bouncy Castle. *Please see these projects for any further licensing issues.*

If you can't find what you are looking for, have a look at the index or try to use the search:

- *genindex*
- *search*

A

C

D

E

G

H

I

J

K

M

O

P

R

S

T

U

W

Z